TRIE-TREE DATA STRUCTURE FOR IP LOOKUP IN VIRTUAL ROUTERS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

DİLEK BAYSAL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

JANUARY 2014

Approval of the thesis:

TRIE-TREE DATA STRUCTURE FOR IP LOOKUP IN VIRTUAL ROUTERS

submitted by **DİLEK BAYSAL** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** ______

Prof. Dr. Gönül Turhan Sayan
Head of Department, **Electrical and Electronics Eng.** ______

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı
Supervisor, **Electrical and Electronics Eng. Dept., METU** ______

Assist. Prof. Dr. Oğuzhan Erdem
Co-Supervisor, **Electrical and Electronics Eng. Dept., Trakya Univ.** ______

Examining Committee Members:

Prof. Dr. Semih Bilgen
Electrical and Electronics Engineering Dept., METU ____________________

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı
Electrical and Electronics Engineering Dept., METU ____________________

Assoc. Prof. Dr. Ece Güran Schmidt
Electrical and Electronics Engineering Dept., METU ____________________

Assoc. Prof. Dr. İlkay Ulusoy
Electrical and Electronics Engineering Dept., METU ____________________

Dr. Atilla Özgit
Computer Engineering Dept., METU ____________________

Date: **23/01/2014**

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Dilek BAYSAL

Signature :

ABSTRACT

TRIE-TREE DATA STRUCTURE FOR IP LOOKUP IN VIRTUAL ROUTER

Baysal, Dilek
M.Sc., Department of Electrical and Electronics Engineering
Supervisor: Assoc. Prof. Dr Cüneyt F. Bazlamaçcı
Co-Supervisor: Assist. Prof. Dr. Oğuzhan Erdem

January 2014, 95 pages

Virtual router is an essential solution to fulfill the increasing demands of network services. A virtual router, having a single hardware platform, serves several networks concurrently and hence provides cost saving.

A virtual router maintains multiple forwarding tables that belong to separate internet service providers (ISPs) and performs IP lookup and forwarding functionality for each ISP in one common platform. IP lookup in a virtual router is performed by inspecting the incoming packets that also carry information about their ISPs. There exist various software and hardware IP lookup solutions in the literature. In software solutions tree or trie based data structures are usually employed which are relatively slower. Hardware solutions use TCAMs or SRAMs and are much faster. Limited on-chip memory is the main bottleneck for hardware implementations. If all ISP forwarding tables of a virtual router are stored separately then a large amount of memory is required and there occurs no benefit for having a virtual router. Therefore tables are usually merged in such a way that the overlapping parts are stored in one common data structure more efficiently.

Decreasing the size of the memory and increasing the performance of look up and update tasks are among the primary concerns and challenges in virtual routers. Lookup performance is considered by means of latency and throughput issues.

In this thesis, we investigate and propose an efficient trie overlapping approach and aim to decrease the memory requirement while achieving a good IP lookup and update performance. During this study, we examine real life prefix tables that belong to existing routers. We first merge these IPv4/IPv6 core and edge router tables in a simple manner into a single trie and observe that some parts of the trie use a large number of nodes to store a small number of prefixes. This observation has motivated and led us to reduce the size of the trie by truncating some of these low density subtries without destroying the advantageous trie structure too much. 2-3 tree data structure is then proposed to be used as a secondary storage to keep the deleted prefixes from the trie separately. 2-3 tree is preferred because of its support for incremental updates. Our thesis identifies truncation metrics and we use them to keep most of the prefixes still in the trie.

Our approach is evaluated and we have shown that it is possible to achieve around 5% reduction in memory size for IPv4 core routers and around 35% reduction for other type of routers in comparison to existing trie merging solutions. Hence memory efficiency is increased while supporting an update process that is possible using a single write bubble. Our solution achieves one lookup per clock cycle throughput and operates with a latency that is standard among other trie based solutions.

Keywords: IP lookup, virtual router, table update, memory efficiency

ÖZ

SANAL YÖNLENDİRİCİLERDE IP ARAMA İÇİN AĞAÇ VERİ YAPISI

Baysal, Dilek
Yüksek Lisans, Elektrik Elektronik Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Cüneyt F. Bazlamaçcı
Ortak Tez Yöneticisi : Y. Doç. Dr. Oğuzhan Erdem

Ocak 2014, 95 sayfa

Sanal yönlendiriciler ağ servislerinin artan taleplerini karşılamak için önemli bir çözüm olmuştur. Sanal yönlendiriciler tek bir donanım platformu kullanarak birden fazla ağa eş zamanlı olarak hizmet sunabilirler; böylece tasarruf sağlarlar.

Sanal yönlendiriciler farklı internet servis sağlayıcılarına ait birden fazla yönlendirme tablosunu idame ettirebilir, her bir internet servis sağlayıcı için IP adresi arama ve yönlendirme faaliyetlerini ortak bir platform kullanarak yürütebilirler. Sanal yönlendiricilerde IP adresi arama işlemi paketlerin taşıdığı servis sağlayıcı bilgisi kullanılarak gerçekleştirilir. IP adresi arama için çeşitli yazılım ve donanım çözümleri geliştirilmiştir. Donanımsal çözüm olarak TCAM ve SRAM kullanılırken, yazılım tabanlı çözümlerde daha yavaş olan ağaç yapıları yer alır. Donanımsal ortamın bellek imkanlarının kısıtlı olması uygulamalardaki temel darboğazı oluşturur. Servis sağlayıcılara ait yönlendirme tabloları ayrı ayrı saklanabilir ancak bunun için büyük miktarda bellek gerekmektedir, bu durumda sanal yönlendiriciler önemli bir bellek kazancı sağlayamamaktadır. Tablolardaki benzerliklerden yararlanabilmek için tablolar genellikle birleştirilerek tek bir ortak veri yapısı oluşturulmaktadır.

Bellek büyüklüğünü azaltmak, adres arama performansını yükseltmek ve güncelleme işlemleri sanal yönlendiriciler için öne çıkan önemli konulardır. Adres

arama performansı gecikme süresi ve birim zamanda gerçekleşen arama sayısı kriterleri ile değerlendirilir.

Bu tez çalışmasında, birleştirilmiş bir ağaç (trie) veri yapısı önerilmiş ve bellek ihtiyacının azaltılması hedeflenirken, arama ve güncelleme işlemlerinde de verimlilik sağlanmıştır. IPv4/IPv6 çekirdek ve kenar yönlendiricilere ait gerçek prefix tabloları birleştirilerek tek bir ağaç (trie) oluşturulmuştur. Birleştirilmiş ağaçtaki bazı bölümlerin az sayıda prefix verisi saklamak için yüksek sayıda boş hücre bulundurduğu gözlenmiştir. Çalışmanın çıkış noktası, ağaçtaki bu düşük yoğunluklu bölgeleri ağacın yapısını değiştirmeden ayıklayarak bellek kullanımını azaltmaktır. Esas ağaçtan alınan prefix verileri ikinci bir ağaç (2-3 tree) kullanarak saklanacaktır. 2-3 ağaç yapısı diğer ağaçlara göre hızlı güncelleme yeteneğinden dolayı tercih edilmiştir. Budama ölçütü prefix verilerinin çoğunun esas ağaçta (trie) kalmasını sağlayacak şekilde belirlenmiştir.

Önerdiğimiz yapı mevcut çözümlerle kıyaslandığında, IPv4 çekirdek yönlendiricilerinde %5, diğer tipteki yönlendiricilerde %35 seviyelerinde bellek tasarrufu elde etmiştir. Böylece bellek etkinliği arttırılmış ve her bir güncellemenin tek bir yazma mesajı ile gerçekleştirilmesine olanak sağlanmıştır. Önerdiğimiz yapı her bir adres arama işlemini bir saat döngüsünde gerçekleştirebilir niteliktedir ve gecikme süresi de trie yapısı kullanan diğer çözümler ile eşdeğerdir.

Anahtar Kelimeler: IP arama, sanal yönlendirici, tablo güncelleme, bellek etkinliği

To my mother

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor Assoc. Prof. Dr. Cüneyt Bazlamaçcı for his guidance, support, encouragement, trust, patience, kindness and valuable contributions throughout the preparation of my thesis. I would also like to express my special appreciation to my co-supervisor Assist. Prof. Dr. Oğuzhan Erdem for his technical support, effort, suggestions and comments throughout the development of this thesis.

I would like to thank the other members of my committee, Prof. Dr. Semih Bilgen, Assoc. Prof. Dr. Ece Güran Schmidt, Assoc. Prof. Dr. İlkay Ulusoy and Dr. Atilla Özgit for their review and evaluation.

I would also like to thank to Assist. Prof. Dr. Aydın Carus for his valuable contributions throughout the preparation of this thesis.

I am grateful to my family, friends and colleagues who have given me encouragement and support.

I would like to acknowledge the support of ASELSAN Inc. for the realization of this thesis.

Finally I would like to thank all of those who helped me work on this thesis and throughout my Master of Science education.

TABLE OF CONTENTS

ABSTRACT v
ÖZ vii
ACKNOWLEDGEMENTS x
TABLE OF CONTENTS xi
LIST OF TABLES xiii
LIST OF FIGURES xv

CHAPTERS

1 INTRODUCTION 1
1.1 PROBLEM DEFINITION 3
1.2 MOTIVATION 3
1.3 CONTRIBUTION 4
1.4 THESIS ORGANIZATION 5

2 LITERATURE OVERVIEW 7
2.1 BACKGROUND 7
2.1.1 ROUTER ARCHITECTURE 9
2.1.2 VIRTUAL ROUTER 10
2.1.3 IP LOOKUP 12
2.1.4 DATA STRUCTURES FOR IP LOOKUP 13
2.1.4.1 TRIE 13
2.1.4.2 BINARY SEARCH TREE (BST) 17
2.1.4.3 2-3 TREE 19
2.1.4.4 COMPARISON 21
2.2 IP LOOKUP ALGORITHMS FOR VIRTUAL ROUTERS 23
2.2.1 EFFICIENT IP-ADDRESS LOOKUP WITH A SHARED FORWARDING TABLE FOR MULTIPLE VIRTUAL ROUTERS 23
2.2.2 A TRIE MERGING APPROACH WITH INCREMENTAL UPDATES FOR VIRTUAL ROUTERS 25
2.2.3 MEMORY-EFFICIENT AND SCALABLE VIRTUAL ROUTERS USING FPGA 27
2.2.4 EFFICIENT TRIE BRAIDING IN SCALABLE VIRTUAL ROUTERS 28

3 TRIE PRUNING AND TRIE-TREE DATA STRUCTURE FOR IP LOOKUP 31
3.1 DATA ANALYSIS 31
3.2 TRUNCATION ALGORITHM 34
3.3 DATA STRUCTURE DETAILS 38
3.3.1 TRIE 38
3.3.2 TREE 40
3.4 IP LOOKUP 42
3.5 UPDATE 44
4 REAL LIFE LOOKUP TABLE ANALYSIS AND DETAILED DATA STRUCTURE DESIGN 47
4.1 MEMORY SIZE CALCULATION 51
4.1.1 TRIE NODE SIZE 52
4.1.2 TREE NODE SIZE 54
4.1.3 TOTAL DATA STRUCTURE SIZE 57
5 COMPARATIVE EVALUATION 61
5.1 MEMORY SIZE COMPUTATION FOR THE EXISTING SOLUTIONS 61
5.1.1 SIMPLE MERGING WITHOUT LEAF PUSHING 61
5.1.2 SIMPLE MERGE WITH LEAF PUSHING 62
5.1.1 TRIE MERGING WITH INCREMENTAL UPDATES 66
5.1.2 SET BOUNDED LEAF PUSHING 68
5.2 COMPARISON AND REMARKS 74
6 CONCLUSION 81
REFERENCES 83
APPENDICIES
A IPv4 CORE ROUTER ANALYSIS 85
B IPv4 EDGE ROUTER ANALYSIS 89
C IPv6 CORE ROUTER ANALYSIS 91
D IPv6 EDGE ROUTER ANALYSIS 93

LIST OF TABLES

TABLES

Table 2-1 FIB table example 14
Table 2-2 Modified prefixes (disjoint prefix set) 18
Table 2-3 Complexity comparisons 22
Table 3-1 Example FIBs belongs four virtual routers 32
Table 3-2 Truncated prefixes 37
Table 4-1 Prefix numbers in the merged FIBs 48
Table 4-2 Number of prefixes and nodes in the merged trie 48
Table 4-3 Truncation criteria and results 50
Table 4-4 List of used notations 51
Table 4-5 Size of child pointer field 53
Table 4-6 Size of next hop array pointer field 53
Table 4-7 Size of trie node 54
Table 4-8 Tree child pointer size 56
Table 4-9 Tree node size 57
Table 4-10 Trie-Tree structure total memory size 58
Table 4-11 Next hop array table size 59
Table 5-1 Memory results for simple merge without leaf pushing algorithm 62
Table 5-2 Node size results for simple merge with leaf pushing algorithm 64
Table 5-3 Memory results for simple merge with leaf pushing algorithm 65
Table 5-4 Node size results for trie merging with incremental updates algorithm 67
Table 5-5 Memory results for trie merging with incremental updates algorithm .. 68
Table 5-6 Number of nodes in T_1 and T_2 70
Table 5-7 Memory results for set bounded leaf pushing algorithm 71
Table 5-8 Memory footprint (in MB) 75
Table 5-9 Memory gain of the proposal with respect to other trie based methods 80

Table A-1 IPv4 core router analysis 85
Table B-1 IPv4 edge router analysis 89
Table C-1 IPv6 core router analysis 91
Table D-1 IPv6 core router analysis 93

LIST OF FIGURES

FIGURES

Figure 2-1 Network topology example 8
Figure 2-2 Router architecture 10
Figure 2-3 Binary trie corresponding to FIB in Table 2-1 15
Figure 2-4 Leaf pushed trie 16
Figure 2-5 BST corresponding to the example disjoint FIB 18
Figure 2-6 2-3 tree nodes 20
Figure 2-7 Example 2-3 tree 21
Figure 2-8 Trie node structure in direct merging 23
Figure 2-9 Proposed lookup data structure in [1] 24
Figure 2-10 Trie merging data structure 26
Figure 2-11 Stages of the set bounded leaf pushing algorithm 28
Figure 2-12 Trie braiding example 30
Figure 3-1 Example trie with *(a,b)* parameters 33
Figure 3-2 Truncation algorithm flowchart 35
Figure 3-3 Truncated trie 36
Figure 3-4 2-3 tree composed of truncated prefixes 38
Figure 3-5 Trie structure with next hop 40
Figure 3-6 Pipeline structure 43
Figure 3-7 A pipeline stage with WBT 45
Figure 5-1 Effects of updates on set bounded leaf pushing algorithm 73
Figure 5-2 Memory footprint comparison 76
Figure 5-3 Memory footprint comparison among trie based solutions 79

LIST OF ABBREVIATIONS

CIDR	: Classless Inter-domain Routing
CPU	: Central Processing Unit
FIB	: Forwarding Information Base
IP	: Internet Protocol
ISP	: Internet Service Provider
LPM	: Longest Prefix Match
QoS	: Quality of Service
RIB	: Routing Information Base
SRAM	: Static Random Access Memory
TCAM	: Ternary Content Addressable Memory
WBT	: Write Bubble Table

CHAPTER 1

INTRODUCTION

Number of users, the amount of information and the diversity of services has dramatically increased in the Internet in recent decades. Data centers that maintain large amounts of data are essential parts of the network infrastructures. A virtual router allows management of multiple networks on a single hardware platform and arises as a solution for reducing capital and operational costs.

In a virtual router multiple prefix tables that belong to different networks share a single hardware platform. Each prefix is associated with an outgoing port of a router. A router consults to its prefix table to make a forwarding decision, which is called as IP lookup. In IP lookup, when a packet arrives at a router, destination address of the incoming packet is extracted. This address is then searched in the prefix table for a possible match with an existing prefix. Prefixes have various lengths. If multiple matches occur, longest matching prefix determines the next hop router.

Trie is an advantageous data structure which is commonly encountered in implementing IP lookup operation. In a trie, a prefix is stored explicitly but it is rather denoted implicitly by the path from the root to a final node, where this node keeps the next hop router information associated with the prefix as its data field. Nodes that store next hop information in addition to pointer values are named as full nodes, whereas others that store pointer values only are named as empty ones.

In a virtual router, a single trie is constructed using the prefix tables of several routers belonging separate networks to achieve a memory gain. Existence of overlapping prefixes in these tables increase the memory efficiency of the

constructed trie. IPv4 core router tables constitutes compact trie structures, however IPv4 edge router tables and IPv6 tables have scattered prefix distributions. Especially for IPv6 routers, numbers of long length prefixes are high, which makes the trie grow enormously. Although some enhancements are proposed to achieve a reduction in the trie memory size, they usually make the update operations more complex in return. Updates are aggregated in virtual routers also. Therefore, quick update capability of a data structure to be used is still essential.

Tree data structure is an alternative for the trie. Although it provides memory efficiency, lookup operation requires one comparison at each stage; number of comparisons increases due to the number of stored prefixes.

Virtual routers can be implemented in software or hardware. Hardware solutions are preferable due to their speed advantage. Hardware solutions are divided into two categories, i.e., SRAM/DRAM based and TCAM based. A TCAM achieves parallel lookup among prefixes in one clock cycle. On the other hand, it is power hungry, expensive and has limited adaptability. An SRAM's power consumption is lower than a TCAM but it requires multiple memory accesses during a single search. Therefore pipelining is generally employed to improve the throughput of SRAM implementations by enabling parallel search in different levels of the tree structure.

Memory consumption is the main bottleneck due to the limited number of memory blocks on chips. Quick update capability, latency and throughput are also highly important concerns for virtual routers.

In the present work we propose a method that provides a memory saving without sacrificing lookup and update performance by considering a pipelined SRAM based implementation. We achieve this memory reduction by truncating sparse subtries existing in the merged trie and hence decreasing the number of empty nodes in the trie. The eliminated prefixes are then stored in an auxiliary 2-3 data structure. Most of the prefixes are still stored in the trie and 2-3 tree is used only as a secondary storage that enables us to reduce the memory size required. Both trie and 2-3 tree

structure support quick updates. Latency of our proposal is the same with other existing trie based solutions and it achieves one lookup per clock cycle throughput conceptually.

1.1 PROBLEM DEFINITION

There are m FIB (forwarding information base) tables that belong to m different routers, $i = 1,\ldots,m,$ and each having N_i prefixes. These tables are to be merged for virtual routing and combined IP packet forwarding. A single data structure is usually constructed by using the merged table. Our aim is to minimize the total memory requirement in this task compared to existing approaches while achieving quick updates at the same time.

1.2 MOTIVATION

Within the scope of this thesis, we carried out an extensive analysis on real prefix tables and have observed that trie structure is not fully utilized in terms of memory in simple merging approach. Prefix lengths are not evenly distributed in lookup tables. Especially for IPv6, the corresponding trie includes long empty node chains to reach a few valid prefixes. These paths are among the reasons for having a large overall memory size and long latency during lookup. Well known techniques such as level compression, path compression and multi-bit trie need extra information to reduce the memory requirement or the latency and moreover cause increased update overhead in general.

Tree is a memory efficient structure, that each node stores a prefix. However search is done by comparing address value with entries thus computational complexity arises. For this reason, tree is not that useful for storing large amounts of data.

Memory size reduction is a critical issue for pipelined structures while maintaining quick update capability, which is essential since each update operation blocks a lookup run.

In our proposal, we use a trie along with a tree to take advantage of both structures. We use a trie as the main data structure but with a smaller and more manageable size because of its advantage in achieving fast lookup and easy update and use an auxiliary 2-3 tree structure for storing some of the prefixes that are deleted from the main structure to make it small.

1.3 CONTRIBUTION

In our study we used four different types of router FIBs, namely IPv4/ IPv6 edge and core routers. We have merged 10 FIBs for each type and analysed the result. Different characteristics of IPv4/ IPv6 edge and core router prefix tables have been observed with respect to their prefix distributions over simply merged trie.

For this purpose a tool, which traverses the simply merged trie, detects and truncates target subtries using a set of predetermined truncation criteria, is developed. The tool produces the results of the truncation in terms of the number of remaining trie nodes and number of truncated prefixes for each criteria.

Size of the merged trie is reduced by truncating sparse subtries. A lookup data structure, which is composed of a simply merged trie and a 2-3 tree is then proposed to be used in virtual routers. Both these data structures support incremental updates. Our study is shown to be the winner among existing competitors in terms of memory efficiency also it is advantageous in terms of fast updates for IPv4 core routers.

Our proposal also achieves a memory reduction of around 35% for IPv4 edge and IPv6 routers in comparison to existing algorithms that also support incremental update and one lookup per clock cycle hence being fairly comparable.

1.4 THESIS ORGANIZATION

The rest of the thesis is organized as follows. Chapter 2 covers background information and reviews the previous studies about virtual routers. Chapter 3 presents our proposed method for increasing memory efficiency. In Chapter 4, experimental results of real lookup tables are given and memory footprint of the proposed structure is calculated and its update performance is evaluated. Chapter 5 presents a comparison of our solution with the existing algorithms and finally, Chapter 6 summarizes and concludes our work.

CHAPTER 2

LITERATURE OVERVIEW

2.1 BACKGROUND

Internet is a system of interconnected computer networks. Hosts are end users of the Internet. Internet Protocol (IP) defines all the necessary rules to transport data packets from source host to the destination host. Every host has an IP address, which is represented as a binary number being 32 bits for IPv4 and 128 bits for IPv6. A data packet has a header and a payload part. The payload contains user data whereas the header includes information to control the packet's journey.

Destination IP address is the key information for routing network packets. Destination address is partitioned into two sub-addresses: first part represents network address and the following part represents host address. Routing operation is performed by using the network address. IP addresses were organized by using Classful Addressing in the early times of the Internet [16]. Network addresses were grouped into 3 variable length fields, 8, 16 and 24 bits, and the rest of the bits were assigned to hosts. However, in this scheme, we run out of IP addresses as network size grow because of having a limited number of host address bits.

Classless Inter-domain Routing (CIDR) addressing scheme was later introduced as a solution, which is still in use. In CIDR, the network address part can be of any length, which in fact no longer defines the network address but rather defines the aggregation of networks. The routing process is then accomplished by a search of the destination address on a prefix table. This search process is called IP lookup.

IP lookup process is a functionality of a router. Routing service is implemented in the network layer of a computer network. Routers are placed at crossroad points of a network topology, which is effectively a map, constituting of routers and links. Core routers operate on network backbones and connect ISP networks to networks. Hosts are connected to ISP networks via edge routers. A core router serves with a larger FIB with respect to an edge router. The update frequency is lower for core routers also because changes in ISPs are rarer than changes in end users. An example router placement in a network topology is shown in Figure 2-1. C*i* denotes core routers, E*j* denotes edge routers and H*k* denotes hosts.

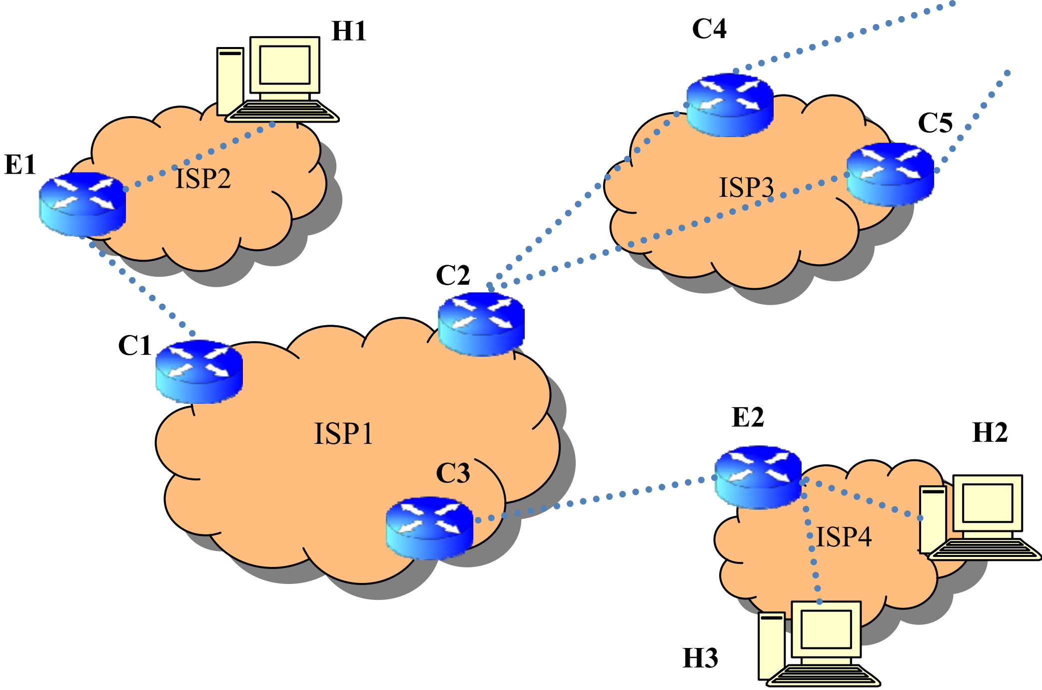

Figure 2-1 Network topology example

2.1.1 ROUTER ARCHITECTURE

A router is a networking device that forwards data packets to specific destinations. A router can be separated into two main parts by functionality: control unit and forwarding unit. Control unit includes CPU and Routing Information Base (RIB) table. Each RIB entry is constituted of a destination IP prefix and corresponding route information. For the same packet, more than one routing information entry can be defined in a RIB. The control unit processes the header part of data packet to determine the route and the protocol. It also runs routing protocols and manages route updates. Forwarding unit is constituted of line cards. Each line card has its own processor, Forwarding Information Base (FIB) table and a forwarder. FIBs are derived from RIBs and only one entry is defined for a packet. Each FIB entry includes a prefix, also next hop address and the corresponding output port for the next hop. Forwarding unit's memory must have higher performance than the control unit's to catch up with the line speed. There exists a switch fabric and a scheduler between the control unit and the forwarding unit. Each outgoing port of a router is linked to a next hop station. Once the next hop port information is retrieved from the lookup process, the packet is sent through the corresponding output port. Router architecture is illustrated as a block diagram in Figure 2-2.

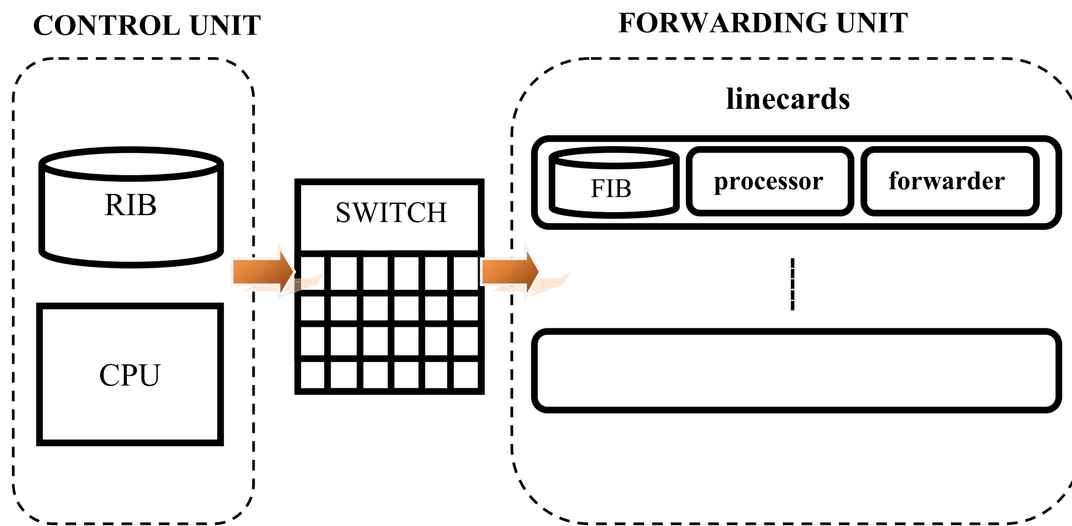

Figure 2-2 Router architecture

2.1.2 VIRTUAL ROUTER

While Internet data traffic grows rapidly, diversity of the services provided by routers also increases. Internet users are spread on a wide spectrum and as a consequence their demands can be variable. For example, speed is the most important factor for multimedia involved users, whereas privacy is indispensable for finance companies and government. 'Best effort' routers were sufficient in early days of the Internet, but today's ISPs have to satisfy strict QoS requirements. Customer specific routing, policy based routing, multi topology routing and network virtualization are some of the services to fulfill variant demands of customers [6]. Customer specific routing is a technology that enables to select different routes

instead of choosing one best route for every packet according to one single criterion. Policy based routing takes packet size, protocol and the application into consideration besides the destination address. Multi topology routing gives ability to create secured routes that are dedicated to only a group of flows. The commonality of these services leads to the need of consulting multiple FIBs to constitute different routes. System management and equipment cost then becomes challenging in fulfilling these requirements.

Virtual router concept rises as a solution to satisfy these requirements in a more advantageous way. Router virtualization is a technique that allows multiple router instances to run on a single physical router platform.

In classical manner, every physical router has only one FIB. A virtual router can assign different specified routes to different customers without needing a separate physical router for each customer. A virtual router encapsulates more than one FIBs. With the help of the ability to keep more than one FIBs, packet specific behavior can be developed. Each FIB table can be dedicated to a definite protocol, so various services can be implemented by the same router. In this way, power consumption is reduced, services are maintained in compact manner and space saving is provided. That leads to increased efficiency in investments.

Virtual router concept also contributes to Internet research area [17]. Experiments on developing network services, migration from IPv4 to IPv6 might require reorganization of the already running system. Virtual router preserves standing areas at the same time giving the ability to make changes on some research areas.

Two main approaches can be followed to enable router virtualization: different router FIBs can be kept in a separate or merged form. Separate scheme assigns isolated memory blocks for each FIB [18]. A discrete data structure is constructed for each virtual router. This method provides perfect isolation among routers. Updates can be achieved by directly reorganizing the related router's data block

without disturbing any other virtual router area. It also provides a saving of the preprocessing time required to merge the FIBs.

Memory usage and performance are the weak points in this approach. The amount of memory increases linearly with the number of supported router instances. This is an undesirable scalability issue. Cache usage is one of the reasons for unsatisfactory performance. Incoming data packet may belong to any one of the virtualized routers meaning that several FIBs should be used concurrently. Today's caches are insufficient to store even one single FIB. Taking in and out several FIBs in cache in short unit of times prevents using the cache efficiently and decreases the lookup performance.

In the shared approach, a common data structure is constructed to store all FIBs together [1-3,5]. It is advantageous in aspect of memory bandwidth. With the help of an appropriate scheduling, memory bandwidth can be used at full load without any waste. Its weak point however, is in isolation and single point failures. Modification affects the whole data structure, so update process requires more effort also.

2.1.3 IP LOOKUP

The main function of a router is forwarding of data packets. IP lookup is necessary to make the forwarding decision. IP lookup operation takes the destination IP address from the packet and then finds the next hop address and the outgoing port. Use of CIDR turns the lookup process into prefix search naturally. Variable length network addresses are represented as prefixes in FIB tables. Prefixes are formed from bits and wildcard characters to denote their variable length. Each prefix is placed with its length and its associated output port in a FIB.

When a data packet comes to router, the header processor extracts the destination address. Line card processor searches the destination address of this packet in its FIB table. Lookup process is completed according to Longest Prefix Matching (LPM)

principle. LPM is a requirement of variable length prefix. For example 101001* and 10100101* prefixes can both be placed at the same FIB with different output ports dedicated. Next hop information from potentially matching prefixes should therefore be temporarily stored and search should be continued until the last entry in case of a longer prefix exists. If more than one prefix matches an IP address, the longest one is valid.

2.1.4 DATA STRUCTURES FOR IP LOOKUP

2.1.4.1 TRIE

Trie is a very suitable data structure for IP lookup by its nature [7]. Trie comes from the word 'retrieve' and it is a special tree data structure. The path from the root node to another one represents the prefix value stored by the corresponding node. Root node represents the wildcard (*) and covers up all addresses. In a binary trie, every node has two pointers, left child pointer is 0 bit path and right child pointer is 1 bit path. Given a FIB table, trie construction process is as follows: Prefixes in the table are examined one by one starting from the leftmost bit in each prefix. According to these values, child pointers and new nodes are created by starting from the root node when necessary. The node at the end of a path, whose value defines a prefix in the FIB stores the corresponding next hop information also. Other nodes have only child pointers and contribute to create paths towards valid prefix nodes. All leaf nodes and some of the intermediate nodes indicate valid prefixes.

Lookup operation starts from the root node. Destination address is used as the key during the search. Every bit leads the traversal of a path. Left pointer is followed when 0 bit comes and right pointer is followed when 1 bit comes. If a match occurs with any node, the next hop information at that node is stored on a temporary

memory. The traversal continues until a leaf is reached. An example FIB table is given in Table 2-1 and the corresponding trie for the example FIB is illustrated in Figure 2-3.

Table 2-1 FIB table example

	name	**value**	**next hop information**
prefixes	a	0*	H1
	b	11*	H2
	c	000*	H3
	d	100*	H4
	e	0010*	H1
	f	0111*	H5
	h	1100*	H6
	i	1101*	H3

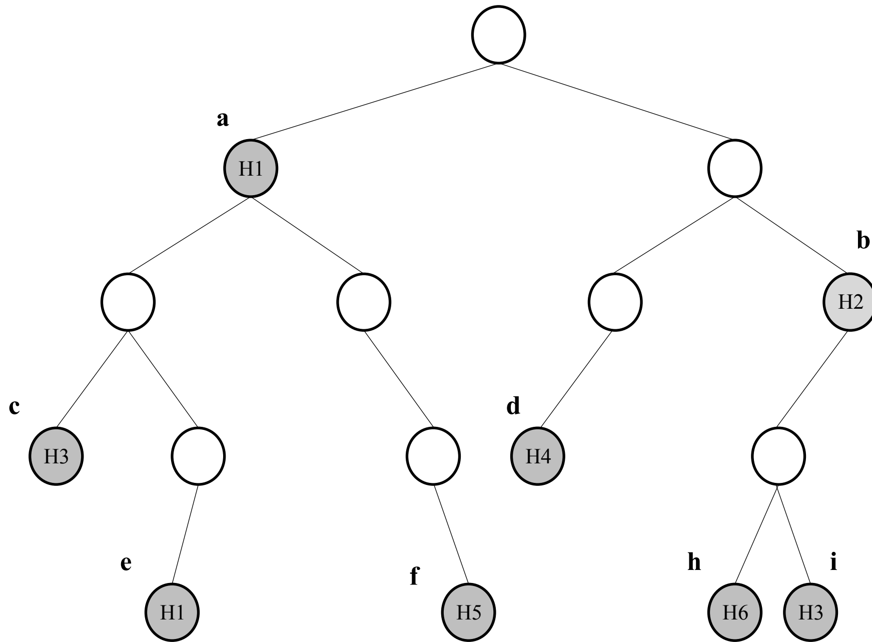

Figure 2-3 Binary trie corresponding to FIB in Table 2-1

Although trie is an appropriate data structure for IP lookup, several enhancements are also proposed. These are leaf pushing, level compression and path compression.

Leaf pushing (aka. child pushing) [9] grows the trie fully to push the prefixes in internal nodes to the leaf nodes. Leaf pushed example trie is seen in Figure 2-4. The aim here is to get rid of the information in internal nodes. This results in a disjoint set of prefixes, which means there is no need to LPM any more, each match actually being the exact match. Two disadvantages of this method are increasing of the number of nodes and update complexity.

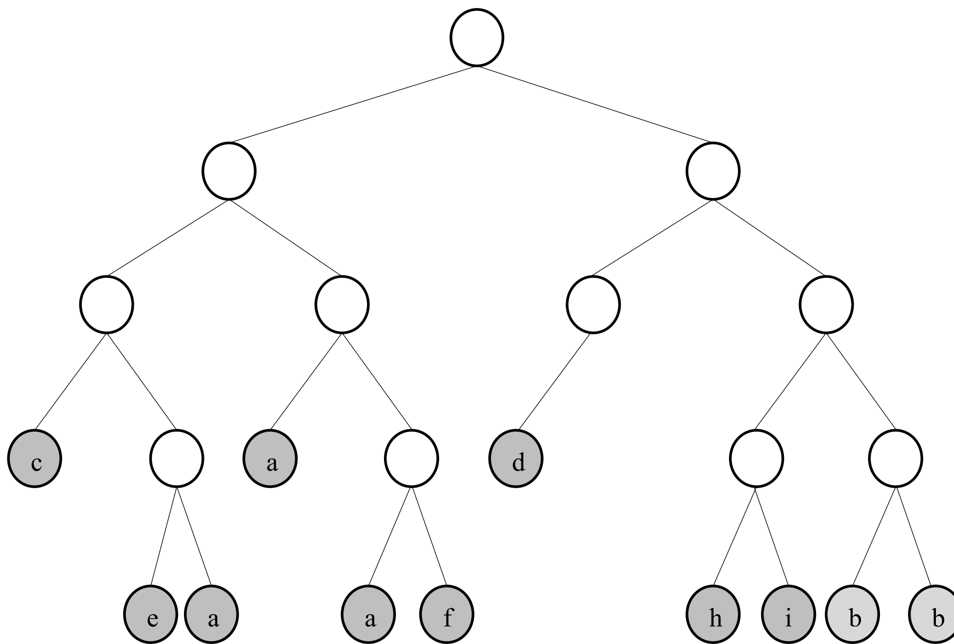

Figure 2-4 Leaf pushed trie

Level compression [10] is another method, which is used to decrease the total number of nodes and the number of trie levels. Multiple bits are used to represent the paths instead of just one bit. n^{th} complete level of the trie is replaced by a single node of degree 2^n. Here, n is the number of bits represented by a path. Although it achieves memory saving and decreases the number of memory accesses, it causes update and computational complexity.

Path compression is a technique in which intermediate nodes with one child are removed recursively [11]. The number of removed branches is kept as 'skip value' at the node where the removal starts. Path compressed tries are also known as patricia trees. One disadvantage of this method is the additional memory used to store the

skip value and the other one is backtracking being not easy due to lack of removed nodes.

2.1.4.2 BINARY SEARCH TREE (BST)

BST is a tree data structure in which prefix values are directly stored into its nodes [8]. The values in the left subtree are smaller than the corresponding node value and the values in the right subtree are greater than the corresponding node value. Each node represents a discrete value.

Every node has two child pointers. IP lookup in a BST is achieved by comparing the searched value with the node value. If the searched value is greater than current node value, traversal operation continues with the right child; otherwise if the searched value is smaller than the current node value, left child is followed. Overlap between the prefixes should be eliminated to use a BST. Child pushing is the most preferable technique to create non-overlapping prefix sets. The FIB in Table 2-1 is transformed to one in Table 2-2 by using the leaf pushed prefixes in Figure 2-4. The BST should also be constructed using equal length prefixes for comparison. Hence an upper length should be chosen and the prefixes are then padded with 1s or 0s up to this predetermined length. For this example, we chose the upper bound on length as 4 bits and shorter prefixes are padded with 0s. Then the prefixes should be sorted in ascending or descending order. A middle valued prefix should be chosen as a pivot. Left and right subtrees should then be constructed recursively using the same principle. The example BST is seen in Figure 2-5.

Table 2-2 Modified prefixes (disjoint prefix set)

	name	**value**	**length**	**next hop information**
prefixes	a1	0011	4	H1
	a2	0100	3	H1
	a3	0110	4	H1
	b1	1110	4	H2
	b2	1111	4	H2
	c	0000	3	H3
	d	1000	3	H4
	e	0010	4	H1
	f	0111	4	H5
	h	1100	4	H6
	i	1101	4	H3

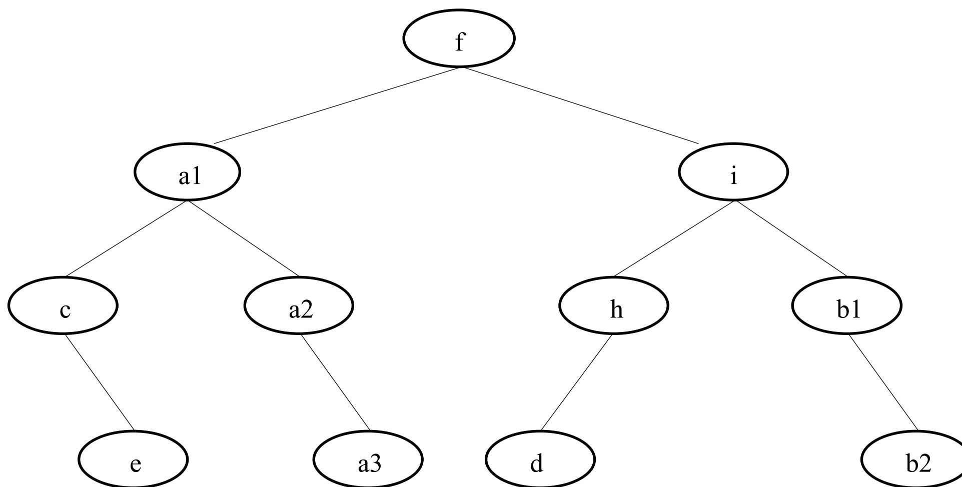

Figure 2-5 BST corresponding to the example disjoint FIB

A memory size enhancement can be done by eliminating child pointers on the complete BST. The nodes at each level could be saved in contiguous addresses. If root index is x, then left child index is 2x and right child index is 2x+1. If the searched data is greater than key; 1 bit is concatenated with the address of current node, otherwise 0 bit is concatenated. Next address is defined on the fly without using any separate pointer field.

2.1.4.3 2-3 TREE

A 2-3 tree is a kind of B-tree [12]. Every node of the tree has two or three children. There are two kinds of leaf nodes: 1-node and 2-node. There are two kinds of internal nodes also: 2-node with two child pointers and 3-node with three child pointers. Node structure is illustrated in Figure 2-6.

The search is done using the following principle. If the current node is a 2-node, search is the same as BST. If it is a 3-node, the pointer is chosen by determining which range does the search data belongs to. 's' denotes searched data and k_1 and k_2 denote first and second data fields in this node. If $s < k_1$ then left pointer is followed, else if $k_1 < s < k_2$ then middle pointer is followed, else if $k_2 < s$ right pointer is followed.

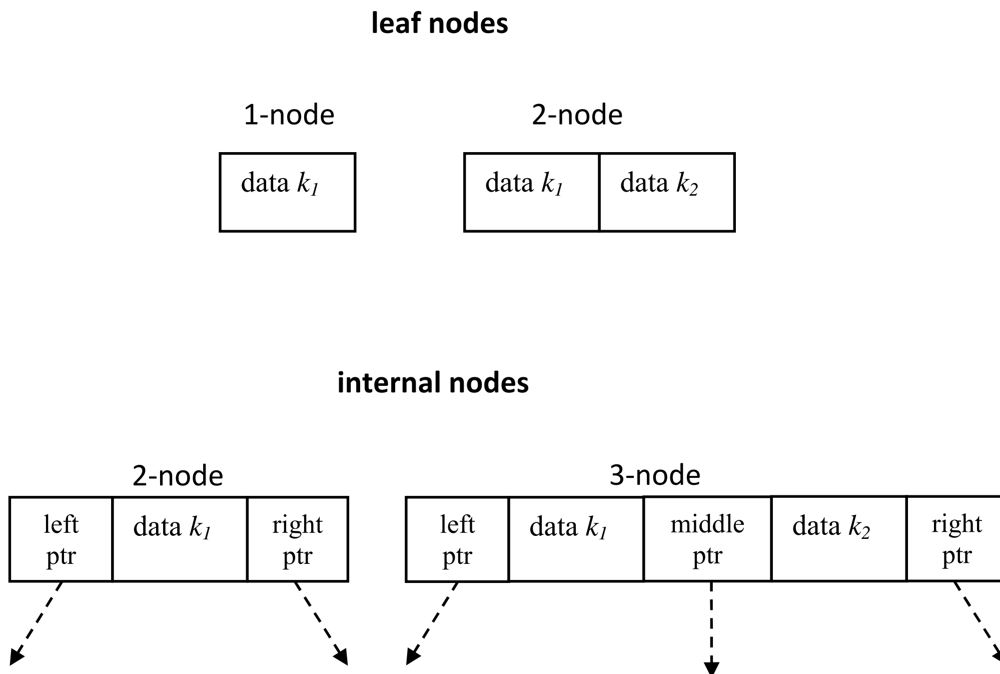

Figure 2-6 2-3 tree nodes

A 2-3 tree is always balanced; height of an n node tree can at most be log_2 (n+1). Disjoint and equal length prefix set is also a requirement for 2-3 tree representation. The constructed 2-3 tree for the example set in Table 2-1 and Table 2-2 is shown in Figure 2-7.

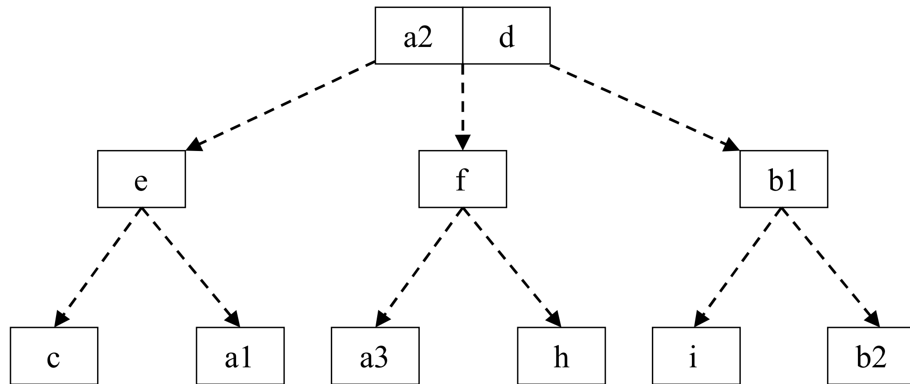

Figure 2-7 Example 2-3 tree

2.1.4.4 COMPARISON

Trie is more advantageous in IP lookup for two main reasons. Firstly, there is no need to store any data on the nodes, enabling one to get rid of processor overhead caused by comparison at each node. This overhead can be a critical issue for fully loaded CPUs. The second advantage is about computational complexity. The set of prefixes must be processed to construct tree structures. This necessity brings difficulty especially in update operations. On the other hand update for trie is as easy as traversing the trie. Any trie update can be done in two steps: find the target node and then insert or delete one node at the corresponding level. For insert operations on tree, however, the new prefix must be checked out if it disrupts the disjoint set property of prefixes by causing any overlap with the existing prefixes.

BST structure may need to be partially or fully reconstructed according to the place of update.

On the other hand, the tree structure (BST or 2-3 tree) is a more memory efficient data structure compared to trie. The depth of the trie depends on the maximum length of the prefixes. Especially for IPv6, longer paths in the trie decrease the lookup performance resulting in increased latency. But prefix length is not an issue for a tree.

Storage and update complexities of the three data structures discussed can be seen in Table 2-3. N indicates the number of nodes and W indicates the maximum length of prefixes.

Table 2-3 Complexity comparisons

	storage complexity	**worst case lookup complexity**	**average case lookup complexity**
Trie	$O(NW)$	$O(W)$	$O(W)$
BST	$O(N)$	$O(N)$	$O(log_2 N)$
2-3 tree	$O(N)$	$O(log_2 N)$	$O(log_2 N)$

2.2 IP LOOKUP ALGORITHMS FOR VIRTUAL ROUTERS

Virtual routers have attracted interest in recent years. But mainly two different approaches are followed in the related area, i.e., building separate data structures for each FIB or building a single structure by merging all FIBs. In the following section, a brief overview is given for the most notable data structures and lookup algorithms proposed for virtual routers using the merging approach.

2.2.1 EFFICIENT IP-ADDRESS LOOKUP WITH A SHARED FORWARDING TABLE FOR MULTIPLE VIRTUAL ROUTERS

A single data structure is built for multiple FIBs in [1]. The proposed structure aims to take advantage of common prefixes among FIBs. The goal is to use a single memory access for each node during the trie traversal.

First, merging multiple FIBs directly into a single trie approach is evaluated. This type of trie node requires fields for child pointers and next hop information for each router. Number of cells for next hop information equals the number of merged virtual routers. If the prefix denoted by a node is valid for a virtual router, its corresponding cell includes the next hop address. Different virtual routers can use different next hop addresses for the same prefix value. The node structure of such a trie, which is constructed by merging *n* virtual routers, is given in Figure 2-8.

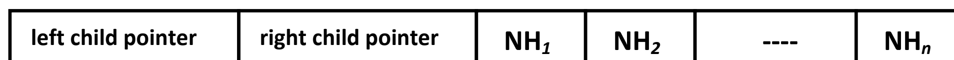

Figure 2-8 Trie node structure in direct merging

Limitations of this structure are handled in [1]. As the number of virtual routers increases, the node size also increases.

[1] aims to remove the necessity of storing next hop pointer in each node. The proposed solution is to turn the LPM into an exact match search so that next hop information need not be kept at every node except matching ones. Prefix set is therefore transformed into a "*common prefix set*" to provide a suitable set for exact match. This transformation requires making the prefix set both complete and disjoint. Complete prefix set means the whole address range is covered so that no packets can get out without a match. A default prefix is placed at the root of the trie, which has the value (*). Even if a packet with an IP address outside the prefix set arrives, it matches with this dummy prefix and then it will be dropped. Disjoint prefix set means that the address ranges defined by all prefixes are disjoint. To make a prefix set disjoint, child pushing operation should be applied. If the prefixes are kept only in leaf nodes, next hop pointer field is necessary only for the leaf nodes. Since intermediate nodes do not keep prefixes, a child pointer is placed instead of next hop pointer.

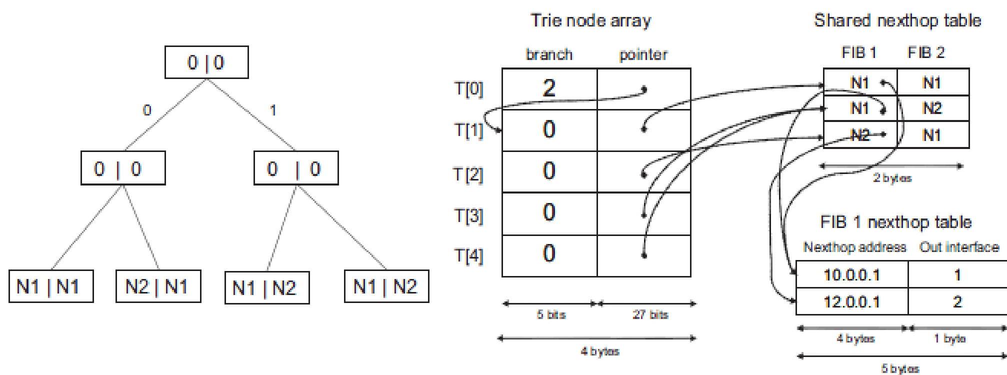

Figure 2-9 Proposed lookup data structure in [1]

The data structure can be constructed after the common prefix set is obtained. The proposed data structure has three parts: *trie node array, shared next hop table, virtual router's next hop tables*. The structure for an example leaf pushed trie can be seen in Figure 2-9. Each matching node in the merged trie is represented in the trie node array as a cell. The nodes at the same level are stored contiguously. Trie node array cell contains the branch factor and a pointer. Root node is stored at the first cell of the array, hence T[0]. Branch factor indicates the number of descendants of the root node. Root's pointer shows the location of the leftmost child among the first descendants of root. Leaf nodes' branch values are 0 and their pointer points to the shared next hop table. In the shared next hop table, each element contains pointers to FIBs individual next hop tables. A FIB next hop table is also an array and contains the next hop address and the corresponding output port.

When an IP search request arrives, branch factor amount of leftmost bits are extracted from the address. Search starts from the root T[0]. The pointer of T[0] is followed and leftmost child is reached. Once the leftmost child is found; target node can be derived by adding extracted bits from the address to the leftmost child address value. Search continues until a leaf node is reached, then by following leaf node's pointer, shared next hop table is reached. For each entry in the shared next hop table, a next hop address and an output port is defined as in individual FIBs next hop table.

2.2.2 A TRIE MERGING APPROACH WITH INCREMENTAL UPDATES FOR VIRTUAL ROUTERS

In [2], an efficient memory usage and fast incremental update are aimed at the same time. The key limitation of the shared FIB approach in [1] is criticized by this study. Storing the next hop pointer information for every router directly in the

nodes enlarges node size enormously. On the other hand, using child pushing to get rid of next hop storage requirement brings increased update overhead.

In [2], a sizeable memory reduction is aimed without sacrificing fast update performance. A single pointer is used to address the next hop pointer array, instead of directly storing a next hop pointer array into the node structure. Also, a prefix bitmap is used to indicate valid prefixes in a node. The proposed structure can be seen in Figure 2-10. There is also a next hop table for each FIB separately.

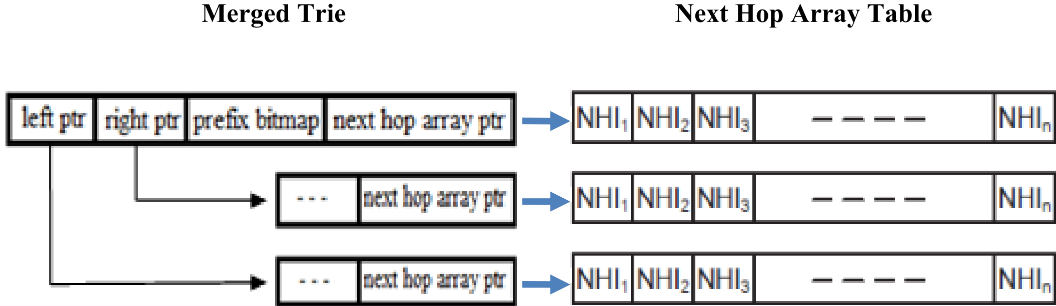

Figure 2-10 Trie merging data structure

This structure shows better update performance with respect to [1]. Child pushing increases the number of nodes per prefix indication and also creates the necessity of reconstruction of the structure during the update operation. [2] takes advantage of not disrupting the trie structure and hence all kinds of updates can be done using one write bubble .

2.2.3 MEMORY-EFFICIENT AND SCALABLE VIRTUAL ROUTERS USING FPGA

Authors of [3] focuses on scalability and update issues and claim that the storage requirement of the proposed structure in [3] is not dependent on the number of supported virtual routers but it depends on the total number of prefixes in the merged set. Hence they propose to use 2-3 tree data structure instead to provide both memory reduction and incremental update simultaneously.

However, tree based structures require non-overlapping prefix sets because of the comparison task needed during search. Hence they choose the child pushing technique again to obtain a disjoint prefix set. Child pushing creates new nodes to push all prefix nodes into leaf nodes. As a result, tree is expanded with a coefficient of approximately 1.7 as was reported previously in [4]. Authors of [3] then propose an algorithm called *"set bounded leaf pushing"* is to decrease this coefficient. The whole data set is first divided into two groups such that prefixes in each group are disjoint. Then these two groups of prefixes are kept in two 2-3 trees separately.

The stages of this algorithm is illustrated in Figure 2-11. First, all of the prefixes from each virtual router FIB are prepended with the corresponding virtual router's ID. An ordinary trie is built by these prepended prefixes (i). First step of algorithm gathers leaf prefixes and stores them into the first 2-3 tree structure. The remaing trie with redundant nodes (ii) is trimmed afterwards (iii). Leaf pushing is applied to this smaller trie (iv) and the leaf nodes are again moved to another 2-3 structure. At the end of the *"set bounded leaf pushing"* algorithm, two 2-3 trees are formed.

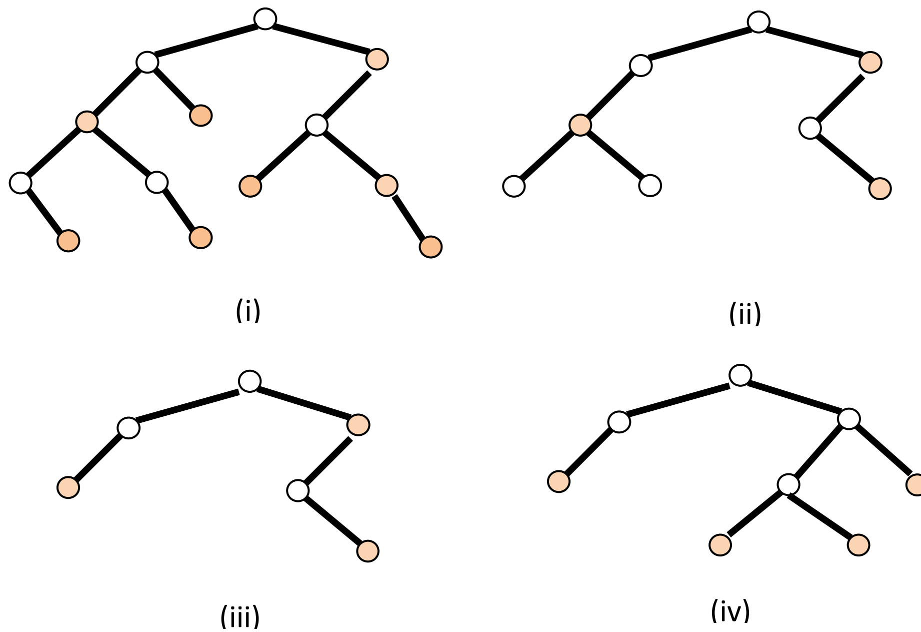

Figure 2-11 Stages of the set bounded leaf pushing algorithm

The search is should be performed in parallel in both trees during IP lookup. If there are match results from both of these parallel tree searches, the first tree has priority since it includes original leaf nodes indicating a longer prefix match.

2.2.4 EFFICIENT TRIE BRAIDING IN SCALABLE VIRTUAL ROUTERS

The study in [5] mainly emphasizes the limitations of simple merging technique, which has been proposed in [1]. Simple merging algorithm takes the advantage of common prefixes among different virtual router tables. As a result, it may not be suitable for virtual tables that does not share much prefixes. The proposed algorithm

in [5], which is called *"trie braiding"*, aims to construct a compact trie structure in case there exists a large diversity among the FIBs to be merged.

The main idea in *trie braiding* algorithm is to modify the shape of the trie to maximize node sharing. A *braiding* bit is stored at every trie node for each virtual router. The value of the braiding bit is decided according to overlapping shape of the node.

Trie braiding algorithm uses graph theory. During the merging operation, one node is mapped onto another node. There are two options for mapping a node's children; directly or after swapping. A cost function is used and the cost of each mapping option is calculated and the minimum cost mapping is chosen. The mapping cost is calculated by starting from the leaves and going up the root. Basically, if a node does not exist in the place where we want to map the other router's node but the sibling node is standing than we swap the place of the merged node. Braiding bit is set to 1 if swapping occurs or set to 0 otherwise.

The theory can be demonstrated by merging two tables. Two example virtual router tries are given in Figure 2-12(a) and the resulting merged scheme is given in Figure 2-12(b). Lookup operation starts from the root node, at each node the corresponding braiding bit is compared with the address bit sequentially. If they are equal, search goes on by following the left pointer, if not it continues with the right pointer.

Running time of this algorithm grows exponentially with the depth of the merged tries. It is deemed to be not practical to be used although it presents a substantial memory reduction.

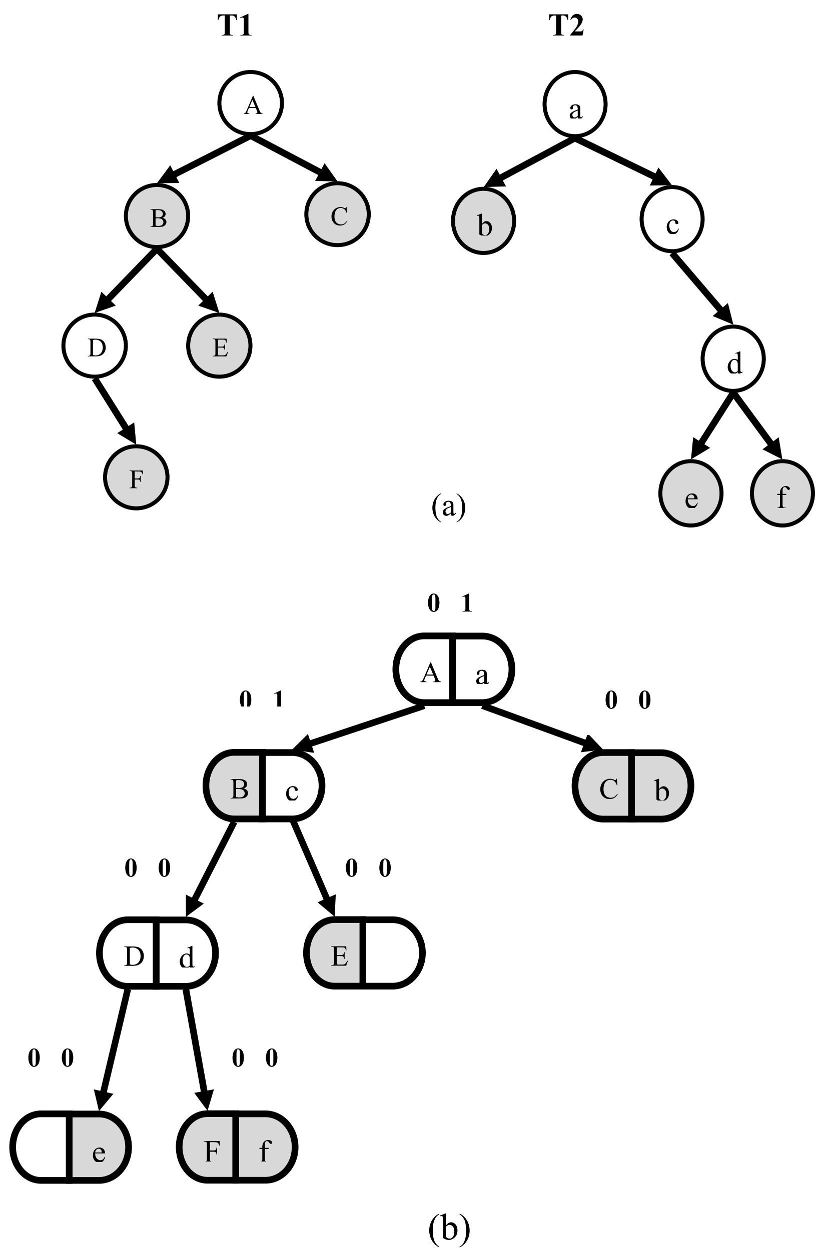

Figure 2-12 Trie braiding example

CHAPTER 3

TRIE PRUNING AND TRIE-TREE DATA STRUCTURE FOR IP LOOKUP

Trie is a common data structure used to represent FIBs. Some of the trie nodes are used to store valid prefixes for lookup whereas some of the nodes are only used as intermediate nodes to reach valid prefix nodes. This type of nodes is termed as empty or non-prefix node. On the other hand, a node that represents a valid prefix and that stores next hop information is termed as full or prefix node. Prefix lengths are not evenly distributed in lookup tables. 24 bit length prefixes dominate IPv4 prefix tables [13]. It is predictable that, congestion of prefixes at certain levels can create ‘one child paths’. One child path is actually a subtrie that keeps a chain of empty nodes to reach a full node. These kind of long paths cause memory inefficiency and an undesirably long latency. Especially for IPv6 FIBs, sparsely distributed prefixes on leafs cause a large number of ‘one child paths’. There are several methods to decrease the search depth such as level compression, path compression and multi-bit trie [10, 11, 19]. However, these methods require extra information to keep track of eliminated nodes and increase computational complexity also ending up with more complex updates.

The present chapter presents our data analysis technique, pruning strategy and representation of the resulting merged table as well as its corresponding lookup and update scheme.

3.1 DATA ANALYSIS

In our experiment, we simply merge virtual router FIBs into a single trie first. In this merged trie, each node has a bitmap array. The size of this bitmap is equal to

the number of FIBs merged, i.e., number of supported router instances. Each cell in this array belongs to a virtual router and indicates whether the node is full or empty for that router. When a packet arrives, the first task is to determine, which router this packet belongs to. A data packet carries an ID in its header part and thereby the virtual router can easily identify which router this packet belongs to. Packet's destination address is then searched on the dedicated areas of the merged trie.

Every node in a trie is actually the root node of a subtrie. We designate a pair of paramaters, *(a,b)*, for the root of each subtrie. *a* indicates the number of prefixes that a subtrie includes whereas *b* represents the number of nodes in the same subtrie. *(a,b)* values are determined for each node by traversing the trie in post order. Table 3-1 shows a set of four virtual router FIBs. The merged trie for this set with labels *(a,b)* can be seen in Figure 3-1.

Table 3-1 Example FIBs belongs four virtual routers

	virtual router							
	rrc0 ID=00		**rrc1 ID=01**		**rrc2 ID=10**		**rrc3 ID=11**	
prefixes	name	value	name	Value	name	value	name	value
	P_{11}	00*	P_{21}	0010*	P_{31}	0110*	P_{41}	110011*
	P_{12}	101*	P_{22}	1001*	P_{32}	10010*	P_{42}	1001*
	P_{13}	1001*	P_{23}	0110*	P_{33}	0101101*	P_{43}	1100*
			P_{24}	10010*				
			P_{25}	0101101*				

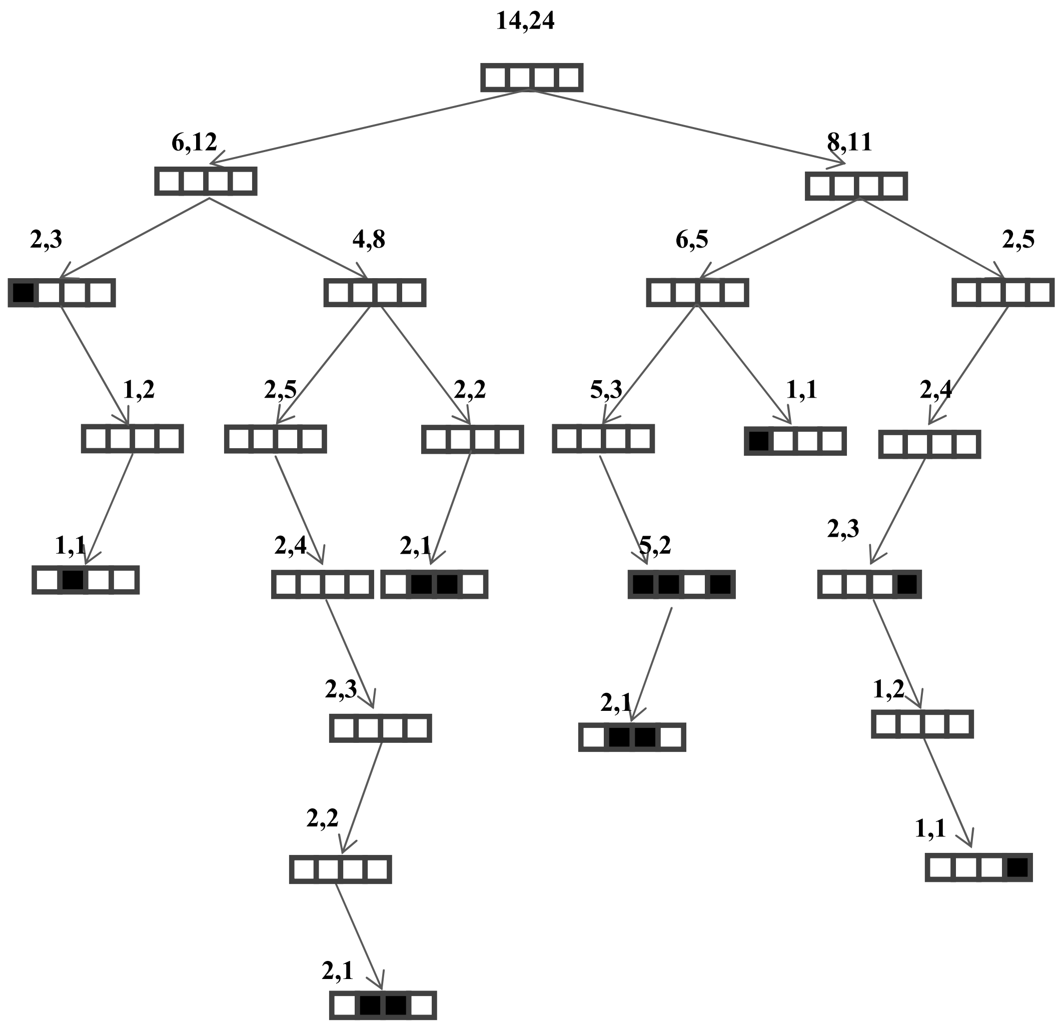

Figure 3-1 Example trie with *(a,b)* parameters

3.2 TRUNCATION ALGORITHM

We scan the simply merged trie by considering *(x,y)* limits where x is the upper bound for the number of prefixes and y is the lower bound for the number of nodes in a subtrie. We truncate a subtrie if and only if a and b satisfy the inequalities $a \leq x$ and $b \geq y$. Following this truncation, remaining trie is traversed once more to detect paths that do not include a full node. Such redundant empty nodes are also trimmed from the trie. Truncation algorithm flowchart is given in Figure 3-2.

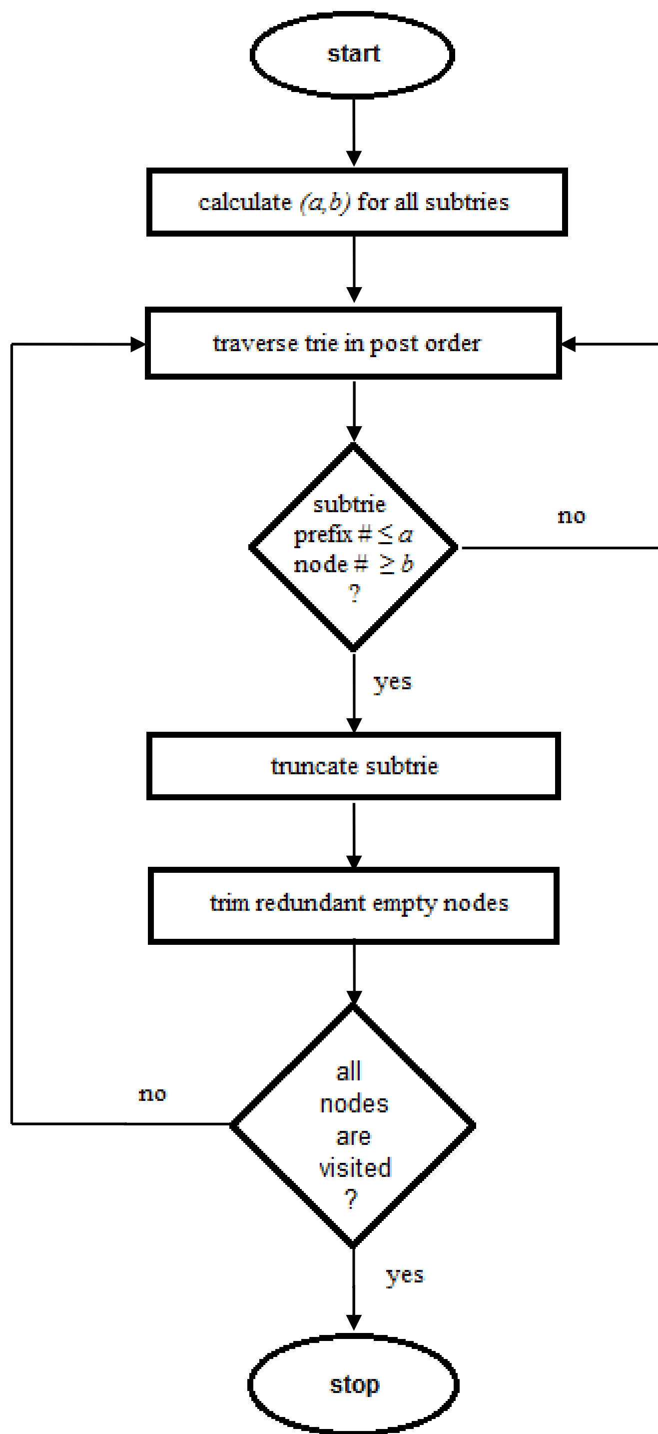

Figure 3-2 Truncation algorithm flowchart

If truncation parameters are chosen as (x,y)=(2,4) for the example in Figure 3-1, the truncated trie in Figure 3-3 is obtained.

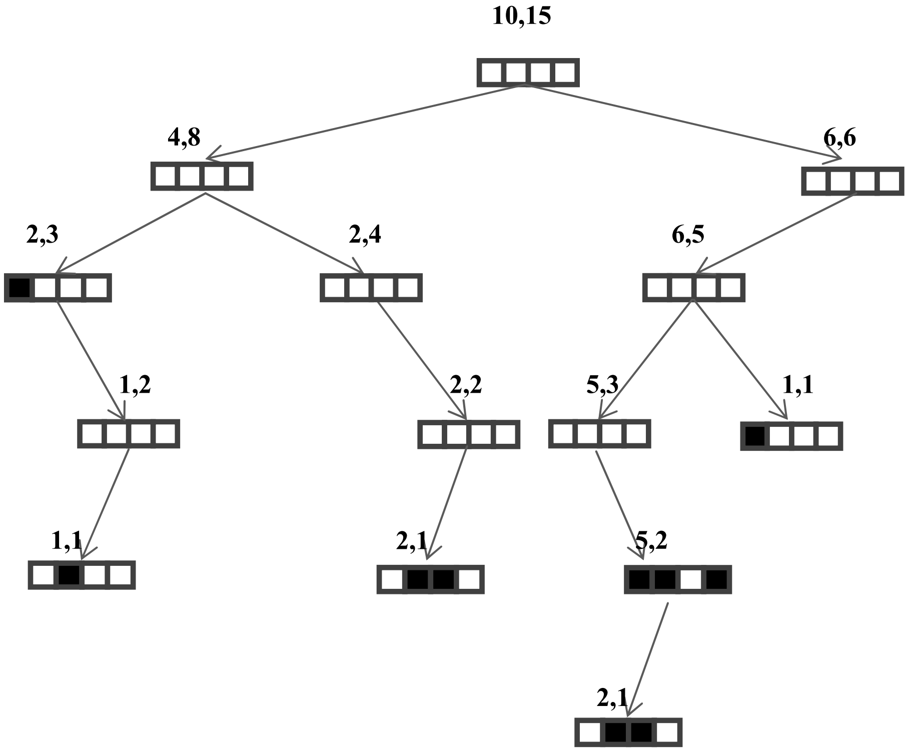

Figure 3-3 Truncated trie

Excluded prefixes should then be moved to an auxiliary data structure. A tree structure can be regarded as advantageous for small data sets because of small sorting overhead. In this case the following are necessary to follow:

Lookup in a tree structure is based on comparison of the key value with the node values and hence node values must have two specialties: i) they must be disjoint ii) they must have the same length. The same prefix value can be placed in more than one router FIBs. All virtual routers have an ID to distinguish the packets. All of the prefixes are prepended with the virtual router ID which they belong to. A trie is constructed from ID prepended truncated prefixes. If there is an overlap between prefixes, it can be eliminated by using leaf pushing method where all the prefixes in non-leaf nodes are pushed to leaf nodes. Then, all the prefixes in leaf nodes are collected. Let L_m denote the maximum length for prefixes. Each prefix should then be padded with 0s until its length is equal to L_m.

In Table 3-2, truncated prefixes are processed to be placed in a tree structure. L_m for this set of prefixes is found to be 8.

Table 3-2 Truncated prefixes

prefix name	original prefix	virtual ID prepended	leaf pushed prefix	0's padded
P_{41}	110011*	11110011	11110011*	111100110
P_{25}	0101101*	010101101	010101101*	010101101
P_{34}	0101101*	100101101	100101101*	100101101
$P_{43\text{-}a}$	1100*	111100	11110010*	111100100
$P_{43\text{-}b}$			1111000*	111100000

The 2-3 tree constructed from using the prefix set in Table 3-2, is shown in Figure 3-4.

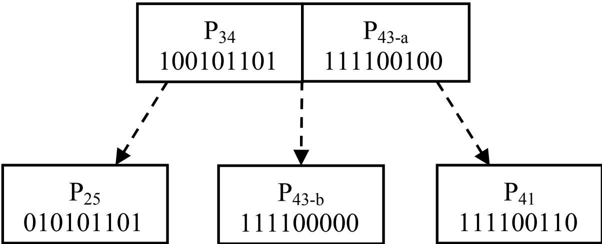

Figure 3-4 2-3 tree composed of truncated prefixes

3.3 DATA STRUCTURE DETAILS

3.3.1 TRIE

A trie node is composed of fields for child pointers and next hop information. For a single FIB, trie is an advantageous data structure in terms of node size. When we construct a single trie for several FIBs, each router's next hop information must be placed separately in the node. Next hop information defines an outgoing port of the router. For example, a router supporting 64 output ports should keep 6 bits to maintain this information. If *n* virtual routers run on this router, 6*n* bits are needed to store next hop information in each trie node. This increase in node size due to the

number of supported virtual routers, although linear, brings a scalability challenge. In the previous studies, for example [1], leaf pushing method is used to get rid of the next hop information in internal nodes. We do not prefer this method in our study because of its update overhead. The next hop field is not accessed unless a match occurs with the node, so its usage frequency is not the same as node access frequency. For this reason, we believe that next hop information can be kept in a separate storage space other than trie storage space. The most suitable trie structure for our study is demonstrated by [2]. Instead of storing all routers' next hop information directly in the node, a pointer is kept in each node to show the location of next hop array of all routers. There is a prefix bitmap in each node to understand whether the node is empty or full for a specified virtual router. Trie bitmap size equals n bits where n is the number of supported virtual routers. This trie structure can be seen in Figure 3-5. If any of the bits in prefix bitmap is equal to one, in other words, if the node is full for any virtual router, next hop array pointer reaches an array cell in next hop array table. Next hop array table has at most m entries, m being the number of full nodes in the simply merged trie.

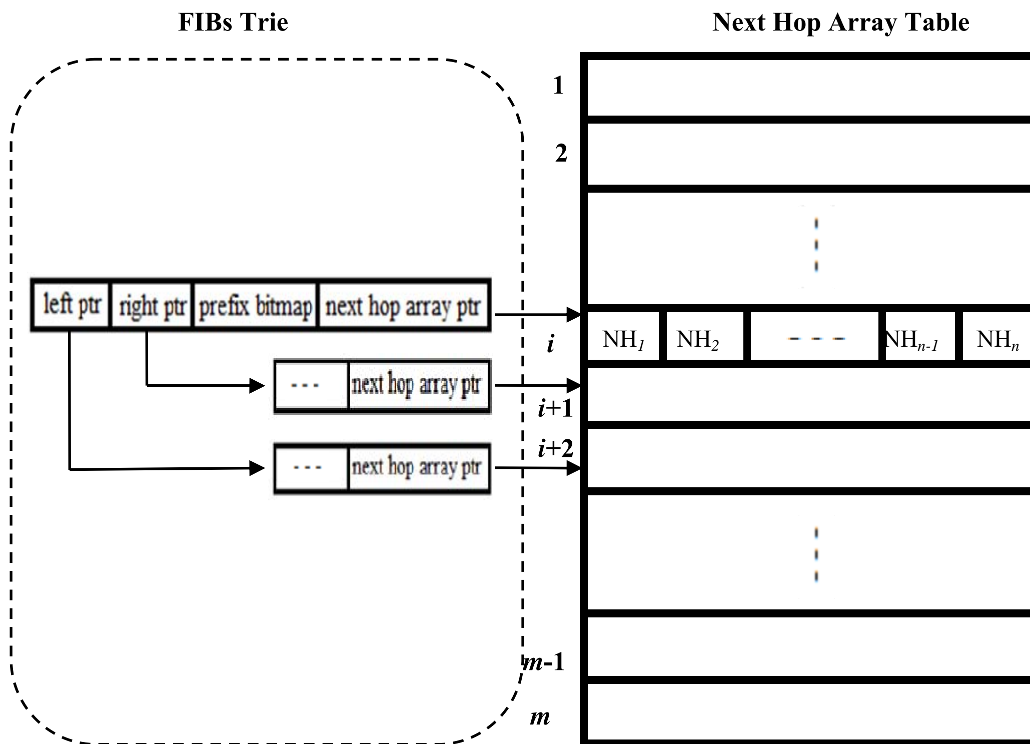

Figure 3-5 Trie structure with next hop

3.3.2 TREE

The truncated prefix set is kept in a tree structure. There are two tree type candidates: binary search tree (BST) and 2-3 tree. A comparison of these two structures is necessary.

Properties of BST

- Each node keeps a key and two child pointers.

- The values in the left subtree is smaller than key and the values in the right subtree is greater than key.

- Keys should be ordered to build a BST.

- Even though BST is constructed as a balanced tree it is not guaranteed to stay balanced after several insertions and deletions.

- Average case lookup time complexity is $O(\log N)$ however it is $O(N)$ for the worst case, where N denotes the number of nodes.

- Hardware memory management is easy, thanks to equal size nodes.

- Child pointers can be eliminated by saving same level nodes in a contiguous manner.

- Update operations require partial reconstructions.

Properties 2-3 tree

- There are two kinds of internal nodes: 2-node or 3-node and two kinds of leaf nodes 1-node and 2-node

- 2-node includes a key and two pointers while 3-node includes 2 keys and three pointers. The values in the left subtree is smaller than key and the values in the right subtree is greater than key in 2-node. For 3-node, left subtree contains less values than first key, middle subtree contains values between first and second key and right subtree contains values greater than second key.

- Leaf 1-node has one key and leaf 2-node has two keys.

- Leafs are always at the lowest level.

- 2-3 tree is always balanced.

- Both average and worst case lookup time complexity is O(*log N*), where *N* is the number of nodes.
- Keys need not be ordered before they are placed into the tree.
- 2-3 tree has difficulty in hardware management because of variable size node structure.
- All kinds of updates can be performed by using a single write bubble.

Although BST presents a memory efficient scheme with respect to 2-3 tree, we prefer 2-3 tree because its update is easier. 2-3 tree doesn't require any reconstruction due to updates and addition or deletion of a prefix is achieved by using a single write bubble in the pipelined memory. Memory size difference between two trees becomes insignificant for us because memory requirement for the storage of the trie will be the larger in our structure anyway.

3.4 IP LOOKUP

We are proposing a pipelined structure for both trie and 2-3 tree schemes. Pipelining is to store each level of tree in a separate memory block [15]. During the lookup operation, only one node is accessed in each level at every clock. If we store the whole tree structure in one memory block, to start a new search, we have to wait for the completion of the ongoing search. Several clock cycles is required to perform one lookup operation. In pipelining, one node in each level can be accessed simultaneously. One search behind another can be performed in parallel. Number of pipeline stages is equal to the number of tree levels. The maximum stage size is determined by the most crowded level of the tree. The result in stage i is transmitted to stage $i+1$. The matches from each pipeline stage are collected and longest prefix match is selected as the final result. Pipeline structure is depicted in Figure 3-6.

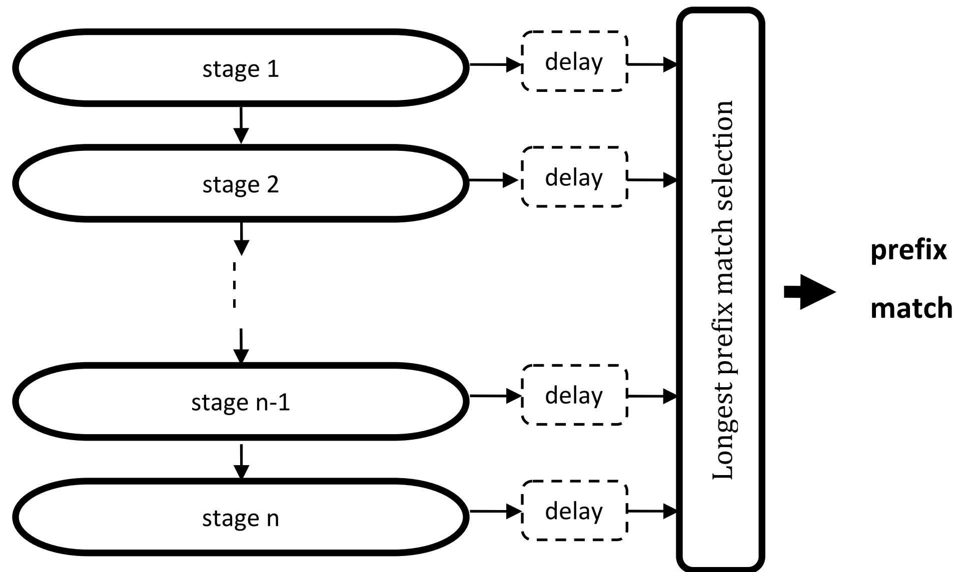

Figure 3-6 Pipeline structure

When a packet arrives, it is destination IP address (key) is searched directly on the trie and the prepended IP address with the corresponding router ID is searched on in 2-3 tree simultaneously. Latency of the trie memory block depends on the maximum length of a prefix, 32 bit for IPv4 and 64 bit for IPv6. Latency of the tree memory block is logaritmically proportional to the number of entries in tree, hence tree memory block will finish lookup before trie anyway. A delay block should be employed at the end of tree to accomplish the synchronization between both search. If a match occurs in both trie and 2-3 tree, tree structure has priority because it keeps the prefixes that are normally placed in higher level nodes before the truncation of the trie. Throughput depends on the access time to memory. If node size exceeds the SRAM entry size more than one memory access are required to

obtain contents of several entries. If a node fits into a single SRAM entry, single memory access is enough for each node in each level. Our pipelined trie memory block guarantees one lookup per clock cycle theoretically [2]. For IPv6 tables, tree node size is larger than trie node size, hence degradation in throughput with respect to simple trie implementations is expected. On the other hand , it is reported in [3] that a 2-3 tree based solution achieves one lookup per clock cycle by using dual port SRAMs. Consequently, our solution can support a comparable throughput among competitive solutions.

3.5 UPDATE

In the present work, we address not only memory utilization but also update performance. There are three types of update operations in IP networks: announcement, withdraw, next hop change. A new prefix is added to FIBs in announcement and an existing prefix is deleted from FIB in withdraw. These two operations require addition or deletion of nodes, but next hop change does not affect the data structure but affecting only the next hop array table. The changes in our data structure are computed offline by using a shadow data structure, which is in an initial state. New contents are then written in write bubble tables (WBT), which are placed at every pipeline stage. A pipeline stage and WBT is illustrated in Figure 3-7. A write bubble table has three fields: i) target to be updated (node address) ii) new content of the node to be updated iii) write enable bit. Update operations are performed by using write bubbles in both of the pipelined structures. Write bubbles are inserted into the pipelines similar to looking up task. When a write bubble is injected into pipeline, lookup up operation is blocked for one clock cycle. Write bubble is more like a trigger agent than being an information carrier. Every write bubble has an ID. When it reaches the stage prior to the stage to be updated, it checks the WBT by using its ID and triggers the write enable bit of the

matched line of WBT. The content of that WBT line then passes through the pipeline memory. Thanks to dual port SRAMs, a write bubble can change at most two node contents at each stage. Update overhead is measured by the number of write bubbles.

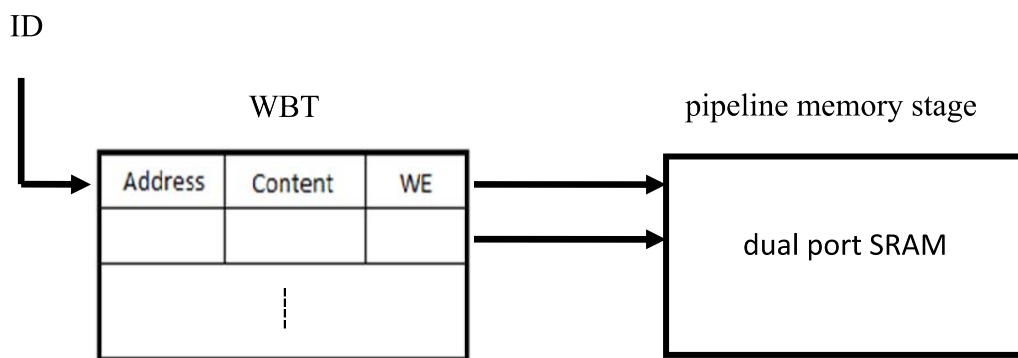

Figure 3-7 A pipeline stage with WBT

Both trie and tree data structures support incremental update. A prefix update affects single node at each level of the trie [15] and at most two nodes at each level of the 2-3 tree structure [14]. Any kind of update is therefore achieved in both trie and 2-3 tree by using a single write bubble.

In this section, we propose an update strategy for different kinds of updates:

Next hop information update: the prefix is either in the trie or the 2-3 tree. If it is in the trie, just change the content of the corresponding next hop array table. If it is in the tree, change the corresponding node content effortlessly.

Prefix announcement: Insert all new announced prefixes in the trie structure. The main aim here is to benefit from the advantages of trie for update operations. Trie does not require any precalculation to insert a new prefix, all we have to do is to traverse the trie and find the appropriate place to create the new node. By creating at most one node at each level, any prefix addition can be done using a single write bubble.

Prefix withdrawal: For the withdrawed prefixes there are two possibilities. If it is in the trie, again search for its place. Delete the corresponding full node and redundant empty nodes if there exist any. Because only a single node is effected at each level, these updates can be performed using a single write bubble. If the prefix to be deleted is in the tree in a single node, the update cost is again a single write bubble. However, if it is placed in more than one tree node due to to leaf pushing, this will cost more than one write bubble to update.

CHAPTER 4

REAL LIFE LOOKUP TABLE ANALYSIS AND DETAILED DATA STRUCTURE DESIGN

In this chapter, results of an extensive analysis study, which use real life router table instances, are reported. Based on this analysis, the size of data structure proposed is determined to make it possible to compare our proposal with the existing approaches.

In our analysis study, we used four different types of router FIBs; IPv4 edge, IPv4 core, IPv6 edge and IPv6 core routers. IPv4 core router tables are collected from Ripe Network Coordination Center [20]. IPv4 edge router tables are obtained by using a synthetic generator tool [21]. IPv6 tables are generated by using the method in [22]. Ten selected FIBs are merged for each type of router. Edge and core router FIBs has different characteristics. In our real experimental data sets, each edge router has 100K and each core router has 320K prefixes. Number of overlapping prefixes is higher in IPv4 core routers in comparison to other types. Besides, number of common prefixes among FIBs is higher in IPv4 core routers with respect to IPv4 edge routers. Although the same tendency is valid for IPv6, the difference between edge and core routers is not as visible as IPv4. Number of prefixes in each FIB used is given in Table 4-1. The number of nodes in the merged trie, the total number of prefixes in all merged FIBs and the total number of full nodes are given in Table 4-2.

Table 4-1 Prefix numbers in the merged FIBs

table	# of prefixes			
	IPv4		IPv6	
	CORE	EDGE	CORE	EDGE
rrc1	332,117	95,043	332,034	100,000
rrc2	324,172	98,390	324,131	100,000
rrc3	372,743	94,780	321,611	100,000
rrc4	321,617	90,867	347,145	100,000
rrc5	347,231	95,358	322,996	100,000
rrc6	322,996	98,305	319,913	100,000
rrc7	321,577	97,410	323,608	100,000
rrc8	322,557	95,007	320,015	100,000
rrc9	325,797	97,394	323,986	100,000
rrc10	323,986	97,426	328,295	100,000

Table 4-2 Number of prefixes and nodes in the merged trie

	merged trie		
	full nodes #	total prefix #	total node #
IPv4 CORE	425,177	3,214,793	1,032,497
IPv4 EDGE	930,660	959,990	3,323,861
IPv6 CORE	3,258,914	3,263,734	62,295,196
IPv6 EDGE	999,999	999,999	22,184,609

The simply merged trie is scanned for a range of (*x,y*) limits. For each (*x,y*) pair, truncation is performed. The number of nodes left in the trie and the number of truncated prefixes are recorded. The results of the truncation algorithm for a set of (x,y) pairs are given in Appendicies for each kind of router.

Although tree is a memory efficient structure for FIBs, its complexity is its main drawback. Latency is proportional to *log N* where *N* is the number of nodes in the tree. Our main aim is to provide memory saving without sacrifising lookup performance and we don't want to grow the tree too much to take advantage of the trie during the lookup operation.

Based on our experimental observations, we prefer to fix the ratio of the number of truncated prefixes to all prefixes at around 30% except IPv4 core case. IPv4 core routers use common prefixes substantially (Table 4-2) and approximately half of the nodes are already full, which indicates a very compact data structure. As a result, inefficient node usage ratio is very low and just around 1% of the prefixes can be truncated from the trie in this case.

Other types of routers are similar to each other in terms of prefix distribution and common prefix usage behaviours. The chosen truncation limits, the number of nodes left in the trie and number of prefixes truncated and moved to tree is given in where

N(Ti) total number of nodes in truncated trie Ti

N(Ts) total number of nodes in simply merged trie

Pt number of truncated prefixes

P total number of prefixes in a set of FIBs

'node%' and 'prefix%' are calculated using formulas (1) and (2), respectively.

$$node\% = \frac{\mathrm{N(Ti)}}{\mathrm{N(Ts)}} * 100 \quad (1) \qquad\qquad prefix\% = \frac{\mathrm{Pt}}{\mathrm{P}} * 100 \quad (2)$$

Table 4-3 Truncation criteria and results

	truncation criteria		truncation results			
	X upper bound for # of prefixes	*y* lower bound for # of nodes	# of nodes in the truncated trie, $N(T_i)$	# of truncated prefixes, P_t	node %	prefix %
IPv4 CORE	5	8	953,599	25,469	92	1
IPv4 EDGE	2	7	2,079,068	263,355	63	27
IPv6 CORE	1	21	39,245,265	997,284	63	31
IPv6 EDGE	1	24	13,085,374	354,113	59	35

4.1 MEMORY SIZE CALCULATION

In the present section, we detail the data structure and compute the required memory size. List of notations used for these calculations is given in Table 4-4.

Table 4-4 List of used notations

notation	Description
$N(Tx)$	total number of nodes in tree T*x*
$E(Tx)$	total number of empty nodes in tree Tx
$F(Tx)$	total number of full nodes in tree Tx
P	total number of prefixes in a set of FIBs
Pt	number of truncated prefixes
$ptr(Tx)$	number of child pointer bits for tree T*x*
H(Tx)	number of next hop information bits for tree Tx
$C(Tx)$	number of nodes in the most crowded level of tree T*x*
Ts	simply merged trie
Ti	truncated trie
Te	2-3 tree
VID	virtual router ID number
L	prefix length

4.1.1 TRIE NODE SIZE

Every level of the trie is kept in a separate memory block. A trie node includes four fields: left and right child pointers, next hop array table pointer and prefix bitmap. The size of each field can be calculated as follows:

Child pointer bits: Same level nodes share the same memory block. Hence the capacity of the memory blocks is determined by the most crowded level in the trie. Address bits are assigned to support maximum number of nodes in a level plus 10% extra space for additional nodes that will be required in update operations. The number child pointer bits is therefore

$$\mathrm{ptr}(\mathrm{T}i) = \lceil \log_2(1.1(\mathrm{C}(\mathrm{T}i))) \rceil \quad (3)$$

Next hop array pointer bits: This is calculated due to an array cell exists for each full node in truncated trie. Although there is a possibility that more than one node may point to the same next hop array cell, we assume that each node points to a different cell in the worst case. Therefore the number of next hop array pointer bits should be

$$\mathrm{H}(\mathrm{T}i) = \lceil \log_2(F(\mathrm{T}i)) \rceil \quad (4)$$

Router bitmap: The number of bits in the bitmap array is equal to the number of supported virtual routers, which is 10 in our study.

Using the above, trie node size is found as in (5).

$$\mathrm{T}i \text{ node size} = 2 * \mathrm{ptr}(\mathrm{T}i) + \mathrm{H}(\mathrm{T}i) + \text{Router bitmap} \quad (5)$$

Computed ptr(Ti) and H(Ti) values are given in Table 4-5 and Table 4-6 respectively for the truncated trie. Maximum node size calculation for experimental data sets is given in Table 4-7 according to the implementation of selected truncation metrics in Table 4-3.

Table 4-5 Size of child pointer field

	$C(Ti)$	node # lower bound	supported node #	$ptr(Ti)$
IPv4 CORE	212,317	233,549	262,144	18
IPv4 EDGE	348,949	383,844	524,288	19
IPv6 CORE	2,041,151	2,245,267	4,194,304	22
IPv6 EDGE	634,204	697,625	1,048,576	20

Table 4-6 Size of next hop array pointer field

	$F(Ti)$	lower bound array cell #	supported array cell #	$H(Ti)$
IPv4 CORE	401,106	441,216	524,288	19
IPv4 EDGE	667,341	734,076	1,048,576	20
IPv6 CORE	2,261,630	2,487,794	4,194,304	22
IPv6 EDGE	645,886	710,475	1,048,576	20

Table 4-7 Size of trie node

	ptr(T*i*)	H(T*i*)	router bitmap	total node size
IPv4 CORE	36	19	10	65
IPv4 EDGE	38	20	10	68
IPv6 CORE	44	22	10	76
IPv6 EDGE	40	20	10	70

4.1.2 TREE NODE SIZE

Leaf pushing is used to obtain disjoint prefixes for storing on the tree. Prefix number increase factor was referred as 1.6 in previous works [4] but in fact, this claim was based on IPv4 routers. In our experiments, it is revealed that the leaf pushing coefficient in IPv4 case is not 1.6 but rather relatively small. This is not surprising because we truncate the prefixes in the higher levels of the trie and by this means we eliminate one child paths mostly. We observed an increase coefficient of leaf pushing on truncated prefixes for is 1.07 and 1.06 for IPv4 core and edge routers, respectively. Also leaf node to total node ratio is %99.9 for IPv6 case. Hence almost all of the prefixes are disjoint in IPv6 tables and therefore leaf pushing operation doesn't change the number of truncated prefixes for IPv6 routers anyway.

Tree node includes four fields similar to the trie: child pointers, next hop information, prefix value and prefix length. The size of each field is calculated as follows:

Prefix value: Although a prefix may be used by several virtual routers, each such prefix is saved as separate nodes. A prefix value is prepended with the virtual router's ID which it belongs to. We merged 10 virtual routers hence each virtual router ID is represented by four bits. Maximum prefix size is 32 bits for IPv4 and 64 bits for IPv6. Although IPv6 address is 128 bit in length, only the first 64 bits are used to identify the network address, hence maximum prefix length is 64 bits.

Child pointer bits: In 2-3 tree for 2-nodes there are 1 key and 2 pointers, number of pointers for a key is 2 and for 3-nodes there are 2 keys and 3 pointers, number of pointers for a key is 1.5. In our memory calculations, we consider 2 pointers for a key not to underestimate the memory size.

Half of the tree nodes are placed in leafs. A 2-3tree is always balanced and the most populated level is the last one. Tree structure is pipelined meaning that nodes in the same level are to be saved in the same memory block. The child pointer bits can then be calculated as

$$\mathrm{ptr}(\mathrm{T}e) = \left\lceil \log_2 \left(\frac{N(Te)}{2} \right) \right\rceil \qquad (6)$$

An enhancement for total tree size calculation can be done by noting that 50% of the nodes are leaf nodes. Since leaf nodes do not need any child pointers, the child pointer bits can be excluded for half of the nodes. This means that we can count node size as if we use only one pointer for all nodes.

Next hop bits: Because we keep every prefix distinctly, next hop information should be saved in a node directly. 6 bits are adequate to support 64 output ports.

Prefix length: 0 bits are padded to prefix values to obtain same length prefixes for comparison but we need the original prefix lengths for longest prefix match. Prefix length can be represented by 5 bits for IPv4 and 6 bits for IPv6.

Child pointer size calculation is given in Table 4-8, 2-3 tree node size is found as in equation (7) and final results are given in Table 4-9.

$$\mathrm{T}e \text{ node size} = \mathrm{ptr}(\mathrm{T}e) + \mathrm{H}(\mathrm{T}e) + \text{Prefix value} + \text{prefix length} \quad (7)$$

Table 4-8 Tree child pointer size

	P_t	$N(\mathrm{T}e)$	$N(\mathrm{T}e)/2$	supported array cell #	$\mathrm{ptr}(\mathrm{T}e)$
IPv4 CORE	25,469	27,340	13,670	16,384	14
IPv4 EDGE	263,355	280,285	140,143	262,144	18
IPv6 CORE	997,284	997,284	498,642	524,288	19
IPv6 EDGE	354,113	354,113	177,057	262,144	18

Table 4-9 Tree node size

	$\mathrm{ptr}(\mathrm{T}e)$	$\mathrm{H}(\mathrm{T}e)$	prefix value	prefix length	total node size
IPv4 CORE	14	6	36	5	61
IPv4 EDGE	18	6	36	5	65
IPv6 CORE	19	6	68	6	99
IPv6 EDGE	18	6	68	6	98

4.1.3 TOTAL DATA STRUCTURE SIZE

The proposed lookup structure has three parts: trie-tree structure, next hop array table and next hop lists.

Trie and tree size is determined using the total number of nodes and the node size. Node sizes are calculated in sections 4.1.1 and 4.1.2. Total memory size is given in Table 4-10.

Table 4-10 Trie-Tree structure total memory size

	$N(\mathrm{T}i)$	trie node size (bit)	$N(\mathrm{T}e)$	tree node size (bit)	Total memory size (MB)
IPv4 CORE	953,599	65	27,340	61	7.95
IPv4 EDGE	2,079,068	68	280,285	65	19.94
IPv6 CORE	39,245,265	76	997,284	99	385.17
IPv6 EDGE	13,085,374	70	354,113	98	118.83

Although the next hop array table is only reached when a match occurs, it doesn't have to be stored on a memory as fast as trie structure's memory. Number of entries in the next hop array table can be at most $F(\mathrm{T}i)$ in the case that each full node in truncated trie points to distinct next hop information. Number of next hop pointer bits is 6 to support 64 output ports. An entry in the next hop array table has 10 cells because we merge ten virtual routers. Therefore the next hop array table size can be computed as 60 * $F(\mathrm{T}i)$ and the results are given in Table 4-11.

Table 4-11 Next hop array table size

router type	$F(\mathrm{T}i)$	next hop array table size (MB)
IPv4 CORE	401,106	3.01
IPv4 EDGE	667,341	5.01
IPv6 CORE	2,261,630	16.96
IPv6 EDGE	645,886	4.84

Memory is also required to store the complete next hop pointer tables that consist of the list of next hops corresponding to an output port number and next hop IP address. We consider a 64-output port router in our study, for this reason the size of this complete next hop table is negligible in our memory calculations.

CHAPTER 5

COMPARATIVE EVALUATION

Our proposed truncation approach and the accompanying structure evaluated and compared in terms of memory efficiency and quick update capability with the exiting solutions. The most competitive algorithms that have the best memory efficiency or the best update performance are chosen as the benchmark cases. In this chapter, the tradeoff between memory and update complexity is discussed also.

5.1 MEMORY SIZE COMPUTATION FOR THE EXISTING SOLUTIONS

Node size and overall size of the data structure in each competitive algorithm is computed below in the following subsections using our selected experimental data set given in Table 4-1 and Table 4-2.

5.1.1 SIMPLE MERGING WITHOUT LEAF PUSHING

This algorithm was mentioned in [1] as the starting point of all the discussion. The structure of the node was given in Figure 2-8. We recall that the number of child pointer bits is calculated by $\mathrm{ptr}(\mathrm{T}i) = \lceil \log_2(\mathrm{C}(\mathrm{T}i)) \rceil$. A 10% extra memory area for additional nodes for handling updates are also taken into account. Six bits are used for next hop information to support 64 output ports. Ten virtual routers are merged again. The node size and overall trie memory size results are given in Table 5-1.

Although this algorithm always supports single write bubble updates, its memory performance needs to be improved.

Table 5-1 Memory results for simple merge without leaf pushing algorithm

	trie node size calculation				total memory size calculation	
	$C(Ts)$	$ptr(Ts)$	$H(Ts)$	total node size (bits)	$N(Ts)$	total size (MB)
IPv4 CORE	222,888	36	60	96	1,032,497	12.39
IPv4 EDGE	584,987	40	60	100	3,323,861	41.55
IPv6 CORE	3,037,584	44	60	104	62,295,196	809.84
IPv6 EDGE	988,271	42	60	102	22,184,609	282.85

5.1.2 SIMPLE MERGE WITH LEAF PUSHING

This algorithm targets a compact data structure solution by taking advantage of common prefixes among different virtual router FIBs [1]. Child pushing method is employed to turn the longest prefix match into exact match. A processor is used for the experimental platform; hence they are able to reduce the number of pointers from two to one by using consecutive addresses for the same level nodes. Only one pointer is used in each node to indicate the next hop information or left

most descendant node. If this structure is implemented by using a hardware platform, both child pointers and next hop array pointer are needed. Next hop information is stored in a separate memory than trie memory and reached by using next hop pointer. Although this algorithm works well with similar FIB tables like in IPv4 core routers, child pushing brings an update overhead and also increases the overall memory.

The number of next hop pointer bits is calculated by the $\lceil \log_2(F(\mathrm{T}i)) \rceil$ formula and the number of child pointer bits is calculated by the $\lceil \log_2(1.1(\mathrm{C}(\mathrm{T}i))) \rceil$ formula. In [4], it was reported that child pushing increases the number of nodes in trie by 1.3 times and number of prefixes by 1.7 times. Our analysis however revealed that %99.9 of the full nodes are placed in leafs in IPv6 tables. Considering this situation, prefix number increase coefficient is used only for IPv4 calculations. The memory size results for trie node array are given in Table 5-2 and Table 5-3.

Table 5-2 Node size results for simple merge with leaf pushing algorithm

	trie node size calculation					
	N(T*s*)	F(T*s*)	C(T*s*)	H(T*s*)	ptr(Ts)	total node size (bits)
IPv4 CORE	1,342,247	722,800	289,755	20	38	58
IPv4 EDGE	4,321,020	1,582,122	760,484	21	40	61
IPv6 CORE	62,295,196	3,258,914	3,037,584	22	44	66
IPv6 EDGE	22,184,609	999,999	988,271	21	44	65

Table 5-3 Memory results for simple merge with leaf pushing algorithm

total memory size calculation		
total trie size (MB)	next hop array size (MB)	total size (MB)
9.73	5.42	15.15
32.95	11.87	44.81
513.94	24.44	538.38
180.25	7.50	187.75

Although this algorithm achieves a memory reduction in trie size with respect to merging without leaf pushing algorithm, total memory size is increased due to the leaf pushing for IPv4 tables. Moreover, its main drawback is its update complexity. The study in [2] shows that during a 12-hour update trace 0.003% of the total trace requires over 100 write bubbles per update. Updates that require a high number write bubbles seem to occur rarely but each write bubble blocks the lookup for one clock cycle. It has been reported that the worst case update operation requires 21.200 write bubbles. Such long term blockings may decrease the throughput dramatically. However the unmodified simply merged trie structure guarantees one write bubble usage per any kind of update.

5.1.1 TRIE MERGING WITH INCREMENTAL UPDATES

This structure proposed in [2] aims to decrease the memory size by not storing next hop info directly in the node on the other hand child pushing is not preferred due to its update complexity. Trie node includes data fields for left and right child pointers, prefix bitmap and next hop array table pointer.

Length of child pointer field is determined according to the most crowded level of simply merged trie and length of next hop array pointer field is determined by the number of full nodes. Size of the trie bitmap is equal to the number of merged FIBs, which in ten in our study. We recall from section 3.3.1 that field length for child pointer is calculated as $ptr(Ts) = \lceil \log_2(1.1 * C(Ts)) \rceil$ and field length for next hop array pointer is calculated as $H(Ts) = \lceil \log_2(F(Ts)) \rceil$. Trie node trie size is given in Table 5-4 and overall size is given in Table 5-5 for the trie merging with incremental updates method.

Table 5-4 Node size results for trie merging with incremental updates algorithm

	trie node size calculation					
	C(Ts)	ptr(Ts)	F(Ts)	H(Ts)	router bitmap	total node size (bits)
IPv4 CORE	222,888	36	425,177	19	10	65
IPv4 EDGE	584,987	40	930,660	21	10	71
IPv6 CORE	3,037,584	44	3,258,914	22	10	76
IPv6 EDGE	988,271	42	999,999	21	10	73

Table 5-5 Memory results for trie merging with incremental updates algorithm

total memory size calculation			
N(Ts)	trie size (MB)	next hop array size (MB)	total size (MB)
1,032,497	8.39	3.19	11.58
3,323,861	29.50	6.98	36.48
62,295,196	591.80	24.44	616.24
22,184,609	202.43	7.50	209.93

This algorithm represents the most effective solution in terms of total memory size in the group of solutions that guarantee a single write bubble per update.

5.1.2 SET BOUNDED LEAF PUSHING

This is a tree based solution, which supports quick updates by usıng 2-3 trees [3]. Tree based solutions requires leaf pushing to obtain disjoint prefixes. But leaf pushing increases the number of prefixes. By using two prefix sets, the extension coefficient of leaf pushing is decreased.

Two trees that encapsulate two distinct set of prefixes are indicated as T_1 and T_2, respectively. Overall memory size equals the sum of the sizes of T_1 and T_2. Each

node includes child pointers, prefix value, prefix length, virtual ID and next hop pointer fields. It is assumed that each node has two child pointers. The overall memory size is given by (8).

$$\sum_{k=1}^{2}[\mathrm{N}(\mathrm{T}k) * (\text{prefix value} + \mathrm{VID} + \mathrm{L} + 2\mathrm{C}(\mathrm{T}k) + \mathrm{H}(\mathrm{T}k))] \quad (8)$$

All formulas for tree node size calculation, which are given in section 4.1.2, are also valid for this algorithm. Experimental results for this study show that total number of prefixes is increased by a factor of 1,12 for IPv4 core, 1,17 for IPv4 edge and 1,01 for IPv6 edge routers [3]. We also know the leaf ratios in the simply merged trie. Although IPv6 core table has not been studied in this work, node distribution characteristics are almost the same in IPv6 for both core and edge routers. Number of nodes $N(T_1)$ and $N(T_2)$ for our data set is computed by using prefix increase factor and leaf ratio given in Table 5-6.

Table 5-6 Number of nodes in T_1 and T_2

	# of leaf nodes / $N(T_s)$	$N(T_1)+N(T_2)/P$	P	$N(T_1)$		$N(T_2)$	
IPv4 CORE	0.90	1.12	3,214,793	0.9P	2,893,314	0.22P	707,255
IPv4 EDGE	0.77	1.17	959,990	0.77P	739,192	0.4P	383,997
IPv6 CORE	0.99	1.01	3,263,734	0.99P	3,231,097	0.01P	32,637
IPv6 EDGE	0.99	1.01	999,999	0.99P	989,999	0.01P	10,000

Overall memory size for this structure is shown in Table 5-7.

Table 5-7 Memory results for set bounded leaf pushing algorithm

	trie node size calculation								total memory size calculation		
	prefix value	VID	L	C(Tk)		H(Tk)	total node size (bits)		$N(T_1)$	$N(T_2)$	total size (MB)
				T_1	T_2		T_1	T_2			
IPv4 CORE	32	4	5	22	20	6	69	67	2,893,314	707,255	30.88
IPv4 EDGE	32	4	5	20	18	6	67	65	739,192	383,997	9.31
IPv6 CORE	64	4	6	22	15	6	102	95	3,231,097	32,637	41.58
IPv6 EDGE	64	4	6	21	14	6	101	94	989,999	10,000	12.62

Set bounded leaf pushing algorithm achieves a good memory performance except for IPv4 core router. The worst memory performance in this case belongs to this algorithm, the reason being Algorithm 5 storing each prefix in all FIBs discretely even though some of them are common among different tables. This algorithm cannot benefit from the commonality of prefixes in FIBs. We recall from Table 4-2 that IPv4 core tables include mostly common prefixes.

2-3 tree structure doesn't require reconstruction for updates. 2-3 tree requires change in at most two nodes at the same level for both addition or deletion of a prefix. Changes in two nodes in the same level can be achieved with a single write bubble by using dual-port SRAM. On the other hand, although 2-3 tree structure supports quick updates, there is a problem caused by the algorithm itself. Firstly,

every update disrupts the division of the two sets. For each additional update, the decision of which set the prefix belongs to has to made again. Besides, updated prefix can cause a prefix to move from T_1 to T_2 or vice versa. Although the two tree structure operates is parallel, a single processor usually manages all of the above described operations. Several modifications per update create a serious work load. Secondly since leaf pushing is used, one prefix change may affect several nodes. In

Figure 5-1, such an example is illustrated. Scheme (i) shows a branch of the simply merged trie. Set bounded leaf pushing algorithm collects the leaf nodes from this branch, which results in scheme (ii). Remaining prefixes in the truncated branch is leaf pushed and put in T_2, which is shown in (iii). A possible update operation is shown in part (iv). Prefix *y* is added to FIB. The changes in nodes for each update are calculated by using a shadow simple merged trie. Added prefix *y* also requires leaf pushing, which means several node contents should be changed because of this change in one single prefix. This situation is illustrated in part (v). Therefore cases where several write bubbles are needed for a single prefix update are possible. The worst case write bubble study, which has been mentioned in [2], for simple trie merging with leaf pushing algorithm [1] is also valid for set bounded leaf pushing algorithm.

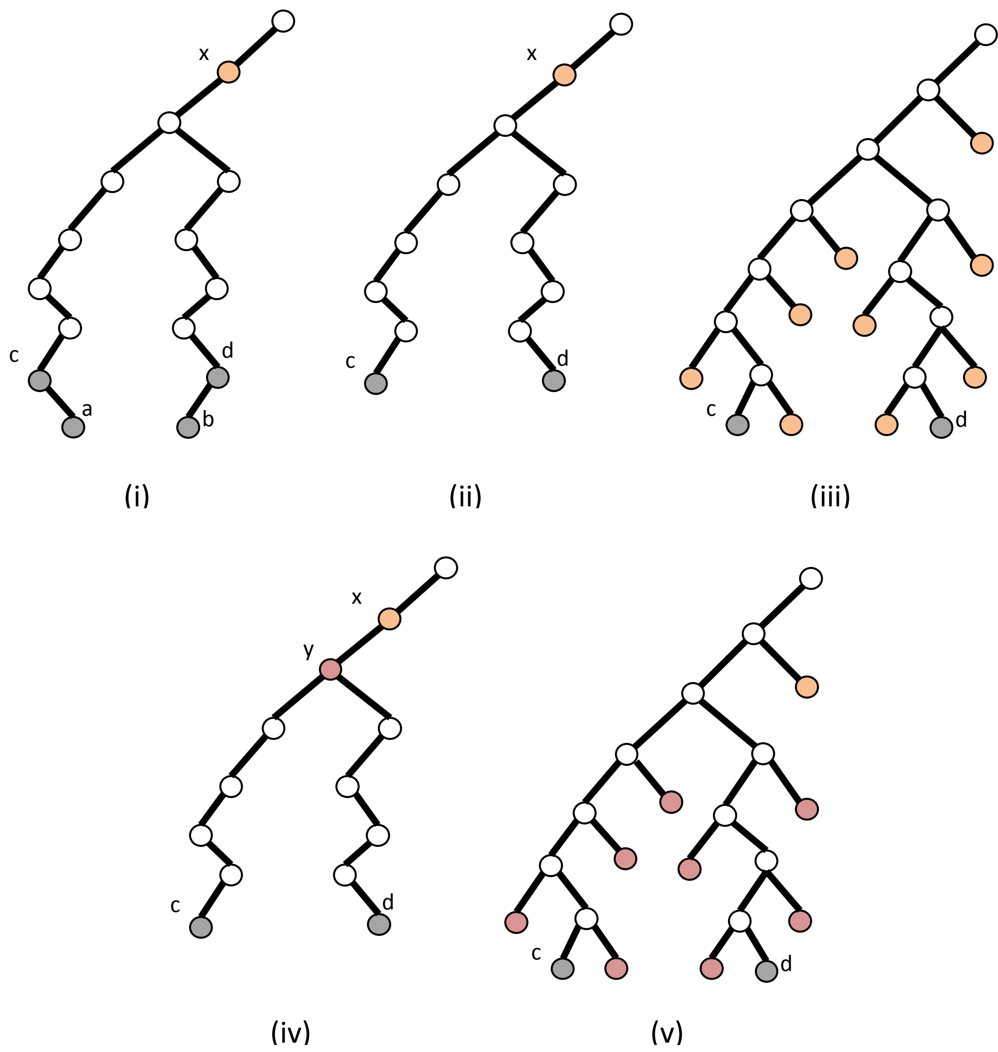

Figure 5-1 Effects of updates on set bounded leaf pushing algorithm

5.2 COMPARISON AND REMARKS

Our proposed method is compared with four existing solutions in terms of three challenging areas for virtual routers: memory footprint, quick update capability and throughput. Competitive methods are named as Method 2 (trie merging with incremental updates), Method 3 (simple merge without leaf pushing), Method 4 (simple merge with leaf pushing) and Method 5 (set bounded leaf pushing).

Memory footprint: Total memory size is evaluated as the sum of the trie size and the next hop table size for Method 1 (our proposal), Method 2 and Method 4. Only the trie size is evaluated for Method 3 and Method 5, since these methods store next hop information directly in nodes. Memory footprint results for the existing solutions and our proposal are given in MBs in Table 5-8 and Figure 5-2.

Our solution has the best memory performance among five competing methods for IPv4 core router tables and ranked second after Method 5 for the other types. IPv4 core routers have the denser prefix distribution among all kinds of routers where 41% of the nodes are full. Moreover, unlike the other types of routers, distinct router tables use prefixes commonly in IPv4 core routers. Method 5 prepends a virtual router ID to each prefix and stores each one separately even they are common in several tables. For these reasons, although tree presents a memory efficient scheme, Method 5 is not suitable for IPv4 core routers.

Table 5-8 Memory footprint (in MB)

	1. proposed solution	2. trie merging with incremental updates	3. simple trie merging without leaf pushing	4. simple trie merging with leaf pushing	5. set-bounded leaf pushing
IPv4 CORE	10.96	11.58	12.39	15.15	30.88
IPv4 EDGE	24.95	36.48	41.55	44.81	9.31
IPv6 CORE	402.13	616.24	809.84	538.38	41.58
IPv6 EDGE	123.67	209.93	282.85	187.75	12.62

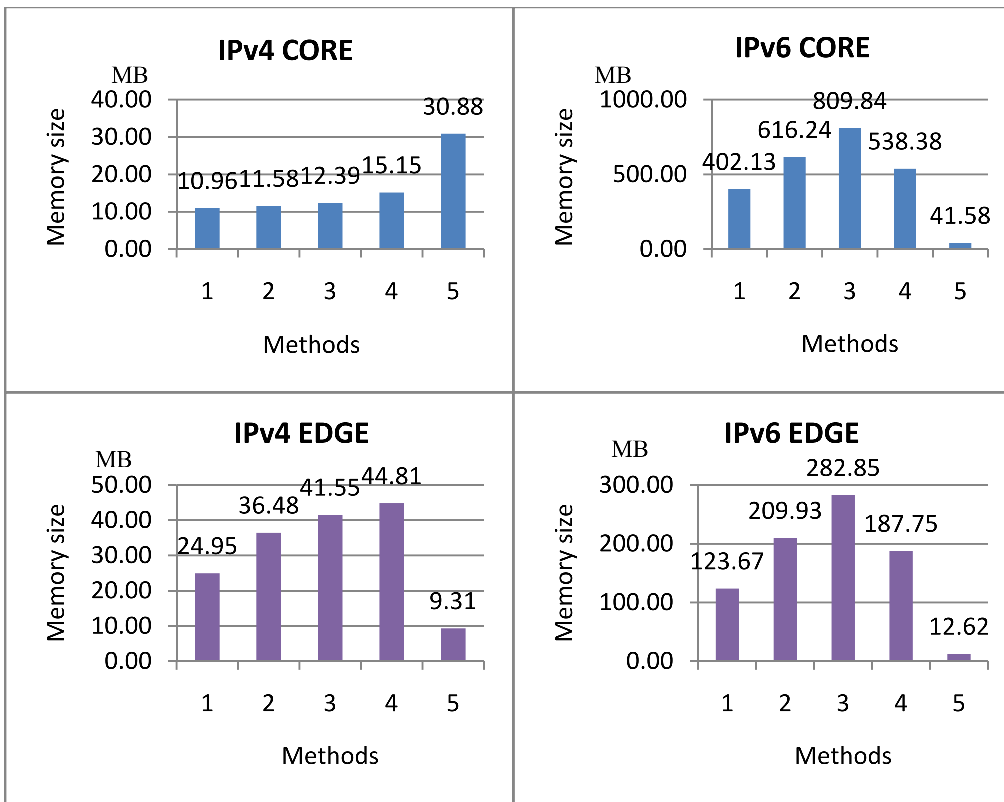

Figure 5-2 Memory footprint comparison

Quick update: Method 4 and Methods 5 are disadvantageous because of their update overhead. Method 4 has a trie based and Methods 5 has a 2-3 tree based data structure where both data structures support quick updates with a single write bubble for up to two node insertion or deletion in the same level. But because of leaf pushing used in Methods 4 and 5 one prefix changes may require so many node changes, which means several write bubbles are needed. Every write bubble

blocks the pipeline during a clock cycle and long term blockings are undesirable due to increased latency.

Our method is more advantageous than these two in terms of update overhead. We guarantee single write bubble update for all announcements. For withdraws, if the prefix to be deleted is stored in the trie, update is achieved with single write bubble. If it is in the tree, the update may affect more than one node due to leaf pushing expands prefixes. But we note that leaf pushing increases the total number of prefixes in all FIBs by 1.0006 and 1.0176 for IPv4 core and edge routers, respectively in our approach. This coefficient is 1,7 in Method 4 for IPv4 routers and it is 1.12 and 1.17 in Method 5 again for IPv4 core and edge routers, respectively. Leaf pushing effect in our method is negligible compared to 4 and 5 for IPv4 routers, thus our scheme presents a better update performance.

Methods 2 and 3 performs all kinds of updates by using a single write bubble hence, they are advantageous in terms of quick updates.

Throughput: Although throughput statistics depends on hardware implementation platform, we make a theoretical comparison. Throughput depends on the access time to memory. If node size exceeds the SRAM entry size more than one memory access are required to obtain contents of several entries. If a node fits into a single SRAM entry, single memory access is enough for each node in each level. In [2], it was reported that one lookup per clock cycle has been achieved. Trie part in our structure is the same with trie based solutions [1,2] hence provides one lookup per clock cycle conceptually. Method 3 has the disadvantage of large node size that grows with the number of merged tables; it affects its throughput negatively. In [3] 400 million packets per second throughput has been achieved with a 200MHz frequency. Dual port SRAM was employed for this implementation. As a result a 2-3 tree based method provides one lookup per clock cycle, too. Therefore, we expect an equivalent throughput performance in our solution by referring to these studies.

We store 99% of IPv4 core router prefixes and approximately 70% of other kinds of routers' prefixes in the trie structure. We prefer to use trie as the main data structure and use tree only as a memory saving secondary data structure. Furthermore the truncation process doesn't disrupt the trie structure too much. Thus a detailed comparison results among trie based solutions can be seen in Figure 5-3 and the exact memory reduction ratios are given in Table 5-9. Our proposal, i.e., Method 1, achieves around 5% memory reduction for IPv4 core routers and around 35% reduction for other types of routers when compared to most memory efficient solutions for different kinds of router.

We utilize the advantages of the trie on both lookup and update operations. Besides, we achieve a more manageable sized trie structure with a standard latency without a significant increase in the total number of prefixes.

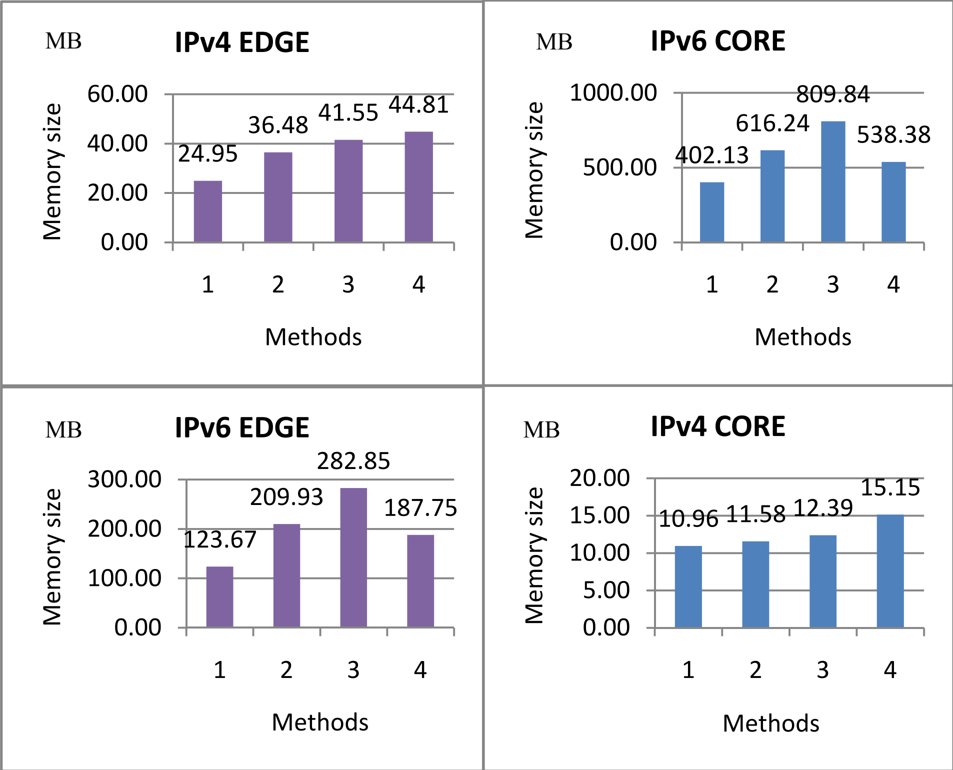

Figure 5-3 Memory footprint comparison among trie based solutions

Table 5-9 Memory gain of the proposal with respect to other trie based methods

	memory gain wrt Alg.2	memory gain wrt Alg.3	memory gain wrt Alg.4
IPv4 CORE	5,35%	11,54%	27,66%
IPv4 EDGE	31,61%	39,95%	44,32%
IPv6 CORE	34,74%	50,34%	25,31%
IPv6 EDGE	41,09%	56,28%	34,13%

CHAPTER 6

CONCLUSION

Virtual routers are among prominent solutions used to fulfill increasing network demands. They manage large amount of data and serve several networks using a single hardware platform. Thus, the most challenging issues are memory size and update overhead for virtual routers.

Several data structures and methods have been proposed in the literature for IP lookup in virtual routers. The solutions can be implemented in hardware or software platforms. SRAM is one of the hardware solutions. It is fast and power efficient and its throughput is improved by pipelining. On the other hand SRAMs are limited on chips. The tradeoff between the memory size and update performance stands out as a potential area to improve in virtual router studies.

In the present thesis, we address an SRAM implementation and we aim to achieve memory reduction in comparison to existing solutions without sacrificing the lookup and update performance. We analyze the characteristics of real life IPv4/IPv6 core and edge router prefix tables. In our experiments, we merge 10 FIBs of the same kind of router into a single trie. Our experimental analysis on real prefix sets has shown that, some parts of the simply merged trie include a high number of empty nodes for storing a relatively small number of prefixes. Our motivation then is to detect these underutilized parts of the simply merged trie and eliminate these parts without disrupting the simplicity of the trie. A truncation criteria and an associated truncation process are proposed. In this, we keep most of the prefixes still in the trie by selecting the truncation metric accordingly. 2-3 tree is then proposed as a secondary and auxiliary data structure to store the truncated

prefixes. 2-3 tree is a memory efficient data structure and supports single write bubble updates, thus matches our aim of memory size reduction while supporting incremental updates. According to a memory size analysis, we demonstrate that our proposal achieves around 5% memory reduction for IPv4 core routers and around 35% reduction for other types of routers in comparison to simple trie merging solutions. The increase in the number prefixes in the tree due to leaf pushing is found to be negligible in our scheme and therefore we may claim that the new structure also supports single write bubble updates substantially. The throughput depends on the implementation but our trie-tree structure supports one lookup per clock cycle in theory by referring to other trie and tree studies.

REFERENCES

[1] J. Fu and J. Rexford, *“Efficient IP-address lookup with a shared forwarding table for multiple virtual routers”,* in Proc. of the ACM CoNEXT Conference, pp. 1-12, 2008.

[2] L. Luo, G. Xie, K. Salamatian, S. Uhlig, L. Mathy and Y. Xie, “*A trie merging approach with incremental updates for virtual routers”,* in Proc. of the IEEE INFOCOM, pp. 1222 – 1230, 2013.

[3] H. Le, T. Ganegedara and V. K. Prasanna, *“Memory-efficient and scalable virtual routers using FPGA”*, in Proc. of the 19th ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays, pp. 257-266, 2011.

[4] H. Song, M. S. Kodialam, F. Hao and T. V. Lakshman, “*Scalable IP lookups using shape graphs”*, in Proc. of ICNP, pp. 73-82, 2009.

[5] H. Song, M. S. Kodialam, F. Hao and T. V. Lakshman, *“Efficient trie braiding in scalable virtual routers”,* IEEE/ACM Transactions on Networking, Vol. 20, pp. 1489-1500, 2012.

[6] V. P. Kumar, T. V. Lakshman, and D. Stiliadis, “*Beyond best effort: router architectures for the differentiated services of tomorrow's Internet”,* IEEE Communications Magazine, Vol.36, pp. 152-164, 2002.

[7] E. Fredkin. *”Trie memory”,* Communications of the ACM, Vol.3, pp. 490-499. 1960.

[8] T. H. Cormen, C. E. Leiserson, R. L.Rivest, *Introduction to Algorithms,* MIT Press, McGRaw-Hill Company, pp.244, 1990.

[9] V. Srinivasan and G. Varghese, *“Faster IP lookups using controlled prefix expansion”,* in Proc. ACM SIGMETRICS, pp. 1–10, 1998.

[10] S. Nilsson, G. Karlsson, *“IP-address lookup using LC-tries”*, IEEE Journal on Selected Areas in Communications, Vol. 17, pp. 1083-1092, 2002.

[11] D. R. Morrison. Patricia *“Practical algorithm to retrieve information coded in alphanumeric”*, Journal of ACM, pp. 514–534, 1968
[12] P. F. Dietz,*“Maintaning order in a linked list”,* in Proceedings of STOC, pp.122-127, 1982.
[13]
http://www.circleid.com/posts/visibility_of_prefix_lengths_in_ipv4_and_ipv6
(last visited on 08.01.2014)
[14] Y. H. E. Yang and V. K. Prasanna. *“High throughput and large capacity pipelined dynamic search tree on FPGA”*, in Proc. FPGA’10, pp. 83-92, 2010.
[15] A. Basu and G. Narlikar, *“Fast incremental updates for pipelined forwarding engines”,* in Proc. IEEE INFOCOM, 2003.
[16] A. S. Tanenbaum, *Computer Networks*, Pearson Education, pp. 437-444, 2003.
[17] T. Anderson, L. Peterson, S. Shenker, J. Turner, *”Overcoming the Internet impasse through virtualization”,* IEEE Computer, vol.38, no.4, pp. 34-41, April 2005.
[18] D. Unnikrishnan, R. Vadlamani, Y. Liao, A. Dwaraki, J. Crenne, L. Gao, and R. Tessier, *“Scalable network virtualization using FPGAs”,* in Proc. of ACM/SIGDA FPGA ‘10, pp. 219-228, 2010.
[19] K. S. Kim, S. Sahni, “*Efficient construction of pipelined multibit-trie router-tables*”, IEEE Trans. on Computers, vol.56, pp.32-43, 2007.
[20] RIS RAW DATA [Online]. [http://data.ris.ripe.net]. (last visited on 06.03.2010)
[21] T. Ganegedara, W. Jiang, and V. Prasanna, *“Frug: A Benchmark for Packet Forwarding in Future Network”* IPCCC ’10: Proc. IEEE Int’l Performance Computing and Comm. Conf., 2010.
[22] M. Wang, S. Deering, T. Hain, and L. Dunn, *“Non-Random Generator for IPv6 Table”,* HOTI ’04: Proc. Ann. IEEE Symp. High Performance Interconnects, pp. 35-40, 2004.

APPENDIX A

IPv4 CORE ROUTER ANALYSIS

Table A-1 IPv4 core router analysis

x upper bound for # of prefixes	*y* lower bound for # of nodes	N(Ti) total number of nodes in truncated trie	Pt number of truncated prefixes	node%	prefix %
1	10	1.027.932	384	99,56	0,01
1	15	1.031.959	33	99,95	0,00
1	20	1.032.435	3	99,99	0,00
1	25	1.032.497	0	100,00	0,00
1	30	1.032.497	0	100,00	0,00
1	35	1.032.497	0	100,00	0,00
1	40	1.032.497	0	100,00	0,00
1	45	1.032.497	0	100,00	0,00
1	50	1.032.497	0	100,00	0,00
1	55	1.032.497	0	100,00	0,00
1	60	1.032.497	0	100,00	0,00
1	65	1.032.497	0	100,00	0,00
1	70	1.032.497	0	100,00	0,00
2	10	1.020.778	1.502	98,87	0,05
2	15	1.029.083	354	99,67	0,01
2	20	1.031.484	88	99,90	0,00
2	25	1.032.279	16	99,98	0,00
2	30	1.032.466	2	100,00	0,00
2	35	1.032.497	0	100,00	0,00
2	40	1.032.497	0	100,00	0,00

Table A-1 (continued)

2	45	1.032.497	0	100,00	0,00
2	50	1.032.497	0	100,00	0,00
2	55	1.032.497	0	100,00	0,00
2	60	1.032.497	0	100,00	0,00
2	65	1.032.497	0	100,00	0,00
2	70	1.032.497	0	100,00	0,00
3	10	1.013.265	3.292	98,14	0,10
3	15	1.026.005	843	99,37	0,03
3	20	1.029.785	301	99,74	0,01
3	25	1.031.394	109	99,89	0,00
3	30	1.032.140	29	99,97	0,00
3	35	1.032.331	12	99,98	0,00
3	40	1.032.445	3	100,00	0,00
3	45	1.032.445	3	100,00	0,00
3	50	1.032.445	3	100,00	0,00
3	55	1.032.497	0	100,00	0,00
3	60	1.032.497	0	100,00	0,00
3	65	1.032.497	0	100,00	0,00
3	70	1.032.497	0	100,00	0,00
4	10	997.095	8.744	96,57	0,27
4	15	1.022.398	1.645	99,02	0,05
4	20	1.028.544	499	99,62	0,02
4	25	1.030.544	217	99,81	0,01
4	30	1.031.454	109	99,90	0,00
4	35	1.032.029	44	99,95	0,00
4	40	1.032.319	15	99,98	0,00
4	45	1.032.400	7	99,99	0,00
4	50	1.032.445	3	100,00	0,00
4	55	1.032.497	0	100,00	0,00
4	60	1.032.497	0	100,00	0,00
4	65	1.032.497	0	100,00	0,00
4	70	1.032.497	0	100,00	0,00
5	5	867.787	64.496	84,05	2,01
5	6	885.574	57.209	85,77	1,78
5	7	904.629	49.729	87,62	1,55
5	8	953.599	25.469	92,36	0,79
5	9	964.663	21.608	93,43	0,67

Table A-1 (continued)

5	10	979.187	15.789	94,84	0,49
5	15	1.016.694	3.292	98,47	0,10
5	20	1.026.714	874	99,44	0,03
5	25	1.029.553	378	99,71	0,01
5	30	1.030.810	199	99,84	0,01
5	35	1.031.669	87	99,92	0,00
5	40	1.032.033	45	99,96	0,00
5	45	1.032.154	32	99,97	0,00
5	50	1.032.383	8	99,99	0,00
5	55	1.032.435	5	99,99	0,00
5	60	1.032.435	5	99,99	0,00
5	65	1.032.497	0	100,00	0,00
5	70	1.032.497	0	100,00	0,00
6	5	858.837	69.678	83,18	2,17
6	6	873.524	63.469	84,60	1,97
6	7	889.398	56.707	86,14	1,76
6	8	935.464	33.747	90,60	1,05
6	9	945.929	29.941	91,62	0,93
6	10	959.451	24.377	92,93	0,76
6	15	1.007.302	6.412	97,56	0,20
6	20	1.023.922	1.560	99,17	0,05
6	25	1.028.220	633	99,59	0,02
6	30	1.029.805	361	99,74	0,01
6	35	1.031.044	168	99,86	0,01
6	40	1.031.485	111	99,90	0,00
6	45	1.031.731	80	99,93	0,00
6	50	1.032.097	38	99,96	0,00
6	55	1.032.252	23	99,98	0,00
6	60	1.032.366	11	99,99	0,00
6	65	1.032.428	6	99,99	0,00
6	70	1.032.497	0	100,00	0,00
7	10	945.179	30.855	91,54	0,96
7	15	998.650	9.574	96,72	0,30
7	20	1.019.744	2.845	98,76	0,09
7	25	1.026.873	961	99,46	0,03
7	30	1.029.210	478	99,68	0,01
7	35	1.030.804	204	99,84	0,01

Table A-1 (continued)

7	40	1.031.281	140	99,88	0,00
7	45	1.031.619	94	99,92	0,00
7	50	1.031.987	52	99,95	0,00
7	55	1.032.194	30	99,97	0,00
7	60	1.032.366	11	99,99	0,00
7	65	1.032.428	6	99,99	0,00
7	70	1.032.497	0	100,00	0,00
8	10	914.120	46.297	88,53	1,44
8	15	964.730	25.778	93,44	0,80
8	20	1.011.718	5.631	97,99	0,18
8	25	1.024.546	1.589	99,23	0,05
8	30	1.028.089	731	99,57	0,02
8	35	1.030.222	304	99,78	0,01
8	40	1.030.709	230	99,83	0,01
8	45	1.031.087	176	99,86	0,01
8	50	1.031.642	100	99,92	0,00
8	55	1.032.002	54	99,95	0,00
8	60	1.032.174	35	99,97	0,00
8	65	1.032.362	14	99,99	0,00
8	70	1.032.497	0	100,00	0,00

APPENDIX B

IPv4 EDGE ROUTER ANALYSIS

Table B-1 IPv4 edge router analysis

x upper bound for # of prefixes	*y* lower bound for # of nodes	N(Ti) total number of nodes in truncated trie	Pt number of truncated prefixes	node%	prefix %
1	5	2.422.054	153.651	72,87	16,01
1	10	3.287.194	3.227	98,90	0,34
1	15	3.323.224	42	99,98	0,00
1	20	3.323.862	0	100,00	0,00
1	25	3.323.862	0	100,00	0,00
1	30	3.323.862	0	100,00	0,00
1	35	3.323.862	0	100,00	0,00
2	5	1.201.193	496.084	36,14	51,68
2	6	1.702.776	358.728	51,23	37,37
2	7	2.079.068	263.355	62,55	27,43
2	8	2.405.616	180.568	72,37	18,81
2	9	2.615.018	131.643	78,67	13,71
2	10	2.861.244	78.200	86,08	8,15
2	15	3.274.760	6.102	98,52	0,64
2	20	3.322.139	162	99,95	0,02
2	25	3.323.762	8	100,00	0,00
2	30	3.323.862	0	100,00	0,00
2	35	3.323.862	0	100,00	0,00
3	5	579.662	706.989	17,44	73,65
3	10	2.076.221	265.595	62,46	27,67

Table B-1 (continued)

3	15	2.988.313	57.130	89,90	5,95
3	20	3.271.157	7.258	98,41	0,76
3	25	3.318.995	554	99,85	0,06
3	30	3.323.642	21	99,99	0,00
3	35	3.323.862	0	100,00	0,00
4	5	313.846	806.613	9,44	84,02
4	10	1.354.078	471.849	40,74	49,15
4	15	2.479.945	170.498	74,61	17,76
4	20	3.082.257	40.847	92,73	4,25
4	25	3.270.243	7.755	98,39	0,81
4	30	3.314.823	1.141	99,73	0,12
4	35	3.323.422	48	99,99	0,01
4	40	3.323.821	4	100,00	0,00
4	45	3.323.862	0	100,00	0,00
4	50	3.323.862	0	100,00	0,00
5	5	198.684	853.050	5,98	88,86
5	10	841.487	634.820	25,32	66,13
5	15	1.902.486	320.953	57,24	33,43
5	20	2.739.971	113.740	82,43	11,85
5	25	3.131.902	32.524	94,22	3,39
5	30	3.273.150	7.506	98,47	0,78
5	35	3.311.912	1.603	99,64	0,17
6	5	144.183	877.167	4,34	91,37
6	10	518.996	745.029	15,61	77,61
6	15	1.383.076	470.892	41,61	49,05
6	20	2.296.947	221.285	69,10	23,05
6	25	2.878.693	84.900	86,61	8,84
6	30	3.163.591	27.092	95,18	2,82
6	35	3.272.190	7.846	98,45	0,82

APPENDIX C

IPv6 CORE ROUTER ANALYSIS

Table C-1 IPv6 core router analysis

x upper bound for # of prefixes	*y* lower bound for # of nodes	N(Ti) total number of nodes in truncated trie	Pt number of truncated prefixes	node%	prefix %
1	5	568.640	3.225.242	0,91	98,82
1	10	3.929.961	2.976.521	6,31	91,20
1	15	13.809.347	2.335.529	22,17	71,56
1	20	33.373.181	1.281.765	53,57	39,27
1	21	39.245.265	997.284	63,00	30,56
1	22	45.440.429	708.500	72,94	21,71
1	25	58.473.510	142.074	93,87	4,35
1	30	61.754.564	15.322	99,13	0,47
1	35	61.961.881	8.846	99,46	0,27
1	40	62.245.532	1.230	99,92	0,04
2	5	63.476	3.259.797	0,10	99,88
2	10	883.625	3.205.234	1,42	98,21
2	15	3.667.438	3.033.419	5,89	92,94
2	20	9.498.740	2.721.164	15,25	83,38
2	25	21.209.840	2.158.725	34,05	66,14
2	30	27.084.022	1.804.336	43,48	55,28
2	35	34.389.072	1.355.432	55,20	41,53
2	36	36.524.908	1.234.426	58,63	37,82
2	37	38.352.274	1.133.742	61,57	34,74
2	38	40.727.119	1.006.326	65,38	30,83

Table C-1 (continued)

2	39	42.723.443	902.339	68,58	27,65
2	40	45.299.603	771.589	72,72	23,64
3	35	16.039.612	2.396.487	25,75	73,43
3	40	22.311.934	2.042.543	35,82	62,58
3	45	30.302.565	1.620.524	48,64	49,65
3	50	37.317.640	1.246.822	59,90	38,20
3	51	38.480.509	1.181.774	61,77	36,21
3	52	39.730.426	1.112.859	63,78	34,10

APPENDIX D

IPv6 EDGE ROUTER ANALYSIS

Table D-1 IPv6 core router analysis

x upper bound for # of prefixes	*y* lower bound for # of nodes	N(Ti) total number of nodes in truncated trie	Pt number of truncated prefixes	node%	prefix %
1	5	49.554	997.261	0,22	99,73
1	6	87.611	994.985	0,39	99,50
1	7	151.701	990.954	0,68	99,10
1	8	252.607	984.330	1,14	98,43
1	9	395.366	974.646	1,78	97,46
1	10	570.578	962.715	2,57	96,27
1	11	775.238	949.081	3,49	94,91
1	12	1.014.101	933.762	4,57	93,38
1	13	1.308.924	915.544	5,90	91,55
1	14	1.683.667	893.081	7,59	89,31
1	15	2.157.618	865.499	9,73	86,55
1	16	2.733.353	832.965	12,32	83,30
1	17	3.402.593	796.451	15,34	79,65
1	18	4.147.257	757.433	18,69	75,74
1	19	4.978.529	715.652	22,44	71,57
1	20	5.944.707	668.862	26,80	66,89
1	21	7.121.068	613.853	32,10	61,39
1	22	8.611.573	546.431	38,82	54,64
1	23	10.577.586	460.383	47,68	46,04
1	24	13.085.374	354.113	58,98	35,41

Table D-1 (continued)

1	25	15.975.243	235.693	72,01	23,57
2	15	474.325	973.138	2,14	97,31
2	16	624.134	965.294	2,81	96,53
2	17	812.223	955.624	3,66	95,56
2	18	1.024.493	944.947	4,62	94,49
2	19	1.247.726	934.097	5,62	93,41
2	20	1.487.265	922.724	6,70	92,27
2	21	1.758.251	910.334	7,93	91,03
2	22	2.082.950	895.851	9,39	89,59
2	23	2.507.560	877.677	11,30	87,77
2	24	3.127.053	851.884	14,10	85,19
2	25	4.039.156	815.153	18,21	81,52
2	26	5.166.278	770.498	23,29	77,05
2	27	6.159.882	731.551	27,77	73,16
2	28	6.799.094	705.493	30,65	70,55
2	29	7.118.929	691.081	32,09	69,11
2	30	7.322.721	679.721	33,01	67,97
2	35	8.491.656	609.474	38,28	60,95
2	36	8.875.999	587.839	40,01	58,78
2	37	9.243.978	567.667	41,67	56,77
2	38	9.728.045	541.798	43,85	54,18
2	39	10.190.487	517.751	45,93	51,78
2	40	10.793.024	487.206	48,65	48,72
2	41	11.360.508	459.239	51,21	45,92
2	42	12.089.690	424.113	54,50	42,41
2	43	12.763.911	392.315	57,53	39,23
2	44	13.607.700	353.269	61,34	35,33
2	45	14.378.447	318.379	64,81	31,84
2	46	15.339.385	275.804	69,14	27,58
2	47	16.185.495	239.110	72,96	23,91
2	48	17.233.885	194.590	77,68	19,46
2	49	18.108.832	158.202	81,63	15,82
2	50	19.133.944	116.418	86,25	11,64
2	51	19.872.740	86.904	89,58	8,69
2	52	20.635.742	57.010	93,02	5,70
2	53	21.070.654	40.294	94,98	4,03
2	54	21.473.700	25.094	96,80	2,51

Table D-1 (continued)

2	55	21.683.976	17.310	97,74	1,73
2	115	4.598.344	789.354	20,73	78,94
2	120	4.792.514	780.677	21,60	78,07
2	125	4.988.399	771.930	22,49	77,19
2	130	5.180.002	763.401	23,35	76,34
2	135	5.377.059	754.621	24,24	75,46
2	140	5.567.561	746.129	25,10	74,61
2	145	5.758.864	737.594	25,96	73,76