PAPER

Towards Dynamic and Scalable High-Speed IP Address Lookup Based on B+ Tree

Gang WANG[†a], **Yaping LIN**[†*b], **Rui LI**[†c], **Jinguo LI**[†d], **Xin YAO**[†e], ***Nonmembers***, ***and*** **Peng LIU**[†f], ***Student Member***

SUMMARY High-speed IP address lookup with fast prefix update is essential for designing wire-speed packet forwarding routers. The developments of optical fiber and 100 Gbps interface technologies have placed IP address lookup as the major bottleneck of high performance networks. In this paper, we propose a novel structure named Compressed Multi-way Prefix Tree (CMPT) based on B+ tree to perform dynamic and scalable high-speed IP address lookup. Our contributions are to design a practical structure for high-speed IP address lookup suitable for both IPv4 and IPv6 addresses, and to develop efficient algorithms for dynamic prefix insertion and deletion. By investigating the relationships among routing prefixes, we arrange independent prefixes as the search indexes on internal nodes of CMPT, and by leveraging a nested prefix compression technique, we encode all the routing prefixes on the leaf nodes. For any IP address, the longest prefix matching can be made at leaf nodes without backtracking. For a forwarding table with u independent prefixes, CMPT requires $O(\log_m u)$ search time and $O(m \log_m u)$ dynamic insert and delete time. Performance evaluations using real life IPv4 forwarding tables show promising gains in lookup and dynamic update speeds compared with the existing B-tree structures.

key words: *IP address lookup, forwarding table, longest prefix matching, dynamic update, B+ tree*

1. Introduction

1.1 Background and Motivation

The rapid growth of Internet traffic and the growing complexity of packet processing are placing extreme demands on the design of high performance routers [1]. One of the core functionalities of router is to compare the destination IP address of every incoming packet against a set of rules in forwarding table to determine the next forwarding hop. IP address lookup is the most time-consuming task in routers since the incoming packets should be processed in wire-speed even as the sizes of routing tables and the rates of packets arrival are dramatically increasing [2].

Currently, IP addresses are divided into two levels of hierarchy: a network prefix and a host address. To utilize the address space efficiently, Classless Inter-Domain Routing (CIDR) is used to allocate prefixes in arbitrary length [3]. However, the IP address lookup process requires the lookup engine searching in variable length prefixes for the longest prefix matching (LPM).

Due to its importance, IP address lookup with longest prefix matching has received significant attention by researchers in the past few years [4]–[10]. Recently, with the rapid growth of internet traffic and IPv6 applications, there has been much renewed research interest in developing dynamic and scalable data structures for high-speed IP address lookup in a large-scale and dynamic forwarding table [11]–[13].

There are several driving factors motivating the research on dynamic and scalable high-speed IP address lookup problem. Firstly, the fiber-optical and high speed interface technologies have been rapidly developed to transmit packets in the speed of 100 Gbps. Therefore, packets processing speed of routers has become the major bottleneck of high performance networks. To keep up with 100 Gbps networks, high-speed IP address lookup is essential for designing wire-speed packet forwarding routers [13]. Secondly, due to the prevalence of the Internet and lots of new emerging network applications, the network traffic in backbone routers is doubling every two months. It's reported that a core router has contained more than 350 K routing prefixes on May 1st of 2011 and the forwarding tables are expending exponentially [14]. Therefore, it necessitates not only the high speed performance of IP address lookup but also dynamic data structures to support fast prefix update. Thirdly, the forwarding tables in backbone routers can be updated hundreds of times every second. It is urgent to develop an IP address lookup algorithm that supports the update rate of at least one thousand times every second to pace the update rate of forwarding tables. Fourthly, the Internet is fetching in sustained growth of users. More IP addresses are required to provide global communication. Despite the Network Address Translation (NAT) technique is involved as a temporary scheme to share one IP addresses within several hosts, the switch to IPv6 address space seems inevitable. Therefore, the algorithms whose lookup speeds are related to the IP address length become less attractive. More scalable IP address lookup technique is imperative.

1.2 Limitation of Prior Art

There are three main categories for the technique of dy-

Manuscript received December 21, 2011.
Manuscript revised April 5, 2012.

[†]The authors are with the College of Information Science and Engineering in Hunan University, P. R. China.
[*]Corresponding author
a) E-mail: wg@hnu.edu.cn
b) E-mail: yplin@hnu.edu.cn
c) E-mail: lirui@hnu.edu.cn
d) E-mail: lijg1985@hnu.edu.cn
e) E-mail: xinyao@hnu.edu.cn
f) E-mail: liupeng@hnu.edu.cn

DOI: 10.1587/transinf.E95.D.2277

Copyright © 2012 The Institute of Electronics, Information and Communication Engineers

namic IP address lookup: schemes based on Ternary Content Address Memory (TCAM), schemes based on trie tree and schemes based on range tree. TCAMs have been implemented in high performance routers for their high-speed lookup by only one single memory access. However, TCAMs are very expensive and suffer from high power consumption. Trie-based schemes are widely used in high-speed routers and easy to implement. But their lookup speeds are highly related to the length of IP address and would decrease dramatically when extending to IPv6 networks. Range tree is a promising technique due to its high matching speed and well-designed scalability.

The most related work to this paper is [13]. The authors in [13] proposed an improved dynamic IP lookup algorithm named MMSPT based on the work in [10]. MMSPT performs with a high speed for dynamic update and scalability in processing IP address lookup. However, there are still two shortcomings of MMSPT: (1) The lookup speed suffers from backtracking on MMSPT for the prefixes which are not the most specific prefixes defined in [13]. Especially with the rapid growth of Internet traffic, the times of backtracking lookup could increase, and this would dramatically decrease the throughput of routers. (2) MMSPT stores all the rules of a forwarding table without compressing. The exponentially expending of forwarding table size may lead to space explosion.

1.3 Our Approach and Contributions

In this paper, we focus on developing a dynamic and high-speed IP address lookup algorithm with good scalability. We are interested in B+ tree to resolve this problem, since for any search keys, the matching operation on B+ tree can be hit on leaf nodes without any backtracking. Furthermore, B+ tree can be constructed with ranges and therefore can be easily extended to IPv6 addresses with good scalability.

Therefore, we firstly build the forwarding table as a Multi-way Prefix Tree (MPT) based on B+ tree whose leaf nodes store all the routing rules and internal nodes store independent prefixes as search indexes. The longest prefix matching operation is performed in multi-way only once and matched in leaf nodes without backtracking. The MPT structure improves the lookup speed, but it is not efficient in memory usage because the independent prefixes are repeatedly stored in the B+ tree. To reduce the memory usage of this structure, we then construct the data structure as a Compressed Multi-Way Prefix Tree (CMPT) by proposing a prefixes compression technique based on the observation that prefixes are nested to each other. To compress prefixes, we design a nested prefix mask for several nested prefixes. Therefore, the leaf nodes of CMPT only store independent prefixes and masks instead of all the routing prefixes. To support the longest prefix matching on CMPT, we propose an IP lookup technique to calculate the longest matching prefix. To support dynamic IP lookup, the dynamic prefix insertion and deletion algorithms are proposed, which can effectively support the dynamic update of routing table. Because the data structure is not related to the length of IP address, it's also suitable for IPv6 addresses and can be easily extended to IPv6 networks.

The key contributions of this paper can be summarized as follows: (1) We propose a dynamic IP lookup algorithm based on B+ tree which improves the IP lookup speed effectively. (2) We propose a compact data structure to store prefixes which increases the space usage rate dramatically. (3) We propose a dynamic prefix update algorithm on B+ tree which can efficiently support the dynamic IP lookup. (4) The data structure we proposed can be easily extended to IPv6 networks.

The rest of this paper is organized as follows. Section 2 briefly summarizes the related work on dynamic IP lookup. Section 3 presents the formulations of routing rules and prefixes. The proposed CMPT structure is described in Sect. 4. Section 5 elaborates the IP address lookup algorithm with longest prefix matching. Section 6 introduces the dynamic update algorithms. Experimental results using our proposed method are reported in Sect. 7 and the paper is concluded in Sect. 8.

2. Related Work

Dynamic IP address lookup has received significant attentions in the past few years. Both hardware based and software based schemes are proposed [15]. However, although TCAMs perform high speed for IP lookup [12], they suffer from high cost and power consumption. The shortcomings are even more serious when extended to IPv6 networks. Therefore, software based solutions are still popular alternatives [1], [2], [4]–[8], [10], [11], [13], [16]–[19].

The trie is a tree-based data structure for IP address lookup applying linear search on prefix length [1], [2], [7], [8], [11]. The routing prefixes are stored in the nodes of trie. The level of a node storing prefix corresponds to the length of the prefix. The search operation proceeds to the right or left at each node according to the sequentially inspected bit of the IP address. The binary trie is widely used as IP address lookup data structure for its easiness to implement and good scalability for large forwarding tables. However, the binary tire is not a memory efficient data structure since there are lots of empty nodes not involving prefixes. Furthermore, since the shorter prefixes are stored higher than longer prefixes, shorter prefixes are compared earlier than longer prefixes when performing the longest prefix matching operation [1]. Lastly, the lookup speed of binary tire which is highly related to the prefix length would decrease dramatically when extends to IPv6 address.

Multi-way range search is a promising way for high-speed IP address lookup due to its good dynamics and scalability. Warkhede et al. firstly proposed a B-tree based data structure for IP address lookup named Multi-Way Range Tree (MRT) [17]. MRT performs the longest prefix matching operation in the time of $O(\log_m n)$, and dynamically inserts or deletes a rule in $O(m \log_m n)$. MRT is suitable for both ranges and prefixes. However, the endpoints of a range are stored repeatedly in internal nodes. A prefix is stored at most in $m-1$ nodes in each level. This property greatly increases the update complexity and memory usage. To resolve this problem, Lu et al. proposed a structure called Range in B-Tree (RIBT) [10]. RIBT stores each prefix only once at each level. In [10], the authors conducted experiments to compare RIBT with MRT and found out that the performance of the two algorithms were almost the same except that RIBT was more memory efficient than MRT.

Kim et al. proposed a dynamic structure called Collection of Red-Black Trees (CRBT) based on the endpoints of each range in [5]. CRBT performs IP lookup, dynamic prefix insertion and deletion in $O(\log n)$ time for a routing table with n rules. In [6], the authors proposed a dynamic IP lookup algorithm called Priority Search Tree (PST) which can find the longest matching prefix and dynamic update in $O(\log n)$. Lu et al. conducted experiments to compare the performance of CRBT and PST [18]. The experimental results show that CRBT outperforms PST in lookup speed. But PST performs much better than CRBT in update speed and memory usage. Lu and Sahni also proposed a dynamic algorithm called Binary Tree on Binary Tree (BOB) in [19]. For real forwarding tables, the time complexity of BOB and Prefix BOB (PBOB) are both $O(\log n)$ in prefix search, insertion and deletion. Furthermore, BOB and LMPBOB (Longest Matching Prefix BOB) outperform PST in prefix search, dynamic update and memory usage.

Chang et al. proposed a Multi-Way Most Specific Prefix Tree (MMSPT) based on B-tree [13]. MMSPT classifies the prefixes of a forwarding table into two categories: Specific Prefix and Cover Prefix. MMSPT stores each prefix only once in its nodes and performs the longest prefix matching operation in $O(\log_m n)$ time and performs dynamic prefix insertion and deletion in $O(m \log_m n)$ time. For a routing table with n prefixes, the memory usage of MMSPT is $O(n)$. However, for some prefixes, MMSPT needs to backtrack along the search path to find the longest matching prefix. Note that the probability of backtracking may increase and should not be ignored since the internet traffic is growing rapidly. More backtracking means more time cost. Furthermore, MMSPT still needs to store all the prefixes of routing tables. Since the size of forwarding tables are increasing exponentially, it's urgent to develop more memory efficient data structure for large forwarding tables.

3. Formulations

This section presents the formally definitions for the concepts of forwarding table, rule, prefix, interval, independent prefix and the relationships among prefixes. A forwarding table RT is made of n rules. Each rule r can be represented as $r : p \rightarrow NH$ where NH is the next hop. p is a set of IP addresses and can be denoted either a prefix or a nonempty nonnegative integer interval. A prefix is a binary string in the form of $\{0,1\}^k\{0\}^{w-k}$, where k is the prefix length and w is the length of IP address (e.g. for IPv4, $w = 32$, for IPv6, $w = 128$). The expression $\{0,1\}^k$ de-

Table 1 A forwarding table with 18 prefixes, $w = 6$.

#	prefix	interval	NH	#	prefix	interval	NH
p_1	0100**	[16,19]	N_1	p_{10}	000***	[0,7]	N_2
p_2	00010*	[4,5]	N_3	p_{11}	100***	[32,39]	N_1
p_3	11010*	[52,53]	N_4	p_{12}	10110*	[44,45]	N_4
p_4	00****	[0,15]	N_1	p_{13}	10****	[32,47]	N_3
p_5	01001*	[18,19]	N_2	p_{14}	101***	[40,47]	N_2
p_6	10010*	[36,37]	N_3	p_{15}	110101	[53,53]	N_1
p_7	11****	[48,63]	N_4	p_{16}	010***	[16,23]	N_3
p_8	01****	[16,31]	N_2	p_{17}	1101**	[52,55]	N_3
p_9	110***	[48,55]	N_2	p_{18}	1001**	[36,39]	N_1

notes a string with k 0 s or 1 s. '*' denotes a wildcard or the "don't care" bit which means '*' can be matched by both 0 and 1. The prefix $\{0,1\}^k\{*\}^{w-k}$, is equal to the interval $\left[\{0,1\}^k\{0\}^{w-k}, \{0,1\}^k\{1\}^{w-k}\right]$ which means $\{0,1\}^k\{0\}^{w-k}$ is the beginning IP address and $\{0,1\}^k\{1\}^{w-k}$ is the ending IP address that the interval can match. For a prefix p, we use $len(p)$, $start(p)$ and $finish(p)$ to denote the length, starting IP address and ending IP address of p respectively.

$$start(p) = \sum_{i=1}^{len(p)} b_i 2^{w-i} \tag{1}$$

$$finish(p) = \sum_{i=1}^{len(p)} b_i 2^{w-i} + \sum_{i=len(p)+1}^{w} 2^{w-i} \tag{2}$$

where b_i is the ith bit of prefix p. For example, when w=6, the prefix 110*** denotes the interval [110000, 110111] (which means the integer interval [48, 55]). And for this prefix, $len(p) = 3$, $start(p) = 48$, and $finish(p) = 55$.

Table 1 shows an example of a forwarding table. Using the concepts of prefix and interval, we define the relationships between two prefixes as follows.

Definition 1: For two prefixes p_1 and p_2, p_1 is disjoint with p_2 if and only if $p_1 \bigcap p_2 = \phi$. Otherwise, p_1 is related to p_2. As shown in Table 1, p_2 is related to p_4, but p_2 is disjoint with p_5.

Definition 2: For two disjointed prefixes $p_1$and p_2, define $p_1 < p_2$ if and only if $finish(p_1) < start(p_2)$. For example, in Table 1, $p_2 < p_1$, $p_8 < p_7$.

Definition 3: For two prefixes p_1 and p_2, define p_1 covers p_2 (denoted as $p_1 \supset p_2$), if and only if $start(p_1) \leq start(p_2)$ and $finish(p_1) \geq finish(p_2)$. For example, in Table 1, $p_7 \supset p_9$, $p_{13} \supset p_{18}$.

Definition 4: In the prefix set P of routing table RT, for any prefix $p \in P$, we say prefix p is an independent prefix if and only if there is no prefix q in the set P-$\{p\}$ satisfying $q \subset p$. Therefore, there is no prefix in P-$\{p\}$ can be covered by p. For example, in Table 1, p_2, p_6, and p_{15} are all independent prefixes.

Theorem 1: For two prefixes p_1 and p_2, if $p_1 \neq p_2$ and $len(p_1) = len(p_2)$, then $p_1 \bigcap p_2 = \phi$.

Proof: Let the first l bits of p_1 and p_2 are the same, p_1 and

p_2 are different from the $l+1$ bit. Since each bit is either o or 1, we may assume that the $l+1$ bit of p_1 is 0 and the $l+1$ bit of p_2 is 1. Therefore

$$finish(p_1) = \sum_{i=1}^{l} b_{i.1}2^{w-i} + 0 \cdot 2^{w-(l+1)} + \sum_{i=l+2}^{len(p_1)} b_{i.1}2^{w-i} + \sum_{i=len(p_1)}^{w} 2^{w-i}$$

$$start(p_2) = \sum_{i=1}^{l} b_{i.2}2^{w-i} + 1 \cdot 2^{w-(l+1)} + \sum_{i=l+2}^{len(p_2)} b_{i.2}2^{w-i}$$

Where $b_{i.1}$ is ith bit of p_1, $b_{i.2}$ is the ith bit of p_2. Since

$$\begin{aligned}\sum_{i=l+2}^{len(p_1)} b_{i.1}2^{w-i} + \sum_{i=len(p_1)}^{w} 2^{w-i} &< \sum_{i=l+2}^{len(p_1)} 2^{w-i} + \sum_{i=len(p_1)}^{w} 2^{w-i} \\ &= \sum_{i=l+2}^{w} 2^{w-i} \\ &< 2^{w-(l+1)}\end{aligned}$$

Therefore, $finish(p_1) < start(p_2)$. According to definition 2, we know p_1 is disjoint with p_2. And according to definition 1, $p_1 \bigcap p_2 = \phi$.

Definition 5: For two prefixes p_1 and p_2, $l_1 = len(p_1)$, $l_2 = len(p_2)$, and $l_1 \leq l_2$, if the first l_1 bits of p_2 are the same as p_1, we define p_1 is a sub string of p_2. As shown in Table 1, p_9 is a sub string of p_{15}.

Theorem 2: For two prefixes p_1 and p_2, if $len(p_1) < len(p_2)$ and p_1 is not a sub string of p_2, then $p_1 \bigcap p_2 = \phi$.

Proof: Since $len(p_1) < len(p_2)$ and p_1 is not a sub string of p_2, the first l bits of p_1 are different from the first l bits of p_2, according to Theorem 1, p_1 is disjoint with the sub string of the first l bits of p_2. Therefore, the proof process is similar to the proof of Theorem 1.

Theorem 3: For two prefixes p_1 and p_2, if $len(p_1) < len(p_2)$, and p_1 is a sub string of p_2, then $p_1 \bigcap p_2 = p_2$.

Proof: p_1 is a sub string of p_2, therefore, the first l bits of p_2 is the same as p_1, so $l = len(p_1)$, and $b_{i,1} = b_{i.2}$, where $1 \leq i \leq l$. Therefore,

$$start(p_1) = \sum_{i=1}^{l} b_{i,1}2^{w-i}$$

$$finish(p_1) = \sum_{i=1}^{l} b_{i,1}2^{w-i} + \sum_{i=l+1}^{w} 2^{w-i}$$

$$\begin{aligned}start(p_2) &= \sum_{i=1}^{len(p_2)} b_{i,2}2^{w-i} \\ &= \sum_{i=1}^{l} b_{i,2}2^{w-i} + \sum_{i=l+1}^{len(p_2)} b_{i,2}2^{w-i} \\ &= \sum_{i=1}^{l} b_{i,1}2^{w-i} + \sum_{i=l+1}^{len(p_2)} b_{i,2}2^{w-i} \\ &\geq \sum_{i=1}^{l} b_{i,1}2^{w-i} = start(p_1)\end{aligned}$$

$$\begin{aligned}finish(p_2) &= \sum_{i=1}^{l} b_{i,2}2^{w-i} + \sum_{i=l+1}^{len(p_2)} b_{i,2}2^{w-i} + \sum_{i=len(p_2)+1}^{w} 2^{w-i} \\ &= \sum_{i=1}^{l} b_{i,1}2^{w-i} + \sum_{i=l+1}^{len(p_2)} b_{i,2}2^{w-i} + \sum_{i=len(p_2)+1}^{w} 2^{w-i} \\ &\leq \sum_{i=1}^{l} b_{i,1}2^{w-i} + \sum_{i=l+1}^{w)} 2^{w-i} \\ &= finish(p_1)\end{aligned}$$

According to definition 3, $p_1 \supset p_2$, so $p_1 \bigcap p_2 = p_2$.

4. Proposed Data Structure

This section presents the proposed data structure which we call Compressed Multi-Way Prefix Tree (CMPT). The objective of CMPT is to design a structure satisfying: (1) the search operation on the structure should be able to hit at leaf node without backtracking. (2) The structure should be able to support dynamic prefix update. (3) The structure should be not related to address length. (4) The structure should be able to store forwarding rules compressively.

We first present our MPT structure based on B+ tree, and then introduce the compression technique for prefixes on the leaf nodes of MPT to construct CMPT.

4.1 Multi-Way Prefix Tree

MPT is an m-way search tree built with B+ tree in the order of m. For a given prefix set P of a forwarding table, we convert each prefix to a range, and then construct P as a MPT based on B+ tree. Like the traditional B+ tree, our MPT is defined as follows.

1. Each node x contains the following information:
   - $n(x)$, the number of keys stored in node x;
   - $leaf(x)$, defined in bool. If x is a leaf node, it is set to TRUE. If x is an internal node, it is FALSE.
   - The $n(x)$ keys $key_i(x)$, where $1 \leq i \leq n(x)$. If $leaf(x)$ is FALSE, the $n(x)$ keys follow

     $$key_1(x) < key_2(x) < \cdots < key_{n(x)}(x)$$

     If $leaf(x)$ is TRUE, the $n(x)$ keys follow

     $$key_1(x) \subset key_2(x) \subset \cdots \subset key_{n(x)}(x)$$
2. If x is an internal node, x contains $n(x)$ pointers pointing to its $n(x)$ children. A leaf node do not have child.
3. The first key of each node (except the root) is repeatedly stored in its parent node as the search index.
4. All the leaf nodes are in the same layer, the depth is h.
5. The number of keys stored in internal node can be ex-

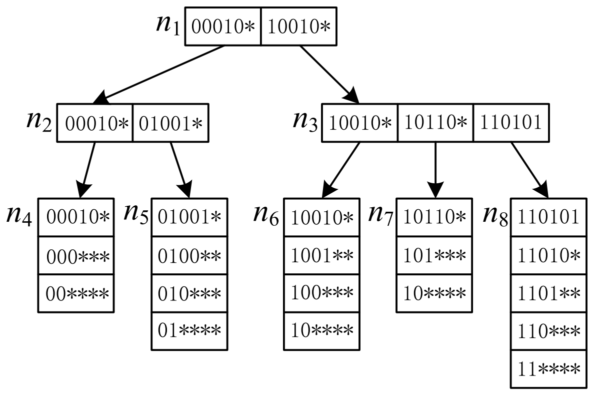


Fig. 1 An example of MPT tree constructed from Table 1.

pressed with the integer m as follows:

- There are at least $m/2$ keys stored in each internal node (except the root), and at least $m/2$ children pointed by $m/2$ pointers. If the tree is non-empty, there is at least one key stored in the root.
- There are at most m keys stored in each internal node and m pointers pointing to the conrresponding children. A internal node is full if there are m keys stored in it.

6. There are at most w (w is the length of IP address) keys stored in each leaf node, and at least one key.

Figure 1 shows an example of MPT tree constructed from Table 1, where m is 4. As illustrated in Fig. 1, the keys in n_1, n_2 and n_3 are all independent prefixes and stored in increasing order. In the leaf nodes of n_4, n_5, n_6, n_7 and n_8, only the first key in each node is independent prefix, the other keys in each node all cover the keys above them. For example, the first key in n_4 is 00010* which is an independent prefix. The other two keys are 000*** and 00****. The second key 000*** covers the first key 00010*, and the third key 00**** covers both 00010* and 000***. The internal node n_2 stores two keys 00010* and 01001*. Therefore, n_2 contains two pointers pointing to its children n_4 and n_5.

According to the definition of MPT, the routing prefixes are all stored in leaf nodes sequentially in coverage increasing order. As search indexes, independent prefixes are stored repeatedly in internal nodes. This property ensures the longest prefix matching can be hit at leaf node for every IP address. However, the repeatedly stored independent prefixes increase the memory usage. To reduce the memory usage of MPT, we develop a nested prefixes compression technique to construct a Compressed Multi-way Prefix Tree (CMPT).

4.2 Prefixes Compression

We present a prefixes compression technique in this subsection to resolve the memory usage problem of MPT. The technique is mainly based on the coverage relationship of prefixes. According to the definition of MPT, the prefixes in each leaf node of MPT have the following theorems.

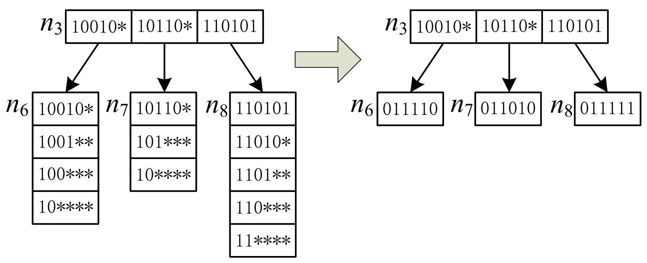


Fig. 2 An example of generating PNMs for leaf nodes.

Theorem 4: For any two prefixes p_1 and p_2 in the leaf node x of MPT, assume their lengths are l_1 and l_2, if $p_1 \neq p_2$, then $l_1 \neq l_2$.

Proof: From the definition of MPT, we know that the prefixes in leaf nodes are stored in increasing order of coverage. Since $p_1 \neq p_2$, suppose p_1 covers p_2 in leaf node x, that's to say, $p_1 \supset p_2$, according to Theorem 3, $p_1 \bigcap p_2 = p_2$ and $l_1 \neq l_2$.

Theorem 5: For any two prefixes p_1 and p_2 in the leaf node x of MPT, assume their lengths are l_1 and l_2, if p_1 is stored before p_2 in x, which means $p_1 \subset p_2$, then $l_1 > l_2$.

Theorem 5 can be proved in the similar way as the proof of Theorem 4.

Based on Theorem 4 and Theorem 5, we generate a Prefixes Nested Mask (PNM) to replace several prefixes stored in each leaf node to reduce the memory usage. The PNM of leaf node x, denoted as $Mask(x)$, can be generated as the following definition.

Definition 6: The $Mask(x)$ of leaf node x is a prefix with the length w (for IPv4, $w = 32$, for IPv6, $w = 128$). All bits of $Mask(x)$ are originally set to 0 s. When the key set of node x includes a prefix with length l, the lth bit of $Mask(x)$ is set to 1. For example, in Fig. 2, $Mask(n_6) = 011110$, $Mask(n_7) = 011010$.

After we generate the PNMs, each leaf node only stores one PNM prefix. For the convenience of IP lookup, we combine the PNMs in leaf nodes with the corresponding nodes of the last internal layer. After combination, we build our Compressed Multi-Way Prefix Tree (CMPT). Therefore, each leaf node of CMPT stores $n(x)$ keys and PNMs.

Figure 2 illustrates the process of generating PNMs for n_6, n_7 and n_8. The node n_6 stores four prefixes which are 10010*, 1001**, 100*** and 10****. According to definition 6, 011110 would be the PNM for n_6. Therefore, the PNM 011110 is used to replace the four prefixes in n_6.

IP lookup structure has to store the next hop for each routing prefix. Note that we can't use pointers pointing to each next hop for each prefix since CMPT stores the independent prefixes and PNMs instead of all the routing prefixes. Therefore, we design a next hop table (NH table) for each key in leaf nodes to ensure the correct IP lookup result. The next hops stored in NH table are in the same order

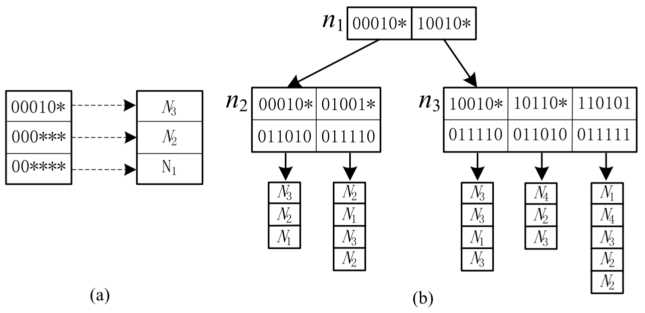


Fig. 3 A CMPT constructed from Table 1.

Algorithm 1 Longest Prefix Matching in leaf node

```
$index \leftarrow 0$
$lmp \leftarrow$ the smallest value of $key_i(x)$
$pl \leftarrow Mask_i(x)$
while ($pl$) do
    $m \leftarrow \sim((\sim pl)\&(pl-1)$
    if ($m\&lmp == m\&p$) then
        return $NH[index]$
    end if
    $pl \leftarrow pl\&(pl-1)$
    $index++$
end while
return default NH
```

of corresponding prefixes stored in the leaf nodes of MPT. Figure 3 (a) shows the mapping structure of NH table. In the leaf node, there are three prefixes, 00010*, 000*** and 00****, stored in the decreasing order according to their length. The next hops of each prefix are stored in the same order in NH table. Figure 3 (b) shows the CMPT structure constructed from Table 1. As shown in Fig. 3 (b), there are only 3 nodes in CMPT after compression and the 26 prefixes stored in MPT are reduced to 12 prefixes in CMPT.

5. Longest Prefix Matching

This section presents the IP address lookup algorithm with longest prefix matching on CMPT. The search process for an IP address p begins at the root of CMPT. For each internal node x that p encounters, p makes a comparison with each element in the prefix set $P(x) = \{key_1(x), key_2(x) \dots key_{n(x)}(x)\}$ of x and trances the path downward to a leaf node. In the comparison process, firstly, we make an XOR operation between p and $key_i(x)$. Then, the search process trance downward to the sub tree that $key_i(x)$ points to, where the result of XOR between p and $key_i(x)$ addresses the maximum number of sequential 0 s starting from the left. If p encounters a leaf node x, the longest matching prefix for p is calculated using $key_i(x)$ and the corresponding $Mask_i(x)$, where the result of XOR between p and $key_i(x)$ addresses the maximum number of sequential 0 s from the left. This technique addresses the correct longest prefix matching in leaf node.

To calculate the longest matching prefix by $key_i(x)$ and $Mask_i(x)$, as shown in Algorithm 1, firstly, let *lmp* to be the minimal value of the range that the prefix $key_i(x)$ involves. Then, we construct a Prefix Matching Model (PMM), denoted as *Model*. *Model* is calculated as the Eq. (3). Note that the first l bits of *Model* are resulted as 1 s where l is the length of the longest matching prefix. All the other bits are resulted as 0 s.

$$Model = \sim((\sim Mask_i(x))\&(Mask_i(x)-1) \tag{3}$$

As shown in Algorithm 1, if the result of $Model\&lmp$ is equal to the result of $Model\&p$, lp is the longest matching prefix and the algorithm returns the corresponding next hop. Otherwise, $Mask_i(x)$ is updated using Eq. (4), and then update Model using Eq. (3). The updated Model drops the 1 at

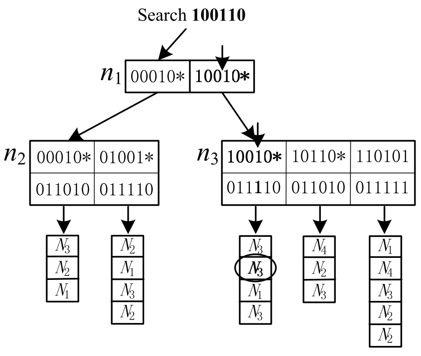


Fig. 4 Longest prefix matching on CMPT.

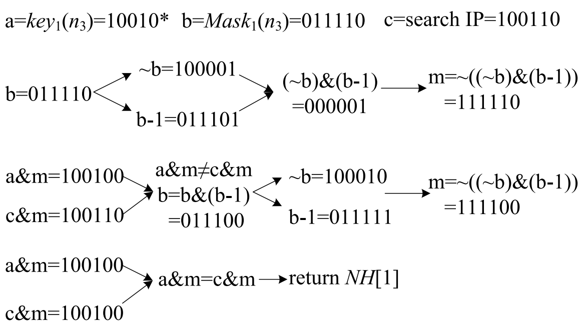


Fig. 5 An example of calculating the longest prefix matching.

its last bit. So the producer becomes the second longest prefix matching. Therefore, we continue to compare the value of $Model\&lmp$ and $Model\&p$. By this way, we can always find the longest matching prefix for an IP address.

$$Mask_i(x) = Mask_i(x)\&(Mask_i(x)-1) \tag{4}$$

Figure 4 depicts an example of longest prefix matching process on a CMPT. When searching the address 100110, we firstly search in the root n_1. The XOR result of 100110 and 00010* is 10001*, the XOR result of 100110 and 10010* is 00001*. 00001* begins with four 0 s and 10001* begins with 1. Therefore, we continue to search in the child that 10010* points to. The same operation is performed in n_3. Therefore, the longest matching prefix is calculated by the prefix 10010* and its Mask 011110. The calculation process is depicted in Fig. 5. For simple, in Fig. 5, we use a, b and c to denote $key_i(x)$, $Mask_i(x)$ and the IP address for search respectively.

Assume there are u independent rules in a routing table with totally n rules. The search speed highly depends on the number of memory access since memory access is the most time-consuming operation in search process. In CMPT, IP lookup operation needs to access memory at one node of each layer. Therefore, the search speed is highly based on the height of CMPT.

Theorem 6: For a CMPT tree constructed from a forwarding table with u independent prefixes in totally n prefixes, in the worst case, the height h satisfies

$$h = \log_{m/2} u \tag{5}$$

Proof: There is at least one key in the root and $m/2$ keys in other nodes of CMPT. At the height of 1, there are at least $m/2$ nodes, at the height of 2, there are at least $(m/2)^2$ nodes, at the height of h, there are at least $(m/2)^h$ nodes. Therefore, in the worst case, the number of independent prefixes stored in leaf nodes satisfying $u = (m/2)^h$. Solve the equation would be $h = \log_{m/2} u$.

Search Complexity: CMPT is an m-way search tree built from routing table with u independent prefixes in n prefixes. Known from Theorem 6, CMPT height in the worst case is $\log_{m/2} u$. To ensure the least utilization, every node except the root in CMPT stores at least $m/2$ keys. We use binary search in each node since m is not very large, the complexity of binary search is $O(\log_2 m)$ and the total search layer is $\log_{m/2} u$. Since the worst case matching times to perform the longest prefix matching in a leaf node is a constant w, the worst case search time would be $O(\log_m u)$.

6. Dynamic Update

Since the forwarding tables of core routers can be updated hundreds of times per second. To stabilize the routing, the data structure for IP address lookup has to support a thousand times of dynamic prefix update every second. This section presents the dynamic update algorithms for our CMPT structure.

6.1 Dynamic Prefix Insert

Since CMPT stores independent prefixes and PLMs instead of all the routing prefixes, to insert a prefix p into CMPT, the prefix can be inserted as the following two forms.

1. If p is not an independent key, p should be inserted into the $Mask_i(x)$ of leaf node x.
2. If p is an independent key, p should be inserted as a new key of CMPT.

Note that whether p is an independent prefix can be decided in the process for searching p on CMPT. For any keys encountered in the search path, if p covers a key, p is not an independent prefix. Otherwise, p is an independent prefix.

For the first case, p is not an independent prefix. Firstly,

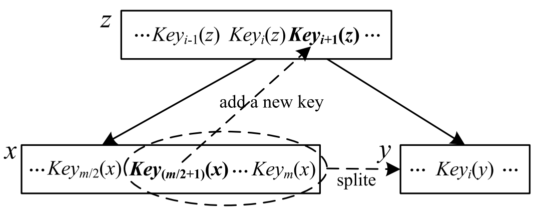

z
$\cdots Key_{i-1}(z)\ Key_i(z)\ \boldsymbol{Key_{i+1}(z)} \cdots$
add a new key
x
$\cdots Key_{m/2}(x)(\boldsymbol{Key_{(m/2+1)}(x)} \cdots Key_m(x)$
y
splite
$\cdots\ Key_i(y)\ \cdots$

Fig. 6 Node splitting.

we locate to the leaf node x along the search path where p should be inserted. Then in the node x, we compare p with each $key_i(x)$. If p covers $key_i(x)$, p should be inserted into $Mask_i(x)$ which is the PNM of $key_i(x)$. Finally, insert p into $Mask_i(x)$ is to set a bit of $Mask_i(x)$ from 0 to 1 according to the length of p. Note that there may be one or more keys to be covered by p, to ensure the efficiency of longest prefix matching, we insert p to each $Mask_i(x)$ that the corresponding $key_i(x)$ is covered by p. Since the presentation of $Mask_i(x)$ is a binary string of length w and inserting a prefix to $Mask_i(x)$ doesn't change its length. Therefore, inserting p to any other $Mask_i(x)$ doesn't increase the memory usage.

For the second case, p is an independent prefix. When we insert p into CMPT, some other independent keys may cover p. Therefore, if p is disjoint with all the other keys on the search path, p should be inserted as a new key of node x. If p is covered by another key $key_i(x)$, we replace each $key_i(x)$ with p in the search path and update the corresponding $Mask_i(x)$.

When p is inserted as a new key of node x, $n(x)$ is counted and may be larger than m. If it does, we need to split x into two nodes (shown in Fig. 6) as the following steps:

Step1: Split x into two nodes x and y at the place of the $(m/2)$th key. Each node stores at least $m/2$ keys.

Step2: Insert $key_1(y)$ into the parent node of x (denoted as z) and save it as a new key of z and make a pointer pointing to y.

Step3: Check if $n(z)$ is larger than m (z is the parent node of x), the node z should be spitted in the same way.

Step4: Repeat this process till the key number of a node is less than m.

6.2 Dynamic Prefix Delete

This sub section presents the dynamic prefix deletion technique on CMPT.

For a forwarding table, all the routing prefixes are stored as keys or PNMs in leaf nodes of CMPT. If the prefix p is involved in PNM of leaf node x, to delete p is to change the corresponding bit of $Mask_i(x)$ from 1 to 0 and delete the corresponding next hop in the NH table. If p is stored as a key of leaf node x, p is also stored in other internal nodes. So we have to delete every p stored in internal nodes along the search path. Note that deleting a key from node x could change the value of $n(x)$. If $n(x)$ is less than $m/2$ after deleting p, CMPT has to be adjusted based on the neighbour node of x as the following two cases.

1. If the number of keys in *neighbour*(x) is larger than $m/2$, x borrows a key from *neighbour*(x). Therefore, the number of keys in x and *neighbour*(x) are not less than $m/2$.
2. If the number of keys in *neighbour*(x) is equal to $m/2$, we merge x and *neighbour*(x) to a new node and delete the key in their parent that points to *neighbour*(x). Note that this process may cause the number of keys in their parent less than $m/2$, the technique should be repeated in the parent node till CMPT is adjusted to correct.

For the first case, assume that the nearest right neighbour of x is y and $n(y)$ is larger than $m/2$. As shown in Fig. 7, $key_1(y)$ is the first key of y, $key_i(z)$ is a key of node z (z is the parent of x and y). $Child_i(z) = x$, $Child_{i+1}(z) = y$. x borrows a key from y is shown as the following steps:

Step1: Make a copy of $key_1(y)$ and insert it after $key_{(m/2)-1}(x)$ in x to be the $key_{(m/2)}(x)$ of x. Move the corresponding $Mask_1(y)$ and NH table to x along with $key_{(m/2)}(x)$.

Step2: Replace $key_{i+1}(z)$ in z with $key_2(y)$ of y.

Step3: Delete $key_1(y)$ in y as well as $Mask_1(y)$ and NH table. Then move all the other keys of y one step left.

For the second case, assume that the nearest right neighbour of x is y and $n(y)$ is equal to $m/2$. As shown in Fig. 8, $key_i(z)$ and $key_{i+1}(z)$ are the keys of node z (z is the parent of x and y). $Child_i(z) = x$, $Child_{i+1}(z) = y$. x and y will be merged as the following steps:

Step1: Make a copy of all the $key_i(y)$ $(1 \leq i \leq n(y))$ and insert them at the end of x. If x and y are leaf nodes, copy the corresponding $Mask_i(y)$ and NH table to insert them into x.

Step2: Delete the node y;

Step3: Delete the key $key_{i+1}(z)$ and the corresponding pointer that points to y. Change $n(z)$ to $n(z) - 1$.

Step4: Inspect the value of $n(z)$. If $n(z)$ is less than $m/2$, the node z should be adjusted by borrowing a key or merging z with its neighbour.

Step5: Repeat this process till the key number of a node is not less than $m/2$.

To insert a prefix p in CMPT, firstly a search operation

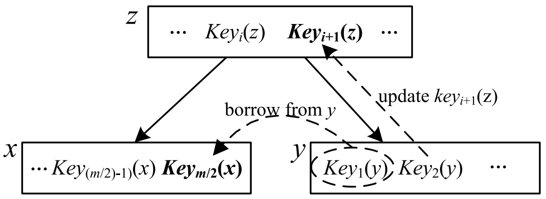

z
··· Key_i(z) Key_{i+1}(z) ···
update key_{i+1}(z)
borrow from y
x
··· Key_{(m/2)-1}(x) Key_{m/2}(x)
y
Key_1(y) Key_2(y) ···

Fig. 7 Moving a key form node y to node x.

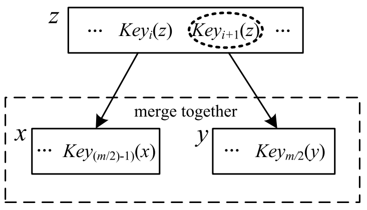


Fig. 8 Merging node x and y.

for p is performed. Then, p has to be inserted into every PNMs of the keys that p covers. Therefore, the insert complexity is $O(m \log_m u)$. When deleting a prefix, since p may be included in more than one PNMs, Therefore, the delete complexity is $O(m \log_m u)$.

7. Experimental Results

7.1 Evaluation Setup

We conduct our experiments on a PC with 2.0 GHz Inter(R) Core(TM)2 CPU and 2.0 GB memory running Windows XP Sp3. The program is written in C++ and simulated in Visual Studio 2008.

We obtain 5 real life routing tables from [14]. In [14], the website records many kinds of statistical information of the BGP routing table of AS6447. We obtain 5 routing tables in different time from 2007 to 2011, denoted as Router1 to Router5 (shown in Table 2). Furthermore, we generate IP traces for every routing table using the technique of [20].

We conduct experiments to contrast our CMPT with MRT of [17], PBOB of [18], POBT of [19] and MMSPT of [13] in the metric of IP lookup speed, dynamic prefix insert and delete speed, and memory usage. MMSPT, MRT and PIBT are all built on m-way search tree (m is 32 in [13]). PBOB is built on a red-black tree and our CMPT is built on an m-way B+ tree. Experiments in [19] illustrated that the best value of m is 32. Since B+ tree and B-Tree are similar in their structure, m is also evaluated as 32 in our CMPT.

7.2 IP Lookup Speed

To evaluate the IP lookup speed, we generate 1000000 IP traces for each routing table from Router1 to Router5 using the technique of [20]. We record the IP lookup time of the 1000000 IP traces for the five different algorithms of MMSPT, MRT, PIBT, PBOB and our CMPT. The recorded time dividing 1000000 would be the average IP lookup time for every IP address. Using this method, we repeat 10 times on each routing table with the 5 algorithms respectively to record the average IP lookup time of each algorithm. Note that it is difficult to record the search time of single lookup [10], we record the search time of 1000000 IP traces to get the average search time [13].

Table 3 and Fig. 9 show the experimental results of IP lookup speed for each algorithm. Note that PBOB requires much more time to perform IP lookup than any other schemes since PBOB is performed as a binary search which is more time consuming than multi-way search. For a routing table with n routing rules, PIBT and MRT stores $2n$ keys while MMSPT stores n keys. Therefore, the IP lookup speed

Table 2 Five real routing tables for experiments.

Name	Router1	Router2	Router3	Router4	Router5
Source	AS6447	AS6447	AS6447	AS6447	AS6447
Time	5/1/2007	5/1/2008	5/1/2009	5/1/2010	5/1/2011
Number	179368	234963	278147	326479	367158

Table 3 IP lookup time of each scheme (μ sec).

Algorithm	Router1	Router2	Router3	Router4	Router5
CMPT	0.27	0.28	0.29	0.3	0.31
MMSPT	0.36	0.39	0.41	0.44	0.48
PIBT	0.41	0.43	0.47	0.51	0.54
MRT	0.46	0.48	0.51	0.53	0.56
PBOB	0.75	0.96	1.1	1.3	1.53

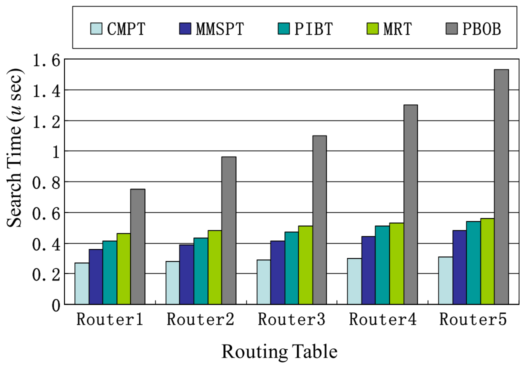


Fig. 9 Figure of IP lookup time for each scheme.

Table 4 Search complexity on CMPT.

	Router1	Router2	Router3	Router4	Router5
n	179368	234963	278147	326479	367158
u	85796	108748	135284	168625	194739
h	4	4	4	4	4

Table 5 Memory usage of each scheme (KB).

Algorithm	Router1	Router2	Router3	Router4	Router5
CMPT	4375	4962	5617	6374	6983
MMSPT	3276	4667	5869	7198	8269
PIBT	7053	9754	13308	16828	19676
MRT	6587	9161	12752	16198	18876
PBOB	3035	4672	5637	6806	7748

is faster in MMSPT than in PIBT and MRT. MMSPT needs to backtrack for some IP addresses while CMPT hits the longest matching prefix at leaf node every time. Furthermore, CMPT only stores independent prefixes in the search path and therefore reduces the times for comparing. Therefore, CMPT is more time efficient than MMSPT.

To further evaluate the search complexity of CMPT, we record the height of each CMPT structure built from Router1 to Router5. The results are shown in Table 4. As an example, the recorded search depth of Router5 is 4. According to Theorem 6, the worst case search height for Router5 on CMPT is $h = \log_{m/2} u = \log_{16} 194739 = 4.39$. The experiment result coincides with Theorem 6.

7.3 Memory Usage

To evaluate the memory usage, we construct CMPT, MRT, PBOB, POBT and MMSPT using the five real life routing tables from Router1 to Router5 respectively. The results are shown in Table 5 and Fig. 10. As illustrated, MMSPT and PBOB use about 50% less memory than PIBT and MRT.

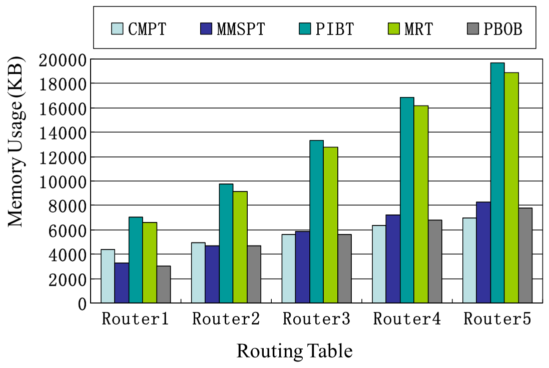


Fig. 10 Memory usage of each scheme of five routing tables.

Since for each prefix of a forwarding table, PIBT and MRT store the prefix in duplicate in constant nodes, while MMSPT and PBOB store the prefix only in one node.Therefore, MMSPT and PBOB are more memory efficient than PIBT and MRT.

For Router1 and Router2, CMPT uses 33.5% and 6% more memory than MMSPT uses. However, For Router3, Router4 and Router5, CMPT uses 4%, 11.4% and 15.5% less memory than MMSPT uses. Therefore, CMPT is more memory efficient for today's large forwarding tables.Note that although CMPT stores independent prefixes on internal nodes repeatedly, the nested prefixes are stored compressively on its leaf nodes. Furthermore, CMPT reduced a layer by combining leaf nodes with their parent nodes. Therefore, CMPT is more memory consuming when the forwarding table is not too large, but it is more memory efficient than MMSPT with the increase of forwarding table size. The prefixes compression algorithm leads to the result that the memory utilization rate of CMPT is higher than other algorithms.

7.4 Insert and Delete Speed

To measure the prefix insertion and deletion speed, we first construct each scheme of MMSPT, PIBT, MRT, POBO and our CMPT using the five real life routing tables from Router1 to Router5. For each scheme, we randomly delete 10% prefixes and record the deletion time. Then the total elapsed time is divided by the number of these 10% prefixes to get the average deletion time for single prefix. Next, the deleted 10% prefixes are inserted back into the corresponding structure consecutively. The total elapsed time is divided by the number of these 10% prefixes to get the average insertion time for each prefix.

Table 6 shows the measured insertion and deletion times. These times are histogrammed in Fig. 11. It can be seen that the update time of PIBT and MRT are much more than that of other schemes, since PIBT and MRT partition the range of a prefix into two endpoints which are inserted or deleted respectively, and then insert or delete the prefix itself. While MMSPT, PBOB and our CMPT only insert or delete a prefix itself. Therefore, PIBT and MRT are more

Table 6 Dynamic prefix update time of each scheme (μ sec).

	Router1		Router2		Router3		Router4		Router5	
	Insert	Delete	Insert	Delete	Insert	Delete	Insert	Delete	Insert	Delete
CMPT	0.55	0.61	0.55	0.62	0.56	0.62	0.57	0.63	0.56	0.62
MMSPT	0.61	0.75	0.63	0.75	0.63	0.78	0.64	0.79	0.65	0.81
PIBT	1.81	2.14	1.82	2.14	1.82	2.15	1.83	2.14	1.82	2.15
MRT	1.24	2.25	1.25	2.25	1.26	2.26	1.27	2.25	1.26	2.24
PBOB	0.82	0.84	0.83	0.85	0.84	0.86	0.82	0.84	0.83	0.85

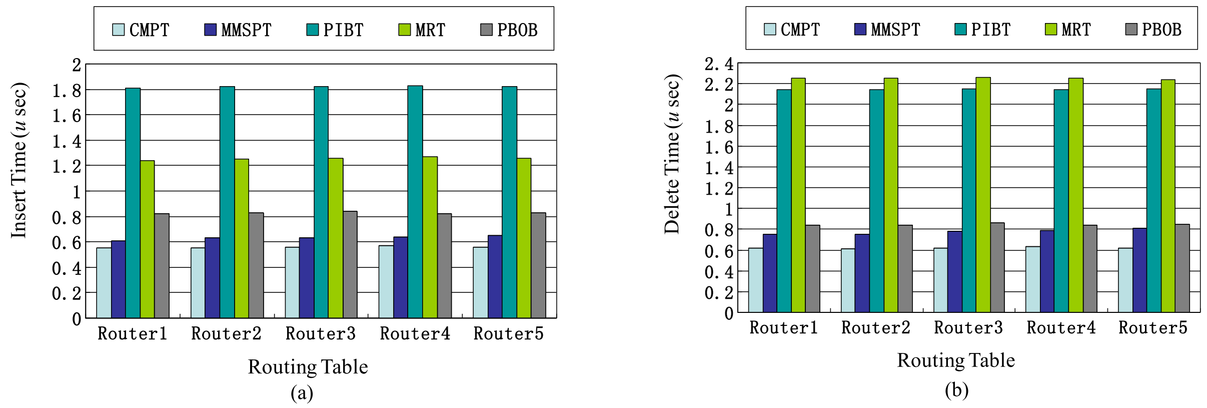


Fig. 11 Figure of dynamic prefix update speed for each scheme.

time consuming than other algorithms.

PBOB is built on a binary tree which may be unbalanced after insert and delete prefixes many times. The additional operation for balancing the binary tree is regarded as reducing the update speed of PBOB. Although MMSPT and CMPT have to split or merge nodes, the probabilities of splitting and merging are very small since m is large enough. Therefore, the updating speeds of MMSPT and CMPT are faster than PBOB. As shown in Fig. 11, the insert speed of CMPT is about 13% faster than MMSPT, and the delete speed is about 18% faster than MMSPT. Since MMSPT needs backtracking search in the lookup process while CMPT does not. Therefore, MMSPT is more time consuming than CMPT in prefix update.

8. Conclusion

In this paper, the authors propose a dynamic and scalable IP address lookup algorithm based on B+ tree. The algorithm builds the forwarding table as a compressed multi-way prefix tree named CMPT to perform IP address lookup. Using CMPT, the lookup speed is improved by finishing every longest prefix matching operation at leaf node without backtracking. The authors also propose a prefixes compression technique to store some of instead of all the routing prefixes on CMPT to improve the memory utilization rate. Based on the CMPT structure, the authors make further efforts to propose dynamic prefix insert and delete algorithm to support dynamic prefix update. Exhaustive experiments on real life forwarding tables show promising gains of the proposed algorithm in IP address lookup speed and dynamic prefixes update speed than the existing B- tree algorithms. Furthermore, since CMPT is unrelated to the length of IP address, it is also suitable for IPv6 networks.

There is still room to improve IP lookup speed. Future work will investigate new approach based on our proposed method. Furthermore, it will be very practical to consider parallel algorithm for independent prefix to further improve search speed.

Acknowledgements

This work was supported in part by NSFC under grant No. 60973031, and by the Postgraduate Research and Innovation Project of Hunan Province of China under grant No.CX 2011B138.

References

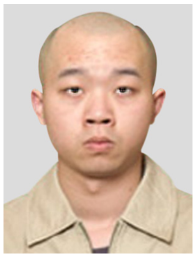
[1] H. Lim, C. Yim, and E.E. Swartzlander, "Priority tries for IP address lookup," IEEE Trans. Comput., vol.59, no.6, pp.784–794, 2010.
[2] K. Huang, G. Xie, Y. Li, and A.X. Liu, "Offset addressing approach to memory-efficient IP address lookup," Proc. IEEE INFOCOM, 2011.
[3] Y. Rekhter and T. Li, "An architecture for IP address allocation with CIDR," RFC 1518.
[4] H. Song, M. Kodialam, F. Hao, and T.V. Lakshman, "Scalable IP lookups using shape graphs," Proc. IEEE ICNP, 2009.
[5] S. Sahni and K.S. Kim., "An $O(\log n)$ dynamic router-table design," IEEE Trans. Comput., vol.53, no.3, pp.351–363, 2004.
[6] H. Lu and S. Sahni, "$O(\log n)$ dynamic router-tables for prefixes and ranges," IEEE Trans. Comput., vol.53, no.10, pp.1217–1230, 2004.
[7] S. Nilsson and G. Karlsson, "IP address lookup using LC-tries," IEEE J. Sel. Areas Commun., vol.17, no.6, pp.1083–1092, 1999.
[8] M. Bando and H.J. Chao, "FlashTrie: Hash-based prefix-compressed trie for IP route lookup beyond 100 Gbps," Proc. IEEE INFOCOM, 2010.

[9] M. Behdadfar, H. Saidi, M. Hashemi, A. Chiasian, and H. Alaei, "IP lookup using the novel idea of scalar prefix search with fast table updates," IEICE Trans. Inf. & Syst., vol.E93-D, no.11, pp.2932–2943, Nov. 2010.
[10] H. Lu and S. Sahni, "A B-tree dynamic router-table design," IEEE Trans. Comput., vol.54, no.7, pp.813–824, 2005.
[11] S.Y. Hsieh, Y.L. Huang, and Y.C. Yang, "Multiprefix trie: A new data structure for designing dynamic router-tables," IEEE Trans. Comput., vol.60, no.5, pp.693–706, 2011.
[12] W. Lu and S. Sahni, "Low power TCAMs for very large forwarding tables," IEEE Trans. Netw., vol.18, no.3, pp.948–959, 2010.
[13] Y.K. Chang and Y.C. Lin, "A fast and memory efficient dynamic IP lookup algorithm based on B-Tree," Proc. IEEE AINA, 2009.
[14] Huston G, "BGP routing table analysis reports," http://bgp.potaroo.net/, 2011.
[15] H. Lim, "Survey and proposal on packet classification algorithms," Proc. IEEE HPSR, 2010.
[16] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated software router," Proc. ACM SIGCOMM, 2010.
[17] P. Warkhede, S. Suri, and G. Varghese, "Multiway range trees: Scalable IP lookup with fast updates," Int. J. Comp. and Telecom. Networking, vol.44, no.3, pp.289–303, 2004.
[18] H. Lu, K. Kim, and S. Sahni, "Prefix and interval-partitioned dynamic IP router-tables," IEEE Trans. Comput., vol.54, no.5, pp.545–557, 2005.
[19] H. Lu and S. Sahni, "Enhanced interval trees for dynamic IP router-tables," IEEE Trans. Comput., vol.53, no.12, pp.1615–1628, 2004.
[20] E.T. David and S.T. Jonathan, "ClassBench: A packet classification benchmark" IEEE/ACM Trans. Netw., vol.15, no.3, pp.499–511, 2007.

Gang Wang was born in 1985, received his B.E. degree in information security and M.E. degree in Computer Science from Hunan University, China, in 2007 and 2010 respectively. Since 2009, he has been a Ph.D. candidate in College of Information Science and Engineering of Hunan University. His research interests include high performance router design, parallel computing architecture, and high speed deep packet inspection algorithm.

Yaping Lin received the B.S. degree in Computer Application from Hunan University, China, in 1982, the M.S. degree in Computer Application from National University of Defense Technology, China in 1985. He received the Ph.D. degree in Control Theory and Application from Hunan University in 2000. He has been a professor and Ph.D. supervisor in Hunan University since 1996. During 2004–2005, he worked as a visiting researcher at the University of Texas at Arlington. His research interests include machine learning, network security and wireless sensor networks.

Rui Li received his B.E. degree from Xi'an University of Architecture and Technology of China in 1998 and M.E. degree in computer science from Central South University in 2004. Since 2004, he has been worked in Hunan University. Currently, he is a Ph.D. candidate in College of Information Science and Engineering of Hunan University. His research interests include data structure and algorithm design in sensor network and access control.



Jinguo Li was born in 1985. He is a Ph.D. student in College of Information Science and Engineering in Hunan University. He received his B.E. Degree in Information Security, Hunan University. He is interested in the analysis and design of network algorithms and privacy-preserving system for wireless networks.

Xin Yao was born in 1989. He is an M.E. Graduate student in College of Information Science and Engineering, Hunan University. He received his B.E. Degree in Computer Science, XiDian University. His research interests are in networking, security and dependable system.

Peng Liu is a Ph.D. student in College of Information Science and Engineer, Hunan University. He received the B.E. Degree in automation from Xiangtan University, in 2006, and M.E. Degree in College of Conformation Science and Engineer, Hunan University. His research interests include low power testing, low cost test and test generation.