A Multi-index Hybrid Trie for IP Lookup and Updates

Chia-Hung Lin, Chia-Yin Hsu, and Sun-Yuan Hsieh *Senior Member, IEEE*

Abstract—High-performance routers require high-speed IP address lookup to achieve wire-speed packet forwarding. This study proposes a new data structure, the Multi-Index Hybrid Trie (MIHT), for dynamic router table designs. This data structure was constructed by combining the useful characteristics of the B^+ tree and priority trie. IP lookup operations can be performed efficiently by associating each prefix with a key value in the MIHT. Furthermore, because the required tree height and number of prefixes were reduced, dynamic router table operations were performed efficiently using the MIHT. To reduce the memory requirement, each prefix stored its corresponding suffix in a node of the MIHT, rather than storing a full prefix. Experiments using IPv4 and IPv6 routing databases indicated that the proposed data structure has efficient memory usage and performs well for lookup, insertion, and deletion operations. This study reports the results of the experiments performed to compare the proposed data structure with other structures using the benchmark IPv4 and IPv6 prefix databases AS1221, AS4637, AS6447, AS65000, AS1221*, and AS6447* with 407,067, 219,581, 417,995, 406,973 , 12,155, and 12,278 prefixes, respectively, where AS1221* and AS6447* are IPv6 BGP routing tables.

Index Terms—Classless inter domain routing (CIDR), dynamic router tables, IP address lookup, longest matching prefix, multi-index hybrid trie.

1 Introduction

Internet traffic growth is the result of Internet applications such as real-time entertainment and P2P file-sharing. Therefore, to maintain good quality of service for the Internet, Internet routers resolve issues such as link speed, data throughput, and packet forwarding rate. Because there are solutions for issues regarding link speed and data throughput, this study focuses on the packet forwarding rate. Internet routers consult the destination address of each packet received and perform IP lookups in their router tables to determine the next hop for packets. Because of the IP address shortage problem, classless inter domain routing (CIDR) [10] has replaced previous addressing architectures, such as the classful routing protocol. The CIDR architecture allows prefixes of arbitrary length and address aggregation at arbitrary levels. Each routing entry of CIDR constitutes a pair of (p/l, o), where p=$p_0p_1 \ldots p_{l-1}$* is a prefix formed by binary bits, l is the length of p, and o is an output port identifier. Let W denote the maximum possible length of a prefix. For IPv4 [21], W=32, and for IPv6 [6], W=128. Although the use of CIDR has reduced the size of router tables, IP lookups in routers have become more complex because they must search the router table for the longest matching prefix (LMP) that matches the destination address of the incoming packet to determine the most specific route. To determine the LMP, the operation first compares all N prefixes in the routing table to the destination address bit-by-bit, and thereafter determines the longest in the set of matching prefixes. However, this matching scheme for determining the LMP has the time complexity $O(N)$ and is not scalable.

Because insert/delete operations for static router tables are batched, the entire router table must be rebuilt when a single prefix is inserted/deleted, causing a negative impact on the lookup performance. Therefore, this study focuses on dynamic router tables to support real-time instant updates. Numerous trie-based router table schemes have been proposed, but they have the following shortcomings:

- Extra memory space is required because some nodes in these structures do not contain routing information [7], [20], [22], [25], [27], [28].
- The tree height is bound by $O(W)$, possibly causing the lookup operation to traverse up to the W nodes [2], [14], [20].
- In the search path of a lookup operation starting from the root, there are too many prefixes that do not match the given destination address [2], [11], [14].
- The router tables in [7], [20], [22], [25], [28] are static and cannot be updated in a timely manner.

To reduce the time required by router table operations,

- *This research was supported in part by National Science Council, Taiwan under grant NSC 100-2221-E-006-116-MY3.*
- *Chia-Hung Lin is with the Department of Computer Science and Information Engineering, National Cheng Kung University, No. 1, University Road, Tainan 701, Taiwan, R.O.C. E-mail: nick@csie.ncku.edu.tw.*
- *Chia-Yin Hsu is with the Department of Computer Science and Information Engineering, National Cheng Kung University, No. 1, University Road, Tainan 701, Taiwan, R.O.C. E-mail: sam_751004@hotmail.com*
- *Sun-Yuan Hsieh is with the Department of Computer Science and Information Engineering and Institute of Manufacturing Information Systems, National Cheng Kung University, No. 1, University Road, Tainan 701, Taiwan, R.O.C. E-mail: hsiehsy@mail.ncku.edu.tw.*

Digital Object Indentifier 10.1109/TPDS.2013.214

this study developed a novel data structure, the multi-index hybrid trie (MIHT), which combines the best features of the B^+ tree and priority trie to design dynamic router tables. The proposed data structure preserves the advantages of the B^+ tree and the trie-based data structure and avoids the mentioned shortcomings. By associating each prefix with a key value, we transformed the problem of searching for an LMP into a problem of searching for a corresponding index. Based on this transformation, the height of the MIHT is less than W, accelerating lookup speed and update operations. Furthermore, the superfluous bits of each prefix do not have to be saved in the proposed data structure, further reducing the memory requirement. For most instances of updating, node splitting/merging does not occur, meaning that update operations only occur in tries with a small size. In addition, based on the MIHT, we propose another data structure, the *partitioning multi-index hybrid trie* (PMIHT), to reduce the tree height and expedite router table operations. The two data structures proposed in this study can be applied to both IPv4 and IPv6.

The remainder of this study is organized as follows: Section 2 reviews related studies. Section 3 introduces the proposed MIHT data structure. The dynamic router table operations for the proposed data structure are described in Section 4. Section 5 examines the PMIHT and its operations. Section 6 presents the experimental results that compare the proposed data structures with other solutions using IPv4 and IPv6 router tables. Section 7 provides the concluding remarks.

2 Related Work

Determining the longest matching prefix for IP lookup is more complex than exact matching because it involves two dimensions: length and value [23]. Previous methods have formulated the IP lookup problem as a range inclusion problem [3], [5], [14], [18], [19], [24], [29]. For an example of the length five IP address, the range representation of prefix 10* is [10000, 10111]=[16, 23]. The range $[e, f]$ matches the destination address DA iff $e \leq DA \leq f$. Some schemes define the rule to compare two prefixes. Therefore, performing binary searches in lookup operations is a feasible method [4], [15], [16], [30], [31]. However, because of the enclosure property between prefixes, a binary search in a sorted list of prefixes can cause failure. For instance, for $P_1 = 10$* and $P_2 = 1011$*, because the range of prefix P_2 is in range of prefix P_1, P_1 is an enclosure (enclosure prefix) of P_2. When the prefixes are not enclosure prefixes of each other, they are disjoints (disjoint prefix). To solve this problem, Yazdani and Min [30] distributed all of the included prefixes in a subtree rooted at the enclosure. Chang [4] generated auxiliary prefixes that inherit the routing information of enclosure prefixes. Lim et al. [15] proposed a *disjoint prefix tree* (DPT) that pushes enclosure prefixes to leaves. The *multiple balanced prefix tree* (MBPT) was proposed in [16]. Multiple balanced trees only have disjointed prefixes. Numerous trie-based router table schemes have been proposed [2], [7], [8], [11], [12], [14], [20], [22], [25]–[27]. The patricia trie [27] removes the one-way dummy nodes of binary tries to reduce the number of intermediary nodes. Nilsson et al. [20] proposed *level compressed trie* (LC-trie), which combines path compression and level compression to reduce tree height. The modified LC-trie [22] improves on the backtracking disadvantage of the traditional LC-trie. The prefix tree [2] is a combination between a trie and a tree, and in each node a comparison is performed similar to that in a tree. This structure does not contain a dummy node to reduce the memory requirement. The lulea algorithm [7] combines multibit tries with compression and bitmap. The structure is divided into three levels in a 16-8-8 pattern. The priority trie [14] removes the dummy nodes in a binary trie by relocating the longest prefixes in the subtrie that are rooted by the dummy nodes. The tree bitmap (TBM) proposed by Eatherton et al. [8] is similar to the lulea algorithm and uses multibit trie and bitmap. In TBM, leaf pushing is avoided, supporting the incremental update. The dynamic tree bitmap (DTBM) [26] is a variant of TBM that enables the enhanced performance of update operations. Hsieh et al. [11] proposed the multi-prefix trie (MPT) and indexed multi-prefix trie (IMPT) for a dynamic router table design, where each node can store more than one prefix and can return LMP immediately when found for some internal nodes for lookup operations, reducing the number of memory accesses.

3 The Proposed Data Structure

In this section, a new data structure, MIHT, for designing dynamic router tables is proposed. First, it is necessary to define terms that are used in the remainder of this study. For a prefix $p = p_0p_1 \dots p_{l-1}$*, let $q = p_ip_{i+1} \dots p_{l-1}$ for $0 \leq i \leq l-1$ be a suffix of p. In addition, p is a suffix of itself. The *length* of a prefix p, denoted by $len(p)$, is the number of non-$*$ symbols. For example, $len(p_0p_1 \dots p_{l-1}*) = l$.

An *undirected graph* (*graph*) $G = (V, E)$ is a pair consisting of the *vertex set* V and an *edge set* E, where V is a finite set and E is a subset of $\{(u, v) |\ (u, v)$ is an unordered pair of the distinct elements of $V\}$. A *tree* is a connected, acyclic, and undirected graph. A *rooted tree* is a tree in which one of the vertices is distinguished from the others. The distinguished vertex is the *root* of a tree. The *level* of the node v in a rooted tree, denoted by $level(v)$, is the number of edges in the path from the root to v.

Definition 1. Let $p = p_0p_1 \dots p_{l-1}$* be a prefix, and let $k \leq l$ be an integer. The *k-prefix key* of p, denoted by $prefix\_key_k(p)$, is the value of $(p_0p_1 \dots p_{k-1})_2$. The *k-suffix* of p, denoted by $suffix_k(p)$, is defined as $suffix_k(p) = p_kp_{k+1} \dots p_{l-1}$, where $0 \leq k \leq l$.

Example 1. $prefix\_key_4(00010*) = prefix\_key_4(00011*) = (0001)_2 = 1$ and $suffix_4(00010*) = 0*$. Note that both $prefix\_key_4(01^*)$ and $prefix\_key_4(010^*)$ are undefined because $len(01*) = 2 < 4$ and $len(010*) = 3 < 4$.

TABLE 1
4-Prefix keys of selected prefixes

Prefix#	Prefix	Next-hop	$prefix\_key_4(.)$	$suffix_4(.)$
P1	00010∗	A	1	0∗
P2	00011∗	B	1	1∗
P3	00111∗	C	3	1∗
P4	0011110∗	D	3	110∗
P5	010110∗	E	5	10∗
P6	011100∗	F	7	00∗
P7	01∗	G	-	-
P8	100101∗	H	9	01∗
P9	1000111∗	I	8	111∗
P10	10010∗	J	9	0∗
P11	010∗	K	-	-
P12	0110100∗	L	6	100∗
P13	100000∗	M	8	00∗
P14	001011∗	N	2	11∗
P15	10001∗	O	8	1∗

B^+ tree is a generalization of a binary search tree in that a node can have more than two children. A B^+ tree of order m is an ordered tree which satisfies the following properties:

- Each node has at most m children.
- Each node, except the root, has at least $\frac{m}{2}$ children.
- The root has at least 2 children.
- All leaves occur on the same level.
- For a B^+ tree, stores all the satellite information in the leaves and stores only keys and child pointers in the internal nodes.

The proposed data structure uses the priority trie defined in [14] as an auxiliary substructure, which is briefly described in the following subsection.

3.1 Priority Tries

The priority trie is proposed to store loner prefixes in higher-level nodes and shorter prefixes in lower-level nodes. In a priority trie, each node store routing information to improve the scalability of memory usage. The data structure is similar to the binary trie [§] that each node has at most two children (left child and right child) and the branch is based on the address bits. Each node in a priority trie contains one prefix, causing the size of the priority trie to be equal to that of the routing table. Two types of nodes are in a priority trie: *priority nodes* and *ordinary nodes*. If p is a prefix stored in node v of a priority trie with $len(p) = level(v)$, v is an ordinary node and p is an ordinary prefix. Otherwise, v is a priority node and p is a priority prefix. Note that, we can construct the priority trie by inserting the prefixes in random order since the priority trie is a dynamic structure. Assume that we insert the prefix set in order to construct priority trie associated with Table 1. We first construct a root to insert the first prefix P_1. Then we insert P_2 starting from the root. Since the root is not "NULL", we check whether $len(P_2) \leq len(P_1)$, or $len(P_2) > len(P_1)$ and P_2 doesn't match P_1. If the condition holds, we insert P_2 to next level. Because the 1st bit of P_2 is 0, we insert P_2 into the left child of the root node. With the same processing, we insert P_3 into the left child of the node which storing P_2. For inserting prefix P_4, the above condition holds until comparing P_4 with P_3. Since $len(P_4) > len(P_3)$ and P_4 matches P_3, hence we replace P_3 with P_4 and insert P_3 into next level. According the 3rd bit of P_3 is 1, we then insert P_3 into right child of the node which storing P_4. Constructing process continuing until the whole prefixes insert into priority trie. As shown in Fig. 1, each node contains a prefix and two pointers that pointed to successive tree nodes. The terms PRIORITY_LOOKUP, PRIORITY_INSERT, and PRIORITY_DELETE represent the algorithms for lookup, insertion, and deletion operations, respectively, in a priority trie in $O(h)$ time, where h is the height of a priority trie with N prefixes [2]. The PRIORITY_LOOKUP algorithm can determine the LMP, and when DA matches prefix p at a priority node, p is LMP.

Definition 2. Let $PT[i]$ for $i = 0, \ldots, 2^k - 1$ be the priority trie that is constructed using the suffix $suffix_k(p)$, where $prefix\_key_k(p) = i$. Specifically, let $PT[-1]$ be the priority trie containing all of the prefixes p with $len(p) < k$.

Example 2. Because $prefix\_key_4$(00011*)=$prefix\_key_4$(00 010*)=1, the two suffixes $suffix_4$(00011*)=1* and $suffix_4$(00010*)=0* are stored in $PT[1]$. Note that 01* and 010* are stored in $PT[-1]$ because len(01*)=2 $<$ 4 and len(010*)=3 $<$ 4.

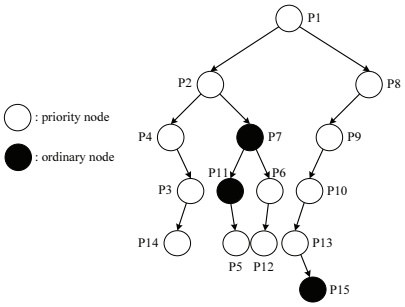


Fig. 1. A priority trie which associated with Table 1.

3.2 Multi-Index Hybrid Tries

For string $x = x_1 \cdots x_l$ of length l and string $y = y_1 \cdots y_n$ of length n, the *concatenation* of x and y, denoted by $x \cdot y$, is the string obtained by appending y to the end of x, as in $x_1 \ldots x_l y_1 \ldots y_n$.

Definition 3. A (k, m)*-multi-index hybrid trie* ((k, m)-MIHT) is a data structure combining a B^+ tree T_b of order m and priority tries, which contains two types of nodes: *index nodes* (i-node) and *data nodes* (d-node).

§. Binary trie is the most natural and basic trie structure to represent prefixes. In a binary trie, there are only two pointers in each node, and the bits of prefixes is used to direct the branching. The prefix stored in a node v at level l is equal to the string of bits labeling the path from root to node v.

Let $I_i(u)$ and $child_j(u)$ be the i-th index and j-th child pointer stored in a index node u respectively. According to B$^+$ tree property, $i = 1, \ldots, m-1$ and $j = 0, \ldots, m-1$. Those i-node and d-node have the following data fields and structure characterizations:

1) The $root$ is a pointer array with a size of two, $root[0]$ points to the root of $PT[-1]$ and $root[1]$ points to the root of T_b.
2) An i-node having at least one child is an *internal i-node*. Otherwise, it is an *external i-node*.
3) Each internal i-node u contained the following fields:
   (a) $leaf(u)$ is a field with a boolean value, indicating whether u is a leaf. For the internal i-node u, $leaf(u)$ is set as **FALSE**.
   (b) The number of children for each internal i-node (except for the root) is between $\lceil \frac{m}{2} \rceil$ and m. Field $t(u)$ records the number of indices stored in u. Because $t(u)$ must be one less than the number of children of u, $\lceil \frac{m}{2} \rceil - 1 \leq t(u) < m$. The number of children of the root is between 2 and m.
   (c) The $t(u)$ indices, denoted by $I_1(u), I_2(u), \ldots, I_{t(u)}(u)$, are stored in i-node u with an ascending order $I_1(u) < I_2(u) < \cdots < I_{t(u)}(u)$.
   (d) There are $t(u)+1$ *child-pointers*, denoted by $child_0(u), child_1(u), \ldots, child_{t(u)}(u)$, point to $t(u)+1$ subtrees such that for each index x in the sub-tree pointed to by $child_i(u)$, the following three conditions hold: (i) $x < I_1(u)$ if $i = 0$; (ii) $x \geq I_{t(u)}(u)$ if $i = t(u)$; and (iii) $I_i(u) \leq x < I_{i+1}(u)$ if $0 < i < t(u)$.
   (e) The *content* of an internal i-node can be represented as $(leaf(u), t(u), child_0(u), I_1(u), child_1(u), I_2(u), \ldots, I_{t(u)}(u), child_{t(u)}(u))$ (Figure 2).
4) The fields of each external i-node v were similar to those of an internal i-node, but had the following two differences:
   (a) $leaf(v)$ is set as **TRUE**.
   (b) $child_i(v)$ points to the root of $PT[I_i(v)]$ for $i = 1, \ldots, t(v)$.
5) Each d-node w had the following fields:
   (a) $priority(w)$ recorded the boolean value, indicating whether w is a priority node. If the prefix corresponding to w is a priority prefix, it is set as **TRUE**; otherwise, it is set as **FALSE**.
   (b) $s(w)$ is the suffix stored in w.
   (c) If w is in $PT[I_i(v)]$, where v is an external i-node, then $port(s(w))$ is the output port of the prefix $I_i(v) \cdot s(w)$.
   (d) $left(w)$ is a pointer indicating the left-child (d-node) of w if it exists; otherwise, the pointer is set as "NULL".
   (e) $right(w)$ is a pointer indicating the right-child (d-node) of w if it exists; otherwise, the pointer is set as "NULL".

Figure 3 shows a (4,4)-MIHT. A pointer array $root$ points to the $PT[-1]$ and a B$^+$ tree. There are four i-nodes, a, b, c, and d, in which a is an internal i-node, and b, c, and d are external i-nodes. Each i-node has at least $\lceil \frac{m}{2} \rceil$=2 and at most m=4 child-pointers. The indices in each i-node satisfy Definition 3 3(c). For example, indices $I_1(c) = 5, I_2(c) = 6$, and $I_3(c) = 7$ are greater than or equal to $I_1(a)$=5, and less than $I_2(a)$=8. For node d (external i-node), $child_1(d)$ pointed to the root of $PT[8]$, and $child_2(d)$ pointed to the root of $PT[9]$. Furthermore, two suffixes, 0* and 1*, were stored in $PT[1]$, in which 0* was a suffix corresponding to the original prefix $I_1(b) \cdot 0$*=00010*, and 1* was a suffix corresponding to the original prefix $I_1(b) \cdot 1$*=00011*.

Definition 4. The *index-set* of the i-node v in a (k, m)-MIHT, denoted by $iSet(v)$, is defined as $iSet(v) = \{I_1(v), \ldots, I_{t(v)}(v)\}$ (i.e., the set of all indices in v).

Example 3. *In Fig. 3,* $iSet(a) = \{5, 8\}$, $iSet(b) = \{1, 2, 3\}$, $iSet(c) = \{5, 6, 7\}$, *and* $iSet(d) = \{8, 9\}$.

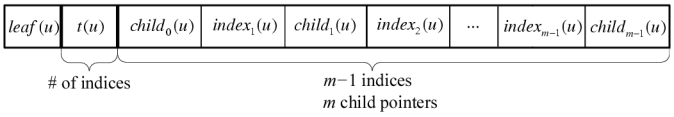


Fig. 2. Data fields of the i-node u.

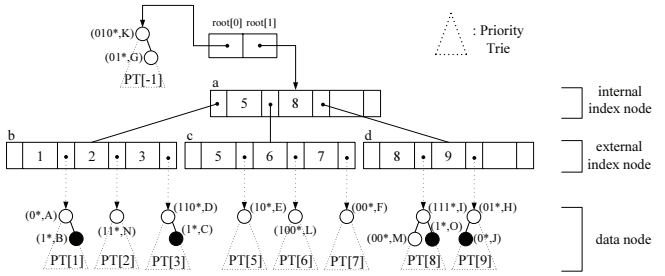


Fig. 3. A (4,4)-MIHT.

For a tree T, the *height* of T, denoted by $h(T)$, equals the $\max\{level(v) | \ v$ is a node in $T\}$. The height of (k, m)-MIHT T equals the $\max\{level(v) | \ v$ is a d-node in $T\}$. Let h_b be the height of a B$^+$ tree in T. When the number of child-pointers of each i-node reached the minimum and the number of indices of each i-node reached the maximum (i.e., 2^k indices), there were $\lceil \frac{m}{2} \rceil^{h_b} = \frac{2^k}{\lceil \frac{m}{2} \rceil - 1}$ external i-nodes, resulting in $h_b = O(\frac{k}{\log m})$. Let h' be the height of $PT[-1]$ and $h = \max\{h(PT[i]) | \ i = 1, 2, \ldots, 2^k - 1\}$. In addition, $h' = O(W - k)$. Because $h(T) \leq h_b + h + h'$, the following result was obtained.

Proposition 1. *Let W be the length of the IP address and let T be a (k, m)-MIHT. Therefore, $h(T) = O(\frac{k}{\log m} + W)$.*

4 Dynamic Router Table Operations

4.1 Construction

Assume that the array $root$ is allocated prior to constructing the (k, m)-MIHT. First, the following algorithm was

used to create a root node of $PT[-1]$ and a root node of a B^+ tree. Following the mentioned initialization process, an insertion algorithm (described in Section 4.2) was used to insert prefixes in the router table. The procedures ALLOCATE_D-NODE and ALLOCATE_I-NODE allocated the memory space of the d-node and i-node, respectively. Both of these required $O(1)$ time.

Algorithm 1: MIHT_CREATE($root$)

```
1 root[0] := ALLOCATE_D-NODE()
2 root[1] := ALLOCATE_I-NODE()
3 leaf(root[1]) := TRUE
```

4.2 Insertion

The insertion algorithm for the (k, m)-MIHT was in Algorithm MIHT_INSERT. The concept of this algorithm is described as follows. To insert the prefix $p = p_0p_1 \dots p_{l-1}$* into the (k, m)-MIHT T, operations were executed based on the following scenario:

Case 1: $len(p) \geq k$. By regarding $prefix\_key_k(p)$ as a key value, T was traversed from the root to a leaf v and split it (by executing the algorithm for splitting) if necessary. If $prefix\_key_k(p) \notin iSet(v)$, $prefix\_key_k(p)$ was inserted into v as a new index and split it if necessary. Therefore, an index $I_i(v)$ must be in an i-node v with $I_i(v) = prefix\_key_k(p)$. Thereafter, $suffix_k(p)$ was inserted into $PT[I_i(v)]$.

Case 2: $len(p) < k$. Insert p into $PT[-1]$ directly.

MIHT_INSERT($p, root[0], root[1]$) was the initial call to insert prefix p. Illustration of Algorithm I-NODE_SPLIT, examples for inserting prefixes, and time complexity analysis of the insertion algorithm are provided in Appendix A.

Theorem 1. *Algorithm* MIHT_INSERT(p, u, v) *can insert the prefix p into a (k, m)-MIHT T in $O(\frac{km}{\log m} + W)$ time.*

4.3 Lookup

The process of the search algorithm, MIHT_LOOKUP, is described as follows. Given the destination address DA, the algorithm searches a (k, m)-MIHT T starting from the root of T pointed by $root[1]$ in a top-down manner. A simple tree traversal from the root of T to a leaf v is performed, incorporated with a binary search using key value $prefix\_key_k(DA)$ to search each visited node. If $prefix\_key_k(DA) = I_i(v)$, where $i = 1, \dots, t(v)$ (i.e., $prefix\_key_k(DA) \in iSet(v)$), then the algorithm searches $suffix_k(DA)$ in $PT[I_i(v)]$. If we find the best match of $suffix_k(DA)$ in some priority trie, then it is the LMP, and the search process is terminated. Otherwise, the algorithm searches DA in $PT[-1]$. The algorithm is detailed in Algorithm MIHT_LOOKUP($DA, root$), where $next\_hop$ is used to record the output port of the current better matching prefix, and $default\_route$ is used to record the default output port.

Algorithm 2: MIHT_INSERT(p, u, v)

```
1  if len(p) ⩾ k then // p should be insert into B+ tree
2      if t(root[1]) = m − 1 then
3          new_root :=ALLOCATE_I-NODE()
4          child_0(new_root) := root[1]
5          root[1] := new_root
6          I-NODE_SPLIT(root[1], 0, child_0(root[1]),
           prefix_key_k(p))
7      find i such that I_i(v) ≤ prefix_key_k(p) < I_{i+1}(v)
8      if leaf(v) then // v is an external i-node
9          if (prefix_key_k(p) ≠ I_i(v)) or (i = 0) then
           // prefix_key_k(p) ∉ iSet(v)
10             i := i + 1
11             move [I_i, child_i], ..., [I_{t(v)}, child_{t(v)}] in i-node v to
               [I_{i+1}, child_{i+1}], ..., [I_{t(v)+1}, child_{t(v)+1}] in i-node
               v
12             I_i(v) := prefix_key_k(p)
13             child_i(v) := NULL
14             t(v) := t(v) + 1
15         PRIORITY_INSERT(child_i(v), suffix_k(p)) // insert
           suffix_k(p) into PT[I_i(v)]
16     else // v is an internal i-node
17         c := child_i(v)
18         if t(c) = m − 1 then
19             if (leaf(c) and prefix_key_k(p) ∉ iSet(c)) or
               ¬leaf(c) then
20                 I-NODE_SPLIT(v, i, c,
                   prefix_key_k(p)) // split node c
21                 if prefix_key_k(p) ≥ I_{i+1}(v) then
22                     i := i + 1
23         MIHT_INSERT(p, u, child_i(v))
24 else // p should be insert into PT[−1]
25     PRIORITY_INSERT(u, p)
```

Algorithm 3: I-NODE_SPLIT(x, y_pos, y, $pKey$)

```
1  z :=ALLOCATE_I-NODE()
2  leaf(z) := leaf(y)
3  move [I_{y_pos+1}, child_{y_pos+1}], ..., [I_{t(x)}, child_{t(x)}] in i-node x to
   [I_{y_pos+2}, child_{y_pos+2}], ..., [I_{t(x)+1}, child_{t(x)+1}] in i-node x
4  child_{y_pos+1}(x) := z
5  t(x) := t(x) + 1
6  g := ⌈m/2⌉
7  if leaf(y) then
8      if pKey ≥ I_g(y) then
9          move [I_{g+1}, child_{g+1}], ..., [I_{m−1}, child_{m−1}] in i-node y
           to i-node z
10         I_{m−g}(z) := pKey
11         arrange the indices of z in an increasing order
12     else
13         move [I_g, child_g], ..., [I_{m−1}, child_{m−1}] in i-node y to
           i-node z
14         I_g(y) := pKey
15         arrange the indices of y in an increasing order
16     t(y) := g
17     t(z) := m − g
18     I_{y_pos+1}(x) := I_0(z)
19 else
20     move child_g, [I_{g+1}, child_{g+1}], ..., [I_{m−1}, child_{m−1}] in
       i-node y to i-node z
21     t(y) := g − 1
22     t(z) := m − g − 1
23     I_{y_pos+1}(x) := I_g(y)
```

The initial call is MIHT_LOOKUP($DA, root[0], root[1]$). Example explains how this algorithm works and the time complexity analysis are provided in Appendix B.

Algorithm 4: MIHT_LOOKUP(DA, u, v)

```
next_hop := default_route
while ¬leaf(v) do
    find i such that I_i(v) ≤ prefix_key_k(DA) < I_{i+1}(v)
    v := child_i(v)
find i such that I_i(v) ≤ prefix_key_k(DA) < I_{i+1}(v)
if prefix_key_k(DA) = I_i(v) then
  // prefix_key_k(DA) ∈ iSet(v)
    next_hop := PRIORITY_LOOKUP(suffix_k(DA), child_i(v))
    if next_hop ≠ default_route then // LMP is found
        return next_hop
return PRIORITY_LOOKUP(DA, u)
```

Theorem 2. *Algorithm* MIHT_LOOKUP(DA, u, v) *can search* DA *in* $O(\frac{km}{\log m} + W)$ *time.*

4.4 Deletion

The deletion algorithm for the (k, m)-MIHT T is presented in this subsection. To delete the prefix p in T, the algorithm first determines the length of p. If $len(p) < k$, the algorithm uses procedure PRIORITY_DELETE($p, root[0]$) to delete p in $PT[-1]$ and releases the storage allocation of the corresponding d-node containing p. Otherwise, if $len(p) \geq k$, the algorithm uses $prefix\_key_k(p)$ as a key value to traverse T from the root to an external i-node v. If $prefix\_key_k \in iSet(v)$ for $1 \leq i \leq t(v)$, then $prefix\_key_k(p)$ must equal $I_i(v)$, and $child_i(v)$ points to the priority trie $PT[I_i(v)]$. Thereafter, a suffix $suffix_k(p)$ stored in a d-node can be deleted from $PT[I_i(v)]$. Following the deletion of the mentioned d-node, the algorithm further determines whether $PT[I_i(v)]$ is empty. If the condition holds, index $I_i(v)$ should be removed from v. However, this action may result in a situation where the i-node v must be merged with an other node. The proposed deletion algorithm allows the number of indices to be smaller than the lower bound (i.e., $t(v) < \lceil \frac{m}{2} \rceil - 1$) following the removal of an index from node v. The algorithm merges two neighboring external i-nodes, for example, x and y, into one node when the sum of cardinalities of their indices equals $m - 1$ (i.e., $t(x) + t(y) = m - 1$). Consider the merger of two neighboring internal i-nodes x and y, the parent of which is z. Without losing generality, assume that x and y are the i^{th} and $(i+1)^{\text{th}}$ children of z, respectively. To merge y and z, $I_{i+1}(x)$ must be moved to y, and it is subsequently necessary to retain one space for $I_{i+1}(x)$. Thereafter, y and z are merged into one node when $t(y) + t(z) + 1 = m - 1$.

The deletion algorithm is detailed in MIHT_DELETE(p, u, v). This algorithm uses three auxiliary procedures: FREE_I-NODE, FREE_D-NODE, and I-NODE_MERGE. The first two procedures release the storage allocations for an i-node and a d-node, respectively, in $O(1)$ time, and procedure I-NODE_MERGE merges two i-nodes (if necessary) after an index has been removed from an i-node. Algorithm MIHT_DELETE(p, u, v) first searches for an external i-node that contains the $prefix\_key_k(p)$. During this search, the visited i-nodes and the indices of the child-pointers used for moving are stored in the arrays *nodes*[·] and *branch*[·], respectively.

Algorithm 5: MIHT_DELETE(p, u, v)

```
x := 0
if len(p) ≥ k then // p should be in B-plus tree T_B
    while ¬leaf(v) do
        find i such that I_i(v) ≤ prefix_key_k(p) < I_{i+1}(v)
        nodes[++x] := v
        branch[x] := i
        v := child_i(v)
    find i such that I_i(v) ≤ prefix_key_k(p) < I_{i+1}(v)
    if prefix_key_k(p) = I_i(v) then
    // prefix_key_k(p) ∈ iSet(v)
        PRIORITY_DELETE(suffix_k(p), child_i(v))
        FREE_D-NODE
        if child_i(v) is an empty trie then
            remove I_i(v) from v; t(v) := t(v) − 1
            I-NODE_MERGE(nodes, branch, x)
else // p should be in PT[−1]
    PRIORITY_DELETE(p, u)
    FREE_D-NODE
return
```

Algorithm 6: I-NODE_MERGE($nodes, branch, x$)

```
while x > 0 do
    f := nodes[x]; b := branch[x]
    c := child_b(f); c_l := child_{b−1}(f); c_r := child_{b+1}(f)
    set full = m − 1 when c is a external i-node; otherwise, set
    full = m − 2
    if t(c_l) + t(c) = full then
        if c is an internal i-node then
            move I_b(f) in i-node f to i-node c_l
        else
            remove I_b(f) from f
        move all indexes and child-pointers in i-node c to
        i-node c_l
        t(c_l) := m − 1; FREE_I-NODE(c)
    else if t(c) + t(c_r) = full then
        if c is an internal i-node then
            move I_{b+1}(f) in i-node f to i-node c
        else
            remove I_{b+1}(f) from f
        move all indexes and child-pointers in i-node c_r to
        i-node c
        t(c) := m − 1; FREE_I-NODE(c_r)
    t(f) := t(f) − 1; x := x − 1
    if x = 1 and t(root[1])=0 then
        temp_root := root[1]
        root[1] := child_0(root[1])
        FREE_I-NODE(temp_root)
return
```

The initial call to delete the prefix p is MIHT_DELETE($p, root[0], root[1]$). The example shows how to delete prefixes using the proposed algorithm and the time complexity analysis are provided in Appendix C.

Theorem 3. *Algorithm* MIHT_DELETE(p, u, v) *can delete prefix* p *in* $O(\frac{km}{\log m} + W)$ *time.*

5 Partitioning the Multi-index Hybrid Trie

We partitioned the (k, m)-MIHT into several smaller (k, m)-MIHTs to reduce the tree height. The new data structure is the (k, m)-*partitioning multi-index hybrid trie* ((k, m)-PMIHT). This concept has been used in many routers [13]. The partitioned (k, m)-MIHTs were merged through partition table $P[0, 1, \cdots, 2^{\alpha} - 1]$. For a fixed length α, the table had 2^{α} entries corresponding to the 2^{α} possible values from $\underbrace{00\ldots0}_{\alpha \text{ bits}}$ to $\underbrace{11\ldots1}_{\alpha \text{ bits}}$. The entry $P[i]$ indicated that (k, m)-MIHT stores prefixes with original prefixes that have a common sub-prefix $p_0p_1 \ldots p_{\alpha-1}$, where $p_0p_1 \ldots p_{\alpha-1}$ is the α-bit binary representation of i. An illustration of a (k, m)-PMIHT is shown in Appendix D. Furthermore, the value of α should not be larger than the length of the shortest prefix in the router table, to prevent a situation where a prefix is stored in more than one (k, m)-MIHT. For a more detailed description, if we insert prefix p into (k, m)-PMIHT, we obtain α bits of p to determine the index value of the partitioned table. If $len(p) \geq \alpha$, we can insert p into the corresponding entry. Otherwise, if $len(p) < \alpha$, p corresponds to more than one index value. For example, assuming that p =101000* and $\alpha = 8$ (the first eight bits of a prefix represent the index value), because 101000* contains $(10100000)_2 = 160$, $(10100001)_2 = 161$, $(10100010)_2 = 162$, and $(10100011)_2 = 163$, p must be inserted into the (k, m)-MIHTs, as indicated by $P[160], P[161], P[162]$, and $P[163]$, respectively. Because storing duplicate prefixes is inefficient, a proper α value must be chosen to reduce the storage requirement. Therefore, α was the length of the shortest prefix in the proposed router table.

To execute the router table operations (i.e., lookup, insertion, and deletion) in a (k, m)-PMIHT, the corresponding partition table entry was determined and operations in the corresponding (k, m)-MIHT were executed. The algorithms for the lookup, insertion, and deletion operations for a (k, m)-PMIHT are presented in Appendix D.

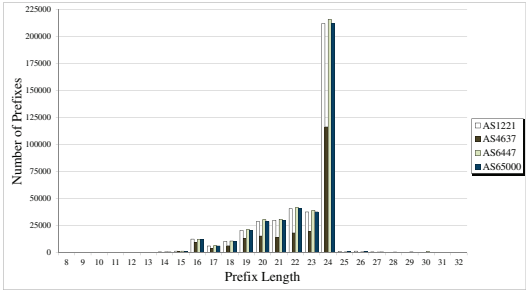
6 Experimental Results

The experiments were performed using C language for four benchmark IPv4 and two benchmark IPv6 prefix databases obtained from [1], as shown in Table 2†. The codes were run on a 3.40GHz Pentium 4 PC that had 1.99GB of memory. Figure 4 shows the total prefix-population of the six router tables by prefix length. For comparison with other data structures, we selected the variables k and m for (k, m)-MIHT and (k, m)-PMIHT, respectively, and analyzed their performances, as shown in Section 6.1. Section 6.2 presents a comparison of the proposed data structures with other data structures.

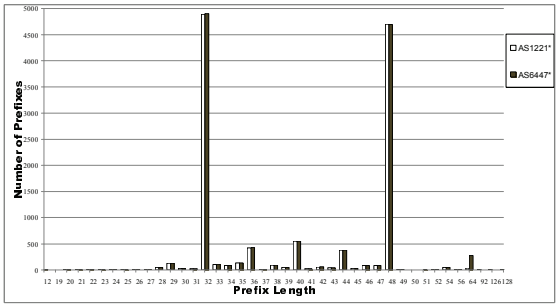
†. Both AS1221* and AS6447* are IPv6 BGP routing tables. Because IPv4 stay popular in modern Internet protocol, the benchmark IPv6 prefix databases may smaller and fewer than IPv4's currently.

TABLE 2
Six BGP Routing Tables for Experiment.

Database	AS1221	AS4637	AS6447	AS65000	AS1221*	AS6447*
date(year.month)	2012.04	2012.04	2012.04	2012.04	2013.02	2013.02
# of prefix	407,067	219,581	417,995	406,973	12,155	12,278
# of PTs	25,922	18,584	26,368	25,919	5,078	5,232
Size of $PT[-1]$	3,167	1,781	3,173	3,166	288	295
Max. size of PTs	368	232	367	367	254	261
Avg. size of PTs	15.70	11.82	15.85	15.70	10.92	10.97
# of split/merge per update	0.0091	0.0123	0.0090	0.0090	0.0083	0.0083



(a)



(b)

Fig. 4. Prefix-population in the router tables: (a) for IPv4, and (b) for IPv6.

6.1 Selecting k and m for MIHT

Table 2 shows the results of the PTs for the (k, m)-MIHTs for four IPv4 router tables with k = 16 and two IPv6 router tables with k = 16, respectively, and Figure 4 shows the length distributions of each router table. For each IPv4 router table, more than 99% of the prefixes had lengths $\geq$ 16. Therefore, if k is 16, a small number of prefixes are stored in $PT[-1]$. In addition, the maximum number of prefixes in the $PT[i]$ of an MIHT with $k = 16$ is less than 369, and the average number of prefixes in $PT[i]$ is at most 15.85 prefixes, where $i = 0, \ldots, 2^{16} - 1$. Therefore, we set $k = 16$. For each IPv6 router table, we have similar observations and then we set $k = 32$. Because the order may affect the performance of the proposed data structures, selecting a proper order is essential. Because a (k, m)-MIHT may contain several priority tries, the *average height* of T is the average height of all the priority tries plus the height of the B^+ tree. Furthermore, because a (k, m)-PMIHT T may contain

several (k, m)-MIHTs, the *maximum height* of T is defined as the $\max\{h(T')|\ T'$ is a (k, m)-MIHT in $T\}$, and the *average height* of T is the average of all the average heights of the (k, m)-MIHTs in T. Table 3 shows the experimental results, including the tree height, the memory requirements, the average times of the lookup and update, and the number of memory accesses (MAs) for lookup and update for the $(16, m)$-MIHTs and $(32, m)$-MIHTs with various orders of m based on the four IPv4 and two IPv6 router databases, respectively, as shown in Table 2. To ascertain the updating performance, the proposed MIHT was constructed according to 90% of the original prefix databases. Thereafter, we inserted and deleted the remaining 10% of prefixes to calculate the average update time. The experiment results showed that when the order increased, all three aspects of performance, including storage, time, and MA, improved when $m \leq 16$ (This situation is similar to IPv6, when $m \leq 32$). However, when the order reached 18 (as 34 in IPv6), the performance stopped improving, and some performance metrics decreased. Although the size of the i-node increased when the order increased, the size of the entire routing table decreased because the number of i-nodes decreased at the same time. To achieve enhanced performance, we used (16,16)-MIHT for IPv4 and (32,32)-MIHT for IPv6 to compare the proposed data structure with other data structures, as addressed in Section 6.2.

For the $(16, m)$-PMIHT for IPv4 and $(32, m)$-PMIHT for IPv6, because there are no prefixes with a length <8 (as 12 in IPv6), we set $\alpha = 8$ for IPv4 and $\alpha = 12$ for IPv6 to avoid duplicate storage of the same prefix. Table 4 shows the experimental results of the $(16, m)$-PMIHT for IPv4 and $(32, m)$-PMIHT for IPv6 with different orders m. We have performance observations similar to those for the $(16, m)$-MIHT and $(32, m)$-MIHT. Hence, we adopt the (16,16)-PMIHT for IPv4 and $(32, 32)$-PMIHT for IPv6 to compare with other data structures in the next subsection.

6.2 Comparison with other data structures

In this section, we present a comparison of the proposed data structures, (16,16)-MIHT and (16,16)-PMIHT, with the following structures: the binary trie, LC-trie [20], prefix tree [2], priority trie [14], dynamic tree bitmap (DTBM) [26], 4-MPT [11], and 4-PCMST [12]. In addition, we also perform the comparison of (32,32)-MIHT and (32,32)-PMIHT with all the same structures mentioned above by IPv6 routing databases‡. First, we compared the average tree height and the storage requirements of the various data structures when implementing the six router databases, as shown in Table 5 and Table 6. For IPv4, the heights of the (16,16)-MIHT and (16,16)-PMIHT were less than those of the other data structures, reducing the number of memory accesses required for dynamic router table operations. For the storage

‡. Considering the address length for IPv6 is much longer than IPv4, we adopt 5-MPT and 5-PCMST for an objective comparison.

TABLE 3
Comparison of $(16, m)$-MIHTs/$(32, m)$-MIHTs for IPv4/IPv6 with Various Orders m.

Database	Order(m)	Height	Avg. Height	Storage(KB)	Lookup (Avg.)		Update (Avg.)	
					# Clock Cycles	#MAs	# Clock Cycles	#MAs
AS1221	6	22	12.04	7031	2434	13.17	2015	8.69
	8	20	10.04	6962	2320	11.17	1904	7.68
	10	19	9.04	6927	2267	10.17	1811	7.17
	12	19	9.04	6907	2249	10.17	1830	7.16
	14	18	8.04	6893	2251	9.17	1743	6.66
	16	18	8.04	6883	2229	9.17	1803	6.66
	18	18	8.04	6875	2250	9.17	1810	6.66
AS4637	6	22	11.66	3884	2084	13.51	1920	8.53
	8	20	8.66	3835	2005	11.51	1797	7.51
	10	19	8.66	3810	1950	10.51	1729	6.99
	12	19	8.66	3795	1948	10.51	1749	6.99
	14	18	7.66	3785	1950	9.51	1660	6.48
	16	18	7.66	3778	1936	9.51	1670	6.48
	18	18	7.66	3773	1978	9.51	1659	6.48
AS6447	6	25	12.07	7215	2441	13.51	1954	8.70
	8	23	10.07	7144	2317	11.51	1832	7.68
	10	22	9.07	7109	2271	10.51	1747	7.18
	12	22	9.07	7088	2260	10.51	1744	7.17
	14	21	8.07	7074	2265	9.51	1692	6.67
	16	21	8.07	7064	2240	9.51	1679	6.67
	18	21	8.07	7056	2230	9.51	1680	6.66
AS65000	6	22	12.04	7029	2425	13.51	2019	8.70
	8	20	11.04	6960	2322	11.51	1883	7.68
	10	19	9.04	6926	2246	10.51	1815	7.17
	12	19	9.04	6905	2242	10.51	1817	7.17
	14	18	8.04	6891	2252	9.51	1747	6.66
	16	18	8.04	6881	2231	9.51	1767	6.66
	18	18	8.04	6874	2249	9.51	1770	6.66
AS1221*	24	19	10.60	6096	997	11.59	1610	7.65
	26	18	8.84	6078	990	9.83	1534	6.76
	28	17	7.96	6066	990	8.95	1587	6.31
	30	17	7.96	6057	981	8.94	1593	6.30
	32	16	7.08	6050	990	8.07	1544	5.86
	34	16	7.08	6046	993	8.07	1506	5.86
	36	16	7.08	6041	1001	8.07	1500	5.86
AS6447*	24	22	10.67	6291	1004	11.96	1546	7.70
	26	20	8.90	6273	999	10.13	1543	6.80
	28	19	8.02	6260	1001	9.25	1497	6.35
	30	19	8.02	6252	990	9.25	1486	6.35
	32	19	7.13	6245	986	8.42	1487	5.90
	34	19	7.13	6239	1001	8.42	1496	5.90
	36	19	7.13	6236	1010	8.42	1456	5.89

TABLE 4
Comparison of $(16, m)$-PMIHTs/$(32, m)$-PMIHTs for IPv4/IPv6 with Various Orders m.

Database	Order(m)	Height		Storage(KB)	Lookup (Avg.)		Update (Avg.)	
		Max.	Avg.		# Clock Cycles	#MAs	# Clock Cycles	#MAs
AS1221	6	18	5.67	7017	2111	10.14	1768	9.18
	8	17	5.02	6951	1995	9.30	1730	8.45
	10	16	4.44	6914	1977	8.54	1701	7.89
	12	16	4.34	6898	1946	8.46	1727	7.78
	14	16	4.23	6886	1937	8.42	1685	7.69
	16	16	4.11	6875	1937	8.31	1680	7.56
	18	16	3.90	6867	1974	8.16	1692	7.43
AS4637	6	18	4.86	3874	1807	9.89	1785	9.10
	8	17	4.27	3828	1715	9.04	1671	8.36
	10	16	3.71	3801	1700	8.23	1626	7.74
	12	16	3.65	3790	1661	8.12	1618	7.63
	14	16	3.56	3781	1663	8.09	1634	7.55
	16	16	3.47	3776	1667	8.05	1654	7.46
	18	16	3.24	3767	1700	7.90	1595	7.31
AS6447	6	21	5.76	7201	2154	10.17	1724	9.21
	8	20	5.06	7132	2007	9.30	1690	8.46
	10	19	4.50	7096	1979	8.60	1649	7.92
	12	19	4.39	7079	1938	8.46	1632	7.79
	14	19	4.29	7067	1933	8.42	1654	7.70
	16	19	4.16	7056	1934	8.30	1676	7.57
	18	19	3.98	7048	1966	8.19	1631	7.45
AS65000	6	18	5.66	7016	2107	10.13	1780	9.18
	8	17	5.01	6949	1992	9.30	1726	8.45
	10	16	4.43	6913	1967	8.54	1702	7.89
	12	16	4.34	6896	1929	8.46	1708	7.79
	14	16	4.23	6884	1932	8.42	1687	7.69
	16	16	4.11	6873	1927	8.31	1676	7.56
	18	16	3.90	6865	1956	8.15	1677	7.42
AS1221*	24	16	4.99	6084	870	8.92	1497	8.08
	26	15	4.42	6070	856	8.23	1520	7.44
	28	14	3.91	6060	852	7.56	1483	6.94
	30	14	3.82	6050	852	7.49	1478	6.85
	32	14	3.72	6043	829	7.45	1489	6.77
	34	14	3.62	6036	866	7.35	1470	6.65
	36	14	3.43	6031	853	7.22	1470	6.54
AS6447*	24	19	5.09	6280	875	9.02	1459	8.15
	26	18	4.47	6265	857	8.23	1444	7.49
	28	17	3.98	6254	854	7.60	1464	7.01
	30	17	3.88	6245	855	7.49	1483	6.89
	32	17	3.79	6237	826	7.45	1443	6.81
	34	17	3.68	6230	865	7.36	1439	6.70
	36	17	3.52	6224	857	7.25	1455	6.59

requirement, the binary trie, LC-trie, and DTBM contained dummy nodes, substantially increasing storage costs. Although the size of each node in the binary trie was smaller than that of the (16,16)-MIHT and (16,16)-PMIHT, the storage requirements of (16,16)-MIHT and (16,16)-PMIHT were smaller than those of the binary trie. The LC-trie and DTBM contained dummy nodes and used larger nodes, causing the storage requirements to be larger than those of the (16,16)-MIHT and (16,16)-PMIHT. Above observations also similar to the experimental results for IPv6.

TABLE 5
The Average Tree Heights and Storage Comparisons of the Various Data Structures.

Data Structure	Avg. Tree Height				Storage (KB)			
	AS1221	AS4637	AS6447	AS65000	AS1221	AS4637	AS6447	AS65000
Binary Trie	32	32	32	32	11,519	6,731	11,870	11,513
LC-Trie	23	21	25	23	13,772	8,266	14,011	13,762
Prefix Tree	29	29	32	29	6,360	3,430	6,531	6,358
Priority Trie	28	27	32	29	6,360	3,430	6,531	6,358
DTBM	10	10	10	10	25,291	14,506	25,997	25,285
4-MPT	11	10	10	10	21,484	11,391	22,049	21,501
4-PCMST	10.08	9.72	10.13	10.04	22,172	12,463	23,051	22,043
(16,16)-MIHT	8.04	7.66	8.07	8.04	6,883	3,778	7,064	6,881
(16,16)-PMIHT	4.11	3.47	4.16	4.11	6,875	3,776	7,056	6,873

TABLE 6
The Average Tree Heights and Storage Comparisons in IPv6.

Data Structure	Avg. Tree Height		Storage (KB)	
	AS1221*	AS6447*	AS1221*	AS6447*
Binary Trie	128	128	9,215	9,496
LC-Trie	89.21	89.65	11,018	11,209
Prefix Tree	116	120	5,088	5,747
Priority Trie	112	117	5,088	5,747
DTBM	27	28	20,233	20,798
5-MPT	12.56	12.78	15,038	15,875
5-PCMST	8.08	8.83	15,520	16,597
(32,32)-MIHT	7.07	7.14	6,057	6,251
(32,32)-PMIHT	3.61	3.68	6,084	6,244

Because each memory access requires substantial time, the number of memory accesses affects the operating time. In addition, the operations in each node affect the operating time. We compared the average lookup times and average number of memory accesses required for the various data structures, as shown in Table 7 and Table 9. In addition, the (16,16)-MIHT and (16,16)-PMIHT required fewer memory accesses than did the other data structures, except for the LC-Trie, implying that their lookup times were shorter than those of the other data structures for two key reasons: 1) All external i-nodes in the proposed data structures were in the same level, and 2) the average size of the searched PTs was small when we reached the external i-node. Although the 4-MPT returns the longest matching prefixes when located, all prefixes in each visited p-node must be compared and the corresponding secondary structure must be searched, increasing the number of memory accesses for lookup. For the proposed data structures, because the indices in each i-node were sorted, we only had to compare the key values of small parts of the indices in each i-node, and search at most two secondary structures (including PT[-1]), rather than searching numerous secondary structures. Because each prefix was a dual-dimension datum, the cost of comparing two prefixes was greater than that of comparing two values. Therefore, the (16,16)-MIHT and (16,16)-PMIHT required fewer memory accesses for lookup. Although the number of memory accesses of the LC-trie was similar to that of the (16,16)-MIHT, the LC-trie required more time to perform lookup because when a node is visited in an LC-Trie, the corresponding branch and skip must be read to proceed to the next node. Although the lookup time of the (16,16)-MIHT is slightly slower than that of the DTBM, the (16,16)-PMIHT is faster than the DTBM for lookup time because its tree height is less. For IPv6, the IP address length is much longer, will cause larger memory storage demands. However, the other structures also suffer this problem. According to the experimental results, (32,32)-MIHT and (32,32)-PMIHT still perform well.

TABLE 7
Performance Comparisons for Lookup for the Various Data Structures.

Data Structure	Avg. Lookup Time (# of Clock Cycles)				Avg. # of Memory Accesses			
	AS1221	AS4637	AS6447	AS65000	AS1221	AS4637	AS6447	AS65000
Binary Trie	2,990	2,684	3,018	3,000	23.56	23.40	23.59	23.56
LC-Trie	3,006	2,857	3,020	3,015	9.25	9.75	9.26	9.24
Prefix Tree	3,206	2,793	3,215	3,206	21.71	21.31	21.72	21.71
Priority Trie	2,864	2,481	2,872	2,862	20.09	19.65	20.09	20.09
DTBM	2,094	1,912	2,091	2,114	8.34	8.29	8.35	8.34
4-MPT	2,878	2,509	2,850	2,866	15.69	14.71	15.61	15.70
4-PCMST	2,012	1,805	2,112	2,151	10.16	9.52	9.57	9.16
(16,16)-MIHT	2,222	1,920	2,227	2,221	9.51	9.17	9.51	9.51
(16,16)-PMIHT	1,954	1,657	1,935	1,953	8.31	8.05	8.30	8.30

The performance comparisons for updating are shown in Table 8 and Table 9. The experiments inserted and deleted 10% of the prefixes from each of the four experimental router databases. We compared the update operations with the various dynamic data structures. Because the LC-trie was static, the entire table must be reconstructed when inserting or deleting a prefix. Therefore, this data structures was excluded from the comparison. The (16,16)-MIHT and (16,16)-PMIHT required fewer memory accesses than did the other data structures, except for the DTBM. The experimental results for the update operations showed that the (16,16)-MIHT and (16,16)-PMIHT had shorter update times than did all of the other data structures. Because a binary trie can insert or delete the longest prefix, the number of memory accesses was 32 for the IPv4 (128 for the IPv6). Furthermore, the number of memory accesses for the binary trie, the prefix tree and priority trie, exceeded 20 times. This is because, in each structure, the insertion and deletion of numerous routing entries in the router databases often follows a downward path from the root to the leaves to execute the operations. Therefore, the update time of the binary trie, prefix tree, and priority trie were longer than those of the proposed data structures. Although the DTBM has a shorter tree height and fewer memory accesses than the (16,16)-MIHT, it requires more time for updating because of two reasons: 1) Because all external i-nodes in the (16,16)-MIHT are in the same level, we reached the external i-node faster and accessed the corresponding substructure (PTs). 2) The average size of PTs was small (less than 15.85), causing the updating to be performed in a small-priority trie.

The (16,16)-PMIHT not only had the properties of the (16,16)-MIHT, but had improved updating performance.

TABLE 8
Performance Comparison for Updating for the Various Data Structures.

Data Structure	Avg. Update Time (# of Clock Cycles)				Avg. # of Memory Accesses			
	AS1221	AS4637	AS6447	AS65000	AS1221	AS4637	AS6447	AS65000
Binary Trie	4,138	4,021	4,216	4,181	26.55	26.70	26.59	26.54
Prefix Tree	2,445	2,395	2,426	2,438	20.87	20.33	20.77	20.85
Priority Trie	3,834	3,830	3,831	3,849	21.42	21.00	21.43	21.42
DTBM	2,755	2,705	2,749	2,754	8.90	8.91	8.91	8.90
4-MPT	2,144	2,419	2,178	2,277	6.24	6.26	6.21	6.24
4-PCMST	1,345	1,517	1,079	1,153	4.04	4.05	3.02	3.24
(16,16)-MIHT	1,907	1,780	1,853	1,900	8.90	8.70	8.90	8.90
(16,16)-PMIHT	1,679	1,693	1,622	1,679	7.56	7.46	7.56	7.57

TABLE 9
Performance Comparisons for Lookup and Update in IPv6.

Data Structure	Lookup (Avg. # of Clock Cycles)		Lookup (Avg. # of MAs)		Update (Avg. # of Clock Cycles)		Update (Avg. # of MAs)	
	AS1221*	AS6447*	AS1221*	AS6447*	AS1221*	AS6447*	AS1221*	AS6447*
Binary Trie	1,196	1,268	14.14	15.10	2,483	2,614	18.05	14.19
LC-Trie	1,202	1,268	5.55	5.93	-	-	-	-
Prefix Tree	1,282	1,13	13.02	13.90	1,467	1,504	14.19	14.75
Priority Trie	1,146	1,206	12.05	12.86	2,300	2,375	14.57	15.22
DTBM	838	878	4.97	5.34	1,653	1,704	6.05	6.33
5-MPT	1,151	1,197	9.41	9.99	1,286	1,350	4.24	4.41
5-PCMST	805	887	6.10	6.12	807	669	2.75	2.14
(32,32)-MIHT	889	935	5.71	6.09	1,144	1,149	6.05	6.32
(32,32)-PMIHT	782	813	4.99	5.31	1,007	1,006	5.14	5.37

7 Concluding Remarks

We propose two novel data structures, (k, m)-MIHT and (k, m)-PMIHT, for IP lookup and updates. The chief features of the proposed data structures are summarized as follows: 1) By expending a small amount of effort to search the B^+ tree, the proposed data structures performed lookup and updates in a priority trie that contained fewer prefixes than did the original prefix set. Therefore, the lookup and update performances were significantly improved. 2) The proposed data structures only stored indices (suffixes), rather than storing prefixes in the nodes, and did not contain any dummy nodes, thereby requiring less storage. 3) Because the size of indices was smaller than those of the prefixes, we selected larger orders to achieve faster lookups and updates. An order of 16 resulted in shorter tree heights, reducing the number of memory accesses for the router table operations. The simulation results showed that the proposed data structures are superior to the trie-based data structures for IP lookup time and update time.

References

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

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

[3] Y.-K. Chang, Y.-C. Lin, and C.-C. Su, "Dynamic multiway segment tree for IP lookups and the fast pipelined search engine," *IEEE Transactions on Computers*, vol. 59, no. 4, pp. 492–506, Apr. 2010.

[4] Y.-K. Chang "Fast binary and multiway prefix searches for packet forwarding," in *Computer Networks*, vol. 51, no. 3, pp. 588–605, Feb. 2007.

[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, Jun. 2007.

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

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

[8] W. Eatherton, G. Varghese, and Z. Dittia, "Tree bitmap: hardware/software IP lookup with incremental updates," in *Proceedings of ACM SIGCOMM Computer Communication Review*, 34, 2, Apr. 2004.

[9] R. Elmasri and S. B. Navathe, Fundamental of Database Systems, 5-th ed. Boston: Addison Wesley, 2007, ch.14.

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

[11] S.-Y. Hsieh, Y.-L. Huang, and Y.-C. Yang, "Multi-prefix trie: a new data structure for designing dynamic router-tables," *IEEE Transactions on Computers*, vol. 60, no. 5, pp. 693–706, May 2011.

[12] S.-Y. Hsieh and Y.-C. Yang, "A classified multisuffix trie for IP lookup and update," *IEEE Transactions on Computers*, vol. 61, no. 5, pp. 726–731, May. 2012.

[13] B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search," *IEEE/ACM Transactions on Networking*, vol. 7, no. 3, pp. 324–334, Jun. 1999.

[14] H. Lim, C. Yim, and E. E. Swart, "Priority tries for IP address lookup," *IEEE Transactions on Computers*, vol. 59, no. 6, pp. 784–794, Jun. 2010.

[15] H. Lim, W. Kim, and B. Lee, "Binary search in a balanced tree for IP address lookup," in *Proceedings of IEEE workshop high performance switching and routing*, pp. 490–494, 2005.

[16] H. Lim, W. Kim, and B. Lee, "Binary searches on multiple small trees for IP address lookup," *IEEE comunications letters*, vol. 9, no. 1, pp. 75–77, Jan. 2005.

[17] 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.

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

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

[20] 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, Jun. 1999.

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

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

[23] 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.

[24] 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.

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

[26] S. Sahni and H. Lu, "Dynamic tree bitmap for IP lookup and update," in *Proceedings of the Sixth International Conference on Networking*, pp. 79–84, 2007.

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

[28] V. Srinivasan and G. varghese, "Fast address lookups using controlled prefix expansion," *ACM Transactions on Computer Systems*, vol. 17, no. 1, pp. 1–40, 1999.

[29] P. R. Warkhede, S. Suri, and G. Varghese, "Multiway range tree: scalable IP lookup with fast updates," *Computer Networks*, vol. 44, no. 3, pp. 289–303, Feb. 2004.

[30] N. Yazdani and P. S. Min, "Fast and scalable schemes for the IP address lookup problem," in *Proceedings of IEEE workshop high performance switching and routing*, pp. 83–92, 2000.

[31] C. Yim, B. Lee, and H. Lim, "Efficient binary search for IP address lookup," *IEEE comunications letters*, vol. 9, no. 7, pp. 652–654, Jul. 2005.

Chia-Hung Lin received the BS degree from the department of information management of National Taiwan University of Science and Technology, Taiwan, in 2001, and the MS degree in computer science and information engineering from National Cheng-Kung University, Taiwan, in 2004. He is currently working toward the PhD degree also in computer science and information engineering of National Cheng-Kung University. His research interests include parallel and distributed computing, design and analysis of algorithms, human-machine interface design and machine learning.

Chia-Yin Hsu received the BS degree in the Department of Computer Science and Information Management from Soochow University, Taiwan, in 2010 and the MS degree in Computer Science and Information Engineering from National Cheng Kung University, Taiwan, in 2012. His research interests include IP routing and searching algorithms.

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 distinguished professor. He received the 2007 K. T. Lee Research Award, Presidents Citation Award (American Biographical Institute) in 2007, the Engineering Professor Award of Chinese Institute of Engineers (Kaohsiung Branch) in 2008, the National Science Councils Outstanding Research Award in 2009, and IEEE Outstanding Technical Achievement Award (IEEE Tainan Section) in 2011. He is Fellow of the British Computer Society (BCS). His current research interests include design and analysis of algorithms, fault-tolerant computing, bioinformatics, parallel and distributed computing, and algorithmic graph theory.