Computer Networks 56 (2012) 231–243

Contents lists available at SciVerse ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

An efficient IP address lookup algorithm based on a small balanced tree using entry reduction

Hyuntae Park, Hyejeong Hong, Sungho Kang *

Department of Electrical and Electronic Engineering, Yonsei University, Seoul, Republic of Korea

ARTICLE INFO

Article history:
Received 23 July 2010
Received in revised form 29 June 2011
Accepted 5 September 2011
Available online 13 September 2011

Keywords:
IP address lookup
Balanced binary search
Multi-way search
Entry reduction

ABSTRACT

Due to a tremendous increase in internet traffic, backbone routers must have the capability to forward massive incoming packets at several gigabits per second. IP address lookup is one of the most challenging tasks for high-speed packet forwarding. Some high-end routers have been implemented with hardware parallelism using ternary content addressable memory (TCAM). However, TCAM is much more expensive in terms of circuit complexity as well as power consumption. Therefore, efficient algorithmic solutions are essentially required to be implemented using network processors as low cost solutions.

Among the state-of-the-art algorithms for IP address lookup, a binary search based on a balanced tree is effective in providing a low-cost solution. In order to construct a balanced search tree, the prefixes with the nesting relationship should be converted into completely disjointed prefixes. A leaf-pushing technique is very useful to eliminate the nesting relationship among prefixes [V. Srinivasan, G. Varghese, Fast address lookups using controlled prefix expansion, ACM Transactions on Computer Systems 17 (1) (1999) 1–40]. However, it creates duplicate prefixes, thus expanding the search tree.

This paper proposes an efficient IP address lookup algorithm based on a small balanced tree using entry reduction. The leaf-pushing technique is used for creating the completely disjointed entries. In the leaf-pushed prefixes, there are numerous pairs of adjacent prefixes with similarities in prefix strings and output ports. The number of entries can be significantly reduced by the use of a new entry reduction method which merges pairs with these similar prefixes. After sorting the reduced disjointed entries, a small balanced tree is constructed with a very small node size. Based on this small balanced tree, a native binary search can be effectively used in address lookup issue. In addition, we propose a new multi-way search algorithm to improve a binary search for IPv4 address lookup. As a result, the proposed algorithms offer excellent lookup performance along with reduced memory requirements. Besides, these provide good scalability for large amounts of routing data and for the address migration toward IPv6. Using both various IPv4 and IPv6 routing data, the performance evaluation results demonstrate that the proposed algorithms have better performance in terms of lookup speed, memory requirement and scalability for the growth of entries and IPv6, as compared with other algorithms based on a binary search.

1. Introduction

The deployment of optical fibers as a transmission media provides a broad bandwidth for internet systems. The optical transmission technologies, such as the dense wave division multiplexing (DWDM), can accommodate several

* Corresponding author. Tel.: +82 2 2123 2775; fax: +82 2 313 8053.
E-mail address: shkang@yonsei.ac.kr (S. Kang).

1389-1286/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2011.09.004

hundred channels, transmitting data at several gigabits per second such as OC-192(10 Gbps) and OC-768(40 Gbps). Backbone routers must have the capability to forward massive incoming packets at this high link speed, even as the packet arrival rates and forwarding table sizes are dramatically increasing. If the processing speed for packet forwarding is less than the link speed, the packet forwarding operations become a major bottleneck which obstructs the fast transmission of the data [1]; the IP address lookup is one of the most challenging tasks among the packet forwarding operations.

In order to aggregate IP addresses in the same network, an IP address consists of two parts: a network part and a host part. The network part, called the *prefix*, identifies the network to which hosts are attached. The IP address lookup determines the output port of the incoming packets by looking up the prefix of the destination IP address on the packets in the forwarding table. In the classful addressing scheme, only three different sizes of networks were allowed; their prefix lengths fixed at 8, 16 and 24 bits in IPv4. Accordingly, IP address lookup was very easy, using a standard exact search method such as a native binary search. However, because only a small fraction of the allocated addresses were actually in use, the address space was used inefficiently and rapidly exhausted. In order to avoid wasting address space, classless inter-domain routing (CIDR) was deployed. CIDR allows for arbitrary aggregation of IP addresses so that the prefix has variable lengths from 8 to 32 bits in IPv4 and from 12 to 64, including 128, in IPv6. Therefore, prefixes have a nesting relationship dependent on the hierarchy of the networks. When a prefix is a sub-string of another prefix, the shorter (ancestor) prefix encloses the longer (descendent) prefix. As a result, since there are multiple matching results with different lengths, IP address lookup can no longer be performed by the standard exact search methods thereby complicating this task which must now find the longest matching prefix (LMP) among multiple matching prefixes [2]. Furthermore, as the long term solution for insufficient address resource, the migration of the address scheme from IPv4 to IPv6 is in progress. It also makes the address lookup becomes complicated because the address lengths of IPv6 is expanded to 128 bits. Even existing IP address schemes, which have excellent performance for IPv4, can hardly be competent for IPv6.

IP address lookup in high-performance routers have been implemented with hardware parallelism using specialized memories called ternary content addressable memory (TCAM). With TCAM, an address lookup can be performed with a single memory access for high-speed packet forwarding [3]. Despite this advantage, the use of TCAMs is prohibitive by the following three disadvantages: (1) low TCAM utilization and updating issues due to sorting entries, (2) high power consumption due to the parallel execution of TCAM entries, and (3) high TCAM manufacturing cost due to low chip density.

In respect of (1) updating issue, because the priority encoder logic selects the entry at a lowest physical memory address for a longest prefix matching issue, entries should be stored in decreasing order of prefix lengths in TCAMs. Under internet instability, this need to keep a sorted list of entries in TCAMs makes updates slow so that this updating issue becomes the main bottleneck of using TCAMs. Besides, since the arrangement of the empty space is crucial to speeding up TCAM updating, the TCAM utilization is degraded. In respect of (2) power consumption issue, the high density TCAMs consume power up to 12–15 W per chip when the entire memory is enabled. In order to support increasing entries for IP address lookup, vendors use four to eight such TCAM chips. The power consumption by a large number of chips causes the increase of the cooling cost and the limitation of the router design to fewer ports [3]. In addition, the growth of the entries in TCAM increases linearly with the power consumption. In respect of (3) TCAM manufacturing cost issue, one bit in a TCAM requires 10–12 transistors, while that in an SRAM requires 4–6 transistors. Thus, TCAMs is less dense than SRAMs, and the TCAM manufacturing cost is very high. As a result, the current solution for IP address lookup using TCAM is still confronted by the cost problems, and has a scalability issue toward IPv6 in the future network. According to Varghese [4], CAM technology is rapidly improving and is supplanting algorithmic methods in smaller routers. However, for large core routers that may have routing databases of a million routes in the future, it may be better to have solutions that scale with ordinary memory technologies such as SRAM and DRAM. In addition, the customers are asking for performing the newly various functions. In order to support various packet functions, the packet engines for IP address lookup are implemented using network processors in the recent routers. Thus, software-based address lookup algorithms are well worth developing.

Several metrics are considered in evaluating the performance of the software-based IP address lookup algorithm. First, the lookup speed is the most crucial metric. The cost of computation in the lookup process is dictated by the number of memory accesses [4]. Therefore, the lookup speed can be evaluated by the average and the worst-case numbers of memory accesses. In the case of a tree based lookup algorithm, the worst-case number of memory accesses depends on the maximum depth of a tree. Second, the memory requirement is an important metric. Currently, a forwarding table should accommodate several hundred thousand entries. Thus, the node size for each entry and the total number of entries should be kept to a minimum, and the pre-computation results should not be stored in each node. Third, an efficient updating mechanism is also important since route changes can occur up to 100 times per second in the edge routers [5]. A complicated updating operation may interfere with the lookup operation and degrade the search performance. Thus, incremental updating should be supported. Finally, scalability is another important metric. Algorithms should easily accommodate a rapid growth of entries in the forwarding table, and have good scalability toward IPv6.

Many software-based IP address lookup algorithms have been developed in recent years, in consideration of the above metrics. A binary trie provides an easy way to handle prefixes with arbitrary lengths [1]. However, the binary trie includes many empty internal nodes, resulting in a decrease in the search speed and wasted memory space. In the sense that binary search algorithms based

on prefix values involve no internal empty nodes, these algorithms have attracted the attention of researchers. However, due to the nesting relationship among prefixes, the ancestor (shorter) prefix must be searched for prior to the search for the descendent (longer) prefix. Hence, a binary search tree cannot be constructed merely by sorting prefixes and a native binary search cannot be directly applied. Furthermore, even if a match occurs in the middle of a tree, a search should continue to a leaf because another longer prefix may exist. These limitations make the binary search tree unbalanced, and hence the lookup speed is decreased. In order to overcome these limitations, the leaf-pushing technique [6] provides an elimination method for the nesting relationship among the prefixes. Using this technique in a binary trie, all entries become the completely disjointed prefixes. A balanced tree for a native binary search can be constructed simply by sorting these disjointed entries. However, the leaf-pushing creates an abundance of duplicated prefixes. The growth of entries in the forwarding table leads to longer lookup time as well as larger memory requirements. Therefore, this is a critical issue with the algorithms using leaf-pushing.

This paper proposes an efficient IP address lookup algorithm based on a small balanced tree using entry reduction. The leaf-pushing is used for constructing the balanced tree. Then, the search tree is minimized by the use of a new entry reduction method. In the leaf-pushed prefixes, there are a large number of pairs of adjacent prefixes which are similar in prefix strings and output ports. The entry reduction method merges these prefix pairs into a single prefix. Accordingly, the number of entries is significantly reduced and a small balanced tree can be constructed. Thus, a native binary search can be effectively used in both IPv4 and IPv6 address lookup. As well, a new multi-way search algorithm is proposed to improve a binary search for IPv4 address lookup. As a result, the proposed algorithms offer excellent lookup performance along with reduced memory requirements. Besides, the proposed algorithms provide good scalability for large amounts of routing data and for the address migration toward IPv6.

The rest of this paper is organized as follows. Section 2 describes previous IP address lookup algorithms using a binary search. Section 3 presents the proposed IP address lookup algorithms. Section 4 discusses the performance evaluation results using various IPv4 and IPv6 routing data, as compared to other algorithms, and Section 5 provides the conclusions.

2. Previous works

2.1. Binary search algorithms based on a trie

A binary trie provides a natural way in which to find the longest matching prefix [1]. This is a tree-based structure which executes a linear search based on the prefix length. Each prefix is associated with a node defined by the path from the root. The search proceeds to the left or the right by a sequential bit-by-bit inspection, starting with the most significant bit. Fig. 1 shows the binary trie for the example set of prefixes presented in Table 1. The gray nodes represent the prefixes, and P_x^o denotes that the output port of the prefix P_x is o. Otherwise, the white nodes represent the empty internal nodes. The binary trie structure is simple, as well as easy to implement and update. It provides good scalability for the growth of entries in the forwarding table. However, since many empty internal nodes are required, as shown in Fig. 1, much memory space is wasted. Besides, the depth of the trie is the maximum prefix length which results in a slow lookup speed.

In order to avoid the limitation by empty internal nodes in a trie, the multi-bit trie and the path compression methods have been suggested. In the multi-bit trie structure [7], more than one bit at a time is inspected. In the level-compressed trie (LC-trie) [8], the multi-bit trie is applied along with path compression. But, using a node array to store the LC-trie makes incremental updates very difficult. Some approaches have been proposed a data structure that can represent large forwarding tables in a compact form by compressing the multi-bit trie to fit into processor's cache. Lulea algorithm [9] has been suggested using the bitmap compression in the leaf-pushed multi-bit trie. This can reduce the number of elements in a trie and save memory requirements. However, it requires two memory accesses per node during the search procedure, and it is almost impossible to perform incremental updates because of its tight coupling property. In most cases, the whole table may need to be completely rebuilt for updating. In addition, its dedicated memory organization is also un-scalable for large forwarding tables and IPv6. Tree bitmap algorithm [10] has been proposed based on multi-bit expanded tries without any leaf pushing and the bitmaps to compress wasted storage in trie nodes. This requires single memory access per node during the search, and has faster update time than the Lulea algorithm. However, since there are two bitmaps per node to avoid the leaf pushing,

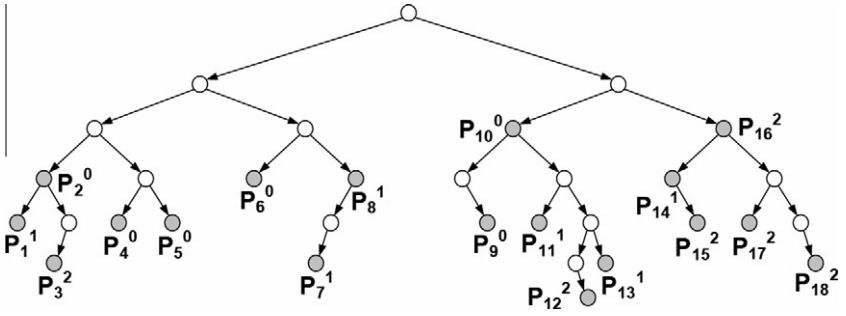


Fig. 1. Binary trie.

Table 1
Example set of prefixes.

Prefix		Output-port	Prefix		Output-port
P_1	0000	1	P_{10}	10	0
P_2	000	0	P_{11}	1010	1
P_3	00010	2	P_{12}	101101	2
P_4	0010	0	P_{13}	10111	1
P_5	0011	0	P_{14}	110	1
P_6	010	0	P_{15}	1101	2
P_7	01100	1	P_{16}	11	2
P_8	011	1	P_{17}	1110	2
P_9	1001	0	P_{18}	11111	2

it makes update inherently slow as in the Lulea algorithm. In some cases, the updating makes an excessive number of memory accesses. It is caused by the fact that the children of a node and the output-ports array of the forwarding information are allocated and stored in contiguous memory blocks. It requires more than 3000 memory accesses each for insertion or deletion operations in the worst case [11].

As solutions for eliminating the empty internal nodes in a trie, the longest prefix first search has been presented. In a priority trie (P-Trie) [12], the empty internal nodes of a trie are replaced by the longest prefix among the descendent prefixes whose are belonged to a sub-tree rooted by the empty node. Similarly, in a longest prefix first search tree (LPFST) [13], when the first sub-string bits of a prefix are equal to the value of a node associated with its position in a trie, the prefix is allocated to the corresponding node. Accordingly, the longer prefix is always located on the upper node. If the entire string of a prefix has the same value as a node associated with a position in the trie, this prefix overlaps with the previously allocated prefix. Both methods are able to eliminate the empty internal nodes in a trie and search for a longer prefix in decreasing order of prefix lengths. However, in order to continually allocate the longer prefix on the upper node in a trie structure, recursive exchanges and movements of nodes are required for the building and updating procedure.

2.2. Binary search algorithms based on prefix values

Considering that a binary search algorithm based on prefix values includes no empty internal nodes, it is a better approach than the binary trie. But, the sorting method of prefixes with various lengths is required. In a binary prefix tree (BPT) [14], the binary search scheme using a comparison method for sorting prefixes with different lengths has been suggested. For two prefixes of different lengths, the magnitude is decided by comparing the first m bits of the two prefixes, where m is the length of the shorter prefix. If the first m bits of the two prefixes are the same, then the $(m+1)$th bit of the longer prefix is observed. If this bit is 1, the longer prefix is larger, otherwise the shorter prefix is larger. However, due to the nesting relationship among prefixes, the BPT cannot be constructed by simply sorting the entries. Among prefixes with the nesting relationship, the ancestor (shorter) prefix should be allocated on the upper node than the descendent (longer) prefix in a tree. This can result in an unbalanced tree with a deep depth. In order to construct more balanced BPT, the weighted prefix tree (WPT) [15] considers the number of the descendent prefixes while selecting the root of each level. However, the WPT is also limited by the necessity of searching for ancestors prior to descendents. In order to overcome this limitation, it is necessary to completely eliminate the nesting relationship among the prefixes.

If all of the entries are completely disjointed by eliminating the nesting relationship, a binary search tree for IP address lookup can be perfectly balanced, thus allowing the application of a native binary search. The depth of a perfectly balanced tree can be bound to $O(logN)$, where N is the number of entries. Thus, an optimum lookup speed can be achieved, dependent only on the number of entries. Besides, the left and the right pointers on each node for a binary search are not required, thus saving memory space.

Several algorithms have been presented in the sense that a binary search tree composed of completely disjointed entries is perfectly balanced. In IP packet forwarding based on partitioned lookup table (IFPLUT) [16], a partitioning technique of multiple tables has been suggested for creating a set of disjointed prefixes. Using the fact that prefixes with the same output port are mutually disjointed, all entries are partitioned according to their output port. However, since this scheme is parallel processed in multiple tables, additional hardware is required so that it is unsuitable for software implementation. For the multi-way binary prefix tree (MBPT) [17], a forwarding table is divided into multiple trees, each consisting of a subset of each ancestor prefix based on different prefix levels. Multiple trees have a hierarchical structure and each tree becomes balanced. However, because a hierarchical binary search is performed in multiple stages, the search path on multiple trees is longer than that on a single tree. In a binary search scheme based on search space reduction (SSR) [18], in order to reduce the search path, a forwarding table is partitioned into two-level tables according to the nesting relationship. Although the search path of this algorithm is smaller than that of the MBPT, the search path remains long. In the algorithm using prefix vectors (PV) [19], the binary search tree is constructed with the prefixes only on the leaves of the binary trie. Since the prefixes on leaves are disjointed, the tree is perfectly balanced. However, each entry contains the forwarding information of the corresponding ancestor prefixes as prefix vectors. Thus, each node requires much memory space for storing the prefix vectors. Especially, the node size in IPv6 is impractical so that scalability to IPv6 is very poor.

In other methods which eliminate the nesting relationship, the prefixes are represented as ranges [20–22]. The IP address lookup issue for the prefixes is transformed into a binary search for range. The range of each prefix consists of start points and end points which are padded with zeros and ones to a maximum length. Ranges are divided by disjointed intervals, and the best matching prefix (BMP) for each interval is pre-computed and stored. As a result, the native binary search can be applied to disjointed interval entries. However, since the start points and end points of a range are stored in the forwarding table, the number of entries might be twice the number of the actual prefixes in the worst case. Besides, an incremental update is impossible due to the pre-computation of the BMP for each entry.

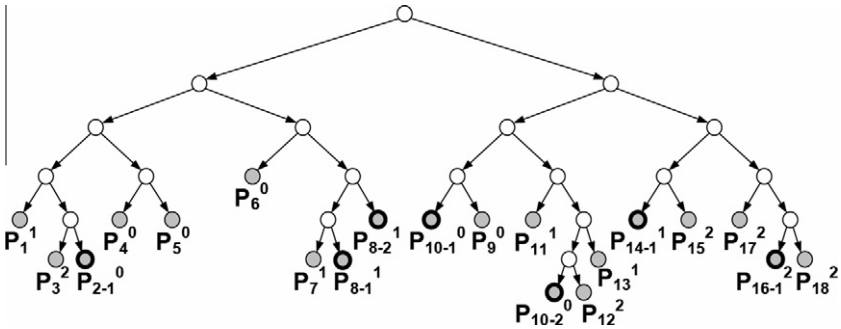


Fig. 2. Leaf-pushing binary trie.

The leaf-pushing technique has been presented as another method for eliminating the nesting relationship [6]. This technique removes ancestor prefixes in the internal nodes and generates new prefixes by pushing the removed prefix down to the leaves in a binary trie. For an example set of prefixes in Table 1, the ancestor prefixes are P_2, P_8, P_{10}, P_{14} and P_{16}. These prefixes are pushed down to their leaves in the trie with their output ports. Fig. 2 shows the leaf-pushing binary trie for the binary trie of Fig. 1. The gray nodes represent the original prefixes, and the bolded nodes represent the pushed prefixes. As shown, all of the entries are located at the leaves in a trie and thus become disjointed. In a disjointed prefix tree (DPT) [23], a balanced binary search tree is constructed by sorting these disjointed entries using the comparison method in the BPT. The search path of the DPT is shorter than that of the algorithms based on multiple trees such as the MBPT and the SSR. Besides, pre-computation is not required, contrary to the binary search algorithms for ranges. Nevertheless, a critical problem arises due to the significant number of increased entries resulting from using leaf-pushing. As shown in Fig. 2, multiple pushed prefixes are produced from a single ancestor prefix such as P_8 and P_{10}. In [24], in an attempt to reduce the number of duplicated prefixes which were increased by the use of leaf-pushing, adjacent prefixes with the same output port are merged. However, since this merge operation is restricted according to the output ports of the prefixes, the large number of entries cannot be reduced.

3. Proposed algorithm

The leaf-pushing technique is initially used to create the completely disjointed. As mentioned earlier, this technique increases a number of entries. We have investigated the leaf-pushed prefixes and identified that there are a large number of pairs of adjacent prefixes which are similar in prefix strings and output ports; therefore, these similar prefix pairs are merged into the single common prefix to reduce the number of entries. By means of sorting these reduced entries, a small balanced binary search tree can be efficiently constructed without any complicated building process. In addition, in order to improve a binary search for IPv4 address lookup, a new multi-way search algorithm is proposed. Because these search trees are perfectly balanced, the lookup operation of the proposed algorithm is quickly accomplished. Furthermore, the proposed algorithm can provide incremental updating and the scalability for the growth of entries and IPv6.

3.1. Entry reduction method

We first define the notation for the prefix as follows. Let $p = \{P_1, P_2, \ldots, P_N\}$ be the set of prefixes, where N is the number of prefixes. Let the bit string of a prefix be $P_x = b_1, \ldots, b_l$ where l is the length and b_k is either 0 or 1. Let $b_{x,k}$ be the kth bit of prefix P_x. Let $L(P_x)$ and $O(P_x)$ be the length and the output ports of the prefix P_x, respectively.

Definition 1 (*Prefix type*). According to the number for the forwarding information of a prefix, if the prefix has a single output port, the prefix type is defined as *ordinary*. Otherwise, if the prefix has dual output ports, the prefix type is defined as *merger*.

Definition 2 (*Twin prefix*). For two *ordinary* prefixes P_A and P_B, P_A is defined as the *twin prefix* of P_B and vice versa, if, and only if, the length of the two prefixes is the same, and the last significant 1 bit of two prefixes is only the different and the rest bits of them are the same. $P_A \coloneqq P_B$ represents the twin prefixes P_A and P_B, therefore,

$$P_A \coloneqq P_B \quad \text{iff} \quad L(P_A) = L(P_B) = l \quad \text{and} \quad b_{A,1} = b_{B,1}, \ldots, b_{A,l-1} = b_{B,l-1}, \quad b_{A,l} \neq b_{B,l}.$$

Definition 3 (*Uniform prefix*). For two *ordinary* prefixes P_A and P_B, P_A is defined as the *uniform prefix* of P_B and vice versa, if, and only if, the two prefixes are mutual twin prefixes and the output ports of the two prefixes are the same. $P_A \equiv P_B$ represents the uniform prefixes P_A and P_B, therefore,

$$P_A \equiv P_B \quad \text{iff} \quad P_A \coloneqq P_B \quad \text{and} \quad O(P_A) = O(P_B).$$

Definition 4 (*Parent of twin prefixes*). For the *twin prefixes* P_A and P_B, P_C is defined as the *parent* of the twin prefixes P_A and P_B, if, and only if, the length of prefix P_C is smaller than that of the twin prefixes by one bit and the bit string of P_C is equal to the sub-string of the twin prefixes; P_C is the parent of the twin prefixes P_A and P_B iff $L(P_A) = L(P_B) = l$, $L(P_C) = l - 1$ and $b_{A,1} = b_{B,1} = b_{C,1}, \ldots, b_{A,l-1} = b_{B,l-1} = b_{C,l-1}$, $b_{A,l} \neq b_{B,l}$.

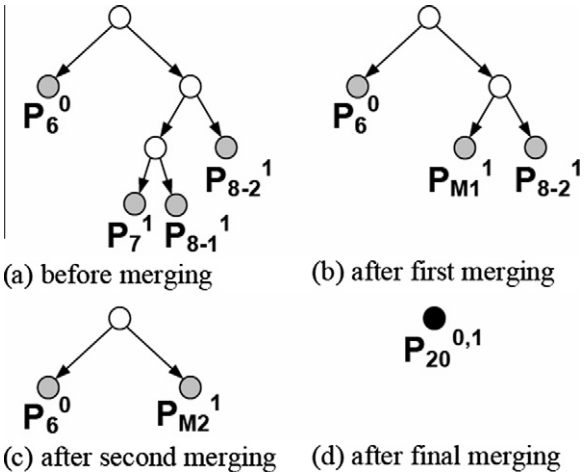


Fig. 3. Example of performing the proposed entry reduction method.

The proposed entry reduction method performs merging the twin prefixes into their parent in the leaf-pushing binary trie as follows. For the twin prefixes P_A and P_B, let the last significant 1 bit of P_A be 0 and that of P_B be 1; $b_{A,l} = 0$ and $b_{B,l} = 1$, when $l = L(P_A) = L(P_B)$. The string of the merged prefix P_M, which is the parent, is the common bit string of the twin prefixes P_A and P_B; $P_M = b_1 b_2, \ldots, b_{l-1}$. The length of the merged prefix is less than that of the twin prefixes by one bit; $L(P_M) = l - 1$. The merged prefix type is determined according to the output ports of the twin prefixes. If the two prefixes are uniform prefixes, according to Definition 3, then the forwarding information of P_M is the output port of the twin prefixes and the merged prefix is an ordinary prefix type. Otherwise, if the two prefixes are not uniform prefixes, the forwarding information of P_M are both output ports of the twin prefixes and the merged prefix is a merger prefix type. This merge operation is repeatedly performed until there are no remaining twin prefixes in the leaf-pushing trie.

Fig. 3 illustrates the reduction of entries using the proposed entry reduction method, where P_x^o denotes the output port o of the prefix P_x. In Fig. 3(a) as a subset of Fig. 2, P_7^1 and P_{8-1}^1 are uniform prefixes. These prefixes are merged into P_{M1}^1 as shown in Fig. 3(b) and this merged prefix is an ordinary prefix type. Then, P_{M1}^1 and P_{8-2}^1 are also uniform prefixes and are merged into P_{M2}^1 in Fig. 3(c). Finally, P_6^0 and P_{M2}^1 are twin prefixes, not uniform prefixes, therefore these prefixes are merged into $P_{20}^{0,1}$ as a merger prefix type. Fig. 4 shows the reduced leaf-pushing trie with a small number of entries using the proposed entry reduction method from the leaf-pushing trie of Fig. 2. As shown, the number of entries can be reduced by more than just the number of twin and uniform prefixes. In this example, we confirm that the number of entries in Fig. 4 can be reduced by half, as compared with the original leaf-pushing trie shown in Fig. 2.

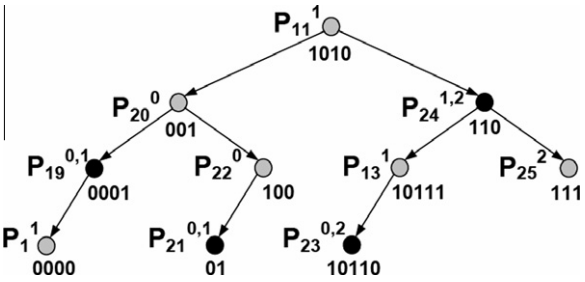


Fig. 5. Proposed balanced binary search tree.

3.2. Building

Building the binary search tree in the proposed algorithm comprises the following three steps: (1) in a binary trie, the leaf-pushing technique is used to transform the prefixes with the nesting relationship into completely disjointed prefixes. (2) The proposed entry reduction method by merging is repeatedly performed until there are no other twin prefixes in the leaf-pushing trie. (3) After the merge operation, the entries are sorted in ascending order using the comparison method in the BPT. This sorted list becomes both the small balanced tree and the forwarding table for a native binary search.

Fig. 5 shows the proposed balanced binary search tree created by sorting entries from the reduced leaf-pushing trie of Fig. 4. The black nodes represent the merger type prefixes, and the gray nodes represent the ordinary type prefixes. This tree is perfectly balanced and has an optimum depth. Fig. 6 illustrates the entry structure for the proposed binary search tree. The node is composed of the prefix type, the prefix string, the prefix length, and the two output-port fields. The value of the type field represents whether the prefix type is ordinary or merger. Every node is either a type 0 or a type 1. For ordinary type pre-

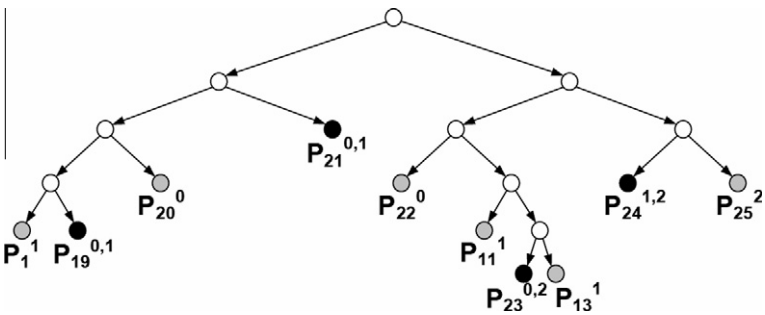


Fig. 4. Reduced leaf-pushing trie using the proposed entry reduction method.

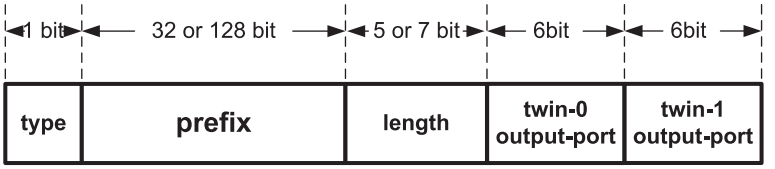


Fig. 6. Entry structure for the proposed binary search tree.

fixes, the type 0 node includes only one output port in the twin-0 output-port field. Otherwise, for merger type prefixes, the type 1 node includes two output ports of twin prefixes in the twin-0 and the twin-1 output-port fields. Prefix strings and their associated lengths are stored in the second and third fields, respectively. The width of the prefix field is 32 bits for IPv4 or 128 bits for IPv6. Accordingly, the width of the length field is 5 bits in IPv4 or 7 bits in IPv6. Because the proposed binary search tree is perfectly balanced, the left and right pointers are not required for a binary search. This yields a very simple entry structure which results in reduced memory requirements for the small node size. For the proposed binary search tree in Fig. 5, its associated forwarding table is shown in Table 2. This table can be constructed by the sorted entries itself without any building operation.

3.3. Searching

Searching in the proposed binary search algorithm is performed by a native binary search. Let the input A be the destination IP address of an incoming packet and $S(A,k)$ be a sub-string of the most significant k bits of A. Let P_x be the prefix stored at node x. Accordingly, A is defined to match P_x if $S(A,L(P_x)) = P_x$. Let $T(P_x)$ be the prefix type for P_x, and O^* represent the output port of the matched prefix as the final search result. The pseudo code for the search procedure is shown in Fig. 7.

In code lines 2–3, the search result O^* and a node x are initialized. When there is no matching prefix in the forwarding table, the incoming packet is forwarded to a default output port. Hence, O^* is set to the default output port. Node x is set to the root. The root in the binary search tree is the entry at the index $N/2$, where N is the total number of entries in the forwarding table. In code lines 5–12, if the input A matches the prefix P_x at node x, the output port is determined according to the matched prefix type. If the prefix type is ordinary (0), the twin-0 output-port is assigned to O^*. On the other hand, if the prefix type is merger

```
1  Search (IP Address A)
2    O* ← default output-port;
3    x ← root;
4    do
5      if ( S( A , L(Px) ) = Px )
6        if ( T( Px ) = 0 )
7          O* ← O0( Px ); break;
8        else
9          if ( bA,L(Px)+1 = 0 )
10           O* ← O0( Px ); break;
11         else
12           O* ← O1( Px ); break;
13     else
14       if ( A < Px )
15         xN is the left child node of x
16       else
17         xN is the right child node of x
18       x ← xN;
19   while ( x is a valid node )
20   return O*;
```

Fig. 7. Pseudo code for the search procedure.

(1), the $(L(P_x) + 1)$th bit of input A is checked. If that bit is 0, the twin-0 output-port is assigned to O^*. Otherwise, the twin-1 output-port is assigned to O^*. Since all of the entries are disjointed, the search result is unique. Therefore, if there is the matched prefix, the search is immediately completed and the output port of the matched prefix is returned at code line 20. In code lines 14–18, if input A does not match the prefix, the binary search is continued. If the input is smaller than the prefix at the present node, the next node is the medium entry of the smaller half, which is the left child node. Otherwise, the next node is the medium entry of the bigger half, which is the right child node. This is continued until there are no more valid nodes.

Based on the forwarding table, shown in Table 2, of an example of the incoming address, 000110*, the input is

Table 2
Forwarding table of the proposed binary search tree.

Index	Type	Prefix	Length	Twin-0 output-port	Twin-1 output-port
0	0	0000	4	1	–
1	1	0001	4	0	1
2	0	001	3	0	–
3	1	01	2	0	1
4	0	100	3	0	–
5	0	1010	4	1	–
6	1	10110	5	0	2
7	0	10111	5	1	–
8	1	110	3	1	2
9	0	111	3	2	–

first compared with the prefix 1010 at index 5 in the root node. The input does not match so the magnitude of the input is compared with that of the prefix at the node. Since the input is smaller than the prefix 1010, the next comparison is performed at index 2. Again, it does not match and is smaller than the prefix 001. Next, the input is matched with the prefix 0001 at index 1. Because the matched prefix type is merger, the next bit in the input is checked. This bit is 1 which is the fifth bit of the input. Therefore, the search result is decided by 1 in the twin-1 output-port field at index 1 and the search is immediately completed. In another example of an incoming address of 10111*, the input is compared with the prefixes at indices 5, 8 and 7 sequentially. At index 7, since the input matches the prefix as an ordinary type, the search result is set to 1 in the twin-0 output-port field and the search is completed.

3.4. Updating

The proposed algorithm supports incremental updating. The prefix insertion operation depends on three factors: the matched prefix with the input, the output ports of the input and the matched prefix, and the matched prefix type. In order to find the appropriate position for a prefix insertion, a prefix matching the input prefix should be searched for using the proposed search process. If there are no matched prefixes in the proposed tree, the input prefix is inserted at a new leaf node. Otherwise, if there is a matched prefix, the output ports of the input and the matched prefix are inspected. When the output ports of both prefixes are the same, the matched prefix implies the input prefix; therefore, an input prefix insertion is not required. However, if the output ports of both prefixes are different, the node itself, as well as the child nodes of the matched prefix, may be affected by the insertion of the input prefix. For this case, according to the type of the matched prefix and the parent relationship with the matched prefix and the input prefix, there are three cases for which the tree is modified.

The first and the second cases are when the matched prefix type is ordinary. The first case is when the matched prefix is the parent prefix of the input prefix. The matched prefix type is transformed into a merger. Then, the output port of the input prefix is stored in the corresponding output port field and the existing output port of the matched prefix is stored into another output port field. For example, when the new input prefix 1000 with an output port 1 is inserted into the proposed tree of Fig. 5, the input prefix is matched with the ordinary prefix 100, P^0_{22}. Since the matched prefix P^0_{22} is the parent of the input, this is transformed into a merger prefix. In this case, the output port of the matched prefix is moved to the twin-1 output-port field and the output port of the input is stored into the twin-0 output-port field. Therefore, the new merged prefix 100 is $P^{1.0}_{22}$.

The second case is when the matched prefix is not the parent prefix of the input prefix. The matched prefix should be leaf-pushed and the new leaf-pushed prefixes are re-merged, if necessary. These prefixes are then reallocated in ascending order. For example, for the input prefix 00110 with output port 1, as shown in Fig. 5, the input is matched with the prefix 001. Because the matched prefix 001 is not the parent of the input, the prefix 001 is leaf-pushed. Hence, the new leaf-pushed prefixes are 0010 and 00111 with the original output port 0. Then, since the new prefix 00111 and the input 00110 are twin prefixes, these prefixes are re-merged into the prefix 0011 with the output port 1 in twin-0 and the output port 0 in twin-1. Therefore, the new entries are the ordinary prefix 0010, P^0_{26} and the merger prefix 0011, $P^{1.0}_{27}$. For reallocation, the prefix $P^{1.0}_{27}$ is located at node P^0_{20} and the prefix P^0_{26} is located at the new right leaf node $P^{0,1}_{19}$. Fig. 8 shows the updated tree for this insertion case. This prefix insertion affects only two entries.

The third case is when the matched prefix type is merger. Since the matched prefix implies twin prefixes, the prefixes of the matched prefix are first separated. One of the separated prefixes is the ancestor prefix of the input. This prefix is leaf-pushed and the other prefix becomes an ordinary prefix. As in the second case, if necessary, the new leaf-pushed prefixes may be re-merged and reallocated in ascending order. For example, for the input prefix 0110 with output port 2, the input is matched with the merger prefix 01, $P^{0,1}_{21}$. The matched prefix is separated into two ordinary prefixes 010 and 011. The prefix 011 is leaf-pushed to the prefix 0111. Then, the input prefix 0110 and the leaf-pushed prefix 0111 can be re-merged. Therefore, the new prefixes are the ordinary prefix 010, P^0_{26} and the merger prefix 011, $P^{2,1}_{27}$. For sorting in ascending order, the prefixes P^0_{22}, P^0_{26} and $P^{2,1}_{27}$ are reallocated in the tree. Fig. 9 shows the updated tree for this insertion case. This prefix insertion affects only three entries.

For prefix deletion, the search process is also used to find a deleted entry. This operation depends on the location and the type of the matched prefix. When the matched prefix type is merger, the corresponding entry, which is the one of the twin prefixes in a matched prefix, is deleted and the matched prefix type is changed to ordinary. Otherwise, when the matched prefix type is ordinary, the location of the matched prefix is checked. If there is a matched prefix on a leaf in the tree, the matched prefix is simply eliminated without any processing. However, if there is a matched prefix in the middle of the tree, after deleting the matched prefix, a reallocation of entries is required.

3.5. Scalability to IPv6

The building, the searching and the updating procedures of the proposed algorithm are not affected by longer

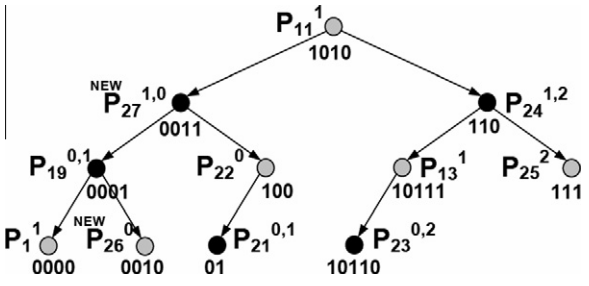


Fig. 8. Proposed tree updated by inserting the prefix 00110* with the output port 1.

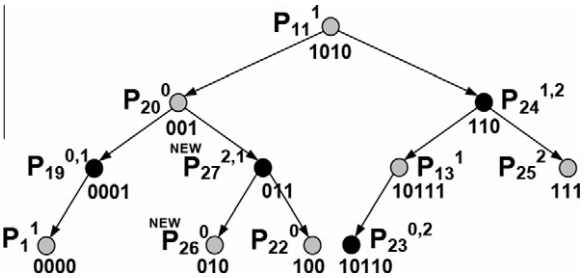


Fig. 9. Proposed tree updated by inserting the prefix 0110* with the output port 2.

prefix length as in IPv6. These operations in IPv6 address scheme are the same as those in IPv4 address scheme. Thus, the lookup and the update performance of the proposed algorithm are absolutely not degraded by increasing address lengths of IPv6. In IPv6, the modification of the proposed algorithm is uniquely that the widths of the prefix field and the length field are increased in the entry structure. This is unavoidable cost in all kinds of algorithms which should store prefix values in each node, such as binary search algorithms based on prefix values and the longest prefix first search algorithms in a trie. Therefore, the proposed algorithm has good scalability toward IPv6. Otherwise, the performance of a trie based algorithms depends on the prefix length so that these algorithms have poor scalability to IPv6. The detailed review is provided in [1].

3.6. 4-Way search algorithm

Modern general-purpose CPUs support various kinds of caches for speeding up the data processing. Recently, a single cache line in general-purpose computing devices is usually more than 30 bytes [19]. Because the node size of the proposed binary search algorithm in IPv4 is very small, getting more search information is possible at single memory access. Inspired by this fact, the multi-way search algorithm is proposed to improve the lookup speed of a binary search for IPv4. In case of IPv6, since the node size is originally 148 bits, the multi-way search cannot be applied.

Since the node size of the proposed binary search algorithm in IPv4 is 50 bits, the node of the proposed multi-way search algorithm can include three entries so that the 4-way search is possible. Fig. 10 shows the proposed balanced 4-way search tree from the reduced entries of Fig. 4. The entry structure of the 4-way search tree is shown in Fig. 11. For building the binary search tree, the forwarding table can be constructed by simply sorting the entries and has no pointers for a search path. On the other hand, for building the 4-way search tree, the quarter entry, the middle entry and the three-quarters entry among the sorted entries are grouped into one node of the 4-way search tree. In addition, the four pointers are stored to indicate the next child nodes.

The proposed 4-way search procedure is similar to the proposed binary search procedure. If there is the matched prefix among the three entries on each node, the search is immediately completed and the output port of the matched prefix is resulted. If the input does not match the prefixes, the magnitude comparison with three entries is performed and the corresponding pointer is selected to progress the next child node. This is continued until there are no more valid nodes or the matched prefix is found.

In order to keep the perfect balance of the tree, the updating of the proposed 4-way search tree may be more complicated than that of the proposed binary search tree. However, since the leaf nodes of the proposed 4-way search tree have empty slots, the inserted prefix can be allocated to empty slots of the leaf nodes. Thus, the incremental update is partially possible.

3.7. Complexity of the proposed algorithms

In both IPv4 and IPv6 networks, the complexity of the proposed algorithms is the follows. *N* denotes the number

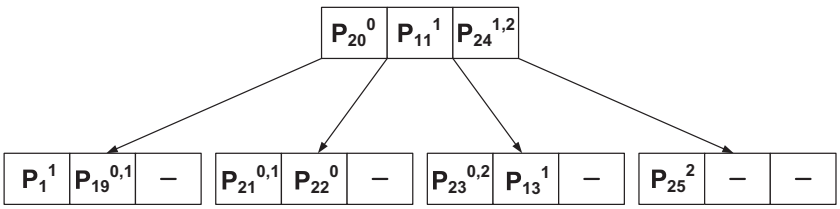


Fig. 10. Proposed balanced 4-way search tree.

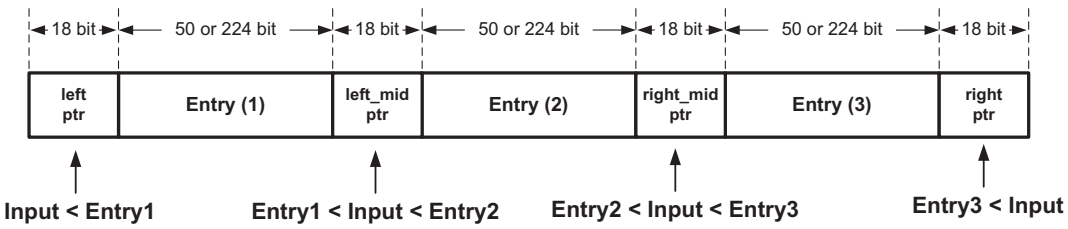


Fig. 11. Entry structure for the proposed 4-way search tree.

of entries of the proposed algorithms, which is reduced by the proposed entry reduction method. The search complexity of the proposed algorithm is $O(log_2 N)$ for a binary search and $O(log_k N)$ for a k-way search because the proposed trees are perfectly balanced. The update times of the proposed algorithms are $log_2 N + \alpha$ and $log_k N + \alpha$ for a binary and a k-way search, respectively. It takes $log_2 N$ and $log_k N$ for the search process for the appropriate positioning of an input prefix insertion or deletion. For a prefix insertion, α is the time required for leaf-pushing a matched prefix and the reallocation of the new prefixes. For a prefix deletion, α is the time required for the reallocation of entries in ascending order. Because time α is affected by a small number of entries, the update complexity is $O(log_2 N)$ for a binary search and $O(log_k N)$ for a k-way search. The memory requirement is determined by multiplying the node size by the number of entries. Since the small node size in the proposed algorithm is fixed, the complexity of memory requirements is $O(N)$.

4. Performance evaluation

The performance evaluation of the proposed algorithms and other algorithms is simulated by using C language programming, based on various IPv4 and IPv6 routing data. The six IPv4 and the two IPv6 routing data are obtained from actual backbone routers [25]. IPv6 real-world routing data is not sufficient for experimental evaluations. Since IPv6 is in its initiation period, only a small portion of address blocks are allocated up to now. Accordingly, the number of the prefixes in IPv6 environment is barely about six thousand, while the number of the prefixes in IPv4 environment is about hundreds of thousands. But, according to RFC 2928, 3177, 3578, and 3587, the organization and the allocation policies of IPv6 addresses have been settled. Thus, the future IPv6 prefix distribution is predictable. As referred to [26], we generate and use the three IPv6 random data, which have similar lengths and output ports distributions to the AS2.0 real routing data.

For both IPv4 and IPv6 routing data, the performance of the proposed algorithm and other algorithms are evaluated in the following terms: the number of original routing prefixes ($\boldsymbol{N}$), the number of routing prefixes after leaf-pushing ($\boldsymbol{N_{LP}}$), the number of pairs of twin and uniform prefixes in the leaf-pushing trie ($\boldsymbol{N_{TW}}$), the number of routing prefixes in the proposed algorithm ($\boldsymbol{N_P}$), the depth of the tree in the proposed algorithm ($\boldsymbol{D}$), the average number of memory accesses for an address lookup ($\boldsymbol{T_a}$), and the memory requirements ($\boldsymbol{M}$) for various routing data. For the proposed binary search algorithm, the reduction rate of entries ($\boldsymbol{R_E}$) show the efficiency of the proposed entry reduction method. On the other hand, for the proposed 4-way search algorithm in IPv4, the reduction rate of entries ($\boldsymbol{R_M}$) is presented by using the proposed entry reduction method and the proposed 4-way search. The reduction rates represent the percentage of the reduced routing prefixes, as compared with the number of routing prefixes after leaf-pushing ($\boldsymbol{N_{LP}}$).

Tables 3 and 4 show the performance evaluation results of the proposed algorithm, respectively. As we expect, the leaf-pushing causes a significant increase in the number of entries; the number of routing prefixes after leaf-pushing ($\boldsymbol{N_{LP}}$) is increased from 23% to 50% in IPv4 and by 3 to 5 times in IPv6. Hence, the algorithms using leaf-pushing requires much memory space resulting in degraded search performance. We have observed that there are a large number of twin prefixes in the leaf-pushing trie as shown by the number of pairs of twin prefixes ($\boldsymbol{N_{TW}}$). Accordingly, the merge operation in the proposed reduction method can be plentifully performed with the leaf-pushed prefixes. As a result, the total number of routing prefixes ($\boldsymbol{N_P}$) can be reduced to a small value. The efficiency of the proposed reduction method is also verified in terms of the reduction rate of entries ($\boldsymbol{R_E}$). The number of routing prefixes in the proposed algorithm is less than that in a leaf pushing trie, from 40% to 80% less, in both IPv4 and IPv6. As a result of entry reduction, the average number of memory accesses ($\boldsymbol{T_a}$) can be from 13 to 16 in IPv4 and from 11 to 16 in IPv6. Because the binary search tree in the proposed algorithms is perfectly balanced, the depth of the tree ($\boldsymbol{D}$) is optimal, and it is bounded by the $log_2 N_P$. Therefore, the maximum number of memory accesses is a small value as possible. The memory requirement ($\boldsymbol{M}$) is evaluated by multiplying the node size by the number of entries. The node size in the proposed binary search algorithm is 50 bits in IPv4 or 148 bits in IPv6, as shown in Fig. 6. Since the data path for memory access is the byte unit, the actual node size is 7 bytes in IPv4 or 19 bytes in IPv6. Due to this small node size and the small number of entries, the memory requirement of the proposed algorithms is extremely small. For the proposed 4-way search as shown in Table 3, although the memory requirement ($\boldsymbol{M}$) is increased, the average ($\boldsymbol{T_a}$) and the maximum ($\boldsymbol{D}$) number of memory accesses of the proposed 4-way search are decreased by half, as compared with the proposed binary search. On the other hand, the reduction rate for the large IPv4 routing data such as with PORT80, Telstra and AS2.0, is larger than that for the small routing data such as PacBell, MaeWest and Funet. In the IPv6 random data, the reduction rate of the routing data with a large amount is also better than that with a small amount. Therefore, the proposed algorithms provide good scalability for entry growth. In addition, as shown in Table 4, the lookup speed and the memory requirement of the proposed binary search algorithms do not depend on the increased address length in IPv6, and these performances are only affected by the reduction rate and the number of entries. Thus, the proposed binary search algorithm has good scalability to IPv6.

Table 5 shows performance comparisons with other algorithms for IPv4 routing data. The binary trie includes many empty internal nodes, therefore, it has the largest number of entries; accordingly, the lookup speed and memory requirements are the poorest. Overall, the lookup performance of the tree bitmap is better than the previous binary search based algorithms and the proposed binary search algorithm. But, the lookup speed of the proposed 4-way search algorithm is faster than that of the tree bitmap. In addition, the memory requirement of the proposed binary search algorithm is significantly smaller than that of the tree bitmap. The number of entries in the DPT is increased by using leaf-pushing. However, since the DPT is

Table 3
Performance evaluations of the proposed algorithm for IPv4 routing data.

Prefix set	N	N_{LP}	N_{TW}	Binary search					4-Way search				
				N_P	R_E (%)	T_a	D	$M(KB)$	N_P	R_M (%)	T_a	D	$M(KB)$
PacBell	20,519	25,337	4311	15,267	39.74	12.94	14	104.36	5461	78.45	6.66	7	149.32
MaeWest	29,584	42,908	9088	19,444	54.68	13.33	15	132.92	8522	80.14	6.87	8	233.02
Funet	40,904	58,993	15,383	19,410	67.10	13	15	132.69	8488	85.61	6.74	8	232.09
PORT80	112,310	145,267	40,905	38,596	73.43	14.3	16	263.84	21,845	84.96	7.44	8	597.32
Telstra	227,223	285,741	93,752	49,745	82.59	14.78	16	340.05	21,845	92.35	7.53	8	597.32
AS2.0	362,079	508,318	163,506	133,480	73.74	16.03	18	912.46	87,381	82.81	8.35	9	2389.32

Table 4
Performance evaluations of the proposed algorithm for IPv6 routing data.

Prefix set	N	N_{LP}	N_{TW}	Binary search				
				N_P	R_E (%)	T_a	D	$M(KB)$
AS2.0	5970	18,482	1768	5770	68.78	11.61	13	107.06
AS6447	5956	19,455	1753	11,353	41.64	12.59	14	210.65
rand_30k	30,000	74,399	3430	35,486	52.30	14.15	16	658.43
rand_50k	50,000	176,596	7781	68,979	60.94	15.09	17	1279.88
rand_100k	100,000	581,994	27,115	159,408	72.61	15.68	18	2957.77

able to perform a native binary search, the lookup speed is not the worst. Alternately, because there are no duplicated entries and no internal empty nodes in the BPT and the MBPT, the numbers of entries are equal to the original number. The numbers of entries with the LPFST and the SSR is slightly reduced by means of overlapping specific prefixes. However, since the trees of these algorithms are unbalanced, the lookup speed is inadequate. In particular, the worst-case lookup speeds of these algorithms depend on the degree of balance of their associated trees, according to the routing data. In some cases, the depth of the BPT, the MBPT and the SSR may be longer than that of a trie based algorithm whose depth is the maximum prefix length. On the contrary, in the PV and the proposed binary search algorithm, a native binary search is used on a balanced tree with small disjointed entries which results in good lookup speed. Moreover, by using proposed reduction method, the number of entries in the proposed algorithm is the smallest. As a result, the lookup speed in the proposed binary search algorithm is faster than that in other algorithms. The proposed 4-way search algorithm improves the lookup performance of the proposed binary search algorithm so that the lookup speed of the proposed 4-way search algorithm is the best.

In respect to the memory requirement, the PV has the largest node size and hence requires a large amount of memory. Alternately, since the algorithms based on a balanced tree such as with the MBPT, the SSR and the DPT do not require left and the right pointers for a binary search, their node size is small. Therefore, the memory requirement of these algorithms is generally smaller than that of the algorithms based on an unbalanced tree. In the proposed binary search algorithm, the search tree is not only balanced, but the node size and the number of entries are also the smallest, resulting in the extremely small memory requirement. In case of the proposed 4-way search algorithm, since the four pointers on each node are required, the memory requirement is slightly increased. But, the memory requirement of the proposed 4-way search algorithm is also smaller than that of other algorithms, except the bitmap tree.

Tables 6 and 7 shows performance comparisons for the real and the random IPv6 routing data. The binary trie has the poorest lookup performance because of increased empty internal nodes and the depth of the tree. Comparatively, the performance of the tree bitmap is excellent in IPv6. But, the depth of the tree bitmap is in proportion to the address length so that this is deep. Since the tree of

Table 5
Performance comparisons with other algorithms for IPv4 routing data.

Algorithm	PORT80 (112,310)				Telstra (227,223)				AS2.0 (362079)			
	N_P	T_a	D	M	N_P	T_a	D	M	N_P	T_a	D	M
Binary trie [1]	225,217	22.15	32	1319.63	452,905	24.47	32	2653.74	888,267	23.51	32	5204.69
Tree bitmap [10]	46,603	9.16	15	519.27	86,719	10.08	14	984.08	147,047	7.44	12	1646.00
LPFST [13]	83,736	18.93	25	981.28	183,025	21.8	33	2144.82	340,984	20.48	30	3995.91
BPT [14]	112,310	25.96	44	1316.13	227,223	30.94	66	2662.77	362,079	20.91	46	4243.11
MBPT [17]	112,310	26.29	47	987.10	227,223	29.56	58	1997.08	302,079	19.63	39	3182.33
SSR [18]	84,810	19.13	28	1022.23	184,706	22.9	37	2199.42	353,329	18.65	34	4439.01
PV [19]	70,418	15.33	17	1719.19	162,103	16.52	18	3957.59	327,507	17.45	19	7995.78
DPT [23]	145,267	16.19	18	851.17	285,741	17.17	19	1674.26	508,318	17.97	19	2978.43
Prop. binary	38,596	14.3	16	263.84	49,745	14.78	16	340.05	133,480	16.03	18	912.46
Prop. 4-way	21,845	7.44	8	597.32	21,845	7.53	8	597.32	87,381	8.35	9	2389.32

Table 6
Performance comparisons with other algorithms for real IPv6 routing data.

Algorithm	AS2.0 (5,970)				AS6447 (5,956)			
	N_P	T_a	D	M	N_P	T_a	D	M
Binary trie[1]	42243	39.78	128	247.52	43297	40.12	128	253.69
Tree bitmap[10]	8882	11.15	35	75.00	9158	11.24	36	77.00
LPFST[13]	5913	29.17	49	144.36	5901	29.18	49	144.07
BPT[14]	5970	14.22	28	145.75	5956	14.14	27	145.41
MBPT[17]	5970	13.09	27	128.26	5956	13.00	27	127.96
SSR[18]	5955	13.47	46	153.10	5942	13.28	45	152.79
PV[19]	5538	11.59	13	567.86	5510	11.58	13	564.99
DPT[23]	18482	13.23	15	324.88	19455	13.32	15	341.98
Prop. Binary	5770	11.61	13	107.06	11353	12.59	14	210.65

Table 7
Performance comparisons with other algorithms for random IPv6 routing data.

Algorithm	rand_30k (30,000)				rand_50k (50,000)				rand_100k (100,000)			
	N_P	T_a	D	M	N_P	T_a	D	M	N_P	T_a	D	M
Binary trie[1]	1035743	56.47	128	6068.81	1676919	56.77	128	9825.70	3064322	57.88	128	17955.01
Tree bitmap[10]	20147	15.47	36	206.00	28578	15.56	35	300.00	59256	15.91	36	618.00
LPFST[13]	29938	19.43	49	730.91	49929	19.22	49	1218.97	99896	19.61	49	2438.87
BPT[14]	30000	15.26	28	732.42	50000	15.94	29	1220.70	100000	16.98	32	2441.41
MBPT[17]	30000	14.29	30	644.53	50000	14.96	28	1074.22	100000	15.92	29	2148.44
SSR[18]	29985	14.58	48	783.41	49984	15.12	48	1301.87	99985	15.94	51	2579.01
PV[19]	27995	13.89	15	2870.58	43831	14.63	16	4494.39	73945	15.49	17	7582.25
DPT[23]	74399	15.23	17	1307.79	176596	16.51	18	3104.23	581994	18.17	20	10230.36
Prop. Binary	35486	14.15	16	658.43	68979	15.09	17	1279.88	159408	15.68	18	2957.77

the LPFST based on a trie is more unbalanced, the LPFST has also poor lookup speed. The IPv6 features of the other binary search based schemes such as BPT, MBPT and SSR, are similar to their IPv4 features. The performances of these algorithms are not seriously dependent upon the increased address length in IPv6. Otherwise, since the PV requires more prefix vectors in IPv6, the memory requirement of the PV is the worst. Actually, the node size of the PV in IPv6 is impractical so that the PV cannot be used in IPv6 networks. The DPT requires more memory space because of the increased entries by the leaf-pushing as well. Considering the various aspects of other algorithms, the proposed binary search algorithm has good scalability toward IPv6 and will be useful in the future IPv6 routers.

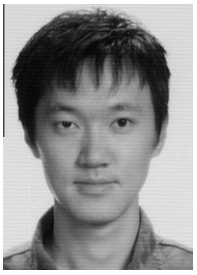
5. Conclusion

In this paper, we propose an efficient IP address lookup algorithm based on a small balanced tree by the use of a new entry reduction. In order to construct a balanced tree, the leaf-pushing technique is used for making the completely disjointed entries. In an attempt to reduce the increased number of entries created by using leaf-pushing, the proposed entry reduction method merges similar pairs of prefixes, in terms of prefix strings and output ports, into single common prefixes in the leaf-pushing binary trie. In this way, a smaller number of completely disjointed prefix entries can be achieved. In the proposed binary search algorithm, the small balanced binary search tree is constructed by sorting these reduced disjointed entries with a small node size. In the proposed multi-way search algorithm, the lookup speed of the binary search algorithm is improved. As a result, the proposed algorithms offer faster lookup speed along with reduced memory requirements, as compared with other algorithms. Furthermore, these provide incremental updating and good scalability for a large routing data and IPv6.

References

[1] H.J. Chao, Next generation routers, Proceedings of the IEEE 90 (9) (2002) 1518–1558.
[2] M.A. Ruiz-Sanchez, E.W. Biersack, W. Dabbous, Survey and taxonomy of IP address lookup algorithms, IEEE Network 15 (2) (2001) 8–23.
[3] V.C. Ravikumar, R.N. Mahapatra, L.N. Bhuyan, EaseCAM: an energy and storage efficient TCAM-based router architecture for IP lookup, IEEE Transactions on Computers 54 (5) (2005) 521–533.
[4] G. Varghese, Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices, Morgan Kaufmann Publishers, Elsevier Inc., 2005.
[5] C. Labovitz, G.R. Malan, F. Jahanian, Internet routing instability, IEEE/ACM Transactions on Networking 6 (5) (1999) 515–528.
[6] V. Srinivasan, G. Varghese, Fast address lookups using controlled prefix expansion, ACM Transactions on Computer Systems 17 (1) (1999) 1–40.
[7] S. Sahni, K.S. Kim, Efficient construction of multibit tries for IP address lookup, IEEE/ACM Transactions on Networking 11 (4) (2003) 650–662.
[8] S. Nilsson, G. Karlsson, IP address lookup using LC-tries, IEEE Journal on Selected Areas in Communications 17 (6) (1999) 1083–1092.
[9] M. Degermark, A. Brodnik, S. Carlsson, S. Pink, Small forwarding tables for fast routing lookups, in: Proc. ACM SIGCOMM, 1997, pp. 3–14.
[10] W. Eatherton, G. Varghese, Z. Dittia, Tree bitmap: hardware/software IP lookups with incremental updates, ACM SIGCOMM Computer Communication Review 34 (2) (2004) 97–122.
[11] S. Sahni, L. haibin, Dynamic Tree Bitmap for IP Lookup and Update, in: Proc. International Conference on Networking (ICN), 2007, pp. 79–84.
[12] H. Lim, C. Yim, E. Swartzlander Jr., Priority tries for IP address lookup, IEEE Transactions on Computers 59 (6) (2010) 784–794.
[13] L. Wuu, T. Liu, K. Chen, A longest prefix first search tree for IP lookup, Computer Networks 51 (12) (2007) 3354–3367.
[14] N. Yazdani, P.S. Min, Fast and scalable schemes for the IP address lookup problem, in: Proc. IEEE HPSR, 2000, pp. 83–92.

[15] C. Yim, B. Lee, H. Lim, Efficient binary search for IP address lookup, IEEE Communications Letters 9 (7) (2005) 652–654.
[16] M.J. Akhbarizadeh, M. Nourani, Hardware-based IP routing using partitioned lookup table, IEEE/ACM Transactions on Networking 13 (4) (2005) 769–781.
[17] H. Lim, B. Lee, W. Kim, Binary searches on multiple small trees for IP address lookup, IEEE Communications Letters 9 (1) (2005) 75–77.
[18] H. Park, H. Kim, H.-S. Kim, S. Kang, A fast IP address lookup algorithm based on search space reduction, IEICE Transactions on Communications E93-B (4) (2010) 1009–1012.
[19] H. Lim, H. Kim, C. Yim, IP address lookup for internet routers using balanced binary search with prefix vector, IEEE Transactions on Communications 57 (3) (2009) 618–621.
[20] B. Lampson, V. Srinivasan, G. Varghese, IP lookups using multiway and multicolumn search, IEEE/ACM Transactions on Networking 7 (3) (1999) 324–334.
[21] X. Sun, Y. Zhao, An on-chip IP address lookup algorithm, IEEE Transactions on Computers 54 (7) (2005) 873–885.
[22] H. Lu, S. Sahni, Enhanced interval trees for dynamic IP router-tables, IEEE Transactions on Computers 53 (12) (2004) 1615–1628.
[23] H. Lim, W. Kim, B. Lee, Binary Search in a Balanced Tree for IP Address Lookup, in: Proc. IEEE HPSR, 2005, pp. 490–494.
[24] Y. Chang, Fast binary and multiway prefix searches for packet forwarding, Computer Networks 51 (3) (2007) 588–605.
[25] BGP Table obtained from <http://www.potaroo.net/>.
[26] K. Zheng, B. Liu, V6Gene: a scalable IPv6 prefix generator for route lookup algorithm benchmark, in: Proc. IEEE AINA, 2006, pp. 147–152.



Hyuntae Park received the B.S. degree and the M.S. degree in electrical and electronic engineering from Yonsei University, Seoul, Korea, in 2004 and 2006, respectively. He is currently working toward a Ph.D degree in Electrical and Electronic Engineering at Yonsei University. His research interests include computer networking, IP switches/routing, and system-on-a-chip (SoC)/VLSI design related to network systems.

Hyejeong Hong received her B.S. degree in electrical and electronic engineering from Yonsei University, Seoul, Korea, in 2006. She is currently working toward a Ph.D degree in Electrical and Electronic Engineering at Yonsei University. Her research interests include multiprocessor architectures, reconfigurable computing, network security and networking systems.

Sungho Kang received the B.S. degree from Seoul National University, Seoul, Korea, and the M.S. and Ph.D. degrees in electrical and computer engineering from the University of Texas (UT), Austin. He was a Postdoctoral Fellow with UT, a Research Scientist with the Schlumberger Laboratory for Computer Science, Schlumberger, Inc., and a Senior Staff Engineer with Semiconductor Systems Design Technology, Motorola, Inc. Since 1994, he has been a Professor with the Department of Electrical and Electronic Engineering, Yonsei University, Seoul. His current research interests include system-on-a-chip (SoC)/VLSI design, VLSI computer-aided design, and SoC testing and design for testability.