IP Address Lookup and Packet Classification Algorithms

by

Xuehong Sun, B.Sc., M.Sc.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

School of Mathematics and Statistics
Ottawa-Carleton Institute for Mathematics and Statistics

Carleton University
Ottawa, Ontario, Canada

November, 2003

© Copyright
2003, Xuehong Sun

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisitions et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-89896-2

Our file Notre référence

ISBN: 0-612-89896-2

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this dissertation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de ce manuscrit.

While these forms may be included in the document page count, their removal does not represent any loss of content from the dissertation.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

The Internet system consists of packet transmission media and packet processing elements. Optical fibers are used as packet transmission media and Internet routers implement the functions of packet processing elements.

In order to keep up with the wire rate speed (optical fiber transmission speed), Internet routers must forward the Internet packets at a very high speed, while keeping the cost low. Though many efforts have been devoted to address this issue, there is still much to be improved.

IP address lookup and packet classification are one of the most important functions (bottlenecks) performed by Internet routers. We need to perform IP address lookup and/or packet classification at the wire rate speed. Many solutions exist to tackle this problem. There are the early days software solutions and recently hardware solutions such as TCAM, ASIC. The problem has attracted much attention from both academic and industrial societies. Developing high performance algorithms to overcome the drawbacks of existing solutions is the objective of this thesis.

In this thesis, we propose three algorithms for IP address lookup and packet classification in Internet routers: the first one is for IP address lookup; the second one is a dynamic algorithm for IP address lookup; the third one is for packet classification.

Although several Ph.D. theses [64, 74, 26] have been generated in recent years in the packet classification field, our work represents one of the accomplishments

in this field. Our goal is to develop algorithms, which consume as little memory as possible so that the algorithms can be implemented on a single chip. Since on-chip memory access is much faster than off-chip memory, the speed for IP address lookup or packet classification can be significantly increased. We have achieved this goal. The prominent feature of our algorithms is that they consume the least amount of memory compared to the other current solutions. Another feature is that all the algorithms are amenable to pipelined and/or parallel implementations. Thus, the speed is only restricted by memory access latency. In the future, when the speed requirement increases due to the advancement of other technologies, such as the increased wire rate speed, algorithms that are suitable for on-chip SRAM implementation will be the only solution. We believe we represent the trend of future technology.

First, we give a background and review of the existing technologies in the field of IP address lookup and packet classification. After that, three algorithms are detailed in each of the following chapters. We conclude in the last chapter.

Acknowledgements

First of all, I would like to thank my supervisor Professor Yiqiang Q. Zhao for his continuing support, guidance, valuable suggestions and enlightening comments during the period of my study at Carleton University. Working with him has always been an enjoyable and rewarding experience. I also thank him for his very kind concerns about my life and career during these years.

I also would like to thank Professor Amitava Bose from Carleton University and Professor David McDonald from the University of Ottawa, who conducted the comprehensive exam for my Ph.D. program at the beginning of writing the thesis. They have given me invaluable suggestions and comments. I would like to thank Professor Sartaj K. Sahni from the University of Florida. We first met at IEEE Symposium on Computers and Communications in Turkey 2003. He carefully read the packet classification algorithm and gave insightful comments and suggestions. To my delight, he will be the external examiner for my thesis.

Dr. Gerard Damm and Mr. Milan Zoranovic of Alcatel have read part of the thesis and made comments and suggestions on the packet classification algorithm using independent sets. Dr. Greg Soprovich of SiberCore Technologies has read the IP address lookup algorithm using compressed keys. He specifically provided several experimental scenarios and made comments regarding industrial applications. I am really grateful for their invaluable comments and feedback.

I would like to thank the School of Mathematics and Statistics, Carleton Univer-

sity, for providing the necessary research facilities for completing this thesis. Many thanks go to the members and staff of the School of Mathematics and Statistics, who helped me during my studies at Carleton University.

Special thanks are given to Mr. Luc Lalande of Carleton University. With his help and tireless support, we (Yiqiang and I) co-founded Sourithm Corp., a start-up R&D company, which can help to bring our research to real life application.

At this moment, I am thinking about my father and mother. From my childhood, I began to learn from them about the merits of hard work, diligence and being positive towards life under any circumstance. I cannot find words to express my gratitude to their love and care. I would like to thank my wife, Xiaodan Du. She is always there to support me. She just brought a new source of joy to the family, our son Rick. I also miss my father-in-law and mother-in-law, my brother and sisters and uncle and aunt in China. They have always given me encouragement and support through these years.

Contents

Title Page	i
Acceptance Sheet	ii
Abstract	iii
Acknowledgements	v
Contents	vii
Acronyms	xii
List of Figures	xiii
List of Tables	xvi
1 Introduction	1
2 Preliminaries	5
2.1 Background	5
2.1.1 Internet	6

2.1.2	IPv4 Addressing Architecture	13
2.1.3	IPv6 Addressing Architecture	15
2.1.4	Background for Packet Classification	17
2.1.5	Performance Metrics for Lookups and Packet Classification Algorithms	18
2.1.6	Notation for Algorithm Complexity	20
2.2	IP Address Lookup Algorithms	21
2.2.1	Naive Algorithms	21
2.2.2	Trie	23
2.2.3	Prefix Range Search	27
2.2.4	Binary Search on Prefix Lengths	31
2.2.5	Ternary CAM	35
2.2.6	Other Algorithms	36
2.3	Packet Classification Algorithms	37
2.3.1	The Relationship between Prefix and Arbitrary Range	38
2.3.2	Classifier and its Geometry Representation	40
2.3.3	Naive Algorithms	43
2.3.4	Parallel Algorithms	43
2.3.5	Serial Algorithms	46
2.3.6	Combined Dimensional Search Algorithms	51
2.3.7	Tuple Space Search	53
2.3.8	Ternary CAM	53
2.4	Summary	54

3	An IP Address Lookup Algorithm	55
3.1	A Hardware Design Reference Model	57
3.2	Convert Longest Prefix Match to Range Search Problem	58
3.3	Put the Tree in Memory	63
3.3.1	Compressed Endpoints and Shared Pointers	63
3.3.2	Build the Tree from Bottom-Up	68
3.3.3	Optimizations	76
3.4	Experimental Study	77
3.4.1	Port Merge Effect	77
3.4.2	Comparisons on Variants	78
3.4.3	Results Using Real Life Tables	79
3.4.4	Results Using the Expanded IPv4 Tables	81
3.4.5	Results Using the Expanded IPv6 Tables	83
3.5	Previous Work	86
3.6	Summary	89
4	A Fast Update Algorithm	91
4.1	The Worst-case Scenario for Update	92
4.2	Partitioning the Prefixes into Independent Sets	93
4.2.1	Definition	93
4.2.2	Partitioning	93
4.2.3	Experimental Results	96
4.3	Update Algorithm	97
4.3.1	Deleting a Prefix	98

4.3.2	Adding a Prefix	101
4.3.3	Estimation on Complexity	102
4.4	Previous Work	102
4.5	Summary	103
5	A Packet Classification Algorithm	104
5.1	Preliminary	105
5.2	Packet Classification Problem	106
5.3	Developing the Algorithm	107
5.3.1	Independent Sets	108
5.3.2	Finding a Global Maximum Independent Set	110
5.3.3	Basic Data Structure for a Group of Independent Sets	115
5.3.4	Search	118
5.3.5	Update	119
5.3.6	Complexities of the Algorithm	127
5.3.7	Variations	128
5.3.8	A Property of the Set of Independent Sets	129
5.4	Experimental Study	130
5.4.1	Classifiers without Wildcards	131
5.4.2	Classifiers with Wildcards	133
5.4.3	One Scenario of Implementation	135
5.5	Previous Work	136
5.6	Summary	137

6	Conclusions and Industrial Outlook	139
6.1	Conclusions	139
6.1.1	Contributions of the IP Address Lookup Algorithm	140
6.1.2	Contributions of the IP Address Table Update Algorithm . . .	141
6.1.3	Contributions of the Packet Classification Algorithm	141
6.2	Industrial Outlook	142
	Bibliography	145

Acronyms:

ALU: Arithmetic Logic Unit

ASIC: Application Specific Integrated Circuit

BGP: Border Gateway Protocol

CIDR: Classless Inter-Domain Routing

DiffServ: Differentiated Services

IntServ: Integrated Services

IP: Internet Protocol

ISP: Internet Service Providers

MPPS: Million Packets Per Second

NAP: Network Access Point

QoS: Quality of Service

RISC: Reduced Instruction Set Computer

SONET: Synchronous Optical NETwork

TCAM: Ternary Content Addressable Memory

TCP: Transmission Control Protocol

VPN: Virtual Private Network

WDM: Wavelength Division Multiplexing

List of Figures

2.1	The Internet.	7
2.2	IP router.	8
2.3	An example architecture of the data plane.	9
2.4	An example of memory hierarchy.	10
2.5	The growth of IP forwarding table.	12
2.6	A binary trie.	24
2.7	A compressed trie.	26
2.8	A multibit trie.	27
2.9	Range representation of prefixes.	28
2.10	A range search tree.	29
2.11	A multiway range tree.	31
2.12	Binary search on prefix lengths.	34
2.13	TCAM.	36
2.14	Convert ranges to prefixes.	39
2.15	Geometric representation of a classifier.	42
2.16	Trie of tries.	47
2.17	Trie of tries with switch points.	48

2.18 FIS tree.	49
2.19 Area-based quadtree.	52
3.1 Hardware design reference model.	57
3.2 Prefixes and their ranges and ports.	60
3.3 Converted endpoints with the corresponding ports.	61
3.4 A tree structure.	62
3.5 The row data structure for a node of the tree.	65
3.6 Tree in memory.	66
3.7 Memory size for IPv4 as a function of table size with different memory widths.	84
3.8 Memory size for IPv6 as a function of table size with different memory widths.	88
4.1 The worst-case scenario for update.	92
4.2 The scenario for the need of moving prefixes.	99
4.3 Proof of the property.	101
5.1 An example of an independent set.	109
5.2 A two dimension classifier.	111
5.3 The corresponding graphs of the two dimension classifier.	112
5.4 A heuristic algorithm for finding a maximum independent set in a graph.	113
5.5 Merge sets of begin points into a master set.	116
5.6 Add a virtual point to B_k	117
5.7 A data structure example.	119

5.8	Index array pointed to by points.	120
5.9	Add new rules.	121
5.10	The case 1.1 and case 1.2 for adding new rules.	124
5.11	Original maximum independent sets.	126
5.12	After the deletion of four rules.	127

List of Tables

2.1	A mini forwarding table (address length=5).	16
2.2	Indexing method(address length=5).	22
2.3	A hash table.	32
2.4	Grouping prefixes in lengths.	33
2.5	An example classifier.	41
2.6	A product table of a classifier.	45
2.7	Tuples.	54
3.1	A mini routing table (Maximum prefix length=6)	59
3.2	Real life IPv4 endpoints.	64
3.3	Calculating the “common leading bits”.	69
3.4	An example.	75
3.5	Port merge effect.	78
3.6	Experiments with IPv4 using 144bits memory width.	79
3.7	Results of IPv4 using 144bits memory width.	81
3.8	Results of IPv6 using 168 bits memory width.	82
3.9	Memory size for IPv4 as a function of table size and memory width. .	83

3.10	Number of memory accesses for IPv4 as a function of table size and memory width.	85
3.11	Memory size for IPv6 as a function of table size and memory width. .	86
3.12	Number of memory accesses for IPv6 as a function of table size and memory width.	87
3.13	Bits consumed for each IPv4 prefix.	89
4.1	The size of the Independent Sets.	97
5.1	Experiments with different repeating factors.	132
5.2	Experiments with different table sizes.	133
5.3	Experiments with different percent of wildcards.	134

Chapter 1

Introduction

The Internet system consists of packet transmission media and packet processing elements. Optical fibers are used as packet transmission media and Internet routers implement the functions of packet processing elements.

Optical transmission systems provide high bandwidth. SONET, a system using optical fiber, can transmit data in tens of gigabits per second per fiber. This is called the *wire rate* speed. SONET uses only one channel for each strand of fiber to transfer data, therefore such optical systems are inefficient in using the bandwidth. Wavelength Division Multiplexing (WDM) technology [53] is proposed as an alternative method for carrying data. With WDM, one strand of fiber can accommodate about 100 channels nowadays (2003) and more in the future.

Internet routers implement Internet Protocols (IP) to carry out the functions of packet processing. The main goal of packet processing is to forward the Internet packets as fast as possible. The functions of packet processing are pipelined. One

of the functions is the IP address lookup and/or packet classification function. In order to keep up with the wire rate speed, while keeping the cost low, many efforts have been devoted to make IP address lookup and/or packet classification fast. Generally speaking, there are algorithmic solutions and Content Addressable Memory (CAM)-based solutions. While there are a lot algorithms being developed, there are still many more to be expected. Though CAM-based solutions currently dominate the market, they have drawbacks such as high power dissipation, expensive cost, difficultly scale to large address tables, etc. In fact, some CAM companies are seeking alternative advanced algorithm-based solutions. There are already a couple of companies that solely apply algorithms in IP address lookup and/or packet classification [82] [83].

Under this background, the thesis was developed. In the beginning, our goals were to develop algorithms to address the current market needs. For example, we need high speed, we need to store large tables, we need to reduce power consumption, and we need to reduce costs. These factors led us to develop IP address lookup and/or packet classification algorithms using the smallest possible amount of memory. In order to achieve that, we might give up the goal of achieving the smallest number of memory accesses. The philosophy of this guideline is that if the memory needed is small, we can put the lookup table in on-chip memory instead of off-chip memory. Since on-chip memory access is much faster than off-chip memory access, sacrificing the number of memory accesses still makes our algorithms much faster. We also hope that the algorithms developed will be amenable to parallel or pipelined implementation so that they would be scalable to the performance requirement. The algorithms presented in this thesis accomplished these goals.

In this thesis, we propose three algorithms; two for IP address lookup and one for packet classification in Internet routers: the first one is for IP address lookup; the second one is a dynamic algorithm for IP address lookup and update; the third one is for packet classification. Though they address different aspects of the packet forwarding problem, they are related. The common features of them are: very small memory requirements, inherently pipelined and/or parallel implementation, targeted at hardware implementation, flexible in implementation, scalable to the number of entries, and applicable to IPv6. We summarize the features of these algorithms as follows:

1. The IP address lookup algorithm uses compressed keys. In order to achieve minimum memory requirement, we combined the following three techniques.
 - We compress the keys in a tree node;
 - We use a shared pointer in a tree node;
 - We use bottom-up from the leaves to the root scheme to build the tree.

The data structure is neat and the creation of the data structure from scratch is also fast. The memory requirement is the smallest compared to all the solutions in the literature and in industry. According to current technology advances, a table with 1M IPv4 entries can be put in a single chip and the search speed is several times faster than the products available in the market.

2. For the dynamic algorithm for IP address lookup and update, the general idea is to use the concept of independent sets to eliminate the worst case update scenarios. The algorithm is also so general that it does not depend on any

search algorithm. Any static algorithm can potentially use this algorithm to convert to a dynamic algorithm. The precondition of using this algorithm is that we need resources for parallel implementation. However, the number of parallel resources is not high.

3. The packet classification algorithm uses the concept of independent sets similar to the above dynamic algorithm. Essentially, when the rules are grouped in independent sets, they avoid being divided (or saved) in several pieces and therefore memory is saved. Using the data structure similar to the fractional cascading technique, several independent set searches can be merged to one independent set search.

The algorithm has very small memory requirements. The search speed is neither sensitive to the size of the rule table nor to the percentage of wildcards in the fields. It also scales well from two dimensional classifiers to high dimensional ones. We envision that it scales to IPv6. In particular, the algorithm is inherently parallel. Hardware tailored to this algorithm can achieve very fast search speeds. The algorithm proposed for update is also very fast in general.

The thesis is organized as follows. First, we give a background and review of the existing technologies in the field of IP address lookup and packet classification. Later, three algorithms are detailed, one in each of the following chapters. The last chapter contributes to the conclusion.

Chapter 2

Preliminaries

This chapter first gives the background of IP address lookup and packet classification applications and then illustrates existing IP address lookup algorithms and packet classification algorithms, respectively. The purpose of this chapter is to provide a brief introduction to this field. Emphasis is on the broadness of the material rather than the depth of the techniques.

2.1 Background

Internet speed is not fast enough to satisfy what people need. We experience delay everyday while we are surfing web sites, downloading a file or talking through the Internet. For multimedia applications, we take a long time to first buffer a short piece of video or audio and then play it. Real time Internet conferencing is not ready for practical use yet due partially to the slow Internet speed. Increasing the speed

of the Internet is a challenge facing the Internet community. Industrial companies compete to offer advanced Internet solutions. Academic organizations put many efforts to develop new Internet technologies. Under this background, we investigate the IP address lookup and packet classification problems. This section introduces the background knowledge of IP address lookup and packet classification problems. Section 2.2 and Section 2.3 detail existing IP address lookup and packet classification algorithms respectively.

2.1.1 Internet

Internet Architecture

The Internet system consists of a number of interconnected packet networks supporting communication among host computers using Internet protocols. As shown in Figure 2.1, individual host computers are connected to form a subnet. The subnet can be a campus network or an enterprise network. These networks are connected to the Internet core through the edge routers. The Internet core consists of core routers that are connected by high speed links. Originally, the Internet used copper lines as transmission media. Nowadays, the copper lines are gradually being replaced by optical fiber. Optical transmission systems provide high bandwidth. SONET is an early system using optical fiber. It can transmit data in tens gigabites per second per fiber. OC48c and OC192c are common in the current market. OC768c is the goal of the near future. (The speed of OC48c is 2.4Gb/s; The speed of OC192c is

Figure 2.1: The Internet.

10Gb/s; The speed of OC768c is 40Gb/s.) This speed is called *wire speed*. Since it uses only one channel for each strand of fiber to transfer data, such optical systems are inefficient in using the bandwidth. Wavelength Division Multiplexing (WDM) [53] technology is proposed as an alternative method for carrying data. With WDM, one strand of fiber can accommodate more than 100 channels nowadays (2003) and possibly more in the future. This amounts to terabit per second transmission speed on optical fiber. In order to keep pace with this speed, Internet routers need to increase packets processing speed.

Internet Router

An Internet router implements functions of the networks. It consists of two parts. One is the control plane and the other is the data plane (Figure 2.2). The control

Figure 2.2: IP router.

plane executes the routing protocols (such as BGP, RIP or OSPF) to learn the structure of the network's topology. The result of this operation is the routing table. The routing table is a set of entries. Each entry consists of a packet IP destination address (IP prefix) as key and a next hop address corresponding to the key. The router first creates the routing table and then keeps updating it according to network topology change or route policy change. A simplified (sometimes the same) version of the routing table is copied to the data plane, which is called a *forwarding table*. It is used to forward packets in the data plane. The data plane has four main tasks, which are packet classification (IP address lookup), packet modification, queue/policy management and packet forwarding. One of the data plane designs is shown in Figure 2.3 [49]. In this design, the data plane consists of multiple line cards (each supporting one or more network interfaces) and forwarding engine cards, all plugged into a high-speed switch. When a packet arrives at a line card, its header is removed and passed through the switch to a forwarding engine. (The remainder of the packet remains on the inbound line card). The forwarding engine reads the header to determine how to forward the packet and then updates

Figure 2.3: An example architecture of the data plane.

the header and sends the updated header and its forwarding instructions back to the inbound line card. The inbound line card integrates the new header with the rest of the packet and sends the entire packet to the outbound line card for transmission.

Let us take a close look at the processor in the forward engine. In [49], the processor is a 415MHz Digital Equipment Corporation Alpha 21164 processor. The Alpha 21164 consists of registers and three internal caches and supports for an external cache (Figure 2.4). (4lpc means four instruction loading per cycle; 8c means eight cycles latency.) The Alpha 21164 is a 64-b 32-register super-scalar reduced

Figure 2.4: An example of memory hierarchy.

instruction set computer (RISC) processor. There are two integer logic units, called E0 and E1, and two floating point units, called FA and FM. The four logic units are distinct. While most integer instructions (including loads) can be done in either E0 or E1, a few important operations, most notably byte extraction, shift operations, and stores, can only be done in E0. Floating point operations are more restricted, with all but one instruction limited to either FA or FM. In each cycle the Alpha attempts to schedule one instruction to each logic unit. For integer register-to-register operations, results are almost always available in the next instruction cycle. Floating results typically take several cycles. The Alpha processes instructions in

groups of four instructions (hereafter called quads). All four instructions in a quad must successfully issue before any instructions in the next quad are issued. In practice this means that the programmer's goal is to place either two pairs of integer instructions that can issue concurrently, or a pair of integer instructions plus a pair of floating point instructions, in each quad.

The instruction and data caches (Icache and Dcache) are the first-level caches and are 8 kB each in size. Since Alpha instructions are 32-b long, this means that the Icache can store 2048 instructions. There is a 96-kB on-chip secondary cache (Scache) which caches both code and data. Loads from the Scache take a minimum of eight cycles to complete, depending on the state of the memory management hardware in the processor. The tertiary cache (Bcache) is external memory (off-chip memory) of several megabytes managed by the processor. One load from the off-chip memory is equal to a 32 byte data which is called a cache line. Loads from the Bcache can take a long time. While the Bcache uses 21ns memory, the total time to load a 32B cache line is 44 ns.

The smallest packet in the Internet is 40 B which is 320 bits. In order to achieve 10 gigabit per second wire speed, we need to forward a packet in 32 ns or forward 31 million packets per second (MPPS) in the worst case.

Figure 2.5 shows the growth of entries in the forwarding table [79]. In 2003, the largest table had more than 140 k entries. If a route entry takes 64b, we need 1120kB memory to store the forwarding table.

An Internet packet consists of payload and header fields. The header fields correspond to the Internet protocol layers: application layer, transport layer, network layer and link layer [51, 14]. The traditional Internet only used the network layer

Figure 2.5: The growth of IP forwarding table.

header, i.e. IP destination address to provide best effort packet forwarding service. The Internet now provides more sophisticated services which involve multiple header fields checking. This is called packet classification. Algorithms for IP address lookup are special cases (one dimension case) of algorithms for packet classification (multiple dimension case). Most packet classification algorithms are based on techniques for IP address lookup algorithms. So we will discuss IP address lookup algorithms first and then packet classification algorithms. In order to understand IP address lookup algorithms, we need to understand how the IP addresses are organized. We

next discuss the IPv4 and IPv6 Internet addressing schemes.

2.1.2 IPv4 Addressing Architecture

IPv4 addresses are 32 bits long. It is represented in dotted-decimal notation: 32 bits are divided into four groups of 8 bits with each group represented as decimal and separated by a dot. For example, 134.117.87.15 is a computer IP address at Carleton University. Sometimes we use the binary or decimal representation of IP addresses for convenience. An IP address is partitioned into two parts: a constituent network prefix (hereafter we call it prefix) and a host number on that network. For example, 134.117.87.15 has two parts: 134.117.87 is the network part; 15 is the host part. 134.117.87 is the prefix usually represented as 10000110 01110101 01010111* (The blanks are added for the purpose of easy reading). How to decide the prefix and a host number is called the addressing scheme. There are two schemes: the classical IP addressing, and the Classless Inter-Domain Routing (CIDR) IP addressing [2]. Nowadays, only CIDR IP addressing is used.

The Classical IP Addressing Architecture

The simplest classical network prefix is the Class A, B, C, D, or E network prefix. These address ranges are discriminated by observing the value of the most significant bits of the address. The address is broken into simple prefix and host number fields. Class A has 8 bits as prefix with 0 as the most significant bit. Class B has 16 bits as prefix with 10 as the most significant bits. Class C has 24 bits as prefix with 110

as the most significant bits. Class D has 28 bits as prefix with 1110 as the most significant bits. Class D is used for IP multicast. Class E with 1111 as the most significant bits is reserved for experimental uses. So, in classical IP addressing, 134.117.87.15 (10000110 01110101 01010111 00001111) is a Class B address with prefix of 16 bits: 10000110 01110101*. Its most significant bits are 10.

The Classless Inter Domain Routing (CIDR)

The classical IP addressing only allows four different prefix lengths: 8, 16, 24 and 28, corresponding to classes A, B, C and D, respectively. This scheme is neither efficient nor flexible in using the IP addresses. There are 16M host addresses in an A class network and there are 64K host addresses in a B class network. If an organization needs 1M host addresses, then an A class network is too big and a B class network is too small. The CIDR changes this situation by allowing a network address to have any length between 0 and 32 (In reality, we seldom see a prefix with a length between 0 and 7 or 31).

The CIDR uses a notation to explicitly mention the bit length for the prefix. Its form is “IP address/prefix length.” For example, 134.117.87.15/24 means that 134.117.87 is for the network and 15 is for the host. 134.117.87.15/22 means that 134.117.84 is for the network and 3.15 is for the host (We need some calculations here). Essentially, we have 22 bits as prefix and 10 bits as host. So 134.117.87.15 (10000110 01110101 01010111 00001111) is divided into two parts: 10000110 01110101 010101* (134.117.84) and 11 00001111(3.15).

Sometimes, we use a mask. For example, 134.117.87.15/255.255.255 is equal

to 134.117.87.15/24, since the binary form of 255.255.255 is 24 bits of 1s (note that binary form of 255 is 11111111). 134.117.87.15/255.255.253 is equal to 134.117.87.15/22, since the binary form of 255.255.253 is 22 bits of 1s.

We can look at the prefix from other perspective. The IPv4 address space is the set of integers from 0 to $2^{32} - 1$. The prefix represents a subset in the IPv4 address space. The longer the prefix is, the smaller the subset is. For example, 10000110 01110101 010101* (length 22) has 2^{10} IP addresses in it; 10000110 01110101 01010111* (length 24) has only 2^8 IP addresses in it. We next explain the concept of more specific address. For example, we have two prefixes 110* and 11000*. We can see 11000* is longer than 110* and the first three bits of 11000* is the same as that of 110*. In this case, we say 11000* is more specific than 110*.

An IP forwarding table consists of destination prefix and next-hop address (or port number). Table 2.1 is an example of the forwarding table. The IP address lookup task is, given the destination address of a packet, to find the most specific prefix it belongs to and then index to the corresponding port. The more specific a prefix is, the longer it is. So the IP address lookup is looking for the longest matching prefix.

2.1.3 IPv6 Addressing Architecture

The research of next-generation Internet protocol (IP) IPv6 [16] was triggered by solving the IPv4 address space exhaustion problem among other things. In IPv6, the IPv6 addressing architecture [31] is used.

An IPv6 address is 128 bits long. An IPv6 address is represented

Table 2.1: A mini forwarding table (address length=5).

Prefixes	Ports
0*	a
01000	b
0111*	c
1*	d
100*	e
1100*	f
1101*	g
1110*	h
1111*	i

as text strings in the form of `x:x:x:x:x:x:x`, where the `x`'s are the hexadecimal values of the eight 16-bit pieces of the address. For example, `"FE0C:BC98:7654:A210:FEDC:B098:7054:3A10."` Another example is `"2070:0:0:0:8:80:200C:17A."` Note that it is not necessary to write the leading zeros in an individual field. We can use `::` to indicate multiple groups of 16-bits of zeros and to compress the leading and/or trailing zeros in an address, but it can only appear once in an address. For example, `"2070:0:0:0:8:80:200C:17A"` may be represented as `"2070::8:80:200C:17A."` `"0:0:0:0:1:0:0:0"` as `::1:0:0:0` or `"0:0:0:0:1::"` but not as `:::1::` When dealing with a mixed environment of IPv4 and IPv6 nodes, the form of `x:x:x:x:x:d.d.d.d` is used, where the `x`'s are the hexadecimal values of the six high-order 16-bit pieces of the address, and the `d`'s are the decimal values of the four

low-order 8-bit pieces of the address (standard IPv4 representation). For example, “0:0:0:0:0:ABCD:129.140.50.38” or in compressed form “::ABCD:129.140.50.38.”

A form similar to the CIDR notation for IPv4 addresses is used to indicate the network part of a IPv6 address. For example, “1200:0:0:CD30:1A3:4567:8AAB:CDEF/60” means that the first 60 bits are the network part and the other 68 bits are the host part. It also represents a 60 bits length prefix.

2.1.4 Background for Packet Classification

Internet service providers (ISP) are expected to provide more value-added services to end users. Examples of these services are firewall packet filtering [4], policy routing [35], virtual private network (VPN) implementations [25], traffic billing [20] and quality of service (QoS) applications such as integrated services (IntServ) [8] and differentiated services (DiffServ) [6]. An example of policy routing is sending all traffic arriving from network A and destined to network B via a particular port. In this case, we need to check the IP source address in addition to the IP destination address of a packet. In order to provide these services, the router needs to classify the packets into flows according to different criteria. These criteria form rules that are based on L2/L3/L4/L5 fields in the packet header. In IPv4, the possible fields are the destination address (32 bits), the source address (32 bits), the protocol field (8 bits), the destination port (16 bits), the source port (16 bits) and TCP flags (8 bits), etc. For example, the protocol field can be the tcp or udp. If the destination port is specified as range 20-21, it represents ftp application. (20 is the port

number for ftp data transfer, 21 is the port number for ftp control). A rule such as [135.207.*; 12.*; 1024-65535; 20-23] may be used to allow IP addresses within AT&T Labs Research to contact IP addresses within WorldNet either via ftp, ssh, or telnet [23].

2.1.5 Performance Metrics for Lookups and Packet Classification Algorithms

To develop and discuss an IP address lookup or packet classification algorithm, performance metrics need to be specified. Some of the performance metrics are briefly discussed as follows:

- Search times. This metric gives the number of memory accesses for a table lookup in the worst case. The measurement in the average case is not as important as in the worst case. The reason is that there is no queue for the IP address lookups/packet classification function. The queues are put in the switching function, which is done after the lookup or classification has been done. Since there is no queue for the IP address lookups/packet classification, it does not make sense to take measurements in the average case. (However, if we use some special techniques, such as cache, it makes sense to take measurements in the average case, or by the hitting rate.) This metric reflects the search speed of the lookup algorithm. The Internet is targeting to achieve 40Gb/s wire speed in the near future. Given 40B the shortest packet size (worst case), we need to lookup 125 million packets per second. That amounts to less than (in addition to lookup, packet needs other processing)

8ns (nanosecond = 10^{-9} second) for a packet lookup. Nowadays, one access to on-chip memory needs 1-5ns for SRAM and 10ns for DRAM; One access to off-chip memory needs 10-20ns for SRAM and 60-100ns for DRAM. From this figure, we can see if we use on-chip memory to store the table, it only allows several memory accesses for a lookup in order to achieve 40Gb/s wire speed. Off-chip memory cannot achieve 40Gb/s wire speed, since one memory access needs 10-20ns for SRAM and 60-100ns for DRAM which is higher than 8ns. The above analysis shows that it is very difficult for algorithms to achieve ideal wire speed. Integrating a parallel or pipeline mechanism into the algorithms is a must for the future lookup algorithms.

- Memory storage requirements. Nowadays, the on-chip memory can be as large as several hundred KB and the off-chip memory several MB. The IP address lookup table can have more than 140k entries (Figure 2.5). Packet classification table can be small. Reference [75] analyzes 793 packet classification tables and finds that only 0.7% of the tables contain more than 1000 rules, with a mean of 50 rules. Ideal algorithms need to achieve linear space complexity with respect to the size of table.
- Update speed. The Internet is a dynamic system. New networks can be added to the Internet. Old networks can be pulled down. The route from one network to another network can be changed due to various reasons. Corresponding to these changes, the entries in the IP address lookup table need to be updated. The IP address lookup table may have hundreds of updates per second [39]. Algorithms need to adapt to these changes. Algorithms can create the table

from scratch or incrementally add or delete entries in the table.

- Scalability. IP address lookup algorithms must be scalable with respect to the number of prefixes. IP address lookup algorithms must also be scalable to IPv6. In addition to these, packet classification algorithms must be scalable with respect to the number of fields.

2.1.6 Notation for Algorithm Complexity

As in algorithm theory, we can use asymptotic notation to measure the complexities of lookup algorithms. In lookup algorithms, we assume that the table size is N . Performance metrics of an algorithm are expressed by asymptotic functions of N , which are defined as follows.

$O(f(N))$ implies that there exists a k and an N_0 such that the complexity is less than $kf(N)$ for $N > N_0$. Sometimes, we call $O(f(N))$ big O notation. This is the frequently used notation. $o(f(N))$ implies that there exists a k and an N_0 such that the complexity is greater than $kf(N)$ for $N > N_0$. $\Theta(f(N))$ implies that there exists k_1 , k_2 and N_0 such that the complexity is between $k_1f(N)$ and $k_2f(N)$ for $N > N_0$.

For example, if an IP algorithm uses $O(N)$ memory, and N is the table size, then with table size of 1000, we can figure out that the size of the memory for this algorithm is around several KB. If another IP algorithm uses $O(N^2)$ memory, we can estimate that we need a memory of size of several MB for a table of size of 1000 IP lookup addresses.

The asymptotic notation is a coarse measurement of an algorithm. Sometimes

the value of the constant involved is very big. So it just gives an idea of the complexity of the algorithm. For IP address lookup and packet classification, the search time is very crucial, so we usually count the exact number of memory accesses for an algorithm. Sometimes, we may take advantage of hardware functions such as cache memory, CPU instruction system, etc. In that case, we need fine measurements for the complexity of the algorithms. The latency to access the memory is much higher than that to access the cache. One multiplication operation takes more time than one addition operation. Therefore, we need to distinguish memory access and cache access, addition operation and multiplication operation, etc. in the algorithm. Whenever possible, we should implement the algorithm for a test.

2.2 IP Address Lookup Algorithms

This section introduces IP address lookup algorithms, the related data structures and various techniques involved.

2.2.1 Naive Algorithms

The *address indexing* and *linear search* are two very simple algorithms. In the address indexing, an array is created as large as the address space. The size of the array is 2^{32} for IPv4 and 2^{128} for IPv6. Each IP address is used as an index to the array. So we need only one memory access for a lookup. Update is the same as lookup except that lookup is a read operation and update is a write operation. So the update is also fast and needs only one memory access. Table 2.2 is an

Table 2.2: Indexing method(address length=5).

Index	Address	Port
0	00000	a
...		...
7	00111	a
8	01000	b
9	01001	a
...		...
11	01011	a
12	01100	c
...		...
31	11111	i

illustrative example if the address length is five. This algorithm gives one extreme situation of the IP address lookup algorithms: lower bound for lookup and update (one memory access) and upper bound for memory storage $O(2^W)$ (W is the length of IP addresses).

In the linear search, we store all the prefixes (not IP addresses) in an array. When lookup occurs, we use the packet IP address to match the prefixes in the array one by one. Remember the longest matching prefix thus far until we exhaust all prefixes in the array. This algorithm needs $O(N)$ memory and $O(N)$ memory accesses for a lookup if the size of the array is N . Update is the same as lookup except that lookup is a read operation and update is a write operation. So an update also needs $O(N)$

memory accesses. This algorithm gives another extreme situation of the IP address lookup algorithms: upper bound for lookup and update ($O(N)$ memory access) and lower bound for memory storage ($O(N)$).

2.2.2 Trie

Most IP address lookup algorithms use *trie* data structures or its variants. This section gives a detailed explanation of trie.

Tree is one of the most important nonlinear structures arising in computer algorithms. Reference [37] uses a recursive definition that is close to computer implementation. The definition is as follows. A tree has a root. The root has several (can be 0) trees (subtrees) as its children. In each subtree, there exists a root and its own children, and so on. Graphically, tree can be expressed as nodes and links. The nodes that have no child are called leaves or external nodes. The nodes that have at least one child are called internal nodes. A tree with its nodes having at most M children is called an M -ary tree. Specially, a tree with its nodes having at most 2 children is called a *binary tree*. Trie [38, 24, 50] is a tree structure to store keys in order to search the keys afterwards. The name trie comes from the word retrieval. The keys can be binary digits (e.g. IP address) or words of English composed of 26 the alphabet characters. A trie storing binary digits is a binary tree. A trie storing words of English is a 26-ary tree. The way to store a key is as follows: starting from the root, we get the first digit or character of the key as the branching decision. For example, if the first digit of the key is 0, we go left; if the first digit of the key is 1, we go right. We proceed the same way until we exhaust the digits or characters of

Figure 2.6: A binary trie.

the key. At that point, we arrive at a node in the tree. Thus, the key is stored on the links that go from root to this node. This node can store information related to the key, such as next hop if the key is an IP address. Figure 2.6 is a trie example for the IP forwarding table in Table 2.1. The same procedure is used to search a key.

Two types of tries will be discussed below: binary tries and multibit tries. We will consider the trie under the context of IP address lookup applications.

Binary tries

The simplest trie is the binary trie. In a binary trie, each node has at most two children. For IP address lookup, we essentially do a sequential prefix search bit by bit. For an IPv4 address, at most 32 steps are needed for a lookup. For an IPv6 address, at most 128 steps are needed for a lookup. This is slow and not suitable for practical use.

We sum up the performance measurements as follows. Assume that W is the length of address. In the worst case, the search complexity is $O(W)$, and the update complexity is also $O(W)$. The memory complexity is $O(WN)$. This is because we have N prefixes and each prefix spans at most $W + 1$ nodes. Therefore, we have at most $(W + 1)N$ nodes. However, this upper bound is very loose, for some prefixes may share the same nodes.

We can use a technique called path compression to reduce trie memory. The idea is to remove all nodes that have only one child and use an extra field to indicate how many bits to skip in a search. The compressed trie is called PATRICIA [47]. Figure 2.7 is the compressed trie of Figure 2.6. The numbers beside the nodes indicate the position of bit in IP address to check. Since some bits are skipped, the prefixes themselves must be stored in the nodes to compare with the key. References [63, 18] are example implementations of the PATRICIA structure. The PATRICIA is efficient when the nodes in the trie are sparse. However, it does not reduce the lookup steps in the worst case.

Multibit Trie

In the binary trie, we check one bit at a time in the IP address lookup. Instead of one bit at a time, we may check several bits at a time in the IP address lookup. Corresponding to this idea, we create a multibit trie structure to facilitate this search. Figure 2.8 is a multibit trie created from the forwarding table in Table 2.1. The root has 4 children. For lookup, we essentially check the first two bits of the key to make a branching decision. So the number of bits checked at a time is called

Figure 2.7: A compressed trie.

stride. We note that prefix 0^* has been expanded to 00^* and 01^* and prefix 1^* to 10^* and 11^* . This is called *prefix expansion*. Due to the expansion, the memory is potentially increased. The multibit trie is divided into levels. In each level, all nodes may have the same stride. In that case, the multibit trie has the fixed strides. Otherwise, it has varying strides. Comparing with binary trie, the multibit trie is doing the tradeoff between memory size and memory access times. The bigger the stride (i.e. the number of children of the node) is, the bigger the memory size is and the less the number of memory access is. For example, if we use a two-level multibit trie with each level having fixed stride of 16 to store an IPv4 forwarding table. We need only two accesses to the memory for a lookup. However, the memory could increase rapidly. For prefix of length one, we need to expand it to the $2^{16} = 64k$ prefixes, though many of them are duplicate prefixes in the original forwarding table.

Multibit trie has received extensive study in the literature [56, 57, 48, 17, 67, 30, 15, 32, 72, 13]. Reference [67] gives a general analysis of optimally choosing

Figure 2.8: A multibit trie.

the number of levels and the strides for nodes. Other papers are special cases or improvements of [67]. However, different aspects are emphasized in these papers. New techniques such as data compression of the trie structure are proposed in these papers.

2.2.3 Prefix Range Search

Prefix represents a range of addresses. For example, in the 5-bit-length addresses, the prefix 100* represents the addresses between 10000 and 10011. We use the left close right open interval $[16, 20)$ to represent these addresses. We call 16 and 20 the

Figure 2.9: Range representation of prefixes.

endpoints of prefix 100^* . This definition of endpoints is different from that in [55]. Each prefix can be converted to two endpoints. For any two prefixes, their intervals either contain one another or are disjoint. They cannot overlap with each other. Two prefixes may share one same endpoint. For example, prefix 101^* has endpoints 20 and 24. It shares the endpoint 20 with prefix 100^* . Therefore, N prefixes can be converted to at most $2N$ different endpoints. Figure 2.9 is the mapping of prefixes in Table 2.1 to endpoints and intervals. The endpoints are $\{0, 8, 9, 14, 16, 20, 24, 26, 28, 30, 32\}$. The smallest endpoint 0 and the largest endpoint 32 can be omitted. We define the interval $[a, b)$ of two consecutive endpoints as a *prefix range*. All addresses in the same prefix range share the same next hop address. So we can lookup the endpoint to find the next hop of an address.

Figure 2.10: A range search tree.

We use the range search tree for the IP address lookup. Figure 2.10 is the range search tree created from the set of endpoints $\{8, 9, 14, 16, 20, 24, 26, 28, 30\}$. The procedure to create the range search tree is as follows. We first sort the endpoints. Then we choose the value in the middle position. If the number of endpoints is even, we are free to choose one of the two values in the middle. This value is stored in the root of the tree. Thus, the original set of endpoints is divided into two parts. One set consists of the endpoints that are smaller than the value chosen; the other consists of the endpoints that are bigger than the value chosen. We choose the middle position values from the small value group and from big value group and

store them to the left child and right child of the root respectively. We recursively follow the same procedure until all the sets of endpoints consist of only one endpoint. In order to facilitate the programming, we copy all the nodes that have one child as one of their own children. In this way, all nodes in the tree have two children except the leaves. For example, in Figure 2.10, the node with value 9 has a node with value 9 as its child. Thus we can store two next hop addresses in the leaf, one for those addresses whose values are smaller than the value stored in the leaf and the other for those addresses whose values are bigger than the value stored in the leaf. The IP address lookup is simple. Starting from the root, we compare the IP address key with the value stored in the node. If the key is smaller, we go to the left child; otherwise, we go to the right child. Thus the search complexity is $O(\log(2N))$, where N is the number of the prefix and the base of the logarithm is 2. Storage complexity is $O(N)$. Update complexity in [41] is $\Theta(N)$. Refer to [41] for details of the update algorithms. The complexities of a range tree only depend upon the number of prefixes rather than the length of the IP address; therefore, the algorithm is scalable to the IPv6 addresses. Using more complicated techniques, the search complexity can be reduced to $O(\log W)$ [78], where W is the length of address.

Corresponding to the multitrie, we can use a multiway range tree [41, 71]. The algorithm exploits the cache lines of memory access. The number of bytes the processor gets from the memory for one access is called *cache line*. The cache line size is 32 bytes for most of the processor. If we use 4 bytes to store the value, we can get 8 values for one memory access. Thus, we can store more than one value in the range tree nodes so that the total number of memory accesses will be reduced. Figure 2.11 is the 4-way range tree of Figure 2.10. In the worst case, 2

Figure 2.11: A multiway range tree.

memory accesses are needed (plus one access to the next hop address). The search complexity is $O(\log_k N)$, where k is the branch of the node. Storage complexity is $O(N)$. Update complexity is $O(N)$.

2.2.4 Binary Search on Prefix Lengths

Before giving an introduction to the binary search on prefix length algorithm [75, 78, 5], we introduce the *hash table*.

A *hash table* is a data structure to store a set of elements. Here we assume the elements are integers between 0 and $2^5 = 32$. For example, 0111, 1100, 1101, 1110, 1111 is a set of elements in binary form. The corresponding decimal form is 7, 12, 13, 14, 15 (as in Table 2.3). With each hash table there is a hash function [33] that

Table 2.3: A hash table.

x	$x(\text{mod})5$
0111*	2
1100*	2
1101*	3
1110*	4
1111*	0

is used to map the elements into the indices of the hash table. For example, we can use $x(\text{mod})y$ as a hash function where x is the element and y is the size of the hash table. Table 2.3 is the result of the mapping, where $y = 5$. For example, in order to lookup 13, we evaluate $13(\text{mod})5$ which is 3. Thus 3 is the index of memory where all information related to element 13 is stored. It is possible that two different elements map to the same index (In Table 2.3, both elements 0111 and 1100 map to 2). This is called *collision*. Refer to [38] for the hash and collision resolution.

We first introduce the linear search on prefix lengths algorithm. The idea is that we group the prefixes according to their lengths. Thus, for IPv4 address, we can have at most 32 groups. For each group which has more than one member, we create a hash table. For IP address lookup, we start from the group with the longest prefix length. We do the hash lookup. If there is a match, we find the longest matching prefix. If there is not a match, we proceed to the next longest prefix length group to do the hash lookup.

Table 2.4 is an example of grouping prefixes in lengths. Assume we need to

Table 2.4: Grouping prefixes in lengths.

Prefix length	1	3	4	5	6	7
Prefixes	0*	100*	0111*	01000*	011111*	0000001
	1*		1100*	11100*	111000*	1000000
			1101*		111110*	1111100
			1110*			
			1111*			

search a match to 0100001. We first use 0100001 as the key to search the hash table associated with the group numbered 7. We do not have a match. We continue to use 010000 as the key to search the hash table associated with the group numbered 6. We miss again. We use 01000 as the key to search the hash table associated with the group numbered 5. We find a match 01000. Therefore, 01000* is the longest prefix match for 0100001. For IPv4 address lookup, we need at most 32 hash table lookups. The complexity of hash table lookup is $O(1)$, and storage complexity is $O(N)$. Dynamic hash tables [42] can be used if updates are frequent.

Instead of linear search on prefix lengths, we can do the binary search on prefix lengths. Using the prefixes in Table 2.4, the binary search tree on prefix lengths is created in Figure 2.12. We do the hash lookup from the group with middle prefix length. If we find a match, we can conclude that the longest matching prefix cannot be in those groups with a shorter prefix length. Therefore, we only need to do the hash lookup from the groups with longer prefix length. The problem is that if we do not have a match in the current group, we cannot say that the longest matching

Example search: 0100001

Figure 2.12: Binary search on prefix lengths.

prefix is in those groups with shorter prefix lengths. Reference [75] uses the *Markers* to solve the problem. Recently, reference [36] proposes some improvements to the binary search on the prefix lengths algorithm.

In binary search on prefix lengths, each lookup can reduce the search by half the number of groups. Therefore the search complexity is $O(\log W)$. Storage complexity is $O(N \log W)$. Since expensive precomputation is used, update is difficult.

2.2.5 Ternary CAM

CAM (Content Addressable Memory) [12, 45] is different from the conventional memory. The key words are stored in CAM directly. Search is performed by comparing the IP address and the keys in parallel. CAM is efficient in storing and retrieving data.

Ternary CAM (TCAM) [73] is similar to CAM in technique. The difference is that the ternary CAM has three bit values: 0, 1 and *. The value * is the don't-care value. It matches both 0 and 1. In TCAM, the memory is divided into cells. Each cell contains the key word and mask. Figure 2.13 is an example of the TCAM. The TCAM has four cells. In the first cell (the first column), 01000 is the key word; 11000 is the mask word. Data is stored in the first available free cell. We can access the cells in parallel. For example, in order to search 10100, we just broadcast 10100 to all the cells. In each cell, 10100 is masked by the mask word and compared with the key word. The result of the search is the output. Multiple matches could happen. So there is a priority decoder to choose the highest priority output. The output is used as an index to the address-based memory to retrieve the information related to the key.

TCAM introduces the parallel searches in the memory access. The data storing and retrieving is very fast. There is no memory access bottleneck as in the address-based memory. However, the volume of the memory is rather small and it is expensive. The power dissipation is rather high. Compared with on-chip SRAM, the memory access latency of TCAM is a bit higher.

In practical use, the prefixes are grouped according to the length. The prefixes

Figure 2.13: TCAM.

with same length are stored in adjacent cells. The update may incur many memory writings. Reference [62] proposes an algorithm to perform fast update in TCAM.

2.2.6 Other Algorithms

Reference [44] uses priority search trees [46] data structure for range search and update. The complexity is $O(\log n)$ for both search and update.

Reference [13] proposes using caching for IP address lookup. Usually, several packets may form a traffic flow where the packets are destined for the same address.

Therefore, when we see a packet destined for an address, there is high possibility that the following packets will also be destined for the same address. Thus, it is beneficial to cache the destination address of the first packet. There has been some debate about the applicability of caching on backbone routers because there are a large number of flows in a backbone router. However, the results reported in [13] demonstrated that caching is very effective for edge routers.

While most of the papers consider the worst case performance, reference [11] uses the average case performance. It presents a general framework for generating optimal tables for table-driven IP routing. A cost model that models the access times and sizes of the hierarchical memory structure of the processor is formulated. Specifically, each prefix has an associated probability and each prefix may be stored in different layers of memory in the hierarchical memory structure, which has different memory access latency. Assuming independent IP packet arrival, the average prefix retrieval time is formulated in terms of the three parameters. A dynamic programming algorithm is used to find an optimal solution.

2.3 Packet Classification Algorithms

IP address lookup checks one field (IP destination address) in the header of packets. Packet classification may check more than one field in the header of packets. IP address lookup is a special case of packet classification. In IP address lookup, keys are prefixes, a special set of IP addresses. The goal of a search is to find the longest matching prefix. In packet classification, not only is the prefix allowed, but also an

arbitrary range is allowed. Here a range refers to a set of consecutive integers. In this thesis, ranges are expressed using intervals. For example, the range $\{4, 5, 6, 7\}$ can be written as $[4, 7]$ or $[4, 8)$. We define 4 and 8 as the endpoints of the range $\{4, 5, 6, 7\}$. Note that 7 is not an endpoint. For convenience, sometimes, the largest IP address is also regarded as an endpoint. We discuss the relationship between prefixes and arbitrary ranges first before we study the packet classification algorithms.

2.3.1 The Relationship between Prefix and Arbitrary Range

From above, we know that each prefix can be converted to two endpoints. A set of N prefixes can be converted to at most $2N$ endpoints. Now, we first discuss the conversion of one range to a set of prefixes and then the conversion of N ranges to a set of prefixes.

We consider the range in $[0, 2^W - 1]$. A range can be represented by a set of prefixes. The maximum number of prefixes needed is $2W - 2$ [23]. For example, for $W = 4$, the range $[1, 14]$ can be represented by prefixes 0001, 001*, 01**, 10**, 110* and 1110 [27]. This is because $0001 = \{1\}$, $001* = \{2, 3\}$, $01** = \{4, 5, 6, 7\}$, $10** = \{8, 9, 10, 11\}$, $110* = \{12, 13\}$ and $1110 = \{14\}$.

We assume that $[0, 2^W - 1]$ is divided into N non-overlapped ranges by $N + 1$ endpoints (including 0 and $2^W - 1$). We may think that up to $N(2W - 2)$ prefixes are needed to represent the N ranges. In fact, this is not the case. Reference [23] points out that at most $2N + 2$ prefixes are needed.

We give an example to show how to convert ranges to prefixes. Assume $W = 4$.

Figure 2.14: Convert ranges to prefixes.

The ranges are $[0, 4)$, $[4, 6)$, $[6, 11)$, $[11, 14)$ and $[14, 15]$. Therefore, the endpoints are $\{0, 4, 6, 11, 14, 15\}$ (remember the definition of endpoints). The endpoints include 15 because 15 is the largest integer when $W = 4$. The endpoints are changed to binary representation. We build a trie of them as in Figure 2.14. Find all the nodes with two children as circled in Figure 2.14. If u is the binary string from a root to a circled node, then $u0^*$ and $u1^*$ are in the prefixes to represent the ranges. Since there are five circled nodes, ten prefixes are produced. They are $\{0^*, 1^*, 00^*, 01^*, 010^*, 011^*, 10^*, 11^*, 1110, 1111\}$ which are the prefixes that the ranges are converted to.

This relationship is important. With this result, any prefix algorithm can be used to solve range problems. The complexity is only increased by a factor of two.

So, in the remainder, we can use prefix instead of range whenever it is convenient.

2.3.2 Classifier and its Geometry Representation

There are many packet classification algorithms in the literature. Notation may be different in different papers. So it is necessary to clarify notation in this thesis.

Packets are classified into flows according to some policy or route information. The policy is specified using fields in the header of packets. The specifications of fields are called *rules*. So flows are specified by rules applied to incoming packets. A collection of rules is called a *classifier* [27]. A classifier is the counterpart of the lookup table in the IP address lookup. Each rule usually consists of several fields. Each field is a prefix or range. Each rule also has a priority number. Usually rules are sorted according to their priorities. The priority is needed, since a packet may match more than one rule. In that case, the rule with the highest priority is chosen.

Reference [27] gives a survey of packet classification algorithms. In this thesis, we will discuss algorithms from the perspective of methodology. Algorithms using a similar method are introduced in the same section.

We will use Table 2.5 as an example classifier to explain algorithms. The smaller the rule number is the higher the priority of the rule is. So rule one has the highest priority. We assume that the width of the fields is 4 bits. We can treat the rules in the classifier as rectangles in the $2D$ space. The geometric representation of the classifier is shown in Figure 2.15. This is obtained by projecting field1 to the horizontal axis and projecting field2 to the vertical axis. With this representation, any packet is a point in the space. To classify a packet is the same thing as to find

Table 2.5: An example classifier.

Field1	Field2	Rule
00*	00*	1
011*	00*	2
00*	111*	3
*	0111*	4
1000	*	5
11*	11*	6
*	*	7

the rectangle that contains the packet point.

With the help of Figure 2.15, we discuss the storage requirement of the geometry representation. We assume that there are N rules in a classifier. In Figure 2.15, we have seven rectangles representing seven rules in the classifier. For example, Rule one has 00* and 00* as its fields. The range of the fields is [0000, 0011] and [0000, 0011] in binary. So rule one shows at the left low corner of the diagram. Please note that in the diagram, the coordinate represents a unit interval rather than a point. We need at least $O(N)$ memory to store them. This is the lower bound. However, reference [29] points out that there is 8% redundancy in the classifiers they studied. Reference [19] uses a technique to make an optimal forwarding table, therefore reducing the number of entries in the forwarding table. In this way, memory requirements can be reduced.

Furthermore, we can see from Figure 2.15 that the rules can overlap. This

Figure 2.15: Geometric representation of a classifier.

makes packet classification more complicated. Many algorithms divide a rule into several small rules if it overlaps with other rules. For example, R_5 in Figure 2.15 is divided into three rules due to R_4 . The three new rules are represented by the three rectangles that are below, above and overlapped with R_4 . Rules with higher priority will not be divided due to lower priority rules. So R_4 is not divided. In the worst case, a rule can be divided into N small rules. In the extreme, we divide the rectangles according to the intervals projected in the x -axis and y -axis. Any combination of an x -axis interval and a y -axis interval forms a rectangle. We have six intervals along the x -axis and five intervals along the y -axis. Thus, we obtain

30 small rectangles that do not overlap with each other. With the division, in the worst case, we need $O(N^2)$ memory for a 2-dimension classifier and $O(N^d)$ memory for a d -dimension classifier. With $N = 1000$, and $d = 4$, we need at least 10^{12} bytes of memory, which is impractical. So any algorithm with this division is only suitable for lower dimensions and small classifiers.

2.3.3 Naive Algorithms

As in the IP address lookup, we have two very simple algorithms for packet classification: the address indexing and linear search. In the address indexing, we concatenate the bits in all fields to form a big address space. For example, if the classifier has four fields: destination address (32 bits), the source address (32 bits), the destination port (16 bits) and the source port (16 bits), then we have an address space as large as 2^{96} . Though we need only one access to memory, we do not have such big memory available. For linear search, we use the smallest memory $O(N)$, but the number of memory accesses $O(N)$ is intolerable, where N is the number of rules in the classifier.

2.3.4 Parallel Algorithms

Since the packet classification is a multi-field extension of the IP address lookup, a natural method is to look up each field in parallel and then the results are combined. Cross-producting [66], Recursive Flow Classification (RFC) [29] and bit vector intersection [40, 1] belong to this category. We discuss the ideas as follows.

The first step is to project rules into the non-overlapping intervals along each field. For example, where Figure 2.15 is a two-dimension classifier, we can project it along field1 and along field2 respectively. We get intervals [0000, 0100), [0100, 0110), [0110, 1000), [1000, 1001), [1001, 1110) and [1110,1111] along field1 and intervals [0000, 0100), [0100, 0111), [0111, 1000), [1000, 1110) and [1110,1111] along field2. Then we assign a best matching rule to each product of these intervals. These would produce N^d entries for a d -dimension classifier in the worst case. Table 2.6 is the resulting table of Figure 2.15. Cross-producting [66] uses this data structure. Query is performed by a range lookup in each dimension, results are combined as an index to the table. The algorithm is not practical for large classifiers for the need of $O(N^d)$ memory.

Reference [40] uses a different structure to store the rules. It associates a bit vector with each interval to indicate if a rule covers the interval or not. If the rule covers the interval, the bit related to it is set to 1, otherwise, 0. The first bit of the bit vector relates to rule one; the second bit relates to rule two, and so on. So the number of bits in the bit vector is equal to the number of rules. The bit vectors that associate to the field1 intervals in Figure 2.15 are: [0000, 0100)-1011001, [0100, 0110)-0001001, [0110, 1000)-0101001, [1000, 1001)-0001100, [1001, 1110)-0001001 and [1110,1111]-0001011. Doing the packet classification, we first search the intervals along all the fields. Then we do the logic-and with all the bit vectors of the intervals. In the resulting vector, the first 1 bit position corresponds to the matching rule. Since we have at most $(2N - 1)$ intervals, each interval has a N -bit vector, for d fields, we need at most $d(2N - 1)$ bits of memory. Compared with the cross-product algorithm, the memory is reduced. However, the time complexity increases to $O(d(t + N/c))$ where

Table 2.6: A product table of a classifier.

Field1	Field2	Rule
[0000, 0100)	[0000, 0100)	R_1
[0000, 0100)	[0100, 0111)	R_7
[0000, 0100)	[0111, 1000)	R_4
[0000, 0100)	[1000, 1110)	R_7
[0000, 0100)	[1110, 1111]	R_3
.....		...
[1110, 1111]	[0000, 0100)	R_7
[1110, 1111]	[0100, 0111)	R_7
[1110, 1111]	[0111, 1000)	R_4
[1110, 1111]	[1000, 1110)	R_7
[1110, 1111]	[1110, 1111]	R_6

c is the number of bits for one memory access and t is the one dimension search time. N/c is the worst case time for fetching the bit vector. Reference [40] also gives an algorithm to reduce the storage requirement at the cost of increasing time complexity. Reference [1] uses aggregation and rearrangement of rules techniques to improve the query time in [40].

Instead of using the range search, we can use the address indexing such as in the RFC algorithm [29]. As we know, the indexing technique always reduces the memory access time at the cost of storage requirement. Another advantage is that it is easy to implement. We introduce the RFC algorithm as follows.

We observe that, in each field, the number of different prefixes is much smaller than the number of addresses in the field. For example, in Table 2.5, the number of addresses in field1 is 16. However, the number of different prefixes in field1 is only 5. We give each prefix an ID number. The ID number is used to index the rule table. Say the number of different prefixes in the fields are $a_1, a_2, \dots, a_d$ respectively. Then the rule table size is $a_1 * a_2 * \dots * a_d$. In addition to that, we need another $2^{f_1} + 2^{f_2} + \dots + 2^{f_d}$ of memory, where f_i is the length of field i . The memory is used to store the address to ID mapping tables. For a query, we use the value in each field as an index to the mapping table to find the ID. The IDs from all the fields are combined as the index to the rule table to find the corresponding rule. We need only $d + 1$ memory access. However, the storage requirement is gargantuan. Next step, we try to trade off the memory access with the storage requirement. We divide the fields into trunks. Several trunks form a group. We perform the same query as described above in a group. This will result in an ID from each group. These IDs are combined to form an index to a last step rule table. With this technique, the memory is reduced, the access time increased. We can recursively use this technique to do the memory and access time tradeoff [29].

2.3.5 Serial Algorithms

Parallel algorithms perform lookup in all fields in parallel. Serial algorithms perform lookup in one dimension first and then based on the result, perform lookup in the next dimension. Exhausting all the dimensions, we will get the result. Grid of tries [66] and Fat-Inverted-Segment tree (FIS tree) [23] are examples of serial algorithms.

Figure 2.16: Trie of tries.

Grid of tries applies to prefixes and the FIS tree applies to ranges (including prefixes).

We can use trie of tries to represent the multiple dimension classifier. The idea is that we use a trie to represent the prefixes in one dimension as usual. For each prefix, we have prefixes from the next dimension combining this prefix to form rules. We create a trie with these prefixes and use a pointer in the prefix to point to it. We continue the process until we exhaust all dimensions. Figure 2.16 is the trie-of tries of the classifier in Table 2.5.

For example, if we want to classify the packet with fields (0111, 0011), we can follow the tries to get the rule R_2 . However, if we want to classify the packet with

Figure 2.18: FIS tree.

structure is called *grid of tries*. However, the switch points can only be used for the last two dimensions, therefore, the time complexity for d dimensions is only reduced to $O(W^{d-1})$.

The FIS tree [23] uses a similar idea to store ranges for two-dimensional classification. The classifier is projected into one dimension, resulting in non-overlapping intervals. For example, in Figure 2.18, the set of non-overlap intervals is $\{I_1, I_2, \dots, I_9\}$. Use these intervals to create an inverted segment tree [52]. Each interval corresponds to a leaf. Several leaves, say t leaves, are grouped together. Each group forms a parent node. Each leaf has a pointer to point to its parent node. t groups also form

a big group. This big group is represented as a parent node with pointers from its children pointing to it. Recursively perform the grouping until we get the last big group that is the root of the tree. In Figure 2.18, $t = 3$. The rules are stored in the nodes. The way to store the rules is as follows. Each leaf associates with an interval. The parent node also associates with an interval which is the union of the intervals associated with its children. A rule is stored in a node if the rule's projected range is not contained in the node's interval but is contained in its parent node's interval. The lower part of Figure 2.18 is the FIS tree derived from the rules above.

In each node, we may store several rules. The endpoints of the next dimension (say y dimension) of these rules are extracted to form a one-dimensional search structure. The endpoints point to rules associated with them. The query is performed as follows. The value of the first field of the packet is extracted. The value is used as a key to search the first dimension intervals to find the interval which it belongs to. The interval points to the corresponding leaf in the FIS tree. From the leaf, we proceed to the root. We use a variable to store the best-matched rule. The variable is first set to the default rule. Along the path from leaf to the root, at each node, we use the value of the second field of the packet as the key to search the y dimension structure stored in the node. If there is a match rule, we compare it with the current best-matched rule. If this rule has higher priority than the best-matched rule, we assign it to the best-matched rule.

2.3.6 Combined Dimensional Search Algorithms

Instead of using the values from the fields as keys for a query, we may combine the digits of these values to form a search key. Area-based Quad Tree (AQT) [9] and HiCuts [28] can be classified into this group. These two algorithms are easily understood from a geometry perspective. We will explain them using a two-dimension classifier.

Assume we have a $2^W \times 2^W$ search space. We divide the search space into four areas where each area has an equal size of $2^{W-1} \times 2^{W-1}$. We number the north-west part as 00, the north-east part as 01, the south-west part as 10 and the south-east part as 11. We recursively divide each part into four smaller parts and number these parts accordingly. We use a 4-way branching tree to represent the decomposition. The root corresponds to the total search space. The root has four children corresponding to the four areas. The numbers are used as branching indices. This is called *quadtree* [60]. Figure 2.19 is an example.

Reference [9] uses quadtree to store two-dimension prefix rules. Each rule consists of two prefixes. According to the property of the prefix, for each rule, we can find a partition area that exactly contains the rule. This means that the length of the longer dimension of the rule is equal to the length of the area. We assign the rules to the areas that exactly contain them. So the rules are stored in the nodes corresponding to the areas. The rules are called the *cross filter set* of the node (see Figure 2.19). A rule crosses an area either from the x direction or y direction. So the cross filter set of a node is divided into two parts: one contains the rules that cross the area from the x direction, the other contains the rules that cross the

Figure 2.19: Area-based quadtree.

area from the y direction. This makes search easier in the area. In fact, we only need to perform two one-dimensional searches in the area. To perform a packet classification, we extract the values from the two fields of the header. The values are represented as digital bits. We successively choose bits from each value to form a decision branch. The search will finish when we reach a leaf of the quadtree. The storage complexity is $O(N)$. The time complexity is $O(W \log N)$. Using fractional cascading [10], time complexity is reduced to $O(W + \log N)$.

HiCuts [28] uses a different scheme to cut the area. At each stage of cutting, it chooses a dimension and a number of parts to cut. They use heuristic to choose these parameters. These parameters are stored in the nodes. The search decision is formed by these parameters. It also has a threshold for the number of the rules. If the number of rules in an area is below the threshold, we will not cut the area further. The query is performed as follows. From the root of the HiCuts tree, we get the cut dimension variable and number of cuts variable. If the dimension variable

indicates the first field to cut, we extract the bits from the first field of the packet according to the value of the number of cuts variable. If the dimension variable indicates the second field, we extract the bits from the second field, and so on. The value of the extracted bits is used as the index to the children. We continue this procedure until we reach a leaf. At this point, we perform a linear search on the rules stored in the leaf.

2.3.7 Tuple Space Search

Tuple space search [65] is the binary search on prefix length algorithms [75] extended to multiple fields. A tuple is a d -dimension integer vector with each element representing the length of the prefix in each dimension. In Table 2.7, the rule $(0^*, 00^*)$ has tuple $(1, 2)$ since the first prefix has length 1 and the second prefix has length 2. We group all the rules with the same tuple. The rules are searched using a hash table. The rationale for the algorithm is the observation that the number of tuples is relatively small compared to the number of rules. The total search is equal to the number of tuple hash table accesses. The storage complexity is $O(N)$ since each rule is stored in exactly one hash table. Techniques exist [65, 76] to improve tuple space search performance.

2.3.8 Ternary CAM

Ternary CAM can be used for packet classification. In order for TCAM to be used, the ranges in any fields must be converted to prefixes. This potentially increases

Table 2.7: Tuples.

Rule	Fields	Tuple
R_1	$(0^*, 00^*)$	$(1, 2)$
R_2	$(1^*, 10^*)$	$(1, 2)$
R_3	$(011^*, 0^*)$	$(3, 1)$
R_4	$(^*, 00^*)$	$(0, 2)$
R_5	$(111^*, 1^*)$	$(3, 1)$
R_6	$(110^*, 1^*)$	$(3, 1)$

memory requirement. On the other hand, two prefixes can be compacted [43]. For example, rule $(00^*, 1010^*)$ and rule $(00^*, 0010^*)$ can be compacted to $(00^*, ^*010^*)$, if the two rules share the same port information.

2.4 Summary

In this chapter, we introduced the background for our research. In particular, we outlined some IP address lookup algorithms and packet classification algorithms. We tried to grasp the essence of these algorithms rather than the details of the techniques. It helps us to get a bird's eye view of the problem. It also reveals the relationship between various algorithms. Thus, it becomes easier to sort out such large numbers of work.

In the next chapter, we present an IP address lookup algorithm using compressed keys. The content first appears in [70].

Chapter 3

An IP Address Lookup Algorithm

As introduced in the previous chapter, Internet routers need to forward packets at wire speed. Software based algorithms that run on general processors are too slow to meet the speed demand. Although TCAM provides a temporary solution, the problems facing it, such as high power dissipation, expensive price, small memory size and memory access latency being limited by about 10 *ns*, will make them eventually obsolete. This motivated us to develop an algorithm which uses a small amount of memory and can be implemented into a single chip hardware.

In fact, our research represents the industry trend. Industry responds to the aforementioned demand by offloading the processing tasks to *coprocessors* [86]. A coprocessor is a system on single chip to perform a single task. Previously, the general purpose processor was used in the Internet nodes. Recently, Network Processors [91] are gaining popularity in processing Internet data. Network Processors have the flexibility to adapt to protocol changes. The fixed functions such as IP ad-

dress lookup, packet classification, security, traffic management, etc., are offloaded to use single chips. As mentioned above, one of the advantages of an on-chip system is that the on-chip memory access latency is very low. Another advantage of on-chip systems includes a larger bus width to on-chip memory than to off-chip memory. The number of pins of the chip is smaller if the memory goes on chip rather than off chip. The data sheets provided by IBM [84] show that the maximum memory width to DRAM is 1168b and to SRAM is 144b. The number of pins for off chip memory would increase a lot with the same memory width.

The restriction of an on-chip system is that the memory cannot be large. Reference [84] shows that for the embedded DRAM, the maximum macro capacity/size is 72.95 Mb/31.80mm² with random access time of 9.0ns using a Cu-08 process; For the embedded SRAM, the maximum macro capacity is 1 Mb with random access time of 1.25ns using a Cu-11 process.

The algorithm we developed uses a small amount of memory. Moreover the algorithm is inherently suitable for parallel or pipeline implementation. With the merits, the algorithm can be implemented in one single chip to achieve very high performance. Our approach is to convert the longest prefix match problem into a range search problem and use a tree structure to do the search. Our main contributions are that:

1. We compress the keys in a tree node;
2. We use a shared pointer in a tree node;
3. We use bottom-up from the leaves to the root scheme to build the tree.

Figure 3.1: Hardware design reference model.

The rest of this chapter is organized as follows. Section 3.1 gives a hardware design reference model for our analysis. In Section 3.2, we give the concepts and definitions related to the range search problem. Details of our new algorithm are presented in Section 3.3. Results from an experimental study are presented in Section 3.4. In Section 3.5, we highlight comparison results with some existing algorithms. Concluding remarks are made in Section 3.6.

3.1 A Hardware Design Reference Model

Figure 3.1 is a reference model for the hardware design. The left part is a chip and the right part is external SRAM memory. The IP destination address enters the

chip as a key for looking up the next hop information. The output of the chip is an index to the external SRAM where the port information can be found. The chip is an ASIC that consists of a memory system and an ALU part. The memory system uses on-chip SRAM. Using an IBM blue logic Cu-08 ASIC process, the I/O width of the SRAM to the control logic unit can be as wide as 144bits. The memory access time can be as low as 1.25ns. The size of the on-chip memory is about 1 Mbits. Assuming there are 144 bits in each row, the chip has less than 2^{13} rows altogether. This means 13 bits is enough to index into any row of the memory. The ALU receives keys from outside and produces outputs to the outside. It may access the memory system and performs some simple logic and arithmetic operations. According to different design goals, the chip can be configured or programmed. After reading this chapter, we may notice that this reference model may be modified to tailor to different situations.

The reason to choose SRAM instead of DRAM as the on-chip memory is that the SRAM access time is several times faster than the DRAM. Although the memory size is constrained to 1M bits, our algorithm only needs a small amount of memory, which compensates for this drawback.

3.2 Convert Longest Prefix Match to Range Search Problem

The IP address lookup problem and IPv4/IPv6 address schemes are introduced in Chapter two. In this chapter, we use IPv4 addresses as an example to explain the

Table 3.1: A mini routing table (Maximum prefix length=6)

prefix	port	prefix	port
0*	A	10*	C
00010	B	10001*	A
001*	C	1001*	A
01*	C	1011*	D
01000*	D	11*	D
0111*	A	110*	A
1*	B	1101*	B

concept for the purpose of simplicity.

In this section, we will give some more definitions related to converting a prefix to a range of addresses. We use a small artificial routing table in Table 3.1 as an example. We first give the following definition.

Definition: A prefix P represents address(es) in a range. When an address is expressed as an integer, the prefix P can be represented as a set of consecutive integer(s), expressed as $[b, e)$, where b and e are integers and $[b, e) = \{x : b \leq x < e, x \text{ is an integer}\}$. $[b, e)$ is defined as the *range* of the prefix P . b and e are defined as the *left endpoint* and *right endpoint* of the prefix P respectively, or *endpoints* of the prefix P .

For example, for the 6-bit-length addresses, the prefix 001* represents the addresses between 001000 and 001111 inclusive (in decimal form, between 8 and 15 inclusive). $[8, 16)$ is the *range* of the prefix 001*. 8 and 16 are *left endpoint* and *right*

Figure 3.2: Prefixes and their ranges and ports.

endpoint of the prefix 001^* respectively. Readers may notice that the definition of right endpoint is different from that in the literature (e.g. [55]). With this definition, each endpoint can have at least as many trailing zeros as the length of the host part of the IP address. As in the example given above, 16 has four trailing zeros in its binary form while 15 has no zero in its binary form. In real life forwarding tables, most of the prefixes have a length of 24 bits. Thus most of the endpoints have at least 8 bits of trailing zeros. We will exploit this property using the tree structure in the following section.

For any two prefixes, their ranges either contain one another or are disjoint. They cannot overlap with each other. Two prefixes may share at most one same endpoint. For example, in Table 3.1, prefix 001^* has endpoints 8 and 16 and prefix 01^* has endpoints 16 and 32. They share the same endpoint 16. Since each prefix can be converted to two endpoints, N prefixes can be converted to at most $2N$ different endpoints. Figure 3.2 is the mapping of prefixes in Table 3.1 to endpoints and ranges. Notice that 14 prefixes produce 17 endpoints in this example.

Figure 3.3: Converted endpoints with the corresponding ports.

If two consecutive ranges have the same port, the shared endpoint can be eliminated. Hence the two ranges can be merged into one range. For example, since range $[34, 36)$ and range $[36, 40)$ both map to port A, we can merge them into the range $[34, 40)$. We can assign a unique port to each range according to the rule of longest prefix match. We can use an endpoint to represent the range to its right and thus assign the port of the range to the endpoint. For example, let a be an endpoint and b its successor. If port A is assigned to a , it means any address that is in $[a, b)$ is mapped to port A. Two steps are involved to convert the routing table into (endpoint, port) tuples: Step one is to sort the prefixes. The host part of a prefix is filled with 0's and the prefix is treated as an integer. Prefixes are sorted according to the value of the integer. The smaller value prefix is sorted in the front. If two prefixes have the same value, the one whose length is smaller is sorted in the front. Step two is to assign port to each endpoint according to the rule of longest prefix match.

Notice that for simplicity this algorithm does not do the possible merge of endpoints mentioned above. In fact, this can be easily done by using a variable to record the port assigned to the previous endpoint. Whenever there is a port assignment, we check the port with the recorded port. If they are equal, this endpoint will

Figure 3.4: A tree structure.

be eliminated; otherwise, assign the port as usual. Figure 3.3 is the result of the conversion of the routing table in Table 3.1 into (endpoint, port) tuples. With the merging of ports, the number of endpoints is reduced from 17 to 15.

The set of endpoints from Figure 3.3 is $\{0, 4, 6, 8, 16, 24, 25, 28, 32, 34, 36, 40, 44, 48, 52, 56, 64\}$. Usually, 0 and the maximum value 64 can be eliminated. With the above process, the IP address lookup problem is converted into the *predecessor finding* problem in a priority queue [21]. The *predecessor finding* problem is defined as follows. Given a set of sorted integers S and an integer i , find an integer e in S such that e is the biggest one which is less than or equal to i . One of the solutions is to use a tree structure to tackle the predecessor finding problem. Figure 3.4 is an example of the tree structure created from Figure 3.3. We start the search from the root, then go through internal nodes until we reach a leaf. The leaf has a pointer which leads to the corresponding port information. For example, in order to lookup 39, we search in the root and find that 34 is the the biggest one which is less than

or equal to 39. Following the pointer, we search the third node in the next stage. We find 39 is less than all the stored values. It points to port A.

The data structure we propose is essentially a tree structure, too. However, the details and the techniques are different from that in the literature. In order to be convenient for explanation, we divide the tree into levels. The root is level one. The next stage nodes are defined as level two and so on. The leaves are the last level. There are two levels in Figure 3.4.

3.3 Put the Tree in Memory

In this section, we will explain how to put the range tree structure into memory. This is considered as the main contribution in this algorithm. We will discuss how to compress the endpoints, how to use the shared pointer in a node and how to use bottom up approach to build a range tree in the memory.

3.3.1 Compressed Endpoints and Shared Pointers

Let us take a close look at the endpoints of IPv4 addresses from Table 3.2. This is a set of consecutive endpoints that are taken from a routing table. We can find that all the endpoints have the same head bits “110000001110110” (15 bits) and each has at least eight trailing zeros. This is the source of compression. We can compress the common leading bits and trailing zeros. The more the common leading bits and trailing zeros are, the better the endpoints are compressed. The number of common leading bits is related to the number of endpoints in a forwarding

Table 3.2: Real life IPv4 endpoints.

11000000111011000100000000000000
11000000111011010010000000000000
11000000111011010100000000000000
11000000111011010111001000000000
11000000111011010111001100000000
11000000111011010111010000000000

table. Intuitively, the larger the number of endpoints is in a fixed address space, the larger the number of common leading bits tends to be, since the endpoint space is fixed and the endpoints tend to cluster together when the number of endpoints is large. Human being assigned numbers such as IP prefixes tend to be correlated. For example, consecutive numbers tend to be assigned. So endpoints from real life IP tables tend to have a large number of common leading bits. We will verify this by experimental study.

Assume the endpoints in Table 3.2 are all the endpoints stored in a same node of the tree structure. We create a data structure shown in Figure 3.5 to represent the node. In this data structure, we use two fields to indicate the number of common bits to skip and the number of trailing zeros that each endpoint at least has. So we only need to store the middle bits of each endpoint as a key for the endpoint. In this example, we only need to store a nine-bit key for each endpoint instead of an endpoint of 32 bits. Referring to the hardware design model in Figure 3.1, there are 144 bits in each memory row. Each row is used to store a node of the tree.

internal node or leaf bit	number of keys	number of skip bits	number of trailing zeros	key1 key2 key3	next tree pointer
------------------------------	-------------------	------------------------	-----------------------------	---------------------	----------------------

Figure 3.5: The row data structure for a node of the tree.

With the endpoints compressed, more endpoints can be stored in a node using 144 bits. Figure 3.5 is the data structure for the 144 bits. We explain it in detail in the following. The data structure has six fields. The first field is a one-bit field to indicate whether the node is an internal node or a leaf node. The second field is the number of endpoints stored in this node. If the number of endpoints stored in any node does not exceed 16, then 4 bits are needed for this field. The third field records the number of head bits to skip in the IP address (endpoint) and the fourth field is the number of zeros to ignore at the trail of the IP address (endpoint). In general, for IPv4 addresses, these two fields need at most five bits each. For IPv6 addresses, these two fields need at most seven bits each. We will further discuss how to reduce the number of bits for these fields in later subsections. The next field is for the keys. The last field is the pointer indexing to next level node of the tree structure. We use 20 bits in this field for supporting 500k entry forwarding tables. This is because each entry can produce at most two endpoints and each endpoint has its corresponding port information. Thus the external SRAM which stores port information can have as many as 1M entries in the worst case which can be indexed by 20 bits. To sum up, for IPv4, the first four fields and the last field use $1+4+5+5+20=35$ bits. The leftover 109 bits are used to store as many keys as possible. For IPv6, the first four fields and the last field use $1+4+7+7+20=39$ bits. The leftover 105 bits are used

Figure 3.6: Tree in memory.

to store as many keys as possible. The order of the fields is not important. This is the basic structure, but it can be modified for different variants. For example, the first field may be removed if possible. The pointer field must be at least 22 bits long if we want to support 2M entry forwarding tables. We will mention later that the pointer field in a different level node may have different bits. The 144 bits can be varied.

We may notice that each node has only one pointer rather than many pointers, as in the case shown in Figure 3.4. This is where our shared pointer goes. This scheme saves memory tremendously. As defined in the previous section, the tree is divided into levels. The nodes in each level are stored in ordered consecutive rows such as in Figure 3.6. In this way, all the endpoints in a node can share one pointer. The exact pointer of a subtree can be determined by the position of

the corresponding endpoint. For example, assume a node with sorted endpoints p_1, p_2, p_3, p_4 . Traditionally, we would need five pointers. Here, we only need the pointer that points to the node whose endpoints are smaller than p_1 . Since other nodes are stored in ordered consecutive rows, we can find the node of a search by knowing the position of the destination address in the searched node. For example, if the searched destination address is greater than or equal to p_3 but less than p_4 , we can find the node by adding 3 to the stored pointer.

The search of the keys in a node is carried out in the registers (or using logic circuit). It is very fast compared to memory access. Also the time can be overlapped with memory access. We can also use a binary search in the keys of a node to speed up the search in the registers.

Two functions are needed for endpoints compression: one is for calculating the number of the common leading bits of a group of endpoints and another is for calculating the common trailing zeros of the group. For calculating the number of common leading bits of the group, we only need to calculate the number of common leading bits of the smallest and the biggest endpoints in the group. The number of common trailing zeros can be calculated recursively as follows. The trailing zeros of the first endpoint are counted and recorded as the candidate number of common trailing zeros. Then the trailing zeros of the next endpoint are counted. If the number of the trailing zeros of the next endpoint is smaller than the candidate number, the candidate number is changed to the smaller one; otherwise, the candidate number is unchanged. This operation is performed for the left endpoints until we exhaust all the endpoints. Using this approach to find the number of common trailing zeros of n endpoints, $O(n)$ operations are needed. $O(2)$ operations are needed

to find the number of common leading bits of any sorted group.

3.3.2 Build the Tree from Bottom-Up

Given a set of sorted endpoints, we next discuss how to create a tree structure from it. We adopt a bottom-up approach. First we assign endpoints to leaf nodes and then to the next highest level until the root level. Beginning with the smallest endpoint, we try to store as many endpoints to a node as possible. We use IPv4 for the explanation in the following. From the analysis of the the previous subsection, we know that there are 109 bits to store the compressed keys. Since each endpoint is 32 bits long, we can store at least three endpoints ($3 * 32 = 96 < 109$) even without key compression. Therefore, we initially select the first four endpoints as a group and calculate the total bits of the compressed keys. If the total bits is bigger than 109, then we cannot store these four keys. Instead, we store the first three keys. If the total bits is equal to 109, we store these four keys. Otherwise, if the total bits is smaller than 109, we have potential to store more keys. Thus we need to probe further. We add the next endpoint to the group and repeat the above procedure. In this way, we can find how many keys can be stored in the 109 bits.

Several variants are proposed in the following to describe the details of the tree creation. Specifically, they differ in how to calculate the compressed keys and select endpoints to store in the higher level nodes. They represent some tradeoffs between the memory size and the number of memory accesses.

Variant One

Table 3.3: Calculating the “common leading bits”.

10000000	11001000	01000000	00000000
10000000	11001001	00100000	00000000
10000000	11101101	01000000	00000000
10000000	11101101	01110010	00000000
10000000	11101101	01110011	00000000
10000000	11101101	01110100	00000000
10000000	11101101	01111101	00000000
10000110	11101111	00001101	10000000
11010110	11101111	00001110	00000000
11010110	11101111	00100111	00000000
11011000	11101111	00101000	00000000
11011000	11101111	00110000	00000000
11111000	11110000	00010000	11000000

Let $\{e_1, e_2, e_3, \dots, e_n\}$ be the set of endpoints to be stored in a tree structure. Assume that the first four endpoints $\{e_1, e_2, e_3, e_4\}$ are stored in the first leaf node, then the endpoint $\{e_5\}$ will be stored in the next higher level node. Assume the

next five endpoints $\{e_6, e_7, e_8, e_9, e_{10}\}$ are stored in the second leaf node, then the endpoint $\{e_{11}\}$ will be stored in the next higher level node, and so on.

For this scheme, the endpoint(s) in the next higher level node must be involved in the leaf nodes to calculate the compressed keys. Specifically, in the aforementioned example, two endpoints $\{e_1\}$ and $\{e_5\}$ are involved to find the common leading bits of the first leaf node; $\{e_1, e_2, e_3, e_4\}$ are used to find the common trailing zeros of the first leaf node. Two endpoints $\{e_5\}$ and $\{e_{11}\}$ are involved to find the common leading bits of the second leaf node; $\{e_6, e_7, e_8, e_9, e_{10}\}$ are used to find the common trailing zeros of the second leaf node, and so on. Next, we will explain why the higher level endpoints will be involved in the calculation of compressed keys using a concrete example.

For example, in Table 3.3 (the blank between bits is for convenience of reading) the first seven endpoints are stored in the first leaf node. The next endpoint “10000110 11101111 00001101 10000000” will be stored in a higher level node. The next four endpoints following “10000110 11101111 00001101 10000000” will be stored in the next leaf node. “11111000 11110000 00010000 11000000” will be stored in a higher level node, and so on.

The common leading bits of the first leaf node are “10000” instead of “1000000011”. The number of common trailing zeros is eight. The common leading bits of the second leaf node are “1” (Because “10000110 11101111 00001101 10000000” and “11111000 11110000 00010000 11000000” are involved to calculate the common leading bits.) instead of “1101”. The number of common trailing zeros is eight.

The reason is as follows. Let us assume that we are searching endpoint

“10011111 11111111 11111111 00000000”. This endpoint is greater than the endpoint “10000110 11101111 00001101 10000000” and less than the first endpoint in the second group. If we took “1101” as the leading bits of the second group, i.e. we skipped four bits, we would mistake “10011111 11111111 11111111 00000000” as greater than the last endpoint in the second group.

After constructing the leaf nodes, we proceed to the next level using the same method. The number of endpoints in this level are reduced to approximately N/k , where N is the number of endpoints in the leaf level and k is the average number of endpoints in a leaf node. For $k > 4$, it quickly converges to the root.

As we can see, the cost for constructing the tree structure is not high. Let N be the total number of endpoints. Sorting the endpoints may take $O(N \log N)$ time. However, after the first sorting, we can incrementally add or delete an endpoint which takes only $O(\log N)$ time using binary search. Assigning ports to endpoints takes $O(N)$ time. Creating the nodes takes $O(N)$ time if k is not small, say $k > 4$. Putting them together, we need $O(N)$ time to create the tree structure.

The preferred architecture for a router is to use two banks of memory for IP packet forwarding. One bank is for updating and the other is for searching. One advantage of this architecture is that the updating will not interfere with the searching. With this architecture, our algorithm is fast enough for the updating. This architecture is even comparable to some dynamic data structures in terms of memory utilization. For example, the memory utilization of a basic B-tree [3] is 50% in the worst case, which is the same as the two-banks-of-memory architecture.

A word for IPv6: if the endpoints of an IPv6 address cannot be compressed, we cannot store even one address in a row, since an IPv6 address is 128 bits long

and the key field of our data structure is only 105 bits. In this case, we may allow two rows to store the key with a minor modification of the data structure. IPv6 addresses may have consecutive zeros in the middle rather than in the trail. We may compress these zeros. However, we do not have enough IPv6 routing tables for the experiment, and we will not go into details. Nonetheless, experiments on small IPv6 routing tables are presented in this thesis for the unmodified data structure.

Variant Two

The essential difference between the variant one and the variant two is that a new endpoint is created to store in the higher level node rather than an existing endpoint. We describe the algorithm first, then explain the reason to do so.

We first explain how to create the new endpoints for the high levels. Let $\{e_1, e_2, e_3, \dots, e_n\}$ be the set of endpoints to be stored in a tree structure. Assume the first four endpoints $\{e_1, e_2, e_3, e_4\}$ are stored in the first leaf node and the next five endpoints $\{e_5, e_6, e_7, e_8, e_9\}$ are stored in the second leaf node and the next four endpoints $\{e_{10}, e_{11}, e_{12}, e_{13}\}$ are stored in the third leaf node, and so on. The first endpoint to be stored in the higher level node is simply the common leading bits of $\{e_1, e_2, e_3, e_4\}$ padded with trailing zeros to form a 32 bit endpoint $\bar{e}_1$. This new endpoint will be stored in the higher level node. The second endpoint is created according to e_4, e_5 and the number of common leading bits of $\{e_5, e_6, e_7, e_8, e_9\}$. Let n_1 be the number of common leading bits of e_4 and e_5 ; Let n_2 be the number of common leading bits of $\{e_5, e_6, e_7, e_8, e_9\}$. Let $n_3 = \max\{n_1 + 1, n_2\}$. Truncate the n_3 most significant bits of e_5 and padded with trailing zeros to form a 32 bit end-

point $\bar{e}_2$. This procedure continues for all the leaf nodes left. When the endpoints for the higher level nodes are created, we can use the procedure recursively to create the tree structure.

We may notice that when creating a new endpoint, essentially we make all the *un-needed* trailing bits zero. By *un-needed* trailing bits, we mean that these bits are not needed for a search. For example, assume that we need to search a key k with the two endpoints 10010101 and 11110001. In this case, we can compare the key k with a newly created endpoint 11000000. If $k > 11000000$, we search the 11110001 branch; otherwise, we search the 10010101 branch. The benefit is that 11000000 has more trailing zeros. The closer the endpoints are to the root, the more trailing zeros they have. We know that the trailing zeros help to compact more endpoints in the same memory. However, this does not mean that the total memory is reduced (explained later). This can help when the table is sparse or the prefix is long, such as IPv6. For dense IPv4, this gain does not compensate the price of creating new endpoints.

With this scheme, the search procedure needs to be modified. For searching an endpoint e_0 in a node, the number of skip bits in the data structure is used to truncate the most significant bits of e_0 to form a search key k_0 which is padded with trailing zeros. The biggest key k_i in the node which is less than or equal to k_0 is found. This leads us to search in the next lower level node (root node) in subtree t_i . If $k_0 = k_i$, a search of the keys in the root node of this subtree is needed as usual. If $k_0 > k_i$, the endpoint e_0 is bigger than all the keys in this root node, thus a search is not needed.

In general, variant one uses less memory than variant two, and variant two tends

to need less memory accesses than variant one. We can explain this intuitively. Since the memory usage is dominated by the number of leaf nodes, assuming a leaf node can store five endpoints on average, variant one uses about 1/6 less nodes than variant two. On the other hand, for variant two, when we approach the root, the endpoints in the nodes tends to have a larger number of trailing zeros and therefore the number of endpoints stored in a node tends to be larger. Thus less memory accesses are needed. We believe that for an IP table dominated by long prefixes, such as 32 length for IPv4 or 128 for IPv6, variant two is a better choice. If an IP table is dominated by short prefixes, variant one is a better choice.

Variant Three

To compromise variant one and two, we may combine them to form a new variant. We may use variant one first. When we are in a level of the tree approaching the root, we may change to variant two. An experimental study shows that this is a good compromise of the two variants. In the experiment, variant one is used for leaf nodes. Variant two is applied to all higher level nodes.

Variant Four

Variant four is to use the front end array similar to [41]. This essentially divides the endpoints into groups according to their leading bits. In each group, a respective tree structure is created using techniques described in this thesis. The number of bits to choose to divide the table is a parameter to be optimized by the particular table. On one hand, the longer the number of bits is, potentially, the smaller the

number of memory accesses is needed; on the other hand, the longer the number of bits is, the more groups will have empty or a small number of endpoints in it, and therefore, waste the memory. It is a tradeoff between memory size and the number of memory accesses. The memory size is the largest among the four variants and the number of memory accesses is the smallest among the four variants.

Table 3.4: An example.

10000000	00001000	01000000	00000000
10000000	10001001	00100000	00000000
10000000	11101101	01000000	00000000
10000000	11101101	01110010	00000000
10000000	11101101	01110011	00000000
10000000	11101101	01110100	00000000
10000000	11101101	01111101	00000000
11111111	00001111	00001101	10000000
11111111	00101111	00001110	00000000
11111111	10001111	00100111	00000000
11111111	10101111	00101000	00000000
11111111	11101111	00110000	00000000
11111111	11110000	00010000	11000000

As mentioned above, when the number of empty groups is large, a large amount of memory is wasted. To remedy the drawback in this situation, we can modify the

algorithm. Instead of using the front end array, we may use a tree structure. The idea is that we truncate a number of leading bits of all the endpoints to form a set of new endpoints (with duplicates removed). With the set of new endpoints, variant one or two is used to create a search tree. The search result is a pointer to a subtree instead of a port entry. The subtree is created using variant one or two. In fact, this is a layered approach which can be applied recursively. It is effective for long bit endpoints such as IPv6 and sparse tables. In general, the cost of this is a bit of memory inefficiency.

For an extreme example in Table 3.4, we are better off to divide the table into two obvious new tables and use 10000000 and 11111111 for a front search to lead to one of the tables.

3.3.3 Optimizations

Several suggestions for optimization for the data structure are provided in this subsection. From Figure 3.6, we can see the level one (root) always occupies one row of memory. The level two always starts from the next row. So the pointer in the root always points to its next row and therefore is not necessary. This saves 20 bits for the root so that we can store potentially more endpoints in the root. Provided that the maximum number of keys stored in a node is 16, the number of nodes in the third level will be less than $17^2 = 189$, hence the pointer field in the level two needs 8 bits instead of 20 bits, and so on.

The next optimization step is to put the root in the registers. In practice, the number of endpoints in the root could be small. In that case, we can store these

endpoints in the registers, thus saving one memory access.

Another optimization factor is for the skip bits field. We can define this field as the additional bits to skip relative to the previous level. For example, if we skip 2 bits in the first level and 7 bits in the second level, we only need to store $7 - 2 = 5$ as additional bits to skip. From the experiments, 3 bits is enough for IPv4.

3.4 Experimental Study

We download IPv4 routing tables from [88] and IPv6 routing tables from [87] for the experiments. They are the basis for our experiments study. We plan three groups of the experiments. One is to use the original tables; Another is to create new tables by expanding original tables; The third one is to generate random tables. Results are analyzed in the following subsections.

3.4.1 Port Merge Effect

If two consecutive intervals have the same port, the two consecutive intervals can be merged into one interval. This reduces the number of endpoints. Table 3.5 shows the port merge effect for IPv4 addresses. The *merge rate* is equal to the number of endpoints with merge divided by the number of endpoints without merge. From the table, we can see that the merge effect is very significant. For example, for pacbell, the merge rate is 57%. This implies that we can save about 43% memory if the merge effect is taken into account. We also conduct experiments by randomly assigning port numbers to prefix. The merge effect is not significant. This indicates that

Table 3.5: Port merge effect.

name	entries	endpoints without merge	endpoints with merge	merge rate
aads	17,487	27,250	16,807	62%
mae-east	18,661	29,871	19,410	65%
mae-west	29,608	45,826	27,792	61%
pacbell	24,193	37,771	21,389	57%
paix	15,332	24,126	16,836	70%

in real life tables, the port assignment has some correlation between neighboring intervals. By observing the port numbers in a real life forwarding table, we can see the port numbers tend to be clustered. For IPv6, due to the lack of large real life tables, we cannot analyze the effect. In the following experiments, we will not perform port merge for the set of endpoints.

3.4.2 Comparisons on Variants

Table 3.6 gives a general picture of the performance of the four variants. It shows the memory requirement and the number of memory accesses for the original tables under different variants. Columns with heading one, two and three correspond to variant one, two and three. The column with heading four is the result of variant four with 6 as the number of the front end bits. The column with heading five is the result of variant four with 14 as the number of the front end bits. We can see

Table 3.6: Experiments with IPv4 using 144bits memory width.

name	memory (bits)					number of accesses				
	one	two	three	four	five	one	two	three	four	five
aads	400,896	423,360	403,776	435,744	2,677,248	6	5	5	4	3
mae-east	421,344	440,064	423,792	454,752	2,707,776	6	5	5	5	3
mae-west	644,256	672,912	648,864	690,768	2,929,824	6	5	5	5	3
pacbell	541,872	565,632	545,760	580,176	2,818,224	6	5	5	4	3
paix	359,712	379,152	362,016	391,824	2,649,600	6	5	5	5	3

variant one uses the smallest amount of memory but the number of memory accesses is the highest. Variant three is well balanced. Other experiments we conducted show similar results with one exception that when a table is dominated by short prefixes, variant one shows the best performance for both memory size and accesses. In order to avoid swamping pages due to huge experimental results, we choose variant three for the following experiments.

3.4.3 Results Using Real Life Tables

This subsection shows the results using real life tables. We first describe the characteristics of the forwarding tables. The five IPv4 tables are dominated by 24 bit prefixes of around 50%. The next largest number of prefixes are 19, 22, 23 and 16 bits, about 7% each. The IPv6 tables are dominated by 48, 35 28 and 24 bits of

length. Some tables have as high as 70% 48 bit prefixes.

Table 3.7 is the result of IPv4 using variant three. The first column is the name for NAPs. The second column is the number of entries (prefixes) in the routing tables. The third column is the *ME ratio*. ME ratio is defined as the number from dividing the memory requirement in bits by the number of prefixes. It measures the average number of bits needed for one prefix. The table shows that about 22 bits are needed for a prefix. Note that if the port merge is taken into account, fewer bits are needed.

Arity of a node is defined as the number of subtrees of the node. The average arity measures the average number of subtrees of all the nodes in the tree. The last column is the average arity from each routing table. From the average arity, we can roughly estimate the performance of a routing table with a different size. For example, assuming the average arity of 8, a routing table with 1M prefixes will need roughly $\log_8 1000000 = 7$ memory accesses.

The above results are obtained without using any optimizations mentioned in the previous section. We should mention that our data structure is also suitable for a pipeline implementation. This is because our algorithm is essentially a tree structure. The searches in the different levels of the tree do not interfere with each other and can be pipelined. For a pipeline implementation, the lookup speed is only restricted by the memory access latency.

Table 3.8 is the result of IPv6 using variant three. We use 168 bits memory width so that the key field is 128 bits long, which can accommodate at least one IPv6 endpoint. The low ME ratio (about 30) for t3 and t4 is probably due to a high percentage (above 70%) of length 48 prefixes. For other tables, no prefix has a

Table 3.7: Results of IPv4 using 144bits memory width.

name	entries	ME ratio	average arity
aads	17,487	23.09	9.7
mae-east	18,661	22.71	10.2
mae-west	29,608	21.92	10.2
pacbell	24,193	22.56	10.0
paix	15,332	23.61	9.6

percentage above 40%.

3.4.4 Results Using the Expanded IPv4 Tables

This subsection provides results for tables expanded from real life tables. The six real life IPv4 tables are combined and the duplicates are removed to form a large table as a basis. The size of the resulting table is 50449. We can expand the basis table to a larger size table. The method of expanding the table is to pick a prefix of length 24 and expand it to $2^8 = 256$ prefixes of length 32. Given a size of a table we want to expand and the total number of prefixes of length 24, we can roughly figure what percentage of the prefixes of length 24 needs to be expanded. Then we randomly select this percentage of prefixes of length 24 to expand. Using this method, we generated five tables of sizes of 66004, 128479, 258784, 591050 and 939889. We have done experiments with different memory widths. We make the

Table 3.8: Results of IPv6 using 168 bits memory width.

name	entries	memory (bits)	ME ratio	memory accesses
t1	58	3696	63.72	4
t2	181	11760	64.97	4
t3	534	17304	32.40	4
t4	488	15456	31.67	4
t5	248	15792	63.68	4
t6	284	16128	56.79	4

length of the key field in a node as 32, 64, 128, 256, 512 or 1024 bits. Assuming that the total length of the other fields is 36 bits for IPv4, the memory width comes to 68, 100, 164, 292, 548 or 1060 bits respectively. Table 3.9 shows the memory size as a function of table size and memory width. The result is also plotted in Figure 3.7. From the results, we can see that 164 bits and 292 bits memory width are better choices than other memory widths for this set of tables. We can also see that the memory requirement increases proportionally to the increase of the number of prefixes, but at a small speed. For example, using a 164 bit memory width, the ME ratio is about 19.4 for tables of size around 50k; The ME ratio is only about 9.5 for tables of size around 1M.

The number of memory accesses is shown in Table 3.10. We can see it is not sensitive to the size of the tables. It indicates that our algorithm scales well to the size of the forwarding table.

Table 3.9: Memory size for IPv4 as a function of table size and memory width.

memory width (bits)	table size (number of prefixes)					
	50449	66004	128479	258784	591050	939889
68	1411000	1613300	2415700	4084828	8292736	12685196
100	1090300	1251300	1895400	3220800	6558500	10028200
164	978752	1123236	1687396	2860816	5822820	8903396
292	953088	1099964	1681044	2868608	5842920	8955640
548	987496	1144772	1755792	3028796	6201716	9477112
1060	1038800	1208400	1876200	3294480	6954660	10767480

3.4.5 Results Using the Expanded IPv6 Tables

Since we do not have large IPv6 tables, we randomly generate the tables and then expand them. The method is as follows. First we randomly generate a table of size 10k with about 80% of prefix length 48 and lengths from 12 to 64 are uniformly distributed. A second table of size 10k is generated in the same way with about 80% of prefix length 64 instead of 48. Then the two tables are combined with duplicates removed. This is the basis table for expansion which has about 40% of prefixes of length 48 and 64 each. When expanding, a prefix with length 64 is randomly selected and expanded to a number of prefixes of length 128. The exact number is randomly chosen between 128 and 1024. The bits that expand from bit 65 to bit 128 are also randomly generated. Expansion continues until we reach the desired size of the table.

Figure 3.7: Memory size for IPv4 as a function of table size with different memory widths.

Using this scheme, we generated seven tables of sizes of 20000, 31284, 48210, 144124, 223112, 530601 and 1208666. The memory widths for this set of experiments are 168, 296, 552 and 1064, respectively. Assuming that the total length of other fields is 40 bits for IPv6, the length of the key field in a node is 128, 256, 512 and 1024 respectively. Shorter memory widths cannot accommodate a single IPv6 endpoint in the worst case and modification needs to be done.

Table 3.11 shows the memory size as a function of table size and memory width.

Table 3.10: Number of memory accesses for IPv4 as a function of table size and memory width.

memory width	table size (number of prefixes)					
	50449	66004	128479	258784	591050	939889
68	9	9	10	10	10	11
100	6	7	7	8	8	8
164	5	5	6	6	6	6
292	4	5	5	5	5	5
548	4	4	4	4	4	4
1060	3	3	4	4	4	4

The result is plotted in Figure 3.8. From the results, we can see 1064 bits memory width is the best choice among all memory widths for this set of tables. We can also see that the memory requirement increases proportionally to the increase of the number of prefixes. The ME ratio does not change much when the number of prefixes increases. For example, the ME ratio is 115.8 for tables of size of around 20k and 125.6 for tables of size of around 1M if the memory width is 1064. All ME ratios are rather high compared with the small real life tables. The reason for this is that the endpoints in real life tables are correlated. The endpoints in randomly generated tables are not. For example, when a 64 bit prefix expands to 128, the trailing 64 bits are randomly generated. If we select 128 to 1024 entries from a space of size of 2^{64} , it is rare that any two of them correlate. (We borrow the word *correlate* to roughly express the following idea: for example, we can say 11111100

Table 3.11: Memory size for IPv6 as a function of table size and memory width.

memory width (bits)	table size (number of prefixes)						
	20000	31284	48210	144124	223112	530601	1208666
168	3078264	5032440	7962696	24559080	38229576	91463568	208804848
296	2644760	4343800	6890880	21319400	33197880	79448472	181433200
552	2390712	3881112	6115056	18782904	29196936	69782184	159235992
1064	2316328	3736768	5866896	17931592	27878928	66544688	151784920

and 11111111 correlate, but 10100100 and 11111111 not so much).

The number of memory accesses is shown in Table 3.12. Similar to IPv4, it is not sensitive to the size of the tables.

3.5 Previous Work

Surveys on address lookup algorithms were given in [59, 68, 55]. Here, we highlight performance measurements for some of the algorithms. Multiway search [41] is the most similar approach to our method. However, due to the lack of techniques to optimize the data structures, larger memory requirements than ours are reported and the number of memory accesses is comparable to ours. Using 32 byte cache line (which equals to 256 bits memory width), 6-way search can be done. For a routing table with over 32,000 entries, 5.6M bits memory is needed. In the worst case, 4 memory accesses are needed. If 256 bit memory width is used, we can report much

Table 3.12: Number of memory accesses for IPv6 as a function of table size and memory width.

memory width (bits)	table size (number of prefixes)						
	20000	31284	48210	144124	223112	530601	1208666
168	6	7	8	8	8	9	9
296	5	7	7	7	7	8	8
552	5	5	6	6	6	7	7
1064	4	4	5	5	5	6	6

better results than that in the previous section. Even with 144 bit memory width, we reported more favorable results than that from [41].

Reference [75] proposed an algorithm which requires a worst case of $\log W$ hash lookups, where W is the length of the address in bits. Hence, at most 5 hash lookups for IPv4 and at most 7 hash lookups for IPv6 are needed. However, data from [41] showed that this algorithm needs 1600K bytes for an IPv4 routing table with roughly 40k entries. In addition to that, the building of the algorithm is not fast.

Reference [48] uses LC-trie for storing the table. They reported that about 4M bits are needed for the LC-trie. According to the paper, they need at least 100 bits for each prefix.

Reference [32] presented an IP address lookup scheme and a hardware architecture. The routing table is meant to be put in off-chip memory, for they need 450-470k bytes memory for a routing table with 40,000 entries. They reported one to three memory accesses. They compared their scheme favorably with those proposed in

Figure 3.8: Memory size for IPv6 as a function of table size with different memory widths.

[17] and [30].

Reference [61] was also aimed at hardware implementation. They used a compact stride multibit trie data structure to search the prefixes. They reported 4.3MB memory for a forwarding table of 104547 entries. The algorithm allows for incremental updates but is not scalable to IPv6.

Although [44] achieves $O(\log n)$ complexity for both update and search, the memory required is much larger than us.

Table 3.13: Bits consumed for each IPv4 prefix.

papers	[41]	[75]	[48]	[32]	[61]	this thesis
ME ratio	175	320	100	90	329	15

Table 3.13 shows roughly the bits consumed for each IPv4 prefix in the above mentioned papers. These figures are derived from the experimental reports from each paper. They just provide a general picture rather than an accurate measure, because data used for experiments, data structures and resource assumption, etc., are different for each paper. For example, reference [61] used 19 bits for the next hop index field, and we use 22 bits.

There are several papers, e.g. [41, 75], that deal with IPv6 address lookup. They have different merits from different perspective view of performance. Our algorithm has the advantage of using the lowest memory and the search speed is comparable to others.

3.6 Summary

We developed a novel algorithm which is tailored to hardware technology. The distinguishing merit of our algorithm is that it has a very small memory requirement. With this merit, a routing table can be put into a single chip, thus, the memory latency to access the routing table can be reduced. With the pipeline implementation of our algorithm, the IP address lookup speed can only be limited by the memory access technology.

Experiment analysis shows that, given the memory width of 144bits, our algorithm needs only 400kb memory for storing a 20k entry IPv4 routing table and 5 memory accesses for a search. For a 1M entry IPv4 routing table, 9Mb memory and 7 memory accesses are needed. With a memory width of 1068 bits, we estimate that we need 150Mb of memory and 6 memory accesses for a routing table with 1M IPv6 prefixes.

Real life tables have some structures. Modifying our algorithm to exploit these structures can achieve better performance.

In the next chapter, we describe an update algorithm. The update algorithm is so general that it can be used to convert many static algorithms to dynamic ones. Combined with the algorithm in this chapter, it provides a good alternative to the two-bank memory architecture proposed in this chapter.

Chapter 4

A Fast Update Algorithm

This chapter describes a new update algorithm for IP forwarding tables. This update algorithm is so general that it can be combined with many IP address lookup algorithms for fast update.

This is done by partitioning the IP forwarding table into independent sets. For each independent set, we can create a data structure such as in the previous chapter for search with fast lookup. This algorithm supports fast update since the independent set avoids the worst-case update scenario. The number of independent sets is small, hence, the number of parallel data structures is small.

The rest of the chapter is organized as follows. Section 4.1, points out the worst-case scenario for update. The concept of independent sets and the details of how to partition the forwarding table into independent sets are described in Section 4.2. In Section 4.3, the update algorithm is presented. In Section 4.4, results from previous work are highlighted. Concluding remarks are made in Section 4.5.

Figure 4.1: The worst-case scenario for update.

4.1 The Worst-case Scenario for Update

The longest prefix match is more difficult than the general table lookup problems, especially for update. There is the worst-case scenario for update for the longest prefix match. Figure 4.1 gives an example of the worst-case scenario. It shows that the IP addresses of prefixes $P_1, P_2, \dots, P_n$ are covered by prefix P_{n+1} . According to the longest prefix match rule, the port number for each address interval is shown in the lower part. If there is an update for P_{n+1} , e.g., the port number d is changed to e , then all the intervals with port d will be changed to e . The complexity is $O(n)$, where n is the total number of prefixes. If n is large, a lot of memory accesses will be involved. If n is small, the memory accesses incurred can be tolerated. In order to solve this problem, intuitively, we need to separate the prefix P_{n+1} from other prefixes. This leads to the concept of *independent sets* which is defined in the next section.

4.2 Partitioning the Prefixes into Independent Sets

We first give the definition of the independent set and then give the algorithm of partitioning the prefixes into independent sets.

4.2.1 Definition

Definition: If the ranges of two prefixes overlap with each other, then the two prefixes are *dependent*, otherwise they are *independent*. For a set of prefixes, if the prefixes in it are pairwise independent, then the set is called an *independent set*.

For example, in Figure 4.1, prefixes $P_1, P_2, \dots, P_n$ are independent from each other and form an independent set. Prefix P_{n+1} is independent of none of them.

Note that for any two prefixes, their ranges either contain one another or are disjoint. They cannot overlap with each other.

4.2.2 Partitioning

Given a set of prefixes, we are trying to partition out an independent set such that the independent set is as large as possible. For the remainder set, we do the same thing. This procedure continues until the set of prefixes is partitioned into independent sets. The algorithm to find an independent set from a set of prefixes is provided in the following. It also preserves the property that for any two prefixes which are covered by one another, the longest prefix is chosen first.

Let $P[]$ be the set of the prefixes and $P[i]$ an element of $P[]$. n is the number of prefixes. $P[]$ is sorted according to the left endpoint of prefixes as integers in increasing order. If two left endpoints are equal, then the prefix with longer length is put in front of the shorter one. Let I be the independent set and L be the remainder set. Let $le(P[i])$ be the left endpoint of $P[i]$ and $re(P[i])$ the right endpoint of $P[i]$.

```

/*Begin Pseudocode*/
i = 0;
temp = P[0];
i ++;
while (i < n) {
    /*The left endpoints of these prefixes are same. */
    while (i < n && le(temp)=le(P[i])) {
        put P[i] to L;
        i ++;
    }
    /*The left endpoints of the prefixes are not same. */
    if (i < n) {
        if (re(temp)>le(P[i])) {
            put temp to L;
            temp = P[i];
            i ++;
        } else {

```

```

        put temp to I;

        temp = P[i];

        i ++;

    }

    } else {

        put temp to I;

    }

}

/*End Pseudocode*/

```

Theoretically, we could partition an IPv4 forwarding table into at most 32 independent sets and an IPv6 forwarding table into at most 128 independent sets. However, in reality, the number of independent sets is much smaller. The following subsection provides the experimental results from real forwarding tables for the independent sets.

Not accounting for the preprocessing part, the complexity of this algorithm is $O(n)$, since each prefix is checked once. Obviously, the complexity of partitioning an IPv4 table is upper bounded by $32O(n)$. In fact, the set of IP addresses can be looked upon as interval graph [77]. More efficient algorithms could be found by exploiting the property of the interval graph. On the other hand, we cannot expect much improvement theoretically, for the complexity of any algorithm for this purpose is lower bounded by $O(n)$.

This is not balanced. The search performance will be dominated by the large

independent set. Fortunately, if an algorithm, such as that introduced in Chapter 3, is used as a range search algorithm, the search unbalance is not so bad because the search performance of the algorithm in Chapter 3 is not very sensitive to the number of endpoints. However, the parallel processing elements should share the same memory, otherwise the memory usage of the total system will not be efficient. On the other hand, we may attempt to balance the load between independent sets. This will destroy some properties between the independent sets and make the following update algorithms impossible.

4.2.3 Experimental Results

We download IPv4 routing tables from [88] for the experiments. We have five forwarding tables from five different NAPs. We also combine these five tables and remove the duplicate to form a larger table. The results are shown in Table 4.1. The first row is the name of the NAPs. The second row is the sizes of the forwarding tables. The following rows show the sizes of the independent sets. We can see all the original tables have five independent sets. The combined table has six independent sets. For all the tables, the first independent set is much larger than the following independent sets. This is not balanced. The performance of search will be dominated by the large independent set. Fortunately, if an algorithm, such as introduced in Chapter 3, is used as range search algorithm, the unbalance of search is not so bad, because the performance of search of the algorithm in Chapter 3 is not much sensitive to the number of endpoints. However, the parallel processing elements should share the same memory, otherwise, the memory usage of the total system

Table 4.1: The size of the Independent Sets.

name	aads	mae-east	mae-west	pacbell	paix	combined
total	17486	18661	29609	24193	15332	50449
1st	16191	17635	27635	22728	14237	46597
2nd	1161	919	1759	1296	960	3305
3rd	124	94	195	148	118	469
4th	9	12	18	20	15	72
5th	1	1	1	1	2	5
6th	0	0	0	0	0	1

will not be efficient. On the other hand, we may attempt to balance the load between independent sets. This will destroy some properties between the independent sets and make the following update algorithms impossible.

4.3 Update Algorithm

In order to achieve a fast update, the first step is to partition the prefix set into independent sets as described in the previous section. Then, parallel searches are performed on all the independent sets. When there are multiple matches, the first (largest) independent set has the highest priority, then the second and so on. Specifically, all the searches return a result, indicating either a successful search or an unsuccessful search. Among the successful searches, the result from the largest independent set is selected.

Let us take the aads forwarding table in Table 4.1 (the first column) as an example. The total of 17486 prefixes are partitioned into 5 independent sets. We could use 5 parallel searches. However, we can see that the first independent set is very large. The following independent sets become smaller and smaller. When the size of an independent set is very small, the worst-case update will not have much affect for the update, because even recreating the data structure will not take many operations. For example, we can combine the 4th and the 5th independent sets into one prefixes set of 10 prefixes. Even in the worst case, only about tens operations are needed. According to the application requirement, we could combine the last three independent sets or the last four independent sets. More independent sets are combined, less parallel searches are needed, but updates become slower.

Let $I_1, I_2, \dots, I_t$ be the partitioned sets obtained by the procedure described before. $I_1, I_2, \dots, I_{t-1}$ are independent sets. I_t may not be an independent set. However, the number of prefixes in I_t is so small that even recreating a data structure from I_t will not take a lot steps in the case that I_t is not an independent set.

We give the update algorithm which involves both adding a prefix and deleting a prefix. If only deleting or adding a prefix is involved, the update algorithm will be much simpler than the one described in the following subsections.

4.3.1 Deleting a Prefix

Let P_1 be the prefix needs to be deleted. A simple-mind solution would be such that: find the prefix first, and then delete it. This may cause problems when we add a prefix. Consider the case in Figure 4.2. Assume P_1 is deleted and then P_4 needs to

Figure 4.2: The scenario for the need of moving prefixes.

be added. However, since P_4 covers P_2 (as a prefix, P_2 is longer than P_4), P_2 should be in I_k and P_4 should be in I_{k+1} . Therefore, in order to preserve the property of the independent sets obtained from the previous algorithm, we must move P_2 to I_k and P_3 to I_{k+1} and so on.

The algorithm for deleting a prefix is given in the following.

Step one: let P_1 be the prefix that needs to be deleted. Find P_1 from $\{I_1, I_2, \dots, I_t\}$.

Assume P_1 is in I_k . Delete P_1 from I_k .

Step two: move prefixes between independent sets. Let P_s and P_b be variables for prefixes.

$P_s = P_1;$

$i = 1;$

$P_b =$ a prefix in I_{k+i} that covers P_s and is independent of other prefixes in $I_{k+i-1};$

while(P_b is not empty) {

```

    move  $P_b$  to  $I_{k+i-1}$ ;

     $i++$ ;

     $P_s = P_b$ ;

     $P_b =$  a prefix in  $I_{k+i}$  that covers  $P_s$  and is
    independent of other prefixes in  $I_{k+i-1}$ ;

}

```

The algorithm stops when there is no prefix in the next independent set that covers the previous prefix and is independent of other prefixes in I_{k+i} . One may argue that though there is no prefix in the next independent set that covers the previous prefix and is independent of other prefixes in I_{k+i} , we may find one after the next independent set. We will prove this case cannot happen.

Property: Let $P_1 = [b_1, e_1)$ be in I_k . There is no prefix P in I_{k+1} such that P covers P_1 and is independent of other prefixes in I_k . Furthermore there is no prefix P in $I_{k+2}, I_{k+3}, \dots, I_t$ such that P covers P_1 and is independent of other prefixes in I_k .

Proof: We prove it by contradiction.

Assume there is a prefix $P_2 = [b_2, e_2)$ in I_{k+2} such that P_2 covers P_1 and is independent of other prefixes in I_k . With the help of Figure 4.3, we can see that there must be a prefix P in the interval of $[b_2, e_2)$ in I_{k+1} , otherwise, P_2 should have been moved to I_{k+1} . The prefix P cannot be in the interval of $[b_1, e_1)$, otherwise, P should have been moved to I_k since P_1 covers P . Thus, P should be in either $[b_2, b_1)$ or $[e_1, e_2)$. However, in either case, P should have been moved to I_k since P_2 is independent of other prefixes in I_k and P should be independent of other prefixes

Figure 4.3: Proof of the property.

in I_k too. This indicates that there is no such prefix, which is a contradiction. $\triangle$

4.3.2 Adding a Prefix

For adding a prefix P , we first check I_1 with prefix P to see if they are independent. If yes, then prefix P is inserted in I_1 and we are done; otherwise, there is a prefix in I_1 , say P' , which is dependent of P . There are only two cases: P contains P' or P' contains P . If P contains P' , then proceed to check I_2 with prefix P ; if P' contains P , then remove P' from I_1 , insert P in I_1 and then proceed to check I_2 with prefix P' . This procedure continues until the prefix is inserted in a set, or all the sets are exhausted. In the latter case, the single prefix forms a new set.

4.3.3 Estimation on Complexity

Assume that we use the algorithm in Chapter three for the independent sets. Assume that a search takes about s memory accesses. For deleting a prefix, about ts memory accesses are needed for step one. Less than $2ts$ memory accesses are needed for step two. Thus, a maximum of $3ts$ memory accesses is needed for deleting a prefix. If $s = 5, t = 4$, less than 60 memory accesses are needed. (We do not consider combining tree nodes here for simplicity. Not combining tree nodes could deteriorate the memory usage after a lot of updates. This could be solved by recreating the data structure after a long time or using a dynamic tree structure.) For adding a prefix, a similar result can be obtained.

4.4 Previous Work

Reference [58, 44] propose algorithms with $O(\log n)$ complexity for both update and search. They are the first to achieve this performance for both update and search. Reference [71] also proposes a fast update for multiway range trees. These algorithms are restricted to the particular data structure. Our algorithm is much more general. It does not restrict us to any particular data structure or algorithm.

The algorithm in [61] supports an incremental update. Again, the supporting update is a characteristic of their search algorithm rather than a new algorithm.

For IP address lookup algorithms, readers are referred to the survey papers [59, 68, 55] and the references therein. By carefully studying these algorithms, we can convert some of them into a dynamic algorithm with fast update using the

scheme proposed in this chapter.

4.5 Summary

We developed a novel update algorithm based on independent sets. In fact, it can be viewed as a general approach to potentially turn a static algorithm into a dynamic one with high speed updates. This is done by introducing a parallel mechanism in the algorithm.

When a particular algorithm is combined with our update algorithm, some modification cannot be avoided. As is the case for any dynamic algorithm, memory management needs to be elaborated to achieve memory efficiency. Since our proposed update algorithm does not incur any resource penalty, such as a large memory tradeoff, it provides a good alternative for the practical design.

In the next chapter, we describe a packet classification algorithm. Since packet classification is a multi-field lookup problem, the algorithms described in this chapter and the previous chapter can be used in the packet classification algorithm.

Chapter 5

A Packet Classification Algorithm

This chapter describes a new algorithm for packet classification using the concept of independent sets which is similar to that presented in the previous chapter. The algorithm has very small memory requirements. The search speed is neither sensitive to the size of the rule table nor to the percentage of wildcards in the fields. It also scales well from two dimensional classifiers to high dimensional ones. We anticipate that it scales to IPv6. In particular, the algorithm is inherently parallel. Hardware tailored to this algorithm can achieve very fast search speeds. The update algorithm proposed is also very fast in general. This chapter is an expansion of [69].

The rest of the chapter is organized as follows. In Section 5.1, more background of the packet classification is provided. The packet classification problem is defined and the related notation is developed in Section 5.2. The concept of independent sets and the details of the new algorithm are described in Section 5.3. In Section 5.4, results of experimental studies are presented. In Section 5.5, we highlight results from some existing algorithms. A summary is made in Section 5.6.

5.1 Preliminary

As discussed in Chapter two, Internet service providers (ISP) are expected to provide more value-added services to end users. In order to provide these services, the router needs to classify the packets into flows according to different criteria. These criteria form rules which are based on L2/L3/L4 fields in the packet header. This function of router is called *packet classification*.

The speed of a packet classification algorithm is measured by the times of memory accesses. Other performance metrics for a packet classification algorithm include memory storage requirements, update speed and scalability to both the number of the rules, and the number of fields included in the rules, among others.

We describe a new algorithm using the concept of independent sets. The new algorithm is theoretically sound. Experimental studies have shown that its performance is at least comparable to best available algorithms. Specifically, the algorithm could convert a higher dimensional classification problem into a lower dimensional one. Thus, the lower bound of a d -dimensional classification problem, which requires to perform at least d one-dimensional range searches, is expected to break. The algorithm has very small memory requirements. The memory factor is expected to be smaller than two (the *memory factor* is the ratio of the total amount of memory used to that needed to store the rules). The best memory factor reported in existing algorithms is four [23]. The search speed of our algorithm is neither sensitive to the size of the rule table nor the percentage of wildcards in the fields. It scales well from two dimensional classifiers to high dimensional ones. We envision that it scales to IPv6. The update algorithm proposed is also very fast in general. In particular, the

algorithm is inherently parallel. It is easy to exploit the parallel mechanism in the hardware. One of the possible limitations of the new algorithm is that it depends on the characteristics in the rule table. Experiments show that the number of memory access can be as low as 15 and as high as 100 for two-dimensional classification problems with a table size of 30000 (assuming that a one dimensional range search uses four memory accesses).

5.2 Packet Classification Problem

In the packet classification application, packets are classified into flows according to policy or routing information. The policy is specified using fields in the header of a packet. Specifications of fields are called *rules*. So flows are specified by rules applied to incoming packets. Each rule consists of several fields, say d . A collection of rules is called a *classifier*. Each field is either an exact value or a prefix or a range. In fact, exact values and prefixes are special ranges. In this thesis, we treat fields as arbitrary ranges. Each rule also has a priority index number. Usually, the rules in a classifier are sorted according to their priorities. This index number is necessary since a packet may match more than one rule. In this case, the rule with the highest priority index is chosen. Let C be the classifier; or $C = \{R_1, \dots, R_N\}$, where R_i ($i = 1, \dots, N$) is a rule and N is the number of the rules in the classifier. For each rule R_i , let $R_i = (F_1^i, \dots, F_d^i)$, where F_j^i ($j = 1, \dots, d$) is the j th field. Throughout the thesis, a field or a range of integers is expressed as $F = [b, e]$, which means all integers greater than or equal to b and smaller than or equal to e . b and e are called

the *begin point* and *end point* of the field F respectively. For example, if the field is an IPv4 destination address, then either point is an integer between 0 and $2^{32} - 1$.

When a packet is arriving, the values f_i ($i = 1, \dots, d$) from the relevant d fields are extracted and expressed as $P = (f_1, \dots, f_d)$. We say that a rule $R = (F_1, \dots, F_d)$ is matched by the packet, if $f_i \in F_i$ for all $i = 1, \dots, d$. Among all the matched rules, the rule with the highest priority index defines the flow that the packet belongs to.

With the above definitions, a rule can be considered as a hyperrectangle in the d -dimensional space. A classifier is a set of such hyperrectangles. Hyperrectangles in the classifier might be overlapped. A packet is then a point in the d -dimensional space. Thus, packet classification is equivalent to finding all hyperrectangles which contain the query point. This resembles the point location problem in computational geometry [52]. The difference between the packet classification and the point location problem is that hyperrectangles in the point location problem are nonoverlapping, while hyperrectangles in the packet classification problem may overlap. Hence, the packet classification is more complex than the point location problem. However, structures and characteristics in classifiers could be exploited to develop high performance packet classification algorithms. Such packet classification algorithms may break the performance upper bound achieved in the point location background. The algorithm to be developed in this chapter serves as one such example.

5.3 Developing the Algorithm

5.3.1 Independent Sets

We first give the formal definition of the independent sets and then explain the motivation behind the concept. **Definition:** Let $C = \{R_1, \dots, R_N\}$, where R_i ($i = 1, \dots, N$) is a rule. For index k , $1 \leq k \leq d$, two rules $R_i = (F_1^i, \dots, F_d^i)$ and $R_j = (F_1^j, \dots, F_d^j)$ are called *independent* along dimension k , if $F_k^i \cap F_k^j = \phi$. For a set S of rules, if any two rules in it are independent along dimension k , we call S an *independent set* along dimension k , which is denoted as an I_k -set or simply an I -set if no confusion arises.

The number of elements in a set A is referred to as the size of the set A , denoted by $|A|$. For a classifier, consider its all possible independent sets along dimension k . An independent set with the largest size is defined as a *maximum independent set* along dimension k in C . A classifier C may have more than one maximum independent set. An independent set with the largest size among all independent sets along all dimensions is called a *global maximum independent set*.

The motivation of introducing I_k -sets is that rules in an I_k -set are easy to distinguish. For example, $S = \{R_1, R_2, R_3\}$ as in Figure 5.1 is an I_1 -set in the two-dimensional space. b_1, b_2 and b_3 are the *begin* points of field one in rules R_1, R_2 and R_3 respectively. b_1, b_2 and b_3 define the search intervals, say $[b_1, b_2)$, $[b_2, b_3)$ and $[b_3, \infty)$, in a one dimensional space. Apparently, each interval contains only one rule. Let each interval store all fields of the corresponding rule. In order to query a point, say, $P = (f_1, f_2)$, we only need to search for the interval that f_1 belongs to in one dimension. After the interval has been found, we compare the point with the rule stored in the interval. If the point is contained in the rule, there is a match.

Figure 5.1: An example of an independent set.

Otherwise, there is no rule matched by the point.

We can easily find two advantages here. One is that we only need the begin points of the field in rules to form a range search structure instead of using both begin points and end points. This reduces the search points to at most one half of a traditional range search algorithm, e.g. in [23]. The other advantage is that we only need to search in one dimension rather than in all dimensions. This means that we may only need to perform one range search for a multidimensional packet classification problem.

Based on the concept of independent sets, we can develop a new algorithm. The general procedure of implementing the algorithm is described as follows. Given a classifier $C = \{R_1, \dots, R_N\}$, we try to separate from it a global maximum independent set. Then, from the set of the remaining rules (treated as a new classifier), we separate a global maximum independent set with respect to the new classifier,

and continue the process until the set of the remaining rules is empty. More formally, assume that $I[1]$ is a global maximum independent set separated from C . Let $C_1 = C - I[1]$, we then find a global maximum independent set $I[2]$ with respect to C_1 . Let $C_2 = C_1 - I[2]$, we continue the process until the resulting classifier is empty. Thus, C is divided into a collection of independent sets $\{I[1], I[2], \dots, I[s]\}$. Next, we group the independent sets $\{I[1], I[2], \dots, I[s]\}$ according to the dimension. Two independent sets belong to the same group if they are independent along the same dimension. Let $G_1, G_2, \dots, G_d$ be the resulting groups, where d is the number of dimensions for rules in the classifier C . G_i ($i = 1, \dots, d$) is the group of independent sets which are independent along dimension i . Note that a group may be empty. We search in all nonempty groups of $G_1, G_2, \dots, G_d$ respectively. The query result can be obtained by comparing all the matches to find the rule with the highest priority.

In the following, we will discuss how to find a maximum independent set and how to create a data structure for searching G_i ($i = 1, \dots, d$).

5.3.2 Finding a Global Maximum Independent Set

For a classifier, there are two steps for finding a global maximum independent set. First, a maximum independent set along each dimension is found and then by comparing the size of these maximum independent sets, a global maximum independent set is identified. Finding a maximum independent set can be converted into the independent set problem in graph theory.

For each dimension k ($k = 1, \dots, d$), a graph $\mathcal{G}_k$ corresponding to the independence of rules along dimension k can be created as follows.

Figure 5.2: A two dimension classifier.

A vertex in the graph $\mathcal{G}_k$ corresponds to a rule in the classifier. There is an edge between two vertices if and only if the corresponding rules are not independent along dimension k . A set of vertices is called an *independent set* if no pair of vertices in the set defines an edge in $\mathcal{G}_k$. An independent set with the maximum size is called a *maximum independent set* in $\mathcal{G}_k$. For example, in Figure 5.2 a two dimensional classifier is illustrated. The corresponding graphs are shown in Figure 5.3. The maximum independent sets in $\mathcal{G}_1$ are $\{1, 2, 3\}$ and $\{2, 3, 4\}$. The maximum independent sets in $\mathcal{G}_2$ are $\{1, 4\}$, $\{3, 4\}$ and $\{3, 5\}$.

To find a maximum independent set along dimension k in the classifier is equivalent to finding a maximum independent set in the graph $\mathcal{G}_k$. The maximum independent set problem in graph theory is an NP-problem [34]. Exact algorithms are not scalable to graphs with a large number of vertices. Numerous references where heuristic solutions are provided for solving this problem can be found in [7]. In our preliminary study, we choose a simple one as shown in Figure 5.4, for finding

Figure 5.3: The corresponding graphs of the two dimension classifier.

a computed maximum independent set along one dimension. The intuition behind the algorithm is that the vertices of small degrees are less likely dependent on other vertices in the graph. Therefore, they have high priority to be chosen for the testing of independence. Note that a computed maximum independent set may not be a true maximum independent set.

In order to find a global maximum independent set of the classifier, we need to create d graphs which correspond to the d dimensions. For each graph, the above algorithm is used to find a computed maximum independent set along one of the dimensions. An independent set with the largest size in all graphs corresponds to a global maximum independent set of the classifier.

The above step is first applied to C to obtain the first global maximum independent set of the classifier. Let us analyze the complexity in this step. The most expensive operation is to find the degree of each vertex in a graph. Since for each

Let $V[]$ be the vertex set of the graph $\mathcal{G}_k$. n is the number of vertices in the graph. $V[]$ is sorted according to the vertex degrees in increasing order. Let I be the computed maximum independent set.

```

/*Begin Pseudocode*/
I={V[0]};
for i from 1 to n-1
    if V[i] is independent of all vertices in I
        then I=I+{V[i]} endif
endfor
/*End Pseudocode*/

```

Figure 5.4: A heuristic algorithm for finding a maximum independent set in a graph.

vertex we need to compare it with all the other vertices once, the complexity of finding the degrees of all vertices in a graph is $O(n^2)$, where n is the number of vertices in the graph. So the complexity of finding the degrees of vertices in d graphs is $O(dn^2)$. All other operations are negligible. Therefore, the complexity of finding a computed maximum independent set of the classifier is $O(dn^2)$.

After a global maximum independent set of a classifier C has been found, we need to apply the same step to C_1 to find a global maximum independent set of a classifier C_1 and continue the process until the end. In each step, we need to recreate graphs and recalculate the degrees. The operations are expensive when using this approach. Therefore, we need to improve this approach.

Observe that the degrees of vertices are essentially used as a criterion to decide which vertex is first to be put in the maximum independent set. Instead of dynamically changing the criterion (the degrees of vertices), we define another static criterion. For each rule in classifier C , we calculate the number of rules that are not independent of it (along any dimension) as its *grand degree*. The rules are sorted according to their grand degrees in increasing order. Rules which have the same grand degree are sorted by their priority. The grand degrees and the orders of rules will not change in the subsequent steps for finding a global maximum independent set for $C_1, C_2, \dots$. With this approach, we only need to calculate the grand degrees once and sort the rules once. Our experimental study shows that this approach works nearly well as the previous one. Specifically, we got the same number of I-sets for each experiment by using these two approaches, though the elements and the number of elements in the corresponding I-sets may differ for each approach.

The following is an algorithm for dividing the classifier C into a set of independent sets.

Input: Classifier C .

Output: Groups of independent sets $G_1, G_2, \dots, G_d$.

Calculate the grand degree of each rule in C . Sort the rules in C according to the grand degrees of rules increasingly and their priorities. $I[]$ is an array to store the I-sets.

*/*Begin Pseudocode*/*

count=0;

while(C is not empty) {

```

    I[count]=Find a maximum independent sets in C;

    count++;

    C=C-I[count];
}

/*End Pseudocode*/

```

Divide $I[]$ into groups $G_1, G_2, \dots, G_d$ according to dimension along which they are independent.

5.3.3 Basic Data Structure for a Group of Independent Sets

In this section, we develop a data structure to facilitate the search in a group of independent sets obtained from the last section. We were inspired by the fractional cascading technique [10] in developing this data structure.

Let $G = \{I[0], I[1], \dots, I[s^G]\}$ be a group of independent sets along dimension D obtained in the last section, where $I[k]$ ($k = 1, \dots, s^G$) is an independent set and s^G is the number of independent sets in G . Let $I[k] = \{R_1^k, \dots, R_{n_k}^k\}$, where R_i^k ($i = 1, \dots, n_k$) is a rule and n_k is the number of rules in $I[k]$. For each $I[k]$ ($k = 1, \dots, s^G$), we extract the begin point b_i^k of each rule R_i^k ($i = 1, \dots, n_k$) along dimension D which gives a set of points $B_k = \{b_1^k, \dots, b_{n_k}^k\}$ for each $I[k]$. All points in B_k are different, since rules in $I[k]$ are independent. We assume that all points in B_k are sorted in increasing order. Next, we merge the points in all s^G sets $B_1, \dots, B_{s^G}$ into a master set B_0 . Apparently, the number of points in B_0 satisfies $|B_0| \leq \sum_{i=1}^{s^G} |B_i|$, since two different sets B_i and B_j may contain a common point.

Figure 5.5: Merge sets of begin points into a master set.

Let $N^G = |B_0|$ and $B_0 = \{b_1^0, \dots, b_{N^G}^0\}$. Refer to Figure 5.5 for an example. The number in the rectangle (rule) is the priority of the corresponding rule. The smaller the number is, the higher the priority is. For the convenience of explanation, we assume that the priority is also the index value of the corresponding rule.

Next, for each k ($k = 1, \dots, s^G$), we add “*virtual points*” to the set B_k . We first explain why we need the virtual points by the example in Figure 5.5. The interval of $[b_1^3, b_2^3]$ of B_3 corresponds to the interval of $[b_4^0, b_7^0]$ of B_0 . There are four points b_4^0, b_5^0, b_6^0 and b_7^0 in the interval of $[b_4^0, b_7^0]$. Note that any rules between b_4^0 and b_5^0 and between b_5^0 and b_6^0 are dependent of rule 3; any rules between b_6^0 and b_7^0 are independent on rule 3. In order to distinguish these two situations, we add the point

Figure 5.6: Add a virtual point to B_k .

b_6^0 to B_3 as a virtual point. Next, we give the definition.

Let b_m^0 be the largest element in B_0 . For each $b_i^k \in B_k$, let e_i^k be the end point corresponding to b_i^k . Let $b_{i+1}^k \in B_k$ be the successor of b_i^k . If the successor does not exist, let $b_{i+1}^k = b_m^0$. Assume $b_j^0, b_{j+l}^0 \in B_0$ such that $b_j^0 = b_i^k$ and $b_{j+l}^0 = b_{i+1}^k$. If there is an point $b_{j+l_0}^0 \in B_0$ with $b_j^0 < b_{j+l_0}^0 < b_{j+l}^0$ such that $b_{j+l_0}^0$ is the smallest one that $e_i^k < b_{j+l_0}^0$ then add $b_{j+l_0}^0$ to B_k as a virtual point. For convenience, if the smallest $b_1^0 \notin B_k$, then add b_1^0 to B_k as a virtual point. Each virtual point is assigned -1 as its index. Refer to Figure 5.6 for the illustration. Also, in Figure 5.7, the points with -1 as indices are all virtual points.

Our algorithm consists of two parts. One is an array that stores the classifier called the *classifier array*; each entry of the classifier array stores a rule which includes the fields, priority and port number. The other part is a one-dimensional range search structure based on B_0 . Algorithms for a one-dimensional range search are plenty in the literature. They have advantages and disadvantages. Some have a

small number of memory accesses, while others are easy to update. The algorithms in Chapter three and four, and van Emde Boas trees [78] are among the best algorithms. However Van Emde Boas trees are complicated to implement. Here we choose the algorithms of Chapter three and four. We use B_0 to create a range search tree as in Chapter three and four where each element in B_0 has an index which points to an entry of an array which stores the indices of the rules as follows. For $b_i^0 \in B_0$ ($i = 1, \dots, N^G$), we find in each B_j ($j = 1, \dots, s^G$) the largest $b_{k_j}^j$ such that $b_{k_j}^j \leq b_i^0$. Each such a $b_{k_j}^j$ corresponds to an index of a rule (or -1 for a virtual point). There are s^G indices. The indices are stored in an array pointed to by the index of an element in B_0 .

Figure 5.7 and Figure 5.8 illustrate an example with such a data structure. In Figure 5.7, -1 is the index (and priority) of a virtual point. -1 has the lowest priority. Figure 5.8 shows the indices array pointed to by the points.

5.3.4 Search

In Section 5.3.3, we demonstrated how to create the data structure for one group of I -set. There are at most d such data structures which need to be created. The search and update can be performed in parallel or in serial in the d groups of I -sets.

For the search of a packet in a group, the value in the relevant field of the packet is used as the key for the search in the range search tree. Find the point with the value that is the largest one among all which are smaller than or equal to the key. Fetch the indices pointed to by the point. Use the indices (ignore the index -1) one by one to get the rule fields by indexing into the classifier array. Compare them

Figure 5.7: A data structure example.

with the relevant fields of the packet. If there is a match, choose the matching rule with the highest priority in this group. Continue this process until all groups have been searched. Compare all the matching rules found from the groups and choose the one with the highest priority. This rule defines the flow to which the packet belongs.

5.3.5 Update

For the dynamic packet classification, we need to update the algorithm to accommodate the changes in the packet classification table. There are two kinds of updates.

$\mathbf{b}_1^0 \rightarrow$	4	-1	-1
$\mathbf{b}_2^0 \rightarrow$	4	2	-1
$\mathbf{b}_3^0 \rightarrow$	1	8	-1
$\mathbf{b}_4^0 \rightarrow$	-1	8	3
$\mathbf{b}_5^0 \rightarrow$	12	-1	3
$\mathbf{b}_6^0 \rightarrow$	5	10	-1
$\mathbf{b}_7^0 \rightarrow$	7	9	11
$\mathbf{b}_8^0 \rightarrow$	6	-1	-1

Figure 5.8: Index array pointed to by points.

One is to create the table data structure from scratch whenever there is a change. Sometimes, this is called *preprocessing*. Another is to modify the table data structure whenever there is a change. This is called *incremental updating*. Usually incremental updating is faster than recreating a table data structure from scratch. However, incremental updating may make the table data structure non-optimal. We discuss both types of updates as follows.

Incremental update

For an update, we distinguish between adding a rule and deleting a rule. There are two steps for adding a rule. The first step is to find a group containing a set that the new rule is independent of. The second step is to make a modification of the

Figure 5.9: Add new rules.

relevant data structure of the group found. We first illustrate these steps by the following example and then describe them in detail.

If we add a new rule with priority 13 to the rule sets in Figure 5.7 (refer to Figure 5.9). We first find the group B_3 which contains the set of rules that are independent of the new rule. As a result of adding the new rule, we need to modify the data structure. Specifically, we need to modify the entries pointed to by b_2^0 and b_3^0 . This is because a was the largest point in B_3 that was smaller than or equal to b_2^0 and b_3^0 . Now, it becomes b . Therefore, the entry $(4, 2, -1)$ pointed to by b_2^0 is modified to $(4, 2, 13)$ and the entry $(1, 8, -1)$ pointed to by b_3^0 is modified to $(1, 8, 13)$.

In order to find a group containing a set that the new rule is independent of, we

need to search the groups one by one. How to choose a group to search first can be decided by the real application. In searching a group, we extract the **end** point of the relevant field of the rule as the key. In the group, find the largest point that is smaller than or equal to the key in the corresponding range search tree. Assume the point we found is b_k^0 . Let E_k be the entry of indices pointed to by the point b_k^0 . For example, we add new rule 14 as in Figure 5.9. We find $b_k^0 = b_8^0$ and the entry pointed to by b_8^0 is $(6, -1, -1)$. The index 6 in the entry corresponds to rule 6. We compare the new rule 14 with rule 6. If they are not independent, we conclude that the new rule 14 is not independent of the independent set B_1 . If they are independent, we conclude that the new rule 14 is also independent of the other rules in the independent set B_1 and therefore is independent of the independent set B_1 . The second index in the entry is -1 which corresponds to a virtual point. Since a virtual point does not correspond to a real rule, we need to find a real rule in -1 's shoe. For this -1 , we find rule 9 which is the predecessor point of this -1 . We compare the new rule 14 with rule 9. If they are not independent, we conclude that the new rule 14 is not independent of the independent set B_2 . If they are independent, we conclude that the new rule 14 is also independent of the other rules in the independent set B_2 and therefore is independent of the independent set B_2 . For the second -1 , we find rule 11. The same argument follows. According to the above example, we conclude that if the entry of indices E_k contains -1 , we need to replace it with the index of the predecessor rule. The following explains how to replace -1 in E_k with the index of a rule.

If there is any -1 in E_k , fetch E_{k-1} , which is pointed to by the point b_{k-1}^0 , the predecessor of b_k^0 . Replace any -1 index in E_k , with the index in E_{k-1} which is

in the same column as that in which the -1 index lies in. This results in a new entry. For example, in Figure 5.9, the entry pointed to by b_8^0 is $(6, -1, -1)$. There are two -1 indices in it. The entry pointed to by b_7^0 is $(7, 9, 11)$. We replace the two -1 s with 9 and 11 respectively resulting in a new entry $(6, 9, 11)$. If the new entry still contains -1 s, then we fetch the entry E_{k-2} pointed to by b_{k-2}^0 . Replace the -1 indices with the same column indices in E_{k-2} . We continue this procedure until there is no -1 index in the resulting entry or we exhaust all the predecessor points in which case the -1 s are kept as the indices. Then we check the indices in the resulting entry one by one and fetch the rule fields by indexing into the classifier array. Thus we can check whether the rule is independent of the new rule or not. As soon as we find a rule that is independent of the new rule, we conclude that the independent set that contains the rule is independent of the new rule. Note that, if an index is -1 , it indicates that the corresponding independent set is independent of the new rule. It is possible that we cannot find an independent set in any group that is independent of the new rule. So there are two cases for the result. Case one: we find a rule that is independent of the new rule (hence the independent set that the new rule is independent of). In this case, assume the index of the found rule is i_f . Case two: there is no rule in any independent set that is independent of the new rule. We separately describe the modification of the range search tree and the index array for these two cases.

In case one, we have two subcases. In subcase one, the begin point of the relevant field of the new rule is not in the master set. In subcase two, the begin point of the relevant field of the new rule is already in the master set (refer to Figure 5.10 for these two cases). In subcase one, we need to add a new point corresponding to

Figure 5.10: The case 1.1 and case 1.2 for adding new rules.

this point to the range search tree (we omit the procedure here) and we also need to insert the corresponding entry of indices in the indices array formed as follows.

Assume the index of the new rule is i_n . Get a copy of the indices pointed to by b_k^0 . In the copy the index i_f is changed to i_n . This forms the entry of indices of the new point. It is inserted in the array after the entry pointed to by b_k^0 . (In Figure 5.10, $b_k^0 = b_2^0$ and $i_f = -1$.) We still need to modify the index i_f in other entries. Starting from the entry of indices of b_{k+1}^0 , we check the same position as where i_f is in the entry of indices if the entry contains i_f . If it contains i_f , change i_f into i_n and check the next point; otherwise finish the modification.

There is a tradeoff here. We could use a link rather than an array to store

the indices. This avoids moving all entries after b_k^0 when inserting the new entry. However, a link structure needs more memory than an array. Moreover, we still need to modify some entries after b_k^0 which, in worst-case, is as same expensive as moving all entries after b_k^0 .

In subcase two, we do not need to insert a new point. We only need to modify the indices starting from b_k^0 . The modification procedure is the same as in subcase one.

In case two, we need to add a new I -set which only contains the new rule to a chosen group. (The group is chosen randomly or with some criteria. If deleting a rule is considered, we may keep track of groups if they are empty. The empty group is the right place to put the new rule.) Thus the length of the indices in the chosen group will become one index length longer. For this reason, we can appoint a group as the “chosen group.” In this group, the entries of indices are deliberately made large as a backup for new I -sets (the size of the backup needs to be decided by real world applications). The backup indices in the entries are all set to -1 . The remaining modification is the same as in case one.

For deleting a rule, we first find the group that contains the rule and then check if the deletion of the rule results in the deletion of the corresponding point in the range search tree. In order to do that, we just need to compare the begin point of the rule with the begin points of the rules in the same entry. If there is a match, we do not need to remove a point; otherwise the corresponding point is removed. The rest of the work is a modification of entries of indices, which is the reverse procedure for adding a rule. We will not detail it here.

Figure 5.11: Original maximum independent sets.

Recreate the data structure

After many incremental updates, the data structure may become non-optimal. For example, in Figure 5.11, B_1 and B_2 are maximum independent sets. After the deletion of four rules (in Figure 5.12), B_1 and B_2 are not maximum independent sets any more in this case. We would have combined B_1 and B_2 to form a maximum independent set. This situation will make the data structure work inefficiently. In order to avoid this situation, we may recreate the data structure. How often we need to recreate the data structure can be decided by a real application.

Figure 5.12: After the deletion of four rules.

5.3.6 Complexities of the Algorithm

Globally, we need an array to store the classifier. The size of the array is linear to the number of rules in the classifier. In addition, a range search tree will be constructed corresponding to each of at most d groups of independent sets. Assume that the group G_i consists of s^{G_i} independent sets and contains N^{G_i} rules. Then the memory storage requirement for the range search tree corresponding to group G_i is linear to the number of N^{G_i} . The number of indices stored in the points is at most $N^{G_i} s^{G_i}$, since there are at most N^{G_i} points and each point stores s^{G_i} indices. To add all these together, the memory storage requirement for the algorithm is upper bounded by $O(sN)$, where N is the number of rules and s is the number of total

independent sets. We will see later that this bound is very loose.

The number of memory access consist of the that needed to search in the range search trees and to fetch the indices plus s accesses to the rules.

The update includes the search part and modification part. The complexity of the search part is similar to the search of a matching rule for a packet. The modification part needs $O(N)$ in the worst-case, where N is the number of rules. However, in reality, the worst-case is very rare. On average, the update is very fast.

The performance of our algorithm relies on s , the number of independent sets that the classifier is partitioned into. The smaller the number is, the fewer number of memory access and less memory storage are required. Fortunately, s is not expected to be high. Our experimental studies and studies in the literature support this conclusion. Reference [1] observed that every packet matches at most 4 rules. Similar small numbers have been seen in [28]. The prefix containment is quite rare in the backbone table and is limited to at most 6 [1, 65]. These characteristics of classifiers guarantee that s is small. The quantitative study is provided in Section 5.4.

5.3.7 Variations

So far, a basic version of the new algorithm has been described. In fact, many variations of this algorithm, which tailor to particular applications or hardware architectures, could be developed.

One variation is that we can increase the average search speed by sorting the indices in each entry of indices. Thus, the high priority rules are searched before the low priority rules; whenever we have a match, we will stop the search procedure.

However, this change will affect the update. This will make it difficult to find the predecessor or successor of a rule in the I -set. This can be solved by using extra memory (not larger than the index arrays) to keep track of the predecessor or successor of a rule.

Another variation is that in the index arrays we replace the index by the corresponding rule itself. This removes the need of index access and increases the search and update speed. However, this will increase the memory storage.

In the preliminary study, we use the linear search for each index in the index arrays. In fact, we can use other techniques such as a multiway search, binary search or hash search along other dimension(s) to speed up the search.

We will detail the variations in our future work.

5.3.8 A Property of the Set of Independent Sets

Our algorithm is scalable to the high dimension classification problem. The following property gives the exact meaning of this scalability.

We first give the following definition.

Definition: Let $C = \{R_1, \dots, R_N\}$ be a d -dimensional classifier, where $R_i = [F_1^i, \dots, F_d^i]$ ($i = 1, 2, \dots, N$) is a rule consisting of d fields F_j^i ($j = 1, 2, \dots, d$). We add one more field to each rule in C such that $R_i^e = [F_1^i, \dots, F_d^i, F_{d+1}^i]$ and $C^e = \{R_1^e, \dots, R_N^e\}$. The new $(d+1)$ -dimensional classifier C^e is called the $(d+1)$ -dimensional extension of C . The rule R_i^e is called the $(d+1)$ -dimensional extension of R_i .

We have the following property for a group of I -sets.

Property: Assume that C is a d -dimensional classifier. Let $\{I[0], I[1], \dots, I[s]\}$ be a group of I -sets of C . Let C^e be the $(d + 1)$ -dimensional extension of C . Let $\{I^e[0], I^e[1], \dots, I^e[s]\}$ be the corresponding $(d + 1)$ -dimensional extension of $\{I[0], I[1], \dots, I[s]\}$. Then $\{I^e[0], I^e[1], \dots, I^e[s]\}$ is a group of I -sets of C^e .

This property shows that instead of analyzing the set of I -sets with multifields, we can analyze the corresponding set of I -sets with two fields. The performance of searching a multiple dimensional classifier is not worse than that of searching the corresponding two dimensional classifier.

Specifically, we can use this property in the following way. In a multidimensional classifier, some fields may have a small number of distinct entries and therefore do not contribute much to the I -sets. In that case, we can avoid separating I -sets along these dimensions. Thus, the high dimension classification problem is converted to a low dimension classification problem. This shows why our algorithm could break the lower bound in which a d -dimensional classification must perform at least d times one dimensional range searches. This property also shows that our algorithm scales very well with respect to the dimension.

In the following section, we only do experiments for a two dimensional classification for illustrative purposes.

5.4 Experimental Study

For a classifier, we first define the repeating factor of a field. For a specified field, let n_1 be the total number of entries of the field in a classifier. Let n_2 be the

number of distinct entries of the field. The *Repeating factor* of the field is defined as n_1/n_2 . In the IP destination address field, the repeating factor measures the average times that an IP address is used in the rules. The performance of our algorithm is directly related to the repeating factor of each field. The data used in [23] shows that repeating factors are around 50 for large classifiers (with about 1M entries). The data used in [65] shows that repeating factors are around 5 for small classifiers (with about 200 entries).

For a preliminary study, as in the thesis, we construct classifiers with characteristics we have seen from field data. To do this, we download IP routing tables from [88] as bases and conduct several groups of experiments for two dimensional classifiers. The results are presented as follows.

5.4.1 Classifiers without Wildcards

We first study classifiers without wildcards. In the first group of experiments, we study the effect of the repeating factor on the performance of our algorithm. In the second group we show that the size of the classifier is not sensitive to the performance of our algorithm when the repeating factor is fixed.

We use the routing table at Mae-west taken on March 15, 2002 from [88] as a basis for constructing a classifier C . Each rule in the classifier contains two fields: the IP destination address, and the source address.

In the first group of experiment, we generate four classifiers of size 30000, with different repeating factors. In order to generate a classifier, the expected repeating factor, say f , is first given. Then from the Mae-west routing table 30000/ f

Table 5.1: Experiments with different repeating factors.

f	des	rf	src	rf	I-set
2	15422	1.95	14321	2.09	9
10	2807	10.69	2960	10.14	23
20	1422	21.10	1610	18.63	33
60	489	61.35	498	60.24	74

addresses are sampled. Next, from the $30000/f$ addresses, 30000 addresses are randomly selected as the IP destination addresses. For each destination address, the corresponding source address is randomly selected from the $30000/f$ addresses to form a rule. If the rule just formed is a duplicate rule of previously generated ones, we reselect a source address until the rule formed is unique.

Table 5.1 shows the resulting classifiers. The first column is the expected repeating factor (f). The second column is the number of distinct destination addresses (des), the third column is the repeating factor (rf) of destination address field and so on. The last column is the number of independent sets of the classifiers. We can see that the number of independent sets increases as the repeating factor increases.

In the second group of experiment, six classifiers of different sizes with the same expected repeating factor of 30 are constructed. The base prefixes are from the AADS routing table taken on March 15, 2002 from [88]. The results are shown in Table 5.2. It shows that the number of independent sets of the classifiers is not sensitive to the size of classifiers provided that the repeating factor is unchanged. This shows that our algorithm is not sensitive to the size of classifiers with a similar

Table 5.2: Experiments with different table sizes.

rules	des	rf	src	rf	I-set
2000	65	30.77	62	32.26	34
10000	323	30.96	360	27.78	40
20000	677	29.54	710	28.17	43
100000	3499	28.58	3287	30.42	45
200000	7155	27.95	6567	30.46	48
1000000	32778	30.51	33698	29.68	61

repeating factor. By observing data used in the literature, we found that it is rare that the repeating factor exceeds 100 even for large size classifiers. Together with our experimental study, we believe that our algorithm scales well to the size of classifiers. We also envision that our algorithm scales well to IPv6, since the number of addresses in IPv6 is much larger than that in IPv4 and therefore the repeating factors should be much lower.

5.4.2 Classifiers with Wildcards

We construct three classifiers with different percentages of wildcards in them. The base prefixes are from the AADS routing table taken on March 15, 2002 from [88]. The size of base prefixes is 30000. Given a percentage p , $p/2$ percent of the rules have wildcards as their destination fields and $p/2$ percent of the rules have wildcards as their source fields. Altogether, p percent of the rules have wildcards. The number of

Table 5.3: Experiments with different percent of wildcards.

% wildcards	des rf	src rf	I-set
10	1.46	1.46	6
30	1.59	1.58	7
50	1.72	1.72	6

rules is 30000 for all three classifiers. Table 5.3 shows that the number of independent sets of the classifiers is not sensitive to the percentage of wildcards in the classifiers. We note that the repeating factor is small and hence the number of independent sets is small. In fact, in a two-dimensional classifier, it is not possible to generate high percentage wildcards in one field with a high repeating factor in the other field. We can prove the following property.

Property: Assume that C is a two-dimensional classifier without duplicated rules. If there are $p\%$ rules that have wildcards in one field, then the repeating factor in the other field is less than $100/p$.

Proof: If two rules are wildcarded in one field, then the entries of these rules in the other field are distinct. Since there are $p\%$ rules that have wildcards in one field, the entries of these $p\%$ rules in the other field are distinct. Therefore, the number of distinct entries in the other field is greater than $p\%$ of the total rules. Hence the repeating factor in the other field is less than $100/p$. $\triangle$

For example, if 50% wildcards were generated in the destination field, then the repeating factor in the source field cannot be higher than two.

5.4.3 One Scenario of Implementation

We choose the third classifier in Table 5.1 for the discussion of implementation. There are 33 I -sets for this classifier, among which 7 are I_1 -sets and 26 I_2 -sets. Corresponding to these two groups of I -sets, two range search trees are created. The range search tree corresponding to field one has at most 1422 points, since the number of distinct destination addresses is 1422. Each point points to 7 rule indices. Since the number of rules is 30000, each index uses 16 bits (in fact 15 bits are enough). So the memory for the index arrays is less than 19908 bytes. Using the same argument, we find the memory for the index arrays in the other range search tree is less than 83720 bytes. For the array storing the rules, we assume that each rule uses 128 bits. Among the 128 bits, 32 bits are for the begin point of the destination address field and 5 bits for the length of the destination prefix. Another 37 bits are for the source address field. 15 bits are used for specifying priority, 32 bits for the port number and 7 bits are empty. So 30000 rules need 480000 bytes of memory. Together, 583628 bytes memory are needed, excluding the range search trees. Comparing the original rule table, our algorithm increases the memory requirements by less than a factor of one. This comes with no surprise. On one hand, the memory requirement is large if the number of I -sets is large. On the other hand, if the number of I -sets is large, the repeating factors should be large. Hence the number of distinct values in a field is small, thus the number of points in the range search tree is small. Therefore, only small memory requirements are needed to store the indices array.

The number of memory access for a search depends on the cachline. We assume

that a 128 bit cacheline is used. Then, 5 memory accesses are needed for fetching the $7 + 26$ indices and 33 memory accesses are needed for fetching the rules, including the port number. Excluding the range search, we need 38 memory accesses.

The update is fast. Since there are only 1610 distinct source addresses, the maximum number of entries of the index arrays to be modified is 1610, rather than 30000.

5.5 Previous Work

Surveys on packet classification algorithms were given in [27, 68]. Here, we highlight performance measurements for some of the algorithms.

The Recursive Flow Classification (RFC) [29] is very fast for a search. However, the memory requirement is so large and the preprocessing time is so slow that it is not suitable for large classifiers. Reference [40] proposed a Bit Vector (BV) search algorithm. For a d -dimensional classifier, the storage requirement is $O(dN^2)$, where N is the number of rules in the classifier. Query time is d times that of the time needed for a range search, plus d times that of the time for a bit vector fetching. This is equal to N/w , where w is the size of cacheline. Reference [1] added new techniques to the BV algorithm and reported an order of magnitude improvement on performance over the standard BV algorithm, with a small price of increasing memory requirements. The tuple space search algorithm [65] has a small memory requirement ($O(N)$), however, the search speed depends on the number of tuples in the classifier and it supports only prefixes rather than arbitrary ranges.

In addition, the use of hashing makes the time complexity of searches and updates nondeterministic [27]. The Fat Inverted Segment tree (FIS-tree) was proposed in [23]. The level of the FIS-tree can be adjusted to make a tradeoff between the search speed and memory requirements. Under the assumption that the cacheline is 32 bytes large and the entry size of a rule is 12 bytes, [23] reported that for a two dimensional classifier with more than 10^6 rules, the search needs less than 22 (17 respectively) memory accesses using three (two respectively) levels of the FIS-tree. The memory is at most 4.1 (7 respectively) times that of the rule table size. They did not report any experimental study on multifield classifiers. However, they pointed out that the memory requirement and memory accesses increase with a factor of l as the dimension d increases, where l is the number of levels in the FIS-tree.

5.6 Summary

We developed a novel packet classification algorithm based on independent sets. We proposed a basic data structure and an update algorithm for the data structure. We also conducted an experimental study on our algorithm.

As mentioned earlier, packet classification algorithms are measured by times of memory access, memory storage requirements, update speed and scalability, etc. Existing algorithms could perform well with respect to one or two of these measurements. Our algorithm performs well in all aspects of the criteria. It seems that our algorithm is the first to achieve such success: small memory requirements, fast search and update speed, and scalability to large classifier, to multidimensional

classifier and to IPv6. The algorithm is feasible for parallel implementation. With these merits, our algorithm could be among the best for the future high speed packet classification task.

Chapter 6

Conclusions and Industrial Outlook

In the previous chapters, we detailed our contributions to the thesis. In this chapter, we summarize these contributions. At the same time, we discuss the current and future situation in the market of IP address lookup and packet classification applications. Conclusions are made in Section 6.1 and the industrial situation is presented in Section 6.2.

6.1 Conclusions

Three high performance algorithms are presented in this thesis. The three algorithms are the IP address lookup algorithm, the IP address table update algorithm, and the packet classification algorithm. Though they address different aspects of the packet forwarding problem, they are related. The common features of them are: very small

memory requirements, inherently pipelined and/or parallel implementation, they are aimed towards hardware implementation, flexible in implementation, and scalable to the number of entries and to IPv6.

6.1.1 Contributions of the IP Address Lookup Algorithm

Chapter three presented the IP address lookup algorithm using compressed keys. In order to achieve minimum memory requirement, we combined the following three techniques.

1. We compress the keys in a tree node;
2. We use a shared pointer in a tree node;
3. We use bottom-up from the leaf to the root scheme to build the tree.

In order for the first technique to work efficiently, we give a different definition of the right point of a prefix so that points may have long trailing zeros.

The data structure is neat and the creation of the data structure from scratch is also fast. The memory requirement is the smallest, compared with all the solutions in the literature and in industry. According to current technology advances, a table with 1M IPv4 entries can be put in a single chip and the search speed is several folds faster than the products available in the market.

6.1.2 Contributions of the IP Address Table Update Algorithm

The general idea is to use the concept of independent sets to eliminate the worst case update scenarios. The algorithm is also so general that it does not depend on any search algorithm. Any static algorithm can potentially use this algorithm to convert to a dynamic algorithm. The precondition of using this algorithm is that we need resources for parallel implementation. However, the number of parallel resources is not supposed to be high.

We can combine this algorithm with the algorithm in Chapter three. In Chapter three, we proposed using two banks of memory for update. These two alternatives have their own advantages and disadvantages. Particular applications can make a suitable decision.

6.1.3 Contributions of the Packet Classification Algorithm

The packet classification algorithm presented in Chapter five uses the concept of independent sets similar to that in Chapter four. Essentially, when the rules are grouped in independent sets, they avoid being divided (or saved) in several pieces and therefore memory is saved. Using the data structure similar to the fractional cascading technique, several independent set searches can be merged to one independent set search.

As demonstrated in Chapter five, the algorithm has very small memory requirements. The search speed is neither sensitive to the size of the rule table nor to the

percentage of wildcards in the fields. It also scales well from two dimensional classifiers to high dimensional ones. We envision that it scales to IPv6. In particular, the algorithm is inherently parallel. Hardware tailored to this algorithm can achieve a very fast search speed. The update algorithm proposed is also very fast in general.

6.2 Industrial Outlook

The current market of IP address lookup and packet classification is dominated by TCAM solutions. Many companies have search engine products which use TCAM for IP address lookup and packet classification. To name a few, IDT [85], Cypress [80], NetLogic Microsystems[90], SiberCore Technologies [92] and Mosaid Semiconductor[89] are leading vendors. The market is rather dynamic. We witnessed the emergence of new startups, acquisitions of existing startups, failures of startups, and the ceasing of operations from the market of established companies.

The size of the market is difficult not only to predict but also to review. To quote Denali Memory Report v2, Issue 3, March 2003, page 8 [81]: “In 2002, the merchant NSE (network search engine) market was about \$175M, Cisco alone, which makes proprietary NSEs to its own spec, is about \$100M of consumption by itself. Adding to the confusion of NSE market sizing is the fact that the NSE ‘function’, which is addressed by a host of alternate SRAM-based, DRAM-based, and ASIC-plus-SRAM-based solutions, and is therefore, arguably 3-4 times as large.”

As we mentioned several times in this thesis, TCAM does not come without drawbacks. Usually, TCAM consumes large amount of power and is expensive. The

maximum capacity of TCAM today is 19Mb. When the table is large, several chips are needed. For example, for 1M IPv4 table, at least 4 chips (70% usage for dynamic table) are needed. This could cost up to \$1000.

Applications of IP address lookup and packet classification algorithms are standard and well defined in the Internet communication market. The reason that they did not compete well with TCAM solutions is that algorithms in the earlier times did not have better performance than that produced by TCAM solutions. Nowadays, the situation is changing. We have witnessed several companies adopting algorithm-based solutions, such as Ezchip Technologies [82] and Hywire [83], etc.. They demonstrate advantages over TCAM solutions. Though the detailed techniques of these companies are not publicly available, from the information we can gather, we believe the techniques presented in this thesis are superior to theirs.

We predict that the algorithm-based solutions will gradually replace the TCAM-based solutions and dominate the future market. The reasons are as follows:

1. The algorithm-based solutions are more flexible. The algorithm can be implemented as a single standalone chip, or FPGA, or embedded as a function unit in a network processor. The memory used can be SRAM, DRAM, or even TCAM. The memory can be on-chip, or off-chip. The performance and cost tradeoff can be easily achieved.
2. TCAM-based solutions are limited by TCAM memory latency. The current TCAM memory latency is about 10ns. It can achieve 100 Million searches per second (Msps), which is quite suitable for the current wire rate of 10Gbps and 40Gbps. If, in the future, the wire rate exceeds 40Gbps, TCAM will

not be suitable for the speed demands. In contrast, the memory latency of on-chip SRAM is about 1-2ns. On-chip SRAM can achieve 5-10 times faster speeds than TCAM. Only algorithm-based solutions can exploit this advantage. When the speed becomes un-compromised, the TCAM-based solutions will become dated.

3. Large tables will exclude the TCAM-based solutions because of the big power consumption and cost. As analyzed in [22], algorithm-based solutions can reduce router chip-count, power and cost by 80% compared to TCAM solutions.
4. Advanced algorithms already exist to provide solutions that are better than TCAM. As exemplified by the algorithms in this thesis, our algorithms have exceeded TCAM in memory requirements, speed and other performances. For example, for IPv4 address lookup, our algorithm needs about 15b per entry, while TCAM needs 32b per entry.

Bibliography

- [1] Florin Baboescu and George Varghese, “Scalable Packet Classification,” *ACM SIGCOMM*, 2001, pp. 199-210.
- [2] F. Baker, “Requirements for IP version 4 routers,” *RFC 1812*, June 1995.
- [3] R. Bayer and E. McCreight, “Organization and maintenance of large ordered indexes,” *Acta Informatica* 1, 3, Sept. 1972, pp. 173-189.
- [4] S. Bellovin and W. Cheswick, “Network Firewalls,” *IEEE Communications Magazine*, v32, n9, 1994, pp. 50-57.
- [5] M. Berg, M. Kreveld and J. Snoeyink, “Two- and Three-dimensional point location in rectangular subdivisions,” *Journal of Algorithms*, 18 (2), 1995, pp. 256-277.
- [6] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang and W. Weiss, “An Architecture for Differentiated Services,” *RFC 2475*, December 1998.
- [7] I. Bomze, M. Budinich, P. Pardalos and M. Pelillo, “The Maximum Clique Problem,” In D.-Z. Du and P. M. Pardalos, editors, *Handbook of Combina-*

- torial Optimization*, volume 4, Kluwer Academic Publishers, Boston, MA, 1999.
- [8] R. Braden, D. Clark and S. Shenker, "Integrated Services in the Internet Architecture: an Overview," *RFC 1633*, June 1994.
- [9] Milind M. Buddhikot, Subhash Suri and Marcel Waldvogel, "Space Decomposition Techniques for Fast Layer-4 Switching," *Proceedings of the IFIP Sixth International Workshop on Protocols for High Speed Networks (PfHSN '99)*, August 1999, pp. 25-41.
- [10] B. Chazelle and L. J. Guibas, "Fractional Cascading I: A Data Structuring Technique," *Algorithmica*, v1, n2, 1986, pp. 133-162.
- [11] Gene Cheung and Steven McCanne, "Optimal Routing Table Design for IP Address Lookups Under Memory Constraints," *IEEE INFOCOM*, 1999, pp. 1437-1444.
- [12] L. Chisvin and R.J. Duckworth, "Content-addressable and associative memory: Alternatives to the ubiquitous RAM," *IEEE Computer*, 2w, July 1989, pp. 51-64.
- [13] T. Chiueh and Prashant Pradhan, "High Performance IP Routing Table Lookup using CPU Caching," *proceedings of IEEE INFOCOM'99*.
- [14] D. Comer, "*Internetworking With TCP/IP Volume 1: Principles, Protocols, and Architecture, 4th edition*," Prentice Hall, Englewood Cliffs, NJ, 2000.

- [15] P. Crescenzi, L. Dardini, and R. Grossi, "IP lookups made fast and simple," *In 7th Annual European Symposium on Algorithms*, July 1999.
- [16] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," *RFC 2460*, December 1998.
- [17] M. Degermark, A. Brodnik, S. Carlsson and S. Pink, "Small forwarding tables for fast routing lookups," *Proceedings of ACM SIGCOMM*, Sept. 1997, pp. 3-14.
- [18] W. Doeringer, G. Karjoth and M. Nassehi, "Routing on longest matching prefixes," *IEEE/ACM Transactions on Networking*, 4(1), February 1996, pp. 86-97.
- [19] Richard P. Draves, Christopher King, Srinivasan Venkatachary and Brian D. Zill, "Constructing Optimal IP Routing Tables", *Proceedings of the Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE INFOCOM '99)*, New York, March 21-25, 1999.
- [20] Richard Edell, Nick McKeown, and Pravin Varaiya, "Billing Users and Pricing for TCP," *IEEE Journal on Selected Areas in Communications*, v13, n7, September 1995, pp. 1162-1175.
- [21] P. van Emde Boas, R. Kaas and E. Zijlstra, "Design and implementation of an efficient priority queue," *Mathematical Systems Theory*, 10, 1977, pp. 99-127.

- [22] EZchip technologies, "Reducing Router Chip-count, Power and Cost by 80%," *White paper*, 2003, available: <http://www.ezchip.com/html/linktowhitepaper.html>
- [23] A. Feldman and S. Muthukrishnan, "Tradeoffs for Packet Classification," *Proceedings of Infocom*, v3, March 2000, pp. 1193-2002.
- [24] Edward Fredkin, "Trie memory," *Communications of the ACM*, Volume 3, Issue 9 (August 1960), pp. 490 - 499.
- [25] B. Gleeson, A. Lin, J. Heinanen, G. Armitage and A. Malis, "A Framework for IP Based Virtual Private Networks," *RFC 2764*, February 2000.
- [26] Pankaj Gupta, "Algorithms for Routing Lookups and Packet Classification," *Ph. D. Thesis, Stanford University*, December 2000.
- [27] Pankaj Gupta and Nick McKeown, "Algorithms for Packet Classification," *IEEE Network Special Issue*, March/April 2001, v15, n2, pp. 24-32.
- [28] Pankaj Gupta and Nick McKeown, "Packet Classification Using Hierarchical Intelligent Cuttings," *IEEE Micro*, v20, n1, January/February 2000, pp. 34-41.
- [29] Pankaj Gupta and Nick McKeown, "Packet Classification on Multiple Fields," *Proc. Sigcomm, Computer Communication Review*, v29, n4, September 1999, pp. 147-160.
- [30] Pankaj Gupta, Steven Lin and Nick McKeown, "Routing Lookups in Hardware at Memory Access Speeds," *Proc. Infocom*, April 98, San Francisco.

- [31] R. Hinden and S. Deering, "Internet Protocol Version 6 (IPv6) Addressing Architecture," *RFC 3513*, April 2003.
- [32] Nen-Fu Huang and Shi-Ming Zhao, "A Novel IP-Routing Lookup Scheme and Hardware Architecture for Multigigabit Switching Routers," *IEEE Journal on Selected Areas in Communication*, Vol.17, No.6, June 1999, pp. 1093-1104.
- [33] R. Jain, "A Comparison of Hashing Schemes for Address Lookup in Computer Networks," *IEEE Transactions on Communications*, vol. 40, no. 10, 1992, pp. 1570-1573.
- [34] D. S. Johnson and M. A. Trick (eds.), "Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge," v26, *DIMACS*, American Mathematical Society, 1996.
- [35] Y. Jyh-haw, C. Randy and N. W. Richard, "Interdomain Access Control with Policy Routing," *Proceedings of the IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, Oct. 1997, pp. 46-52.
- [36] K. Kim and S. Sahni, "IP lookup by binary search on length," *IEEE Symposium on Computers and Communications*, 2003, pp. 77-82.
- [37] D. Knuth, "*The Art of Computer Programming*," volume 1, Addison Wesley, 1973.
- [38] D. Knuth, "*The Art of Computer Programming*," volume 3, Addison Wesley, 1973.

- [39] C. Labovitz, G. R. Malan and F. Jahannian, "Internet routing instability," *Proceedings of ACM SIGCOMM '97*, 1997, pp. 115-126.
- [40] T. V. Lakshman and D. Stiliadis, "High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching," *Proceedings of ACM Sigcomm*, September 1998, pp. 191-202.
- [41] B. Lampson, V. Srinivasan and G. Varghese, "IP Lookups Using Multi-way and Multicolumn Search," *IEEE/ACM Transactions on Networking*, 7, 1999, pp. 324-334.
- [42] P.A. Larson, "Dynamic Hash Tables," *Communications of the ACM*, vol. 31-4, 1988, pp. 446-457.
- [43] Huan Liu, "Routing Table Compaction in Ternary-CAM," *IEEE Micro*, Jan/Feb, 2002, pp. 58-64.
- [44] H. Lu and S. Sahni, "O(log n) dynamic router-tables for ranges," *IEEE Symposium on Computers and Communications*, 2003, pp. 91-96.
- [45] A. McAuley, P. Francis, "Fast Routing Table Lookup Using CAMs," *IEEE INFOCOM 1993*, vol. 3, March 1993, pp. 1382-1391.
- [46] E. McCreight, "Priority Search Trees," *SIAM Jr. on Computing*, 14, 1, 1985, pp. 257-276.
- [47] D. R. Morrison, "PATRICIA - Practical Algorithm to Retrieve Information Coded in Alphanumeric," *Journal of the ACM*, 15(4), Oct 1968, pp. 514-534.

- [48] S. Nilsson and G. Karlsson, "IP-address lookup using LC-tries," *IEEE Journal on Selected Areas in Communication*, 17(6), June 1999, pp. 1083-1092.
- [49] C. Partridge, etc., "A 50-Gb/s IP Router," *IEEE/ACM Transactions on Networking*, vol. 6, no. 3, June 1998, pp. 237-248.
- [50] T. Pei and C. A. Zukowski, "Putting routing tables into silicon," *IEEE Network*, January 1992, pp. 42-50.
- [51] J. Postel, "Internet Official Protocol Standards," RFC 1780, *Internet Architecture Board*, March 1995.
- [52] F. Preparata and M. I. Shamos, "*Computational Geometry: an Introduction*," Springer-Verlag, 1985.
- [53] R. Ramaswami and K.N. Sivarajan, "*Optical Networks: A practical perspective*," Morgan Kaufmann, San Francisco, CA, 1998.
- [54] Y. Rekhter and T. Li, "An Architecture for IP Address Allocation with CIDR," *RFC 1518*, September 1993.
- [55] M. A. Ruiz-Sanchez, E.W. Biersack and W. Dabbous, "Survey and taxonomy of IP address lookup algorithms," *IEEE Network* 15, 2 March/April, 2001, pp. 8-23.
- [56] S. Sahni and K. Kim, "Efficient Construction Of Fixed-Stride Multibit Tries For IP Lookup," *Proceedings 8th IEEE Workshop on Future Trends of Distributed*, 2001, pp. 178-184.

- [57] S. Sahni and K. Kim, "Efficient Construction Of Variable-Stride Multibit Tries For IP Lookup," *Proceedings IEEE Symposium on Applications and the Internet*, SAINT, 2002, pp. 220-227.
- [58] S. Sahni and K. Kim, "O(log n) dynamic packet routing," *IEEE Symposium on Computers and Communications*, 2002, pp. 443-448.
- [59] S. Sahni, K. Kim and H. Lu, "Data structures for one-dimensional packet classification using most-specific-rule matching," *International Journal on Foundations of Computer Science*, 14, 3, 2003, pp. 337-358.
- [60] H. Samet, "*Design and Analysis of Spatial Data Structures*," Addison Wesley, ISBN 0-201- 50255-0, 1990.
- [61] Kari Seppänen, "Novel IP address lookup algorithm for inexpensive hardware implementation," *WSEAS Transactions on Communications*, 2002, Vol. 1, No. 1, pp. 76-84.
- [62] Devavrat Shah and Pankaj Gupta, "Fast Updates on Ternary-CAMs for Packet Lookups and Classification," *IEEE Micro*, vol. 21, no. 1, January/February 2001.
- [63] K. Sklower, "A tree-based packet routing table for Berkeley Unix," *Proceedings of the 1991 Winter USENIX Conference*, Dallas, 1991, pp. 93-99.
- [64] V. Srinivasan, "Fast and Efficient Internet Lookups," *Ph. D. Thesis, Washington University in St. Louis*, June 1999.

- [65] V. Srinivasan, S. Suri and G. Varghese, "Packet Classification Using Tuple Space Search," *Proceedings of ACM Sigcomm*, September 1999, pp. 135-146.
- [66] V. Srinivasan, S. Suri, G. Varghese and M. Waldvogel, "Fast and Scalable Layer four Switching," *Proceedings of ACM Sigcomm*, September 1998, pp. 203-214.
- [67] V. Srinivasan and G. Varghese, "Fast address lookups using controlled prefix expansion," *ACM Transactions on Computer Systems*, 17(1), February 1999, pp. 1-40.
- [68] X. Sun, "IP Address Lookups and Packet Classification: A Tutorial and Review," Technical Report, No. 002, Carleton University, 2002.
- [69] X. Sun and Y. Zhao, "Packet Classification Using Independent Sets," *IEEE Symposium on Computers and Communications*, 2003.
- [70] X. Sun and Y. Zhao, "An IP Address Lookup Algorithm Using Compressed Keys," Technical Report, No. 005, Carleton University, 2003. (patent pending.)
- [71] S. Suri, G. Varghese and P.R. Warkhede, "Multiway range trees: scalable IP lookup with fast updates," *Proc. IEEE GLOBECOM '01*, v3 2001, pp. 1610 -1614.
- [72] Henry Hong-Yi Tzeng and Tony Przygienda, "On fast Address-Lookup Algorithms," *IEEE Journal on Selected Areas in Communication*, Vol.17, No.6, June 1999, pp. 1067-1082.

- [73] J. P. Wade and C. G. Sodini, "A Ternary Content Addressable Search Engine," *IEEE Journal of Solid State Circuits*, 1989, pp. 1003-1013.
- [74] M. Waldvogel, "Fast Longest Prefix Matching: Algorithms, Analysis and Applications," *Ph. D. Thesis, Swiss Federal Institute of Technology, Zurich*, 2000.
- [75] M. Waldvogel, G. Varghese, J. Turner and B. Plattner, "Scalable high speed IP routing lookups," *ACM Computer Communication Review* 27(4), October 1997, pp. 25-36.
- [76] Priyank Ramesh Warkhede, Subhash Suri and George Varghese, "Fast Packet Classification for Two-Dimensional Conflict-Free Filters," *Proceedings IEEE INFOCOM 2001*, vol. 3, April 2001, pp. 1434-1443.
- [77] D. B. West, "Introduction to Graph Theory," Prentice Hall, Upper Saddle River, NJ, 2001.
- [78] D. E. Willard, "Log-logarithmic Worst-case Range Queries Are Possible in Space $\Theta(N)$," *Information Processing Letters*, 17(2), 1983, pp. 81-84.
- [79] <http://bgp.potaroo.net/>
- [80] <http://www.cypress.com/>
- [81] <https://www.ememory.com/docs/servlet/ememory.drs.EmemoryDmrHome>
- [82] <http://www.ezchip.com/>
- [83] <http://www.hywire.com/>

- [84] <http://www-3.ibm.com/chips/products/asics/products/ememory.html>
- [85] <http://www.idt.com/>
- [86] <http://www.linleygroup.com/usefullinks.html#>
- [87] <http://www.mcvax.org/~jhma/routing/ipv6/>
- [88] http://www.merit.edu/ipma/routing_table/
- [89] <http://www.mosaid.com/corporate/home.htm>
- [90] <http://www.netlogicmicro.com/>
- [91] <http://www.npforum.org/>
- [92] <http://www.sibercore.com/>