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

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

Fast IP Address Lookup Using Index Table

by

Xiaojun Nie, M.Eng

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

MASTER OF SCIENCE

in Information and Systems Science

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

Carleton University

Ottawa, Ontario, Canada

August 22, 2003

© Copyright
2003 Xiaojun Nie

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: 0-612-88781-2

Our file Notre référence

ISBN: 0-612-88781-2

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.

Canada

The undersigned recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis

Fast IP Address Lookup Using Index Table

Submitted by

Xiaojun Nie

in partial fulfillment of the requirement for the degree of
Master of Science in Information and Systems Science

Professor Yiqiang Q. Zhao

Chair, School of Mathematics and Statistics

Carleton University

© August, 2003

Abstract

The explosive growth of the Internet and employment of all kinds of new applications over IP make routers as the bottleneck in the Internet, which is mainly caused by IP address lookup. This thesis proposes a new IP address lookup algorithm to improve the performance of routers. In our proposed algorithm, an index table is used to reduce the search range from the whole database to a small search group whose size is limited to 16. All prefix entries are stored in a hash table that is organized by different length hash keys. To lookup a destination address, the first eight bits of the address are extracted to check the index table. Based on the head information in the index table, the hash table is searched. The search result is either the next hop pointer or a small search group. In the latter case, it is necessary to search the small search group. The index table and the small search group are prefetched into cache memory on general-purpose processors (GPP). The implementation of the algorithm on GPP is carried out. As a result, one main memory access and several fast caching searches are needed to lookup an address. The memory requirements can be controlled by selecting the size of the small search group. Since a prefix insertion/deletion only involves a small number of memory modifications, the algorithm performs quick update. The adaptation of the algorithm to NP architecture will be studied in the future.

Acknowledgements

I would like to express my high appreciation to my supervisor, Professor Yiqiang Q. Zhao, for his knowledge, guidance and supervision during my studies. I would also thank him for believing in my ideas and devoting time and energy in challenging and inspiring discussion. Many thanks to people from Alcatel Networks for their suggestion, especially to Gerard Damm, Jerome Cornet, David J. Wilson, Milan Zoranovic and Vincent Mouilleron. . I would also like to thank my colleges, Xuehong Sun, Zhen Xu and my friend Gerg Boye for their friendship and help. The financial support of Alcatel Networks is highly appreciated. Finally I would like to express my deep gratitude to my lovely wife Lianne Zhou and our unborn baby boy, my Aunt Youdong Zhou and my parents for their unfailing love, encouragement and support.

Acronyms

ASIC	Application-Specific Integrated Circuit
BGP	Boarder Gateway Protocol
BMP	Best Matching Prefix
CAM	Content Addressable Memory
CIDR	Classless Inter-Domain Routing
CPU	Center Processor Unit
DRAM	Dynamic Random Access Memory
DWDM	Dense Wavelength Division Multiplexing
FPGA	Field-Programmable Gate Array
GPP	General-Purpose Processor
IP	Internet Protocol
IPMA	Internet Performance Measurement and Analysis
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
LC-trie	Level Compressed Trie
NP	Network Processor
OC-192	Optical Carrier 192
PC	Personal Computer
SRAM	Static Random Access Memory

Table of Contents

Acceptance Form	ii
Abstract	iii
Acknowledgments.....	iv
Acronyms	v
Table of Contents	vi
List of Tables	ix
List of Figures	x
1. Introduction	1
1.1 Route Lookup and Route	1
1.1.1 Route Lookup.....	1
1.1 .2 Route Table.....	2
1.2 The Difficulties in IP Address Lookup.....	4
1.2.1 Classless Inter-Domain Routing (CIDR).....	4
1.2.2 Routing Table Size.....	5
1.2.3 Internet Traffic.....	6
1.2.4 High Speed Internet.....	6
1.3 Existing Solutions.....	7
1.3.1 Multi-way and Multi-column.....	9
1.3.2 The Lulea Algorithm	11
1.3.3 Binary Search on Prefix Lengths.....	14
1.4 Outline of Rest of Thesis	16
2. Cache and Hash Tables	17
2.1 Cache Memory	17
2.2 Hash Tables	20

3. Motivation.....	23
3.1 Problems in IP Lookup	23
3.1.1 Difficult Updates	23
3.1.2 Worst Case Lookup Speed	24
3.1.3 Low Flexibility	24
3.1.4 High Memory Usage	25
3.2 Motivation, Statement and Objective.....	25
3.3 Scope of Thesis	26
 4. Proposed Algorithm.....	 27
4.1 Basic Idea.....	27
4.2 Data Structures of the Proposed Algorithm.....	31
4.2.1 Data Structure of Index Table.....	31
4.2.2 Data Structure of Hash Table.....	34
4.2.3 Data Structure of Small Search Groups.....	38
4.3 Pseudo codes for the Proposed Algorithm.....	40
4.3.1 Pseudo code Notations.....	41
4.3.2 Lookup Table Creation.....	42
4.3.2.1 Building Index Table.....	42
4.3.2.2 Building the Hash Table.....	47
4.3.3 IP Address Search.....	49
4.3.4 Lookup Table Updates.....	53
4.3.4.1 Insertion of Prefix.....	53
4.3.4.2 Deletion of Prefix.....	56
4.4 Analysis of Algorithm.....	59
4.4.1 Memory Accesses for Lookup.....	59
4.4.2 Memory Usage of Lookup Table.....	61
4.4.3 Prefix Update Rate.....	62
4.4.4 Flexibility of Algorithm.....	64
 5. Implementation and Performance Analysis.....	 66
5.1 Prefix Expansion Results.....	67
5.2 Memory Access Results for Lookup.....	69
5.2.1 Main Memory Access Results.....	69

5.2.2 Cache Memory Access Results.....	73
5.3 Results of Lookup Table Memory Usage	76
5.4 Lookup Table Update Results.....	78
 6. Conclusions and Future Work	 80
6.1 Conclusions.....	80
6.2 Summary of contributions	81
6.3 Future Work	82
 References	 84

List of Tables

Table 1.1: A simple forwarding table	5
Table 4.1: Memory access comparisons of different lookup algorithms	61
Table 4.2: Memory requirement comparisons of several lookup algorithms	62
Table 4.3: Update comparisons of different lookup algorithms	64
Table 4.5: Flexibility comparisons of different lookup algorithms	64
Table 5.1: The number of prefixes for four databases.	66
Table 5.2: The prefix expansion results for four databases.	67
Table 5.3: The hash lookup results on the implementation.	73
Table 5.4: Results on cache memory accesses for checking the index table.	74
Table 5.5: The results on cache memory accesses for searching small search groups.	75
Table 5.6: Implementation results of updates.	79

List of Figures

Figure 1.1: Diagram of the architecture of a router.	2
Figure 1.2: (a) An Internet example with five networks and four routers.	3
Figure 1.2: (b) The routing table in R.	3
Figure 1.3: Prefixes are encoded into different ranges.	9
Figure 1.4: The initial 16-bit array with pointers to corresponding six-way search nodes	11 12
Figure 1.5: The three levels of the data structure	12
Figure 1.6: Finding the pointer index	13
Figure 1.7: Improved branching decisions due to markers.	15
Figure 2.1: Relationship between CPU and cache.	18
Figure 2.2: The cache memory consists of a directory a cache data memory buffers (to isolate the cache subsystem from the system bus) and the cache controller	20 20
Figure 2.3 (a) A hash of size 13 storing 10 integer keys with collisions resolved by the chaining method. The hash function is $h(k)=k \bmod 13$	21 21
Figure 2.3 (b) The number of collisions is reduced when N is changed to 11	21
Figure 4.1: The process of breaking larger groups into small groups.	29
Figure 4.2: Basic process of IP lookup using the index table.	31
Figure 4.3: The structure of an element in the index table.	32
Figure 4.4: An example of an index table structure.	33
Figure 4.5: The structure of a sub-index table.	34
Figure 4.6: The structure of an element in the hash table.	35
Figure 4.7: Converting an original forwarding table into a hash table with hash key length 2.	36

Figure 4.8: Converting an original forwarding table into a hash table with hash key length 2 and 3.	36
Figure 4.9: Chaining resolution for hash collisions.	38
Figure 4.10: The structure of the small search group.	39
Figure 4.11: An example of converting a prefix into a word.	40
Figure 4.12: Initialization of the index table.	43
Figure 4.13: Data collections in building the index table.	44
Figure 4.14: Building the rough index table.	45
Figure 4.15: Extension of prefixes.	46
Figure 4.16: Refining the index table	47
Figure 4.17: Initialization of the hash table.	48
Figure 4.18: Building the hash table	49
Figure 4.19: An example of IP address lookup using index table.	51
Figure 4.20: Pseudo code of IP address lookup using index table.	52
Figure 4.21: Pseudo code of inserting prefix.	55
Figure 4.22: Pseudo code of prefix deletion.	58
Figure 5.1: Results of prefix extensions results on simulation.	68
Figure 5.1: (a) The hash table size versus average collisions.	68
Figure 5.1: (b) The hash table size versus maximum collision.	68
Figure 5.2: (a) The hash functions versus average collision.	70
Figure 5.2: (b) The hash functions versus maximum collisions.	70
Figure 5.3: (a) Average collisions versus the hash functions.	72
Figure 5.3: (b) The maximum collisions versus the hash function.	72

Figure 5.5: Frequency histogram of the number of items in the small search groups. 76

Figure 5.6: The hash table size versus the memory usage. 77

1. Introduction

This chapter will give a background of this research. Firstly we introduce the basic knowledge of Internet Protocol (IP) route tables and route lookup. Secondly we briefly address the difficulties in the route lookup in today's Internet routers. Thirdly, the existing solutions for route lookup are given. Finally the rest of this thesis is overviewed.

1.1 Route Lookup and Route Table

Route lookup is a key operation in an Internet router, while a route table in the router stores all routes, one of which will be selected as a path to forward a packet. This section briefly introduces the route lookup and route tables.

1.1.1 Route Lookup

An Internet router is a special computer dedicated to interconnect all kinds of networks. However, its primary function is to transfer packets from a set of input links to a set of output links; which is called packet forwarding. To forward packets, the router must make a decision which next-hop router the incoming packets should be sent to. Besides transferring packets, the router also implements routing protocols, runs software to configure and manage the router, and performs special processing on incoming packets such as filtering packets for security reasons, delivering packets according to their priorities, and maintaining statistics on the number of packets sent by different networks.

Figure 1.1 shows the diagram of the basic architecture of a generic router, which consists of four components including input ports, output ports, a switching fabric and a routing processor. If a packet is coming at an input port, the processor extracts the

destination address from the header of this packet. Then the processor looks up the forwarding database with the destination address as a key, which produces the address of the next-hop router and determines the output ports through which the packet should be sent. This lookup operation in the router is called *route lookup* or *IP address lookup* [1]. The packet is transferred by a switching system from the input port to the output port identified by the address lookup operation. This operation is called *switching* operation. The route lookup and switching is called packet processing, which is a time consuming operation. A router may perform other functions such as scheduling at the output link to ensure fairness among competing packet streams, providing delay bounds for time-critical traffic etc.

Figure 1.1: Diagram of the architecture of a router.

1.1.2 Route Table

Routing table (also called IP routing table) is used in a router to store information about possible destinations and how to reach them. When the IP software in a router

needs to transmit a packet, it consults the routing table to decide where to send the packet[2].

A routing table contains pairs (N, R) , where N is the IP address of a destination network (or network identifier, *net-id*), and R is the IP address of the next router from which the network N can be reached. Here R is also called next hop. Thus, the routing table in a router only specifies one step along the path from this router to the destination network N , so the router does not know the complete path to N . In the following, we use a concrete example, as shown in Figure 1.2, to illustrate a routing table.

(a)

Network	Next hop
10/8	124.5.0.2
124.5/16	Deliver directly
192.16.67/24	134.56.0.4
134.56/16	Deliver directly
134.32/16	134.56.0.6

(b)

Figure 1.2: (a) An Internet example with five networks and four routers; (b) the routing table in router Q.

Figure 1.2 (a) consists of five networks connected by four routers. In Figure 1.2 (b), the routing table gives the routes that router Q uses. Q connects directly to network 124.5/16 and 134.56/16; it can use direct delivery to send packets to a host on either of these networks. If a packet with the destination address 192.16.67.25 arrives at the router Q, Q looks up the routing table to find the next hop 134.56.0.4, and Q routes the packet to address 134.56.0.4, the address of router T. T will then deliver the packet directly.

The size of the routing table depends on the number of networks. When the number of networks increases, the size of the routing table grows. The size of the routing table varies in different level routers. Backbone routers connecting a few thousand smaller networks usually have a large route table.

1.2 The Difficulties in IP Address Lookup

IP address lookup did not face any difficulties during the period when class system addresses were being used. But the lookup has become the most important challenge in today's Internet routers. The factors making the IP address lookup more difficult include classless inter-domain routing (CIDR), route table size, Internet traffic and high speed Internet, among which CIDR is preliminary reason for IP lookup difficulties.

1.2.1 Classless Inter-Domain Routing (CIDR)

In CIDR, to reduce the size of the routing table, only IP prefixes are stored in the routing table [3]. Each prefix covers a number of IP addresses. With a destination IP address, the IP router searches its routing table to find which prefixes match this address. Since CIDR allows hierarchical allocation of IP addresses, a short prefix may cover several long prefixes. So when a router looks up the prefix for a destination address, it

may result in several different prefixes. In this case, only the longest prefix is selected as the prefix matching the destination address. This is called longest prefix matching (LPM) or best prefix matching (BPM) [4]. We use an example to illustrate LPM.

Table 1.1: A simple forwarding table.

Prefix	0101*	0101101*	010110101011*
Next hop	H1	H2	H3

Table 1.1 is a forwarding table example with three entries. Given an IP address 010101101011, only prefix 0101* matches this address. So the next hop is H1. If the address is 010110101011, all three prefixes match this address. But the longest prefix 010110101011* is the best matching, hence the next hop is H3.

Unlike an IP address in the class system in which matching prefix can be determined according to its head bits, in CIDR the length of the longest matching prefix cannot be determined based on the destination address of an arriving packet. To search for the longest matching prefix, a router needs to determine both the matching prefixes and length of the longest matching prefix. Thus, regular search algorithms such as hashing, binary searches in which the search key is a fixed length cannot directly be employed to find the best matching prefix. Consequently, more sophisticated search algorithms are employed to do the best prefix matching.

1.2.2 Routing Table Size

In spite of the aggregation in CIDR, the sizes of route tables in Broader Gateway Protocol (BGP) routers keep growing. The size of a BGP route table can reach 100,000 by the year 2003 [5]. According to [6], factors that contribute to the routing table growth

include multi-homing, load balancing, address fragmentation, and failure to aggregate to route table size.

Packet forwarding speed is decreased with a dramatic growth of a route table. With more and more prefix entries being added into the table, route table characteristics such as prefix length distribution will be changed. But the performance of some routing lookup algorithms heavily depend on these characteristics. In addition, the growth of the table size requires more memory space to store the prefix entries. The LPM lookup schemes based on the hardware-oriented solution such as Content Addressable Memories (CAMs) and caches scale poorly with the routing table size.

1.2.3 Internet Traffic

Internet traffic is doubling every three months from 1995 [7]. This is partly due to increased number of users [8], but new Internet applications such as e-business, videoconference also make significant contributions to the traffic growth. Hence fast packet processing is required in the Internet routers to accommodate heavy traffic. Since the IP address lookup is the most time consuming operation in packet processing, a fast lookup algorithm is necessary to improve the performance of packet forwarding.

1.2.4 High Speed Internet

In the early stages of the Internet, the bandwidth requirement was fairly low so that the routers based on software running on general-purpose microprocessors could support low speed links such as 10Mbps Ethernet. But with the advent of many new bandwidths consuming applications such as voice over IP and videoconferencing, high speed Internet is required to satisfy these applications. With the introduction of the optical fiber

technology such as DWDM [9], the data rates of links have increased rapidly. The requirement of throughput for routers also goes up. Internet routers are required to support high-speed Internet links such as OC-192 (or 10Gps) by 2002 [10]. New technologies on hardware such as high-speed processors, and fast access Dynamic Random Access Memory (DRAM) can accelerate packet processing, but this improvement is not enough for the growing throughput requirement. Therefore, to remove the bottleneck in routers, the packet processor requires a high performance algorithm for route lookup.

1.3 Existing Solutions

To look up the longest matching prefix in a routing table, three earlier and simpler searching schemes, binary trie [11], Patricia trie [12] and path-compressed trie [13] were developed. These algorithms work well in the early stages of CIDR, but could not satisfy high requirements of packet forwarding due to the quick developments of the Internet industry. To meet the requirements, many new algorithms have been proposed to speed up the IP address lookup rate.

In general, these new algorithms can be categorized into software-based algorithms, hardware-based algorithms and hybrid algorithms. Software based algorithms include level-compressed tries (LC-Trie) [14][15], controlled prefix expansion [16], binary search on prefix lengths [17] and dynamic tries [18]. These algorithms improve their search performance mainly through efficient data structures. By arranging the prefixes into the data structures, the algorithms can perform IP address lookup with fewer memory accesses. Most of these approaches use tries as the data structures. Software based algorithms are more flexibly applied in different devices, since they usually do not

have constraints of hardware. But these kinds of lookup schemes have more difficulties in satisfying the high-speed requirement in today's backbone routers.

In hardware-based algorithms such as parallel multi-hash paths [19], multi-bit tries in hardware [20][21], four-level binary trie [22], encoding and compression algorithm [23], trie bitmap [24], CS-Trie [25], Content Addressable Memories (CAMs) [26][27] and IP lookup using CPU cache [28], an efficient approach to achieve the improvement for lookup is to make the routing table small enough to fit into faster cache memory. Furthermore, hardware solutions can potentially use parallelism to gain lookup speed. However, the penalty of using hardware-based algorithms is the cost of expensive main memory accesses, due to the cache miss. Moreover, the implementation of hardware-based algorithms cannot rely on an existing apparatus like large on-chip caches or multiple pipeline execution units.

In practice, there is no absolute boundary between software based schemes and hardware based schemes. Without considering hardware, software-based lookup would have difficulties to improve search performance. Similarly, hardware based solutions are not easy to be implemented without software assistant. From many proposed approaches in literature, the hybrid algorithm, a combination of software and hardware, provide an efficient way to achieve high performance address lookups.

There are other proposed algorithms [29][30][31][32][33], some of which are not based on software or hardware, but on the Protocol. All of these algorithms have their own advantages regarding the implementation environment, the routing table, flexibility, scalability, or even the routers according to [34][35][36]. In this section, three new

algorithms: Multi-way and multi-column algorithm [37], the Lulea algorithm [38], and the binary search on prefix lengths algorithm [17], will be described in detail.

1.3.1 Multi-Way and Multi-Column

This algorithm was developed to improve the performance of any best prefix matching scheme using an initial array indexed by the first X bits of an IP address. It also uses the cache lines to do a multi-way search with 6-way branching and claims to be able to extend the binary search solution and the multi-way search solution to IP Version 6 (IPv6).

This algorithm treats each prefix as a range, converts it using the start and the end of the range, and arranges ranges in a binary search table. For example, there are three prefixes: 1*, 101*, 101010* representing 6-bit addresses, each of which is encoded into a 6-bit range, as shown in Figure 1.3. Any range between two consecutive numbers such as 101000 and 101111 corresponds to a unique prefix. For instance, address 101110 is located in the range 101000 to 101111, so the corresponding matching prefix is 101*. In this case, every exact match corresponds to a unique prefix, which makes it possible to map the range in the binary search table to the corresponding prefix.

Figure 1.3: Prefixes are encoded into different ranges.

With this arrangement, a binary structure can be modified into an n -way structure by adding $n-1$ keys and $2n-1$ pointers in each node in the binary structure, so the average search time from $\log_2 N$ for a binary search is reduced to $\log_n N$ for an n -way search, where N is the number of prefix entries in a routing table. To fit one search node in the search table into a 32-byte cache line, 6 is the best value for n . Because each key or pointer is a 16-bit string and each six-way searching node has 5 keys and 11 pointers, there are a total of 256 bits (or 32 bytes) for each node. When the first element in this search node is read, the entire node would be pre-fetched into a cache line. Thus searching the remaining elements from this node only involves cache memory accesses that is much faster than a main memory access.

With 6-way search, this algorithm require six main memory accesses in the worst case to search a routing table with 30,000 prefixes. To further reduce the number of memory accesses, a pre-computed prefix table (or initial table) is employed. The initial table is an array of 2^{16} elements, which can be accessed by using the first 16 bits of the prefix as the index to this table. For example, for a prefix 12.24.78/24 the first 16 bits, 12.24, corresponds to the index 120 to the table. So with this index, we can directly go to the 120th element of this table without searching other elements. For a prefix whose length is smaller than or equal to 16, the table gives an exact prefix match. Otherwise the search result in the initial table is a pointer to a multi-way search table. Therefore, with the initial table, the search is either an exact match, or converts to a multi-way search at the cost of one main memory access. This is explained in Figure 1.4. In Figure 1.4, if the result of searching the initial table is a pointer to the search table, then the search node referenced by the pointer will be pre-fetched into one whole cache line.

Figure 1.4: The initial 16-bit array with pointers to corresponding six-way search nodes [42].

To search a given IP address, the first 16 bits of this address are extracted as an index to the initial table. If the result of searching the initial table is a next hop pointer, the lookup is over. If the result is a multi-way search table pointer, the node pointed to by this pointer is retrieved into a cache line for further searching. If the matching prefix is found in this node, the lookup is terminated, otherwise, another search node in this search table will be fetched into the cache line, and so on.

1.3.2 The Lulea Algorithm

This is a forwarding table data structure designed for quick route lookup. The structure is small enough to fit the whole table in the cache memory of a regular purpose processor [41]. The data structure that is a completed prefix tree, with a depth of 32, is divided into three levels as shown in Figure 1.5. Level one covers the prefix tree down to depth 16, level two from depth 17 to 24, and level three from depth 25 to 32. If a part of the prefix tree extends below level one, then level two describes that part of the tree. The

result of searching one level of the data structure is either an index to the next-hop table or an index to an array of chunks for the next level. In the latter case, the next level will be further searched to find longest match prefix.

Figure 1.5: The three levels of the data structure [41].

The data structure of level one consists of four parts: A base index array, a code word array, a maptable and a pointer array. The base index array contains the offset to find a big group of pointers. The code word consists of two parts: A ten-bit part and a six-bit part. The six-bit part determines a small group of pointers in the big group found in the base index array. The ten-bit part combined with four bits of an IP address is used to locate the exact pointer in the small group of pointers, which is stored in the maptable. The pointer array contains the pointers to the next-hop table or next level. Detailed explanation of these four part can be found in [41].

Searching level one can be described using Figure 1.6. Given an IP address, the first 10 bits of the address are extracted to find a big group of pointers in the base index array. Then the first 12 bits of the IP address are used to find a code word in the code word array, say W . The code word W has two parts: A six-bit part and a ten-bit part. The six-bit part is the position of the small group of pointers, and the ten-bit part and the low four bits of the first 16 bits of the address are combined to search a maptable to find the exact

position of the pointer, as shown in Figure 1.6.

Figure 1.6: Finding the pointer index [41].

Level two and level three in the data structure are made of chunks. A chunk covers a sub-tree of height eight and contain at most 256 nodes. The sub-trees can be divided into three kinds of chunks according to the number of nodes: Sparse (one to eight nodes), dense (9 to 64 nodes) and very dense (65 to 256 nodes). Sparse chunks are represented by an array of the 8-bit indices of the nodes, plus eight 16-bit pointers, and values 1 to 8 are placed in decreasing order using the binary search tree. The dense and very dense trunks have a similar structure to level one, and are searched analogously.

To establish such a forwarding table, the size of the forwarding table for the database Mae-East with 32732 entries is 160K bytes, and for the database AADS with 1109 entries it is only 28K bytes. Therefore, such a forwarding table can be pre-fetched into a secondary cache and do quick lookups. With the Alpha 21164 processor, in the worst case the search time is 444ns.

1.3.3 Binary Search on Prefix Lengths

Waldvogel, Varghese, Turner and Plattner [17] developed this algorithm by modifying the linear search of hash tables. In such a linear search, there are W different hash tables organized according to the length of prefixes, where W is the number of the distinct lengths in a routing database. Multiple hash tables are used to find the best matching prefix starting with the longest length and going backwards in decreasing order. The search complexity for a linear search is $O(W)$, thus in the worst case 32 memory accesses are needed for an IPv4 address. If a binary search on prefix lengths are performed rather than a linear search, the number of hash lookups is cut down to $O(\log_2 W)$ from $O(W)$. Suppose that there are 32 possible prefix lengths in a prefix database. The binary search algorithm first does a hash lookup on the first 16 bits of the destination address to see whether it matches any of the prefixes of length 16. If it does, the algorithm continues to look for a longer prefix, so the hash table with prefixes of length 24 is searched. Otherwise, it will go on to prefixes of length 8, and so forth.

However, there is a problem with binary search on prefix lengths. For instance, suppose that there are three prefixes 101*, 1010* and 10111*. Given a 6-bit input IP address 101110, first we check the middle length (here it is four) with 1011*, and it fails. So we go down and check length three (here it is 101*), and it matches. Thus we will think the prefix 101* matches the IP address. But actually the prefix 10111* is the best match for this address. This problem can be solved by adding a marker into the prefixes of median length (here the marker 1011 is added). Now we have a match for the marker of prefixes of length four, then go up to length five and find the best matching prefix 10111*. With this approach, all prefixes larger than the middle length must have a

marker in the prefixes of middle length (here it is four). The structure is shown in Figure 1.7. The longest prefixes will have more markers than other prefixes. That means that one prefix contributes at most $\log_2 W$ markers.

Figure 1.7: Improved branching decisions due to markers.

Markers can prevent the longer prefix from match failure, but it also causes another problem. We use the above example to search IP address 101100. A match happens in the prefixes of length 4, but we fail when we check length 5. The search results in no match, but apparently the prefix 101* can match this IP address. To find the prefix 101*, we have to go back and check length 3. If we do this, in the worst case the search time is not $\log_2 W$, but is still $O(W)$, which is the same as a linear search.

To avoid backtracking, a method called pre-computation is employed in this algorithm. When a marker is inserted, the best matching prefix of this marker is attached to this marker. In the above example, the best matching prefix for the marker 1011 is 101*. When we hit the marker 1011, the prefix 101* is kept as the best matching prefix. We continually check length five and fail. So the BMP is the prefix 101* and

backtracking is avoided.

In this algorithm, a series of optimizations were also explored to reduce the average number of hash lookups. An asymmetric binary search trie was applied to improve the average case performance, in which the most promising prefix length layers are searched first rather than the medium length. However, this approach has a negative impact on the worst case, because some searches would take more steps to find the BMP. Using a mutating binary search trie could improve the worst case, but it also increases the storage requirement and makes prefix updates more difficult. Finally a better scheme, rope and index array are proposed to refine the algorithm, which reduces the average number of hashes to smaller than two. With optimization, this algorithm reduces the average search time and memory requirements.

1.4 Outline of Rest of Thesis

Chapter 2 reviews some basic knowledge about cache memory and hash tables, which are closely related to our proposed algorithm. The problems in the existing IP lookup schemes are addressed in Chapter 3. In Chapter 4, a thorough description of the proposed solution is provided. The data structures of three main components in the algorithm are illustrated with diagrams and examples. The description of the establishment of the hash table and the index table, search and updates are also given in this chapter. The algorithm is also analyzed by comparing it with some other algorithms. Chapter 5 provides implementation and performance analysis including the prefix extension, memory accesses, memory usage, and updates. Chapter 6 concludes the thesis work, and makes suggestions for future work.

2. A review of Cache and Hash Table

In computer architectures, the huge bandwidth difference between center processor unit (CPU) and main memory is reduced by using cache memory, which can significantly improves the system performance. With a fast access speed, cache memory is now becoming more attractive for IP address lookup. A hash table is an efficient data structure for doing fast searches. By selecting a suitable hash function, data can be stored in the hash table sparsely. Since cache and hash tables will be used as key elements in our proposed algorithm, the features of these two elements will be demonstrated in the following sections.

2.1 Cache Memory

In this section, we will answer questions such as: What is cache, why is cache used, and how does cache work? In many hardware based or hardware-assistant IP lookup algorithms, using cache memory is an important means to improve search performance. Cache memory is also a critical component in modern personal computers. Without it, the performance of Personal computers (PCs) will degrade significantly.

Cache memory is a kind of static random access memory (SRAM), which is faster than regular RAM. Cache memory is categorized into primary cache (L1) and secondary (L2) cache in terms of the cache's closeness and accessibility to the microprocessor [39]. An L1 cache is on the same chip as the microprocessor. The size of an L1 cache varies with different kinds of microprocessors. A sixth generation AthlonXP (2.3GHz) microprocessor manufactured by Advanced Micro Device (AMD) has 128K bytes of L1 cache, but the same generation Pentium 4 (2.8GHz) microprocessor provided by Intel

only have 8K bytes of L1. L2 cache is usually a separate SRAM chip, but in recent products, L2 cache is often integrated into the same chip as microprocessors to improve their performance. The size of L2 can reach 512K bytes in Intel chips. Since cache memory is more expensive than DRAM, the processor designers must make a trade-off between its optimal speed capability and its cost.

Today's processors in computers can run up to 3GHz internally. With a 64-bit (8 bytes) processor data bus width, a processor can require data to arrive at the rate of 16GB/s ($2.0\text{G} \times 8 \text{ bytes}$) to 22.4GB/s ($2.8\text{G} \times 8 \text{ bytes}$.) However, even with the fast Pentium 4's 533MHz front side bus (FSB), the data rate can only be increased to 4.2GB/s [40], which is far slower than the requirement of the Pentium 4 processor. However, if data is supplied to processors from a cache rather than from main memory, the data rate gap between processors and the data bus can be reduced significantly.

Figure 2.1: Relationship between CPU and cache.

The relationship between cache and processor is shown in Figure 2.1. L1 cache is integrated on the same chip as the processor, so the data rate can match the CPU speed if data comes from the L1 cache. An L2 cache usually is off the chip and its access speed is

one half or one third of that of the processor. But the L2 cache, which is integrated in the same chip as the processor as in recent products, can reach the same speed as L1 cache.

There are four fundamental parts of a cache memory: Cache data memory, cache directory, bus buffer and cache controller, which are shown in Figure 2.2. The cache data memory is a small, fast memory, which is used to store instructions and data. The cache directory is a list of the main memory addresses of recently accessed data. These data are stored in the corresponding locations of the cache data memory. The bus buffers can be controlled to keep a copy of a main memory location. Cache controller is a logic component implementing the algorithm and moving the data into and out of the cache data memory and the cache directory [41].

In the following, we will use Figure 2.2 to illustrate how processors use the cache to improve the data rate. As the CPU needs data, it first goes to the cache directory with the main memory address of the data, and puts the address into the address buffer at the same time. If the address is found in the cache directory, the data in the cache data memory is retrieved at rate of a cycle clock. Otherwise the cache controller indicates that the address in the buffer is sent to the main memory, and the CPU will do a more time-consuming reading of data from the main memory. At the time of reading data from main memory, the data is also written into the cache. That means the cache always keeps the most recently used data. Actually the data in the cache has a higher probability of being used than other data in main memory. If the processor keeps a high cache hit ratio, the system could get high performance in reading data.

Figure 2.2: The cache memory consists of a directory, a cache data memory, buffers (to isolate the cache subsystem from the system bus) and the cache controller [41].

2.2 Hash Tables

A hash table consists of two major components: An array and a hash function. The array A of a hash table has size N , which defines the capacity of the array. An element with a key can be inserted into the array. Any cell in the array is associated with one or more keys. To store an element with an arbitrary key into the array, it is necessary to find a way to map a given type of key into an integer among the range $[0, N-1]$. An integer assigned to a key is called the hash code for that key. The hash code should be mapped down to the range $[0, N-1]$ in a uniform manner, which can be done by using a function. That function is called the hash function [42]. Therefore if we search an element in the array with a key, by using the hash function to get the index associated with the key in the array, we only do one search operation to find the element.

Since the key in the hash table can be of any type, different keys may be assigned to the same integer, and different keys may have the same hash code. In this case, two or

more different elements may be mapped to the same cell of the array A . This event is called a collision. A simple and efficient way for dealing with a collision is to store a sequence of elements with the same hash code in one cell of the array A . This is known as chaining resolution. We will use a hash table with a simple hash function to give an illustration of chaining.

Example: $N=13$, keys $\{10,12,18,25,28,36,38,41,54,90\}$ and hash function: $h(k)=k \bmod N$. Here $h(k)$ is the hash code. Key 28, 41 and 54 have the same hash value 2, so these keys, with their elements, will be put into the same cell $A[2]$. The same thing happens in keys 36, 10 and 12, 25, 38, 90, respectively. Figure 2.3 shows the chaining collision for the hash table.

Figure 2.3: (a) A hash table of size 13, storing 10 integer keys, with collisions resolved by the chaining method. The hash function is $h(k)=k \bmod 13$, [42]. (b) The number of collisions is reduced when N is changed to 11.

The more collisions the hash table has, the worse the search performance becomes.

From Figure 2.3, if we want to find the element with key 90, we have to do four search

operations rather than one. If we change the array size to 11, in the worst case, the number of collisions is reduced to two. To find an element in Figure 2.3, we need at most two searches rather than four. Therefore, selecting an appropriate table size, N , makes the hash table more efficient. According to C.E Radke's study in 1970 [43], a good choice for a table size is a prime number of the form $4k+3$. Sometimes, however, choosing N to be a prime number is not enough, because if some key values have a repeated pattern $iN+j$ for several different i 's, many collisions will still happen. In this case, we should try other common kinds of hash functions. One of which is:

$$h(k) = (ak+b) \bmod N$$

where N is a prime number, $a \neq 0$ and b is a non-negative integer. The purpose of this hash function is to eliminate the repeated patterns in the set of hash codes. With this hash function, any two different keys have at most a $1/N$ probability of colliding. There are many other proposed hash functions that might reduce collisions, and researchers are still investigating new hash functions.

To resolve collisions, there are alternate schemes such as linear probing, quadratic probing and double hashing. The advantage of these schemes is the save of space, since no auxiliary structures are used, like chaining, but it is more complicated to handle collisions with these schemes.

3. Motivation

Many new algorithms have been developed to speed up the IP lookup operation in Internet routers. With these algorithms, a forwarding engine in routers can easily forward packets at gigabit rates. However, when these lookup algorithms employ novel data structures or technologies, new problems arise which prevent these algorithms from achieving expected performance.

3.1 Problems in IP Lookup Algorithms

Some researchers have developed fast IP address lookup algorithms at the cost of other performance measures such as updates, worst case speed, flexibility, and memory usage. In this section, these issues will be addressed with some existing IP lookup algorithms.

3.1.1 Difficult Updates

Many existing lookup schemes only concentrate on the fast IP address lookups at the expense of table updates. In gigabit backbone routers which handle frequent routing updates, slow updates will result in serious packet delay. Difficulties of table updates arise from complicated or compact data structures. In the Lulea algorithm, in which four complicated data structures are employed, updating a prefix entry will change all the entries below this target prefix entry. In the worst case, the entire forwarding table will be rebuilt.

A high degree of prefix extensions is another reason for slow updates. In the DIR-24-8 algorithm, a prefix will potentially be extended to 2^{16} extended prefixes. So updating this kind of prefix involves 2^{16} prefix entries, or *TBLlong* segments, being

changed. Although route updates do not happen as frequently as address lookups, slow update rate might have a fairly negative impact on the address lookups.

3.1.2 Worst Case Lookup Speed

A main concern in some lookup schemes is to ensure fast lookup speed on the average. However, the address lookup becomes time consuming operation in the worst case. In the multi-way and multi-column search algorithm, the sizes of the search tables are unbalanced. For the routing database Mae-East, the size of a search table ranges from one to 384. Even six-way search technology is used to reduce the memory accesses, there are still five main memory accesses and 44 cache memory accesses in the worst case.

The reason for allowing the low performance in the worst case is that the worst case rarely occurs, so it is considered not to affect the average lookup rate. However, the worst case might happen frequently in a short time when a number of sequential packets destined to the same destination network are coming. A high frequency of worst case instances will queue more packets in the router, so the low lookup rate in the worst case cannot be ignored.

3.1.3 Low Flexibility

Flexibility is an important performance metric in efficient lookup schemes, but it is usually ignored by many researchers. With high flexibility algorithms, the high-speed performance will not change or will only slightly change on different routing databases or different Internet routers.

The Lulea algorithm is considered as a high performance lookup scheme in relation to lookup speed and memory usage. Although the memory usage in the Lulea scheme is

only between 150 to 160K bytes for a database with 40,000 entries, it requires cache memory to hold the lookup table. Actually, not all routers can provide enough cache memory space for this table. Furthermore, since routing tables do not stop growing, the size of the lookup table in the Lulea algorithm may increase with size of the large routing tables. Therefore, the forwarding table can not easily be fit into the cache memory.

Internet routers in different networks may have different hardware and contain different routing tables. High performance lookup schemes should keep high flexibility to satisfy all kinds of requirements.

3.1.4 High Memory Usage

Memory usage is not a critical problem in IP lookup because memory size in Internet routers is growing at the same time. But it is possible for a router to be unable to hold a lookup table. In the Dir-24-8 algorithm, all prefixes with lengths less than 24 are extended to prefixes with length 24. In spite of the routing table size, any lookup table in this algorithm has at least 2^{24} entries that consume about 33M bytes of memory usage, which increases the cost of implementing this algorithm. In CIDR, only network addresses (or prefixes) are stored in the routing table to slow routing table growth. To achieve high speed IP address lookup by huge prefix extension is not practical.

3.2 Motivation, Statement and Objective

To overcome the problems in the existing schemes, this thesis aims at developing a novel IP address lookup algorithm for IPv4 address. Like all existing schemes, this algorithm must perform high-speed address lookups. Simultaneously, this algorithm will keep reasonable memory requirements and relatively fast updates. The search rate in the

worst case will be bounded to a certain value to avoid negative effects on search performance. This scheme does not require specific hardware such as Application-Specific Integrated Circuit (ASIC), or field-programmable gate array (FPGA), except for on-chip memory, which is available in most Internet routers. So this algorithm should possess high flexibility.

3.3 Scope of Thesis

Our work concentrates on developing a high performance algorithm. A part of the ideas of this scheme come from three existing algorithms including: The multi-way and multi-column search, the binary search on prefix length, and the Lulea algorithm, which are reviewed in Section 1.3 and will be evaluated in Section 4.6.

A new algorithm is proposed to perform IP lookup using an index table that is small enough to be located into on-chip memory, such as cache memory, on a general-purpose processor (GPP). Three data structures (index table, hash table and small search group) are designed in this algorithm. Using an index table narrows the search region from the whole database into a small search group, while the hash table allocated into main memory is employed to reduce the number of main memory accesses.

This algorithm will be analyzed according to performance metrics including the number of memory accesses, memory usage, updates and flexibility. Comparisons with some existing algorithms will be provided.

This algorithm will be implemented on general-purpose processors. The performance of this algorithm will be evaluated based on the results of implementation on the GPP.

4. Proposed Algorithm

A new algorithm that addresses the problems described in Section 3.1 is presented in this chapter. Section 4.1 gives the basic idea of this scheme based on four concepts: Index table, hash table, small search group and prefetching. The relevant data structures of the index table, hash table and small search group are detailed in Section 4.2. In order to explicitly describe our solution, the pseudo of creating look up table, search operation and the lookup table updates are provided in Section 4.3. Finally, Section 4.4 analyzes the algorithm based on metrics including memory accesses for lookup, memory usage for the lookup table, prefix updates and the algorithm flexibility.

4.1 Basic Idea

To speed up lookup rates, many efficient search approaches are employed in IP address lookup algorithms. Multiway and multicolumn algorithms deploy an initial table to narrow the search range and utilize prefetching technology to decrease the number of main memory accesses. In the Lulea algorithm, a compacted data structure that is small enough to fit into on-chip memory is designed to speed up the lookup rate. Hash tables are used in binary searches on prefix length algorithms to cut down the number of lookup times. New problems, however, occur in these algorithms, which are discussed in Section 3.1. We propose a new algorithm based on the ideas from these three algorithms, while also eliminating many of the problems found in the aforementioned three algorithms.

The proposed algorithm consists of four basic components: Small search groups, hash tables, index tables, and prefetching.

Small search group: One of the ideas in our algorithm is to efficiently narrow the search range down from the whole routing table to a small range. The small range, called “small search group” in the proposed algorithm, contains the best matching prefix if one exists. Thus the lookup operation is only carried out in the small search group in the next step, which significantly reduces the number of lookup times. The number of prefixes in a small search group is bounded to control the worst case. To realize this idea, we need to divide a routing table into a number of small search groups, which will be depicted with an example of the structure of a book.

In a book, if the size of a section is too large, this section will be broken into several small subsections to make the book easy to read. If we categorize all prefixes in a forwarding table into many groups (corresponding to sections in a book) according to the first 8 bits of the prefixes, it will be found that some groups are extremely large and others very small ones. Similar to organizing a book, we can break larger groups into smaller groups (corresponding to subsections in a book) according to their first 16 or 24 bits. In the following, we illustrate how to break larger groups into smaller groups.

Example: Suppose that the number of prefixes in a group is limited to three. Given a forwarding table in Figure 4.1, the table is first divided into two groups according to the first 8 bits of prefixes, referred to as group 12 and group 24 (here 12 and 24 are called group names). Since the size of group 24 is greater than three, it will be further split into two small groups (group 24.12 and group 24.13) according to the first 16 bits of prefixes in this group. So there are three groups so far: Group 12, group 24.12 and group 24.13. Group 24.12 has five prefixes, which is still greater than three. Based on the first 24 bits of prefixes in this group, group 24.12 is broken into four small groups: Group 24.12.0,

group 24.12.1, group 24.12.5 and group 24.12.6. Finally the forwarding table is categorized into 6 groups, each of which has less than 3 prefix entries.

Figure 4.1: The process of breaking larger groups into small groups.

Hash table: Hash tables are often used in exact matching, in which the search key has same length. To reduce a big search range into a small search group, we employ a hash table to hold all small search groups. The search keys (or hash keys) have different lengths or the group names mentioned above. With the hash key, the small search group can be found in one search step.

Index table: To search a small search group in the hash table, we first need the hash key for a given destination IP address. The hash key information of all small search

groups are placed into a table named “index table.” With a similar purpose to index pages in a book, the index table provides a clue for a quick search. Before searching a small search group, we first check the index table with the address. The result of checking the index table is the hash key.

Prefetching: On-chip memory has faster memory access than main memory, so putting the data into this kind of memory is an efficient way to speed up a search. However, a routing database is normally too large to fit into memory. The Lelua algorithm places a whole table into on-chip memory by compacting the database, but it causes other problems mentioned in Section 3.2. In the proposed algorithm, the index table will be prefetched into the fast on-chip memory, such as the cache memory, to do fast index table checking. The small search group is also small enough so that it can be loaded into the cache memory. But the index table always resides in the cache memory, while the small search group is only temporarily fetched into the cache memory.

Figure 4.2 shows the basic process of an IP lookup using the index table. Given an incoming IP address, we first check the index table to obtain the name of the group (or hash key) that the address belongs to. With the hash key and the hash function, we are able to find the small search group in the hash table in one step. To find the BMP for the address, we only need to search the small search group. Consequently, we reduce the search range size from searching the whole database to searching the small search group. To further improve the lookup speed, the small search group is prefetched into the fast cache memory before we search for the BMP from the small search group. This will be discussed in following sections.

Figure 4.2: Basic process of IP lookup using the index table.

4.2 Data Structures of the Proposed Algorithm

The data structures of our proposed algorithm consist of three parts: The index table, the hash table and the small search group. The index table stores the information about the hash keys. The hash table, which contains all small search groups, is organized according to the group name (hash key). The result of a hash lookup is a small search group, in which the BMP can be found, if it exists.

The index table would be designed to be small enough to fully fit into the primary cache memory. To be fit into the cache lines, a small search group is compacted to at most 128 bytes. In Section 4.2.1, we will describe the structure of the index table, while the structure of the hash table and the small search table will be discussed in Sections 4.2.2 and Section 4.2.3 respectively.

4.2.1 Data Structure of Index Table

The index table is an array of 256 elements, each of which has two fields, called the head and the sub-index table. The data structure of the index table is shown in Figure 4.3.

The head is one of the four possible values: ‘0’, ‘1’, ‘2’ or ‘3’, each of which has different meanings, explained below.

‘0’: There is no matching prefix in the routing table for the input address. Lookup is over and the next hop will be the default next hop.

‘1’: The first 8 bits of the input IP address are used as the hash key to perform a hash table lookup.

‘2’: The first 16 bits of the input IP address are used as the hash key to perform a hash table lookup.

‘3’: The first 16 or 24 bits are extracted from the address as the hash key. In this case, the second 8 bits of the address are used to check the sub-index table. If this 8-bit sequence is found in the sub-index table, the first 24 bits of the address are used as the hash key. Otherwise, the first 16 bits of the address are extracted as the hash key. An example for searching the index table is given in Figure 4.4.

Figure 4.3: The structure of an element in the index table.

Example: Given an input IP address 00001100 0011101 01111100 01111110 that can also be represented as 12.45.124.126 for simplicity, the first 8 bits representing “12” are used to index into the index table. After checking the element corresponding to index 12, if the head ‘1’, then the first 8 bits, 12, are used as the hash key to search the hash table. Consider another address 64.34.45.32, if the head is ‘2’ according to the search

result in the index table, then the high 16 bits (64.34) of the address are used as the hash key. If the address is 192.138.129.45, and the head is '3', it is necessary to check the sub-index table of the element with the next 8 bits 138. The 8-bit sequence is found in the sub-index table, so the first 24 bits 192.138.129 of the address are used as the hash key.

Figure 4.4: An example of an index table structure.

The sub-index table stores the information regardless of whether or not the 24 bits of the input address are used as the hash key. Since an 8-bit sequence corresponds to a total of 256 ($2^8=256$) possibilities, to save memory space we use a bit vector as the sub-index table. Each bit position in this vector represents a possible 8-bit value from 00000000 to 11111111. The structure of the sub-index table is shown in Figure 4.5. The sub-index table consists of 8 32-bit words. To check an 8-bit string, the first 3 bits of the string bits are an index to the proper word, and the remaining 5 bits are the bit position in this word. If an 8-bit value is in the sub-index table, the value in the corresponding bit

position is set to one. We use an example to illustrate how to check a sub-index table for an 8-bit value.

Figure 4.5: The structure of a sub-index table.

Example: Given an IP address 11000000 10001101 11101010 00101101 (or 192.141.234.45), the high 8 bits 11000000 are the index to an element in the index table. The head of this element is 3, so we extract the second 8 bits 10001101 of this address. To check the sub-index table, the first 3 bits, 100, are extracted as the index that is word 4. The remaining 5 bits, 01101, are the index of bit position 13 in word 4. Since one is in this bit position, which means the 8-bit string “10001101” is found in the sub-index table, the first 24 bits of this address 11000000 10001101 11101010 will be used as the hash key. If the second 8 bits are 0001100, then the bit position is 12 in word 4. Since zero is in this bit position, which means that 1001100 is not found in the sub-index table, the first 16 bits 11000000 10001100 are used as the hash key.

4.2.2 Data Structure of Hash Table

The hash table employed in this algorithm is an array of n elements, where n is the size of this array or called the “table size” in the algorithm. The table size varies with the number of hash table elements (or prefix entries in the algorithm). The structure of an

element in the hash table is shown in Figure 4.6. Each hash table element has four fields: Identifier, size, pointer and next pointer, which are explained as follows.

Figure 4.6: The structure of an hash table element in the hash table.

- **Identifier:** Differentiates hash keys which are mapped into the same hash index.. This field is 24 bits. If a hash key is less than 24 bits, then zeros are added to its left.
- **Size:** The number of items in the small search group whose address is stored in the pointer field.
- **Pointer:** The addresses of a small search group in which the BMP can be found, if it exists.
- **Next pointer:** The address of the next link in a linked list structure deployed to solve the hash collisions discussed in Section 4.2.3.

Figure 4.7: Converting an original forwarding table into a hash table with hash key length 2.

Figure 4.8: Converting an original forwarding table into a hash table with hash key length 2 and 3.

In the following, the hash table is illustrated by an example. Figure 4.7 shows the process of converting an original routing table into a hash table. If we use the first two bits of a prefix as hash key, the hash table only has two hash table elements. That means that the routing table is divided into two small search groups. Small search group (a) has three prefixes, in which all prefixes start with bits 01. Small search group (b) has five prefixes and all prefixes start with the two bits 11. If the size of a small search group is limited to four, entire small search (b) group must be further divided. In this case, we can choose the first three bits of a prefix in small search group (b) as the key, and put the prefixes with the same key into the same small search group. This results in a hash table with three hash table elements, which are shown in Figure 4.8. The small search group (b) in Figure 4.7 is divided into two small search groups in Figure 4.8 by extending the key length to three.

The critical problem in the hash table structure is hash collisions. There are two commonly used solutions to collision: chaining and probing. Chaining is a simple solution to implement, but will waste some memory space. Probing solution can make a more efficient use of the memory space, but it is more complicated to implement. In addition, if the array is full in the probing solution, the array must be resized for new prefix entries. In this case, all elements in the hash table must be reorganized. Therefore updates of a forwarding table will become more difficult. Since the search speed is more critical than memory space in our proposed algorithm, the chaining resolution is the best choice for us. Figure 4.9 shows how to handle the hash collisions by the chaining solution, in which only the identifier and next pointer fields are displayed for simplicity.

Figure 4.9: Chaining solution for hash collisions.

If more than one hash key maps into the same hash index, the colliding elements are put into one linked list. When one hash key is mapped into this index, the identifier of the first link is compared with the hash key. If they are identical, the search is over. Otherwise, the second link pointed to by the next pointer in first link is checked, and so on. If all links are checked without matching, the default next hop will be returned.

4.2.3 Data Structure of Small Search Groups

A small search group is an array whose size is controlled from 1 to 16 by selecting the length of its hash key. Each small search group element in the array consists of two 32-bit words. The first word represents part of a prefix (24 bits), an extension flag (1 bit) and the prefix length (7 bits). The second word is the pointer to the next hop. The structure of a small search group is shown in Figure 4.10 (a). To save memory space, only a part of a prefix is put into the small search group. Here we explain how to divide a prefix.

With a prefix length l , if the first m bits of this prefix are its hash key, then the remaining $l-m$ bits (the suffix) are put into the small search group. If the suffix is less than

24 bits, it is extended to 24 bits by trailing zeros. If l equals m , the prefix is its hash key and the suffix is empty. In this case, 24 zeros are put in this field. If l is less than m , the prefix has to be extended to $2^{(m-l)}$ prefixes with length m . If a prefix is an extended prefix, we use a 1-bit field called “extension flag” that is set to one to represent this. The extension flag will be used in the update discussed in Section 4.5. The prefix length field has 7 bits, which tells how many bits of an IP address are used to compare the suffix.

Figure 4.10: The structure of the small search group.

Figure 4.11 is an example of how to convert part of a prefix and its length into a 32-bit long word. Given the prefix 12.32.64/20, if the first 16 bits of the prefix are used as the key, then the 4-bit suffix is 0100. By extending the suffix and adding the extension flag and the length, the binary format of the long word 01000000 00000000 00000000 0010100 is formed. In this long word, the first three bytes represent the suffix, and in the last byte, the leftmost bit 0 is extension flag and the rightmost 7 bits 010100 stand for the length 20. Since the next hop is a 32-bit address, it is easy to convert the next hop into a

long word. The information in the first word is used to check if the input address matches the prefix. If so, the second word is the next hop. If not, the algorithm continues to check the next small search group element.

Figure 4.11: An example of converting a prefix into a word.

In a small search group, all prefixes are stored in order of decreasing length for linear search. That means that the search starts the prefix with the longest prefix length. When a match occurs, the search is terminated without going through all prefixes to find the BMP.

4.3 Pseudo codes for the Proposed Algorithm

The proposed algorithm consists of two kinds of operations: Address lookups and prefix updates. Before performing address lookups and prefix updates, the index table and the hash table, two main data structures of the forwarding table (called lookup table

in this algorithm), must be established.. These procedures will be described in this section using pseudo codes.

4.3.1 Notation in Pseudo Codes

To describe the processes of the creation of a lookup table, the IP address lookup and the update of lookup table, we use pseudo code to explain the processes in detail. In the pseudo code, some variables, values or functions will be represented as symbols. These symbols used in this algorithm are not standard, which are explained as follows:

- N_{m-n} : The number of prefixes which have the same bits from the m th bit to the n th bit.

E.g.: N_{1-8} is the number of prefixes with the same first 8 bits.

- len : The length of prefix pre , which ranges from 8 to 32.
- H : The value of a head in the index table, which may be '0', '1', '2', or '3'.
- V_{m-n} : The value of bits from the m th bit to the n th bit.

E.g.: For prefix 106.84/16, V_{1-8} is 106, V_{9-16} is 84, V_{1-16} is 27220.

- S_{key} : A set of prefixes with the same hash key.

E.g.: $S_{01101000}$ denotes all prefixes starting with 01101000

- key_n : A hash key consisting the first n bit.

E.g.: For prefix 01101010 01010100 (binary), key_8 is 01101010 and key_{12} is 01101010 0101

- T_{forw} : Forwarding table (original prefix entities).
- T_{index} : The index table.
- T_{hash} : The hash Table.
- $Item_{key}$: A small search group with key key .

- T_{temp1} : A temporary array of 256 integers storing the data collection from the first 8 bits of all prefixes.
- T_{temp2} : A temporary array of 65,536 integers storing the data collection from the first 16 bits of all prefixes.

4.3.2 Lookup Table Creation

The creation of the lookup table creation consists of creating index table and the hash table. Before building the hash table, the index table must be established first.

4.3.2.1 Building Index Table

There are five steps to building the index table: Step 1 initialization, step 2 data collection, step 3 building, step 4 extension, and step 5 refining. We detail these steps as follows.

Step 1: In this step, the pseudo code for creating and initializing an index table is presented, in Figure 4.12. Each index table element in the table is called `index_table_entry` and consists of two fields: Head and sub-index table. An empty sub-index table is an array of eight 32-bit words that are initialized to zero.

Input: none
Output: index table, T_{index}

```

Initialize( )
    create an empty array  $T_{\text{index}}$  of 256 elements;
    For  $i \leftarrow 0$  to 255
         $T_{\text{index}}[i] \leftarrow \text{index\_table\_entry}$ ;
         $T_{\text{index}}[i].\text{head} \leftarrow '0'$ ;
        create an empty sub-index table
         $T_{\text{index}}[i].\text{sub\_index\_table} \leftarrow \text{sub-index table}$ ;
    return  $T_{\text{index}}$ ;

```

Figure 4.12: Initialization of the index table.

Step 2: The main task of data collection is to obtain statistical data according to the first n bits of all prefixes, where n is 8 or 16. In this step, a temporary array of 256 integers T_{templ} is created to store the data N_{l-8} . First of all, initialize each integer in T_{templ} to zero, and then go through all the prefixes in the original forwarding table, T_{forw} . For each prefix, extract its first 8 bits to get V_{l-8} . With V_{l-8} as an index, we can get an integer $T_{\text{templ}}[V_{l-8}]$ in T_{templ} , and increase this integer by one at index $T_{\text{templ}}[V_{l-8}]$. After going through all prefixes, we collect all N_{l-8} . This process is called data collection and is shown in Figure 4.13.

Input: Forwarding table, T_{forw} ,
Output: Temporary table 1, T_{templ}

```

collect( $T_{forw}$ )
    create an empty array  $T_{templ}$  of 256 integers
    for  $i \leftarrow 0$  to 255 do
         $T_{templ}[i] \leftarrow 0$ ;
        for  $j \leftarrow 1$  to # of all prefix entries in  $T_{forw}$  do
             $V_{1-8} \leftarrow$  high 8 bits from prefix in  $T_{forw}[j]$ ;
             $T_{templ}[V_{1-8}] \leftarrow T_{templ}[V_{1-8}] + 1$ ;

    return  $T_{templ}$ ;

```

Figure 4.13: Data collections in building the index table.

Step 3: With T_{templ} , we can fill the index table. Go through T_{templ} and check each N_{1-8} . If N_{1-8} is greater than zero and less than or equals to 16, the corresponding head in the index table, T_{index} , is changed to '1'. Since T_{templ} and T_{index} are the same size of 256, checking T_{templ} and modifying T_{index} can be done simultaneously. For example, if $T_{templ}[12]$ is 15 which is less than 17, the head in $T_{index}[12]$ is changed to '1'. If $T_{templ}[12]$ is 20, which is great than 16, the head is changed to '2'. This process is shown in Figure 4.14.

Input: empty index table, T_{index} and temporary Table1 T_{templ} .

Output: index table, T_{index}

Build_index_table (T_{index} , T_{templ})

For $i \leftarrow 0$ to 255 do

if ($T_{templ}[i] == 0$) then do nothing;

else if ($T_{templ}[i] \leq 16$) then $T_{index}[i].head \leftarrow '1'$;

else if ($T_{templ}[i] > 16$) then $T_{index}[i].head \leftarrow '2'$;

return T_{index} ;

Figure 4.14: Building the rough index table.

Step 4: A prefix needs to be extended if this prefix length is greater than 8 and less than 16, and its head in the index table T_{index} is '2'. This process is described as follows. Go through all prefixes in T_{forw} , then check T_{templ} with its V_{l-8} . If N_{l-8} is greater than 16 and the prefix length, len , is greater than 8 and less than 16, extend this prefix to multiple prefixes with length 16. If the length is 8, this prefix will not be extended to length 16. Its next hop will be used as the default next hop for destination addresses whose first 8 bits are the same as those of this prefix. All the extended prefixes are put into the original forwarding table, T_{forw} . After extending the prefixes, we need to collect all the N_{l-16} values. Using a similar approach used in the data collection, each N_{l-16} is stored into another temporary table, T_{temp2} . Figure 4.15 shows this method.

Input: Forwarding table, T_{forw} and Temporary Table 1, T_{temp1}

Output: Temporary Table 2, T_{temp2}

Extend (T_{forw}, T_{temp1})

```

for  $i \leftarrow 1$  to # of all prefix entries in  $T_{forw}$  do
     $V_{1-8} \leftarrow$  high 8 bits of  $T_{forw}[i]$ ;
    if (  $T_{temp1}[V_{1-8}] > 16$  ) then
         $len \leftarrow$  length of  $T_{forw}[i]$ ;
        if (  $len < 16$  and  $len > 8$  ) then
            extend prefix  $T_{forw}[i]$  to  $2^{(16-len)}$  prefixes with  $len=16$ ;

```

Create an empty array T_{temp2} , and initialize all values in T_{temp2} as zero

```

for  $j \leftarrow 1$  to # of all prefix entries in  $T_{forw}$  do
     $V_{1-8} \leftarrow$  high 8 bits of  $T_{forw}[j]$ ;
    if (  $T_{temp1}[V_{1-8}] > 16$  ) then
         $V_{1-16} \leftarrow$  high 16 bits of  $T_{forw}[j]$ ;
         $T_{temp2}[V_{1-16}] \leftarrow T_{temp2}[V_{1-16}] + 1$ ;

```

return T_{temp2} ;

Figure 4.15: Extension of prefixes.

Step 5: This is the last step to complete the index table, in which more information will be added. Go through T_{temp2} , and check each value N_{1-16} . If N_{1-16} is greater than 16, the corresponding head in the index table, whose value is '2', will be changed to '3'. Also the second 8 bits of those prefixes are put into the corresponding sub-index table in the index table. In the following we use an example to explain the rule.

Suppose that there is a total of 18 prefixes which share the same first 16 bits, say 00011000 01011111 (or 24.95), the 01011111 is stored in the sub-index table of the element indexed by 24. So the first 24 bits of the prefixes starting with 24.95 will be used

as the hash key. But the prefixes with lengths less than 24 and greater than 16 will be extended to prefixes with length 24. The extension method is the same as the one in step Extension. Figure 4.16 is the process of refining the index table.

```

Input: Index Table,  $T_{index}$  and Temporary Table 2,  $T_{temp2}$ 
Output: Index Table  $T_{index}$ 

Refine_Index_table(  $T_{index}, T_{temp2}$  )

  for  $i \leftarrow 1$  to # of elements in  $T_{temp2}$  do
    if ( $N_{1-16} > 16$ ) then
       $V_{1-8} \leftarrow$  first 8 bits from  $i$ 
       $V_{9-16} \leftarrow$  second 8 bits from  $i$ 
       $T_{index}[V_{1-8}].head \leftarrow '3';$ 
      check  $T_{index}[V_{1-8}].sub\_index\_table$  with  $V_{9-16}$ 
      if ( not found ) then
        add  $V_{9-16}$  into  $T_{index}[V_{1-8}].sub\_index\_table$ 
  return  $T_{index}$ ;

```

Figure 4.16: Refining the index table.

4.3.2.2 Building the Hash Table

The hash table is established based on the index table. In the proposed algorithm, two steps are taken to build the hash table, including step 1 initialization and step 2 building.

Step 1: In this step, it is important to choose the ideal size of hash table. This size may impact the performance of hash lookups such as average hash collisions and maximum hash collisions. Each hash table element (called hash_table_entry in the pseudo code) has been allocated the same memory space. The field size is initialized to

zero and all other fields are initialized as NULL. Figure 4.17 shows the process of this step.

```

Input:  none
Output: hash table,  $T_{hash}$ 

Initialize()
    create an empty array  $T_{hash}$  of table_size ;
    For  $i \leftarrow 0$  to table_size  do
        create an empty hash_table_entry;
         $T_{hash}[i] \leftarrow$  hash_table_entry;
         $T_{hash}[i].key \leftarrow$  NULL;
         $T_{hash}[i].number \leftarrow 0$ ;
         $T_{hash}[i].item \leftarrow$  NULL;
         $T_{hash}[i].next \leftarrow$  NULL;

    return  $T_{hash}$ ;

```

Figure 4.17: Initialization of the hash table.

Step 2: This step will fill the initialized hash table with the prefixes in the forwarding table, T_{forw} , that include the original prefixes and the extended prefixes. The building process is described in Figure 4.18. Go through T_{forw} and for each prefix check the head in the index table with V_{l-8} , the first 8 bits of this prefix. The hash key can be extracted from the index table entry in terms of the head. When a head is '3', the sub-index table must be checked against the second 8 bits of this prefix to determine the hash key. The prefix is put into a small search group in the hash table according to the hash key. If the small search group with that hash key does not exist, a new small search group

is created to hold the prefix. Otherwise, a memory location is allocated for the small search group, and the prefix is placed into it.

Input: Hash table, T_{hash} , Index Table, T_{index} and Forwarding Table, T_{forw}

Output: Hashtable, T_{hash}

Setup_hashTable($T_{hash}, T_{index}, T_{forw}$)

```

for  $i \leftarrow 1$  to # of elements in  $T_{forw}$ 
     $V_{1-8} \leftarrow$  first 8 bits of  $T_{forw}[i]$ ;
     $H_i \leftarrow T_{index}[V_{1-8}].head$ ;
    if ( $H_i == '1'$ ) then
        put  $T_{forw}[i]$  into  $T_{hash}$  with  $key_8$ ;
    else if ( $H_i == '2'$ ) then
        put  $T_{forw}[i]$  into  $T_{hash}$  with  $key_{16}$ ;
    else if ( $H_i == '3'$ ) then
         $V_{9-16} \leftarrow$  the next 8 bits of  $T_{forw}[i]$ ;
        check  $V_{9-16}$  in  $T_{index}[V_{1-8}].sub\_index\_table$ ;
        if (found) then
            put  $T_{forw}[i]$  into  $T_{hash}$  with  $key_{24}$ ;
        else
            put  $T_{forw}[i]$  into  $T_{hash}$  with  $key_{16}$ ;
return  $T_{hash}$ 

```

Figure 4.18: Building the hash table

4.3.3 IP Address Search

Before searching, the whole index table will be prefected into the cache memory. There are three steps to perform the route lookup, including checking the index table, searching the hash table and searching the small search group. It is not necessary for an IP lookup to perform all three steps since the next hop can be determined in step 1, step 2 or step 3. The search process in the proposed algorithm goes as follows.

Given an input IP address, the first 8 bits of the address, V_{1-8} , are extracted to index in to the index table to get the head. If the head is '0', the search is terminated and the default next hop is returned. If the head is '1' or '2', the first 8 bits or 16 bits of the address, respectively, are extracted as its hash key to do hash lookup. If the head is '3', the next 8 bits of the address, V_{9-16} , are taken to check the sub-index table. If V_{9-16} is found in the sub-index table, the head is still kept as '3', while the first 24 bits of the address are extracted as the hash key. Otherwise, the head will be '2'. In this case, the hash key is the first 16 bits of the address. Now we give an illustration of the search with an example and Figure 4.19.

Suppose that we have an input IP address P, 24.30.108.172, in dotted decimal form. First, the first 8 bits (here it is 24) are extracted to check the index table. If the head is given as '3', the next 8 bits (30) are taken to check the sub-index table. Since there is no match in the sub-index table, the head is considered as '2', and the first 16 bits of P will be the hash key (00011000 00011110(24.30)). With the hash key, only one hash lookup is needed to find the result that is a pointer to a small search group. Before further searching, the whole small search group is prefetched into the primary cache, then the best matching prefix 00011000 00011110 0* is found using a linear search. The corresponding next hop is 198.32.136.124. Figure 4.20 shows the pseudo code of the search process.

Figure 4.19: An example of IP address lookup using index table.

Input: IP address, P_{addr} , Hash table, T_{hash} , and Index table, T_{index}

Output: next hop

```

Search( $T_{hash}$ ,  $T_{index}$ ,  $P_{addr}$ )
     $V_{1-8} \leftarrow$  the first 8 bits of  $P_{addr}$ ;
     $H \leftarrow T_{index}[V_{1-8}].head$ ;
    if ( $H = '0'$ ) then
        return default next hop;
    else if ( $H = '1'$ ) then  $key \leftarrow key_{1-8}$ ;
    else if ( $H = '2'$ ) then  $key \leftarrow key_{1-16}$ ;
    else if ( $H = '3'$ ) then
         $V_{9-16} \leftarrow$  the next 8 bits of  $P_{addr}$ ;
        check  $V_{9-16}$  in  $T_{index}[V_{1-8}].sub\_index\_table$ ;
        if (found) then  $key \leftarrow key_{1-24}$ ;
        else  $key \leftarrow key_{1-16}$ ;
     $ptr \leftarrow get(T_{hash}, key)$ ;
    if ( $ptr = NULL$ ) then
        return default next hop;
    else if ( $ptr$  is pointer of info) then
        compare  $P_{addr}$  with  $ptr \rightarrow prefix$ ;
        if (same) then return  $ptr \rightarrow next\_hop$ ;
        else return default next hop;
    else if ( $ptr$  is pointer of array) then
        prefetch( $ptr \rightarrow item$ );
        do_linear_search( $ptr \rightarrow item$ );

```

Figure 4.20: Pseudo code of IP address lookup using index table.

4.3.4 Lookup Table Updates

There are three kinds of updates in the routing table, including change of the next hop, insertion of a prefix, and deletion of a prefix. Among them, the change of the next hop is the simplest update. The approach of changing the next hop is similar to searching an IP address in the routing table. The difference is that the next hop is returned to forward the packet in searching an IP address, while the next hop will be replaced by a new next hop in the changing next hop process. Like searching an IP address, changing the next hop only needs one external memory access. Therefore, we will not discuss this kind of update. Compared with the change of the next hop, the insertion and the deletion are more complicated to operate. We will discuss these two kinds of updates separately.

4.3.4.1 Insertion of a Prefix

A new prefix with its next hop will be inserted into the routing table in a router, to which if a new network is connected. When the new prefix is inserted into the hash table, the index table may be changed in some cases. We will describe the insertion with the pseudo code shown in Figure 4.21. Given a prefix P to be inserted, we first check its head in the index table with its first 8 bits. There are several cases according to the value of the head.

- If the head is '0', change the head in the index table to '1'. The first 8 bits of P are extracted as the hash key, then put the prefix into a small search group in the hash table with the key.
- If the head is '1', search the hash table with the first 8 bits of P as its key to get the size of corresponding the small search group. If the size is smaller than 16,

place P into this small search group. If the size is equal to 16, the head in the index table is changed to '2'. In this case, all prefixes in this small search group will be rehashed using their first 16 bits as hash keys. This means that these prefixes will not be in one group and these 17 prefixes (16 old prefixes plus P) will be regrouped into a few new small search groups according to their first 16 bits. If some of these 17 prefixes have lengths less than 16, they will be extended into prefixes with length 16. The old small search group will be deleted to preserve memory space.

- If the head is '3', check the sub-index table with the second 8 bits of the prefix, V_{9-16} . If V_{9-16} is found in the sub-index table, the head remains '3' and place P with its first 24 bits as the hash key into the hash table. In this case, if the length of P is less than 24 and greater than 16, P will be extended to a number of prefixes that are inserted into the small search groups with 24-bit hash keys. If V_{9-16} is found in the sub-index table, the head will be considered as '2'. In this case, the insertion operation will be described as follows.
- If the head is '2', the first 16 bits of the prefix are used as the hash key to look up the hash table. Check the size of the small search group. If the size is smaller than 16, place the prefix into the small search group. If the size equals 16, the head in the index table is changed to '3'. In this case, the next 8 bits of these prefixes are put into the corresponding sub-index table in the index table. All prefixes in this small search group and the new prefix will be regrouped into different small search groups with their first 24 bits. Before inserting the prefix into the hash table, the prefix is extended if required.

Input: Index table, T_{index} , Hash table, T_{hash} , and prefix, P_{pre}
output: none

```

Insert ( $T_{index}, T_{hash}, P_{pre}$ )
   $V_{1-8} \leftarrow$  first 8 bits of  $P_{pre}$ ;
   $len \leftarrow$  the length of  $P_{pre}$ ;
   $H \leftarrow T_{index}[V_{1-8}].head$ ;
  if ( $H == '0'$ ) then  $T_{index}[V_{1-8}].head \leftarrow 1$ ;  $key \leftarrow key_{1-8}$ ;
    put  $P_{pre}$  into  $T_{hash}$  with  $key$ ;
  if ( $H == '1'$ ) then  $key \leftarrow key_{1-8}$ ;
     $N_{1-8} \leftarrow \#$  of  $S_{key}$  from get ( $T_{hash}, key$ );
    if ( $N_{1-8} < 16$ ) then put  $P_{pre}$  into  $T_{hash}$  with  $key$ ;
    else  $T_{index}[V_{1-8}].head \leftarrow '2'$ ;

    if ( $len < 16 \ \& \ len > 8$ ) then  $S_{ext} \leftarrow extend(P_{pre})$ ;
      Insert ( $T_{index}, T_{hash}$ , every  $P_{pre}$  of  $S_{ext}$ );

    else Insert ( $T_{index}, T_{hash}, P_{pre}$ );
      Insert ( $T_{index}, T_{hash}$ , every  $P_{pre}$  of  $S_{key}$ );
      delete  $S_{key}$  with  $key_{1-8}$ ;

  if ( $H == '3'$ ) then  $V_{9-16} \leftarrow$  decimal value of second 8 bits of  $P_{pre}$ ;
    check  $V_{9-16}$  in  $T_{index}[V_{1-8}].sub\_index\_table$ ;
    if (found) then
      if ( $len < 24 \ \& \ len > 16$ ) then  $S_{ext} \leftarrow extend(P_{pre})$ ;
        Insert ( $T_{index}, T_{hash}$ , every  $P_{pre}$  of  $S_{ext}$ );
      else put  $P_{pre}$  into  $T_{hash}$  with  $key_{1-24}$ ;
    else  $H \leftarrow '2'$ ;

  if ( $H == '2'$ ) then  $key \leftarrow key_{1-16}$ ;
    if ( $len < 16 \ \& \ len > 8$ ) then  $S_{ext} \leftarrow extend(P_{pre})$ ;
      Insert ( $T_{index}, T_{hash}$ , each of  $S_{ext}$ );
    else  $N_{1-16} \leftarrow \#$  of  $S_{key}$  from get ( $T_{hash}, key$ );
      if ( $N_{1-16} < 16$ ) then put  $P_{pre}$  into  $T_{hash}$  with  $key$ ;
      else  $V_{9-16} \leftarrow$  next 8 bits of  $P_{pre}$ ;
         $T_{index}[V_{1-8}].head \leftarrow '3'$ ;
        Insert ( $T_{index}, T_{hash}, P_{pre}$ );
        Insert ( $T_{index}, T_{hash}$ , every  $P_{pre}$  of  $S_{key}$ );
        delete  $S_{key}$  with  $key_{1-16}$ ;

```

Figure 4.21: Pseudo code of inserting prefix.

4.3.4.2 Deletion of a Prefix

Deletion operations are similar to the insertion. But the process is a reversal of the insertion process. The pseudo code for deletion is shown in Figure 4.22. For a prefix P to be deleted, get its head from the index table. Several different deletions can occur depending on the value of the head.

- If the head is '0', do nothing (this means that there is no such prefix in the forwarding table).
- If the head is '1', search P from the small search group in the hash table. Delete P from the small search group and check the size of this group. If the size is 0, change the corresponding head in the index table to '0' and delete the small search group to free memory space.
- If the head is '3', check the sub-index table with the next 8 bits of P , V_{9-16} .
 - ▶ If V_{9-16} is found in the sub-index table, the head is considered as '3'. Then find P with its first 24-bit as the hash key in the hash table and delete it. If the length of P is less than 24, extend P to extended prefixes with length 16 and delete all these extended prefixes. After each deletion, compute the number of prefixes that share the same first 16 bits with P . If the number is less than 17, delete V_{9-16} from the sub-index table. Regroup all these prefixes into one small search group with the hash key being the first 16 bits of P . Then delete the old small search groups in which these prefixes are stored. Check the sub-index table. If there is no value in the sub-index table, change the head to '2'.
 - ▶ If V_{9-16} is not found in the sub-index table, the head is considered as '2'. The deletion operation is discussed as follows.

- If the head is '2', first check the prefix length of P . If the length is greater than or equal to 16, find P from the hash table and delete it. If the length is greater than eight and less than 16, extend P to extended prefixes with length 16. For each of the extended prefixes, we first must find it in the hash table. Then we check whether the prefix is an extended prefix or an original one in term of the value of the extension flag. If the flag is set to one, which means it is an extended prefix, delete this extended prefix. If the flag is zero, it indicates that this prefix is an original one. In this case, we keep this prefix. After deleting a prefix, compute the number of prefixes that share the same first 8 bits with P . If the number is less than 17, regroup all these prefixes into one small search group, with a hash key being the first 8 bits of P . Then delete the old small search groups in which these prefixes are stored. The head in the index table is changed to '1'.

Input : Index table, T_{index} , Hash table, T_{hash} , and Prefix P_{pre} ;

Output: none;

Delete (T_{index} , T_{hash} , P_{pre})

$V_{1-8} \leftarrow$ the first 8 bits of P_{pre} ;

$len \leftarrow$ the length of P_{pre} ;

$H \leftarrow T_{\text{index}}[V_{1-8}].\text{head}$;

If ($H='0'$) **then** do nothing;

If ($H='1'$) **then** $key \leftarrow key_{1-8}$;

$P_{\text{pre}} \leftarrow \text{get}(T_{\text{hash}}, key)$;

delete P_{pre} ;

$N_{1-8} \leftarrow \# \text{ of } S_{\text{key}} \text{ from } \text{get}(T_{\text{hash}}, key)$;

If ($N_{1-8} < 1$) **then** $T_{\text{index}}[V_{1-8}].\text{head} \leftarrow '0'$;

If ($H='3'$) **then** $V_{9-16} \leftarrow$ the second 8 bits of P_{pre} ;

check $T_{\text{index}}[V_{1-8}].\text{sub-index-table}$ with V_{9-16} ;

If (found) **then**

If ($len < 24$ and $len > 16$) **then** $S_{\text{ext}} \leftarrow \text{extend}(P_{\text{pre}})$;

Delete (T_{index} , T_{hash} , every P_{pre} of S_{ext});

else $key \leftarrow key_{1-24}$;

$P_{\text{pre}} \leftarrow \text{get}(T_{\text{hash}}, key)$;

delete P_{pre} ;

check N_{1-16} ;

If ($N_{1-16} < 17$) **then** remove V_{9-16} from $T_{\text{index}}[V_{1-8}].\text{sub-index-table}$;

regroup all prefixes in S_{1-24} with key_{1-16} ;

If (there is no value in $T_{\text{index}}[V_{1-8}].\text{sub-index-table}$) **then**

$T_{\text{index}}[V_{1-8}].\text{head} \leftarrow '2'$;

else $H \leftarrow '2'$;

If ($H='2'$) **then** $key \leftarrow key_{1-16}$;

$P_{\text{pre}} \leftarrow \text{get}(T_{\text{hash}}, key)$;

delete P_{pre} ;

check N_{1-8} ;

If ($N_{1-8} < 17$) **then** $T_{\text{index}}[V_{1-8}].\text{head} \leftarrow '1'$;

regroup all prefixes in S_{1-16} with Key_{1-8} ;

Figure 4.22: Pseudo code of prefix deletion

4.4 Analysis of Algorithm

In this section, we will analytically evaluate the proposed algorithm in terms of memory accesses, memory usage, updates and flexibility. By comparing it with other algorithms, we are able to give an illustration of our design goal.

4.4.1 Memory Accesses for Lookup

In our proposed design, the major part of the search time is from memory accesses which are required by looking up hash table, checking index table and searching small search group. In each one of these three operations, the number of memory accesses is different in term of different address lookups.

Memory accesses of checking index table: For every address lookup, it is necessary to check the index table to get the head information, which involves one memory access. In case of head '3', a sub-index table needs to be checked which results in one additional memory access. Therefore, checking the index table requires one to two memory accesses.

Memory accesses of hash table lookup: To lookup the hash table with the hash key, only one memory access is required to get the hash table entry in the best case, as the hash key can be converted into the index to the hash table by the hash function. More memory accesses, however, may be needed to get the entry belonging to the hash key if there are hash collisions with the key. The maximum memory access of hash table lookup depends on the hash collisions in the worst case. So one hash lookup needs one to n memory accesses, where n are the maximum hash collisions in the hash table. The

maximum number of hash collisions varies for different routing tables. The hash table size increases as this maximum number increases. Therefore, employing appropriate hash functions can reduce maximum collisions.

Memory accesses of searching small search group: The memory accesses in this operations depends on not only the destination addresses, but also on the size of a small search group. In the best case, the first element in the small search group is the best matching prefix, which only consume one memory access. But in the worst case, all the elements in this group will be checked before the BMP is found. Hence, the memory accesses of searching a small search group ranges one to n , where n is the size of the small search group that is 16 in the worst case.

For the algorithm of binary search on prefix lengths in which different long prefixes are stored in different hash tables, each IP lookup requires one to five hash lookups. Therefore, one lookup will results in n to $5n$ hash lookups, where n is the average hash collisions.

In the multi-way and multi-column search algorithms, when the result of searching the initial table is a pointer to a small search table, the small search table is prefetched into one cache line. If six-way search is used, there are 11 possible cache memory accesses for each small search table. However, searching the small search table may result in a pointer to another small search table. In this case, a new small search table will be loaded into one cache line to find the BMP. So one IP lookup may involve several main memory accesses (searching small search tables) and many internal memory accesses. The number of internal memory accesses cannot be controlled. For the Mae-

East database, there are five main memory accesses and 44 internal memory accesses.

Table 4.1 presents the memory access comparisons of different lookup algorithms.

Table 4.1: Memory access comparisons of different lookup algorithms

Algorithm	Number of external memory accesses	n	m
Multi-way and multi-column	$1 \sim (1+12\log_6^n)$	The maximum number of prefix entries in a small	-
The Lulea	4~12		-
Binary search on prefix	$1 \sim 5m$	-	The maximum hash collisions in a hash table
The proposed	$1 \sim n + m$	The maximum size of a small search time, ≤ 16 .	The maximum hash collisions in the hash table

4.4.2 Memory Usage of Lookup Table

While establishing the index table and the hash table, memory is allocated to hold these tables. In this algorithm, memory usage depends on not only the number of prefixes, but also the number of extended prefixes. In this section, the memory usage is analyzed based on the index table, hash table and extended prefixes. Some results will be compared with other algorithms.

Each element in the index table consists of a one-byte head and an eight-byte sub-index table (256 bits), so the index table has a total size of 2.25K Bytes (256×9 bytes). Thus the index table can be easily fit into the primary cache.

There are four fields in a hash table element including a 24-bit identifier, an 8-bit size, a 32-bit pointer and a 32-bit next pointer. So each element in the hash table has 12 bytes. Each element in a small search group contains two 32-bit words that are 8 bytes.

Since each prefix is an element in a small search group, for a database with 30,000 prefix entries, all small search groups have a total of 240K Bytes. Hash table size depends on the number of small search groups. Suppose each small search group has two elements on average, and there are 15,000 small search groups. If we choose the load factor (the number of prefixes over the hash table size) as 0.75, the hash table size is 20,000. So the memory usage for the hash table is 240K Bytes. Total main memory requirements for this database is 480K Bytes, in which the extended prefixes are not considered. If we permit 50% extended prefixes, an extra 240K Bytes of memory space is required. Table 4.2 shows memory requirements for several algorithms.

Table 4.2: Memory requirement comparisons of several lookup algorithms.

Algorithm	The Lulea	Multi-way and multi-column	Binary search on prefix length	Proposed algorithms
Memory usage for around 30K routing prefixes (K Bytes)	160	950	1600	720

Although it is not a serious problem in most existing lookup algorithms, memory usage must be considered in developing a lookup algorithm. Some algorithms use huge memory space to reduce the memory accesses, which is impractical.

4.4.3 Prefix Update Rate

Insertion and deletion are similar to searching a best matching prefix, but it is more complicated than the lookup as inserting or deleting a prefix may lead to regrouping the prefix entries in a small search group. If the prefix needs to be extended, updating one prefix may result in a few prefixes being updated. In spite of these difficulties, the update in the proposed algorithm has some advantages compared to other algorithms on the main

memory accesses and prefix extension.

In the proposed algorithm, since each prefix belongs to a small search group, updating this prefix only involves memory location modifications within this small search group. So the main memory accesses are limited to within a certain value. In the Lulea algorithm employing a complicated compact data structure, inserting or deleting a prefix may cause the whole forwarding table to be rebuilt. Although the forwarding table is placed into the cache memory, a lot of main memory accesses are required to load the prefix entries into the cache memory. In multi-way and multi-column algorithms, consecutive search ranges are put into one or more small search tables. Updating one prefix may rebuild all these small search tables, in the worst case scenario. In the binary searches on prefix length algorithms, updating a prefix is relatively easier than the other two algorithms. However in the worst case, it will add /remove a hash table and insert / delete several markers into/from several different hash tables. In addition, adding a new hash table needs an additional hash function.

In the proposed algorithm, in the worst case, updating a prefix will cause 128 extended prefixes to be inserted into or deleted from the small search group if this prefix needs to be extended. Multi-way and multi-column algorithms extend all prefixes with prefix lengths less than 16 to extended prefixes with length 16. So a prefix will be extended to at most 256 extended prefixes. In the Lulea algorithm, although there is only one prefix extension if extension is required, but updating one prefix may change the whole data structure. Table 4.3 shows the update comparisons of different lookup algorithms.

Table 4.3: Update comparisons of different lookup algorithms

Algorithm	Number of memory modifications	Number of extended prefixes
Multi-way and multi-column	$2 \sim 11(\log_6^n)$	$0 \sim 256$
The Lulea	$4 \sim k$	$0 \sim 1$
Binary search on prefix lengths	$1 \sim 5m$	$0 \sim 5$
The proposed	$2 \sim (m+17)$	$0 \sim 128$

n : The maximum number of prefix entries in a small search table. (n equals to 384 for Mae-East).

m : The maximum hash collisions in a hash table.

k : The number of memory accesses to rebuild the whole forwarding table.

4.4.4 Flexibility of Algorithm

An efficient lookup algorithm should propose a high flexibility feature. This means that the performance of this algorithm will not change or will change slightly when different routing databases are applied into the algorithm. Meanwhile, this algorithm should not require specific implementation environments to reach high performance lookup. The flexibility feature comparison of different lookup algorithms is given in Table 4.5.

Table 4.5: Flexibility comparisons of different lookup algorithms

Algorithm	Performance changing with different condition	Flexibility
Multi-way and multi-column	Lookup speed (in the worst case)	Medium
The Lulea	Lookup speed and memory usage	Low
Binary search on prefix lengths	Memory usage	High
The proposed	Memory usage	High

In the proposed algorithm, the index table loaded into on-chip memory has a fixed size of 2.25K bytes, which could be easily satisfied in an Internet router. The hash table size depends on the size of a routing database, but it will not require much memory space. This was analyzed in Section 4.4.2.

In the Lulea algorithm, the whole forwarding table is fit into the cache memory. But this algorithm cannot guarantee that all kinds of routing databases can be fit into the cache memory as some Internet routers have a small size cache memory. In this case, the performance of this algorithm will be significantly degraded.

In multi-way and multi-column algorithms, all prefixes are stored in main memory and only a small search table with 32bytes is prefetched into one cache line. So this algorithm can easily be implemented in the routers, but the lookup speed in this algorithm depends on the size of a search table, in the worst case. Hence, the routing databases will affect its worst-case performance.

5.Implementation and Performance Analysis

In the analysis of the algorithm, some performance metrics such as worst-case internal memory access, worst-case prefix extension, and flexibility can be theoretically determined. Non-deterministic performance metrics such as the average prefix extension, the number of memory accesses, the memory usage in hash table and the update rate, can be analyzed based on statistical data observed from the implementation of the algorithm. In this chapter, non-deterministic performance metrics will be analyzed by implementing the proposed algorithm, part of which will be compared with the theoretical results of the algorithm. Before the implementation, the implementation environment is described as follows.

Four snapshot databases from the Internet Performance Measurement and Analysis (IPMA) project [44] are employed in this implementation, which include Mae-East, Mae-West, AADS and PAIX. Each database has a different size, as shown in Table 5.1.

Table 5.1: The number of prefixes for four databases.

Database	Mae-East	Mae-West	AADS	PAIX
Number of prefixes	24446	29601	29349	10859

The proposed algorithm is implemented on a GPP with 256K bytes of cache memory and 64M bytes of main memory. The implementation is done by creating a C program without threads that models the data structures in the proposed algorithm. For each database, an index table of size of 2.25K bytes is loaded into cache memory. The hash table is placed into main memory. Since the hash table size is determined by the

number of prefix entries, the main memory will be dynamically allocated for the hash table. A small search group stored in the hash table will be retrieved into the cache memory if the size of the small search group is greater than one.

5.1 Prefix Expansion Results

In building the hash table, some prefixes may be extended as required. The number of expanded prefixes depends on the size of the small search group. Table 5.2 presents the results of the implementation. To fit a small search group into the cache line(s), the maximum size of the small search groups are selected as 8, 12, 16, 20 or 24. The results do not include the original prefixes. The extended prefixes are one of the two types: Prefixes extended to length 16, and prefixes extended to length 24.

Table 5.2: The prefix expansion results for four databases.

The maximum size of small search group	The number of prefixes			
	Mae-East	Mae-West	AADS	PAIX
8	34696	39535	41251	11022
12	18574	20751	21233	5625
16	9532	10328	11065	2890
20	4257	4360	2702	1438
24	1232	1591	0	0

The analysis of the results in Table 5.2 is given in Figure 5.1. From the results in Table 5.2, we can observe that the number of extended prefixes (denoted by N_{extended}) is determined by the maximum size of a small search group (denoted by N_{small}). With the increasing N_{small} , N_{extended} drops sharply. When N_{small} is a small value, such as 8, N_{extended} even exceeds the original prefix number for all four databases. This will require more

memory storage. If N_{small} grows to 24, N_{extended} is reduced to a small value. For the database AADS and PAIX, no prefixes is needed to be extended. It means that we do not need to extend the prefixes to lengths 16 or 24 if the N_{small} is large enough.

Figure 5.1: The prefix extension results on implementation. (a) The number of extended prefixes versus the size of the small search group. (b) Amount of extended prefixes versus the size of the small search group.

However, a large N_{small} causes two problems. The first one is that the search in the cache memory will take more much time, which will slow down the lookup rate. The second is a problem on memory. In our algorithm, consecutive memory blocks are needed to store a small search group so that it can fit into the cache lines. But the consecutive memory blocks cannot always be found, especially for large memory blocks. Therefore, in selecting N_{small} , there is a trade-off between the number of extended prefixes and the lookup speed. In our implementation, we choose 16 for N_{small} , with which at most 4 cache lines are used to fit a small search group. N_{small} can be selected based on different memory requirements. Thus, IP lookup performance can be controlled by adjusting the value of N_{small} .

5.2 Memory Access Results for Lookup

Memory accesses in the proposed algorithm consist of two types: Main memory accesses and cache memory accesses (or on-chip memory accesses). Hash table lookup (this operation is called hash lookup in this algorithm) involves with main memory accesses, while checking the index table and searching the prefetched small search group involves with accesses to cache memory. In this section, we will provide an analysis of the lookup rate by the implementation results of memory accesses.

5.2.1 Main Memory Access Results

Hash lookup speed is decided by hash collisions. So the average hash collisions reflect the average hash lookups, while the maxims hash collisions indicate the worst-case hash lookups. In the proposed algorithm, we expect that both collisions are as small as possible. Since two factors, the hash table size and the hash function, influence these two cases, we will analyze main memory accesses both on average case and in the worst case hash collisions based on the hash table size and hash functions.

Hash table size: The hash lookup implementation results for different hash table sizes are presented in Figure 5.2. Figure 5.2 (a) shows average collisions versus the hash table size, and Figure 5.2 (b) the maximum collision versus the hash table size.

From Figure 5.2 (a), we see that when the hash table size is small, there are more than one half of the prefixes colliding with each other for database Mae-East, Mae-West and AADS. Since the size of database PAIX is much smaller than that of the other three

databases, collisions only happen among 30 percent of prefixes for PAIX. As the hash table size increases to 52115, the average collisions drop to about 20 percent for Mae-East, Mae-West and AADS, and 6 percent for PAIX. With a fixed hash table size, the larger the size of the database is, the fewer the average collisions occur. Although the average collisions are reduced when we enlarge the hash table, the hash lookups do not benefit much on the average number of collisions when the size reaches a certain value, such as 52115.

Figure 5.2: The analysis of the hash table results on implementation. (a)The average collisions versus the hash table size (b)The maximum collisions versus the hash table size.

As we see from Figure 5.2 (b), the worst case in the hash lookups is 9~ 10 or 9 for Mae-East, Mae-West or AADS, respectively, and the maxim number of collisions is six for PAIX. For the Mae-East, Mae-West and AADS databases, which have approximately the same database size, almost the same worst case. But the database PAIX that has a smaller size apparently has a better worst case than that for the other three databases. This indicates that different databases have the same worst case when their hash table sizes and database sizes are approximately the same. A large hash table size can improve the worst case for all databases, but the improvement is significant only when the hash table size is very large.

Hash functions: To investigate the effect of hash functions on the hash lookups, we employ four different hash functions with a fixed hash table size of 42115. The hash functions are based on the model: $h(k) = ak + b \bmod N$ where N is the table size, and a and b are variable integers. These four hash functions are:

$$F1: h(k) = k \bmod Size,$$

$$F2: h(k) = (k * k + 3) \bmod Size,$$

$$F3: h(k) = (k + k \bmod Size) \bmod Size$$

$$F4: h(k) = ((k * p + q) \bmod m) \bmod Size$$

The implementation results are given in Figure 5.3. The results show that changing hash functions has little effect on the hash performance. There is no clear relationship between the variations of the average collision or the maximum collisions and the hash functions. For instance, the hash function $h(k) = (k + k \bmod Size) \bmod Size$ improves the average collisions for Mae-East, but worsens AADS and PAIX. So the hash performance could not be improved by selecting different hash functions. Additionally, complicated

hash functions cause more multiplication operations in hash lookups, which will add more search time. Therefore, $h(k) = k \bmod \text{Size}$ is selected in our algorithm simply as

Figure 5.3: The analysis of hash function results on implementation. (a)Average collisions versus the hash functions. (b)The maximum collisions versus the hash function.

Summary: With the condition that the hash table size is selected as 42115 and the hash function is $h(k) = k \bmod \text{Size}$, the implementation results for the hash lookup are given in Table 5.3. The number of hash lookups is shown row 1 to row 6. The integers in the first row represent the number of 1-time hash lookups; the integers in the second row are the number of 2-time hash lookups, and so on. The 7th row is the average hash lookups for each database and the last row gives the maximum number of hash lookups or the worst case for each database. The percentiles are also shown in rows 1 to 6, which will be used to analyze the average case and the worst case.

From the result in the first row in the Table 5.3, we see that more than 70 percent of the prefixes only need a one-time hash lookup. In other words, hash collisions only occur among less than 30 percent of the prefixes. More than 94 percent of the prefixes only need at most two hash lookups to be found in the hash table. From the average hash

lookups, this implies that that large hash tables can lead to a high performance in the hash lookups.

Table 5.3: The hash lookup results on the implementation.

Times of hash lookup	East		West		AADS		PAIX	
	Number	%	Number	%	Number	%	Number	%
1	14696	73.34	15489	70.52	16089	71.17	8273	92.97
2	4514	22.53	5363	24.42	5286	23.38	591	6.64
3	739	3.69	985	4.48	1084	4.79	33	0.37
4	85	0.42	117	0.53	139	0.61	2	0.02
5	3	0.01	8	0.04	10	0.04	0	0
6	0	0	1	0.005	0	0	0	0
Average	1.31		1.35		1.35		1.07	
Worst case	5		6		5		4	

In the analysis presented in Section 4.4.1, the average speed to search main memory is one hash lookup, each of which requires one main memory access. The implementation results of four databases for searching the main memory access are between 1.07 and 1.35 hash lookups. This indicates that the implementation results are very close to the results of the algorithm analysis. The worst case in the algorithm analysis is m hash lookups, which varies between four and six in the implementations. But m can be reduced by increasing the hash table size if the memory requirements are met.

5.2.2 Cache Memory Access Results

Random destination IP addresses can range from 0.0.0.0 to 255.255.255.255, but in fact the destination IP addresses are not uniformly distributed over the range 0.0.0.0 to 255.255.255.255 according to the investigation in [45]. To make the implementation more practically, we use the addresses that must have matching prefixes in the database.

Checking the Index Table: The results of checking the index table are given in Table 5.4, in which the first and the second row are the percentile of the number of cache memory accesses and the last row is the average cache memory accesses.

For all databases, around 50 percent of the IP lookups need one cache memory access, and 50 percent of IP lookups require two cache memory accesses. The average cache memory accesses are between 1.35 and 1.52. From the results, we can conclude that the proposed algorithm can efficiently cut the search range down with the index table at the cost of one or two cache memory accesses.

Table 5.4: Results on cache memory accesses for checking the index table.

Number of cache memory accesses	Mae-East	Mae-West	AADS	PAIX
1	48.24 %	55.40%	54.02%	65.32%
2	51.76 %	44.60%	45.98%	34.68%
Average cache memory accesses	1.52	1.45	1.46	1.35

Searching Small Search Group: In the proposed design, only these small search groups prefetched into cache memory need cache memory accesses. To investigate how many small search groups need prefetching in this implementation, the number of the small search groups with one element are given, as shown in Table 5.5.

Table 5.5 presents implementation results on the sizes of small search groups. Without loss of generality, we use small ranges between size 2 and 15 instead of individual sizes to analyze the distribution. The rows from 1 to 6 represent the size distribution of all small search groups for the four databases. The seventh row shows the average cache memory accesses, while the last row gives the worst-case cache memory

accesses. When the size of a small search group is 1, there is no cache memory access. We consider the range 2 to 5 as low cache memory accesses, 6~9 and 10~12 as medium cache memory accesses, and 13~15 as high cache memory accesses. When the size is 16, it is the worst case in the small search group.

Table 5.5: The results on cache memory accesses for searching small search groups.

Size of small search groups		Mae-East		Mae-West		AADS		PAIX	
		number	%	number	%	number	%	number	%
1		22771	86.6	25103	84.5	26102	85.5	7775	81.3
2~5		3115	11.8	3994	13.4	3857	12.6	1548	16.1
6~9		264	1.0	411	1.38	390	1.30	161	1.68
10~12		84	0.32	104	0.35	95	0.31	55	0.58
13~15		47	0.18	56	0.19	54	0.18	20	0.21
16		15	0.05	16	0.05	13	0.04	2	0.02
Cache memory accesses	Average case	0.43		0.50		0.47		0.62	
	Worst case	16		16		16		16	

Figure 5.5 is the frequency histogram of the size of the small search groups based on Table 5.5. According to the results in Figure 5.5, over 80 percent of the small search groups have only one item. This indicates that on the average more than 80 percent of the destination addresses do not need to search prefixes in cache memory. Furthermore, even for the small search groups with more than one item, about 98 percent of them have at most five cache memory accesses. For all databases, the worst case is 16 cache memory accesses. But the worst case may only happen among less than 0.06 percent small search groups. The results in Table 5.5 also show that there are around only 0.5 cache memory accesses to search small search groups on average.

Figure 5.5: Frequency histogram of the number of items in the small search groups.

The majority of cache memory accesses come from checking the index table. Searching small search groups contributes to only a small part of the on-chip memory accesses. This result is consistent with the analyzing results in Section 4.4.2 which states that a major part of small search groups do not need on-chip memory accesses.

5.3 Results of Lookup Table Memory Usage

There are three data structures consuming memory space: The hash table, the index table and the small search groups. The index table resides in the cache memory (or SRAM), while the hash table is placed in the main memory (or DRAM). The small search group that is a part of the hash table will probably be retrieved into the cache. Since it has a special structure discussed in Section 4.2.1, the index table has a fixed size of 2.25 Kbytes that is independent of the size of the forwarding tables. Therefore, the major memory usage in the algorithm will be used to hold the hash table and the small search group. Figure 5.6 shows the memory usage against the hash table size.

Figure 5.6: The hash table size versus the memory usage.

As we see from the diagram, the memory usage increases gradually with the hash table size. This implies that a larger hash table does not consume too much memory space, which is usually the case for data structures in other algorithms. Therefore we can conclude that with sufficient memory, the hash table size should be as large as possible to improve the hash lookup performance. From Figure 5.6, the memory usage for three large databases increases slower than the small database PAIX. That implies that it is more efficient to improve the hash table performance for large databases than for small databases as the hash table size increases.

For databases with around 30K prefix entries such as Mae-West and AADS, the memory usage is about 900Kbytes when the number of extended prefixes reaches 40% of total prefix entries. This memory requirement is closed to the memory requirements in the algorithm analysis discussed in Section 4.4.2. So in this proposed algorithm, the memory usage is determined by the size of a database, not the data structures.

5.4 Lookup Table Update Results

Lookup table updates consist of the following types of changes: Insertion of a new prefix, deletion of an existing prefix, and next hop change of an existing prefix. For simplification, a prefix that will be updated is called a new prefix in this algorithm. The process of changing the next hop of an existing prefix is similar to that of looking up an IP address. The speed of changing the next hop is the same as the lookup rate. So, in the following we will concentrate on the analysis of updates on insertion and deletion on the number of memory accesses required per routing table update.

For both insertions and deletions, there are two cases to be considered. In an insertion operation, the first case is that the new prefix does not belong to any small search groups. In the second case, the new prefix belongs to an existing small search group. In a deletion operation, the first case is that the new prefix belongs to a small search group that has a size of one, while the second case is that the new prefix belongs to a small search group whose size is greater than one.

In both first cases, the update operation only needs one-memory access. However, both second cases are complicated update operations, since all prefixes in the small search groups must be placed in consecutive memory locations. When one prefix is added or deleted, all other prefixes in that small search group must be relocated into new consecutive memory locations. Another case in insertions and deletions that must be considered is that the new prefix may need to be extended before being updated. In this case, updating a prefix will cause several extended prefixes to be updated. The implementation results of updates are given in Table 5.6.

Table 5.6: Implementation results of updates.

		Mae-East	Mae-West	AADS	PAIX
# of main memory accesses for updating a single prefix.	Average case	3.29	3.34	3.32	3.44
	Worst case	18	18	18	18
# of the extended prefixes for an original prefix.	Average case	6.66	8.08	7.11	3.04
	Worst case	128	128	128	128
# of main memory accesses for updating an original prefix.	Average case	6.09	7.26	6.75	4.13
	Worst case	356	346	354	324

The results in the first row in Table 5.6 are the average case and the worst case of main memory accesses for updating a single prefix which is an original prefix or an extended one. The average case and the worst case of the number of extended prefixes are listed in the second row. The results of updating an original prefix that may have been extended are shown in the third row.

To insert or delete a single prefix, only about three memory accesses are required on average, though the number of memory accesses can be as high as 18 in the worst case. This value should be between two and $m+17$, where m is the maximum collisions. This shows that there are no hash collisions when the worst case happens. Although a new prefix may create several extended prefixes on average, the number of main memory accesses does not increase very much for updating this new prefix. This implies that most of the extended prefixes belong to small search groups of a small size, which need a minimum of two main memory accesses. Consequently, the worst case memory accesses in updates are limited to around 350 main memory accesses.

6. Conclusions and Future Work

This thesis has concentrated on developing a new algorithm for IP address lookup, called IP address lookup using index table. Before detailing the proposed algorithm, we introduced the background of route lookup and addressed issues of IP lookups. Three existing IP lookup schemes were reviewed and the current status of the research in this field was presented.

6.1 Conclusions

After presenting the proposed algorithm and implementing of the algorithm on a GPP, we have drawn the following conclusions.

- Using an index table, the proposed algorithm can perform fast IP address lookups. An index table is employed to reduce the search range from the whole routing table to a small search group with the cost of one or two fast cache memory accesses. The main search time is around one hash table lookup, which needs one or two main memory accesses based on the implementation environment. Searching a small search group requires a half number of the cache memory accesses on the average or 16 cache memory accesses in the worst case. So, with this algorithm, looking up an IP address needs one main memory access and three fast cache memory accesses in the average case.

Updating a prefix may involve modifications of the memory location, but only in a small search group. The prefix may be extended to a few extended prefixes before being updated. In the worst case, updating the prefix leads to 128 extended prefixes

being updated. The average number of memory accesses required by all four databases for inserting or deleting a prefix is between four and seven.

- To search the BMP for a given IP address, two cache memory accesses for checking index table, five hash lookups for looking up the hash table and 16 cache memory accesses for searching the small search group are theoretically required in the worst case in the database Mae-East. The worst case hash lookup can be reduced by increasing the hash table size.
- The cache memory usage for the index table is fixed to 2.25K bytes for all kinds of databases. The main memory requirement of the hash table depends on the hash table size and the size of the database.
- With low memory requirements and relatively simple data structures, this algorithm can be easily implemented on Internet routers without specific hardware requirements, therefore providing high flexibility.

6.2 Summary of contributions

- A new IP address lookup algorithm has been developed. This algorithm is designed for fast IP lookup, fast prefix updates, low memory requirement and high flexibility. The index table and the small search group with ideal structures are proposed to speed up the lookup rate and update rate. Using a single hash table to hold all prefix entries not only reduces the hash lookup times, but also simplifies the complexity of a prefix search and table establishment.

- The implementation of this algorithm on GPP is presented, which also illustrates the algorithm's high flexibility. The snapshots of four routing tables are employed in the implementation. The algorithm performance is evaluated by comparing the implementation results and theoretical results.

6.3 Future Work

This thesis has been on target for developing a new IP lookup algorithm with which the packet processing capacity of IP routers can keep up with the exponential increase in data rates of physical links. Although it achieves good performance according to the simulation results, the algorithm still has a little room for improvement.

- **Implementation on different databases**

The simulation results are based on four typical routing databases which have some similarities, such as the distribution of prefixes. Since the length distribution has an effect on the number of extended prefixes, we need more simulation results based on databases with extremely different length distributions to evaluate the flexibility of the proposed algorithm. Additionally, it is important to implement the algorithm on databases with different sizes. Practically, it is not easy to find appropriate databases to evaluate the performance of the algorithm. Therefore, these databases need to be created randomly using a C program.

- **Scalability to IPv6.**

IPv6 is designed for the next generation network. There are a significant difference in the architectures between IPv4 and IPv6. The proposed algorithm designed for IPv4

network cannot be employed directly to IPv6 networks. To scale the algorithm to the future network, the data structure of the algorithm (especially the index table), must be modified to store more information on the prefix lengths of IPv6 addresses whose lengths vary from 1 to 128.

- **Adapting the algorithm into NP architecture**

The proposed algorithm was simulated on a GPP. To apply the algorithm to the routers based on Network Processors (NP), the algorithm must be adapted into NP architecture.

References

- [1] S. Keshav and R. Sharma, *Issues and Trends in Router Design*, IEEE Communications Magazines, May 1998.
- [2] Douglas E. Comer, *Internetworking with TCP/IP vol I: Principles, Protocols, and Architecture*, Prentice-Hall, 1995.
- [3] Internet Engineering Steering Group, *Applicability Statement for the Implementation of Classless Inter-Domain Routing (CIDR)*, RFC 1517, Available at <http://www.ietf.org/rfc/rfc1517.txt>, 1993.
- [4] S. Fuller, T.Li, J.Yu and K. Varadhan, *Classless inter-domain routing(CIDR): an address assignment and aggregation strategy*, RFC 1517, Available at <http://www.ietf.org/-rfc/rfc1517.txt>, 1993.
- [5] AS4637 BGP Table Data, Available at <http://bgp.potaroo.net/as4637/>, February 2003.
- [6] C.Labovitz, G.Malan and F.Jahanian , *Internet Routing instability*, Proceedings of ACM sigcomm, p115-16, September 1997.
- [7] Reed E. Hundt, *You Say You Want a Revolution*, Yale University Press, New Haven, 2000.
- [8] *Internet Domain Survey*, Available at <http://www.isc.org>, 2003.
- [9] Krishna M. Sivalingam and Suresh Subramaniam (Editors), *Optical WDM Networks: Principles and Practice*, Kluwer Academic Publishers, Mar. 2000
- [10] *Rambus DRAM for OC192 Data Rate Line Card Applications*, Available at http://www.rambus.-com/downloads/OC192_App_Note.pdf, 2002
- [11] D.E. Knuth, *Optimum Binary Search Trees*, Acting Information, 1 (1971) pp14-25.
- [12] D.R. Morrison, *PATRICIA — practical algorithm to retrieve information coded in alphanumeric*, Journal of the ACM, vol. 15, No.14, pages 514-34, October 1968.

- [13] K.Sklower, *A tree-based packet outing table for Berkeley Unix*, In Proceedings of 1991 Winter USENIX Conference, Dallas, Pages 93 –99,1991.
- [14] Stefan Nilsson and Gunnar Karlsson, *IP Address Lookup using LC-Tries*, IEEE Journal on Selected Areas in Communications, Vol. 17, No. 6, pp. 1083-1092, June 1999.
- [15] S.Nilsson and G.Karlsson, *Fast address lookup for Internet routers*, In Proceedings of Broadband Communications: The Future of Telecommunications, Pages 11-22, Chapman &Hall, 1998.
- [16] V.Srinivasan and G.Va ghese, *Faster IP lookups using controlled prefix expansion*, Proceedings of SIGMETRICS 98, Madison pages 1 –10,1998.
- [17] M.Waldvogel, G.Varghese, J.Tu ner, and B.Plattne, *Scalable high speed IP outing lookups*, ACM Computer Communication Review 27(4): 25 –36,October 1997.
- [18] W.Doeninger, G.Karjoth, and M.Nassehi, *Routing on longest matching prefixes*, IEEE/ACM Transactions on Networking4 (1): 86 –97, February 1996.
- [19] A.Moestedt and P.Sjodin, *IP address lookup in hardware for high-speed routing*, In Proceedings of Hot Interconnects VI, Stanford, August 1998.
- [20] P.Gupta,S.Lin,and N.McKeown, *Routing lookups in hardware at memory access speeds*, In Proceedings of Infocom'98, San Francisco, April 1998.
- [21] Pankaj Gupta, *Algorithms for routing lookups and packet classification*, Ph.D thesis, Stanford University, December 2000.
- [22] Derek Pao, Cutson Liu, Angus Wu, Lawrence Yeung and K. S. Chan, *Efficient hardware architecture for fast IP address lookup*, IEEE INFOCOM 2002, pp. 555-561, June 2002.
- [23] 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 Communications, 17(6): 1093–1104, 1999.

- [24] W. N. Eatherton, *Hardware-Based Internet Protocol Prefix Lookups*, M. Sc thesis, Washington University in St. Louis, 1998.
- [25] Kari Seppänen, *Novel IP address lookup algorithm for inexpensive hardware implementation*, WSEAS Transactions on Communications, Vol. 1, No. 1, pp. 76-84, 2002.
- [26] A. McAuley and P. Francis, *Fast routing table lookup Using CAMS*, In Proceedings of INFOCOM, Pages 1382-1391, March-April 1993.
- [27] Anthony J. Bloomfeld, NJ McAuley, Paul F. Lake, Hopatcong NJ Tsuchiya, and Daniel V. Rockaway, *Fast Multilevel hierarchical routing table using content-addressable memory*, U.S. Patent, serial number 034444.
- [28] Tzi-cker Chiueh and P. Pradhan, *High-Performance IP Routing Table Lookup Using CPU Caching*, IEEE INFOCOM, pp. 1421-1428, 1999.
- [29] C.P. et al, *A 50-Gb/s IP router*, IEEE/ACM Transactions on Networking 6(3): 237-248, 1998.
- [30] David E. Taylor, John W. Lockwood, Todd Sproull, Jonathan S. Turner and David B. Parlour, *Scalable IP Lookup for Programmable Routers*, In Proceedings of IEEE INFOCOM, 2002.
- [31] Belenkiy Andrey, *Deterministic IP Table Lookup at Wire Speed*, Available at http://www.isoc.org/inet99/proceedings/4j/4j_2.htm. New Jersey Institute of Technology USA, July 2000.
- [32] Marcel Waldvogel, *Fast Longest Prefix Matching: Algorithms, Analysis and Applications*, PhD thesis, Swiss federal institute of technology, 2002.
- [33] H.-Y. Tzeng, *Longest prefix search using compressed trees*, in Proc. IEEE Global Communication'98 Conf., Sydney, Australia.

- [34] Henry H., Y. Tzeng and Tony Przygienda, *On Fast Address-Lookup Algorithms*, IEEE Journal on Selected Areas in Communications, Vol. 17, No. 6, pp. 1067-1082, June 1999.
- [35] Kari Seppänen, *Novel IP address lookup algorithm for inexpensive hardware implementation*, WSEAS Transactions on Communications, Vol. 1, No.1, pp. 76-84, 2002.
- [36] M.A.Ruiz-Sanchez, E.W.Biersack, and W.Dabbous, *Survey and taxonomy of IP address lookup algorithms*, IEEE Network Vol.15, pp.8 –23, March./April, 2001.
- [37] B.Lampson, V.Srinivasan, and G.Varghese, *IP lookups using multi-way and multicolumn search*, In Proceedings of IEEE Infocom'98, San Francisco, 1998.
- [38] M.Degemak, A.Bodnik, S.Calsson, and S.Pink, *Small forwarding tables for fast routing lookups*. ACM Computer Communication Review 27(4): 3-14, October 1997.
- [39] Betty Prince, *High Performance memories: new Architecture DRAMs and SRAMs – evolution and function*, Chichester, New York, 1996.
- [40] *Faster Memory for the Pentium 4*, Available at <http://www.extremetech.com/article2/0,3973,336-59,00.asp>, 2002.
- [41] Jim Handy, *The cache memory book*, Academic Press, Boston, 1993.
- [42] Michael T. Goodrich and Roberto Tamassia, *Data Structure and Algorithms in Java*, Wiley, New York, 1998.
- [43] Michael Main, *Data structures & other objects using Java*, Addison Wesley, 1999.
- [44] *Internet Performance Measurement and Analysis Project*, University of Michigan and Merit Network, Available at <http://www.merit.edu/ipma>, 2002.

- [45] Ryo Kawabe, Shingo Ata and Masayuki Murata, *Performance Prediction Methods for Address Lookup Algorithms of IP Routers*, Available at <http://www.anarg.jp/achievements/web2001/papers/rkawabe02ICOINAddressLookup.pdf>, 2000.