Priority Tries for IP Address Lookup

Hyesook Lim, *Member*, *IEEE*, Changhoon Yim, *Member*, *IEEE*, and
Earl E. Swartzlander, Jr., *Fellow*, *IEEE*

Abstract—High-speed IP address lookup is essential to achieve wire speed packet forwarding in Internet routers. The longest prefix matching for IP address lookup is more complex than exact matching because it involves dual dimensions: length and value. This paper presents a new formulation for IP address lookup problem using range representation of prefixes and proposes an efficient binary trie structure named a priority trie. In this range representation, prefixes are represented as ranges on a number line between 0 and 1 without expanding to the maximum length. The best match to a given input address is the smallest range that includes the input. The priority trie is based on the trie structure, with empty internal nodes in the trie replaced by the priority prefix which is the longest among those in the subtrie rooted by the empty nodes. The search ends when an input matches a priority prefix, which significantly improves the search performance. Performance evaluation using real routing data shows that the proposed priority trie is very good in performance metrics such as lookup speed, memory size, update performance, and scalability.

Index Terms—Internet, router, IP address lookup, binary trie, priority trie, range representation.

1 Introduction

The rapid growth of Internet traffic requires routers to perform high-speed packet forwarding. Address lookup is one of the most challenging tasks since it should be performed at wire speed for the incoming packets, even as packet arrival rates and routing table sizes are dramatically increasing [1]. Address lookup determines the output port using the destination Internet protocol (IP) address of an incoming packet in order to forward the packet toward its final destination.

IP addresses have two levels of hierarchy: a network part and a host part. The network part is called the prefix. Classless interdomain routing (CIDR) structure allows prefixes of arbitrary length and address aggregation at arbitrary levels. As a result, the address lookup in routers requires searching the forwarding table for the longest prefix that matches the destination address of the input packet to find the most specific route. Determining the longest matching prefix (LMP) or the best matching prefix (BMP) involves two dimensions: length and value [2].

Several metrics are useful to evaluate the performance of IP address lookup algorithms. Search speed is the primary metric and is highly dependent on the number of memory accesses for table lookup, since memory access is the most time-consuming operation in the search process [3]. The size of required memory is also an important metric as the routing tables have grown to hundreds of thousands of entries. For routing tables that support dynamic routing, the ability to provide incremental updates is also important. Scalability is another important metric to accommodate growing numbers of routing entries.

High-performance routers based on ternary content addressable memory (TCAM) have been implemented. With TCAM, an address lookup is performed with a single memory access [4], [5]. TCAM is much more expensive than ordinary memory in circuit complexity as well as power consumption. Many algorithms and architectures performing the longest prefix match using ordinary memories have been proposed.

As a basic address lookup structure, binary tries are simple and easy to implement [6]. The binary trie structure facilitates incremental update and provides good scalability. However, the speed of binary tries is limited and requires large memory because of empty internal nodes. Most of the successful methods for IP address lookup practically used are essentially high-performance variants of the basic binary trie [7] such as a multibit trie [2] and the tree bitmap [8].

Range matching algorithms for the IP address lookup problem represent prefixes as ranges in a number line between 0 and $2^{32}-1$ [9], [10], [11]. In order to remove the length dimension in the IP address lookup problem and present prefixes as ranges, the start points and the end points of the ranges are padded with zeros and ones to span the maximum length. Ranges are divided by disjoint intervals, and the BMP for each disjoint interval is precomputed and stored. Binary search based on entry values is applied in the range matching algorithm.

This paper presents a new mathematical formulation using the range representation of prefixes and proposes a new IP address lookup structure named a priority trie. Here, the prefixes are represented as a range between 0 and 1. The IP address lookup problem is formulated as the range inclusion problem finding the smallest range that includes the given input address. In the proposed range representation, the start and the end points are not necessarily padded to

- *H. Lim is with the Department of Electronics Engineering, Ewha W. University, 11-1 Daehyun-dong, Seodaemun-gu, Seoul 120-750, Korea. E-mail: hlim@ewha.ac.kr.*
- *C. Yim is with the Department of Internet and Multimedia Engineering, Konkuk University, 1 Hwayang-dong, Kwangjin-gu, Seoul 143-701, Korea. E-mail: cyim@konkuk.ac.kr.*
- *E.E. Swartzlander, Jr., is with the Department of Electrical and Computer Engineering, The University of Texas at Austin, 1 University Station, Austin, TX 78712. E-mail: eswartzla@aol.com.*

Manuscript received 31 July 2008; revised 11 Dec. 2009; accepted 11 Jan. 2010; published online 11 Feb. 2010.
Recommended for acceptance by C.-L. Wang.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2008-07-0389.
Digital Object Identifier no. 10.1109/TC.2010.38.

the maximum length. Many interesting characteristics of prefixes are exploited using the proposed range representation. An earlier version of the priority trie was presented in [12] by one of the authors. In the proposed priority trie, empty internal nodes in the binary trie are removed by relocating the longest prefixes included in the subtrie rooted by empty nodes. The relocated prefixes are denoted as priority prefixes. Since each empty internal node is replaced by the longest prefix belonged to its subtrie, longer prefixes are compared earlier than shorter prefixes, and hence, the search performance and the memory requirement in the priority trie are significantly improved compared to the binary trie.

This paper is organized as follows: Section 2 briefly summarizes related work. Section 3 presents the IP address lookup problem as a range inclusion problem. The binary trie is characterized using the proposed range representation in Section 4. Section 5 presents the proposed algorithm and some characteristics of the proposed priority trie in the range representation. In Section 6, an implementation example of the proposed algorithm is illustrated. Section 7 shows the performance evaluation results, and Section 8 concludes the paper.

2 RELATED WORK

2.1 Algorithms Based on Hashing

Hashing has been popularly used for layer 2 address lookup which requires exact matching [13]. Hashing converts a long string into a smaller memory address. Collisions are an intrinsic problem of hashing. Broder and Mitzenmacher proposed to use multiple hash functions to reduce collisions [14]. For IP address lookup, hashing is applied to each length of prefixes, and the longest prefix among matched prefixes is selected as the best match [15], [16]. Waldvogel et al. proposed binary searching on hash tables organized by prefix lengths [15]. Lim et al. proposed to use multiple hash functions in reducing collisions in hashing and perform parallel search for every hash table in each length [16]. Another interesting approach is to combine hashing and binary search [17], where hashing is first applied to prefixes of the same length, and for prefixes that collide into the same entry, binary search is applied. Recently, it is proposed to use Bloom filters in reducing the number of hash table accesses [18], [19].

2.2 Algorithms Based on Binary Tries

The binary trie is an attractive data structure for IP address lookup [2], [6], [7], [8], [20], [21], [22]. It is a tree-based data structure which applies linear search on length. Each prefix resides in a node of the trie in which the node level corresponds to the prefix length. At each node, the search proceeds to the left or right according to sequential inspection of address bits starting from the most significant bit. Fig. 1 shows the binary trie for an example set of prefixes. In Fig. 1, black nodes represent prefixes and white nodes represent empty internal nodes. The binary trie structure is simple and easy to implement. It provides good scalability as the number of routing table entries becomes large. However, since the binary trie has many empty nodes which are not involved with routing prefixes, it is inefficient in memory usage and search speed. Moreover, in

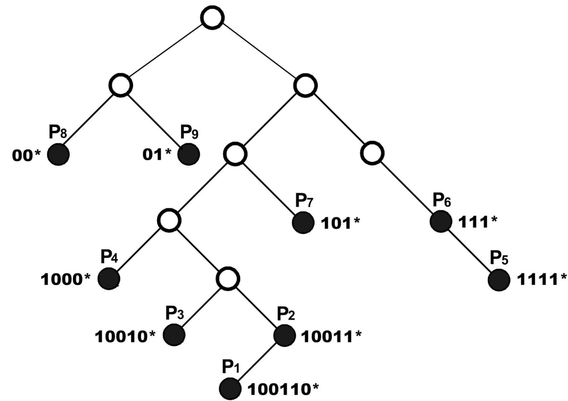

Fig. 1. The binary trie for an example set of prefixes.

the binary trie, shorter prefixes are located at higher levels than longer prefixes, and hence, shorter prefixes are compared earlier than longer prefixes in the search process. In this paper, a higher level is assumed to be a smaller level. For example, the root node is at level 0 which is the highest level, and a leaf node P_1 is at level 6 which is the lowest level in Fig. 1. Therefore, even though a match to an input address is found, the search must continue until a leaf is visited since there could exist a longer prefix that matches the input address. This reduces the search efficiency. The multibit trie inspects more than 1 bit at a time [20], and the path-compressed trie collapses one-way branch nodes [2]. The level-compressed trie applies the multibit trie with path compression [21]. The bitmap algorithm [8] employs an encoding scheme to a multibit trie to reduce the memory penalty associated with a naive implementation. In order to save memory by compression, the Lulea algorithm proposed a compact trie structure for fast lookup [22], but it requires a lot of preprocessing and does not allow incremental updates.

2.3 Algorithms Based on Binary Search for Prefix Values

The binary search tree (BST) [23] algorithm performs binary search on prefix values. To do this, prefixes are sorted according to their values. The BST scheme provides a set of new definitions for comparison of prefixes of different lengths. The BST does not have empty internal nodes, and hence, it minimizes the required memory size. However, binary search on the sorted list of prefix values has a limitation that an ancestor prefix should be compared earlier than its descendent. Depending on the depth of prefix hierarchy, trees can be highly unbalanced and the depth can become very large, as will be shown in Section 7.

As an attempt to reduce the depth of tree, the weighted binary search tree (WBST) [24] considers the number of descendents in selecting the root of each level. The constructed WBST has a shorter depth and is more balanced than the BST. Since disjoint prefixes construct a perfectly balanced tree, the multiple balanced prefix tree (MBPT) [25] constructs multiple balanced trees only with disjoint prefixes. The disjoint prefix tree (DPT) [26] constructs the BST for leaf-pushed prefixes, and hence, it is a perfectly

balanced tree for the extended set of prefixes generated by the leaf-pushing. Recently, it is proposed to build a perfectly balanced binary tree only with the set of disjoint prefixes and to store a prefix vector in each node representing the prefix hierarchy [27].

Binary search on range (BSR) [9], [10], [11] treats each prefix as a range which has a start point and an end point. The start and end points are defined by padding with zeros and ones to the maximum length, respectively. Hence, the length dimension is removed, and a prefix is mapped into a range on the address space between 0 and $2^{32}-1$. By sorting the start points and the end points of ranges based on their values, the address space is divided by disjoint intervals. For each disjoint interval, the BSR scheme precomputes and stores the BMP and the binary search is performed on the sorted list of start and end points. The worst-case number of routing table entries could be two times of the actual number of prefixes. Because of the precomputation of BMPs for each interval, complicated extra data structure is required for the incremental update as in [11] and [28].

3 IP Address Lookup Problem

This section formulates the IP address lookup problem as a range inclusion problem. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_N\}$ be the set of routing prefixes, where N is the number of prefixes, $b_{i,k}$ is the kth bit of prefix P_i ($b_{i,k}$ is either 0 or 1), and n_i is the prefix length of P_i.

The prefix can be represented as a half-open range $r(P_i) = [l_i, u_i) \in [0, 1)$:

$$l_i = \sum_{k=1}^{n_i} b_{i,k} 2^{-k},$$
$$u_i = \sum_{k=1}^{n_i} b_{i,k} 2^{-k} + 2^{-n_i},$$

where l_i and u_i are the lower and upper bounds of the prefix range, respectively. In other words, the length of each prefix determines the width of the prefix range, and each prefix corresponds to the range of 2^{-n_i} starting from the lower bound in the number line between 0 and 1. An IP address A can be represented as a value:

$$v(A) = \sum_{k=1}^{W} a_k 2^{-k}, \tag{1}$$

where a_k is the kth bit of the IP address A and a_k is either 0 or 1. In (1), W is the number of bits in an IP address (W is 32 in IPv4).

Let P_i and P_j be two distinct prefixes in the prefix set $\mathcal{P}$.

Definition 1. *Prefixes P_i and P_j are* disjoint *if $r(P_i) \cap r(P_j) = \emptyset$. For example, prefixes 0^* and 1^* are disjoint, because $r(0*) = [0, 0.5)$, $r(1*) = [0.5, 1)$, and $r(0*) \cap r(1*) = \emptyset$.*

Definition 2. *Prefix P_i is* enclosed *in prefix P_j if $r(P_i) \in r(P_j)$. For example, prefix 11^* is enclosed in prefix 1^*, because $r(11*) = [0.75, 1) \in [0.5, 1) = r(1*)$.*

Lemma 1. *If prefix P_j is a substring of prefix P_i, then prefix P_i is enclosed in prefix P_j.*

Proof. See the Appendix. □

Lemma 2. *If the first n_i bits starting from the most significant bits of IP address A are the same as prefix P_i of length n_i, then $v(A) \in r(P_i)$, i.e., the value of A is enclosed in the range of prefix P_i.*

Proof. See the Appendix. □

Corollary 3. *If an IP address $v(A) \in r(P_i)$, then the IP address A matches the prefix P_i.*

An IP address could match multiple prefixes. The IP address lookup problem is to find the longest prefix among the matched prefixes, namely, the longest prefix matching (LPM).

Let $\mathcal{M}(A)$ represent the set of prefixes that an IP address A matches:

$$\mathcal{M}(A) = \{P_i \in \mathcal{P} : v(A) \in r(P_i)\}. \tag{2}$$

Let $P_{i_1}, P_{i_2}, \ldots, P_{i_n}$ be the elements in $\mathcal{M}(A)$ such that $v(A) \in r(P_{i_1}) \in r(P_{i_2}) \in \cdots r(P_{i_n})$. If the elements are sorted in this way, then P_{i_1} corresponds to the smallest range in $\mathcal{M}(A)$, and hence, the P_{i_1} is the longest prefix that includes the address A by Lemmas 1 and 2.

In this range representation for the IP address lookup problem, the IP lookup problem is to find the prefix corresponding to the smallest range in $\mathcal{M}(A) \in \mathcal{P}$ for a given IP address A.

4 Binary Trie

This section presents the conventional binary trie in the context of the IP lookup problem in Section 3. In the binary trie, a node has a fixed location which can be defined by a bit string. For example, the left child node of the root node has the bit string of 0, and the right child node has the bit string of 1. If a prefix corresponding to the bit string of a node exists in an IP routing table, the routing information of the prefix is stored in the node and the node is nonempty. If there is no prefix corresponding to the bit string of a node, the node is empty. If the longest number of prefix bits W is 32 (as in IPv4), there are $2^0 + 2^1 + \cdots + 2^{32} = 2^{33} - 1$ possible nodes for a binary trie in IPv4.

Let $B(x)$ represent the bit string of a node x. In a nonempty node x in the binary trie, the prefix corresponding to the node x is the bit string $B(x)$. In the binary trie, an empty node x is generated if there is no prefix corresponding to the bit string $B(x)$ and there exists at least one descendant node corresponding to a prefix which has the bit string $B(x)$ as its substring. The number of empty nodes in the binary trie is dependent on the prefix set of the IP routing table.

Ranges can be defined for the bit strings in the binary trie in a similar way to that used for prefixes in Section 3. Let $b_{x,k}$ be the kth bit in bit string $B(x)$ of node x and let n_x be the length of bit string $B(x)$. A bit string can also be represented as a half-open range $r(B(x)) = [l_x, u_x) \in [0, 1)$:

$$l_x = \sum_{k=1}^{n_x} b_{x,k} 2^{-k},$$
$$u_x = \sum_{k=1}^{n_x} b_{x,k} 2^{-k} + 2^{-n_x},$$

where l_x and u_x are the lower and upper bounds of bit string $B(x)$, respectively.

Let x and y be two distinct nodes in the binary trie.

Definition 3. *Nodes x and y are* disjoint *if* $r(B(x)) \cap r(B(y)) = \emptyset$.

Definition 4. *Node x and prefix P_i are* disjoint *if* $r(B(x)) \cap r(P_i) = \emptyset$.

Definition 5. *Prefix P_i is* enclosed *in node x if* $r(P_i) \in r(B(x))$.

Lemma 4. *If $B(x)$ is a substring of a prefix P_i, then prefix P_i is* enclosed *in node x.*

Lemma 4 can be proved in a similar way as the proof for Lemma 1.

Lemma 5. *If nodes x and y are distinct nodes at the same level in the binary trie, then $r(B(x)) \cap r(B(y)) = \emptyset$, and the nodes x and y are disjoint.*

Proof. See the Appendix. □

Corollary 6. *If $B(x)$ is a substring of prefix P_i and $B(y)$ is a substring of prefix P_j, and the nodes x and y are distinct nodes at the same level in the binary trie, then the prefixes P_i and P_j are disjoint.*

5 The Proposed Algorithm

5.1 Priority Trie

Let $|r(P_i)|$ represent the width of range $r(P_i) = [l_i, u_i)$. Then, $|r(P_i)| = u_i - l_i = 2^{-n_i}$, where n_i is the length of prefix P_i. Prefix P_i has a higher priority than prefix P_j if the prefix length of P_i is longer than that of prefix P_j ($n_i > n_j$), i.e., $|r(P_i)| < |r(P_j)|$. Hence, a longer prefix has a higher priority. If $n_i = n_j$, the priority can be arbitrary. Using this priority, the elements of prefix set $\mathcal{P} = \{P_1, P_2, \ldots, P_N\}$ can be sorted such that P_i has higher priority than P_j for $i < j$.

Besides the fact that the binary trie has many empty internal nodes, another intrinsic problem is that longer prefixes are stored at lower levels, and hence, they are compared later than shorter prefixes. Therefore, even if a match is found, search has to be continued until a leaf is visited. If prefixes are reversely assigned, in other words, if longer prefixes are associated with higher level nodes and shorter prefixes are associated with lower level nodes, searching finishes immediately when there is a match. However, in order for a shorter prefix to be associated with a node in the lower level than its length, the prefix has to be duplicated by 2^{L-n_i} times, where n_i is the prefix length and L is the level. To avoid the duplication of prefixes, the proposed priority trie associates higher priority prefixes with the empty nodes of a binary trie.

Let $\mathcal{E}(x)$ represent the set of prefixes that are enclosed in node x. Let $P(x)$ represent the prefix with the highest priority in $\mathcal{E}(x)$. If a node x is empty in the binary trie, prefix $P(x)$ is stored in empty node x in the proposed priority trie. Once prefix $P(x)$ is stored in node x, the node is marked as a *priority node*. If node y is nonempty in the binary trie, the prefix at node y is bit string $B(y)$ (the prefix is not changed), and node y is marked as an *ordinary node*. Let $S(x)$ represent the stored prefix in node x. If the node x is a priority node, $S(x) = P(x)$, and otherwise, $S(x) = B(x)$.

Lemma 7. *If nodes x and y are two distinct ordinary nodes at the same level in the priority trie, the prefixes at nodes x and y are disjoint.*

Proof. See the Appendix. □

Lemma 8. *If nodes x and y are two distinct priority nodes at the same level in the priority trie, the prefixes at nodes x and y are disjoint.*

Proof. See the Appendix. □

Lemma 9. *If node x is an ordinary node and node y is a priority node at the same level in the priority trie, the prefixes at nodes x and y are disjoint.*

Proof. See the Appendix. □

Lemma 10. *Let P_i and P_j be the prefixes at nodes x and y, respectively. If nodes x and y are distinct nodes at the same level in the priority trie, then prefixes P_i at node x and P_j at node y are disjoint.*

Proof. See the Appendix. □

Since higher priority prefixes are only associated with empty nodes and each prefix is located in the same level or in a higher level than its prefix length, there is no need for prefix duplication in the proposed algorithm.

5.2 Build Process

There are two possible ways of building the priority trie. First, a conventional binary trie is built in which a prefix with length n_i is stored at a node in level n_i, and then, the binary trie is traversed. If a nonempty node is found, the node is marked as an ordinary prefix. If an empty node x is found, the prefixes enclosed in x, i.e., $\mathcal{E}(x)$, are sorted, the node storing the highest priority prefix $P(x)$ is removed, and the prefix $P(x)$ is stored into node x. The node x is now marked as a *priority node*. Since the highest priority prefix which was a leaf node is relocated to a higher level node, there can be empty nodes which do not have a prefix in lower levels. They also have to be deleted. Note that the prefix can be moved only to a higher level in this step. If prefix P_i at level n_i in the binary trie is moved to level $L (L < n_i)$ in the priority trie, the level is reduced by $n_i - L$ for this prefix. This step is continued until the binary trie is completely traversed or there are no empty nodes. This method of building the priority trie causes high computational complexity.

The second way of building the priority trie is by incremental updates. Prefixes are primarily sorted in the decreasing order of their lengths. Prefixes with the same length are not necessarily sorted. Starting from a prefix with the longest length, the prefix is stored into the root node and the node is marked as a *priority node*. A next prefix is stored in the left or right child of the root node depending on the first bit of the prefix and marked as a *priority node*. This processing is repeated for every prefix. However, a prefix with length n_i should be stored at a node in level L, which is less than or equal to n_i. If $L = n_i$, the node is marked as an *ordinary node*. In other words, in storing a prefix, if the node of the level which is the same as the length of the prefix was already a priority node, that is, if the node was already occupied by another priority prefix, the node should be converted to an ordinary node and the priority prefix

should be relocated into a lower level node. In relocating the priority prefix, if a priority node is encountered and the node has a shorter length prefix matching to the priority prefix, the priority prefix needs to be stored in the node. The same processing is repeated for the prefix which lost its place. The pseudocode for the incremental build process can be represented as follows:

```
BuildPriorityTrie ( ) {
    Sort Prefixes as P_1, P_2, ..., P_N;
    for (i = 1; i ≤ N; i++) {
        x = root; p = P_i; L = 0;
        BuildNode (x, p, L);
    }
}
BuildNode (node x, prefix p, level L) {
    if (x is empty) {
        S(x) = p; // Store p at x
        if (n(p) > L) Mark x as a priority node;
        else Mark x as an ordinary node;
        return;
    }
    else { // x is nonempty
        if (n(p) == L and x is a priority node) {
            tmp = S(x); S(x) = p; p = tmp;
            Mark x as an ordinary node;
        }
        else if ( (n(p) > n(S(x))) and (r(p) ∈ r(S(x))) and
                (x is a priority node) ) {
            tmp = S(x); S(x) = p; p = tmp;
        }
        L++;
        y is the child node of x identified by the Lth bit of p;
        // 0: left child, 1: right child
        BuildNode(y, p, L);
    }
}
```

As shown in the pseudocode, once the BuildNode function is called, it is recursively called if the node x is nonempty. The function is completed when node x is empty and the prefix is stored into x. Hence, in the build process, a single node is created for each P_i, and this process is repeated N times for $i = 1, \dots, N$. Therefore, the number of nodes in the final priority trie becomes N, the number of prefixes.

The complexity in inserting a prefix is proportional to the current depth of the trie. Hence, the complexity in building the priority trie incrementally is less than O(ND_p), where D_p is the depth of the priority trie with N prefixes.

5.3 Search Process

Lemma 11. *Let x be a node at level L. If an input IP address A matches P_m at a priority node x, then P_m has the smallest prefix range in $\mathcal{M}(A)$, i.e., P_m is the BMP in the given prefix set $\mathcal{P} = \{P_1, P_2, \dots, P_N\}$.*

Proof. See the Appendix. □

The search process starts from the root node. At each node, an input address is compared with the stored prefix to find out whether the input matches the prefix at the node. By Lemma 2, if the first n_i bits of the input address are the same as those of the stored prefix with length n_i, they are determined as matched. Lemma 11 states that the search process can be finished at a priority node in any level without searching lower levels if the input matches the prefix at a priority node. This means that the BMP for IP address A can be found without searching all the prefixes in $\mathcal{M}(A)$, which is a very important characteristic of the proposed algorithm for reducing the number of memory accesses. If the input matches the prefix at an ordinary node, the stored prefix is remembered as the current best match. If the input matches the prefix at an ordinary node or the input does not match in the current level L, the $(L+1)$th bit of the input address is examined. If the bit is 0, the search continues on the left child, and otherwise, the search continues on the right child. The pseudocode for the search process can be represented as follows and the search complexity is O(D_p):

```
Search (IP address A, node x)
{
    BMP = *; // default prefix
    x = root; L = 0;
    do {
      if (v(A) ∈ r(S(x))) {
      // Input IP address matches a stored prefix S(x)
            BMP = S(x);
            if (x is a priority node)
            // Matched prefix is a priority prefix
            break;
        }
        L++;
        y is the child node of x identified by the Lth bit of A;
        // 0: left child, 1: right child
        x ← y;
    } while (x is a valid node);
    return BMP;
}
```

As shown in the pseudocode of search procedure, the input address is compared with the stored prefix at each node in the proposed priority trie, while the comparison is not required in the regular binary trie. However, search speed is evaluated by the number of memory accesses required to perform the search procedure since the memory lookup is the slowest operation. Both tries require a memory lookup in accessing each node. The time to perform the comparison is negligible compared with the time to access the memory.

5.4 Update

The proposed priority trie provides incremental update for a prefix deletion or insertion. For the insertion of a prefix in the proposed algorithm, there are two cases that multiple nodes are affected by a prefix insertion. The first case is when the inserted prefix matches a priority prefix and is longer than the priority prefix. The second case is that the node of the inserted prefix to be stored as an ordinary prefix was preoccupied by another priority prefix. In these cases, the inserted prefix takes the place, and for the prefix, for which its place was taken away, the same procedure is repeated. If the number of nested networks is small as

TABLE 1
An Example Prefix Set

P_i	Prefix	Lower bound(l_i)	Upper bound(u_i)	$\mathcal{E}(P_i)$
P_1	100110*	0.59375	0.609375	$\emptyset$
P_2	10011*	0.59375	0.625	$\{P_1\}$
P_3	10010*	0.5625	0.59375	$\emptyset$
P_4	1000*	0.5	0.5625	$\emptyset$
P_5	1111*	0.9375	1.0	$\emptyset$
P_6	111*	0.875	1.0	$\{P_5\}$
P_7	101*	0.625	0.75	$\emptyset$
P_8	00*	0.0	0.25	$\emptyset$
P_9	01*	0.25	0.5	$\emptyset$

expected from the known statistics [28], the number of nodes affected by a prefix insertion is statistically limited to small numbers. In other cases, the inserted prefix is stored at a leaf node by creating a new leaf. The pseudocode for the insertion process can be represented as follows and the insertion complexity is O(D_p):

Insertion (prefix p) {
  $x =$ root; $L = 0$;
  BuildNode (x, p, L);
}

For the deletion of a prefix in the proposed priority trie, the prefix is searched and deleted. Now, the node becomes empty. If the empty node has any child nodes, the prefix stored in one of its child nodes is relocated to the node. If the relocated prefix was a priority prefix, the node is marked as a priority node, and otherwise, the node is marked as an ordinary node. In case an ordinary node is relocated into a higher level node, even though the prefix is stored in a level shorter than its length, the node type should not be changed since it is possible that longer prefixes exist in lower levels. The deletion is simple in this way. This process is repeated until a leaf node is deleted. The deletion complexity is O($2D_p$) since each node from the deleted node down to a leaf node is read once and written once. The pseudocode for the deletion process can be represented as follows:

Deletion (prefix p) {
  $q =$ Search(p, x);
  if ($p == q$) // Prefix p exists in priority trie
    DeleteNode (x, p);
}

DeleteNode (node x, prefix p) {
  z and w are the left and right child node of x, respectively;
  if (z and w are null) // no child (leaf node)
    Delete x; **return**;
  else {
    if (z is nonnull) $y \leftarrow z$;
    else $y \leftarrow w$;
    $S(x) = S(y)$;
    if (y is a priority node) Mark x as a priority node;
    else Mark x as an ordinary node;
    DeleteNode (y, $S(y)$);
  }
}

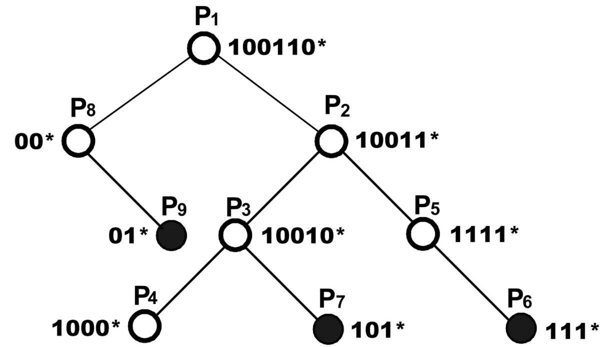

Fig. 2. The proposed priority trie for the example set of prefixes.

6 An Implementation Example

Table 1 shows the characteristics of each prefix for the example prefix set in Fig. 1. The prefixes in Table 1 are sorted in the order of decreasing priority. The range that each prefix covers is shown. The enclosed prefixes of each prefix are also shown. Longer prefixes have higher priorities, and if the lengths are the same, the priority order is arbitrary.

Fig. 2 shows the proposed priority binary trie built using the proposed build algorithm for the example prefix set in Fig. 1. Priority nodes are represented by white nodes and ordinary prefixes are represented by black nodes. As shown, the depth of the trie is reduced to 3 in the proposed priority trie from 6 in the binary trie in Fig. 1.

Table 2 shows the routing table implementing the proposed priority trie. The first field of the routing table is the priority field. If the stored prefix in this entry is the priority prefix, the field is set, and otherwise, it is reset. The next two fields are the stored prefix and the length of the prefix. By Lemma 2, if the first n_i bits of a given input address are the same as the stored prefix where n_i is the length of the stored prefix, then the input is included in the range covered by the stored prefix and determined as a match. The next two fields are two pointers for the children of the current node, and the entry address is assumed to start from 1. The routing information is not shown in this table for simplicity. As

TABLE 2
An Example Routing Table Implementation for the Proposed Trie

Priority/ Ordinary	Prefix P_i	Prefix length (n_i)	Left pointer	Right pointer
1	100110*	6	8	2
1	10011*	5	3	5
1	10010*	5	4	7
1	1000*	4	-	-
1	1111*	4	-	6
0	111*	3	-	-
0	101*	3	-	-
1	00*	2	-	9
0	01*	2	-	-

TABLE 3
The Number of Memory Accesses for Each Prefix Match

BMP	Binary trie	Priority trie
P_1	7	1
P_2	6	2
P_3	6	3
P_4	5	4
P_5	5	2
P_6	4	4
P_7	4	4
P_8	3	2
P_9	3	3

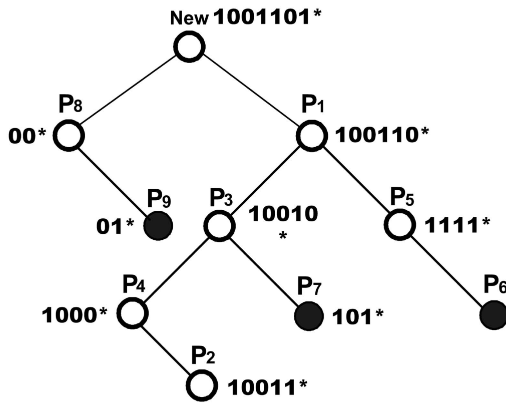

Fig. 3. The priority trie updated by adding a new prefix as a priority node.

shown in Table 2, the number of memory entries is the same as the number of prefixes in the proposed priority trie.

Table 3 compares the number of memory accesses between the binary trie and the proposed priority trie for cases where the inputs match with each prefix as the BMP. If the probability of each prefix being matched with inputs is assumed to be equal, the average number of memory accesses would be 2.78 for the proposed scheme, while it is 4.78 for the binary trie.

Fig. 3 shows the case where a new prefix 1001101^* is inserted into the priority trie in Fig. 2. Since the new prefix matches P_1 and is longer than the prefix P_1, the new prefix takes the node of P_1. Similarly, the prefix P_1 takes the node of the prefix P_2. Since the prefix P_2 does not match any other lower level prefix, it is stored as a new priority leaf. In this case, a prefix insertion affects three entries.

Fig. 4 shows the case where the prefix P_2 is deleted from the priority trie in Fig. 2. The node has two children. The left child, which is the prefix P_3, is relocated to the node and the node is marked as a priority node. Similarly, the prefix P_4 is relocated to the node of P_3 and marked as a priority node. The leaf node is empty, and hence, it is deleted. In this case, a prefix deletion affects three entries.

7 Simulations and Performance Evaluation

Simulations have been performed using C language for six real prefix sets downloaded on May 2006 from backbone routers [29]. The proposed priority binary trie and the proposed priority multibit (2 bit) trie have been built using the incremental build process, as shown in Section 5.

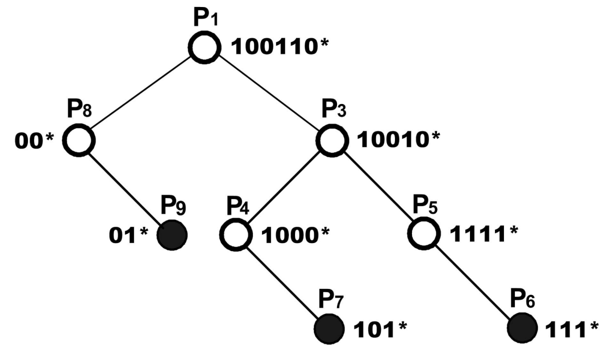

Fig. 4. The priority trie updated by deleting prefix P_2.

TABLE 4
Performance of the Proposed Priority Binary Trie

Prefix Set	N	N_p	W	D_p	T_a	M(Kbyte)
MAE-West1	14,553	14,199	32	24	16.66	119
Aads	20,204	19,568	32	24	17.43	170
MAE-West2	29,584	26,671	32	24	18.22	249
PORT 80	112,310	50,091	32	28	20.35	1001
Grouptlcom	170,601	70,525	32	24	20.76	1,562
Telstra	227,223	119,149	32	32	22.86	2,080

TABLE 5
Performance of the Proposed Priority Multibit Trie

Prefix Set	N	N_p	N_e	W	D_p	T_a	M(Kbyte)
MAE-West1	14,553	14,388	2,337	32	12	9.70	214
Aads	20,204	19,124	3,208	32	12	10.05	297
MAE-West2	29,584	23,016	6,273	32	13	10.54	455
PORT 80	112,310	36,362	21,630	32	16	11.27	1,790
Grouptlcom	170,601	47,073	32,732	32	13	11.45	2,710
Telstra	227,223	75,929	41,850	32	16	12.51	3,850

Tables 4 and 5 show the performance evaluation results of the proposed priority binary trie and priority multibit trie, respectively, in terms of the number of routing prefixes (N), the number of priority prefixes (N_p) among routing prefixes, the maximum prefix length (W), the depth of the proposed priority trie (D_p), the average number of memory accesses (T_a) for an address lookup, and the memory requirement (M).

The memory requirement of the proposed priority trie is directly proportional to the number of prefixes since there are no empty nodes. The memory requirements shown in the tables account for the data structure of the proposed priority trie shown in the example in Table 2. The entry width of the routing table can be designed with 39 bits (1 bit for the node identity, i.e., priority node or ordinary node, 25 bits for the prefix considering that the shortest prefix length is 8 bits, 5 bits for the prefix length, and 8 bits for routing information) plus two fields for child pointers. The number of bits for the child pointers depends on the size of routing data set. If 20 bits are allocated for each of the child pointers (assuming that the number of prefixes is no more than a million), the memory requirement to store a prefix is 10 bytes. Therefore, once the estimated number of prefixes is known for a router to be designed, the required memory

TABLE 6
Insertion Performance of the Proposed Priority Binary Trie

Prefix Set	N_{75}	N_i	P_w	P_a	C_w	C_a
MAE-West1	10,915	3,638	25	18.00	3	2.01
Aads	15,153	5,051	25	18.75	3	2.01
MAE-West2	22,188	7,396	25	19.72	4	2.03
PORT 80	84,233	28,077	29	21.42	6	2.26
Grouptlcom	127,951	42,650	25	21.74	5	2.25
Telstra	170,418	56,805	33	23.87	5	2.19

TABLE 7
Deletion Performance of the Proposed Priority Binary Trie

Prefix Set	N	N_d	P_w	P_a	C_w	C_a
MAE-West1	14,553	3,638	25	18.00	7	2.01
Aads	20,204	5,051	25	18.75	7	2.02
MAE-West2	29,584	7,396	25	19.71	9	2.06
PORT 80	112,310	28,077	29	21.21	14	2.55
Grouptlcom	170,601	42,650	25	21.54	15	2.50
Telstra	227,223	56,805	33	23.71	18	2.39

amount to store the forwarding table is determined since the node size of the proposed priority trie is fixed and the number of entries is the same as the number of prefixes. Hence, the proposed algorithm is well suited for hardware implementation.

For IPv6, the binary trie for IPv6 prefixes is expected to be very sparse and there will be many empty nodes since it has 128-bit address space. Since all of the empty nodes are replaced by priority prefixes in the proposed priority trie, the performance improvement will be much larger for IPv6.

As shown in the number of priority prefixes in Table 4, more than 90 percent of the prefixes for the first three prefix sets are stored by priorities, and this means that the ordinary binary trie has many empty nodes. The more priority nodes would result in the better search performance with the proposed algorithm. The average number of memory accesses is about 16-23, and it does not degrade much as the prefix set grows.

In constructing the priority multibit trie, prefixes are replicated to be extended to a certain length depending on the predetermined stride. Hence, extra nodes are created in the multibit trie. The result of the proposed priority multibit trie shown in Table 5 is for the 2-bit stride. In Table 5, N_e is equal to the total number of nodes in the proposed priority multibit trie minus the number of prefixes in the prefix set. Each empty internal node is replaced by the longest prefix belonging to the subtrie of the empty node in the proposed priority multibit trie. As shown in Table 5, the depth is 12-16 and the average number of memory accesses of the priority multibit trie is about 10-13. Since the performance of the proposed algorithm in terms of the trie depth, the average number of memory accesses, and the required memory size does not degrade much as the prefix set grows, the proposed scheme is good in scalability with large routing data.

Simulations for evaluating the incremental update performance of the proposed priority binary trie have been performed. For insertion, the priority trie was primarily built using the 75 percent of prefixes randomly selected for those prefix sets in Table 4, and the performance was measured for inserting the remaining prefixes to the trie one by one. The performance was evaluated in two terms: the number of pass-through nodes and the number of changed nodes. The pass-through nodes mean the nodes that need to be read and compared in inserting a prefix. The changed nodes mean the nodes that need to be read, compared, and replaced with another prefix in inserting a prefix. Here, the changed nodes are included in counting the pass-through nodes. Table 6 shows the number of prefixes used in building a proposed priority trie (N_{75}), the number of prefixes used in evaluating the insertion performance (N_i), the worst-case number (P_w) and the average number (P_a) of pass-through nodes, and the worst-case number (C_w) and the average number (C_a) of changed nodes. As shown in Table 6, the average number of changed nodes is 2.0 to 2.3, and hence, the proposed algorithm provides the incremental insertion.

For deletion, the priority trie was built using the 100 percent of prefixes in the prefix sets. The performance was measured in deleting the 25 percent of the prefixes one by one from the trie. The performance was evaluated using the same two terms: the number of pass-through nodes and the number of changed nodes. Table 7 shows the number of prefixes in the prefix set (N), the number of prefixes deleted from the trie (N_d), the worst-case number (P_w) and the average number (P_a) of pass-through nodes, and the worst-case number (C_w) and the average number (C_a) of changed nodes. As shown in Table 7, the average number of changed nodes are 2.0 to 2.5, and hence, the proposed algorithm provides the incremental deletion.

Simulations have been performed with existing binary search schemes for two routing sets, one with 112K entries and the other with 227K entries. Simulation using those two prefix sets can show the scalability of each algorithm when the number of prefixes is doubled for the sets with a large number of prefixes. Table 8 shows the performance comparison in terms of the worst-case number of memory accesses (T_w), the average number of memory accesses (T_a), the required memory size (M), and the number of extra nodes required in the algorithms (N_e). As shown in Table 8, the proposed priority multibit trie is the best in the worst-case number of memory accesses. For the average number of memory accesses, the LC-trie is the best and the BSR and the proposed priority multibit trie are the next.

Considering the required memory size, the BSR and the proposed priority binary trie show the best performance. Even though the proposed priority trie requires storing a prefix value in each node, while it is not required in the binary trie, the required memory size is smaller than the binary trie since the proposed priority trie does not include empty nodes. LC-trie requires huge memory because of the redundant nodes created during the process of level compression as shown in N_e. Since the BSR algorithm requires precomputation of the best matching prefixes in each entry, it requires a very complicated data structure for the incremental update [11], [28], while the proposed algorithm provides incremental update.

Regarding the scalability, the binary trie and the BSR show very good characteristics since the performance was not degraded much when the size of prefix sets is doubled. However, it is shown that the LC-trie has a scalability issue

TABLE 8
Comparison with Other Algorithms

Algorithm		Binary trie [2]	BST [23]	WBST [24]	BSR [9]	LC-trie [21]	Proposed priority (binary)	Proposed priority (multi-bit)
Incremental update		yes	no	no	complex	no	yes	yes
Port80 (112,310)	T_w	32	44	36	18	25	28	16
	T_a	22.15	25.82	20.44	11.42	10.6	20.35	11.27
	M (Mbyte)	1.29	1.25	1.25	0.96	7.3	1.00	1.79
	N_e	112,907	0	0	72,063	548,295	0	21,630
Telstra (227,223)	T_w	32	66	39	19	19	32	16
	T_a	24.64	30.80	23.96	11.07	11.2	22.86	12.51
	M (Mbyte)	2.59	2.60	2.60	1.76	105	2.08	3.85
	N_e	225,082	0	0	124,795	9,350,438	0	41,850

since the required memory amount was grown 14 times and the number of extra nodes was grown 17 times when the size of prefix sets is doubled. It is also shown that the BST has a scalability issue in the worst-case number of memory accesses. The proposed algorithm shows very good scalability in which all of the performance metric is not degraded when the size of prefix sets is doubled.

8 Conclusion

Network devices face two areas of scaling challenges: bandwidth scaling and population scaling [3]. Bandwidth scaling occurs because links are getting faster and the Internet traffic keeps growing. Population scaling occurs as more end points are added to the Internet. These scaling issues makes wire speed forwarding in the Internet routers more challenging.

This paper presents a new mathematical formulation of the IP address lookup problem as the range inclusion problem. In the previous range matching algorithm of [9], each prefix is extended to the maximum length for the start point and the end point of its range, ranges are divided by disjoint intervals, and BMPs for each disjoint interval need to be precomputed in order to apply binary search. In the proposed range representation, it is not necessary to extend each prefix into the maximum length and each prefix is represented as a range included in a number line between 0 and 1. Each prefix covers the range of 2^{-n_i}, where n_i is the length of prefix. The BMP is the smallest range that includes the given input address. The characteristics of the binary trie are described using the proposed range representation. A new binary search algorithm based on the priority trie is proposed in this paper. The proposed algorithm is based on a binary trie, but each empty node in the binary trie is removed by placing the priority prefix which is the longest prefix included in the subtrie rooted by the empty node. Search in the proposed algorithm finishes either at a match with a priority prefix or at a leaf, and hence, the search performance is improved significantly. The proposed algorithm also provides incremental update by limiting the number of changed nodes in inserting or deleting a prefix. Since the proposed algorithm is based on a binary trie, it is conceptually simple and can be implemented easily. Simulation results show that the required memory size and the number of memory accesses in the proposed algorithm increase moderately as the size of prefix set is increased, and hence, the proposed algorithm is good in scalability toward large routing data.

Appendix

Proof of Lemma 1. Assume that prefix P_j is a substring of prefix P_i. The lower and the upper bounds of prefix ranges $r(P_j) = [l_j, u_j)$ and $r(P_i) = [l_i, u_i)$ can be represented as

$$l_j = \sum_{k=1}^{n_j} b_{j,k} 2^{-k}, u_j = \sum_{k=1}^{n_j} b_{j,k} 2^{-k} + 2^{-n_j}, l_i = \sum_{k=1}^{n_i} b_{i,k} 2^{-k}, \text{ and}$$

$$u_i = \sum_{k=1}^{n_i} b_{i,k} 2^{-k} + 2^{-n_i},$$

where $n_j < n_i$ and $b_{j,k} = b_{i,k}$ for $k = 1, \dots, n_j$. Then,

$$\begin{aligned} l_i &= \sum_{k=1}^{n_j} b_{i,k} 2^{-k} + \sum_{k=n_j+1}^{n_i} b_{i,k} 2^{-k} \\ &= l_j + \sum_{k=n_j+1}^{n_i} b_{i,k} 2^{-k} \\ &\geq l_j, \end{aligned}$$

and

$$\begin{aligned} u_i &= \sum_{k=1}^{n_j} b_{i,k} 2^{-k} + \sum_{k=n_j+1}^{n_i} b_{i,k} 2^{-k} + 2^{-n_i} \\ &\leq \sum_{k=1}^{n_j} b_{i,k} 2^{-k} + \sum_{k=n_j+1}^{n_i} 2^{-k} + 2^{-n_i} \\ &= \sum_{k=1}^{n_j} b_{j,k} 2^{-k} + (2^{-n_j} - 2^{-n_i}) + 2^{-n_i} \\ &= \sum_{k=1}^{n_j} b_{j,k} 2^{-k} + 2^{-n_j} = u_j. \end{aligned}$$

Hence, $r(P_i) \in r(P_j)$ and prefix P_i is enclosed in prefix P_j.

Proof of Lemma 2. Let $v(A) = \sum_{k=1}^{W} a_k 2^{-k}$, $l_i = \sum_{k=1}^{n_i} b_{i,k}$, and $u_i = \sum_{k=1}^{n_i} b_{i,k} + 2^{-n_i}$. Assume that the n_i bits of A are the same as the n_i bits of P_i, i.e., $a_k = b_{i,k}$ for $k = 1, \dots, n_i$. Then,

$$\begin{aligned} l_i &= \sum_{k=1}^{n_i} b_{i,k} 2^{-k} = \sum_{k=1}^{n_i} a_k 2^{-k} \\ &\leq \sum_{k=1}^{n_i} a_k 2^{-k} + \sum_{k=n_i+1}^{W} a_k 2^{-k} \\ &= \sum_{k=1}^{W} a_k 2^{-k} = v(A), \end{aligned}$$

and

$$\begin{aligned} v(A) &= \sum_{k=1}^{W} a_k 2^{-k} = \sum_{k=1}^{n_i} b_{i,k} 2^{-k} + \sum_{k=n_i+1}^{W} a_k 2^{-k} \\ &\le \sum_{k=1}^{n_i} b_{i,k} 2^{-k} + \sum_{k=n_i+1}^{W} 2^{-k} \\ &= \sum_{k=1}^{n_i} b_{i,k} 2^{-k} + 2^{-n_i} - 2^{-W} \\ &< \sum_{k=1}^{n_i} b_{i,k} 2^{-k} + 2^{-n_i} = u_i. \end{aligned}$$

Hence, $v(A) \in [l_i, u_i) = r(P_i)$.

Proof of Lemma 5. Assume that nodes x and y are distinct nodes at the same level L. Then, the number of bits is same, and $n_x = n_y = L$. Assume that $l_x < l_y$, without loss of generality, and it is necessary to prove that $u_x \le l_y$. Let $b_{x,m}$ be the first bit such that $b_{x,m} \ne b_{y,m}$. Then, $b_{x,k} = b_{y,k}$ for $k = 1, \ldots, m-1$ and $b_{x,m} = 0, b_{y,m} = 1$ since $l_x < l_y$:

$$\begin{aligned} u_x &= \sum_{k=1}^{L} b_{x,k} 2^{-k} + 2^{-L} \\ &= \sum_{k=1}^{m-1} b_{x,k} 2^{-k} + b_{x,m} 2^{-m} + \sum_{k=m+1}^{L} b_{x,k} 2^{-k} + 2^{-L} \\ &\le \sum_{k=1}^{m-1} b_{x,k} 2^{-k} + \sum_{k=m+1}^{L} 2^{-k} + 2^{-L} \\ &= \sum_{k=1}^{m-1} b_{y,k} 2^{-k} + (2^{-m} - 2^{-L}) + 2^{-L} \\ &\le \sum_{k=1}^{m-1} b_{y,k} 2^{-k} + 2^{-m} + \sum_{k=m+1}^{L} b_{y,k} 2^{-k} \\ &= \sum_{k=1}^{L} b_{y,k} 2^{-k} = l_y. \end{aligned}$$

Hence, $r(B_x) \cap r(B_y) = [l_x, u_x) \cap [l_y, u_y) = \emptyset$, and the nodes x and y are disjoint.

Proof of Lemma 7. In the priority trie, prefixes $B(x)$ and $B(y)$ are stored at the ordinary nodes x and y, respectively. By Lemma 5, $r(B(x)) \cap r(B(y)) = \emptyset$. Hence, prefixes $B(x)$ at node x and $B(y)$ at node y are disjoint.

Proof of Lemma 8. In the priority trie, prefixes $P(x)$ and $P(y)$ are stored at priority nodes x and y, respectively. Since prefix $P(x)$ has the highest priority in $\mathcal{E}(x)$, where $\mathcal{E}(x)$ is the set of prefixes that are enclosed in node x, $r(P(x)) \in r(B(x))$. Similarly, $r(P(y)) \in r(B(y))$. By Lemma 5, $r(B(x)) \cap r(B(y)) = \emptyset$. Hence, $r(P(x)) \cap r(P(y)) = \emptyset$, and prefixes $P(x)$ at node x and $P(y)$ at node y are disjoint.

Proof of Lemma 9. In the priority trie, the prefix which has bit string $B(x)$ is stored at ordinary node x and prefix $P(y)$ is stored at priority node y. Since $P(y) \in \mathcal{E}(y)$, $r(P(y)) \in r(B(y))$. By Lemma 5, $r(B(x)) \cap r(B(y)) = \emptyset$. Hence, $r(B(x)) \cap r(P(y)) = \emptyset$, and the prefixes having the bit strings $B(x)$ at node x and $P(y)$ at node y are disjoint.

Proof of Lemma 10. There can be four cases for node types. Case 1: Nodes x and y are ordinary nodes. Case 2: Node x is an ordinary node and node y is a priority node. Case 3: Node x is a priority node and node y is an ordinary node. Case 4: Nodes x and y are priority nodes.

In Case 1, $P_i = B(x)$ and $P_j = B(y)$. By Lemma 7, $B(x)$ and $B(y)$ are disjoint. Hence, prefixes P_i and P_j are disjoint. In Case 2, $P_i = B(x)$ and $P_j = P(y)$. By Lemma 9, $B(x)$ and $P(y)$ are disjoint. Hence, prefixes P_i and P_j are disjoint. Case 3 is similar to Case 2. In Case 4, $P_i = P(x)$ and $P_j = P(y)$. By Lemma 8, $P(x)$ and $P(y)$ are disjoint. Hence, prefixes P_i and P_j are disjoint.

Proof of Lemma 11. The proof is by induction. If $L = 0$, x is the root node. Since x is a priority node, $P(x) = P_1$ and $v(A) \in r(P_1)$. Since P_1 has the highest priority in $\mathcal{P}$, P_1 at priority node x is the BMP.

The search proceeds from level 0 to lower levels. Assume that there is no priority node that matches A in levels 0 through $L-1$ and the BMP does not exist in levels 0 through $L-1$ for induction. Now, assume that A matches P_m at a priority node x at level L.

There can be three cases that the BMP can exist. Case 1: The BMP exists at levels $L+1, \ldots$. Case 2: The BMP exists at another node at level L. Case 3: P_m at node x of level L is the BMP.

If P_n is the BMP at node y in a lower level $K (K > L)$, then $v(A) \in r(P_n)$ and the P_n has the smallest range including $v(A)$, and hence, $r(P_n) \in r(P_m)$ and P_n has higher priority than P_m. In this case, P_n should have been stored at a priority node at a higher level $L' (L' \le L)$ in the build process and the address A should have matched P_n at the level L' in the search process. This is a contradiction, and hence, Case 1 is removed.

By Lemma 8, two distinct prefixes at the same level in the priority trie are disjoint. If P_n is a prefix at a node $y (y \ne x)$ at the level L, $r(P_n) \cap r(P_m) = \emptyset$. Since $v(A) \in r(P_m)$, $v(A)$ cannot be enclosed in $r(P_n)$, i.e., $r(A) \notin r(P_n)$. Hence, the BMP cannot be located at another node $y (y \ne x)$ at the level L, and hence, Case 2 is removed. Thus, the matched prefix P_m at priority node x in level L is the BMP.

ACKNOWLEDGMENTS

The research of the first author (H. Lim) was supported by LG Yonam Foundation under the overseas research professor program and by the MKE(The Ministry of Knowledge Economy), Korea, under the HNRC-ITRC support program supervised by the NIPA (NIPA-2010-C1090-1011-0010). This work was also supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (2010-0000483). The work of the second author (C. Yim) was supported by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD) (KRF-2008-013-D00083) and by the MKE under the ITRC support program supervised by the NIPA (NIPA-2010-C1090-1031-0003). The preparation of this paper would not have been possible without the efforts of our students in SOC Design Laboratory at Ewha W. University on simulations. The authors are particularly grateful for Ju Hyoung Mun, A.G. Alagu Priya, and Geum Dan Jin.

REFERENCES

[1] H.J. Chao, "Next Generation Routers," *Proc. IEEE*, vol. 90, no. 9, pp. 1518-1558, Sept. 2002.

[2] M.A. Ruiz-Sanchex, E.W. Biersack, and W. Dabbous, "Survey and Taxonomy of IP Address Lookup Algorithms," *IEEE Network*, vol. 15, no. 2, pp. 8-23, Mar./Apr. 2001.

[3] G. Varghese, *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices.* Morgan Kaufmann Publishers/Elsevier, Inc., 2005.
[4] V.C. Ravikumar, R.N. Mahapatra, and L.N. Bhuyan, "EaseCAM: An Energy and Storage Efficient TCAM-Based Router Architecture for IP Lookup," *IEEE Trans. Computers,* vol. 54, no. 5, pp. 521-533, May. 2005.
[5] D. Shah and P. Gupta, "Fast Updating Algorithms for TCAM," *IEEE Micro,* vol. 21, no. 1, pp. 36-47, Jan./Feb. 2001.
[6] D. Mehta and S. Sahni, *Handbook of Data Structures and Applications.* Chapman and HALL/CRC, 2005.
[7] H. Song, J. Turner, and J. Lockwood, "Shape Shifting Tries for Faster IP Route Lookup," *Proc. IEEE Int'l Conf. Network Protocols (ICNP),* 2005.
[8] W. Eatherton, "Fast IP Lookup Using Tree Bitmap," master's thesis, Washington Univ., 1999.
[9] B. Lampson, V. Srinivasan, and G. Varghese, "IP Lookups Using Multiway and Multicolumn Search," *IEEE/ACM Trans. Networking,* vol. 7, no. 3, pp. 324-334, June 1999.
[10] X. Sun and Y. Zhao, "An On-Chip IP Address Lookup Algorithm," *IEEE Trans. Computers,* vol. 54, no. 7, pp. 873-885, July 2005.
[11] H. Lu and S. Sahni, "O(log*n*) Dynamic Router-Tables for Prefixes and Ranges," *IEEE Trans. Computers,* vol. 53, no. 10, pp. 1217-1230, Oct. 2004.
[12] H. Lim and J. Mun, "An Efficient IP Address Lookup Algorithm Using a Priority-Trie," *Proc. IEEE GLOBECOM,* pp. 1-5, 2006.
[13] R. Jain, "A Comparison of Hashing Schemes for Address Lookups in Computer Networks," *IEEE Trans. Comm.,* vol. 40, no. 10, pp. 1570-1573, Oct. 1992.
[14] A. Broder and M. Mitzenmacher, "Using Multiple Hash Functions to Improve IP Lookups," *Proc. IEEE INFOCOM,* pp. 1454-1463, 2001.
[15] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High Speed IP Routing Lookups," *Proc. ACM SIGCOMM,* pp. 25-35, 1997.
[16] H. Lim and Y. Jung, "A Parallel Multiple Hashing Architecture for IP Address Lookup," *Proc. IEEE Workshop High Performance Switching and Routing,* pp. 91-98, 2004.
[17] H. Lim, J.-H. Seo, and Y.-J. Jung, "High Speed IP Address Lookup Architecture Using Hashing," *IEEE Comm. Letters,* vol. 7, no. 10, pp. 502-504, Oct. 2003.
[18] S. Dharmapurikar, P. Krishnamurthy, and D.E. Taylor, "Longest Prefix Matching Using Bloom Filters," *IEEE/ACM Trans. Networking,* vol. 14, no. 2, pp. 397-409, Apr. 2006.
[19] K. Lim, K. Park, and H. Lim, "Binary Search on Levels Using a Bloom Filter for IPv6 Address Lookup," *Proc. ACM/IEEE Symp. Architectures for Networking and Comm. Systems (ANCS),* pp. 185-186, 2009.
[20] S. Sahni and K.S. Kim, "Efficient Construction of Multibit Tries for IP Address Lookup," *IEEE/ACM Trans. Networking,* vol. 11, no. 4, pp. 650-662, Aug. 2003.
[21] S. Nilsson and G. Karlsson, "IP Address Lookup Using LC-Tries," *IEEE J. Selected Areas in Comm.,* vol. 17, no. 6, pp. 1083-1092, June 1999.
[22] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small Forwarding Tables for Fast Routing Lookups," *Proc. ACM SIGCOMM,* pp. 3-14, 1997.
[23] N. Yazdani and P.S. Min, "Fast and Scalable Schemes for the IP Address Lookup Problem," *Proc. IEEE Workshop High Performance Switching and Routing,* pp. 83-92, 2000.
[24] C. Yim, B. Lee, and H. Lim, "Efficient Binary Search for IP Address Lookup," *IEEE Comm. Letters,* vol. 9, no. 7, pp. 652-654, July 2005.
[25] H. Lim, B. Lee, and W.-J. Kim, "Binary Searches on Multiple Small Trees for IP Address Lookup," *IEEE Comm. Letters,* vol. 9, no. 1, pp. 75-77, Jan. 2005.
[26] H. Lim, W. Kim, and B. Lee, "Binary Search in a Balanced Tree for IP Address Lookup," *Proc. IEEE Workshop High Performance Switching and Routing,* pp. 490-494, 2005.
[27] H. Lim, H. Kim, and C. Yim, "IP Address Lookup for Internet Routers Using Balanced Binary Search with Prefix Vector," *IEEE Trans. Comm.,* vol. 57, no. 3, pp. 618-621, Mar. 2009.
[28] S. Sahni and K.S. Kim, "An O(log*n*) Dynamic Router-Table Design," *IEEE Trans. Computers,* vol. 53, no. 3, pp. 351-363, Mar. 2004.
[29] http://www.potaroo.net, 2010.

Hyesook Lim (M'91) received the BEng and MS degrees from the Department of Control and Instrumentation Engineering, Seoul National University, Korea, in 1986 and 1991, respectively, and the PhD degree from the University of Texas at Austin, in 1996. From 1996 to 2000, she had been employed as a member of technical staff at Bell Labs in Lucent Technologies, Murray Hill, New Jersey. From 2000 to 2002, she worked for Cisco Systems, San Jose, California. She is currently an associate professor in the Department of Electronics Engineering, Ewha Womans University, Seoul, Korea. Her research interests include router design issues such as address lookup and packet classification, and hardware implementation of various network algorithms. She is a member of the IEEE.

Changhoon Yim (M'91) received the BEng degree from the Department of Control and Instrumentation Engineering, Seoul National University, Korea, in 1986, the MS degree in electrical engineering from Korea Advanced Institute of Science and Technology, in 1988, and the PhD degree in electrical and computer engineering from the University of Texas at Austin, in 1996. He was a research engineer working on HDTV at Korean Broadcasting System, from 1988 to 1991. From 1996 to 1999, he was a member of technical staff in HDTV and Multimedia Division, Sarnoff Corporation, New Jersey. From 1999 to 2000, he worked at Bell Labs, Lucent Technologies, New Jersey. From 2000 to 2002, he was a software engineer in KLA-Tencor Corporation, California. From 2002 to 2003, he was a principal engineer at Samsung Electronics, Suwon, Korea. He is currently an associate professor in the Department of Internet and Multimedia Engineering, Konkuk University, Seoul, Korea. His research interests include multimedia communication, digital image processing, packet classification, and IP address lookup. He is a member of the IEEE.

Earl E. Swartzlander, Jr. (S'64-M'72-SM'79-F'88) received the degrees in electrical engineering from Purdue University, the University of Colorado, and the University of Southern California. He is a professor of electrical and computer engineering at the University of Texas at Austin. In his current position, he and his students conduct research in computer engineering with emphasis on application-specific processor design, including high-speed computer arithmetic, processor architecture, VLSI technology, and rapid prototyping. As of December 2009, he has supervised 33 PhD students. From 1975 to 1990, he held a variety of positions at TRW including the director of Independent Research and Development in the TRW Defense Systems Group, the manager of the Digital Processing Laboratory in the Electronics and Technology Division, and the manager of the Advanced Development Office in the System Development Division. He was the editor-in-chief of the *IEEE Transactions on Computers* from 1990 to 1994 and was the founding editor-in-chief of the *Journal of VLSI Signal Processing*. In addition, he has served as an associate editor for the *IEEE Transactions on Computers*, the *IEEE Transactions on Parallel and Distributed Systems*, and the *IEEE Journal of Solid-State Circuits*. He has been a member of the Board of Governors of the IEEE Computer Society (1987-1991), the IEEE Signal Processing Society (1992-1994), and the IEEE Solid-State Circuits Council/Society (1986-1991). He has been a member of the IEEE History Committee (1996-2004), the IEEE Fellows Committee (2000-2003), and currently is the chair of the IEEE James H. Mulligan, Jr., Education Medal Committee. He has chaired a number of conferences. He is the author of one book, editor of seven books and the author or coauthor of 62 refereed journal papers, 33 book chapters, and 257 conference papers. He is a fellow of the IEEE. He has been honored with the IEEE Third Millennium Medal, the Distinguished Engineering Alumnus Award from the University of Colorado, the Outstanding Electrical Engineer and Distinguished Engineering Alumnus Awards from Purdue University, and the IEEE Computer Society Golden Core Award.