Diss. ETH No. 13266

Fast Longest Prefix Matching: Algorithms, Analysis, and Applications

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of
DOCTOR OF TECHNICAL SCIENCES

presented by
MARCEL WALDVOGEL
Dipl. Informatik-Ing. ETH
born August 28, 1968
citizen of Switzerland

accepted on the recommendation of
Prof. Dr. B. Plattner, examiner
Prof. Dr. G. Varghese, co-examiner

2000

In memoriam Ulrich Breuer

An ingenious mind
in a physical world

Abstract

Many current problems demand efficient best matching algorithms. Network devices alone show several applications. They need to determine a *longest matching prefix* for packet routing or establishment of virtual circuits. In integrated services packet networks, packets need to be classified by trying to find the *most specific match* from a large number of patterns, each possibly containing wildcards at arbitrary positions. Other areas of applications include such diverse areas as geographical information systems (GIS) and persistent databases.

We describe a class of best matching algorithms based on slicing perpendicular to the patterns and performing a modified binary search over these slices. We also analyze their complexity and performance. We then introduce schemes that allow the algorithm to "learn" the structure of the database and adapt itself to it. Furthermore, we show how to efficiently implement our algorithm both using general-purpose hardware and using software running on popular personal computers and workstations.

The research presented herein was originally driven by current demands in the Internet. Since the advent of the World Wide Web, the number of users, hosts, domains, and networks connected to the Internet seems to be exploding. Not surprisingly, network traffic at major exchange points is doubling every few months. The Internet is a packet network, where each data packet is passed from a router to the next in the chain, until it reaches destination. For versatility and efficient utilization of the available transmission bandwidth, each router performs its decision where to forward a packet as independent of the other routers and the other packets for the same destination as possible.

Five key factors are required to keep pace if the Internet is to continue to provide good service: (1) higher link speeds, (2) better router data throughput, (3) faster packet forwarding rates, (4) quick adaptation to topology and load changes, and (5) the support for Quality-of-Service (QoS). Solutions for the first two are readily available: fiber-optic cables using wavelength-division multiplexing (WDM) and switching backplane interconnects. We present longest matching prefix techniques which help solving the other three factors. They allow for a high rate of forwarding decisions, quick updates, and can be extended to classify packets based on multiple fields.

The best known longest matching prefix solutions require memory accesses proportional to the length of the addresses. Our new algorithm uses binary search on hash tables organized by prefix lengths and scales very well as address and routing table sizes increase: independent of the table size, it requires a worst case time of $\log_2(\textit{address bits})$ hash lookups. Thus only 5 hash lookups are needed for the current Internet protocol version 4 (IPv4) with 32 address bits and 7 for the upcoming IPv6 with 128 address bits. We also introduce *mutating binary search* and other optimizations that, operating on the largest available databases, reduce the worst case to 4 hashes and allow the majority of addresses to be found with at most 2 hashes. We expect similar improvements to hold for IPv6.

We extend these results to find the best match for a tuple of multiple fields of the packet's header, as required for QoS support. We also show the versatility of the resulting algorithms by using it for such diverse applications as geographical information systems, memory management, garbage collection, persistent object-oriented databases, keeping distributed databases synchronized, and performing web-server access control.

Kurzfassung

Viele aktuelle Probleme erfordern effiziente Algorithmen zur Bestimmung der besten Übereinstimmung zwischen einem Suchwort und einer vorgegebenen Datenbank von jokerbehafteten Mustern. Allein der Netzwerkbereich liefert bereits diverse Anwendungen dafür. Die Geräte müssen den *längsten passenden Präfix* bestimmen, um Pakete weiterzuleiten oder virtuelle Verbindungen zu erstellen. In dienstintegrierenden Paketnetzwerken müssen die Pakete darüberhinaus noch feiner klassiert werden. Dies wird erreicht, indem bestimmte Felder aus den Paketen mit einer grossen Menge von Vergleichsmustern verglichen wird, wovon jedes Joker an möglicherweise beliebigen Positionen beinhalten kann. Aus den passendsten Mustern wird das mit der *grössten Übereinstimmung* gewählt. Neben dem Netzwerkbereich werden solche Algorithmen auch in vielen anderen Bereichen benötigt, so fuer geografische Informationssysteme oder persistente Datenbanken.

In dieser Arbeit beschreiben wir eine Klasse von Algorithmen zur Bestimmung des längsten passenden Präfixes. Sie alle basieren auf demselben Algorithmus, der eine gegebene Musterdatenbank, senkrecht zur Orientierung der Vergleichsmuster, in parallele Schichten zerteilt. Über diese Schichten wird dann eine modifizierte binäre Suche durchgeführt. Danach führen wir Schemata ein, welche es dem Basisalgorithmus erlauben, aus der Struktur der Datenbank zu “lernen” und sich ihr anzupassen. Desweiteren zeigen wir, wie unsere Algorithmen effizient implementiert werden können, sowohl mittels Standardkomponenten in Hardware als auch in Software auf beliebten Personalcomputern und Workstations.

Die hier vorgestellte Arbeit wurde durch aktuellen Forderungen im Internet initiiert. Seit der Einführung des World Wide Web ist die Zahl der Benutzer, Rechner, Domänen und Netzwerken, welche am Internet angeschlossen

sind, am Explodieren. Es überrascht deshalb nicht, dass der Netzwerkverkehr an den wichtigen Austauschknoten sich alle paar Monate verdoppelt. Das Internet ist ein Paketnetzwerk, in welchem jedes Datenpaket von Router zu Router weitergeleitet wird, bis es sein Ziel erreicht. Aus Gründen der Vielseitigkeit und effizienten Ausnützung der vorhandenen Übertragungsbandbreite leitet im Internet jeder Router jedes Paket möglichst unabhängig von den anderen Routern und anderen, auch gleich adressierten, Paketen weiter.

Damit das Internet weiterhin mit den steigenden Bedürfnissen Schritt halten kann, müssen fünf Punkte erfüllt werden: (1) höhere Geschwindigkeiten auf den Datenleitungen, (2) besserer Durchsatz innerhalb der Router, (3) schnellere Entscheidung über die Paketweiterleitung, (4) rasche Anpassung an Topologie- und Laständerungen und (5) die Unterstützung von Dienstgüte (QoS). Lösungen für die ersten beiden Punkte sind bereits verfügbar: Glasfaserkabel mit Wellenlängenmultiplexing (WDM) und Paketvermittlung anstelle von Datenbussen in den Routern. Wir stellen Techniken zur Bestimmung des längsten passenden Präfixes vor, die bei der Lösung der letzten drei Faktoren mithelfen. Sie ermöglichen eine schnelle Festlegung des nächsten Wegstückes eines Paketes, rasche Aktualisierung und können erweitert werden, und so Pakete auch aufgrund mehrerer Felder gleichzeitig zu klassieren.

Die Anzahl Speicherzugriffe der besten bekannten Techniken zur Präfixbestimmung ist proportional zur Länge der verglichenen Adressen. Unser neues Verfahren benutzt eine binäre Suche über Hashtabellen, welche nach Länge der Präfixe aufgeteilt ist. Es skaliert sehr gut mit dem Wachstum der Adresslängen und Routingtabellen: unabhängig von der Tabellengrösse werden maximal $\log_2(\textit{Adressbits})$ Hashzugriffe benötigt. Dadurch werden beim aktuellen Internetprotokoll Version 4 (IPv4) mit 32 Bit langen Adressen nur 5 Hashzugriffe gebraucht, oder 7 für das bevorstehende IPv6 mit 128 Adressbits. Wir führen ebenso *mutierende binäre Suche* sowie weitere Optimierungen ein, welche es erlauben, den schlechtesten Fall auf 4 Zugriffe zu limitieren sowie die Mehrzahl der Adressen in maximal 2 Zugriffen zu finden. Wir erwarten ähnliche Verbesserungen für IPv6.

Wir erweitern diese Verfahren zur Suche nach den beste Übereinstimmung für Tupel von mehreren Felden des Paketkopfes, welche für QoS-Unterstützung benötigt wird. Darüberhinaus zeigen wir die Vielseitigkeit der resultierenden Algorithmen, indem wir sie für so unterschiedliche Anwendungen einsetzen wie geografische Informationssysteme, Speicherverwaltung, Garbage Collection, persistente objekt-orientierte Datenbanken, Synchronisation in verteilten Datenbanken und Zugriffskontrolle von Webservern.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many current problems demand efficient best matching algorithms. Network devices alone show several applications. They need to determine a *longest matching prefix* for packet routing or establishment of virtual circuits. In integrated services networks, packets need to be classified by trying to find the *most specific match* from a large number of patterns, each possibly containing wildcards at arbitrary positions. Other areas of applications include such diverse areas as geographical information systems (GIS) and persistent databases.

We describe a class of best matching algorithms based on slicing perpendicular to the patterns and performing a modified binary search over these slices. We also analyze their complexity and performance. We then introduce schemes that allow the algorithm to "learn" the structure of the database and adapt itself to it. Furthermore, we show how to efficiently implement our algorithm both using general-purpose hardware and using software running on popular personal computers and workstations.

1.1 The Internet

The Internet is becoming ubiquitous: everyone wants to join in. Since the advent of the World Wide Web, the number of users, hosts, domains, and net-

works connected to the Internet seems to be exploding [Gra96]. Not surprisingly, network traffic at major exchange points is doubling every few months. The proliferation of multimedia networking applications and devices is expected to give traffic another major boost.

The increasing traffic demand requires four key factors to keep pace if the Internet is to continue to provide good service: higher link speeds, better router data throughput, faster packet forwarding rates, and quick adaptation to routing changes. Readily available solutions exist for the first two factors: for example, fiber-optic cables can provide faster links and switching technology can be used to move packets from the input interface of a router to the corresponding output interface at multi-gigabit speeds [PC$^+$98]. We deal with the other two factors: Forwarding packets at high speeds while still allowing for frequent updates to the routing table.

The Internet resulted out of the first packet-switched networks. Unlike the then-predominant circuit-switched networks, such as the public telephone network, and other packet-switched networks such as X.25 [IT96], every packet travels across the network independently of all others. While this makes optimal use of the available bandwidth through the inherent high-resolution multiplexing, it also requires that each packet is almost self-sufficient. Each packet must be able to be treated independently of all the other packets in its stream. This usually implies that each packet is labelled with both a globally unique source and destination address, which it must carry along.

In a router, the major step in packet forwarding is to lookup the destination address of an incoming packet in the routing database. While there are other chores, such as updating time-to-live (TTL) fields and checksums, these are computationally inexpensive compared to the major task of address lookup (see Section 2.2.1). Data link bridges have been doing address lookups at 100 Mbps [Spi95] for many years. However, bridges only do exact matching on the destination (MAC) address, while Internet routers have to search their database for the *longest prefix* matching a destination IP address. Thus, standard techniques for exact matching, such as perfect hashing, binary search, and standard Content Addressable Memories (CAMs) cannot directly be used for Internet address lookups. Also, the most widely used algorithm for IP address lookups, BSD Patricia Tries [Skl93], have bad worst-case behavior.

Prefix matching was introduced in the early 1990s, when it was foreseen that the number of endpoints and the amount of routing information would grow enormously. Then, only address classes A, B, and C existed, giving in-

dividual sites either 24, 16, and 8 bits of address space, allowing up to 16 Million, 65,534, and 254 host addresses, respectively. The size of the network could easily be deducted from the first few address bits, making hashing a popular technique. The limited granularity imposed by the three address classes turned out to be extremely wasteful on address space. To make better use of this scarce resource, especially the class B addresses, bundles of class C networks were given out instead of class B addresses. This would have resulted in massive growth of routing table entries over time. Therefore, Classless Inter-Domain Routing (CIDR) [RL93, FLYV93] was introduced, which allows for aggregation of networks in arbitrary powers of two to reduce routing table entries. With this aggregation, it was no longer possible to identify the number of bits relevant for the forwarding decision from the address itself, but required a prefix match, where the number of relevant bits was only known when the matching entry had already been found in the database.

The use of best matching prefix in forwarding has allowed IP routers to accommodate various levels of address hierarchies, and has allowed parts of the network to be oblivious of details in other parts. Given that best matching prefix forwarding is necessary for hierarchies, and hashing is a natural solution for exact matching, a natural question is: "Why can't we modify hashing to do best matching prefix?" However, for several years now, it was considered not to be "apparent how to accommodate hierarchies while using hashing, other than rehashing for each level of hierarchy possible" [Skl93]. In 1997, we started the race for faster lookups by introducing a technique which will perform significantly better than that [WVTP97, WVTP98]. This thesis covers and extends on the work started then.

1.2 Beyond Packet Forwarding

As the Internet became more popular, the applications demanding the majority of the bandwidth moved from bulk file transfer and e-mail to interactive browsing of the World Wide Web and audiovisual broadcasts or conferencing. For the latter, delays and packet drops due to congestion are much more noticeable. Therefore, the user demands for a network serving these applications are much higher, leading to requests for bandwidth reservation and preferential treatment.

To meet these demands, several approaches have been proposed, all requiring *packet classification* at several or all routers within the network to function properly. Packet classification defines the process of matching packets against a pre-defined database of often partially defined header fields, such as source and destination addresses, protocol, and even application-specific parameters.

Another application requiring packet classification in increasing demand are firewalls, devices which block or allow packets based on their matching against a database.

1.3 Claims

This thesis addresses several topics, which are listed below and revisited in Chapter 10.

1. **Fast and Scalable Longest Prefix Matching**
   We introduce a fast and scalable, yet generic algorithm which allows for matching query items against a large database of possibly overlapping prefixes.

2. **Fast Forwarding**
   We apply this algorithm to Internet packet forwarding and analyze its performance using the largest available databases.

3. **Adaptivity**
   We extend the generic algorithm to a scheme which can self-adapt to structures discovered in the search database, resulting in a further performance boost.

4. **Efficient Building and Updating**
   We present efficient algorithms for building both the generic and the self-adapting data structures. We also show how to update them both quickly and efficiently.

5. **Fast Hashing**
   We explain and analyze practical schemes for fast hashing. This scheme is required for the operation of the presented algorithms. We also show that the search structures and hashing can be efficiently combined to yield even better results.

6. **Two Dimensions**
   We extend the scheme to perform two-dimensional prefix-style packet classification. This is required for basic packet classification on source/ destination address pairs.

7. **Packet Classification**
   We further enhance the algorithm to an efficient full-fledged five-dimensional Internet packet classification, thanks to known properties of the additional three dimensions.

8. **Versatility**
   We show that our algorithm is not limited to theory and Internet. Instead, the availability of our prefix matching scheme makes a series of other applications practical for the first time or improves them significantly.

1.4 Overview

This thesis is structured as follows. Chapter 2 introduces the basics of packet networks, forwarding lookups, packet classification, and hashing. Chapter 3 discusses related work in these fields.

Chapter 4 describes how to efficiently search a forwarding database, Chapter 5 explains build and update procedures and documents practical techniques for fast hashing, Chapter 6 implements efficient software searches and presents cheap hardware for multi-gigabit lookups. The results are evaluated in Chapter 7.

Chapter 8 first describes multi-dimensional packet classification and then introduces additional matching problems. Chapter 9 presents further applications for the algorithms and techniques presented herein. Chapter 10 concludes this thesis.

Chapter 2

Background

In this chapter, we will give some background on the Internet Protocol and then state the underlying problems of prefix matching and packet classification in more detail. We will also mention the classical algorithms on which this work builds.

2.1 The Internet Protocol

2.1.1 History and Evolution

In 1961, Kleinrock [Kle61] proposed and analyzed the use of packet switched networks. This work resulted 1969 in the ARPANET, from which our current Internet evolved. Since its inception in 1978 [Rob97], the Internet protocol is dubbed "version 4" [Pos81a]. The reasons for starting with version 4 are obscure, but the version numbers below 4 have never been officially assigned or are reserved. With the years, it turned out that the Internet developed differently from what the original protocol designers had thought. That the Internet in fact bloomed much better than the designers had imagined even in their wildest dreams turned out to be a major problem. Through the immense growth, address space was getting extremely scarce. Also it was predicted that quality of service and security would become issues in the near future. To support them and any other issues that might show up, the protocol should

be designed in an extensible, yet efficient way. Items such as these were put on the agenda for proposals for designing and engineering the next generation Internet (IPng) [BM93].

Among the different proposals, the one which had been assigned the experimental version number 6, received most attention and started to evolve and integrate promising features from the other proposals. The result is now known as *Internet Protocol Version 6* (IPv6) [DH98]. Version number 5 had been allocated for the experimental Stream Protocol [Top90, DB95], which is not part of the official Internet protocol family, but was also designed as an evolution from IPv4.

2.1.2 Internet Protocol Layer

In Tables 2.1 and 2.2, the headers of the IP version 4 and 6 protocols, respectively, are depicted. In all the protocol figures in this thesis, each row represents four bytes, and the important fields are set in bold.

0		1	2	3
Vers	HLen	**ToS**	Packet Length	
IP ID			Fragment Info/Offset	
TTL		**Protocol**	Header Checksum	
Source Address				
Destination Address				
IP Options (optional, variable length)				

Table 2.1: *IP Version 4 Header Format*

Probably the most important field to determine packet handling in IPv4 (Table 2.1) is the destination address. Each router between source and destination will have to look at this field and determines the direction to send the packet to based on its contents. This is the normal forwarding operation. For packet classification, the source address is also looked at. This address pair together define a *flow* in its coarsest possible notion. This so-called "host-based" association groups all uni-directional traffic between the two hosts. Another important field is the protocol ID, which defines the transport-level protocol that is encapsulated within this IP packet. The most common values for it are TCP (used for reliable and well-behaving traffic) and UDP (used for real-time

services and group communication). In addition to these two, a number of other protocols are defined, mostly for control and management purposes, with ICMP (for control messages, such as "your packet could not be delivered") as the main representative. The type-of-service field (ToS) was created to identify the packet's priority, it's queueing, throughput, and dropping behavior to the routers. Several applications (e.g., telnet [PR83] and ftp [PR85]) do set these flags. Nevertheless, they were only used seldomly for packet classification in routers. With Differentiated Services (DiffServ, Section 2.2.3) being explored right now, two previously reserved bits in the ToS field may be used in core routers to determine packet treatment without classification.

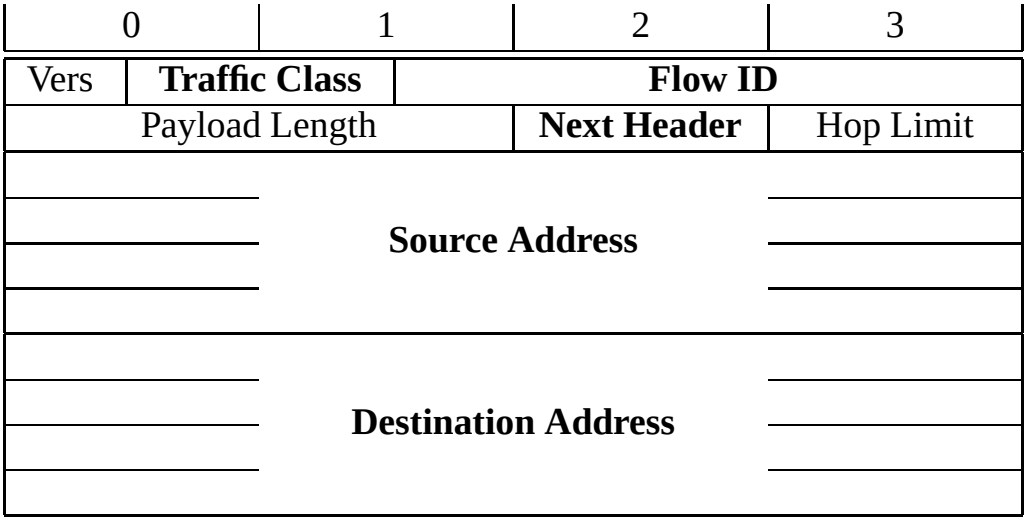


Table 2.2: *IP Version 6 Header Format*

The IPv6 header as shown in Table 2.2 has a structure similar to IPv4. The most notable changes are that addresses have been extended from 32 to 128 bits, the protocol ID field has been renamed "Next Header" to stress the improved modularity, and several fields have been removed to streamline the protocol. The Traffic Class field is intended to extend the semantics of the IPv4 Type-of-Service field, but its use is currently only experimental.

To simplify packet classification, a flow ID has been added. The tuple (source address, flow ID) should uniquely identify a flow for any non-zero flow ID. The exact definition of "flow" is left to the application, but it should cover only packets that require homogeneous treatment in transit. Nevertheless, many routers will need to perform full-fledged classification. The flow ID was not meant to entirely replace classification, only to simplify parsing of the packets. Especially if the classification is performed for security reasons, the flow ID does not help.

2.1.3 Transport Protocol Layer

On the transport layer, only two protocols, UDP (Table 2.3) and TCP (Table 2.4), provide information that is commonly used to further classify packets: Source and destination port numbers. These numbers are used to dispatch packets to the receiving application and represent the fine-grained (and more common) variety of flows. Within the network, they can be used to identify a pair of communicating applications. Thus, with appropriate signalling support, applications can let the network know about their requirements, commonly in the form of resource reservations.

Many of the port numbers have a publicly known meaning. These so-called *well-known ports*, have been assigned for, and are reserved to, common services. For example, port number 80 is assigned to communicate with World Wide Web (WWW [Wil98]) servers using the Hypertext Transfer Protocol (HTTP), so traffic to or from port 80 will most likely be WWW traffic.

0	1	2	3
Source Port		**Destination Port**	
UDP Data Length		Checksum	

Table 2.3: *UDP Header Format*

0	1	2	3
Source Port		**Destination Port**	
Sequence Number			
Acknowledgement Number			
Offset —	**Flags**	Window Size	
Checksum		Urgent Pointer	
TCP Options (optional, variable length)			

Table 2.4: *TCP Header Format*

If you desire to know more about the Internet protocol, [Ste94] is highly recommended reading for the technically inclined reader and contains all the information you might desire to know.

2.2 The Problems

The Internet poses several problems related to per-packet decisions. Forwarding and packet classification are discussed below.

2.2.1 Internet Packet Forwarding

As we have seen in Chapter 1, it was the desire of the designers of the Internet to keep the router's routing and forwarding databases small. This was originally done by not requiring the routers to keep track of all individual nodes of the network, but by grouping them in networks of different size. Each router outside a given network only had to know about how to forward packets to these networks, where routers with knowledge about the internal topology would make sure the packet reached its final destination. Two additional goals were also set up and met by the original designers:

1. To provide for several sizes of networks, as to closely fit the needs of the organizations connected to the Internet.
2. To simplify the extraction of the part of the address that named the destination network.

The first goal was met by designing the network sizes so that the individual networks could contain up to 254, 65534, or 16 million hosts, named Class C, B, and A, respectively. The second goal was met by having the first few bits of the address indicate the length of the network part. This resulted in 126 Class A networks, 16382 Class B networks, and 2 million Class C networks, leaving one eighth of the address space for future wild ideas (Table 2.5).

To achieve maximum routing table space reduction, aggregation is done aggressively: Suppose all the subnets in a big network have identical routing information except for a single, small subnet with different information. Instead of having multiple routing entries for each subnet in the large network, just two entries are needed: one for the overall network, and one entry showing the exception for the small subnet. Now there are two matches for packets addressed to the exceptional subnet. Clearly, the exception entry should get preference there. This is achieved by preferring the more specific entry, resulting in a *Best Matching Prefix* (BMP) operation. In summary, CIDR traded

Class	First Bits	# of Networks	# of Hosts per Network
A	0	126	16777214
B	10	16382	65534
C	110	2097150	254
D[a]	1110	—	—
E[b]	1111	—	—

[a] Originally unassigned, later used for Multicast [DC90]
[b] Unassigned

Table 2.5: *Classful Internet Addressing (Outdated)*

off better usage of the limited IP address space and a reduction in routing information for a more complex lookup scheme.

Today, an IP router's database consists of a number of *address prefixes*. A prefix is a specific pattern, which represents a bit sequence consisting of two distinct areas. One area consists of bits with an exactly defined value (chosen from either 0 or 1), which makes up the most significant bits ("left-aligned"). The other area consists of all don't-care bits and is "right-aligned". Either of the areas may be empty, but the sum of the lengths of the two areas must equal the length of the address they should be compared with. The *prefix length* is defined as the number of non-wildcard bits.

When an IP router receives a packet, it must compute which of the prefixes in its database has the longest match when compared to the destination address in the packet. The packet is then forwarded to the output link associated with that prefix, directed to the next router or the destination host. For example, a forwarding database may have the prefixes $P_1 = 0000*$, $P_2 = 0000\_111*$ and $P_3 = 0000\_1111\_0000*$, with $*$ meaning all further bits are unspecified and _ being used to visually group the bits. An address whose first 12 bits are 0000_0110_1111 has longest matching prefix P_1. On the other hand, an address whose first 12 bits are 0000_1111_0000 matches all three prefixes, but its longest matching prefix is P_3. For one-dimensional prefix databases, if it contains a match at all, there is always a single distinct prefix in this database having the longest prefix length associated with it.

Current backbone routers have to deal with forwarding databases containing up to 40,000 prefixes. Several millions of packets per second have to be compared against this database for each network link. Today (June 1999), the biggest routers contain a dozen OC-192 (9.6Gb/s) links, corresponding to

more than 300 million minimum-sized packets passing through the router each second. The forwarding databases, the link speeds, and the link counts are still growing, with the number of packets per link doubling every few months. Current Internet (IPv4) addresses are 32 bits long, with 128 bits upcoming (IPv6 [DH98]) to relieve the current IP address scarcity and allow for further growth.

Besides the forwarding decision, routers have to perform other tasks when forwarding a packet [Bak95]:

Checksum verification The standard makes it a requirement to check the header checksum of any IPv4 packet before performing looking at any other field. But hardly any router verifies the checksum, because the checksum calculation is considered to be too expensive, despite the speed optimizations possible to the naïve algorithm [BBP88]. Checksumming is omitted under the assumption that (1) packets hardly ever get corrupted in transit with current technology, especially fiber-optics, and (2) end systems (hosts) will recognize the corruption. IP version 6 [DH98] therefore no longer has an IP header checksum, the relevant header fields (source and destination address) are only checked by inclusion into the transport protocol checksum (already the IPv4 transport protocols include the IP addresses into their checksum).

Fragmentation A packet may need to be fragmented, because the outbound link cannot handle the size of the packet. This is very unusual for high-speed links, since they are designed to handle large enough packets.

Time-to-Live The Time-to-Live (TTL) field is used to detect packets looping in the network. A host sending a packet typically initializes the TTL with 64 (recommended by [RP94]) or 255 (the maximum). Each router then decrements it. The packet is discarded and an error message generated when the TTL reaches zero before reaching the destination.

Checksum update Since a header field—the TTL—was changed, the checksum needs to be recalculated. [MK90] describes how to efficiently do this incrementally, if only the TTL was decremented. Using incremental updates also allows end systems to recognize corrupted headers and does not run the risk that routers unknowingly overwrite the unchecked bad checksum with a good one.

Although there are a lot of chores to be performed, in almost all cases, they are reduced to decrementing and checking the TTL and incrementally updating

the checksum. These two operations combined are much cheaper than making a forwarding decision.

2.2.2 Multicast Packet Forwarding

Multicast routing in the Internet is currently being done based on either a *Distance-Vector* algorithm [DPW88, DC90, Dee91] or so-called *Core-Based Trees* [Bal97b, Bal97a].

For core-based trees, a shared spanning tree is established for each multicast group, containing all the group's recipients, including those who are also senders. Routers within the group's tree determine, which spanning tree to use based on an exact match of the destination (multicast group) address with their forwarding database. There are also routers outside the tree, those serving the nodes which only send to the group. These routers forward all packets addressed to the group towards a router within the tree, called *core*. All forwarding is done using exact prefix matching on the destination address, and thus, there is no need to perform any prefix matching. A similar scheme is also used for Protocol Independent Multicast–Sparse Mode (PIM-SM [EFH$^+$98]).

The most popular multicast scheme utilized in the current Internet is distance-vector multicast routing (DVMRP). Each sender is allocated its own multicast distribution tree. This results in the fastest and most ressource-efficient delivery of messages. Unfortunately, with many senders, this approach also requires a lot of state at all multicast routers: For each pair of sender and multicast group this sender transmits to, the router needs to maintain the list of outgoing interfaces the packets need to be multicast. To suppress loops in the routing and build and update the initial distribution tree, it also needs to store the interface the packets are expected to come in on.

To reduce the information kept for DVMRP, aggregation also takes place. Since the multicast groups (the destination address) have no exploitable structure, the senders are aggregated based on their addresses. This happens based on the unicast routing information. For lookups, DVMRP thus requires first an exact match on the group (destination) address, followed by a longest prefix match on the source address. This can also be combined into a longest prefix match based on the concatenation of destination and source addresses. As such, although two addresses ("dimensions") are involved, it can be reduced to a one-dimensional prefix matching.

Header Field	Priority	IntServ	DiffServ	Firewalls
Type of Service	Exact	—	—	—
Source Address	(Exact)	Wildcard	Prefix	Prefix
Destination Address	(Exact)	Wildcard	Prefix	Prefix
Protocol ID	Exact	Wildcard	(Wildcard)	Wildcard
Source Port	Exact	Wildcard	(Wildcard)	Range
Destination Port	Exact	Wildcard	(Wildcard)	Range
TCP SYN	—	—	—	Wildcard

Table 2.6: *Header Field Matching*

MASC/BGMP [KRT+98] extends the aggregation performed for DVMRP. Not only are the senders grouped based on their address, also multicast addresses can be tied together, if they are expected to have similar structure and their addresses were allocated to make prefix aggregation possible. Such structural similarities across groups is e.g. caused by protocols using multiple multicast groups to transmit related material. A prominent representative of this category is layered multicast [MJV96]. Thus, MASC/BGMP requires two-dimensional longest prefix matching. Multi-dimensional matching ("classification") is covered in the next section.

2.2.3 Internet Packet Classification

In the recent years, we've seen an increasing demand for Quality of Service (QoS) provisioning in the Internet. With more and more organizations connected, they become more and more concerned about the security of their internal network. This usually leads to the set-up of Internet firewalls, devices that decide whether to allow or deny a packet based on a set of rules, that may be changing dynamically.

Priority-based Forwarding

For a long time, routers supported priority-based forwarding, where the network administrator could determine which packets should be treated preferentially. This was usually done by performing exact matches against either of the header fields, with type of service and the port fields used most fre-

quently. Often, the fields could be searched in order until a match was found (Table 2.6). The prioritization was static, inflexible, and coarse-grained, yet still needed a lot of manual tuning.

Integrated Services

QoS is currently being used in the context of Integrated Services (IntServ) [BCS93, BCS94, Wro97]. These schemes currently require all routers along the path between source and destination to keep state for each individual "flow". For further understanding it is sufficient to think of a flow as an end-to-end connection. Obviously, this flow-by-flow accounting does not scale well. On the other hand, due to the missing aggregation, the matching rules are also simple. Classically, packet filtering takes the five classical header fields into account while comparing a packet against the database: source and destination address, protocol ID, and potentially the source and destination port numbers, which identify the originating and receiving application.

As is shown in Table 2.6, the current IntServ mechanisms only require wildcard matching, signifying each of the fields is either completely specified (exact match) or left entirely unspecified (wildcard). This limited matching makes it easier to implement than prefix or range matching. There is currently research going on, which tries to aggregate reservations, thus reducing the amount of information being kept at the backbone routers [DWD+97, DDPP98, BV98]. Unfortunately, this will also require more complex matching schemes.

Differentiated Services

Differentiated Services (DiffServ, [BBC+98, FKSS98, AFL97]) try to reduce the amount of information being kept in backbone routers to a minimum: none. Instead, they use a number of bits in the packet header to specify whether the packet does not need any special treatment (i.e., should receive "best effort" treatment) or is part of a reserved flow. For the latter, it generally is further specified whether that packet is within the limits specified by the traffic contract or not. According to this information, the priority of the packet is decided upon by the router, without the need to revert to databases, keeping additional effort to a minimum.

Unfortunately, this scheme invites cheating: Everyone can happily set the "high-priority" bits in the outgoing packets to get better treatment. Unfortunately, those that require preferential treatment and may even have paid for that privilege, will notice degraded traffic quality. Therefore, at each boundary between administrative domains, i.e., between the end user and its Internet service provider (ISP) or between ISPs, the receiving partner must check that the priority bits are indeed set according to a pre-arranged traffic contract. It is at these points, that high-speed packet classification is required to "police" the traffic according to the contract. As can be seen from Table 2.6, they do require prefix matching for source and destination addresses, and (depending on the exact nature of the contract) possibly also wildcard matching on the remaining fields.

Firewalls

Firewalls [CZ95] do require the most sophisticated pattern matching algorithms (Table 2.6. Not only should it be possible to aggregate source and destination addresses arbitrarily, requiring prefix matching. But we also need to support wildcards on the protocol ID and the even more complex range matching on the port numbers. Luckily, only a limited number of ranges is generally being matched ($1 \ldots 1023, 1024 \ldots 65536, 6000 \ldots 6099$), everything else is plain wildcard matching. Unfortunately, firewall rules usually require an additional field, namely, whether the TCP SYN bit is set. This is necessary to differentiate whether a connection is set up from the "inside" (trusted, protected region) or the "outside" (insecure region), increasing the number of relevant dimensions to 6.

2.2.4 The Knowledge Meta-Problem

Although much is known about the protocol and its related algorithms, our knowledge about traffic patterns is limited. This is partly because monitoring the high-speed backbone networks is difficult [TMW97], and partly because measurements cannot keep track with the pace user behavior and thus the Internet traffic is changing. This lack of knowledge poses a major problem for designers and vendors of Internet equipment.

Nevertheless, there are a few resources available providing for up-to-date data. The Internet Performance and Measurement Project [Int] collects and

analyzes [Lab96, LMJ97] routing information. The U.S. National Laboratory for Applied Network Research (NLANR) keeps information on data flows, such as packet size distributions [Nat97] and has recently started to supply real-time packet statistics [Nat].

Until now, there unfortunately exists no paired routing *and* traffic data from the same location, which would help in analyzing algorithms. Also, for several of the advanced problems, there is only rudimentary knowledge about the datasets available the algorithms will have to deal with in the near future.

2.3 Matching Techniques

"Matching" is most often associated with algorithms that search for a single data item in a large string of constant data ("exact matching", "substring matching"). It is often forgotten, that this also applies to comparing a *single pattern* against a large set of constant data ("pattern matching"). Even less is known about the problem we face with Internet forwarding and classification: Comparing a constant entry against a large *set of patterns* ("classification", "point location"). This is summarized in Table 2.7. Matching pattern items against pattern databases ("best matching") and matching inexact items against inexact databases ("closest fit") are out of scope of this thesis. See [Gus97] for a discussion.

		Database Entries	
		Fully Specified	Partially Specified
Search Item	Full	Exact Matching	Classification
	Partial	Pattern Matching	Best Matching

Table 2.7: *Classification of Matching Techniques*

The following sections give an *overview* over some existing techniques in these categories. The algorithms closely related to the topic of this thesis will be discussed in detail in Chapter 3.

2.3.1 Exact Matching

A large variety of techniques is known for exact matching in different data structures. The number of techniques is so large, that we're only able to mention general techniques, there exists a large number of solutions, each tailored to a specific problem. On linear memory, well-known strategies are linear search, binary search, and hashing. Using structured memory (i.e., pointers), we are able to add trees and tries to the list. [Knu98] gives an excellent overview over the different techniques and also explains a vast number of specialized sub-forms.

Content Addressable Memories (CAM) add hardware parallelism to the matching. Each memory cell is equipped with a comparator, verifying whether the contents of its cell equals the search item. While many CAMs are used for exact matching, often they are also capable of matching against a search pattern. Modern "Ternary CAMs" even allow their memory locations to contain wildcards, which makes them suitable for classification. For a more detailed discussion of CAMs, see Section 3.2.4.

Hashing

Hashing is a very prominent candidate among the exact matching group, since—on average—they can provide for $O(1)$ access in $O(1)$ memory per database entry. Probably the most prominent representative is known as *perfect hashing* [FKS84], providing for $O(1)$ *worst-case* access and memory. Unfortunately, finding a hash function which meets these criteria is very dependent on the entries in the database. Thus, database build and update time can take non-polynomial time and lead to expensive hash functions.

A more practical solution is *dynamic perfect hashing* [DMR$^+$94], providing for $O(1)$ access with modest update times. Hash methods are analyzed in more detail in Section 5.5.

Wildcard Matching

Wildcard matching extends exact matching by providing for a "fall-back" or "match-all" entry, which is considered to be matched if no exact match was found. While this is trivial to implement when matching a single item against

a database (one-dimensional matching), it adds another level of complexity, when tuples of items are matched against a tuple database (multi-dimensional matching), and each of the items in the tuples can be individually wildcarded.

2.3.2 Substring Matching

Substring matching is a variation of exact matching. Again, both the search item and the database are fully defined. This time, the database does not consist of several independent entries, but of a single large sequence of symbols, a string, and the search item is to be compared with every possible substring. Two well-known representants of solutions to this problem are the algorithms by Knuth-Morris-Pratt [KMP77] and Boyer-Moore [BM77]. Both are also discussed in [Knu98, Gus97], together with further algorithms.

2.3.3 Pattern Matching

Pattern matching is in wide use for matching wildcarded substrings in a large string. The best-known solutions in this field are variations on the Knuth-Morris-Pratt and Boyer-Moore algorithms [Gus97]. If the pattern is to be matched against a database of individual entries and contains a non-trivial leading prefix, trie matching schemes are also in widespread use.

Nievergelt et al. [NHS84] describe the Grid File, a way to structure multi-dimensional data on disks so that subset queries can be answered efficiently.

2.3.4 Point Location

Given a database of objects in a d-dimensional space and a query point in this space, which object(s) is this point in? This is the basic question that defines the point location problem. Depending on the constraints of the number of dimensions and the shape of the objects, a vast variety of algorithms has been created to optimally tackle this field [BKOS97].

Packet classification is considered a sub-problem of the general problem space, since a potentially large set of constraints on the form of the objects is known in advance. As will be explained in more detail in Chapter 8, packet

classification per se has a number of constraints which allow for more efficient solutions. Additionally, the databases that will be fed to these algorithms often contain a significant amount of regularities, which can additionally be used to improve on the algorithms.

2.4 Relations Between One-Dimensional Matching Problems

Another common matching problem we have already seen besides prefix matching is *range matching*. Simple prefix matching problem—without overlapping—is a special case of range matching. Prefixes are limited ranges, whose size is a power of two and whose start and thus also its end is a multiple of its size. Any range delimited by integer positions can be split into at most $2W$ prefixes, where W is the number of bits used to represent the size of the range. This encoding is entirely independent of the remainder of the database, as long as there are no overlaps.

As soon as we introduce overlaps into either the prefix or the range database, we require a mechanism to differentiate between the overlaps. In prefix matching, there is a natural way. Any two prefixes are either entirely contained within one another or do not overlap at all. This naturally assigns the higher priority to the contained prefix. When choosing priorities according to another criteria, some or all of the contained prefixes would never match, so there would be no need for them to be part of the database.

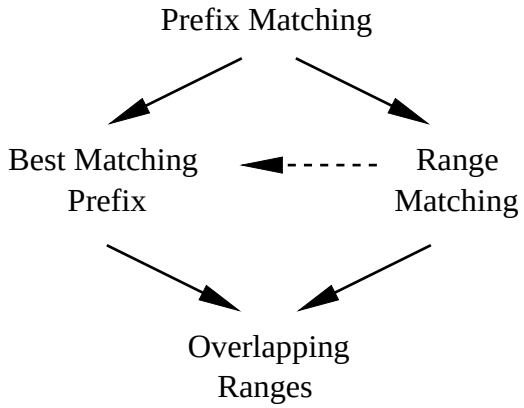


Figure 2.1: *"Inheritance Hierarchy" of Matching Problems*

On the other hand, such an inherent distinction is not apparent for overlapping ranges, requiring explicit priorities in the general case. This implies that when allowing for overlaps, the search algorithms need to more elaborate.

The relation between the different facettes of range matching is depicted in Figure 2.1. Looking at this figure, the question of the relationship between range matching and best matching prefix arises. As can be seen, any database of N potentially overlapping prefixes can be split into at most $2N$ ranges. Unfortunately, an addition or deletion of a single prefix can cause N ranges to be created or deleted, which is clearly undesirable.

Chapter 3

Related Work

As our main topic is longest prefix matching, this chapter will mainly cover the algorithms which either directly solve this problem or are easily so adapted. We will also cover multi-dimensional longest prefix matching techniques. Firstly, we introduce the performance metrics used.

3.1 Performance Metrics

Performance is classically measured in both time and space requirements. For algorithms, *space* usually corresponds to memory consumption, while *time* is based on the number of operations required on a virtual CPU (such as the Random Access Machine (RAM) model [AHU74] model) or the number of seconds spent on a real CPU.

We will see both time and space metrics in this chapter, discussing space and memory complexities ($O(x)$ notation). For the time requirements, several different measures will be used:

Average-case search time The main unresolved goal in IP forwarding is speed, therefore, it will obviously be one of the main factors for evaluating and comparing algorithms. Since IP routers are statistical devices anyway, i.e., are equipped with buffers to accomodate traffic fluctua-

tions, average-case speed seems like an adequate measure. Unfortunately, it is hard to come up with reliable average-case scenarios, since they heavily depend on the traffic model and traffic distribution.

We believe that using publicly available backbone databases and a uniform distribution of destination addresses across the entire *address space* results in a reasonable approximation of the average case. Assuming a uniform distribution tends to err towards shorter lookup times, since shorter prefixes (which were typically assigned earlier, when the address space was only sparsely populated) tend to be more sparsely populated. In addition, most of the popular sites showing heavy traffic are located in relatively small prefixes.

Unfortunately, the resulting skew cannot be quantized reliably. To give a bound on the error, we often will also mention the other extreme: When traffic per *prefix* is constant. These two metrics will be used below when referring to "bounding" the average case.

Worst-case search time Unfortunately, it is unknown how prefix and traffic distributions will evolve in the future. The Internet so far has rather successfully tried to escape predictability. Therefore, known worst-case bounds are important for designing a system that should work well over the next several years, making worst-case bounds at least as important as knowledge about the average case. In some cases, such as for implementation in hardware, or together with hardware, constant time or constant worst time lookups are a prerequisite.

3.2 Existing Approaches to Longest Prefix Matching

We survey existing approaches to IP lookups and their problems. We discuss approaches based on modifying exact matching schemes, trie based schemes, hardware solutions based on parallelism, proposals for protocol changes to simplify IP lookup, and caching solutions.

3.2.1 Trie-Based Schemes

The classical solutions for longest prefix matching have been tries. The following sections explain the evolution of this scheme.

Binary Trie

The binary trie represents the simplest species in this tree-like family. The name is apparently based on "re*trie*val" [Bri59, Fre60]. Unlike trees, branching in tries is not based on a ordered comparison with the key stored in the node. Instead, bits are extracted sequentially from the search key and used to index into a pointer table, going left or right. A simple binary trie is shown in Figure 3.1. The grey nodes are terminal nodes, i.e., nodes that do contain more information on what to do if the search terminates there. White nodes only guide the search process. Assume searching for the bit sequence 011. First, the most significant bit (0) is extracted and used to decide which way to branch from the root to the gray node directly below it to the left. This gray node is remembered as our current longest match, in case nothing better will be found. Then, the next bit (the middle 1) is extracted from the search string and branched along the path labeled "1" to its white right child node. The third bit, another 1, cannot be used, since there is no link labeled "1" from that node. Thus, the remembered gray node is the best match.

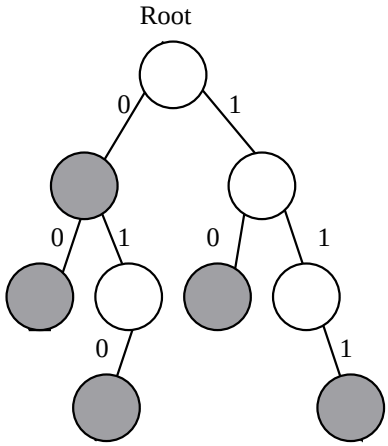


Figure 3.1: *Simple Binary Trie*

This solution requires $O(W)$ time and $O(NW)$ memory. The average case according to the current Internet is bounded by 11 and 22 trie steps.

Path-Compressed Tries

Often, these binary tries contain long sequences of nodes without branching. For exact matching, Gwehenberger [Gwe68] presents an algorithm to reduce these long chains of "white" nodes (compare Figure 3.1), by introducing a *skip count* in each node. This technique of using skip counts later became known as *path compression* and reduces the number of trie nodes to $2N-1$, independent of the data set. At about the same time, Morrison [Mor68] independently created a library indexing system based on the same idea, which he called PATRICIA. This name has stuck.

The most commonly available IP lookup implementation was implemented for the BSD Unix kernel by Sklower [Skl93]. It combines the idea of path path-compression and longest prefix matching. In path-compressed tries, not all of the intermediate bits are stored in the trie nodes traversed. Although this optimization reduces the work required for exact matches, it increases the cost of longest prefix matching searches: When the search reaches a terminal node, it may turn out that the entry stored there no longer matches the search argument. This then leads to expensive backtracking. Because the structure of these single-bit tries closely models the aggregation hierarchy, updates are very fast. Despite this, the search implementation requires up to $2W$ costly memory accesses—64 or 256 for IPv4 or IPv6, respectively.

PATRICIA exhibits the same time complexity as the binary trie, but generally is faster. Since the average cost depends much stronger than the basic trie on the actual prefixes stored, the average case would be too variable to make a reasonable statement.

LC-Tries

Nilsson and Karlsson [NK98] enhance the radix trie's path compression by *level compression*. Whenever a trie node does not have terminal nodes for the next t levels, the fan-out trie is compressed into a single array of size 2^t and t bits are used to index into this array of pointers to nodes. Unlike all the trie variants mentioned below, this structure is still self-contained. This means that it does not require a simpler helper trie to be built from. Unfortunately, insertions and deletions can become expensive, if they require large fan-out arrays to be built or destroyed. It also does not gain as much speed advantage as the read-only tries below, yet it is quite complex to handle.

As PATRICIA, LC-tries do not improve time complexity, although the average search depth is reduced from 19 (binary trie) to 8 (LC-trie) in the authors' analysis.

Controlled Prefix Expansion

Srinivasan and Varghese [SV99a] improved binary tries by extracting multiple bits at a time and using them as an index into an array of pointers to child nodes. While the idea is extremely simple, the implementation is tricky. Since there is no longer a one-to-one relation between terminal nodes and database entries, updating can be painful and requires a conventional binary trie as a helper structure. Also, memory consumption can be high if the number of bits to take per step (*stride*) does not match the database well. But probably the idea's biggest disadvantage is that it does only improve search speed by a constant factor and thus does not scale to longer address lengths.

The time complexity is $O(W/S)$, where S is the stride, with memory growing to $O(N * 2^S)$. With S being tunable, average-case numbers cannot be given.

Compact Tries

Degermark et al. [DBCP97] present a method for compactly encoding the largest forwarding databases to fit into the second-level cache of DEC Alpha [Sit92] processors. Their scheme is quite complex and shows similarities to data compression algorithms. The search requires a large number of different steps. As the operations are done in the cache and not in slow main memory, they still perform well.

Again, the time complexity remains at $O(W)$. Due to the entirely different per-search-step operation, an average comparison with the other tries does not make sense.

All these trie variants improve the speed only by at most a constant factor. The search acceleration for 32 bits is good, but they will run into performance problems with the migration to IPv6 addresses. These schemes can also exhibit bad update behavior. Many of the faster algorithms cannot be updated

dynamically and require a traditional trie as a helper structure to rebuild the fast search structure from. A good summary can also be found in [SV99b].

3.2.2 Modifications of Exact Matching Schemes

Classical fast lookup techniques such as hashing and binary search have been used to match network addresses for a long time [Jai89, Spi95]. Unfortunately, they only are used for exact matching and do not directly apply to the Best Matching Prefix (BMP) problem. We will present two of these adaptations below.

Binary Search on Prefixes

A modified binary search technique, originally due to Butler Lampson, is described in [Per92] and improved in [LSV98]. There, the overlapping prefixes are expanded to non-overlapping ranges, possibly doubling the number of entries. Then the entries representing either the border between to ranges, or the limit between a range and undefined space are inserted into a large array.

To search, This method requires $\log_2 2N$ steps, with N being the number of routing table entries. With current routing table sizes, the worst case would be 17 data lookups, each requiring at least one costly memory access, not far from the 32 memory accesses for a straightforward implementation of radix tries. Also, modifications to the database require $O(N)$ time. As with any binary search scheme, the average number of accesses is $\log_2(n) - 1$, with an average of one less operation. n is the total number of entries for the binary search, which is $2N$ when searching for prefixes, as each prefix is split into a start and an end entry.

Linear Search on Prefix Lengths

A second classical solution would be to re-apply any exact match scheme for each possible prefix length [Skl93]. This is even more expensive, requiring W iterations of the exact match scheme used (e.g. $W = 128$ for IPv6).

When doing shortest-to-longest search, the result is only clear when the entire table has been traversed, as a better (=longer) prefix may be ahead.

Starting with the longest entry, the first hit can immediately end the search, as there will be nothing better.

3.2.3 Point Location

Finding a given point's enclosing body from a given database is one of the most popular fields in Computational Geometry. A special case are one-dimensional spaces, which could represent the address space. Typically, the one-dimensional structures on which the search is performed, are subdivisions (non-overlapping, contiguous ranges).

Probably the best algorithm for one-dimensional point location is due to Mehlhorn and Näher [MN90], later improved by De Berg et al. [BKS95] and bases on *stratified trees* [Emd75, EKZ77]. A stratified tree is probably best described as a self-similar priority queue, where each node internally has the same structure as the overall queue. Although starting from an entirely different background, the resulting data structure resembles the one used in our basic scheme. Their scheme only works for subdivision and does not support overlapping or priorities, both required properties for IP lookups and longest prefix matching in general.

It seems possible to resolve the overlaps in a pre-processing step, which translates the incoming prefixes into subdivisions. This would degrade their scheme to $O(N)$ update time: A single prefix insertion or deletion requires up to N subdivision insertions or deletions. This scheme is also unable to take advantage of regularities in the (routing) databases, which our algorithm does efficiently, as shown in Section 4.2. Also, preprocessing for multi-dimensional matching results in a possible $O(N^2)$ explosion in memory (and thus update) performance.

3.2.4 Hardware Solutions

Hardware solutions can potentially use parallelism to gain lookup speed. For exact matches, this is done using Content Addressable Memories (CAMs) in which every memory location, in parallel, compares the input key value to the content of that memory location. A natural approach would be to use several CAMs to parallelize the lookups for each prefix length.

Recently, so-called “ternary CAMs” have become available, which allow storing a mask together with the entries. It has been shown that these CAMs can be used to do BMP lookups [MF93, MFTW95]. But these solutions are usually very costly, because CAMs are much slower, more expensive, and several orders of magnitude smaller than conventional memory. Updates in ternary CAMs can also be very costly, since the priority of a filter can only be encoded by its memory address.

Probably the most fundamental problem with CAMs is that CAM designs have not historically kept pace with improvements in RAM memory sizes and speeds, and the speed of general-purpose CPUs. Mostly due to the extensive hardware requirements per memory cell, CAMs provide several orders of magnitudes less storage than conventional memory and one to two orders of magnitude longer access times. Thus, they probably will also be unable to follow the required database growth. Thus a CAM based solution (or indeed *any* hardware solution) runs the risk of being made obsolete, in a few years, by software technology running on faster processors and memory. Known CAMs allow priorization of entries only based on their address. Inserting entries thus requires careful address selection, which seems to require at least $O(N)$ effort, another undesirable factor.

Besides using CAMs, other techniques have been developed, where custom hardware is used to accelerate trie searches. [Eat99] implements a multibit trie heavily tuned for hardware operations, using an application-specific integrated circuit (ASIC) to control dynamic RAM. [ZHMB97] uses hardware to walk a path-compressed trie.

3.2.5 Caching

For years, designers of fast routers have resorted to caching to achieve high speed IP lookups. This is problematic for several reasons. First, information is typically cached on the entire address, potentially diluting the cache with hundreds of addresses that map to the same prefix. Second, a typical backbone router of the future may have hundreds of thousands of prefixes and be expected to forward packets at Multi-gigabit rates. Although studies have shown that caching in campus networks and sometimes even in the backbone can result in hit ratios up to and exceeding 90 percent [Par96, LM97], the simulations of cache behavior were done on large, fully associative caches which commonly are implemented using CAMs. CAMs, as already mentioned, are

usually expensive. It is not clear how set associative caches will perform and whether caching will be able keep up with the growth of the Internet. So caching does help, but does not avoid the need for fast BMP lookups, especially in view of current network speedups.

3.2.6 Protocol Based Solutions

One way to get around the problems of IP lookup is to have extra information sent along with the packet to simplify or even totally get rid of IP lookups at routers. Today's major proposal along these lines is Multi-Protocol Label Switching (MPLS [RVC01]), which evolved from IP Switching [NMH97, LM97] and Tag Switching [CV95, RDK+97]. All these schemes require large, contiguous parts of the network to adopt their protocol changes before they will show a major improvement. The speedup is achieved by adding information on the destination to every IP packet.

In IP Switching [NMH97, LM97], this is done by associating a flow of packets with an ATM Virtual Circuit; in Tag Switching, this is done by adding a "tag" to each packet, where a "tag" is a small integer that allows direct lookup in the router's forwarding table. Tag Switching is based on a concept originally described by Chandranmenon and Varghese ([CV95]) using the name "threaded indices". The current Tag Switching [RDK+97] and Multi-Protocol Label Switching [RVC01] proposals go further than threaded indices by adding a stack of indices to better deal with hierarchies. Nevertheless, to support these hierarchies, much more internal routing information needs to be distributed, which previously was heavily aggregated, to allow backbone scalability. It is therefore unclear whether these schemes will scale well and deliver the performance improvements their inventors expect.

Neither scheme can completely avoid ordinary IP lookups. Both schemes require the ingress router (to the portions of the network implementing their protocol) to perform a full routing decision. In their basic form, both systems potentially require the boundary routers between autonomous systems (e.g., between a company and its ISP or between ISPs) to perform the full forwarding decision again, because of trust issues, scarce resources, or different views of the network. Scarce resources can be ATM VCs or tags, of which only a small amount exist. Thus towards the backbone, they need to be aggregated; away from the backbone, they need to be separated again.

Different views of the network can arise because systems often know more details about their own and adjacent networks, than about networks further away. Although Tag Switching addresses that problem by allowing hierarchical stacking of tags, this affects routing scalability. Tag Switching assigns and distributes tags based on routing information; thus every originating network now has to know structural details about the possible destination networks. Thus while both Tag Switching and IP Switching can provide good performance within a level of hierarchy, neither solution currently does well at hierarchy boundaries without scaling problems.

3.2.7 Network Restructuring

A number of proposals originating in graph theory were put forward recently. They do not address the fast lookup problem but provide some insight in how to model the network and change the routing protocol, if nodes want to keep minimal requiring minimum routing information, yet still delivering well-defined performance.

Interval Routing

[Fre96] proposes to number the network nodes such that routes can be easily aggregated to intervals. Unfortunately, this requires a static network, or alternatively, a system that allows for frequent and automatic renumbering. Even worse is the requirement for the network to be an outerplanar graph: While there may be several short-cut connections, all nodes are required to be arranged in a ring, which is clearly impractical. Other research results along similar lines using on interval routing in specific environments include [SK85, LT86].

Smallest Spanning Trees

A related paper [Fre97] proposes an algorithm for easily maintaining a number of minimum spanning trees, along which can be routed, even if the network is dynamic in its nodes and links. While this is fine for maintaining connectivity, it has the major disadvantage that routing is only allowed along a spanning tree, heavily concentrating all available network traffic along a few links.

While it is possible to maintain multiple spanning trees with this algorithm, the management overhead grows with each additional spanning tree.

Compact Routing

Eilam et al. [EGP98] present an algorithm allowing for compact routing tables. It bases on the idea that the farther away a destination is, the less accurate information has to be known for packets to reach the destination "fast enough", i.e. within a predetermined factor of the optimal. The relative speed is defined as the *stretch*, the ratio between the path selected by their algorithm and the shortest existing path. They prove that their algorithm achieves routing with an average stretch of at most 3 and a worst case stretch of at most 5. Unfortunately, they still require large amounts of memory: $I = 2\sqrt{N(1 + \ln N)}$ intervals need to be stored per link. They do not provide for any network aggregation, so N would be around 30 million in the current Internet. The algorithm also requires complete network renumbering on every topology change. Still, a binary search operation over I times the number of links would be required for each packet, slower than what current algorithms provide.

3.3 Multi-Dimensional Packet Classification

As we have seen in Section 2.3.4, multi-dimensional matching is a specialization of the generic point location problem known from computational geometry. Table 2.6 showed that typical rulesets require five or six dimensions. Unfortunately, the best known general point location algorithms require either space or time exponential to the number of dimensions, which is clearly impractical.

Since some of the possible constraints of the rules for packet classifications are a priori known, a number of adapted solutions have been published recently [SVSW98, LS98, DDPP98, SSV99, GM99, BSW99]. Before publication of these algorithms, linear search along the database was considered state of the art. Since network administrators were not willing to sacrifice speed for some features which were considered mere cosmetics, these tables were generally very small. This led to a kind of self-sustaining prophecy: Since the majority of users didn't need more complex filter databases, no ven-

dor would work on improving the database lookups, further discouraging users to take advantage of filter functions.

But with the upcoming demand for QoS support, this barrier is being broken. Researchers are proposing practical ways to support large filter databases at high speeds, vendors are thinking of incorporating them into their product line, and users start considering to take advantage of them. Unfortunately, the resulting lack of real-world data makes it hard for researchers and vendors to tune their algorithms to the sort of databases that will be used a few years from now.

Most of the current solutions

- require incredible amounts of hardware parallelism: Very wide ternary CAMs (not currently available) or the multi-dimensional scheme in [LS98],
- have slow performance and possibly require some amount of backtracking: Cecilia [FT91], [DDPP98],
- suffer from memory explosion: Cross-producting [SVSW98] (traded off against speed using a caching scheme),
- are limited to two dimensions: Grid-of-tries [SVSW98], [LS98], or
- suffer from a combination of the above problems.

3.4 Summary

In summary, all existing schemes have problems of either performance, scalability, generality, or cost. Lookup schemes based on tries and binary search are (currently) too slow and do not scale well; CAM solutions are expensive and carry the risk of being quickly outdated; Tag and IP Switching solutions require widespread agreement on protocol changes, and still require BMP lookups in portions of the network; finally, locality patterns at backbone routers make it infeasible to depend entirely on caching.

We now describe a scheme that has good performance, excellent scalability, and does not require protocol changes. Our scheme also allows a cheap, fast software implementation, and is also amenable to hardware implementations.

Chapter 4

Scalable Longest Prefix Matching

4.1 Basic Binary Search Scheme

Our basic algorithm is based on three significant ideas:

- First, we use hashing to check whether an address D matches any prefix of a particular length;
- second, we use binary search to reduce number of searches from linear to logarithmic;
- third, we use pre-computation to prevent backtracking in case of failures in the binary search of a range.

Rather than present the final solution directly, we will gradually refine these ideas in Section 4.1.1, Section 4.1.2, and Section 4.1.4 to arrive at a working basic scheme. We describe further optimizations to the basic scheme in the next section.

4.1.1 Linear Search of Hash Tables

Our point of departure is a simple scheme that does linear search of hash tables organized by prefix lengths. We will improve this scheme shortly to do binary search on the hash tables.

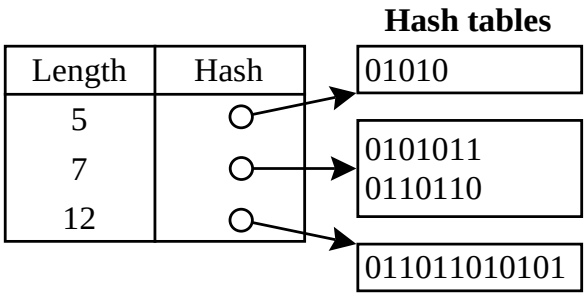


Figure 4.1: *Hash Tables for each possible prefix length*

The idea is to look for all prefixes of a certain length l using hashing and use multiple hashes to find the best matching prefix, starting with the largest value of l and working backwards. Thus we start by dividing the database of prefixes according to lengths. Assuming a particularly tiny routing table with four prefixes of length 5, 7, 7, and 12, respectively, each of them would be stored in the hash table for its length (Figure 4.1). So each set of prefixes of distinct length is organized as a hash table. If we have a sorted array L corresponding to the distinct lengths, we only have 3 entries in the array, with a pointer to the longest length hash table in the last entry of the array.

To search for destination address D, we simply start with the longest length hash table l (i.e. 12 in the example), and extract the first l bits of D and do a search in the hash table for length l entries. If we succeed, we have found the longest match and thus our BMP; if not, we look at the first length smaller than l, say l' (this is easy to find if we have the array L by simply indexing one position less than the position of l), and continuing the search.

4.1.2 Binary Search of Hash Tables

The previous scheme essentially does (in the worst case) linear search among all distinct string lengths. Linear search requires $O(W)$ time (more precisely, $O(W_{dist})$, where $W_{dist} \leq W$ is the number of distinct lengths in the database.)

A better search strategy is to use binary search on the array L to cut down the number of hashes to $O(\log W_{dist})$. However, for binary search to make its branching decision, it requires the result of an ordered comparison, returning whether the probed entry is less than, equal, or greater than our search key. When searching among hash tables, it seems only possible to know the result of this comparison whenever we have found a match containing further information. We will see that the insertion of guiding entries containing such additional branching information, called *markers*, plays an important role.

But where do we need markers, and how many are there? Naïvely, it seems that for every entry, there would be a marker at all other prefix lengths; maybe even multiple markers in the tables associated with longer prefixes, to cover the same range which is covered by a shorter prefix. Recall that shorter prefixes have more "don't care" bits, thus covering a larger area. Luckily, this "marker explosion" does not need to happen when parameters are chosen wisely.

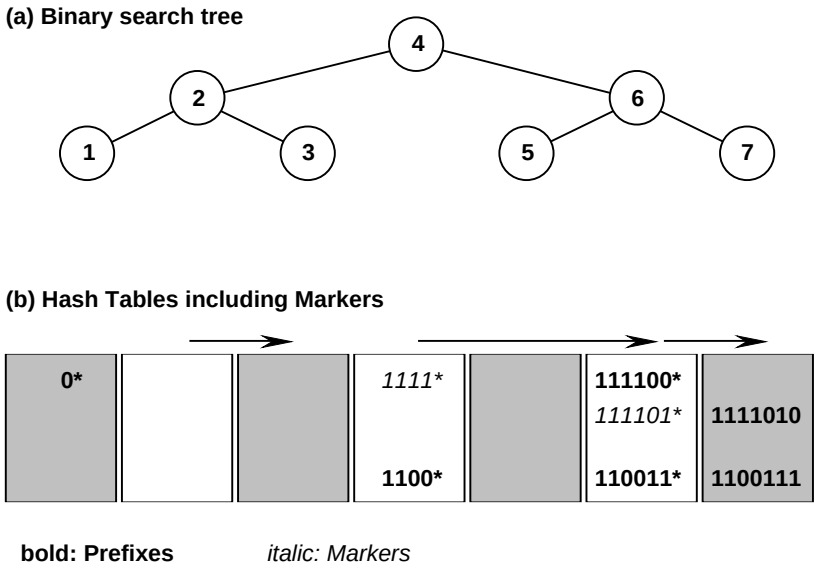


Figure 4.2: *Branching Decisions*

First of all, markers do not need to be placed at all levels. Figure 4.2(a) shows a binary search tree. At each node, a branching decision is made, going to either the left or right subtree, until the correct entry or a leaf node is met. Clearly, at most $\log W$ internal nodes will be traversed on any search, resulting in at most $\log W$ branching decisions. Also, any search that will end up at a given node only has a single path to choose from, eliminating the need to place markers at any other levels.

Secondly, we observe that detecting the *absence* of a marker at any given level may also convey information, this being another key point we rely on. Directing the search by absence or presence of a marker at the current length gives two possibilities: markers could direct the search either towards longer or shorter prefixes.

Shorter Directing from longer towards shorter prefixes would require the placement of many longer prefixes (each covering a smaller area) to cover the whole range of the shorter prefix. Figure 4.2(b) shows some entries (in bold) and the necessary markers (in italics) when branching according to the binary search tree (a). The arrows show which level a successful marker match directs the search to.

Longer Guiding the search the other way round, from shorter towards longer prefixes, as depicted in Figure 4.2(b), does not suffer from this marker multiplication, it even allows multiple prefixes to share the same marker.

To limit memory utilization, we define the comparison rule as follows: finding marker when searching for an entry means that the search has to continue towards longer prefixes, and the absence directs the search towards shorter prefixes. This results in the pseudo-code as illustrated in Figure 4.3 to perform the search. Let L be an array of records, with $L[i]$.*length* being the prefix length of the ith distinct length and $L[i]$.*hash* the corresponding hash table (see also Figure 4.1).

Function NaiveBinarySearch(D) (* search for address D *)
Initialize search range R to cover the whole array L;
While R is not a single entry do
  Let i correspond to the middle level in range R;
  Extract the most significant $L[i]$.*length* bits of D into D';
  Search(D', $L[i]$.*hash*); (* search hash table for D' *)
  If found then set $R \leftarrow$ lower half of R (*longer prefixes*)
    Else set $R \leftarrow$ upper half of R; (*shorter prefixes*)
  Endif
Endwhile

Figure 4.3: *Naïve Binary Search*

4.1.3 Problems with Backtracking

Unfortunately, the algorithm shown in Figure 4.3 is not correct as it stands and *does not* take logarithmic time if fixed straightforwardly. The problem is that while markers are good things (they lead to potentially better prefixes lower in the table), can also cause the search to follow false leads which may fail. In case of failure, we would have to modify the binary search (for correctness) to backtrack and search the upper half of R again. Such a naïve modification can lead us back to linear time search. An example will clarify this.

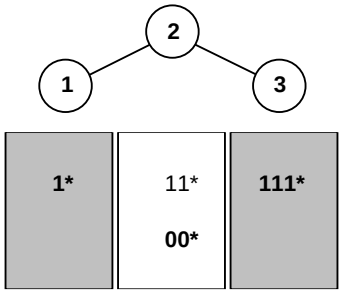


Figure 4.4: *Misleading Markers*

First consider the prefixes $P_1 = 1$, $P_2 = 00$, $P_3 = 111$ (Figure 4.4). As discussed above, we add a marker to the middle table so that the middle hash table contains 00 (a real prefix) and 11 (a marker pointing down to P_3). Now consider a search for 110. We start at the middle hash table and get a hit; thus we search the third hash table for 110 and fail. But the correct best matching prefix is at the first level hash table — i.e., P_1. The marker indicating that there will be longer prefixes, indispensable to find P_3, was misleading in this case; so apparently, we have to go back and search the upper half of the range.

The fact that each entry contributes at most $\log_2 W$ markers may cause some readers to suspect that the worst case with backtracking is limited to $O(\log^2 W)$. This is incorrect. The worst case is $O(W)$. The worst-case example for say W bits is as follows: we have a prefix P_i of length i, for $1 \leq i < W$ that contains all 0s. In addition we have the prefix Q whose first $W - 1$ bits are all zeroes, but whose last bit is a 1. If we search for the W bit address containing all zeroes then we can show that binary search with backtracking will take $O(W)$ time and visit every level in the table. (The problem is that every level contains a false marker that indicates the presence of something better below.)

4.1.4 Pre-computation to Avoid Backtracking

We use pre-computation to avoid backtracking when we shrink the current range R to the lower half of R (which happens when we find a marker at the mid point of R). Suppose every marker node M is a record that contains a variable $M.bmp$, which is the value of the best matching prefix of the marker M. $M.bmp$ can be precomputed when the marker M is inserted into its hash table. Now, when we find M at the mid point of R, we indeed search the lower half, but we also *remember* the value of $M.bmp$ as the current best matching prefix. Now if the lower half of R fails to produce anything interesting, we need not backtrack, because the results of the backtracking are already summarized in the value of $M.bmp$. The new code is shown in Figure 4.5.

Function BinarySearch(D) (* search for address D *)
Initialize search range R to cover the whole array L;
Initialize *BMP* found so far to null string;
While R is not empty do
  Let i correspond to the middle level in range R;
  Extract the first $L[i].$*length* bits of D into D';
  $M \leftarrow$ Search(D', $L[i].$*hash*); (* search hash for D' *)
  If M is nil **Then** set $R \leftarrow$ upper half of R; (* not found *)
  Else-if M is a prefix and not a marker
  Then *BMP* $\leftarrow M.$*bmp*; break; (* exit loop *)
  Else (* M is a pure marker, or marker and prefix *)
    BMP $\leftarrow M.$*bmp*; (* update best matching prefix so far *)
    $R \leftarrow$ lower half of R;
  Endif
Endwhile

Figure 4.5: *Working Binary Search*

The standard invariant for binary search when searching for key K is: "K is in range R". We then shrink R while preserving this invariant. The invariant for this algorithm, when searching for key K is: "EITHER (The Best Matching Prefix of K is BMP) OR (There is a longer matching prefix in R)".

It is easy to see that initialization preserves this invariant, and each of the search cases preserves this invariant (this can be established using an inductive proof.) Finally, the invariant implies the correct result when the range shrinks

to 1. Thus the algorithm works correctly; also since it has no backtracking, it takes $O(\log_2 W_{dist})$ time.

4.2 Refinements to Basic Scheme

The basic scheme described in Section 4.1 takes just 7 hash computations, in the worst case, for 128 bit IPv6 addresses. However, each hash computation takes at least one access to memory; at gigabit speeds each memory access is significant. Thus, in this section, we explore a series of optimizations that exploit the deeper structure inherent to the problem to reduce the average number of hash computations.

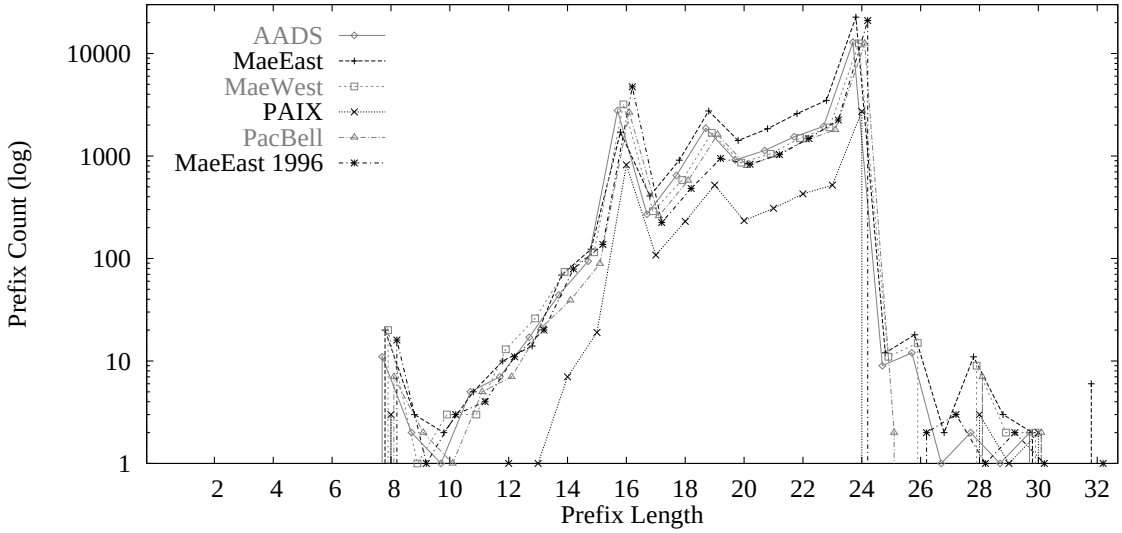


Figure 4.6: *Histogram of Backbone Prefix Length Distributions (log scale)*

4.2.1 Asymmetric Binary Search

We first describe a series of simple-minded optimizations. Our main optimization, mutating binary search, is described in the next section. A reader can safely skip to Section 4.2.2 on a first reading.

The current algorithm is a fast, yet very general, BMP search engine. Usually, the performance of general algorithms can be improved by tailoring them to the particular datasets they will be applied to. Figure 4.6 shows the prefix length distribution extracted from forwarding table snapshots from five major

	Prefixes	W_{dist}	$W_{dist \geq 16}$
AADS	24218	23	15
Mae-East	38031	24	16
Mae-West	23898	22	14
PAIX	5924	17	12
PacBell	22850	20	12
Mae-East 1996	33199	23	15

Table 4.1: *Forwarding Tables: Total Prefixes, Distinct Lengths, and Distinct Lengths longer than 16 bit*

backbone sites in January 1999 and, for comparison, at Mae-East in December 1996 [Int]. As can be seen, the entries are distributed over the different prefix lengths in an extremely uneven fashion. The peak at length 24 dominates everything by at least a factor of ten, if we ignore length 24. There are also more than 100 times as many prefixes at length 24 than at any prefix outside the range $15 \ldots 24$. This graph clearly shows the remnants of the original class A, B, and C networks with local maxima at lengths 8, 16, and 24. This distribution pattern is retained for many years now and seems to be valid for all backbone routing tables, independent of their size (Mae-East has over 38,000, while PAIX has less than 6,000 entries).

These characteristics visibly cry for optimizations. Although we will quantify the potential improvements using these forwarding tables, we believe that the optimizations introduced below apply to any current or future set of addresses.

As the first improvement, which has already been mentioned and used in the basic scheme, the search can be limited to those prefix lengths which do contain at least one entry, reducing the worst case number of hashes from $\log_2 W$ (5 with $W = 32$) to $\log_2 W_{dist}$ ($4.1 \ldots 4.5$ with $W_{dist} \in [17, 24]$, according to Table 4.1). Figure 4.7 applies this to Mae-East's 1996 table. While this numerically improves the worst case, it harms the average performance, since the popular prefix lengths 8, 16, and 24 move to less favorable positions.

A more promising approach is to change the tree structure to search in the most promising prefix length layers first, introducing asymmetry into the binary tree. While this will improve average case performance, introducing asymmetries will not improve the maximum tree height; on the contrary, some

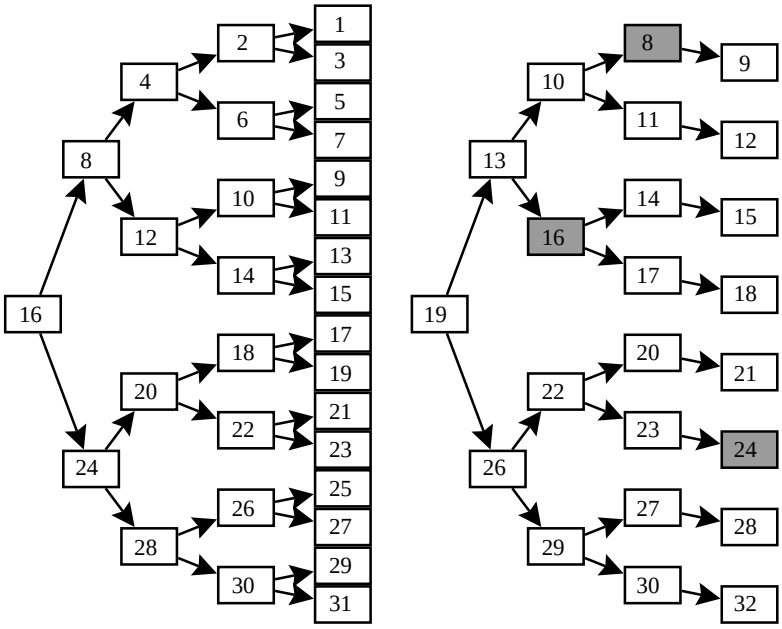


Figure 4.7: *Search Trees for Standard and Distinct Binary Search*

searches will make a few more steps, which has a negative impact on the worst case. Given that routers can temporarily buffer packets, worst case time is not as important as the average time. The search for a BMP can only be terminated early if we have a "stop search here" ("terminal") condition stored in the node. This condition is signalled by a node being a prefix but no marker (Figure 4.5).

Average time depends heavily on the traffic pattern seen at that location. Optimizing binary search trees according to usage pattern is an old problem [Knu98]. By optimizing the average case, some data sets could degenerate towards linear search (Figure 4.8), which is clearly undesirable.

To build a useful asymmetrical tree, we can recursively split both the upper and lower part of the binary search tree's current node's search space, at a point selected by a heuristic weighting function. Two different weighting functions with different goals (one strictly picking the level covering most addresses, the other maximizing the entries while keeping the worst case bound) are shown in Figure 4.8, with coverage and average/worst case analysis for both weighting functions in Table 4.2. As can be seen, balancing gives faster increases after the second step, resulting in generally better performance than "narrow-minded" algorithms.

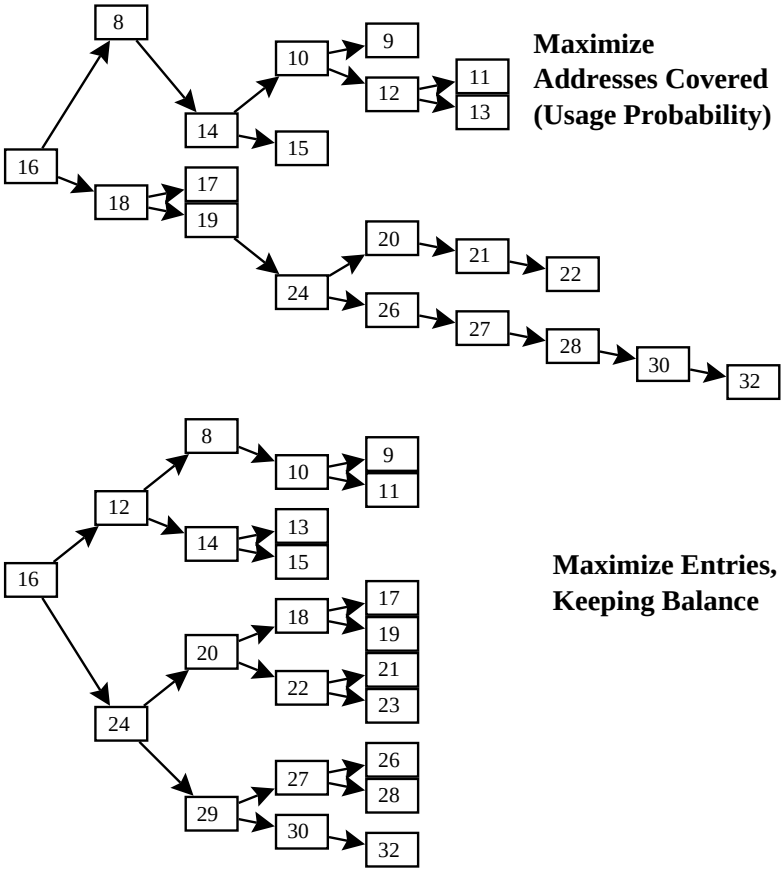


Figure 4.8: *Asymmetric Trees produced by two Weighting Functions*

4.2.2 Mutating Binary Search

In this subsection, we further refine the basic binary search tree to change or *mutate* to more specialized binary trees each time we encounter a partial match in some hash table. We believe this a far more effective optimization than the use of asymmetrical trees though the two ideas can be combined.

Previously, we tried to improve search time based on analysis of prefix distributions sorted by prefix lengths. The resulting histogram (Figure 4.6) led us to propose asymmetrical binary search, which can improve average speed. More information about prefix distributions can be extracted by further dissecting the histogram: For each possible n bit prefix, we could draw 2^n individual histograms with possibly fewer non-empty buckets, thus reducing the depth of the search tree.

Steps	Usage		Balance	
	A	P	A%	P%
1	43%	14%	43%	14%
2	83%	16%	46%	77%
3	88%	19%	88%	80%
4	93%	83%	95%	87%
5	97%	86%	100%	100%
Average	2.1	3.9	2.3	2.4
Worst case	9	9	5	5

Table 4.2: *Address (A) and Prefix (P) Count Coverage for Asymmetric Trees*

	1	2	3	4	5	6	7	8	9
AADS	3467	740	474	287	195	62	11	2	1
Mae-East	2094	702	521	432	352	168	53	8	1
Mae-West	3881	730	454	308	158	70	17	3	—
PAIX	1471	317	139	56	41	31	1	—	—
PacBell	3421	704	442	280	168	42	9	—	—
Mae-East 1996	5051	547	383	273	166	87	27	3	—

Table 4.3: *Histogram of the Number of Distinct Prefix Lengths* ≥ 16 *in the 16 bit Partitions*

When partitioning according to 16 bit prefixes[1], and counting the number of distinct prefix lengths in the partitions, we discover another nice property of the routing data. We recall the whole forwarding databases (Figure 4.6 and Table 4.1) showed up to 24 distinct prefix lengths with many buckets containing a significant number of entries and up to 16 prefix lengths with at least 16 bits. Looking at the sliced data in (Table 4.3), none of these partial histograms contain more than 9 distinct prefixes lengths; in fact, the vast majority only contain one prefix, which often happens to be in the 16 bit prefix length hash table itself. This suggests that if we start with 16 bits in the binary search and get a match, we need only do binary search on a set of lengths that is much smaller than the 16 possible lengths we would have to search in naïve binary search.

[1]There is nothing magic about the 16 bit level, other than it being a natural starting length for a binary search of 32 bit IPv4 addresses.

In general, every match in the binary search with some marker X means that we need only search among the set of prefixes for which X is a prefix. Thus, binary search on prefix lengths has an advantage over conventional binary search: on each branch towards longer prefixes, not only the range of prefix lengths to be searched is reduced, but also the number of prefixes in each of these lengths. Binary search on prefix lengths thus narrows the search in *two dimensions* on each match, as illustrated in Figure 4.9.

Thus the whole idea in mutating binary search is as follows: *whenever we get a match and move to a new subtrie, we only need to do binary search on the levels of new subtrie.* In other words, the binary search *mutates* or changes the levels on which it searches dynamically (in a way that always reduces the levels to be searched), as it gets more and more match information.

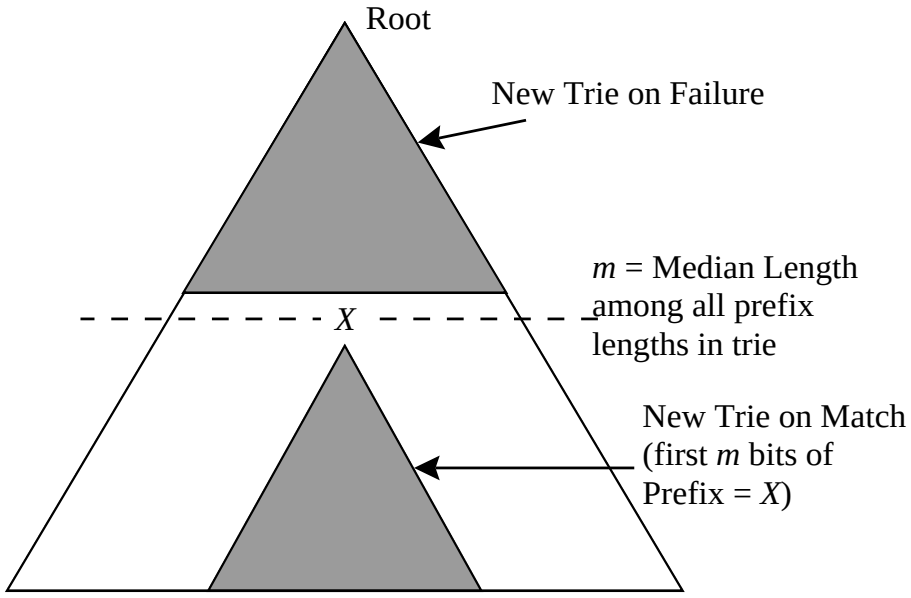


Figure 4.9: *Showing how mutating binary search for prefix P dynamically changes the trie on which it will do binary search of hash tables.*

Thus each entry E in the search table could contain a description of a search tree specialized for all prefixes that start with E. The optimizations resulting from this observation improve lookups significantly:

Worst case: In all the databases we analyzed, we were able to reduce the worst case from five hashes to four hashes.

Average case: In the largest two databases, the majority of the addresses is found in at most two hash lookups. The smaller databases take a little bit longer to reach their halfway point.

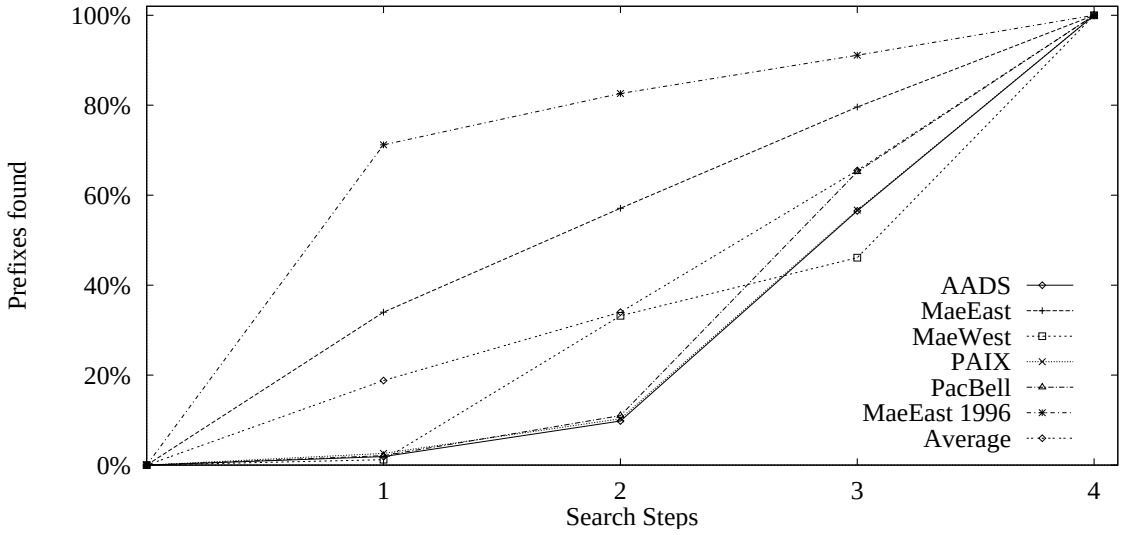


Figure 4.10: *Number of Hash Lookups*

Using Mutating Binary Search, looking for an address (see Figure 4.11) is different. First, we explain some new conventions for reading Figure 4.11. As in the other figures, we continue to draw a binary search tree on top. However, in this figure, we now have multiple partial trees, originating from any prefix entry. This is because the search process will move from tree to tree, starting with overall tree. Each binary tree has the "root" level (i.e., the first length to be searched) at the left; the left child of each binary tree node is the length to be searched on failure, and whenever there is a match, the search switches to the more specific tree.

Consider now a search for address 1100110, matching the prefix labelled B, in the database of Figure 4.11. The search starts with the generic tree, so length 4 is checked, finding A. Among the prefixes starting with A, there are known to be only three distinct lengths (4, 5, and 6). So A contains a description of the new tree, limiting the search appropriately. This tree is drawn as rooting in A. Using this tree, we find B, giving a new tree, the empty tree. The binary tree has *mutated* from the original tree of 7 lengths, to a secondary tree of 3 lengths, to a tertiary empty "tree".

Looking for 1111011, matching G, is similar. Using the overall tree, we find F. Switching to its tree, we miss at length 7. Since a miss (no entry found) can't update a tree, we follow our current tree upwards to length 5, where we find G.

In general, whenever we go down in the current tree, we can potentially move to a specialized binary tree because each match in the binary search is

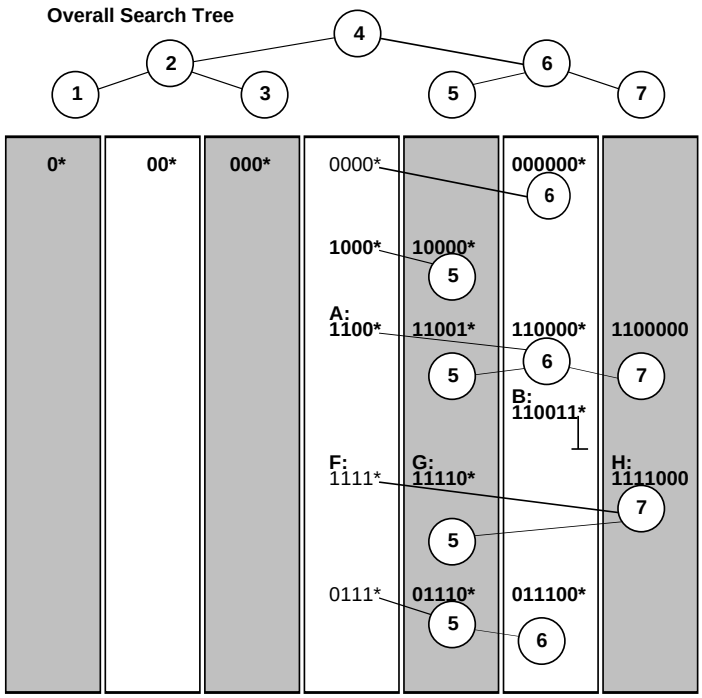


Figure 4.11: *Mutating Binary Search Example*

longer than any previous matches, and hence may contain more specialized information. Mutating binary trees arise naturally in our application (unlike classical binary search) because each level in the binary search has *multiple entries* stored in a hash table. as opposed to a *single entry* in classical binary search. Each of the multiple entries can point to a more specialized binary tree.

In other words, the search is no longer walking through a single binary search tree, but through a whole network of interconnected trees. Branching decisions are not only based on the current prefix length and whether or not a match is found, but also on what the best match so far is (which in turn is based on the address we're looking for.) Thus at each branching point, you not only select which way to branch, but also change to the most optimal tree. This additional information about optimal tree branches is derived by *pre-computation* based on the distribution of prefixes in the current dataset. This gives us a faster search pattern than just searching on either prefix length or address alone.

Two possible disadvantages of mutating binary search immediately present themselves. First, precomputing optimal trees can increase the time to insert a new prefix. Second, the storage required to store an optimal binary

tree for each prefix appears to be enormous. We deal with insertion speed in Chapter 5. For now, we only observe that while the forwarding information for a given prefix may frequently change in cost or next hop, the addition or deletion of a new prefix (which is the expensive case) is be much rarer. We proceed to deal with the space issue by compactly encoding the network of trees.

Bitmap One short encoding method would be to store a bitmap, with each bit set to one representing a valid level of the binary search tree. While this only uses W bits, computing a binary tree to follow next is an expensive task with current processors.

Rope A key observation is that *we only need to store the sequence of levels which binary search on a given subtrie will follow on repeated failures to find a match.* This is because when we get a successful match (see Figure 4.9), we move to a completely new subtrie and can get the new binary search path from the new subtrie. The sequence of levels which binary search would follow on repeated failures is what we call the Rope of a subtrie, and can be encoded efficiently. We call it Rope, because the Rope allows us to swing from tree to tree in our network of interconnected binary search trees.

If we consider a binary search tree, we define the *Rope* for the root of the trie node to be the sequence of trie levels we will consider when doing binary search on the trie levels while failing at every point. This is illustrated in Figure 4.12. In doing binary search we start at Level m which is the median length of the trie. If we fail, we try at the quartile length (say n), and if we fail at n we try at the one-eight level (say o), and so on. The sequence $m, n, o, \ldots$ is the Rope for the trie.

Figure 4.13 shows the Ropes containing the same information as the trees in Figure 4.11. Note that a Rope can be stored using only $\log_2 W$ (7 for IPv6) pointers. Since each pointer needs to only discriminate among at most W possible levels, each pointer requires only $\log_2 W$ bits. For IPv6, 64 bits of Rope is more than sufficient, though it seems possible to get away with 32 bits of Rope in most practical cases. Thus a Rope is usually not longer than the storage required to store a pointer. To minimize storage in the forwarding database, a single bit can be used to decide whether the rope or only a pointer to a rope is stored in a node.

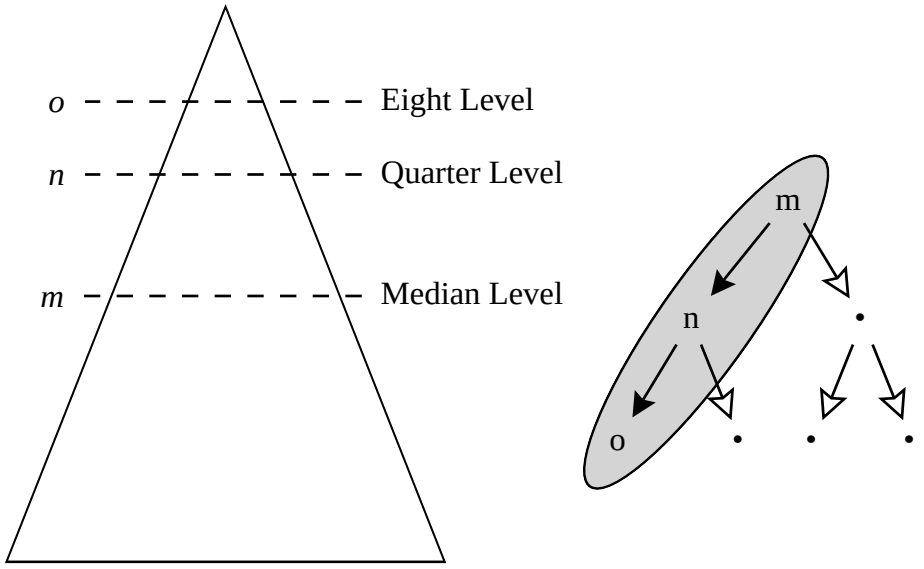


Figure 4.12: *In terms of a trie, a rope for the trie node is the sequence of lengths starting from the median length, the quartile length, and so on, which is the same as the series of left children (see dotted oval in binary tree on right) of a perfectly balanced binary tree on the trie levels.*

Using the Rope as the data structure has a second advantage: it simplifies the algorithm. A Rope can easily be followed, by just picking pointer after pointer in the Rope, until the next hit. Each *strand* in the Rope is followed in turn, until there is a hit (which starts a new Rope), or the end of the Rope is reached. Following the Rope on processors is easily done using "shift right" instructions.

Pseudo-code for the Rope variation of Mutating Binary Search is shown below. An element that is a prefix but not a marker (i.e., the "terminal" condition) specifies an empty Rope, which leads to search termination. The algorithm is initialized with a starting Rope. The starting Rope corresponds to the default binary search tree. For example, using 32 bit IPv4 addresses, the starting Rope contains the starting level 16, followed by Levels 8, 4, 2, 1. The Levels 8, 4, 2, and 1 correspond to the "left" pointers to follow when no matches are found in the default tree. The resulting pseudo-code (Figure 4.14) is elegant and simple to implement. It appears to be simpler than the basic algorithm.

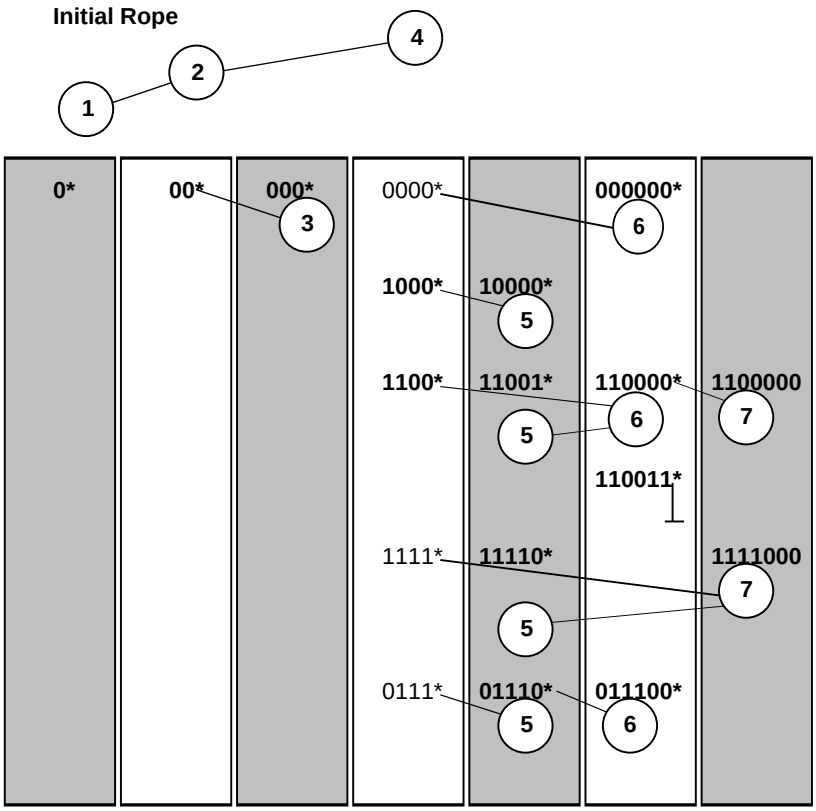


Figure 4.13: *Sample Ropes*

4.2.3 Using Arrays

In cases where program complexity and memory use can be traded for speed, it might be desirable to change the first hash table lookup to a simple indexed array lookup, with the index being formed from the first w_0 bits of the address, with w_0 being the prefix length at which the search would be started. For example, if $w_0 = 16$, we would have an array for all possible 2^{16} values of the first 16 bits of a destination address. Each array entry for index i will contain the *bmp* of i as well as a Rope which will guide binary search among all prefixes that begin with i. An initial array lookup is not only faster than a hash lookup, but also results in reducing the average number of lookups, since there will be no misses at the starting level, which could direct the search below w_0.

4.2.4 Halving the Prefix Lengths

It is possible to reduce the worst case search time by another memory access. For that, we halve the number of prefix lengths by e.g. only allowing all the

```
Function RopeSearch(D) (* search for address D *)
Initialize Rope R containing the default search sequence;
Initialize BMP so far to null string;
While R is not empty do
   Pull the first strand (pointer) off R and store it in i;
   Extract the first L[i].length bits of D into D';
   M ← Search(D', L[i].hash); (* search hash table for D' *)
   If M is not nil then
      BMP ← M.bmp; (* update best matching prefix so far *)
      R ← M.rope; (* get the new Rope, possibly empty *)
   Endif
Endwhile
```

Figure 4.14: *Rope Search*

even prefix lengths, decreasing the $\log W$ search complexity by one. All the prefixes with odd lengths would then be expanded to two prefixes, each one bit longer. For one of them, the additional bit would be set to zero, for the other, to one. Together, they would cover the same range as the original prefix. At first sight, this looks like the memory requirement will be doubled. It can be shown that the worst case memory consumption is not affected, since the number of markers is reduced at the same time.

With W bits length, each entry could possibly require up to $\log(W) - 1$ markers (the entry itself is the $\log W$th entry). When expanding prefixes as described above, some of the prefixes will be doubled. At the same time, W is halved, thus each of the prefixes requires at most $\log(W/2) - 1 = \log(W) - 2$ markers. Since they match in all but their least bit, they will share all the markers, resulting again in at most $\log W$ entries in the hash tables.

A second halving of the number of prefixes again decreases the worst case search time, but this time increases the amount of memory, since each prefix can be extended by up to two bits, resulting in four entries to be stored, expanding the maximum number of entries needed per prefix to $\log(W)+1$. For many cases the search speed improvement will warrant the small increase in memory.

4.2.5 Very Long Addresses

All the calculations above assume the processor's registers are big enough to hold entire addresses. For long addresses, such as those used for IP version 6, this does not always hold. We define w as the number of bits the registers hold. Instead of working on the entire address at once, the database is set up similar to a multibit trie [SV99a] of stride w, resulting in a depth of $k := W/w$. Each of these "trie nodes" is then implemented using binary search. If the "trie nodes" used conventional technology, *each of them* would require $O(2^w)$ memory, clearly impractical with modern processors, which manipulate 32 or 64 bits at a time.

Slicing the database into chunks of w bits also requires less storage than unsliced databases, since not the entire long addresses do not need to be stored with every element. The smaller footprint of an entry also helps with hash collisions (Section 5.5).

This storage advantage comes at a premium: Slower access. The number of memory accesses changes from $\log_2 W$ to $k + \log_2 w$, if the search in the intermediate "trie nodes" begins at their maximum length. This has no impact on IPv6 searches on modern 64 bit processors (Alpha, UltraSPARC, Itanium), which stay at 7 accesses. For 32 bit processors, the worst case using the basic scheme raises by 1, to 8 accesses. Generally, as long as $W/w < 4$, which is the case for IPv6 addresses and modern processors, this does not increase the number of worst-case steps; average-case scenarios do not seem to be impaired either. Only at $W/w = 4$, a first additional memory access becomes necessary.

4.2.6 Internal Caching

Figure 4.6 showed that the prefixes with lengths 8, 16, and 24 cover most of the address space used. Using binary search, these three lengths can be covered in just two memory accesses. To speed up the search, each address that requires more than two memory accesses to search for will be cached in one of these address lengths according to Figure 4.15. Compared to traditional caching of complete addresses, these cache prefixes cover a larger area and thus allow for a better utilization.

Function CacheInternally(A, P, L, M)
(* found prefix P at length L after M memory accesses searching for A *)
If $M > 2$ then (* Caching can be of advantage *)
    Round up prefix length L to next multiple of 8;
    Insert copy of P's entry at L, using L most significant bits of A;
Endif

Figure 4.15: *Building the Internal Cache*

4.3 Special Cases

The previous sections have always implied that for an address length of W bits there are W distinct prefix lengths and that binary searching them requires $\log_2 W$ steps. Both statements are incorrect. Nevertheless, assuming their validity both simplified the explanations and represents the behavior seen. Note that the modifications below do not change any of the $O(\cdot)$ results.

4.3.1 Prefix Lengths

Instead of W distinct prefix lengths for addresses of length W, the maximum is in fact $W + 1$, since all prefix lengths from 0 to W, inclusive, are possible. But prefix length 0 is special. With a prefix length of 0, there are 0 bits of information to distinguish between the entries of prefix length 0. Thus, there can be at most one of these so-called *default* entries in the database. This lends itself to special treatment.

Even if there is no default route in the database, this entry is nevertheless recommended, especially in modular router designs, such as the BBN GigaRouter [PC$^+$98]. In this router, the processor responsible for special packet treatment, such as sending a "Host unreachable" ICMP control message back to the sender is addressed in the same way as output interfaces. Thus it is possible to remove this test from the fast path. Thus it is recommended to always have a default entry, but not store it in a hash table, but refer to it on search failures.

Additionally, the CIDR specification [RL93, FLYV93] for IPv4 does not introduce the notion of further aggregating ("supernetting") the classical class

A networks with eight bit prefix length. Therefore, the set of valid prefix lengths is $\{0, 8 \ldots 32\}$, resulting in only 26 *allowed* distinct prefix lengths, out of 33 possible. Most databases do contain even less than that.

A similar line of thought applies to IPv6. There will be $W + 1$ prefix lengths (W is 128 for IPv6), but prefixes shorter than 3 are not used. IPv6 provides an option to use addressing schemes for other protocols in a "compatibility" mode. It is unclear whether this option is ever going to be used. Even then, these non-native addresses will most likely only be used in some restricted environments. Assuming native-only addressing, the interesting prefix lengths start at 16, and most likely there will be no prefixes with lengths between 64 and 128, resulting in valid prefixes of $\{0, 16 \ldots 64, 128\}$, or a total of just 51 prefix lengths. The large gap between 64 and 128 is a natural fit for hashing. The 128-bit addresses will be distributed sparsely within their corresponding 64-bit prefixes. Tree-based algorithms will thus perform badly; path compression may not perform well (a possible branch at each prefix length), and level compression is unsuitable, since it would lead to (possibly multiple) tables of 2^{64} addresses each, clearly impractical.

4.3.2 Binary Search Steps

Each binary search step tests the middle level and then goes on to proceed in the *remaining* half, excluding the middle level. Thus, the maximum number of levels covered c after v search steps is $C(v) = 1 + 2C(v-1)$, with $C(1) = 1$. In closed form, this equals $C(v) = 2^v - 1$, or $v = \lceil \log_2(C+1) \rceil$. Thus, 32 distinct prefix lengths would already require 6 steps in the worst case, only below lengths, 5 steps would be sufficient. As we have seen above, in reality, this would not make a difference. And we also considered it unfair, since the improvements of asymmetric or Rope search, such as seen in Figure 4.10 would look better than they really are.

4.4 Summary

This chapter developed a working binary search on prefix lengths. It then introduced asymmetric binary search and analyzed its impact. Then this was improved to Rope search, which refines the search after each search step,

adapting snuglily to the database. Further modifications such as using an initial array and halving the number of prefix lengths were presented. A closer analysis on the number of prefix lengths and search step concluded this chapter.

Chapter 5

Building and Updating

Having fast searching readily available is key. A necessary requirement is to have algorithms ready to build the data structures used in searching. Besides that, it is highly desirable to provide for quick builds, dynamic updates, and a small memory footprint. This chapter will present solutions to all these issues.

A predominant idea used throughout this thesis is *pre-computation.* Every hash table entry has an associated *bmp* field and (possibly) a Rope field, both of which are precomputed. Pre-computation allows fast search but requires more complex Insertion routines. However, as mentioned earlier, while the routes stored with the prefixes may change frequently, the addition of a new prefix (the expensive case) is much rarer. Thus it is worth paying a penalty for Insertion in return for improved search speed, especially if that penalty is as low as we will see below.

5.1 Basic Scheme Built from Scratch

Setting up the data structure for the Basic Scheme is straightforward, as shown in Figure 5.1, requiring a complexity of $O(N \log W)$. For simplicity of implementation, the list of prefixes is assumed to be sorted by increasing prefix length in advance ($O(N)$ using bucket sort). For optimal search performance, the final hash tables should ensure minimal collisions (see Section 5.5).

```
Function BuildBasic;
For all entries in the sorted list do
    Read next pair (Prefix, Length) from the list;
    Let Index be the index for the Length's hash table;
    Use Basic Algorithm on what has been built by now
        to find the BMP of Prefix and store it in BMP;
    Add a new prefix node for Prefix in the hash table for Index;
    (* Now insert all necessary markers "to the left" *)
    For ever do
        (* Go up one level in the binary search tree *)
        Clear the least significant one bit in Index;
        If Index = 0 then break; (* end reached *)
        Set Length to the appropriate length for Index;
        Shorten Prefix to Length bits;
        If there is already an entry for Prefix at Index then
            Make it a marker if it isn't already;
            break; (* higher levels already do have markers *)
        Else
            Create a new marker Prefix at Index' hash table;
            Set it's bmp field to BMP;
        Endif
    Endfor
Endfor
```

Figure 5.1: *Building for the Basic Scheme*

To build a basic search structure which eliminates unused levels or to take advantage of asymmetries, it is necessary to build the binary search tree first. Then, instead of clearing the least significant bit, as outlined in Figure 5.1, the build algorithm really has to follow the binary search tree back up to find the "parent" prefix length. Some of these parents may be at longer prefix lengths, as illustrated in Figure 4.2. Since markers only need to be set at shorter prefix lengths, any parent associated with longer prefixes is just ignored.

5.2 Rope Search from Scratch

There are two ways to build the data structure suitable for Rope Search:

Simple: The search order does not divert from the overall binary search tree, only missing levels are left out. This results in only minor improvements on the search speed and can be implemented as a straightforward enhancement to Figure 5.1.

Optimal: Calculating the shortest Ropes on all branching levels requires the solution to an optimization problem in two dimensions. As we have seen, each branch towards longer prefix lengths also limits the set of remaining prefixes.

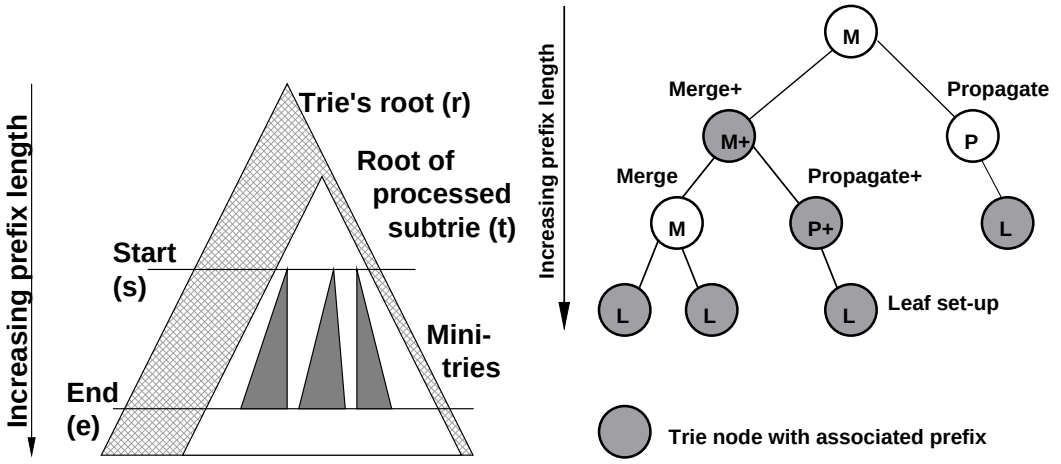
We present the algorithm which globally calculates the minimum Ropes, based on dynamic programming. The algorithm can be split up into three main phases:

1. Build a conventional (uncompressed) trie structure with $O(NW)$ nodes containing all the prefixes ($O(NW)$ time and space).
2. Walk through the trie bottom-up, calculating the cost of selecting different branching points and combining them on the way up using dynamic programming ($O(NW^3)$ time and space).
3. Walk through the trie top-down, build the Ropes using the results from phase 2, and insert the entries into the hash tables ($O(NW \log W)$ time, working on the space allocated in phase 2).

To understand the bottom-up merging of the information in phase 2, let us first look at the information that is necessary for bottom-up merging. Recall the Ropes in Figure 4.13. At each branching point, the search either turns towards longer prefixes and a more specific branching tree, or towards shorter prefixes without changing the set of levels. The goal is to minimize worst-case search cost, or the number of hash lookups required. The overall cost of putting a decision point at prefix length x is the maximum path length on either side plus one for the newly inserted decision. Looking at Figure 4.13, the longest path on the left of our starting point has length two (the paths to $0*$ or $000*$). When looking at the right hand side, the longest of the individual searches require two lookups ($11001*$, 1100000, $11110*$, and 0111000).

Generalizing, for each range R covered and each possible prefix length x splitting this range into two halves, R_l and R_r, the program needs to calculate the maximum depth of the *aggregate* left-hand tree R_l, covering shorter prefixes, and the maximum depth of the *individual* right-hand trees R_r. When

trying to find an optimal solution, the goal is to minimize these maxima, of course. Clearly, this process can be applied recursively. Instead of implementing a simple-minded recursive algorithm in exponential time, we use dynamic programming to solve it in polynomial time.



(a) Structures (b) Cases treated

Figure 5.2: *Rope Construction, Phase 2*

Figure 5.2(a) shows the information needed to solve this minimization problem. For each subtree t matching a prefix P, a table containing information about the depth associated with the subrange R ranging from start length s to end length e is kept. Specifically, we keep (1) the maximum over all the *individual* minimal-depth trees (T_I), as used for branching towards longer prefixes and (2) the minimal *aggregate* tree (T_A), for going to shorter prefixes. Each of these trees in turn consists of both a left-hand aggregate tree and right-hand individual branching trees.

Using the dynamic programming paradigm, we start building a table (or in this case, a table per trie node) from the bottom of the trie towards the root. At each node, we combine the information the children have accumulated with our local state, i.e. whether this node is an entry. Five cases can be identified: (L) setting up a leaf node, (P) propagating the aggregate/individual tables up one level, (P+) same, plus including the fact that this node contains a valid prefix, (M) merging the child's aggregate/individual tables, and (M+) merging and including the current node's prefix. As can be seen, all operations are a subset of (M+), working on less children or not adding the current node's prefix. Figure 5.3 lists the pseudo-code for this operation.

```
Function Phase2MergePlus;
Set p to the current prefix length;

(* Merge the children's T_I below p *)
Forall s, e where s ∈ [p + 1 ... W], e ∈ [s ... W];
   (* Merge the T_I mini-trees between Start s and End e *)
   If both children's depth for T_I[s, e] is 0 then
      (* No prefixes in either mini-tree *)
      Set this node's depth for T_I[s, e] to 0;
   Else
      Set this node's depth for T_I[s, e] to the
         the max of the children's T_I[s, e] depths;
   Endif
Endforall

(* "Calculate" the depth of the trees covering just this node *)
If the current entry is a valid prefix then
   Set T_I[p, p] = T_A[p, p] = 1; (* A tree with a single entry *)
Else
   Set T_I[p, p] = T_A[p, p] = 0; (* An empty tree *)
Endif

(* Merge the children's T_A, extend to current level *)
For s ∈ [p ... W];
   For e ∈ [s + 1 ... W];
      (* Find the best next branching length i *)
      Set T_A[s, e]'s depth to min(T_I[s + 1, e] + 1), (* split at s *)
         min_{i=s+1}^{e}(max(T_A[s, i − 1] + 1, T_I[i, e])); (* split below *)
      (* Since T_A[s, i − 1] is only searched after missing at i, add 1 *)
   Endfor
Endfor

(* "Calculate" the T_I at p also *)
Set T_I[p, ∗] to T_A[p, ∗; (* Only one tree, so aggregated=individual *)
```

Figure 5.3: *Phase 2 Pseudo-code, run at each trie node*

As can be seen from Figure 5.3, merging the T_As takes $O(W^3)$ time per node, with a total of $O(NW)$ nodes. The full merging is only necessary at nodes with two children, shown as (M) and (M+) in Figure 5.2(b). In any trie, there can be only $O(N)$ of them, resulting in an overall build time of only $O(NW^3)$.

If the optimal next branching point is stored alongside each $T_A[s, e]$, building the rope for any prefix in Phase 3 is a simple matter of following the chain set by these branching points, by always following $T_A[s_{\mathit{prev}} + 1, \mathit{previous\ branching\ point}]$. A node will be used as a marker, if the higher-level rope lists its prefix length.

5.2.1 Degrees of Freedom

The only goal of the algorithm shown in Figure 5.3 is to minimize the worst-case number of search steps. Most of the time multiple branching points will result in the same minimal T_A depth. Therefore, choosing the split point gives a further degree of freedom to optimize other factors within the bounds set by the calculated worst case. This freedom can be used to (1) reduce the number of entries requiring the worst case lookup time, (2) improve the average search time, (3) reduce the number of markers placed, (4) reduce the number of hash collisions, or (5) improve update behavior (see below). Because of limitations in space and scope, they will not be discussed in more depth.

5.3 Insertions and Deletions

As shown in [LMJ97], some routers receive routing update messages at high frequencies, requiring the routers to handle these messages within a few milliseconds. Luckily for the forwarding tables, most of the routing messages in these bursts are of pathological nature and do not require any change in the routing or forwarding tables. Also, most routing updates involve only a change in the route and do not add or delete prefixes. Additionally, many wide-area routing protocols such as BGP [RL95] use timers to reduce the rate of route changes, thereby delaying and batching them. Nevertheless, algorithms in want of being ready for further Internet growth should support sub-second updates under most circumstances.

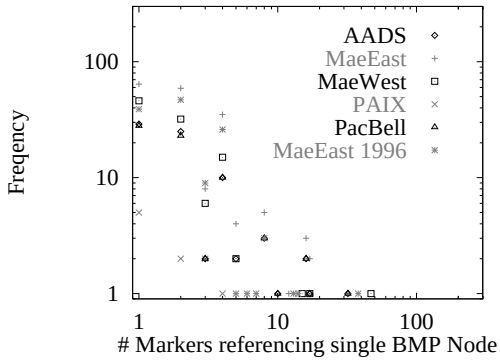
Adding entries to the forwarding database or deleting entries may be done without rebuilding the whole database. The less optimized the data structure is, the easier it is to change it.

5.3.1 Updating Basic and Asymmetric Schemes

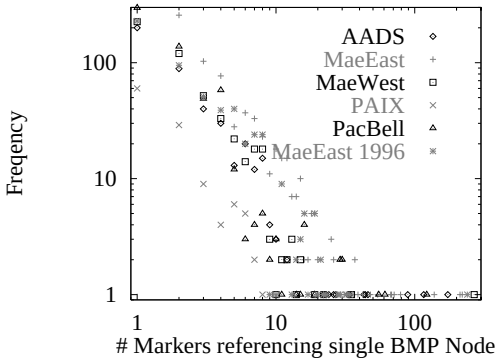
We therefore start with basic and asymmetric schemes, which have only eliminated prefix lengths which will never be used. Insertion and deletion of leaf prefixes, i.e. prefixes, that do not cover others, is trivial. Insertion is done as during initial build (Figure 5.1). For deletion, a simple possibility is to just remove the entry itself and not care for the remaining markers. When unused markers should be deleted immediately, it is necessary to maintain per-marker reference counters. On deletion, the marker placement algorithm from Figure 5.1 is used to determine where markers would be set, decreasing their reference count and deleting the marker when the counter reaches zero.

Should the prefix p being inserted or deleted cover any markers, these markers need to be updated to point to their changed BMP. There are a number of possibilities to find all the underlying markers. One that does not require any helper data structures, but lacks efficiency, is to either enumerate all possible longer prefixes matching our modified entry, or to walk through all hash tables associated with longer prefixes. On deletion, every marker pointing to p will be changed to point to p's BMP. On insertion, every marker pointing p's current BMP and matching p will be updated to point to p. A more efficient solution is to chain all markers pointing to a given BMP in a linked list. Still, this method could require $O(N \log W)$ effort, since p can cover any amount of prefixes and markers from the entire forwarding database. Although the number of markers covered by any given prefix was small in the databases we analyzed (see Figure 5.4), Section 5.4 presents a solution to bound the update efforts, which is important for applications requiring real-time guarantees.

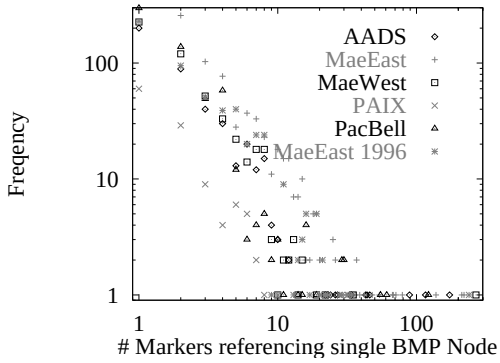
During the previous explanation, we have assumed that the prefix being inserted had a length which was already used in the database. In Asymmetric Search, this may not always be true. Depending on the structure of the binary search trie around the new prefix length, adding it is trivial. The addition of length 5 in Figure 5.5(a) is one of these examples. One possibility, shown in the upper example, is to re-balance the trie structure, which unlike balancing a B-tree can result in several markers being inserted: One for each pre-existing prefix not covered by our newly inserted prefix, but covered by its parent. This structural change can also adversely affect the average case behavior. Another possibility, shown in the lower right, is to immediately add the new prefix length, possibly increasing the worst case for this single prefix. Then we wait for a complete rebuild of the tree which takes care of the correct re-balancing.



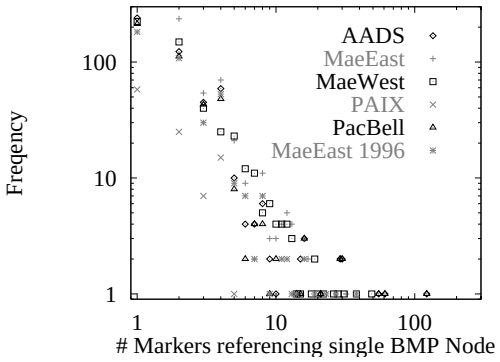
(a) "Pure Basic" (without Length Elimination)



(b) Basic



(c) Asymmetric



(d) Rope

Figure 5.4: *Histogram of Markers depending on a Prefix (log scales)*

We prefer the second solution, since it does not need more than the plain existing insertion procedures. It also allows for updates to take effect immediately, and only incurs a negligible performance penalty until the database has been rebuilt. To reduce the frequency of rebuilds, the binary search tree may be constructed as to leave room for inserting the missing prefix lengths at minimal cost. A third solution would be to split a prefix into multiple longer prefixes, similar to the one used by Causal Collision Resolution Section 5.5.1.

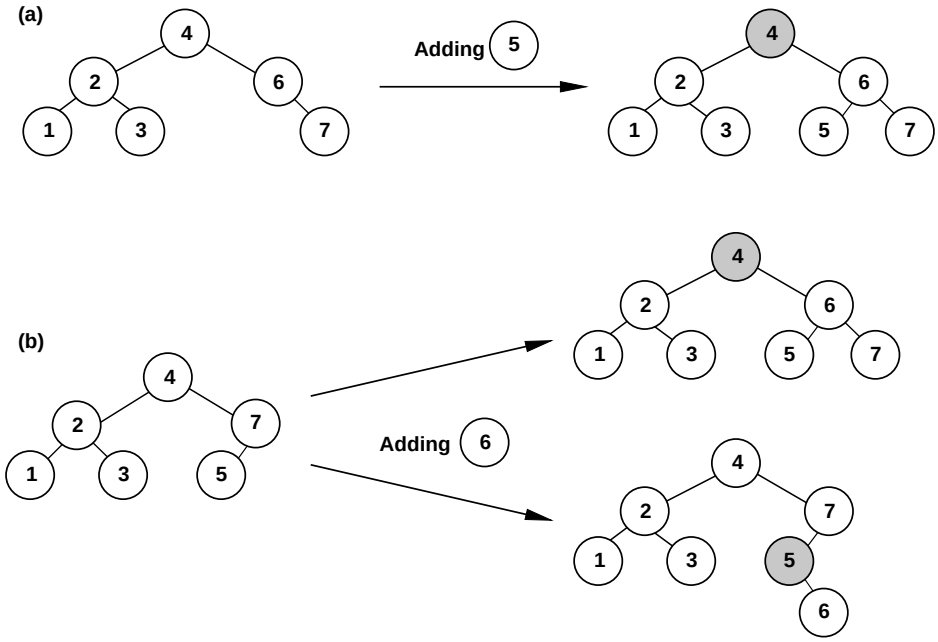


Figure 5.5: *Adding Prefix Lengths*

5.3.2 Updating Ropes

All the above insights also apply to Rope Search, and even more so, since it uses many local asymmetric binary search trees, containing a large number of uncovered prefix lengths. Inserting a prefix has a higher chance of adding a new prefix length to the current search tree, but it will also confine the necessary re-balancing to a small subset of prefixes. Therefore, we believe the simplest, yet still very efficient, strategy is to add a marker at the longest prefix length shorter than p's, pointing to p. If this should degrade the worst-case search time, or anyway after a large number of these insertions, a background rebuild of the whole structure is ordered. The overall calculation of the optimal branching points in phase 2 (Figure 5.3) is very expensive, $O(NW^3)$, far more expensive than calculating the ropes and inserting the entries Table 5.1. Just recalculating to incorporate the changes induced by a routing update is much cheaper, as only the path from this entry to the root needs to be updated, at most $O(W^4)$, giving a speed advantage over simple rebuild of around three orders of magnitude. Even though Rope Search is optimized to very closely fit around the prefix database, Rope Search still keeps enough flexibility to quickly adapt to any of the changes of the database.

The times in Table 5.1 were measured using completely unoptimized code on a 300 MHz UltraSparc-II. We would expect large improvements from opti-

	Basic	Rope			Entries
	Hash	Phase 2	Ropes	Hash	
AADS	0.56s	11.84s	0.59s	0.79s	24218
Mae-East	1.82s	14.10s	0.85s	1.69s	38031
Mae-West	0.58s	11.71s	0.60s	0.85s	23898
PAIX	0.09s	4.16s	0.18s	0.07s	5924
PacBell	0.48s	11.04s	0.57s	0.73s	22850
Mae-East 1996	1.14s	13.08s	0.75s	1.12s	33199

Table 5.1: *Build Speed Comparisons (Built from Trie)*

mizing the code. “Hash” refers to building the hash tables, “Phase 2” is phase 2 of the rope search, “Ropes” calculates the ropes and sets the markers. Just adding or deleting a single entry takes orders of magnitudes less time.

5.4 Marker Partitioning

The scheme introduced below, *recursive marker partitioning*, significantly reduces the cost of marker updates identified as a problem above. It does this by requiring at most one additional memory access per entire search, whenever the last match in the search was on a marker. Using rope search on the examined databases, an additional memory lookup is required for $2 \ldots 11\%$ of the addresses, a negligible impact on the average search time. Of the searches that require the identified worst case of four steps, only $0 \ldots 2\%$ require an additional fifth memory access.

Furthermore, prefix partitioning offers a tunable tradeoff between the penalty incurred for updates and searches, which makes it very convenient for a wide range of applications.

5.4.1 Basic Partitioning

To understand the concept and implications of partitioning, we start with a single layer of partitions. Assume an address space of 4 bits with addresses

ranging from 0 to 15, inclusive. This space also contains nine markers, labeled $a1$ to $c3$, as shown in Figure 5.6. For simplicity, the prefixes themselves are not shown. Recall that each marker contains a pointer to its BMP. This information requires update whenever the closest covering prefix is changed.

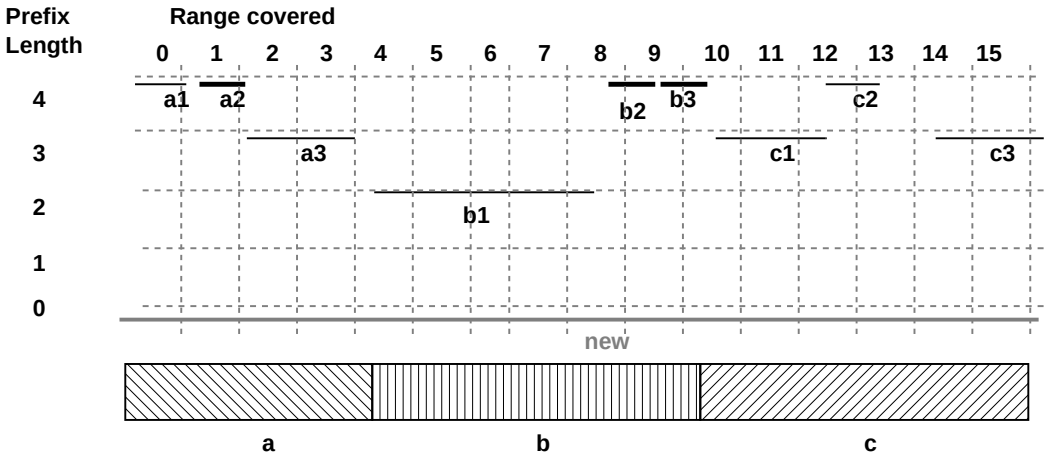


Figure 5.6: *Simple Partitioning Example*

Assume the prefix designated *new* is inserted. Traditional approaches would require the insert procedure to walk through all the markers covered by *new* and correct their BMP, taking up to $N \log W$ steps. *Marker partitioning* groups these markers together. Assume we had grouped markers a_1 to a_3 in group a, markers b_1 to b_3 in b, and c_1 to c_3 in c. Note that the prefixes in the group are disjoint and hence, we can store a single overlapping BMP pointer information for all of them instead of at each of them individually. Thus, in this example, we would remember only three such entries — one per group or partition. This improves the time required from updating each entry to just modifying the information common to the group. In our example above (Figure 5.6), when adding the *new* prefix, we see that it entirely covers the partitions a, b and c. Thus, our basic scheme works well as long as the partition boundaries can be chosen so that no marker overlaps them and the new prefix covers entire groups.

But when looking at one more example in Figure 5.7, where partition A contains markers a_1, a_2, a_3, partition B contains b_1, b_2, b_3 and partition C contains c_1, c_2, c_3. Clearly, the partition boundaries now overlap. Although in this example it is possible to find partitionings without overlaps, prefixes covering a large part of the address space would severely limit the ability to find enough partitions. Thus, in the more general case, the boundaries between the splits are no longer well-defined; there are overlaps. Because of the nature of prefix-

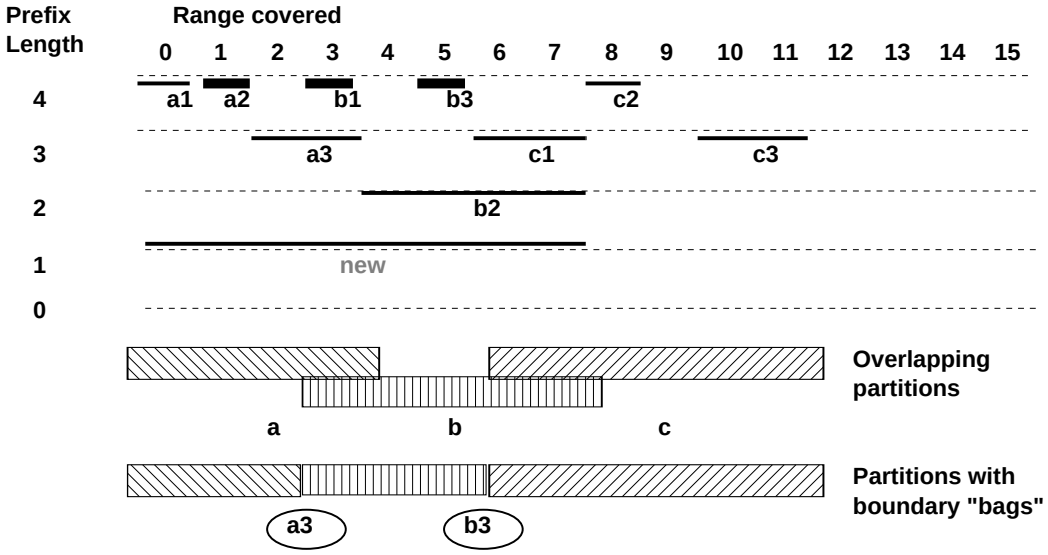


Figure 5.7: *Partitions with Overlaps*

style ranges, at most W distinct ranges may enclose any given point. This is also true for the markers crossing boundary locations. So at each boundary, we could store the at most W markers that overlap it and test against these special cases individually when adding or deleting a prefix like *new*. It turns out to be enough to store these overlapping markers at only a single one of the boundaries it crosses. This is enough, since its *bmp* will only need to change when a modification is done to an entry covering our prefix.

For simplicity of the remaining explanations in this section, it is assumed that it is possible to split the prefixes in a non-overlapping fashion. One way to achieve that would be to keep a separate marker partition for each prefix length. Clearly, this separation will not introduce any extra storage and the search time will be affected by at most a factor of W.

Continuing our example above (Figure 5.7), when adding the *new* prefix, we see that it entirely covers the partitions a, b and partially covers c. For all the covered partitions, we update the partitions' Best Match. Only for the partially covered partitions, we need to process their individual elements. The changes for the BMP pointers are outlined in bold in the Table 5.2. The real value of the BMP pointer is the entry's value, if it is set, or the partition's value otherwise. If neither the entry nor the entry's containing partition contain any information, as is the case for c_3, the packet does not match a prefix (filter) at this level.

Entry/Group	Old BMP stored	New BMP stored	Resulting BMP
a_1	—	—	new
a_2	—	—	new
a_3	—	—	new
a	—	new	(N/A)
b_1	a_3	a_3	a_3
b_2	—	—	new
b_3	b_2	b_2	b_2
b	—	new	(N/A)
c_1	—	new	new
c_2	—	—	—
c_3	—	—	—
c	—	—	(N/A)

Table 5.2: *Updating Best Matching Prefixes*

Generalizing to p partitions of e markers each, we can see that any prefix will cover at most p partitions, requiring at most p updates.

At most two partitions can be partially covered, one at the start of the new prefix, one at the end. In a simple-minded implementation, at most e entries need to be updated in each of the split partitions. If more than $e/2$ entries require updating, instead of updating the majority of entries in this partition, it is also possible to relabel the container and update the minority to store the container's original value. This reduces the update to at most $e/2$ per partially covered marker, resulting in a worst-case total of $p + 2e/2 = p + e$ updates.

As $p * e$ was chosen to be N, minimizing $p + e$ results in $p = e = \sqrt{N}$. Thus, the optimal splitting solution is to split the database into $\sqrt{N}$ sets of $\sqrt{N}$ entries each. This reduces update time from $O(N)$ to $O(\sqrt{N})$ at the expense of at most a single additional memory access during search. This memory access is needed only if the entry does not store its own BMP value and we need to revert to checking the container's value.

5.4.2 Dynamic Behavior

Insertion and deletion of prefixes often goes ahead with the insertion or deletion of markers. Over time, the number of elements per partition and also in the total number of entries, N, will change. The implications of these changes are discussed below. For readability, S will be used to represent $\sqrt{N}$, the optimal number of partitions and entries per partition.

The naïve solution of re-balancing the whole structure is to make all partitions equal size after every change to keep them between $\lfloor S \rfloor$ and $\lceil S \rceil$. This can be done by 'shifting' entries through the list of partitions in $O(S)$ time. This breaks as soon as the number of partitions needs to be changed when S crosses an integer boundary. Then, $O(S)$ entries need to be shifted to the partition that is being created or from the partition that is being destroyed, resulting in $O(N)$ entries to be moved. This obviously does not fit into our bounded update time.

We need to be able to create or destroy a partition without touching more than $O(S)$ entries. We thus introduce a deviation factor, d, which defines how much the number of partitions, p, and the number of elements in each partition, e_i, may deviate from the optimum, S. The smallest value for d which allows to split a maximum-sized partition (size Sd) into two partitions not below the minimum size S/d and vice versa is $d = \sqrt{2}$. This value will also satisfy all other conditions, as we will see.

Until now, we have only tried to keep the elements e_i in each partition within the bounds set by S and d. As it turns out, this is satisfactory to also force the number of partitions p within these bounds, since $N/\min e_i > S/d$ and $N/\max e_i < Sd$.

Whenever a partition grows too big, it is *split* into two or *distributes* some of its contents across one or both of its neighbors, as illustrated in Figure 5.8. Conversely, if an entry is getting too small, it either *borrows* from one or both of its neighbors, or *merges* with a suitably small neighbor. Clearly, all these operations can be done with touching at most Sd entries and at most 3 partitions.

The *split* operation is sufficient to keep the partitions from exceeding their maximum size, since it can be done at any time. Keeping partitions from shrinking beyond the lower limit requires both *borrow* (as long as at least one

of the neighbors is still above the minimum) and *merge* (as soon as one of them has reached the minimum).

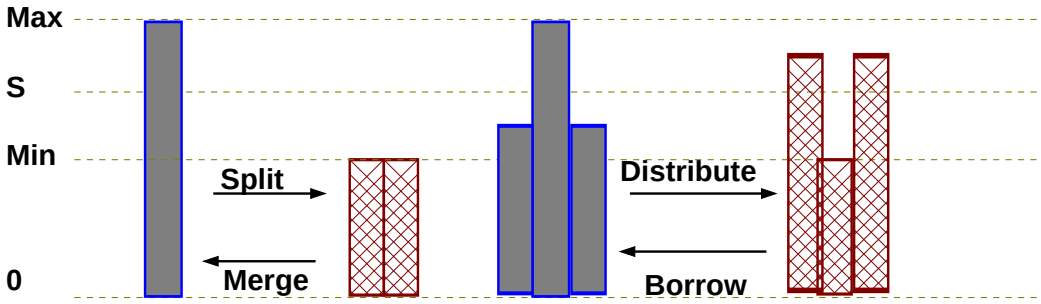


Figure 5.8: *Dynamic Operations*

S crossing an integer boundary may result in all partitions to become either too big or too small in one instant. Obviously, not all of them can be *split* or *merged* at the same time without violating the $O(S)$ bound. Observe that there will be at least $2S + 1$ further insertions or $2S - 1$ deletions until S crosses the next boundary. Also observe that there will be at most S/d maximum-sized entries and Sd minimum-sized entries reaching the boundaries.[1] If we extend the boundaries by one on each side, there is plenty of time to perform the necessary splits or merges one by one before the boundaries change again.

Instead of being 'retro-active' with splitting and joining, it can also be imagined to be pro-active. Then, always the partition furthest away from the optimal value would try to get closer to the optimum. This would make the updates even more predictable, but at the expense of always performing splits or joins.

To summarize, with the new bounds of $S/d-1$ to $Sd+1$, each insertion or deletion of a node requires at most $2(Sd+1)$ updates of BMP pointers, moving $Sd/2$ entries to a new partition, and on boundary crossing $Sd + 1$ checks for minimal size partitions. This results in $O(Sd)$ work, or with d chosen a constant $\sqrt{2}$, $O(S) = O(\sqrt{N})$. All further explanations will consider $d = \sqrt{2}$. Also, since we have $O(s)$ partitions, each with $O(s)$ pointers, the total amount of memory needed for the partitions is $O(N)$.

[1] If there are more than $Sd/2$ minimum-sized entries, than some of them have to be right beside each other. Then a single *merge* will eliminate two of them. Therefore, there will be at most $Sd/2$ operations necessary to eliminate all minimum-sized entries.

5.4.3 Multiple Layers of Partitioning

We have shown that with a single layer of partitions, update complexity can be limited to $O(\sqrt{N})$ with at most a single additional memory access during search.

It seems natural to extend this to more than one layer of grouping and to split the partitions into sub-partitions and sub-sub-partitions, similar to a tree. Assume we defined a tree of α layers (including the leaves). Each of the layers would then contain $s = \sqrt[\alpha]{N}$ entries or sub-partitions of the enclosed layer. As will be shown below, the update time is then reduced to $O(\alpha \sqrt[\alpha]{N})$ at the expense of up to $\alpha - 1$ memory accesses to find the Best Match associated with the innermost container level who has it set.

Prefix updates At the outermost layer, at most sd containers will be covered, with at most two of them partially. These two in turn will contain at most sd entries each, of which at the most $sd/2$ need to be updated, and at most one further split partition. We continue this until the innermost level is found, resulting in at most $sd + (\alpha - 1)2sd/2$ changes, or $O(s)$.

Splitting and Joining At any one level, the effort is s. In the worst case, α levels are affected, giving $O(s\alpha)$.

Boundary Crossing of s The number of insertions or deletions between boundary crossings is $(s+1)^\alpha - s^\alpha$, while the number of minimal-sized partitions is $\sum_{i=1}^{a-1} s^i = (s^\alpha - s)/(s-1)$. So there is enough time to amortize the necessary changes over time one by one during operations that do not themselves cause a split or join.

An application of this technique to multi-dimensional packet classification is documented in [BSW99].

5.4.4 Further Improvements

For many filter databases it would make sense to choose α dynamically, based on the real number of entries. The total number of markers for most databases will be much less than the worst case. If optimal search time should be achieved with bounded worst-case insertion, it seems reasonable to reduce

the partition nesting depth to match the worst-case update. Often, this will reduce the nesting to a single level or even eliminate it.

5.5 Fast Hashing

There are many ways to implement hashing. Since we have mentioned a single memory access per lookup, the number of collisions needs to be tightly bounded. One well-known solution is *perfect hashing* [FKS84]. Unfortunately, true perfect hashing requires enormous amounts of time to build the hash tables and also requires complex functions to locate the entries. While perfect hashing is one of the solutions fulfilling the $O(1)$ access requirement, it is often impractical. An improvement, dynamic perfect hashing [DMR$^+$94], also achieves $O(1)$ lookup time at amortized cost of $O(1)$ per insertion, by having a two-level hierarchy of randomly chosen hashing functions. Thus, it requires two memory accesses per hash lookup, making it an attractive option.

With increasing memory densities (both for static and dynamic RAM) at stagnant and falling prices, memory cost is no longer one of the main limiting factor in router design. Therefore, it is possible to relax the hashing requirements. First, we no longer enforce optimal compaction, but allow for sparse hash tables. This already greatly reduces the chances for collisions.

Second, we increase the hash bucket size. With current DRAM technologies, the cost of a random access to a single bit is almost indistinguishable from accessing many bytes sequentially. Modern CPUs take advantage of this and always read multiple consecutive words, even if only a single byte is requested. The amount of memory fetched per access, called a *cache line*, ranges from 128 to 256 bits in modern CPUs. This cache line fetching us to store a (small) number of entries in the same hash bucket, with no additional memory access penalty (recall that for most current processors, access to main memory is much slower than access to on-chip memory and caches or instruction execution.)

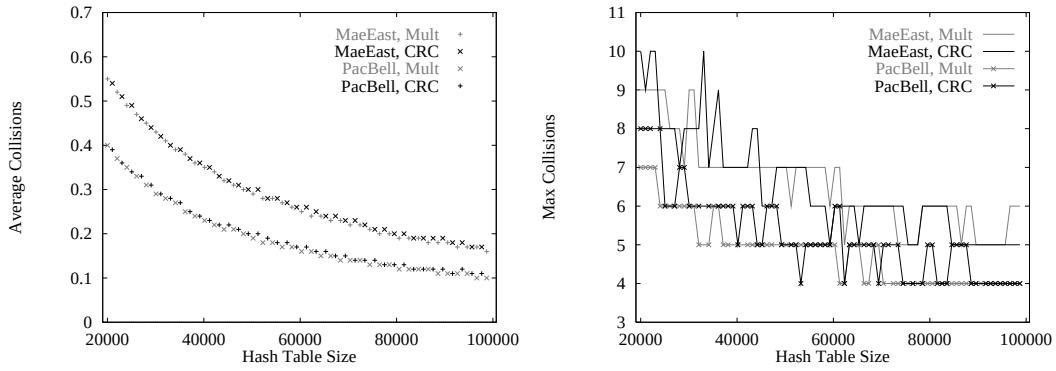
We have seen several key ingredients: Randomized hash functions (usually only a single parameter is variable), over-provisioning memory, and allowing a limited number of collisions, as bounded by the bucket size. By combining these ingredients into a hash function, we were able to achieve single memory access lookup with almost $O(1)$ amortized insertion time.

In our implementations, we have been using several hash functions. One group of functions consists of non-parametric functions, each one utilizing several cheap processor instructions to achieve data scrambling. Switching between these functions is achieved by changing to a completely new search function, either by changing a function pointer or by overwriting the existing function with the new one.

The other group consists of a single function which can be configured by a single parameter, using $\mathrm{frac}(\mathit{Key} * \mathit{Scramble}) * \mathit{BucketCount}$, where frac is a function returning the fractional part, *Key* is the key to be hashed, $\mathit{Scramble} \in (0 \ldots 1]$ is a configurable scrambling parameter, and *BucketCount* is the number of available hash buckets. This function does not require floating point and can be implemented as fixed-point arithmetic using integer operations. Since multiplication is generally fast on modern processors, calculation of the hash function can be hidden behind other operations. Knuth [Knu98, Somewhere] recommends the scrambling factor to be close to the conjugated golden ratio ($(\sqrt{5}-1)/2$). This function itself gives a good tradeoff between the collision rate and the additional allocation space needed.

It is possible to put all the hash entries of all prefix lengths into one big hash table, by using just one more bit for the address and setting the first bit below the prefix length to 1. This reduces the collision rate even further with the same total memory consumption. Since multiplication is considered costly in hardware, we also provide a comparison with a 32-bit Cyclic Redundancy Check code (CRC-32), as used in the ISO 3309 standard, in ITU recommendation V.42, and the GZIP compression program [Deu96]. In Figure 5.9(b), a soft low-pass filter has been applied to increase readability of the graph, eliminating single peaks of $+1$. Since only primes in steps of about 1000 apart are used for the table sizes, there is always a prime hash table size available nearby which fulfills the limit.

Depending on the width of the available data path, it might thus be more efficient to allow for more collisions, thus saving memory. Memory requirements are still modest. A single hash table entry for 32 bit lookups (IPv4) can be stored in as little as 6 or 8 bytes, for the basic schemes or rope search, respectively. Allowing for five entries per hash bucket, the largest database (Mae East) will fit into 1.8 to 2.4 megabytes. Allowing for six collisions, it will fit into 0.9 to 1.2 MB.



(a) Average Collisions (b) Maximum Collisions

Figure 5.9: *Collisions versus Hash Table Size*

5.5.1 Causal Collision Resolution

As can be seen from Figure 5.9, only very few entries create collisions. If we could reduce collisions further, especially at these few "hot spots", we could optimize memory usage or reduce the number of operations or the data path width. In this section, we present a technique called "Causal Collision Resolution" (CCR), which allows us to reduce collisions by adapting the marker placement and by relocating hash table entries into different buckets. We have seen that there are several degrees of freedom available when defining the binary search (sub-)trees for Asymmetric and Rope Search (Section 5.2.1), which help to move markers.

Moving prefixes is also possible by turning one prefix colliding with other hash table entries into two. Figure 5.10(a) illustrates the expansion of a prefix from length l to two prefixes at $l + 1$, covering the same set of addresses. This well-known operation is possible whenever the l is not a marker level for $l + 1$ (otherwise, a marker with the same hash key as the original prefix would be inserted at l, nullifying our efforts). When expansion doesn't work, it is possible to "contract" the prefix (Figure 5.10(b)). It is then moved to length $l - 1$, thus covering too large a range. By adding a prefix C at l, complementing the original prefix within the excessive range at $l - 1$, the range can be corrected. C stores the original BMP associated with that range.

The two binary search trees shown in Figure 5.10 are only for illustra-

tive purposes. Expansion and contraction also work with other tree structures. When other prefixes already exist at the newly created entries, precedence is naturally given to the entries originating from longer prefix lengths. Expansion and contraction can also be generalized in a straightforward way to work on more than ± 1 prefix lengths.

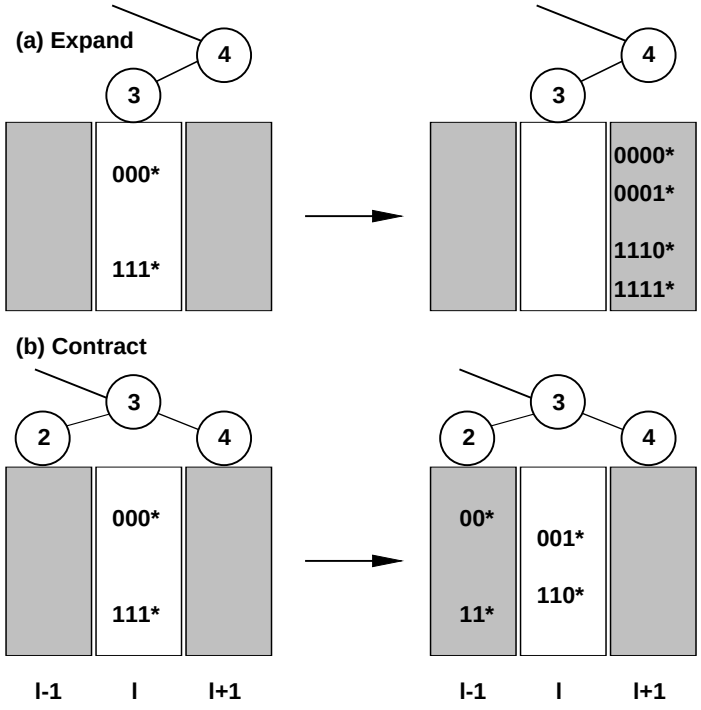
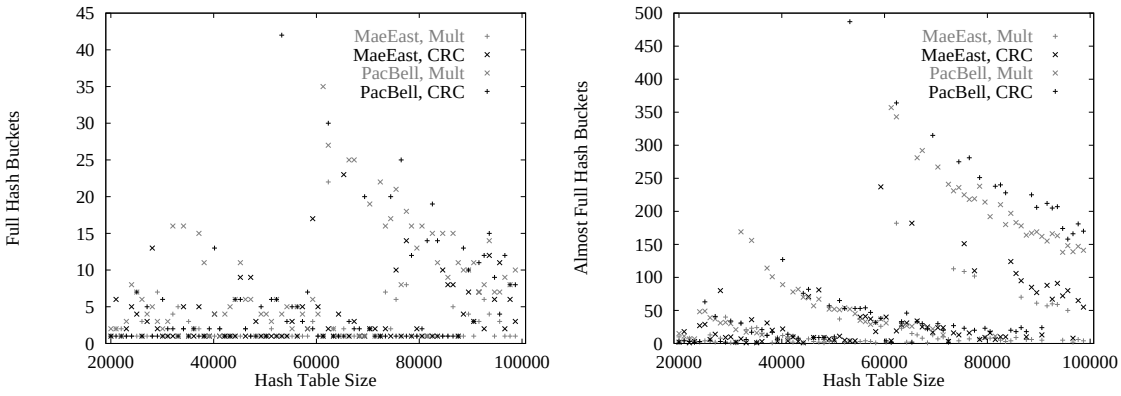


Figure 5.10: *Causal Collision Resolution*

In Figure 5.11 the number of buckets containing the most collisions and those containing just one entry less are shown. As can be seen, for the vast majority of hash table configurations, only less than a handful of entries define the maximum bucket size. In almost half of the cases, it is a single entry. Even for the buckets with one entry less than the maximum size (Figure 5.11(b)), a negligible amount of buckets (less than 1 per thousand for most configurations) require that capacity.

Using causal collision resolution, it is possible to move one of the "surplus" entries in the biggest buckets to other buckets. This makes it possible to shrink the bucket sizes by one or two, reducing the existing modest memory requirements by up to a factor of two.



(a) Number of Hash Buckets with Maximum Collisions

(b) Number of Hash Buckets with One Entry Less

Figure 5.11: *Number of (Almost) Full Hash Buckets*

5.6 Summary

This chapter starts by presenting routines to build the search structures described in Chapter 4 from scratch. It also shows how to optimally calculate the Ropes in order to reduce the worst-case search time. It then evolves to schemes allowing the real-time update of these structures. For the non-Rope structures, it shows that these changes can be done cheaply and do not require and reshaping. For Rope search, the performance may slowly degrade over time. But it is shown that the most expensive part of that operation, recalculating the optimal Ropes, can be done incrementally. This considerably reduces the process of rebuilding the structure from scratch.

The chapter further introduces the concept of marker partitioning to considerably reduce update time at only minimal search-time expenses. It also describes how we can do practical fast hashing, requiring only a single memory access during search. We also show that this can be done in a small memory footprint. It also shows that this footprint can be reduced even further by applying a technique we call *Causal Collision Resolution*, which can be used to reduce the size of a maximum-size hash bucket or reduce the overall number of hash buckets.

Chapter 6

Implementation

In Chapter 4, we have seen that the search code is extremely simple per iteration and only very few iterations are needed. Yet it turns out that in general, compiled code is not executed at the speeds that can be expected from the code's simplicity. After explaining our principles for measuring speed, the individual optimization steps and their effect on the speed will be discussed. Optimizations were performed for both the Sparc v9 (UltraSparc [WG94]) and Pentium II [Int97c, Int97a] architectures. More background on the analysis results can be found in [Mey99].

6.1 Measurement Principles

The main metric for speed measurements is *lookups per second* (lookups/s). The accuracy and meaningfulness of the measurement results depends on two key criteria:

Test Data How is the test sequence chosen?

System Interferences How does the operating system and other processes interfere with our measurements?

6.1.1 Test Data Generation

For the creation of test data, several simple generation mechanisms come to mind:

Linear Walking linearly through the search (=network address) space, either in single increments or in larger steps. This has the advantages of simplicity and perfect reproducibility, but may show artifacts from unrealistic cache locality and—if run only over a partial region—may only be representative for the address range covered, bearing no relevance for the rest of the database.

Random Walking pseudo-randomly keeps the simplicity and the reproducibility, while reducing cache locality and taking samples from all regions of the database. Because it probably has least locality, it can serve well as a worst case.

Trace Driven Simulating using a trace from a router whose routing table is also available would give an excellent real-world result. Unfortunately, for none of the backbone routers both routing information and packet traces are currently available.

Entry Driven Searching for each entry in the database would again be heavily biased. The prefixes each cover ranges of vastly different size, the largest covering more than 16 Million times more addresses than the smallest. It is unlikely that both will have even remotely similar traffic densities. An inspection of the available databases further reveals that roughly 95% of the prefixes lie in the range $192.0.0.0 \ldots 210.255.255.255$.

"Random" seems to be the fairest strategy from those available. Yet a closer look at the database shows that only roughly 55% of the entire address space is covered by prefixes. This means that in 45% of the queries, the default entry will be returned. These routing tables were retrieved from the so-called "default-free" Internet. This means that these routers in principle should never see packets addressed to destinations they cannot find in their routing tables. If such a packet should ever reach them, they will not know where to send it further and will bounce an appropriate ICMP error packet back to the sender.

Therefore, only random addresses which can be found in the database should be generated. Our approach is to pick random entries one by one and

then search for them in the database to see whether it is a valid entry. Then they are stored in a table that is later being consulted to perform timed measurements. This random search also gives a minimal chance for the caches to "warm up" [PH96], i.e. contain some of the data they would if the router had been doing lookups for some time.

Nevertheless, a fully random search also has its uses. If the router utilizing our algorithm would not be in the default-free Internet, but just outside, e.g., within a large ISP, it would contain a large number of routes *and* a default route towards the backbone. Thus, we will perform our measurements and optimizations with both scenarios:

Backbone Router Only addresses contained in the database are to be looked up (default-free).

Border Router addresses matching the (implicit) default entry, resulting in packets being forwarded to a more knowledgeable router. Using pure random data, this results in about three quarters of the lookups returning the default route (the prefixes have about 25% coverage).

6.1.2 Avoiding System Interferences

Since the measurements were performed on networked multitasking machines, there would be influences from the multitasking system, which might activate some background process or system cleanup function on a regular basis. And there might also be other users working on the machine. The latter can be easily checked for and avoided, but the former is hard to avoid, even if the system seems unloaded.

To minimize system interferences, we ran the simulation multiple times. Since we knew that the program flow was deterministic (each run would use the same database and test addresses), we knew that any variations would come from background load and system maintenance. Therefore, instead of averaging the run-times, we took the minimum time. Firstly, prolongations of the measured time can easily be explained by system activities, whereas faster run-time—besides explanations involving system failures—can only be due to "too good" caching. "Too good" caching cannot happen out of the blue, so this also indicates the absence of system interferences polluting the caches. Secondly, the minimum execution time was very close to the average (only a few percent away), and closer even to the median.

Using performance measurement tools such as Quantify [Ratb] does not help. Since the commonly available tools are software-based, they have to "instrument" the code (insert additional measurement instructions), which require additional computing cycles and memory accesses. Though the tools try to eliminate this time from their measurements, the effects on pipelining and memory stalls become noticeable when profiling compact pieces of code.

Thus we do not doubt the validity of our results under these circumstances.

6.2 Implementation Overview

The code piece we will analyze and show how to improve does asymmetric binary search for 32 bit IPv4 addresses on hash tables that use a simple multiplicative hash. The database used contains the 1996 prefixes from Mae East.

The hash function gets the search key as a right-aligned prefix, i.e., occupying the least significant bits, multiplies it by a constant and masks the result by the length of the hash table. Each prefix length uses its own hash table, with the number of elements in the hash table being a power of two. Each hash table entry consists of 8 bytes (see Table 6.2). Its first four bytes contain the prefix, so the hash lookup code can determine whether the right entry was found. The next byte contains flag bits, such as whether this entry is a prefix, only a marker, there are collisions, and the entry is valid at all. The latter is necessary to distinguish an all-zeroes (or all-ones) entry from an empty slot.

0	1	2	3
Prefix			
Flag Bits	Collision Count	Next Hop Index	

Table 6.1: *Format of a Hash Table Entry*

If there are collisions, the collision bit is set in the flags and the *collision count* byte contains a counter, the number of entries to search in linear order until a match is found or the end is reached. The last two bytes are an index into the next hop table, containing information such as outgoing link, IP, and—if necessary—MAC layer address to use.

Each hash table is described by a structure as shown in Table 6.2. It contains a pointer to the hash table entries and a mask to apply to the output of the hash function, to trim it down to a valid index into the hash table. It also contains the minimal number of significant bits the hash function needs to deliver. In this case it is used as padding, to make the descriptor a power of two. Additionally, it contains the current prefix length, and which prefix lengths to branch to when branching "left" (shorter prefix lengths) or "right" (longer), in relative and absolute measures.

0	1	2	3
Pointer to the Hash Table			
Mask for Index ($2^b - 1$)			
Bits Needed ($\geq b$)			
Prefix Length	Right Absolute	Right Relative	Left Relative

Table 6.2: *Hash Table Descriptor*

In this scenario, the memory required by the hash tables amounts to just below 1.2 MB, the descriptors occupy just above 0.5 KB.

The code we started with was organized in a very modular fashion, not only to provide for a solid basis, but also to simplify performing a wide range of functional experiments. The first obvious optimization was to inline the code, which made the following logical flow also the physical flow of events: No additional parameter passing and subroutine calls. This basic structure can be found in Figure 6.1 and closely corresponds to the pseudo-code in Figure 4.5 on page 40. The diagram also contains the relative frequency of taking both paths of a branching decision in the border router scenario.

The algorithm in Figure 6.1 after inlining runs as follows: First, there is a check that there are still hash tables to be searched (box "prefixLen>0?"). Then, after devising which entry in which hash table should be consulted, the entry is looked at. If it matches our query data (first box "entry found?"), it is analyzed further below. If not, the algorithm checks whether there are any collisions ("collision?"). If so, the entries in the chain are checked for a match.

Then, depending on whether the above strategy has found an entry (second box "entry found?"), everything is prepared for a left (shorter prefixes) or right (longer prefixes) branch. But if the entry found is not also a marker, we can short-cut the search and terminate here.

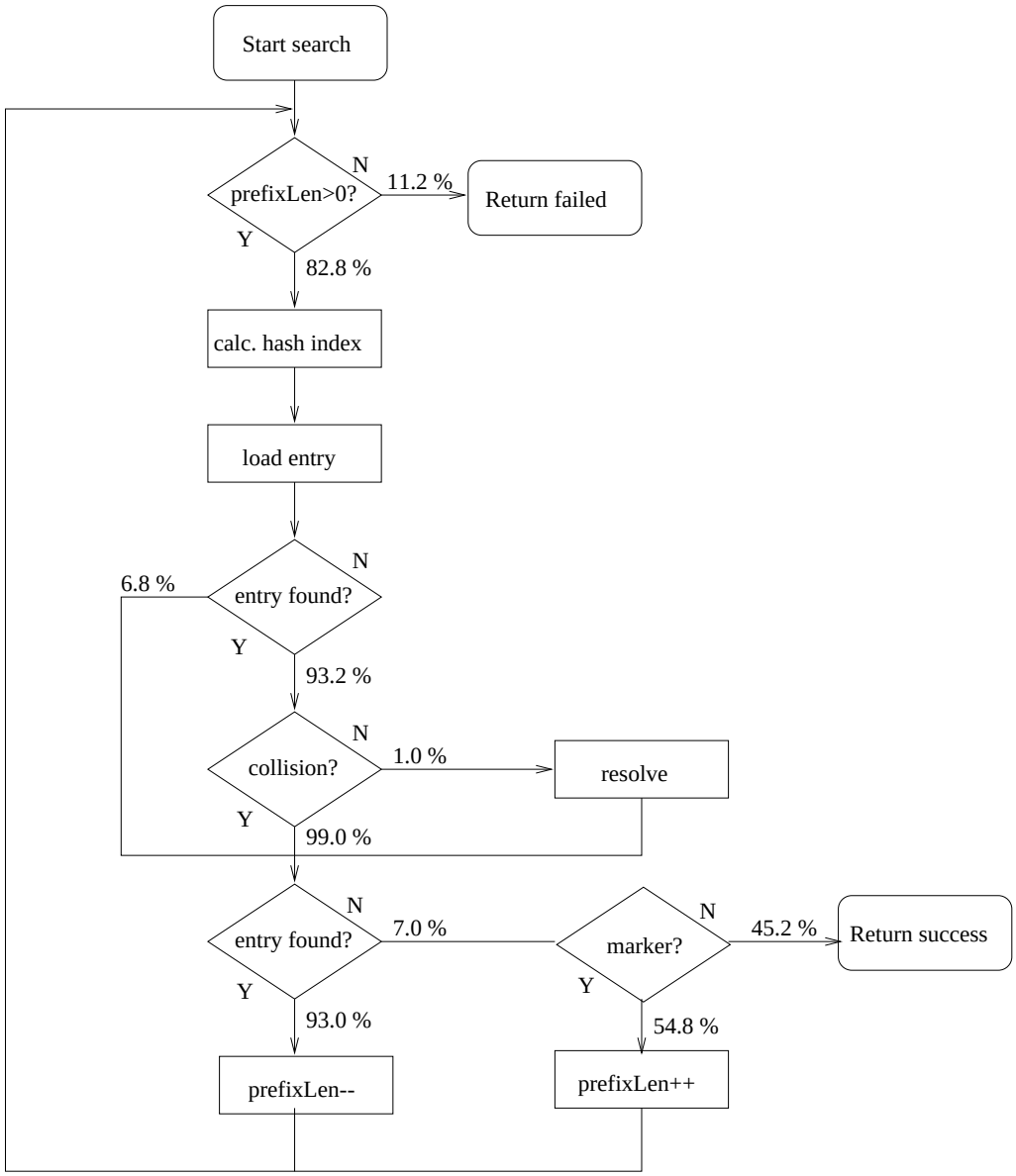


Figure 6.1: *Flow Chart for Basic Binary Search [Mey99]*

6.3 Sparc Version 9 Architecture Optimizations

In view of this knowledge of the algorithm, let us review the relevant hardware characterstics of the machine used and then proceed to perform appropriate optimizations.

6.3.1 The Hardware and Its Restrictions

Our experiments were run on a Sun Ultra 60, where a UltraSPARC-II processor runs at 370 MHz. This processor implements the Sparc V9 architecture [WG94] and was equipped with dual instruction execution units, a first-level cache (16 KB data and instruction each) which can be accessed with a two cycle latency, and an external 4 MB second-level cache with an access time of 30 ns, requiring around 11 clock cycles per access, during which 22 instructions could be executed. Memory has a latency of 150 ns, leading to a response time of seemingly endless 55 clock cycles. In this time more than 110 instructions could have been executed, enough for two complete lookups, assuming no memory latency [Rij00].

With so few instructions per loop cycle, it is therefore imperative to avoid memory accesses and drastically cut down on second-level cache accesses. Unfortunately, fine-grained control on cache allocation and replacement policies is not available to the user. Since the hash tables scarcely fit into second-level cache, it is to be expected that external memory accesses will occur only in extremely seldom cases. Also, with the compact size of the hash table descriptors and their frequent access, they will probably remain in first-level cache for most of the time. Thus, memory access latency can be expected to be minimal.

Besides memory access timing, modern processors have another bottleneck: Pipelines. To achieve today's clock frequencies, deep pipelines are needed. Unfortunately, they may require flushing and refilling at (mispredicted) branches. Looking at Figure 6.1, it is obvious that almost the entire algorithm consists of decisions and branches, and there is almost no computation to be done in-between them.

Another source for pipeline stalls are data dependencies, when an instruction needs to work on data that a previous instruction has not yet delivered. This may be due to a required memory access or a long execution time of the previous instruction. Besides that, an instruction that is just before the current instruction may be executed in parallel to it, requiring a stall for the latter even if the former does not induce any latency.

These two factors will be the main issues to be addressed in the upcoming optimizations. Each of the ideas for optimizations will be discussed separately below. Unless specified otherwise, all the optimizations were done entirely in

the C programming language, although the assembly source output from the compiler was used to analyze some of the impacts.

6.3.2 Tighter Coupling

Instead of just performing inlining, it is also possible to couple the modules tighter. As an example, the generic function to search an entry in the hash table "returns" a pointer to the entry found or NULL, if nothing was found. This requires an additional test for NULL in the "calling" code. Instead, the hash table test for a match could be combined with the left-right-branching test for a match, reducing comparisons and, generally, overhead. This simple trick yields a speedup of $15 \ldots 20\%$.

6.3.3 Expanding Bitfields

Another possibility is to expand the bit-fields into their own integers, to simplify loading and masking. Unfortunately, this increases the memory footprint of the data structures, thus reducing the cache hit rate. Therefore, this "optimization" would slow down the program by $50 \ldots 100\%$.

6.3.4 Changing Data Layout

Trying to do some fine-tuning with the data structure layout turned out to yield $0.1 \ldots 3\%$ improvement, at the limit of our measurements. Although this could be considered a minor success, together with the preceding result, it became clear that memory layout was not the problem.

6.3.5 Overlaps Between Loop Iterations

The next optimization step is to move instructions from the beginning of the current iteration to the end of the previous iteration. So, we can reduce the data dependencies at the beginning of the loop and thus reduce pipeline stalls. Figure 6.2 outlines this for fetching the address of the hash table for the next round. For clarity, this figure displays a restricted view onto Figure 6.1.

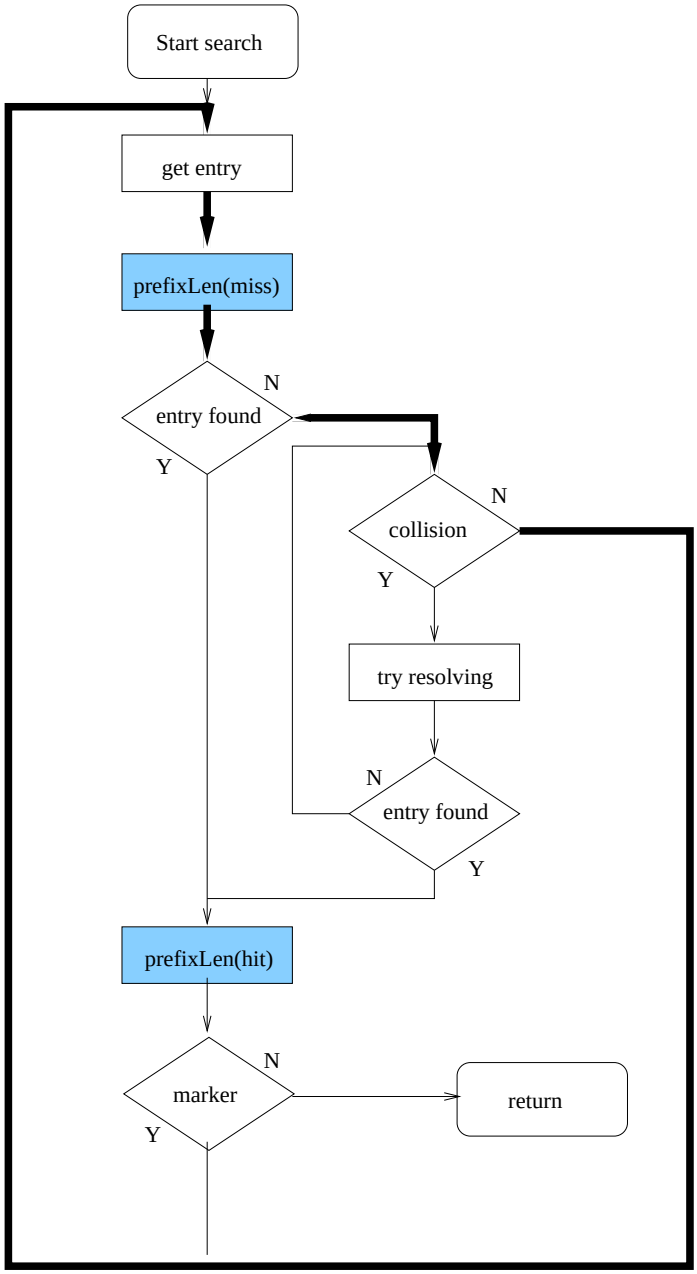


Figure 6.2: *Loop Overlapping (main data path in bold) [Mey99]*

As can be seen from Figure 6.1, hash table misses are much more common than hits (93% vs. 7% in the border router scenario, 69% vs. 31% for the backbone router). The instruction flow for this case is emphasized by the thick line in Figure 6.2. So it is a "safe" guess to assume that there will be a miss this round and optimize for this case by determining the hash table address for the next iteration early (box "prefixlen(miss)"). As can be seen, the calculation is done an entire iteration earlier than was done originally.

If there should be a hash table hit, then there is still time before the test for a marker to start calculating the next hash table address (box "prefixlen(hit)"). This optimization leads to an additional $15 \ldots 25\%$ speedup. Other re-arrangements, such as interleaving multiple parallel statements to reduce pipeline stalls, improved this by further 2%.

The result of all optimizations was that the average instruction now took two clock cycles (CPI=2). The theoretical maximum lies at CPI=0.5, when always two instructions are executed in parallel.

6.3.6 Dual Address Search

We have seen that interleaving statements gives an improvement. The ultimate in interleaving is performing two independent address lookups in parallel. Unfortunately, searches do not always require the same amount of loops. Measurements showed that the average number of iterations was just below the maximum of five. Therefore, the terminating condition was changed to always make five passes.

Additionally, each of the individual searches could have different results and therefore branch differently. The combined algorithm had to take this into account and handle all possible combinations. Figure 6.3 shows the flow chart of such an attempt.

This resulted in a speedup of $18 \ldots 25\%$. Further speedups might be achieved by looking for a third or even fourth address in parallel. It has to be kept in mind that each of these parallel paths requires its own processor registers, imposing a hardware limit on growth. Besides, each additional path doubles the number of possible combinations, quickly leading to a state explosion.

6.3.7 Loop Unrolling

Now that the number of iterations is known in advance, loop unrolling can entirely avoid counting the iterations and free a processor register for other chores. This resulted in accelerating lookups by another $17 \ldots 33\%$.

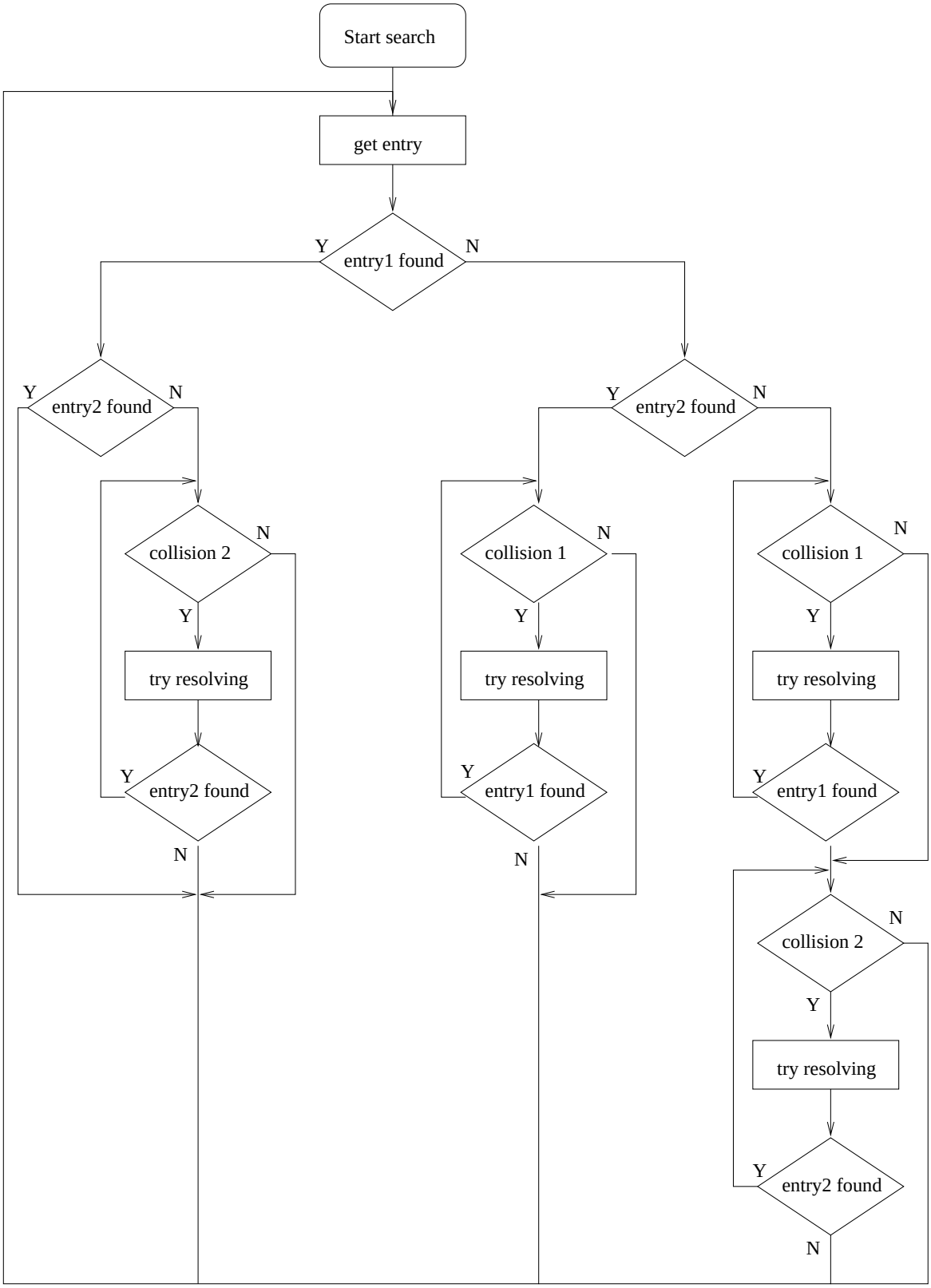


Figure 6.3: *Dual Address Search [Mey99]*

6.3.8 Assemly Optimizations

After improving the C source, re-arranging statements at assembly language level did not yield any further improvements. Neither did adding explicit predictions to the conditional branches, or use of branches anulling the instruction in the branch delay slot [WG94].

Overall, the optimizations to the existing C code totalled to speeding up the search by a factor of $3.9 \ldots 4.6$, or a factor of 2.2 compared to the inlined version.

6.4 Intel Pentium Architecture Optimizations

The same improvements were tried on an AMD K6-II processor clocked at 300 MHz. This processor is instruction-set compatible to the Intel Pentium processor series, but has a different internal architecture. Therefore, not all results may be directly transferred. A comparison between the two results are given in Table 6.3. Each step is relative to the most recent optimization, that achieved a speedup for this processor.

Optimization step	Speedup on			
	UltraSPARC-II		AMD K6-II	
	Border	Backbone	Border	Backbone
Plain Code	1.000	1.000	1.000	1.000
Inlining	2.142	1.780	2.091	1.867
Tighter Coupling	1.155	1.186	(included)	
Expanding Bitfields	0.795	0.497	0.854	0.584
Data Layout	1.001	1.034	1.243	1.162
Loop Overlaps 1	1.246	1.149	1.047	0.992
Loop Overlaps 2	1.019	1.021	(included)	
Dual Search	1.245	1.182	0.796	0.739
Loop Unrolling	1.171	1.333	1.214	1.222
Total	4.620	3.906	3.302	2.269

Table 6.3: *Speedup Comparison (based on [Mey99]). Speedup is relative to the previous successful optimization step (> 1), total counts successful steps only.*

As can be seen, changing the data layout gives an improvement on the AMD K6-II, which probably results from a different prefetching logic. Loop overlaps do not give as much improvement on the K6-II, probably due to a better instruction scheduling inside the processor score. Its major drawback is the slowdown for dual searches, no wonder in view of the extremely limited number of available registers in the Intel x86 architecture [Int97b].

As mentioned at the beginning of this chapter, *lookups/s* is the most important measure. For each of the four scenarios, we therefore give the number of lookups, in Mega-lookups/s in Table 6.4. Using real-world packet distribution from June 1997 [Nat97], the equivalent line speed has also been calculated. This table also introduces the results from the BBN GigaRouter project [PC$^+$98] on DEC Alphas at 500 MHz, using the basic binary search algorithm. These numbers are complete packet forwarding decisions per second (decreasing the number of lookups), but also apply a limited amount of caching (resulting in a speed improvement).

Processor	Scenario	Mlookups/s	Gbit/s
UltraSPARC	Border	5.8	23.2
	Backbone	4.6	18.4
AMD K6-II	Border	2.7	10.8
	Backbone	2.0	8.0
DEC Alpha	Backbone (BBN results)	10 . . . 13	40 . . . 52

Table 6.4: *Optimized Lookup Speed*

We believe that the limited number of translation look-aside buffers (TLB) available in the processor results in a major performance decrease due to the resulting TLB traps and lookups. We expect that direct access to physical memory would result in a significant speedup. Please keep in mind that all the simulations were done under worst-case conditions, i.e., there is no traffic locality, as there would be under realistic traffic conditions.

6.5 Implementation in General-Purpose Hardware

As we have seen in both Figure 4.5 and Figure 4.14, the search functions are very simple, so ideally suited for implementation in hardware. This chapter has shown that in spite of its simplicity, current processors do not perform well, since out of little data a lot of information can be gathered, requiring a number of branches, each based on a very simple condition. It can be concluded that the decisions are too simple, but too many for modern processors. Hardware lends itself well to performing these checks in parallel.

The inner component, most likely done as a hash table in software implementations, can be implemented using (perfect) hashing hardware such as described in [Spi95], which stores all collisions from the hash table in a CAM. Instead of the hashing/CAM combinations, a large binary CAM could be used. Besides the hashing function described in [Spi95], Cyclic Redundancy Check (CRC) generator polynomials are known to result in good hashing behavior (see also the comparison to other hashing functions in Section 5.5).

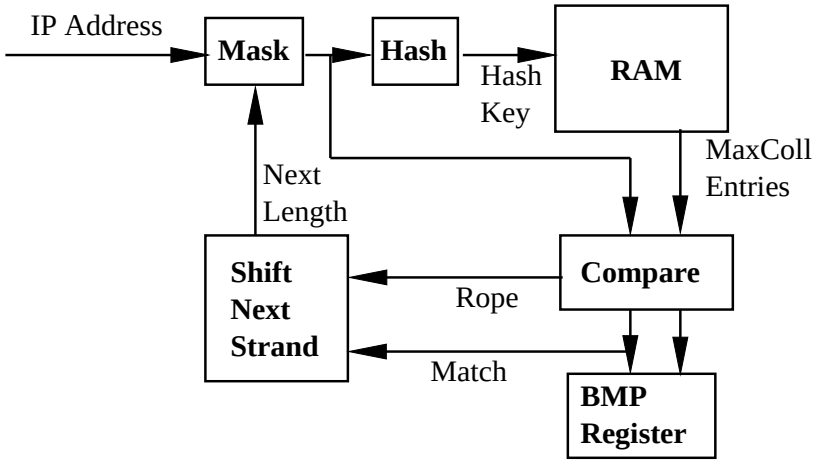


Figure 6.4: *Hardware Block Schematic*

The outer loop in the Rope scheme can be implemented as a shift register, which is reloaded on every match found, as shown in Figure 6.4. This makes for a very simple hardware unit. For higher performance, the loop can be unrolled into a pipelined architecture. Pipelining is cheaper than replicating the entire lookup mechanism: in a pipelined implementation, each of the RAMs can be smaller, since it only needs to contain the entries that can be retrieved in its pipeline stage (recall that the step during which an entry is found depends

only on the structure of the database, and not on the search key). Consult Figure 4.10 for a distribution of the entries among the different search steps. As is true for software search, Rope search will reduce the number of steps per lookup to at most 4 for IP version 4 addresses, and hardware may also use an initial array. Pipeline depth would therefore be four (or five, in a conservative design). If a pipelined design is infeasible, but multiple memory banks exist, changing from binary to k-ary branching will lead to a reduction in the number of search steps. The resulting performance improvement heavily depends on how well the access to memory banks can be parallelized in the actual implementation.

Consider a pipelined approach using early 1999 DRAM technology (SDRAMs at 125 MHz [Gri99]) with identical information in both banks of the SDRAM. We expect the hardware cost to be around US$ 100. Using this simple device, it is possible to achieve a throughput of one lookup every two cycles, resulting in 62.5 million packets per second. This speed is equivalent to 20 Gbit/s with minimum size packets or around 250 Gbit/s using measured packet distributions from June 1997 [Nat97]. Using custom hardware and pipelining, we thus expect a speedup of roughly 10 to 20 relative to software performance, allowing for affordable IP forwarding reaching far beyond the single-device transmission speeds reached in high-tech research labs (Table 6.4).

6.6 Summary

In this chapter, we presented software optimization techniques which were used to improve the speed of the search function. Using Sparc and Intel/AMD processors as examples, we compared the effects of the optimizations. We showed which optimizations perform well on both, neither, or just one of the processor families. Of course, the measurement methodology is discussed in detail. This chapter also outlined the more relevant details of our implementation and concluded with a presentation of fast search hardware.

Chapter 7

Performance Summary

Recollecting some of the data mentioned earlier, we show measured and expected performance for our scheme. We will first discuss the memory requirements as due to the expansion resulting from markers, compare complexities between different lookup techniques, recapitulate the speed under realistic assumptions, and mention expected behavior under IPv6.

7.1 Memory Requirements

One open question is the number of markers needed in realistic workloads. Theoretically, they could multiply to number of total entries by up to $\log_2 W$. Yet in the typical case, many prefixes will share markers (Table 7.1), reducing the marker storage further. Notice the difference between "Max Markers", the number of markers requested by the entries, and "Effective Markers", how many markers really needed to be inserted, thanks to marker sharing. In our sample routing databases, the additional storage required due to markers was only a fraction of the database size. However, it is easy to give a worst case example where the storage needs require $O(\log_2 W)$ markers per prefix. (Consider N prefixes whose first $\log_2 N$ bits are all distinct and whose remaining bits are all 1's). The numbers listed below are taking from "Plain Basic" scheme, but the amount of sharing is comparable with other schemes.

	Total	Basic: Request for					Max	Effective
	Entries	0	1	2	3	4	Markers	Markers
AADS	24218	2787	14767	4628	2036	0	30131	9392
Mae-East	38031	1728	25363	7312	3622	6	50877	13584
Mae-West	23898	3205	14303	4366	2024	0	29107	9151
PAIX	5924	823	3294	1266	541	0	7449	3225
PacBell	22850	2664	14154	4143	1889	0	28107	8806
Mae-East 1996	33199	4742	22505	3562	2389	1	36800	8342

Table 7.1: *Marker Overhead for Backbone Forwarding Tables*

7.2 Complexity Comparison

Table 7.2 collects the (worst case) complexity necessary for the different schemes mentioned here. Be aware that these complexity numbers do not say anything about the absolute speed or memory usage. See Chapter 3 for a textual comparison between related work and more background information. For Radix Tries, Basic Scheme, Asymmetric Binary Search, and Rope Search, W is the number of distinct lengths. Memory complexity is given in W bit words. The three closely related algorithms presented in this thesis (Basic Scheme, Asymmetric Binary Search, and Rope Search) allow for a flexible and tunable tradeoff between build and search times. It can be seen that the schemes performing binary search on prefix lengths outperform the other schemes in most categories. In the other categories, they are only marginally worse.

7.3 Measurements for IPv4

Many measurements on the number of memory accesses for real-world data have been included earlier in this paper. The number of memory accesses have been analyzed in Chapter 4, software and hardware speeds in Chapter 6, respectively. To summarize, we have shown that with modest memory requirements of less than a megabyte and simple hardware or software, it is possible to achieve fast best matching prefix lookups with at most four memory accesses, some of them may even be resolved from cache.

Algorithm	Build	Search	Memory	Update
Binary Search	$O(N \log N)$	$O(\log N)$	$O(N)$	$O(N)$
Trie	$O(NW)$	$O(W)$	$O(NW)$	$O(W)$
Radix Trie[a]	$O(NW)$	$O(W)$	$O(N)$	$O(W)$
Ternary CAMs	$O(N)$	$O(1)^b$	$O(N)$	$O(N)$
Basic Scheme *or*	$O(N \log W)$	$O(\log W)$ $O(\alpha + \log W)$	$O(N \log W)$	$O(N)$ $O(AW \log W)$
Asymmetric BS *or*	$O(N \log W)$	$O(\log W)$ $O(\alpha + \log W)$	$O(N \log W)$	$O(N)$ $O(AW \log W)$
Rope Search *or*	$O(NW^3)$	$O(\log W)$ $O(\alpha + \log W)$	$O(N \log W)^c$	$O(N)$ $O(AW \log W)$

$A = \alpha \sqrt[\alpha]{N}$

[a] As in current NetBSD implementations

[b] Using theoretical very large CAMs (not available today)

[c] For the search structure; for building and fast updates $O(NW^3)$ is needed, possibly outside the router

Table 7.2: *Speed and Memory Usage Complexity*

7.4 Projections for IP Version 6

Although there originally were several proposals for IPv6 address assignment principles, the *aggregatable global unicast address format* [HOD98] is at the verge of being deployed. All these schemes help to reduce routing information. In the optimal case of a strictly hierarchical environment, it can go down to a handful of entries. But with massive growth of the Internet together with the increasing forces for connectivity to multiple ISPs ("multi-homing") and meshing between the ISPs, we expect the routing tables to grow. Another new feature of IPv6, Anycast addresses [HD98, DH98], may (depending on how popular they will become) add a very large number of host routes and other routes with very long prefixes.

So most sites will still have to cope with a large number of routing entries at different prefix lengths. Likely, there will be more distinct prefix lengths, so the improvements achieved by binary search will be similar or better than those achieved on IPv4.

For the array access improvement shown in Section 4.2.3, the improvement may not be as dramatic as for IPv4. Although it will improve performance for IPv6, after length 16 (which happens to be a "magic length" for the

aggregatable global unicast address format), only a smaller percentage of the address space will have been covered. Only time will tell whether this initial step will be of advantage. All other optimizations are expected to yield similar improvements.

7.5 Summary

We have designed a new algorithm for best matching search. The best matching prefix problem has been around for twenty years in theoretical computer science; to the best of our knowledge, the best theoretical algorithms are based on tries. While inefficient algorithms based on hashing [Skl93] were known, we have discovered an extremely efficient algorithm that scales with the logarithm of the address size.

Our algorithm contains both intellectual and practical contributions. On the intellectual side, after the basic notion of binary searching on hash tables, we found that we had to add markers and use pre-computation, to ensure logarithmic time in the worst-case. Algorithms that are trying to use binary search of hash tables without using markers and pre-computation are unlikely to provide logarithmic time bounds. Among our optimizations, we single out mutating binary trees as an esthetically pleasing idea that leverages off the extra structure inherent in our particular form of binary search.

On the practical side, we have a fast, scalable solution for IP lookups that can be implemented in either software or hardware, reducing the number of expensive memory accesses required considerably. We expect most of the current characteristics of this address structure to remain and possibly even becoming stronger in the future, especially with the transition to IPv6. Even if our predictions, based on the little evidence available today, should prove to be wrong, the overall performance can easily be restricted to that of the basic algorithm which already performs remarkably well.

We have also shown that updates to our data structure can be very simple, with a tight bound around the expected update efforts. Furthermore, we have introduced causal collision resolution. Thanks to knowledge outside the hash function itself, it greatly simplifies collision resolution compared to known algorithms working only inside the hash tables.

With algorithms such as ours, we believe that there is no more reason for router throughputs to be limited by the speed of their lookup engine. We also do not believe that hardware lookup engines are required because our algorithm can be implemented in software and still perform well. If processor speeds should not keep up with the expectancies, extremely affordable hardware (around US$ 100) enables forwarding speeds of around 250 Gbit/s, much faster than any single transmitter can currently achieve even in the research laboratories. Therefore, we do not believe that there is a compelling need for protocol changes to avoid lookups as proposed in Tag and IP Switching. Even if these protocol changes were accepted, fast lookup algorithms such as ours are likely to be needed at several places throughout the network.

Anyone capable of holding a soldering iron can thus achieve IP forwarding speeds far exceeding the physical layer speeds currently achieved in high-tech research laboratories.

Our algorithm has already been successfully included into the BBN multi-gigabit per second router [PC$^+$98], which can do the required Internet packet processing and forwarding decisions for $10 \ldots 13$ million packets per second per forwarding engine. Each forwarding engine is based on a single off-the-shelf microprocessor, a member of the DEC Alpha family [Sit92], clocked at 500 MHz.

Chapter 8

Advanced Matching Techniques

We have seen that the basic binary search of hash tables (Section 4.1.2) and its various enhancements such as Rope Search (Section 4.2.2) provide dramatic improvements to traditional search schemes. Now the question arises whether these advantages seen in the one-dimensional case can also be transformed and applied to multi-dimensional classification.

First, we formalize the problem of multi-dimensional prefix matching. Then, we have a look at some existing classifiers, which might take advantage of our scheme. Later, we evolve the one-dimensional algorithm to two dimensions. Later, we extend it to more than two dimensions.

We will conclude this chapter with a closer look at how longest prefix matching and range matching relate to each other and can take advantage of each other, both in single and multiple dimensions.

8.1 Properties of Multi-Dimensional Matching

Two-dimensional matching is very similar to one-dimensional matching, but instead of having a database of prefixes such as $\{111\_00*, 0110*, \ldots\}$, we

have a database of prefix pairs, e.g., $\{(1*, 0000\_00*), (1111\_11*, 0*), \ldots\}$. This database is consulted for fully-specified tuples, such as—assuming 12 bit address length—$(1100\_0000\_1111, 0000\_1111\_1111)$. These pairs are ordered tuples, with each of the tuple's fields representing a range of coordinates in the corresponding dimension.

Extending it to $d > 2$ dimensions is straightforward, but instead of 2-tuples, d-tuples are being used.

Obviously, each prefix (or prefix tuple) can also be represented by the set of addresses (or address tuples) it matches. In one-dimensional matching, when multiple matching entries exist in the database, the sets representing these entries can always be completely ordered by a subset relation. Otherwise said, from each pair of matching entries, one of the representing sets was a subset of the other. Therefore, the most specific entry could be determined easily and unambiguously.

For d-dimensional matching (with $d \geq 2$), ambiguities may—and in general, will—exist. Assume again our two-dimensional prefix database $\{(1*, 0000\_00*), (1111\_11*, 0*)\}$. If we would search this database for entries matching $(1111\_1111\_1111, 0000\_0000\_00000)$, both entries would match. Neither of them can be considered more specific: The second entry is more specific in the first dimension, but the first entry is more specific in the second dimension. Also, the size of the sets represented by either tuple is the same. Therefore, it is impossible to find a natural ordering between the two; the ambiguity cannot be resolved.

If it is known in advance that only few entries will contain ambiguities, it may be possible to split the entry into several sub-entries to resolve ambiguities, as described in [Har99].

To resolve ambiguity, several solutions have been proposed:

Unspecified There is no simple way to know in advance which of the matching entries will be returned. This is the simplest solution, but seldom satisfactory, unless ambiguities can be prevented to appear in the database in the first place [Har99].

Priorities of Dimensions The dimensions are prioritized against each other. Without loss of generality, it can be assumed that the dimensions are sorted in order of decreasing priority. When resolving ambiguities, the

prefix lengths of the individual numbers are concatenated as digits in a $W + 1$-ary number and the entry with the highest number wins.

It is not obvious how this scheme can be implemented more efficiently than the next proposal. Therefore, applications for this solution are not apparent.

Explicit Priorities Each entry is assigned an explicit priority, which can be considered constant during the presence of that entry in the database. This priority is then used to resolve ambiguities. We assume that entries having a subset relation will have priorities set so they do not conflict with the subset relation. If they ever do, i.e., if a more specific entry should have a lower priority than a less specific entry, the lower priority entry will never be consulted. Instead of solving this problem at search time, it can be avoided at database build time by ignoring the hidden entry.

This scheme is the the most flexible of these three, and includes the others as subset. Providing a solution for this problem therefore implies having a solution for the others.

The following discussions will assume the assignment of explicit priorities and provide solutions for this case.

8.2 Use In Existing Classifiers

In [SVSW98], Srinivasan et al. describe two schemes for packet classification. The first, *Grid of Tries* allows for efficient classification in two dimensions. As the name implies, it consists of two intermingled tries, one for each of the two dimensions. The first trie is searched for a longest matching prefix in traditional fashion. Only the second trie than contains non-standard constructs, which cannot be lead back to a prefix operation. So, any of the algorithms for finding the longest matching prefix discussed in this thesis may be used to speed up the first trie. Although this might lead to a significant performance boost, the second trie search limits still remains at $O(W)$ complexity. Therefore, the performance boost, although real, remains unseen in the complexity analysis.

The second algorithm described in [SVSW98] is called *Cross-producting* and allows for an arbitrary number of dimensions. When it is being used, all

the dimensions are being searched independently for their best match. Later, the results are combined using a huge table. Since this table is of exponential size, it is maintained as a virtual table, i.e., only the entries that are needed are being calculated and are cached. For the back-end procedure, they give way to linear search through the classifiers. For sensible packet classification databases, the improvement over naïve caching of fully-specified entries is significant, since a typical cache entry can be used for many more flows than caching fully-specified entries.

For this scheme, there are even two possible applications of the algorithms described herein: Not only can the search algorithm for the individual dimensions be replaced by our one-dimensional algorithm, the back-end multi-dimensional algorithm can also be replaced by a multi-dimensional algorithm as described herein.

8.3 Two-Dimensional Extension

Recall the one-dimensional solution. There we perform a binary search over multiple collections. Each of these collections has a pre-defined set of significant bits and can therefore be accessed in $O(1)$ using a hash table. Among these sets of significant bits, we can set up two ordered comparisons with the operators $x \subset y$ and $x \succ y$, ordering them both in gestalt and priority. Incidentally, these two relations return the same ordering for both criteria. It seems that this definition is both necessary and sufficient to enable binary search over hierarchical prefixes.

How can this definition be applied to two or even more dimensions? Figure 8.1 shows the increasing lengths for one-dimensional matching on the left-hand side. Each square represents a hash table with all the prefixes of length i. Moving up results in a more specific prefix. A natural placement of the length pairs can be seen on the right-hand side. Again, the tuples represent the number of significant bits in each of the two dimensions, and label the hash table represented by the enclosing square. Moving right or up in this matrix results in a more specific entry (one of the prefixes becomes more specific). Moving left or down results in a less specific entry. Moving two steps, one left and one up (or one right and one down), results in a more specific entry along one dimension, and in a less specific entry along the other, resulting in ambiguity.

4	(0,4)	(1,4)	(2,4)	(3,4)	(4,4)
3	(0,3)	(1,3)	(2,3)	(3,3)	(4,3)
2	(0,2)	(1,2)	(2,2)	(3,2)	(4,2)
1	(0,1)	(1,1)	(2,1)	(3,1)	(4,1)
0	(0,0)	(1,0)	(2,0)	(3,0)	(4,0)

Figure 8.1: *One- vs. Two-Dimensional Search*

As seen in Section 8.1, ambiguity cannot be avoided in structural ways. We therefore apply the proven *divide and conquer* strategy [Cae50]. Each of the columns (or rows) in Figure 8.1's matrix fulfills the non-ambiguity criteria, when taken by itself. An obvious solution would be to search each of the columns (or rows) using the one-dimensional binary search scheme. For two addresses of W bits each, this would require searching a $(W+1) \times (W+1)$ matrix, using binary search in one dimension and linear search in the other. Thus, the number of steps would be $O(W \log W)$ or, more concrete, $(W + 1) \cdot \lceil \log_2(W+1) \rceil$. For $W = 32$, this would amount to 198 search steps, too much for modern routers.

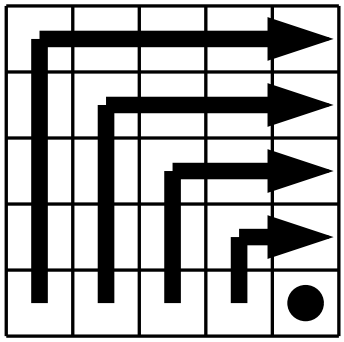
Fortunately, there is hope. Not only is a better solution available, we also expect the classification databases to exhibit a large amount of structure, which can be exploited.

Please note, that unlike the one-dimensional case discussed in Section 4.3, the row (and column) corresponding to prefix length zero is necessary in the multi-dimensional search. This is due to the fact that—except for the prefix length pair $(0, 0)$—the other prefix length is non-zero, providing for non-zero information.

8.3.1 Faster Than Straight

To improve on the row-by-row scheme presented above, recall that the number of memory accesses for binary search grows logarithmically to the number of prefix pairs covered. It is thus better to use fewer binary searches, each covering more ground.

Further recall that the entries get more specific both in vertical (up in Figure 8.1) and in horizontal direction (left). By combining a path in both directions, it is possible to create a sequence of completely order prefix lengths which is longer than a single row or column. Figure 8.2 shows a set of such longest paths. Let us call such a path collecting unambiguous prefix length pairs a *Line*.



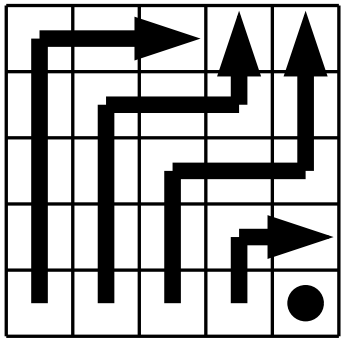
9 7 5 3 1

Figure 8.2: *Longest Possible Lines in a* 5×5 *Matrix*

In the naïve solution, a 5×5 matrix was covered by 5 Lines of length 5 each, each requiring 3 search steps, totaling 15 search steps. Now, the matrix is covered with 5 Lines of varying length, summing up to $4+3+3+2+1 = 13$ search steps. Larger matrices allow for a higher yield. So the ratio for an 8×8 matrix equals $32 : 24$.

Against the intuition presented earlier, making lines as long as possible is not the optimal solution. Recall that the number v of binary search steps required to cover C prefix length pairs is $v = \lceil \log_2(C+1) \rceil$ (Section 4.3). Going from a Line of length 7 to one covering 8 cells also increases the number of steps from 3 to 4, going from a coverage of $7/3$ cells per search step down to $8/4$.

Therefore, it is not only advisable to make the lines as long as possible, but also to cut some off just below powers of two. Figure 8.3 shows an example with a better line configuration. The longest line has been cut off at length 7 to save a search step, the second line is kept at that size, but changes path to keep nestling up against the first. The third line is extended to length 7 to cover the two cells freed by the other two lines, at no additional cost. Thus, the total number of search steps amounts to $3 + 4 + 3 + 2 + 1 = 12$, a further improvement.



7 7 7 3 1

Figure 8.3: *Optimal Lines in a* 5×5 *Matrix*

It can be shown that this solution is optimal. Lines with optimal length can be built by the algorithm in Figure 8.4. "Spare" cells are cells that could be covered at no additional cost, if the current lines would be extended to the maximum length

Now we have reached one goal, making lines longer. But we haven't yet reduced the number of lines. Unfortunately, the number of elements on the co-diagonal is one of the limiting factors. Since all of the prefix length pairs on the co-diagonal are ambiguous to each other, they provide a lower bound for the number of Lines. So do the other cuts parallel to the co-diagonal and generally all other sets of mutually ambiguous prefix length pairs.

8.3.2 Expected Two-Dimensional Classification Databases

As discussed in Section 3.3, until very recently, no feasible approach for multi-dimensional classification was available, short of slow and tedious linear search through the entire database. Therefore, no one has started creating large classification databases. Nevertheless, we expect demand for using

Function OptimalLines(W) (* Build Lines for Size W *)
(* Calculate memory accesses for each Line *)
Initialize s to 0; (* spare cells *)
For $l \leftarrow 1$ to W do
  (* Calculate longest Line along the outer border of a square *)
  (* of length l, taking into account and updating spare cells. *)
  (* $2l - 1$ is border length, c_l is coverage, m_l is search steps. *)
  $c_l \leftarrow$ smallest (power of 2) $-1, \leq 2l - 1 - s$;
  $m_l \leftarrow \lceil \log_2(c_l + 1) \rceil$;
  (* Update spare counter by surplus/borrowed cells *)
  $s \leftarrow s + c_l - (2l - 1)$;
  If cells were borrowed then
    Extend the most recent Lines which can cover more cells
      ($m_i^2 - 1 > c_i$) until borrows are satisfied;
  Endif
Endfor

Figure 8.4: *Build Optimal Lines For Full Matrices*

such databases to become real within the next few years. Until then, we cannot but generate our own sample databases. To provide for a wide variety of databases, covering a large part of the possible spectrum, we devised four benchmark scenarios, described below.

Full This is the simplest scenario, but the most expensive to solve: All possible prefix length pairs will show up in the database, giving a full $(W + 1) \times (W + 1)$ matrix.

Chess In the manner of a checkerboard, only every alternating matrix cell of prefix pair lengths contains prefixes.

CIDR This pattern consists of the prefix lengths that are most likely to appear, so all (x, y), where $x, y \in \{0, 8 \ldots 30, 32\}$ are assumed to contain prefixes. Lengths $1 \ldots 7$ are excluded since they are not part of the CIDR [RL93, FLYV93] specification. Length 31 is not part of the set since most of the checked one-dimensional routing databases do not contain entries of that length. This is due to the fact that the two addresses included in that range cover more than a single host, but not enough to cover a reasonable network (the first and last address in each network cannot be assigned to machines).

Random This is actually based on real entries, and comes in two flavors, Random1000 and Random5000. To create this database, 1000 (5000) random prefixes were picked from the Mae-East database. Of these, 1000 (5000) random pairs were constructed. 10% of these pairs had one entry prefix replaced by a default prefix (with zero length). This is based on the assumption that classifiers will be biased towards some prefixes, and that tuples only specifying either source or destination filters will also be common. Additionally, the database contains the "default" prefix pair with a (0,0) length tuple.

These five benchmarks (Full, Chess, CIDR, Random1000, and Random5000) will be used for analysis below.

8.3.3 Lines for Sparse Matrices

In Figure 8.4, we have seen how to build optimal Lines for full matrices. For sparse matrices, no algorithm for building optimal Lines is known, short of exhaustive search. We have devised a number of heuristics. Each of these tries to build the largest possible Lines, but some of them cut Lines down.

To find the longest Lines, a directed acyclic graph of subset relations is built. By labeling each vertex with its depth in the graph, the vertex with the highest number is the end of the longest Line. This Line is removed, and the process repeated, until the graph is exhausted.

The algorithms for cutting Lines are as follows:

Simple No cutting is done, the longest Lines are used.

Log All Lines are cut to the maximal length in the form $2^x - 1$, optimizing the coverage per search step.

AlmostLog Only Lines "just above" an optimal length are cut down, i.e., those with lengths of the form $2^x \ldots 1.5 \cdots 2^x$.

8.3.4 Performance Analysis

In the previous sections, five benchmark databases (full, chess, CIDR, Random1000, and Random5000) have been introduced. Table 8.1 compares the

performance of different Line selection algorithms for these benchmark scenarios. The algorithms are called Original (Lines are parallel, either all rows or all columns), Optimal (Figure 8.4, and the three heuristics for sparse matrices (Simple, Log, and AlmostLog).

As can be seen, Selection according to the AlmostLog criteria is up to 25% faster than our first, naïve idea, especially on the random distributions, those we deem most representative for future classification databases.

Note that for reasonably sparse matrices, such as the results from the random distributions, two-dimensional classification is only an order of magnitude more expensive than the fastest one-dimensional lookups. We expect that a two-dimensional generalization of the Rope paradigm would give an additional performance boost.

Benchmark	Full	Chess	CIDR	Rnd1000	Rnd5000
DB Size	1089	544	625	1001	5001
Prefix Pairs	1089	544	625	88	141
Original	198	165	125	42/40[a]	56
Simple	168	140	119	30	46
Log	164	132	111	31	48
AlmostLog	159	130	111	30	45
Optimal	159	*Unknown*			

[a] 42 was achieved when splitting along the columns, 40 when splitting along the rows. This is the only test where data organization made a difference.

Table 8.1: *Line Search Performance (Memory Accesses)*

Since many of the benchmarks do not deal with prefixes, but only with the utilized prefix lengths, we cannot do any Rope search simulations, since they would require real data. It is expected that Rope search will improve the performance quite dramatically. While we believe in the representativeness of the prefix length pair simulations above, we doubt that Rope search results based on our synthetic data would bear any resemblance to real data.

8.4 Adding a Third Dimension

Analogous to adding a second dimension, further dimensions may be added. Unfortunately, the lower bound on the number of Lines grows impracticable. In the two-dimensional case, we have seen that the number of occupied cells in the co-diagonal, any of its parallels, and in fact any group of cells which are mutually ambiguous imposes a lower bound on the number of Lines. Similarly, the co-diagonal plane in the three-dimensional cube and all its relatives provide a lower bound for three dimensions. Thus, with all prefix length triples in use, there are $O(W^2)$ lines of $O(\log W)$ search steps each, totaling $O(W^2 \log W)$, clearly impractical, even if many databases will perform much better than that. Generally, for d dimensions, $O(W^{d-1} \log W)$ effort is required. If the dimensions should differ in size, with W_i the number of bits necessary to represent the address range in dimension i, $O(\log W_d \prod_{i=1}^{d-1} W_i)$.

8.5 Collapsing Additional Packet Classification Dimensions

As can be seen from the previous section, adding dimensions after the second does apparently not lead to efficient solutions. Therefore, we try to change our goal and reduce the number of dimensions needed, instead of adding dimensions we can support.

For this section, knowledge of the additional header fields used in classification is advantageous. If you do not feel familiar enough with the semantics of these fields, it is advisable to refresh Section 2.2.3 before proceeding.

8.5.1 Collapsing a Wildcard Matching Dimension

As can be seen from Table 2.6 on page 15, full prefix matching is only required for the source and destination addresses. For all the other fields, much more limited matching methods are sufficient. Assume the addition one of the *wildcard* fields, such as the protocol ID. Instead of adding this as full-fledged dimension in its own right, we add it as an additional layer to dispatch between multiple two-dimensional search structures.

To dispatch, all the valid protocol IDs are stored in an array or a hash table. Each of these entries points to a Line search structure, to perform the source/ destination address matching. Each of these Line search structures only contain the entries of the database which contain the appropriate protocol ID. Additionally, there is a Line search structure containing all entries where the protocol ID is a wildcard. Given a packet, it is classified as follows. First, the protocol ID is looked up in the initial array or hash table. If found, the referenced two-dimensional structure is searched and the best match remembered. Independent of the execution of the previous step, the additional structure for wildcard protocols is searched. Then, the search result with the higher priority is used to further process the packet.

In the worst case, this approach only doubles the search steps, compared to an up to fourfold slow-down if the protocol ID were considered a full-fledged eight-bit dimension. By sacrificing some memory, the addition of the third dimension may not even affect the performance. By including all the relevant data from the wildcard structure into the individual fully-specified sub-databases, the additional search of the wildcard structure can be avoided entirely. Although no large databases are available to support this claim, we believe that the search structures associated with the defined protocol IDs will not be extended significantly.

8.5.2 Collapsing a Limited Range Matching Dimension

Some fields, such as the port fields, do not only require exact matching and wildcard fall-back, they also require a small number of ranges. The only known ranges used for ports are $[1, 1023]$ (privileged ports and well-known services [Pos81b, RP94]) and $[6000, 6063]$ (X Window System (X11) [Nye95, RP94]).[1] Extending wildcard matching to support these is straightforward. After searching the exact match (if it exists), a range is searched (if appropriate), and then the wildcard default is searched. This requires only three times as many searches as a plain two-dimensional classification.

Since the number of supported ranges is a small constant, it makes sense to avoid the third search in the wildcard database, by including the relevant

[1] Some sources refer to the X11 reserved range as $[6000, 6100]$. According to both [RP94, IAN], the authoritative sources for Internet number assignments, only the first 64 ports are in fact reserved for X11.

entries into the two range databases, sacrificing a small amount of memory for a significant worst-case speed improvement.

8.5.3 Multiple Fields

Obviously, this strategy can also be extended to matching multiple of these fields. The classical problem in Internet packet classification is the five-tuple matching: source/destination addresses, protocol ID, and source/destination ports. One of the factors that simplify this five-tuple matching is that only two protocols, UDP and TCP, do have port numbers defined. All others have no notion of protocols, so port matching is not necessary when searching the wildcard protocol ID. Instead, these entries are just added to both the UDP and TCP databases, again sacrificing a negligible amount of memory for a significant speedup.

Figure 8.5 shows the decision tree that is to be used, limiting the two-dimensional searches for any path to at most 5. If each of these dimensions were treated as prefix matches, the three fields of length 8, 16, and 16 bits (protocol, source port, destination port, respectively), would have resulted in an increase in the number of search steps by several orders of magnitude. Although Figure 8.5 shows a sequential order, all the decisions can be made before starting the first lookup, and all lookups can be performed in parallel, since there are no interdependencies.

Although the idea is similar to the one described in [SVSW98], this description goes much further and even allows for a limited amount of ranges on port numbers.

8.6 Matching Ranges

Although matching prefixes is the natural form of matching for Internet addresses, many other problems are more commonly treated as range matching. Already in the Internet world, we have seen that port numbers sometimes are treated in ranges (e.g., port numbers Section 2.2.3). In this chapter, we will explore how to extend binary search to match ranges. Additionally, a short discussion on optimizing Internet packet forwarding using ranges is also included.

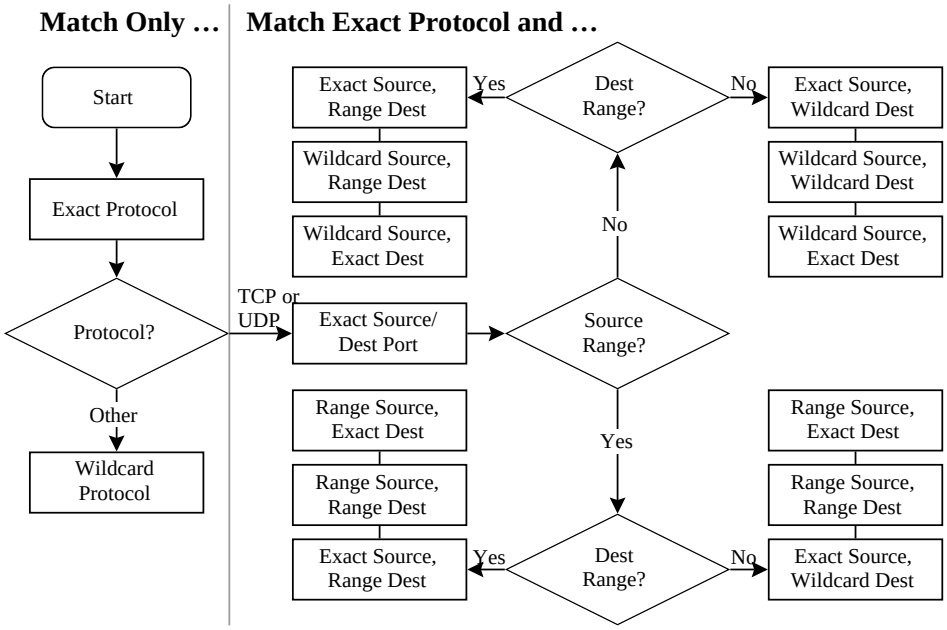


Figure 8.5: *Collapsing Multiple Dimensions*

As discussed earlier in Section 2.4 and visualized in Figure 2.1 on page 21, (integer-valued) Range matching and Longest Prefix Matching are specializations of the Prefix Matching problem. Figure 2.1 showed that there is some relation between the two descendants. Their relationships are the topic of this section. A short section each is dedicated on how to map one onto the other and vice versa.

8.6.1 Ranges Expressed Using Non-Overlapping Prefixes

Before using longest prefixes to express ranges, we start by looking at using non-overlapping prefixes. Figure 8.6 shows an example of covering a range with multiple prefixes.

The general algorithm to cover arbitrary ranges by prefixes is shown in Figure 8.7. 2^p, the size of the next prefix, is calculated so that it is a prefix and does not extend beyond either end of the range. Recall that a prefix is defined such that its length is a power of two and a prefix of length l must start at an integer multiple of l. Unfortunately, this listing does not show an apparent upper bound on the number of prefixes necessary to cover the given range.

By changing the algorithm to narrow from both sides, we are able to limit the number of iterations and thus the number of prefixes necessary. This is

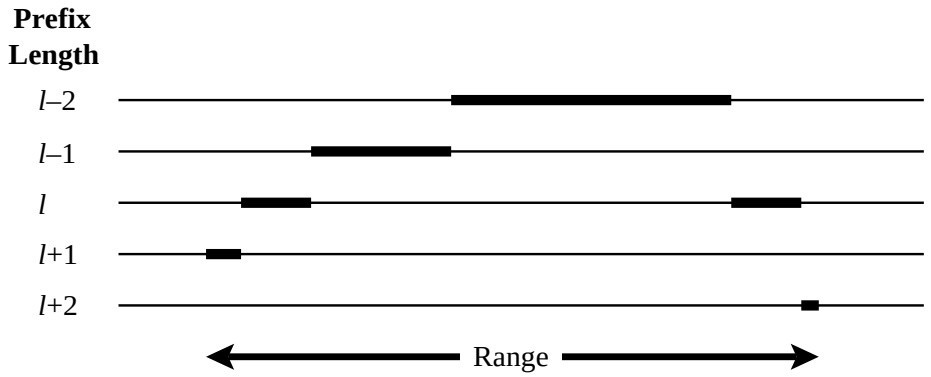


Figure 8.6: *Covering Ranges Using Non-Overlapping Prefixes*

Function RangeCovering(s, e) (* Build Lines for Range $[s, e)$ *)
While $s < e$ do
  (* $LSB(x)$ is the bit number of the least significant bit set in x *)
  $p \leftarrow \max(LSB(s), \lfloor \log_2(e - s) \rfloor)$;
  Emit prefix covering $[s, s + 2^p)$;
  (* Limit range to what remains *)
  $s \leftarrow s + 2^p$;
Endwhile

Figure 8.7: *Range Covering Algorithm*

shown in Figure 8.8. As can be seen, each iteration emits one prefix and clears the least significant bit of either s or e. If both s and e are W-bit numbers, this process has to terminate after at most $2W$ steps. A closer limit is $2(W - I)$ bits, where I is the number of identical contiguous bits, measured from the most significant bit.

It is even possible to calculate the exact number of prefixes required without running the algorithm. The number of prefixes created on the upper end (near e) is simple. Because the subtraction clears the least significant set bit without side-effects, this requires as many bits as there are bits set to one in the least significant $W - I$ bits of e, i.e., in the "non-common" area.

On the lower end (near s), things are a little bit more complicated, since the addition which clears the least significant bit creates a carry, possibly changing higher bits. This can be fixed by first calculating the two's complement of the least significant $W - I$ bits of s before counting the set bits.

Function RangeCovering(s, e) (* Build Lines for Range $[s, e)$ *)
While $s < e$ do
  If *LSB*$(s) <$ *LSB*(e) then
    (* Start requires smaller prefix to cover, do this first *)
    $p \leftarrow$ *LSB*(s);
    Emit prefix covering $[s, s + 2^p)$;
    $s \leftarrow s + 2^p$;
  Else
    (* End requires smaller prefix to cover, do this first *)
    $p \leftarrow$ *LSB*(e);
    Emit prefix covering $[e - 2^p, e)$;
    $e \leftarrow e - 2^p$;
  Endif
Endwhile

Figure 8.8: *Narrowing Range From Both Ends Simultaneously*

Now we have covered both ranges. Concluding, the number of prefixes to cover the range $[s, e)$ is the sum of the number of bits set in the least significant $W - I$ bits of e and the number of bits set in the least significant $W - I$ bits of the two's complement of s or *bitcount*$(e \star (2^{W-I}) - 1) +$ *bitcount*$(-s \star (2^{W-I}) - 1)$. In the previous formula, *bitcount* counts the number of set bits, $\star$ is the bitwise and, and the unary minus calculates the two's complement.

8.6.2 Ranges Expressed Using Longest Prefix Matching

In the previous section, we have shown that it is possible to cover an arbitrary integer-valued range by at most $2W$ non-overlapping prefixes. Since Section 2.4 pointed out that overlapping prefixes with longest prefix matching are more powerful than their non-overlapping brothers, it should be possible to reduce the number of prefixes needed to cover a range by switching to the more powerful mechanism.

Indeed, switching to longest prefix match gets rid of one of the restrictions we have seen in Section 8.6.1. The need to confine the prefixes entirely within the covered range is gone. Instead, they can cover a larger area and the surplus region can be corrected using additional prefixes, as already employed in Section 5.5.1.

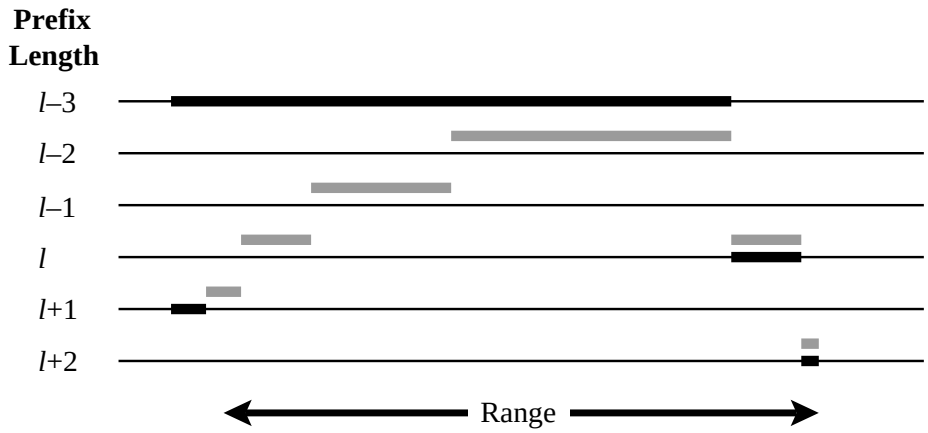


Figure 8.9: *Covering Ranges By Longest Prefixes*

Figure 8.9 shows the same range to be covered. In gray is the solution achieved in Figure 8.6. This is a bad example for plain prefixes: On one end, a range just smaller than a power of two is to be covered. Using longest prefix matching, instead of requiring almost W ranges, only two are needed. The entry labeled 1 covers the power of two, just a little bit too much. This is corrected by entry number 2, containing the information that would have been returned in that area had entry 1 not covered too much. Since entry number 2 is more specific than number 1, it automatically takes precedence. If the neighboring range is covered, the corrective entry has already been inserted when the neighboring range was set up, so the number of entries due to "our" range is reduced even further.

We have seen that in many cases, the number of prefixes can be reduced significantly. But does it improve the upper bound? Recall that the number of prefixes depends on the number of bits set in the coordinates of the endpoints. If the bits set is more than $W/2$, we can now cover the next higher power of two with an entry, toggling all bits. Therefore, there are now less than $W/2$ bits set, requiring *less than* $W/2$ prefixes. Together with the helper entry used to toggle the bits, *at most* $W/2$ prefixes. This can be done on both sides, reducing the worst case prefixes from $2W$ to W by switching from non-overlapping prefixes to a longest prefix matching algorithm.

In many cases, it is possible to optimize this even further. Should the reverted range still contain a bit sequence with high frequency of set bits, these can be toggled by inserting further prefixes. Geometrically, this repeated toggling represents alternatively overshooting the assigned border, changing between covering too much and too little. Generally, toggling a bit sequence

requires 2 prefixes, but if it terminates at $W - I$ bits, only a single prefix is required. Table 8.2 shows an example of repeatedly overshooting one end, when covering the range $[0000\_0000\_0000, 0111\_0000\_1111)$. This reduces the number of prefixes required from 8 using plain prefixes, over 5 using longest prefix matching to 3.

Extracted Prefix	Remaining Range	Position
Start	[0000_0000_0000, 0111_0000_1111)	In
0*	[0111_0000_1111, 1000_0000_0000)	Out
0111*	[0111_0000_0000, 0111_0000_1111)	In
0111_0000_1111	$\emptyset$	

Table 8.2: *Modeling a Range By Overshooting Repeatedly*

8.6.3 Longest Prefixes Expressed Using Ranges

Above, we have seen the close resemblance between sequences of prefixes and ranges. This can also be advantageous the other way round, by expressing prefixes as ranges. Having both possibilities will allow us to switch from one representation to the other, resulting in a way to improve the number of prefixes needed to represent the original information, e.g., minimizing the number of entries needed in complex routing tables.

As discussed earlier in Section 2.4, a single change in a database of N prefixes expressed as up to $2N$ ranges, can require changes in N ranges, which is clearly impractical whenever response time to updates is an issue. But there are applications where the advantages of coding entries as prefixes outweigh this disadvantage. This is especially true when the ranges are encoded as overlapping prefixes again, in which case the above-mentioned drawback no longer applies, and fast updates can be achieved using the technique described in Section 5.4. Such an application are Internet forwarding tables. Our work involving such tables has shown that these tables are often described inefficiently and that further or different aggregation may reduce the routing table size significantly. It is common to find successive prefixes, which all contain the same information, i.e., point to the same next hop. In this case it may be desirable to recode such sequences more optimally, using less prefixes. One technique is described below.

Minimizing Routing Table Entries

A simple technique is to describe a sequence of prefixes having the same information by a single range, and then optimally encoding such a range by possibly overlapping prefixes, such as described in Section 8.6.2. This can be done in $O(NW)$ time and $O(W)$ space.

Besides optimally encoding successive prefixes, this also optimally encodes overlapping prefixes, where a more specific prefix contains the same information as its outer prefix. As an additional bonus, it also covers the case where prefixes only almost succeed, and either contain a small hole between them, which does not contain any information, or a small range containing different information. This is already done by the scheme in Section 8.6.2. There, ranges which are covered by overlapping longest prefixes whose borders overlap will generate some identical prefixes, which are automatically merged. A more complex scheme achieving a similar result, including an optimality proof, can be found in [DKSZ99].

Beyond the Optimum

Although [DKSZ99] contains an optimality proof, it is possible to improve on that. We do not show that the proof shown therein is in error, instead we show that one of the boundary conditions they assume and adhere often can be relaxed.

Above, we have seen that almost successive ranges may contain holes. Although these holes are rendered efficiently when recoded using ranges, there is an even better way to encode these holes: Not at all. This works efficiently, if some router further downstream still has the knowledge that there really is a hole. In a default-free world, this automatically happens whenever the packet addressed to a non-existent address in the hole reaches a choke point, a point where the hole is left alone, and not merged with either side. This happens latest, when the range just below the hole needs to contain information different from the range just above the hole.

Should the packet leave the default-free part of the Internet, then this solution still works perfectly, if the hole belongs to an address space allocated to the same entity as the bordering ranges. Then this entity (usually an Internet Service Provider, ISP) can return the correct ICMP error message to the

source, notifying it of its use of an unused address. If this is not the case, the packet will bounce forth and back between the default-free zone and the ISP, and eventually be discarded because it exceeded its lifespan. Then, some more resources have been utilized and a possibly misleading message that the packet looped before it reached the destination will be sent back. Since packets should not be sent to an inexistent destination frequently, we believe that this is just a minor inconvenience and will not cause any network problems.

8.7 Summary

In this chapter, we have shown several techniques enabling efficient two-dimensional longest prefix matching in general. To test its performance, we developed models for possible future two-dimensional classification patterns. For more than two dimensions, a native three-dimensional algorithm has been introduced that is able to perform fast searches when the sizes of sets of mutually ambiguous prefix length tuples remains small. More importantly, we have shown an efficient scheme to match 5-tuples used for Internet packet classification.

We have further shown the close relation between range matching and longest prefix matching has been laid out. Techniques for mapping them onto each other have been introduced, including analyses on the impact of the conversions. Based on the two conversions, an algorithm optimizing routing table entries, thus minimizing memory requirements has been presented. Further, a strategy for further improvement has been proposed, which can also further improve another algorithm which was proved optimal.

Chapter 9

Applications

In the previous chapters, we have introduced an algorithm to perform fast and scalable longest prefix matching. We have also discussed several optimizations that allow this algorithm to improve the number of search steps required by adapting closely to the database searched. We have also shown some generalizations, notably to match arbitrary ranges and the extension to match multiple dimensions. All of this work has been focusing on application in an Internet world, to be utilized by network nodes such as routers or firewalls.

In this chapter, we will show the versatility of the algorithm, its improvements, and extensions by introducing applying it to solve a number of different problems unrelated to packet networks. We will cover topics from such diverse topics such as geographical information systems (GIS), memory management, databases, and access control management. Many of these are still topics of ongoing research and require further work.

9.1 Geographical Information Systems

Geographical information systems [BM98, LGMR99] often have queries related to locating points in landscapes [KNRW97], similar to classical point location [BKOS97]. A number of algorithms and publications exist on point lo-

cation, which often relates to range queries with non-integer boundaries. The best known two-dimensional algorithms achieve $O(N)$ storage and $O(\log N)$ search time with $O(N \log N)$ build time, which sounds excellent. To achieve these results, [Sei91] has to revert to randomization, impractically high constants, and a number of tricks such as shearing all the data, when two points happen to have the same X coordinate. Other solutions have similar drawbacks. In higher dimensions, the point location problem is still considered very much open [BKOS97]: No general solutions with reasonable bounds have been found.

Below, we show several practical cases where longest prefix matching opens new possibilities or leads to improved algorithms.

9.1.1 Proximity Queries

One area where prefix matching performs well is what we call the *proximity query*. The goal of proximity queries is to find the point closest to the given query point offering some particular service. E.g., in an in-flight emergency, it can prove vital to quickly locate the nearest airfield to land on.

The database consulted for such a query might be structured as follows. For each airfield, the region of points closer to this than to any of the others is calculated in advance and stored in a database. Although the area usually is pretty symmetrical, it will consist of ranges, possibly even describing the borders using floating-point numbers or a function. To store such two-dimensional range information using prefixes will require large amounts of prefixes, impractical for most situations.

There are better possibilities to store such coarse-grain tiles requiring fine-resolution borders. Instead of modeling the border exactly within the data structure, we only model the border coarsely. Then, within the record found, we define the borders in the desired resolution or even as a function. Then, the decision which side of the border the query point lies can be solved in a short time. When the borders consist of lines, the answer is found after at most b comparisons, where b is the number of directly bordering regions. When the borders are defined by a function, the function defines the cost. Figure 9.1 shows an example.

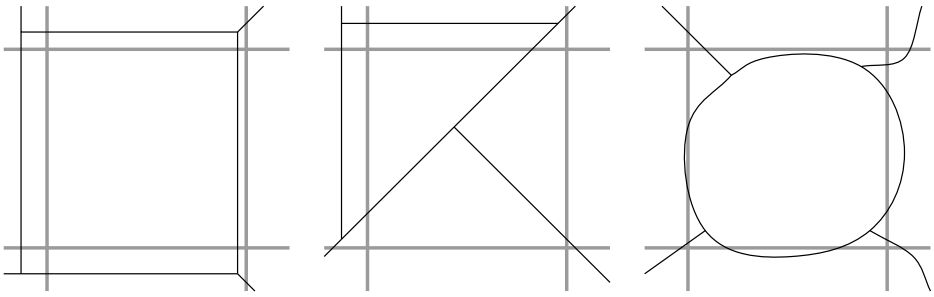


Figure 9.1: *Examples of Coarse-Grain Tiles With High-Resolution Borders*

9.1.2 Squares

In the proximity query introduced above, all regions have borders whose outline is within a constant factor from a square, i.e., $length/width \leq f$. Then there is a better algorithm than using two-dimensional classification, which allows for easy extension to arbitrary dimensions. This is done by bit-wise interleaving of all the coordinates. As such, regions which would be represented using a pair of prefixes of the same length in the proximity query scheme (Section 9.1.1), will be represented using a single database entry. A region which whose prefix lengths differ by one will require one or two entries; a region whose prefix lengths differ by two requires two or four, and so on (Table 9.1). In this table, question marks (?) symbolize a single unknown bit and, as usual, Asterisks (*) mark the remaining bits as unknown. The changing bits in the database entries are marked in **bold**.

Coordinate		Interleaved	Database
First	Second	Pattern	Entries
1111*	0000*	10101010*	10101010*
11111*	0000*	101010101*	101010101*
1111*	00000*	10101010?0*	1010101**0**0*, 1010101**1**0*
111111*	0000*	101010101?1*	1010101**0**1*, 1010101**1**1*
1111*	000000*	10101010?0?0*	1010101**0**0**0**0*, 1010101**0**0**1**0*, 1010101**1**0**0**0*, 1010101**1**0**1**0*

Table 9.1: *Address Interleaving*

In general, each region is extended to at most $2^{(d-1)f}$ database en-

tries, each with the flexibility described in Section 9.1.1, allows search in $O(\log(dW))$, where d is the number of dimensions, and W the number of significant bits in each coordinate. This allows for extremely fast lookups. Still, for small values of f, the resulting memory expansion is negligible.

In case the extents of the regions should only be comparable in some of the dimensions only, these dimensions can be merged into one. This merged dimension can then be used as a single dimension, together with the unmergeable dimensions, in "conventional" multi-dimensional lookups.

9.1.3 Efficient Bit Interleaving

Before entering the realm of more complex shapes than the square, we need to gain some background knowledge on bit interleaving, which simplifies the understanding of the following section.

Unfortunately, modern processors do not support bit interleaving well. This means that interleaving the coordinates would be slow and tedious. Instead of interleaving the bits, the regions to would be interleaved in the algorithms described above, can also be extracted from the the different coordinates and simply be concatenated. If all interleaving operations in an algorithm are replaced by concatenations of the appropriate bits, the algorithms can remain otherwise unmodified. This simple change results in a significant performance improvement.

When using binary search on hash tables as a back-end, is not even necessary to perform the interleaving. It is enough to feed the relevant prefixes of the coordinates into different hash functions each and combine the resulting partial keys. To check for a match in the hash table, each coordinate is then compared on an individual basis.

This shows that the bits themselves do not really need to be interleaved for the operations to work. This also means that specifying the number of bits per coordinate is sufficient and there is no need to specify their exact order. This is crucial for what we will encounter in the next section.

9.1.4 Rectangles

In Section 9.1.2, we have seen that squares, cubes, and hypercubes can be efficiently stored searched by interleaving coordinates bitwise before adding them to the database or searching for them. It has also become clear that the further the region shape evolves from a square, cube, or hypercube, the more entries are needed to describe them. Reducing the number of entries for these rectangular items is thus the topic in this section.

Looking at Table 9.1, it can be noticed that the number of entries required does not only depend on the difference in prefix length, but also on the order of the coordinates. Put differently, the efficiency of storing a region in the database depends on the relation between the exact sequence of interleaving and the shape of the region. If a majority of the regions should have a similar, non-rectangular shape, then the interlacing scheme can be changed, e.g., to take a single bit from the first coordinate, then two from the second, then again one from the first, and so on. This results in an effect similar to performing a coordinate transformation. While it can be used for simple stretching transforms, it can also be used to apply nonlinear transforms. This makes it possible for large-scale objects to have one preferred orientation while smaller-scale objects may have a different form factor and tiny objects looking differently again.

Conceptually, the global skew towards some address bits resembles asymmetric binary search as discussed in Section 4.2.1. This gives rise to the question whether there is also a dimensional equivalent to Rope search.

There is. As search proceeds in Rope search, the prefix lengths to be searched can be progressively narrowed down, adapting to the database. Instead of only narrowing down the prefix lengths, in multi-dimensional search, we can also adapt to localized form factors. This is achieved by adding a different amount of bits for each dimension. In this generalized form, each strand[1] of the Rope not only consists of single prefix length, but consists of d prefix lengths, indicating the number of bits to extract from each of the coordinates.

Figure 9.2 shows a sample layout and Table 9.2 an excerpt of the corresponding database, corresponding to the four quadrants, staring at the upper left and proceeding clockwise. The Rope entries specify the prefix lengths for

[1] An individual prefix length specification in the Rope, see Section 4.2.2 on page 44.

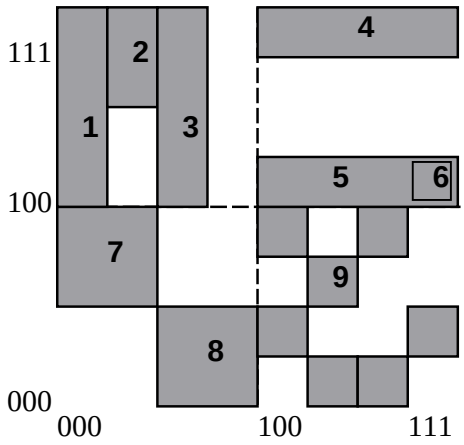


Figure 9.2: *Sample Rectangular Layout (Coordinates Given in Binary Notation)*

Coordinate			
x	y	Type	Rope
*	*	Start	(1,1)
0*	1*	Marker	(3,2), (3,1)
000	1*	Entry 1	—
001	11*	Entry 2	—
010	1*	Entry 3	—
1*	1*	Marker	(1,3)
1*	111	Entry 4	—
1*	100	Entry 5	(3,3)
111	100	Entry 6	—
0*	0*	Marker	(2,2)
00*	01*	Entry 7	—
01*	00*	Entry 8	—
1*	0*	Marker	(3,3)
101	010	Entry 9	—
⋮			

Table 9.2: *Excerpt of Database Corresponding to Figure 9.2*

each coordinate in a tuple. Already in this small coordinate system the advantages of two-dimensional Ropes can be perceived. Two-dimensional binary search as introduced in Chapter 8 would require 4 search steps, while only three steps are needed here. With larger universes, the advantage will grow even further.

Often a large number of different form factors exist within the same region or a number of vastly different form factors coexist. Then, it is also possible to trade off some search speed for a smaller memory footprint. This is done by enhancing or—depending on the standpoint—slightly abusing the Rope mechanism. In the one-dimensional Rope, walking and guiding only occurs along a single subset relation, i.e., between ordered prefix lengths. Multi-dimensional Ropes can be freed from this restriction and can guide among the predominant rectangle geometries.

9.2 Memory Management

There are a number of issues in memory management, where fast longest matching prefix algorithms may help providing solutions. Exemplarily, management for persistent objects and application-transparent garbage collection and memory compaction will be discussed.

9.2.1 Persistent Object Management

Management of object persistence [Ses96] can be classified whether the application needs to be aware of the underlying persistent storage or not. While it is easier to write applications that do not need to be aware of the database and look like plain object-oriented applications, the underlying middleware needs to know when and how objects have been accessed. For read accesses, it may need to (re-)fetch the object from the database, and for write accesses, it needs to mark the object *dirty*, scheduling a later write-back.

The detection of accesses is done by first protecting the object's memory region against all accesses. The first time the object is accessed, the processor traps to the operating system, which forwards this event to the middleware. The middleware then fetches the object from the database, changes permissions to allow reads, and asks the operating system to continue the trapped

instruction. On a write access, the middleware is again notified, marks the object as dirty, removes all access limitations, and resumes program operation.

The best performance is achieved if applications can run natively, i.e. outside an interpreter or virtual machine. Unfortunately, current processors do not support fine-grained memory management, but rely on segmenting memory into pages of 4 or 8 KBytes [Tan87]. This usually results in either coarse-grained access control, when multiple objects are put into the same page, or a waste of physical memory.

Itzkovitz and Schuster [IS99] recently proposed a system allowing multiple objects to share the same physical page while keeping fine-grained read/write logging. Their system clobbers entries in the Translation Lookaside Buffers (TLB), where the processor's memory management unit (MMU) keeps a cache of address translations and access permissions, thus reducing memory access performance. Additionally, the middleware will become more complex as a result of this change. Further, access faults, such as addressing beyond the end of an array, will be harder to catch, complicating program debugging.

We believe that the solution is to provide for a finer-grain memory management. Moving from fixed-size memory pages (typically 2 to 8 KBytes today) to variable sizes allows to adapt better to today's needs: Large, contiguous blocks of memory, which currently use up many pages (and thus many of the very limited entries in the TLB cache), can be represented in a single translation entry. Blocks smaller than the page size, which need to be protected individually, can use a more compact "local page size" and do not waste an entire memory page.

Such fine-grain memory management can easily be accomplished by using binary search on prefix lengths together with a small TLB cache made of content-addressable memory (CAM). Hashing for page table lookups already is a common practice in the PowerPC processor [MSSW94], as a second-level cache beyond the TLB cache. Modifying the algorithm to do binary search on the prefix lengths will give additional flexibility at minimal cost.

Besides object persistence, many operating system extensions [CT90, BCWP98] try to improve I/O throughput by zero-copying. Fine-grain memory management helps these efforts by reducing the granularity. In addition, also coarse-grain management is made possible by switching to a longest matching prefix solution. Coarse-grain memory management also often a desirable

feature in current computer systems, since it improves the hit rate of the TLB.

Other applications where we envisage improvements include faster inter-process message passing through fine-granular memory mapping, and improved memory protection and access checks. Such hardware-assist can be used for improved run-time bounds checking, similar to what is currently being done by several products in software [Rata].

9.2.2 Garbage Collection and Memory Compaction

Many popular processor architectures, such as Alpha [Sit92] and Sparc [WG94] have already performed the transition from 32 bit address spaces to 64 bit addresses, and others, such as Merced (now Itanium), the result from the Intel-HP joint processor development program, are following this trend. While this only doubles the number of address bits, the available address space immediately grows by a factor of 2^{32}. Physical memory growth behaves closer to Moore's law, which predicts doubling of processor speed ever 18 months. While the address space growth will remove all practical limits on virtual memory for the foreseeable future, physical memory will become relatively scarce, compared to the vast address space. If the memory requirements of applications continues its fast growth, the relative memory scarcity will also turn absolute.

Many modern run-time environments support garbage collection, the automatic deallocation of unused memory. This provides for an efficient use of memory resources and at the same time frees the developers from the error-prone task of remembering the allocation status of individual objects and variables. To make garbage collection efficient, the remaining regions of allocated memory need to be compacted. This separates the wild pattern of free and allocated regions from each other, separating the two and clustering them together. This reduces fragmentation of the free space, making it ready for re-allocation by differently sized objects. Also, some chunks of free space may be returned to the operating system to make them available to other applications in need of memory.

Memory compaction is a complex process, especially if it has to happen in the background. Not only do all objects need to be copied, polluting the processor's data cache, but all references to these relocated objects need to

be updated, requiring more memory operations and severely restricting the efficiency of background operation.

This efficiency can be greatly improved by taking advantage of large address spaces and the flexible fine-grain memory address translation as introduced in Section 9.2.1. Then, physical memory can be compacted on an as-required basis, invisible to the application, by retaining its virtual memory addresses and only performing the collection in physical memory. Parallel operations are also simplified, since the same mechanism also allows physical memory to be protected from application accesses during data relocation.

9.3 Updating Views in Distributed Databases

In a distributed database, several nodes may store data based on an as-needed basis, e.g., using a *View* paradigm. Or a client wants to register a call-back with a server, to be notified, whenever entries matching certain criteria change.

When the distributed system is large or many call-backs have been registered, the traditional mechanism of sequential matching quickly becomes unwieldy. If the matches are exact, the solution is trivial. If the matches can be expressed as longest matching prefixes (or postfixes), binary search on prefix lengths comes in extremely handy. For applications using character strings, it makes sense not to do bit-by-bit matching, but character-by-character matching, a straightforward extension.

Another matching problem that is to be expected to be frequent, is substring matching [Sri99]. We are currently working on extending our prefix matching solution to substrings. Our intermediate results on research in this direction look extremely promising.

9.4 Access Control

Many data services such as web servers [Wil98] provide a hierarchical address space [BMM94] and perform access control based on this hierarchy. Currently, this is done by searching the hierarchy level-by-level after parsing it for level separators character-by-character. With the availability of fast prefix matching solutions which scale well with the length of the prefixes, such

linear algorithms could soon become obsolete and replaced by binary search on prefix lengths.

9.5 Summary

We have seen that the availability of fast prefix matching allows for a number of new and elegant solutions to existing problems, such as geographical information systems, object persistence, garbage collection, distributed databases and access control for Web servers. The vast variety of these issues suggests that prefix matching has more general problems than assumed until now.

Chapter 10

Conclusions and Outlook

In the present thesis, we have developed a new algorithm for longest prefix matching. We have then analyzed and improved it, to take optimal use of the structure hidden in the database to be searched. We have also seen that it compares very favorably with other algorithms known. Based on our one-dimensional results, we have extended the algorithm to two and more dimensions. We have then shown the generality of prefix matching by applying the results to other problem domains.

10.1 Contributions

In Section 1.3 on page 4, we have listed a number of claims to be addressed in this thesis. Now it is time to revisit them. Below, you will find each of these claims assessed.

1. **Fast and Scalable Longest Prefix Matching**
   We introduce a fast and scalable, yet generic algorithm which allows for matching query items against a large database of possibly overlapping prefixes.

   The analysis shows that the prefix matching algorithm based on binary search of hash tables not only requires very few and simple search steps,

logarithmic in the prefix length. It also shows that the performance is completely independent of the size of the database. This not only provides the possibility for the algorithm to be fast, it is extremely scalable in matters of database size and prefix length. The analysis of our implementation in Chapters 6 and 7 supports these statements.

2. **Fast Forwarding**
   We apply this algorithm to Internet packet forwarding and analyze its performance using the largest available databases.

   We have seen that the algorithm works like a charm using even the largest available Internet forwarding databases of around 40,000 prefixes and that the theoretical results hold in practice. With modest amounts of memory high-performance results can be achieved.

   BBN's choice to license and use this algorithm in their GigaRouter [PC$^+$98] supports this claim. The simplicity and performance of our algorithm impressed them so much that they did not use any of the freely available algorithms nor the algorithm they had licensed earlier on for that purpose.

   Our results of using binary search on prefix lengths to improve lookup time also have found their way into several recent textbooks discussing Internet forwarding [PD00, Per99, KR00].

3. **Adaptivity**
   We extend the generic algorithm to a scheme which can self-adapt to structures discovered in the search database, resulting in a further performance boost.

   The novel Rope paradigm introduced allows for even faster convergence to the search result than binary search on prefix lengths. After each successful search step, the range of the remaining lengths to be searched can be narrowed down further, depending on the database. This is done by encoding the next search prefix lengths to search in turn until a hash table hit supplies its own Rope.

   Even for the largest databases, we could demonstrate that the worst case improved to mere four hash lookups. We are confident that with longer addresses the improvement will turn out to be much better.

4. **Efficient Building and Updating**
   We present efficient algorithms for building both the generic and the self-adapting data structures. We also show how to update them both quickly and efficiently.

The analysis of the algorithms presented in Chapter 5 provides evidence to the efficiency of building and updating both structures. The generic algorithm can be built in expected $O(N \log W)$ time, with N denominating the size of the database and W the length of the addresses. Updating will mostly be $O(\log W)$, although the bound for expected performance lies at $O(\alpha \sqrt[\alpha]{N} \log W)$.

Adapting the data structure for the first time requires $O(NW^3)$, each change at most $O(W^3)$. Rebuilding then requires the same cost as the generic algorithm. Incremental updates cost $O(\alpha \sqrt[\alpha]{N} \log W + W^3)$, fast enough for most real-time applications. When new prefix lengths are added, changes in the Ropes are necessary. Although they can be done incrementally, the worst-case search time can degenerate slowly, rebuilds may be necessary after many changes.

5. **Fast Hashing**
   We explain and analyze practical schemes for fast hashing. This scheme is required for the operation of the presented algorithms. We also show that the search structures and hashing can be efficiently combined to yield even better results.

   We have shown how to make hash table lookups possible in a single memory access with modest memory requirements, without requiring complex hash functions or prohibitively long build times. The resulting scheme is well-suited for cheap implementation in hard- or software. Our analysis demonstrated that (re-)moving a few entries would reduce the memory requirement even further. The introduction of *causal collision resolution* allows for an efficient relocation of misbehaving entries, further reducing the memory requirements. Using causal collision resolution, a single entry can be split into two, with different hash keys.

6. **Two Dimensions**
   We extend the scheme to perform two-dimensional prefix-style packet classification. This is required for basic packet classification on source/ destination address pairs.

   With Line search, an elegant extension of the one-dimensional longest prefix matching scheme to two dimensions has been introduced. Although no large two-dimensional prefix databases are available, we have developed a sensible model for what these databases might look like. We have then analyzed the algorithm under different conditions. The results show that the algorithm performs exceptionally well for expected future databases.

We have also presented an algorithm to optimally cover full matrices with Lines and a series of heuristics for covering sparse matrices. The *AlmostLog* heuristics, which tries to use the use the longest possible Lines, unless their lengths are just a little above a power of two, in which case it is cut down to this power minus one. At this number, maximum coverage per number of memory accesses is achieved.

7. **Packet Classification**
   We further enhance the algorithm to an efficient full-fledged five-dimensional Internet packet classification, thanks to known properties of the additional three dimensions.

   We have shown that the naïve way of performing five-dimensional lookups, i.e., doing longest prefix matching in each dimension, will result in bad performance. By knowing that the three additional fields (protocol, source/destination port) do not require prefix matching, but only wildcard matching, maybe with a small number of range matches, our solution requires only 5 two-dimensional lookups instead of 100 to perform five-dimensional packet classification. This makes our classifier both more versatile and more efficient than other known solutions.

8. **Versatility**
   We show that our algorithm is not limited to theory and Internet. Instead, the availability of our prefix matching scheme makes a series of other applications practical for the first time or improves them significantly.

   Longest prefix matching can be used to efficiently model arbitrary range matching. Thamks to our Ropes, it is even possible to perform better than the best known dedicated range matching algorithms, which cannot handle simple overlaps, let alone priorities. We demonstrate how a conversion to range matching and back can shrink the size of the forwarding database, and even propose a scheme that is able to reduce forwarding tables below what others previously had proven to be the optimal solution.

   Besides that, we show that longest prefix matching techniques can provide for major improvements in geographical information systems, middleware supporting persistent databases, garbage collection, distributed databases, and access control.

As the above list shows, our efficient longest prefix matching algorithms fulfills all the claims stated in Section 1.3.

10.2 Outlook

In this section, we present some open issues and unsolved problems related to the improvement of the data structures, their implementations, applications, and Internet routing.

Incremental Updates It is still unclear whether it is possible to perform incremental updates in the Rope search structure. We believe that some restructurings necessary during insertions can require work equivalent to a complete rebuild, unless a localized degradation of performance is acceptable. We believe that an algorithm trying to do all the operations only at insertion time will run into this dilemma sooner or later.

So we put our hopes in pro-active algorithms, that try to foresee critical situations and relax them in advance. As it is impossible to predict the upcoming changes in general, the only practicable way seems to identify all locations where an insertion might require restructuring and change them in appreciation of this change.

Optimal Lines Another unsolved problem is how optimal lines can be calculated in sparsely occupied matrices in polynomial time. Our best heuristic results in the same performance for fully-covered matrices than the known optimal solution. Yet we do not know whether the Lines generated by the heuristic for sparse matrices are also optimal.

Co-Optimizing Ropes and Lines An even less studied problem which might lead to major improvements is the parallel creation of Lines and the Ropes making up these Lines. Then, the optimal Line length will no longer be of the form $2^x - 1$, but will allow for more coverage with the same or a lesser number of search steps.

Optimal Bit Interleaving During the discussion of geographical information systems in Section 9.1, we have seen that bit interleaving can be done flexibly. Defining metrics for optimality of this type of search is hard, since there is a flexible trade-off between memory requirements and search speed involved. Even if criteria were set, it is unclear how to optimize bit interleaving to meet them.

Conditional Moves We are still working on ideas how to take optimal advantage of new processor instructions, such as conditional moves, to further improve software performance of the search algorithms.

Testing Applications Although a number of applications for longest prefix matching have been defined and analyzed in theory, it remains to check out their impact and performance under test and real-world conditions.

Vector Distance Finding the closest match in a multi-dimensional vector space is a challenging problem. There is some relation to finding a longest prefix, or to point location, but it is not obvious how to solve this efficiently. Such problems do occur e.g. in speech recognition. There matching is additionally complicated because the vector to be matched can start anywhere in a sequence of numbers.

Improving Routing We have seen that graph theory has resulted in several routing algorithms that require much less information than the current Internet. Some of these algorithms have very good performance. Are there ways to change our current Internet routing to take advantage of some of the theoretical results? For IP version 6, it is already planned to allow for painless renumbering of network addresses [DH98] and a lot of brain power has been invested into trying to reduce routing table size [HD98, HOD98]. But it is foreseeable that the routing database will either grow fast or routing will be suboptimal. Taking advantage of foreign results therefore seems necessary if the Internet should be able to grow and prosper.

Dynamic Routing The Internet was designed as a packet network in order to automatically route around failing nodes and links. Recent events [SWI99] show that already a single link failure can cause long outages.[1] Besides political issues related to traffic and peering contracts, it is also a technical issue. Since most Internet service providers rely on static or semi-static routing, problems like these aggravate. Finding a solution to either the political or the technical problem would improve the reliability of the Internet.

I hope that the present thesis has raised the awareness for longest prefix matching problems and their background. Longest prefix matching in its various aspects not only shows simplicity and elegance, but also provides for incredible amounts of flexibility. The latter allows it to be tweaked to adapt to the current situation, and to become extremely versatile. I have been amazed about the possibilities. Please let me know when you solve your next problem using longest prefixes.

[1] Due to a link failure in New York, most Swiss research institutions lost connectivity to most of the Internet for about 20 hours.

Acknowledgements

I would like to express my sincere gratitude to my advisors, Prof. Dr. Bernhard Plattner and Prof. Dr. George Varghese, for introducing me into the world of academic research and their invaluable support and patience. Designing and improving the algorithm together with Prof. Dr. George Varghese and Prof. Dr. Jon Turner proved extremely helpful and shaped my understanding of algorithms. All this would not have been possible without the generous agreement of Prof. Dr. Burkhard Stiller, who let me go abroad when my support for the core components of Da CaPo++ was most needed.

Heartfelt thanks go to Thomas Meyer, who analyzed where time was lost during search and who improved the search speed on real-life machines. His work [Mey99] provided the basis for much of Chapter 6; many of the figures in this chapter were done by him. Many thanks also to Marcel Dasen, who co-advised him and brought in his extensive knowledge about processor internals.

I am very grateful to Dr. Christina Class, Dr. Marcus Brunner, and Dr. Thomas Walter, the members of the informal and ad-hoc “SAD”, which provided helpful discussions and constructive criticism about this work.

My thanks go to everyone at the Communication Systems group at TIK, who provided a most enjoyable collaborative and challenging environment: Dr. Daniel Bauer, Christian Conrad, Dr. Dan Decasper, George Fankhauser, Ulrich Fiedler, Gabriela Joller, Ralph Keller, Gabriel Sidler, Caterina Sposato, Nathalie Weiler, Dr. Erik Wilde. The discussions with Matthias Gries and Andreas Romer improved my knowledge about RAM technologies.

I would like to thank the entire Crossbow team for their valuable input, feedback, and support, especially Dr. Hari Adiseshu, Zubin Dittia, Fred Kuhns, and Prof. Dr. Guru Parulkar. I also thank Dr. V. “Cheenu” Srinivasan

for providing us with the Radix Trie code, suggesting fast hash functions for the Pentium, and helping me with my attempts in doing dynamic programming. Subhash Suri gave me insight about the state of the art in point location and geographical information systems.

I am also very thankful for the feedback and the many ideas I received in discussions with Divesh Srivastava, Antony Hosking, and Karsten Schwan.

The biggest thanks of all go to my parents, who enabled me to live a life leading to the current achievements, and, most notably, my wife, Nicola Aschwanden, who had to endure many of my (physical and mental) absences while working on this thesis. Her love and moral support proved invaluable.

Biography

Marcel Waldvogel was born in Winterthur, Switzerland, on August 28, 1968. He attended primary school, secondary school, and gymnasium, from where he graduated with Matura Type C (mathematical/scientific degree) in fall of 1987. After the compulsory military service, he started studying computer science at the Eidgenössische Technische Hochschule (ETH) in Zürich, Switzerland, in October 1988. During his studies, he worked on a wide range of software projects. He was active as a member of the steering committee of the computer science students union (VIS), the CS education committee, and the student parliament (VSETH). In April 1994, he graduated with a diploma degree (Eidg. Dipl. Informatik-Ingenieur ETH). He then started working full-time for the company he had co-founded in 1992 (Uptime Object Factory Inc. in Zürich). During his work as both a networking consultants and a member of the board, the company started flourishing.

In 1996, he left the company to search for new challenges in research and joined the Computer Engineering and Networks Laboratory (TIK) at ETH Zürich. There, he worked on configurable protocols [SCW+99], security for large and dynamic groups [WCS+99], and on scalable packet forwarding and classification techniques. The latter work, resulting in this thesis, started while he was visiting Washington University in St. Louis, Missouri, USA.

He is currently working as Assistant Professor in Computer Science at Washington University in St. Louis.

For parts of the work presented herein, he has been co-awarded U.S. patent number 6,018,524 [TVW00]; another patent is pending.

Bibliography

[AFL97] Werner Almesberger, Tiziana Ferrari, and Jean-Yves Le Boudec. Scalable resource reservation for the Internet. In *IEEE Conference on Protocols for Multimedia Systems – Multimedia Networking*, November 1997.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

[Bak95] Fred Baker, editor. Requirements for IP version 4 routers. Internet RFC 1812, June 1995.

[Bal97a] Tony Ballardie. Core based trees (CBT) multicast routing architecture. Internet RFC 2201, September 1997.

[Bal97b] Tony Ballardie. Core based trees (CBT version 2) multicast routing. Internet RFC 2189, September 1997.

[BBC+98] Steven Blake, David Black, Mark A. Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. An architecture for differentiated services. Internet RFC 2475, December 1998.

[BBP88] Robert Braden, D. Borman, and Craig Partridge. Computing the Internet checksum. Internet RFC 1071, September 1988.

[BCS93] Robert Braden, David Clark, and Scott Shenker. RSVP: A new resource reservation protocol. *IEEE Network,* 7(9):8–18, September 1993.

[BCS94] Robert Braden, David Clark, and Scott Shenker. Integrated services in the Internet architecture: An overview. Internet RFC 1633, June 1994.

[BCWP98] Milind Buddhikot, Jane Xin Chen, Dakang Wu, and Guru Parulkar. Extensions to 4.4 BSD UNIX for networked multimedia in project MARS. In *IEEE Conference on Multimedia Computer Systems*, Austin, TX, USA, June 1998.

[BKOS97] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 1997.

[BKS95] Mark de Berg, Marc van Kreveld, and Jack Snoeyink. Two- and three-dimensional point location in rectangular subdivisions. *Journal of Algorithms*, 18(2):256–277, 1995.

[BM77] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[BM93] Scott Bradner and Alison Mankin. IP: Next generation (IPng) white paper solicitation. Internet RFC 1550, December 1993.

[BM98] Peter A. Burrough and Rachael A. McDonnel. *Principles of Geographical Information Systems*. Oxford University Press, 1998.

[BMM94] Tim Berners-Lee, Larry Masinter, and Mark McCahill, editors. Uniform resource locators (URLs). Internet RFC 1738, December 1994.

[Bri59] R. de la Briandais. File searching using variable length keys. In *Proceedings of the Western Joint Computer Conference*, pages 295–298, 1959.

[BSW99] Milind M. Buddhikot, Subhash Suri, and Marcel Waldvogel. Space decomposition techniques for fast Layer-4 switching. In Joseph D. Touch and James P. G. Sterbenz, editors, *Protocols for High Speed Networks IV (Proceedings of PfHSN '99)*, pages 25–41, Salem, MA, USA, August 1999. Kluwer Academic Publishers.

[BV98] Steven Berson and Subranamiam Vincent. Aggregation of Internet integrated services. Technical report, USC Information Sciences Institute, February 1998. http://www.isi.edu/∼berson/iwqos.ps.

[Cae50] Gajus Julius Caesar. De bello Gallico. Report to the Roman Senate, 50BC.

[CT90] David Clark and Dave Tennenhouse. Architectural considerations for a new generation of protocols. *ACM Computer Communication Review*, 20(4):200–208, September 1990.

[CV95] Girish Chandranmenon and George Varghese. Trading packet headers for packet processing. In *Proceedings of ACM SIGCOMM*, Boston, August 1995. Also in *IEEE Transactions on Networking*, April 1996.

[CZ95] D. Brent Chapman and Elizabeth D. Zwicky. *Building Internet Firewalls*. O'Reilly and Associates, 1995.

[DB95] Luca Delgrossi and Louis Berger, editors. Internet stream protocol version 2 (ST2) protocol specification — version ST2+. Internet RFC 1819, 1995.

[DBCP97] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. In *Proceedings of ACM SIGCOMM*, pages 3–14, September 1997.

[DC90] Stephen Deering and David R. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–110, May 1990.

[DDPP98] Daniel S. Decasper, Zubin Dittia, Guru Parulkar, and Bernhard Plattner. Router plugins: A modular and extensible software framework for modern high performance integrated services routers. In *Proceedings of ACM SIGCOMM*, September 1998.

[Dee91] Stephen E. Deering. *Multicast Routing in a Datagram Internetwork*. PhD thesis, Stanford University, December 1991.

[Deu96] L. Peter Deutsch. GZIP file format specification. Internet RFC 1952, May 1996.

[DH98] Stephen Deering and Robert Hinden. Internet protocol, version 6 (IPv6) specification. Internet RFC 2460, 1998.

[DKSZ99] Richard P. Draves, Christopher King, V. Srinivasan, and Brian D. Zill. Constructing optimal IP routing tables. In *Proceedings of IEEE INFOCOM*, 1999.

[DMR+94] Martin Dietzfelbinger, Kurt Mehlhorn, Hans Rohnert, Anna Karlin, Friedhelm Meyer auf der Heide, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal of Computing*, 23(4):748–761, 1994.

[DPW88] Stephen E. Deering, Craig Partridge, and David Waitzman. Distance Vector Multicast Routing Protocol. Internet RFC 1075, 1988.

[DWD+97] Daniel S. Decasper, Marcel Waldvogel, Zubin Dittia, Adiseshu Hari, Guru Parulkar, and Bernhard Plattner. Crossbow — a toolkit for integrated services over cell switched IPv6. In *Proceedings of the IEEE ATM '97 workshop*, Lisboa, Portugal, May 1997.

[Eat99] William N. Eatherton. Hardware-based Internet protocol prefix lookups. Master's thesis, Washington University in St. Louis, St. Louis, MO, USA, May 1999.

[EFH+98] Deborah Estrin, Dino Farinacci, Ahmed Helmy, David Thaler, Stephen Deering, Mark Handley, Van Jacobson, Ching-gung Liu, Puneet Sharma, and Liming Wei. Protocol independent multicast—sparse mode (PIM-SM): Protocol specification. Internet RFC 2362, June 1998.

[EGP98] Tamar Eilam, Cyril Gavoille, and David Peleg. Compact routing schmes with low stretch factor. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 11–20, Puerto Vallarta, México, June–July 1998.

[EKZ77] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[Emd75] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pages 75–84, 1975.

[FKS84] Mchael L. Fredman, Janós Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[FKSS98] Wu chang Feng, Dilip Kandlur, Debanjan Saha, and Kang Shin. Adaptive packet marking for providing differentiated services in the Internet. In *Proceedings of ICNP '98*, October 1998.

[FLYV93] Vince Fuller, Tony Li, Jessica Yu, and Kannan Varadhan. Classless Inter-Domain Routing (CIDR): An address assignment and aggregation strategy. Internet RFC 1519, September 1993.

[Fre60] Edward Fredkin. Trie memory. *Communications of the ACM*, 3:490–499, 1960.

[Fre96] Greg N. Frederickson. Searching intervals and compact routing tables. *Algorithmica*, 15(5):448–466, May 1996.

[Fre97] Greg N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM Journal on Computing*, 26(2):484–538, April 1997.

[FT91] Paul Francis Tsuchiya. A search algorithm for table entries with non-contiguous wildcarding. Known as "Cecilia," available from the Author <francis@aciri.org>, 1991.

[GM99] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *Proceedings of ACM SIGCOMM*, pages 147–160, Cambridge, Massachusetts, USA, September 1999.

[Gra96] Matthew Gray. Internet growth. http://www.mit.edu/people/mkgray/net/internet-growth-summary.html, 1996.

[Gri99] Matthias Gries. A suvey of synchronous ram architectures. TIK Technical Report TIK-57, TIK, ETH Zürich, April 1999. ftp://ftp.tik.ee.ethz.ch/pub/publications/TIK-Report71.ps.gz.

[Gus97] Dan Gusfield. *Algorithms on strings, trees, and sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[Gwe68] Gernot Gwehenberger. Anwendung einer binären Verweisketten-methode beim Aufbau von Listen (Use of a binary tree structure for processing files). *Elektronische Rechenanlagen*, 10:223–226, 1968.

[Har99] Adiseshu Hari. *Bandwidth Management in the Internet*. PhD thesis, Washington University in St. Louis, St. Louis, MO, USA, May 1999.

[HD98] Robert Hinden and Stephen Deering. IP version 6 addressing architecture. Internet RFC 2373, 1998.

[HOD98] Robert Hinden, Mike O'Dell, and Stephen Deering. An IPv6 aggregatable global unicast address format. Internet RFC 2374, 1998.

[IAN] IANA. Internet Assigned Number Authority. http://www.iana.org/.

[Int] Internet Performance and Measurement Project. Internet routing table statistics. http://www.merit.edu/ipma/routing_table/.

[Int97a] Intel Corporation. *Intel Architecture Optimization Manual*, 1997.

[Int97b] Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, 1997.

[Int97c] Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set*, 1997. http://developer.intel.com/design/PentiumII/manuals/.

[IS99] Ayal Itzkovitz and Assaf Schuster. MultiView and Millipage — Fine-grain sharing in page-based DSMs. In *Proceedings of the Third Symposium on Operating Systems Design and Implementations (OSDI '99)*, Operating Systems Review, February 1999.

[IT96] International Telecommunication Union (ITU-T). Interface between data terminal equipment (DTE) and data circuit-terminating equipment (DCE) for terminals operating in the packet mode and connected to public data networks by dedicated circuit. ITU-T Recommendation X.25, October 1996.

[Jai89] Raj Jain. A comparison of hashing schemes for address lookup in computer networks. Technical Report DEC-TR-593, Digital Equipment Corporation Western Research Laboratory, February 1989.

[Kle61] Leonard Kleinrock. Information flow in large communication nets. RLE Quarterly Progress Report, July 1961.

[KMP77] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(1):323–350, 1977.

[KNRW97] Mark van Kreveld, Jürg Nievergelt, Thomas Roos, and Peter Widmayer, editors. *Algorithmic Foundations of Geographic Information Systems*. Number 1340 in Lecture Notes in Computer Science. Springer, 1997.

[Knu98] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 2nd edition, 1998.

[KR00] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison-Wesley, 2000.

[KRT+98] Satish Kumar, Pavlin Radoslavov, David Thaler, Cengiz Alaettinoğlu, Deborah Estrin, and Mark Handley. The MASC/BGMP architecture for inter-domain multicast routing. In *Proceedings of ACM SIGCOMM*, pages 93–104, September 1998.

[Lab96] Craig Labovitz. Routing analysis. http://www.merit.edu/ipma/analysis/routing.html, 1996.

[LGMR99] Paul A. Longley, Michael F. Goodchild, David J. Maguire, and David W. Rhind, editors. *Principles and Technical Issues*, volume 2 of *Geographical Information Systems*. Wiley, 2nd edition, 1999.

[LM97] Steven Lin and Nick McKeown. A simulation study of IP switching. In *Proceedings of ACM SIGCOMM*, pages 15–24, 1997.

[LMJ97] Craig Labovitz, G. Robert Malan, and Farnam Jahanian. Internet routing instability. In *Proceedings of ACM SIGCOMM*, pages 115–126, 1997.

[LS98] T. V. Lakshman and Dimitrios Stiliadis. High speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of ACM SIGCOMM*, pages 203–214, September 1998.

[LSV98] Butler Lampson, V. Srinivasan, and George Varghese. IP lookups using multiway and multicolumn search. In *Proceedings of IEEE INFOCOM*, San Francisco, 1998.

[LT86] Jan van Leeuwen and Richard B. Tan. Routing with compact routing table. In *The Book of L*, pages 259–273. Springer, 1986.

[Mey99] Thomas Meyer. Schnelle Hashingverfahren für Routing im Internet. Semester project, ETH Zürich, February 1999. http://marcel.wanda.ch/Archive/SA-Meyer.pdf (In German).

[MF93] Anthony J. McAuley and Paul Francis. Fast routing table lookup using CAMs. In *Proceedings of INFOCOM*, pages 1382–1391, March–April 1993.

[MFTW95] Anthony J. McAuley, Paul Francis Tsuchiya, and Daniel V. Wilson. Fast multilevel hierarchical routing table using content-addressable memory. U.S. Patent 5,386,413, January 1995.

[MJV96] Steven McCanne, Van Jacobson, and Martin Vetterli. Receiver-driven layered multicast. In *Proceedings of ACM SIGCOMM*, pages 117–130, August 1996.

[MK90] Tracy Mallory and A. Kullberg. Incremental updating of the Internet checksum. Internet RFC 1141, January 1990.

[MN90] Kurt Mehlhorn and Stefan Näher. Bounded ordered dictionaries in $O(\log\log N)$ time and $O(n)$ space. *Information Processing Letters*, 35(4):183–189, 1990.

[Mor68] Donald R. Morrison. PATRICIA—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15:514–534, 1968.

[MSSW94] Cathy May, Ed Silha, Rick Simpson, and Hank Warren, editors. *The PowerPC Architecture*. Morgan Kaufmann, 2nd edition, 1994.

[Nat] National Laboratory for Applied Network Research, Measurement and Operations Analysis Team. NLANR network analysis infrastructure. http://moat.nlanr.net/.

[Nat97] National Laboratory for Applied Network Research. WAN packet size distribution. http://www.nlanr.net/NA/Learn/packetsizes.html, 1997.

[NHS84] Jürg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.

[NK98] Stefan Nilsson and Gunnar Karlsson. Fast address lookup for Internet routers. In P. Kühn and R. Ulrich, editors, *Broadband Communications: The Future of Telecommunications*, pages 11–22, 1998.

[NMH97] Peter Newman, Greg Minshall, and Larry Huston. IP Switching and gigabit routers. *IEEE Communications Magazine*, 35(1):64–69, January 1997.

[Nye95] Adrian Nye, editor. *X Protocol Reference Manual: Volume Zero for X11, Release 6*, volume 0 of *Definitive Guide to X Windows*. O'Reilly and Associates, February 1995.

[Par96] Craig Partridge. Locality and route caches. In *NSF Workshop on Internet Statistics Measurement and Analysis*, San Diego, CA, USA, February 1996. Available at http://www.caida.org/outreach/9602/positions/partridge.html.

[PC+98] Craig Partridge, Philip P. Carvey, et al. A 50-Gb/s IP router. *IEEE/ACM Transactions on Networking*, 6(3):237–248, June 1998.

[PD00] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann, 2nd edition, 2000.

[Per92] Radia Perlman. *Interconnections: Bridges and Routers*. Addison-Wesley, 1992.

[Per99] Radia Perlman. *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols*. Addison-Wesley, 2nd edition, 1999.

[PH96] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.

[Pos81a] Jon Postel, editor. Internet protocol. Internet RFC 791, 1981.

[Pos81b] Jon Postel, editor. Transmission control protocol. Internet RFC 793, September 1981.

[PR83] Jon Postel and Joyce Reynolds, editors. Telnet protocol specification. Internet RFC 854, May 1983.

[PR85] Jon Postel and Joyce Reynolds, editors. File transfer protocol (FTP). Internet RFC 959, October 1985.

[Rata] Rational Software Corporation. Purify. http://www.rational.com/products/purify_unix/.

[Ratb] Rational Software Corporation. Quantify. http://www.rational.com/products/quantify/.

[RDK+97] Yakov Rekhter, Bruce Davie, Dave Katz, Eric Rosen, and George Swallow. Cisco systems' tag switching architecture overview. Internet RFC 2105, February 1997.

[Rij00] Chris Rijk. The UltraSPARC III. http://www.aceshardware.com/Spades/read.php?article_id=115, 2000.

[RL93] Yakov Rekhter and Tony Li. An architecture for IP address allocation with CIDR. Internet RFC 1518, September 1993.

[RL95] Yakov Rekhter and Tony Li. A border gateway protocol 4 (BGP-4). Internet RFC 1771, 1995.

[Rob97] Lawrence G. Roberts. Internet chronology. http://www.ziplink.net/∼lroberts/InternetChronology.html, August 1997.

[RP94] Joyce K. Reynolds and Jon Postel. Assigned numbers. Internet RFC 1700, October 1994.

[RVC01] Eric C. Rosen, Arun Viswanathan, and Ross Callon. Multiprotocol label switching architecture. RFC 3031, Internet Engineering Task Force, January 2001.

[SCW+99] Burkhard Stiller, Christina Class, Marcel Waldvogel, Germano Caronni, and Daniel Bauer. A flexible middleware for multimedia communication: Design, implementation, and experience. *IEEE Journal on Selected Areas in Communications*, 17(9):1580–1598, September 1999.

[Sei91] Raimund Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry Theory Applications*, 1:51–64, 1991.

[Ses96] Roger Sessions. *Object Persistence: Beyond Object-Oriented Databases*. Prentice-Hall, 1996.

[Sit92] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[SK85] Nicola Santoro and Ramez Khatib. Labelling and implicit routing in networks. *The Computer Journal*, 28(1):5–8, 1985.

[Skl93] Keith Sklower. A tree-based packet routing table for Berkeley Unix. Technical report, University of California, Berkeley, 1993. Also at http://www.cs.berkeley.edu/∼sklower/routing.ps.

[Spi95] Barry A. Spinney. Address lookup in packet data communications link, using hashing and content-addressable memory. U.S. Patent Number 5,414,704. Assignee Digital Equipment Corporation, Maynard, MA, May 1995.

[Sri99] Divesh Srivastava. Personal communication, April 1999.

[SSV99] V. Srinivasan, Subhash Suri, and George Varghese. Packet classification using tuple space search. In *Proceedings of ACM SIGCOMM*, pages 135–146, Cambridge, Massachusetts, USA, September 1999.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[SV99a] V. Srinivasan and George Varghese. Fast address lookups using controlled prefix expansion. *Transactions on Computer Systems*, 17(1):1–40, February 1999.

[SV99b] V. Srinivasan and George Varghese. A survey of recent IP lookup schemes. In *Proceedings of the IFIP Sixth International Workshop on Protocols for High Speed Networks (PfHSN '99)*, Salem, MA, USA, August 1999.

[SVSW98] V. Srinivasan, George Varghese, Subhash Suri, and Marcel Waldvogel. Fast and scalable layer four switching. In *Proceedings of ACM SIGCOMM*, pages 191–202, September 1998.

[SWI99] Switch trouble ticket 19990318-2. http://www.switch.ch/cgi-bin/tt/ttview?view=19990318-2, March 1999.

[Tan87] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, 1987.

[TMW97] Kevin Thompson, Gregory J. Miller, and Rick Wilder. Wide-area Internet traffic patterns and characteristics. *IEEE Network*, 11(6):10–23, November–December 1997.

[Top90] Claudio Topolcic, editor. Internet stream protocol version 2 (ST2) protocol specification. Internet RFC 1190, October 1990.

[TVW00] Jonathan Turner, George Varghese, and Marcel Waldvogel. Scalable high speed IP routing lookups. U.S. Patent Number 6,018,524, January 2000.

[WCS+99] Marcel Waldvogel, Germano Caronni, Dan Sun, Nathalie Weiler, and Bernhard Plattner. The VersaKey framework: Versatile group key management. *IEEE Journal on Selected Areas in Communications*, 17(9):1614–1631, September 1999.

[WG94] David L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual, Version 9*. Prentice-Hall, Menlo Park, 1994.

[Wil98] Erik Wilde. *Wilde's WWW: Technical Foundations of the World Wide Web*. Springer, November 1998.

[Wro97] John Wroclawski. The use of RSVP with IETF integrated services. Internet RFC 2210, September 1997.

[WVTP97] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable high speed IP routing table lookups. In *Proceedings of ACM SIGCOMM*, pages 25–36, September 1997.

[WVTP98] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable best matching prefix lookups. In *Proceedings of PODC '98*, page 311, Puerto Vallarta, México, June 1998.

[ZHMB97] Martina Zitterbart, T. Harbaum, D. Meier, and D. Brökelmann. HeaRT: high performance routing table look up. In *Proceedings of IEEE HPCS '97*, 1997.