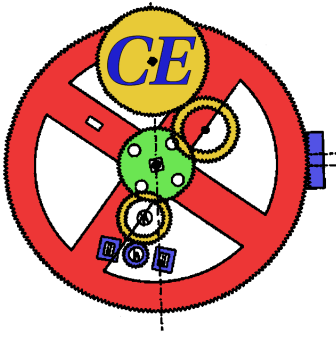




MSc THESIS

Longest Prefix Match and Incremental Updates for Range Tries

Sri Harsha Katamaneni

Abstract

CE-MS-2010-22

Address lookup has become a major bottleneck in the performance of today's network routers with rapid increase in the urge for faster and high bandwidth communication. The most crucial metrics for a lookup algorithm being lookup speed, scalability and memory usage, the Range Trie algorithm provided the much needed solution to the diminishing benefits of the current address lookup algorithms. The proposed design provides Longest Prefix Match and Incremental Update support to the existing Range Trie method. The design offers faster update rates maintaining the inherent properties of the Range Trie with increasing address width and routing table size. The design only requires D memory accesses for address lookup and $4 \cdot D$ memory accesses ($2 \cdot D$ read and $2 \cdot D$ write) for an update operation where D =depth of the tree. Each update operation requires a bubble of 4 cycles in the pipeline. The pipelined hardware design is prototyped on a Xilinx ML410 Embedded development platform (Virtex4 XC4VFX60). The Range Trie 90-nm ASIC implementation can perform 630 million lookups with no updates and more than 625 million lookups with upto 1 Million updates per second. The Range Trie design outperforms the existing designs showing better scalability and performance with minimum resource requirements.

Longest Prefix Match and Incremental Updates for Range Tries

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Sri Harsha Katamaneni
born in Khammam, India

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Longest Prefix Match and Incremental Updates for Range Tries

by Sri Harsha Katamaneni

Abstract

Address lookup has become a major bottleneck in the performance of today's network routers with rapid increase in the urge for faster and high bandwidth communication. The most crucial metrics for a lookup algorithm being lookup speed, scalability and memory usage, the Range Trie algorithm provided the much needed solution to the diminishing benefits of the current address lookup algorithms. The proposed design provides Longest Prefix Match and Incremental Update support to the existing Range Trie method. The design offers faster update rates maintaining the inherent properties of the Range Trie with increasing address width and routing table size. The design only requires D memory accesses for address lookup and $4 \cdot D$ memory accesses ($2 \cdot D$ read and $2 \cdot D$ write) for an update operation where D =depth of the tree. Each update requires a bubble of 4 cycles in the pipeline. The pipelined hardware design is prototyped on a Xilinx ML410 Embedded development platform (Virtex4-FX60). The Range Trie 90-nm ASIC implementation can perform 630 million lookups with no updates and more than 625 million lookups with upto 1 Million updates per second. The Range Trie design outperforms the existing designs showing better scalability and performance with minimum resource requirements.

Laboratory : Computer Engineering
Codenumbr : CE-MS-2010-22

Committee Members :

Advisor:	Dr. Ioannis Sourdis, CE, TU Delft
Chairperson:	Dr. K.L.M. Bertels, CE, TU Delft
Member:	Dr.ir. Stephan Wong, CE, TU Delft
Member:	Dr. Christian Doerr, NAS, TU Delft
Member:	Dr.ir. Georgi N. Gaydadjiev, CE, TU Delft

*To my grandmother, my parents and all those who are helping me
in the process of being perfected*

Contents

List of Figures	viii
List of Tables	ix
Acknowledgements	xi
1 Introduction	1
1.1 Problem Statement	2
1.2 Motivation	3
1.3 Goals and Objectives	5
1.4 Thesis Overview	5
2 Background	7
2.1 Related Data Structures for Address lookup	7
2.1.1 Search on length	7
2.1.2 Search on values	9
2.2 Range Trie for Address lookup	9
2.2.1 Range Trie Rules	10
2.2.2 Range Trie Structure	14
2.2.3 Lookup Procedure	16
2.3 Range Tree Data Structures for LPM and Updates	17
2.3.1 Multiway Range Tree	17
2.3.2 B-Tree	18
2.3.3 Dynamic Segment Tree	18
2.4 Summary	18
3 Range Trie with LPM and Dynamic Updates	21
3.1 Issues involved	21
3.2 Associating prefixes to nodes	24
3.3 Longest Matching Prefix	26
3.4 Dynamic Updates	28
3.4.1 Prefix Traversal	28
3.4.2 Inserting a prefix	31
3.4.3 Deleting a prefix	34
3.5 Spare levels	35
3.6 Rules to update the Range Trie data structure	38
3.7 Complete Range Trie Structure	39
3.8 Summary	42

4	Range Trie Hardware Design	43
4.1	Background	44
4.1.1	Processing Unit	44
4.1.2	Memory Structure	47
4.1.3	Node data structure	48
4.2	Pipeline Structure	49
4.3	Updates	50
4.3.1	Prefix Traversal	50
4.3.2	Storing prefix lengths and action pointers:	53
4.3.3	Updating Prefix lengths	54
4.3.4	Updating Action pointers	56
4.4	Spare levels	58
4.4.1	Memory structure	59
4.4.2	Processing unit	63
4.5	Complete Range Trie pipeline	64
4.5.1	Iteration stage in the fixed Range Trie	65
4.5.2	Iteration stage in the spare levels	66
4.6	FPGA Prototype	67
4.7	Summary	68
5	Experimental Results	71
5.1	Random Input	71
5.2	Real data	74
5.2.1	Real IPv4 routing tables	74
5.2.2	Real IPv6 routing tables	77
5.3	Comparison	79
5.3.1	For synthetic routing tables	80
5.3.2	For Real routing tables	83
5.3.3	For IPv4 routing tables projected to IPv6	85
5.4	Update results	87
5.5	Summary	89
6	Conclusion	93
6.1	Summary	93
6.2	Contributions	95
	Bibliography	98

List of Figures

1.1	Wide Area Network	1
1.2	A Network Router	2
1.3	Address lookup problem	2
1.4	Ranges Stored as Prefixes for LPM	3
1.5	Projected Internet traffic[3]	4
2.1	Search on Length : Trie	8
2.2	Search on Values : Range Tree	9
2.3	A Range Trie node	10
2.4	An example for Rule1	12
2.5	An example for Rule2	12
2.6	An example for Rule3	13
2.7	An example for Rule4	13
2.8	An example for Rule5	14
2.9	An example for Range Trie structure	15
2.10	An example for Range Trie node and its children	16
3.1	Storing prefixes 1	22
3.2	Storing prefixes 2	23
3.3	Handling Action pointers	23
3.4	Processing non-existing bounds	24
3.5	Prefix storing rule 1	25
3.6	Prefix storing rule 2	26
3.7	Associating prefixes to nodes	27
3.8	Range Trie structure supporting LPM	28
3.9	Range Trie showing the nodes in the range of prefixes	29
3.10	Prefix Traversal in the Range Trie	30
3.11	Storing Action Pointers and Action in Range Trie Structure	31
3.12	Update of a prefix in a Range Trie	32
3.13	Node split when a prefix is inserted	33
3.14	Inserting non-existing bounds	34
3.15	Deleting a prefix resulting in a node merge	35
3.16	Sub-tree in spare level	37
3.17	Rebuilt sub-tree in spare level	37
3.18	Node structure in the Range Trie	40
3.19	Node structure in the spare levels	41
3.20	Complete Range Trie Structure	41
4.1	Processing Unit	45
4.2	Memory organization	47
4.3	Node data structure	49
4.4	One level in the Range Trie Structure	49

4.5	Pipeline of the initial design	51
4.6	Prefix Traversal in the Range Trie	52
4.7	Prefix Traversal in the pipeline for the scenario of Figure 4.6	53
4.8	Modified node data structure in fixed part of the Range Trie	54
4.9	Updating prefix lengths in the Range Trie Structure	55
4.10	Updating action pointers in the Range Trie Structure	57
4.11	Finding a new action pointer	58
4.12	Node data structure in spare levels	60
4.13	Memory allocation in spare levels	61
4.14	Memory de-allocation in spare levels	61
4.15	Block Diagram of Spare memory	63
4.16	Inserting a new bound into a node in the spare levels	64
4.17	Logic in Processing Unit to insert or delete bounds into the node	65
4.18	Block Diagram of Iteration stage in the static Range Trie	65
4.19	Block Diagram of Pipeline structure of one level of sub-tree in spare levels	67
4.20	Block Diagram of the prototype	68
5.1	Number of tree levels with varying routing table sizes	72
5.2	Memory with varying routing table sizes	73
5.3	IPv4 prefix distribution in rrc06 Exchange point	74
5.4	Number of tree levels for real routing tables	76
5.5	Memory requirements for real routing tables	76
5.6	Number of tree levels of real routing tables of IPv6	77
5.7	Memory requirement for real IPv6 routing tables	78
5.8	Number of tree levels(latency) for synthetic IPv4 routing tables	81
5.9	Memory requirement for synthetic IPv4 routing tables	81
5.10	Number of tree levels(latency) for synthetic IPv6 routing tables	82
5.11	Memory requirement for synthetic IPv6 routing tables	82
5.12	Number of tree levels(latency) for real IPv4 routing tables	83
5.13	Memory requirement for real IPv4 routing tables	84
5.14	Number of tree levels(latency) for IPv4 routing tables projected to IPv6	86
5.15	Memory requirement IPv4 routing tables projected to IPv6	86
5.16	Update requests for a period of a year	87
5.17	Number of sub-tree rebuilds over a period of a year	88
5.18	Overhead time due to updates	88
5.19	Performance overhead with real time updates	89
5.20	Overhead when fixed number of updates are fed	90

List of Tables

5.1	Uniform Distribution:IPv4 Routing Tables	72
5.2	Memory requirement of routing tables with uniformly distributed prefixes	73
5.3	IPv4 Routing Tables	75
5.4	Memory real IPv4	75
5.5	IPv6 Routing Tables	77
5.6	Memory for real IPv6 routing tables	78
5.7	Comparison of synthetic IPv4 routing tables	80
5.8	Comparison for real IPv4 routing tables	83
5.9	Comparison for real IPv4 routing tables projected to IPv6	85
5.10	Complexity Comparison	90

Acknowledgements

Most of all, I would like to thank my advisor Ioannis Sourdis for his invaluable support and guidance and for his belief in me to take up the thesis topic.

I would also like to thank Gerooge Stefanakis and Ruben de Smet who helped in transferring their knowledge in the initial stages of my thesis.

Sri Harsha Katamaneni
Delft, The Netherlands
August 30, 2010

Introduction

1

The Internet has empowered the modern societies to share information that transformed the pace of human life. Its proliferation has created the urge for faster and high bandwidth communication. Apart from computer systems and personal computers, smart phones and hand-held devices overload the world-wide-area networks consuming large part of their bandwidth as shown in Figure 1.1.

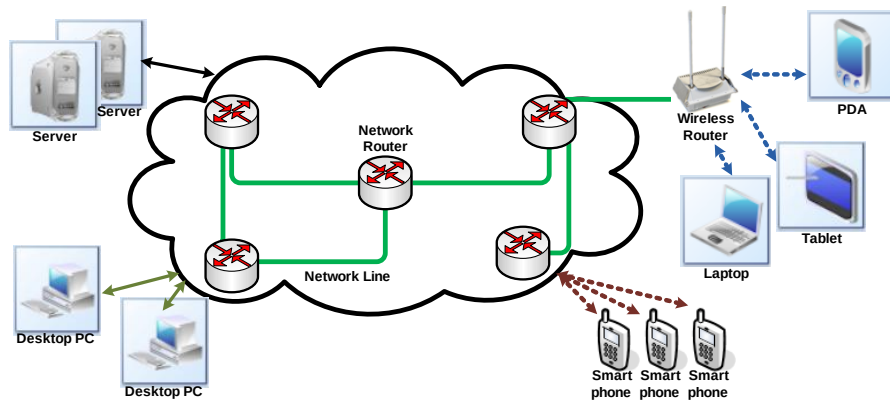


Figure 1.1: *Multiple devices connected to the Internet*

Network traffic travels through network lines and in order to find its destination needs to be properly routed through high-speed network routers. As illustrated in Figure 1.2, a network router processes each incoming network packet and determines its correct network-path using a forwarding engine. The forwarding engine finds the correct output port for each packet to be sent after searching in a routing table using the destination address of a packet; this function is also called address-lookup. Subsequently, a switch fabric will guide the packets to the output.

The rest of the section is organized as follows: Section 1.1 provides a brief description of the problem being addressed in this thesis. The motivation behind the thesis is presented in section 1.2. Section 1.3 states the goals of this thesis and its contributions to the Range Trie structure. An overview of the thesis is provided in section 1.4 that concludes the chapter.

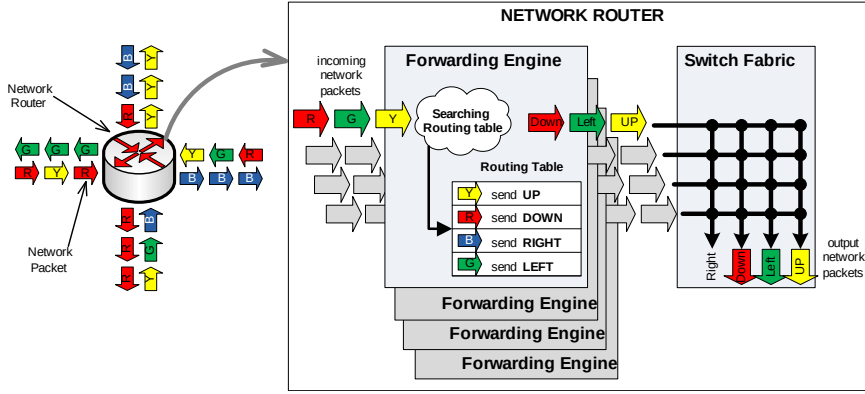


Figure 1.2: A Network Router is the crossroad of multiple Network Lines; it consists of multiple Forwarding Engines and Switch Fabrics. A Forwarding engine finds the proper output to which an incoming network packet will be sent; this is achieved by searching in a Routing Table

1.1 Problem Statement

IP address lookup is one of the significant functions of a network router. It involves searching the routing table using the destination address of an incoming packet. Each entry in the routing table corresponds to an address range in the IP address space. The k unique address bounds B_k generated from the available set of unique prefixes divides the address space $[0, 2^n)$ into $k+1$ variable size address ranges R_j where $j = 1, 2, \dots, k+1$ as shown Figure 1.3. The basic address lookup problem is to determine the range R_j to which the incoming address A_{IN} (destination address of a packet) belongs to.

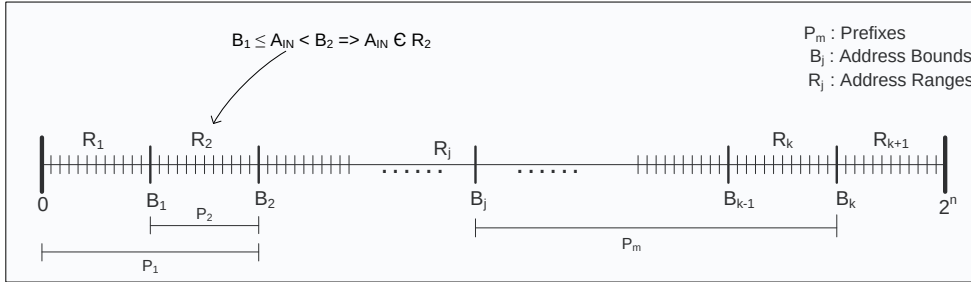


Figure 1.3: Address lookup problem

When IP was first standardized in 1981[2], the address space was divided into three classes namely Class A, B and C with fixed size prefixes. However, with the unimaginable growth of internet, the size of routing tables increased rapidly with increasing number of networks. Inefficient distribution of address space led to fast exhaustion of the address space. To address the problem of bloating routing table sizes and for efficient use of the address space a new scheme called *Class-less Inter Domain Routing* (CIDR)[13] was introduced in 1993. With this new scheme, length of the prefix address is no longer

fixed. Now, a router has to perform lookup based on the variable sized address prefix. As several routing table entries might match a certain prefix, a router must identify the best desirable match, which would be the *Longest Prefix Match* (LPM). Now, the address ranges are stored as prefixes as shown in Figure 1.4.

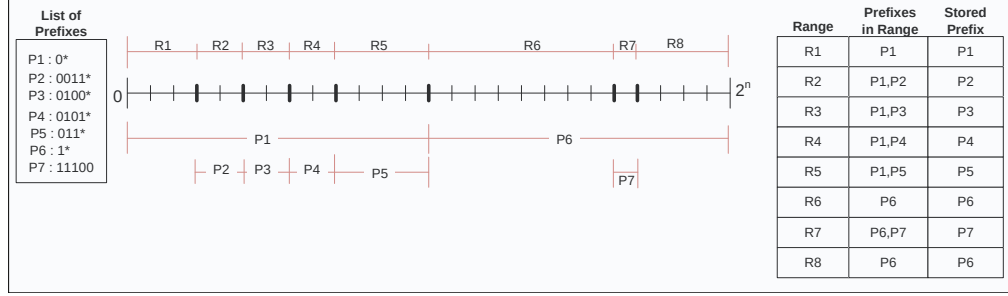


Figure 1.4: Ranges Stored as Prefixes for LPM

The novel Range Trie algorithm introduced by I.Sourdis in [23] performs address lookup based on exact range match. Since the current network routers perform lookup based on LPM, the Range Trie data structure should be modified to support LPM.

Today's network routers use Border Gateway Protocol (BGP) to address the task of path determination [1]. BGP is an incremental protocol that sends network updates only when there is a change in the network. A BGP update can either be an announcement that indicates a new network addition or a withdrawal that indicates the network is no longer reachable[17]. Thus *a router should have the ability to handle incremental updates*.

A routing table stores information about the topology of the computer network. A good routing algorithm should be able to adapt dynamically to the changes in the network topology. Whenever there is a change in the topology of the network routing tables should be dynamically updated at run time. Since the time and effort spent on updating the routing table can adversely affect the network performance, it is desirable to have an update mechanism to handle fast update rates. The Range Tries should also support incremental update to be an integral part of real world network routers.

1.2 Motivation

Address lookup operation is a major bottleneck in today's network routers[5]. With the increasing difficulty for wide-area networks to keep pace with the network evolution and to provide efficient high-speed and high-bandwidth network solutions and higher network bandwidth requires increasingly more searches (address-lookups) per second. ***The increasing growth of the use of end-network devices created a rapid increase in network traffic***(Figure 1.5) and the need for independent identification of these devices populates the routing tables increasing them in size. This emphasized

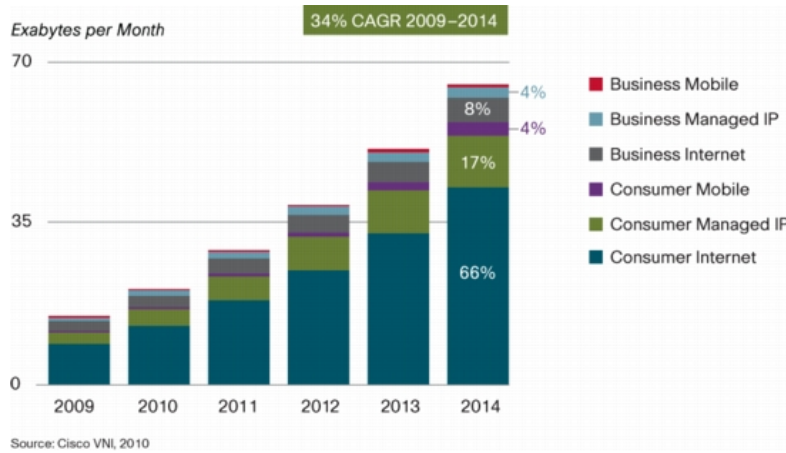


Figure 1.5: Projected Internet traffic[3]

the need for high performance routing algorithms.

The number of devices connected to Internet is rapidly increasing the size of routing tables. Due to the exponential increase in the geographical use of internet, the IPv4 address space is close to exhaustion. The unique IP-addresses, used for identifying each device connected to the Internet, will need to become wider in order to accommodate the increasing number of network devices, thus the IPv4 addresses would soon be replaced by the four times wider IPv6 ones. *Wider IP-addresses is the most significant limitation for building forwarding engines and is expected to significantly increase their cost and latency.* The use of IPv6 addresses, the rapidly increasing size of routing tables as well as the increasing needs for higher network bandwidth introduce challenging requirements for future forwarding engines. Based on current research of Internet number resources, APNIC¹ believes the pool of unallocated IPv4 address space will be consumed at some point in 2010 or 2011. After that time the cost and complexity of operating a stable and growing Internet will escalate significantly, unless IPv6 deployment is well underway ².

The existing IP lookup algorithms in literature do not scale well with increase in address width and increasing routing table sizes. Some of these algorithms will be discussed in the next chapter. In the present scenario, the Range Trie algorithm[23] is one such algorithm that shows better scalability and high performance. The motive behind the thesis is to enable the Range Trie algorithm to be a part of real world network routers which is only possible when it supports LPM and dynamic updates.

¹<http://www.apnic.net>

²<http://www.apnic.net/community/ipv6-program/ipv4-exhaustion>

1.3 Goals and Objectives

The Range Trie method introduced in [23] performs lookup using exact range matching. In the real world view, a lookup is performed to find the longest matching prefix of an incoming address as described in section 1.3.1. The goal of the thesis is to adapt the existing Range Trie algorithm to support Longest Prefix Match and handle the incremental updates dynamically, maintaining the inherent characteristics of the Range Trie structure such as (a) low lookup latency (b) high throughput (c) low memory requirements (d) better scalability in terms of address width.

The main objectives of this thesis are:

- ***Support for Longest Prefix Match*** : To devise an efficient method to store the prefixes in the Range Trie structure to perform lookups with Longest prefix match avoiding the redundant storage of prefixes in the data structure.
- ***Fast Dynamic Updates*** : To design a specialized hardware design for faster update rates.
- ***A complete pipelined hardware design of the algorithm*** : The hardware design is completely pipelined to handle IP address lookups and to handle update operations. The specialized hardware for LPM and updates is positioned in the pipeline such that the operating frequency of the design does not differ from the initial design that performed exact range match lookup. So there is no latency overhead due to the updated hardware design.
- ***FPGA Prototype of the hardware design***

A detailed description of the contributions are presented in the subsequent chapters of this thesis.

1.4 Thesis Overview

This chapter provided an overview of the problem network routers are facing in the present world. It also presented the need to support longest prefix match and the need for faster update times. The rest of the section provides an overview of the following chapters.

A brief overview of the existing address lookup algorithms and the Range Trie algorithm will be presented in Chapter 2. In chapter 3, the design details in order to support LPM and dynamic updates in the Range Trie will be presented. A detailed hardware description and implementation of the new design will be presented in Chapter 4. Chapter 5 evaluates the performance, memory requirements and update rates of the new Range Trie structure.

Chapter 6 concludes the thesis summarizing the contributions to the thesis and presenting the possible future contributions to the present work.

Background

The exponential growth in the Internet usage and the rapid rise in internet based devices lead to the fast depletion of IPv4 address space. This emphasized the need to introduce a new addressing scheme with an increase in address width, which would probably be IPv6 that is 4 times the address width of the current IPv4 addresses. The existing lookup algorithms lack scalability as their memory requirements and the latency increase linearly with the growing address width. Apart from this, these algorithms are lagging behind to withstand the growing internet speeds, internet traffic and increasing routing table sizes.

Some of the noteworthy algorithms for address lookup proposed in the past have been summarized in the survey by Ruiz-Sanchez et al. in [21]. Following the same taxonomy of [21], we give a glimpse of those approaches in section 2.1. Later, the Range Trie algorithm that performs lookup based on exact range matching which is the basis for this thesis is presented in section 2.2. Section 2.3 discusses some related work in the existing literature that support LPM and updates. A brief summary is provided in section 2.4.

2.1 Related Data Structures for Address lookup

The address space $[0, 2^n)$ where n =address width(IPv4=32,IPv6=128 etc.), is organized as a set of ranges. These ranges are defined either as a set of intervals where full length addresses are compared to perform lookup or extracted from a set of prefixes where lookup result is the longest matching prefix. Address lookup involves searching in two dimensions: (a) length and (b) value, as mentioned in [21]. Based on this, the existing address lookup schemes are classified in two approaches, *search on length* and *search on value*.

2.1.1 Search on length

A Trie is a tree based data structure allowing the organization of prefixes on a digital basis by using the bits of prefixes to direct the branching[21]. A position in the Trie corresponds to the length of the prefix being compared. That means at each level, all the descendants of a node share a common prefix. A lookup starts search for prefixes of length 1 at level 1, and continues until finding the better match i.e the search ends when there are no more branches to traverse the Trie. An example is shown in Figure 2.1. The depth of the Trie is the length of the longest prefix which in the worst case is the address width.

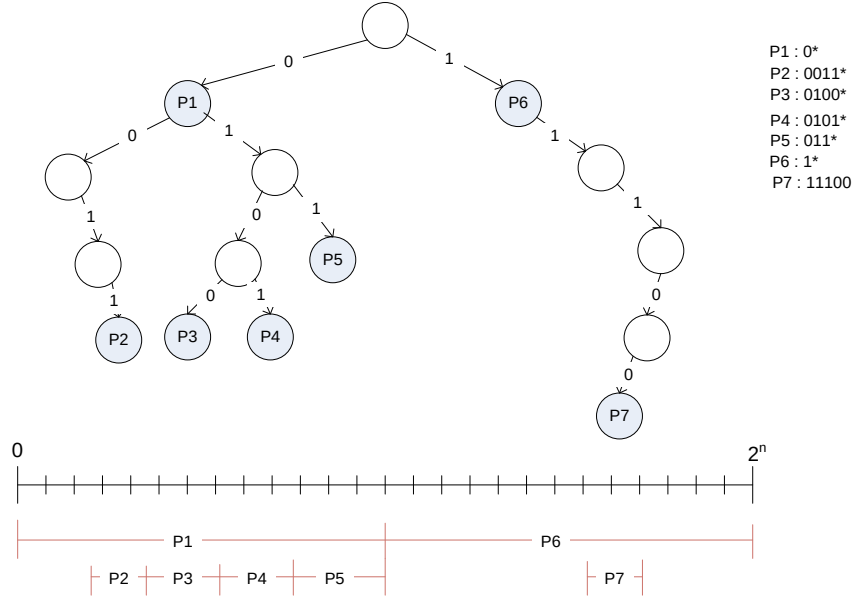


Figure 2.1: Search on Length : Trie

Multibit tries [25] using controlled prefix expansion speeds up the search by inspecting multiple bits simultaneously at each node. Lookup is faster as the tree depth decreases due to the increase in branching factor. Choosing optimal strides is a trade-off between latency and memory requirements. In [25] and [14], approaches to find optimal strides were presented but the memory requirement grows exponentially with increase in stride length at each level.

PATRICIA performs address lookup using path compression technique introduced by [19] and was extended to LPM in [22]. Nilsson et al.[20] combined multibit tries with path compression technique to improve search time using Level Compressed tries(LC Tries).

The Tree bitmap method introduced by Eatherton et al. in [12] involves the construction of a variable stride multibit trie also using the compression techniques of Lulea algorithm[9]. The main effort was to reduce the memory access widths by compressing the address prefixes as much as possible. To further reduce the latency and memory requirements, Eatherton et al. proposed several other optimizations like (a)initial array optimization(b) end node optimization (c) split tree bit map (d) Segmented bitmap.

All the Trie based algorithms inherently support LPM, but the number of tree levels increase linearly to address width which is the major drawback of this approach. Tree depth is reduced using multibit tries but does not improve the scalability with address width as the memory size increases significantly.

2.1.2 Search on values

The search in a Range Tree is performed similar to binary search. As the Range Tree method is a search on values approach, search is performed based on the value of the incoming address. So comparisons are performed on full length addresses. The tree is traversed based on the outcome of the comparison at each stage of the tree. Range Trees lead to balanced trees as shown in Figure 2.2 as opposed to search on length schemes. They store complete addresses and hence have a considerable memory requirement.

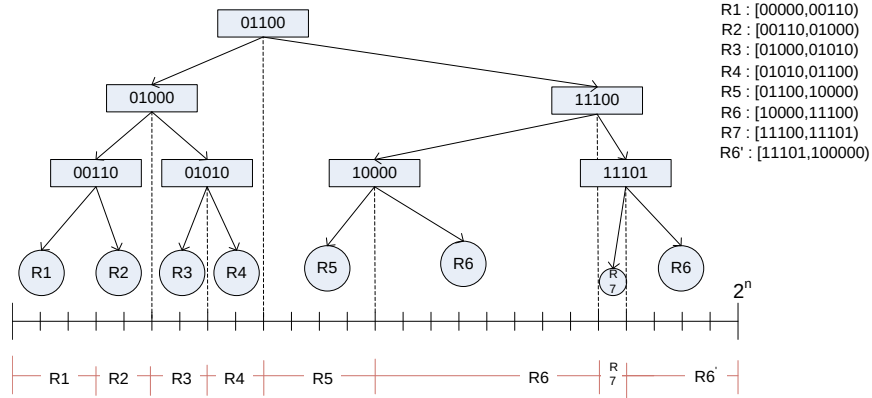


Figure 2.2: Search on Values : Range Tree

Multiway Range Trees[27] is a version of Range Trees performing multiple comparisons at each level of the Tree. These approaches do not inherently support LPM as mentioned in [18][21][27]. Thus, additional information must be stored in order to perform address lookup based on LPM.

Several other approaches like TCAM[7] based on Content Addressable Memory(CAM) were introduced that have very good performance but are very expensive in terms of power and memory. Cache based approaches presented in [15][16] tried to perform lookup exploring the associativity of caches but proved to be complex. Address lookup based on Bloom Filters[4] were introduced in [10][11] but have worst case complexities due to false positives.

2.2 Range Trie for Address lookup

Numerous lookup algorithms were invented in the past of which an important few were presented in the previous section. Even though a few are just serving the purpose(e.g Tree bitmap[12] used in Cisco's CRS-1), they cannot endure with the pace of growing Internet speeds, increasing complexity and size of routing tables and the forthcoming need to move to IPv6. This cited the need for a new address lookup algorithm that would have (a) low latency (b) high throughput (c) low memory requirements and (d) scalability with increasing routing table sizes and address width. The Range Trie

algorithm introduced by I.Sourdis in [23] is one such algorithm that would serve the present and future needs of the Internet world.

A Trie performs exact match on parts of addresses for lookup, while a Range Tree performs comparison on full length addresses. The Range Trie can be considered as a blend of both the Trie and the Range Tree methods as it performs comparisons on parts of addresses to perform address lookup. It tries to borrow the benefits of both the methods.

A set of unique bounds(end points of prefixes) are extracted from the existing prefixes in the routing table. These bounds define ranges that are placed sparser or denser in the address space $[0, 2^n)$. The address space where bounds are sparser have longer ranges and where bounds are denser have shorter ranges. Comparison on sparser ranges might need just few starting bits and the denser ranges have more bits in common that can be shared during the comparison. Taking advantage of this property, given a memory bandwidth(BW), Range Trie tries to reduce the number of bits to be stored of each bound, hence increasing the number of bounds stored in a node.

Range Trie is a tree like data structure resembling Multiway Range Trees while performing comparisons only on parts of addresses. At each node, comparisons are performed to traverse the tree structure. For a given memory bandwidth, more bounds can be stored at a node by reducing the number of bits needed to store a bound. This increases the branching factor(number of outgoing branches) per node reducing the height of the tree which in turn reduces the lookup latency.

In the rest of the section, the fundamental rules of the algorithm to construct the Range Trie will be presented in section 2.2.1. Section 2.2.2 describes the structure of the Range Trie and the lookup procedure using the Range Trie algorithm is presented in 2.2.3.

2.2.1 Range Trie Rules

Each node in the Range Trie maps to an address range $[N_a, N_b)$. Each node contains k address bounds(B_k) that divide the address range of the node into $k+1$ sub ranges $R_1, R_2 \dots R_{k+1}$ as shown in Figure 2.3. The Range Trie algorithm tries to accommodate as

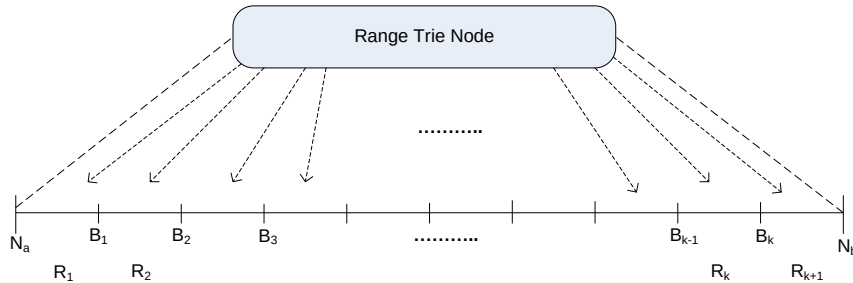


Figure 2.3: A Range Trie node

many bounds as possible per node to perform maximum possible number of comparisons at a node. So, Range Trie is build based on the following observations:

- Nodes closer to the root compare addresses that are sparsely located in the address space. So they may have a longer common suffix.
- Nodes closer to the leaf compare addresses that are densely located in the address space. So more precision is needed to locate the ranges. They may have longer common prefix and hence only the suffix part is to be compared.
- Nodes in the other levels may need to compare addresses that are sparsely or densely located in the address space. Thus the respective parts of the address are compared at these node resulting in a balanced tree structure.

A set of rules presented in [23][24], in order to exploit the above mentioned characteristics to minimize the comparison widths of the bounds thus performing the maximum possible comparisons at each node. The five rules are as follows:

- **Rule 1:** Omit Node common Prefix.
- **Rule 2:** Omit address zero Suffix.
- **Rule 3:** Share address' common prefix.
- **Rule 4:** Share address' common suffix.
- **Rule 5:** Address alignment.

Consider a node N that maps to the address range $[N_a, N_b)$ with each address being W bits wide as shown in Figure 2.3 and in incoming address $A_{IN} \in [N_a, N_b)$ that needs to be compared against the bounds in the node N to determine the subrange to which A_{IN} belongs to. Each comparator determines whether A_{IN} is Less, Equal or Greater or equal with respect to a bound. The rest of the section explains the above five rules in detail.

Rule 1: Omit Node Common Prefix. The common prefix of length L bits ($L < W$) from all the bounds in the node address range $[N_a, N_b)$ can be omitted from comparison. An example for this rule is shown in Figure 2.4. When A_{IN} visits a node, it also belongs to the same address range $[N_a, N_b)$ and thus will have the same common prefix of L bits as the node bounds. Comparison of these L most significant bits(MSB) will always result in equality. So, comparing just the $(W - L)$ bits can produce the required result.

Rule 2: Omit address zero Suffix. If an address bound B_i has L bits($L < W$) of suffix that are zeros, then this suffix of L bits need not be compared against A_{IN} . Figure 2.5 gives an example of this scenario.

If a bound has a L -bit suffix that are zeroes, then comparing these bits against the incoming address will always result in "Greater or Equal". So, comparing just the $(W-L)$ MSB will produce the desired result. If the comparison of $(W-L)$ MSB results in an equality, then the result will be "Greater or Equal" since the L LSB are zeroes. Hence the zero suffix can be omitted from comparison.

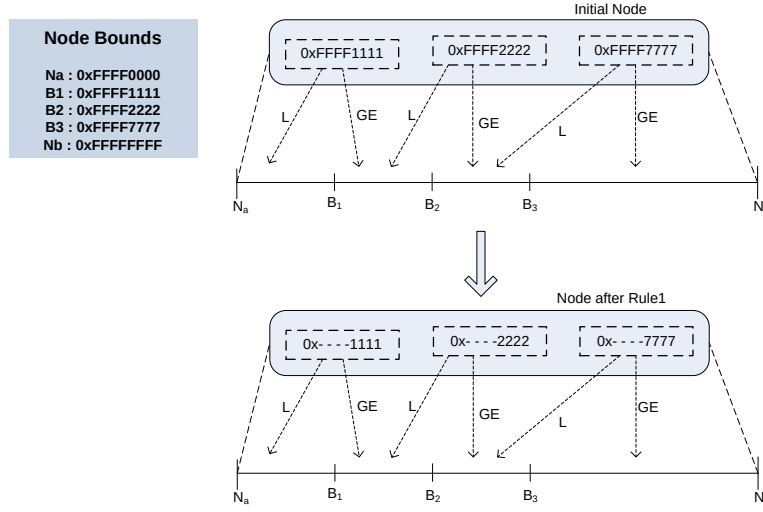


Figure 2.4: An example for Rule1

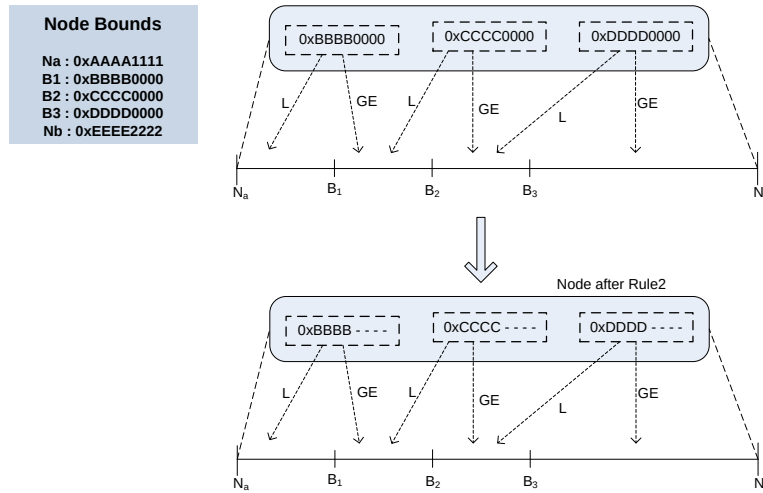


Figure 2.5: An example for Rule2

Rule 3: Share address' common prefix. The common prefix(CP) of length L bits ($L < W$) can be shared among the bounds B_i and can be processed separately. An example of this rule is shown if Figure 2.6. If the L MSB of A_{IN} is less than CP of group of bounds $B_i \dots B_j$, then the comparison results in the range $[B_{i-1}, B_i)$. If the L MSB of A_{IN} is greater than CP, then the comparison results in the range $[B_j, B_{j+1})$. Otherwise the comparison result of the remaining $(W-L)$ bits determines the range A_{IN} belongs to i.e. one of the ranges between $[B_i, B_j)$.

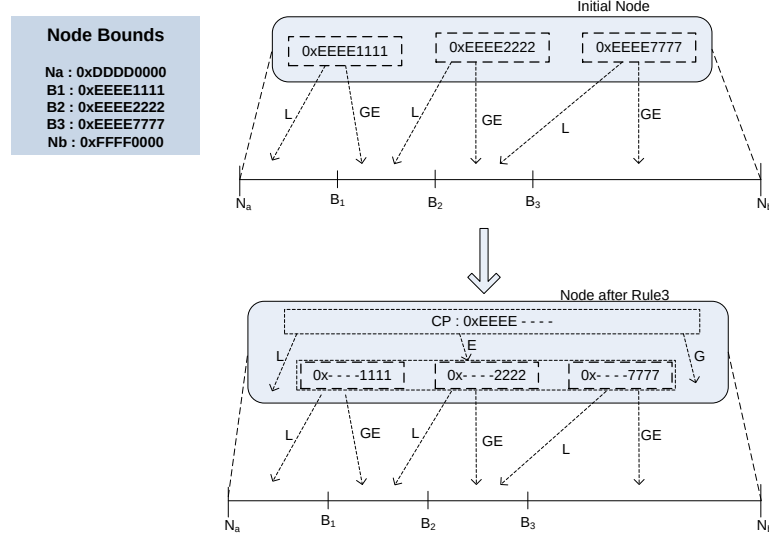


Figure 2.6: An example for Rule3

Rule 4: Share address' common suffix. The common suffix(CS) of length L bits ($L < W$) can be shared among the bounds B_i and can be processed separately. Figure 2.7 gives an example of this rule. The (W-L) MSB of the A_{IN} are compared with

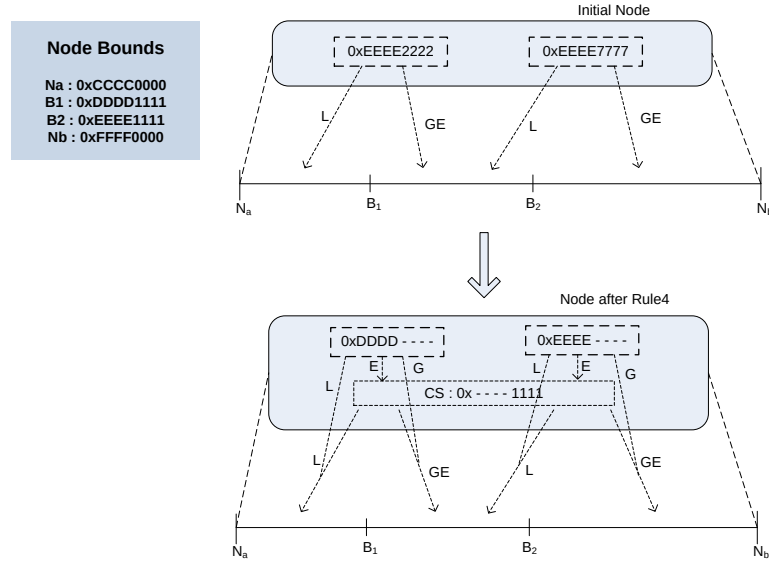


Figure 2.7: An example for Rule4

the (W-L) MSB of the group of bounds $B_i \dots B_j$. If the comparison of A_{IN} with B_i results in a “Less”, then the result would be the range $[B_{i-1}, B_i)$. If the comparison of A_{IN} with B_j results in a “Greater or Equal”, then the result would be the range $[B_j, B_{j+1})$.

Otherwise the result depends on the comparison of L common suffix (CS) bits.

Rule 5: Address alignment. Comparison of an incoming address A_{IN} with the bounds in the node whose address range is $[N_a, N_b)$ is equivalent to the comparison of $A_{IN} - N_a$ with the bounds $B_i - N_a$ in the range $[0, N_b - N_a)$. An example of aligning addresses is given in Figure 2.8. Aligning the addresses in a node gives a wide zero

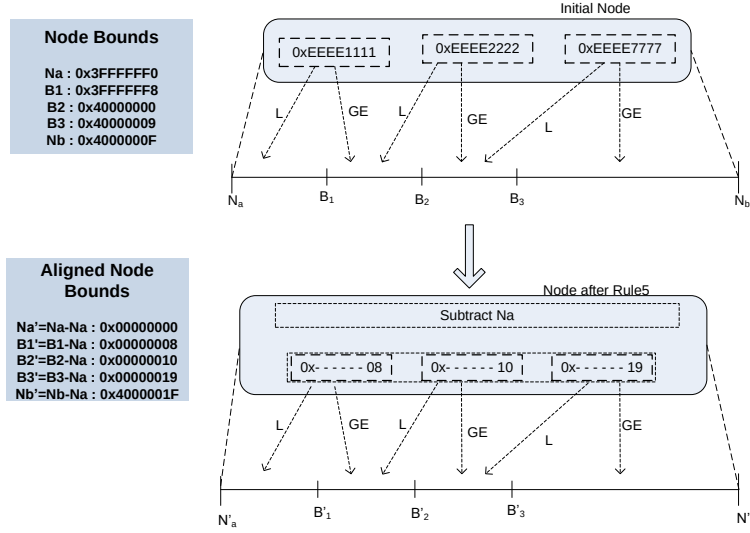


Figure 2.8: An example for Rule 5

valued prefix and hence allows to perform narrower comparisons. In the worst case the width of the comparisons after aligning the addresses would be $L = \log(N_b - N_a)$ bits. So, subtraction of $A_{IN} - N_a$ suffices to be calculated for those L bits.

2.2.2 Range Trie Structure

The Range Trie structure exploits the five rules presented in section above, to create a balanced tree structure. At each node parts of multiple address bounds are compared. The number of bits to be compared per bound is minimized by omitting or sharing the common prefixes or suffixes among the bounds of the node. Re-aligning the bounds with subtraction increases the node common prefix which further reduces the number of bits to be compared at the node.

There are several parameters that restrict the number of comparisons that can be performed at each node. The number of bits that can be compared at each node depends on the bandwidth (BW) parameter. The maximum width of the comparator that can be used is usually the address width. The nodes are formed from the available address bounds using the five rules of the Range Trie algorithm. These nodes form the whole tree structure as shown in Figure 2.9. The characteristics of a node are as follows:

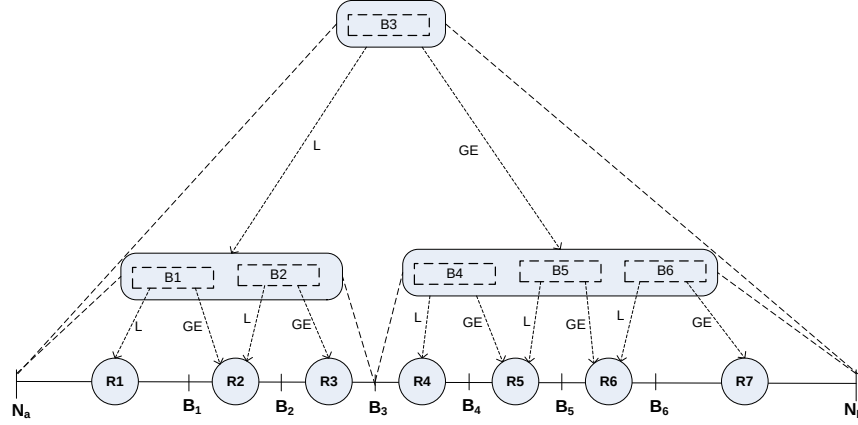


Figure 2.9: An example for Range Trie structure

- The width parts of addresses stored at node is determined by applying the five rules described in the previous section.
- Each node can contain k address bounds where k depends on BW and the width(W) of the parts of addresses that can be stored at the node. Usually $W = 8, 16, 32 \dots N$, where N is the address width(IPv4=32, IPv6=128).
- A node containing k bounds have $k + 1$ out going branches. Each outgoing branch points to a node in the next level of the Range Trie.
- Each comparison performed at a node results in a “Less” or “Greater or Equal”, based on which the outgoing branch is determined.

A Range Trie consists of several nodes at each level of the tree. The information stored at these nodes is as follows:

- the values to be compared with.
- the number of comparisons to be made.
- the parts of incoming address to be compared.
- the branches to the nodes in the next level.
- the shared common prefix and suffix and alignment address if needed.

As already discussed, at each node, the outgoing branch to be taken to reach the next level is determined by the result of comparisons performed at each node. The steps to be followed to determine the outgoing branch is presented in the following section.

2.2.3 Lookup Procedure

The Range Trie is a tree structure that spans multiple levels. At each level there are multiple nodes except at the first level which just has a root node. During lookup, an incoming address A_{IN} visits one node at each level, performing comparisons at each node to determine the next node to be visited in its traversal. So, the lookup process starts at the root node and ends when A_{IN} is matched to an address range in the interval $[0, 2^n)$.

At each node, a decision is taken to select an outgoing branch to reach the next level. Consider a node N with k bounds as shown in Figure 2.10.

The steps to be performed at each node in order to make the decision are as follows:

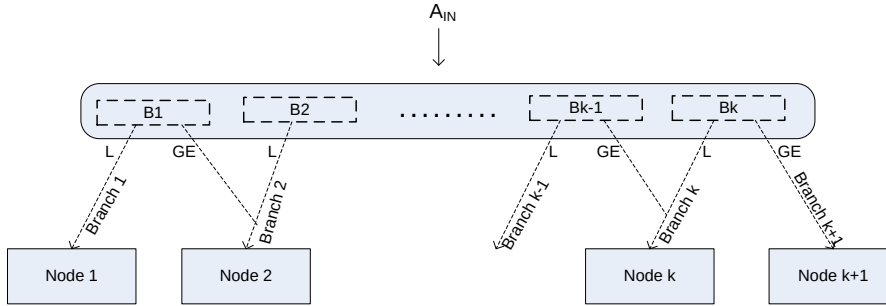


Figure 2.10: An example for Range Trie node and its children

- If bounds are to be aligned, then a subtraction value is subtracted from A_{IN} .
- A_{IN} is compared with the k bounds in the node, and each comparison results in a “Less” or “Greater or Equal(GE)”.
 - If the result of the comparison 1 is Less, then the outgoing branch is B_1 .
 - If the result of the comparison k is GE, then the outgoing branch is B_{k+1} .
 - Otherwise, there exists a bound $j \neq k$ whose comparison results in GE, and a bound $j+1$ whose comparison results in a Less. Then the outgoing branch is B_{j+1} .
- If there is common prefix sharing, the corresponding prefix bits are compared with the prefix bits of A_{IN} .
 - If the result of comparison is Less, then the outgoing branch is set to B_1 .
 - If the result of comparison is G, then the outgoing branch is set to B_{k+1} .
 - Otherwise, if the result of the comparison is Equal, then the outgoing branch depends on the result of step 2.
- If there is common suffix sharing, the corresponding suffix bits are compared with the suffix bits of A_{IN}

- If the result of the comparison is GE, then the outgoing branch depends on the result of comparison in step 2.
- If the result of the comparison is Less, then the outgoing branch is one less than the branch decided in step 2.

All the steps discussed above can be performed in parallel and then put together to determine the branch to be taken to visit a node in the next level of the Range Trie structure. The process of lookup continues until A_{IN} is matched to an address range.

2.3 Range Tree Data Structures for LPM and Updates

The Range Trie does not inherently support LPM, just like Range Tree approaches. So additional information must be stored along with the node information that has comparison values. This section discusses some of the Range Tree approaches supporting LPM, that gave us an insight into several possible schemes that can be employed on Range Tries to support LPM and dynamic updates.

2.3.1 Multiway Range Tree

Multiway Range Trees(MRT) [27] is a version of Range Trees, in which multiple comparisons are performed at each node instead of a single value. In order to support longest prefix match, prefixes are associated with each of the nodes in the tree. The prefix storing rule followed by the MRT method is as follows:

A prefix p is stored at a node or a leaf v if and only if the address range of v is completely inside the address range of p but the address range of the parent of v is not in the address range of p .

This removes the redundancy in storing the prefixes at all the nodes of the tree.

Updates to the Range Tree are performed as follows:

- **Updating the keys and handling splits and merges:** The end nodes of a prefix must be inserted into the Range Tree. Given a memory bandwidth, as the Range Tree performs full length comparisons, each node in the tree can accommodate only $k = BW/N$ bounds, where BW : *Bandwidth* and N : *addresswidth*. When a new key is to be inserted into a node and if the node is already full, the node should be split into two. In a similar approach, when a prefix is deleted, it can be possible to merge adjacent nodes into one node.
- **Updating the address spans and prefix heaps:** When a node is split or two nodes are merged, the address ranges of the parent nodes should be updated. Then there is a possibility of split or a merge of the parent. So the parent of the parent should also be updated. In the worst case the updating the address spans continues until traversing back to the root node.

Each node also maintains a prefix heap that stores all the prefixes that include the address range of the node following the prefix storing rule. So the prefix heaps of all the nodes that are in the range of the new prefix should also be updated.

So in the MRT, a prefix is inserted or deleted in $O(t \log_t n)$ time where t is the arity of the node and n is the number of nodes to be updated. It cannot perform any other operations like lookup until the Range Tree is updated.

2.3.2 B-Tree

The B-Tree is a variant of the MRT as described in [18]. In this method the prefix is stored in only $O(1)$ nodes per level as opposed to $O(m)$ nodes in the MRT where m is the arity of the node. The B-tree also follows the same scheme of splits and merges as in the MRT method for inserting and deleting a prefix. But as the prefix is stored in only $O(1)$ nodes per level, only those nodes are to be updated in the B-Tree method. This reduces the memory requirement and also the update time compared to the MRT method.

2.3.3 Dynamic Segment Tree

Dynamic Segment Trees(DST)[6] is another variant of Range Trees, constructed using elementary intervals that are disjoint and covering the entire address space. Ranges are stored as canonical sets at the node instead of prefixes. The search and update methods are similar to MRT and B-Tree requiring a feedback in the tree to manage splits and merges of the nodes in the tree. Since each node in DST is augmented with a range set, the need for an extra heap as in MRT and B-Tree is removed. Thus they have reduced memory consumption and better update speeds compared to MRT and B-Tree but is worse in terms of search speeds.

2.4 Summary

In section 2.1, a set of algorithms/approaches that perform address lookup were presented. The algorithms were classified into search on length and search on values approaches. The Trie based approaches lack scalability in performance with increase in address width. Range Tree based approaches need to store extra information in order to support LPM and consume considerable memory size to store full length addresses. The memory requirement further increases with increase in address width.

In section 2.2, the Range Trie method designed to answer the growing internet requirements was presented. The Range Trie method is a mixture of both search on length and search on values approaches that deliver (a) low latency (b) high throughput (c) low memory requirement (d) better scalability was presented. Lookup is performed by comparisons on parts of addresses at each node based on the 5 rules that serve as a fundamental concept for the Range Trie development. The Range Trie method maximizes the number of comparisons at each node making an effective use of the

memory bandwidth.

Section 2.3 discussed some related approaches that support LPM and perform dynamic update of routing tables. Similar to these methods, Range Trie also does not inherently support LPM. The Range Trie method should support LPM and incremental updates to process the internet traffic in the real time. The main focus is to support LPM and dynamic updates maintaining the inherent properties of the basic Range Trie design. The design and implementation of the modified Range Trie structure in order to support LPM and incremental updates will be presented in detail in the following chapters.

Range Trie with LPM and Dynamic Updates

3

In this chapter, the necessary changes to be made to the Range Trie data structure described in Section 2.2 in order to support Longest Prefix Match and dynamic updates will be presented. This design effort aims at modifying Range Trie structure to have fast update times and performing lookup on longest prefix match maintaining the inherent characteristics of the Range Trie method.

In a router, a lookup problem is to determine the next hop in a packets routing path. With introduction of CIDR [13] in 1993, routing table entries are associated with variable sized address prefixes. So an incoming address might match with a set of prefixes and the lookup algorithm has to find the best matching prefix which is called the *longest prefix match*. In order to support LPM, information related to the prefixes must be stored at the nodes in the Range Trie.

A routing protocol allows the routing tables in a router to be *updated dynamically* on their own. As the network topology is not completely stable, routers exchange routes whenever there is change in the path. An update is a message broadcasted by one of the routers to notify about new insertions or deletions to the network or changes in routing path for the routers. When a router receives an update, it changes the corresponding routing table entry to reflect the changes in the network. As a router just stores the best possible path to the destination, information related to the old routing path is removed from the routing table.

Section 3.1 first presents the issues involved in modifying the Range Trie structure to support LPM and dynamic updates. Then section 3.2 discusses ways to store prefixes in the Range Trie structure reducing the amount of memory needed to store the prefixes. Integrating LPM into Range Trie structure is described in section 3.3. Section 3.4 describes the design changes to be made to handle insertion and deletion of prefixes to the Range Trie. A detailed description of spare levels that are placed at the end of the Range Trie structure to handle insertion of prefixes is given in Section 3.5. Section 3.6 describes the complete modified Range Trie structure along with LPM and update structures.

3.1 Issues involved

As described in section 2.2, the main objective of the Range Trie method is to have high throughput, efficient memory usage and effective utilization of bandwidth. There are several underlying issues that need to be resolved in order to support longest prefix match and handle dynamic updates, adhering to the principles and objectives of the

Range Trie method.

- **Storing prefixes 1** During lookup, each node must provide an associated prefix to the incoming address A_{IN} whenever it traverses through the node. So each node should store the prefix along with its node data. But storing prefixes at each node in the Range Trie requires a huge amount of memory. In Figure 3.1, $p1$ should be stored all the nodes in the Range Trie, $p2$ in *Node a* and its children, $p3$ in $b1$ and $b2$ and $p4$ in $b3$ and $d1$. At a node, even if we store only the longest prefix that spans the address range of the node, the memory size would be $O(m)$ where m is the number of nodes in the Range Trie.

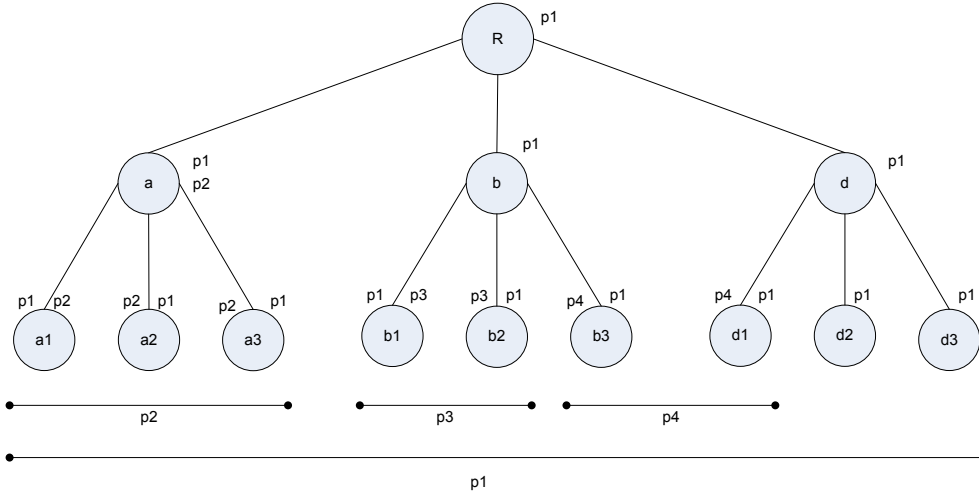


Figure 3.1: Storing prefixes 1

- **Storing prefixes 2** Whenever a new prefix is inserted or a prefix is deleted, all the nodes that are inside the address range of the prefix should be updated. As each node corresponds to a different memory location in memory, we need to perform $O(n)$ writes for each update operation where n is the number of children of the node. As the Range Trie is designed to be implemented on hardware, large number of writes is a huge drawback.
- **Handling Action Pointers** After traversing the Range Trie completely, we retrieve the action corresponding to the longest matching prefix from the *action table*. As we have one big action table at the end of the pipeline, whenever there is an update all the memory entries of action table corresponding to the nodes at the leaf level should be updated with the new action. This also involves a large number of writes to the memory. Figure 3.3 gives an example of this problem. Nodes $N3$ through $N7$ are in the range of prefix p and hence the corresponding action table entries should be updated in the action table i.e. five entries in the action table.
- **Processing non-existing bounds** Updates can be classified into two types. A new prefix with already existing bounds or a new prefix with non-existing bounds.

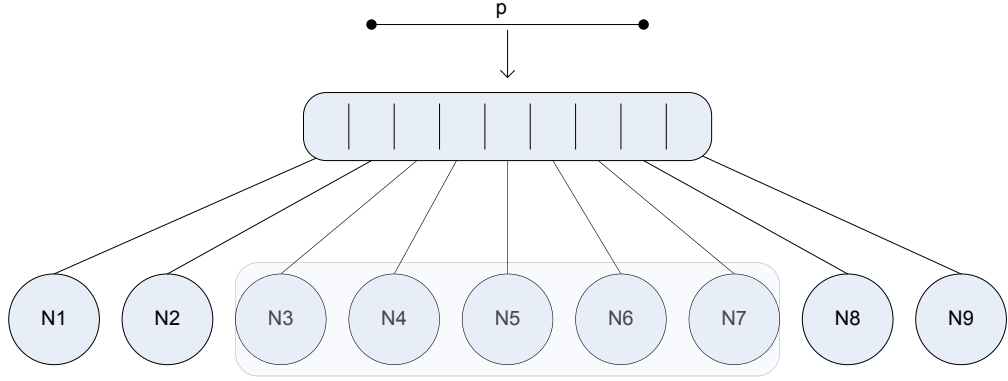


Figure 3.2: Storing prefixes 2

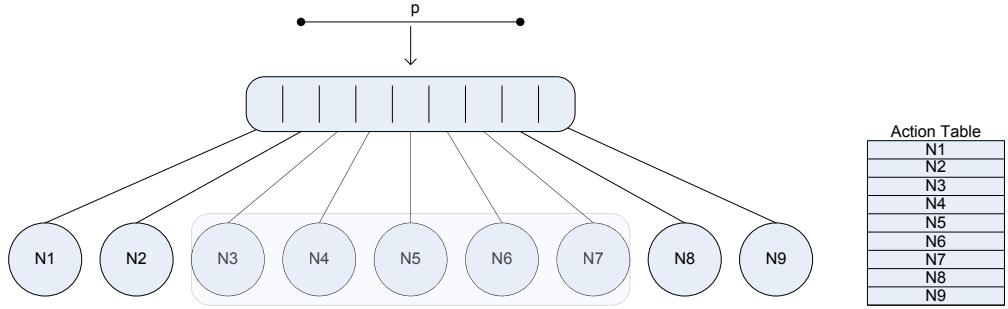


Figure 3.3: Handling Action pointers

Thus, whenever a prefix with non-existing bounds is inserted, the bounds of the prefix need to be included in the Range Trie. According to Range Trie algorithm, each node can only store a limited number of bounds based on the bandwidth and width of bounds stored at that node. When a new prefix is inserted, a lookup is performed on the bounds of the prefix. In the worst case, we know whether the bounds exist only after we traverse the whole Trie. If the final leaf node does not fit the new bound that is inserted, the node needs to be split or merge based on whether the update is a insertion or a deletion respectively. In the worst case, we need to traverse the whole Trie back to root node in order to find a suitable position to be inserted for the new bound. So the whole pipeline should be halted until the update is completed, introducing a high performance overhead. In this approach, updating the Range Trie involves allocating worst case memory required per each level in the Range Trie. As the approaches existing literature are software based approaches, this might not be a problem. Since the Range Trie is a hardware oriented design, this approach introduces a huge memory overhead. Figure 3.4 shows our proposal to the solution having a fixed and updatable part of the Range Trie where $T_1...T_n$ are sub-trees at each address range.

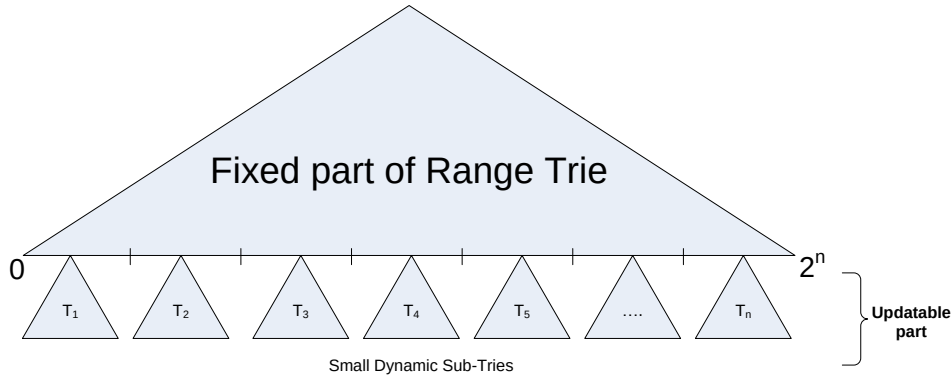


Figure 3.4: Processing non-existing bounds

In the later part of the chapter, solution to Problem 1, 2 will be discussed in section 3.2 and Problem 3 will be discussed in Section 3.4, Problem 4 will be discussed in Section 3.5. The rules to be followed to update the Range Trie structure is presented in Section 3.6. The resulting complete Range Trie structure will be presented in Section 3.7.

3.2 Associating prefixes to nodes

In the Range Trie, additional information corresponding to the prefixes needs to be stored at the nodes in order to support longest prefix match. Similar to the Multiway Range Tree method [27], the idea is to store prefixes at the internal nodes. Since the prefix can be constructed from the node information using the length of the prefix, *only the prefix lengths are stored in the Range Trie*. When lookup is performed at each level of the Range Trie, corresponding prefix length should be found along with the pointer to the child node. After completely traversing the Range Trie, the action associated to that matching prefix should be retrieved from the action table. Each node in the Range Trie consists of some address bounds and each address range (range in between two address bounds) spans a certain part of address space. So, prefix lengths corresponding to all these address ranges need to be stored at each node. The idea seems to be redundant as storing prefix lengths at all the nodes consumes a lot of memory. In the later part of the section, a solution to cope with the problem is discussed.

Consider a node N with address range $[N_a, N_b)$ and has k address bounds. If the node is not a leaf node, it will have $k + 1$ child nodes each with range $[N_i, N_j)$ where $N_a \leq N_i < N_j < N_b$ and $N_a < N_j < N_b$. Let p be a prefix with range $[lb, hb)$. A prefix should be stored at all the nodes whose address range is within the range of the prefix. So, if the range of p covers the address range of node N it is understood that it also covers the address range of all its children. So, potentially each of N and its $k + 1$ child nodes can store the prefix. However during lookup, an incoming address A_{IN} first encounters the parent node N before reaching the child. Thus if we store the prefix at the parent and carry out the result to the child node, prefix need not be

explicitly stored at the child nodes. Instead of storing the prefix at $k + 2$ times (1 for the parent and $k + 1$ for its child nodes), now it is just stored at the parent node thus saving a lot of memory. Here, storing the prefix means storing the length of the prefix. Storing just the length of the prefix would be sufficient because we can calculate the prefix value from address information at the node. Figure 3.5 gives a small example of the Range Trie, with four prefixes $p1$, $p2$, $p3$ and $p4$. Prefix $p2$ can be stored at a instead of also storing them at $a1$, $a2$, $a3$ as $p2$ spans the address ranges of all these nodes. Prefix $p3$ and $p4$ can only be stored at the leaf nodes as they do not cover the address range of the nodes b and d . It is just sufficient to store prefix $p1$ at the root node R , as all the nodes along with the root node fall into the address range of $p1$.

A rule is framed from the solution devised for problem 1 as mentioned in [27] is as follows:

Prefix Storing Rule 1 : Store a prefix p at a node N if and only if the address range of the node is contained in the prefix range but the address range of the parent node of N is not contained in the prefix range. Only the length of the prefix is stored along with the node information.

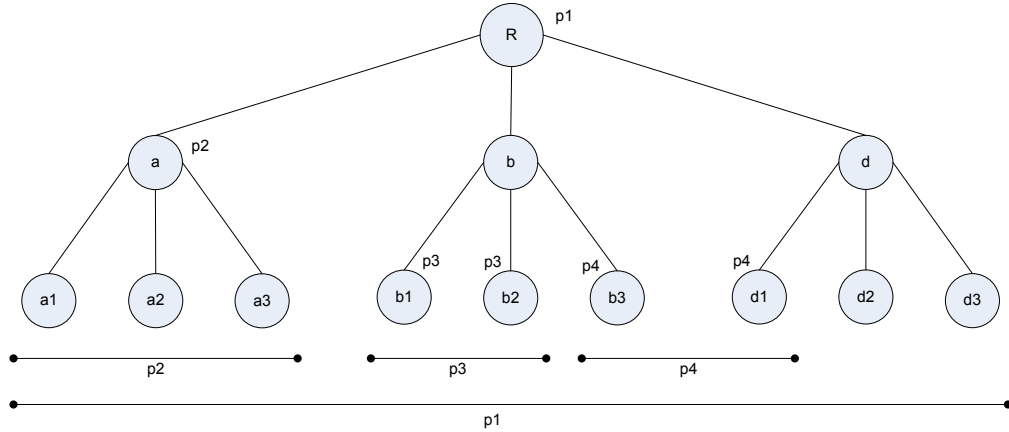


Figure 3.5: Prefix storing rule 1

When there is an update, all the nodes that are in the address range of the prefix must be updated with the new prefix data. At a certain node, m children might be in the address range of the prefix, but the node itself is not in the address range of the prefix. Thus, m nodes should be updated which affects the performance of the Range Trie structure. So the complexity of update in the Range Trie would be $O(L \times m)$ where L is the number of levels in the Range Trie and m being the number of children to be updated at each level. In order to avoid updating large number of nodes at each level of the Range Trie, prefixes of children are stored at the parent. Now information at only one node i.e. the parent node is to be updated. Figure 3.6 gives an example of this scenario. Child nodes N_2 to N_7 are in the range of the prefix p , but the parent node is not in the range of the prefix. When the Range Trie is updated with prefix p , N_2 to N_7

need to be updated. Instead, if the prefixes are stored at the parent, then only the parent node should be updated with prefix p . Now the complexity of update reduces to $O(L \times 1)$.

Prefix Storing rule 2: *Prefixes of children are stored at the parent if and only if the child is inside the address range of the the prefix but the parent is not.*

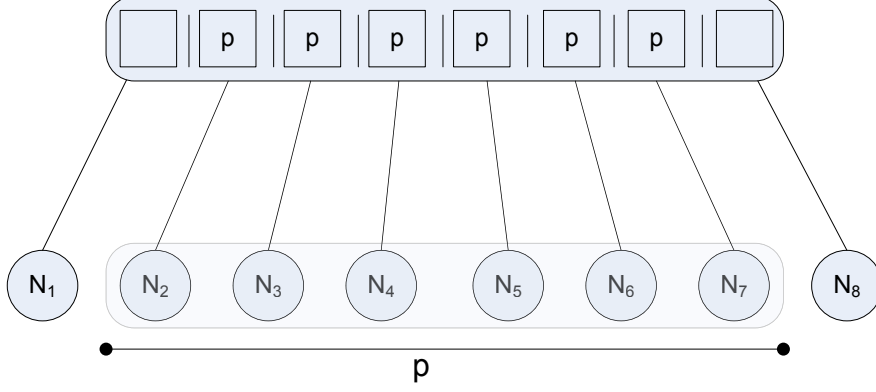


Figure 3.6: Prefix storing rule 2

The whole scenario of associating prefixes with nodes is summarized in Figure 3.7. Initially, all the prefixes were stored at the leaf nodes. So the complexity of storing the prefix lengths and the number of writes is $O(N)$ where N is the number of nodes in the range of the prefix. Applying Prefix storing rule 1, prefix lengths are stored at internal nodes if possible, reducing the complexity of storage and the number of writes to $O(L \times m)$ where $L = O(\log_m N)$ and m is the number of children per node. Then applying prefix storing rule 2, prefix lengths of children are stored at the parent that reduces the complexity of number of writes to $O(L \times 1)$ while the complexity of storage remains $O(L \times m)$.

3.3 Longest Matching Prefix

In the initial Range Trie structure as described by [26], the lookup starts with the incoming address A_{IN} visiting the root node. In general, lookup at each level of the Range Trie can be defined as a two step process.

- Visiting the node (Reading node information from memory).
- Performing comparisons and decide which branch to follow based on the comparison results.

Based on the comparisons performed at each level, A_{IN} traverses the Range Trie. The process ends when A_{IN} reaches the leaf node where it is matched to an address range. As prefixes overlap, the span of each address range might be a part of multiple prefixes. This section describes the adaptation of Range Trie structure to support LPM, i.e.

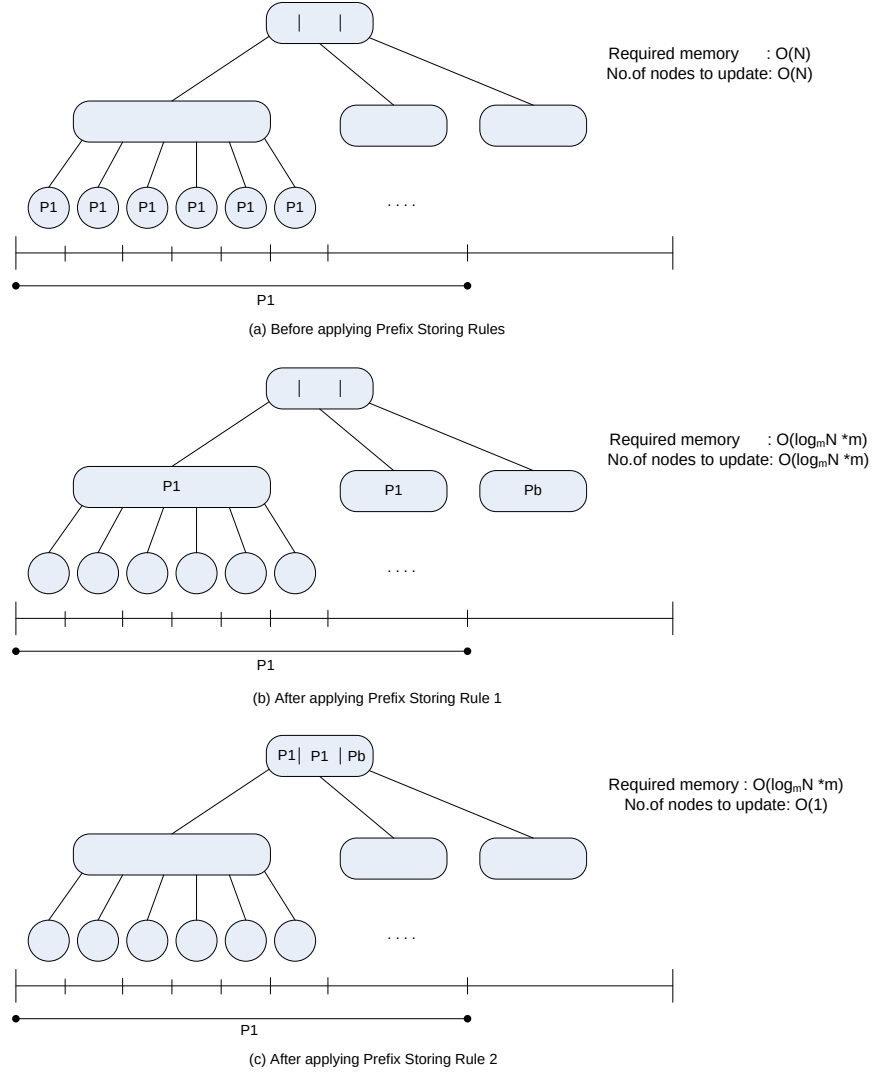


Figure 3.7: Associating prefixes to nodes

to determine the longest matching prefix among the multiple prefixes that span each address range in the interval $[0, 2^n)$.

Figure 3.8 shows a small Range Trie that stored prefixes according to the two prefix storing rules. As already mentioned, in our data structure, storing a prefix at a node means storing the length of the prefix. Now the lookup procedure should be able to determine the longest matching prefix for an incoming address as it traverses down the Range Trie. Consider three incoming addresses A_{IN1} , A_{IN2} and A_{IN3} that traverse the paths $R \rightarrow A \rightarrow A1$, $R \rightarrow B \rightarrow B3$, $R \rightarrow C \rightarrow C2$ respectively. After visiting Root node R , A_{IN1} remembers $p3$ stored at it. As it traverses down the tree it does not find any other prefix, so the longest matching prefix for A_{IN1} would be $p3$. A_{IN2} remembers $p5$

at R and at the next level it finds $p2$. As $p2$ is greater than $p5$ it remembers $p2$ and traverses down the tree. Thus the longest matching prefix for A_{IN2} would be $p2$. While traversing down the tree A_{IN3} remembers $p5$ at the root node and as it is the only prefix it encounters along its path, $p5$ is the longest matching prefix for A_{IN3} .

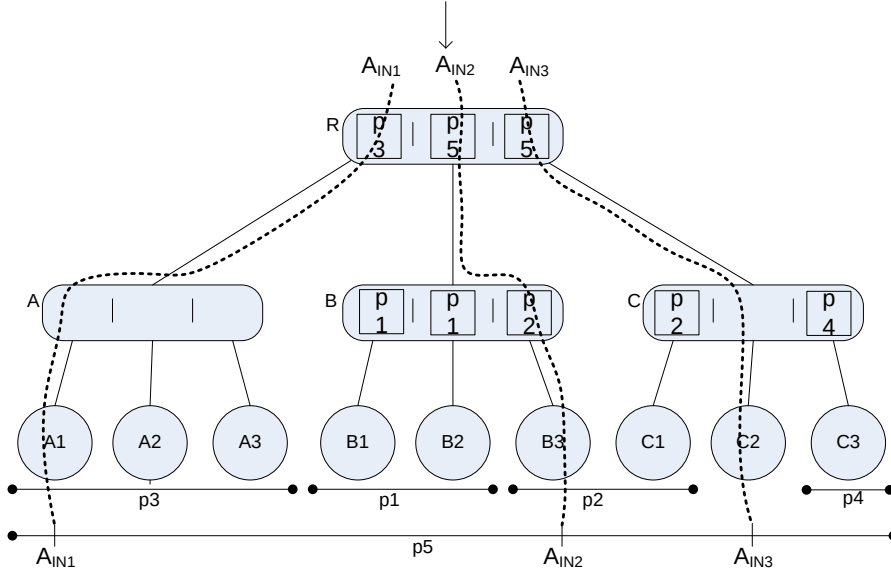


Figure 3.8: Range Trie structure supporting LPM

3.4 Dynamic Updates

As described earlier, the term update refers to inserting or deleting a prefix. A prefix has two bounds, a lower bound and an upper bound. Whenever there is an update, a lookup must be performed with these bounds as incoming addresses to the pipeline. Figure 3.9 shows the nodes in the Range Trie that are eligible for update when prefix $p1$ and $p2$ are inserted. As we already know, an update operation updates the necessary data at all the nodes that are in the range of the prefix. In this section, how the updates are handled in the Range Trie structure and the necessary changes to be made to the Range Trie structure in order to support updates will be discussed.

3.4.1 Prefix Traversal

When a prefix visits a node, lookup is performed on low and high bounds of the prefix. After lookup both the bounds may go to the same outgoing path to reach the child at the next level or each of them might lead a different path from that node. The latter case is termed as '*prefix split*'. A prefix split not occurring at node means that no child of that node is in the address range of that prefix. If a prefix split occurs then all the children of the node that are in the range of the prefix are to be updated with the new prefix length. Here update in the prefix length means that the already stored

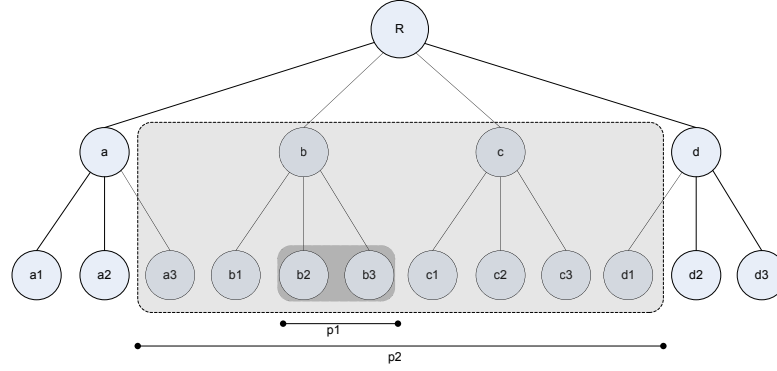


Figure 3.9: Range Trie showing the nodes in the range of prefixes

prefix length is replaced by the new prefix length only when the new prefix length is greater than the existing prefix length.

Assume a small Range Trie of three levels and each node in the Trie consisting of ' k ' bounds and prefixes $p1$ and $p2$ with address ranges $[p_i.lb, p_i.hb)$ as shown in Figure 3.10. When prefix $p1$ visits the root node, it can be seen that both the bounds determine *node b* at level 2 as their next destination. So there is no split yet and hence there is no update at the root node. When the prefix reaches the second level, a prefix split occurs as the low bound and high bound result in different lookup paths. Prefix $p2$ splits at the root node. The second and third children of the root node are in the range of the prefix and they are to be updated with the new prefix lengths. According to prefix storing rule 2, prefixes of children are stored at the parent. So, prefix lengths of *node b* and *node c* can be updated in a single cycle at the parent node. But *node a* and *node d* are only partially covered by the prefix. So we move to the next level to update *node a* and *node d*. Only *a3* and *d1* come under the prefix range $p2$. Hence *a3* and *d1* are updated at their respective parents *node a* and *node d*.

Until a prefix split happens both the low bound and the high bound move together in the Range Trie and the nodes visited by the bounds until that point are not updated. When the prefix splits at a node, all the children of the node that are completely inside the address range (i.e. between the low bound and the high bound) of the prefix are updated. After prefix split, bounds split their ways to different branches, each bound tries to update the node it is visiting until it reaches the final level. The low bound tries to update all the children of the visiting node that are to the right of it and the high bound updates all the children of the visiting node that are to the left of it. The nodes that are between the nodes visited by the prefixes need not be updated as they are already updated at the parent (Prefix storing rule 2).

In section 3.2, it is mentioned that an incoming address A_{IN} reads the next hop information from action table after completely traversing the Range Trie structure.

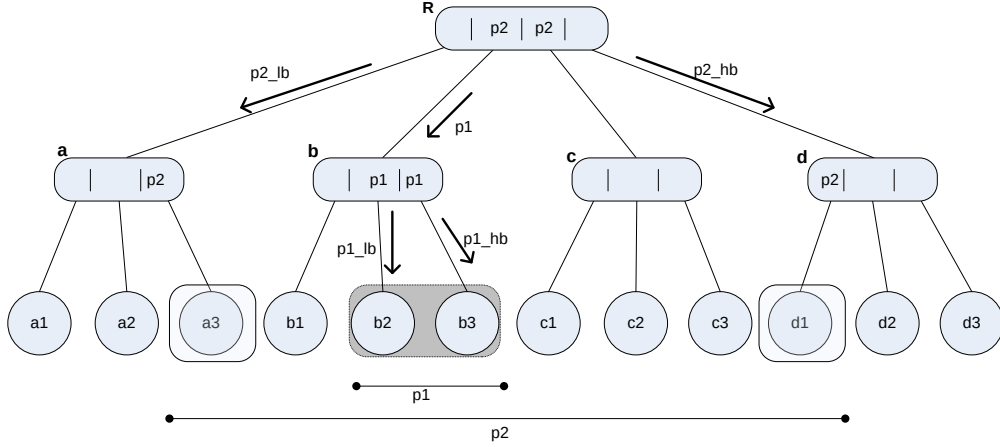


Figure 3.10: Prefix Traversal in the Range Trie

Whenever there is an update operation, action table entries of the nodes corresponding to the updated prefix should also be updated. Here we encounter the same problem of performing multiple writes to the memory. To avoid this problem, it is decided to have address pointers stored at the nodes along with the prefix length. We name these address pointers as *action pointers*. This enables us to have a single write to the action memory and the corresponding action pointer to that memory location is updated at all the possible nodes in the Range Trie structure. However, instead of updating all the nodes that are in the range of the prefix, we update only a few selected nodes according to the prefix storing rules mentioned in section 3.2. With the existing structure, a prefix can only reach the action memory after traversing the complete Trie. But with this structure, action pointer to the final action memory cannot be determined at the earlier levels of the Range Trie where we store the prefix lengths. So, one action table is introduced per level that stores the actions corresponding to the nodes in that level as shown in Figure 3.11. Action table resides along with the node information at each level of the Range Trie. Now the action pointers at each node correspond to the action memory of its child because the action pointers are stored at the parent as we do with prefix lengths. During lookup, as the prefix lengths are remembered while traversing the Range Trie, we also remember the action corresponding to the remembered prefix length. Action corresponding to the node is read from the action table using the action pointer stored at the parent node. This solves the Problem 3 mentioned in section 3.1.

Consider a small Range Trie as shown in Figure 3.11. If action pointers do not exist, when prefix $p2$ is inserted, all memory location corresponding to the leaf nodes i.e. here, all locations of Action_Table.L2 need to be updated. According to the modified Range Trie Structure and following the Prefix storing rules, only action pointer entries at root node are updated and just one location in Action_Table.L1 is to be written. Thus when prefix $p2$ is inserted only one entry in Action_Table.L1 is updated instead

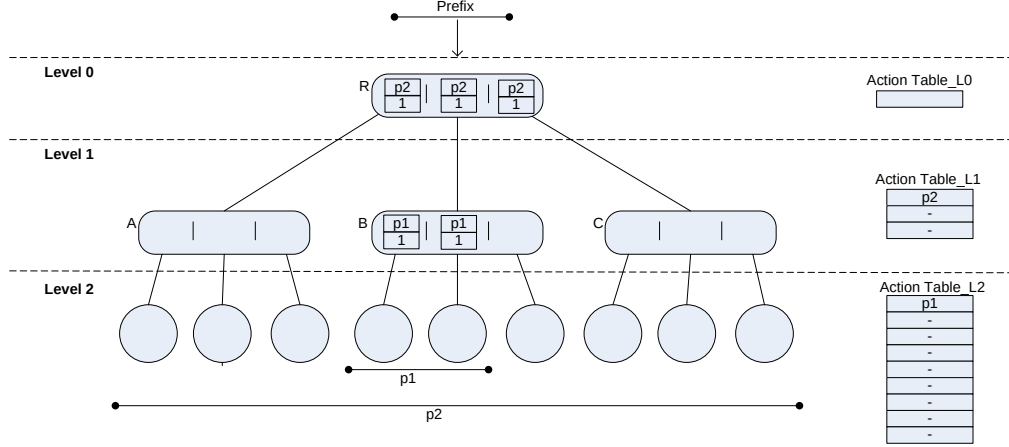


Figure 3.11: Storing Action Pointers and Action in Range Trie Structure

of updating all the entries in Action_Table.L2.

If the traversal path of the Range Trie is carefully observed, only the nodes visited by the bounds are updated. According to the prefix storing rules mentioned in Section 3.2, all the nodes that are completely inside the address range of the prefix are updated at the parent. Bounds move to the next level to update the nodes that are partially inside the prefix address range. In Figure 3.12, as prefix $p2$ traverses the Range Trie, *node B* is updated at the parent. *Node A* and *node C* are partially covered by $p2$ and hence are updated at the next level of the Trie. Prefix $p2$ has split at level 1, so each bound of $p2$ traverses the Trie independently thereafter. At level 2, one child of each *node A* and *node C* are inside the address range of $p2$. Thus low bound of $p2$ updates *node A* and high bound updates *node C*. Here an update refers to updating the corresponding prefix lengths and action pointers of the nodes. The complexity of an update operation reduced from $O(N)$ to $O(\log_m N \times m)$ where N is the number of prefixes and m is the number of children of a node.

3.4.2 Inserting a prefix

An insertion involves inserting a new prefix into the Range Trie structure. However, bounds of the prefix might already exist in the Range Trie because a bound which is an address of width N can have N prefixes with the bound being either its low bound or a high bound. Thus, inserting a prefix can be classified in two ways. One is inserting prefixes with existing bounds and the other being inserting prefixes with non-existing bounds. This section discusses the possible ways to handle the insertion of a prefix to the Range Trie structure.

- *Prefix with already existing bounds*

When a prefix is inserted, a lookup is performed on the bounds of the prefix to traverse down the Range Trie. At each level of the Range Trie, comparisons are

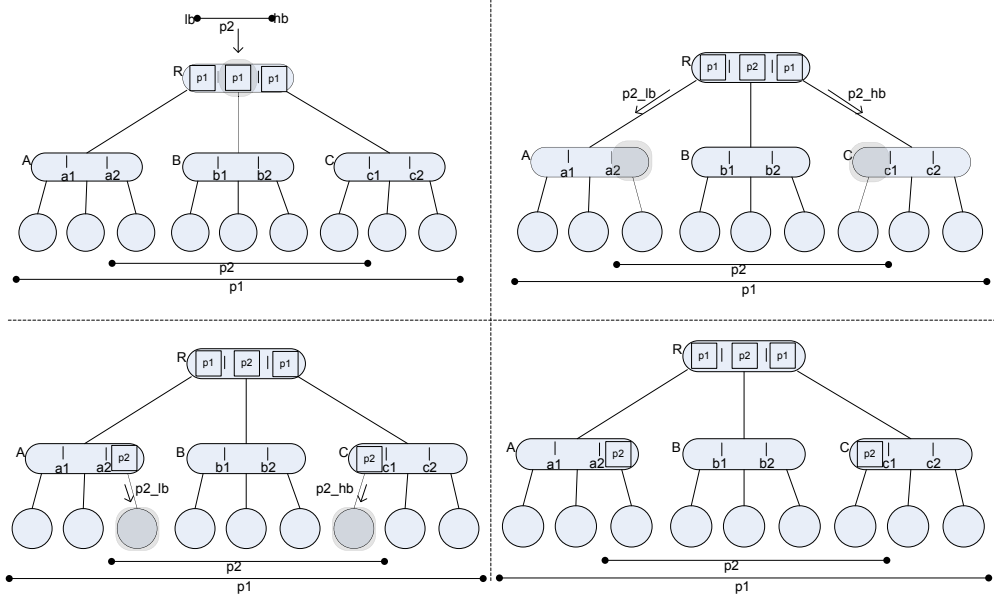


Figure 3.12: Update of a prefix in a Range Trie

performed to determine the branch to be followed to reach the next level of the Range Trie. As the prefix traverses down the Trie, an equality in comparison determines that the bound of the prefix that led to equality already exists in the Range Trie. Thus, if the bounds of the prefix already exist in the Range Trie structure, an update operation is just to update the prefix lengths, action pointers and contents of action memory of the nodes the prefix visited at each level of the Trie as explained in Section 3.4.1. After completely traversing the Trie no more changes are to be made to the Range Trie structure. As the Range Trie structure is designed to have most of the real bounds at the leaf nodes, in most of the cases, the existence of bounds is determined at the last level of the Range Trie. Figure 3.12 explains the process of updating a new prefix with already existing bounds where prefix $p2$ with bounds $[a2, c1)$.

- **Prefix with non-existing bounds**

After completely traversing the Trie, if the bound does not exist, the new bound should be inducted into the Range Trie structure. In Multiway Range Trees[27], authors discussed about updating the keys in the nodes and handling splits of the nodes. Figure 3.13 shows the process of splitting a node, in which each node in the Range Trie can support only three bounds. So if one more bound is inserted into the node, the node is split and the parent is updated. This has a worst case update time of $O(t \log_t n)$ where t is the number of keys at each node. This is because, after traversing the tree, if the bound does not exist, the bound should be inducted into the last node it visits. But if the leaf node is already full it cannot have any more bounds in it, thus resulting in the node to be split and updating

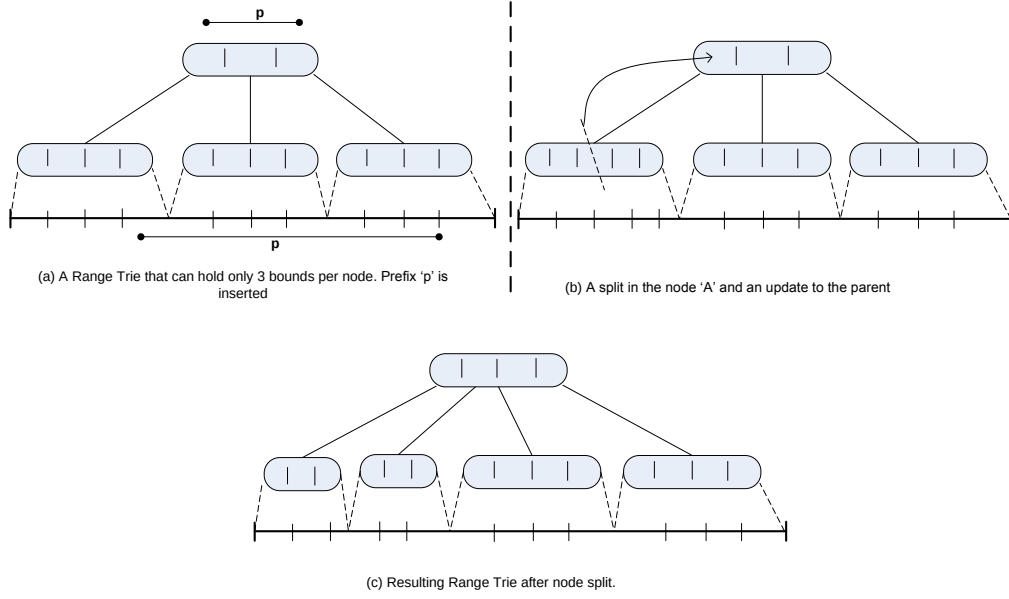


Figure 3.13: Node split when a prefix is inserted

the address span of the parent. So splitting a node requires to modify (add bounds to) the parent node that might also change the address span of the parent, and in the worst case, parent of the parent and so on until the root node. This approach limits the tree not be able to handle any lookup or update requests while processing an update request i.e. traversing down the tree and traversing back to the root node, updating all the nodes in its path. This heavily reduces the throughput of the lookup or update process.

Since the Range Trie is designed to be implemented in hardware, if the whole tree is updatable, worst case memory (memory required for maximum possible number of nodes at a level) is to be allocated per level in the Range Trie. This makes the design memory inefficient. As already mentioned, performing splits have a high performance overhead.

In our effort to support insertion of a prefix with non-existing bounds without significant performance overhead, it is decided to maintain extra levels at the end of the Range Trie which we call them as *spare levels* (updatable part). Each node in this spare level stores full length addresses as in Range Trees [27]. After traversing the Range Trie, if the bound of the prefix does not exist in the Range Trie, a new child node is created for that address range and the bound is inserted into that node as shown in Figure 3.14, where lower bound of prefix p does not exist in the Range Trie. Each node can have r bounds where $r = BW/N$, BW is memory band width and N is the address width ($N=32$ for IPv4 and $N=128$ for IPv6). If the node gets full, an extra node is created at the next level and inserted into that node. In this way a small sub-tree is created at each address range of the Range Trie and the sub-tree only expands when there are more insertions to the

spare levels. The fixed part of the Range Trie and the updatable part are as shown in Figure 3.4. If the node in first spare level has r keys, then the node can have $r+1$ child nodes. We decided to have a maximum of three spare levels in the Range Trie

structure. Thus, each sub-tree can have $\sum_{i=0}^2 (r+1)^i$ nodes and can absorb bounds up to r times the number of nodes. The update cost of inserting non-existing bounds is the same as inserting prefixes with existing bounds with a complexity of $O(\log n)$. However, handling a sub-tree for each of the address ranges of the Range Trie need a huge amount of memory. In order to avoid memory explosion, a fixed amount of memory is allocated for the spare levels which means only a fixed amount of nodes are available to insert new bounds. A detailed description of design and handling of spare levels will be discussed in section 3.5.

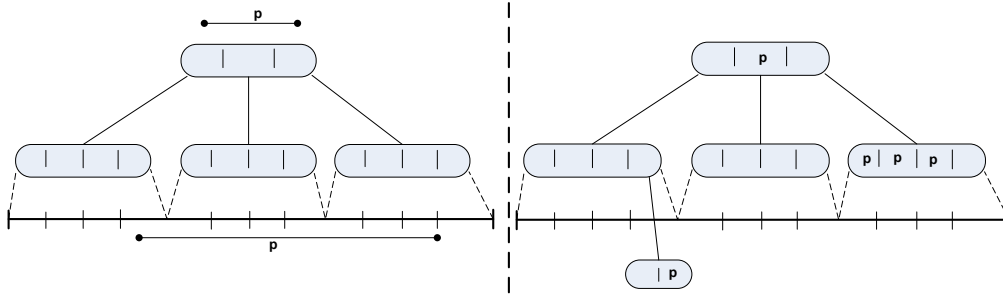


Figure 3.14: Inserting non-existing bounds

3.4.3 Deleting a prefix

Deleting a prefix means withdrawing the prefix from the Range Trie. While traversing down the Range Trie all the nodes that are in the address range of the prefix need to be updated which is handled in the same way as inserting a prefix. In the Range Trie structure we assume that the real bounds only exist at the leaf nodes. As mentioned in section 3.4.2, many prefixes might have the bound as their start or an end point. In order to avoid deleting a bound when there are other prefixes based on the bound, a counter is maintained to count the number of prefixes starting or ending on the bound. The counter for the bound is incremented when a prefix is inserted and is decremented when a prefix is deleted with the bound as its start or end point. A bound is completely deleted from the Range Trie, when the counter becomes zero which means there are no prefixes based on that bound in the Range Trie at that moment.

When a bound is deleted from a node, there is a possibility that bounds of the two adjacent nodes can fit into a single node. In this case the two nodes are merged together into one single node and the address range of the parent should be updated as mentioned in Multiway Range Trees[27]. This is called *merge* of the nodes. However, updating the address range of the parent might require updating the parent of the parent and so on

till the root node(similar to splitting the nodes when prefix is inserted). This also results in a high performance overhead. Figure 3.15 shows the scenario of deleting the prefix in Multiway Range Trees. In order to avoid changes to the Range Trie structure and in

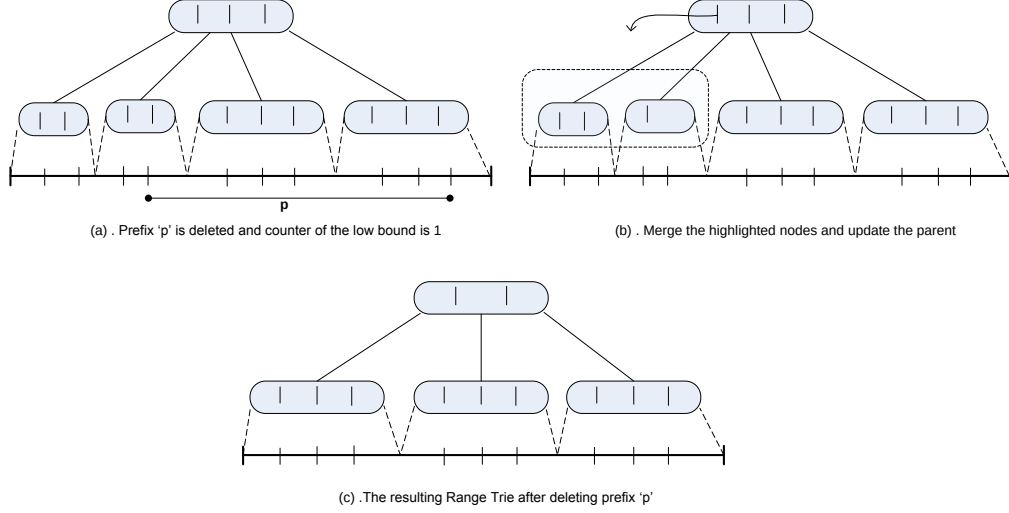


Figure 3.15: Deleting a prefix resulting in a node merge

turn to avoid the performance overhead, it is preferred not to delete the bounds in the fixed part of the Range Trie structure until it is completely rebuilt. However, counters are maintained for the bounds in the spare levels and are removed from the structure when the requirements are met. So *deletion of bounds is allowed only in the spare levels (updatable part)*. Deleting bounds in spare levels is discussed in detail in section 3.4.

3.5 Spare levels

The Range Trie structure is built on some base bounds extracted from a list of prefixes from the routing table. These k unique bounds extracted from these prefixes form $k + 1$ intervals in the address range $[0, 2^n)$. Whenever a new bound is inserted, a suitable place has to be found to induct the bound if it does not exist in the Range Trie. If the bound is placed in the existing structure, the Trie has to be traversed twice, once to the bottom during the prefix lookup and then back to the root updating the nodes. In order to reduce the complexity of inserting prefixes with non-existing bounds, three spare levels are assigned at the bottom of the Range Trie structure. This section discusses our effort to use the spare levels in the best possible way, maintaining the throughput of the Range Trie structure and reducing the complexity of updates compared to the existing designs.

When a prefix is inserted, final iteration stage of the fixed Range Trie determines whether the bounds already exist in the structure. If the bounds do not exist the output

of the final iteration stage would be the interval to which the bound belongs. If a node does not exist at the first spare level of that interval, a node is created and the new bound is inserted in that node. If a node already exists a suitable position for the bound is determined and inserted at that position depending on capacity of the node. If the node is already full a child is created at the next spare level and the bound is inserted to the child node. In this way, each interval in the address range $[0, 2^n)$ is eligible for a small 3-level sub-tree when non-existing prefixes end up in that interval.

A node in the sub-tree stores full length addresses in the available bandwidth as opposed to the Range Trie method in which variable length addresses are stored based on Range Trie rules mentioned in Section 2.2.1. Each node can have r bounds where $r = BW/N$, $BW = \text{Bandwidth}$ and N can be 8,16,32...addresswidth. If the node in first spare level has r keys, then the node can have $r + 1$ child nodes. As the number of spare levels is three, each interval can hold a three level sub-tree. Thus, each sub-tree

can hold $\sum_{i=0}^2 (r + 1)^i$ nodes and can absorb bounds up to r times the number of nodes.

As mentioned earlier, a suitable position is to be found such that all bounds in a node are in ascending order of their values when a new bound is inserted into the sub-tree. If the node is already full, the newly inserted bound traverses to the next level of the sub-tree and the path to be traversed is determined by the processing unit at that level. Prefix lengths, action pointers and contents of the action memories are updated in the same way as is done for the fixed part of the Range Trie. Counters are placed corresponding to each bound in a node and are incremented whenever a new prefix that starts or ends on the bound is inserted.

However, a bound that is inducted into a sub-tree may not find a position in the sub-tree. Three possibilities exist for a bound to be denied a position in the sub-tree consequently denying the update operation:

- All nodes in the traversal path of the sub-tree are full, even though the sub-tree is not full.
- Sub-tree is completely full.
- Total memory allocated for spare levels is full.

In the first case, the sub-tree is restructured to distribute the bounds evenly in the sub-tree. Rebuilding the sub-tree involves reading all the contents of the nodes at the three spare levels of the sub-tree, re-distributing the bounds among nodes and replacing the old sub-tree with the new rebuilt sub-tree. This process takes $(r + 1)(r + 1) + \text{time to redistribute the bounds} + (r + 1)(r + 1)$ cycles (worst case) and the Range Trie structure would not accept any insertions during this period. In the second and the third cases, when the sub-tree is full or the total memory is full the whole Range Trie structure is rebuilt. During rebuild, lookup requests can be processed, as the lookup does not change the contents of the Trie structure.

In the worst case, if all the bounds entering an interval in the range $[0, 2^n)$ are inserted in-order, only the extreme right part of sub-tree becomes full and after inserting $r + r + r$ bounds the sub-tree could not absorb more bounds if the inserting bounds are still in-order. An example of this scenario is given in Figure 3.16 and Figure 3.17. Consider a two level sub-tree with memory bandwidth of 256 bits for node data and accepts IPv4 addresses. So each node can have 8 bounds and the sub-tree can hold up to 10 nodes and 80 bounds. Consider a set of 8 prefixes of length x that arrived in-order and are inserted into the sub-tree as shown in Figure 3.16. Now the root node and ninth child of the root node are full. Even though the sub-tree can theoretically store 64 more bounds into it, because of the structure of the sub-tree, it cannot absorb a new prefix p as the ninth child is already full. So the sub-tree is rebuilt as shown in figure Figure 3.17, and now the prefix p can be inserted into the sub-tree. As we chose not to traverse back the tree doing splits and merges of the nodes, as an alternative, we chose to rebuild the sub-tree or the whole Range Trie depending on the requirement.

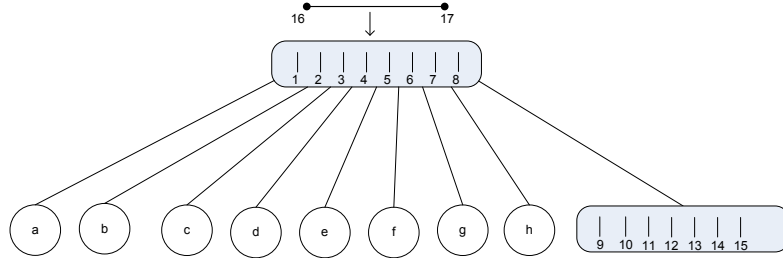


Figure 3.16: Sub-tree in spare level

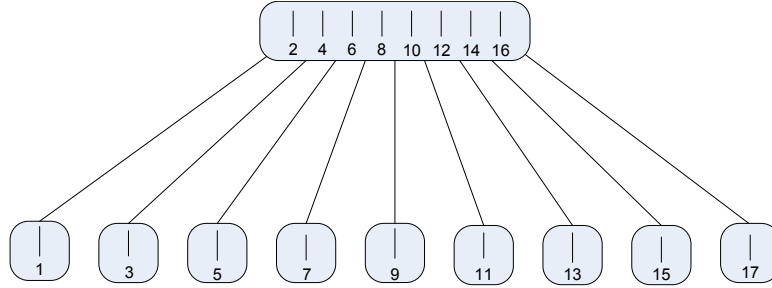


Figure 3.17: Rebuilt sub-tree in spare level

If a prefix is deleted from the spare levels, the counter of the bound is decremented. A bound is completely removed from the sub-tree if the counter becomes zero and there are no child nodes belonging to the node. If a child node exists even if the counter is zero, the bound is removed when the sub-tree or the whole Range Trie is rebuilt. If the child node does not exist, bounds that are to the right of the deleted bound in the node and the corresponding prefix lengths, action pointers of those bounds are shifted to the left. As the action pointers do not change, contents of action memory are not changed.

Thus allowing three spare levels at the end of the static Range Trie structure allows to maintain the throughput of the Range Trie method, until one of the sub-trees is full or if the maximum available nodes in the spare levels are already in use. This section discussed an approach to handle the spare levels that are available at the end of the Range Trie structure which improves the update time and hence the throughput of the Range Trie method compared to the existing designs.

3.6 Rules to update the Range Trie data structure

In sections 3.2 and 3.4, modifications to the Range Trie data structure to support LPM and dynamic updates were presented. An update(Insert/Delete) of a prefix may change the contents of the node such as prefix length and action pointer. This section presents the rules to be followed in order to change the contents of a node that the prefix visits during its traversal in the Range Trie structure.

If the bounds of the prefix exist in the fixed part of the Range Trie, then at each visited node, updating (insert/delete) a prefix requires the following:

- Inserting a prefix requires to
 - update the prefix length if it is longer than the previously stored one in the node.
 - update the action pointers corresponding to the nodes whose prefix lengths are updated.
 - update the action at the address determined by the action pointer.
- Deleting a prefix requires to:
 - update the prefix length with another supplied prefix length, if the length of deleting prefix length equals the existing length of the prefix.
 - update the action pointers corresponding to the node whose prefix lengths are update.
 - update the action at the address determined by the action pointer.

If the bounds of the prefix does not exist in the fixed part of the Range Trie the at each node visited, updating(insert/delete) a prefix requires the following:

- Inserting a prefix requires to
 - If a sub-tree exists at the address range determined at the leaf nodes of the fixed Range Trie
 - * Inserting the bounds if they already does not exist in the sub-tree, and updating the prefix lengths and action pointers.
 - * Otherwise, incrementing the counter along with updating the prefix lengths and action pointers.
 - If a sub-tree does not exist:

- * A new node is created in the updatable part and the bound is inserted into the node.
- * Prefix length is stored and a new action pointer is created.
- Deleting a prefix requires to:
 - decrement the counter
 - * if the counter is greater than 1.
 - * if there is a child node in the address range related to the bound.
 - delete the bound if the counter is 1 and the node does not have a child in the address range related to the bound

3.7 Complete Range Trie Structure

In the above sections, structural changes to the Range Trie in order to support longest prefix match and dynamic updates were presented. This section aims to integrate the existing Range Trie structure with the supporting update structure and spare levels to form the complete Range Trie design with support to dynamic updates.

The initial Range Trie structure was pipelined dividing each level of the Range Trie into a single stage in the pipeline. To further increase the throughput, each stage is again divided into memory and processing stages. To support longest prefix match, prefix lengths were stored at the nodes according to the two prefix storing rules mentioned in Section 3.2. Three spare levels of Range Trie, action pointers and action memory at each level were introduced to reduce the update time and to maintain the throughput of the Range Trie structure.

The Range Trie structure was built applying the five Range Trie rules mentioned in Section 2.2.1 on the set of k prefixes that divide the address range $[0, 2^n)$ into $k + 1$ intervals. Now, each of these address intervals match with multiple prefixes of different lengths. The Range Trie structure spanning multiple levels has to match an incoming address A_{IN} to a longest matching prefix among the prefixes spanning the interval. In order to support dynamic updates three spare levels were introduced into the Range Trie structure to store new bounds. Each level of the Range Trie consists of a number of nodes, except in the first level which just has one (root) node. The algorithm directs the incoming address to traverse the Range Trie structure and finally match to the best matching prefix i.e. by length.

There are two types of nodes in the Range Trie structure. Nodes in the fixed part of the Range Trie and nodes of sub-trees in spare levels. Data structure of each of these nodes is as follows

- **Node structure in the fixed Range Trie**

Each node in the fixed Range Trie contains the following information

- *Comparison values.* These contain the values to be compared with the incoming address.
- *Control values.* These contain information about number of comparisons to be performed, configuration of the comparisons and information suggesting a shared common prefix, shared common suffix and address alignment.
- *Pointer to the child nodes* in the next level.
- *Prefix lengths* of each child of the node.
- *Action pointers* of each child of the node.

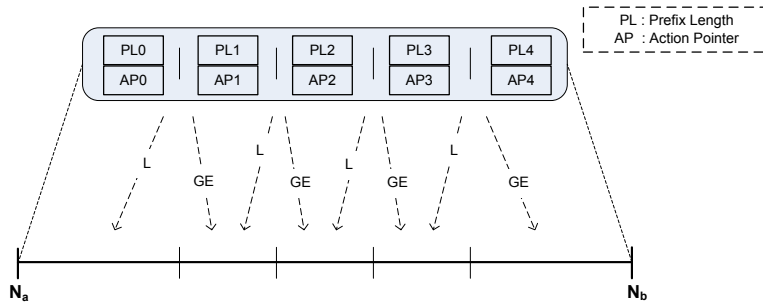


Figure 3.18: Node structure in the Range Trie

• Node structure in the spare level sub-trees

Each node in the spare level contains the following information

- *Comparison values.* These contains the values to be compared with the incoming address. These are full length addresses thus full length comparisons are performed.
- *Control values.* Since full length addresses are stored at the nodes in the spare levels, no. of comparisons at each node is always constant. So, no comparator configuration bits are required. This reduces the number of control bits, compared to the fixed part of the Range Trie.
- *Pointer to the child nodes* in the next level.
- *Prefix lengths* of the children of the node.
- *Action pointers* of the children of the node.
- *Counters for the bounds* in the node. This gives the number of prefixes starting or ending on a bound to keep track if we need a bound or not(i.e. bound should exist if *counter* > 0, can be deleted if *counter* = 0).

These nodes spanning into different levels form the complete Range Trie structure. Consider a small Range Trie as shown in Figure 3.20. Structure of fixed part of the Range Trie nodes and the spare level(updatable part) nodes are as shown in Figure 3.18 and Figure 3.19. Assume an incoming address A_{IN} for lookup. The lookup starts with A_{IN}

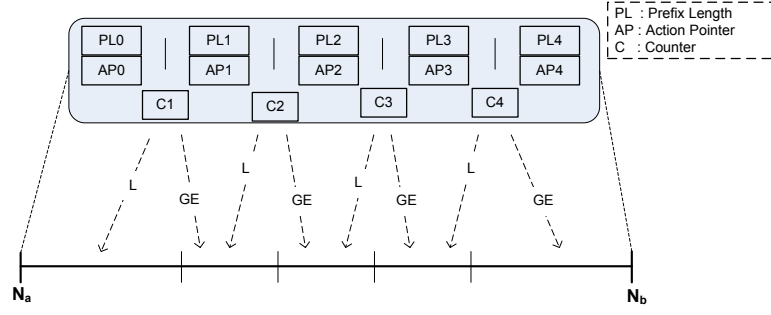


Figure 3.19: Node structure in the spare levels

visiting the root node. The path to reach the next level is determined by the comparisons performed at the node using the necessary information stored at the node. This process continues until A_{IN} reaches the third and last of the spare levels. At each level, A_{IN} keeps track of the prefix length stored along that traversal and action corresponding to the respective action pointer. When A_{IN} reaches the leaf node, if a sub-tree exists in that interval, then the process continues until reaching the third spare level. Otherwise, prefix matched after reaching the leaf node and the corresponding action is the final result.

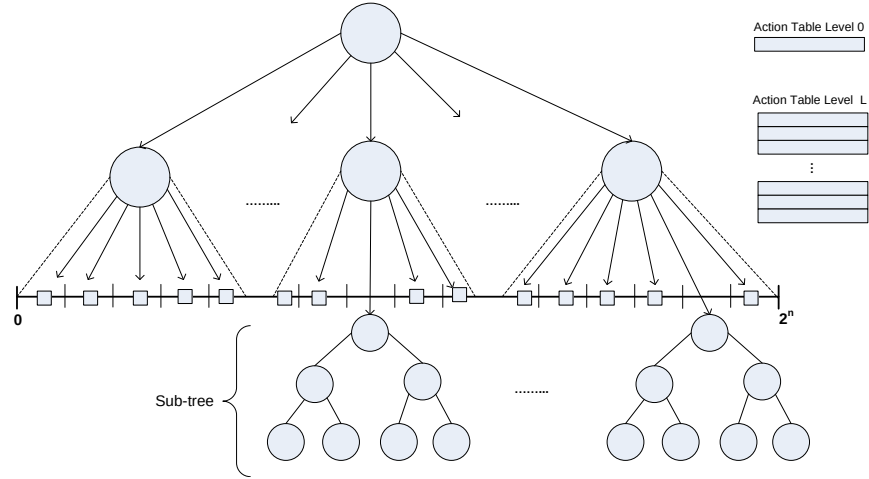


Figure 3.20: Complete Range Trie Structure

Thus the final modified Range Trie structure performs longest prefix match in $O(\log n)$ time for each incoming address A_{IN} and has fast update times when new prefixes are inserted or the existing prefixes are deleted.

3.8 Summary

In this chapter, a modified Range Trie structure to support LPM and dynamic updates has been presented. At each level of the Range Trie, an incoming address retrieves the prefix lengths, action pointers, actions corresponding to the prefix along with the comparison values, while performing lookup. The incoming address remembers the longest prefix it has encountered in its traversal upto that level and proceeds to the next level of the Range Trie. Two prefix storing rules were stated in section 3.2 to minimize the memory overhead incurred to support LPM. Section 3.3 discussed the lookup procedure in the Range Trie supporting LPM. Action pointers and one action table at each level of the Range Trie were introduced in section 3.4 to minimize the number of write operations to the action table which otherwise would incur a huge performance overhead. To support insertion of prefixes with non-existing bounds, we partitioned the tree into fixed and updatable parts as opposed to the existing solutions in the literature that perform splits/merges of the nodes. This approach (a) minimized the memory overhead by avoiding the need to allocate memory for maximum number of children per node in the Range Trie (b) reduced the complexity of updating the Range Trie nodes. The modified data structure has:

- 1 write to memory per tree level per prefix bound during an update.
- no feedback to go back to the tree node performing splits/merges of the nodes.
- a sub-tree rebuild or a full Range Trie rebuild to avoid splits/merges.

Since the Range Trie does support feedback, it is less efficient in modifying the nodes. But when the Range Trie is pipelined updates and lookups can co-exist in different stages of the pipeline which is not possible when supporting feedback. A detailed description of pipelining the Range Trie structure will be given in section 4.2. This effort was to modify the existing Range Trie structure to support updates which offers high throughput and low update times compared to the existing contemporary designs.

Range Trie Hardware Design

In this chapter, hardware design of the modified Range Trie structure to support LPM and dynamic updates will be presented. Based on the data structure changes proposed in the previous chapter, the necessary hardware implementation of the logic and memory management will be discussed in detail. The main objective of the Range Trie design by [26] is to build a fast, efficient and scalable design exploiting the inherent characteristics of the algorithm. However the previous version of the design did not support LPM and dynamic updates. This effort is to improve the design to support LPM and dynamic updates to the Range Trie structure without compromising on the main objectives of the algorithm.

In the Range Trie method, lookup starts with the incoming address visiting the root node. Based on the comparisons performed at each level, A_{IN} traverses the Range Trie. In the previous design, the process ends when A_{IN} reaches the leaf node where it is matched to an address range. In the modified structure, an incoming address should be matched with a prefix that best matches the incoming address in terms of length and updates are to be performed whenever there is a change in the network topology. So, each level of the Range Trie can be defined as a group of processes:

- Visiting the node.
- Perform comparisons and decide which branch to follow based on the comparison results.
- Select the suitable prefix length and action pointer based on the comparison results.
- Update the node information during update requests.

Visiting a node involves reading the node contents from memory, the Processing Unit performs the necessary computations on the node data to determine the path to be traversed and the Update unit updates the prefix lengths, action pointers and node information at necessary instances.

There are several specifications that determine the efficiency of the Range Trie design. They are as follows:

- *Memory bandwidth*: The allowed memory bandwidth to store the node information.
- *Address width* : The Range Trie design should support IPv4 and IPv6 addresses thus the address width (W) would be 32 or 128 bits respectively. This determines the number of comparators available for a given memory bandwidth.

- *Range Trie levels* : This gives the number of levels in the Range Trie structure.
- *Memory Depth* : The required memory depth at each level of the Range Trie. As the number of nodes per level increases from top to bottom, depth of the memory also increases.
- *Update time* : The time required to process an update request.

All the details that lead to a complete pipelined Range Trie design that supports LPM and dynamic updates will be presented in this chapter. A brief overview of the Range Trie hardware design by George Stefanakis in [26] that supports lookups based on range match is presented in section 4.1. Section 4.2 discusses the pipelined hardware design of the initial Range Trie. Section 4.3 discusses prefix traversal in the pipeline as described in 3.4.1 and the additional memory structure and necessary components to update the Range Trie structure. The hardware description of the spare levels that include the memory structure and a modified Processing Unit for the spare levels will be discussed in section 4.4. A complete description of the Range Trie pipeline integrating the update structure to the base design will be presented in Section 4.5. The chapter concludes with a summary in section 4.6.

4.1 Background

In this section a brief overview of the Range Trie hardware designed by George Stefanakis[26] is presented. As mentioned earlier, each level of the Range Trie comprises of a number of nodes except the first level. When an incoming address A_{IN} visits a node, the Processing Unit determines the next branch to be taken by A_{IN} , based on the information stored at the node. When implemented in hardware, each node contains some information which is read from memory. This section gives an overview of the Processing Unit, the information stored in the node data structure, the memory structure and an efficient memory addressing scheme.

4.1.1 Processing Unit

The basic operation of a Processing Unit is, given an incoming address A_{IN} and the node information (a pre-determined set of values), comparisons are performed to determine the traversal path of A_{IN} to reach the next level of the Range Trie. According to the Range Trie method, comparisons are performed on parts of addresses at each level of the Range Trie. The operational structure of the Processing Unit can be divided into a four step process as given in Figure 4.1.

- *Select parts of incoming address* : Based on the information stored at the visiting node, parts of the incoming address A_{IN} are selected. First the common prefix or common suffix is stripped off by feeding the incoming address A_{IN} to a shifter. If needed, a subtraction is performed to align the shifted address according to Rule5 of the Range Trie method. The resulting A_{IN} is input to $k - 1$ comparison value constructor units. As mentioned earlier, the number of comparators used in a Processing Unit is $k = BW/W$ and k^{th} comparator always performs full

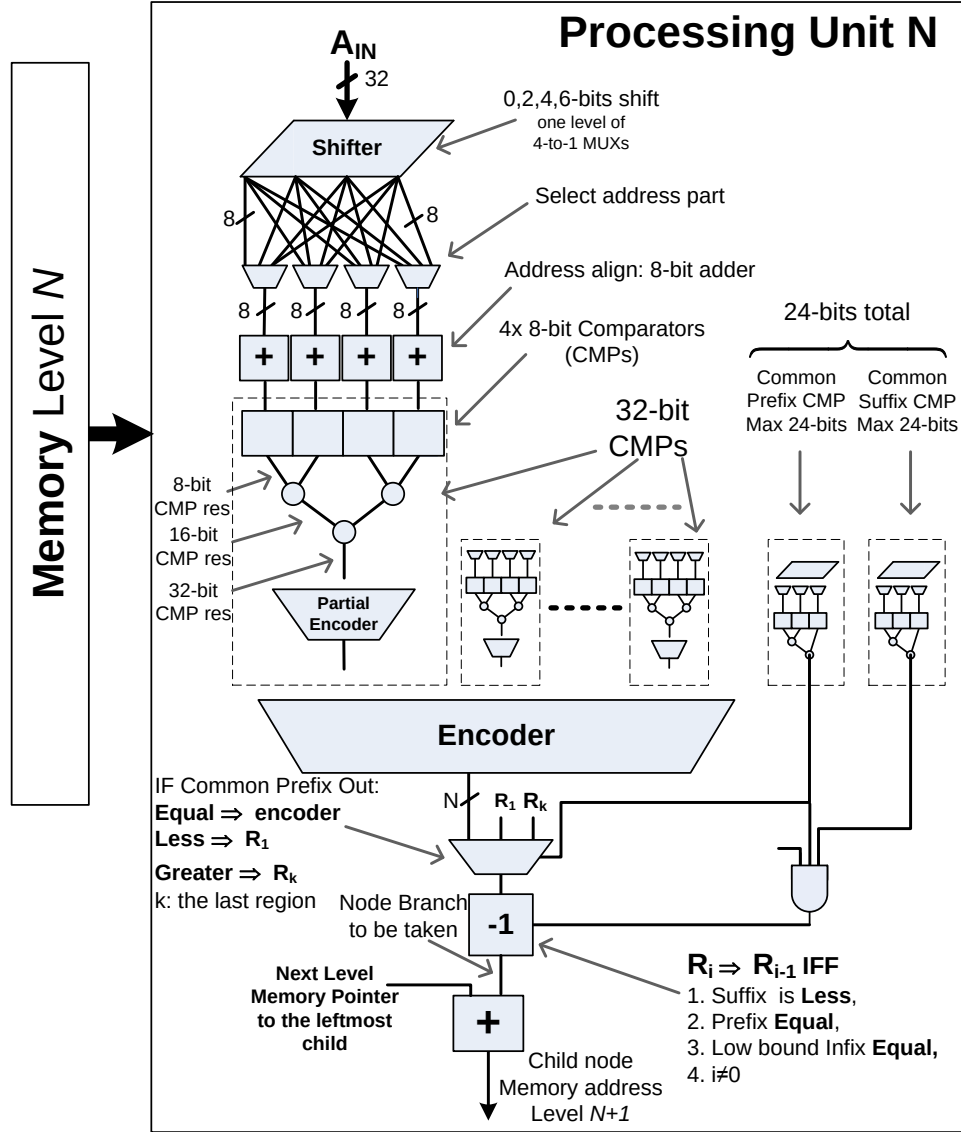


Figure 4.1: Processing Unit

length comparisons. In order to select the parts of the incoming address several parameters are to be stored at the node such as *Shift_ctrl*, *Start_byte*, *SUB* and *Comparator_modes*.

- *Perform comparisons* : Using the k variable width comparators, comparisons are performed between the selected parts of the address and the pre-determined values stored at the node. A common prefix and a common suffix comparison that is $W - 8$ bits wide is performed parallel to the $k - 1$ comparisons. Thus the pre-determined values to be stored at the node are comparison values, common prefix or common suffix values and common prefix/suffix encoded masks. The comparison

results show whether the incoming address A_{IN} is greater-equal or less than the pre-determined comparison values stored at the node.

- *Interpreting the comparison results* : If c pre-determined comparison values exist at each node, comparison results of the k comparators in the previous step are used to determine the branch to be taken among the $c + 1$ possible branches, to reach the next level of the Range Trie. Based on the comparator comparisons, a partial encoder indicates the range to which the selected part of A_{IN} belongs to, and also indicates if the selected part of A_{IN} is equal to one of the c comparison values at the visiting node.
- *Decide the next branch to follow* : As mentioned earlier, apart from the k -comparisons performed on the comparison values, a comparison is also made on the common prefix/suffix values. The result of the prefix/suffix comparison should also be considered in order to make a final decision on the next branch among the $c + 1$ possible branches. In a common prefix comparison If the comparison leads to Less than, the branch to be taken is 0. If the comparison leads to Greater than, the branch to be taken is $c + 1$ If the comparison leads to equal, then the branch to be taken is the determined branch from previous comparisons.

The available memory bandwidth (BW) and the address width (W) are the main factors that affect the Range Trie design. Considering these factors, there are several parameters that are needed to perform the necessary processing at the Processing Unit. They are detailed as follows:

- *Shift_ctrl (2-bits wide)* : The number of bit-shifts performed at the shifter is determined by this value. 0,2,4,6 bit shifts are performed at the shifter.
- *Start_byte($\log(W/8)$ bits)* : The part of A_{IN} to be subtracted is determined by this parameter. This is $\log(W/8)$ bits wide.
- *SUB* : This is the 2's complement of the value to be subtracted from the selected part of the incoming address A_{IN} . If the subtraction is not needed the value will be 0.
- *Common prefix/suffix mask* : This is $(2 * \log(W - 8))/2$ bits wide. The first half is the binary representation of $CP/2$ and the second half is the binary representation of $CS/2$. When there is no shared prefix/suffix, then the respective values will be zeros.
- *Comparison modes* : This determines the comparison widths of each of the $k-1$ comparators at each Processing Unit. The width of comparison modes is 3-bits for IPv4 addresses and 12-bits for IPv6 addresses. Comparators are disabled when not in use. Since the k^{th} comparator, when enabled, performs full length comparisons, only one bit is needed to indicate whether the comparator is enabled or disabled.
- *Common prefix/suffix value*: As mentioned earlier, the final decision on the outgoing branch to be taken is decided based on the common prefix/suffix comparison.

A $W - 8$ bit value holds both the common prefix and common suffix values if any. The remaining bits will be zeros.

- *Comparison values:* The k comparators at each Processing Unit perform variable width comparisons according to the comparator modes. A comparator can be disabled with the comparator mode if the comparator is not used. The comparison values are the pre-determined values stored at the node in order to compare with the incoming address A_{IN} . If a comparison is disabled, the values are set to zeros. The first $(k - 1) * W$ values hold the comparison values. The final W bits either store the comparison value of the k^{th} comparator when used or the $W - 8$ bit wide prefix/suffix comparison.

4.1.2 Memory Structure

The first step in the lookup process is visiting a node. As mentioned earlier, visiting a node means retrieving node information needed by Processing Unit from the memory. At each level of the Range Trie structure, incoming address A_{IN} visits a new node i.e memory is accessed by A_{IN} once at each level of the Range Trie. This section discusses the organization of nodes at each level of the Range Trie structure and a memory addressing scheme suitable to the Range Trie structure.

4.1.2.1 Memory organization

This section discusses the organization of the Range Trie nodes in memory. The basic effort was to minimize the size of memory in terms of width and also in terms of depth. The amount of unused memory space should also be minimized. Consider an L level

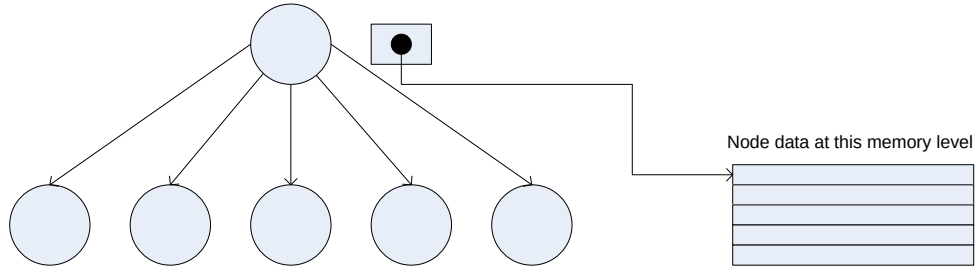


Figure 4.2: Memory organization

Range Trie structure. Memory should be allocated at each level that stores the information of the nodes corresponding to that level. A node in memory corresponds to one location in memory. So, storing nodes of a level in contiguous memory locations reduces the size of memory. There should be no unallocated memory locations between nodes. This also reduces the address width needed to address the memory locations. The first level of the Range Trie just contains the root node. Thus, instead of storing the node information in memory, it can be stored in a register. As we move down the Range Trie structure, the number of nodes at each level increases. Thus the required memory depth

increases at each level of the Range Trie. If the nodes are stored non-contiguously in memory, a worst case memory allocation is needed to fit all the possible number of nodes at each level, thus leading to a large unused memory space.

4.1.2.2 Memory addressing

This section discusses the memory addressing scheme designed to address the memory at each level of the Range Trie structure. Apart from leaf nodes, each node in the Range Trie structure points to its child nodes. During the Range Trie traversal, an incoming address A_{IN} traverses to one child of the visiting node based on the result of the Processing Unit. As mentioned in the previous section, nodes are stored in continuous memory locations at each level to minimize the unused memory address space. But a pointer to the child nodes should be stored at each node in order to traverse the Range Trie. Storing a pointer for each child consumes a lot of memory. Since the nodes at each level are stored in contiguous memory locations and the Processing Unit determines the child to be visited, storing a pointer to the first child i.e. the left most child will satiate our need. So the pointer stored at a node is added to the result of the Processing Unit to get the address of the child node to be visited in the next level. If the nodes are stored non-contiguously in memory, pointer to each child should be stored at a node that leads to a large memory overhead for storing pointers.

4.1.3 Node data structure

This section discusses the data structure of the node and the information that should be stored at the node. All the information needed to select parts of addresses to perform variable width comparisons at a Processing Unit was described in section 4.1.1 and the pointer memory needed to traverse to the child nodes was presented 4.1.2. Based on this information, node information can be classified into three parts:

- *Comparison Values*: This contains the values to be compared against the selected variable width parts of the incoming address A_{IN} . Since the main objective of the Range Trie method is to increase the branching factor at each node, a node should be able to store all the possible values to be compared. So a memory bandwidth (BW) wide memory is allocated for comparison values.
- *Control values*: These values determine the parts of the address to be chosen and the comparator modes. As mentioned in detail in section 1.1, these values are *shift_ctrl*, *start_byte*, *subtract_value(SUB)*, common prefix/suffix mask and the comparator modes. The width of these values is determined by the available bandwidth (BW) and the address width (W).
- *Pointer*: This gives the pointer to the leftmost child of a node. The width of the value depends on the depth of the memory at the next level of the Range Trie.

A representation of the node data structure is shown in Figure 4.3.

This section presented a brief overview of the design details of the existing Range Trie structure. At each level node information is retrieved from memory and comparisons

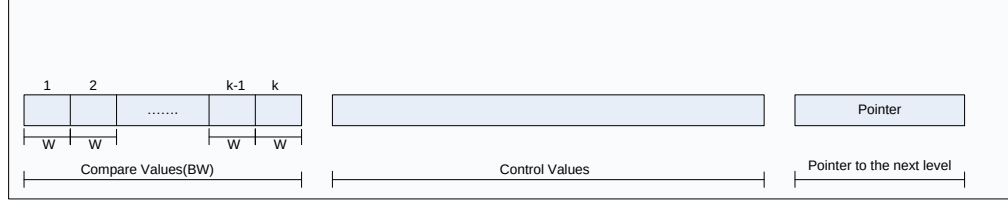


Figure 4.3: Node data structure

were performed in the Processing Unit. Based on the determined child to be visited and the pointer value, node information of the next node to be visited is retrieved from the memory at the next level.

4.2 Pipeline Structure

An incoming address only visits one node at each level of the Range Trie structure and performs comparisons on the contents of the node. This is an iterative process until the incoming address reaches a leaf node. By pipelining the traversal of the Range Trie structure high throughput can be achieved which is one of the objectives of the Range Trie method. As this is an iterative process, each level of the Trie can be considered as an *iterationstage* of the pipeline. Thus, at each clock cycle one new A_{IN} can be fed to the pipeline. Being a two step process each iteration stage can be further pipelined into two stages, a Memory access stage and a Processing stage as shown in Figure 4.4. This reduces the clock cycle delay thus improving the speed of the pipeline. This section describes the integration of the memory and Processing Units presented in section 4.1 into a pipeline.

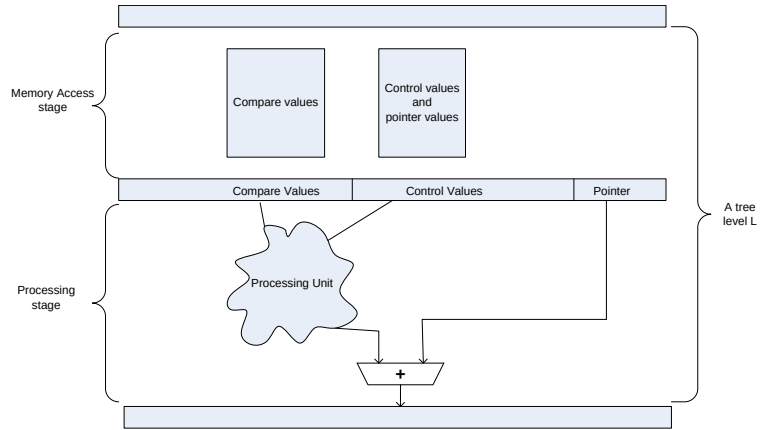


Figure 4.4: One level in the Range Trie Structure

Consider one level of the Range Trie structure. First, A_{IN} should retrieve node information from the memory and perform comparisons to determine the next node

to be visited. The memory structure and the processing unit together correspond to one level of our Range Trie structure as shown in Figure 4.4. One A_{IN} is input to the pipeline every clock cycle. In the first clock cycle it retrieves information from the node. In the next cycle this data is fed to the Processing Unit. The Processing Unit determines the child to be visited in the next stage. The pointer value stored at the node should be added to the number of the child to determine the memory location of the child node.

Several control signals are needed in order to handle the pipeline. They are as follows

- *Memory Read/Write*: To allow reconfiguration of the Range Trie structure or to alter the memory contents a write signal is propagated. In order to check the contents of the memory for debugging purposes a read signal is propagated. As read and write of memory are not needed simultaneously we just need one bit to determine Memory Read/Write (1 / 0) operation.
- *Memory Enable*: The memory units might be disabled when not used. Also when reconfiguring the Range Trie structure or while altering the contents of some memory, the memory units at other levels of the Range Trie can be disabled. The width of this depends on the number of levels of the Range Trie structure.
- *Lookup*: A control signal is needed to dictate the kind of operation performed in the pipeline. As lookup is different from Memory Read/Write (needed for updating purpose), an extra signal is needed to dictate this operation.

Consider an L level Range Trie structure. Since lookup is an iterative process, each tree level in Figure 4.4 can be replicated L times to make the complete pipeline of the Range Trie. It takes 2 cycles to traverse each tree level. Since it takes L-1 iterations to traverse the whole Range Trie, it takes $2*(L-1)$ cycles to output the result of a lookup request. The block diagram of the complete pipeline is shown in Figure 4.5.

4.3 Updates

An update operation inserts or deletes a prefix from the Range Trie structure. Since each prefix has two bounds, the prefix bounds are to be fed into the pipeline in two different cycles. To perform an update, a lookup is performed on the two bounds of the prefix and the node memories that are in the range of the prefix are updated as the prefix propagates down the pipeline. This section first describes the traversal of the prefix in the pipeline in order to perform an update operation. Then the hardware structure designed to work parallel to the Processing Unit, to update the prefix lengths and the action pointers is presented. Then the change in the node data structure and its inclusion into the memory structure is discussed.

4.3.1 Prefix Traversal

The low bound and high bound of a prefix are fed into the pipeline one after the other in different (consecutive) cycles. A lookup is performed on both the bounds in

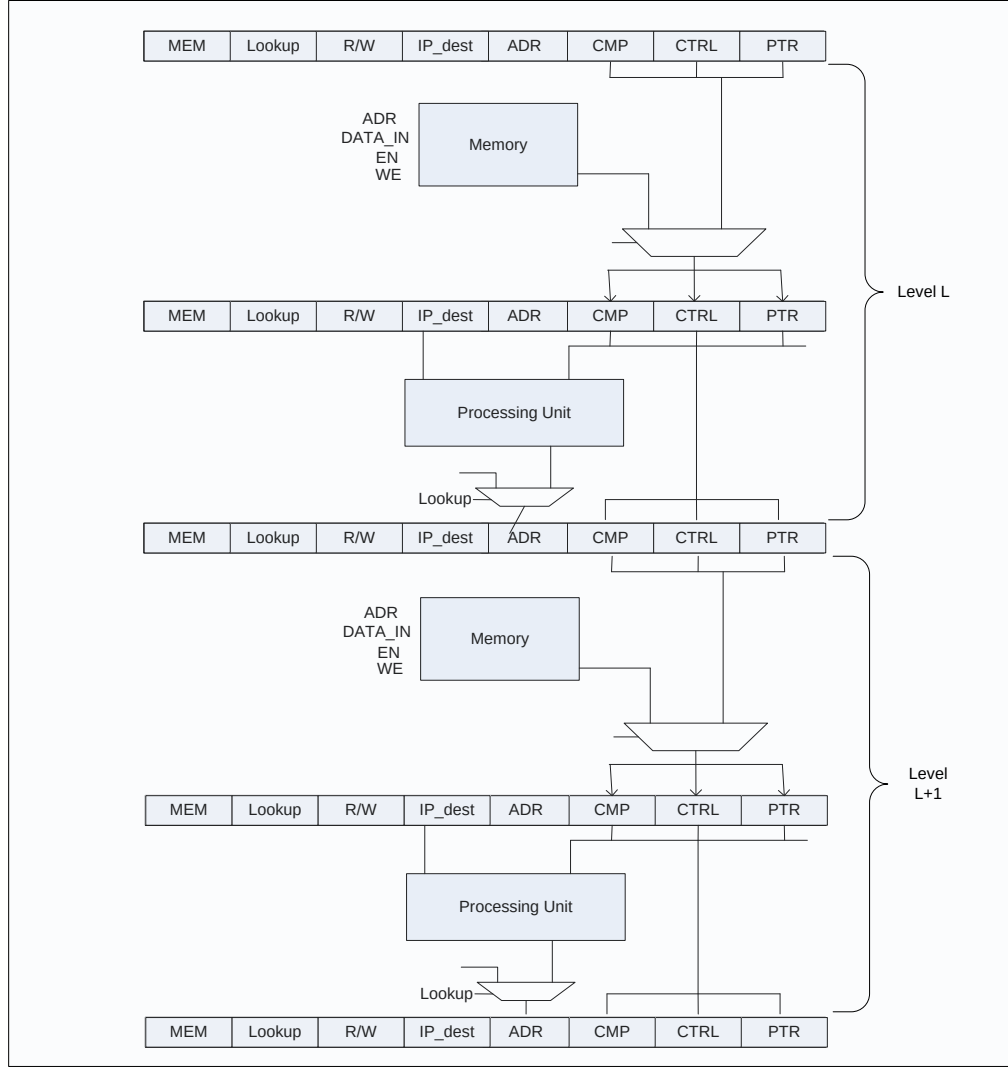


Figure 4.5: Pipeline of the initial design

order to find the Range Trie nodes related to the updated prefix. As we already know, the result of the lookup at each level of the Range Trie is the branch to be taken to visit the next node in the path of the bound. If the lookup result of both the bounds is equal then both the bounds reach the same node in the next level. Otherwise, differences in the lookup results of the bounds mean that the bounds go to different nodes at the next level which leads to a prefix split. This section describes a way to realize prefix split in the pipeline and a way to update the nodes in the range of the prefix.

To determine a prefix split, the lookup result of the low bound and high bound are to be compared. Since they are fed into the pipeline in two consecutive cycles, the result of the low-bound should be stored at the Processing Unit stage until the high-bound is

processed at the Processing Unit. If the comparison of lookup results of low-bound and high-bound leads to an inequality then we know that prefix is split at that node. Then the prefix lengths and action pointers of child nodes whose address ranges are a subset of the address range of the prefix at that node are updated. Since memory is one level above the Processing Unit, a write back in the pipeline is required to write the modified contents to the memory. At this moment, no other request to read the memory at that stage can be processed. After the prefix split, both the high bound and low bound move independent to each other. Now they traverse the pipeline independently updating the memories of the nodes they visit in their path. When the low-bound is processed at the Processing Unit, the high-bound reads the node information of the visiting node. Thus, when the low-bound writes back the updated memory contents, the high bound has already finished its memory access and there is no hurdle for the low-bound to perform write back. Then after the high bound is processed at the Processing Unit stage, it performs write back of the memory contents of the node it has visited in its traversal path. A bubble of two cycles is needed in the pipeline in order to process the write back requests of both the low-bound and high-bound. So, after the pipeline is fed with the bounds of a prefix in two consecutive cycles, no other addresses should be fed for two more cycles. The above process is explained with the following example.

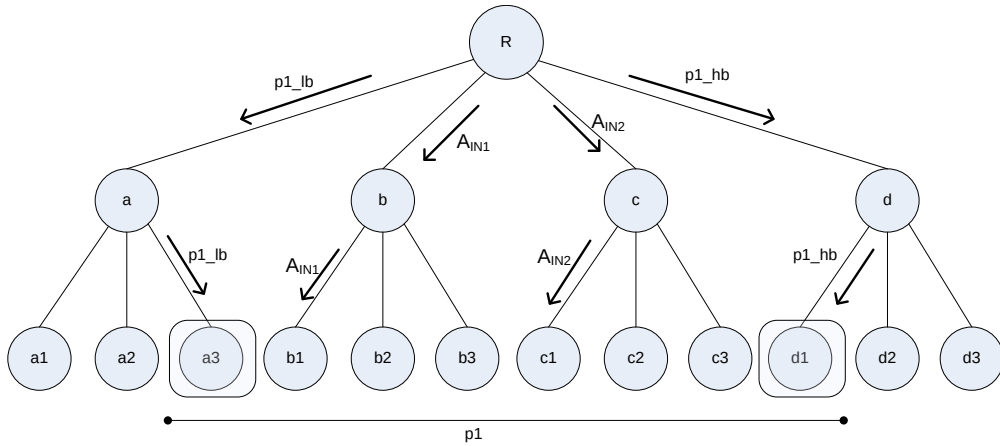


Figure 4.6: Prefix Traversal in the Range Trie

Assume a three level Range Trie as shown in Figure 4.6 and its corresponding pipeline in Figure 4.7. Let addresses A_{IN1} , prefix $p1$ and A_{IN2} be fed into the pipeline in the same order. A_{IN1} and A_{IN2} are addresses for performing lookup. As we know, a lookup traverses down the tree remembering the longest matching prefix at each level of the Range Trie. In the first cycle A_{IN1} is fed into the pipeline and after traversing the three level Range Trie the lookup result is known after the sixth cycle. The propagation of A_{IN1} in the pipeline is shown in Figure 4.7. In the second cycle low-bound of prefix $p1$ is fed into the pipeline and high-bound is fed in the third cycle. From Figure 4.6, we can see that $p1$ splits at the root node. In the pipeline, prefix split is known only after high-bound is processed at the first level Processing unit i.e. after the 4th cycle. Node b

and *Node c* are completely inside the range of the prefix and hence they are updated at the root node. Since the prefix split is known in the 4th cycle, the updated prefix lengths and action pointers are written back in the 5th cycle. Now the prefix is split and low-bound and high-bound move independent to each other in the Range Trie. In the 5th cycle the low-bound have finished its processing at level 2. Since the prefix is already split, the low-bound updates the contents of the visiting node. Since address range of node *a3* is a subset of the address range defined by *p1*, prefix lengths and action pointers corresponding to *a3* are updated in the 6th cycle. The high-bound is fed to the Processing Unit at level 2 in the 6th cycle, so write back of memory by low-bound does not hinder the propagation of high-bound. The processing of high-bound at level 2 is

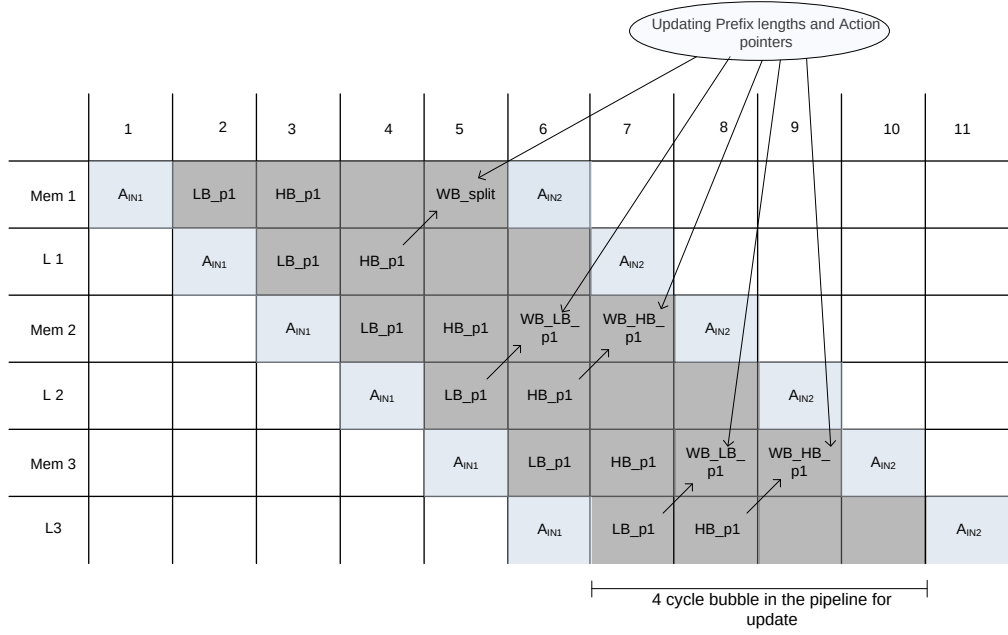


Figure 4.7: Prefix Traversal in the pipeline for the scenario of Figure 4.6

finished in the 6th cycle, so the write back of updated contents of *d1* to memory *Mem2* is done in the 7th cycle. In 6th and 7th cycle, no memory read requests are processed at *Mem2*, since the low-bound and high-bound perform *write.back* in 6th and 7th cycles respectively. In order to realize this, lookup or update requests should not be fed into the pipeline for two cycles after inserting the prefix. In this case, no other request should be fed in the 4th and 5th cycle. In the same way the update scenario of *p1* continues to the third level of the Range Trie. Then A_{IN2} is fed into the pipeline for lookup in the 6th cycle. The propagation of A_{IN1} , A_{IN2} and *p1* in the pipeline is shown in Figure 4.7.

4.3.2 Storing prefix lengths and action pointers:

In the Range Trie method, each incoming address is matched to a range in the interval $[0, 2^n)$. The node data structure of the base design that performs range match was

presented in section 4.1.3. In order to support longest prefix match and dynamic updates, prefix lengths and action pointers should be stored at the node as described in section 4.3.1 and 4.3.2. So, along with the current node information, prefix lengths and action pointers should be added to the fixed part of the Range Trie data structure. The memory required for this purpose is as follows:

- *Prefix length:* If a prefix p matches with an incoming address A_{IN} the first PL (prefix length) bits of the prefix and A_{IN} will be equal. If the length of the prefix is known, the prefix can be extracted from A_{IN} . So, there is no need of explicitly storing the prefix. Storing the length of the prefix just needs $\log_2 W$ bits where W is the address width. So, by storing just the prefix length, memory width can be minimized. The maximum number of bounds in a node is $k = (BW - 32)/N$ where $BW = \text{memorybandwidth}$ and $N = 8, 16, 32, \dots \text{Addresswidth}$. So the maximum number of children in a node supporting IPv4 addresses for $BW=256$ will be $((256-32)/8)+1=29$. Similarly node supporting IPv6 addresses will have $((256-128)/8)+1=17$ children. The memory required to store prefix lengths is $29 * \log_2 32$ and $17 * \log_2 128$ for IPv4 and IPv6 addresses respectively.
- *Action pointer:* This depends on the maximum number of branches from a node i.e the maximum number of children of a node. For example, according to the Range Trie method, a node in the Range Trie built on IPv4 addresses can have a maximum of 29 branches. Since an action corresponds to each branch from the node, the action pointer should address all these 29 locations. So, 5 bits are needed to store action pointer at a node for IPv4 routing tables. To address the action table memory the action pointer is added to the base address which is same as the pointer stored to address the child nodes in the next level of the Range Trie. So the memory required to store the action pointers is $29 * \log_2 29$ and $17 * \log_2 17$ for IPv4 and IPv6 addresses respectively.

Counters for the bounds are not needed as the bounds are not deleted from the fixed part of the Range Trie. The resulting data structure of the node is as shown in Figure 4.8.

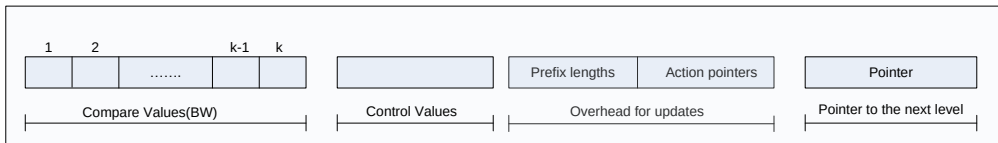


Figure 4.8: Modified node data structure in fixed part of the Range Trie

4.3.3 Updating Prefix lengths

In section 4.3.1, traversal of a prefix in a pipeline inserted or deleted for an update was described. When a prefix visits a node in the Range Trie, only the prefix lengths of the children whose address ranges are subset of the address range of the prefix are

along with each of the low bound and the high bound. At each level of the Range Trie along with the node information prefix lengths of all the children of the node are read from memory. After the Processing stage, based on *lb_up* and *hb_up*, the child nodes that are eligible for update are determined. Then the prefix lengths of the eligible children whose existing prefix length is less than the prefix length of the new prefix then it is replaced by the new prefix length. Otherwise the existing prefix lengths that are greater than the new prefix length are not changed. The resulting array of prefix lengths is written back to memory to the same memory location.

When a prefix is deleted, along with the length of the deleting prefix, a new prefix length to be replaced at appropriate nodes is also provided. Similar to prefix insertion, the deleted prefix traverses down the Range Trie updating the eligible nodes in the address range of the prefix. If the existing prefix length at an eligible node is equal to the length of the deleted prefix, then it is replaced with the provided prefix length. Otherwise, the existing prefix length is not changed. As the address bounds in the fixed part of the Range Trie are not changed, address bounds are not completely deleted from the Range Trie.

4.3.4 Updating Action pointers

When a prefix is inserted or deleted from the Range Trie, all the action entries in the action table corresponding to the nodes that are in the range of the prefix should be updated. During an update, it is evident from previous section that multiple nodes must be updated, as the prefix propagates through the pipeline. If n nodes are in the range of the prefix, n writes must be performed to update entries of n nodes in the action table. During this period, the whole pipeline should be halted as it cannot process any other requests while updating the action table. In order to avoid the above and minimize the number of writes, as explained in section 3.4.1, action pointers are placed along with the prefix lengths at each node and an action table corresponding to the nodes at each level is introduced. This section explains an efficient way to update the action pointers and the corresponding actions in the action table without halting the pipeline i.e. in the worst case *only one write is performed per prefix bound at each level of the Range Trie*.

A prefix in its traversal through the Range Trie visits a node at each level and updates the contents of the children of the node whose address ranges are a subset of the address range of the prefix. Thus at each level, action table entries of each of the children should be updated which again leads to multiple write operations. In order to avoid halting the pipeline, only one write must be performed to the action table at each level. So, action pointers of the children are also placed at the parent node along with the prefix lengths. An action pointer is a location in the action table where the action corresponding to each child of a node is stored. Now the key is to find an action table entry that is eligible for an update and does not invalidate the action table entries of other nodes that are not in the range of the prefix. In the fixed part of the Range Trie, the approach to update the action pointer is same for both an insert or a delete operation.

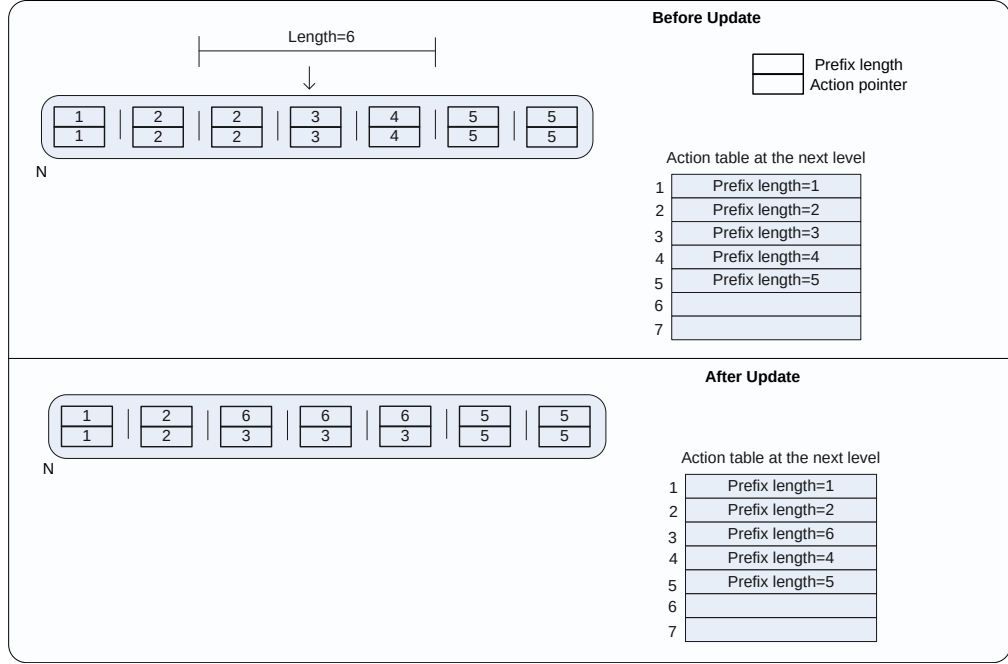


Figure 4.10: Updating action pointers in the Range Trie Structure

Consider a node in a Range Trie with some existing action pointers as shown in Figure 4.10. A new prefix p with prefix length 6 visits the node. The third, fourth and fifth child of the node are in the address range of the prefix. Since the prefix length of the new prefix is greater than the existing prefix lengths of these children, they are eligible for an update. As these child nodes currently hold the locations 2, 3 and 4, we might think that one of these locations might be free to be updated. But updating the action at location 2 invalidates the action corresponding to the second child. So, we cannot write the new action at location 2. Thus the remaining locations that can be chosen to write the new action without invalidating the existing action pointers are 3, 4, and 7. Choosing the first free and eligible location, the new action can be written at location 3 and the action pointers of third, fourth and fifth child are updated to 3. As explained in the previous section, prefix lengths of these child nodes are updated to the new prefix length i.e. 6.

Finding action pointer : To update the action pointers at a node, a location in the action table of its children should be found that is eligible to write the action without invalidating the actions of the other child nodes. Consider IPv4 addressing scheme and a bandwidth of 256 bits. With these specifications a node can have a maximum of 29 children according to the Range Trie method. As there may be upto 29 children, each action pointer consumes 5 bits. So, first each action pointer that is not eligible for an update is decoded using a 5-32 bit decoder. The bit corresponding to the value of the action pointer is set to '1'. All the bits of the child nodes that are in the range of the prefix are set to '0'. Now a bitwise-or is performed on these 29 32-bit values. In the

resulting 32-bit variable, all the bits corresponding to the values that are not eligible as an action pointer are set to '1'. So, an encoder is designed to find the first 0-bit in the 32 bit variable. The location of the first 0-bit is a free location and this can be chosen as an action pointer to be updated. The action pointer is added to the base address which is same as the pointer to the child nodes in the next level of the Range Trie to get the location to be updated in the action table. The new action is written to this location. A detailed explanation is shown Figure 4.11. The new action pointer is updated in the

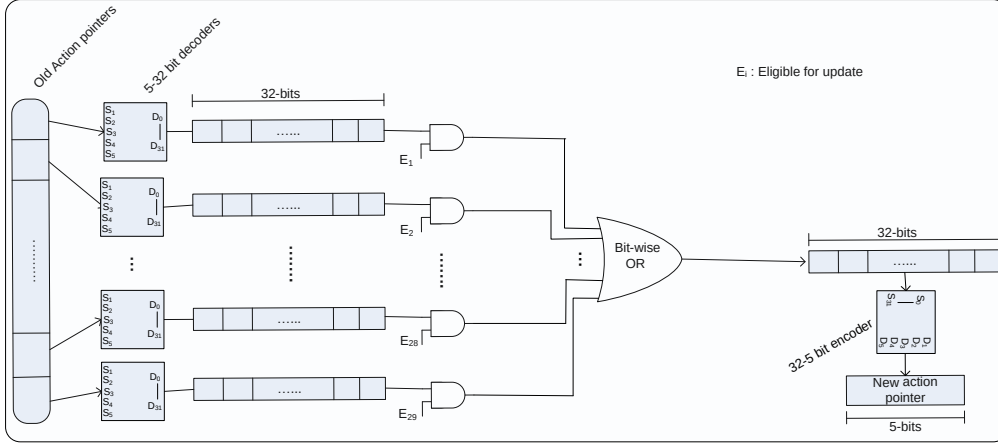


Figure 4.11: Finding a new action pointer

same way as the prefix lengths are updated and is written back to memory along with the prefix lengths. Since the action table resides along with the node information of the child nodes, while *write_back* of action pointers is performed, the new action is written to the action table at that location.

4.4 Spare levels

When a prefix is inserted, if one or both the bounds do not exist in the Range Trie structure, they should be added to the Range Trie data structure. The existence of a bound in the Range Trie is known only after it traverses through the Range Trie to the leaf nodes. Here, if the last level of nodes could not accommodate the new bound the node has to be split and the address range of the parent node should be updated. In the worst case, a bound has to traverse back to the root node of the Range Trie, splitting the nodes and updating the address range of the parent at each level. This involves a number of read and write operations to the memory. The whole pipeline should halt in order to process each update operation as the bound is traversing back in the pipeline, updating the corresponding nodes in its path. In section 3.5, we have discussed the need to introduce spare levels in the Range Trie structure to induct the non-existing bounds inserted during an update and avoid the problem of splitting the nodes. We have also discussed how to handle the insertions and deletions of prefixes into the spare levels.

The processing unit in the spare levels, apart from performing lookups, it should also handle positioning the newly inserted bound into an appropriate position in a node in the spare levels. The structure of the memory is also different from the fixed Range Trie. Since the memory allocated for the spare levels is limited, we decided to have dynamic memory allocation. So the memory in the spare levels is allotted depending on the need for a new node. This section describes the hardware implementation of the Processing unit, the memory structure and memory handling in the spare levels.

4.4.1 Memory structure

As mentioned in the previous chapter, we chose to have 3 spare levels in the Range Trie structure. This section presents the memory structure in these spare levels and the addressing scheme corresponding to this memory.

4.4.1.1 Node data structure

One of the primary objectives of the Range Trie method is to accommodate as many bounds as possible into a node following the Range Trie rules. This improves the branching factor and thus reduces the depth of the Range Trie. But in spare structure, we consider full length comparisons at each spare level of the Range Trie. So, for a given memory bandwidth the number of bounds that can be stored at a node is fixed ($k = BW/W$). Thus the Range Trie rules need not be applied on the nodes in the spare levels.

The contents of each node in a spare level are as follows:

- *Comparison values* : As we consider full length comparisons, the number of bounds in a node is $k=BW/W$. If we consider $BW=256$, then the number of bounds in a node are 8 for IPv4 addresses.
- *Control values* : Comparator modes are always the same since we perform full length comparisons. As the Range Trie rules are not applied on these nodes, the control values are also not needed as opposed to the nodes in the static Range Trie structure.
- *Pointer to the child* : As storing the pointer to each of the children amounts to large memory, we consider to store the children of a node in continuous memory location. So only one pointer to the left most child is stored similar to the nodes in the static Range Trie.
- *Prefix lengths* : This depends on the address width W . So the width of each prefix length is $\log_2 W$ bits. Since we have 8 children for each node memory needed to store the prefix lengths is $8 \times \log_2 W$.
- *Action pointers* : The number of children for a node is fixed. For IPv4 addresses and $BW=256$ the number of children for each node is 8. Thus to address the action table corresponding to the children of a node, just 3 bits are needed per child to

store the action pointer. For IPv6 addresses and $BW=256$ the number of children would be 3 and the required width of the action pointer is just 2 bits per child.

- *Counters* : As the number of prefixes starting or ending on a node is always equal to W , the width of the counter will be $\log_2 W$ bits. The memory needed to store the counters would be $8 \times \log_2 W$.

Figure 4.12 shows the block diagram of the node structure in spare levels.

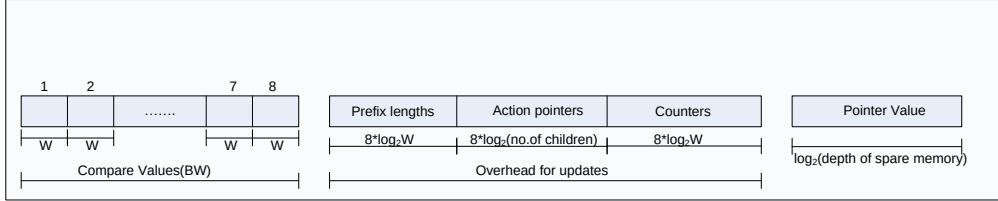


Figure 4.12: Node data structure in spare levels

4.4.1.2 Memory allocation

Each node in the spare level can accommodate $k=BW/W$ bounds. When a node is full, a lookup is performed on the newly inserted bound and is propagated to the appropriate child node. But if a child does not exist, a new child is created and the new bound is inducted into the newly created node. Creating a node is analogous to allocating a memory location corresponding to the node. As already mentioned in the node data structure, each node has a pointer to the leftmost child of the node. So, when a node is full, the first insert request propagated to a child also allocates memory for all the children of the node.

Consider a small sub-tree of the spare levels that just has a root node as shown in Figure 4.13. Consider that the node is full and there is a request for insertion of a new bound with value 5. Since there is no child node already allocated, a block of 8 locations is allotted from memory and the pointer to the leftmost child is updated in the pointer memory of the node.

Nodes in the first spare level are child nodes for the nodes in the last level of the fixed part of the Range Trie. Since the nodes in the Range Trie store variable width addresses, the nodes may have variable number of child nodes. Also for $BW=256$, the maximum of children per node is 29 and 17 for IPv4 and IPv6 addresses respectively. But as described above, the memory for the nodes in the spare levels are allocated in blocks of 8 memory locations. So, we need to store 4 pointers to the child nodes at each leaf node of the fixed Range Trie with IPv4 addresses.

The spare structure in the Range Trie contains three spare levels. The memory for all the spare levels is divided into blocks of 1K locations and dynamically distributed to the 3 spare levels on demand. For the first request for memory at each level, a block of 1K

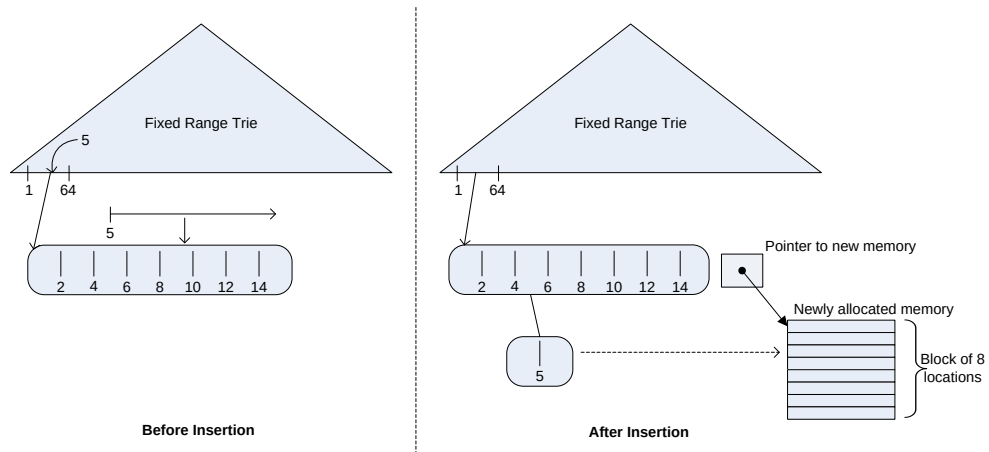


Figure 4.13: Memory allocation in spare levels

locations is allocated to that level. Then at each level when there is an another memory request, memory locations from the already allocated 1K locations for that spare level are provided. Then whenever the block is full a new block of 1K locations from the pool of blocks is allocated to that level.

4.4.1.3 Memory de-allocation

When a prefix is deleted, bounds in the nodes might get deleted. In case there are no bounds in the block of 8 locations allotted to the children of a node, the block is de-allocated and is added to the ‘*Recycle Bin*’ at each spare level. A Recycle Bin is a queue of pointers to the free memory blocks and is handled on the basis of first in-first out principle. Figure 4.14 shows the scenario of de-allocation along with a Recycle Bin at the spare levels. Consider a situation when a node is the only node in the block of 8 memory locations. If this node is deleted the block would be unused and hence moved to the Recycle Bin. So, node A is deleted which frees a block of 8 locations, and the block is added to the Recycle Bin.

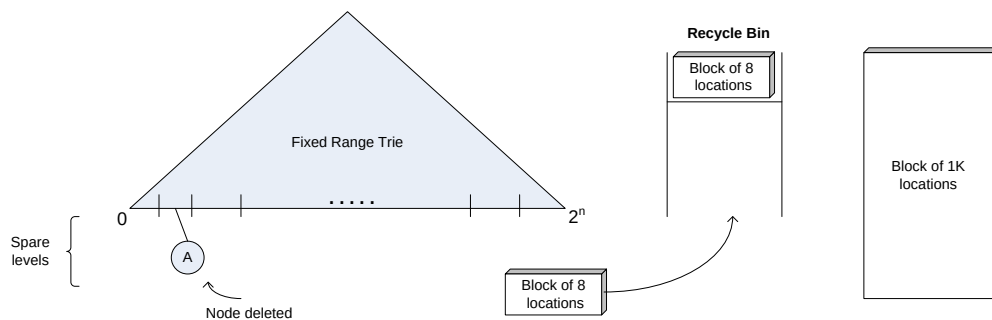


Figure 4.14: Memory de-allocation in spare levels

4.4.1.4 Memory addressing

The spare structure has some spare memory along with it. Memory is allocated to the nodes and in turn to each spare level upon request from this spare memory. As all the children of a node are placed in contiguous memory locations, the memory addressing scheme followed in the fixed Range Trie also applies here in the spare levels. As described in 4.4.1.2, nodes in the first spare level are the child nodes of the nodes in the last level of the fixed Range Trie. Since the memory in the spare level is allocated in blocks of 8, $\lceil 29/8 \rceil = 4$ (assuming BW=256, max. no. of children=29 for IPv4 addresses) pointers are needed at the node in the last level of the fixed Range Trie. So, depending on the lookup result at the last level, the memory address to be added to the lookup result is chosen from among the four memory pointers.

4.4.1.5 Spare level Memory Manager

From the memory allocation and de-allocation schemes discussed above, we need a module that manages the memory in spare levels. The memory available for all the spare levels is divided into blocks of 1K locations which we name as *Mem.blocks*. The pool of these *Mem.blocks* should be globally managed for all the spare levels. Each location can accommodate one node of the spare level. We need to keep track of the blocks of 1K locations allocated to each level and the Recycle Bin at each level. Each spare level requests a block of 8 locations when needed and frees a block of 8 locations when no bounds exist in the respective nodes.

The Memory manager functions as follows:

- When a spare level requests for memory
 - If the Recycle Bin is not empty, then a block from the Recycle Bin is allocated.
 - If the Recycle Bin is empty, then a block from the *Mem.block* in that level is allocated.
 - If the Recycle Bin is empty and *Mem.block* of that level is completely allocated, then a *Mem.block* from the pool of memory for spare levels is allocated to the spare level. Then a block of 8 locations is allocated for the request.
- When a spare level frees memory
 - a block is added to the Recycle Bin at that level (Each entry in the Recycle Bin is a block of 8 locations).

Since the Range Trie structure is pipelined, the manager should handle the memory requests from three spare levels in parallel. The structure of the spare level memory in order to handle the requests from the three spare levels in parallel is as shown in Figure 4.15. Consider memory of 32K locations for all the spare levels. So dividing it into blocks of 1K locations, we have 32 *Mem.blocks*. We need 15 bits to address this memory of which most significant 5 bits are used for addressing the *Mem.blocks* and the least significant 10 bits to address each location in a *Mem.block*.

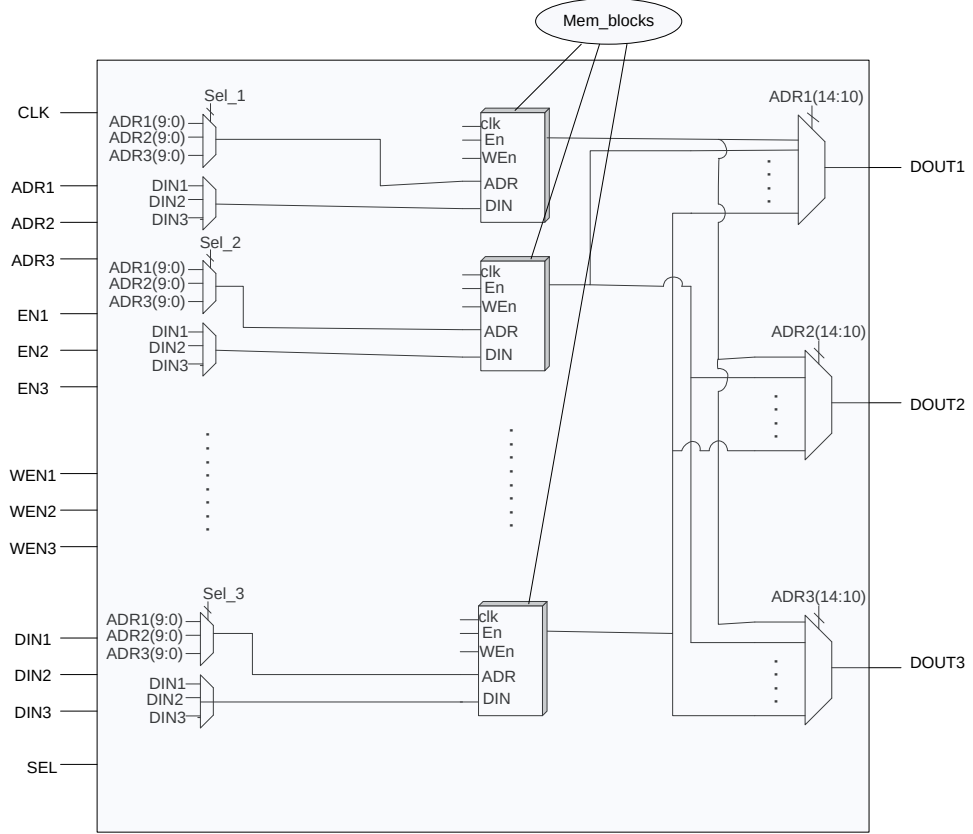


Figure 4.15: Block Diagram of Spare memory

4.4.2 Processing unit

Depending on the node information read from memory, the Processing Unit performs lookup of the incoming address A_{IN} and determines the branch to be taken to reach a node in the next level of the Range Trie. The Processing Unit has an extra role to play in the spare levels. Apart from performing lookup, the Processing Unit has to update the node comparison values when a non-existing bound is inserted into or a bound is deleted from the Range Trie structure. This section describes the hardware implementation of the Processing Unit in the spare levels.

When a new non-existing bound is inserted into a node, first a lookup of the bound is performed on the existing comparison values at the node. The result of the lookup is the appropriate position to insert the new bound. If the new bound is greater than all the existing bounds at the node, then the new bound is just appended to the existing comparison values. If the lookup of the new bound results in a position between the existing bounds then the bounds that are positioned to the right of the new bound should be shifted. In a similar way, if a bound is deleted from a node, all the bounds to the right of the deleted bound are to be shifted to left. Consider comparison values of a node

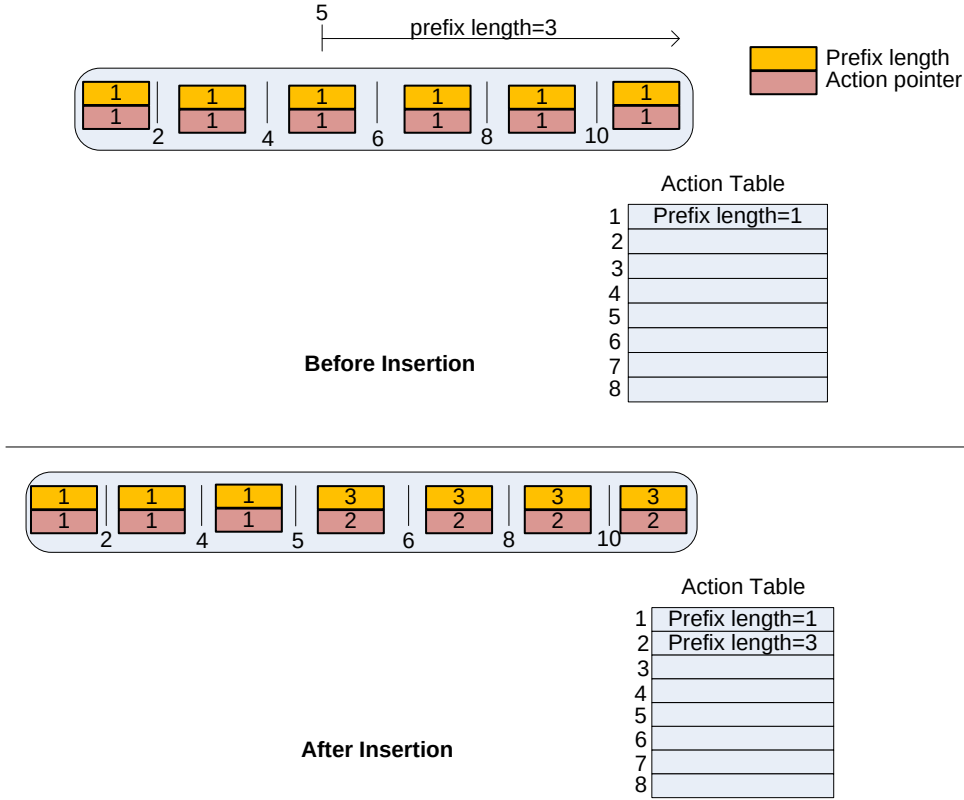


Figure 4.16: Inserting a new bound into a node in the spare levels

as shown in Figure 4.16. A bound with a value of 5 is inserted to the node. The lookup shows that the suitable position to insert the new bound 5 is position 2. So, the values 6 and 8 are shifted to the right, 2 and 4 retain their places and 5 is inserted at position 2.

Based on the kind of update request and the lookup result of the bound, the existing comparison values must be shifted to the right or the left or else retain the previous values. So, a 4 input mux is set for each position of the comparison values in order to update the values. Figure 4.17 shows a block diagram of a part of the Processing Unit that updates the node comparison values.

4.5 Complete Range Trie pipeline

In section 4.2 the pipeline structure of the Range Trie structure that performs lookup to match an incoming address A_{IN} to an address range in the interval $[0, 2^n)$ was discussed. In sections 4.3 and 4.4, the necessary design changes and the respective hardware units required to support longest prefix match and dynamic updates were presented. This section integrates all modifications into the pipeline structure. Each level in the Range Trie consists of two levels in the pipeline, a memory access and the processing unit.

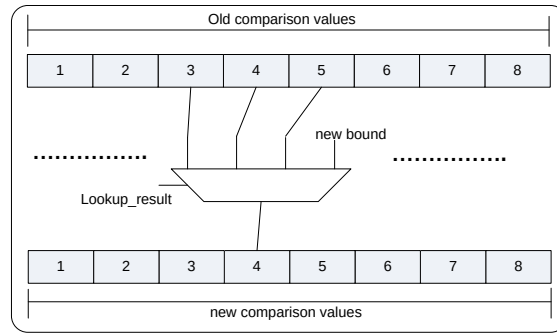


Figure 4.17: Logic in Processing Unit to insert or delete bounds into the node

First, a brief overview of hardware structure of one tree level in the fixed part of Range Trie and the one level of sub-tree in spare levels is presented.

4.5.1 Iteration stage in the fixed Range Trie

The iteration stage in the fixed part of the Range Trie should be able to perform a lookup and update the prefix lengths, action pointers and the corresponding actions in the action table, when there is an insert or a delete request. The block diagram of the iteration stage supporting longest prefix match and dynamic updates is as shown in Figure 4.18.

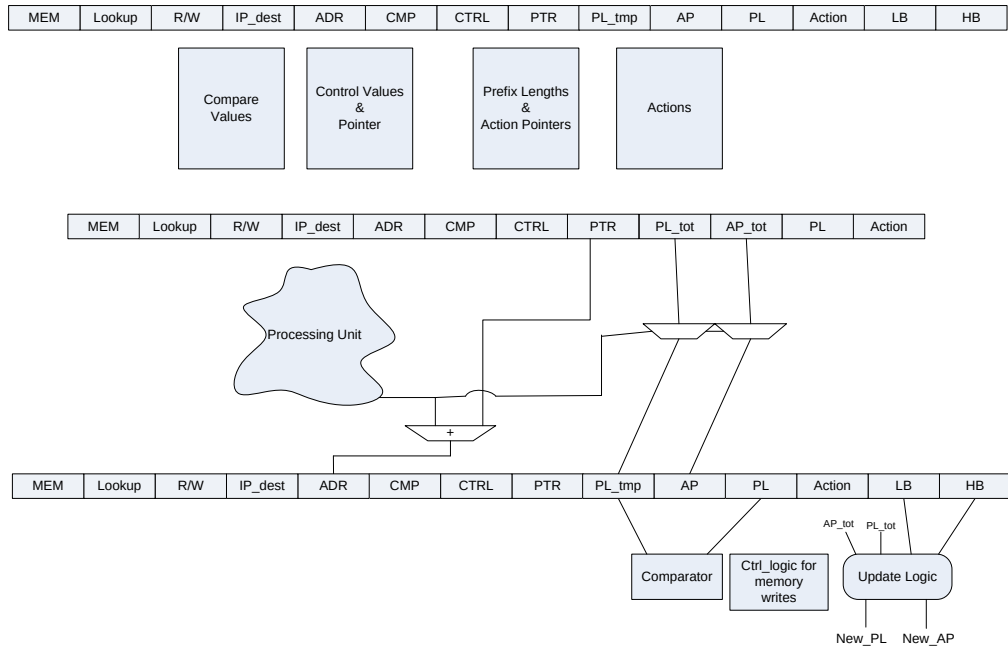


Figure 4.18: Block Diagram of Iteration stage in the static Range Trie

During Lookup first the node information that comprises of comparison values, control values, pointer to the leftmost child, prefix lengths and action pointers are read from the memory. Then the Processing Unit performs comparisons on the node information and determines the branch to be taken to reach a child in the next level of the Range Trie. The output of the Processing Unit is also used to select the prefix length of the respective child. Finally, the selected prefix length is compared with the prefix length remembered by the incoming address A_{IN} and the greater of the two is remembered, as A_{IN} moves the next stage of the pipeline.

During Update when a prefix visits a node in the Range Trie, first the low-bound is fed into the pipeline. In the next cycle the high-bound is fed into the pipeline.

- *no prefix split* : The lookup result of the low-bound is stored as lb . When the high-bound visits the Processing Unit, the lookup result of the high-bound is compared to the stored lb . If the comparison results in an equality, then there is no prefix split, i.e. both the bounds of the prefix branch to the same child of a node. Information in the node need not be updated and hence is unchanged.
- *prefix splits at this stage* : If the lookup result of the high-bound and the stored lb are unequal, the prefix has split at this node. All the children of the node that are in the overlapping region of lb_up and hb_up are updated as shown in node A of Figure 4.9.
- *prefix already split* : If the prefix has already split in a previous level of the Range Trie, then the low-bound and the high-bound are traversing independent to each other. So, the low-bound updates the prefix lengths and action pointers of all the children in the region lb_up and similarly the high-bound updates all the children in the region hb_up as shown in node A2 and node A4 of Figure 4.9.

4.5.2 Iteration stage in the spare levels

The memory structure in the spare levels is significantly changed to support dynamic memory allocation i.e. memory is allocated upon request and is deallocated when all the bounds in the node are deleted. Apart from performing lookup and updating prefix lengths, an Processing Unit in the spare levels should be able to update the node comparison values when a prefix with non-existing bound is inserted or a prefix is deleted and update the counter (*number of prefixes starting or ending on the bound*) of the bound. The block diagram of the pipeline structure of one level of sub-tree in the spare levels is shown in Figure 4.19. The approach to perform lookup for an address A_{IN} and updating prefix lengths and action pointers is similar to the pipeline stages described in 4.5.1. When a prefix is inserted, the existence of bounds of the prefix is unknown until the bounds completely traverse the Range Trie. A lookup performed on the non-existing bound falls into one of the specified address ranges in the interval $[0, 2^n)$.

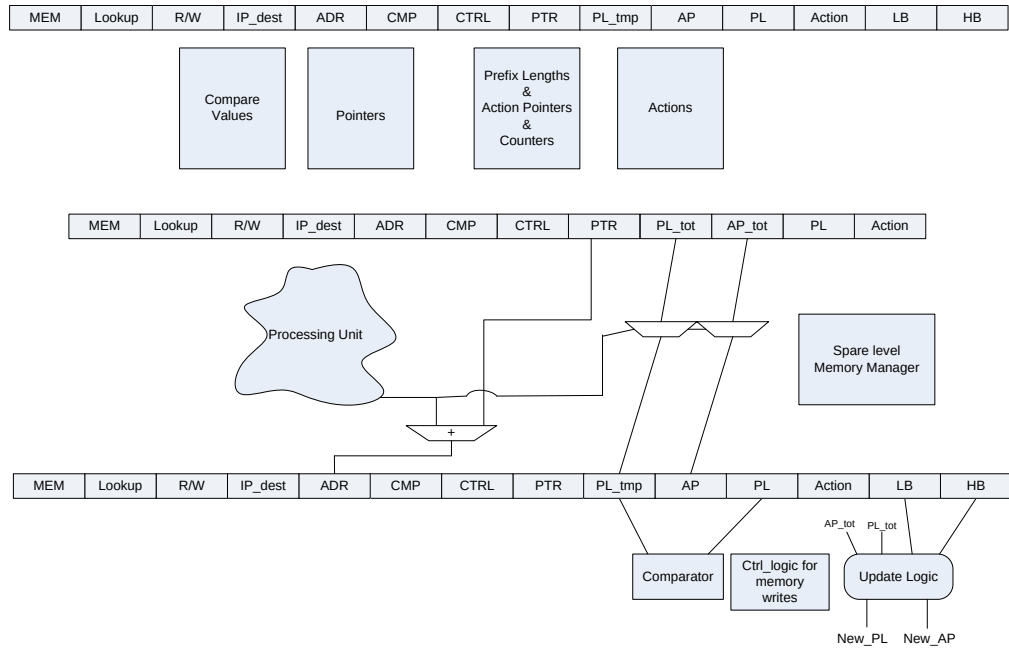


Figure 4.19: Block Diagram of Pipeline structure of one level of sub-tree in spare levels

Inserting a prefix

- If a sub-tree exists in the address range, then the bound traverses the sub-tree and is inducted into a node. The prefix lengths and action pointers of the nodes in the range of the prefix are updated accordingly. If the bound cannot be inserted the sub-tree is rebuilt along with the newly inserted bound. If the sub-tree is already full then the whole Range Trie structure is rebuilt.
- If a sub-tree does not exist in the address range, then memory is allocated from the available spare memory and the newly inserted bound is inserted into that node.
- If the spare memory is full and no more memory allocation is possible, then the whole Range Trie structure is rebuilt.

Deleting a prefix

- All the nodes in the range of the prefix are updated and the counter is decremented. If the counter becomes zero and no children exist in the address ranges of the bound, the bound is deleted from the Range Trie. If children exist even if the counter becomes zero, the bound will be deleted when the Range Trie is rebuilt.

4.6 FPGA Prototype

To ensure that the hardware design is functionally correct, the hardware implementation of the Range Trie structure described above is prototyped using a Xilinx ML410(Virtex4

XC4VFX60) platform. Due to limited resources on the FPGA, a small routing table is considered for the prototype.

The whole design is connected to a Microblaze processor through a Processor Local Bus (PLB) as shown in Figure 4.20 with the help of Xilinx EDK tool.

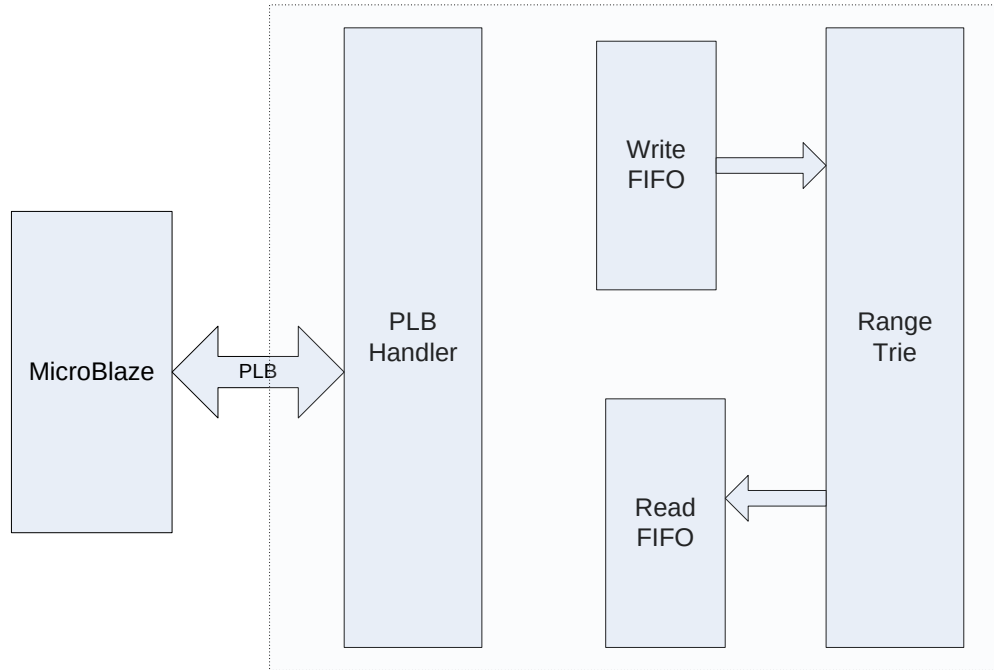


Figure 4.20: Block Diagram of the prototype

Read and Write FIFO's were used to interface with the software. The data to be stored in memory is loaded into the Write FIFO. Then the Range Trie IP reads the inputs from Write FIFO and the results are pushed into the Read FIFO. Each location in the FIFO is 32 bits. A top level VHDL module is designed to feed the input data to the Range Trie hardware design. Initially, multiple cycles are needed to load the node configurations into memory of the Range Trie. So for IPv4 addresses, we need 10 cycles to load the information related to a single node. During an update operation, each bound of a prefix requires 2 cycles to feed the bound and the new prefix lengths. The processor reads the results in the Read FIFO and prints them on screen using a RS232 communication link.

4.7 Summary

This chapter presented the hardware description of the modified Range Trie design. Each level of the Range Trie is divided into two stages, a memory access stage and a processing stage, and are pipelined to increase the throughput of the Range Trie. As it is pipelined, the lookups and updates to the Range Trie can co-exist at different stages in the pipeline.

The initial pipelined design by George Stefenakis [26] was presented in section 4.1. Then the design was modified to be able to perform memory read/write operations which is needed to perform updates. The memory overhead to support LPM and dynamic updates was calculated in section 4.3. The required memory blocks, the prefix lengths, action pointers and action table were then added to the pipeline. Based on the rules to update prefix lengths and action pointers in the previous chapters, logic was developed to update the prefix lengths and action pointers of the appropriate nodes. During an update, each prefix bound only performs single write per level, so effectively each update operation requires a bubble of only 4 cycles in the pipeline (2 read cycles to lookup of prefix bounds and 2 write cycles to update).

A specialized memory structure and processing unit are designed for the spare levels which was presented in Section 4.4. Memory is allocated to the spare levels dynamically upon request and is de-allocated when the allocated memory is no longer used. Memory is allocated in blocks of 8 locations in order to minimize the overhead of addressing the nodes(pointers to the child nodes) in the sub-trees.

Finally, we have a working hardware design that realizes the advantages of the modified Range Trie structure presented in the previous chapter.

Experimental Results

In this chapter, we present the results obtained from the new Range Trie design that supports longest prefix match and dynamic updates. We consider a memory bandwidth (BW) of 256 bits and IPv4 and IPv6 addresses for evaluating the Range Trie design. First we present the results for an input of pseudo random numbers generated using uniform and Gaussian distributions. Then the results obtained from the real routing data from the RIPE data base is presented. These results include the latency and memory requirement of the fixed part of the Range Trie design generated for the corresponding sizes of the routing tables. Heuristic methods were developed by Ruben de Smet[8] to construct the Range Trie. Two types of approaches namely *top-down* and *bottom-up* were developed. For this thesis, we chose *bottom-up* heuristics to construct the Range Trie as it proved to build an efficient and a balanced Range Trie as shown in [8]. Later we present the update statistics generated during the period of an year (June-2009 to May-2010).

The characteristics of the inputs and of the Range Trie design for acquiring the results are as follows:

- *Address width* : IPv4(32 bits) and IPv6(128 bits).
- *Memory Bandwidth(BW)* : 256 bits
- *Heuristics used* : Bottom-up(BU)
- *Distribution for random data* : Uniform Distribution.

5.1 Random Input

Uniform Distribution : In uniform distribution, it is equally likely to generate a number in an address space. A number(len) in the interval $(0, N]$ which is the prefix length and number($addr$) in the interval $[0, 2^N)$ where $N = addresswidth$, are generated using uniform distribution. So the prefix would be $addr$ with a prefix length of len . IPv4 routing tables with varying sizes ranging from 1K prefixes to 8M prefixes are generated as shown in Table 5.1 and IPv6 routing tables from 1K prefixes to 1M prefixes.

Range Trie is constructed with varying routing table sizes and the number of levels of the Range Trie for IPv4 and IPv6 addresses is shown in Figure 5.1. Given a memory bandwidth of 256 bits, the number of bounds stored at a node with IPv4 addresses ranges from 8-28, bounds in a node with IPv6 addresses ranges from 2-16. Since the bounds are uniformly distributed in the address space, Range Trie does not benefit much from the Range Trie rules resulting in full length comparisons in most of the nodes.

<i>Routing Table</i>	<i># of Prefixes</i>	<i># of Bounds</i>
1	1024	1810
2	2048	3623
3	4096	7143
4	8192	14219
4	16384	28420
5	32768	56546
6	65536	111948
7	131072	221409
8	262144	438389
9	524288	865682
10	1048576	1706460
11	2M	3355233
12	4M	6572998
13	8M	12280126

Table 5.1: Uniform Distribution:IPv4 Routing Tables

As we move to the top levels Range Trie makes the best use of the rules increasing the branching factor and in turn reduces the number of levels of the tree. Increase in

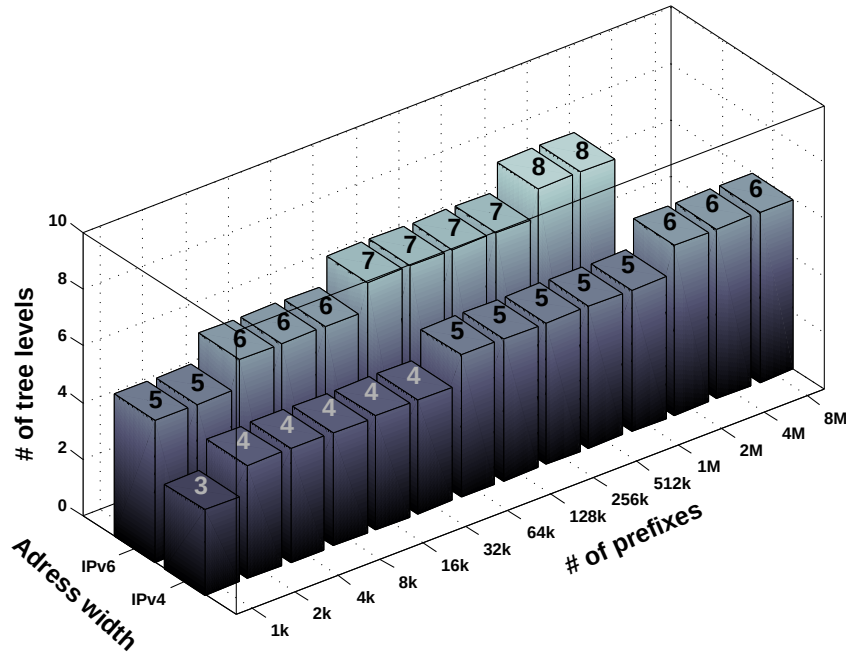


Figure 5.1: Number of tree levels with varying routing table sizes

address width from 32 bits (IPv4) to 128 bits (IPv6) just increases the tree height by 2 levels which shows the scalability of the Range Trie even for uniformly distributed

bounds. The memory requirement of the fixed part of the Range Trie supporting LPM and dynamic updates, with different routing table sizes is given in Table 5.2 and the corresponding plots comparing Range Trie memory with different address widths is as shown in Figure 5.2.

	<i>Memory in MB</i>													
<i>Address</i>	<i>1k</i>	<i>2k</i>	<i>4k</i>	<i>8k</i>	<i>16k</i>	<i>32k</i>	<i>64k</i>	<i>128k</i>	<i>256k</i>	<i>512k</i>	<i>1M</i>	<i>2M</i>	<i>4M</i>	<i>8M</i>
IPv4	0.02	0.04	0.07	0.14	0.28	0.52	1.00	1.80	3.33	5.93	11.09	21.24	40.91	76.66
IPv6	0.08	0.15	0.31	0.61	1.23	2.44	4.85	9.71	19.30	38.50	76.38	-	-	-

Table 5.2: Memory requirement of routing tables with uniformly distributed prefixes

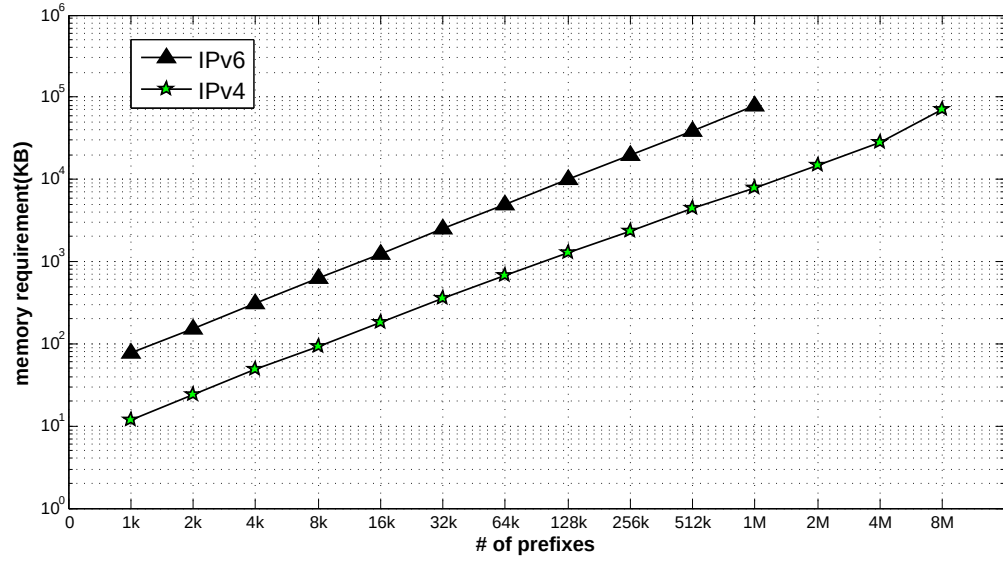


Figure 5.2: Memory with varying routing table sizes

5.2 Real data

The RIPE NCC database in Amsterdam provides raw data corresponding the routing tables collected from various Internet Exchanges. The Routing Information Service (RIS) is a RIPE NCC project to collect internet routing data by deploying Remote Route Collectors (RRC) at various Internet Exchanges. The provide raw data which is converted to a suitable input format in order to run the Range Trie heuristics. The prefix length distribution among the provided routing tables is shown in Figure 5.3¹. The figure

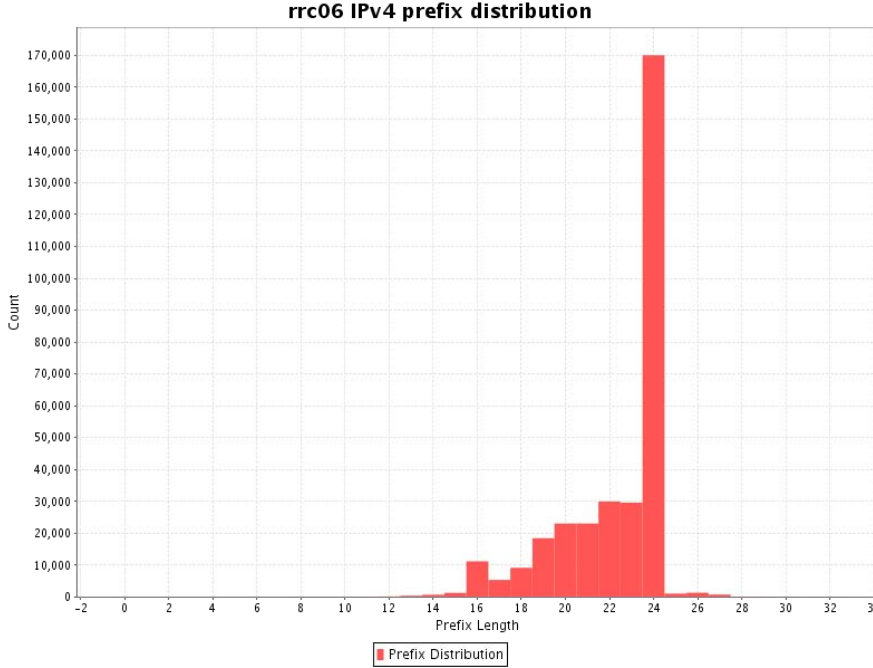


Figure 5.3: IPv4 prefix distribution in rrc06 Exchange point

resembles a Gaussian distribution rather than a uniform distribution. Hence, similar to Gaussian distribution we should be able to share bits and accommodate more bounds in a node. Thus we can say that the Range Trie gives increased performance compared to the Range Tree in real life.

5.2.1 Real IPv4 routing tables

IPv4 routing tables are extracted from raw data collected from 13 different Internet Exchange points. The number of prefixes and the corresponding unique bounds are tabulated in Table 5.3.

To determine the scalability of the Range Trie structure for large routing tables of IPv6 addresses, the real IPv4 addresses of the routing tables obtained from the 13 exchange points mentioned in Table 5.3 are projected to IPv6 addresses using the

¹http://www.ris.ripe.net/risreport/rrc06/total/prefix_n.shtml

<i>Exchange Point</i>	<i>Location</i>	<i># of Prefixes</i>	<i># of Bounds</i>
<i>amsix</i>	Amsterdam, the Netherlands	270771	324856
<i>decix</i>	Frankfurt, Germany	266881	320829
<i>jpix</i>	Otemachi, Japan	265291	329770
<i>linx</i>	London, UK	262310	314756
<i>mix</i>	Milan, Italy	263500	316248
<i>mskix</i>	Moscow, Russia	263869	316361
<i>netnod</i>	Stockholm, Sweden	269473	322180
<i>ny</i>	New York, USA	262942	315338
<i>paix</i>	Palo Alto, USA	265773	319424
<i>ptt</i>	São Paulo, Brazil	269106	323331
<i>ripe</i>	Amsterdam, the Netherlands	275706	330445
<i>sfinx</i>	Paris, France	267675	321554
<i>vix</i>	Vienna, Austria	264496	317036

Table 5.3: IPv4 Routing Tables

following format:

2002: IPv4 address :: zeroes

The Range Trie heuristics are run on these projected routing tables and the results are compared with the Range Trie for IPv4 addresses. The performance of the Range Trie on these real routing tables is shown in Figure 5.4. The total memory requirement for the Range Trie designs is tabulated in Table 5.4 and the corresponding plot is shown in Figure 5.5.

	<i>Memory in KB</i>												
<i>Address</i>	<i>amsix</i>	<i>decix</i>	<i>jpix</i>	<i>linx</i>	<i>mix</i>	<i>mskix</i>	<i>netnod</i>	<i>ny</i>	<i>paix</i>	<i>ptt</i>	<i>ripe</i>	<i>sfinx</i>	<i>vix</i>
IPv4	1400	1395	1370	1343	1364	1350	1374	1355	1385	1400	1440	1366	1353
IPv6	1988	1976	1941	1908	1932	1919	1949	1922	1961	1986	1912	1944	1923

Table 5.4: Memory real IPv4

The distribution of prefixes in the real routing tables resemble Gaussian distribution with the peak at prefixes with length 24. Because the bounds are closely placed in the address space, the Range Trie makes most of the benefit from the Range Trie rules in placing maximum bounds possible in a node. The difference in number of levels of the tree with 4 times increase in address width is just 1, which shows that the Range Trie scales much better for real routing tables.

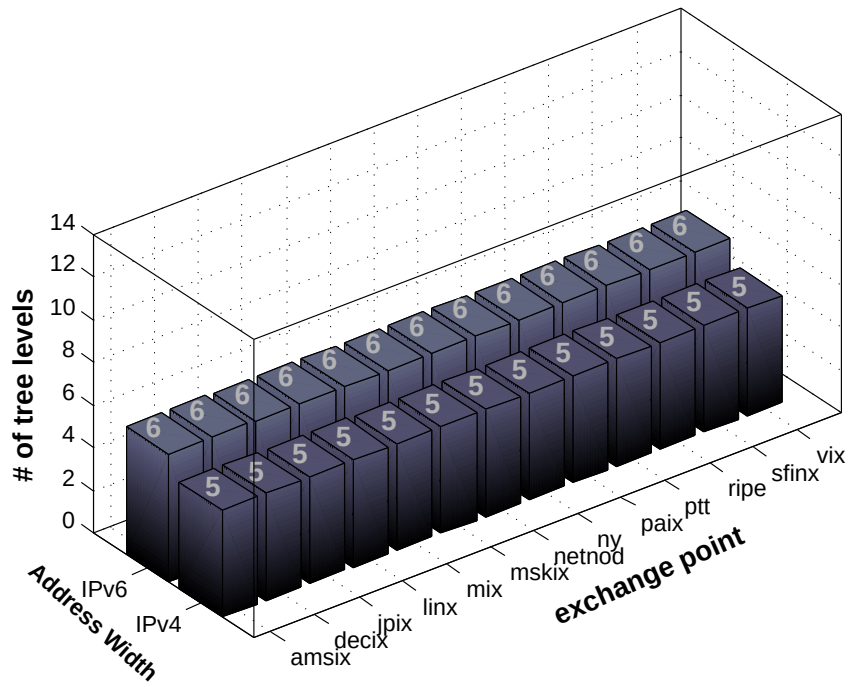


Figure 5.4: Number of tree levels for real routing tables

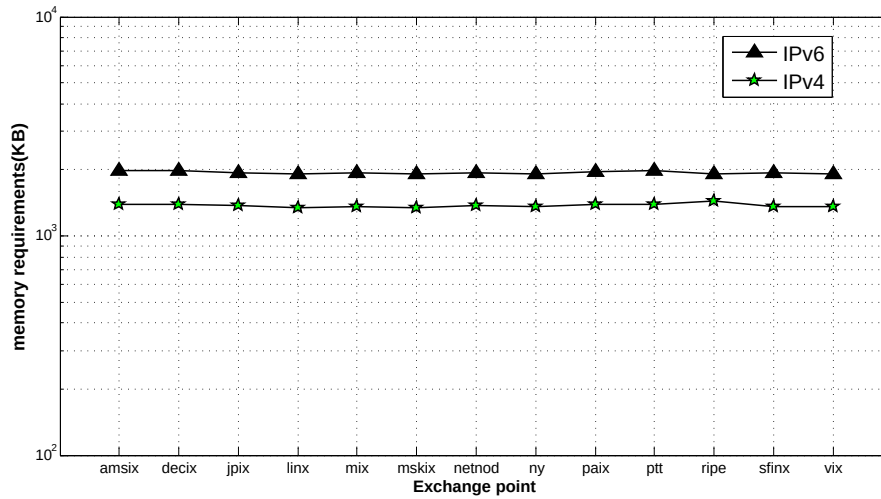


Figure 5.5: Memory requirements for real routing tables

5.2.2 Real IPv6 routing tables

There are just a few Internet Exchange points that are managing IPv6 addresses as it is not completely deployed on Internet. The range Trie heuristics are run on the latest IPv6 routing tables of the Exchange points mentioned in [23] whose prefixes and the corresponding unique bounds are tabulated in Table 5.5.

<i>label</i>	<i>Exchange Point</i>	<i># of Prefixes</i>	<i># of Bounds</i>
<i>as6447</i>	Route-Views.Oregon-ix.net	3189	5654
	Hurricane Electric	3110	5398
<i>as2</i>	APNIC R&D	3101	5967

Table 5.5: IPv6 Routing Tables

Table 5.6 provide the results corresponding to the total memory needed for the fixed part of the Range Trie structure. The plot providing the performance in terms of latency i.e. the number of levels of the Range Trie is shown in Figure 5.6 and plot for total memory is in Figure 5.7. The basic Range Tree needs 12 levels for the approximately 3K prefixes whereas the Range Trie needs just 5 levels. Comparing with the similar size of IPv4 routing tables using uniform distribution, the depth of the Range Tree has more than doubled while the depth of the Range Trie has just increased by 1 level which shows the scalability of the Range Trie structure.

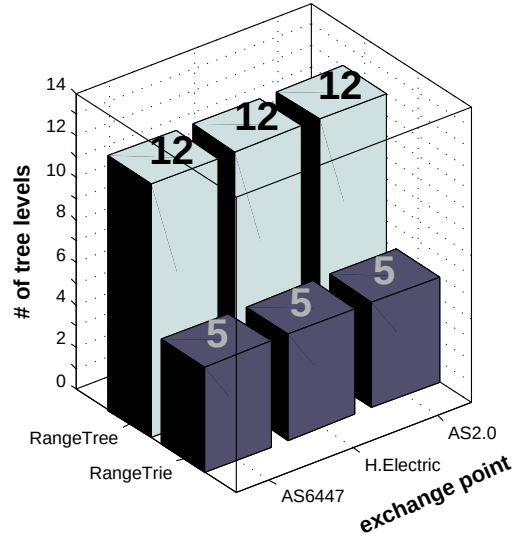


Figure 5.6: Number of tree levels of real routing tables of IPv6

Address	Type	Memory in KB		
		AS6447	H.Electric	AS2.0
IPv6	RTree	378	370	370
	RTrie	58	53	54

Table 5.6: Memory for real IPv6 routing tables

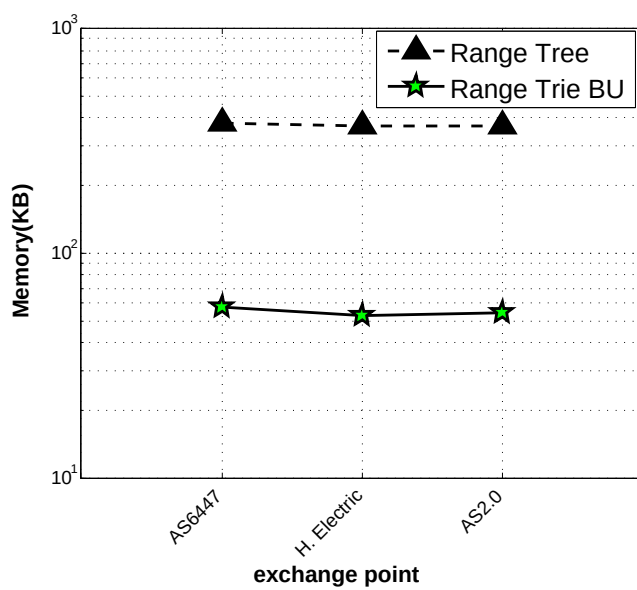


Figure 5.7: Memory requirement for real IPv6 routing tables

5.3 Comparison

In this section, the Range Trie method is compared with other lookup algorithms introduced in Chapter 2, and also to some other algorithms based on Trie and Range Tree methods. For each of the algorithms the latency, memory requirement are presented along with discussing the scalability.

The implementations corresponding to the algorithms were simulated according to the given specifications in the documentation. The memory requirements are measured using a C++ tool that monitors memory allocation mechanism of a process and provides the peak memory usage during its lifetime. All the implementations are run on synthetic routing tables of IPv4 and IPv6, real routing tables of IPv4 and real IPv4 routing tables projected to IPv6.

A Trie is a basic lookup structure, and performs worse in terms of performance and scalability. The depth of the trie grows linearly with the address width and need same number of memory access as the levels of the trie per lookup. Thus the performance and scalability of the Trie is worse.

The Multibit trie is an extension to the trie comparing multiple bits at each node. As discussed in chapter 2, with increase in the stride to reduce the depth of the trie the memory requirement increases exponentially. Here a stride of “8-8-4-4-2-2” is chosen with the trie depth of 7. Using controlled prefix expansion all the prefixes are preprocessed to match the stride widths. An Allotment Routing Table(ART)¹ is used for implementing multibit Tries. Even though the performance is better compared to the Trie, it has a high memory requirement.

Patricia Trie in [19] optimizes the original Trie structure using Path Compression technique by pruning the nodes that have a single child. The structure is a bit complex to update. Lookup performance is improved due to the decrease in Trie depth, but needs backtracking to acquire longest prefix match reducing the performance by half which makes the algorithm less interesting for address lookup. The results are obtained from the Net::PATRICIA module for Perl from CPAN site². The memory requirement is still high and scalability is poor because of the minimal improvements to the regular Trie.

LC Trie combines Path Compression used in PATRICIA Trie with Level Compression technique. A static version of the LC-Trie introduced in [20]³ is used for this purpose. The depth of the LC-Trie reduced significantly compared to the previous algorithms but with the expense of memory. Scalability is also better compared to the regular Trie.

¹<http://www.hariguchi.org/art/>

²<http://search.cpan.org/dist/Net-Patricia/>

³<http://www.nada.kth.se/~snilsson/public/code/router/C/>

Range Tree method is a search on values approach that supports multiple comparisons. The theoretical results of the Range Tree were presented in the previous section to compare with the Range Trie. Here results from a more practical approach by implementing the multiway range tree using B-Tree will be presented. A B-Tree has a minimization factor that limits the minimum number of bounds allowed in a node. If t is the minimization factor, every node should have at least $t-1$ bounds and at most $2t-1$ bounds. Relating these to the parameters of the Range Trie and equation can be derived with respect to memory bandwidth(BW) and the address width(N).

$$(2 * t) - 1 = BW/N$$

As we have a BW of 256 and IPv4 addresses, the value of t would be 4.5. The results are obtained from an existing B-Tree implementation¹. Scalability in performance can be achieved by performing more comparisons per node but with increase in memory requirements.

Tree Bitmap is Trie structure that stores the node information in the form of a bitmap. Here we consider a Tree bitmap which needs 6 levels with strides of “13-4-4-4-4-3”. The memory requirement is reported with a lower and upper bounds considering 11-23 bytes per prefix for IPv4 (32 bit) addresses as mentioned in [12].

5.3.1 For synthetic routing tables

The latency and memory requirements of these algorithms for synthetic routing tables of sizes varying from 1024 – 8 *million* for IPv4 addresses is tabulated in Table 5.7 and the corresponding plots are shown in Figure 5.8 and Figure 5.9. The corresponding plots for IPv6 addresses are shown in Figure 5.10 and Figure 5.11.

Algorithm	Range Trie depth # of Iterations													
	1k	2k	4k	8k	16k	32k	64k	128k	256k	512k	1M	2M	4M	8M
Trie	32	32	32	32	32	32	32	32	32	32	32	32	32	32
PTrie	23	23	23	23	23	23	23	23	23	23	23	23	23	23
MbTrie	7	7	7	7	7	7	7	7	7	7	7	7	7	7
LCTrie	3	3	3	3	4	4	5	5	6	6	7	7	9	9
TBitmap	6	6	6	6	6	6	6	6	6	6	6	6	6	6
RTree	5	5	6	6	7	7	8	8	9	9	10	10	11	11
RTrie	4	4	4	4	4	5	5	5	5	5	6	6	6	6
Algorithm	Memory Usage (KB)													
	1k	2k	4k	8k	16k	32k	64k	128k	256k	512k	1M	2M	4M	8M
Linear	1058	1043	1036	1025	1029	1031	1053	1027	1038	1051	1077	1046	1033	1033
Trie	10520	10520	10520	10520	10520	10520	10520	10520	10520	10520	10520	10520	10520	10520
PTrie	1828	1828	1960	2088	2472	3240	4648	7592	13608	25384	49320	97320	193960	389416
MbTrie	2224	2484	2612	3128	3888	5684	9396	16820	31284	58548	110004	209972	400692	801300
LCTrie	185808	185972	186148	186512	187156	188776	191832	197972	210620	236156	287460	390740	599716	650000
RTree	19	39	78	158	317	638	1282	2590	5238	10588	21464	43596	88801	181501
RTrie	12	24	48	92	182	357	678	1274	2343	4386	7828	14660	28496	70296

Table 5.7: Comparison of synthetic IPv4 routing tables

¹<http://resnet.uoregon.edu/~gurney-j/jmpc/btree.html>

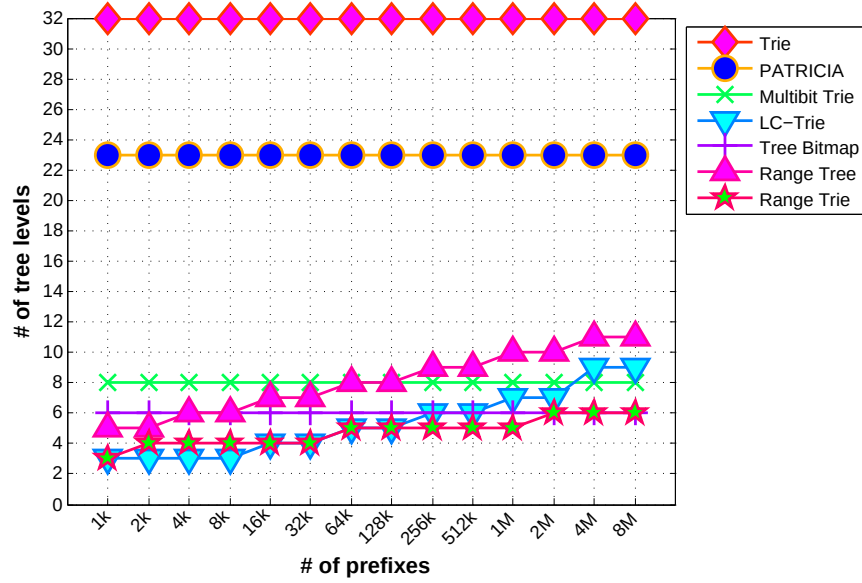


Figure 5.8: Number of tree levels(latency) for synthetic IPv4 routing tables

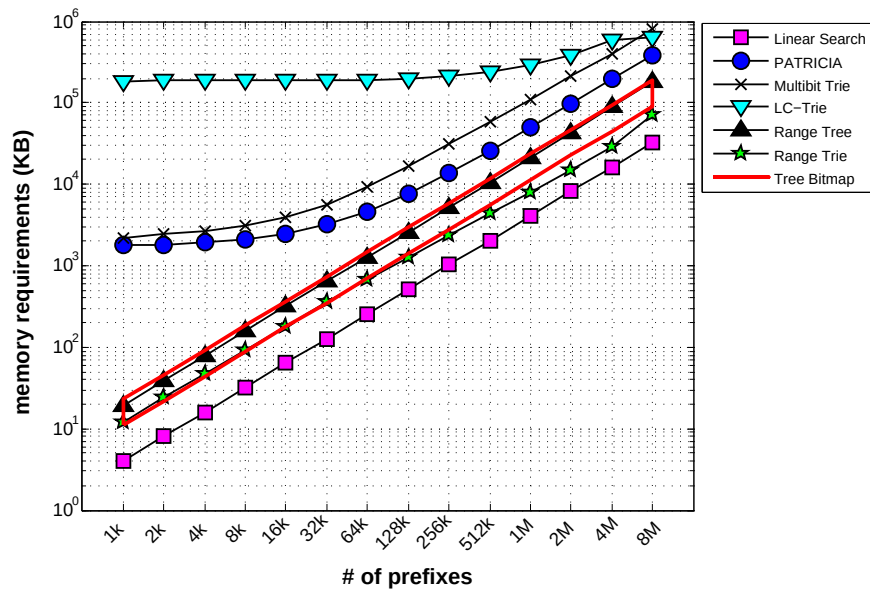


Figure 5.9: Memory requirement for synthetic IPv4 routing tables

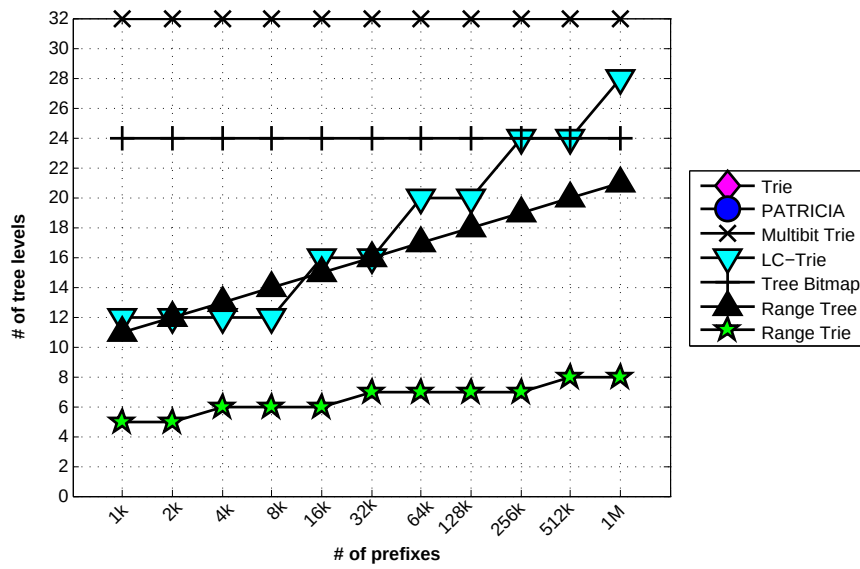


Figure 5.10: Number of tree levels(latency) for synthetic IPv6 routing tables

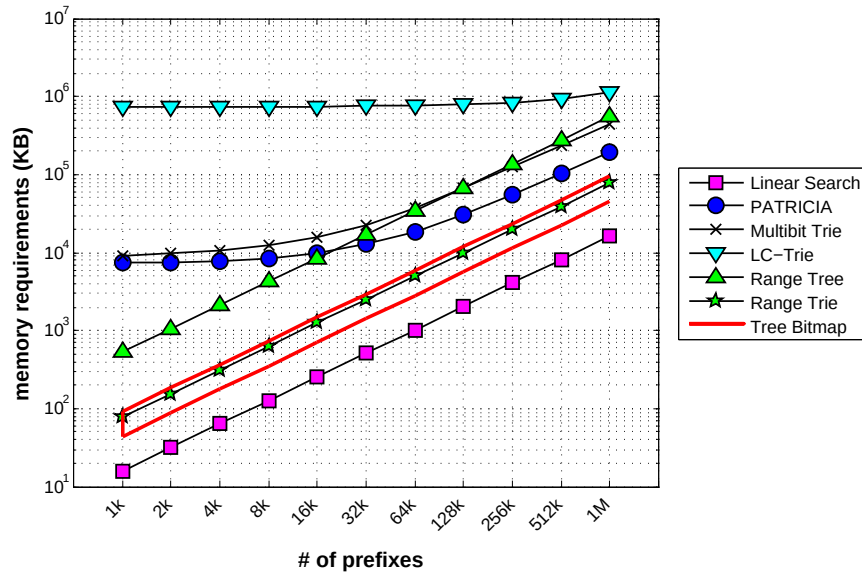


Figure 5.11: Memory requirement for synthetic IPv6 routing tables

5.3.2 For Real routing tables

Table Figure 5.8 gives the latency and memory requirements of the algorithms for real routing tables collected from the 13 exchange points mentioned in Section 5.2.1. The corresponding plots comparing the tabulated values are shown in Figure 5.12 and Figure 5.13.

Algorithm	Exchange Point, # of Iterations												
	amsix	decix	jpix	linx	mix	mskix	netnod	ny	paix	ptt	ripe	sfinx	vix
Trie	32	32	32	32	32	32	32	32	32	32	32	32	32
PTrie	23	23	23	23	23	23	23	23	23	23	23	23	23
MbTrie	7	7	7	7	7	7	7	7	7	7	7	7	7
LCTrie	6	6	6	6	6	6	6	6	6	6	6	6	6
TBitmap	6	6	6	6	6	6	6	6	6	6	6	6	6
RTree	9	9	9	9	9	9	9	9	9	9	9	9	9
RTrie	5	5	5	5	5	5	5	5	5	5	5	5	5
Algorithm	Memory Usage (KB)												
	amsix	decix	jpix	linx	mix	mskix	netnod	ny	paix	ptt	ripe	sfinx	vix
Linear	1058	1043	1036	1025	1029	1031	1053	1027	1038	1051	1077	1046	1033
Trie	10520	10520	10520	10520	10520	10520	10520	10520	10520	10520	10520	10520	10520
PTrie	25552	25168	25040	24784	24912	24912	25552	24912	25168	25424	25936	25396	25040
MbTrie	18172	17912	17792	17536	17660	17660	17928	17668	17784	18052	18560	17788	17668
LCTrie	39388	39104	38912	38664	38836	38848	39352	38684	38932	39320	39844	39140	38880
RTree	8088	7956	8088	7824	7824	7824	7956	7824	7956	7956	8220	7956	7956
RTrie	1400	1395	1370	1343	1364	1350	1374	1355	1385	1400	1440	1366	1353

Table 5.8: Comparison for real IPv4 routing tables

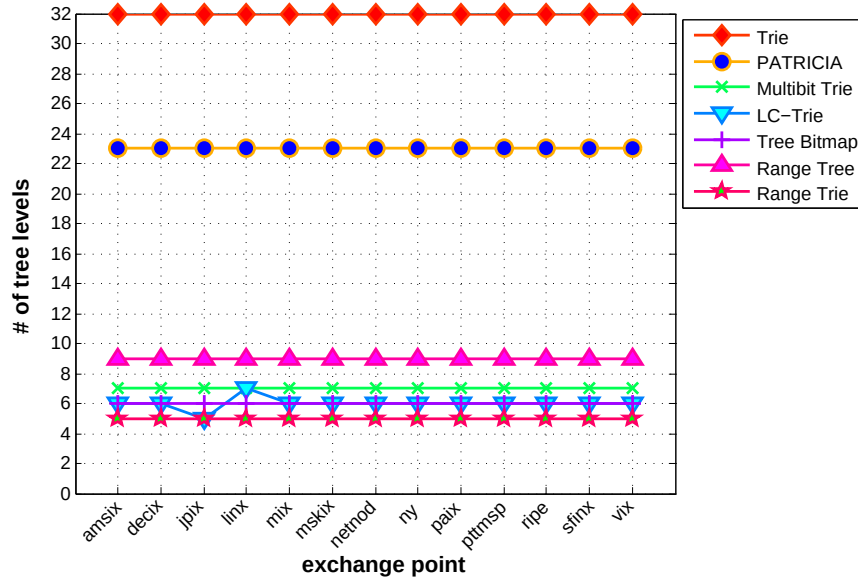


Figure 5.12: Number of tree levels(latency) for real IPv4 routing tables

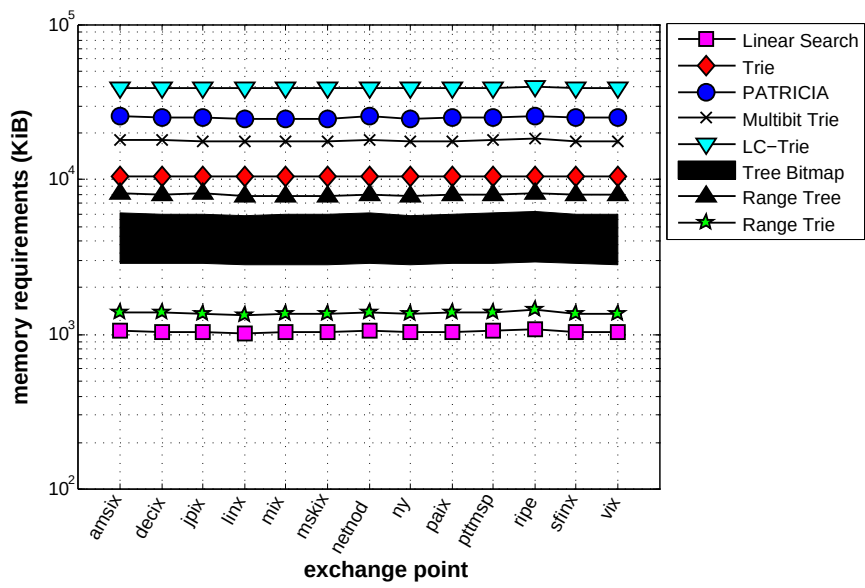


Figure 5.13: Memory requirement for real IPv4 routing tables

5.3.3 For IPv4 routing tables projected to IPv6

The prefixes in the real IPv4 routing tables are project to IPv6 as reported in Section 5.2.1. The latency and memory requirements for the Range Trie are obtained by running the Range Trie heuristics on the projected routing tables. IPv6 implementations of the other algorithms were not available. But as these algorithms scale linearly with address width, the latency and memory requirements may also quadruple with respect to their IPv4 values. Considering this scenario, we theoretically calculated the values for other algorithms which are tabulated in Table 5.9. The corresponding plots comparing the latency and memory requirements of the Range Trie with other algorithms are shown in Figure 5.14 and Figure 5.15 respectively.

Algorithm	Exchange Point, # of Iterations												
	<i>amsix</i>	<i>decix</i>	<i>jpix</i>	<i>linx</i>	<i>mix</i>	<i>mskix</i>	<i>netnod</i>	<i>ny</i>	<i>paix</i>	<i>ptt</i>	<i>ripe</i>	<i>sfinx</i>	<i>vix</i>
Trie	128	128	128	128	128	128	128	128	128	128	128	128	128
PTrie	92	92	92	92	92	92	92	92	92	92	92	92	92
MbTrie	28	28	28	28	28	28	28	28	28	28	28	28	28
LCTrie	24	24	20	28	24	24	24	24	24	24	24	24	24
TBitmap	24	24	24	24	24	24	24	24	24	24	24	24	24
RTree	18	18	18	18	18	18	18	18	18	18	18	18	18
RTrie	6	6	6	6	6	6	6	6	6	6	6	6	6
Algorithm	Memory Usage (KB)												
	<i>amsix</i>	<i>decix</i>	<i>jpix</i>	<i>linx</i>	<i>mix</i>	<i>mskix</i>	<i>netnod</i>	<i>ny</i>	<i>paix</i>	<i>ptt</i>	<i>ripe</i>	<i>sfinx</i>	<i>vix</i>
Linear	4232	4172	4144	4100	4116	4124	4212	4108	4152	4204	4308	4184	4132
Trie	42080	42080	42080	42080	42080	42080	42080	42080	42080	42080	42080	42080	42080
PTrie	25552	25168	25040	24784	24912	24912	25552	24912	25168	25424	25936	25396	25040
MbTrie	72688	71648	71168	70144	70640	70640	71712	70672	71136	72208	74240	71152	70672
LCTrie	157552	156416	155648	154656	155344	155392	157408	154736	155728	157280	159376	156560	155520
RTree	32352	31824	32352	31296	31296	31296	31824	312967	31824	31824	32880	31824	31824
RTrie	1988	1976	1941	1908	1932	1919	1949	1922	1961	1986	1912	1944	1923

Table 5.9: Comparison for real IPv4 routing tables projected to IPv6

From all the comparisons performed in Sections 5.3.1-5.3.3, clearly, the Range Trie outperforms other algorithms even after including the support for Longest Prefix Match and Incremental updates. LC Trie shows better latency for smaller routing tables as shown in Figure 5.8 but has very high memory requirement. The Range Trie implementation shows a perfect balance between the performance and memory usage compared to the other implementations. Also compared to the other algorithms the Range Trie shows better scalability with increasing routing table sizes. Range Trie also scales better than other algorithms with increase in address width from IPv4 to IPv6 addresses. With IPv6 being the future solution to satiate the needs of the Internet usage, Range Trie can probably evolve as the future solution for IP lookup in network routers.

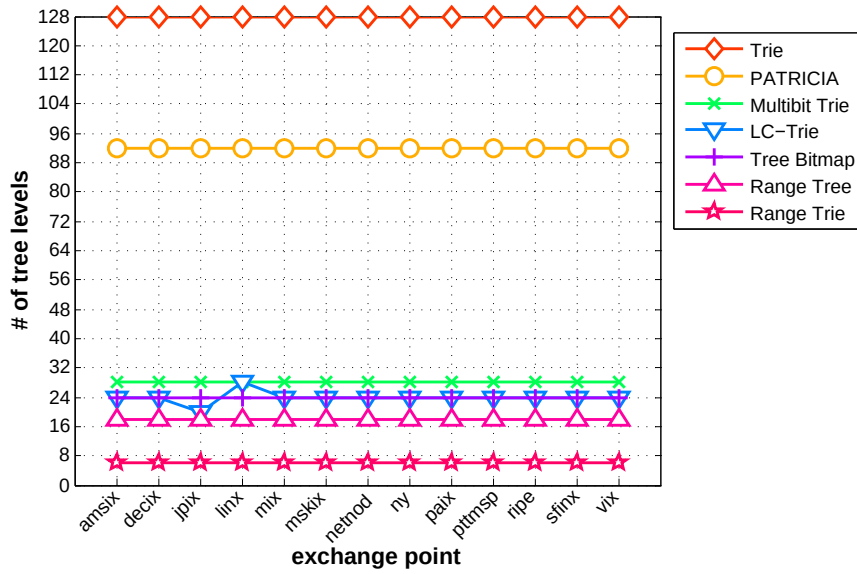


Figure 5.14: Number of tree levels(latency) for IPv4 routing tables projected to IPv6

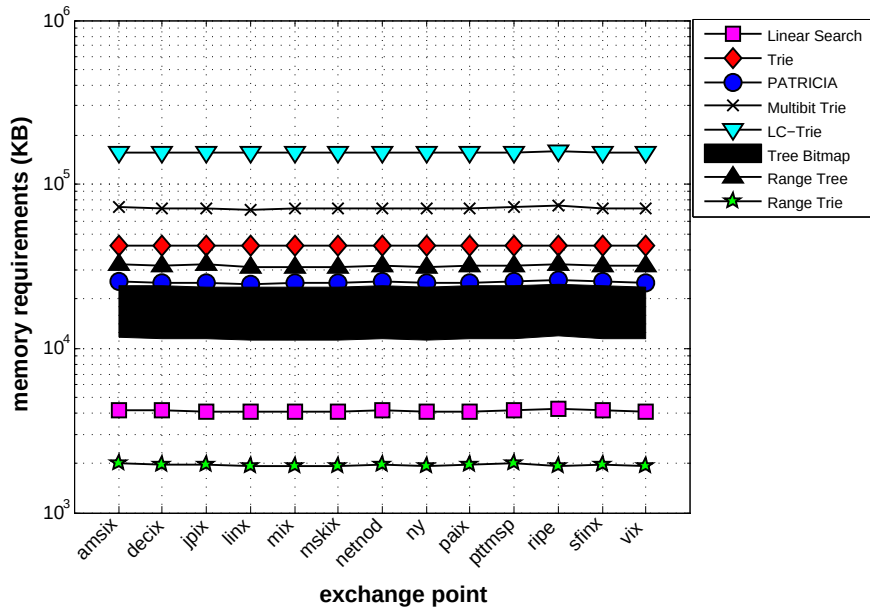


Figure 5.15: Memory requirement IPv4 routing tables projected to IPv6

5.4 Update results

The RIS provides raw data of the routing tables and the updates for each of the Internet Exchange points. The whole routing tables are provided every 8 hours and an update file is generated every 5 minutes. So 3 routing table files and 288 update files are generated each day. We collected routing tables and update files spanning for a period of an year (June-2009 to May-2010) to evaluate our update structure.

As mentioned in section 3.5, when a prefix with non-existing bounds is inserted into the Range Trie, the new bound is inducted into the spare levels. A sub-tree is rebuilt when it cannot accommodate any more bounds even if the sub-tree is not full. The whole Range Trie is rebuilt when a sub-tree is full and a new bound is inducted into the same sub-tree. A simulator that mimics the function of the Range Trie in the spare levels is created. The updates files are extracted and fed to the simulator in the required format.

A plot showing the update requests per second over the period of an year is shown in Figure 5.16. An average of 100 updates per second arrived as can be inferred from the figure.

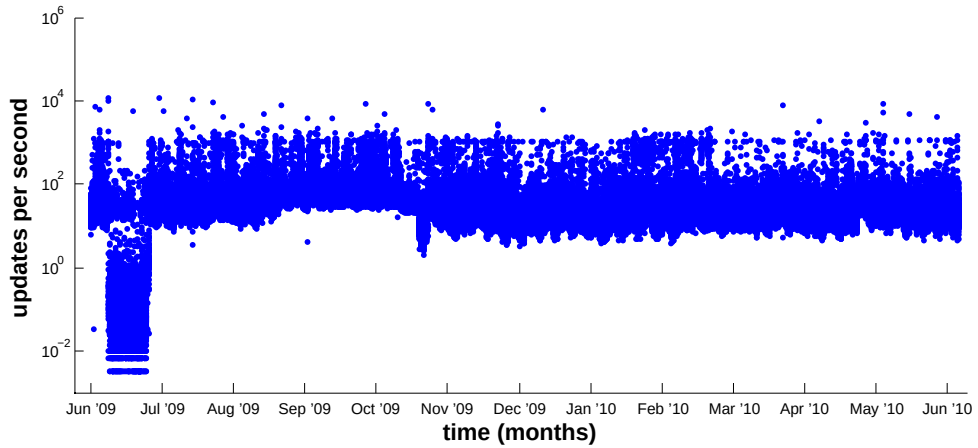


Figure 5.16: Update requests for a period of a year

Figure 5.17 shows the number of occurrences of sub-tree rebuilds per month. Over the period of a year we needed just 6 complete rebuilds of the Range Trie. The number above the bar represent the number of complete Range Trie rebuilds occurred in that month.

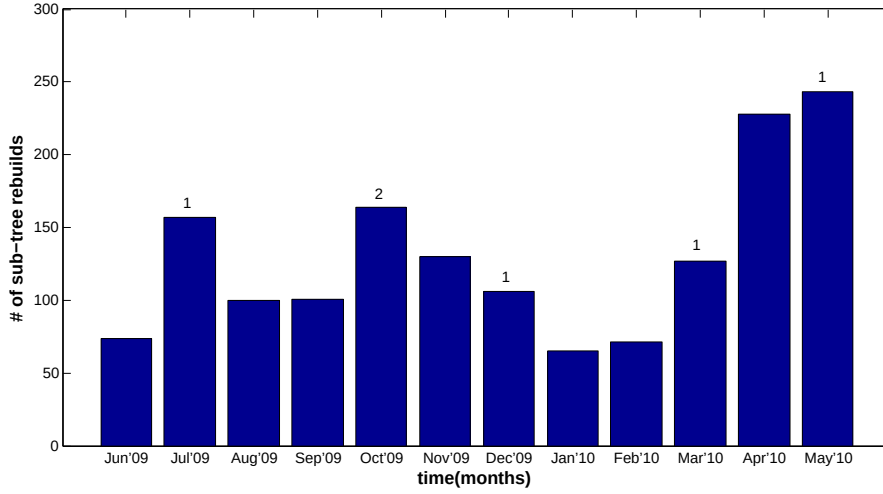


Figure 5.17: Number of sub-tree rebuilds over a period of a year

We consider an operating frequency of 630MHz corresponding to the size of the routing tables as mentioned in [26]. The amount of overhead incurred in number of cycles by the Range Trie design to handle updates is shown in Figure 5.18. There are several hundred of rebuilds of the subtrees over the year and the need to rebuild the whole Range Trie occurred only 6 times in the whole year. There were a lot of subtree rebuilds within a short time due to the in-order arrival of the updates. This phenomenon can be seen highlighted in Figure 5.18.

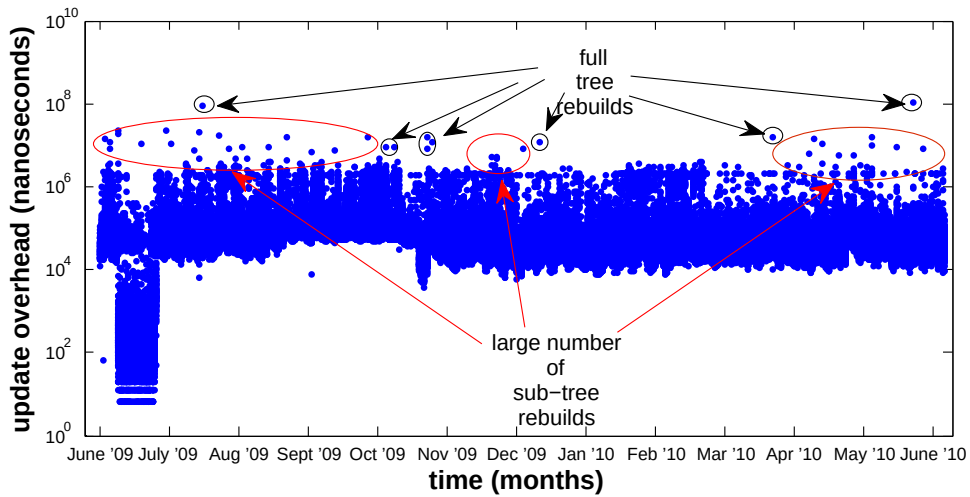


Figure 5.18: Overhead time due to updates

The performance of the Range Trie with incremental updates over the year in real time is as shown in Figure 5.19. Since the number of updates per 5 minutes in the real

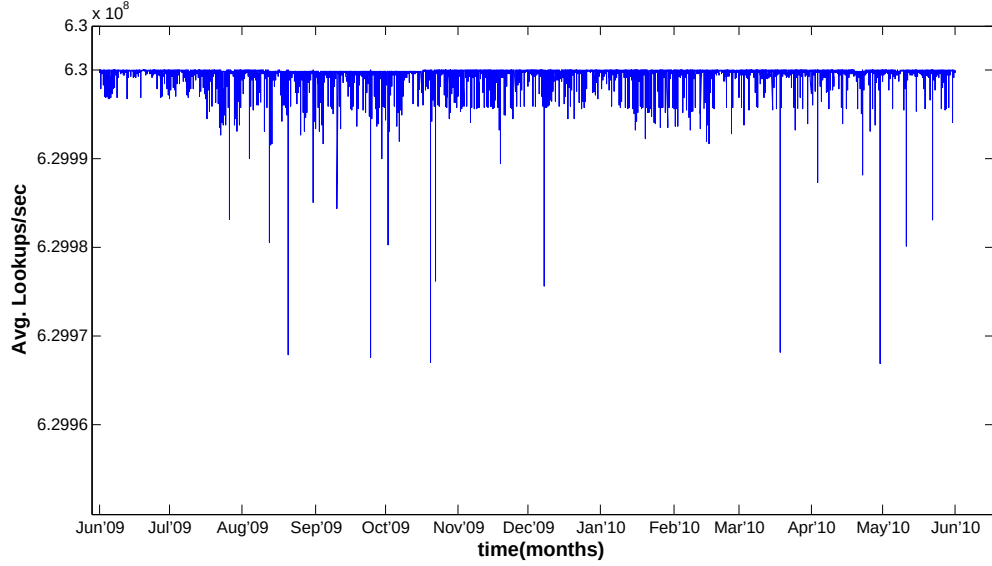


Figure 5.19: Performance overhead with real time updates

routing tables is not constant, we chose to feed in constant number of updates per second into the Range Trie design. We have simulated the Range Trie design ranging from zero updates/second to 1Million updates/sec. With an operational frequency of 630MHz, 630 million lookups can be performed per second without any updates. The overhead incurred by feeding fixed number of updates per second is shown in Figure 5.20. The spikes showing a sudden increase in the overhead time is due to complete rebuild of the Range Trie.

5.5 Summary

A table reporting the lookup and update complexities of the algorithms is obtained from [21] and the Range Trie complexities are added to the table as shown in Table 5.10. The search on length approaches have complexities mentioned in terms of W which is the address width and the search on value based approaches have complexities in terms of N , where N =number of prefixes. There is a factor of $\log_k$ in Range Tree approaches where k is the branching factor per node. For Range Trie, k varies from 8-28 for IPv4 addresses and from 2-16 for IPv6 addresses.

The results of the Range Trie algorithm are compared with the existing solutions and it is clear that the Range Trie outperforms them in terms of performance and memory requirement. The scalability with increasing address width and increasing size of routing tables is also better compared to the other solutions. So Range Trie can be the future solution if the Internet service providers decide to move from IPv4 to IPv6

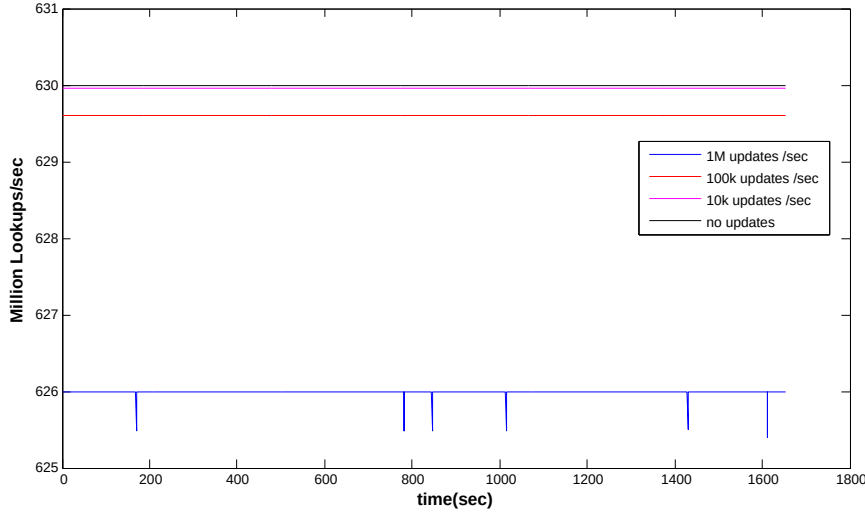


Figure 5.20: Overhead when fixed number of updates are fed

<i>Scheme</i>	<i>Worst Case Lookup</i>	<i>Update</i>	<i>Memory</i>
Binary Trie	$O(W)$	$O(W)$	$O(NW)$
Path-compressed Trie	$O(W)$	$O(W)$	$O(N)$
k-stride Multibit Trie	$O(W/k)$	$O(W/k + 2^k)$	$O(2^k NW/k)$
LC Trie	$O(W/k)$	-	$O(2^k NW/k)$
Lulea Trie	$O(W/k)$	-	$O(2^k NW/k)$
Full Expansion/Compression	3	-	$O(2^k + N^2)$
Binary Search on Prefix Lengths	$O(\log_2 W)$	$O(N \log_2 W)$	$O(\log_2 W)$
Binary Range Search	$O(\log_2 N)$	$O(N)$	$O(N)$
Multiway Range Search	$O(\log_k N)$	$O(N)$	$O(N)$
Multiway Range Tree	$O(\log_k N)$	$O(k \log_k N)$	$O(N k \log_k N)$
Range Trie	$O(\log_k N)$	$O(k \log_k N)$	$O(N k \log_k N)$

Table 5.10: Complexity Comparison

addresses, as IPv4 address space is close to exhaustion which as reported by APNIC ¹ can happen in 2010-2011.

A normal update can be handled with a bubble of 4 cycles in which 2 cycles are used for lookup of the prefix and the other 2 cycles for writing the updated contents to memory. Considering an operating frequency of 630MHz for the Range Trie as mentioned by George Stefanakis in his thesis[26], The Range Trie structure can handle 630 million lookups per second with no updates. Introducing updates gives an overhead which is efficiently handled using the spare levels. The Range Trie is completely rebuilt when it

¹<http://www.apnic.net/community/ipv6-program/ipv4-exhaustion>

cannot handle any updates during which the lookup process is to be halted. Simulations reported only 6 complete rebuilds of the Range Trie over the period of a year and it can still handle 600 million lookups per second during the complete rebuild which we think is quite efficient. We think that the update scenario would be equally efficient for IPv6 addresses.

Conclusion

The Range Trie algorithm has proven that it can be the possible solution to solve the vows of the internet routers. In order to work in the real world internet traffic, the Range Trie algorithm should comply with CIDR addressing scheme. To efficiently handle the bloating routing table sizes and efficient use of address space CIDR uses variable length address prefixes. So, the routing path of the incoming packets is determined by the longest matching prefix among all the matched prefixes.

Routers should support dynamic updates to its routing table in order to adapt to the changes in the network structure. Since the time and effort spent on updating the routing table can adversely affect the network performance, it is desirable to have an update mechanism to handle fast update rates.

The existing Range Trie structure performs lookup using exact range matching. The main objective of this thesis is (a) To adapt the Range Trie structure to support lookup using longest prefix match (b) To develop an update mechanism to support dynamic updates to the routing table maintaining the inherent characteristics of the Range Trie algorithm such as low latency, high throughput, low memory requirement and scalability with address width.

Section 6.1 summarizes the proposed solution for the thesis and my contributions to this thesis are reported in section 6.2. Section 6.3 concludes the thesis with some suggestions for future work.

6.1 Summary

In Chapter 2, a few notable designs and algorithms and their efficiency in address lookup with longest prefix match and their efficiency in handling dynamic updates were presented. They were classified into Search on length and search on value approaches. The Range Trie algorithm, a novel approach for address lookup that delivers the required characters of a lookup algorithm was introduced. Using the five fundamental rules, the Range Trie tries to perform as many comparisons as possible within the limitations of its parameters such as bandwidth. The Range Trie has proved to be a fast, efficient and scalable method with low memory requirements and outperformed the existing algorithms. Since the bottom-up approach proved to be efficient among the developed heuristics, we used bottom-up approach throughout this thesis.

The challenges involved in adapting the Range Trie structure to support LPM and dynamic updates, and their solutions were discussed in detail in Chapter 3. Prefixes

should be associated to the nodes in order to obtain LPM while performing lookup. Prefix lengths were stored at suitable nodes according to the two prefix storing rules that evaded redundant storage of prefixes. This also reduced the number of nodes to be updated when a prefix is inserted or deleted. Small action tables were also introduced at each level of the Range Trie to reduce the update complexity and action pointers to address these action tables were stored along with the prefix lengths. Only two nodes are visited and updated at each level of the Range Trie. To support the insertion of prefixes with non-existing bounds, spare tree levels were introduced. The spare level contains several sub-trees that only handle full length comparisons similar to the Multiway Range Tree method. The sub-trees are rebuilt when no new insertion can be made to the sub-tree even though the sub-tree is not full. A completely full sub-tree establishes the need for the complete rebuild of the Range Trie structure.

Chapter 4 presents the hardware description of the modified Range Trie design. Memory is allocated to store prefix lengths and action pointers at the nodes. The two bounds of the prefix traverse together down the pipeline until there is a split. An existing prefix is updated only when the length of the new prefix is greater than the existing prefix lengths at the node that are eligible for an update. So logic is developed to update only the suitable prefix lengths and logic to find a location in the action table suitable for update. Each bound only performs single write per level, so effectively each update operation requires a bubble of only 4 cycles in the pipeline(2 read cycles for lookup of prefix bounds and 2 write cycles for update). A specialized memory structure is designed to allocate memory dynamically for the sub-trees in the spare levels. Finally, we have a working hardware design of the Range Trie that supports LPM and dynamic updates. The hardware design is prototyped on a Xilinx ML410 Embedded development platform to prove its functional credibility.

In chapter 5 The performance and memory requirements of the modified Range Trie structure is compared with other existing designs in the literature. The Range Trie method after including the support for LPM and updates performs much better compared to the existing designs. The real routing tables and updates for a period of an year(June 2009 to May 2010) were collected from RIPE NCC database¹ in Amsterdam. These are converted into a suitable input format and fed to the simulator that replicates the function of the new Range Trie design. Only 6 complete rebuilds of the Range Trie were required for the whole year with the existing structure. Considering an operating frequency of 630MHz, the new Range Trie design can perform more than 600 million lookups per second in the worst case i.e. when the Range Trie is to be rebuilt completely.

With the increasing complexity of the current routing tables and with IPv4 address space close to exhaustion, the Range Trie design can be the future solution for internet routers.

¹<http://data.ris.ripe.net/rrc00/>

6.2 Contributions

The main focus of this thesis is to find a solution to adapt the existing Range Trie method to support LPM and dynamic updates maintaining the inherent characteristics such as (a) low latency (b) high throughput (c) low memory requirement (d) scalability with address width and increasing routing table sizes.

The contributions of this thesis are:

- ***Support for Longest Prefix Match*** : An efficient method was devised to store the prefixes in the Range Trie structure to perform lookups with Longest prefix match avoiding the redundant storage of prefixes in the data structure.
- ***Fast Dynamic Updates*** : A specialized hardware was designed to support fast update rates. Extra levels of the Range Trie with dynamic memory allocation and a modified Processing Unit to handle updates was designed to accommodate non-existing bounds. Fast update rates were achieved minimizing the number of memory accesses per update.
- ***A complete pipelined hardware design of the algorithm*** : The hardware design was completely pipelined to handle IP address lookups and to handle update operations. The specialized hardware for LPM and updates was positioned in the pipeline such that the operating frequency of the design does not differ from the initial design that performed exact range match lookup. So there was no latency overhead due to the updated hardware design. Each update operation required a bubble of 4 cycles in the pipeline and only $2 \cdot O(\log_m N)$ writes were performed (m =number of children per node and N = total number of bounds) per prefix update unless a sub-tree rebuild or a complete Range Trie rebuild is required.
- ***FPGA Prototype of the hardware design*** : The hardware design is prototyped on to a Xilinx ML410 Embedded development platform to prove that the design is functionally correct.

Bibliography

- [1] *Border gateway protocol(bgp)*, <http://www.pulsewan.com/data101/pdfs/bgp.pdf>.
- [2] *Understanding ip addressing: Everything you ever wanted to know*, Tech. report, 3COM, www.3com.com/other/pdfs/infra/corpinfo/en_US/501302.pdf.
- [3] *Hyperconnectivity and the approaching zettabyte era*, Tech. report, Cisco Systems Inc., 2010, http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/VNI_Hyperconnectivity_WP.html.
- [4] Burton H. Bloom, *Space/time trade-offs in hash coding with allowable errors*, Communications of the ACM **13** (1970), 422–426.
- [5] Chia-Tai Chan, Pi-Chung Wang, Shuo-Cheng Hu, Chung-Liang Lee, and Rong-Chang Chen, *Scalable ip routing lookup in next generation network*, ICOIN, 2003, pp. 46–55.
- [6] Yeim-Kuan Chang and Yung-Chieh Lin, *Dynamic segment trees for ranges and prefixes*, IEEE Trans. Comput. **56** (2007), no. 6, 769–784.
- [7] Ed Spitznagel David, David Taylor, and Jonathan Turner, *Packet classification using extended tcams*, in Proceedings of IEEE International Conference on Network Protocols (ICNP, 2000).
- [8] R. de Smet, *Range trie heuristics for variable-size address region lookup*, Master’s thesis, May 2009.
- [9] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink, *Small forwarding tables for fast routing lookups*, in ACM Sigcomm, 1997, pp. 3–14.
- [10] Sarang Dharmapurikar, Praveen Krishnamurthy, and David E. Taylor, *Longest prefix matching using bloom filters*, SIGCOMM ’03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications (New York, NY, USA), ACM, 2003, pp. 201–212.
- [11] Sarang Dharmapurikar, Haoyu Song, Jonathan Turner, and John Lockwood, *Fast packet classification using bloom filters*, ANCS ’06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems (New York, NY, USA), ACM, 2006, pp. 61–70.
- [12] Will Eatherton, Zubin Dittia, Z. Dittia, and George Varghese, *Tree bitmap : Hardware/software ip lookups with incremental updates*, 2002.
- [13] V. Fuller, T. Li, J. Yu, and K. Varadhan, *Classless inter-domain routing (cidr): an address assignment and aggregation strategy*, 1993.

- [14] Pankaj Gupta, Steven Lin, and Nick Mckeown, *Routing lookups in hardware at memory access speeds*, 1998, pp. 1240–1247.
- [15] Stefanos Kaxiras, *Ipstash: A power-efficient memory architecture for ip-lookup*, In Proc. of MICRO-36, 2003, pp. 361–372.
- [16] ———, *Ipstash: A set-associative memory approach for efficient ip-lookup*, IEEE Infocom (2005), 992–1001.
- [17] Craig Labovitz, G. Robert Malan, and Farnam Jahanian, *Internet routing instability*, IEEE/ACM Trans. Netw. **6** (1998), no. 5, 515–528.
- [18] Haibin Lu and Sartaj Sahni, *A b-tree dynamic router-table design*, IEEE Trans. Comput. **54** (2005), no. 7, 813–824.
- [19] Donald R. Morrison, *Patricia—practical algorithm to retrieve information coded in alphanumeric*, J. ACM **15** (1968), no. 4, 514–534.
- [20] S. Nilsson and G. Karlsson, *Fast address look-up for internet routers*, BC '98: Proceedings of the IFIP TC6/WG6.2 Fourth International Conference on Broadband Communications (London, UK, UK), Chapman & Hall, Ltd., 1998, pp. 11–22.
- [21] Miguel . Ruiz-sánchez, Inria Sophia Antipolis, Ernst W. Biersack, Sophia Antipolis, Walid Dabbous, and Inria Sophia Antipolis, *Survey and taxonomy of ip address lookup algorithms*, IEEE Network **15** (2001), 8–23.
- [22] Keith Sklower, *A tree-based packet routing table for berkeley unix*, 1991, pp. 93–99.
- [23] I. Sourdis, G. Stefanakis, R. de Smet, and G. N. Gaydadjiev, *Range tries for scalable address lookup*, ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), October 2009, pp. 143–152.
- [24] Ioannis Sourdis, Ruben De Smet, and Georgi N. Gaydadjiev, *Range trees with variable length comparisons*, HPSR'09: Proceedings of the 15th international conference on High Performance Switching and Routing (Piscataway, NJ, USA), IEEE Press, 2009, pp. 42–47.
- [25] V. Srinivasan and G. Varghese, *Fast address lookups using controlled prefix expansion*, ACM Transactions on Computer Systems **17** (1999), 1–40.
- [26] G. Stefanakis, *Design and implementation of a range trie for address lookup*, Master's thesis, July 2009, p. 132.
- [27] Subhash Suri, George Varghese, and Priyank Ramesh Warkhede, *Multiway range trees: Scalable ip lookup with fast updates*, Proc. IEEE GLOBECOM 01 , v3 2001, 2001, pp. 1610–1614.