Addis Ababa University

Addis Ababa institute of Technology

Department of Electrical and Computer Engineering

SMALL FORWARDING TABLES FOR FAST ROUTING LOOKUPS

A thesis submitted to the Addis Ababa Institute of Technology, School of Graduate Studies, Addis Ababa University

In partial fulfillment of the requirements for the degree of MASTER OF SCIENCE IN ELECTRICAL ENGINEERING (COMPUTER ENGINEERING)

By

Musa Hayato

Addis Ababa University

Addis Ababa institute of Technology

Department of Electrical and Computer Engineering

SMALL FORWARDING TABLES FOR FAST ROUTING LOOKUPS

A thesis submitted to the Addis Ababa Institute of Technology, School of Graduate Studies, Addis Ababa University

In partial fulfillment of the requirements for the degree of MASTER OF SCIENCE IN ELECTRICAL ENGINEERING (COMPUTER ENGINEERING)

By

Musa Hayato

Advisor

Dr. Getachew Hailu

Addis Ababa University

Addis Ababa institute of Technology

Department of Electrical and Computer Engineering

SMALL FORWARDING TABLES FOR FAST ROUTING LOOKUPS

By

Musa Hayato

APPROVED BY BOARD OF EXAMINERS

Chairman, Department of Graduate Study	**Signature**
Dr. Getachew Hailu	
Advisor	**Signature**
Internal Examiner	**Signature**
External Examiner	**Signature**

Acknowledgements

I would like to express my deep and sincere gratitude to my Advisor Dr. Getachew Hailu for his excellent supervision and helps during this work. I would like to express gratitude to my family who are always beside me on financial support and inspiration to pursue the study.

My gratitude also goes to my colleagues at ECE Department for their collaboration.

Contents

List of Figures

List of Tables

#

List of Acronyms

API	Application Program Interface
AS	Autonomous System
ATM	Asynchronous Transfer Mode
BBN	Backbone Network
BGP	Border Gateway Protocol
CAM	Content Addressable Memory
CIDR	Classless Inter Domain Routing
CPE	Controlled Prefix Expansion
DRAM	Dynamic Random Access Memory
DWDM	Dense Wavelength Division Multiplexing
DP-trie	Dynamic Prefix trie
FDDI	Fiber Distributed Data Interface
ICMP	Internet Control Message Protocol
IDE	Integrated Development Environment
IP	Internet Protocol
IPMA	Internet Performance Measurement and Analysis
ISP	Internet Service Provider
LC-Tries	Level Compressed Trie
LPM	Longest Prefix Matching
MPLS	Multiple Protocol Label Switching
NHA	Next Hop Array

OSPF	Open Shortest Path First
PATRICIA	Practical Algorithm to Retrieve Information Coded In Alphanumeric
PIM	Protocol Independent Multicast
QOS	Quality of Service
RAM	Random Access Memory
RFC	Request for Comment
RIP	Routing Information Protocol
SRAM	Synchronous Random Access Memory
TCAM	Tertiary Content Addressable Memory
TCP/IP	Transmission Control Protocol/Internet Protocol
TTL	Time to Live

Abstract

IP address lookup is an important design issue for the high performance Internet routers. Due to the rapid growth of traffic in the Internet, backbone links of several gigabits per second are commonly deployed. To handle gigabit-per-second traffic rates, the backbone routers must be able to forward millions of packets per second on each of their ports. Fast IP address lookup in the routers, which uses the packet's destination address to determine for each packet the next hop, is therefore crucial to achieve the packet forwarding rates required. In the last few years, various algorithms for high-performance IP address lookup have been proposed. The algorithm that is under scrutiny in this project is based on matching longest IP prefix. This algorithm incorporates a technique of complete trie in building prefix tries. This mechanism enables to build small forwarding tables.

In this project we will present a forwarding table data structure designed for quick routing lookups. The Forwarding tables are small enough to fit in the cache of a conventional general purpose processor. This means that it is feasible to do a full routing lookup for each IP packet at gigabit speeds without using special hardware.

Keywords: Address lookup, Complete Trie, Data Structure

Chapter One

1. Introduction

Internet traffic is increasing at an unimaginable rate, partly because of increased users, but also because of new multimedia applications. The higher-bandwidth need requires faster communication links and faster network routers. Gigabit fiber links are commonplace, and yet the fundamental limits of optical transmission have hardly been approached. Thus the key to improve Internet performance is use faster routers.

The necessity to furnish the Internet with high-speed backbone gives a strong motivation to develop fast routers matching with the sum of their input optical links' speed. One research area for fast routers focuses on speeding up forwarding lookup operations, which is one of main causes for routers' performance bottleneck. Since routing lookups named as LPM (Longest Prefix Matching) problem need to scan the forwarding table to locate an entry with the longest prefix matching the destination IP address of each incoming packets, they are known to hardly accomplish complexity like the exact matching problem. The time complexity of conventional forwarding lookup algorithms tends to grow in proportion to either the routing table size or IP address size [1].

The existing research for hastening lookups can be categorized into three categories: hardware-based, software-based, and protocol-based techniques. Hardware-based solutions try to expedite routing lookup operations by employing special hardware such as high speed SRAM and CAM [2, 3, 4]. Protocol-based schemes such as MPLS (Multiple Protocol Label Switching) try to avoid the complicated IP routing lookups by assigning a small index called *label* to a connection at the edge router, which later will be appended at every incoming packets from that connection.

In contrast to the above two approaches, software-based methods accelerate forwarding lookups simply by new noble forwarding table structures and their accompanying algorithms. Recently many forwarding structures for shortening lookup time up to a few hundred nanoseconds per packet, which is fast enough for catching up with Gigabit links, have been proposed. For speedup, this approach either compacts the forwarding table small enough to sit on cache or transforms the longest partial match problem to the exact match problem. The approach focuses on limiting the number of memory accesses since the memory access time is usually 10 times slower than the execution time of one instruction in modern processors [1].

1.1. Objectives

1.1.1. General Objectives

The general objective of this thesis is to study software based routing table lookup process and to implement Small Forwarding Table algorithms for lookup process

1.1.2. Specific Objectives

- To fully understand software based routing lookup process
- To implement an algorithm for fast routing table lookup
- To compare different approach for routing table lookup

1.2. Methodology

1.2.1. Literature survey

Extensive literature survey was made to fully understand and master the concepts related to routing. This includes thorough study of main component included under routing process such as routing table construction, routing table update and routing table lookups. Besides, the research of algorithms so far developed to facilitate routing lookups was made.

1.2.2. Analysis and discussions

In order to analyze/investigate the effectiveness of the algorithm two types of data are inherently available to work with. Firstly, the actual data that can be collected from different web sites that provides such services as Internet Performance Measurement and Analysis (IPMA). Secondly, artificially generated data was used.

1.3. Theses outline

This report was organized into five chapters and an appendix. Chapter one gives some highlights on the whole thesis work: - problem description, defining the objective and laying down methodology used to achieve the objectives.

Following the introductory chapter is the theoretical background of IP Address Look ups:- concepts and components. In this chapter, the basic concepts of IP Addressing schemes, Path Determination, IP address lookups and design requirements are discussed briefly.

In the third chapter of this thesis, special emphasis is given to the Materials and Methodology. In this chapter, a detailed discussion of the literature survey and methodology are presented. Most of the basic and main components of IP Address Lookups and the attempt so far made to solve the problems of address lookup are discussed. The final section of this chapter is a discussion of specific Lookup algorithms ,namely Small Forwarding Table, and materials used are elaborated.

Following the discussion of Methodologies and materials is the practical aspect of this thesis. In this “Simulation and Result” chapter, simulation environment and procedures followed for simulating Small Forwarding Table are discussed. Finally the results of simulation are presented and discussed.

Finally chapter five presents some conclusions remarks and limitations of this thesis work. In the end, appendices and bibliography are attached. With the appendices, detailed documentation of Small Forwarding Table deployment is presented.

Chapter Two

2. Background of the problem

2.1. Addressing and Internet Service

The Internet comprises a mesh of routers interconnected by links, in which routers forward packets to their destinations, and physical links transport packets from one router to another. Because of the scalable and distributed nature of the Internet, there are more and more users connected to it and more and more intensive applications over it. The great success of the Internet thus leads to exponential increases in traffic volumes, stimulating an unprecedented demand for the capacity of the core network. The trend of such exponential growth is not expected to slow down, mainly because data-centric businesses and consumer networking applications continue to drive global demand for broadband access solutions. This means that packets have to be transmitted and forwarded at higher and higher rates. Advances in fiber throughput and optical transmission technologies have enabled operators to deploy capacity in a dramatic fashion. For instance, dense wavelength division multiplexing (DWDM) equipment can multiplex the signals of 300 channels of 11.6 Gbit/s to achieve a total capacity of more than 3.3 Tbit/s on a single fiber and transmit them over 7000 km. In the future, DWDM networks will widely support 40 Gbit/s (OC-768) for each channel, and link capacities are keeping pace with the demand for bandwidth [5].

Historically, network traffic doubled every year, and the speed of optical transmissions (such as DWDM) every seven months. However, the capacity of routers has doubled every 18 months lagging behind network traffic and the increasing speed of optical transmission. Therefore, the router becomes the bottleneck of the Internet [6].

2.1.1. Concept of Routers

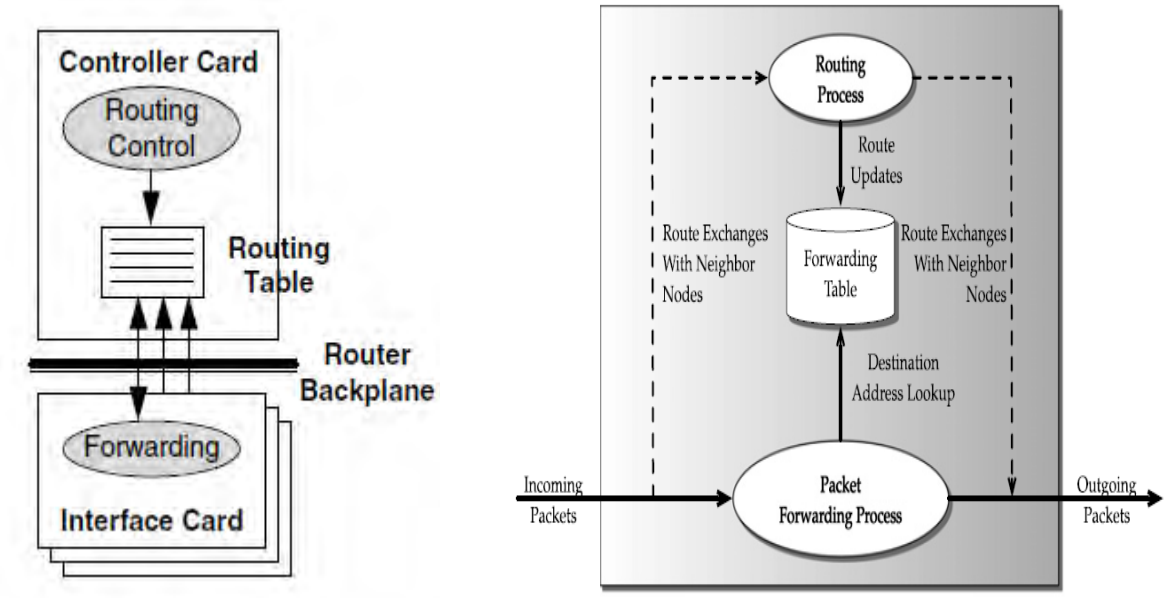
The Internet can be described as a collection of networks interconnected by routers using a set of communication standards known as the Transmission Control Protocol/Internet Protocol

(TCP/IP) suite. TCP/IP is a layered model with logical levels: the application layer, the transport layer, the network layer, and the data link layer. Each layer provides a set of services that can be used by the layer above. The network layer provides the services needed for Internetworking, that is, the transfer of data from one network to another. Routers operate at the network layer, and are sometimes called IP routers. Routers tie together the constituent networks of the global Internet, creating the illusion of a unified whole. In the Internet, a router generally connects with a set of input links through which a packet can come in and a set of output links through which a packet can be sent out.

Each packet contains a destination IP address; the packet has to follow a path through the Internet to its destination. Once a router receives a packet at an input link, it must determine the appropriate output link by looking at the destination address of the packet. The packet is transferred router by router so that it eventually ends up at its destination. Therefore, the primary functionality of the router is to transfer packets from a set of input links to a set of output links [5].

2.1.2. Functions of IP Routers

Broadly speaking, a router must perform two fundamental tasks: routing and packet forwarding, as shown in Figure 2.1(b). Based on the information exchanged between neighboring routers using routing protocols, the routing process constructs a view of the network topology and computes the best paths. The network topology reflects network destinations that can be reached as identified through IP prefix-based network address blocks. The best paths are stored in a data structure called a forwarding table. The packet forwarding process moves a packet from an input interface (“ingress”) of a router to the appropriate output interface (“egress”) based on the information contained in the forwarding table. Since each packet arriving at the router needs to be forwarded, the performance of the forwarding process determines the overall performance of the router [7].



(a)[5] (b)[11]

Figure 2.1: Basic Architecture of Router (a)Basic architecture (b)functional view

A generic architecture of an IP router is given in Figure 2.1(a). Figure 2.1(a) shows the basic architecture of a typical router: the controller card [which holds the central processing unit (CPU)], the router backplane, and interface cards. The CPU in the router typically performs such functions as path computations, routing table maintenance, and reachability propagation. It runs whichever routing protocols are needed in the router. The interface cards consist of adapters that perform inbound and outbound packet forwarding (and may even cache routing table entries or have extensive packet processing capabilities). The router backplane is responsible for transferring packets between the cards. The basic functionalities in an IP router can be categorized as: route processing, packet forwarding, and router special services.

2.1.2.1. Path determination

It is based on a variety of criteria to determine the optimal route where there are a number of available path to the destination. Software implementations of routing algorithms calculate the

appropriate criteria values for each available (known) path to determine the optimal route(s) to a destination. To aid the process of path determination, routing algorithms initialize and maintain routing tables which contain route information. Route information varies depending on the routing algorithm used. Path determination may require a quite complex process. The path determination function may consist of one or more routing protocols that provide exchange of routing information among routers, as well as the procedures (algorithms) that a router uses to convert this information into a routing table. In the case of IP routing, OSPF, BGP, and PIM are examples of such routing protocols [8].

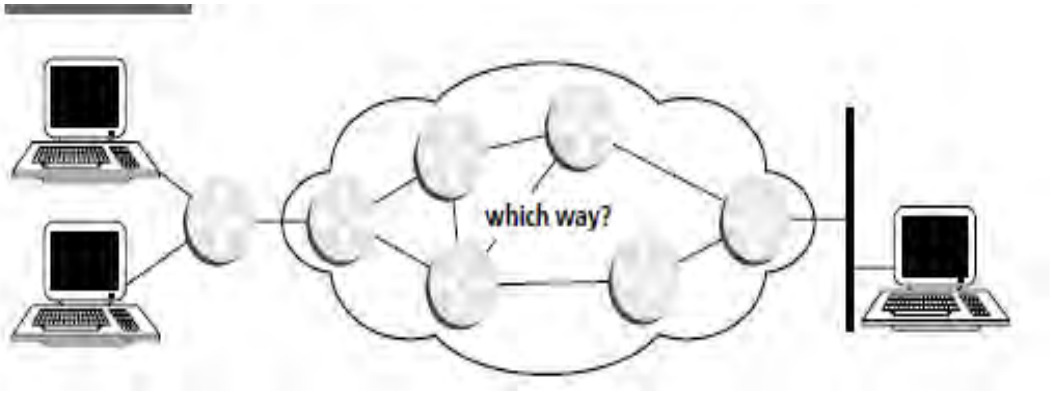


Figure 2.2: Path Determination [8]

2.1.2.2. Packet Forwarding

A router receives an IP packet on one of its interfaces and then forwards the packet out of another of its interfaces (or possibly more than one, if the packet is a multicast packet), based on the contents of the IP header. As the packet is forwarded hop by hop, the packet's (original) network layer header (IP header) remains relatively unchanged, containing the complete set of instructions on how to forward the packet (IP tunneling may call for prepending the packet with other IP headers in the network). However, the data-link headers and physical-transmission schemes may change radically at each hop to match the changing media types.

If the router supports IP routing, the IP header of a datagram is checked for protocol version, header length (must be greater than minimum IP header size, i.e. five 32-bit words), length (must

not be smaller than the header length), and header checksum. The IP datagram length is checked against the size of the received Ethernet frame to determine whether a complete datagram has been received. The header checksum computation serves to verify all the header fields against corruption. In the case of failure of at least one of these initial IP datagram checks, the router sees the datagram as malformed and discards it even without sending out an error notification to the datagram originator. If the above checking was successful, as the next step, the router checks the value of the time-to-live (TTL) field. The TTL field has two functions: to limit the lifetime of IP datagrams and to terminate Internet routing loops. Although TTL initially indicated time in seconds, it now has the attribute of a hop-count, since each router is required to reduce the TTL field by one. TTL expiration is intended to cause datagrams to be discarded by routers, but not by the destination host. Hosts that act as routers by forwarding datagrams must therefore follow the router's rules for TTL. The TTL value should be greater than one because the router has to decrement it by one before further forwarding the datagram to the next hop. Any datagram with zero value of TTL must be discarded (unless addressed to the particular router) and the ICMP *TTL Exceeded* message generated. TTL is checked first but not decremented before actually checking the destination IP address. Note in particular that a router must not modify the TTL of a packet except when forwarding it. A router must not originate or forward a datagram with a TTL value of zero (RFC 1812) [8].

In packet forwarding, the destination IP address is used as a key for the routing table lookup. The best-matching routing table entry is returned, indicating whether to forward the packet and, if so, the interface to forward the packet out of and the IP address of the next IP router (if any) in the packet's path. The next-hop IP address is used at the output interface to determine the link address of the packet, in case the link is shared by multiple parties [such as an Ethernet, Token Ring, or Fiber Distributed Data Interface (FDDI) network], and is consequently not needed if the output connects to a point-to-point link.

In addition to making forwarding decisions, the forwarding process is responsible for making packet classifications for quality of service (QOS) control and access filtering [5].

2.2. Addressing Scheme

2.2.1. Classful Addressing Scheme

IP addresses are used by the IP to identify distinct devices, such as hosts, routers, and gateways, as well as to route data to those devices. Each device in an IP network must be assigned to a unique IP address so that it can receive communications addressed to it. This addressing, known as IPv4 addressing, is written in the bit format, from left to right, where the left-most bit is considered the most significant bit. The hierarchy in IP addressing, similar to the postal code and the street address, is reflected through two parts, a network part and a host part referred as the pair (netid, hostid). Thus, we can think of the Internet as the interconnection of networks identified through netid where each netid has a collection of hosts. The network part (netid) identifies the network to which the host is attached, and the host part (hostid) identifies a host on that network. The network part is also referred as the IP prefix. All hosts attached to the same network share the network part of their IP addresses but must have a unique host part. To support different sizes for the (netid, hostid) part, a good rule on how to partition the total IP address space of 2^{32} addresses was needed. Thus, the IP address space was originally divided into three different classes, Class A, Class B, and Class C, as shown in Figure 2.3 for networks and hosts. Each class was distinguished by the first few initial bits of a 32-bit address. For readability, IP addresses are expressed as four decimal numbers, with a dot between them. This format is called the dotted decimal notation. The notation divides the 32-bit IP address into 4 groups of 8 bits and specifies the value of each group independently as a decimal number separated by dots. Because of 8-bit breakpoints, there can be at most 256 (= 2^8) decimal values in each part. Since 0 is an assignable value, no decimal values can be more than 255. Thus, an example of an IP address is 10.5.21.90 consisting of the four decimal values, separated by a dot or period [9].

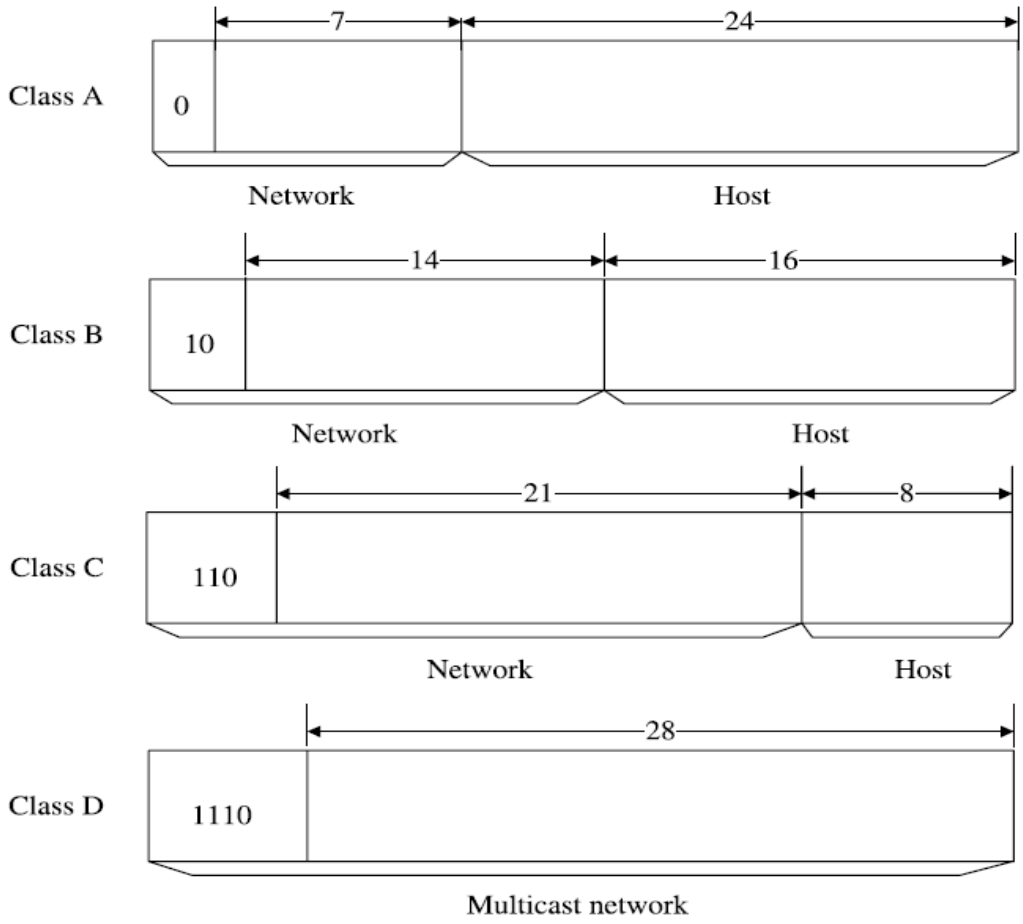


Figure 2.3: Classful IP Addressing [9]

Each Class A address has the first bit set to 0 and is followed by 7 bits for the network part, resulting in a maximum of 128 (= 2^7) networks; this is then followed by a 24-bit host part. Thus, Class A supports a maximum of $2^{24} - 2$ hosts per network. This calculation subtracts 2 because 0s and 1s in the host part of a Class A address may not be assigned to individual hosts; rather, all 0s that follows a netid such as 10.0.0.0 identify the network, while all 1s that follow a netid such as 10.255.255.255 are used as the broadcast address for this network. Each Class B network address has the first two bits set to "10," followed by a 14-bit network part, which is then

followed by a 16-bit host part. A maximum of 2^{14} networks can be defined with up to $2^{16} - 2$ hosts per network.

Finally, a Class C network address has the first three bits set as "110" and followed by a 21-bit network part, with the last 8 bits to identify the host part. Class C provides support for a maximum of 2^{21}*(*= 2,097,152*)* networks with up to 254 ($2^8 - 2$) hosts. In each class, a set of network addresses is reserved for a variety of purposes;

Three address classes discussed so far are used for unicasting in the Internet, that is, for a host-to-host communication. There is another class of IP addresses, known as Class D addressing, that is used for *multicasting* in the Internet; in this case, the first four bits of the 32-bit address are set to "1110" to indicate that it is a multicast address. A host can use a multicast address as the destination address for a packet generated to indicate that the packet is meant for any hosts on the Internet; in order for any hosts to avail this feature, they must use another mechanism to tune into this address.

The original rationale behind classes of different sizes was to provide the flexibility to support different sized networks, with each network containing a different number of hosts. Thus, the total address length can still be kept fixed at 32 bits, an advantage from the point of view of efficient address processing at a router or a host. As the popularity of the Internet grew, several disadvantages of the addressing scheme came to light. The major concerns were the rate at which the IP address blocks that identify netids were being exhausted, especially when it was necessary to start assigning Class C level netids. IP netids are non geographic; thus, all valid netids are required to be listed at the core routers of the Internet along with the outgoing link, so that packets can be forwarded properly. If we now imagine all Class C level netids being assigned, then there are over 2 million entries that would need to be listed at a core router; no current routers can handle this number of entries without severely slowing packet processing. This led to the development of the concept of *classless* addressing [9].

2.2.2. Subnetting/Netmask

Consider the IP address 192.168.40.3 that is part of Class C network 192.168.40.0. A subnet or sub-network is defined through a network mask boundary using the specified number of significant bits as 1s. Since Class C defines networks with a 24-bit boundary, we can then consider that the most significant 24 bits are 1s, and the lower 8 bits are 0s. This translates to the dotted decimal notation 255.255.255.0, which is also compactly written as "/24" to indicate how many most significant bits are 1s. We can then do a bit-wise logical "AND" operation between the host address and the net mask to obtain the Class C network address as shown below:

```
        11000000 10101000 00101000 00000011 → 192.168.40.3
AND     11111111 11111111 11111111 00000000 → netmask (/24)        (1)
        11000000 10101000 00101000 00000000 → 192.168.40.0
```

Note that for network addresses such as Class C address, the net mask is implicit and it is on a /24 subnet boundary. Now consider that we want to change the net mask *explicitly* to /21 to identify a network larger than a 24-bit subnet boundary. If we now do the bit-wise operation

```
        11000000 10101000 00101000 00000011 → 192.168.40.3
AND     11111111 11111111 11111000 00000000 → netmask (/21)        (2)
        11000000 10101000 00101000 00000000 → 192.168.40.0
```

we note that the network address is again 192.168.40.0. However, in the latter case, the network boundary is 21 bits. Thus, to be able to clearly distinguish between the first and the second one, it is necessary to explicitly mention the netmask. This is commonly written for the second example as 192.168.40.0/21, where the first part is the netid and the second part is the mask boundary indicator. In this notation, we could write the original Class C address as 192.168.40.0/24 and thus, there is no ambiguity with 192.168.40.0/21 [9].

2.2.3. Classless Inter Domain Routing

Classless Inter Domain Routing (CIDR) uses an explicit netmask with an IPv4 address block to identify a network, such as 192.168.40.0/21. An advantage of explicit masking is that an address block can be assigned at any bit boundaries, be it /15 or /20; most important, the assignment of Class C level addresses for networks that can show up in the global routing table can be avoided or minimized.

Through such a process, and because of address block assignment at boundaries such as /21, the routing table growth at core routers can be delayed. In the above example, only the netid 192.168.40.0/21 needs to be listed in the routing table entry, instead of listing *eight* entries from 192.168.40.0/24 to 192.168.47.0/24. Thus, you can see how the routing table growth can be curtailed.

2.3. IP Address Lookups

One of the fundamental objectives of the IP is to interconnect networks, so it is a natural choice to route packets based on a network, rather than on a host. Thus, each IPv4 address is partitioned into two parts: netid and hostid, where netid identifies a network and hostid identifies a host on that network; this is called a two-level hierarchy. All hosts on the same network have the same netid but different hostids.

The IP suite of protocols has some routing protocols such as routing information protocol (RIP), open shortest path first (OSPF), or border gateway protocol (BGP). The routers perform one or more routing protocols to exchange protocol messages with neighboring routers. The protocol messages mainly contain a representation of the network topology state information and store the current information about the best-known paths to destination networks, called the reachability information. The reachability information is stored in a router as a table called the Routing Table. In the BGP routing table, an entry often has parameters: Network (Prefix), Nexthop, AS path, Metric, LocPrf, and Weight etc.

The prefix is a unique identifier of a neighboring network: each prefix must have one or more next hops that can forward the packets to the neighboring network. If a prefix has only one next hop, then other parameters would not affect the BGP decision process. If a network has more than one next hop, then it is dependent on the other parameters to select one of next hops. The AS path is a list of autonomous systems (ASs) that packets must transit to reach the network (Prefix). The metric is usually multi-exit discriminator in Cisco routers. Setting the local preference (LocPrf) also affects the BGP decision process. If multiple paths are available for the same prefix, then the AS path with the larger LocPrf is preferred. LocPrf is at the highest level of the BGP decision process (comes after the Cisco proprietary weight parameter); it is considered before the AS path length. A longer path with a higher LocPrf is preferred over a shorter path with a lower LocPrf [5].

The router typically maintains a version of the routing table in all line cards that is directly used for packet forwarding, called the forwarding table. The forwarding table is made from the routing table by the BGP decision process. In the forwarding table, each entry may only keep a prefix, a next hop, and some statistics. In mathematics, the forwarding table is a function *F:*

$$F: \{P1, P2, P3. . . Pn\} \rightarrow \{h1, h2, h3, . . ., hm\} \quad \textbf{(3)}$$

where *Pi* is a prefix, *n* is the number of prefixes in a forwarding table, *h* is the next hop, and *m* is the number of next hops on a router [5].

2.3.1. Concept of IP-Address Lookup

As we have seen, using address prefixes is a simple method to represent groups of contiguous IP addresses, that is to say, a prefix is a set of contiguous IP addresses.

Proposition 2.3.1.1: For an IP address $D = s1s2s3 . . . s32$, (si = 0, 1) and a prefix $P = t1t2t3 . . . tl*$ (ti = 0, 1).

$D \in P$, if and only if $si = ti$, $i = 1, 2 \ldots l$.

Proof: If $D \in P$, then $P = t1t2\ t3 \ldots\ tl\underbrace{000000000}_{32-l} \leq D \leq P = t1\ t2\ t3 \ldots tl\underbrace{1111111}_{32-l}$

$\therefore sk = tk \quad 1 \leq k \leq l$

If $si = ti$, $i = 1, 2, \ldots, l$, $\because 0 \leq si \leq 1$, $i = l + 1, l + 1, \ldots, 32$,

$\therefore P1 = t1t2\ t3 \ldots\ tl\underbrace{000000000}_{32-l} \leq D \leq P2 = t1\ t2\ t3 \ldots tl\underbrace{1111111}_{32-l}\ D \in P$

Definition 2.3.1.1: For an IP address D and a prefix P, if $D \in P$, we say that D matches P.

For an IP packet with destination address D, the router will find a next hop to forward the packet to by looking up in the forwarding table the prefix P that matches D. That is to say, for a given IP address D, we must find the next hop $hi = F(Pk)$, in which Pk is subject to $D \in Pk$. The main problem is to give the algorithm to find the appropriate prefix Pk, in which Pk is subject to $D \in Pk$, called **IP-address Lookup** [5].

Proposition 2.3.1.2: For any two prefixes with different length $P1$ and $P2$, if there is an IP address D such that $D \in P1$ and $D \in P2$, then $D \in P2 \subset P1$ ($D \in P1 \subset P2$), or $P1 \cap P2 = \emptyset$

Proof: Let there be two IP address prefixes: $P1 = t1\ t2\ t3 \ldots tm$, ($ti = 0$ or 1), $P2 = s1\ s2\ s3 \ldots sn$, ($si = 0$ or 1). Suppose $m < n$.

$D = d1\ d2\ d3 \ldots d32 \in P1$, then $di = ti$ $(1 \leq i \leq m)$.

$D = d1\ d2\ d3 \ldots d32 \in P2$, then $dj = sj$ $(1 \leq j \leq n)$.

$\therefore ti = si$ $(1 \leq i \leq m)$

$\forall D1 = s1\ s2\ s3 \ldots smsm + 1 \ldots snxn + 1\ xn + 2 \ldots x32 \in P2$

$D1 = s1\ s2\ s3 \ldots smsm + 1 \ldots sn\ xn + 1\ xn + 2 \ldots x32 = t1\ t2\ t3 \ldots tmsm + 1 \ldots snxn + 1\ xn + 2 \ldots x32 \in P1$.

$P2$ is a subset of $P1$, $D \in P2 \subset P1$.

If $m > n$, then $D \in P1 \subset P2$.

If $P1 \cap P2 \neq \emptyset$, for example $D' \in P1 \cap P2$, then $D' \in P1$, and $D' \in P2$.

$\therefore$ If there is no IP address D such that $D \in P1$ and $D \in P2$, then $P1 \cap P2 = \emptyset$

Proposition 2.3.1.3: For any two prefixes $P1$ and $P2$ with same length if there is an IP address D such that $D \in P1$ and $D \in P2$, then $P1 = P2$, or $P1 \cap P2 = \emptyset$.

2.3.2. Matching Techniques

Matching is most often associated with algorithms that search for a single data item in a large string of constant data. There are many existing matching techniques such as exact matching, substring matching, wildcard matching, pattern matching, range matching, and prefix matching [5].

In the exact matching problem, the result is identical with the input item. A large variety of techniques is known for exact matching in different data structures, such as linear search, binary search, hashing, etc. For example, hashing is a very prominent candidate among the exact matching group, because—on average—it can provide for $O(1)$ access in $O(1)$ memory per input item. Probably the most prominent representative is known as perfect hashing, providing for $O(1)$ worst-case access and memory. Unfortunately, finding a good hash function is difficult, and building and updating time of the data structure can take non polynomial time and lead to expensive hash functions [5].

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 is found. Its complexity is much more than that of exact matching.

Substring matching is a variation of exact matching, in which the database does not consist of several independent entries, but of a single large sequence of symbols, a string, and the search item has to be compared with every possible substring.

Pattern matching is in wide use for matching wild-carded substrings in a large string. The best-known solutions in this field are variations on the Knuth–Morris–Pratt and Boyer–Moore algorithm [5].

Range matching is used to check whether a key belongs to a range. Many applications encounter range-matching problems. For example, in a distributed-storage networking scenario where many hosts might access shared data at very high speeds, the hosts protect their access by locking the accessed address range. Hosts must check whether the range is locked before beginning a memory access. There are two types of range searches: the point intersection problem that determines whether a set of ranges contains a query, and the range intersection problem that determines whether a query range intersects with any range in a set of ranges.

Prefix matching is a special case of range matching if a prefix does not overlap with any other prefixes. Prefixes have a limited range: the size of the range is a power of two and its start, and therefore also its end, is a multiple of its size. The problem to determine whether an IP-address matches with a prefix is the point intersection problem, which is one of exact matching. As soon as we introduce overlaps into the prefix database, we require a mechanism to differentiate between the overlapping prefixes. This implies that when allowing for overlaps, the search algorithms need to be more elaborate. If we assign a higher priority to the longer prefix, the longest prefix can be found. In this case, the prefix matching problem is called the Longest Prefix Matching (LPM) problem.

2.3.3. Design Criteria and Performance Requirement

There are a number of properties that are desirable in all IP lookup algorithms [10]:

- *High Speed*. Increasing data rates of physical links require faster address lookups at routers. For example, links running at OC192c (approximately 10 Gbps) rates need the router to process 31.25 million packets per second (assuming minimum-sized 40 byte TCP or IP packets). We generally require algorithms to perform well in the worst case. If

this were not the case, all packets (regardless of the flow they belong to) would need to be queued. It is hard to control the delay of a packet through the router. At the same time, a lookup algorithm that performs well in the average case may be acceptable, in fact desirable, because the average lookup performance can be much higher than the worst-case performance. For OC192c links (10.0 Gbps), the lookup algorithm needs to process packets at the rate of 3.53 million packets per second, assuming an average Internet packet size of approximately 354 bytes. Table 2.1 lists the lookup performance required in one router port.

- *Average-case search time*—Because IP routers are statistical devices anyway, that is, are equipped with buffers to accommodate traffic fluctuations, average-case speed seems like an adequate measure. It is hard to come up with reliable average-case scenarios, because they depend heavily on the traffic model and traffic distribution. But the traffic distribution is unknown. The average case search time is given with the extreme scenario: traffic per prefix is constant.
- *Worst-case search time*—Unfortunately, it is not known how prefix and traffic distributions will evolve in the future and 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.
- *Low Storage*. Small storage requirements enable the use of fast but expensive memory technologies like synchronous random access memories (SRAMs). A memory-efficient algorithm can benefit from an on-chip cache if implemented in software, and from an on-chip SRAM if implemented in hardware.
- *Low Preprocessing Time*. Preprocessing time is the time taken by an algorithm to compute the initial data structure. An algorithm that supports incremental updates of its data structure is said to be “dynamic.” A “static” algorithm requires the whole data

structure to be recomputed each time an entry is added or deleted. In general, dynamic algorithms can tolerate larger preprocessing times than static algorithms.

- *Low Update Time*. Forwarding tables have been found to change fairly frequently. As links go down and come back up in various parts of the Internet, routing protocol messages may cause the table to change continuously. Changes include addition and deletion of prefixes, and the modification of next-hop information for existing prefixes. These changes often occur at the peak rate of a few hundred prefixes per second and at the average rate of more than a few prefixes per second. A lookup algorithm should be able to update the data structure at least this fast.
- *Flexibility in Implementation*. The forwarding engine may be implemented either in software or in hardware depending upon the system requirements. Thus, a lookup algorithm should be suitable for implementation in both hardware and software. For the Line Card at higher speeds (e.g., for OC192c), it is necessary to implement the algorithm in hardware. In fact, the lookup algorithms are implemented easier in software than in hardware. The algorithm based on hardware should use a simple data structure, because of the difficulty in managing memory, and can be divided into few operations for the wire-speed.
- *Scalability to IPv6*. IPv6 is the IP of the next generation. Due to its longer IP address, 128 bits instead of 32-bit, IPv6 requires especially suited efficient mechanisms for IP-address lookup. The run-time complexity analysis and lookup time measurement on IPv4 routing tables are necessary for a performance analysis of the algorithms.

Table 2.1 Requirement of IP Lookup Speed for Line Cards and Packet Sizes

Line Type in Router	*Line Rate (Gbps)*	*40-Byte Packets (Mpps)*	*84-Byte Packets (Mpps)*	*354-Byte Packets (Mpps)*
T1	0.0015	0.0468	0.0022	0.00053
OC3c	0.155	0.48	0.23	0.054
OC12c	0.622	1.94	0.92	0.22
OC48c	2.50	7.81	3.72	0.88
OC192c	10.0	31.2	14.9	3.53
OC768c	40.0	125.0	59.5	14.1

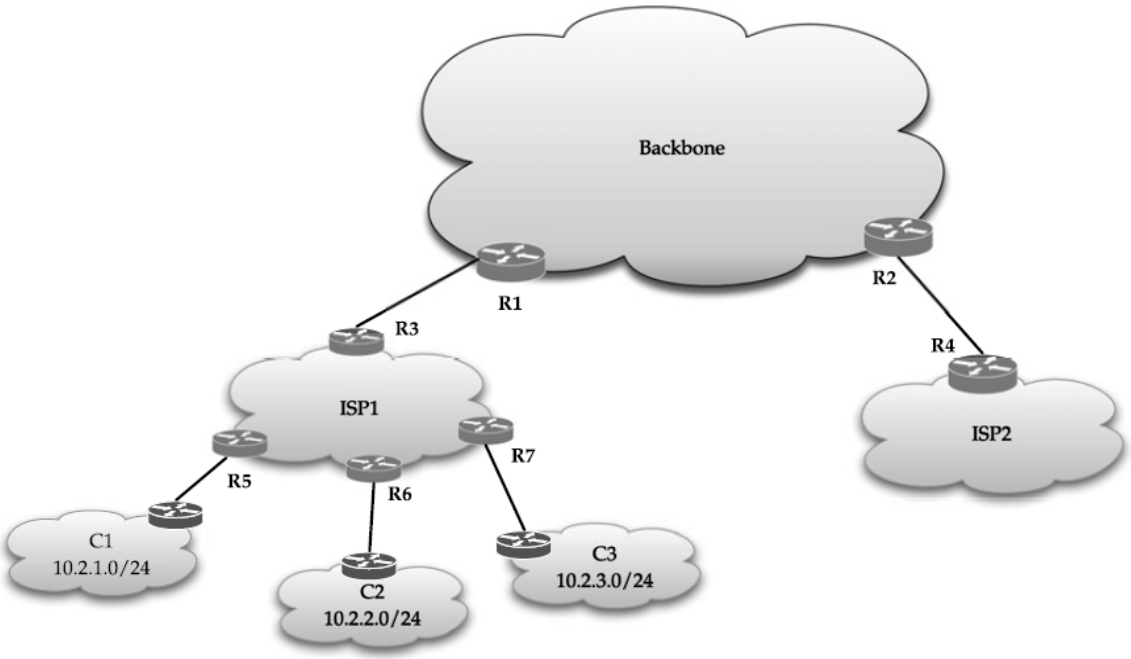
2.3.4. Address aggregation and difficulty of the longest-prefix matching problem

The problem of identifying the forwarding entry containing the longest prefix among all the prefixes matching the destination address of an incoming packet is defined as the longest matching prefix problem. This longest prefix is called the *longest matching prefix*. It is also referred to as the *best matching prefix*.

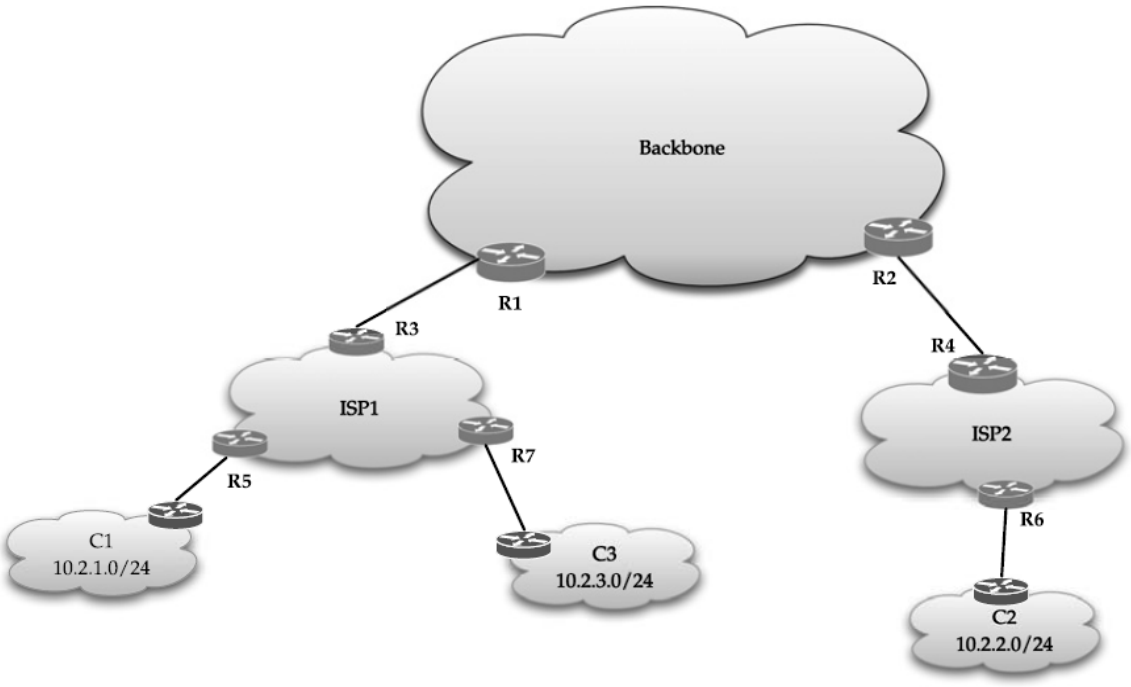
First, let us consider address aggregation. Assume that ISP1, a service provider, connects three customers—C1, C2, and C3—with the rest of the Internet; see Figure 2.4(a). ISP1 is, in turn, connected to some backbone provider through router R1. The backbone can also connect other service providers like ISP2. Assume that ISP1 owns IP prefix block 10.2.0.0/22 and partitions it among its customers. Let us say that prefix 10.2.1.0/24 has been allocated to C1, 10.2.2.0/24 to C2, and 10.2.3.0/24 to C3. Now the router in the backbone R1 needs to keep only a single forwarding table entry for IP prefix 10.2.0.0/22 that directs the traffic bound to C1, C2, and C3 through router R3. As you can see, the hierarchical allocation of prefixes obviates the need for separate routing table entries for C1, C2, and C3 at router R1. In other words, the backbone

routes the traffic bound for ISP1 to R3 and it is the responsibility of the routers within ISP1 to distinguish the traffic between C1, C2, and C3. Next, assume that customer C2 would like to change its service provider from ISP1 to ISP2, but does not want to renumber its network. This is depicted in Figure 2.4(b). Now all the traffic in the backbone bound to prefix 10.2.0.0/22 would need to be directed to ISP1 except for prefix 10.2.2.0/24. We cannot perform aggregation as before at router R1 since there is a "hole" in the address space. This is called *exception to address aggregation or discontinuous networks*.

The difficulty of the longest prefix matching arises because of the following reasons. First, a destination IP address does not explicitly carry the netmask information when a packet traverses through routers. Second, the prefixes in the forwarding table against which the destination address needs to be matched can be of arbitrary lengths; this could be as a result of an arbitrary number of network aggregations. Thus, the search for the longest prefix not only requires determining the length of the longest matching prefix, but also involves finding the forwarding table entry containing the prefix of this length [11].



(a)



(b)

Figure 2.4: Examples of (a) address aggregation, (b) exception in address aggregation [11]

Chapter Three

3. Materials and Methods

In order to understand the routing process and specially software based forwarding process, an extensive literature survey was made that backs me with the relevant data. I have included the most important materials for my research and presented according to the following.

3.1. Literature Survey

The literature so far discovered revealed that there are two main schemes to solve the problem of IP address lookups processes. These are hardware approach in which case the lookups process is furnished by using hardware architecture. The second type of scheme is to use different data structure so that the lookup process can be easily implemented in software using general purpose processors. The details of the two schemes are presented as follows.

3.1.1. Hardware Based

3.1.1.1. DIR-24-8-BASIC

Gupta et al. [13] proposed a route lookup mechanism that, when implemented in a pipelined fashion in hardware, can achieve one route lookup every memory access. It is called the DIR-24-8-BASIC scheme. With a 50-ns DRAM, this corresponds to approximately $20*10^6$ packets per second.

Their techniques are based on the following assumptions:

1. Memory is cheap.
2. The route lookup mechanism will be used in routers where speed is at a premium, for example, those routers that need to process at least 10 million packets per second.

3. On backbone routers there are very few routes with prefixes longer than 24 bits. By analyzing the MAE-EAST backbone routing table, they observed that 99.93% of the prefixes are 24 bits or less.
4. IPv4 is here to stay for the time being. Thus, a hardware scheme optimized for IPv4 routing lookups is still useful today.
5. There is a single general-purpose processor participating in routing table exchange protocols and constructing a full routing table including protocol-specific information, such as the route lifetime, for each route entry. The next-hop entries from this routing table are downloaded by the general-purpose processor into each forwarding table, and are used to make per-packet forwarding decisions.

In order to implement the algorithm they proposed scheme *DIR-24-8-BASIC* that makes use of two tables as shown in the Figure 3.1 both stored in DRAM. The first table (called *TBL24*) stores all possible route prefixes that are up to, and including, 24-bits long. This table has 2^{24} entries, addressed from 0.0.0 to 255.255.255. Each entry in *TBL24* has the format shown in Figure 3.2 The second table (*TBLlong*) stores all route prefixes in the routing table that are longer than 24-bits.

Assume for example that it is required to store a prefix, *X*, in an otherwise empty routing table. If *X* is less than or equal to 24 bits long, it need only be stored in *TBL24*: the first bit of the entry is set to zero to indicate that the remaining 15 bits designate the next-hop. If, on the other hand, the prefix *X* is longer than 24 bits, then the entry in *TBL24 is used* addressed by the first 24 bits of *X*. The first bit of the entry is set to one to indicate that the remaining 15-bits contain a pointer to a set of entries in *TBLlong*. In effect, route prefixes shorter than 24-bits are expanded; e.g. the route prefix 128.23/16 will have entries associated with it in *TBL24*, ranging from the memory address 128.23.0 through 128.23.255 filled with the same next hop information

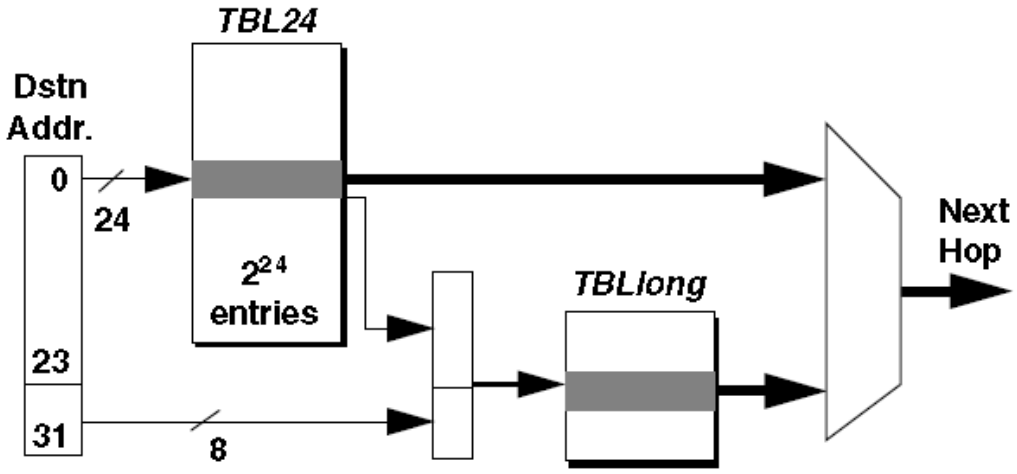


Figure 3.1: DIR-24-8-BASIC [13]

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

When a destination address is presented to the route lookup mechanism, the following steps are taken:

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

TBL24 Entry Format

If longest prefix with 24-bit prefix is < 25 bit long

0	Next hop
1bit	15 bit

If longest prefix with 24 bit is > 25 bit long

1	Index in to 2^{nd} table TBLlong
1bit	15 bit

Figure 3.2: TBL24 Entry Format

The continued decreasing cost of DRAM means that it is now feasible to perform an IPv4 routing lookup in dedicated hardware in the time that it takes to execute a single memory access. The lookup rate will improve in the future as memory speeds increase. The scheme operates by expanding the prefixes and throwing lots of cheap memory at the problem.

3.1.1.2. SRAM

N. Huang et al [14] proposed a fast longest prefix matching mechanism for IP switching routers that only needs tiny SRAM and can be implemented in a pipelined skill in hardware. Based on the proposed scheme, the forwarding database is tiny enough to fit in SRAM with very low cost. For example, a large routing table with 40,000 routing entries can be compacted to a forwarding table of 450-470 Kbytes with the cost less of US$30. Most of the address lookups can be done by a single memory access. In the worst case, the number of memory accesses for a lookup is three. When implemented in a pipeline skill in hardware, the proposed mechanism can achieve one routing lookup every memory access. With a 10ns SRAM, this mechanism furnishes approximately 100 million routing lookups per second. This is much faster than any current commercially available routing lookup schemes.

3.1.1.2.1. IP Routing Lookup Scheme

The most straightforward lookup scheme is to have a forwarding database in which an entry is designed for each 32-bit IP address as shown in Figure 3.3. This design needs only one memory access for IP address lookup but the size of forwarding database (next hop array) is too huge (2^{32} = 4 GB)[14].

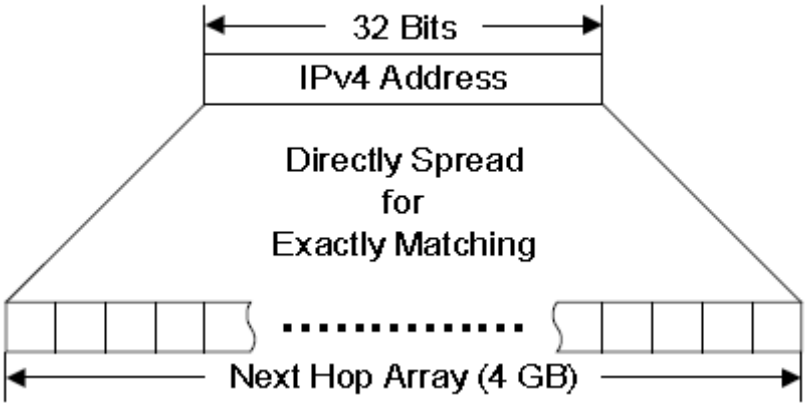


Figure 3.3: Direct Lookup Mechanism [14]

To reduce the size of forwarding database, the skill of indirect lookup shown in Figure 3.4 can be employed [14]. Each IP address is partitioned into two parts: segment (16-bit) and offset (16-bit). The segmentation table is with the size of 64K entries (2^{16}) and each entry (32-bit) records either the next hop (port number, value < 256) of the routing or a pointer (value > 255) points to the associated Next Hop Array (NHA). Each NHA consists of 64K entries (2^{16}) and each entry (8-bit) records the next hop (port number) of the destination IP address. Thus, for a destination IP address a.b.x.y, the a.b is used as the index of the Segmentation Table and the x.y is employed as that of the associated NHA, if necessary. Thus, for a segment a.b, if the length of the longest prefix belongs to this segment is less than or equal to 16 (the segment length), then the corresponding entries of the Segmentation Table store the output port directly, and the associated NHA is not necessary. On the other hand, if the length of the longest prefix belongs to this segment is greater than 16, and then an associated 64KB NHA is required. In

this design, the maximum number of memory accesses for an IP address Lookup is two. In [13], the segment length is of 24-bit and therefore a 16 Mbytes segmentation table is required. Although the indirect lookup scheme furnishes fast lookup (up to 2 memory access), it does not consider the distribution of the prefixes belongs to a segment. A 64KB NHA is required as long as the length of the longest prefix belongs to this segment is greater than 16.

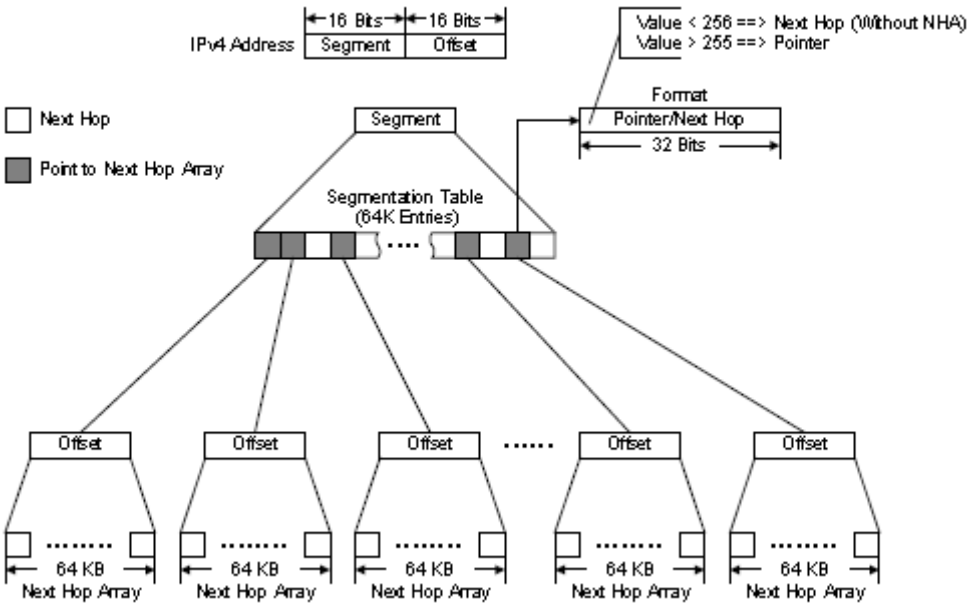


Figure 3.4: Indirect Lookup Mechanism [14]

By considering the distribution of the prefixes within a segment, they are able to further reduce the size of the associated NHA as shown in Figure 3.4. Again, the IP address is still partitioned into segment (16-bit) and offset (≤16-bit). The segment is also 16-bit and the Segmentation table is with the size of 64K entries, each one (32-bit) is divided into two fields: pointer/next hop (28-bit) and offset length (4-bit). The first field records either the next hop (port number, value <256) of the routing or a pointer (value > 255) points to the associated NHA. The second field indicates the length of the offset (k bits, $0 < k \leq 16$). For offset of k-bit length, the associated NHA is with the size of 2^k entries. For each segment, the offset length indicates how many entries are needed

in the associated NHA. This depends on the prefixes of the segment. Thus, for segment a.b, assume there are m prefixes and the longest one is of length $16 < l \leq 32$, then the offset length k for this segment is l-16. This also means that for a destination IP address a.b.x.y, the a.b is used as the index of the Segmentation Table and the left most k bits of x.y (from 16-th bit to (16+k-1)-th bit) is employed as that of the associated NHA, if necessary.

3.1.1.3. TCAM (Tertiary content addressable memory)

3.1.1.3.1. Basic TCAM Scheme

CAM is a specialized matching memory that performs parallel comparison. The CAM outputs the location (or address) where a match is found. Conventional CAM can only perform exact matching, when presenting a parallel word to the input, and cannot be applied to CIDR IP route lookup. A ternary CAM (TCAM) stores each entry with a (val, mask) pair, where val and mask are both *W*-bit numbers. For example, if $W = 6$, a prefix 110* is stored as the pair (110000, 111000). Each TCAM element matches a given input key by checking if those bits of val for which the mask bit is 1 match those in the key.

The logical structure of a TCAM device is shown in Figure 3.5. Whenever a matching operation is triggered, the 32-bit destination IP address will be compared with all the TCAM entries bit by bit, respectively and simultaneously. Since there may be multiple matches found at the same time, priority resolution should be used to select a match with the highest priority as the output. Many commercial TCAM products are order-based, such that the priority is determined by the memory location. The lower the location, the higher is the priority. Namely the *i*th TCAM has higher priority than the *j*th TCAM, if $i < j$. The priority arbitration unit selects the highest priority matched output from the TCAMs. For instance, an IP address 192.168.0.177 fed to the TCAM in Figure 3.5 results in four matches at the locations of 1, 1003, 1007, and 65535. The location of 1 is selected.

The forwarding table is stored in the TCAM in decreasing order of prefix lengths, so that the longest prefix is selected by the priority encoder. As shown in Figure 3.5, the groups of 32-bit prefixes are at the bottom of the TCAM. Note that there are some empty spaces in some groups reserved for adding prefixes in the future. The default route is located at the very top of the TCAM chip. Its mask value is 0, which guarantees that it will match with any input IP address. Only when there is no match from all the locations below it, will it be selected by the priority arbitrator. The output from the TCAM is then used to access RAM, in which the next hop information is stored in the same location as the prefix in the TCAM.

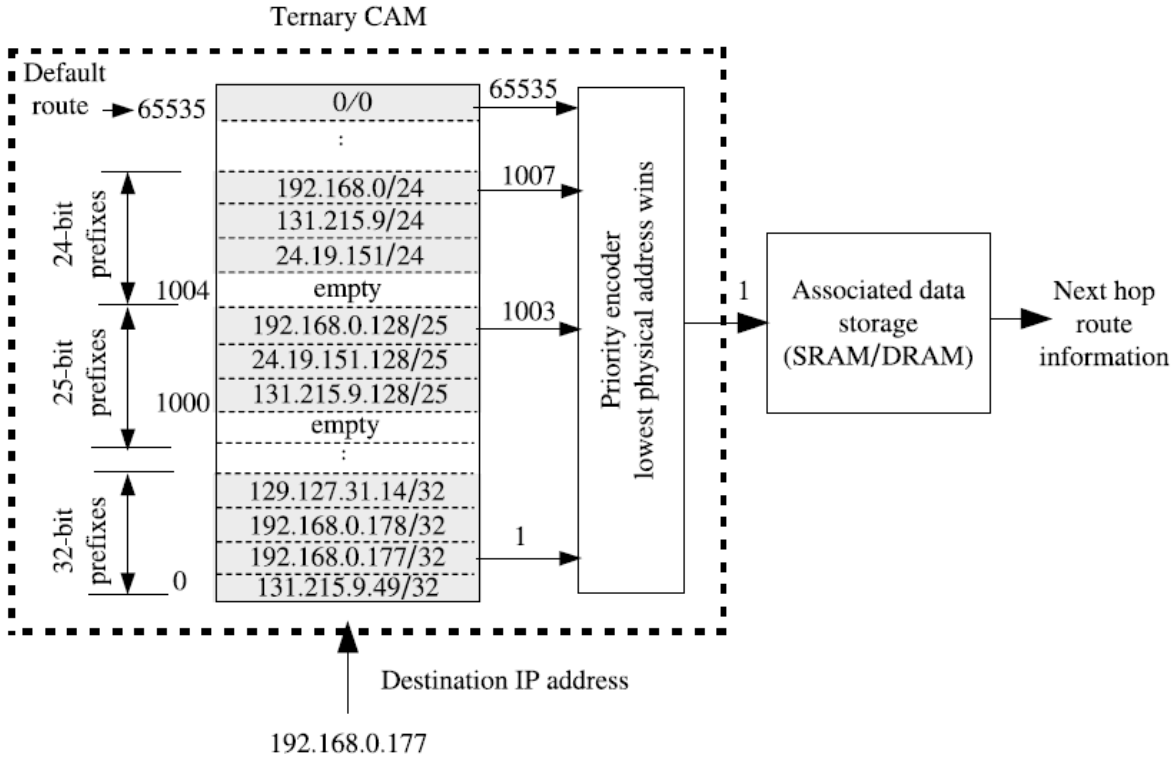


Figure 3.5: logic structure of TCAM [10]

3.1.1.3.2. Performance

TCAM returns the result of the longest matching prefix lookup within only one memory access, which is independent of the width of the search key. And the implementation of TCAM-based lookup schemes is commonly used because they are much simpler than that of the trie-based

algorithms [10]. The commercially available TCAM chips can integrate 18 M-bit (configurable to 256 k × 36-bit entries) into a single chip working at 133 MHz, which means it can perform up to 133 million lookups per second. However, the TCAM approach has the disadvantage of high cost-to-density ratio and high-power consumption (10–15Watts/chip) [10].

In addition to above mentioned hardware based methods, there are different kinds of attempts made to overcome the problem of address look up. These are using caching scheme to enhance the routing lookups, and different kinds of data structures are developed to facilitate the process. In order to explain how the remaining approach enable fast route lookups, first let's assume that we have address prefixes (routing tables) given in the table below.

Table 3.1 Prefixes of addresses

Label	Classifier	Mask	Interval
P1	0*	000000/1	[000000-100000)
P2	1*	100000/1	[100000-111111]
P3	01*	010000/2	[010000-100000)
P4	11*	110000/2	[110000-111111)
P5	100*	100000/3	[100000-101000)
P6	101*	101000/3	[101000-110000)
P7	1011*	101100/4	[101100-110000)
P8	00110*	001100/5	[001100-001110)

We use the example routing table from Table 3.1 to explain the theory behind each of these approaches

A geometrical representation of the address space and the regions imposed by this routing information is shown in Figure 3.6. In this figure, the address space is represented as a horizontal axis ranging from 0 to 2^n. Prefixes are visualized as triangles that define an interval in the address space, labeled with Pi. Based on the prefix descriptions we determine a number of regions Ri that

do not overlap. We can do this by selecting the smallest prefix-interval that contains the region we consider. The union of these regions is equal to the complete address space. If we can determine in which interval an incoming address belongs, we can match the incoming address to a prefix in our routing table [15, 16].

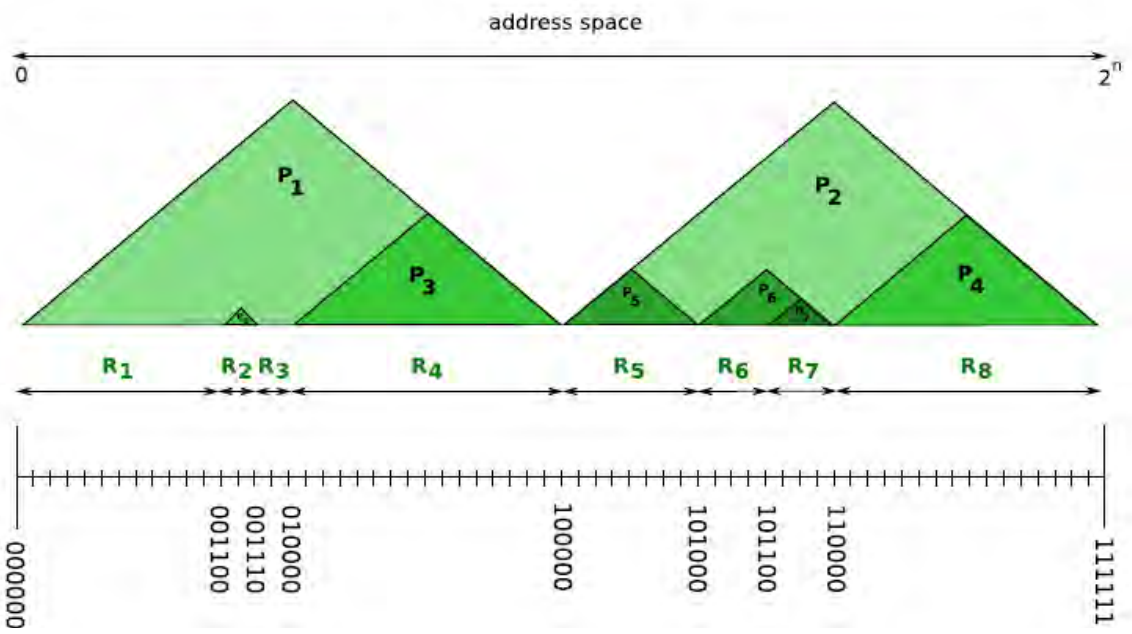


Figure 3.6: Geometric representation of the prefixes in Table 2.1

3.1.1.4. Caches

Caches use hashing to reduce lookup complexity: they focus on storing the results of commonly recurring comparisons in high-performance memory close to the calculation. The architecture is based on a hit & miss principle, where a hit will result in a fast result returned by the cache, while a miss will delegate a lookups to a (slower) higher level cache or a (very slow) external memory architecture. The higher the hit-ratio, the better performance we can expect from the cache. Efficient cache allocation algorithms exist that can reduce the miss-ratio to as low as 4% [5].

We can introduce a cache to any existing lookup-structure to reduce average lookup time, but the efficiency of the cache is highly dependent on the principle of locality, which may or may not be

present in any given data-stream. The worst-case lookup speed is inevitably increased when compared to the same system without the cache, due to the added delay introduced by a potential miss in the cache. A common method to determine whether a request is a hit or a miss is called hashing. One problem with hashing is the possibility of multiple inputs resulting in the same hash, commonly referred to as a collision. This phenomenon increases the worst-case performance of a cache significantly. Recent research has provided new hashing methods that attempt to remove these collisions from the result-set (perfect hashing), but these require additional resources [16].

In [16] an IP Lookup structure called the IP Stash is introduced. They implement a set-associative memory structure to store routing tables. Its functionality and design is very similar to a set-associative cache, but in contrast to a cache that only holds a small part of the data set, the IP Stash is the main storage for the routing table – not a cache for it.

They use Controlled Prefix Expansion (CPE) to restrict routing table prefixes to a small number of prefix lengths. To increase the capacity of the IP Stash device they apply a technique called Skewed-associativity, introduced in [15]. Skewed-associativity reduces the miss ratio of a cache by applying different indexing functions to each way of an associative cache, thereby increasing the entropy of the system by the introduction of randomness in the distribution of the items in the cache.

The architecture they propose offers significant benefits over TCAM, in terms of performance, memory requirements and power consumption. Additionally, incremental updates are effortless. However, this approach has not been embraced by industry researchers, possibly due to the complexity of an implantation.

3.1.2. Trie Based

3.1.2.1. Trie

The trie is an ordered tree data structure, used for storing an associative array of usually (string, value) pairs. The term “trie” stems from the word “re*trie*val”. A position in the trie is associated with the length of the key being compared. This is why a trie is sometimes denoted as a “search-on-length” approach. All the descendants of a node share a common prefix of the string associated with that node, and the root is associated with the empty string. Some benefits of the trie compared to exhaustive linear search include:

- Lookup is fast: a lookup of a key of length n takes O(n) time. In other words, the maximum depth of a trie is always equal to the length of the longest key in the array; keys are stored in nodes corresponding to the length of the key.
- The structure of the trie is well-suited to the problem of Longest Prefix Matching, since nodes are in locations associated with their length. This means that the last matching node will always be the longest match.

However, tries also have their flaws: they result in unbalanced decision trees, and scale linearly in depth with the length of an address. In Figure 3.7 we visualize the trie structure associated with the routing table in Table 3.1. Leaf nodes are drawn with a circles, prefixes use the labels Pi defined in Table 3.1. The axis below the tree depicts the regions from our routing table, similar to the visualization in Figure 3.6. Tracing a lookup for the value “101011” in the trie produces the dotted arrow path. This results in prefix match for P6, (101000/3), which defines the interval [101000, 110000). This is indeed the largest matching prefix for our input, and the interval contains our input value.

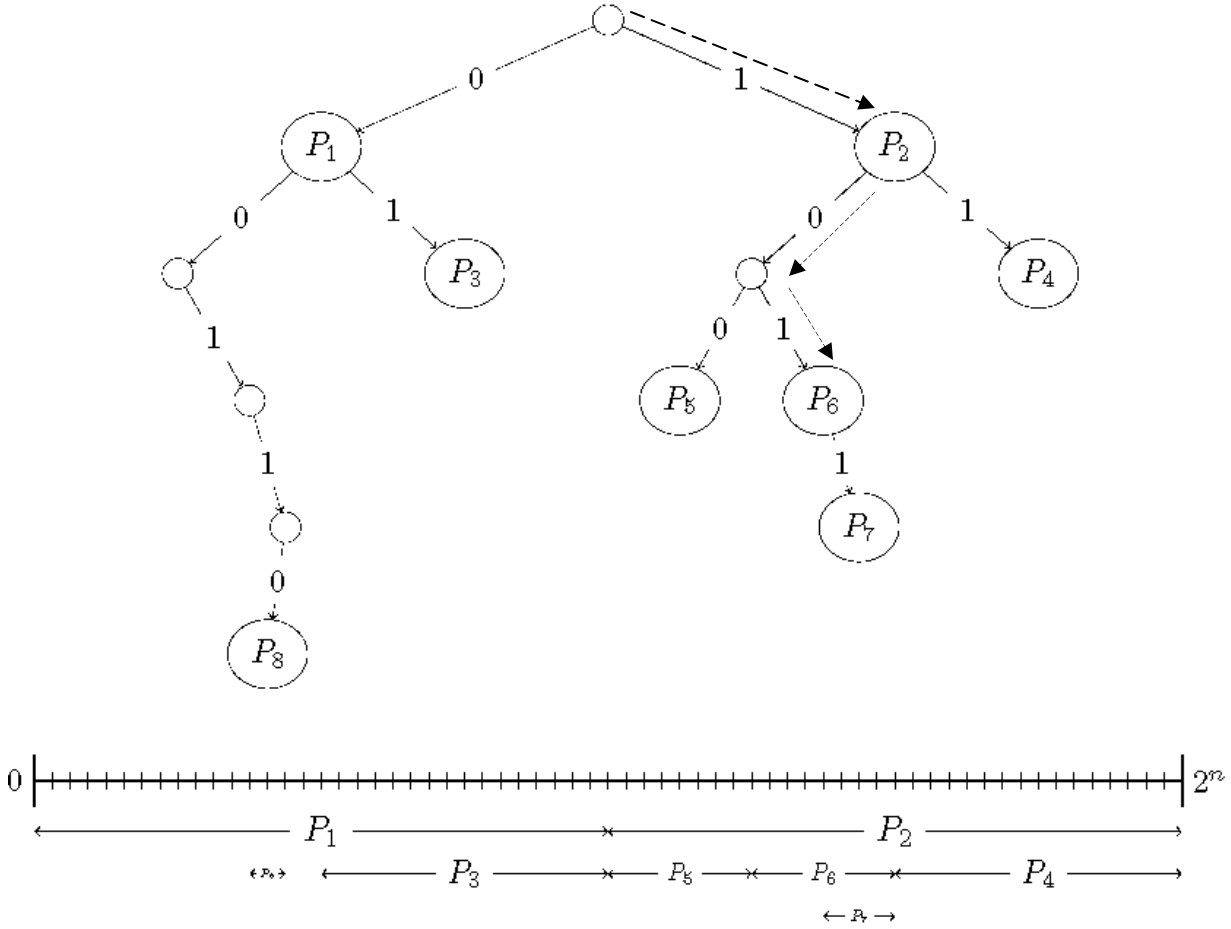


Figure 3.7: Trie Structure

We note that every prefix is stored on a node with a depth equal to the length of the prefix: $P_{1, 2}$ have prefix-length 1 and are stored in level 1 nodes, P_8 is a prefix with length 5 and it is stored in a level 5 node, etc. The axis below the trie illustrates the prefixes associated with regions in the address space.

3.1.2.2. PATRICIA Trie

A PATRICIA Trie is an enhancement of the regular Trie structure, also commonly referred to as a Radix Tree. Like the trie, a PATRICIA trie stores an associative array of key-value pairs. A node still shares a common prefix with all of its parents. Unlike the trie however, the values used for a branch decision are no longer a single bit, but can be any sequence of bits. This allows us to

reduce the depth of the trie, by pruning every path between every node with an only child. The result is a space-optimized Trie, where every internal node has two children. The depth of a node is no longer equal to the length of the prefix stored. Instead, the length of the prefix is now an upper bound for the depth of a node. This approach is still classified as a “search-on-length”. This approach shows significant benefit for smaller sets (especially if the keys are long) and for sets where keys share large common prefixes [7].

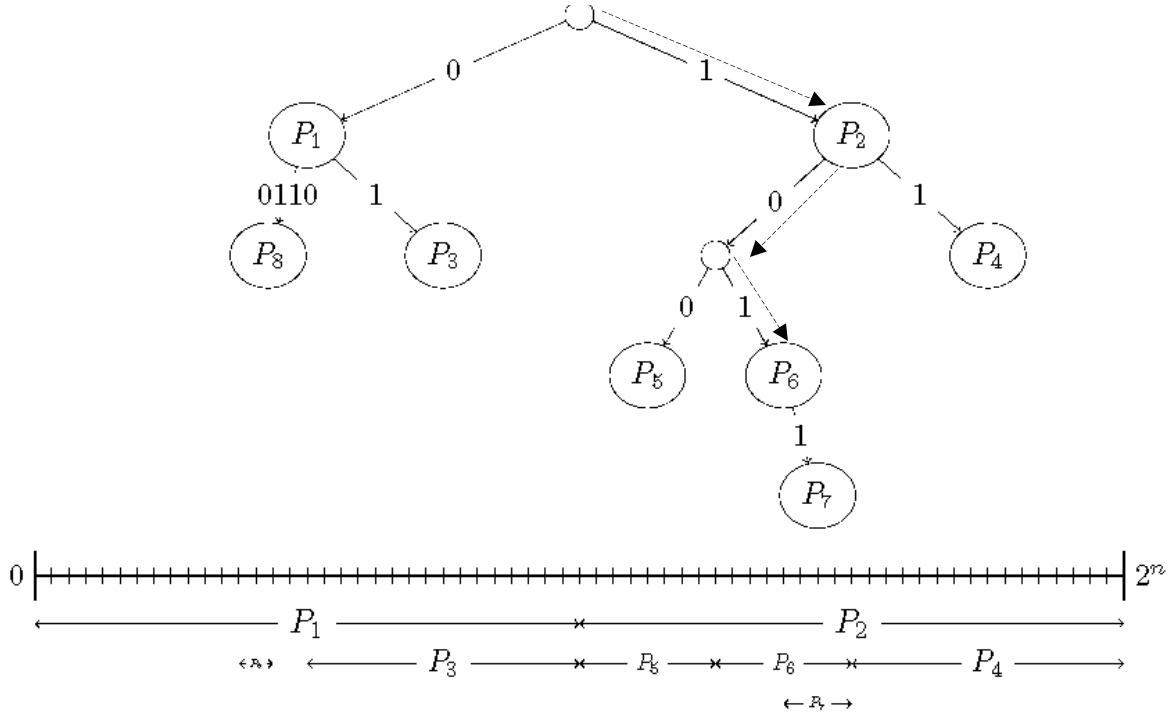


Figure 3.8: Patricia Trie Structure [7]

A lookup of a key of length n now has a worst-case complexity of O(n), as opposed to the constant complexity of O(n) for the basic trie. However, since nodes are no longer stored at a level equal to their length, we can no longer be certain that a match found is the longest possible match. The BSD implementation solves this problem by performing a lookup without checking the prefixes stored at internal nodes [5, 15]. Once at a leaf-node, the path is then backtracked to find the LPM. Note that with this scheme the worst-case complexity becomes O (2*n). The dynamic prefix trie (DP-trie) extends the concept of the PATRICIA trie to support LPM while guaranteeing retrieval times at most linear in the length of the input key, irrespective of the trie

size. If we apply this technique for the trie in Figure 3.7, we get the resulting Patricia Trie in Figure 3.8. We notice that the path to prefix P8 has been reduced in depth from 5 to 2. This reduces the maximum depth of the complete trie from 5 to 4 levels. The lookup for "101011" is identical to that in the regular trie, since there was no path-compression on the critical path to prefix P6 and thus no backtracking is required.

3.1.2.3. Multi-bit Trie

Multi-bit tries are an improvement of the binary tries. They are a sequential "search on length" approach that provides a constant factor improvement. Instead of inspecting one bit at a time, they inspect several bits simultaneously. The number of bits that are inspected per step is called *stride*. As a result, the trie depth decreases and the branching factor of a node increases, as each node has now 2^k child nodes, where k is the stride. A multi-bit trie may have a fixed stride (all nodes of a level have the same stride) or a variable stride. The process of searching in a multibit trie is almost identical to the binary trie, except that more bits may be inspected per step. An example variable-stride multibit trie based on the binary trie of the Table 3.1 is depicted in Figure 3.9.

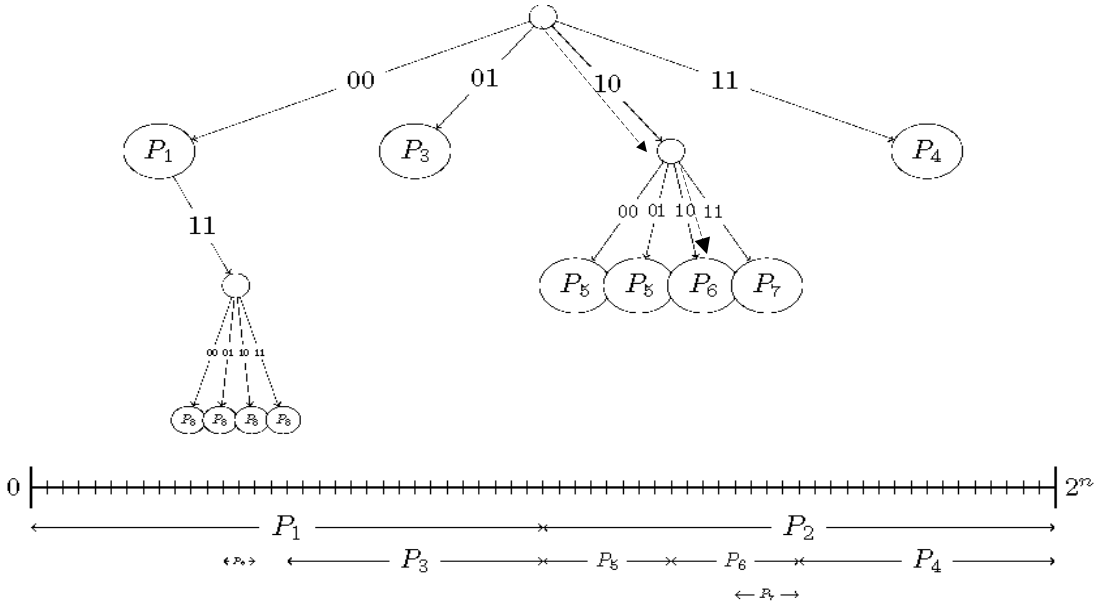


Figure 3.9: Multi bit Trie Structure

Controlled Prefix Expansion is required to compare prefixes that do not match up with the stride-value. Narrow prefixes can be transformed into multiple prefixes of a larger length. Consider comparing a prefix "0*" with a stride of 2 bits; we have to expand this prefix to "00*" and "01*" to compare the prefix in the multi-bit trie. Intuitively, the larger the stride, the shorter the trie, and the faster the lookup is. However, the memory requirements scale exponentially with the stride value [5]. In Figure 3.9 we visualize a Multi-bit Trie structure for our example data. We use a fixed stride of 2 bits at every node. The depth of the original trie is halved compared to the single-bit stride, but we also notice an increase in memory requirements: prefix P8 is duplicated four times, and P5 is stored twice in the multi-bit trie. The lookup path for value "101011" is now 2 levels deep, as highlighted on the graph. We notice that every parent node has 4 children or 2^s in the generic case.

Choosing the optimal strides is a trade-off between search speed (trie depth) and memory requirements (number of nodes). Because multibit tries cannot support arbitrary prefix lengths, it is needed to transform the prefix addresses into a compatible set where some prefixes need to be expanded. In [19] Srinivasan et al. present a method to select the optimal strides for fixed-stride tries and variable-stride tries, based on the given address prefixes. They used dynamic programming in order to minimize memory requirements and guarantee a worst-case search time.

Another practical approach for choosing the strides was followed from Gupta et al. in [13]. They noticed that in a typical backbone router of the time most of the address prefixes had a length of 24 bits or less. So, they opted for a first-level stride of 24 bits and a second-level stride of 8 bits. In this way, only two steps are needed for address lookup in the cost of extra memory resources (i.e. just the first-level memory size is 32 Mbytes).

3.1.2.4. Level Compressed Trie

Nilsson et al. in [17] combined the multibit trie (see Figure 3.9) with the path compression technique (see Figure 3.8) to gain in search time. The new scheme they introduced is also a sequential "search on length" approach and is called *Level-Compressed Tries* (LC-Tries). In

level-compression, *k*-level full binary sub-tries are recursively replaced with a corresponding one-level *k*-stride multibit trie. That way the initial *k* levels are "compressed" into one and the search time is reduced. An example LC-Trie is shown on Figure 3.10 based on the path-compressed trie of Table 3.1.

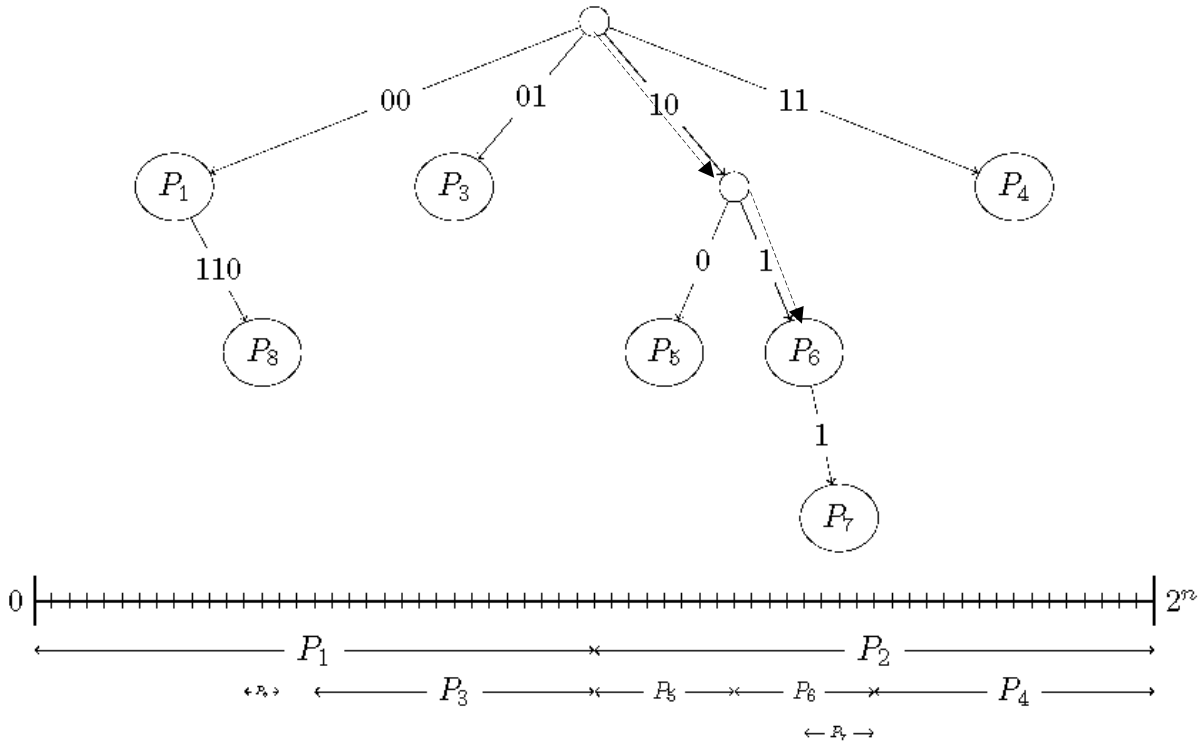


Figure 3.10: LC-Trie Structure

An optimization to the level-compression technique is to loosen the criterion for sub-trie replacement. Instead of demanding a full binary sub-trie, we may demand only a fraction of branches to be present. The required fraction is represented by the *fill factor x* ($0 < x \leq 1$) that may be used as the deciding value. The fill factor offers the possibility of a trade-off between time (trie levels) and memory requirements (number of nodes); using low fill factors decreases the trie depth, by increasing the branching factor, but may introduce unnecessary leaf nodes. It is the software lookup structure of choice for address lookup in the Linux kernel since v2.6.137 [16].

3.1.3. Comparison and Measurements of Schemes

Each of the schemes presented under section 3.1.2 has its strengths and weaknesses. In this section, we compare the different schemes and discuss the important metrics to evaluate these schemes. The ideal scheme would be one with fast searching, fast dynamic updates, and a small memory requirement. The schemes presented make different tradeoffs between these aspects. The most important metric is obviously the lookup time, but update time must also be taken into account, as well as the memory requirements. Scalability is also another important issue, both, with respect to the number of prefixes and the length of prefixes.

3.1.3.1. Complexity Analysis

A complexity comparison for the different schemes is shown in Table 3.2. The next sections carry out a detailed comparison.

3.1.3.2. Tries

In binary tries we potentially traverse a number of nodes equal to the length of addresses. Therefore the search complexity is O(W). Update operations are readily made and need basically a search, so its complexity is also O(W). Since inserting a prefix potentially creates W successive nodes (along the path that represents the prefix), the memory consumption for a set of N prefixes has complexity O(NW). Note that this upper bound is not a tight one, since some nodes are, in fact, shared along the prefix paths.

Path compression reduces the height of a sparse binary trie, but when the prefix distribution in a trie gets denser, height reduction is less effective. Hence, complexity for search and update operations in path compressed tries, is the same as that of classical binary tries. Path compressed tries are full binary tries. Full binary tries with N leaves have N-1 internal nodes. Hence, space complexity for path compressed tries is O(N).

Table 3.2 Comparison of different Schemes

Scheme	Worst case lookup	Update	Memory
Binary trie	O(W)	O(W)	O(NW)
Path-compressed tries	O(W)	O(W)	O(N)
k stride Multibit trie	O(W/ k)	O(W/k+2^k)	O(2^kNW/k)
LC trie	O(W/k)	-	O(2^kNW/k)
Lulea trie	O(W/k)	-	O(2^kNW/k)

Multibit tries still do linear search on lengths, but since the trie is traversed in larger strides the search is faster. If search is done in strides of *k* bits, the complexity of the lookup operation is O(W/k). As we have seen, updates require a search and will modify a maximum of 2^{k-1} entries (if leaf pushing is not used). Update complexity is thus O(W/k +2^k) where *k* is the maximum stride size in bits in the multibit trie. Memory consumption increases exponentially with *k*: each prefix entry may need potentially an entire path of length W/k and paths consist in one-level sub-tries of size 2^k. Hence memory used has complexity O(2^kNW/k).

Since the Lulea and the Full expansion/Compression schemes use compressed multibit tries together with the leaf pushing technique, incremental updates are difficult if not impossible and we have not indicated update complexity for these schemes. The LC trie scheme uses an array layout and must maintain lists of less specific prefixes. Hence, incremental updates are also very difficult.

3.2. Small Forwarding Table

For some time, there was an assumption that networking community had. They claim that it is impossible to do full IP routing lookups in software running on general purpose microprocessors fast enough to support routing at gigabit speeds. In fact, some believes that IP routing lookups cannot be done quickly at low cost in hardware [18].

Routing lookup in a routing table is done to determine the destination of IP datagram's packet. The result of this operation is the next hop information to which the packet is forwarded. An entry in a routing table is conceptually an arbitrary length prefix with associated next-hop information. Thus IP router should perform routing lookups to find the routing entry with the longest matching prefix.

The assumption that IP routing lookups are naturally slow and complex operations results in creation of techniques to avoid them. Various link layer switching technologies below IP, IP layer bypass methods and the development of alternative network layers based on virtual circuit technologies such as ATM, are, to some degree, results of a wish to avoid IP routing lookups [18].

Most current IP router designs use caching techniques where the routing entries of the most recently used destination addresses are kept in a cache. The technique relies on there being enough localities in the traffic so that the cache hit rate is sufficiently high and the cost of a routing lookup is amortized over several packets. These caching methods have worked well in the past. However, as the current rapid growth of the Internet increases the required size of address caches, hardware caches might become uneconomical [18].

3.2.1. Architectures of Generic Routers

The architectures of generic routers can be broadly classified into two categories. One is schematically shown in Figure 3.11. A number of *network interfaces*, *forwarding engines*, and a *network processor* are interconnected with a *switching fabric*. Inbound interfaces send packet headers to the forwarding engines through the switching fabric. The forwarding engines in turn determine which outgoing interface the packet should be sent to. This information is sent back to the inbound interface, which forwards the packet to the outbound interface. The only task of a forwarding engine is to process packet headers. All other tasks such as participating in routing protocols, resource reservation, handling packets that need extra attention and other

administrative duties are handled by the network processor. A BBN multi gigabit router is an example of this design [18].

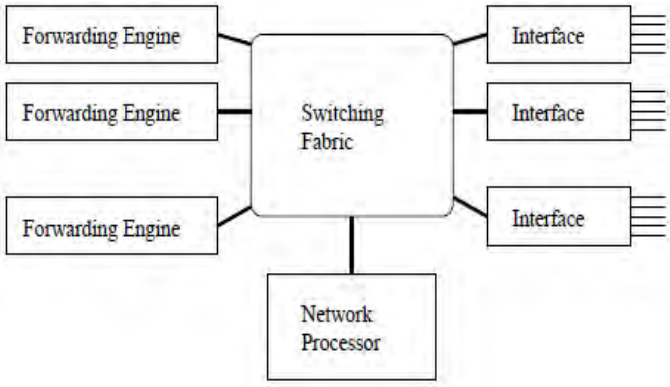


Figure 3.11: router with forwarding engine[18]

Another router architecture is shown in Figure 3.12. Here, processing elements in the inbound interface decide to which outbound interface packets should be sent. The GRF routers from Ascend Communications, for instance, use this design.

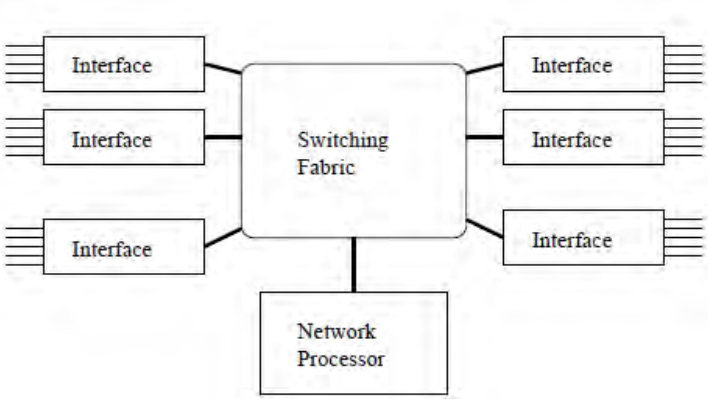


Figure 3.12: router design with processing power on interface [10]

The forwarding engines in Figure 3.11 and the processing elements in Figure 3.12 use a local version of the routing table - a *forwarding table*, downloaded from the network processor-to make their routing decisions. It is not necessary to download a new forwarding table for each routing update. Routing updates can be frequent, but since routing protocols need time on the order of minutes to converge, forwarding tables can be allowed to grow a little stale and need not change more than once per second.

The network processor needs a dynamic routing table designed for fast updates and fast generation of forwarding tables. The forwarding tables, on the other hand, can be optimized for lookup speed and need not be dynamic.

3.2.2. IP Route Lookup Design

When designing the data structure used in the forwarding table, the primary goal is to minimize lookup time. To reach that goal, two parameters should be simultaneously minimized:

- the number of memory accesses required during lookups, and
- The size of the data structure.

Reducing the number of memory accesses required during a lookup is important because memory accesses are relatively slow and usually the bottleneck of lookup procedures. If the data structure can be made small enough, it can fit entirely in the cache of a conventional microprocessor. This means that memory accesses will be orders of magnitude faster than if the data structure needs to reside in memory consisting of the relatively slow DRAM.

If the forwarding table does not fit entirely in the cache, it is still beneficial if a large fraction of the table can reside in cache. Locality in traffic patterns will keep the most frequently used pieces of the data structure in cache, so that most lookups will be fast. Moreover, it becomes feasible to use fast SRAM for the small amount of needed external memory. SRAM is expensive, and the faster it is, the more expensive it is. For a given cost, the SRAM can be faster if less is needed [10].

As secondary design goals, the data structure should

- need few instructions during lookup, and
- keep the entities naturally aligned as much as possible to avoid expensive instructions and cumbersome bit-extraction operations.

These goals have a second-order effect on the performance of the data structure [18].

For the purpose of understanding the data structure, imagine a binary tree that spans the entire IP address space Fig. 3.13. Its height is 32, and the number of its leaves is $2^{32,}$ one for each possible IP address. The prefix of a routing table entry defines a path in the tree ending in some node.

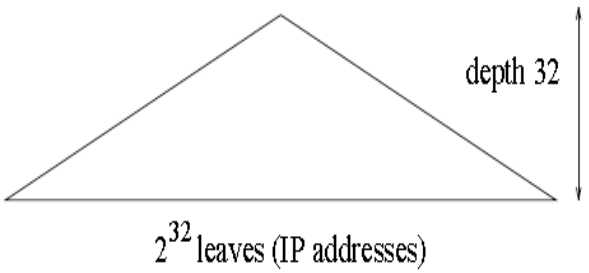


Figure 3.13: Binary tree spanning the entire IP address space [18]

All IP addresses or leaves in the sub tree rooted at that node should be routed according to that routing entry. In this manner each routing table entry defines a range of IP addresses with identical routing information (next-hop IP address).

If several routing entries cover the same IP address, the rule of the longest match is applied; it states that for a given IP address, the routing entry with the longest matching prefix should be used. This situation is illustrated in Figure 3.14; the routing entry e1 is hidden by e2 for addresses in the range *r*.

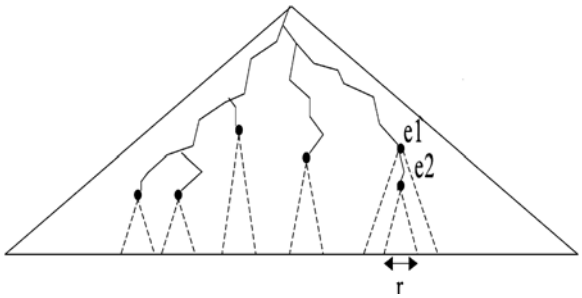


Figure 3.14: Routing entries defining ranges of IP addresses [10].

Degermark et al.[18] proposed a data structure that can represent large routing tables in a very compact form, small enough to fit entirely in a cache. This provides an advantage in that the fast IP route-lookup algorithm can be implemented in software running on general-purpose microprocessors.

The forwarding table is a representation of the binary tree spanned by all routing entries. This is called the *prefix tree*. The prefix tree is required to be full, that is, such that each node in the tree has either two or no children. Nodes with a single child must be expanded to have two children; the children added in this way are always leaves, and their next-hop information will be the same as that of the closest ancestor with next-hop information.

It is not needed to actually build the prefix tree to build the forwarding table. The forwarding table can be built during a single pass over all routing entries. The forwarding table is essentially a tree with three levels. As shown in Figure 3.16, level 1 of the data structure covers the prefix tree down to depth 16; level 2 covers depths 17 to 24, and level 3 depths 25 to 32. Wherever a part of the prefix tree extends below level 16, a level-2 chunk describes that part of the tree.

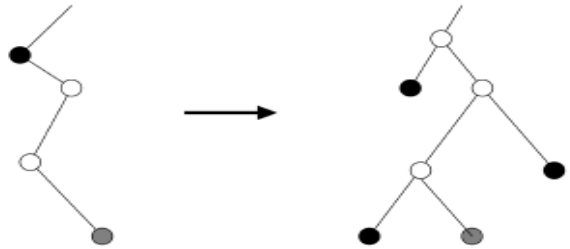


Figure 3.15: Expanding the prefix tree to be complete. [18]

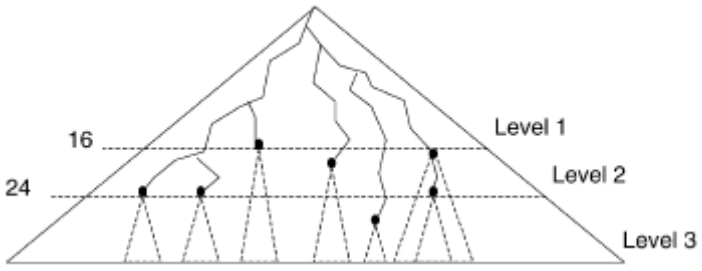


Figure 3.16: The three levels of the data structure. [18]

Similarly, chunks at level 3 describe parts of the prefix tree that are deeper than 24. The result of searching a level of the data structure is either an index into the next-hop table or an index into an array of chunks for the next level.

3.2.3. Level 1 of Data Structure

Imagine a cut through the prefix tree at depth 16. The cut is stored in a bit vector, with one bit per possible node at depth 16. For this, 2^{16} bits=64Kbits=8 Kbyte are required. To find the bit corresponding to the initial part of an IP address, the upper 16 bits of the address is used as an index into the bit vector.

When there is a node in the prefix tree at depth 16, the corresponding bit in the vector is set. Also, when the tree has a leaf at a depth less than 16, the lowest bit in the interval covered by that leaf is set. All other bits are zero. A bit in the bit vector can thus be

- one representing that the prefix tree continues below the cut: a root head (bits 6, 12, and 13 in Fig. 3.17., or
- one representing a leaf at depth 16 or less: a genuine head (bits 0, 4, 7, 8, 14, and 15 in Fig. 3.17., or
- zero, which means that this value is a member of a range covered by a leaf at a depth less than 16 (bits 1, 2, 3, 5, 9, 10, and 11 in Fig.3.17).

For genuine heads an index to the next-hop table is stored. Members will use the same index as the largest head smaller than the member. For root heads, an index to the level-2 chunk that represents the corresponding sub-tree is stored. The head information is encoded in 16-bit pointers stored in an array. Two bits of the pointer encode what kind of pointer it is, and the 14 remaining bits are indices either into the next-hop table or into an array containing level-2 chunks.

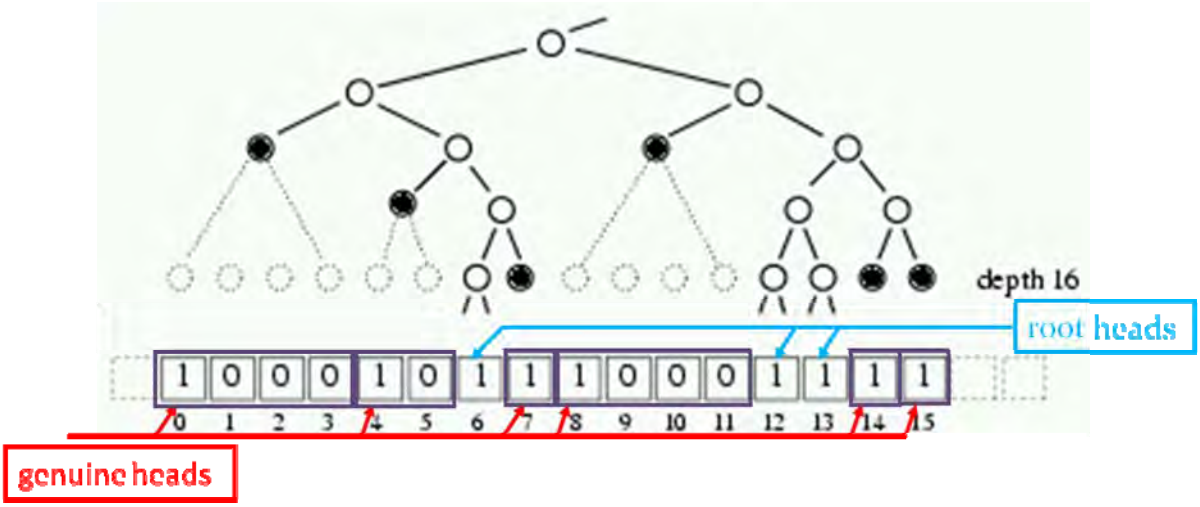


Figure 3.17: Part of cut with corresponding bit vector

So, how to find the proper pointer group? The bit vector is divided into bit masks of length 16, and there are 2^{12} = 4096 of these. The position of a pointer in the array is obtained by adding three entities: a base index, a 6-bit offset, and a 4-bit offset. The base index plus 6-bit offset determines where the pointers corresponding to a particular bit mask are stored. The 4-bit offset specifies which pointer among those to retrieve. Figure 3.18 shows how these entities are found. The following paragraphs elaborate on the procedure.

Since bit masks are generated from a full prefix tree; not all combinations of the 16 bits are possible. A nonzero bit mask of length 2^n can be any combination of two bit masks of length n or the bit mask with value 1. Let $a(n)$ be the number of possible nonzero bit masks of length $2n$. Then $a(n)$ is given by the recurrence

$$\mathbf{a(0)=1,\ a(n)=1+a(n-1)^2} \qquad \mathbf{(4)}$$

The number of possible bit masks with length 16 is thus $a(4)+1=678$; the additional one is because the bit mask can be zero. An index into a table with an entry for each bit mask thus only needs 10 bits.

A table, maptable, is kept to map bit numbers within a bit mask to 4-bit offsets. The offset specifies how many pointers to skip over to find the wanted one, so it is equal to the number of set bits smaller than the bit index. These offsets are the same for all forwarding tables, regardless of what values the pointers happen to have maptable is constant; it is generated once and for all.

The actual bit masks are not needed, and instead of the bit vector an array of 16-bit codewords consisting of a 10-bit index into maptable plus a 6-bit offset is kept. A 6-bit offset covers up to 64 pointers, so one base index per four codewords is needed. There can be at most 64K pointers, so the base indices need to be at most 16 bits (2^{16} =64K).

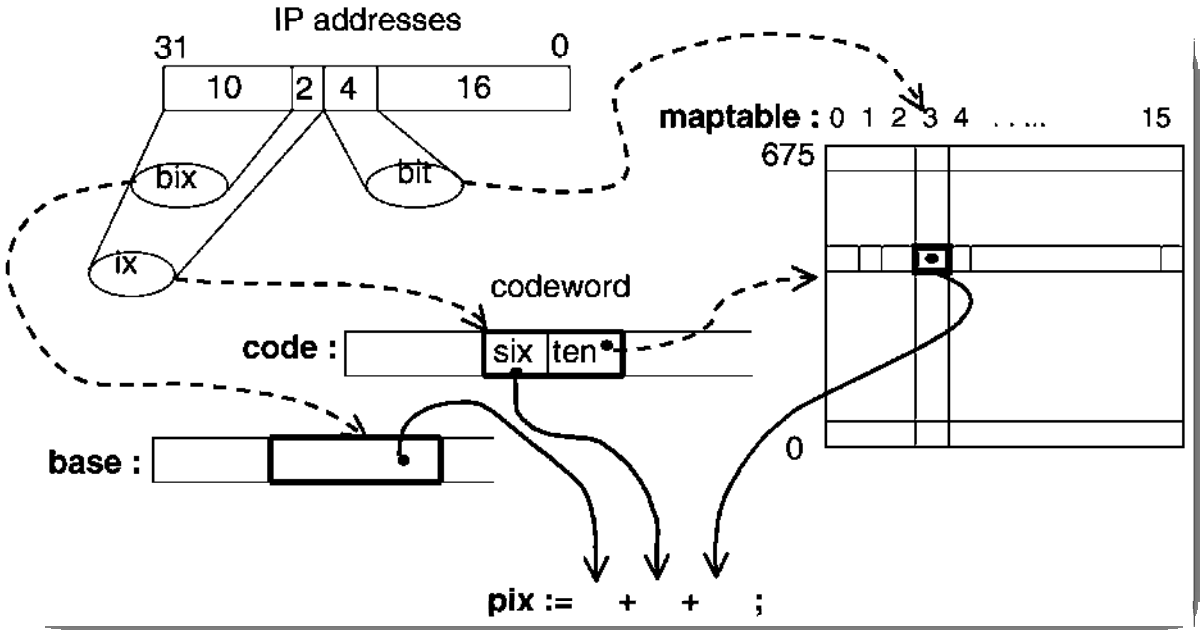


Figure 3.18: Finding pointer index [18]

The following steps are required to search the first level of the data structure; the array of codewords is called code, and the array of base addresses is called base. Figure 3.18 illustrates the procedure.

ix := high 12 bits of IP address

```
bit := low 4 of high 16 bits of IP address
codeword := code[ix]
ten := ten bits from codeword
six := six bits from codeword
bix := high ten bits of IP address
pix := base[bix]+ six + maptable[ten][bit]
pointer := level1_pointers[pix]
```

The code is extremely simple. A few bit extractions, array references, and additions are all that is needed. No multiplication or division instructions are required except for the implicit multiplications when indexing an array.

When the bit mask is zero or has a single bit set, the pointer must be an index into the next-hop table. Such pointers can be encoded directly into the codeword, and thus maptable need not contain entries for bit masks 1 and 0. The number of maptable entries is thus reduced to 676 (indices 0 through 675). When the ten bits in the codeword (*ten* above) are larger than 675, the codeword represents a direct index into the next-hop table. The six bits from the code word are used as the lowest six bits in the index, and ten- 676 gives the upper bits of the index. This encoding allows at most $(1024-676)*2^6=22,272$ next-hop indices, which is more than the 16K we are designing for. This optimization eliminates three memory references when a routing entry is located at depth 12 or more, and reduces the number of pointers in the pointer array considerably. The cost is a comparison and a conditional branch.

3.3. Materials

In order to comprehend and implement the small forwarding tables I have used the following materials to come up with feasible solution to the problems of IP lookup processes.

1. Laptop with dual core processor having clock rate of 1.87 GHz, 2GB RAM and secondary cache of 1MB.
2. NetBean 6.9.1 Integrated Development Environment (IDE) with Java programming languages.

3. In order to facilitate the process of writing java codes I have downloaded two third party libraries from internet.
    i. Java Excel API - A Java API to read, write and modify Excel spreadsheets- from http://sourceforge.net/project/showfiles.php?group_id=79926
    ii. Open Source Libraries for High Performance Scientific and Technical Computing in Java - http://nicewww.cern.ch/%7Ehoschek/colt/index.htm

Chapter Four

4. Simulation and Result

4.1. Simulation Environment

A code was developed to simulate small forwarding table using Java Programming language on Windows Vista Operating System Using NetBean Editor. The justification behind using NetBean IDE is that, it has so many and helpful documentations that it guide us to write working code. Besides, it enables to easily navigate the available library functions to use with comprehensive explanations.

4.2. Simulation Step

In order to develop a code to simulate small forwarding table algorithms, we have followed the following steps

1. The database consisting of input, subnet mask, next hop and the corresponding outgoing port was prepared in tabular form using Excel Spread sheet.
2. We have downloaded java API library from the internet that enables to read Microsoft excel file from hard disk. The API library is then added and integrated to Net Bean editor according to the instructions given in the manual.
3. Using the Net Bean editor we wrote the Java code to read data from the excel spread sheet and accordingly prepare bit vector at depth of 16.
    3.1. According to the algorithm the bit vector represents the cut-through of complete tree at depth of 16.
    3.2. When there is a node in the prefix tree at depth 16, the corresponding bit in the vector is set. Also, when the tree has a leaf at a depth less than 16, the lowest bit in the interval covered by that leaf is set. All other bits are set to zero.

4. In order to determine which bit is set and which is not, the integral equivalent of the first 16 bits of the corresponding IP address is calculated. i.e. we prepared BitVector that have a size of 2^{16} =65336. In order to facilitate this process we downloaded another Java API Library from the internet that has better capacity of handling bitwise operation more than that of BitSet Library in the Java itself.
5. The process of setting the bit within the bit vector is takes place as follow.
   5.1. The IP address is extracted as an array of octet(quad dot representation of the IP address)
   5.2. The integral equivalent the first two octet of the given IP address is calculated as:-

$$\textbf{Integral=octet [0]*}2^{8}\textbf{+octet [1]} \qquad \textbf{(5)}$$

   where octet[0] is the highest octet
   5.3. According to the value of **integral** that we have calculated in equation (5) the corresponding bit will be set and this will enables us to set the bit vector with an index from **0 to 65335**
   5.4. The bit that is set can thus be either **genuine head** or **Root head**.
   5.5. In order to identify whether the set bit is that of genuine head or root head we simultaneously stored **next hop** information or **Second level Chunk** information in the corresponding pointer. i.e. every set bit is associated with pointer and examining the contents of the pointer will determine whether the associated bit is that of genuine or root head
      5.5.1. For genuine heads we stored the corresponding next hop information interface number (with assumption that the number of next hop is limited to 60 and numbering it from 1 to 60 and 0 for default router where no match is found).
      5.5.2. For root heads we stored chunk number in the pointer starting from 500 (to avoid over lapping of next hop with next chunk).
6. In order to identify which entry goes first level and which goes to second level we analyzed the third octet of the IP address. Accordingly if the numerical value of the third octet is

equivalent to zero then the entry will be only first level and the head associated with it will be genuine heads. Otherwise the entry will be that of second level or third level.

6.1. Sometimes it happens that two entries that have the same first and second octet but on the third octet the first have value of zero (**which makes it the first level**) the other will have values other than zero. In that case

6.1.1. Both entries will go to the second level and the next hop information will be modified.

6.1.2. The same principle applies to entries that goes to third level

7. After completing the above steps for each entry in the database we are able to come up with imaginary bit vector that represents the cut-through of the complete tree of the database and simultaneously the pointers with next hop or next chunk information.
8. The following steps show how the remaining part of the algorithm is accomplished.

8.1. From bit vector that generated so far **code words** are formed

8.1.1. The bit vector is divided into bit-masks having a length of 16 bits and for every bit-masks there is code word that stores the number of set bits in the corresponding bit-masks and it contains two parts

8.1.1.1. Six part which contains the number set bits and

8.1.1.2. Ten parts which is related to **Map-table**

8.2. For every four code words there are **base index array** which contains the total number of set bits in the previous code words (or sum of the number set bits within the bit-masks) leaving the last four code words.

8.3. Map-table generated as follows

8.3.1. Every bit-masks are copied to a **two dimensional array** of Boolean value having a column of 16.

8.3.2. If the bit-mask contains all of its bits are cleared or zero it will be skipped and will not added to the table. Otherwise it will be added to Map-table.

8.3.3. If there are more than one bit-masks that have the same bit arrangement, then only single entry will be added to Map-Table.

8.3.4. As a result, the size of Map-table is less than that of bit-masks.

8.3.5. Finally the offset of each bit is within the given row is calculated

9. Finally searching

9.1. In order to search first level it is simply straight forward

9.1.1. Extract the first two octet of the incoming packet

9.1.2. Extract the high twelve bits of the octets and this will serve as the index in to the code word

9.1.3. Extract the high ten bits of the octets and this will serve as index into correct base index

9.1.4. Extract the low four bits of the two higher octet and this will gives us index to offset values within the Map-table

9.1.5. The contents of code word is separated in to six and ten

9.1.6. The ten along with four bit will points us to values within Maptable and from that we find the number of set bits up to and including the value pointed by the two

9.2. The sum of contents of base index, the six part of code word and the number from the Maptable will gives us the index to the pointer. The pointer points to either next hop or next chunk.

9.3. Searching second level is quite tricky. In the step 6 we have described that some entries goes to second level while other ends in the first level. If it ends in the first level that is all fine but if not we set up two processes.

9.3.1. First process stores the next hop information of second level in the vector and while the second keeps track of the number of entries that goes to each chunk within the second level. Later this will serve as index in to vector to search the second level.

9.4. If the pointer points to chunks the chunk number will be extracted and at the same time the number of entries within the previous chunks will be calculated as indicated in 9.3.1.

9.4.1. Similarly the number entries with in the previous chunk including this chunk also calculated.

9.4.2. The value found at 9.4 and 9.4.1 will gives us the limit of indices to the second level vector that stores about next hop information.

4.3. Simulation Results

We have tested my java code on randomly generated routing table database consisting of 96788 entries and found the following.

4.3.1. General Result

1. The core result of this scheme is that a complete tree can be represented by

   - one bit per possible leaf at depth of h plus one base index per 64 possible leaves
   - plus the information stored in the leaves
   - for h > 6 the result can be generalized as

   $$2^{h-3} + b * 2^{h-6} + l * d \quad (6)$$

   where h is the height , b base index of 2 byte , l is the number of leaves and d is the information size contained in leaves.

2. A total of seven bytes needs to be accessed to search the first level:

   - a two-byte codeword, a two-byte base address, one byte (4 bits, really) in maptable, and finally a two-byte pointer.
   - The size of the first level is 8 Kbyte for the code word array, plus 2 Kbyte for the array of base indices, plus a number of pointers.
   - The 5.3 Kbytes required by maptable is shared among all three levels.

4.3.2. Specific Result

1. There are a total of 4096 bit-mask out of which 2308 contain zero bits
2. Thus Maptable consist of 678*16 distinct entries
3. There are a total of 11392 chunks in the second level
   - 8211 Sparse chunks
   - 3113 Dense chunks and
   - 68 Very dense chunks
4. The system has a total head of 14335 out of which 11392 are Root heads while the remaining are genuine heads.
5. The time to search the first level time is about 40 micro seconds while the time to search second level increased very large as compared to first level, and it is about 350 microseconds.
6. My database of next hop information are resides on hard disk and the total time to read the database and create Map-table and the associated code word and base index and forward the packets according to this information is around 17 seconds.

Chapter Five

5. Conclusions and Limitations

5.1. Conclusions

We have demonstrated how IP routing tables can be succinctly represented and efficiently searched by structuring them as compact forwarding table that can be searched quickly to find the longest matching prefix. Our data structure is fully general and does not rely on the old class based structure of the address format for its efficiency.

The compact representation of the forwarding tables scale very well with increase in routing table sizes for IPv4 addresses. The same solution cannot be directly applied to routing on IPv6 addresses for gigabit speeds. This is because IPv6 has an exponentially large address space and compacting and placing most of the forwarding table in the processors cache is not possible. However, with small improvement similar techniques can be applied to the larger addresses of IPv6.

5.2. Limitations

We have tried to find an actual data to test with the simulation unfortunately we got none. As a result, we used a randomly generated data to test the simulation. This has its own flaws as it does not reflect the distribution of prefix in actual data.

The time between two consecutive lines is measured and it is about 8 microseconds. This value is very far from the system clock which is about 1.87 GHz. This can be attributed to the operating system's back ground process which contributes to pollution of caches and it was difficult to control this.

Bibliography

1. Seunghyun Oh, Yangsun Lee *"The Bitmap Trie for Fast Prefix Lookup"*, Web and communication Technologies and Internet-Related Social Issues — HSI 2003 Lecture Notes in Computer Science, 2003, Volume 2713/2003, 172, DOI: 10.1007/3-540-45036-X_57
2. Jan van Lunteren "*Searching Very Large Routing Tables in Fast SRAM*" Proceedings of the 10th IEEE International Conference on Computer Communications and Networks ICCCN'01, pp. 4-11
3. Devavrat Shah, Pankaj Gupta *"Fast incremental updates on Ternary-CAMs for routing lookups and packet classification"* http://*klamath.stanford.edu/~pankaj/paps/hoti00.pdf*
4. Anthony J. McAuley & Paul Francis *"Fast Routing Table Lookup Using CAMS"* INFOCOM '93. Proceedings.Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies. Networking: Foundation for the Future. IEEE, 1382 - 1391 vol.3,
5. Weidong Wu-*Packet Forwarding Technologies*- Auerbach Publications Taylor & Francis Group -2008
6. K. G. Coffman and A. M. *Odlyzko-"Internet growth: Is there a "Moore's Law" for data traffic?*
7. H. Jonathan Chao, Cheuk H. Lam, Eiji Oki *Broadband Packet Switching Technologies: A Practical Guide to ATM Switches and IP Routers* John Wiley & Sons, Inc.
8. Rita Puzˇmanová. *Routing and Switching: TIME OF CONVERGENCE?*-Addison-Wesley-2002
9. Gilbert Held. The ABCs of IP Addressing- Auerbach Publications Taylor & Francis Group -2002
10. H. JONATHAN CHAO and BIN LIU-" *High Performance Switches And Routers"-JohnWiley & Sons, Inc.-2007*

11. D. Medhi, K. Ramasamy, "*Network Routing Algorithms, Protocols, and Architectures*" - *Morgan Kaufmann Publishers -2007*
12. Gupta, P., *Algorithms for Routing Lookups and Packet Classification*, PhD thesis, Stanford University, Palo Alto, California, December 2000, pp. 7–19.
13. P. Gupta, S. Lin, and N. McKeown. "*Routing Lookups in Hardware at Memory Access Speeds.*" *IEEE INFOCOM'98, San Francisco, April 1998, Session 10B-1*.
14. Nen-Fu Huang; Shi-Ming Zhao; Jen-Yi Pan; Chi-An Su INFOCOM '99. "*A fast IP routing lookup scheme for gigabit switching routers*" Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE
15. Ruben de Smet-" *Range Trie Heuristics for Variable-Size Address Region Lookup*"- MSc THESIS-2009
16. Georgios Stefanakis-"*Design and Implementation of a Range Trie for Address Lookup* " MSc THESIS -2009
17. S. Nilsson , G. Karlsson "*Fast address lookup for Internet routers*" Proceedings of "Algorithms and Experiments" (ALEX98) Trento,Italy Feb 9-11,1998
18. Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink "*Small Forwarding Tables for Fast Routing Lookups*" Proceedings of the SIGCOMM '97 Technical Conference, Volume 27, Number 4 (October 1997)
19. V. Srinivasan and G. Varghese, *Fast address lookups using controlled prefix expansion,*ACM Transactions on Computer Systems **17** (1999), no. 1, 1–40.
20. Gary R. Wright, W. Richard Stevens- *TCP/IP Illustrated, Volume 2: The Implementation*- Addison Wesley-1995
21. http://www.scribd.com/doc/47947631/final-P21

Appendix

The Java code to simulate Small Forwarding Table

```
import java.net.*;
import cern.colt.bitvector.BitVector;
import java.util.Vector;
import java.io.*;
//import java.util.Scanner;
import java.io.IOException;
import jxl.*;
import jxl.read.biff.BiffException;

public class Finalized {
public static void main(String[] args) throws IOException {
//-------------------------------------------------------------------------------------------
//--------------------VARIABLE DECLARATION-------------------------------------
//-------------------------------------------------------------------------------------------
boolean[][] z = new boolean[1788][16];    // To store MapTable
int[] pointer = new int[65000];       /*an array to store nexthop or nexttable
                                       * information
                                       */
int  counter = 0, counter1 = 0, counter2 = 0;/*To keep track of how many entries
                                              * are added to nexthop or
```

```
                                        * next chunk
                                        */
int fh = 0, dj = 0, x;/* fh index of array to store entries that goes
                        * in to the same chunk
                        * x an index of bit vectors
                        * dj the number of entries that goes to the same
                        * chunk
                        */
 int[] junta = new int[14336];// to count number of entries that goes to each chunk
 int[] sumofjunta = new int[11392];// serves as index in to nexthop information of chunks
BitVector bv = new BitVector(57088);/* to store the bit vectors
     *that is generated at the depth of 16
    */
BitVector y = new BitVector(16);  //to divide bit vector in to bitmask of length 16
short vx; // the third octet of IP Address
Vector<Integer> Y = new Vector<Integer>();   /* avector that stores the
                                                *nexthop information of second chunk
                                                *with corresonding third octet
                                                */
String source, nexthop; // entry with corresponding nexthop information
int a = 0, chunk = 500, xz = 0; /* a is iterator
                                * chunk
                                * index of vector Y that is to be modified
```

```
                                        */
 byte[] ip1 = {0, 0, 0, 0};//Previous IP address----------
 int nh = 0;                    //The corresponding nexthop of previous IP address
 int fg = 0;
//-------------------------------------------------------------------------------------------
//--------------------READ DATA FROM EXCEL FILE--------------------------------
//-------------------------------------------------------------------------------------------
 try {
Workbook workbook = Workbook.getWorkbook(new File(
"C:\\Users\\ABU\\Desktop\\Book2.xls") );
 Sheet sheet = workbook.getSheet(0); /*
                                        *where the database of entries are stored
                                        */

for (int j = a; j < a + 1; j++)
{
for (int i = 3; i < sheet.getRows(); i++)
{
Cell cell = sheet.getCell(j, i);
source = cell.getContents();              //contains nexthop entry
Cell cell1 = sheet.getCell(j + 3, i);
nexthop = cell1.getContents();          //the port address of nexthop entry
byte zs = Byte.parseByte(nexthop);
```

```
//---------------------------------------------------------------------------
InetAddress ip = null;
try {
ip = InetAddress.getByName(source);
}
catch (UnknownHostException e)
{
e.printStackTrace();
return;
}
//---------------------------------------------------------------------------
byte[] binaryIP = ip.getAddress();/*extract ip address
                                    * from string into
                                    * octet
                                    */
if (binaryIP[0] < 0 && binaryIP[1] < 0) {
x = ((binaryIP[0] + 256) * 256) + (binaryIP[1] + 256);
} else if (binaryIP[0] < 0 && binaryIP[1] >= 0) {
x = ((binaryIP[0] + 256) * 256) + binaryIP[1];
} else if (binaryIP[0] >= 0 && binaryIP[1] < 0) {
x = (binaryIP[0] * 256) + (binaryIP[1] + 256);
 } else {
 x = (binaryIP[0] * 256) + binaryIP[1];
```

```
}
bv.set(x);


//-------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------
 if (binaryIP[2] < 0) {
 vx = (short) (binaryIP[2] + 256);
 } else {
 vx = binaryIP[2];
}
//-------------------------------------------------------------------------------------
//--------------This is to identify the nexthop ports that----------------------
//-----------------found in the second level chunks-----------------------------
if (binaryIP[2] != 0 || binaryIP[3] != 0)
{
Y.add((zs + 100 * vx));
if (ip1[0] == binaryIP[0] && ip1[1] == binaryIP[1] && ip1[2] == 0 && ip1[3] == 0)
{
 Y.add(xz, nh);
}
}
//-------------------------------------------------------------------------------------
//-------------This is the place where the chunk or nexthop---------------------
```

```
//----------is identified----------

counter = bv.cardinality();
counter2 = counter1;
counter1 = bv.cardinality();
if (counter1 == counter2)
{
 dj++;
} else {
dj = 1;
fh++;
chunk++;
}
if (binaryIP[2] == 0)
{
if (binaryIP[3] == 0)
{
pointer[counter] = zs;
} else {
 pointer[counter] = chunk;
}
} else if (binaryIP[2] != 0 || binaryIP[3] != 0) {
pointer[counter] = chunk;
```

```
fg++;
}
junta[fh] = dj;
ip1 = binaryIP;//----------------------------
nh = zs;
xz = Y.size();//to keep track of how many elements are added to the vector
}
j = j + 4;
a = 6;
}
workbook.close();
} catch (BiffException e) {
e.printStackTrace();
}
//-------------------------------------------------------------------------------------
//--------This is to renumber the second level chunk as----------------------------
//------------------named previously unordered-------------------------------------
int n = 0, sh = 500;
for (int i = 1; i < 14336; i++)
{
 if (pointer[i] >= 500)
{
 pointer[i] = sh;
```

```
 sh++;
}
}
//-------------------------------------------------------------------------------------
//--------------This is to calculate the index of Vector----------------------------
//--------------so as to find the nexthop associated with it-----------------------
int dd = 0, sum2 = 0;
for (int i = 0; i < 14336; i++)
{
if (pointer[i] >= 500)
{
sumofjunta[dd] = junta[i] + sum2;
sum2 = sumofjunta[dd];
 dd++;
}
}
System.out.println(Y.get(sum2 - 1));

//-------------------------------------------------------------------------------------------------
//----------------------------VARIABLE DECLARATION FOR---------------------------------
//--------------------CODEWORD, BASE INDEX AND MAPTABLE--------------------------
int j = 0, c, f, ab = 0, kd = 0, dc = 0, fs = 0, r = 100, h = 0,b;
int[] codeword = new int[bv.size() / 16];
```

```
int[] base = new int[bv.size() / 64];

int[] sum = new int[3568];

for (int i = 0; i < bv.size(); i++)

{

y = bv.partFromTo(i, i + 15);

b = y.cardinality();

f = b + dc;

sum[j] = f;

//-----------------------------------------------------------------------------------------------------

//-------------------------------------------codeword of bitmask-----------------------------------

if (j % 4 == 0)

{

codeword[j] = 0 + r;

ab = b;


} else {

codeword[j] = ab + r;

ab = b + ab;

}

//-----------------------------------------------------------------------------------------------------

//----------------------base index for every four code words-----------------------------------

if (j % 4 == 0)

{
```

```
if (j == 0 || j == 1 || j == 2 || j == 3)
{
kd = 0;
} else {
kd++;
}
}
if (kd == 0)
{
base[kd] = 0;
} else {
base[kd] = sum[4 * kd - 1];
dc = b + dc;
}
j++;

//-------------------------------------------------------------------------------------------------------
//--------------------------------Maptable Generated From bit masks--------------------------------
//---------------------------------dropping every bitmasks that contain only bit zero----------------
if (b != 0)
{
for (int l = 0; l < 16; l++)
{
```

```
z[fs][l] = y.get(l);
}
fs++;
r = r + 100;
}
 i = i + 15;
}
System.out.println(z[1787][15]);
//-------------------------------------------------------------------------------------------------------
//------------------------SEARCH THE DATABASE-----------------------------------------------
short twelve, tenofword, six, four;
short mod, div, tenofbase;
boolean bit;
int index, w = 0;
short[] s = new short[4];
 // Scanner in = new Scanner(System.in);
// while(in.hasNextLine()){
long start = System.nanoTime();
String address = "4.200.223.0";
//String address = in.nextLine()
InetAddress ip = null;
try
{
```

```
ip = InetAddress.getByName(address);
}
 catch (UnknownHostException e) {
e.printStackTrace();
return;
}
byte[] addr = ip.getAddress();
for (int i = 0; i < 4; i++)
{
s[i] = addr[i];
if (addr[i] < 0) {
s[i] = (short) (addr[i] + 256);
}
}
div = (short) (s[1] / 16);
mod = (short) (s[1] % 16);
twelve = (short) (s[0] * 16 + div);
tenofbase = (short) ((s[0] * 4) + s[1] / 64);
six = (short) (codeword[twelve] % 100);
tenofword = (short) ((codeword[twelve] / 100) - 1);
four = (short) base[tenofbase];

for (int i = 0; i <= mod; i++)
```

```
{
if (z[tenofword][i] == true)
{
 w++;
}
}
index = four + w + six;//mod
long star1=System.nanoTime();
System.out.println(star1-start);
if (pointer[index] < 500) {
System.out.println("the packet is forwarded to nexthop with port number :" + pointer[index]);
} else {
int ch = pointer[index] - 500;
int yk = sumofjunta[ch];
int iv = Y.get(yk - 1);
int yk1 = sumofjunta[ch - 1];
int mid;
int por = 0;
if (ch == 0) {
mid = yk;
} else {
mid = yk - yk1;
}
```

```
for (int i = 0; i < mid; i++)
{
if (Y.get(yk - 1 - i) / 100 == s[2])
{
por = Y.get(yk - 1 - i) % 100;
}
}
for (int i = 0; i < mid; i++)
{
if (Y.get(yk - 1 - i) / 100 - s[2] < 0) {
por = Y.get(yk - 1 - i) % 100;
break;
}
}
System.out.println(iv + " " + " " + yk);
System.out.println("the packet is forwarded to the chunk: " + ch + " with port no: " + por);
}
long end = System.nanoTime();
System.out.println(end - start);
}
}
```

The derivation of recurrence equation

Since bit masks are generated from a full prefix tree; not all combinations of the 16 bits are possible. A nonzero bit mask of length 2^n can be any combination of two bit masks of length *n* or the bit mask with value 1.

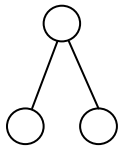
This can be proved by induction.

If a(n) is the number of non-zero bit mask of length 2^n.

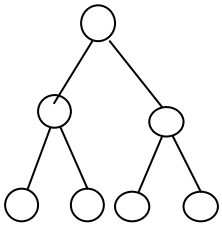
a) For n=0, the mask length is 2^0 = 1. For a mask length of 1, the number of non-zero bit-mask is only 1. It means, a(0) = 1.

b) For n=1, the mask length is 2^1 = 2.

It's tree is:



This is a complete tree. The number of non-zero bit mask of length 2 and the combination is [1 0, 1 1]. Bit mask [0 0] is skipped because it's not a non-zero bit mask. Whereas [0 1] bit mask is invalid due to the fact that the complete trees are filled from left to right.
Thus a(1)=2.
Similarly for n=2, the mask length 2^2=4. It's tree is



For this tree the possible non-zero bit-mask combination is given as

1 0 0 0

1 0 1 0

1 0 1 1

1 1 1 0

1 1 1 1

The other combinations are not legal bit-masks. This implies that a(2)=5.

Hence

a(0) = 1

a(1) = 2 = 1 + 1^2 = 1 + a(0)^2

a(2) = 5 = 1 + 2^2 = 1 + a(1)^2

So by induction

a(n) = 1 + a(n-1)^2

DECLARATIONS

I, the undersigned, hereby declare that this thesis is my original work performed under the supervision of Dr. Getachew Hailu, has not been presented as a thesis for a degree program in any other university and all sources of materials used for the thesis are duly acknowledged.

Musa Hayato

August, 2011.