# A Fast and Memory Efficient Dynamic IP Lookup Algorithm Based on B-Tree

Yeim-Kuan Chang and Yung-Chieh Lin
Department of Computer Science and Information Engineering
National Cheng Kung University
701 Tainan, Taiwan R.O.C.
ykchang@mail.ncku.edu.tw, p7894110@ccmail.ncku.edu.tw

*Abstract*—**This paper deals with the traditional IP address lookup problem with fast updates. We propose a B-tree data structure, called MMSPT (Multiway Most Specific Prefix Tree), which is constructed by using the most specific prefixes in routing tables. MMSPT arranges the most specific prefixes as the keys in B-tree. Unlike the previous schemes, every prefixes in routing tables is stored exactly once in our MMSPT. For a routing table of $n$ prefixes, MMSPT requires $O(n)$ memory, and the time for search, insertion and deletion operations are $O(\log_m n)$, $O(m\log_m n)$, and $O(m\log_m n)$, respectively ($m$ is the order of the B-tree). Our experimental results conducted by using five real IPv4 routing tables show that MMSPT outperforms two existing B-tree data structures, PIBT (Prefix In B-Tree) and MRT (Multiway Range Tree), in all aspects. Moreover, since the complexities of MMSPT is not subject to the length of IP addresses, the proposed MMSPT can be easily extended to fit the IPv6.**

## I. INTRODUCTION

The Internet consists of a mesh of routers interconnected by links. In a router, the major function in packet forwarding process is to lookup the destination addresses for the incoming packets according to the routing table. In the recent years, due to the prevalence of the World Wide Web (WWW) and many emerging multimedia networking applications, network traffic at major exchange points is doubling every few months. The increasing traffic has put great loads on the capacity of routers. To keep pace with the Internet traffic growth and continue to furnish good quality-of-service (QoS) on the Internet, there is an urgent need of the fast IP lookup algorithm with fast update.

Furthermore, the Internet is a dynamic system. New routes and new sub-networks can be added to the Internet. Corresponding to these changes, the routers have to process the changes on the fly and adjust the routing table and the corresponding search structure. However, if these changes (updates) require extensive computation, vast amount of memory accesses, or even total reconstruction of the search structure, it can degrade the performance of a router considerably.

Today, backbone routers typically run the Border Gateway Protocol (BGP). Because some BGP implementations exhibit considerable instability, the routing tables of backbone routers may have a peak of a few hundred BGP updates per second. Thus, a lookup algorithm at least should be able to perform 1k updates per second to avoid routing instabilities. Hence, besides fast lookups, algorithms can support fast updates are also very important.

Besides the lookup speed and the update time, to design a good IP lookup algorithm, there are still three issues needed to be considered. They are storage requirement, scalability, and flexibility in implementation. We describe each of them as follows.

**Storage requirement.** Nowadays, the on-chip memory can be as large as several hundred KB and the off-chip memory several MB. We desire an IP lookup data structure that requires small storage requirement. Small storage requirement means high memory access speed and low power consumption, which can benefit from an on-chip cache if implemented in software, and from an on-chip SRAM (Synchronous Random Access Memories) if implemented in hardware.

**Scalability.** Here the scalability means two things. One is the scalability to the number of routing entries in routing tables and the other is the scalability to the address length. Today, the routing tables of backbone routers contain approximately 200k route prefixes and are growing rapidly. The ability of a lookup algorithm to handle large real-life routing tables is required. Besides, as the Internet evolves into a global communication medium, each user may require multiple addresses. Despite temporary measures such as Network Address Translation (NAT) boxes (where several hosts share a common IP address), the switch to longer addresses (e.g. IPv6) seems inevitable. Since IPv6 uses 128 bit addresses, schemes whose lookup time grows with address length become less attractive.

**Flexibility in implementation.** The forwarding engines in a router may be implemented either in software or hardware depending upon the system requirements. Thus, a lookup algorithm should have the flexibility of being implemented in different ways, such as an ASIC, a network processor, or a generic processor. For the highest speeds (e.g., OC768c), integration parallel or pipeline mechanism into the lookup algorithms is a must for the future lookups.

In the last couple of years, various algorithms for high-performance IP lookups have been proposed. By simply analyzing them, current IP lookup algorithms can be broadly classified into two categories: static and dynamic. The static schemes, like [2], [4], [5], [11], and [17], are constructed by performing a lot of precomputation. The precomputation usually can simplify the entire data structure of the routing table and thus improve the performance of the lookup speed and memory

requirement. However, the disadvantage of the precomputation is that when a single prefix is added or deleted, the entire data structure may need to be rebuilt. This seriously affects the lookup performance. Hence, in terms of the lookup speed and memory consumption, schemes of static category perform well, but not in terms of the update time. On the other hand, some dynamic schemes based on the trie structure, like binary trie, multibit trie [16] and Patricia trie [15], do not use precomputation and can easily support route updates; however, their performances grow linearly with the address length, and thus these schemes lack the scalability when switching to IPv6 or large routing table.

In fact, for an IP lookup algorithm, it is hard to fulfill all the five issues we describe above. In this paper, we try to develop a data structure that can get the balance among these five issues. Since the static algorithms cannot support the dynamic updates (i.e., inserting and deleting prefixes without precomputation and rebuilding), we focus on the dynamic algorithms. We propose a B-tree structure called Multiway Most Specific Tree (MMSPT) to solve the IP lookup problem with updates.

The rest of the paper is organized as follows. In Section II, we review the existing dynamic schemes. In sections III, we illustrate the proposed MMSPT. The intensive experimental results will be shown in Section IV. Finally, a brief conclusion will be remarked in Section V.

## II. Related Work

Despite the extensive research conducted in recent years about the IP lookup problem, algorithms that balance among lookup speeds, memory requirement, update time, scalability, and flexibility in implementation are scarce. Both static and dynamic schemes are surveyed in [12] and [14]. By briefly summarizing them, although the static schemes have fast lookup speed and small memory requirement, they cannot afford frequent insertions and deletions of route prefixes. and the performance of the trie-based dynamic schemes degrade linearly with the address length when switching to IPv6 and large routing tables. In this section, we review the existing dynamic schemes that not only support dynamic insertions and deletions but also scale well to IPv6.

Kim and Sahni [13] developed a dynamic data structure called the collection of red-black trees (CRBT). The basic interval tree of CRBT is constructed from the distinct endpoints of all prefixes. CRBT supports search, insert, and delete operations in $O(\log N)$ time each for a routing table of $N$ prefixes. Lu and Sahni proposed a dynamic scheme based on the priority search tree (PST) [7] which arrives at $O(\log N)$ time complexity for search, insertion, and deletion. The experimental results in [6] showed that PST performs a little worse than CRBT in terms of search time. However, PST performs much better than CRBT in terms of insertion, deletion, and memory usage.

Lu and Sahni also developed an enhanced interval tree which is called the binary tree on binary tree (BOB) [8] for dynamic routing tables. With real routing tables, BOB and prefix BOB (PBOB) perform the operations of insertion, deletion, and search in $O(\log N)$ time. Also, the PBOB and the longest matching prefix BOB (LMPBOB) perform much better than PST in terms of search, insertion, deletion, and memory requirement.

TABLE I. A Prefix Set of 9 Prefixes, for $W = 6$

| Name | Prefix | Interval | Name | Prefix | Interval |
|---|---|---|---|---|---|
| $p_1$ | 00* | [0, 15] | $p_6$ | 11* | [48, 63] |
| $p_2$ | 01* | [16, 31] | $p_7$ | 1100* | [48, 51] |
| $p_3$ | 0001* | [4, 7] | $p_8$ | 110111 | [55, 55] |
| $p_4$ | 1* | [32, 36] | $p_9$ | 100* | [32, 39] |
| $p_5$ | 01011* | [22, 23] | | | |

Feldmann et al. [3] proposed a dynamic multiway fat inverted segment tree (FIS) for dynamic insertions and deletions of ranges. The search time of $O(\log_m N)$ can be achieved in a tree of degree $m$. In [18], a B-tree data structure which is called the multiway range tree (MRT) was proposed to find the longest matching prefix in $O(\log_m N)$ time, and insert or delete a prefix in $O(m\log_m N)$ time. MRT is suitable for both prefixes and ranges. However, there are many duplicate endpoints stored in internal nodes, and a prefix may be stored in at most $m - 1$ nodes per B-tree level. This drawback increases the update time and memory requirement. Another B-tree data structure called range in B-tree (RIBT) in [9] is proposed for solving this drawback by storing a range in only $O(1)$ B-tree nodes per B-tree level. The experimental results conducted by using real routing tables in [9] show that the performance of RIBT and MRT are almost the same. However, RIBT is more memory efficient than MRT by a constant factor.

## III. Proposed Data Structure

### A. Preliminaries

A string $b_0b_1b_2\cdots b_{l-1}*$ represents a *prefix* with length $l$, where $b_i \in \{0, 1\}$, for $i = 0$ to $l - 1$ and '*' represents a wildcard or "don't care" bit. We use $W$ to denote the maximum possible value of $l$ (for IPv4, $W = 32$). Moreover, prefix $b_0b_1b_2 \cdots b_{l-1}*$ represents a range of addresses $[s, f]$, where $s = (b_0b_1b_2\cdots b_{l-1}0^{W-l})_2$, $f = (b_0b_1b_2 \cdots b_{l-1}1^{W-l})_2$, and $\alpha^k$ represents the string obtained by repeating $\alpha$ $k$ times. For a prefix $p$, we use $len(p)$, $start(p)$, and $finish(p)$ to denote the length, the start address (i.e., $s$), and the finish address (i.e., $f$) of $p$, respectively. For example, when $W = 6$, prefix $p$, 1010*, represents the range [40, 43]. Moreover, $len(p) = 4$, $start(p) = 40$, and $finish(p) = 43$.

**Definition 1:** For any two different prefixes $p_1$ and $p_2$, we say $p_1$ *covers* $p_2$ (or $p_2$ is *covered* by $p_1$) iff $start(p_1) \le start(p_2)$ and $finish(p_2) \le finish(p_1)$. We also say $p_2$ is *more specific* than $p_1$ and they are *nested*. For example, in Table I, $p_1$ covers $p_3$, $p_5$ is covered by $p_2$ and $p_8$ is more specific than $p_6$.

**Definition 2:** For any two different prefixes $p_1$ and $p_2$, we say $p_1$ and $p_2$ are *disjoint* iff $finish(p_1) < start(p_2)$ or $finish(p_2) < start(p_1)$. For example, in Table I, $p_1$ and $p_2$ are disjoint.

**Definition 3:** For a given prefix set $P$, we say a prefix $p \in P$ is the *most specific prefix* in $P$ iff no prefixes in $P$-$\{p\}$ are covered by $p$ (i.e., no prefixes in $P$-$\{p\}$ are more specific than $p$). For example, in Table I, $p_3$, $p_5$, $p_7$, $p_8$, and $p_9$ are the most specific prefixes.

**Definition 4:** For any two different prefixes $p_1 = [s_1, f_1]$ and $p_2 = [s_2, f_2]$, we say $p_1 < p_2$ iff (1) $s_1 < s_2$, or (2) $s_1 =$

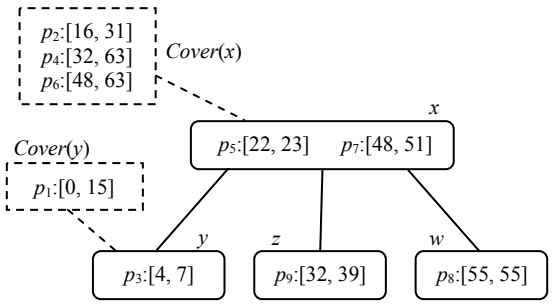



Figure 1. MMSPT for Table I

$s_2$ and $f_1 > f_2$. For example, in Table I, $p_1 < p_2$, $p_3 < p_4$, and $p_6 < p_7$.

### B. Our Multiway Most Specific Prefix Tree Structure

MMSPT is an augmented $m$-way search tree (B-tree of order $m$). For a given prefix set $P$, MMSPT stores the most specific prefixes in $P$ as keys in each node. Just like the traditional B-tree, each node $x$ of MMSPT has the following fields:

a. $n(x)$, the number of keys currently stored in $x$,

b. the $n(x)$ keys $key_i(x)$, for $i = 1 \sim n(x)$, stored in increasing order, so that $key_1(x) < key_2(x) < \cdots < key_{n(x)}(x)$,

c. the $n(x) + 1$ child pointers $child_i(x)$, for $i = 0 \sim n(x)$, where $child_i(x)$ points to the $i$th child (subtree) of $x$.

Notice that, now each of keys in the MMSPT nodes is a prefix instead of a value, hence they are compared (sorted) by using definition 4. Besides the above three fields, we augment $x$ with a prefix set $Cover(x)$, where $Cover(x)$ is a subset of $P$ and each of prefixes in $Cover(x)$ must follow the *prefix allocation rule* which is defined as follows,

**Definition 5:** For a MMSPT node $x$, each prefix $p$ in $Cover(x)$ must cover at least one key in node $x$ and $p$ is disjoint with all keys in the parent node (if it exists) of $x$. This is called the *prefix allocation rule*.

Fig. 1 shows an order 3 MMSPT for the prefix set of Table I. As we can see, there are total four MMSPT nodes $x$, $y$, $z$, and $w$. Node $x$ has two keys where $key_1(x) = p_5 =$ [22, 23] and $key_2(x) = p_7 =$ [48, 51]. Nodes $y$, $z$, and $w$ all have one key, where $key_1(y)$, $key_1(z)$, and $key_1(w)$ are $p_3 =$ [4, 7], $p_9 =$ [32, 39], and $p_8 =$ [55, 55], respectively. All of these keys are the most specific prefixes in Table I. The rectangles with dotted lines beside $x$ and $y$ represent $Cover(x)$ and $Cover(y)$. Moreover, $Cover(z) = Cover(w) = \phi$.

Notice that, by using the most specific prefixes in a prefix set and following the prefix allocation rule, we can use a balanced binary search tree (e.g., AVL trees or red-black trees) instead of B-trees to construct a binary-MSPT. However, inserting or deleting a key may make the binary-MSPT unbalanced. To maintain the balance of the binary search tree, one or more rotations must be performed. The cost of rotations in binary search trees is much higher than split and merge operations in B-tees. Moreover, the search speed of the binary structure is much more than the multiway structure. That is the reason why the proposed structure is a multiway structure instead of a binary structure.

### C. Longest Prefix Matching

Since all the keys stored in MMSPT are disjoint, no more than one key matches the query address $d$. Since all the keys are most specific, this matched key must be the longest matching prefix of $d$. Moreover, the MMSPT search algorithm employs the following lemma.

```
Algorithm MMSPT_search(d){
// return the longest matching prefix of d
// Assume key_0(x) = [-1, -1] and key_{n(x)+1}(x) = [2^W, 2^W]
// array collect[] collects the pointers to nodes with non-empty cover
01  x = root; // root of MMSPT
02  j = 0; // counter for array collect[]
03  while(x ≠ null){
04    Let i be the index such that d < key_i[x];
05    if(key_{i-1}(x) matches d)
06      return key_{i-1}(x) as the longest matching prefix of d;
07    if (cover(x) is not empty) collect[j++] = x;
08    x = child_{i-1}(x);
09  }// end of while-loop
10  for(i = j -1 ; i ≥ 0 ; i--){
11      return the longest prefix in Cover(collect[i]) that match d
12  } // end of for-loop
13  return default prefix; // no prefixes in MMSPT match d
}
```

Figure 2. *MMSPT_search* algorithm for a query address $d$

**Lemma 1:** If a query address $d$ is both matched by two prefixes $p_1$ and $p_2$, where $p_1 \in Cover(x)$ for a MMSPT node $x$ and $p_2 \in Cover(x$'s parent), then $p_2$ covers $p_1$ (i.e., $p_1$ is more specific than $p_2$ and $len(p_1)$ is longer than $len(p_2)$.).

**Proof:** We know two prefixes are either disjoint or nested. Since $p_1$ and $p_2$ contain a common address $d$, $p_1$ and $p_2$ must not be disjoint, and therefore $p_1$ and $p_2$ are nested. Since $p_1$ and $p_2$ must follow the prefix allocation rule, $p_2$ must cover $p_1$. □

The search process for a query address $d$ begins at the root of MMSPT and traces a path downward to a leaf node, as shown in Fig. 2. For each node $x$ encountered, the $key_i(x)$ where $d < key_i(x)$ is found first. For simplicity, in the rest of the paper, we assume $key_0(x) = [-1, -1]$ and $key_{n(x)}(x) = [2^W, 2^W]$ for every node $x$. If $key_{i-1}(x)$ matches $d$, the search terminates. Otherwise, the search goes to the node pointed by $child_{i-1}(x)$. When the search arrives at a leaf node and there still no key that matches $d$, we backtrack the same path upward to the root. For each node $x$ encountered, we find the longest prefix in $Cover(x)$ that matches $d$. If all prefixes stored in MMSPT do not match $d$, it returns the default prefix (Fig. 2, line 13).

In real routing tables, more than 90 percent of prefixes are the most specific prefixes. In other words, only less 10 percent of prefixes are stored in $Cover(x)$ for some node $x$. Hence, most of the queries can find a match when we traverse down from the root. Thus, backtracking is not performed frequently. Table II shows this observation by analyzing five real BGP routing tables obtained from [1] and [10]. As we can see, about half of node their *Cover*()

TABLE II. Analyses of five real BGP Routing Tables

| Routing Table | AS6447 | AS6447 | AS6447 | AS7660 | AS2493 |
|---|---|---|---|---|---|
| Year-month | 2000-4 | 2002-4 | 2005-4 | 2005-4 | 2005-4 |
| # of entries | 79,560 | 124,824 | 163,574 | 159,816 | 157,118 |
| # of most specific prefixes | 73,891 (93%) | 114,745 (92%) | 150,245 (92%) | 145,849 (91%) | 143,683 (91%) |
| Maximum size of *Cover*( ) | 19 | 21 | 22 | 24 | 21 |
| Avg. size of *Cover*( ) | 1.14 | 1.31 | 1.33 | 1.42 | 1.39 |
| # of nodes | 4,925 | 7,648 | 10,015 | 9,722 | 9,579 |
| # of nodes whose *Cover*( ) = ϕ | 2,434 (49%) | 3,616 (47%) | 4,951 (49%) | 4,524 (47%) | 4,432 (46%) |

```
Algorithm MMSPT_insert(p){ // insert a prefix p into MMSPT
01 x = root; // root of MMSPT
02 while(x ≠ null){
03   Let i be the index such that p < key_i[x];
04   if(p covers key_i[x]){
05     add p in Cover(x); return;}
06   if(key_{i-1}(x) = p){ // p has already existed
07     change the routing information associated with key_{i-1}(x);
08     return;}
09   if(key_{i-1}(x) covers p){
10     add key_{i-1}(x) in Cover(x) and set p as the new key_{i-1}(x);
11     return;}
12   y = x;
13   x = child_{i-1}(x);
14 }// end of while-loop
15 right shift the keys of y by 1begining with those at position i;
16 insert p as the new key_i(y);
17 n(y)++;
18 if(n(y) = m)
19   buttom_up(y); // adaptation of split process of traditional B-tree
}
```

Figure 3. Algorithm to insert a prefix $p$ into the MMSPT

is empty. In average, there are less than two prefixes stored in *Cover*($x$) of a node $x$. According to this observation, for each node $x$, we can implement its *Cover*($x$) using a sorted array (according to definition 4). The search in *Cover*($x$) starts from the last element (largest prefix in *Cover*($x$)) of this array and sequentially to the first element (smallest prefix in *Cover*($x$)). Once an element matches the query address, the search terminates (Fig. 2, lines 11-12). Since the array is quite small, in practice, the search, insert, and delete in a small array are more efficient than other data structures.

**Complexity Analysis.** Since more than 90 percent prefixes are the most specific prefixes, most of the search can be done without back tracking process. In that case, each iteration of the **while** loop of Fig. 2 takes $O(\log_2 m)$ time (we assume throughout this paper that, for sufficiently large $m$, a B-tree node is searched using a binary search) and the number of iterations is $O(\log_m n)$ (the height of the B-tree, $n$ is the number of prefixes). So, the overall complexity is $O(\log_2 m \log_m n) = O(\log_2 n)$.The number of node accessed by the algorithm is $O(\log_m n)$.

### D. Inserting a Prefix

The algorithm to insert a prefix $p$ into the MMSPT structure is an adaptation of the standard B-tree insertion algorithm. Just like the search process of MMSPT, the insertion begins at the root of MMSPT and traverses a path downward, as shown in Fig. 3. We search the MMSPT for a key that covers $p$, is equal to $p$, or is covered by $p$. If such a key exists, the insert process terminates at the node that contains this key. And it performs some simple adjustments (Fig. 3, lines 4-11) to follow the constraints of MMSPT. Otherwise, $p$ is inserted as a key in a leaf node. Assume $p$ is inserted into a leaf node $x$ between $key_{i-1}(x)$ and $key_i(x)$, where $key_{i-1}(x) < p < key_i(x)$. When $n(x) <$ m, the described insertion of $p$ is finished. When $n(x) = m$, the described insertion of $p$ into $x$ makes $x$ has one key more than its capacity $m$–1. Just like the traditional B-trees, node $x$ is split into two node around $key_g(x)$, where $g = \lceil m/2 \rceil$. Keys of $x$ to the left of $key_g(x)$ remain in $x$, those to the right are placed into a new node $y$, and $key_g(x)$ is inserted into the parent node of $x$. Moreover, prefixes in *Cover*($x$) that cover $key_g(x)$ are removed from *Cover*($x$) and inserted into *Cover*(parent node of $x$). Consider a prefix $p$ in *Cover*($x$). If $p$ is greater than $key_g(x)$ but not covers $key_g(x)$, it must cover at least one key in $y$ but not cover any prefix in $x$ according to the prefix allocation rule. Therefore, prefixes in *Cover*($x$) that are greater than $key_g(x)$ are also removed from *Cover*($x$) and inserted into *Cover*($y$). The split procedure is performed by the function *buttom_up*() in line 19 of Fig 3. And function *buttom_up*() will be performed iteratively on the nodes on the path from $x$ to the root.

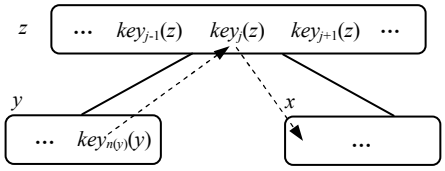



Figure 4. Node $x$ borrows a key from its nearest left sibling $y$

### E. Deleting a Prefix

To delete a prefix $p$ from the MMSPT structure, if $p$ is in *Cover*($x$) for some node $x$, then we remove $p$ from *Cover*($x$) and the deletion for $p$ is finished. Otherwise $p$ is a key for some node $x$. To delete a key from the MMSPT structure considers two cases:

Case 1): $x$ is a leaf node.

Case 2): $x$ is an interior node.

**Case 1): *x* is a leaf node.** To delete a prefix $p$ where $p = key_i(x)$, we first determine is there a prefix $p_2$ with longest length in *Cover*($x$), such that $p_2$ only covers $p$ (i.e., $p_2$ dose not cover $key_{i-1}(x)$ and $key_{i+1}(x)$). If such a prefix exists, we use $p_2$ to replace $p$ as the new $key_i(x)$, and we are done. Otherwise, we remove $key_i(x)$ from $x$ and the keys to the right of $key_i(x)$ are shifted one position left. If the number of keys that remain in $x$ is at least $\lceil m/2 \rceil$ (except $x$ is the root), the deletion of $p$ is finished. Otherwise, $x$ is deficient and we do the following:

1. If a nearest sibling of $x$ has more than $\lceil m/2 \rceil$ keys. $x$ borrows (gains) a key via this nearest sibling, and therefore is no longer deficient.
2. Otherwise, $x$, its nearest sibling, and a key in the parent node of $x$ are merged into one node. This may cause the parent node of $x$ to become deficient, hence this deficiency resolution process is repeated in the parent node of $x$.

**Borrow from a sibling.** Assume $x$'s nearest left sibling $y$ has more than $\lceil m/2 \rceil$ keys, as shown in Fig. 4. $key_{n(y)}(y)$ is the largest (rightmost) key in $y$ and $key_j(z)$ is the key in $z$ (parent of $x$ and $y$) such that $child_{j-1}(z) = y$ and $child_j(z) = x$.

The borrow operation does the following:

1. In $z$, $key_j(z)$ are replaced by $key_{n(y)}(y)$.
2. In $x$, all keys and child pointers are shifted one right. $child_{n(y)}(y)$ and $key_j(z)$ become $child_0(x)$ and $key_1(x)$, respectively.
3. From *Cover*($z$), remove the prefixes that are greater than $key_{n(y)}(y)$ and do not cover $key_{j+1}(z)$, then add these removed prefixes in *Cover*($x$).
4. From *Cover*($y$), remove the prefixes that cover $key_{n(y)}(y)$ and add these removed prefixes in *Cover*($z$).

**Merging two adjacent siblings.** As shown in Fig. 5, $z$ is the parent node of $x$ and $y$, and $x$ and $y$ are pointed by $child_{j-1}(z)$ and $child_j(z)$, respectively. When node $x$ is deficient and its nearest right sibling $y$ has exactly $\lceil m/2 \rceil$ – 1 keys, node $x$, $y$, and $key_j(z)$ are combined into a single

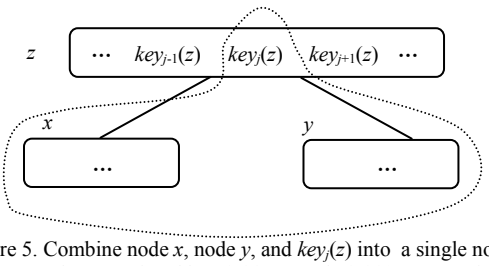



Figure 5. Combine node $x$, node $y$, and $key_j(z)$ into a single node, where $x$ and y are pointed by $child_{j-1}(z)$ and $child_j(z)$, respectively

node $x'$. The resulting $x'$ now has $2\lceil m/2 \rceil - 2$ keys. Besides, $Cover(x)$ and $Cover(y)$ are combined into $Cover(x')$. The prefixes in $Cover(z)$ that only cover $key_j(z)$ are removed from $Cover(z)$ and added into $Cover(x')$.

Since the merging of $x$ and $y$ reduces the number of keys in $z$ by 1, $z$ may become deficient. Hence, the same borrow/merge process may be repeated at the nodes from $z$ all the way up to the root.

If there are $k$ prefixes stored in $Cover(z)$, to find the prefixes in $Cover(z)$ that only cover $key_j(z)$ takes $O(k)$ time. Since $k$ is rarely small, we can treat $k$ like a constant. Hence, each merge operation takes $O(m)$ time and involves three nodes. So does each split operation.

**Case 2): x is an interior node.** As shown in Fig. 5, we assume $p$ is the $i$th key in an interior node $x$ (i.e., $p = key_i(x)$), and $y$ and $z$ are the child nodes of $x$ pointed by $child_{i-1}(x)$ and $child_i(x)$, respectively. Moreover, $key_{n(w)}(w)$ (the last key of node $w$) is the largest key in the subtree rooted at $y$, and $key_1(v)$ (the first key of node $v$) is the smallest key in the subtree rooted at $z$.

Just like case 1), to delete $p$ from $x$, we first determine are there any prefixes in $Cover(x)$ that only cover $p$, if so, we use the one who has the longest length to replace $p$, then we are done. Otherwise, $p$ is replaced either by $key_{n(w)}(w)$ or $key_1(v)$. Assume we replace $p$ by $key_1(v)$, For every node $u$ on the path from $z$ to $v$ (including $z$ and $v$), all prefixes stored in $Cover(u)$ that cover $key_1(v)$ are removed from $Cover(u)$ and added into $Cover(x)$. Then the deletion process of $p$ from node $x$ now becomes the deletion process of $key_1(v)$ from node $v$. Since $v$ is a leaf node, now the deletion process becomes case 1) situation which has been described previously.

## IV. PERFORMANCE EVALUAITON

### A. Simulation Environment

We programmed all tested schemes in C and all codes were run on a 2.4GHz Pentium 4 PC that has 8KB L1, 256KB L2 caches, and 512MB main memory. The gcc-3.2.2 compiler of the Redhat 9.0 with optimization level –O4 was used. Moreover, an instruction of the Intel processor called RDTSC (ReaD Time Stamp Counter) is used. This instruction can keep an accurate count of every clock cycle that occurs on the processor. We use this instruction to estimate the speeds of the search, insert, delete time, respectively.

### B. Tested Data and Tested Schemes

Our experiments are conducted by using five real BGP routing tables with different size. Table II shows the detailed information of these five routing tables. The routing tables having the same name (in AS number) means they were obtained from the same BGP backbone

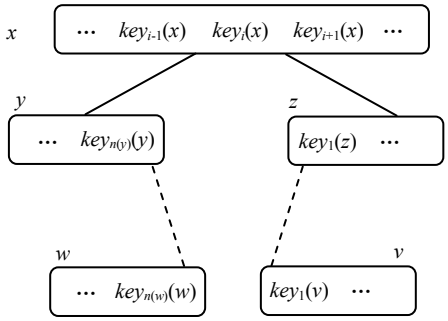



Figure 6. To delete a prefix $p = key_i(x)$ from the MMSPT structure

router at different dates, and the first three routing tables with the same name are obtained from [10] and other two were obtained from [1]. We can see the routing table AS6447, as time goes on, it grows rapidly. Moreover, the number of prefixes shown in the third row of Table II reveals the current BGP routing tables deployed on the Internet are all very large.

We compare our MMSPT with multiway range tree (MRT) [18], prefix in B-tree (PIBT) [9], and prefix binary tree on binary tree (PBOB) [8] in terms of memory usage, lookup speed, insert time, and delete time. MMSPT, MRT, and PIBT are all multiway data structures and we implement them in B-trees of order 32. PBOB is implemented in red-black trees.

### C. Memory Usage

For each prefix in a routing table, MMSPT and PBOB only store this prefix at most in one node. However, MRT and PIBT will store this prefix in duplicate in constant nodes per B-tree level.

Table III and Fig. 7 show the memory usage of each scheme for five BGP routing tables. As we can see, MMSPT uses about 63 percent less memory than MRT and PIBT. Since MMSPT is a B-tree structure, for a MMSPT node $x$ of $n(x)$ keys, we allocate the memory of $m$ - 1 keys plus $m$ pointers (although $x$ only uses $n(x)$ keys

TABLE III.
Memory Usage, Search Time, Insertion Time, and Deletion Time of Each Scheme for Five Real BGP Routing Tables

| Routing Table | | AS6447 2000-4 | AS6447 2002-4 | AS6447 2005-4 | AS7660 2005-4 | AS2493 2005-4 |
|---|---|---|---|---|---|---|
| Memory (KByte) | MMSPT | 1,802 | 2,805 | 3,673 | 3,570 | 3,516 |
| | PIBT | 4,966 | 7,745 | 10,097 | 9,842 | 9,689 |
| | MRT | 4,815 | 7,509 | 9,791 | 9,543 | 9,394 |
| | PBOB | 1,591 | 2,477 | 3,236 | 3,144 | 3,095 |
| Search (μsec) | MMSPT | 0.38 | 0.44 | 0.48 | 0.48 | 0.47 |
| | PIBT | 0.39 | 0.46 | 0.49 | 0.48 | 0.49 |
| | MRT | 0.45 | 0.51 | 0.55 | 0.55 | 0.55 |
| | PBOB | 0.90 | 1.08 | 1.19 | 1.16 | 1.16 |
| Insert (μsec) | MMSPT | 0.73 | 0.75 | 0.75 | 0.75 | 0.75 |
| | PIBT | 1.70 | 1.66 | 1.59 | 1.58 | 1.58 |
| | MRT | 1.22 | 1.21 | 1.21 | 1.24 | 1.26 |
| | PBOB | 0.85 | 0.85 | 0.85 | 0.86 | 0.85 |
| Delete (μsec) | MMSPT | 0.67 | 0.68 | 0.68 | 0.65 | 0.65 |
| | PIBT | 1.83 | 1.84 | 1.85 | 1.84 | 1.82 |
| | MRT | 1.91 | 1.92 | 1.92 | 1.92 | 1.90 |
| | PBOB | 0.71 | 0.71 | 0.73 | 0.75 | 0.74 |

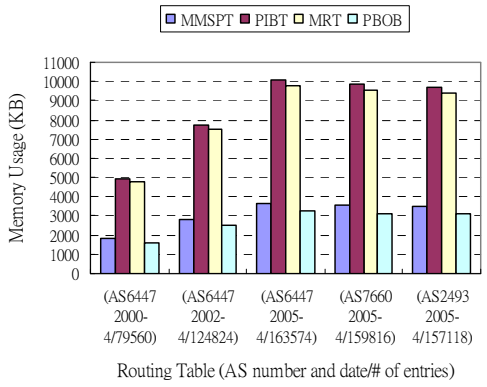



Figure 7. Memory usage of each scheme for five real BGP routing tables

and $n(x) + 1$ pointers). Hence, the memory size per node of MMSPT is larger than that of PBOB. As the results, MMSPT consumes a little more memory than PBOB.

### D. Search Time

To measure the average search time, we first constructed each scheme according to the original routing table. For each prefix $p$ in the original routing table, we randomly choose a address $d \in D(p)$, then we collected these addresses. Further, we disarrange the sequence of these collected addresses as the simulation IP traffic.

Table III and Fig. 8 show the search time of each scheme for five BGP routing tables. Since PBOB is the binary structure, as we expect, the search time of three B-tree structures are all much faster than PBOB. For a routing table of $n$ prefixes, MMSPT stores $n$ keys while MRT and PIBT store $2n$ keys. Hence, the height of MMSTPT is lower than those of MRT and PIBT. That is the reason why the search speed of MMSPT is faster than MRT and PIBT.

### E. Insertion Time and Deletion Time

To measure the insertion time and the deletion time, we first constructed each scheme according to the original routing table; further, we randomly deleted 5 percent prefixes from the structure we built. The total elapsed time to delete these 5 percent prefixes divided by the number of these 5 percent prefixes is the average time for a single deletion. We proceed to insert these deleted 5 percent prefixes back into the structure. The total elapsed time to insert these 5 percent prefixes back divided by the

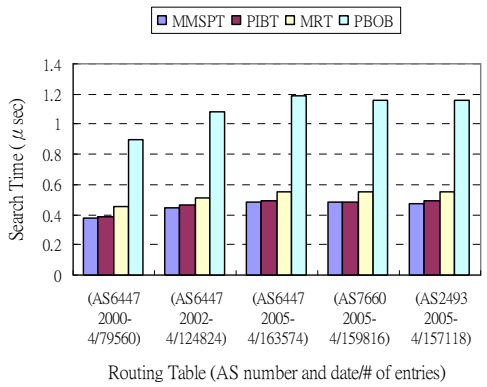
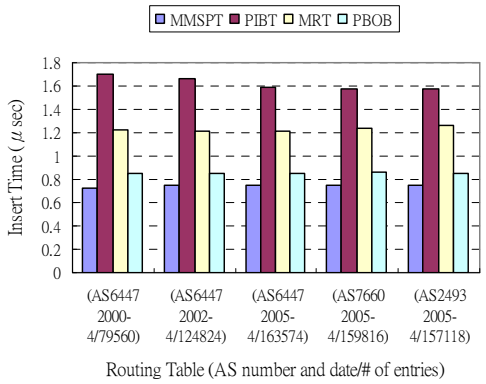



Figure 8. Search time of each scheme for five real BGP routing tables



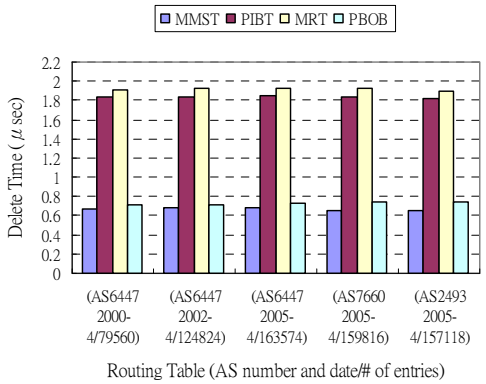




Figure 9. Insertion and deletion times of each scheme for five real BGP routing tables

number of these 5 percent prefixes is the average time for a single insertion.

For a prefix $p$, the insertion/deletion of $p$ in/from MRT and PIBT are a three-pass procedure including 1) inserting/deleting *start*($p$), 2) inserting/deleting *finish*($p$), 3) inserting/deleting $p$ itself. However, the insertion and deletion processes of MMSPT and PBOB only involve the third step, inserting/deleting $p$ itself. As the results, MMSPT and PBOB have better insertion and deletion time than MRT and PBOB.

Table III and Fig. 9 show the insertion and deletion time of each scheme for five BGP routing tables. As we can see, the insertion and deletion time of MRT and PIBT are the worst. Since PBOB is a balanced binary search tree, to maintain the balance of the tree, the insertion and deletion of PBOB often involve one or several rotations. Rotations can seriously affect the performance of insertion and deletion of PBOB. Contrast with MMSPT, the number of split and merge operations caused by insertion and deletion are relatively small (when $m$ is acceptably large). Hence, MMSPT has better insertion and deletion time than PBOB.

## V. CONCLUSION

We have proposed a data structure called multiway most specific tree (MMSPT) for IP lookup problem with fast updates. MMSPT is a balanced $m$-way search tree, where each key in a MMSPT node is the most specific prefix. The non-most-specific prefixes are stored by following the prefix allocation rule we defined. The experimental results conducted by using five real IPv4 touring tables show that the search, insertion, and deletion

speeds of MMSPT are all better than the existing dynamic schemes (PBOB [8], MRT [18], and PIBT [9]). Moreover, since the complexity of MMSPT is not subject to the length of IP addresses, the proposed MMSPT can be easily extended to fit the IPv6.

## REFERENCES


[1] BGP Routing Table Analysis Reports, http://bgp.potaroo.net/.

[2] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," *Proceeding of ACM SIGCOMM*, pp. 3-14, September 1997.

[3] A. Feldman and S. Muthukrishnan, "Tradeoffs for packet classification," *Proceeding of ACM SIGCOMM*, vol. 3, pp. 1193-1202, March 2000.

[4] N. Huang, S. Zhao, J. Pan, and C. Su, "A Fast IP Routing Lookup Scheme for Gigabit Switching Routers," *Proceeding of IEEE INFOCOM*, pp. 1429-1436, March 1999.

[5] B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search," *Proceeding of IEEE INFOCOM*, pp. 1248-1256, April 1998.

[6] H. Lu, K. Kim, and S. Sahni, "Prefix and interval-partitioned dynamic IP router-tables," *IEEE Transactions on Computers*, vol. 54, no. 5, pp. 545-557, May 2005.

[7] H. Lu and S. Sahni, "$O(\log n)$ dynamic router-tables for prefixes and ranges," *IEEE Transactions on Computers*, vol. 53, no. 10, pp. 1217-1230, October 2004.

[8] H. Lu and S. Sahni, "Enhanced interval trees for dynamic IP router-tables," *IEEE Transactions on Computers*, vol. 53, no. 12, pp. 1615-1628, December 2004.

[9] H. Lu and S. Sahni, "A B-tree dynamic router-table design," *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 813-824, July 2005.

[10] D. Meyer, University of Oregon Route Views Archive Project, at http://archive.routeviews.org/.

[11] S. Nilsson and G. Karlsson, "IP-Address lookup using LC-trie," *IEEE Journal of Selected Areas in Communications*, vol. 17, no. 6, pp. 1083-1092, June 1999.

[12] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous, "Survey and taxonomy of IP address lookup algorithms," *IEEE Network*, vol. 15, no. 2, pp. 8-23, March/April 2001.

[13] S. Sahni and K. Kim, "An $O(\log n)$ dynamic router-table design," *IEEE Transactions on Computers*, vol. 53, no. 3, pp. 351-363, March 2004.

[14] S. Sahni, K. Kim, and H. Lu, "Data structure for one-dimensional packet classification using most-specific-rule matching," *Proceeding of Int'l Symp. Parallel Architecture, Algorithms, and Networks (ISPAN)*, pp. 3-14, May 2002.

[15] K. Sklower, "A Tree-Based Packet Routing Table for Berkely UNIX," Technical report, University of California, Berkeley, 1993.

[16] V. Srinivasan and G. Varghese, "Fast IP lookups using controlled prefix expansion," *ACM Transactions on Computer Systems*, pp.1-40, February 1999.

[17] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high-speed IP routing lookups," *Proceeding of ACM SIGCOMM*, pp. 25-36, September 1997.

[18] P. Warkhede, S. Suri, and G. Varghese, "Multiway range trees: scalable IP lookup with fast updates," *Computer Networks: The International Journal of Computer and Telecommunications Networking*, vol. 44, no. 3, pp. 289-303, February 2004.