Multi-Prefix Trie: a New Data Structure for Designing Dynamic Router-Tables

Sun-Yuan Hsieh *Member, IEEE,* Yi-Ling Huang, and Ying-Chi Yang

***Abstract*—IP lookup affects the speed of an incoming packet and the time required to determine which output port the packet should be sent to; hence, it plays an important role in the design of router-tables. In this paper, we propose a new data structure, called a multi-prefix trie, for use in designing dynamic router-tables. One key feature of our data structure is that each node can store more than one prefix, which reduces the number of memory accesses. When performing lookup, the structure can search more prefixes in one node and may find the longest matching prefix in an internal node rather than on a leaf. Moreover, when updating the router-table, it does not need to reconstruct the table. As a by-product, the proposed data structure minimizes the time required for dynamic router-table operations, including lookup, insertion, and deletion, and also reduces the number of memory accesses. We report the results of experiments conducted to compare the proposed data structure with other structures using the benchmark IPv4 prefix database AS4637 with 219,581 prefixes.**

***Index Terms*—Classless inter domain routing, dynamic router tables, IP address lookup, longest matching prefix, multi-prefix trie.**

I. Introduction

The IP addresses of the classful routing protocol* are often fully utilized, but Classless Inter Domain Routing (CIDR) [11] provides a way to alleviate the problem. The routing entries of CIDR are pairs of $(p/l, o)$, where $p = p_0p_1 \ldots p_{l-1}*$ is a prefix formed of binary bits; l is the length of p, which is at most 32 bits for IPv4 [19] and 128 bits for IPv6 [8]; and o is an output port identifier. The prefix length of CIDR is variable; hence it can provide more IP addresses than classful routing.

An internet router classifies incoming packets into flows based on information contained in the packet headers and a table of (classification) rules called the router-table (equivalently, rule-table), which contains pairs of $(p/l, o)$. The rules are used to find the best matching prefix and determine the output port the packet should be sent to. Currently, the best known prefix matching scheme in the Internet protocol is the so-called *longest prefix matching* scheme, which compares all the prefixes in the routing table to the destination address bit-by-bit, and then finds the longest one among the set of matching prefixes. When a packet with a destination address d arrives, we can find the pair of $(p/l, o)$ in the router table if p with length l matches d, and then send the packet to the output o. However, this matching scheme causes a major bottleneck in the router. Clearly, the performance of the longest prefix matching algorithm plays an important role in routing devices.

Since static router-tables require a prohibitive update time to reconstruct a table, they are only suitable for routing environments without insertion and deletion; therefore, we concentrate on dynamic router-tables to support real-time instant updates. Interested readers may refer to [21] for more information about static router-tables, and to [25] for further details about static and dynamic router-tables. In general, four main factors affect the performance of a router-table.

1) Lookup speed: The amount of Internet traffic is so large that the speed of determining which output port an incoming packet should be forwarded to is critical.
2) Space requirement: The space requirement must be small so that the structure of the router-table can be stored in the memory.
3) Scalability: The structure of the forthcoming IPv6 routing protocol can be applied to the IPv6 router. The router-table must have the scalability to deal with routing entries for both IPv4 and IPv6.
4) Update speed: Since variability is a key feature of the Internet, router-table entries usually vary such that updates (insertion and deletion) occur frequently. Each operation must be performed efficiently without reconstructing the router table.

Designing router-tables based on different data structures has received a great deal of attention in recent years [2]–[5], [7], [10], [15]–[18], [20], [22]–[24], [26]–[29], [31], [33], [34]. One line of research focuses on designing dynamic router-tables that manipulate prefixes using contiguous intervals [4], [5], [15], [26], [28], [29], [34]. For instance, when the length of the IP address equals 5, the interval representing the prefix p=0* is $[00000, 01111] = [0, 15]$. An interval filter $[u, v]$ matches the destination address D iff $u \leq D \leq v$. Consequently, the longest matching prefix problem reduces to finding the shortest interval containing the destination address. Another line of research tries to find hardware solutions. To this end, Sarang Dharmapurikar *et al.* [9] employ Bloom filters for matching the longest prefix. Jiang *et al.* [13] propose an SRAM-based parallel multi-pipeline architecture for terabit IP lookup. Sieteng Soh *et al.* [32] utilize lexicographic ordered prefixes to reduce the off-line construction time of the Full Expansion Compression (FEC) algorithm [6] and the Compressed Next-Hop Array/Code Word Array (CNHA/CWA) [12].

The authors are with the Department of Computer Science and Information Engineering, National Cheng Kung University, No. 1, University Road, Tainan 701, TAIWAN. E-mail addresses: hsiehsy@mail.ncku.edu.tw (Corresponding author: S. Y. Hsieh); p7695130@mail.ncku.edu.tw (Y. L. Huang); p7697452@mail.ncku.edu.tw (Y. C. Yang).

*The classful routing protocol has four address classes: A, B, and C have 8-bit, 16-bit, 24-bit network-addresses, respectively, and D is for multicast.

Numerous trie-based router-table schemes have also been proposed [3], [18], [20], [22]–[24], [29]–[31], but they suffer from the following shortcomings.

I. Additional memory space is required because some nodes in these structures do not contain any routing information [18], [20], [22]–[24], [31].
II. Each node stores exactly one prefix, so accessing a node can only match one prefix. Thus, router-table operations, such as lookup, insertion, and deletion take a great deal of time [3], [18], [20], [31].
III. Lookup operations in these structures must compare the prefixes stored in the nodes from the root to a leaf, even if the longest matching prefix is in some internal node. This will increase the time required for lookup operations [3], [18], [20], [22]–[24], [29]–[31].
IV. The router-tables in [18], [20] are static, so they cannot be updated in a timely manner.

Motivated by the above observations, we propose a new data structure called a multi-prefix trie for designing dynamic router-tables. In the proposed data structure, all the nodes keep the corresponding routing information; hence each node can store more than one prefix, which reduces the number of memory accesses required for router-table operations. Moreover, a multi-prefix trie can return the longest matching prefix immediately when it is found in some internal node. This is quite different from the other data structures, which must go to a leaf in order to return the discovered longest matching prefix. The above advantages reduce the time required for router-table operations. In addition, based on the multi-prefix trie, we propose another data structure, called the index multi-prefix trie, which combines the index table with the multi-prefix trie to reduce the height of the trie and expedite router-table operations. Note that the two proposed data structures can be applied to both IPv4 and IPv6.

The remainder of this paper is organized as follows: Section II introduces the proposed multi-prefix trie. The dynamic router-table operations for the proposed data structures are described in Section III. We discuss the index multi-prefix trie and its operations in Section IV. The results of experiments conducted to compare the proposed data structures with other data structures are reported in Section V. We then summarize our findings in Section VI.

II. DESIGNING DYNAMIC ROUTER-TABLES USING A MULTI-PREFIX TRIE

In this section, we propose our data structure, called a multi-prefix trie, for designing dynamic router-tables. The proposed structure utilizes the Prefix-Tree (PT) defined in [3] as an auxiliary sub-structure. Before describing the data structure, we define some terms used throughout the paper. For a prefix $p = p_0p_1 \dots p_{l-1}*$, let $p' = p_0p_1 \dots p_i*$ for $0 \le i \le l-2$ be a *sub-prefix* of p. The *length* of a prefix p, denoted by $len(p)$, is the number of non-$*$ symbols; for example, $len(p_0p_1 \dots p_{l-1}*) = l$. The *level* of a node v in a rooted tree, denoted by $level(v)$, is the number of edges on the path from the root to v. If the last edge on the path from the root of the tree is (y, x), then y is the *parent* of x, and x is a *child* of y.

A. Prefix-trees

Each node in a prefix-tree contains exactly one prefix, so the size of the prefix-tree is equal to the size of the routing table. As shown in Figure 1, each node contains a prefix and two pointers pointing to successive tree nodes. The length of the prefix must be greater than or equal to the level where the prefix is located. Thus, in the figure, the level of the node containing 1* is 1 and len(1*)=1. For convenience, we use PT_LOOKUP, PT_INSERT and PT_DELETE, to represent the respective algorithms for lookup, insertion, and deletion operations in prefix-trees in $O(W)$ time, where W is the length of the IP address. The PT_LOOKUP algorithm can find the longest matching prefix [3].

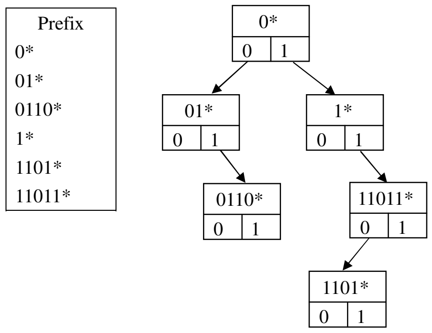


Fig. 1. A prefix-tree

B. Multi-prefix tries

A *trie* is a rooted tree data structure. A k*-stride Multi-Prefix Trie* (k-MPT), where k is the *stride* which is a positive integer, contains two types of nodes, a *primary node* (p-node) and a *secondary node* (s-node), which possess the following properties:

P1. Each p-node v contains the following fields:
   a) $0 \le t \le m$ is the number of prefixes stored in v, where $m = O(k)$.
   b) The t prefixes, denoted by $p_1(v), p_2(v), \dots, p_t(v)$, are stored in non-increasing order with $len(p_1(v)) \ge len(p_2(v)) \ge \cdots \ge len(p_t(v))$.
   c) $port(p_i(v))$, the output port of $p_i(v)$.
   d) $s\_pointer(v)$, a pointer points to a prefix-tree PT composed of s-nodes, which store prefixes of length at least $k \cdot level(v)$, but less than $k \cdot (level(v)+1)^{\dagger}$. For convenience, the tree indicated by $s\_pointer(v)$ is called the PT of v.
   e) The *content* of a p-node v can be represented simply as $(t, p_1(v), p_2(v), \dots, p_t(v), s\_pointer(v))$.

P2. The stride k is the number of bits used by a p-node to determine which branch to take. A p-node whose stride is k has 2^k children corresponding to the 2^k possible

†The s-nodes in a PT of a p-node v store prefixes of length at least $k \cdot level(v)$, but less than $k \cdot (level(v)+1)$. This can be viewed as a classification of prefixes according to their lengthes. There was another prefix partitioning technique proposed in [15] in which the prefixes in each partition may be represented using a dynamic router-table data structure.

values for the k used bits. For ease of presentation, we use $child_0(v), child_1(v), \ldots, child_{2^k-1}(v)$ to represent 2^k children corresponding to 2^k possible values from $\underbrace{00\ldots0}_{k}$ to $\underbrace{11\ldots1}_{k}$. Thus, if $k = 2$, there will be four children, $child_0(v), child_1(v)$, $child_2(v)$, and $child_3(v)$, corresponding to $00, 01, 10$, and 11, respectively.

P3. A p-node with m prefixes is said to be *full*; otherwise, it is *non-full*. An *internal p-node* is a full p-node that has children, and an *external p-node* is a p-node without any children. Note that an external p-node may be non-full.

P4. Let u and v be two consecutive p-nodes on a path in T. If there are two prefixes $p_i(u)$ and $p_j(v)$ such that $p_j(v)$ is a sub-prefix of $p_i(u)$, then $level(u) \leq level(v)$.

P5. Each s-node w has the following fields:

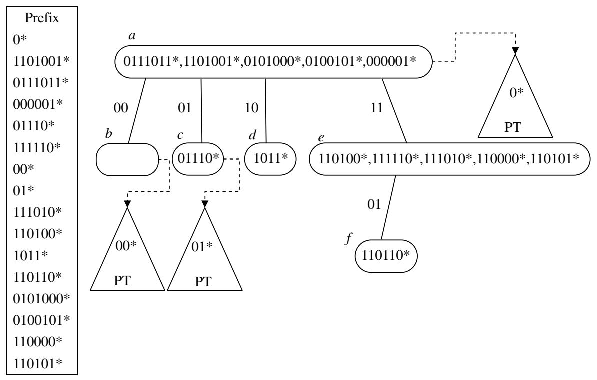
a) $p(w)$, the prefix stored in w.

b) $port(p(w))$, the output port of the prefix stored in w.

c) $left(w)$, a pointer indicating the left s-node of w if it exists; otherwise, the pointer is set as "null".

d) $right(w)$, a pointer indicating the right s-node of w if it exists; otherwise, the pointer is set as "null".

A p-node is described as *empty* if it does not contain any prefix. Figure 2 illustrates a 2-MPT in which a, b, c, d, e, and f are p-nodes. Assume that $m = 5$; hence, every p-node contains at most five prefixes. The internal p-nodes a and e are full, while the external p-nodes b, c, d, and f are non-full. The three dotted pointers represent $s\_pointers$, which point to three PTs containing prefixes 00*, 01*, and 0*, respectively. Note that, since prefix 110100* in e is a sub-prefix 1101001* in a, we have $level(a) = 0 < 1 = level(e)$.



Prefix
0*
1101001*
0111011*
000001*
01110*
111110*
00*
01*
111010*
110100*
1011*
110110*
0101000*
0100101*
110000*
110101*

Fig. 2. A 2-MPT

For a k-MPT T, the *height* of T, denoted by $h(T)$, is $\max\{level(v) \mid v \text{ is a node (p-node or s-node) in } T\}$. To traverse a path in T from the root (a p-node) to a leaf, the current p-node requires k bits to branch out to the next p-node until it arrives at the leaf. Hence, the longest path from the root to an external p-node is bound by $\lfloor \frac{W}{k} \rfloor$. Moreover, the $s\_pointer$ of an external p-node may point to a prefix-tree whose height is bounded by k. Therefore, the following lemma holds.

Lemma 1: Let W be the length of the IP address and let T be a k-MPT. Then, $h(T) \leq \lfloor \frac{W}{k} \rfloor + k$.

If we have string x of length l and string y of length n, the *concatenation* of x and y, written xy, is the string obtained by appending y to the end of x, as in $x_1 \cdots x_l y_1 \cdots y_n$. The following properties are derived from the structural characterization of k-MPT.

Lemma 2: Let u be a p-node in a k-MPT T. Then, the prefixes in u have the common sub-prefix of length kt, where t is the length of the path from the root of T to u.

Proof: Let $P = \langle v_0, v_1, \ldots, v_t \rangle$, where $v_t = u$, be the path from the root, say v_0, to u. Then, the concatenation of the indices (binary strings of length k) of the branches (v_i, v_{i+1}) for all $0 \leq i \leq t-1$ forms the common sub-prefix of the prefixes in u. Since the index of each branch is a binary string of length k and the path P has length t, the length of the common sub-prefix equals kt. **Q.E.D.**

Lemma 3: Let u be a p-node in a k-MPT T. If the prefix-tree PT of u exists, then the prefixes in PT have the common sub-prefix of length kt, where t is the length of the path from the root of T to u.

Proof: According to properties P1, P2, and P5 of the k-MPT, the result can be shown by using a method similar to that utilized to show Lemma 2. **Q.E.D.**

III. Dynamic router-table operations

A. Creating an empty k-MPT

To build a k-MPT T, we first use the following algorithm to create an empty root node and then call an insertion algorithm (described in Section III-B) to insert a new prefix. Both algorithms use an auxiliary procedure called ALLOCATE_P-NODE, which allocates the memory space to be used by a new p-node in $O(1)$ time.

```
Algorithm MPT_CREATE($T$)
1: $v :=$ ALLOCATE_P-NODE()
2: $root(T) := v$
```

B. Insertion operation

In this subsection, we present the insertion algorithm for k-MPT. The following definition is useful in our algorithms.

Definition 1: Let $S = \{(p_0 p_1 \ldots p_{l-1}*, i, j) \mid p_t \in \{0,1\}$ for $0 \leq t \leq l-1$ and $0 \leq i \leq j \leq l-1\}$. Define the function GET : $S \to \mathbb{Z}^+$ as GET$(p_0 p_1 \ldots p_{l-1}, i, j) = \sum_{r=i}^{j} p_r 2^{j-r}$. For example, GET$(0100000*, 0, 3) = (0100)_2 = 4$.

To insert a prefix $p = p_0 p_1 \ldots p_{l-1}*$ into a k-MPT, we start from the root and move downwards until we find a p-node to insert p. Suppose that v is the current p-node visited during this process. If $len(p) < k \cdot (level(v) + 1)$, then p will be inserted into the PT of v. Otherwise, if $len(p) \geq k \cdot (level(v) + 1)$, we determine whether or not node v is full and execute the following operation. Assume that the current content of v is $(t, p_1(v), p_2(v), \ldots, p_t(v), s\_pointer)$. If v is full (i.e., $t = m$) and $len(p_m(v)) < len(p)$, we replace $p_m(v)$ with p, and then insert $p_m(v)$ into $child_j(v)$, where j=GET$(p_m(v), k \cdot$

$level(v), k \cdot (level(v) + 1) - 1)$. The content of v is further updated as $(t, p_1(v), p_2(v), \ldots, p_{m-1}(v), p, s\_pointer)$. However, if v is full and $len(p_m(v)) \geq len(p)$, we insert p into $child_j(v)$, where j=GET$(p, k \cdot level(v), k \cdot (level(v) + 1) - 1)$. Otherwise, v is non-full, i.e., $t < m$, so we just insert p into v. Now, the content of v is updated as $(t+1, p_1(v), p_2(v), \ldots, p_t(v), p, s\_pointer)$. The algorithm is detailed below.

```
Algorithm MPT_INSERT($p$, $v$, $level$)
1:  if $v$ is null then
2:      $v$ :=ALLOCATE_P-NODE()
3:  if IN_PT($len(p)$, $level$) then /* $p$ should be inserted into
        the PT of $v$ */
4:      $u$ := ALLOCATE_S-NODE() /* allocate the storage
            for a new s-node */
5:      PT_INSERT($p, u, s\_pointer(v)$) /* recall that
            PT_INSERT is the insertion algorithm for
            prefix-trees */
6:  else if Is_Full($v$) then
7:      if $len(p_m(v)) < len(p)$ then /* length of the last
            prefix in $v$ < length of $p$ */
8:          replace $p_m(v)$ with $p$ in $v$
9:          arrange the prefixes in $v$ in a non-increasing order
                of their length
10:         $r$ := GET($p_m(v)$, $k \cdot level$, $k \cdot (level + 1) - 1$)
11:         $v$ := $child_r(v)$
12:         MPT_INSERT($p_m(v)$, $v$, $level + 1$)
13:     else
14:         $r$ := GET($p$, $k \cdot level$, $k \cdot (level + 1) - 1$)
15:         $v$ := $child_r(v)$
16:         MPT_INSERT($p$, $v$, $level + 1$)
17: else
18:     insert $p$ into $v$
19:     $t(v) := t(v)+1$ /* increase the number of the prefixes
            in $v$ */
20: return
```

The initial call to insert a prefix p is MPT_INSERT$(p, root, 0)$. Next, we describe how the above algorithms work (see also Figure 3).

Example 1: To insert the prefix 010* into a 2-MPT with $m = 5$, as shown in Figure 3(a), 010* can not be replaced with any prefix in p-node a (because len(000001*)=6$>$ $3 = len$(010*)) or inserted into the PT of a (because len(010*)=3 $> k \cdot (level(a) + 1) = 2(0 + 1)$). We obtain the first two bits $(01)_2$ via GET(010*,0,1) and then go to $child_1(a)$, i.e., p-node c. Since IN_PT(len(010*),1) is TRUE, 010* is inserted into the PT of c (see Figure 3(b)). Let us consider inserting another prefix 0110100*. Since the length of the last prefix 000001* in a is smaller than len(0110100*)=7, 000001* is replaced with 0110100* as shown in Figure 3(c). Next, we insert 000001* into p-node b. Since p-node b is not full, we insert the prefix into b (see Figure 3(d)).

```
Algorithm IN_PT($l, level$)
1:  if $l < k \cdot (level + 1)$ then
2:      return TRUE
3:  else
4:      return FALSE
```

```
Algorithm IS_FULL($v$)
1:  if $v$ is full then
2:      return TRUE
3:  else
4:      return FALSE
```

The following lemmas are useful for demonstrating the correctness of the MPT_INSERT algorithm.

Lemma 4: After executing Algorithm MPT_INSERT to insert the prefixes, the length of each prefix in a p-node v will be larger than that of each prefix in the PT of v.

Proof: Let p be an arbitrary prefix in v and let p' be an arbitrary prefix in the PT of v. Suppose, by contradiction, that $len(p) \leq len(p')$. Then, according to Lines 3–5 of the algorithm and Algorithm IN_PT, we have $len(p') < k \cdot (level(v)+1)$. Hence, $len(p) \leq len(p') < k \cdot (level(v)+1)$, which means p is in the PT of v. This contradicts the assumption that p is in v. **Q.E.D.**

Consider a p-node v in a k-MPT T. Let $root(T)$ be the root of T. Any p-node u on the unique path from the root of T to v is called an *ancestor* of v. If u is an ancestor of v, then v is a *descendant* of u. (Every node is both an ancestor and a descendant of itself.) If u is an ancestor of v and $u \neq v$, then u is a *proper ancestor* of v, and v is a *proper descendant* of u. The *descendent subtree rooted at* v, denoted by $T(v)$, is the tree induced by descendants of v, rooted at v.

Lemma 5: Let x and y be two prefixes that are inserted using Algorithm MPT_INSERT, where y is a sub-prefix of x. If x and y are both in p-nodes, then x and y are in the same p-node; otherwise, the level of the p-node containing x is smaller than the level of the p-node containing y.

Proof: We have the following scenarios.

Case 1: x is inserted before y. Let u be the current p-node containing x. According to the algorithm, all the proper ancestors of u must be full. Moreover, when y is inserted into a p-node using MPT_INSERT$(y, root, 0)$, y must follow the downward path from the root and encounter p-node u because $len(y) < len(x)$. There are three cases.

Case 1.1: u is full and $len(p_m(u)) < len(y)$. According to Lines 6–12 of the algorithm, y will also be inserted into u by replacing $p_m(u)$. Hence, x and y are now in the same p-node and the result holds.

Case 1.2: u is full and $len(p_m(u)) \geq len(y)$. According to Lines 13–16 of the algorithm and by the assumption that y is in a p-node, y will be inserted into a p-node whose level is larger than that of u. Hence, the result holds.

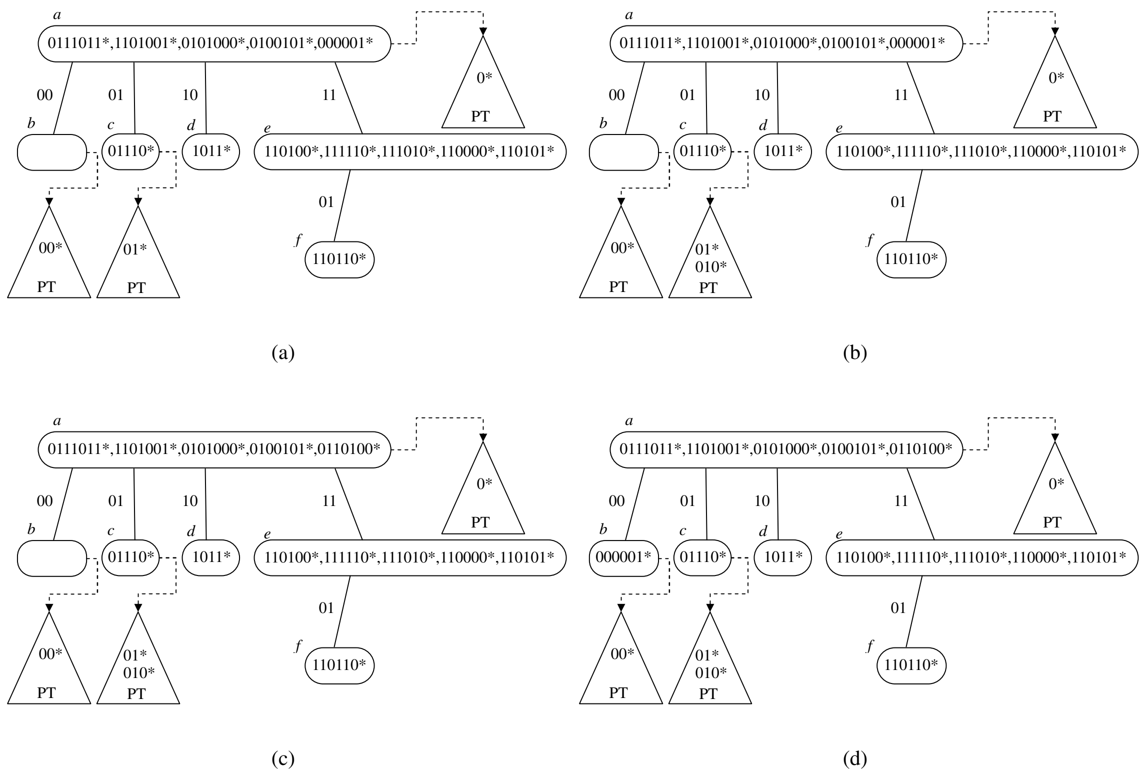


Fig. 3. Illustration of how Algorithm MPT_INSERT works. (a) The original 2-MPT. (b) 2-MPT after inserting prefix 010*. (c) To insert prefix 0110100*, the last prefix in node a is replaced with prefix 0110100*. (d) The 2-MPT after inserting prefix 0110100*.

Case 1.3: u is non-full. According to Lines 17–19, y will also be inserted into u. Hence, x and y are now in the same p-node and the result holds.

Case 2: x is inserted after y. Let v be the current node containing y. According to the algorithm, all the proper ancestors of v must be full and the length of each prefix stored in any proper ancestor of v must be at least $len(y)$. When x is inserted using MPT_INSERT($x, root, 0$), either x is replaced by some prefix in a proper ancestor of v, or it follows the downward path from the root until it encounters a p-node v. In the former case, x will be in a p-node whose level is smaller than $level(v)$; and in the latter case, we have the following three scenarios.

Case 2.1: v is full and $len(p_m(v)) < len(x)$. According to Lines 6–12 of the algorithm, x will be inserted into v by replacing $p_m(v)$. If $y = p_m(v)$, then y will be inserted into a proper descendant of v. Thus, the result holds. Otherwise, if $y \neq p_m(v)$, then x and y will be inserted into the same p-node.

Case 2.2: v is full and $len(p_m(v)) \geq len(x)$. This leads to $len(y) \geq len(p_m(v)) \geq len(x)$, which contradicts the fact that $len(y) < len(x)$ because y is a sub-prefix of x.

Case 2.3: v is non-full. According to Lines 17–19, x will be also inserted into v. Hence, x and y are now in the same p-node and the result holds. **Q.E.D.**

Theorem 1: Algorithm MPT_INSERT($p, root, 0$) can correctly insert a prefix p into a k-MPT T.

Proof: The correctness follows from Lemmas 4–5 and the correctness of Algorithm PT_INSERT.

Q.E.D.

Theorem 2: Algorithm MPT_INSERT($p, root, 0$) can be implemented to run in $O(W)$ time.

Proof: During the execution of the algorithm, we first check whether the given prefix p should be inserted into the PT of some p-node v, according to Lines 3–4. This step can be done in $O(1)$ time. If the condition holds, it takes $O(W)$ time to insert a prefix into the PT according to Line 5. Otherwise, we can check whether or not the current p-node v is full in $O(1)$ time. If v is full and $len(p_m(v)) < len(p)$, we replace $p_m(v)$ with p and insert p into the proper position in v and $p_m(v)$ into $child_j(v)$ with j=GET($p_m(v), k \cdot level(v), k \cdot (level(v)+1)-1$), according to Lines 6–12. Since $m = O(k)$, the above operation takes $O(k)$ time without considering the recursive call. If v is full and $len(p_m(v)) \geq len(p)$, we insert p into $child_j(v)$ with j=GET($p, k \cdot level(v), k \cdot (level(v)+1)-1$), according to Lines 13–16 of the algorithm. Without considering the recursive call, this takes $O(k)$ time. Otherwise, if v is non-full, the algorithm only needs $O(k)$ time to insert p into v. Since the number of the p-nodes from the root to an external p-node in a k-MPT is $O(\frac{W}{k})$ and inserting a prefix into a PT of a p-node takes $O(W)$ time, the total time complexity of the algorithm is $O(k \cdot (\frac{W}{k}) + W) = O(W)$. **Q.E.D.**

C. Lookup operation

The steps of our search algorithm, called LOOKUP, are as follows. Given the destination address DA, we search a k-MPT starting from the root in a top-down manner. When a p-node v is visited, we first search the prefixes of v starting from $p_1(v)$ to determine whether the DA matches some prefix in v. If such a prefix exists, then it is the longest matching prefix according to Properties P1(b) and P4. Otherwise, we search the corresponding prefix-tree PT and determine whether it contains a matching prefix. If a matching prefix is found, we store it and proceed with further iterations until an external p-node is visited. The algorithm is detailed below.

```
Algorithm MPT_LOOKUP($DA, v, level$)
/* $next\_hop$ is used to record the output port of the current better matching prefix, and $default\_route$ is used to record the default output port. */
 1: $level := 0$
 2: $next\_hop := default\_route$
 3: while $v \neq$ null do
 4:     if there is a prefix in $v$ that matches $DA$ then
 5:         find the longest prefix $p_i(v)$ that matches $DA$
 6:         return $port(p_i(v))$
 7:     else
 8:         $next\_hop :=$ PT_LOOKUP($DA, s\_pointer(v)$)
 9:     $r :=$ GET($DA$, $k \cdot level$, $k \cdot (level + 1) - 1$)
10:     $v := child_r(v)$
11:     $level := level + 1$
12: return $next\_hop$
```

The initial call is MPT_LOOKUP($DA, root, 0$), where $root$ is the root of the given k-MPT. We use an example to explain how the algorithm works (see also Figure 2).

Example 2: If DA=11010010, we begin searching the root a and find that 1101001* matches this address. Hence, the longest matching prefix is 1101001*. If DA = 00100110, we begin searching the root, but find that no prefix matches 00100110. Based on $s\_pointer(a)$, we go to the corresponding PT and find a matching prefix 0*. We then store the output port number of 0*, and take the first two bits 00 of DA to go to $child_0(a)$ (i.e., p-node b). No prefix in b matches 00100110, but the corresponding PT has a prefix 00* that matches 00100110. Since p-node b has no child, 00* is thus the longest matching prefix.

Theorem 3: Algorithm MPT_LOOKUP($DA, root, 0$) can correctly find the longest matching prefix for DA if such a prefix exists.

Proof: To verify that the algorithm works correctly, we use the following "loop invariant":

- At the start of each iteration of the **while** loop in Lines 3–11, the output port of either the longest matching prefix or the current best known matching prefix is found.

We need to show that 1) this invariant is true prior to the first loop iteration; 2) each iteration of the loop maintains the invariant; and 3) the invariant provides a useful property to demonstrate the correctness when the loop terminates.

Initialization: The entire k-MPT is not searched prior to the first iteration of the loop; $next\_hop$ is set as $default\_route$ and the loop invariant holds trivially.

Maintenance: To check whether each iteration maintains the loop invariant, we match DA with the prefixes $p_1(v), p_2(v), \ldots, p_t(v)$ stored in the current p-node v. First, we consider the situation where DA matches $p_i(v)$ for some $1 \leq i \leq t$ based on the following two cases.

Case 1. $next\_hop \neq default\_route$. Here, the DA must match some prefix p' in a PT of a p-node u, where $level(u) < level(v)$. According to Property P1(d) of the k-MPT, $k \cdot level(u) \leq len(p') < k \cdot (level(u) + 1)$. Note that $len(p_i(v)) \geq k \cdot level(v)$. Hence, $len(p') < len(p_i(v))$ by $len(p') < k \cdot (level(u) + 1) \leq k \cdot level(v) \leq len(p_i(v))$. Moreover, if there exists another matching prefix p'' in a p-node w or in the PT of w with $level(w) > level(v)$, then, according to Lemma 4 and Properties P1(d) and P4 of the k-MPT, $len(p'') < len(p_i(v))$. Based on the above observations and the fact that $len(p_i(v)) \geq len(p_{i+1}(v)) \geq \cdots \geq len(p_t(v))$, $p_i(v)$ is selected as the longest matching prefix and Line 6 of the algorithm will return the correct output port.

Case 2. $next\_hop = default\_route$. Here, $p_i(v)$ is the first matching prefix. If there is another matching prefix p'' in a p-node w or in the PT of w with $level(w) > level(v)$, then, according to Lemma 4 and Properties P1(d) and P4 of the k-MPT, $len(p'') < len(p_i(v))$. Moreover, because $len(p_i(v)) \geq len(p_{i+1}(v)) \geq \cdots \geq len(p_t(v))$, $p_i(v)$ is selected as the longest matching prefix and Line 6 of the algorithm will return the correct output port.

Next, we consider the situation where the DA does not match any prefix in a p-node. If the DA matches a prefix in the PT of v, then Line 8 of the algorithm will return the output port of the longest matching prefix among the prefixes in the PT. Clearly, the discovered matching prefix is currently the best one. However, if the DA does not match any prefix in the PT of v, then in Lines 9–11, we get the next k bits to determine which child of v should be visited next. Renewing v and incrementing $level$ re-establishes the loop invariant for the next iteration.

Termination: At the termination, v is null. By the loop invariant, each action of renewing $next\_hop$ will keep the output port of the current best matching prefix. Since there are no other nodes to be visited, Line 12 of the algorithm will return the output port of the longest matching prefix if such a prefix exists. **Q.E.D.**

Theorem 4: Algorithm MPT_LOOKUP$(DA, root, 0)$ can be implemented to run in $O(\frac{W^2}{k})$ time.

Proof: For each prefix $p = p_0p_1 \ldots p_{l-1}$*, we use an interval $I_p = [a, b]$ to represent p, where $a = \sum_{r=0}^{l-1} p_r 2^{W-r-1}$ and $b = \sum_{r=0}^{l-1} p_r 2^{W-r-1} + \sum_{r=l}^{W-1} 2^{W-r-1}$. For example, when $W = 8$, $I_{001*} = [32, 63]$. Note that the DA matches a prefix $[a, b]$ if the DA's decimal value is between a and b. Hence, checking whether DA matches a prefix will take $O(1)$ time. In each iteration of the while loop, we can check whether DA matches some prefix in the current p-node in $O(k)$ time. Since the number of the p-nodes from the root to an external p-node in a k-MPT is $O(\frac{W}{k})$ and lookuping a prefix in a PT takes $O(W)$ time, the overall time complexity of the algorithm is $O((k + W) \cdot \frac{W}{k}) = O(\frac{W^2}{k})$. **Q.E.D.**

D. Deletion operation

To delete a prefix $p = p_0p_1 \ldots p_{l-1}$*, we start from the root, follow the downward path, and execute the following operation. Suppose that v is the p-node being visited currently. First, we utilize Algorithm IN_PT to check whether p is in the PT of v. If p is in the PT, we simply delete p and release the storage allocation of the corresponding s-node containing p. However, if p is not in the PT of v, there are two possible scenarios depending on whether p is in v or not. We first consider the case where p is in v. Let $(t, p_1(v), p_2(v), \ldots, p_t(v), s\_pointer(v))$ be the current content of v such that $p_i(v) = p$ for some $1 \leq i \leq t$. If v is an external p-node, we remove p from v and update the content of v as $(t - 1, p_1(v), p_2(v), \ldots, p_{i-1}(v), p_{i+1}(v), \ldots, p_t(v), s\_pointer(v))$. Note that when $t - 1 = 0$ and v has no PT, i.e., $s\_pointer(v)$=null, we only need to release the storage allocation of v. However, if v is an internal p-node, we delete p first. Let $child_r(v)$ for some $0 \leq r \leq 2^k - 1$ be the child containing the longest prefix, say y, among the prefixes stored in the children of v. We insert y into v and update the content of v as $(t, p_1(v), p_2(v), \ldots, p_{i-1}(v), p_{i+1}(v), \ldots, p_t(v), y, s\_pointer(v))$. By executing the above operations, we recursively call the deletion algorithm to delete y from $child_r(v)$.

For the situation where p is not in v, we get the desired k bits from p to branch out from some child of v, and recursively call the deletion algorithm. The complete algorithm is presented in Algorithm MPT_DELETE$(p, v, level)$.

The initial call for deleting a prefix p is MPT_DELETE$(p, root, 0)$.

Example 3: The steps of the MPT_DELETE algorithm are as follows. First, we delete the prefix 01* from the 2-MPT shown in Figure 4(a). Since 01* is not in p-node a or in the PT of a, we get the first two bits $(01)_2$ and go to $child_1(a) = c$. However, 01* is in the PT of c, so we delete it from the PT (see Figure 4(b)). Next, we delete prefix 110100* from the 2-MPT shown in Figure 4(b). Since 110100* is not in p-node a or in the PT of a, we get the first two bits $(11)_2$ and go to

```
Algorithm MPT_DELETE($p, v, level$)
/* This algorithm uses two auxiliary procedures, FREE_S-NODE and FREE_P-NODE, which free the storage allocation of an s-node and a p-node, respectively, in $O(1)$ time. */
1:  if $v$ is null then
2:      output "$p$ is not found"
3:  if IN_PT($len(p)$, $level$) then
4:      PT_DELETE($p, s\_pointer(v)$)
5:      FREE_S-NODE
6:  else if $p$ is in $v$ then
7:      delete $p$ from $v$
8:      if $v$ is an external p-node then
9:          $t(v) := t(v) - 1$ /* decrease the number of the
                prefixes in $v$ */
10:         if $t(v) = 0$ and $s\_pointer(v)$=null then
11:             FREE_P-NODE($v$)
12:     else
13:         find a prefix $y$ in $child_r(v)$ s.t. $len(y) =$
                $\max\{len(p) | p \in child_i(v)$ for $0 \leq i \leq 2^k - 1\}$
14:         insert $y$ as the last prefix in $v$
15:         $v := child_r(v)$
16:         MPT_DELETE($y$, $v$, $level + 1$)
17: else
18:     $r :=$ GET($p$, $k \cdot level$, $k \cdot (level + 1) - 1$)
19:     $v := child_r(v)$
20:     MPT_DELETE($p$, $v$, $level + 1$)
21: return
```

$child_3(a) = e$. We find that 110100* is in p-node e, so we delete it (see Figure 4(c)). As shown in Figure 4(d), 110100* is replaced with prefix 110110* in p-node f, and 110110* is deleted from f. As p-node f is now empty, we release its storage allocation. Finally, we consider to delete 0110100* from p-node a, as shown in Figure 4(d). The intermediate 2-MPT is shown in Figure 4(e). We select the longest prefix among the prefixes stored in the children of a. The length of the longest prefix equals 6, but there are many prefixes of length 6 stored in the children of a. Without loss of generality, we assume that 000001* in p-node b is selected. We then insert it as the last prefix in a and delete it from b (see Figure 4(f)). Note that since the PT of v still exists after deleting 000001*, we do need not to release the storage allocation of b.

The following lemmas are useful for demonstrating the correctness of the deletion algorithm.

Lemma 6: After using Algorithm MPT_DELETE $(z, root, 0)$ to delete a prefix z from a p-node v, each prefix in v will still be longer than any prefix in the PT of v.

Proof: According to Lines 12–16 of the algorithm, the longest prefix in some child of v may be moved to v. Let x be an arbitrary prefix in the PT of v and let y be the prefix moved up to v. Since $len(x) < k \cdot (level(v) + 1)$ and $k \cdot (level(v) + 1) \leq len(y)$, $len(x) < len(y)$. **Q.E.D.**

Lemma 7: Let x and y be two prefixes in a p-node and let y be a sub-prefix of x. Then, after using Algorithm MPT_DELETE$(z, root, 0)$ to delete a prefix $z \notin \{x, y\}$, one

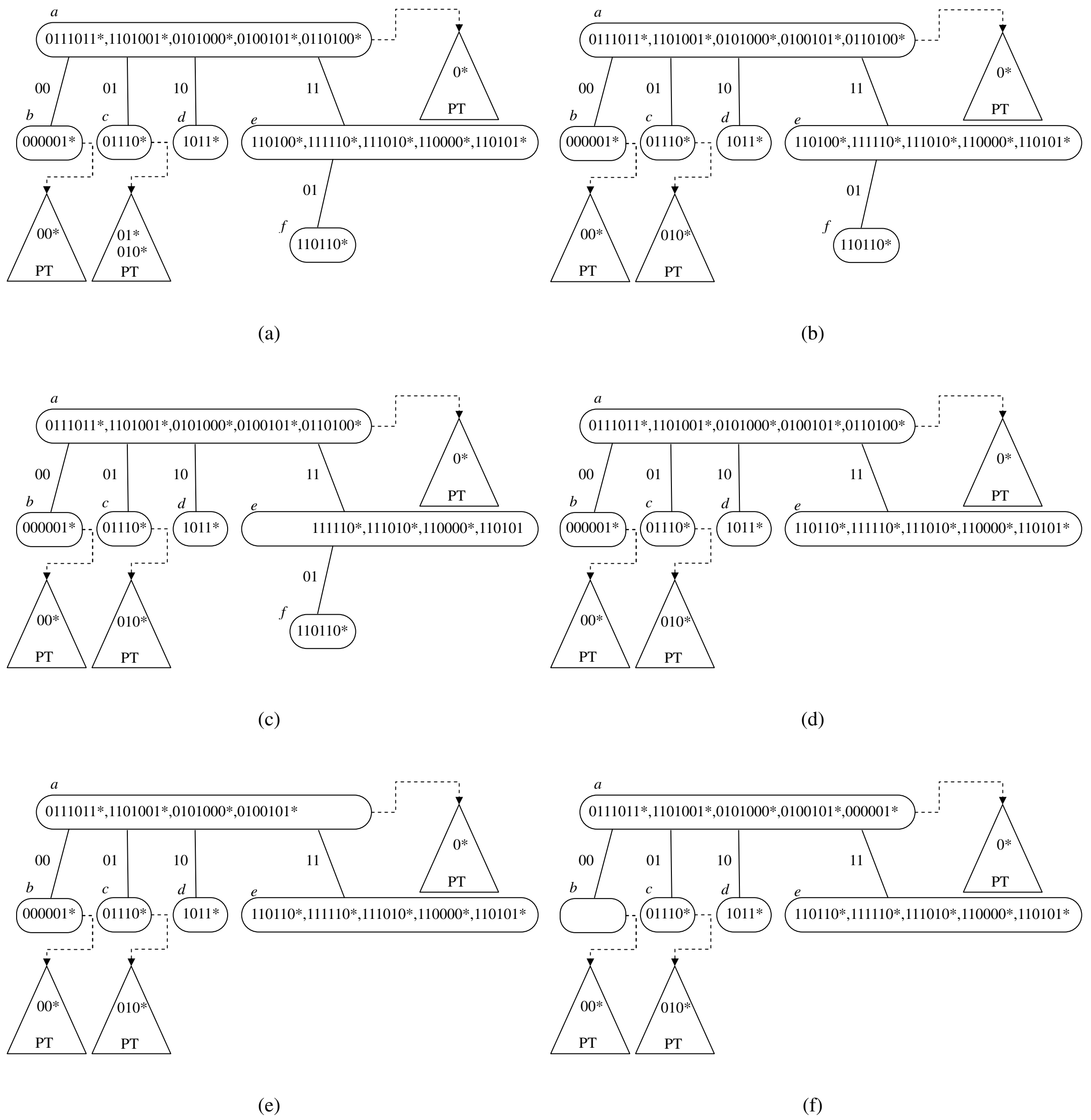


Fig. 4. The operations of the MPT_DELETE algorithm. (a) The original 2-MPT. (b) 2-MPT after deleting prefix 01*. (c) Prefix 110100* is deleted from p-node e. (d) Prefix 110110* is moved to node e and the storage allocation of f is released. (e) Prefix 0110100* is deleted from p-node a. (f) The longest prefix, 000001*, in the children of b is moved to node a such that p-node b is now empty.

of the following two conditions will hold: (1) x and y will be stored in the same p-node such that x will appear before y; or (2) the level of the p-node containing x will be smaller than the level of the p-node containing y.

Proof: Let u and v be two p-nodes containing x and y, respectively, before executing Algorithm MPT_DELETE$(z, root, 0)$. We have the following scenario.

Case 1: $u = v$. Since $len(y) < len(x)$, x appears before y in u. According to the algorithm, there are two sub-cases.

Case 1.1:The first prefix $p_1(u)$ is moved up to the parent of u. If $x = p_1(u)$, then x is moved to the parent of u after the algorithm has been executed; thus, Condition (2) holds. However, if $x \neq p_1(u)$, then x and y will still be in the same p-node after the algorithm has been executed. Hence, Condition (1) holds.

Case 1.2:The first prefix $p_1(u)$ is not moved up to the parent of u. In this sub-case, x and y are still in u and retain their relative order. Hence, Condition (1) holds.

Case 2: $u \neq v$. By Lemma 5, $level(u) < level(v)$. After executing the algorithm, either $level(u) < level(v)$ still holds or $level(u)+1 = level(v)$ and y is moved up to u. In the latter case, since $len(x) > len(y)$, Condition (1) holds according to Line 14 of the algorithm. **Q.E.D.**

Theorem 5: Algorithm MPT_DELETE$(p, root, 0)$ can correctly delete a prefix p from a k-MPT.

Proof: The correctness follows directly from the algorithm and Lemmas 6 and 7. **Q.E.D.**

Theorem 6: Algorithm MPT_DELETE$(p, root, 0)$ can be implemented to run in $O(\frac{2^k W}{k})$ time.

Proof: During the execution of the algorithm, we first check whether p is in the PT of the current p-node v (Lines 3–5) in $O(1)$ time. If p is in the PT, it takes $O(W)$ time to delete it. Otherwise, we check whether p is in v and delete it in $O(k)$ time. If p is in v, then the corresponding operations (without regard to a recursive call) described in Lines 6–16 of the algorithm can be implemented in $O(2^k)$ time because we find the longest prefix among the prefixes in $O(2^k)$ children of v. Otherwise, p is not in v. The corresponding operations (without regard to a recursive call) described in Lines 17–20 of the algorithm can be implemented in $O(k)$ time. Therefore, the total complexity is $O(2^k(\frac{W}{k}) + W) = O(\frac{2^k W}{k})$. **Q.E.D.**

IV. Index Multi-Prefix Trie

We now present a new data structure called the k-stride *Index Multiple Prefix Tree* (k-IMPT for short), which is based on the k-MPT. The k-IMPT partitions a k-MPT into several smaller k-MPTs to reduce its height. This idea has been used in many routers [14]. The partitioned k-MPTs are merged through the index table $tab[\underbrace{00\ldots0}_{\alpha}, \underbrace{00\ldots1}_{\alpha}, \cdots, \underbrace{11\ldots1}_{\alpha}]$. Given a fixed length α, the index table has 2^α entries such that the entry $tab[b_0b_1\ldots b_{\alpha-1}]$ contains a pointer, which indicates the k-MPT that stores the prefixes with the common sub-prefix $b_0b_1\ldots b_{\alpha-1}$ (see Figure 5). To execute the router-table

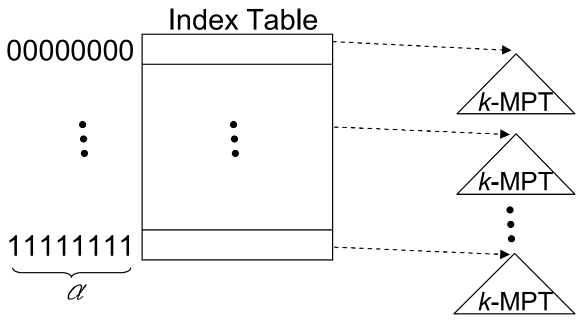


Fig. 5. A k-IMPT

operations (lookup, insertion, and deletion) in a k-IMPT, we first match the index table and then execute the operations in the corresponding k-MPT. For example, if we insert a prefix p into a k-IMPT, we first get the α bits of p to determine the index value. If $len(p) \geq \alpha$, we can insert p into the corresponding array. However, if $len(p) < \alpha$, then p corresponds to more than one index value. For instance, assume that p =101000* and $\alpha = 8$ (i.e., the first eight bits of a prefix represent the index value). Because 101000* contains 10100000, 10100001, 10100010, and 10100011, the prefix p =101000* must be inserted into the k-MPTs indicated by $tab[10100000], tab[10100001], tab[10100010]$, and $tab[10100011]$, respectively. Since storing duplicate prefixes is inefficient, we must choose an α value that can reduce the storage requirement. The algorithms for lookup and deletion work similarly.

Definition 2: Let $S = \{(p_0p_1\ldots p_{l-1}*, \alpha) |\ p_i \in \{0,1\}$ for $0 \leq i \leq l-1$ and $0 \leq l < \alpha\}$. Define the function TAKE_PREFIX_ENDPOINT : $S \rightarrow \mathbb{Z}^+$ as TAKE_PREFIX_ENDPOINT$(p_0p_1\ldots p_{l-1}, \alpha) = \sum_{r=0}^{l-1} p_r 2^{\alpha-r-1} + \sum_{r=l}^{\alpha-1} 2^{\alpha-r-1}$.

For example, TAKE_PREFIX_ENDPOINT$(101000*, 8) = (10100011)_2 = 163$. The algorithms for the lookup, insertion, and deletion operations for a prefix $p = p_0p_1\ldots p_{l-1}*$ on the k-IMPT are detailed below.

```
Algorithm IMPT_LOOKUP($\alpha, p, DA$)
1: $s$ := GET($p$, 0, $\alpha - 1$)
2: MPT_LOOKUP($DA$, $tab[s] \rightarrow root$, 0)
```

```
Algorithm IMPT_INSERT($\alpha, p$)
1: $s$ := GET($p$, 0, $\alpha - 1$)
2: if $len(p) \geq \alpha$ then
3:     if ($tab[s] \rightarrow root$) is null then
4:         MPT_CREATE($tab[s]$)
5:     MPT_INSERT($p$, $tab[s] \rightarrow root$, 0)
6: else
7:     $e$ := TAKE_PREFIX_ENDPOINT($p$, $\alpha$)
8:     for $i := s$ to $e$ do
9:         MPT_INSERT($p$, $tab[i] \rightarrow root$, 0)
```

```
Algorithm IMPT_DELETE($\alpha, p$)
1: $s$ := GET($p$, 0, $\alpha - 1$)
2: if $len(p) \geq \alpha$ then
3:     MPT_DELETE($p$, $tab[s] \rightarrow root$, 0)
4: else
5:     $e$ := TAKE_PREFIX_ENDPOINT($p$, $\alpha$)
6:     for $i := s$ to $e$ do
7:         MPT_DELETE($p$, $tab[i] \rightarrow root$, 0)
```

V. Experimental Results

We conducted experiments on the benchmark IPv4 prefix database AS4637, dated November 21, 2007 [1]. It contains 219,581 prefixes. We implemented the k-MPT and k-IMPT for various k, as well as for other data structures (for comparison with k-MPT and k-IMPT) using C++. All the experiments were performed on a 3.40GHz PC with a 512MB memory. Figure 6 shows the total prefix-population by the prefix length. In Section V-A, we select proper k and m for k-MPT and k-IMPT, and analyze their performance. Then, in Section V-B, we compare the proposed data structures with other data structures.

TABLE I
AVERAGE LOOKUP TIME (# CLOCK CYCLES) OF k-MPTS WITH DIFFERENT k AND m

$m \setminus k$-MPT	1-MPT	2-MPT	3-MPT	4-MPT	5-MPT
k	5953	5440	5529	5788	6116
$k+1$	6029	5514	5560	5642	6223
$k+2$	6362	6075	5646	5760	6162
$k+3$	6608	5872	6124	5804	6477
$2k$	6045	5643	5726	5842	6418
$2k+1$	6327	5395	5817	5902	7056
$2k+2$	6621	6062	5935	5949	6549
$2k+3$	7436	6232	6012	6016	6577
$3k$	6337	6777	6246	6107	6671
$3k+1$	6621	6471	6124	6149	6763
$3k+2$	7537	6370	6265	6233	6863
$3k+3$	7855	6890	6364	6420	7007

TABLE II
AVERAGE UPDATE TIME (# CLOCK CYCLES) OF k-MPTS WITH DIFFERENT k AND m

$m \setminus k$-MPT	1-MPT	2-MPT	3-MPT	4-MPT	5-MPT
k	9805	6999	6741	6342	6269
$k+1$	9553	7016	6986	6137	6273
$k+2$	9743	7193	6889	6157	6382
$k+3$	9515	7478	7230	6503	6365
$2k$	9688	7699	7080	6494	6434
$2k+1$	9588	6965	7117	6156	7004
$2k+2$	9435	7372	7315	6257	6593
$2k+3$	9408	7509	7284	6150	6555
$3k$	9545	7389	7378	6103	6660
$3k+1$	9459	7583	7475	6418	6634
$3k+2$	9429	7502	7822	6142	6711
$3k+3$	9305	8012	7520	6128	6786

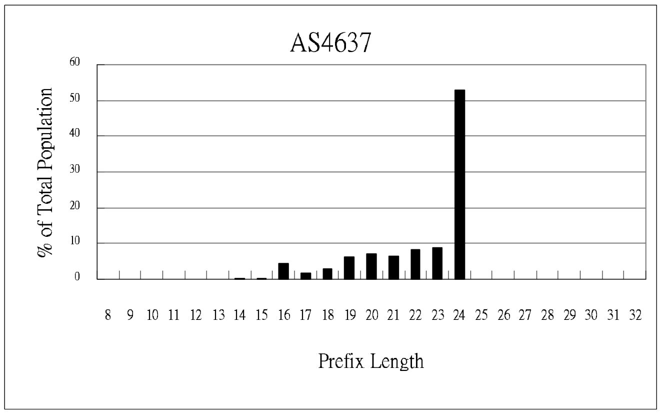


Fig. 6. Prefix-population in AS4637

A. *Selecting k and m for MPT and IMPT*

Since the number $m = O(k)$ for prefixes in the p-node of MPT may affects performance, we need to select proper values appropriate for k and m. The experimental results for the average lookup time and average update time for k-MPT with different k and m based on AS4637 are shown in Table I and Table II, respectively. We insert and delete 5,000 prefixes in AS4637 to compute the average update time. Since the length of each prefix in AS4637 is at least 8, we set $\alpha = 8$ to avoid duplicate storage of the same prefix in k-IMPT. As we can see in Table I, it can achieve the best average lookup time when $k = 2$ and $m = 2k + 1$. For average update time shown in Table II, 4-MPT has better performance. Moreover, it can achieve the best average update time when $k = 4$ and $m = 3k$. Since (average update time on $k = 4$ and $m = 2k + 1$)$-$(average update time on $k = 4$ and $m = 3k$) $= 6156 - 6103 = 53$ is small; hence, we select $m = 2k + 1$ for comparing k-MPT with other data structures in Section V-B. On the other hand, the k-IMPT has similar results. The tables of average lookup time and average update time for k-IMPT with different k and m are shown at http://algorithm.csie.ncku.edu.tw/network/MPT.htm.

We select $m = 2k+1$ for $1 \leq k \leq 5$ to compare the storage requirement by the experiment. The experimental results for the number of nodes, the height, and the storage requirements for building k-MPT and k-IMPT based on AS4637 are shown in Table III. Note that for only one k-MPT, the maximum height and average height are the same. On the other hand, since one k-IMPT T may contain several k-MPTs, the maximum height of T is defined as $\max\{h(T')|\ T'$ is a k-MPT contained in $T\}$; and the average height of T is the average height of all the k-MPTs in T. As shown in Table III, when k increases, the height decreases, but more storage is required. The reason is that larger k can store more prefixes in a p-node, which implies that more storage is required. To achieve a better performance, we need to consider the height, operating time, and storage requirement when selecting k. As shown in

TABLE III
REQUIREMENTS FOR BUILDING k-MPTS AND k-IMPTS BASED ON AS4637

	# Nodes		Max Height	Avg. Height	Storage (KB)
	# p-nodes	# s-nodes			
1-MPT	101353		24	24	1566
	88486	12867			
2-MPT	102236		13	13	2070
	66454	35782			
3-MPT	125994		10	10	2714
	59428	66566			
4-MPT	152340		9	9	3580
	60502	91838			
5-MPT	221827		9	9	4158
	51731	170096			
1-IMPT	101080		24	19.4	1562
	88374	12706			
2-IMPT	153037		13	10.74	2064
	66305	35485			
3-IMPT	125207		10	8.38	2702
	59199	66008			
4-IMPT	113394		9	7.64	3558
	60155	91186			
5-IMPT	146000		9	7.96	4158
	51419	168243			

Table III, 2-MPT requires less storage than 4-MPT. From the above observations, we conclude that the lookup performance of 2-MPT is better, and its average update time is just a little higher than that of 4-MPT. Therefore, we compare 2-MPT with the other data structures in Section V-B. We have similar observations on k-IMPT and select 2-IMPT with $m = 2k + 1$ to compare it with the other data structures in the next section.

B. Comparison with other data structures

Table IV compares the number of nodes, the number of dummy nodes, the height, and the storage requirement when implementing the AS4637 router-table with the following structures: a binary trie, an LC-trie [18], a modified LC-trie [20], a prefix-tree [3], a Dynamic Tree BitMap (DTBM) [30], Fixed-Stride Trie (FST) [22], 2-MPT, and 2-IMPT. For the multi-bit trie based structure, we adopted the k-FST, a fixed-stride trie with the level k. We applied the algorithm proposed by Sahni and Kim [22] to compute the cost and the best stride of k-FST for $3 \leq k \leq 7$. In our experiment, 7-FST is superior than the other k-FSTs ($3 \leq k \leq 6$) with respect to the worst-case lookup time, number of nodes, number of dummy nodes, storage and update time. Moreover, the average lookup time is competitive to other choices of k. Therefore, we select 7-FST to compare with the proposed data structures. Note that the binary trie, the LC-trie, the modified LC-trie, DTBM, and 7-FST contain dummy nodes, which increases the storage cost substantially. Moreover, the binary trie, the LC-trie, the modified LC-trie, the prefix-tree, DTBM, and 7-FST have more nodes than our data structures. The storage requirement of the LC-trie and the modified LC-trie need to store information about the next hop, the prefix, as well as skip and branch operations in each node; hence, their storage requirements are greater than those of the other structures. Specifically, the height (both the maximum height and the average height) of 2-MPT (2-IMPT) is less than that of the other structures except for DTBM and 7-FST, which reduces the number of memory accesses required for router-table operations. Although the height of DTBM (7-FST) is smaller than our data structures, the lookup time of our data structures is competitive to DTBM and 7-FST in the experiment.

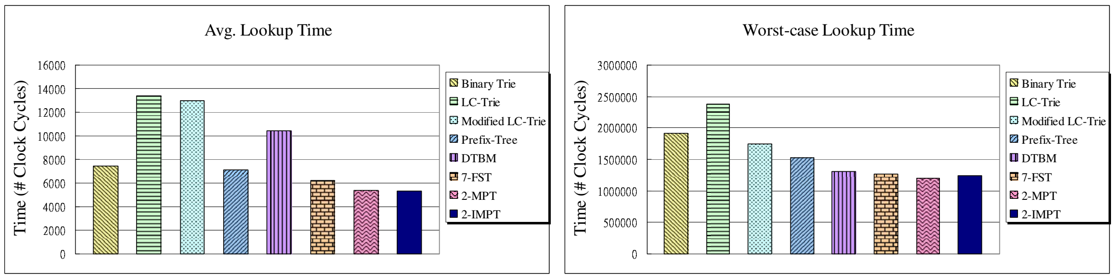
Since each memory access requires a great deal of time, the number of memory accesses affects the operating time. Table V, Figure 7, and Figure 8 compare the performance of the lookup operations of the different data structures. Both 2-MPT and 2-IMPT require fewer memory accesses than the other structures except for DTBM and 7-FST, which implies that the average lookup time also shorter than the compared structures. One key reason is that our data structures can return the longest matching prefix when it is found in an internal node, without going to a leaf to return. Although the DTBM (respectively, 7-FST) requires fewer memory accesses than 2-MPT (respectively, 2-MPT and 2-IMPT), 2-MPT and 2-IMPT are faster in terms of lookup time. The reasons are as follows. (1) When a node v is visited in the DTBM, an auxiliary function is called to determine whether there is a matching prefix matches the destination address. If such a prefix exists, then it will be stored as the currently best matching prefix. Moreover, the corresponding next hop is also kept. (2) When a node is visited in the 7-FST, the corresponding stride need to

TABLE IV
STRUCTURE COMPARISON OF DIFFERENT DATA STRUCTURES

	# Nodes		# Dummy Nodes	Max. Height	Avg. Height	Storage (KB)
	# p-nodes	# s-nodes				
Binary Trie	523793		305408	32	32	2558
LC-Trie	366413		27554	14	14	3847
Modified LC-Trie	366413		27554	14	14	3847
Prefix Tree	219581		0	29	29	2144
DTBM	226673		51893	10	10	2012
7-FST	673576		379155	7	5.2	2631
2-MPT	102236		0	13	13	2070
	66454	35782				
2-IMPT	101790		0	13	10.74	2064
	66305	35485				

TABLE V
LOOKUP TIME AND THE NUMBERS OF MEMORY ACCESSES FOR DIFFERENT DATA STRUCTURES

	Time (# Clock Cycles)		# Memory Accesses	
	Avg.	Worst	Avg.	Worst
Binary trie	7445	1919241	21.65	32
LC-trie	13352	2385763	10.94	29
Modified LC-trie	12983	1748730	7.87	16
Prefix-tree	7107	1525282	20.64	31
DTBM	10396	1301537	7.47	11
7-FST	6199	1267035	4.06	7
2-MPT	5395	1203166	7.61	25
2-IMPT	5334	1245514	7.07	25



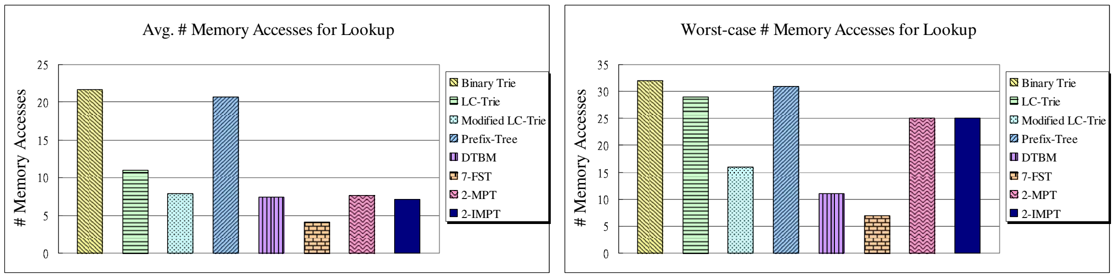
(a) (b)

Fig. 7. The average lookup time and worst-case lookup time for different data structures

be computed in order to go to the next node. These operations take longer time. Table VI, Figure 9, and Figure 10 show the performance comparison of updating 5,000 prefixes in AS4637 with different dynamic data structures. Since the LC-trie, modified LC-trie, and FST are static, the whole table must be reconstructed when inserting or deleting a prefix; hence, the above three data structures are excluded from the comparison. As shown in the table and figures, both 2-MPT and 2-IMPT require shorter update time. Since the binary trie may insert or delete a longest prefix, the memory access is 32 for IPv4 (128 for IPv6). Therefore, the update time of the binary trie is larger than our data structures. Moreover, the update time and the number of memory accesses for the binary trie and the prefix-tree are almost the same because, in both structures, the insertion and deletion of many routing entries in AS4637 often follow downward paths to leaves in order to execute the operations. Note that the update time and the number of memory accesses for the binary trie and the prefix-tree are larger than 2-MPT and 2-IMPT. Although the number of memory accesses of DTBM is less than 2-MPT and 2-IMPT, the update time of DTBM is larger than 2-MPT and 2-IMPT. The reason is that DTBM needs more complex operations to perform a deallocation when deleting a prefix.

VI. CONCLUDING REMARKS

We have proposed two new data structures, MPT and IMPT, for dynamic router-table design. Since the structures do not contain dummy nodes, they require less storage and they are not as high as other trees. In addition, because of the lower height, they require fewer memory accesses for router-

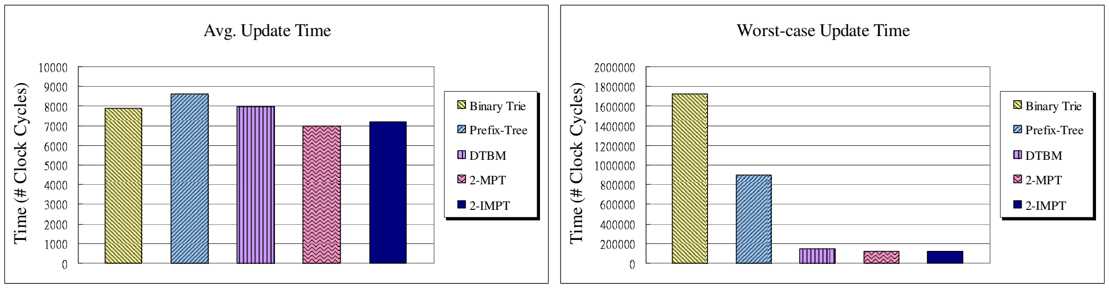


(a) (b)

Fig. 8. The average number memory accesses and the worst-case number of memory accesses for different data structures

TABLE VI
THE UPDATE TIME AND THE NUMBER OF MEMORY ACCESSES FOR DIFFERENT DATA STRUCTURES

	Time (# Clock Cycles)		# Memory Accesses	
	Avg.	Worst	Avg.	Worst
Binary trie	7889	1723333	19.25	32
Prefix-tree	8640	895900	18.3	30
DTBM	7975	148062	5.39	9
2-MPT	6965	118847	8.49	16
2-IMPT	7213	120020	8.48	54



(a) (b)

Fig. 9. The average update time and the worst-case update time for different data structures

table operations. The simulation results reveal that our data structures are superior to those trie-based data structures in respect of storage, IP lookup time, and update time.

Table VII summarizes the structural properties of different trie-based data structures. The binary trie, the (modified) LC-trie and DTBM contain dummy nodes, so it is necessary to go to a leaf to find the longest matching prefix. Our proposed data structures do not suffer from this limitation.

Our future work attempts to reduce the number of memory accesses and the memory requirement of the proposed data structures while preserving fast updating.

ACKNOWLEDGMENT

The authors are deeply appreciative for the comments and suggestions given by the editor and the anonymous referees.

REFERENCES

[1] BGP Table obtained from http://bgp.potaroo.net/.

[2] A. Basu and G. Narlikar, "Fast incremental updates for pipelined forwarding engines," *IEEE/ACM Transaction on Networking*, vol. 13, no. 3, pp. 690–703, June 2005.

[3] M. Berger, "IP lookup with low memory requirement and fast update," in *Proceedings of IEEE High Performance Switching and Routing*, pp. 287–291, June 2003.

[4] Y. K. Chang, "Simple and fast IP lookups using binomial spanning trees", *Computer Communications*, vol. 28, no. 5, pp. 529–539, Mar. 2005

[5] Y. K. Chang and Y. C. Lin, "Dynamic segment trees for ranges and prefixes", *IEEE Transactions on Computers*, vol. 56, no. 6, pp. 769–784, June 2007

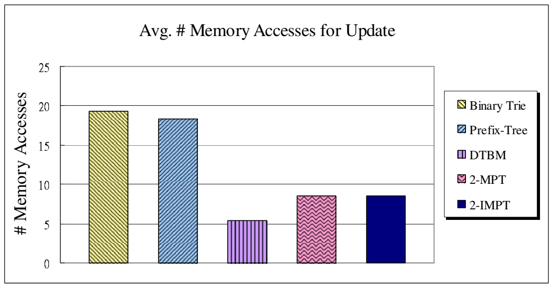
[6] P. Crescenzi, L. Dardini, and R. Grossi, "IP address lookup made fast and simple," in *Proceedings of Seventh Ann. European Symp. Algorithms*, Technical Report TR-99-01, Univ. of Pisa, 1999.

[7] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," in *Proceedings of ACM SIGCOMM*, pp. 3–14, 1997.

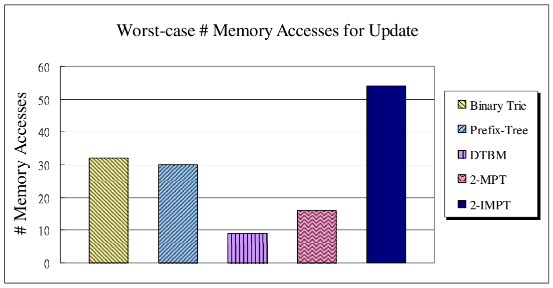
[8] S. Deering and R. Hinden, "Internet protocol version 6 (IPv6) specification", RFC 1883, Dec. 1995.

[9] S. Dharmapurikar, P. Krishnamurthy and D. E. Taylor, "Longest prefix matching using bloom filters", *IEEE/ACM Transactions on Networking*, vol. 14, no. 2, pp. 397–409, Apr. 2006

[10] W. Doeringer, G. Karjoth, and M. Nassehi, "Routing on longest-



(a)



(b)

Fig. 10. The average number of memory accesses and the worst-case number of memory accesses for different data structures

TABLE VII
SUMMARY OF PROPERTIES FOR DIFFERENT DATA STRUCTURES

	Dummy node	Table	Height	Compare-to-Leaf[a]	Backtracking[b]
Binary Trie	Y	dynamic	$\leq W$	Y	N
LC-trie	Y	static	$\leq W$	Y	Y
Modified LC-Trie	Y	static	$\leq W$	Y	N
Prefix-Tree	N	dynamic	$\leq W$	Y	N
DTBM	Y	dynamic	$\leq \lfloor \frac{W}{k} \rfloor$	Y	N
7-FST	Y	static	$\leq k$	Y	N
MPT	N	dynamic	$\leq \lfloor \frac{W}{k} \rfloor + k$	N	N
IMPT	N	dynamic	$\leq \lfloor \frac{W}{k} \rfloor + k$	N	N

[a]The structure must compare the prefixes stored in the nodes from the root to a leaf, even if the longest matching prefix is in some internal node.

[b]The prefix of the leaf in this structure does not match the destination address, so it must backtrack from the leaf to the upper nodes to find the longest matching prefix.

matching prefixes," *IEEE/ACM Transactions on Networking*, vol. 4, no. 1, pp. 86–97, 1996.

[11] V. Fuller, T. Li, J. Yu and K. Varadhan, "Classless inter-domain routing (CIDR): an address assignment and aggregation strategy," RFC 1519, Sep. 1993.

[12] N. F. Huang and S. M. Zhao, "A novel IP-routing lookup scheme and hardware architecture for multigigabit switching routers," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, pp. 1093– 1104, June 1999.

[13] W. Jiang, Q. Wang, and V. Prasanna, "Beyond TCAMs: an SRAM-based parallel multi-pipeline architecture for terabit IP lookup," in *Proceedings of IEEE INFOCOM*, pp. 1786–1794, Apr. 2008.

[14] B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search," in *Proceedings of IEEE INFOCOM*, pp. 1248–1256, Apr. 1998.

[15] H. Lu, K. S. 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

[16] J. H. Mun, H. Lim and C. Yim, "Binary search on prefix lengths for IP address lookup," *IEEE Communications Letters*, vol. 10, no. 6, pp. 492–494, June 2006

[17] S. Nilsson and G. Karlsson, "Fast address look-up for Internet routers," *IEEE Broadband Comm.*, pp. 11–22, 1998.

[18] S. Nilsson and G. Karlsson, "IP-address lookup using LC-trie," *IEEE Journal on selected Areas in Communications*, vol. 17, no. 6, pp. 1083–1092, June 1999.

[19] J. Postel, "Internet protocol darpa internet program protocol specification," RFC791, Sep. 1981.

[20] V. C. Ravikumar, R. Mahapatra, J. C. Liu, "Modified LC-trie based efficient routing lookup," in *Proceedings of the 10th IEEE International Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 177–182, Oct. 2002.

[21] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, "Survey and taxonomy of IP address lookup algorithms," *IEEE Network*, vol. 15, pp. 8–23, 2001.

[22] S. Sahni and K. S. Kim, "Efficient construction of fixed-stride multibit tries for IP lookup," in *Proceedings of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems*, pp. 178–184, 2001.

[23] S. Sahni and K. S. Kim, "Efficient construction of variable-stride multibit tries for IP lookup," in *Proceedings of IEEE Symp. Applications and the Internet (SAINT)*, pp. 220–227, 2002.

[24] S. Sahni and K. S. Kim, "Efficient construction of multibit tries for IP lookup," *IEEE/ACM Transaction on Networking*, vol. 11, no. 3, pp. 650–662, 2003.

[25] S. Sahni, K. S. Kim, and H. Lu, "Data structures for one- dimensional packet classification using most-specific-rule matching," *International Journal of Foundations of Computer Science*, vol. 14, no. 3, pp. 337–358, 2003.

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

[27] S. Sahni and K. S. Kim, "Efficient construction of pipelined multibit-trie router-tables", *IEEE Transactions on Computers*, vol. 56, no. 1, pp. 32–43, Jan. 2007.

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

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

[30] S. Sahni and H. Lu, "Dynamic tree bitmap for IP lookup and update," in *Proceedings of the Sixth International Conference on Networking* (ICN'07), pp.79, 2007.

[31] K. Sklower, "A tree-based packet routing table for Berkeley unix," in *Proceedings of Winter Usenix Conf*, pp. 93–99, 1991.

[32] S. Soh, L. Hiryanto and S. Rai,"Efficient prefix updates for IP router using lexicographic ordering and updatable address set," *IEEE Transactions on Computers*, vol. 57, no. 1, pp. 110–125, Jan. 2008

[33] V. Srinivasan and G.Varghese, "Faster IP lookups using controlled prefix expansion," *ACM Transactions on Computer Systems*, pp. 1–40, Feb. 1999.

[34] P. R. Warkhede, S. Suri, and G. Varghese, "Multiway range tree: scalable

IP lookup with fast updates," *Computer Network*, vol. 44, no. 3, pp. 289–303, Feb. 2004.

PLACE PHOTO HERE

Sun-Yuan Hsieh received the PhD degree in computer science from National Taiwan University, Taipei, Taiwan, in June 1998. He then served the compulsory two-year military service. From August 2000 to January 2002, he was an assistant professor at the Department of Computer Science and Information Engineering, National Chi Nan University. In February 2002, he joined the Department of Computer Science and Information Engineering, National Cheng Kung University, and now he is a full professor. He received the 2007 K. T. Lee Research Award, President's Citation Award (American Biographical Institute) in 2007, the Engineering Professor Award of Chinese Institute of Engineers (Kaohsiung Branch) in 2008, the National Science Council's Outstanding Research Award in 2009, and Distinguished Professor Award at National Cheng Kung University in 2010. His current research interests include design and analysis of algorithms, fault-tolerant computing, bioinformatics, parallel and distributed computing, and algorithmic graph theory.

PLACE PHOTO HERE

Yi-Ling Huang received the B.S. degree in Computer Science and Information Engineering from National Changhua University of Education, Taiwan, in 2006, and the M.S. degree in Computer Science and Information Engineering from National Cheng Kung University, Taiwan, in 2008. In July 2008, she joined the network operations laboratory of Chunghwa Telecom Laboratories, and now she is an associate researcher. Her research interests include IP lookup and packet classification algorithms, programmable routers, the design and analysis of scalable searching algorithms and architectures, and geographic information system algorithms.

PLACE PHOTO HERE

Ying-Chi Yang received the B.S. degree in the Department of Computer Science and Engineering from National Chung-Hsing University, Taiwan, in 2008. She is currently working toward the M.S. degree in computer science and information engineering at National Cheng Kung University, Taiwan. Her research interests include IP routing and searching algorithms.