



UNIVERSITY OF NAIROBI

SCHOOL OF COMPUTING AND INFORMATICS

**OPTIMIZATION AND PERFORMANCE EVALUATION OF IP LOOKUP
ALGORITHMS.**

BY

OTIENO STEPHEN OBARE

P58/70475/2008


SUPERVISOR: Eric Ayienga.

MARCH, 2012

**Submitted in partial fulfillment of the requirements of the Master of Science in
Computer Science.**

DECLARATION

This project, as presented in this report, is my original work and has not been presented for any other University award.

Sign:  Date: 15-08-12
Otieno Stephen Obare
P58/70475/2008

This project has been submitted as partial fulfillment of the requirements for the Master of Science degree in Computer Science of the University of Nairobi with my approval as the University supervisor.

Sign:  Date: 15/8/2012

Mr. Eric Ayienga,
Lecturer,
School of Computing and Informatics,
University of Nairobi.

ABSTRACT:

The number of people accessing the internet grows exponential and soon half of the world's population will have access to internet. New services and applications are also added daily on the popular IP networks and this trend is likely to continue into the future. More precisely, this development is in three major parameters of the internet activity: the number of connected nodes and endpoints is increasing, resulting in growth of routing table sizes, the number of users increases, resulting in larger internet traffic, and the complexity of the provided services increases, also causing an increase in traffic by delivering higher amounts of data per transaction. All these translate into a growing increase in traffic demands, which can only be answered by improvement in the service given by 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 incoming packet the next hop, is therefore crucial to achieve the packet forwarding rates required. IP address lookup is difficult because it requires a longest matching prefix search.

In this research work, I consider the problem of organizing the Internet forwarding tables in such a way as to enable fast routing lookup performance. In the last couple of years, various algorithms for high performance IP address lookup have been proposed. I present a brief survey of state-of-the-art IP address lookup algorithms. I concentrate on four recently proposed methods and try to evaluate their performance. I describe my implementation of the methods and results of performance measurements on artificially generated input data. Some conclusions about the general behavior of all methods, based on the measurements and theoretical reasoning is presented. Finally, I comment on the results, suggesting preference among the methods.

ACKNOWLEDGEMENT

To my supervisor Mr. Eric Ayienga, for making hard work less hard and without whom this project wouldn't have been such a great experience.

To Sue: the core router in my network where the forwarding rates varies instantaneously and over time, and where looping occur . . .

Table of Contents

| | |
|--|-------------|
| LIST OF TABLES | VII |
| LIST OF FIGURES | VIII |
| ABBREVIATIONS | IX |
| PREFIX | XI |
| CHAPTER 1 | 1 |
| INTRODUCTION | 1 |
| 1.1 BACKGROUND | 1 |
| 1.2 PROBLEM STATEMENT: | 2 |
| 1.2 OBJECTIVES: | 4 |
| 1.3 RESEARCH QUESTIONS | 5 |
| 1.4 SIGNIFICANCE OF THE STUDY | 5 |
| LITERATURE REVIEW | 6 |
| 2.1 THE CLASSFUL INTERNET ADDRESSING: | 7 |
| 2.2 THE CIDR ADDRESSING SCHEME | 10 |
| 2.3 IPV6 ADDRESS ARCHITECTURE | 13 |
| 2.4 DIFFICULTY OF THE LONGEST MATCHING PREFIX SEARCH | 15 |
| 2.5 CLASSIFICATIONS OF IP LOOKUP ALGORITHMS | 16 |
| 2.6 SIMULATION | 35 |
| CHAPTER 3 | 40 |
| IMPLEMENTATION | 40 |
| 3.1 INTRODUCTION | 40 |
| 3.2 OBSERVATIONS: | 41 |
| 3.3 DESIGN OF SIMULATOR: | 45 |
| 3.4 MODULES: | 46 |
| CHAPTER 4 | 51 |
| EXPERIMENTAL RESULTS | 51 |
| 4.1 INTRODUCTION | 51 |

| | |
|---|-----------|
| 4.2 The objectives of the experiments:..... | 51 |
| 4.3 Justification for selected plans..... | 52 |
| 4.4 Methodology..... | 52 |
| CHAPTER 5..... | 66 |
| CONCLUSION..... | 66 |
| 5.1 Conclusion..... | 66 |
| 5.2 Recommendation and future work..... | 67 |
| BIBLIOGRAPHY..... | 69 |

LIST OF TABLES

PAGE

| | |
|--|----|
| Table 2.1: Forwarding table. | 7 |
| Table 2.2: Address Aggregations | 9 |
| Table 2.3: Prefix Table. | 17 |
| Table 2.4: Original Forwarding Table Prefixes. | 19 |
| Table 2.5: Forwarding Table Prefixes after expansions. | 22 |

| LIST OF FIGURES | PAGE |
|---|-------------|
| Figure 2.1: Internet Two Level hierarchy | 1 |
| Figure 2.2: Forwarding table | 2 |
| Figure 2.3: Address Aggregations | 4 |
| Figure 2.4: An exception Prefix | 7 |
| Figure 2.5: Allocation Policy for IPv6 | 10 |
| Figure 2.6: Example network with 6-bit IP addresses | 11 |
| Figure 2.7: A Binary Trie | 11 |
| Figure 2.8: Path Compressed Trie | 12 |
| Figure 2.8: Patricia Trie | 14 |
| Figure 2.9: Multibit Node | 16 |
| Figure 2.10: Multibit Node with Pointers | 19 |
| Figure 2.11: Root Unibit Node with Bitmaps | 23 |
| Figure 2.13: Root Tree Bitmap Node with Child Array | 25 |
| Figure 2.14: Optimized Tree Bitmap | 26 |
| Figure 3.1: Distribution of IP Prefixes | 28 |
| Figure 3.2: Projected Forwarding Table Growth | 29 |
| Figure 3.3: Top Level Architecture of the Model | 31 |
| Figure 3.4: User Interfaces Screen Shots | 32 |
| Figure 3.5: Search Module Screen Shot | 35 |
| Figure 3.6: Output Module Screen Shot | 35 |
| Figure 4.1: IP Lookup times under varying IPv4 FIB sizes. | 37 |
| Figure 4.2: IP Lookup times under varying IPv6 FIB sizes. | 37 |
| Figure 4.3 Memory requirements under varying IPv4 FIB sizes. | 40 |
| Figure 4.4: Memory requirements under varying IPv6 FIB sizes. | 40 |
| Figure 4.5: Lookup times various percentages of the FIB | 41 |
| Figure 4.6: Memory required various percentages of the FIB | 42 |

ABBREVIATIONS

| | |
|-----------|---|
| IETF: | Internet Engineering Task Force |
| CIDR: | Classless interdomain routing. |
| IANA: | Internet Assigned Numbers Authority. |
| RIR: | Regional Internet Registries. |
| ISP: | Internet Service Providers. |
| LIR: | Local Internet Registries. |
| EU: | End Users. |
| IAB: | Internet Architecture Board. |
| IESG: | Internet Engineering Steering Group. |
| PATRICIA: | Practical Algorithm To Retrieve Information Coded in Alphanumeric Implementation. |
| BMP: | Best Matching Prefix. |
| RFC: | Request for comments. |
| IPv4: | Internet Protocol version 4. |
| IPv6: | Internet Protocol version 6. |
| RIPE: | Réseaux IP Européens. |
| NCC: | Network Coordination Centre. |
| RIS: | Routing Information Service . |
| TCAM: | Ternary Content Addressable Memory. |
| LIR: | Local Internet Registry. |
| TLA: | Top Level Aggregation. |
| NLA: | Next Level Aggregation. |
| SLA: | Site Level Aggregation. |

DEFINITION OF TERMS:

Routing:

Is the process of running various protocols like BGP, OSPF, LDP, etc to arrive at set of efficient routes towards various network destinations. Routing is the process by which forwarding tables are built.

Forwarding

Refers to the process of receiving packets, performing route lookup on the packet and sending the packet on an output interface. Forwarding involves various other functions like policing, rate-shaping, QOS, etc. Consists of taking a packet, looking at its destination address, consulting a table, and sending the packet in a direction determined by that table

A backbone router:

Is different from any other low-end router. It performs forwarding at very high speeds and does route lookup on large number of routes ($\approx 120,000$). In backbone routers, routing protocol traffic and data traffic follow different paths through the router. These are usually called the slow-path and fast-path respectively.

Throughput

Rate at which packets are sent/received without loss.

Latency

Time spend by a packet inside the router.

Longest prefix match:

Refers to an algorithm used by routers in Internet Protocol (IP) networking to select an entry from a routing table. Because each entry in a routing table may specify a network, one destination address may match more than one routing table entry. The most specific table entry — the one with the highest subnet mask — is called the longest prefix match.

A stride:

Refers to the number of bits to be inspected per step in multibit trie and it can be constant or variable. Multibit tries cannot support arbitrary prefix lengths since they allow traversing the data structure in strides of several bits at a time.

Prefix

The IP prefix identifies the number of significant bits used to identify a network. For example, 192.9.205.22 /18 means, the first 18 bits are used to represent the network and the remaining 14 bits are used to identify hosts.

Binary Trie

Represent the prefix space, with each node for a possible prefix. The prefix of a route table entry defines a path in the trie ending in some node, which is called the Prefix Nodes. If a node itself is not a prefix node but its descendants include prefix nodes, it is called it an Internal Node.

Build time

Time necessary to build the forwarding table from the in memory ordered list of routing table entries. The time to read the values or to perform the sorting is not included.

CHAPTER 1 INTRODUCTION

1.1 Background

The number of people accessing the internet grows exponential and soon half of the world's population will have access to internet. New services, applications, computers, smart phones and hand held devices are added daily on the popular IP networks and this trend will continue into the future. These devices overload the world-wide-area networks consuming lots of IP addresses as shown below.

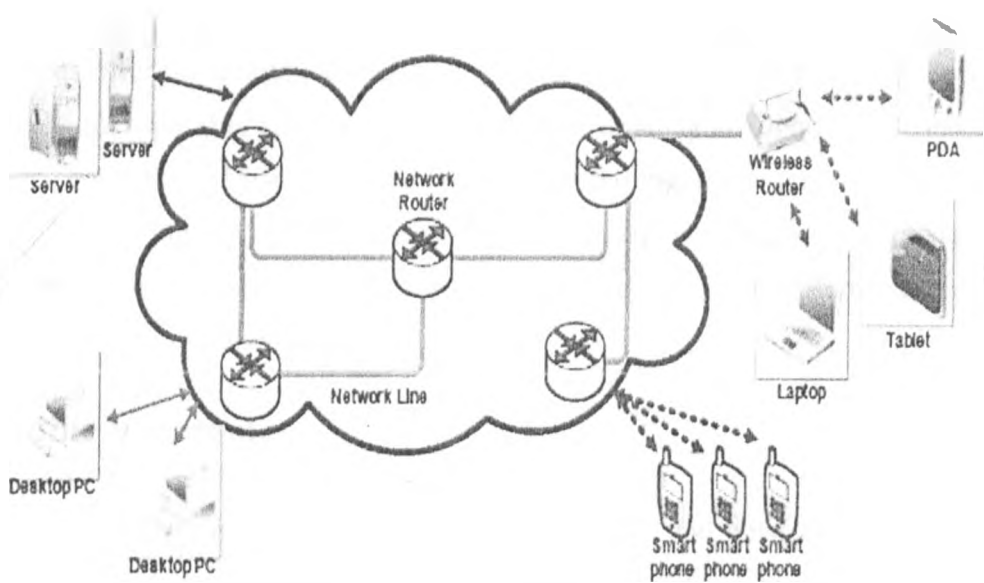


Figure 1: Services, applications and devices

Three key factors that affect a router's performance and that must keep pace with these demand if Internet is to continue to provide good service are: link speeds, router data throughput and packet forwarding rates. Solutions for the first two factors are available: Backbone links of several gigabits per second such as fiber can provide faster links, and current switching technology can be used to move packets from the input interface of a router to the corresponding output interface at gigabit speeds. Packet forwarding is done by the router in order to decide the destination port of the incoming packet. This is done by looking up the packet's destination address in the forwarding database. Therefore if an

improvement in address lookup can be found, especially in relation to the imminent increase in address lengths and growth of the routing database, then the performance of packet forwarding in routers will significantly improve hence improve internet services.

Past evaluations by G. Varghese, W. Eatherton, Z. Dittia, 2010, Miguel A. Ruiz-Sanchez et al, 2001, Jun Wang, Klara Nahrstedt, 2008, Packer Ali, Dhamim, 2000 have studied IP Lookup problem in routers. However these studies were based on algorithmic analysis, hardware implementation and some were performed in testbeds with small number of nodes. When evaluating IP Lookup algorithms it is very important to test its scalability and 1000 nodes are not usually sufficient.

Simulations offer an alternative that makes easier to test algorithms in large scenarios. Simulators are especially suitable when we want to evaluate an algorithm by modifying different parameters or when we want to modify the algorithm (create a new one or improve an existing one). In this sense, simulations allow us to write a model of an algorithm, protocol or system and study their behaviour through different experiments which can include a large number of nodes.

1.2 Problem Statement:

There are 4.3 billion unique Internet Protocol addresses and now they're running out. This growth has led to exhaustion of IPv4 addresses and the transition to IPv6 which offers an 'infinite' number of IP addresses is inevitable. The growth will cause a major increase in the length and number of addresses, and in turn the number of IP address prefixes which are used to aggregate IP addresses into networks. This stresses the need for an address lookup solution that is as less dependent as possible on the length of IP addresses and size of the searched database which is bound to grow significantly.

For every arriving packet, IP routers perform two steps: look up the next-hop of the packet from the forwarding database and forward the packet to the outgoing interface determined by the first step. The first step heavily influences the performance of routers because the longest prefix matching is complicated after the introduction of CIDR

(Classless Inter-Domain Routing). Therefore, an address lookup method easily becomes a performance bottleneck at high-speed routers.

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*.

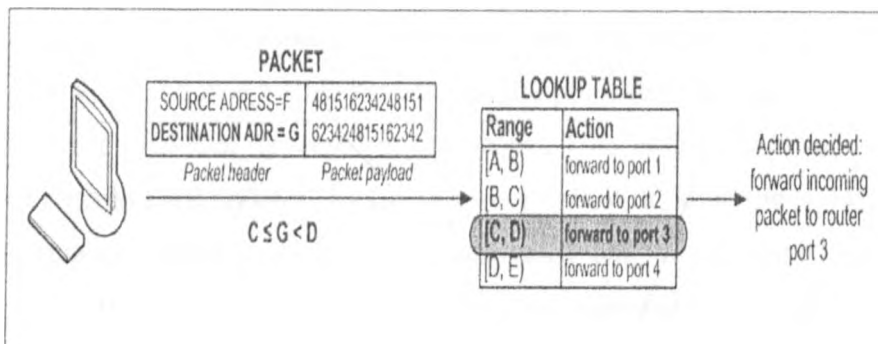


Figure 1.2: Simple depiction of IP Lookup

Given the tree and a candidate address thought of as a sequence of bits, the lookup algorithm is as follows:

1. Set node to the top of the tree.
2. If at a leaf node, stop.
3. Extract a bit position to test.
4. If that bit of the candidate address is on, set the node to the right child of the current node, otherwise set node to the left child.
5. Repeat steps 2 – 4.

Wei, G., Chunhe, X., Nan, L., Haiquan, W., and in, D, in their paper, Research on simulation framework of structured Network, IEEE Computer Society, notes that current evaluations of IP Lookup algorithms are based on algorithmic analysis and experiments that have only been performed with a small number of nodes. Some well known network simulators are Opnet, NS-2 and OMNET++. These simulators perform very well when evaluating network protocols but they do not scale well for IP Lookup algorithm evaluation with a large number of nodes. For instance Omnet++ can't simulate more than

1000 nodes and Narses can simulate up to 600. This is due to the overhead added by the network details.

1.2 Objectives:

1.2.1 General Objective:

This project aims at investigating IP Lookup algorithms and data structures for the longest prefix match operation required for routing IP packets. Specifically, this project aims at investigating and comparing the performance of the widely used Patricia-tree based approaches found in the BSD kernels and the proposed Tree Bitmap algorithm for Cisco routers for IPv6 that appeared in the paper “Tree bitmap: hardware/software IP lookups with incremental updates” by George Varghese, Will Eatherton and Zubin Dittia. Their performance differences are also investigated and an optimized Tree Bitmap Algorithm for efficient Lookup in IPv6 routers is investigated through the following activities.

This project will also aim to create a simulator that will study and evaluate the IP Lookup Algorithms. The simulator will allow testing the algorithms scalability and searching performance in different scenarios.

1.2.2 Specific Objectives:

- a) To design and implement an IP Lookup simulator through the following activities:
 - i) The first goal of this project is to do a thorough study of IP lookup algorithms. At the moment algorithms such as Tree Bitmap Algorithm and BSD Trie are implemented in router hardware.
 - ii) Optimise IP Lookup Algorithm.
 - iii) Design a simulator for testing IP Lookup Algorithms based on predefined parameters.
 - iv) Implement a simulator for testing the Lookup Algorithms.

- b) To test the IP Lookup algorithms through simulations involving experimenting, studying and analyzing the algorithms using the following activities:
 - i) To test memory consumption of IP Lookup Algorithms.

- ii) To test speed of IP Lookup algorithms.
- iii) To test scalability of IP Lookup Algorithms.

1.3 Research Questions

The basic research questions regarding this study are:

- a. What are the different kinds IP Lookup Algorithms?
- b. Which factors influence the IP Lookup in a router and why?
- c. What are the challenges posed by Longest Matching Prefixes in IP address lookup and how are the challenges compounded by the transition to IPv6?
- d. Can a solution to the challenges be developed found for the Longest Matching Prefix problem?
- e. Is the proposed solution a good enough solution for the IPv6 address lookup problem?

1.4 Significance of the Study

The purpose of this research work will address the scientific society about the LMP problem, the impact of IPv6 on IP Address Lookup Algorithms and the implementation of a proposed solution to the problem. It will contribute through the use of large numbers of IP prefixes to analyze the performance of IP Lookup Algorithms; it should reveal the idea, by comparing performances of the selected algorithms, to developing high performance routers for IPv6.

The second purpose of this research work is to come up with a model simulator that can be used in IP Lookup algorithm research. Past studies have shown that network simulators such as OMNET++, NS2 perform very well when evaluating network protocols but they do not scale well for IP Lookup algorithm evaluation with a large number of nodes. For instance Omnet++ can't simulate more than 1000 nodes and Narses can simulate up to 600. This is due to the overhead added by the network details. Creating a simulator for IP lookup algorithm will not just help algorithm developers to improve their algorithms but also researchers in the field of next generation routers.

CHAPTER 2

LITERATURE REVIEW

The primary role of routers is to forward packets toward their final destinations. To this end, a router must decide for each incoming packet where to send it next. More exactly, the forwarding decision consists of finding the address of the next-hop router as well as the egress port through which the packet should be sent. This forwarding information is stored in a forwarding table that the router computes based on the information gathered by routing protocols. To consult the forwarding table, the router uses the packet's destination address as a key; this operation is called address lookup. Once the forwarding information is retrieved, the router can transfer the packet from the incoming link to the appropriate outgoing link, in a process called switching.

The exponential growth of the Internet has stressed its routing system. While the data rates of links have kept pace with the increasing traffic, it has been difficult for the packet processing capacity of routers to keep up with these increased data rates. Specifically, the address lookup operation is a major bottleneck in the forwarding performance of today's routers. This project presents a survey of the latest algorithms for efficient IP address lookup and the optimization. I start by tracing the evolution of the IP addressing architecture. The addressing architecture is of fundamental importance to the routing architecture, and reviewing it will help in understanding the address lookup problem:

Routing Processor

When a packet arrives at a line card, its header is removed and passed to the routing processor. The remainder of the packet remains on the inbound line card. Once the header reaches the routing processor, it is placed in a request first-in first-out (FIFO) queue for processing. The processor reads the header and looks up a routing table to determine how to forward the packet, then makes one or more writes to inform the inbound line card

how to handle the packet. Hence, the routing processor is also called forwarding engine or network processor in the literature.

The main function of a router is to perform route lookup: that is, given a packet with an IP destination address, the router must determine the appropriate output port for this packet. The process of table look-up involves a longest prefix match of the variable destination network address contained in the packet header against multiple entries in the routing table. The one selected contains the most bits that match up with the destination address.

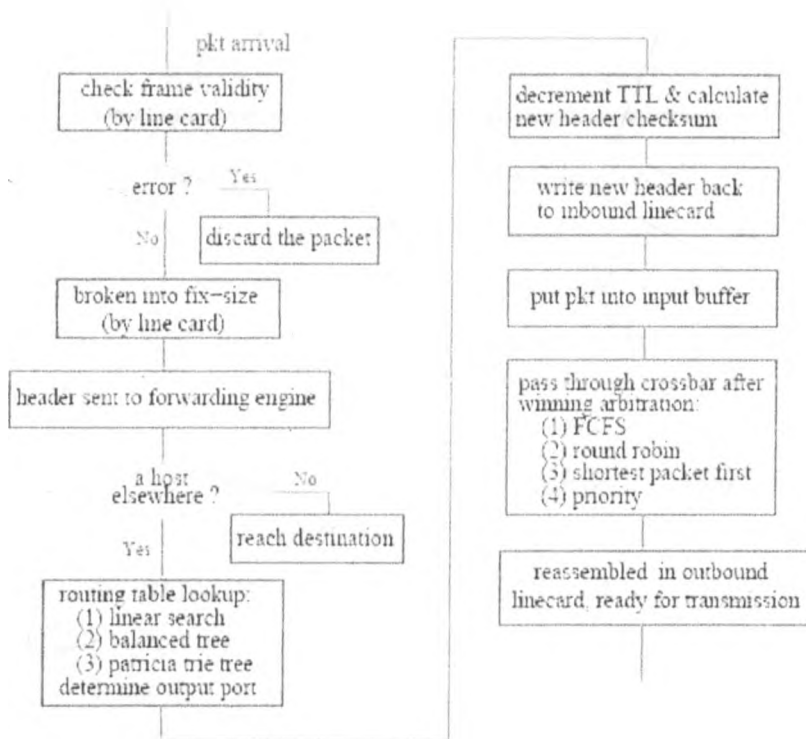


Figure 2.1: Routing Process

2.1 The Classful Internet Addressing:

In IPv4, IP addresses are 32 bits long and when broken up into 4 groups of 8 bits, are normally represented as four decimal numbers separated by dots. For example, the

Address 10000010 01010110 00010000 01000010
Corresponds to 130.86.16.66.

The fundamental objective of Internet Protocol is to interconnect networks, so routing on a network basis was a natural choice rather than routing on a host basis. Thus, the IP address scheme initially used a simple two-level hierarchy, with networks at the top level and hosts at the bottom level.

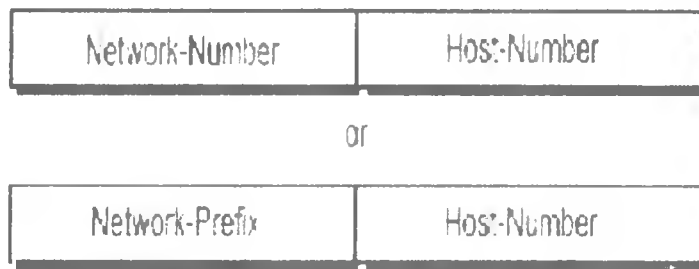


Figure 2.2: Internet Two Level hierarchy

This hierarchy is reflected in the fact that an IP address consists of two parts, a network part and a host part. The network part identifies the network to which a host is attached, and thus all hosts attached to the same network agree in the network part of their IP addresses.

Since the network part corresponds to the first bits of the IP address, it is called the address prefix. Prefixes will be written as bit strings of up to 32 bits in IPv4 followed by a *. For example, the prefix 1000001001010110* represents all the addresses that begin with the bit pattern 1000001001010110. Prefixes can also be indicated using the dotted-decimal notation, so the same prefix can be written as 130.86/16, where the number after the slash indicates the length of the prefix.

With a two-level hierarchy, IP routers forwarded packets based only on the network part, until packets reached the destination network. As a result, a forwarding table only needed to store a single entry to forward packets to all the hosts attached to the same network. This technique is called address aggregation and allows using prefixes to represent a group of addresses. Each entry in a forwarding table contains a prefix, as can be seen in Table 2.1.

| Destination Address Prefix | Next-hop | Output interface |
|----------------------------|----------------|------------------|
| 24.40.32/20 | 192.41.177.148 | 2 |
| 130.86/16 | 192.41.177.181 | 6 |
| 208.12.16/20 | 192.41.177.241 | 4 |
| 208.12.21/24 | 192.41.177.196 | 1 |
| 167.24.103/24 | 192.41.177.3 | 1 |

Table 2.1: Forwarding table.

Finding the forwarding information now requires searching for the prefix in the forwarding table that matches the corresponding bits of the destination address.

The addressing architecture specifies how the allocation of addresses is performed; that is, it defines how to partition the total IP address space of 2^{32} for IPv4 and 2^{128} for IPv6 addresses - specifically, how many network addresses will be allowed and what size each of them should be. When Internet addressing was initially designed, a rather simple address allocation scheme was defined, which is known as classful addressing scheme. Basically, three different sizes of networks were defined in this scheme, identified by a class name: A, B, or C.

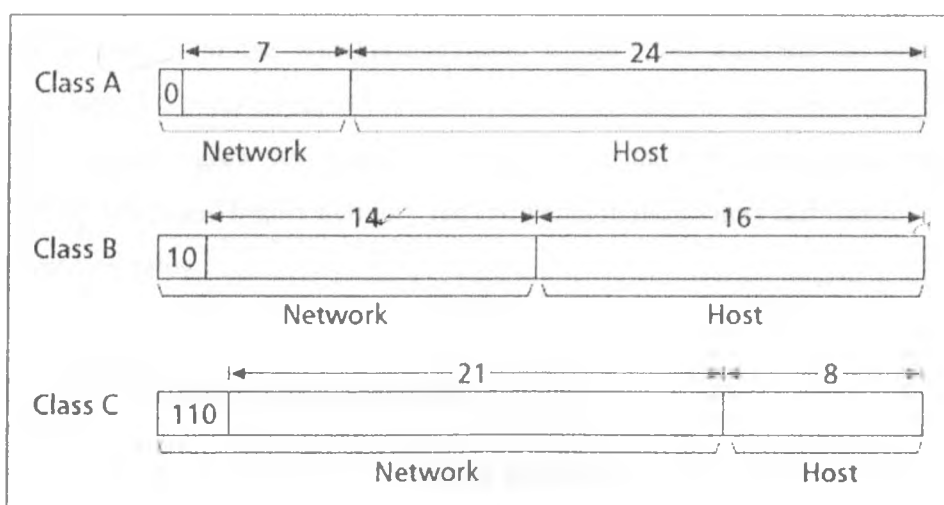


Figure 2.3 Classful addresses.

Network size was determined by the number of bits used to represent the network and host parts. Thus, networks of class A, B, or C consisted of an 8, 16, or 24-bit network part and a corresponding 24, 16, or 8-bit host part.

With this scheme there were very few class A networks and their addressing space represented 50 percent of the total IPv4 address space (2^{31} addresses out of a total of 2^{32}). There were 16,384 (2^{14}) class B networks with a maximum of 65,534 hosts/network and 2,097,152 (2^{21}) class C networks with up to 256 hosts. This allocation scheme worked well in the early days of the Internet. However, the continuous growth of the number of hosts and networks has made apparent three problems with the classful addressing architecture. First, with only three different network sizes from which to choose the address space was not used efficiently and the IP address space was getting exhausted very rapidly even though only a small fraction of the addresses allocated were actually in use. Attempts to reduce the inefficient address space allocation leads to even more router table entries.

Second the lack of internal address flexibility ensured that big organizations are assigned large, “monolithic” blocks of addresses that don't match well the structure of their underlying internal networks.

Third, although the information stored in the forwarding tables did not grow in proportion to the number of hosts it still grew in proportion to the number of networks. This was especially important in the backbone routers, which must maintain an entry in the forwarding table for every allocated network address. As a result, the forwarding tables in the backbone routers grew very rapidly. The growth of the forwarding tables resulted in higher lookup times and higher memory requirements in the routers and thereby impacted their forwarding rates.

2.2 The CIDR Addressing Scheme

To allow more efficient use of the IP address space and to slow down the growth of the backbone forwarding tables, a new scheme called classless interdomain routing (CIDR)

was introduced. Remember that in the classful address scheme, only three different prefix lengths are allowed: 8, 16, and 24, corresponding to classes A, B and C, respectively. CIDR uses the IP address space more efficiently by allowing finer granularity in the prefix lengths. With CIDR, prefixes can be of arbitrary length rather than constraining them to be 8, 16, or 24 bits long.

To address the problem of forwarding table explosion, CIDR allows address aggregation at several levels. The idea is that since the allocation of addresses has a topological significance, then addresses can be recursively aggregated at various points within the hierarchy of the Internet's topology. As a result, backbone routers maintain forwarding information not at the network level, but at the level of arbitrary aggregates of networks. Thus, recursive address aggregation reduces the number of entries in the forwarding table of backbone routers.

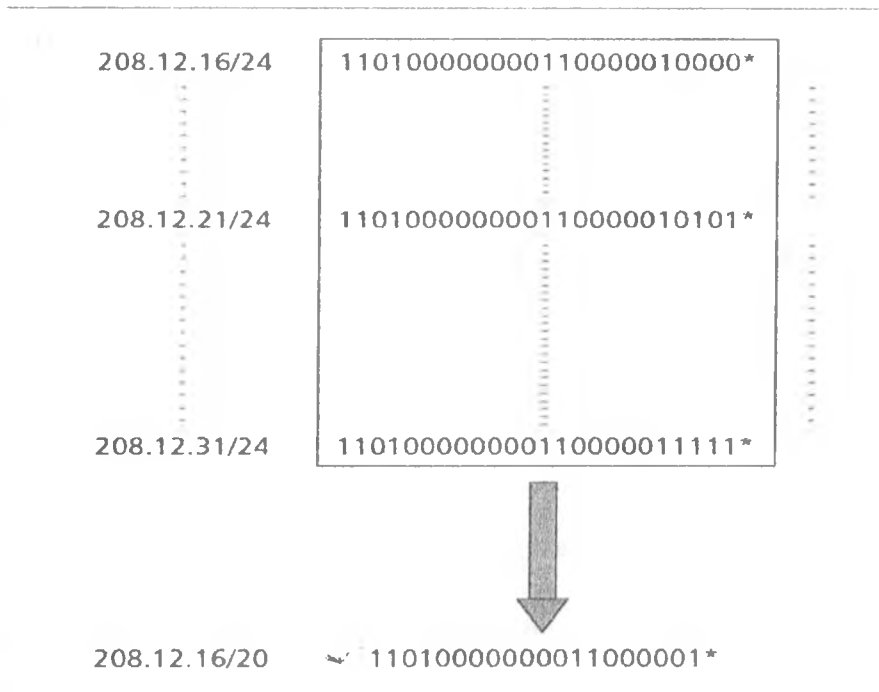


Table 2.2: Address Aggregations:

As an example on how address aggregation works, the networks represented by the network numbers from 208.12.16/24 through 208.12.31/24 are considered. Suppose that in a router all these network addresses are reachable through the same service provider.

From the binary representation we can see that the leftmost 20 bits of all the addresses in this range are the same (11010000 00001100 0001). Thus, we can aggregate these 16 networks into one “supernetwork” represented by the 20-bit prefix, which in decimal notation gives 208.12.16/20. Indicating the prefix length is necessary in decimal notation, because the same value may be associated with prefixes of different lengths; for instance,

208.12.16/20 (11010000 00001100 0001*) is different from
 208.12.16/22 (11010000 00001100 000100*).

While a great deal of aggregation can be achieved if addresses are carefully assigned, in some situations a few networks can interfere with the process of aggregation. For example, if a customer who owns the network 192.2.3/24 changes his service provider and does not want to renumber his network.

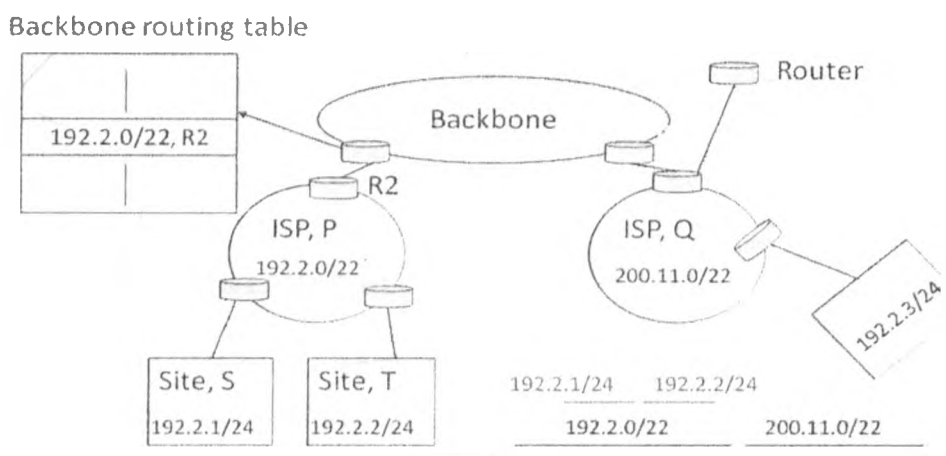


Figure 2.4: Address Aggregations:

Now, all the networks from 192.2.1/24 through 192.2.2/24 can be reached through the same service provider, except for the network 192.2.3/24. Aggregation can now not be performed as before and instead of only one entry, additional entries need to be stored in the forwarding table. One solution that can be used in this situation is aggregating in spite of the exception networks and additionally storing entries for the exception networks. In this example, this will result in only two entries in the forwarding table: 192.2.0/22 and 192.2.3/24.

Some addresses will match both entries because prefixes overlap. In order to always make the correct forwarding decision, routers need to do more than to search for a prefix that matches. Since exceptions in the aggregations may exist, a router must find the most specific match, which is the longest matching prefix.

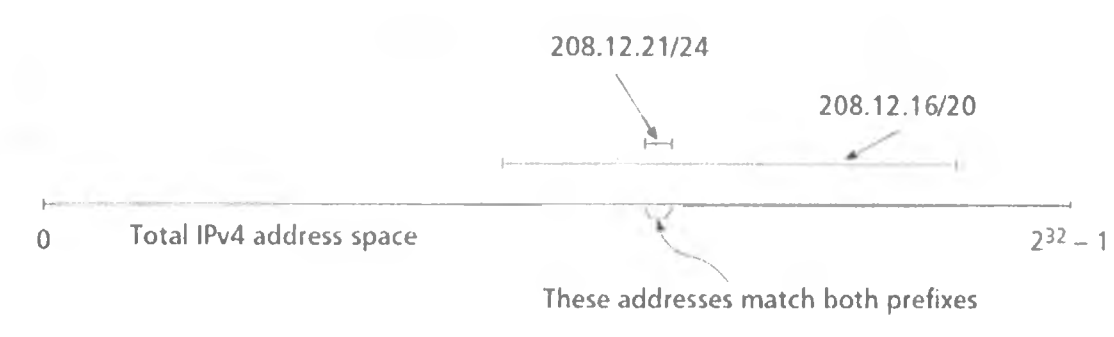


Figure 2.5: An exception Prefix.

In summary, the address lookup problem in routers requires searching the forwarding table for the longest prefix that matches the destination address of a packet.

2.3 IPv6 Address architecture

IPv6 addresses are 128 bits in length and as such IPv6 addresses are very long. The address includes 3 parts: 45-bit global routing prefix (the first three bits should always be '001'), 16-bit subnet ID, and 64-bit interface ID. The global routing prefix identifies a site, the subnet ID identifies a subnet in a specific site, and the interface ID specifies a network interface in a subnet. Usually, only the global routing prefix and the subnet ID, accounting for 64 bits, are used for routing. Thus, routing entries with prefix lengths longer than 64 bits are seldom in the IPv6 backbone BGP routing tables.

2.3.1 Allocation policies for IPv6 address

Responsibility for management of IPv6 address spaces is distributed globally in accordance with the hierarchical structure shown below. The top level of the hierarchy is Internet Assigned Numbers Authority (IANA), which allocates global unicast IPv6 addresses of /23 to Regional Internet Registries (RIRs). RIRs in turn allocate addresses of /32 to subordinate address agencies, ISPs (Internet Service Providers) or LIRs (Local Internet Registries). EUs (End Users) will generally be given /48, sometimes /64 or /128 assignments according to their requirements and scales.

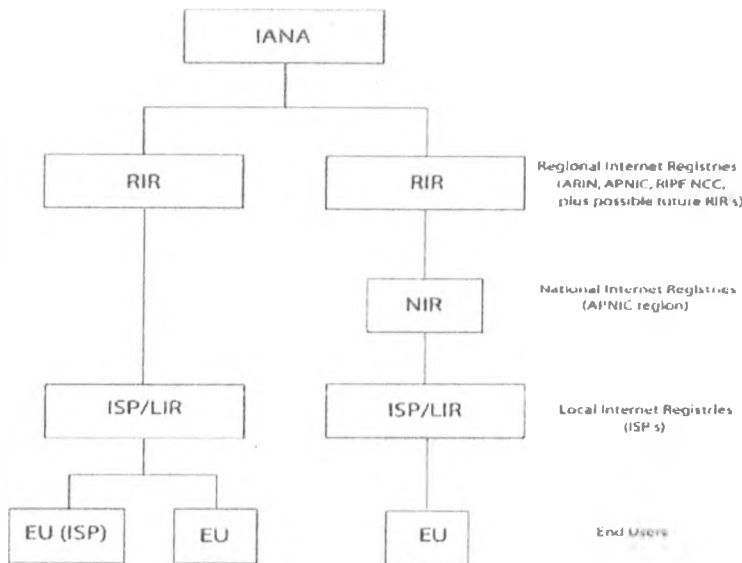


Figure 2.6: Allocation Policy for IPv6.

Moreover, some authorities have established the regulations to guide IPv6 address allocation, such as IAB/IESG recommendations on IPv6 address allocation and the IPv6 address allocation and assignment policy issued jointly by RIRs. This makes IPv6 routing tables to present clear hierarchies. Different hierarchies have different characteristics and thus are suitable for different data structures on which my optimized tree bitmap algorithm is rooted.

2.4 Difficulty of the Longest Matching Prefix Search

In the classful addressing architecture, the length of the prefixes was coded in the most significant bits of an IP address and the address lookup was a relatively simple operation: Prefixes in the forwarding table were organized in three separate tables, one for each of the three allowed lengths. The IP lookup operation amounted to finding an exact prefix match in the appropriate table. The search for an exact match could be performed using standard algorithms based on hashing or binary search.

While CIDR allows the size of the forwarding tables to be reduced, the address lookup problem now becomes more complex. With CIDR, the destination prefixes in the forwarding tables have arbitrary lengths and no longer correspond to the network part since they are the result of an arbitrary number of network aggregations. Therefore, when using CIDR, the search in a forwarding table can no longer be performed by exact matching because the length of the prefix cannot be derived from the address itself. As a result, determining the longest matching prefix involves not only comparing the bit pattern itself, but also finding the appropriate length. Therefore, searching is done in two dimensions: value and length.

The IP address lookup problem can be defined formally as follows:

Let $P = \{P_1, P_2, \dots, P_N\}$ be a set of routing prefixes, where N is the number of prefixes.

Let A be an incoming IP address and $S(A, k)$ be a sub-string of the most significant k bits of A .

Let $n(P_i)$ be the length of a prefix P_i .

A is defined to match P_i if $S(A, n(P_i)) = P_i$.

Let $M(A)$ be the set of prefixes in P that A matches, then $M(A) = \{P_i \in P : S(A, n(P_i)) = P_i\}$.

The longest prefix matching problem is to find the prefix P_j in $M(A)$, such that $n(P_j) > n(P_i)$ for all $P_i \in M(A)$, $i \neq j$. Once the longest matching prefix P_j is determined, the input packet is forwarded to an output port directed by the prefix P_j .

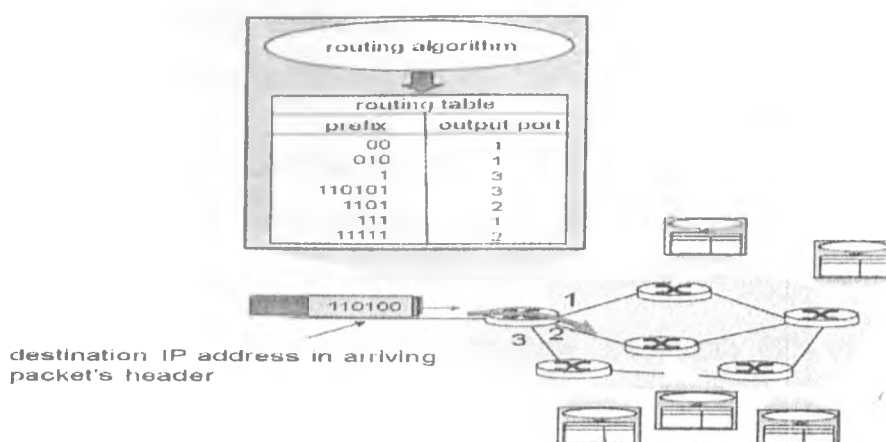


Figure 2.7: Example network with 6-bit IP addresses

Figure shows an example network that has a 6-bit address space (IP address in IPv4 is 32 bits) as an example. Each router obtains a routing table composed of a set of prefixes and the corresponding output port by running routing algorithms. For the example set of prefixes as shown, by searching the routing table for an arbitrary input address, 110100, we obtain $M(A) = \{1^*, 1101^*\}$. Of those two matching prefixes, prefix 1101* is identified as the longest matching prefix or the best matching prefix (BMP); it represents the most specific network that the input has to be forwarded. Hence, the input packet is forwarded toward the network through output port 2.

Since IP Lookup is performed in two dimensions, the search methods I review try to reduce the search space at each step in both of these dimensions.

2.5 Classifications of IP Lookup Algorithms

In this project, I describe various IP address lookup algorithms and compare the characteristics. Miguel Á. Ruiz-Sánchez et al, in their paper, Survey and Taxonomy of IP Address Lookup Algorithms, published a survey on IP address lookup algorithms. However, several algorithms that are more interesting have been proposed since their publication. My approach differs from the approach that was used in that I use a consistent example set to describe the data structure and the search procedure of each algorithm so that each algorithm can be easily understood, compared, and practically

implemented. The evaluation method also differs. While algorithms were evaluated based on Ipv4 addresses only, this project includes both Ipv4 and the projected Ipv6 addresses.

The design of a “good” algorithm requires an algorithm designer to understand the requirements of the problem and how these requirements are expected to evolve. Below listed are the basic requirements for the longest prefix matching:

- **Lookup Speed:** Internet traffic measurements show that roughly 50% of the packets that arrive at a router are TCP-acknowledgment packets, which are typically 40-byte packets. As a result, a router can be expected to receive a steady stream of such minimum size packets. Thus, the prefix lookup has to happen in the time it takes to forward a minimum-size packet (40 bytes), known as wire speed forwarding.
- Similarly, the lookup cannot exceed the budget time of 32 nanosec at 10 Gbps and 8 nanosec at 40 Gbps. The main bottleneck in achieving such high lookup speed is the cost of memory access. Thus, the lookup speed is measured in terms of the number of memory accesses.
- **Memory Usage:** The amount of memory consumed by the data structures of the algorithm is also important. Ideally, it should occupy as little memory as possible. A memory-efficient algorithm can effectively use the fast but small cache memory if implemented in software.
- **Scalability:** The algorithms are expected to scale both in speed and memory as the size of the IP address length and forwarding table increases. While core routers presently contain hundreds of thousands prefixes, it is expected to increase. When routers are deployed in the real network, the service providers expect them to provide consistent and predictable performance despite the increase address lengths and forwarding table size. This is expected since a router needs to have a useful lifetime of at least five years to recuperate the return on investment.
- **Updatability:** It has been observed in practice that the route changes occur fairly frequently. Studies show that core routers may receive bursts of these changes at rates varying from a few prefixes per second to a few hundred prefixes per second. Thus, the route changes require updating the forwarding table data structure, in the order of

milliseconds or less. These requirements are still several orders of magnitude less than the lookup speed requirements. Nonetheless, it is important for an algorithm to support incremental updates.

To summarize, the important requirements of a lookup algorithm are speed, storage, update time, and scalability. Ideally, algorithms are required to perform well in the worst case.

Following is an analysis of the different approaches with respect to lookup time, memory utilization and scalability. The existing approaches have been classified into Trie based approaches. Trie is a general-purpose data structure for storing strings. Each prefix in the routing table is represented by a leaf node in the trie. A trie has labeled branches that is traversed during a search operation using individual bits of the search key. The left branch of a node is labeled 0 and the right-branch is labeled 1. The longest prefix search operation on a given destination address starts from the root node of the trie. The remaining bits of the address determine the path of traversal in a similar manner.

2.5.1 Binary Tries

A binary trie is a tree-based data structure allowing the organization of prefixes on a digital basis using the bits of prefixes to direct the branching. Each node has at most two children in a binary trie. Each prefix maps to a node in the binary trie of which the path and the level are determined by the prefix value and the length, respectively.

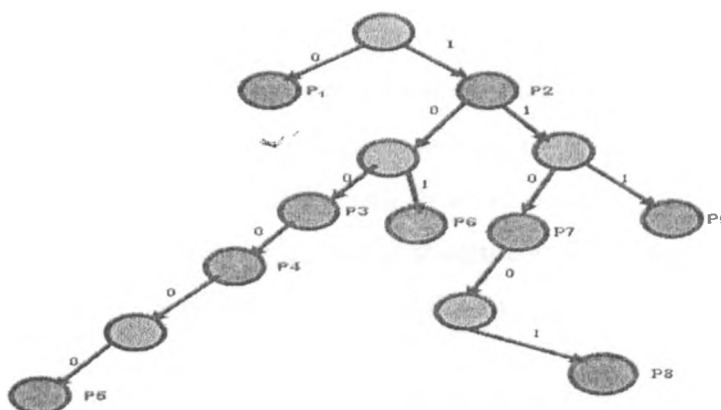


Figure 2.8: A Binary Trie

Figure shows the binary trie for the example set of prefixes, $P = \{P_1(0^*), P_2(1^*), P_3(100^*), P_4(1000^*), P_5(100000^*), P_6(101^*), P_7(110^*), P_8(11001^*), P_9(111^*)\}$. From the root node, the left edge corresponds to a bit value of 0 and the right edge corresponds to a bit value of 1. The depth of the trie is determined by the longest prefix existing in the routing data. As shown, empty nodes that are not associated with any prefix are included in the paths going to prefix nodes (red nodes).

| Next Hop | Prefix | Port Number |
|----------------|---------|-------------|
| P ₁ | 0* | 1 |
| P ₂ | 1* | 2 |
| P ₃ | 100* | 1 |
| P ₄ | 1000* | 1 |
| P ₅ | 100000* | 4 |
| P ₆ | 101* | 3 |
| P ₇ | 110* | 2 |
| P ₈ | 11001* | 1 |
| P ₉ | 111* | 4 |

Table 2.3: Prefix Table

Table 2.3 shows the routing table implementing the data structure of the binary trie. Entries in the routing table have one-to-one correspondence to a node in the binary trie, and the entry address is represented by a red-colored number in each node. The first entry is the root node. Fields of the routing table include a valid bit to distinguish prefix nodes from empty nodes and two memory addresses pointing to its children. It also has a field for an output port used in the case of a match. It is not necessary to store the value and the length in the routing table entries in the trie structure since the value and the length of each prefix are known by the path and the level of the prefix node from the root node.

Search in the binary trie proceeds to a lower level, by examining a bit of the input address at a time. If it is 0, the search proceeds to a left child and otherwise proceeds to a right

child, until it reaches where there is no pointer to follow. While going down the trie, the search process keeps track of the current best matching prefix (BMP), whenever a prefix node is encountered. When the search is over, the currently remembered BMP is returned. Assuming that a 6-bit input IP address 1100000001 is given, the search passes through various entries. The current BMP was the prefix P_2 at entry 1. It is replaced with prefix P_7 at entry 4 and the output port of prefix P_7 is returned when the search is complete.

The search space is reduced by half in a binary trie by accessing a memory entry. Hence, the binary trie provides better search performance than the linear search, which reduces the search space by a single entry every time a memory entry is accessed. However, for the migration to IPv6 that has 128-bit address space, the depth of the binary trie becomes excessive and the search speed will become a major issue.

2.5.2 Patricia Trie

This is the most commonly available IP lookup implementation found in the BSD kernels. The PATRICIA stores the prefix entries in a Trie data structure that is optimized for storage and retrieval. Since the IP addresses are binary, the trie data structure has an alphabet size of 2 with the alphabet set as $\{0, 1\}$. The IP address is then processed bit by bit to produce the best match.

Assume the objective is to reduce the search time and reduce the memory space; what can we do about it? One possibility is not to involve any of the bits corresponding to one child nodes during inspection. If they do not need to be inspected, then we can eliminate them as well. By collapsing the one-way branch nodes, path compression improves search time and consumes less memory space. However, additional information needs to be maintained in other nodes so that a search operation can be performed correctly.

While binary tries allow the representation of arbitrary length prefixes they have the characteristic that long sequences of one-child nodes may exist. For example, P_5 contains a long sequence of child nodes. Since these bits need to be inspected, even though no

actual branching decision is made, search time can be longer than necessary for some cases.

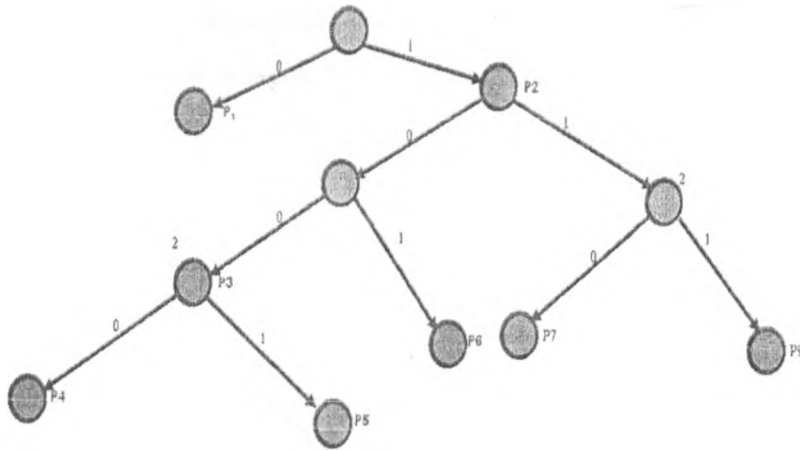


Figure 2.9: Path Compressed Trie

Also, one-child nodes consume additional memory. In an attempt to improve time and space performance, a technique called path-compression was used. Path-compression consists in collapsing one-way branch nodes. When one-way branch nodes are removed from a trie, additional information must be kept in remaining nodes, so that search operation can be performed correctly.

There are many ways to exploit the path-compression technique; perhaps the simplest to explain is illustrated below corresponding to the binary trie in figure above.

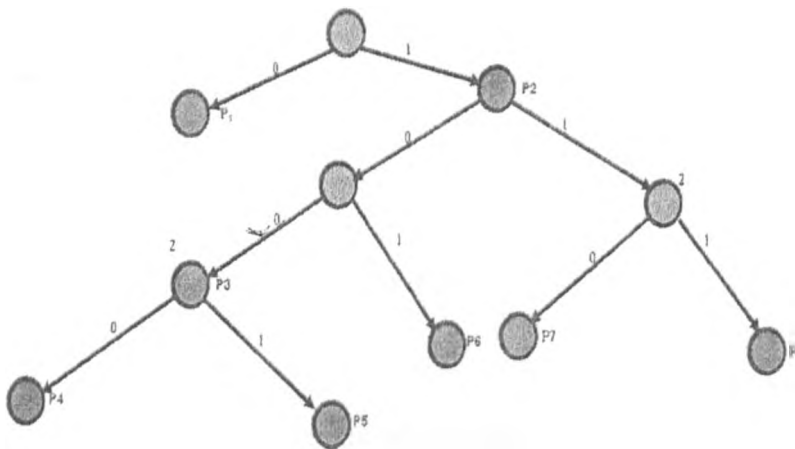


Figure 2.10: Patricia Trie

The two nodes preceding P5 now have been removed and since prefix P5 was located at a one-child node, it is moved to the nearest descendant not being a one-child node. Since in a path to be compressed several one-child nodes may contain prefixes, in general, a list of prefixes must be maintained in some of the nodes. Because one-way branch nodes are now removed, it is possible to jump directly to the bit where a significant decision is to be made, bypassing the bit inspection of some bits. As a result, a bit number field must be kept now to indicate which bit is the next bit to inspect.

A search in this kind of path-compressed tries is as follows: The algorithm performs, as usual, a descent in the trie under the guidance of the address bits; but this time, only inspecting bit positions indicated by the bit-number field in the nodes traversed. When a node marked as prefix is encountered, a comparison with the actual prefix value is performed. This is necessary since during the descent in the trie some bits may be skipped. If a match is found, the trie is traversed while the prefix found as the BMP is kept as the BMP so far. Search ends when a leaf is encountered or a mismatch is found. As usual the BMP will be the last matching prefix encountered.

For instance, a search for the BMP of an address beginning with the bit pattern 010110 in the path compressed trie shown above proceeds as follows:

Step 1

Start at the root node and since its bit number is 1 inspect the first bit of the address. The first bit is 0 so we go to the left.

Step 2

Since the node is marked as prefix compare the prefix P1 with the corresponding part of the address 0.

Since they match proceed and keep P1 as the BMP so far. Since there are no more nodes to traverse, search stops and the last remembered BMP (prefix a) is the correct BMP.

Path-compression makes much sense when the binary trie is sparsely populated. But when the number of prefixes increases and the trie gets denser, using path compression has little benefit. Moreover, the principal disadvantage of path-compressed tries, as well as binary tries in general, is that a search needs to do many memory accesses, in the worst case 32 for IPv4 addresses.

2.5.3 Multibit Trie

While binary tries can handle prefixes of arbitrary length easily, the search can be very slow since bits are examined one at a time. In the worst case, it requires 32 memory accesses for the 32-bit IPv4 address. If the cost of a memory access is 10 nanosecond, the lookup will consume 320 nanosecond. This translates to a maximum forwarding speed of 3.125 million packets per second ($1/320$ nanosecond). At 40 bytes per packet, this can support an aggregate link speed of at most 1 Gbps. However, the increase in Internet traffic requires supporting aggregate link speeds as high as 40 Gbps. Clearly, sustaining such a high rate is not feasible with binary trie-based structures.

After closely examining the binary trie, we can ask: why restrict ourselves to only one bit at a time? Instead, examine multiple bits so that we can speed up the search by reducing the number of memory access. For instance, if we inspect 4 bits at a time, the search will finish in 8 memory accesses as compared to 32 memory accesses in a binary trie. This is the basic idea behind the multibit trie. The number of bits to be inspected per step is called a stride. Strides can be either fixed-size or variable-size. A multibit trie is a trie structure that allows the inspection of bits in strides of several bits. Each node in the multibit trie has 2^k children where k is the stride. If all the nodes at the same level have the same stride size, it is called a fixed stride; otherwise, it is a variable stride.

Since multibit tries allow the data structure to be traversed in strides of several bits at a time, they cannot support prefixes of arbitrary lengths. To use a given multibit trie, a prefix must be transformed into an equivalent prefix of longer length to conform with the prefix lengths allowed by the structure.

An IP prefix associated with the next-hop information can be expressed as an equivalent set of prefixes with the same next-hop information after a series of transformations. One of the most common prefix transformation techniques is prefix expansion. A prefix is said to be expanded if it is converted into several longer and more specific prefixes that cover the same range of addresses. For instance, the range of addresses covered by prefix 0* can be also specified with the two prefixes 00* and 01*, or with the four prefixes 000*, 001*, 010*, and 011*. An appropriate prefix expansion transforms varied prefix lengths into a set of prefixes that have fewer different lengths.

| | |
|-----|---------|
| P1 | 0* |
| P2 | 1* |
| P3 | 100* |
| P4 | 1000* |
| P5 | 100000* |
| P6 | 101* |
| P7 | 110* |
| P8 | 11001* |
| P9 | 111* |
| P10 | 1001* |

Table 2.4: Original Forwarding Table Prefixes

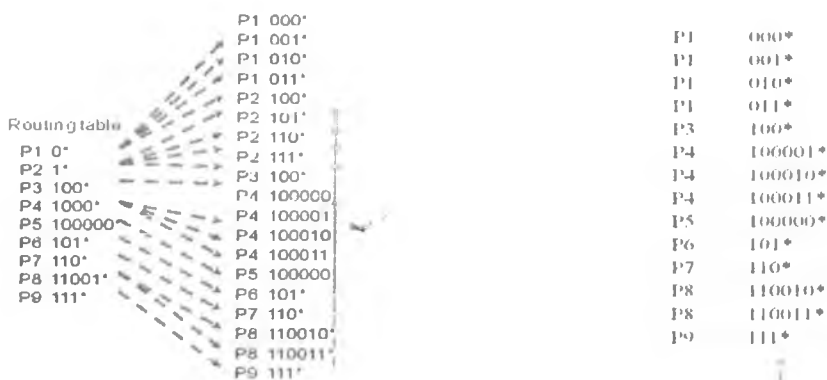


Table 2.5: Forwarding Table Prefixes after expansions

A fixed stride trie is typically implemented using arrays for each trie node and linking those using pointers. The trie nodes at different levels will have different array sizes as determined by the stride at that level. If the stride at a level is k , then the size of the array required will be 2^k . Each entry in the array consists of two fields: the field *nhop* contains the next-hop information and the field *ptr* contains the pointer to the subtrie, if any. The presence of prefix information in an element indicates that the field *nhop* is not empty and stores the next-hop information associated with that prefix. The arrows indicate that the field *ptr* is not empty and point to the subtrie.

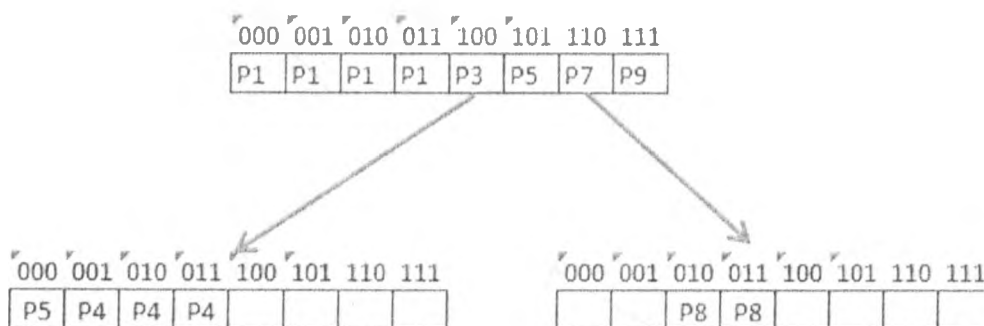


Figure 2.11: Multibit Node

The search proceeds by breaking up the destination address into chunks that correspond to the strides at each level. Then these chunks are used to follow a path through the trie until there are no more branches to take. Each time a prefix is found at a node, it is remembered as the new best matching prefix seen so far. At the end, the last best matching prefix found is the correct one for the given address.

Consider searching for the best matching prefix for the address 1000110 in the fixed stride trie shown in Figure 2.9. First, the address is divided into multiple chunks: a chunk made of the first 3 bits, 100, corresponds to level 1 of the trie; another chunk made of the next three bits, 111, corresponds to level 2 of the trie, and the last incomplete chunk consists of the remaining bits. Using the first chunk 100 at the root node leads to the prefix $P3$ that is noted as the best matching prefix. Next, using the second chunk of 111 leads to the prefix $P4$, which is updated to be the best matching prefix so far. Since the

search cannot proceed further, the final answer is P_4 . The number of memory accesses required is 2 instead of 6 when using a binary trie for the same search.

2.5.4 Tree Bitmap

Tree Bitmap is a multibit trie algorithm that allows fast searches and allows faster update and fewer memory storage requirements. A multibit node has two functions: to point at children multibit nodes, and to produce the next-hop pointer for searches in which the longest matching prefix exists within the multibit node.

A closer look at figure 2.9 reveals that some space is wasted in the leftmost array in the second level that contains only prefix P_5 and P_4 ; the rest of the four elements do not contain any information. This presents an opportunity for improvement of the algorithm. Some form of compression may be employed to minimize the wastage on storage. Using large strides in fixed stride multibit tries results in a greater number of contiguous nodes with the same best matching prefix and next-hop information. The tree bitmap algorithm takes advantage of this fact and compresses the redundant information using bitmaps, thereby reducing storage and still not incurring a high penalty in the search time.

Tree Bitmap algorithm is based on four key ideas. The first idea in the algorithm is that all child nodes of a given trie node are stored contiguously. This allows the use of just one pointer for all children (the pointer points to the start of the child node block) because each child node can be calculated as an offset from the single pointer. This can reduce the number of required pointers by a factor of two compared with standard multibit tries. More importantly it cuts down the size of trie nodes. Using this idea, the same 3-bit stride trie of Figure 2.9 is redrawn as Figure 2.10.

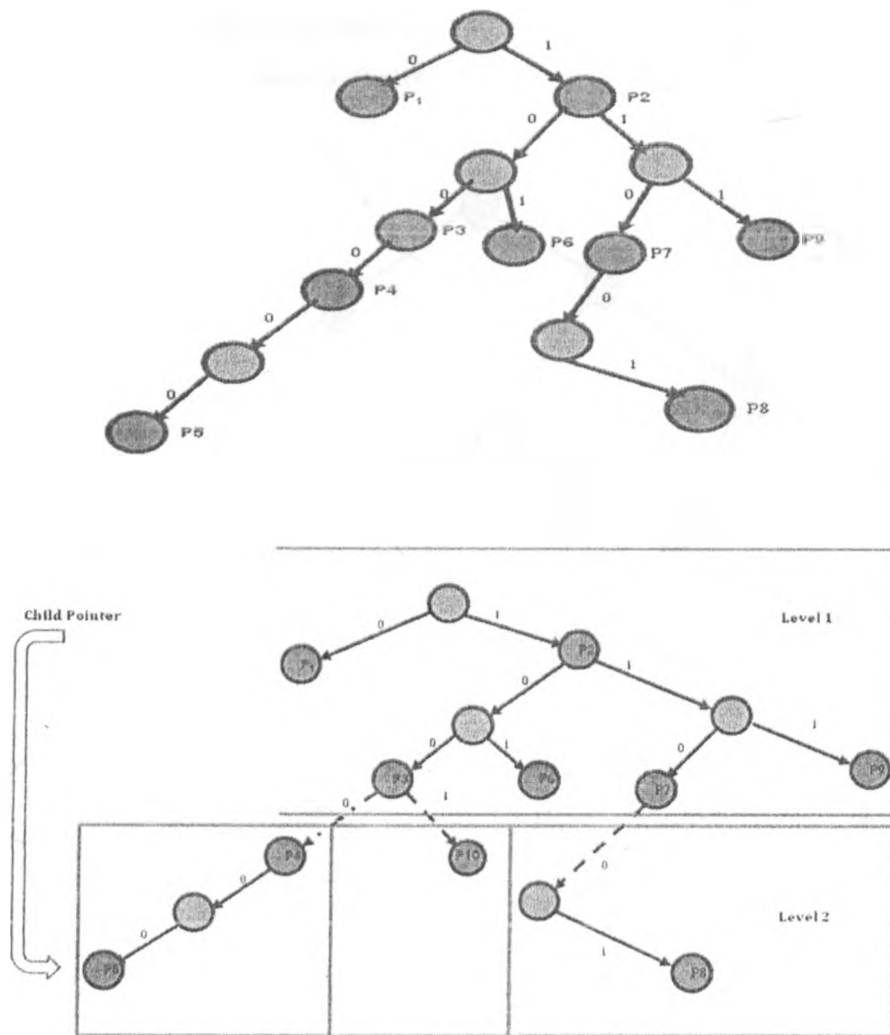


Figure 2.12: Multibit Node with Pointers

The second idea is that there are two bit maps per trie node, one for all the internally stored prefixes and one for the external pointers. Unlike in other schemes such as Lulea that implements leaf pushing, two bit maps are used to avoid leaf pushing. The internal bit map has a 1 bit set for every prefixes stored within this node. Thus for an r bit trie node, there are $2^{(r-1)}$ possible prefixes of lengths $< r$ and thus a 2^r-1 bit map is used. For the root trie node of Figure 2.10, there are six internally stored prefixes: P1, P2, P3, P6, P7 and P9. Suppose our internal bit map has one bit for prefixes of length 0, two following bits for prefixes of length 1, 4 following bits for prefixes of length 2 etc. Then

for 3 bits the root internal bit map becomes 1011000. The first 1 corresponds to P1, the second to P2, the third to P3. This is shown in Figure

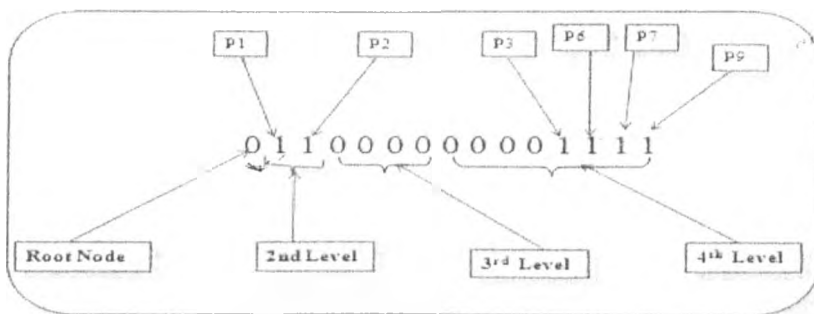
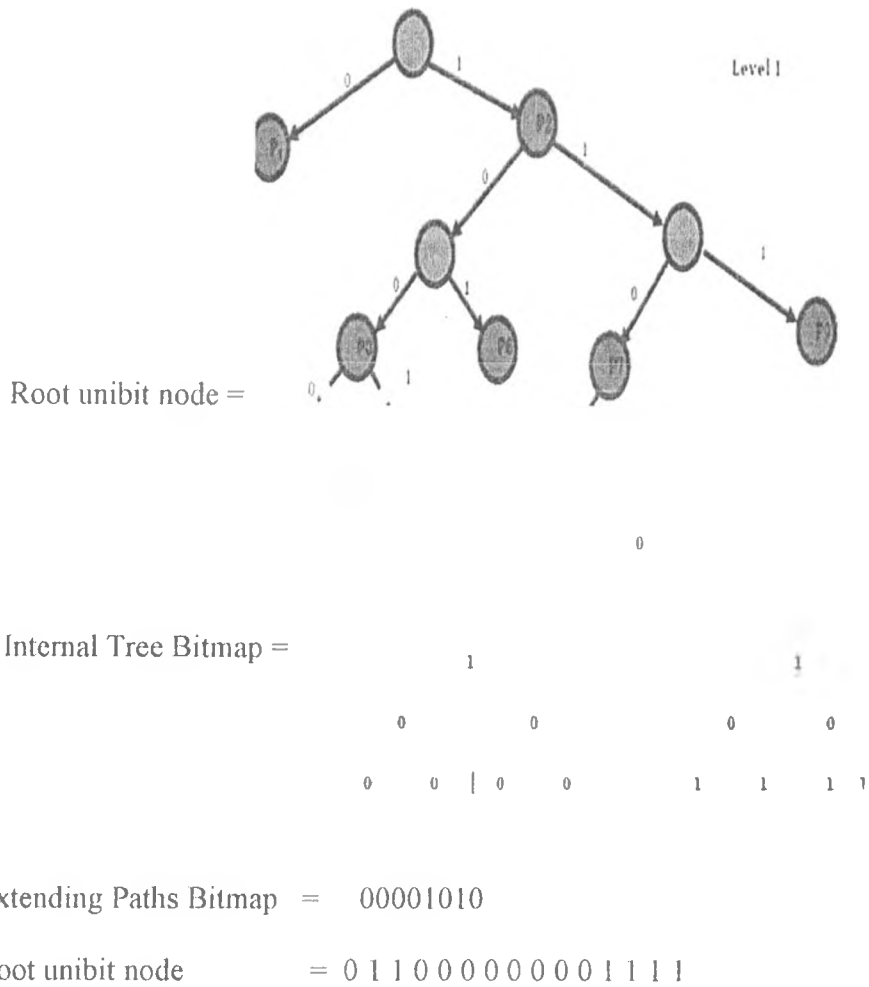


Figure 2.13: Root Unibit Node with Bitmaps

The external bit map contains a bit for all possible 2^r child pointers. Thus in Figure 2.10, we have 8 possible leaves of the 3 bit sub trie. Only the fifth and seventh leaves have pointers to children. Thus the extending paths (or external) bit map shown in Figure 2.11 is 00011001.

The third idea is to keep the trie nodes as small as possible to reduce the required memory access size for a given stride. Thus a trie node is of fixed size and only contains an external pointer bit map, an internal next hop information bit map, and a single pointer to the block of child nodes. The next hops information associated with the internal prefixes is stored within each trie node in a separate array associated with this trie node. Putting next hop pointers in a separate result array potentially requires two memory accesses per trie node (one for the trie node and one to fetch the result node for stored prefixes). The result of a search is not accessed till the search terminates. After the search terminates, the result node corresponding to the last trie node encountered in the path that contained a valid prefix is accessed. This way only a single memory reference at the end is added besides the one memory reference required per trie node.

The tree bitmap algorithm achieves fast search and update by storing two bitmaps: one for pointers to the child and the other for prefixes. The tree bitmap algorithm design considers that a multibit trie node is intended to serve two purposes—one to direct the search to its child nodes and the other to retrieve the forwarding information corresponding to the best matching prefix that exists in the node. It further emphasizes that these functions are distinct from each other. Furthermore, the tree bitmap attempts to reduce the number of child node pointers by storing all the child nodes of a given trie node contiguously. As a result, only one pointer that points to the beginning of this child node block needs to be stored in the trie node. Such an optimization potentially reduces the number of required pointers by a factor of two compared to standard multibit tries. An additional advantage is that it reduces the size of the trie nodes. In such a scheme, the address for any child node can be computed efficiently using simple arithmetic, assuming a fixed size for each child node. The tree bitmap algorithm attempts to keep the trie nodes as small as possible to reduce the size of a memory access for a given stride.

A tree bitmap trie node contains the pointer bitmap, the prefix bitmap, the base pointer to the child block, and the next-hop information associated with the prefixes in the node. If the next-hop information is stored along with the trie node, it would make the size of the trie node much larger. Instead, the next-hop information is stored separately in an array associated with this node and a pointer to the first element is stored in the trie node. The algorithm does not fetch the resulting next-hop information until the search is terminated. Once the search ends, the desired node is fetched. This node carries the next-hop information corresponding to a valid prefix present in the last trie node encountered in the path.

A sample tree bitmap node is shown below. The node consists of two bitmaps.

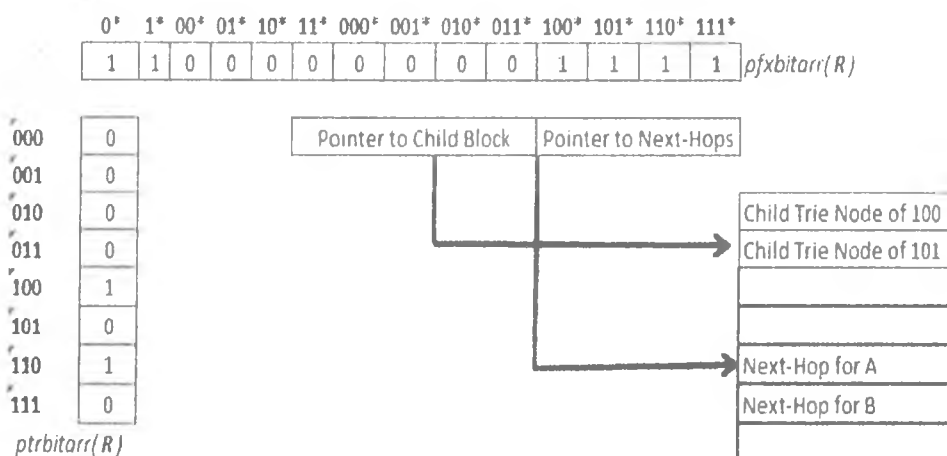


Figure 2.14: Tree Bitmap Node

The first bitmap shown vertically is the pointer bitmap, which indicates the position where child pointers exist; this bitmap is referred to as $ptrbitarr$. It shows that pointers exist for entries 100 and 110. These pointers correspond to the two subtrees rooted at the entries 100 and 110. Now instead of storing explicit child pointers in a separate array, the tree bitmap node stores a pointer to the first child trie node as one of the fields in $ptrblk$. The second bitmap shown horizontally is the prefix bitmap. It stores a list of all the

prefixes within the first 3 bits that belong to the node. The bitmap positions are assigned to the prefixes in the order of 1-bit prefixes followed by 2-bit prefixes and so on. A bit in the prefix bitmap is set if that prefix occurs within the trie node.

The search starts from the root trie node and using the same number of bits as the stride for the first level indexes into the pointer bitmap. If the bit in position P is set, it implies that there is a valid child pointer that points to a subtree. To obtain the value of the child pointer, the number of 1 bits in the pointer bitmap is counted up to the indexed position P . Assuming the count is C and the base address to the child block in root trie node is A , the expression $A + (C - 1) \times S$ gives the value of the child pointer, where S refers to the size of each child trie node.

Before following the child pointer to the next level, the search examines the prefix bitmap to determine if one or more prefixes match. The bits used to match these prefixes are the same set of bits used to index the pointer bitmap.

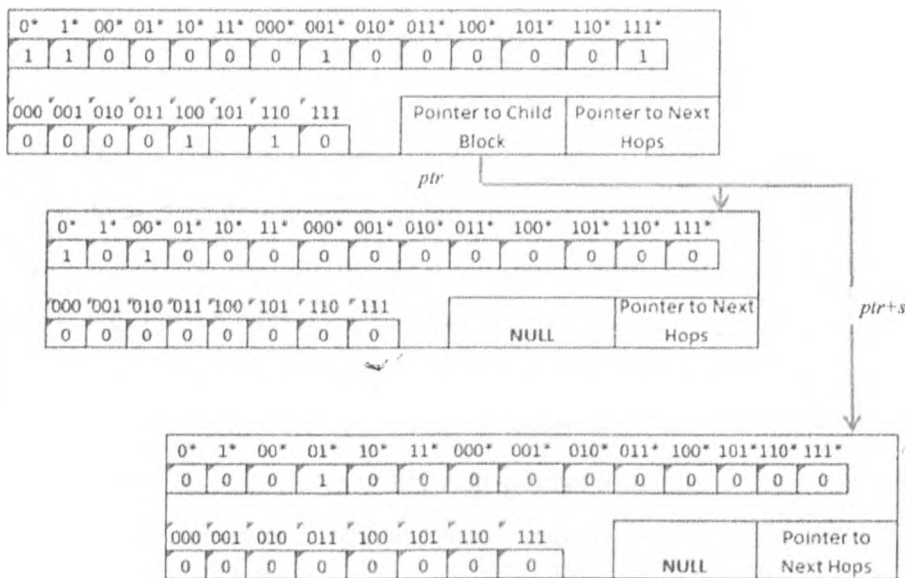


Figure 2.15: Full Tree Bitmap

A search for the address beginning with 10011 in the tree bitmap data structure shown in Figure 2.13 would start by examining the pointer bitmap $ptrbitarr(R)$ of the root node using the first three bits 100. Since the bit is set, the search needs to examine the child subtrie. In the pointer bitmap, as it is the second bit set, the child pointer is computed as $ptr + (2 - 1) \times S = ptr + S$ where ptr is the base address and S is the size of the trie node. Before continuing the search to the child subtrie, the prefix bitmap $pxbitarr(R)$ is examined for matching prefixes. First, the entry corresponding to the first three bits of the address 100 is examined. Since the bit is not set, there is no matching prefix, the search then drops the last bit and examines the entry 10. This indicates there is no match and the search continues to the entry 1. Since the bit is set, a prefix match has been found, the pointer to the nexthop information is computed (similar to the computation of child pointer). This next-hop information is not fetched and instead remembered as the best matching prefix so far. Now the computed child pointer is used to fetch the child node Y . Using the last two bits of the address 11, the child bitmap $ptrbitarr(Y)$ is examined. Since the bit corresponding to entry 11 is not set, there is no more child subtrie to examine. The prefix bitmap $pxbitarr(Y)$ is examined for the entry 11. As the bit is not set, there is no matching prefix and the search continues to entry 1. The bit is set indicating the presence of matching prefix $P6$. This prefix is updated as the best matching prefix; the pointer to its next-hop information is computed and fetched, terminating the search.

The pseudo code for Tree Bitmap search below assumes 3 arrays set up prior to execution of the search code. The first array is called `stride []` contains the search address broken into the stride length. So with a stride of 4 bits, and a search address length of 8 bits, the array `stride []` will have 8 entries each 4 bits in length. The designation `stride[i]` indicates the i th entry of the array. The second array that is required is `node_array []` which contains all of the trie nodes. The third array is `result_array` which contains all the results (next hop pointers). In practice the next hop pointers and node data structures would probably all be in the same memory space with a common memory management tool

```

1.  node:= node_array[0];
    /* node is the current trie node being examined; so we start with
    root as the first trie node which is assumed to be at location 0
    of node_array */
2.  i:= 1;
    /* i is the index into the stride array; so we start with the first stride */
3.  do forever
4.    if (treeFunction(node.internal_bitmap, stride[i]) is not equal to null) then
5.      /* there is a longest matching prefix, update pointer */
6.      LongestMatch:= node.results_address + CountOnes(node.internal_bitmap,
7. treeFunction(node.internalBitmap, stride[i]));
8.  if (extending_bitmap[stride[i]] = 0) then
    /* no extending path through this trie node for this search */
9.  NextHop:= result_array[LongestMatch];
    /* lazy access of longest match      pointer to get next hop pointer */
10. break;
    /* terminate search)
11. else
    /* there is an extending path, move to child node */
12. node:= node_array[node.child_address + CountOnes
    (node.extending_bitmap, stride[i]);
13. i=i+1; /* move on to next stride */
14. end do;

```

Figure 2.16: Pseudocode for Tree Bitmap Algorithm

There are two functions assumed to be predefined by the pseudocode. The first function 'treeFunction' can find the position of the longest matching prefix, if any, within a given node by consulting the internal bitmap. The function treeFunction takes in an internal bitmap and the value of stride[i] where i is for the current multi-bit node. The second function CountOnes simply takes in a bit vector and an index into the bit vector and returns the number of '1' bits to the left of the index. There are several variables assumed. The first is "LongestMatch" which keeps track of an address into the result_array of the longest match found so far. Another variable is 'i' which indicates what level of the search we are on. A final variable is the node which maintains the current trie node data structure under investigation.

The loop terminates when there is no child pointer (i.e., no bit set in extending bitmap of a node) upon which we still have to do our lazy access of the result node pointed to by LongestMatch to get the final next hop.

Refinement to Basic Scheme:

There is an irritating feature of the basic Tree Bitmap algorithm as shown in the Figure 2.10. Prefixes like P10 will require a separate trie node to be created with bitmaps that are almost completely unused. While this cannot be avoided in general, it can be mitigated by picking strides carefully, probably by using dynamic programming. Suppose we have a trie node that only has external pointers that point to prefixes such as P10.

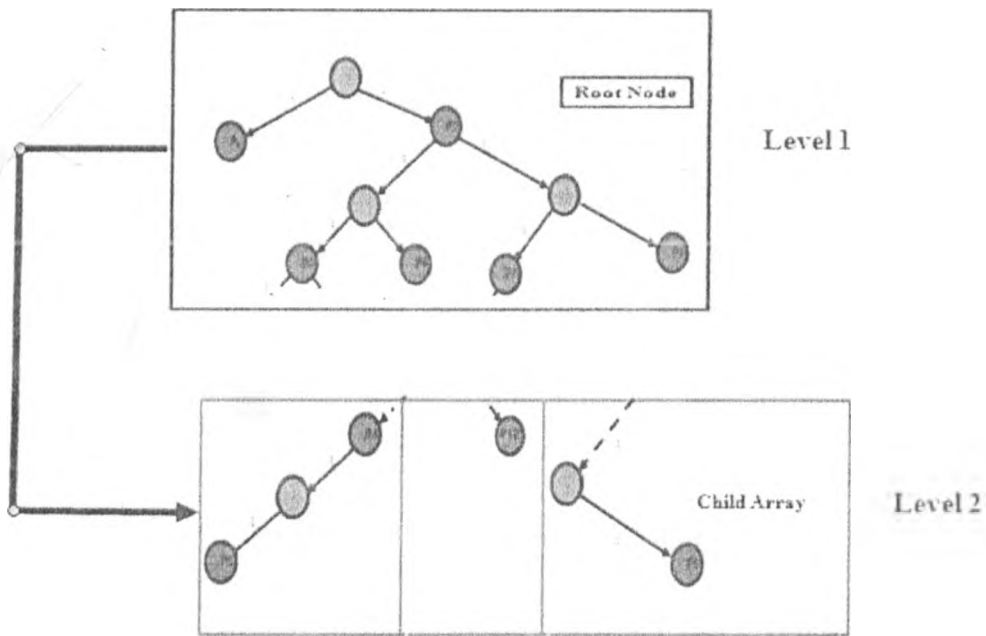


Figure 2.17: Root Tree Bitmap Node with Child Array

Then a special type of trie node can be made in which the external bit map is eliminated and substituted with an internal bit map of twice the length. The new node now has room in its internal bit map to indicate prefixes that were previously stored in such nodes. The nodes can then be eliminated and the corresponding prefixes stored in the final node

itself. Thus P8 is moved up to be stored in the upper trie node which has been modified with a larger bit map.

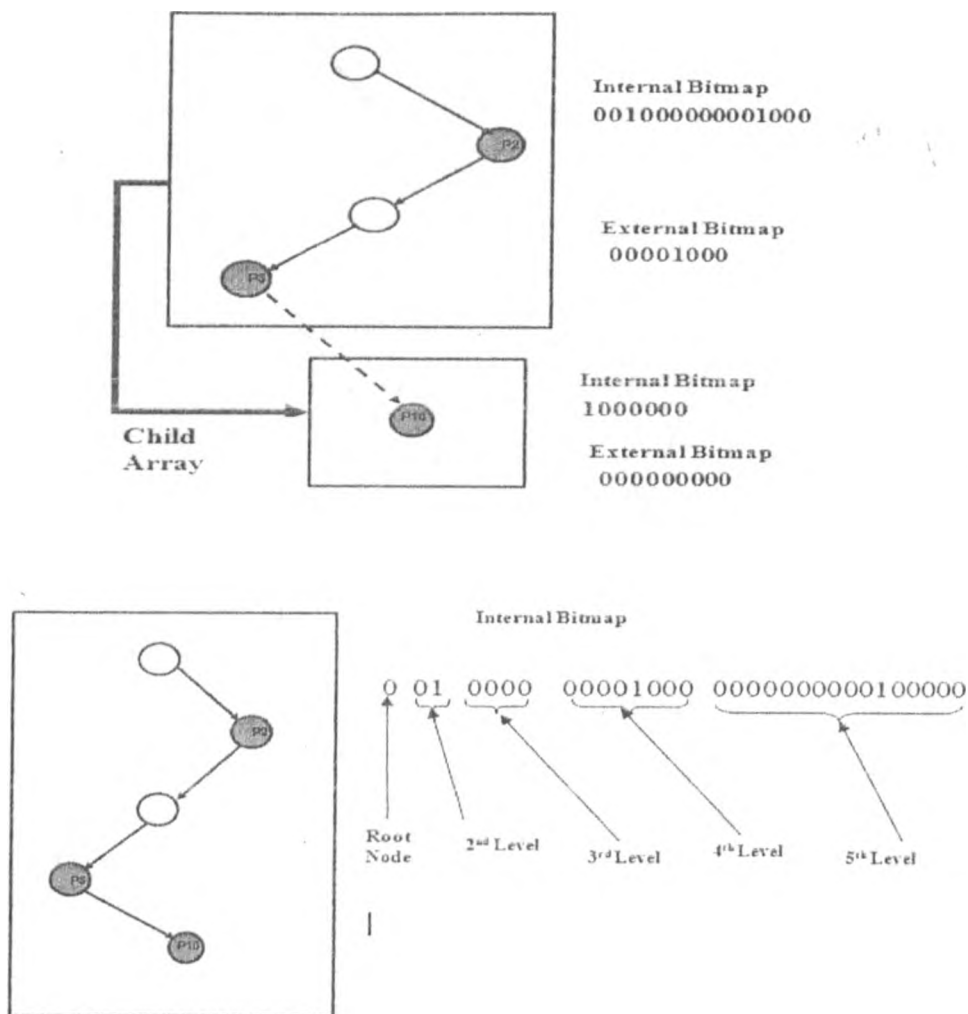


Figure 2.18: Optimized Tree Bitmap

2.6 Simulation

The Internet is greatly heterogeneous and the rate of change makes it a deeply uncontrolled environment. Experiments are difficult to replicate, verify, or even understand without the stability and relative transparency provided by simulators such as Opnet, NS2 and emulator or self-contained experimental test bed. Simulators provide full

control over the network model, meaning the full range of experimental parameters—network topology, end-node protocol behavior, queue drop policies, congestion levels, and so forth.

With simulators, researchers can model unusual situations, or extrapolate current Internet trends to evaluate likely future behavior. Networks can also be modeled with little relationship to Internet reality or with an unknown relationship to Internet reality. This isn't necessarily a problem. Some divergences between models and reality are unimportant, in that they don't affect the validity of simulation results, or useful, in that they clarify behavior in simple cases. Furthermore, some divergence is necessary in order to investigate the Internet of the future instead of the Internet of the past or present. However, the research community has not yet determined which divergences are acceptable and which are not.

This basic question has led to difficulties in this project and in the evaluation of other work. There is a continuing frustration with poor quality of models in general in Internet research, and a need for a critical approach in evaluating the models that are used in simulations and experiments. It is important to know when a model might lead to bad results, and what those results might be. This work needs to be grounded in Internet measurement.

2.6.1 Desirable characteristics of simulator

IP lookup is becoming popular due to their interesting properties such as decentralisation, scalability, self-organization and robustness. Those properties impose some important requirements on the simulator. The following list shows some of the desirable characteristics of an IP Lookup algorithm simulator:

- **Scalability:** simulators need to be scalable to thousands even millions of nodes. An IP Lookup algorithm simulator should be able to run simulations with a large number of peers while making an efficient use of computing resources.
- **Flexibility:** The simulator should be able to run simulations of both structured and unstructured overlay networks. The user of the simulator should also be able to

specify the parameters which relevant for a specific simulation such as the number of nodes, the mobility of the nodes or the churn rate.

- Usability and Documentation Usability is related to how easy to learn and use the simulator is. A simulator should have a clear and understandable API that allows to implement protocols in an easy way. A good documentation explaining how to use the simulator is also very important. Some simulators have very poor documentation and the code is difficult to understand which makes these simulators very hard to use.
- Underlying Network Simulation Some simulators do not model the underlying network or they offer a limited simulation of the network layer. This makes simulations of peer to peer protocols not as realistic as with simulators with a better modeling of the network layer. In some cases researches prefer to focus on the algorithm verification without worrying much about some parameters of the network layer such as latency costs. In other cases, a proper simulation of the network layer is necessary. In those cases, an exchangeable network model would be desirable.
- Statistics: The simulator should be able to collect significant results which are easy to understand and manipulate.
- Repeatability: Mechanisms should exist to allow the repeatability of simulations. Repeatability is important to reproduce experiments, compare different proposals and evaluate the influence of a parameter by changing it in different simulations. Some papers present results that are not reproducible and therefore, comparisons between different proposals are difficult to do.

It is very difficult to build a simulator that satisfies all the requirements. To fill some requirements some simulators need to sacrifice other requirements. For instance, if a simulator wants to offer high scalability probably the network layer will need to be represented by a simple model, without considering the low-level details such as the overhead associated to the communication stack. Sometimes, an accurate level of detail is not necessary to evaluate some protocols.

The difficulties found in satisfying all the requirements have led to different research groups to develop their own simulator to evaluate their protocols.

2.6.2 Study of existing simulators

Although there is a wide range of simulators, most of them are "home-made" solutions built to simulate a specific protocol or systems. Within simulators there are network simulators and overlay simulators. Network simulators provide a framework for accurate simulation of network protocols such as TCP, UDP, IP, AODV etc. These simulators model the network at the packet level, considering parameters such as delay, bandwidth and other lower-level concerns. Some well known network simulators are Opnet, NS-2 and OMNET++. These simulators perform very well when evaluating network protocols but they do not scale well for overlay networks with a large number of nodes. For instance Omnet++ can't simulate more than 1000 nodes and Narses can simulate up to 600. This is due to the overhead added by the network details. On the other hand, overlay simulators are less focused on the lower level and more focused on evaluating the overlay algorithms.

| Simulator | Max Nodes |
|-----------|-----------|
| Narses | 600 |
| Omnet++ | 1,000 |
| Opnet++ | |
| NS2 | (10,000) |

Table 2.6: Maximum number of nodes simulated by each simulator.

2.6.3 The Simulation Model

In order to develop a high performance and suitable route lookup algorithm for next generation high speed routers, it is prudent to first inspect the route databases and make use of the characteristics of the distribution to generate more routes for evaluating the corresponding performance.

The model represents the physical network where nodes communicate with each other by sending route messages. The model is quite simple and it does not introduce details such as latencies or node mobility. However, it can be extended to simulate more complex

scenarios. Considerations have been made on the possibility of integrating the simulator with existing simulator such as NS2 or OMNET++ to simulate more realistic networks.

The model simulator tries to satisfy three requisites:

- Scalability. One of the most important characteristics in a lookup algorithm is its scalability. The Simulator should allow studying the lookup algorithm performance in scenarios with a large number of IP prefixes.
- Extensibility. The design of the simulator is simple and easy to extend.
- Usability. The simulator is easy to use and allows the researcher to configure all the parameters necessary for their tests. Moreover, the simulator provides results which are easy to understand and significant for the evaluation of the simulated algorithm.

The model devised and used in this project tries to meet the above requirements. Due to time constraint imposed on this project work, the model may miss some critical requirements for testing forwarding functions of routers. The model strives to accurately reflect IP lookup functions in routers under a broad range of workloads and in special cases in order to be computationally inexpensive may miss certain requirements. The integration of the components into a single system was done and the model used for testing different IP lookup algorithms.

CHAPTER 3

IMPLEMENTATION

3.1 Introduction

Before I actually show the results of this study and the conclusions derived from them, I sum up the assumptions that were used in this project:

- The entire forwarding table is placed in cache so lookups are performed with an undisturbed cache. That would emulate the cache behavior of a dedicated forwarding engine. However, having access to conventional general-purpose laptop makes it difficult to control the cache contents on such systems. The cache is disturbed whenever input/output is performed, an interrupt occurs, or another process gets to run. It is not even possible to print out measurement data or read a new IP address from a file without disturbing the cache.
- All traffic from the nodes in the network gets through without any loss or delay.
- I assume that the buffer size of a router is large enough to accommodate all the generated prefixes. This probably would not be the case since the router buffer size is not unlimited in size.
- There is no memory management overhead and it is not correlated with database size. The results for traditional IP address lookup schemes traditionally do not contain any memory management overhead.

Various IP lookup algorithms are implemented in this project. In the lack of a wide deployment of IPv6, I generate IPv6 prefixes based on IAB/ RIPE/ IANA allocation recommendations. This distribution captures the expected behavior that majority of the organizations will use the shortest allocated prefix possible. It is my belief that by using these recommendations, I am not compromising the accuracy of the algorithm.

To calculate the time taken for IP lookup using all the algorithms, I have used the inbuilt java time function. I chose to use the inbuilt java function to get an accurate result. Every

time this timer generates an interrupt a counter is incremented. I measure the time taken for each task based on this counter. Output of this program shows the time taken for each algorithm and the memory consumed.

Several parameters are defined for the simulations, which correspond with the main features of the algorithm to be tested. In each simulation, some of the parameters were held fixed in all tests, while others were varied. The measured quantity varied between tests too, depending on the issue I was looking at.

The test stresses the algorithm in incremental steps and measure its ability to perform under load conditions. The effect of the following parameters are investigated: number of prefixes, length of prefixes and composition of the forwarding table. The metrics of performance include packet lookup time, memory consumption and scalability.

3.2 Observations:

Algorithms are ideally expected to perform well in the worst case. However, exploiting some of the following practical observations to improve the expected case is desirable. Usually, the performance of general algorithms can be improved by tailoring them to the particular datasets they will be applied to. Figure 3.1 shows the prefix length distribution extracted from RIPE NCC Routing Information Service (RIS) peers on 1st August 2011 and MAE-East peering point. As can be seen, the entries are distributed over the different prefix lengths in an extremely uneven fashion. This distribution is representative of a large forwarding table used near the core of an IP network.

- For IPv4, most of the prefixes are 24 bits or less in core routers, while more than half are 24 bits. About half of all prefixes have length 24 with most of the remainder distributed between 16 and 23 bits.
- There are not very many prefixes that are prefixes of other prefixes. Practical observations show that the number of prefixes of a given prefix is at most seven.
- It is obvious but important that there is no prefix with length between 64bit and 128bit (excluding 64bit and 128bit).

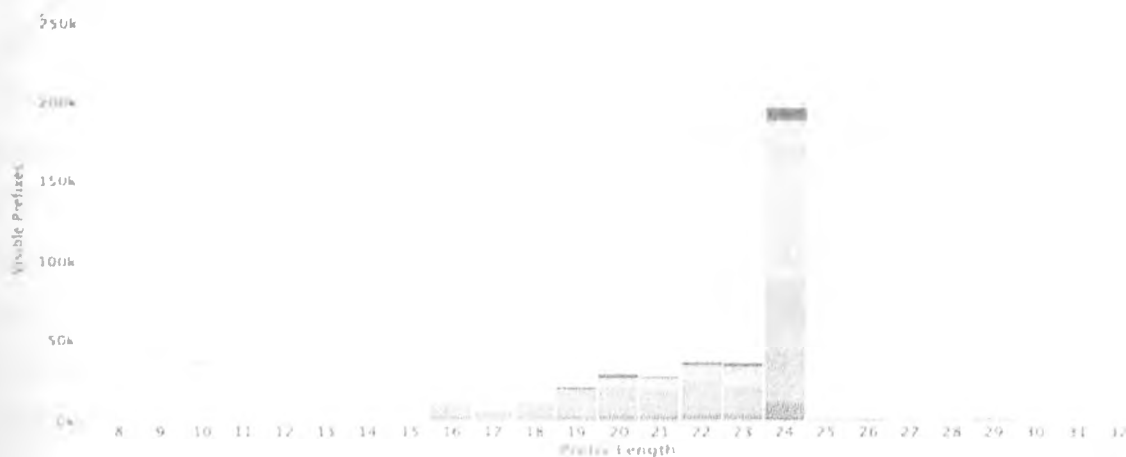
- The majority of the prefixes should be the '/48s', and '/64s' the secondary majority. Other prefixes would be distinctly fewer than the '/48s' and '/64s'.
- Future (or the near future) IPv6 address blocks will be allocated to common subscribers mainly from the current assigned LIRs. This is essential for IPv6 prefix generating.
- Though the address length is bound to increase, the levels of subnet/prefix would not be distinctively scaled (e.g. only 4-5 levels), due to consistent width subnet field.

These practical observations can be further exploited to come up with efficient lookup schemes.

3.2.1 Prefix Length Distribution:

Table below provides a realistic distribution of prefix lengths that have been modeled after real internet measurements.

Visibility of IPv4 prefixes



Source: RIPE NCC Routing Information Service (RIS) peers on 1st August 2011

Figure 3.1: Distribution of IP Prefixes

Visibility of IPv6 prefixes



Source: RIPE NCC Routing Information Service (RIS) peers on 1st August 2011

Figure 3.2: Distribution of IP Prefixes

Figure 3.2 depicts the IPv6 prefix distribution on prefix length of a real-world IPv6 global route table (Route-View IPv6 route table, Data: 2012-10-3, Size: 680 Prefixes). The majority are '/32' prefixes, which is referred to as the initial IPv6 allocation blocks. This kind of IPv6 address blocks are allocated to the LIRs. Some shorter prefixes were assigned to high-level subscribers according to RFC 2374 before it was replaced by RFC 3587. In RFC 2374 and RFC 2928, IPv6 address blocks were organized in a complex aggregatable hierarchy which includes the TLA (Top Level Aggregation) '/16' blocks, sub-TLA blocks, NLA (Next Level Aggregation) '/48' blocks, SLA (Site Level Aggregation) '/64' blocks and the Interface Level address ('/128'). There are only four prefix levels (i.e., subnet levels). This information is useful for IP prefix generation.

3.2.2 Forwarding Table Growth: ~

The main area of growth that is considered here is in database size. To project future database sizes a history of total BGP prefixes from January 1, 2004 to November 1, 2011 was studied. The growth in an eight year time period was approximately linear. In January 2004, the report shows approximately 150,000 prefixes in existence. If a lookup engine was required to hold all the network prefixes, and if the prefix growth rate

continues to be linear then the current IP lookup schemes would slow down internet speeds. Lookup engines with a capacity of 200,000 entries could potentially be large enough until the year 2008. With the projected growth, several reasons would result in not using the same lookup engines and algorithms; reasons includes: database change in distribution causing worse prefix to memory ratio, increase in line rates causing OC-192c rate to become obsolete, change in Internet standards (like increase of IP address length).

Table below provides a realistic growth on the number of prefixes as projected up to 2016.

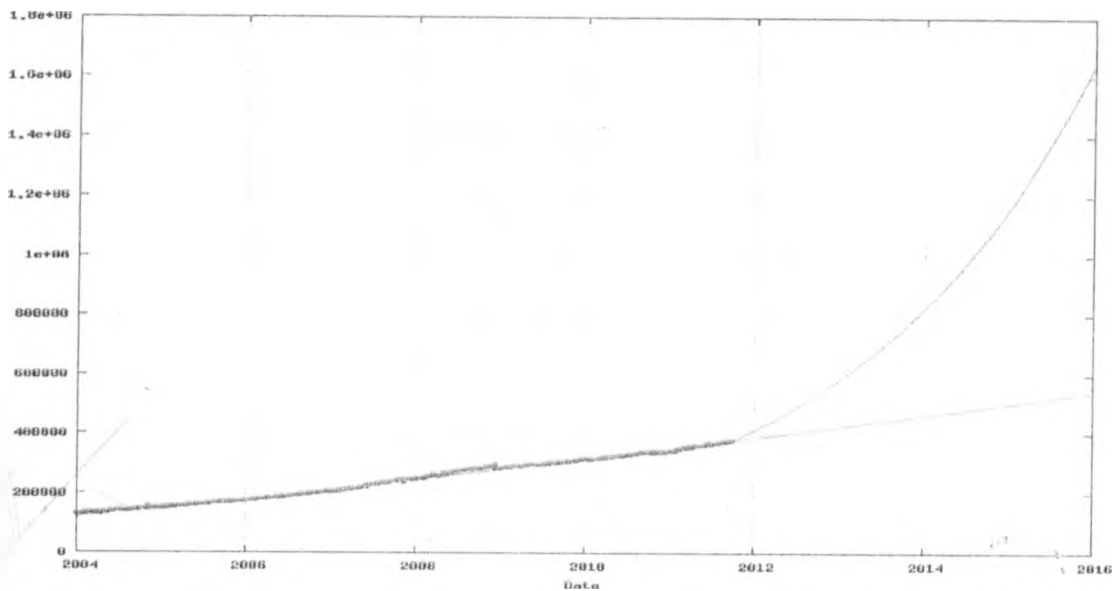


Figure 3.3: Projected Forwarding Table Growth

Source: Asia Pacific Network Information BGP Routing Growth in 2011

I have implemented the Binary Trie, Patricia Trie, MultiBit and Tree Bitmap Algorithms. I have implemented this project on an AMD Ethlon processor using Windows Operating system. Through the research, analysis and design of the key components required for the testing the algorithms were developed in a Java environment. Java is chosen because of its simplicity and the quick ramp-up it enables. A logical diagram of the development environment used is given below.

3.3 Design of Simulator:

The top level view of the simulator through its main modules is described below.

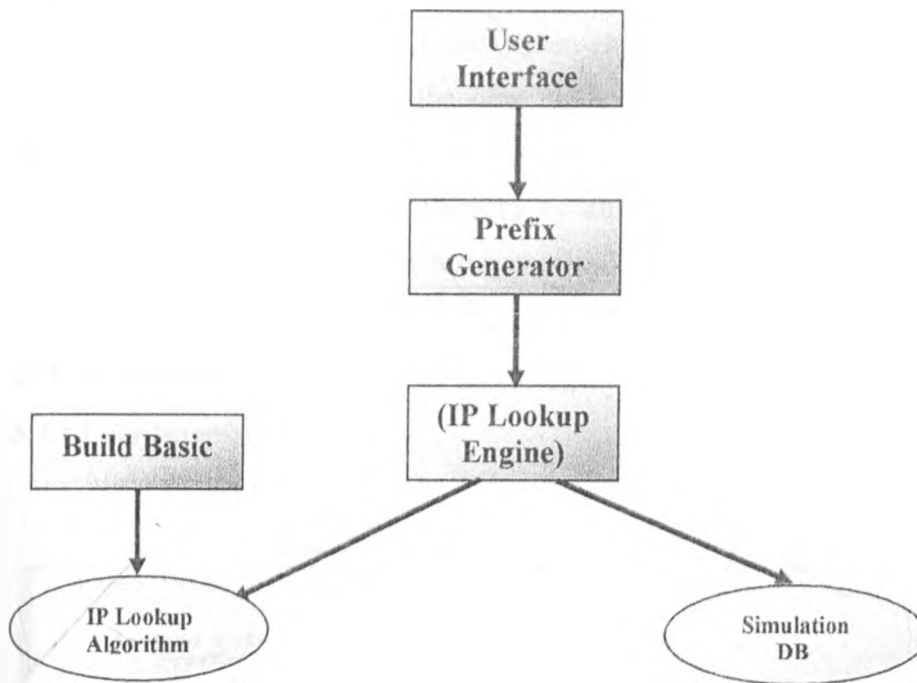


Figure 3.4: Top Level Architecture of the Model

I compare lookup algorithms using a software platform. Software platforms are more flexible and have smaller initial design costs than hardware design. To investigate scalability aspects of IP packet forwarding, I consider the time and memory required to lookup IP prefixes during packet forwarding. The performance comparison is conducted under a common platform using the similar IP prefix database, development and hardware platforms. The platform consists of a Pentium IV 2.1 GHz processor machine with 2 Giga Bytes of RAM. Java language was used for developing the simulation tool and for coding the algorithms because of the ease with which new requirements can be integrated in the tool and familiarity with the development environment. The codes are executed under the windows operating system.

A combination of IPv4 and IPv6 prefixes stream is generated from an IP Prefix generator based on the above observations. The test starts with a specified initial ratio of IPv6/IPv4 traffic at 100% offered load. The lengths and composition of IPv6/IPv4 addresses used are varied based on the RFC recommendations. The load is then increased and the test repeated. The ratio of IPv6 to IPv4 is then changed by a specified increment and the entire test repeated again for the new ratios. Traffic generator advertise a series of different prefix lengths based on the recommended IPv6 and IPv4 distribution in the real internet traffic. Lookup time, memory consumption and scalability for each of the algorithms is measured for every test scenerios.

3.4 Modules:

3.4.1 User Interfaces:

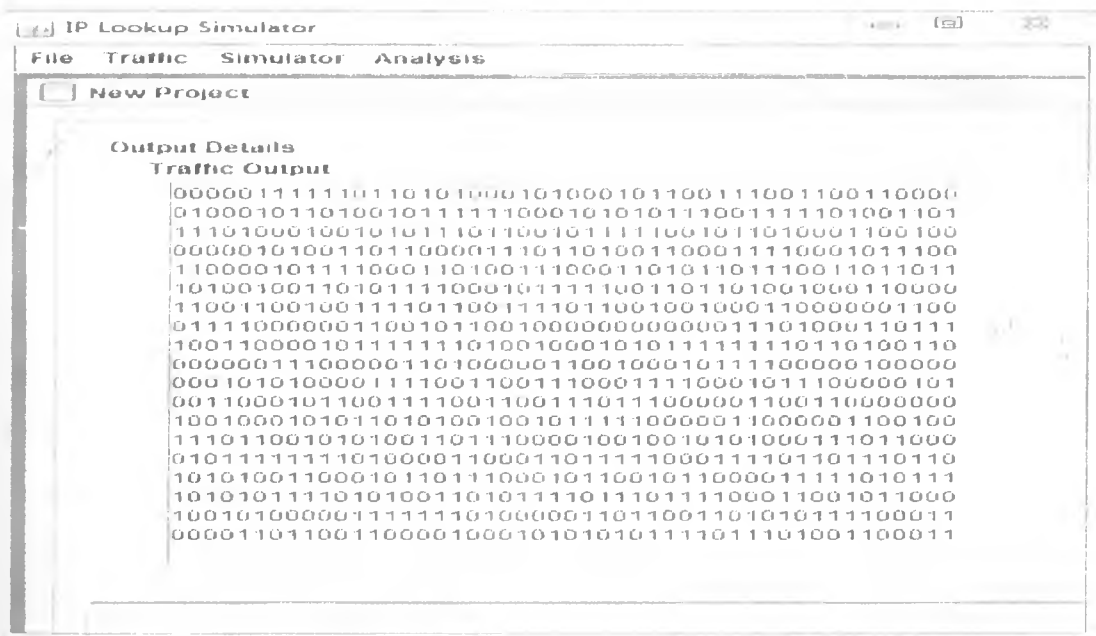
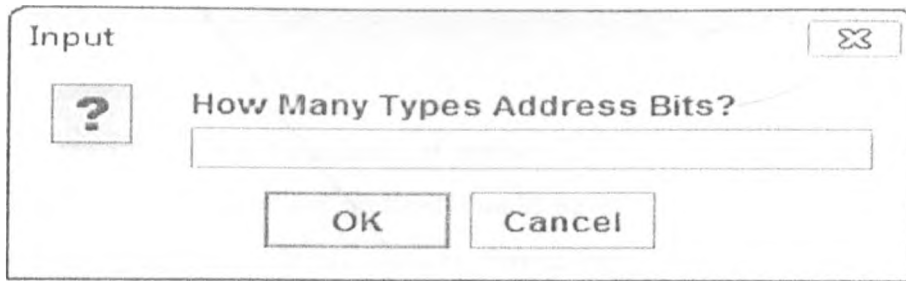


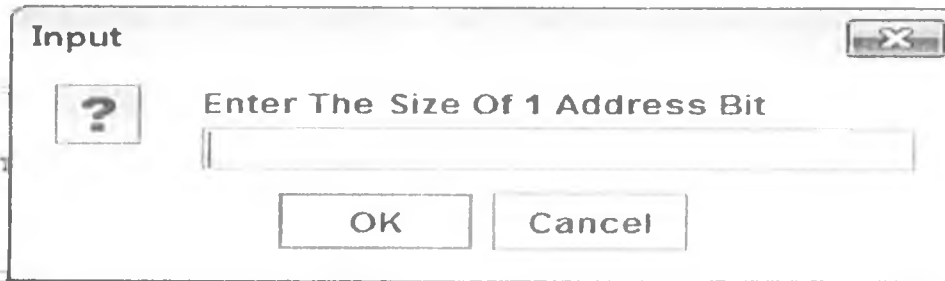
Figure 3.5: Top User Interface Model

This module handles requests for a specific simulation

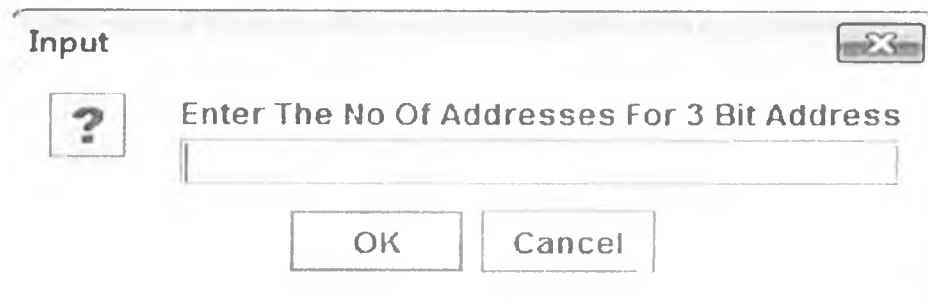
- Distribution of prefix lengths to be added to the database.



- Size of each address distribution to be added to the database.



- Number of prefixes in the database.



- IP Lookup algorithm to be used for simulation. Performs lookups in the database according to the chosen algorithm. Performs all memory management required for the database.

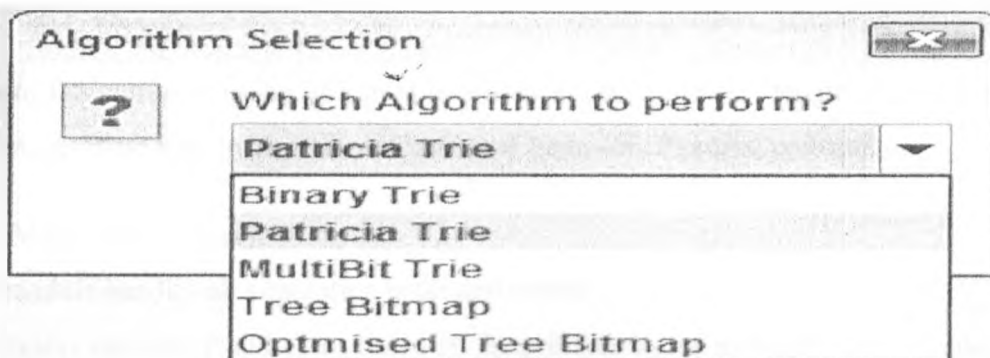


Figure 3.6: User Interfaces Screen Shots

3.4.2 Packet Generator:

This function generates a specific number of prefixes according to the given distribution on prefix length; all the generated prefixes should have a same prefix, i.e., the seed prefix. Then the function randomly assigns forwarding information to each prefix generated. Such information includes the forwarding output port, next hop IP addresses, and so on.

This module creates all the necessary data for the simulation flow:

- Generates based on recommendations from IAB/IPMAP and practical observations of IP address distributions, the required number of prefixes.
- Creates a mirror database of the prefixes – to be used by the model for testing other IP Lookup algorithms.
- Stores the generated prefixes in memory – in case the simulation doesn't immediately follow the pre-simulation phase, and we want to keep the prefix list.

3.4.3 Build Basic:

For all the methods, I take as a source for building the lookup structures a list of routing table entries. Although in the case of real implementation in a router, the tables would be built from some dynamic structure at the central processor, for each method the structure would be different, so as to enable the fastest possible build procedure. It remains to be seen which would be the most convenient one for each method. However to make a fair comparison of the build time, I chose a random list of the entries as the input data to the build process of all methods.

Each of the entries consists of a next hop address, port number, length of the IP prefix. The entries were kept in an array consisting of tuples of IP prefix, port number.

3.4.4 Model (Simulator):

This module handles all simulation tasks and events:

- Creates random IP addresses based on the prefixes stored in the database padded with random data 'forwarding ports'.

- Injects the incoming IP address to the search engine.
- Keeps simulation's statistics – average number of memory accesses, lookup rate, and use of markers and of BMPs etc. and prints runtime simulation information to the screen.

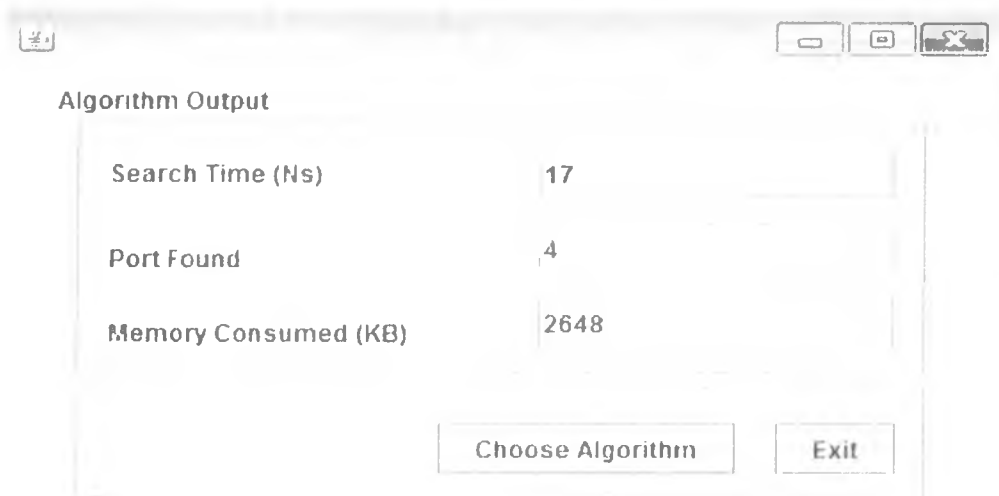


Figure 3.7: Output Module Screen Shot

It displays the port number and the lookup time respectively. The output of the search module displays the port number, lookup time and the memory consumed. This shows the memory used by the system to search for the IP address specified to the system.

3.4.5 General functions:

Not precisely an active module, more a collection of help functions for handling files and manipulation of IP addresses.

3.4.6 Correctness checking:

To enable check the correctness of lookups, it would have been ideal to assign each prefix with a unique number. This unique number would then reside both in the main database and in the simulation database.

When the model creates a new IP address to perform lookup upon, it chooses a random prefix from the simulation database and pads it with random bits. It also stores the

prefix's unique number. When a lookup is completed, the search engine returns the found prefix and its unique number, also called port number, so that all that is to be checked is check whether it is the expected unique number. This process is automated.

This project implements a manual process where each prefix can be manually confirmed to be correct by checking the returned prefix against the input.

CHAPTER 4

EXPERIMENTAL RESULTS

4.1 Introduction

By using synthetically generated data based on recommendations of RFC/IPMA3/RFC and IAB, I study and report some of the numerical results carried out to evaluate the performance of IP lookup algorithms. I propose two different types of tests sessions: one for evaluating the impact of growth of routing and the impact of partial deployment of IPv6. For each type of test, i show results in the form of a graph and a discussion for the same results.

In the lack of a wide deployment of IPv6, i used randomly generated routing entries based on IAB allocation recommendations.

4.2 The objectives of the experiments:

- (a) To test if the simulation model is functional and effective.
- (b) To compare the performances of different lookup algorithms based on the following parameters:
 - i. Lookup speed.
 - ii. Memory consumption.
 - iii. Scalability.
- (c) Based on the results obtained above, propose the best lookup algorithm for high speed Internet routers.

The general objective of these experiments is to characterize the performance and the scalability of the IP lookup algorithms. The IP lookup algorithms such as Binary Trie, Patricia Trie, Tree Bitmap algorithms will be tested and characterized.

4.3 Justification for selected plans

Some of the algorithms suggested which are either software based or hardware based were not chosen, either because they didn't show much significant improvement from the current implementations in terms of time, space complexity or they would require more time for analysis and implementation. Also with the implementations of the four algorithms including the proposed optimised tree bitmap algorithm, i have chosen to use my own implementation due to the very same reason of lack of time for analysis and implementation.

Additionally, there has not been a single simulator singled out for testing IP Lookup algorithms largely due to the fact that simulating IP lookup algorithms for the future internet requires using large prefix databases. Simulating hundreds of thousand or even millions of nodes in a conventional simulator is computationally expensive.

4.4 Methodology

1. Specify the number of IP prefixes to be generated based on the recommended distributions.
2. Specify the lengths of the prefixes based on visibility of the prefixes on the network.
3. Specify the number of prefixes to be generated for each distribution level. IP prefix database will be created based on the above three parameters input by the user.
4. Select the algorithm to use for looking up the incoming IP address.
5. Input the incoming IP address to search.
6. Record the output that consists of lookup time and memory consumed

Input Data Size

- i. Due to system limitations, the maximum number of entries used was restricted.

Prefix Length Distribution

- ii. Constant length.
- iii. Length distribution as in existing tables. The prefix length distribution in existing tables was modeled according to values found at RIPE NCC Routing Information Service (RIS) peers on 1st August 201.

iv. Random length.

For generating the random values a random generator is proposed. The prefix length distribution was modeled using the random generator and a random-variate technique.

4.5.1 Case 1: Projecting Prefix Growth

The impact of growth in forwarding table sizes due to new prefix allocations is projected as this is likely to affect forwarding rates of routers. The focus is on lookup times and memory requirements. The first simulation is a straightforward test of the algorithm's performance in a simple environment, checking how the speed of the search varies as the number of address prefixes increases.

Internet traffic measurements show that roughly 50% of the packets that arrive at a router are TCP-acknowledgment packets, which are typically 40-byte packets. As a result, a router can be expected to receive a steady stream of such minimum size packets. Thus, the prefix lookup has to happen in the time it takes to forward a minimum-size packet (40 bytes), known as wire speed forwarding. At wire speed forwarding, the amount of time that it takes for a lookup should not exceed 320 nanoseconds at 1 Gbps ($= 10^9$ bps), which is computed as follows:

$$\frac{40 \text{ bytes} \times 8 \text{ bits/byte}}{1 \times 10^9 \text{ bps}} = 320 \text{ nanosec.}$$

Similarly, the lookup cannot exceed the budget time of 32 nanoseconds at 10 Gbps and 8 nanoseconds at 40 Gbps. The main bottleneck in achieving such high lookup speed is the cost of memory access. Thus, the lookup speed is sometimes measured in terms of the number of memory accesses.

I generated prefixes with lengths ranging from 16 bits to 24 bits so as the distribution of the length is equal to the distribution of prefix length in major routing tables. That is, I simulate the same distribution as in Mae East database. The information about the prefix length distribution was obtained through IPMA. The results of simulation are displayed in Table 4.1.

| | 500 | 1500 | 4500 | 9500 | 19500 | 39500 |
|-----------------------|-----|------|------|------|-------|-------|
| Binary Trie | 132 | 210 | 218 | 352 | 451 | 814 |
| Patricia Trie | 150 | 165 | 175 | 210 | 296 | 731 |
| Tree Bitmap | 75 | 90 | 125 | 170 | 300 | 650 |
| Optimized Tree Bitmap | 54 | 62 | 76 | 110 | 210 | 236 |

Table 4.1. Lookup parameters for IPv4 length distribution as in existing tables.

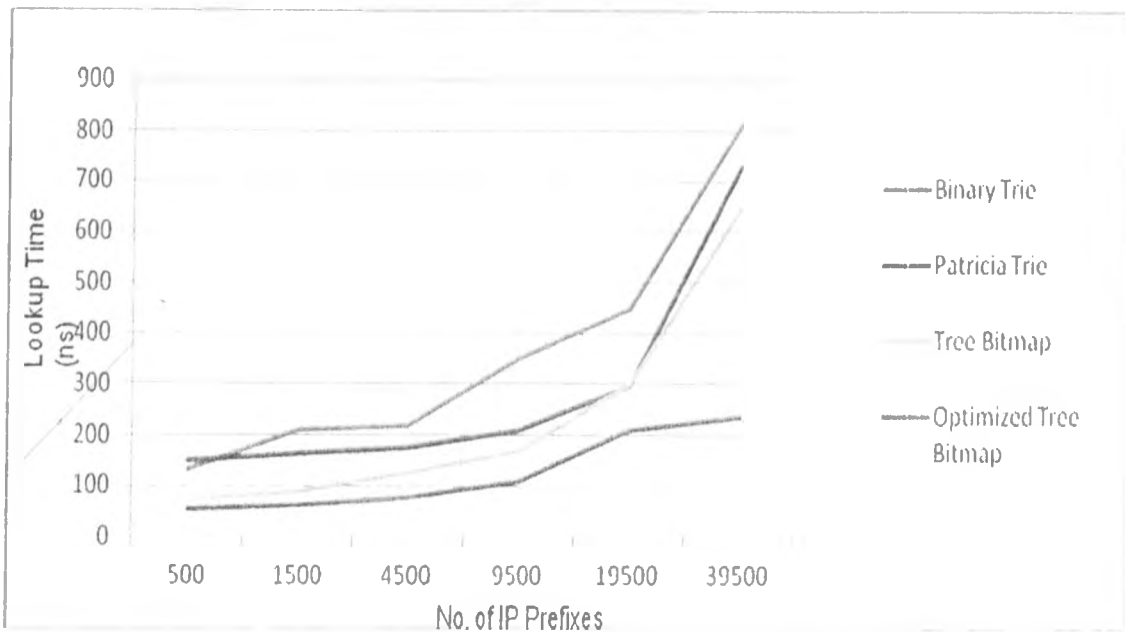


Figure 4.1: IP Lookup times under varying IPv4 FIB sizes.

Observing Figure 4.1 above I note how the increase in routing table impacts the speed of IP lookup. At each simulation, the traffic load is doubled and lookup time measured. The algorithms achieve less than 100 ns in lookup time when the total offered load is less than 5000 routes. When the offered load in the network increases, lookup time also increases at the router, this degrades the performance of the router. This happens after offered load of about 1000 packets. Even at loads of more than 4500 prefixes, Optimized Tree Bitmap still performs lookups at less than 100ns.

| | 500 | 1500 | 4500 | 9500 | 19500 | 39500 |
|-----------------------|-----|------|------|------|-------|-------|
| Binary Trie | 17 | 41 | 59 | 67 | 120 | 198 |
| Patricia Trie | 34 | 54 | 60 | 76 | 112 | 154 |
| Tree Bitmap | 8 | 40 | 53 | 62 | 74 | 94 |
| Optimized Tree Bitmap | 8 | 16 | 22 | 32 | 48 | 56 |

Table 4.2: Lookup parameters for IPv6 uniform length distribution and fixed step.

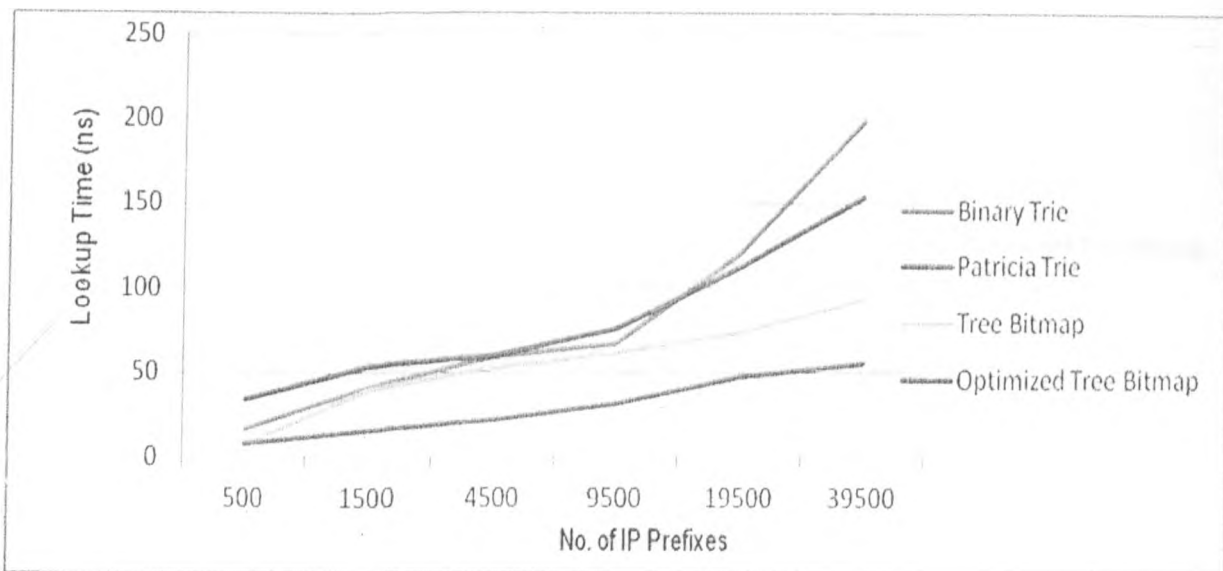


Figure 4.2: IP Lookup times under varying IPv6 FIB sizes.

The amount of memory consumed by the data structures of the algorithm is also important. Ideally, it should occupy as little memory as possible. A memory-efficient algorithm can effectively use the fast but small cache memory if implemented in software.

| | 500 | 1500 | 4500 | 9500 | 19500 | 39500 |
|-----------------------|------|------|-------|-------|-------|-------|
| Optimized Tree Bitmap | 1429 | 8241 | 10053 | 23512 | 14748 | 12866 |
| Tree Bitmap | 982 | 8364 | 10863 | 23734 | 15065 | 13484 |
| Patricia Trie | 1126 | 8639 | 11055 | 23886 | 15204 | 13645 |
| Binary Tree | 1270 | 8761 | 11213 | 24020 | 15349 | 13790 |

Table 4.3: Lookup parameters for IPv6 uniform length distribution and fixed step.

In this case, prefixes are generated with random length of uniform distribution between 32 bits and 64 bits and the step between numbers generated remains fixed. Again with small steps we obtain many duplicates but we can observe in Table 4.3 that with the rising number of generated prefixes, Optimized Tree Bitmap algorithm space requirements grow much slower than those of Tree Bitmap algorithm.

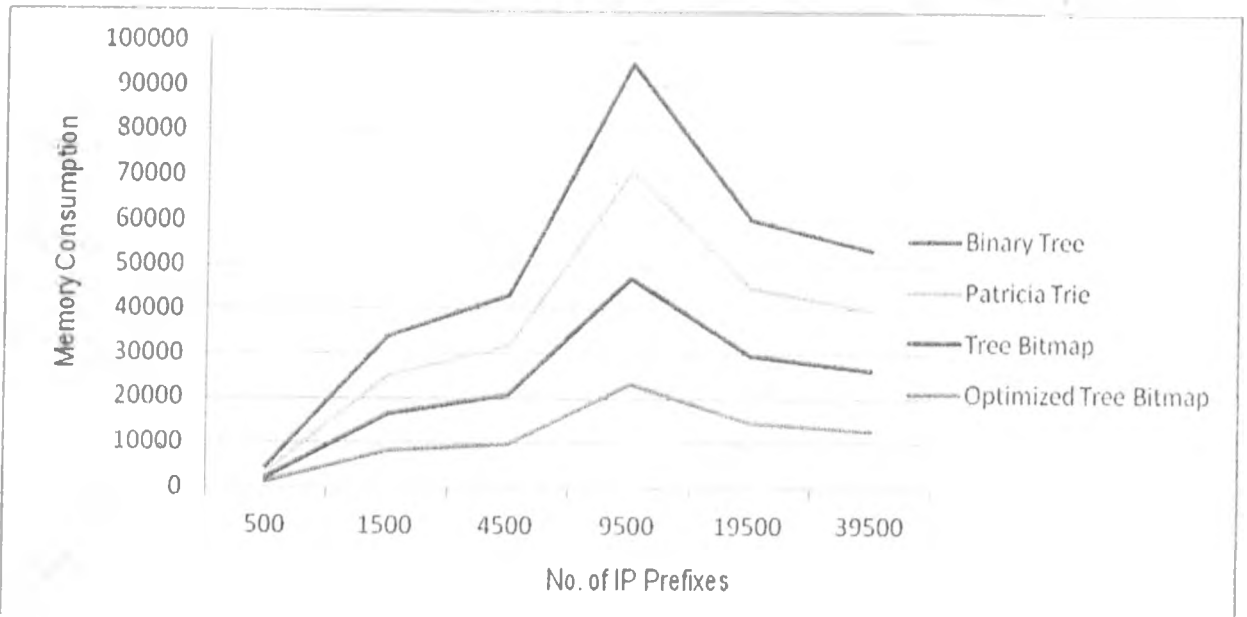


Figure 4.3: Memory requirements under varying IPv4 FIB sizes.

| | 500 | 1500 | 4500 | 9500 | 19500 | 39500 |
|------------------------------|-------|-------|--------|--------|--------|--------|
| Optimized Tree Bitmap | 20787 | 58965 | 104096 | 153430 | 175483 | 200175 |
| Tree Bitmap | 19020 | 56442 | 102588 | 151950 | 173704 | 198722 |
| Patricia Trie | 17576 | 56442 | 101144 | 150502 | 172267 | 197360 |
| Binary Trie | 11647 | 54398 | 98777 | 143807 | 167855 | 194224 |

Table 4.4: Lookup parameters for IPv6 uniform length distribution and fixed step.

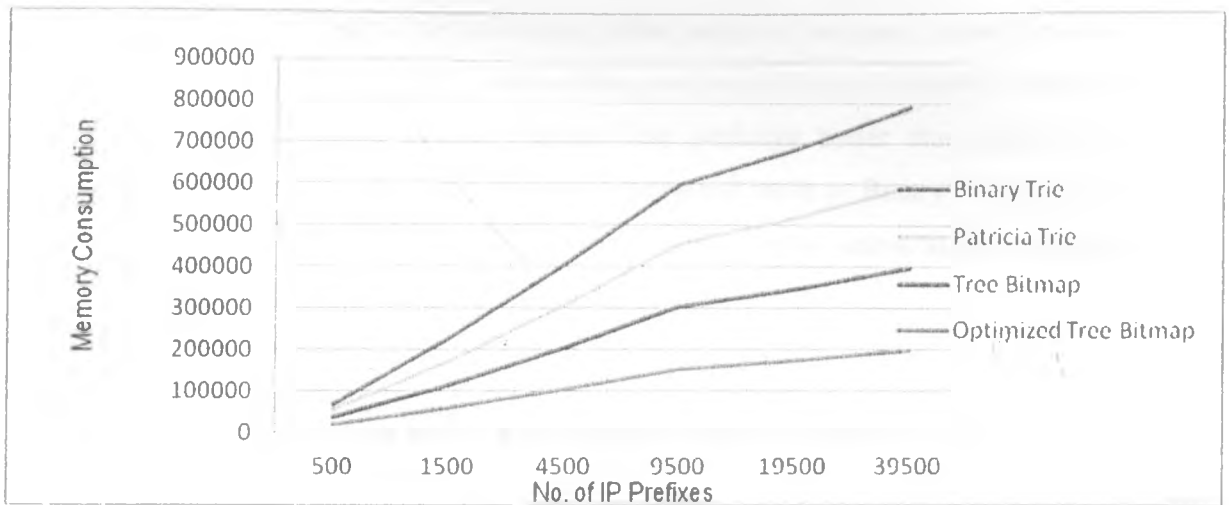


Figure 4.4: Memory requirements under varying IPv6 FIB sizes.

Figure 4.3 and 4.4 given above shows the simulation results using offered load of between 500 and 20000 IP addresses. The result is the average memory requirement of each algorithm to lookup a particular IP address. The initial low memory requirement can be attributed to the fact that the amount of memory required for a data structure of a few addresses is always small. As the number of addresses is increased, the memory consumed by the data structure also increases.

To compare memory consumption by the algorithms based on the different lengths of IP prefixes, memory utilization was plotted using the same offered load as shown in figures 4.3 and 4.4. For each of the offered loads, all the algorithms perform better with IPv4 addresses as compared to IPv6 addresses.

Initially at the start of the experiment, Binary Trie performed better than Patricia Trie and Optimized Tree Bitmap Algorithms. Optimized Tree Bitmap Algorithm should perform better than Binary Trie Algorithm. This could be attributed to the choice of incoming IP address in which case I choose incoming IP addresses of less than 5 bits. As the number of prefixes increases, Optimized Tree Bitmap performs better than the other algorithms because unlike the other algorithms, apart from using multibit nodes to reduce memory consumed, all child nodes of a given trie node are stored contiguously. Only one pointer

that points to the beginning of the child node block needs to be stored in the trie node. Such an optimization reduces the size of the trie nodes hence memory consumed resulting into faster memory access. Patricia Trie performs better than Binary Trie because in Patricia one way branch nodes are compressed while in Binary Trie, one way branch nodes are inspected hence increasing the cost of memory access as the lengths of IP addresses increases.

Studies have shown that in general lookup time and memory consumed are very important factors in the design of next generation routers. In general, lookup time is a function of the depth of the tree and memory depends on the size of the tree. An algorithm that is able to compact data to the size that can be stored in cache memory and is able to reduce the size of the tree drastically will be favorable for next generation routers as the size of the forwarding database continues growing exponentially.

4.5.2 Case 2: Partial Deployment of IPv6 Addresses.

In this experiment, I test the behavior of IP lookup mechanisms based on partial deployment of IPv6 addresses. Since a co-existence of IPv4 and IPv6 is inevitable, I examined the lookup times and memory requirements for IPv4 and IPv6. To achieve this purpose, in all the tests reported in this Section, I use a traffic flow composed by a variable range of random IP prefix addresses of IPv4 and IPv6. Thus, I have performed a test with the aim of analyzing the performance of the discussed IP lookup mechanisms in partial deployment environment. The tests are based on the percentages shown in figure

| | | | | | |
|----------|----------|----------|----------|----------|----------|
| 75% IPv4 | 60% IPv4 | 50% IPv4 | 40% IPv4 | 25% IPv4 | 5% IPv4 |
| 25% IPv6 | 40% IPv6 | 50% IPv6 | 60% IPv6 | 75% IPv6 | 95% IPv6 |

Table 4.5: Percentages of Prefixes

| | 500 | 1500 | 4500 | 9500 | 19500 | 39500 |
|------------------------------|-----|------|------|------|-------|-------|
| IPv4 | 375 | 900 | 2250 | 5700 | 4875 | 1975 |
| IPv6 | 125 | 600 | 2250 | 3800 | 14625 | 37525 |
| Binary Trie | 95 | 63 | 43 | 125 | 237 | 352 |
| Patricia Trie | 49 | 72 | 52 | 145 | 136 | 260 |
| Tree Bitmap | 16 | 25 | 32 | 36 | 54 | 95 |
| Optimized Tree Bitmap | 2 | 5 | 9 | 19 | 22 | 32 |

Table 4.6: Partial Deployment Table Data for IP Lookup

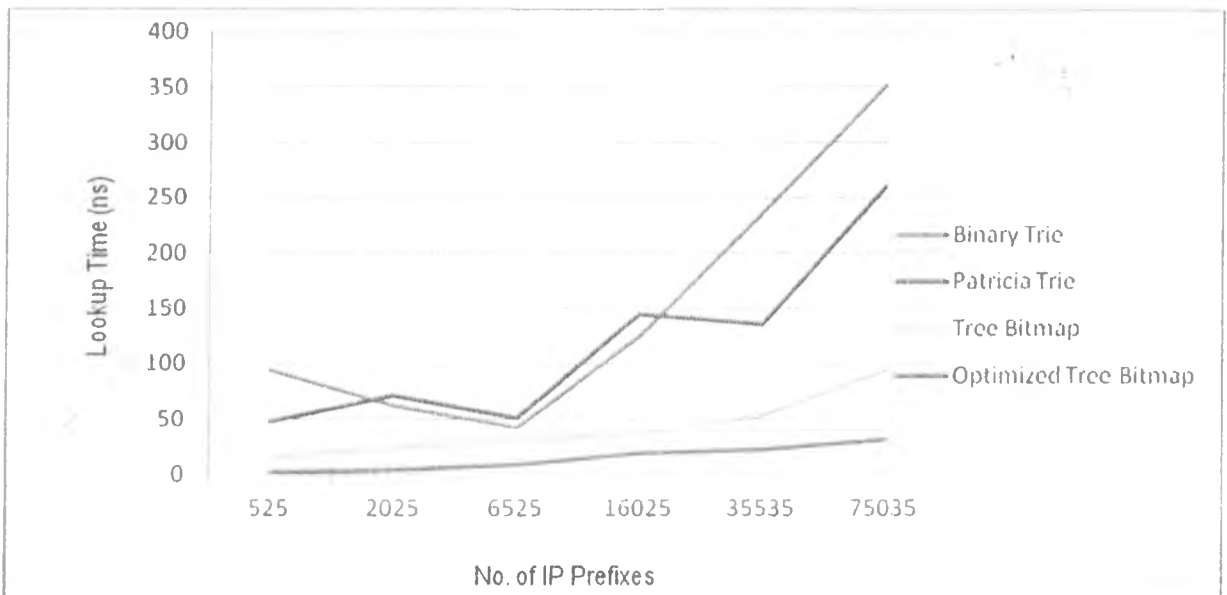


Figure 4.5: Lookup times when IPv4 and IPv6 contribute various percentages of the FIB.

An increasing number of IP addresses cause an enlargement of the trie data structure, and a consequent increase in searching times. This is clearly observable in Figures 4.5 and where the lookup values show a very similar behavior to those in Figure 4.2. The performance decay is not only caused by the increasing entries in the routing database, but also substantially due to the searching time increase in the cache memory since as the forwarding table size increase, there are very high likelihood of part of the data residing in a slow memory within the router.

| | 500 | 1500 | 4500 | 9500 | 19500 | 39500 |
|------------------------------|-------|-------|--------|--------|--------|--------|
| Optimized Tree Bitmap | 47828 | 39397 | 100401 | 161091 | 210877 | 144438 |
| Tree Bitmap | 13376 | 45997 | 101986 | 164662 | 213381 | 154608 |
| Patricia Trie | 14739 | 47393 | 103631 | 165900 | 214670 | 155846 |
| Binary Trie | 16104 | 48652 | 103691 | 167186 | 216033 | 157180 |

Table 4.7: Partial Deployment Table Data for Memory Consumption.

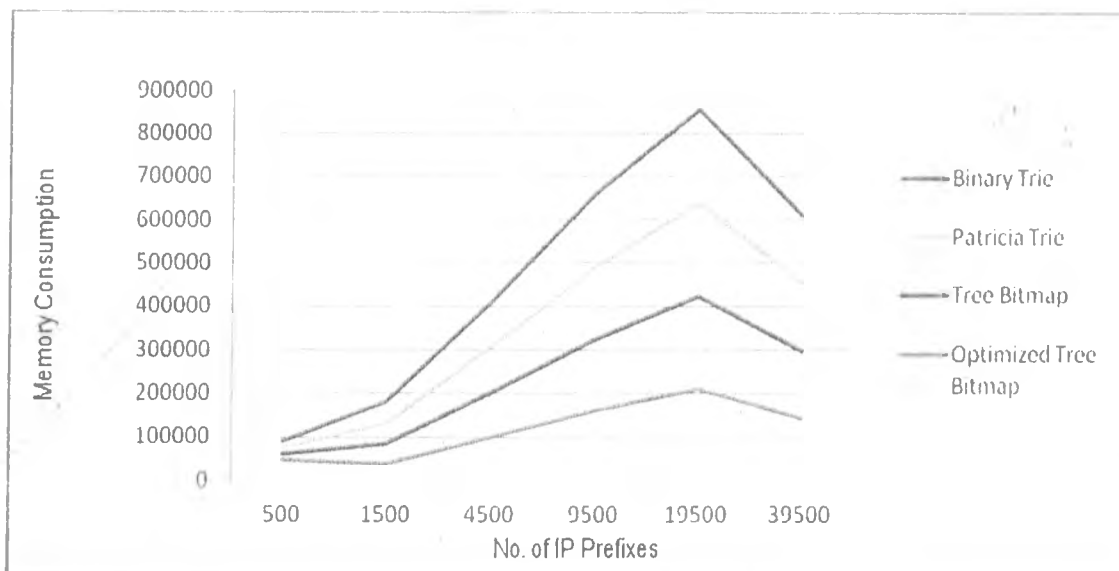


Figure 4.6: Memory required when IPv4 and IPv6 contribute various percentages of the FIB.

Figure 4.6 illustrates the differences in memory requirements for IP lookup mechanisms for variable routing table size and composition. While the memory requirements for Optimized Tree Bitmap algorithm is comparable to that of Tree Bitmap Algorithm, given the cost and availability of memory, the difference is again not extremely significant, e.g., about a factor of for the average case. This difference was to be expected since both schemes employ data structure compression mechanisms and reduced pointer usage for all the destinations. It is these techniques that lacks in Binary Tree and Patricia Trie schemes that corresponds to the differences shown in Figure 4.6.

The transition from IPv4 to IPv6 will cause a major increase in the lengths of addresses, and in the number of address prefixes which are used to aggregate IP addresses into networks. From Figure 4.5 and Figure 4.6, Binary Trie and Patricia Trie perform relatively well when the FIB is composed of shorter prefixes IPv4. As we move to IPv6 addresses, Binary Trie and Patricia Trie performance largely degrades because they are largely dependent on the lengths and size of the searched database, which is projected to grow significantly, and on the length of the address.

4.5.3 Scalability test

When talking about the scalability of an IP address lookup implementation, two different views can be considered. First it is the scaling to more prefixes in the routing table and second the scaling to longer addresses, 128 bit IPv6 addresses.

To investigate the scalability of the routing table, I ran simulations with duplicate scenarios (adding more traffic load, both IPv4 and IPv6 traffic) in order to derive some statistics about lookup time versus load and memory consumed versus load. To be more specific, I used the parameters as stated in Table 4.8, without changing the ratio of IPv4 to IPv6 but by constantly incrementing the scaling factor in each simulation scenario. Table 4.8 presents the composition of the prefix database that was used in the simulation. The first row presents the number of prefixes in the database which ranges from about five hundred over forty thousand. Rows two through six present the search times for the three algorithms.

The key measurement goal is to determine the capability to forward incoming IP addresses under heavy loading. Consequently, as traffic in the router increases the behavior of both lookup time and memory consumption for the different schemes are depicted below.

| | 500 | 2000 | 6500 | 15800 | 34300 | 59500 |
|---------------|-----|------|------|-------|-------|-------|
| Binary Tree | 28 | 79 | 83 | 59 | 109 | 296 |
| Patricia Trie | 73 | 39 | 58 | 82 | 166 | 212 |
| Multibit | 7 | 25 | 26 | 39 | 56 | 92 |
| Tree Bitmap | 2 | 6 | 8 | 9 | 17 | 22 |

Table 4.8: Scalability ratios for IP Lookup.

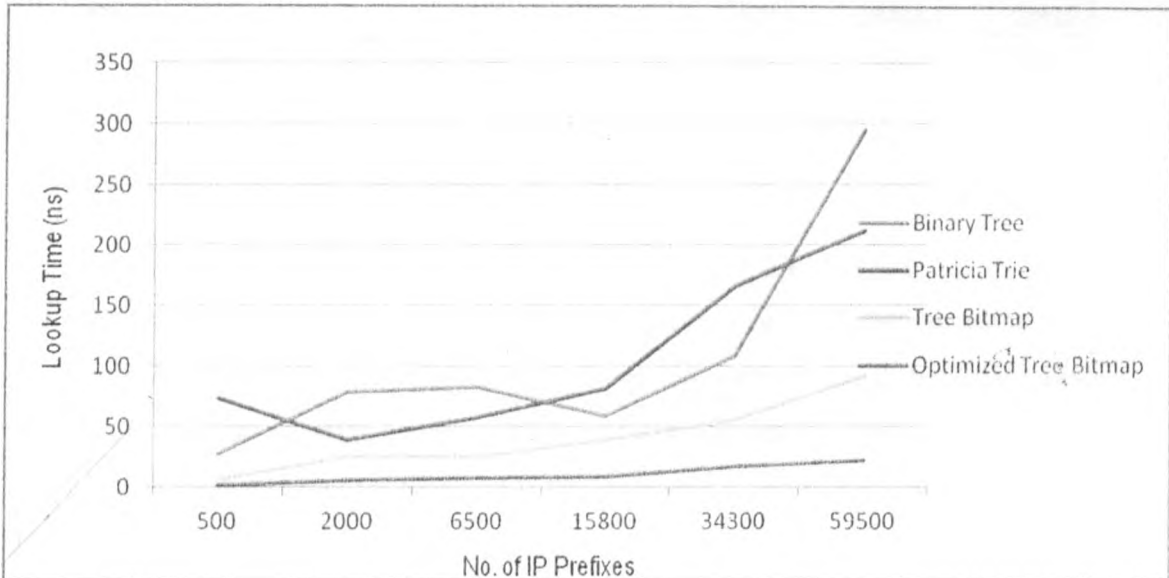


Figure 4.7: Variation of lookup times as number of Prefixes increases.

Looking at the results of the simulation in figure 4.7, one simple observation is that as the number of prefixes increases, the lookup time also increases. The time take to lookup a destination IP address increases sharply after the number of prefix when the number of prefixes is greater than 15800. The ratio of lookup time seems to increase as the database gets larger. For future databases which are expected to grow exponentially, it could be extrapolated that the Optimized Tree Bitmap is likely to perform better than all the other algorithms. Binary Trie performance better in lookup time than Patricia Trie. This difference is attributed to the fact that one way branch nodes were eliminated making the lookup faster.

The second scalability issue is harder to investigate, since there is a lack of differentiation between the memory consumed during data structure construction and the memory consumed during lookup. Over 34000 distinct prefixes are generated and the simulation

gives the results of table 4.9. This factor makes it hard to tell how well this implementation will handle very large IPv6 tables. What is clear, however, is that the optimized algorithm performs well even with IPv6. How the memory consumption is affected is dependent on the number of prefixes.

| | 500 | 2000 | 6500 | 15800 | 34300 |
|--------------------------|------|------|-------|-------|-------|
| Binary Tree | 1585 | 6342 | 12590 | 20408 | 27895 |
| Patricia Trie | 700 | 4623 | 8071 | 14264 | 24562 |
| Multibit | 367 | 650 | 2563 | 6782 | 11864 |
| Tree Bitmap | 250 | 285 | 736 | 1325 | 5620 |
| Path Compression Savings | 24% | 35% | 16% | 23% | 21% |

Table 4.9: Scalability ratios for Memory Consumption.

The first row of Table 8 presents the number of prefixes in the database which ranges from about five hundred over thirty thousand. Rows two through five present the memory occupied by the data structure as organized by the four algorithms. Memory consumed by binary tree is relatively large since there is no compression mechanism and a lot of space is also wasted by empty nodes.

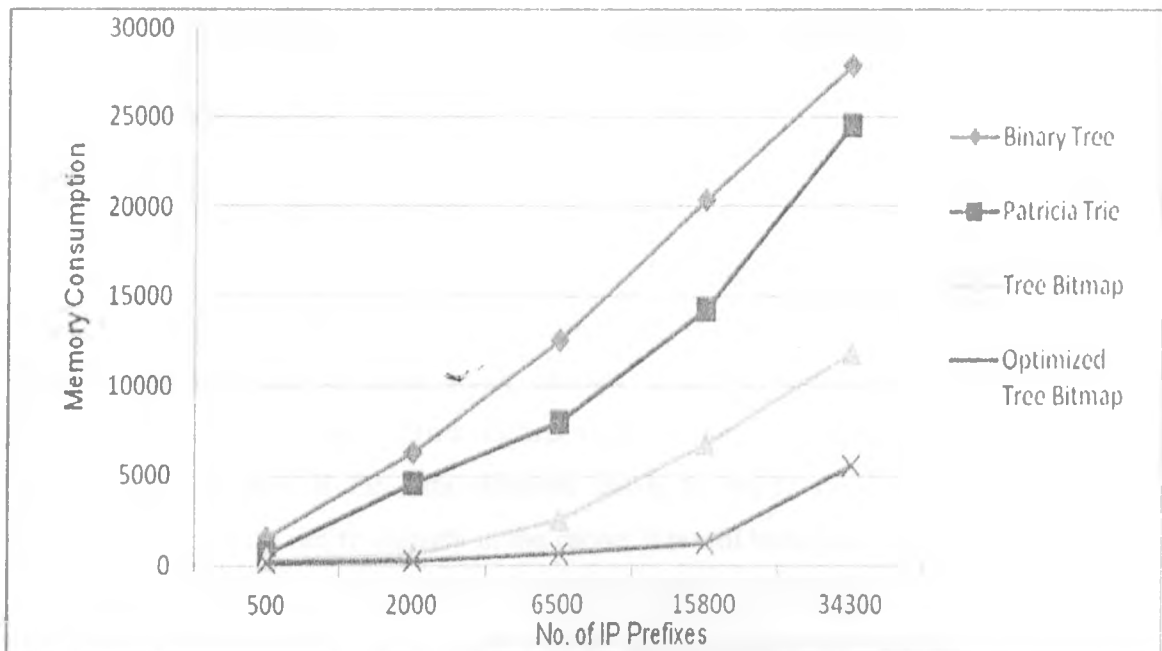


Figure 4.8: Variation of memory requirement as IP Prefixes increases.

From the graph above, it is observed that though there is a rapid increase in memory consumed as the number of IP prefixes increases. Optimized Tree Bitmap algorithm performs better than the other algorithms since the size of the next hop pointer is smaller than for other schemes. Notice that for Optimized Tree Bitmap algorithm, increase is not as rapid as the other algorithms. This implies that the algorithm is more efficient with larger databases with optimization that reduces wastage of unused nodes.

In general, algorithms are expected to scale both in speed and memory as the size of the forwarding table increases. While core routers presently contain as many as 200,000 prefixes, it is expected to increase to 500,000 to 1 million prefixes with the possible use of host routes and multicast routes. When routers are deployed in the real network, the service providers expect them to provide consistent and predictable performance despite the increase in routing table size. This is expected since a router needs to have a useful lifetime of at least five years to recuperate the return on investment. Optimized Tree Bitmap algorithm is a good bet for investment.

4.6 Discussions

When designing the data structure used in the forwarding table, the primary goal was to minimize lookup time. To reach that goal, an attempt was made to simultaneously minimize two parameters;

1. the number of memory required during lookup, and
2. 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 slow memory. 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.

Starting with the memory requirements, I observe that among the various trie structures the Optimized Tree Bitmap generally consumes the least amount of memory for all kinds of input. No other algorithm organizes a data structure to use less space than the Tree Bitmap algorithm.

The IP prefixes in real routers have the same distribution as the distribution used in these experiments and hence the results for this input are very similar to the results for real data. In all the experiments, the average depth in a Optimized Tree Bitmap is considerably smaller than in any of the other tries. This is the single most one reason why the lookup speeds are averagely faster in Optimized Tree Bitmap than other data structure. For large data, the time measurements were greatly affected by the virtual memory assignment policy of the Java virtual machine.

The algorithms spend some time in building the structures. Each time a set of IP prefixes is generated, a reconstruction of the trie takes place. This time is also is taken care of since it is not included in the final computation for lookup though it would have been nice to measure the buildup time. As shown above, the performance of the lookup is highly dependent on the memory access time. So it is crucial to keep the size of the memory as small as possible and definitely below the limit for embedded SRAMs which is some hundred kilobytes.

In summary, with respect to lookup time, memory requirements and scalability of algorithms, the Tree Bitmap algorithm is a clear winner among the various trie structures.

CHAPTER 5

CONCLUSION

5.1 Conclusion

The rapid growth of Internet Traffic causes increase in size of routing table. The current day routers are expected to perform longest prefix matching algorithm to forward millions of datagram each second, and this demand on router is increasing even while the prefix search database is expanding in both the dimensions, i.e., IP address length (128 bits for IPv6) and number of prefixes. When there is migration from IPv4 to IPv6, the routing table size increases exponentially. So this project has modified the Tree Bitmap Algorithm implemented in Cisco-CRS1 routers to reduce lookup latency. This provides an efficient way of searching through forwarding tables.

The goal of this project is to detail a new algorithm for IP Lookups called Optimized Tree Bitmap and then illustrate its capability using a simulation model. The simulation model is very understandable and easily adaptable to different IP Lookup algorithms. The model provides a very easy and fast tool to perform various simulations and analyze the results providing a good analysis and understanding of the search schemes.

A new algorithm for IP lookup is presented. This algorithm is optimized for software implementation and it balances fast lookups and small memory. It contains both intellectual and practical contributions. On the intellectual side, after the basic notion of Tree Bitmap Algorithm, I found that I had to push single prefixes that required separate trie node to be created with bitmaps that are almost completely unused. I singled out that pushing the prefix to the parent has an aesthetically pleasing idea that leverages off the extra structure inherent in the particular form of tree bitmap.

On the practical side, I have a fast, scalable solution for IP lookups that can be implemented in software reducing the number of expensive memory accesses required

considerably. I expect most of the characteristics of address structure to strengthen in the future especially with the imminent transition to IPv6. Even if my predictions based on the little evidence available today should prove to be wrong the overall performance can easily be restricted to that of the optimized algorithm which will perform well.

This research work therefore contributes a solution towards alleviating the problem of longest matching prefix as identified in the research problem in section 1.2.

Lookup time measured for the lookup schemes reflects the dependence on the prefix length distribution. A large variance between time for short prefixes and time for long prefixes is observed because of the height of the data structure constructed by each scheme. For Binary trie and Patricia trie, the height is high. On the contrary, the full expansion/compression scheme employed by tree bitmap and optimized tree bitmap reduces the height considerably always needs fewer memory accesses. Optimized tree bitmap scheme has the best performance for the lookup operation in this experiment. Small variations should be due to cache misses as well as background operating system tasks.

We believe that trie-based schemes will continue to dominate in IPv4-based products. However, the slow, but ongoing, trend towards IPv6 will give a strong edge to schemes scalable in terms of prefix lengths. Optimized Tree Bitmap will be a strong candidate for the transition. Except for tables where path compression is very effective, I believe that this algorithm will be better than trie-based algorithms for IPv6 routers. Tree Bitmap was adopted in the Cisco router in anticipation of such a trend. Optimized tree bitmap will do even better.

5.2 Recommendation and future work.

For future work, I will be attempting to fine tune the algorithm to separate build up time from search time apart from modification processes. I will also be looking for other applications of the algorithm. Thus I will also be studying the effects of caching in lookup based on the optimized algorithm. Besides packet classification, I believe that this algorithm and its improvements may be applicable in other domains besides Internet

packet forwarding. Potential applications that are worth investigating include memory management using variable size pages, access protection in object-oriented operating systems, and access permission management for web servers and distributed file systems.

BIBLIOGRAPHY

APNIC and Routing Registries 2002. IPv6 address allocation and assignment policy. [Online], available at: http://www.arin.net/policy/archive/ipv6_policy.html [Accessed January 2012].

D. Medhi and K, Ramasamy, 2007. Network Routing: Algorithms, Protocols and Architectures. Morgan Kaufmann Publishers, Part 5.

George Varghese, 2012. Recent Research Directions in IP Lookup Algorithms, [Online] available at: <http://www-cse.ucsd.edu/users/varghese/research.html> [Accessed February 2012].

Internet Performance Measurement and Analysis Statistics, [Online] available at: <http://nic.merit.edu/ipma> [Accessed February 2012].

IPv6 information page, [Online] available at: <http://www.ipv6.org> [Accessed December 2011].

H. Jonathan Chao, Sept 2002. Factors to consider in Building Next Generation Routers in: Technical Report Next Generation Routers, Proc. IEEE, vol.90, no.9, pp.1518–1588.

M. A. Ruiz-Sanchez, E. W. Biersack, W. Dabbous, 2001. Survey and Taxonomy of IP Address Lookup Algorithms. Journal of IEEE Network, Vol.15, Issue 2, PP.8-23.

Methodology for Modelling Wireless Routing Protocols Using Opnet Modeller. Rowan University 201 Mullica Hill Rd. Glassboro, NJ 08062 USA, PP2.

Michigan University and Merit Network. Internet performance and analysis (ipma) project. <http://www.merit.edu>.

M. Waldvogel. Fast Longest Prefix Matching: Algorithms, Analysis, and Applications. PhD thesis, Swiss Federal Institute of Technology - Zurich, 2000.

M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high-speed ip routing lookups. In Proceedings of ACM Sigcomm, pages 25–36, October 1997.

N. McKeown, “Fast Switched Backplane for a Gigabit Switched Router”, <http://www.cisco.com/warp/public/733/12000/technical.shtml>.

M. Degermark, A. Brodnik, S. Carlsson, S. Pink, “Small Forwarding Tables for Fast Routing Lookups” Proc. ACM SIGCOMM ‘97, , Cannes (14 - 18 September 1997).

M. Waldvogel, G. Varghese, J. Turner, and B. Plattner 1997. Scalable high speed IP routing lookups. In Proc.SIGCOMM’97, Cannes, France. PP

Niall Murphy, Google, and David Wilson, “The End of Eternity: IPv4 Address Exhaustion and Consequences”, The Internet Protocol Journal, Volume 11, No. 4, [Online] available at <http://www.cisco.com/web/about/ac123/ac147/archived/ipj_11-4/114_eternity.html> [Accessed December 2011].

OPNET Modeler ver. 14.5 OPNET Technologies, Inc®, [online] available at <www.opnet.com> [accessed February 2012].

S. Keshav and Rosen Sharma 2008. Issues and Trends In Router Design¹, Cornell University. Journal of IEEE Communications Magazine. PP.1-5.

S. Nilsson and G. Karlsson, 1998. Fast Address Look-Up for Internet Routers In: pro. IEEE Broadband Communication, PP 2-5.

University of Oregon Advanced Network Technology Center, Route views project, [Online] available at: <<http://www.routeviews.org/>> [Accessed December 2011].

Vasil Hnatyshin, Hristo Asenov, and John Robinson 2011. Challenges in performance measurements of routers forwarding functions using Opnet Modeller in: Practical

V. Fuller et al 1993. Classless Inter-Domain Routing (CIDR): an address assignment and aggregation strategy. RFC1519.

V. Srinivasan and G. Varghese, 1998. Faster IP Lookups Using Controlled Prefix Expansion, Measurement and Modeling of Computer Systems, 1998, vol. 17. PP.

V. Srinivasan, S. Suri, and G. Varghese, 2009. Packet Classification and Lookup models in: Packet classification using tuple space search, ACM SIGCOMM Computer Communication Review, vol.29, no. 4, pp.135–146.

V. Srinivasan and G. Varghese. Faster ip lookups using controlled prefix expansion. ACM Transactions on Computer Systems, 17(1):1–40, February 1999.

W. Wu, Packet Forwarding Technologies 2009: Auerbach Publications, Taylor and Francis Group. Chp 1-4.

Wei, G., Chunhe, X., Nan, L., Haiquan, W., and in, D. (2007). Research on simulation framework of structured Network. In FGCN '07: Proceedings of the Future Generation Communication and Networking, Washington, DC, USA. IEEE Computer Society.

Y. K. Li and D Rao, 2006. Address Lookup algorithms for IPv6”, IEEE Communications Magazine, Vol 153, No.6.