Available online at www.sciencedirect.com
SCIENCE DIRECT
Computer Communications 28 (2005) 529–539
computer communications
www.elsevier.com/locate/comcom

Simple and fast IP lookups using binomial spanning trees

Yeim-Kuan Chang*

Department of Computer Science and Information Engineering, National Cheng Kung University, No. 1, Ta-Hsueh Road, Tainan 701, Taiwan, ROC

Received 18 January 2004; revised 5 October 2004; accepted 26 October 2004
Available online 25 November 2004

Abstract

High performance Internet routers require an efficient IP lookup algorithm to forward millions of packets per second. Various binary trie data structures are normally used in software-based routers. Binary trie based lookup algorithms are simple not only for searching for the longest prefix match but also for updating the routing table. In this paper, we propose a new IP lookup algorithm based on binomial spanning trees. The proposed algorithm has the same advantages of simple search and update processes as the ones based on binary trie. However, the performance of the proposed algorithm in terms of memory requirement and the lookup time is better than the schemes based on binary tries, such as path-compression and level-compression.

Keywords: Binary trie; Binomial spanning tree; IP lookup

1. Introduction

The increase of the Internet traffic continues in an unprecedented rate mostly due to the advent of the World Wide Web (WWW) [7]. Backbone routers with link speed of gigabits per second (e.g. OC-192, 10 Gigabits and OC-768, 40 Gigabits) are thus commonly deployed. Among all the fundamental functions of the routers, IP address lookup is the most critical one. Fast lookup algorithms make packet forwarding rate of the routers keep up with the link speed and router bandwidth. These backbone routers have to forward millions of packets per second at each port.

The routing table in a router that is used to lookup an IP address stores a large number of routing entries, each consisting of a network address that is the prefix of a group of IP addresses and the corresponding output port number to the network. When a router receives a packet, it must determine the next port number through which the packet must be forwarded. The longest prefix in the routing table that matches the destination IP address of the packet is the best match prefix (BMP). Sequential search for the BMP has a time complexity of O(N) which is not scalable.

A large variety of IP lookup algorithms were classified and their worst-case complexities of lookup latencies, update times, and storage consumption were compared [15]. Among them, a category of algorithms is based on a trie/tree structure [4]. The binary trie is the basic data structure used in most of IP lookup algorithms. Binary tries provide an easy way to store and update the prefixes of different lengths and to search for BMP. The binary trie is in fact a binary search tree using the bit value (0 or 1) to guide the search moving toward the left or the right part of the tree. The binary tree structure is usually implemented using a linked list data structure. Each trie node has the left and right pointers pointing to its left and right sub-tree, respectively. The number of nodes and the trie depth may be large in the binary trie for a typical routing table containing over 10k routing entries. Thus, the basic binary trie is not efficient in terms of memory usages and lookup times. The path compression technique can be used to improve the performance of the trie structure by removing the internal nodes with only one child [12,16]. A multibit trie, a simple extension to the binary trie, inspects more than one bit at a time to speedup the search process. A space efficient array implementation for the multibit trie based algorithm is proposed [13,14]. Various multibit trie algorithms were proposed in software and hardware [2,5,8,14,15].

* Tel.: +886 6 2757575x62539; fax: +886 6 2747076.
E-mail address: ykchang@mail.ncku.edu.tw.

0140-3664/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.comcom.2004.10.014

Based upon this primitive trie structure, a set of prefix compression and transformation techniques can also be used to either make the whole data structure small enough to fit in a cache [3], or to transform the set of original prefixes to a different one in order to speed up the tree traversal procedure, e.g. prefix expansion or leaf-pushing [17,18]. The hardware based lookup algorithms using multi-bit trie [5] is in fact a variation of the prefix transformation techniques. The extreme case is a 32-bit extended trie which trades a memory consumption of 32 Gbytes and inefficient prefix updates for only one memory lookup latency. We can classify the 32-bit extended trie as a perfect hashing approach which is obviously not minimal. Since finding a minimal perfect hashing table for the whole set of prefixes is difficult, a binary search on prefix lengths was proposed [18]. In this approach, a binary search scheme is conducted on a set of hash tables, where the prefixes of the same length are organized in one hash table. In [2], the authors use CPU caching hardware to perform routing table caching and lookup directly by using a special design of the cache data structure.

Among all the IP lookup algorithms proposed in the literature, only the binary range search proposed in [1,10] can store the lookup data structure in a sequential array. Instead of trying to store the complete prefixes, the binary range search encodes the prefixes using the start and end addresses of the ranges covered by them. Thus, each prefix is encoded by two full length bit strings. All the start and end addresses of the ranges are sorted and stored in a sequential array. The binary search method can then be applied using the array index. Obviously, a subtle design (e.g. '>' and '=' ports) must be employed to make the binary search on the sequential array correct. The primary idea of the binary range search is to pre-compute the port number when the target IP is equal to one of the start and end addresses of ranges or fall between them.

In this paper, we propose a new IP lookup algorithm that uses a binomial spanning tree [9]. To understand how a binomial spanning tree is constructed, we use the hypercube structure for illustration. A four dimensional hypercube and the corresponding binomial spanning tree are depicted in Fig. 1. In the figure, some of nodes are also shown with their corresponding 4-bit addresses. Each node of the binomial spanning tree is associated with a binary address. To map the prefixes of various lengths onto the nodes of the binomial spanning tree, we convert the prefixes to binary addresses by padding zeros. For example, a prefix 01* in a 4-bit address space is converted to address 0100. Thus, node 0100 of the binomial spanning tree is responsible for storing the routing information of the prefix 01*.

The proposed IP lookup algorithm based on the binomial spanning tree has the similar characteristics to that based on the binary tries. In other words, both of the algorithms based on binomial spanning trees and binary tries have the advantages of simple and easy searching mechanism, tree construction, and updates. The basic searching method for the proposed lookup algorithm is also in a bit-by-bit fashion starting from the most significant bit and following the pointers in the nodes.

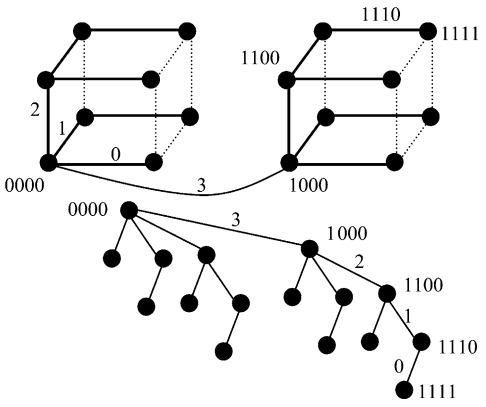

Fig. 1. A 4-cube and its corresponding binomial spanning tree.

The average number of memory accesses and average number of internal pointers for a lookup based on binomial spanning tree are half of that based on the full binary trie. In an n-bit address space, the worst-case number of memory references based on the binomial spanning tree is less than n since we can easily pick a root node which has the maximum hamming distance of less than n from all the prefix nodes. We will then develop some techniques to further reduce the number of nodes and the tree depth in the binomial spanning tree using level and path compression and the special properties of the hypercube. Simulation results that we conducted show that the proposed lookup algorithm based on the binomial spanning trees performs better than that based on the binary tries.

The rest of the paper consists of four sections. Section 2 is preliminaries; notations, terminology, and definitions are introduced. Section 3 is proposed data structure. We describe the data structure of the proposed IP lookup algorithm using binomial spanning tree. Section 4 is performance evaluation. We evaluate the performance of the proposed algorithms in terms of memory requirements and lookup times by analyses and simulation. The last Section 5 is the conclusions.

2. Preliminaries

In this paper we define the notations and terminology as follows.

In an n-cube, the set of node addresses is $\mathcal{N}=\{0,1,\ldots,2^n-1\}$, and the set of dimensions is $\mathcal{D}=\{0,1,\ldots,n-1\}$. The binary address of node i is represented as $(i_{n-1},i_{n-1},\ldots,i_0)$. The bitwise Exclusive-OR operation is denoted as ^ (as in C language).

Definition 1. A binary n-cube is a graph $G=(V,E)$ such that $V=\mathcal{N}$ and $E=\{(i,j)|i\hat{}j=2^m$, for all i and j belong to $\mathcal{N}\}$. An edge (i,j) connects nodes i and j through dimension m.

Definition 2. The Hamming distance between nodes i and j is Hamming $(i,j) = \sum_{m=0}^{n-1}(i_m^j m)$.

Definition 3. A binomial spanning tree (n-BST) with root node $r=(r_{n-1}r_{n-2}\ldots r_0)$ is defined [9] as follows.

The root's children include all the nodes in the set of $\{(r_{n-1}r_{n-2}\ldots r_m\ldots r_0)|m=n-1,\ldots,0\}$. The set of children of another node i with the address $(i_{n-1}i_{n-2}\ldots i_m\ldots i_0)$ is $\{(i_{n-1}i_{n-2}\ldots i_m\ldots i_0)|m=p-1,\ldots,0\}$, where $c=i\hat{}r$ and $c_k=i_k\hat{}r_k$ for $k=n-1,\ldots,0$, and $c_{p-1}=\ldots=c_0=0$ and $c_p=1$. In other words, p is the position of the least significant set bit. The sub-BST containing nodes in the p-cube $0\ldots01_p*$ is connected to the root node along dimension p.

d3.d2.d1.d0/l/p: the *length format* of prefixes. It represents a prefix of length l associated with a next port number p, where d3.d2.d1.d0 is dotted notation of a 32-bit IP address using four octal numbers. The notation d3.d2.d1.d0/l will be used when no confusion is incurred.

$b_{n-1}\ldots b_i*\ldots*/p$: the *ternary format* of prefixes. It represents a prefix of length $n-i$ associated with a next port number p and $b_j=0$ or 1 for $n-1\geq j\geq i$. When we use $t_{n-1},\ldots,t_1t_0$ as the ternary format of a prefix, where $t_i=0$, 1, or * (don't care), we must follow the rule that if t_k is * then t_j must also be * for all $j<k$. For simplicity, a single don't care bit is used to denote a series of don't care bits. Thus, the prefix 1* denotes 1**** in a 5-bit address space.

Prefix enclosure. Consider two prefixes in their ternary format: $A=b_{n-1}\ldots b_j\ldots b_i*$ and $B=b_{n-1}\ldots b_j*$ and $j>i$. Prefix A is said to be enclosed by B since the IP address space covered by A is a subset of that covered by B.

Disjoint prefixes. Two prefixes A and B are said to be disjoint if none of them is enclosed by the other.

3. Proposed data structure

Using the binomial spanning tree as the basic structure to store the routing table is not as simple as we first thought. Each vertex of an n-cube is associated with an n-bit binary address which can be directly mapped to a node in the binomial spanning tree. Storing routing table in the binomial spanning tree will be straightforward when all the prefixes of the routing table are of length 32. Theoretically, the prefix lengths of the routing table for IPv4 are in the range of 0–32. However, in the actual routing tables obtained from typical backbone routers, the prefix lengths range from 8 to 32. We need to devise a method to store a prefix of any length in a node of the binomial spanning tree so that performing an IP lookup will lead to a correct result. What we do is to convert the prefix $b_{n-1}\ldots b_0/l/p$ to $b_{n-1}\ldots b_{n-l}0\ldots0/l/p$ and store it in the node with address $b_{n-1}\ldots b_{n-l}0\ldots0$. Note that the routing tables that are made available on the Internet in fact use this converted strings to record the routing entries.

The above conversion leads to a problem that more than one prefix may be mapped onto one single node of the binomial spanning tree. For example, the node with address $b_{n-1}\ldots b_{n-k+1}l_{n-k}0\ldots0$ may store any prefix in $\{b_{n-1}\ldots b_{n-k+1}l_{n-k}0\ldots0*_x\ldots*_0/l|l=n-1-x$ and $x=-1,\ldots,n-k-1\}$, where $x=-1$ indicates the prefix is of length n. The prefixes stored in a single node of the binomial spanning tree are called the conflicting prefixes. This conflict situation can be solved by two approaches. The first approach called *prefix array* approach uses an additional prefix array in data structure of the node to record the conflicting prefixes. The second approach called *prefix expansion* approach expands the conflicting prefixes of shorter lengths to ones of longer lengths in such a way that no node stores more than one prefix. These two approaches are described as follows.

In the prefix array approach, the node contains a pointer to the prefix array. When a lookup traverses a node of the binomial spanning tree containing a non-empty conflicting prefix array, an additional process would be performed to search the prefix array for the proper match. Since this kind of enclosure situation is rare, the LC trie [14] also uses this approach to solve the problem caused by the enclosure property that internal nodes have no space to store the prefix information. One advantage of this approach is its simple updating process. To delete or insert a new prefix, we just need to go through the pointers of the binomial spanning tree, find the corresponding node, and insert/delete the prefix in/from the prefix array. No other augmented data structure is needed. The optimized node structure to store the array of conflicting prefixes will be described later for improving the lookup performance.

The prefix expansion approach divides the conflicting prefixes into disjoint prefixes. Disjoint prefixes are mapped onto different nodes of the binomial spanning tree. For example, two conflicting prefixes 0*/p and 000*/q in the 4-bit address space are initially mapped to the same node with address 0000. These two prefixes are expanded into 01**/p, 001*/p, and 000*/q. These three disjoint prefixes are then mapped to three distinct nodes with addresses, 0100, 0010, and 0000. One might think that this approach is the same as the approach that removes all the enclosure situations completely, e.g. making the original structure a full tree. The following example shows that it is not. Consider two prefixes, 0*/p and 01*/q in a 4-bit address space. The former prefix encloses the latter. It can be seen that these two prefixes are not conflicting because they are mapped onto two distinct nodes with addresses 0000 and 0100. The advantage of this approach is that no additional array to store the conflicting prefixes is required. However, since the conflicting prefixes are split into many prefixes of longer lengths, deleting an original prefix will not be an efficient process. An additional data structure that records the split prefixes and the associated original prefix would be required.

Fig. 2 shows the binary trie and the corresponding binomial spanning tree for a small routing table. We can see that the depth of the binomial spanning tree is one less than

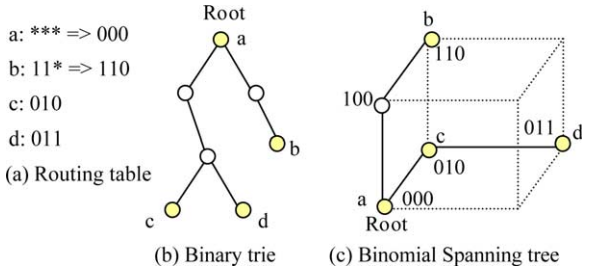

Fig. 2. A small routing table example for the proposed algorithm.

the binary trie and the number of links is two less than that of the binary trie. The number of links is proportional to the required memory space because the links are usually implemented as pointers.

The insertion procedure in the syntax of C programming language for the proposed lookup algorithm is given in Fig. 3. This insertion procedure is the building block of the tree construction and update processes. Formally, the insertion procedure is called every time when a prefix is added in the routing table. The last three parameters of the insertion procedure, *ip*, *len*, and *port*, represent the prefix being added, in the length format. If the hamming distance between the root and the BMP node is *h*, there are at least *h* nodes in the binomial spanning tree that will be traversed or created if necessary in the insertion process. As stated earlier, if conflicting prefixes are split into disjoint prefixes of longer lengths, more nodes will be traversed or created. Lines 2–9 show the core codes that create the necessary nodes along the path from the root to the final BMP node. The destination IP is first Exclusive-ORed with the root's IP. The position of the most significant set bit is then computed first. This operation can be implemented efficiently by using the 'BSR' (bit scan reverse) instruction of the Intel processor family starting from Intel 80386. When coming to the BMP node that is responsible for storing the input prefix, one would check if a conflicting prefix already exists and perform the appropriate conflict resolution operations. Referred to the line 10 in Fig. 3, we assume that if the final node has a port number greater than 0, it is an indication that a conflicting prefix was already assigned to this BMP node. The function *conflict_resolver( )* implements either the prefix array approach or the prefix expansion approach. If no conflicting prefix exists in this final node, we update its *len* and *port* fields. Notice that the node address is not explicitly stored since it can be deduced from traversing the root to the final BMP node.

Finally, we show the lookup procedure in Fig. 4. The lookup process as shown in the figure is similar to the insertion process. However, the matching process implemented in function *matching( )* must be performed in every node traversed by the lookup procedure. The exact implementation of function matching( ) depends on which of the prefix array approach and prefix expansion approach is adopted.

3.1. Node representation

As you may have noticed that the data structure of the node in Figs. 3 and 4 contains predetermined number of pointers depending on where the node locates. The root node has *n* pointers since it may have at most *n* children in an *n*-BST. In general, assuming the address of root node is 0, node *i* ($i_{n-1}\ldots i_p 0\ldots 0$) has *p* pointers if the least significant set bit is the *p*th bit. There are 2^k nodes containing $n-k-1$ pointers, for $k=0$ to $n-1$. A lot of pointer space will be wasted because some of the prefixes may not exist and thus most of pointers are NULLs. We propose to use a bitmap to record which pointers are not NULLs. Take a node with 8 pointers in a 5-bit address space as an example. If only pointers at bits 3 and 5 are not NULL, we use 00101000 as the bitmap and an array of two pointers. When we check whether the pointer at bit *i* is NULL or not, we check if the *i*th bit in the bitmap is set or not. If the *i*th bit of the bitmap is

```
void insertion(node32 *root32, unsigned root_ip,
                    unsigned ip,unsigned len,unsigned port)
{
1   unsigned x = root_ip ^ ip, i;
2   while (x != 0){
3      i =BSR(x);    //i is the most significant set bit of x
4      if (root32->ptr[i] == NULL)
5            root32->ptr[i] = create_node32();
6      root32 = root32->ptr[i];
7      x = x ^ (1<<i);
8      root_ip = root_ip ^ (1<<i);
9   }/*end while */
10     if (root32->port > 0){
11          conflict_resolver(root32, root_ip, ip, len, port);
12     } else {
13          root32->len = len;
14          root32->port = port;
15     }
}
```

```
struct node32 {
   struct node32 *ptr[32];
   unsigned char port;
   unsigned char len;
};
```

Fig. 3. The insertion procedure for the proposed binomial spanning tree.

```
unsigned lookup(node32 *root32, unsigned root_ip,
                unsigned traffic_ip, unsigned default_port)
{
1     unsigned x = root_ip ^ traffic_ip, i, j;
2     while (x != 0){
3           default_port = matching(root32,root_ip, ip);
4           i =BSR(x);    // i is the most significant set bit of x
5           if (root32->ptr[i] == NULL)    return default_port;
6           root32 = root32->ptr[i];
7           x = x ^ (1<<i);
8           root_ip = root_ip ^ (1<<i);
9      }/*end while */
10    default_port = matching(root32,root_ip, ip);
11     return default_port;
}
```

Fig. 4. The lookup procedure for the proposed binomial spanning tree.

zero, the corresponding pointer is NULL. Otherwise, we compute how many set bits that precedes the ith bit (inclusive) to locate the corresponding pointer in the pointer array and follow it to the next level.

For the prefix array approach to dealing with conflicting prefixes, we now present an efficient data structure to store the conflicting prefixes for improving the lookup performance. Assume the address of a node storing many conflicting prefixes is $A = b_{n-1}\ldots b_{n-k+1}l_{n-k}0\ldots0$, where the bit $n-k$ is the least significant set bit of A. The set of conflicting prefixes that may be stored in node A is $\{b_{n-1}\ldots b_{n-k+1}l_{n-k}0\ldots 0*_x\ldots*_0/l | l = n-1-x$ and $x = -1\ldots n-k-1\}$, where $x=-1$ means no don't-care bit exists and thus the prefix length is n. Note that l cannot be less than k because of the following reason. Consider the case when $l = k-1$. We can see that the prefix $b_{n-1}\ldots b_{n-k+1}l_{n-k}0\ldots0/k-1$ should be converted to $b_{n-1}\ldots b_{n-k+1}0_{n-k}0\ldots0/k-1$ and stored in node $b_{n-1}\ldots b_{n-k+1}0_{n-k}0\ldots0$ instead of $b_{n-1}\ldots b_{n-k+1}1_{n-k}0\ldots0$.

A bitmap is used to record which prefixes are stored in the node with address $b_{n-1}\ldots b_{n-k+1}1_{n-k}0\ldots0$. This bitmap can be represented as $0\ldots0_{n-k}p_{n-k-1}\ldots p_0$, where $p_x=1$ for $x=-1$ to $n-k-1$ indicates that prefix $b_{n-1}\ldots b_{n-k+1}l_{n-k}0\ldots0*_x\ldots*/n-1-x$ exists in the node. $x=-1$ means the bitmap is 0 and there is no prefix stored in the node. In addition, a port array is needed to store the corresponding next port numbers to the prefixes stored in the node.

3.2. Optimizations

There are a number of optimization techniques proposed in the literature for the binary trie. Path compression, level compression, and k-level segmentation (called bucketing in [14]) are examples of the optimization techniques. We shall show that these techniques can also be applied to the proposed IP lookup algorithms using the binomial spanning tree approach.

Applying the path compression to the binomial spanning tree is straightforward. The non-prefix node with only one child can be removed by the path compression technique. For example, Fig. 2(c) shows node 100 can be squeezed out and the root directly points to node 110 which stores prefix b.

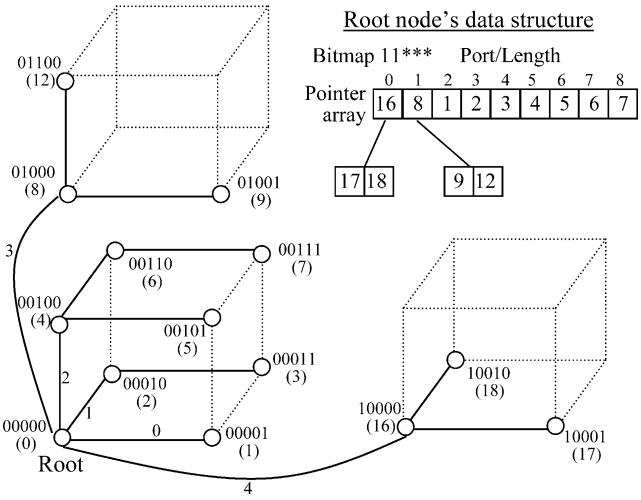

Fig. 5. Level compression example for the proposed algorithm.

Incorporating level compression technique in the binomial spanning tree is similar to finding a subcube. For example, assume there are 14 prefixes that are stored in the nodes marked as circles of a binomial spanning tree shown in Fig. 5. There exists a 3-cube that contains the root node 0 (prefix 0). The data structure of the root node as shown in Fig. 5 contains a modified bitmap, 11***, to indicate that there are nine pointers stored in the node. There are seven pointers pointing the 7 nodes in a 3-cube, 00***, (nodes 1–7) and another two pointers pointing to nodes 8 and 16 along dimensions 3 and 4, respectively. If the incoming IP is 01001, the second pointer (with prefix 8) is first selected because the most significant set bit of IP is in dimension 3. After it reaches node 8, the same process repeats. If the incoming IP is 00101, the 6th pointer will be followed. We can see that the most significant set bit is 2 and it belongs to these three don't care bits. Thus, the position of the selected pointer will be $2+5$ (00101) -1 which is 6.

The k-level segmentation is mostly used in the hardware-based IP lookup algorithms [5,6]. For our binomial spanning tree, we also use the k-level segmentation array of 2^k pointers, each pointing to the corresponding sub-binomial spanning tree of dimension $n-k$. Therefore, when an incoming IP arrives, the most significant k bits are used to locate the corresponding sub spanning tree. Then the usual lookup process can be performed in the sub-spanning tree to find the best matching prefix.

Besides path compression, level compression, and k-level segmentation, an intelligent selection of root node can be used to further reduce the depth of the binomial spanning tree. Consider the example shown in Fig. 2(c), if the node c (010) is selected as root, then the constructed binomial tree rooted at node c has only the depth of one. Nodes a, b, and d

are all one hop from node c. Therefore, the tree depth can be reduced by carefully selecting a root for a sub-tree.

In addition, we use the technique of dual roots, i.e. a pair of diagonally opposite nodes in the n-cube to reduce the depth of the binomial spanning tree. The hamming distance between the pair of diagonally opposite nodes A and B is n, the maximum, in an n-cube system. Node A is the only node that has the hamming distance of n from node B. For example, nodes A and B could be the nodes with addresses 0 and 2^n-1. We will use these two addresses for the dual roots as the default ones when we describe our idea as follows.

The maximum distance between either A or B and any other node in the n-cube is $n/2$. Therefore, we can use this property to build two binomial spanning trees, one rooted at A and the other rooted at B. The previously proposed insertion procedure can be employed directly in the following manner. When the binary address of a prefix using the address conversion scheme described above is closer to root A than B, we insert this prefix in the binomial spanning tree rooted at A. Otherwise, this prefix is inserted into the spanning tree rooted at B. After a prefix has been determined to be inserted in the tree rooted at B, the address conversion scheme is different from the original scheme and will be described later. If there exists an address covered by a prefix that is closer to A than B, then this prefix is inserted into the tree rooted at A. At the same time, it is possible that there is also an address that is covered by a prefix is closer to B than A. If this is the case, then a marker prefix must also be inserted into the binomial tree rooted at B. For example, the prefix, 3.0.0.0/8, has a portion of its addresses closer to A than B, and another portion of addresses closer to B than A. Therefore prefix 3.0.0.0/8 must be inserted into both the trees rooted at A and B. To insert into the tree rooted at node A, we use the original address conversion scheme, while insert into the tree rooted at node B, we use a different conversion scheme in order to reduce the distance between node B and the converted address. What we do is to convert the prefix 3.0.0.0/8 to address 3.255.255.255 by padding ones to the don't-care bits. Thus, when we perform a lookup operation on a destination address 3.255.255.0, we search the tree rooted at B and the BMP will be found at the node with address 3.255.255.255. If the prefix to be inserted is 4.0.0.0/24, then no marker prefix is created, because there is no address covered by the prefix that is closer to B than A.

The lookup process is similar to the insertion. When the incoming IP is closer to root A than B, the lookup process is preformed on the spanning tree rooted at A. Otherwise the lookup process is performed on the tree rooted at B.

4. Performance evaluation

In this section, we shall demonstrate the performance improvements of the binomial spanning trees over the binary trie and Level-Compressed Trie (LC trie) by analyses and computer simulations. Firstly, the number of nodes (memory accesses) traversed for an IP lookup based on the binomial spanning tree will be compared with that based the full binary trie. We shall show by analyses that the average number of memory accesses for an IP lookup based on a full binary trie is double of that based on the corresponding binomial spanning tree. It is known that the links of a graph are normally implemented as pointers and thus taking memory space. The total required memory space for the routing tables based on a binary trie or a binomial spanning tree is proportional to the number of links used. By carefully inspecting how the node space in the binary trie is utilized, we know that the reason for high memory usage in the binary trie is the memory space required for the pointers. Therefore, secondly, we shall show that the number of links used in a full binary trie is double of that in the corresponding binomial spanning tree. The computer simulation results for the memory usages using real routing tables will also be compiled. Finally, we shall conduct experiments based on a common platform to compare the IP lookup times in terms of binomial spanning trees, the binary trie, and the LC trie.

The routing tables of two real routers available on the Internet will be used in the performance evaluation. One table, funet, is obtained from the publicly available site [14] which contains 41,709 routing entries. The other table, oix, containing 120,635 routing entries is obtained from the University of Oregon Route Views Archive project [11]. Since it is almost impossible to obtain the actual IP traffic being routed through the router at the time when the routing table is logged, we use a simulated IP traffic described as follows. Assume the routing entries in the routing table follow the length format. The simulated traffic is constructed first by removing the length and port of the routing entries and then by performing a random permutation. If the final addresses generated above are not in the converted format as stated in Section 3, the address conversion operations must be performed. The simulated traffic assumes that every prefix in the routing table has the same probability of being accessed. The same method was used in [14,15].

Assume the best matching prefix (BMP) of an incoming IP is of length l, say $b_{n-1}\ldots b_0/l$. Consider the binary trie first. The lookup process has to traverse $l+1$ nodes to reach the BMP node. It is possible that this BMP also encloses other prefixes of length longer than l. If it is the case, then the lookup process must continue traversing more nodes downward to make sure that there is no other matching prefix of longer length. Therefore, the number of trie nodes traversed for finding the BMP is at least $l+1$ for the binary trie structure.

Assume the probability that a prefix is of length l is $P(l)$ for $0 \le l \le W$, where W is 32 for IPv4 or 128 for IPv6. $P(l)$ can be easily computed from a routing table. We denote $P(l)$ as $P_{\mathrm{bt}}(l)$ when the routing table is built from the binary trie structure. In other words, the summation of $P_{\mathrm{bt}}(l)$ for all

the prefix lengths must be 1. Therefore, we have the following equation

$$\sum_{l=0}^{l=W} P_{\mathrm{bt}}(l) = 1 \tag{1}$$

Since the converted addresses from the routing tables will be used as the destination IP's, the average number of memory accesses to search for the best matching prefix is at least M_1+1, where

$$M_1 = \sum_{l=0}^{l=W} lP_{\mathrm{bt}}(l) \tag{2}$$

Now, consider the same lookup process for the binomial spanning tree. The number of nodes traversed does not depend on l but on the number of 1's in the converted address of the BMP node. For example, if the BMP prefix is 00110000/6 in an 8-bit address space, then the address of the BMP node is 00110000. The distance between root node (00000000) and the BMP node is 2. Therefore, the number of nodes traversed for the IP lookup of address 00110000 is 3, for accessing the root, the node 00100000, and the node 00110000. The enclosure situation is also applied and thus force the lookup process traverse more nodes downward to make sure that no BMP of longer length exists. Therefore, in general, the number of nodes traversed to reach for the BMP prefix in the binomial spanning tree is at least M_2+1, where M_2 is the number of 1's in the BMP's address.

Consider a prefix of length l, $b_{n-1}\ldots b_0/l$, and the root node with address 0…0. The address of the BMP node is $b_{n-1}\ldots b_{n-l}0\ldots0$. Assume the probability of bit b_i being 0 or 1 is even for $n-1\leq i\leq n-l$. We can easily conclude that the average distance between the root and the node storing the prefix is half of l. For a destination IP address converted from the prefix of length l, the average number of memory accesses for the lookup will be at least $1+l/2$. Since the converted addresses from all the prefixes in the routing table are used as the destination IP's, the average number of memory accesses to search for the best matching prefix is at least M_2+1, where

$$M_2 = \sum_{l=0}^{l=W} \left(\sum_{k=0}^{k=l} \frac{C_k^l}{2^l} k \right) P_{\mathrm{bt}}(l) \tag{3}$$

C_k^l, which is equal to stands for the number of combinations of k elements selected from a set of l elements. We assume that a prefix of length l can be stored in any node of a hypercube of dimension l containing the root node with address 0. Thus, the probability that the node storing this prefix is k hops from the root node is $C_k^l/2^l$. By a simple calculation, we have

$$M_2 = \sum_{k=0}^{k=W} kP_{\mathrm{bst}}(k), \quad \text{where } P_{\mathrm{bst}}(k) = \sum_{l=0}^{l=W} \frac{C_k^l}{2^l} P_{\mathrm{bt}}(l) \tag{4}$$

Based on the above analysis, we illustrate the performance advantage of the binomial spanning tree over binary trie by showing the probability distribution of prefix length in the routing table. We plot the probability $P_{\mathrm{bt}}(l)$ of the oix table, the analyzed $P_{\mathrm{bst}}(l)$, and the computed $P_{\mathrm{bst}}(l)$ in Fig. 6. It can be seen that analyzed $P_{\mathrm{bst}}(l)$ is normally distributed because we assume that the probability of a bit being 0 or 1 is even. The simulated $P_{\mathrm{bst}}(l)$ matches the computed $P_{\mathrm{bst}}(l)$ closely. We can see that the binomial spanning tree performs much better than the binary trie. We also plot the probability distributions of the numbers of memory accesses for the binomial spanning trees using one root and dual roots with 0-level, 8-level, and 16-level segmentations in Fig. 7. The average number of memory accesses for the 16-level segmentation with dual roots is reduced to less than 4 which is comparable with the LC trie. However, we shall show by simulations that the binomial spanning tree in fact performs better than LC trie and binary trie in terms of economized memory usage and shortened average lookup time.

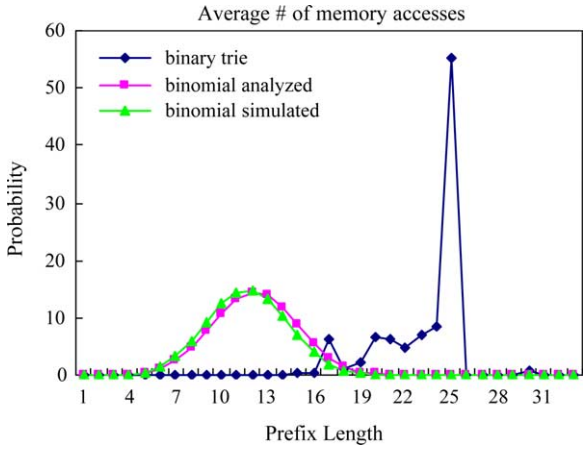

Fig. 6. Probability distribution of the number of memory accesses for binary trie and binomial spanning tree.

Now, we show that the number of links used in a full binary trie is also double of that in the corresponding

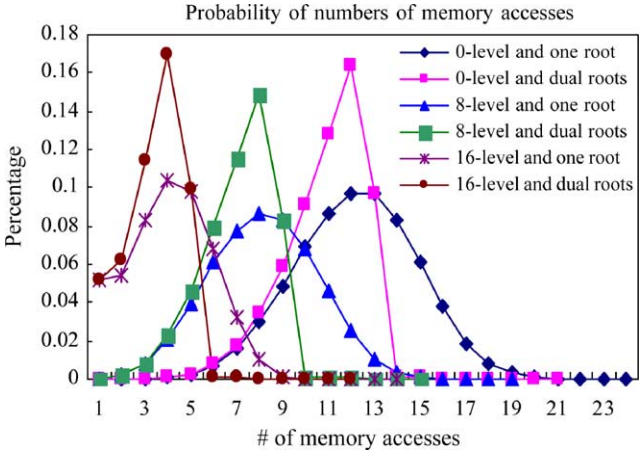

Fig. 7. Probability distribution of the number of memory accesses for the binomial spanning tree with 0-level, 8-level, 16-level segmentation all with one root or dual roots.

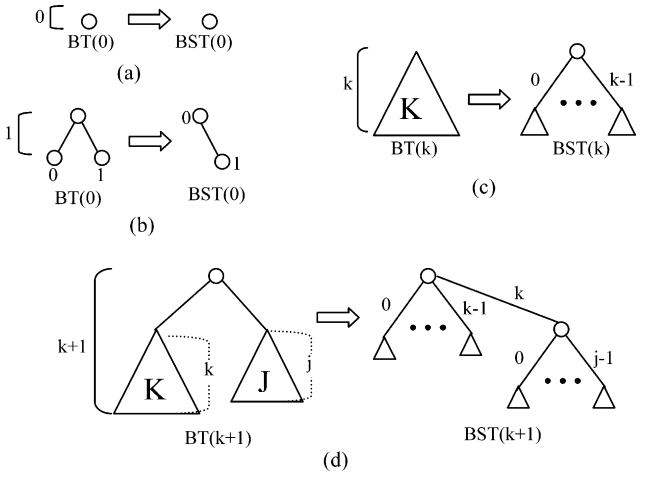

Fig. 8. The full binary tries and the corresponding binomial spanning trees.

binomial spanning tree. Let BT(k) be the number of links in a full binary trie of depth k. Assume BST(k) is the number of links in a binomial spanning tree that corresponds to a full binary trie BT(k). For example, Fig. 8(a) shows a BT(0) and the corresponding BST(0). Both BT(0) and BST(0) have one node and no link. Fig. 8(b) shows BT(1) and BST(1), where BT(1) has three nodes and two links and BST(1) has two nodes and one link.

Theorem 1. *The number of links in a full binary trie is double of that in the corresponding binomial spanning tree*.

Proof. We prove it by induction. Assume the theorem is true and thus BT(k) = 2BST(k). We will prove that the theorem is also true for BT($k+1$) = 2BST($k+1$). Assume a full binary trie of depth $k+1$ is constructed from two full binary tries of depth k and j as shown in Fig. 8(c). Without loss of generality, we assume $j \leq k$. By a trivial conversion, the original full binary trie of depth $k+1$ can be converted to the corresponding binomial spanning tree as shown in Fig. 8(d). Therefore, BT($k+1$) = 2 + BT(k) + BT(j) and BST($k+1$) = 1 + BST(k) + BST(j). Then we have BT($k+1$) = 2BST($k+1$). Thus, the theorem follows. □

Now we compute the memory usages and average lookup times for the schemes based on the binomial spanning tree,

Table 1
Assumptions for node sizes (in bytes) for the binomial spanning tree, the binary trie, and the LC trie

	Length	Port	Pointer	Bitmap
Binomial				
0 level	1	1	4	4
8 level	1	1	4	4
16 level	1	1	4	4
Binary trie	1	1	4	
	Length	Port	String	Prefix index
LC				
Base	1	1	1	4
Prefix	1	1	–	4
Trie node	–	–	4	

Table 2
Total memory requirement in Kbytes for the binomial spanning tree, the binary trie, and the LC trie with fill factor = 0.5

	0-level	8-level	16-level
Binomial			
One root	1703	1533	1670
Dual roots	1831	1631	1896
Binary trie	3062	3020	3212
LC trie	–	2022	2210

the binary trie, and the LC, using the real routing tables. The prefix expansion approach mentioned earlier is used here for solving the prefix conflicts. We have implemented the schemes based on the binomial spanning tree and the binary trie. We obtain the C codes for the LC trie from the web site published by the authors in [14]. In fact, we also implemented a variant of path compressed scheme. Since the performance of the path compressed scheme does not perform better than LC [15], its results are not given in this paper for clarity.

To have a fair comparison for the memory usages, we make similar assumptions for the data structures used in these three lookup schemes, while the data structures of base and prefix arrays in the LC trie [14] actually take more memory storage. These assumptions are shown in Table 1. The port and length are one byte for all the schemes. The pointer field in the binary trie and the binomial spanning tree takes 4 bytes. For LC trie, we do not consider the space for 'nexthop' array [14] since we assume the next hop port number is one byte and is stored in the base vector. Table 2 shows the computed memory usages for the oix routing table. The binomial spanning tree with an 8-level segmentation uses the least memory compared with other schemes. The binomial spanning tree with a 16-level segmentation uses more memory than that with an 8-level segmentation, because the 16-level segmentation table is large. Table 2 does not show the memory usage for the LC trie with no segmentation because the branch at the root will be at least

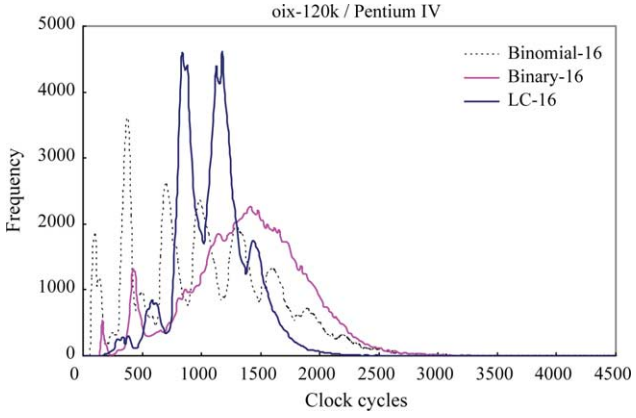

Fig. 9. The binomial spanning tree using one root, the binary trie, and the LC trie with a 16-level segmentation for oix routing table on Pentium IV.

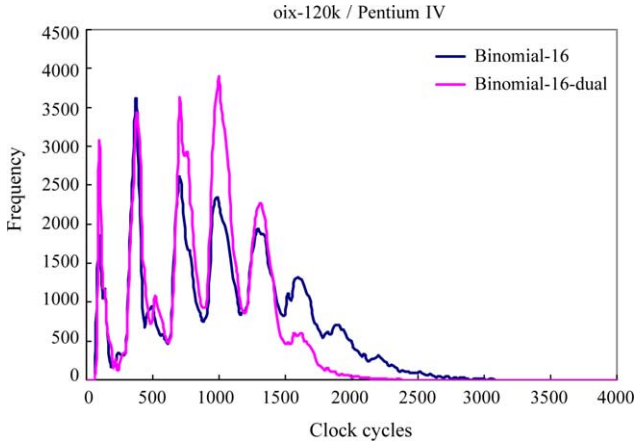

Fig. 10. The binomial spanning trees using one root and dual roots with a 16-level segmentation for oix routing table on Pentium IV.

7 bits when the fill factor is 0.5. The binary tries indeed need a lot more memory than the other schemes.

The experiments for measuring lookup times are conducted with 0-level, 8-level, and 16-level segmentations on the Intel Pentium III and IV processors. As stated earlier that the IP traffic is taken from the routing table and is randomized before feeding into our simulator. The clock cycles are measured by using the special instruction, rdtsc (read time stamp counter), provided by Intel Pentium processor. The clock counts obtained from different CPUs may have different scales. For example, the clock counts on Pentium IV are more than that on Pentium III for the same experiments. It does not necessarily mean that the conducted algorithms perform better on Pentium III than on Pentium IV. We need to convert the clocks to seconds, which is an easy task. However, we do not perform this conversion for maintaining the clarity of the figures shown.

Figs. 9 and 10 show the results of the experiments with a 16-level segmentation in clock cycles on a 2.4 G Pentium IV processor with 8 KB L1 and 256 KB L2 caches. In Fig. 9, we can see that there are similar number of peaks for the binomial spanning tree and the LC trie. However, the peaks of the binomial spanning tree locate toward to the left end. This means the time taken for the binomial spanning tree is

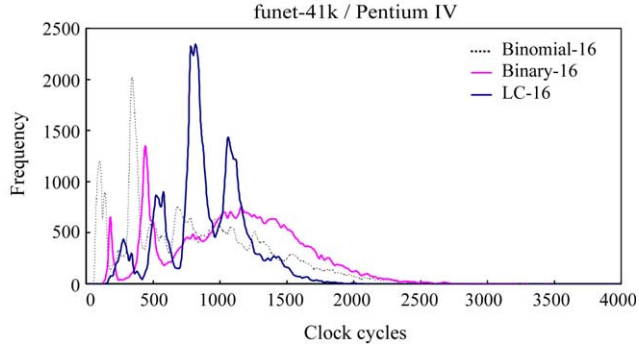

Fig. 11. The binomial spanning tree using one root, the binary trie, and the LC trie with a 16-level segmentation for funet routing table on Pentium IV.

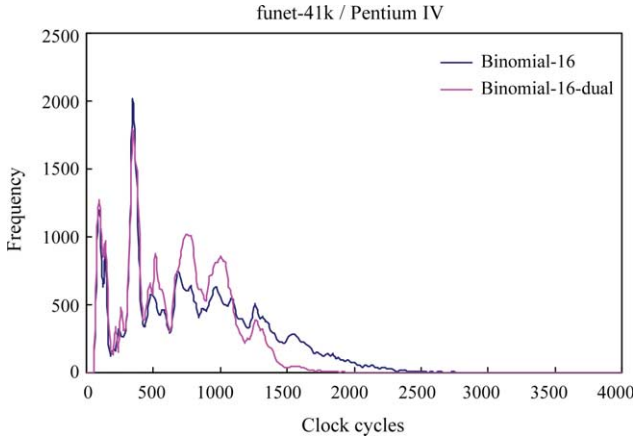

Fig. 12. The binomial spanning trees using one root and dual roots with a 16-level segmentation for funet routing table on Pentium IV.

smaller than the binary trie and LC trie. Note that there are five peaks in LC because the maximum number of tree levels in LC is five. In Fig. 10, we compare the binomial spanning trees with one root and dual roots. The shapes of

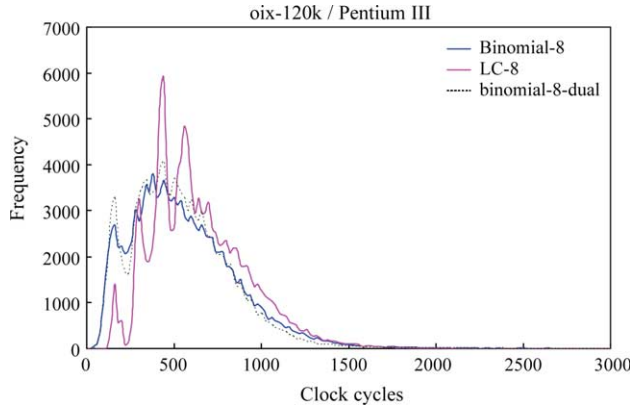

Fig. 13. The binomial spanning trees using one root and dual roots and the LC trie with an 8-level segmentation for oix routing table on Pentium III.

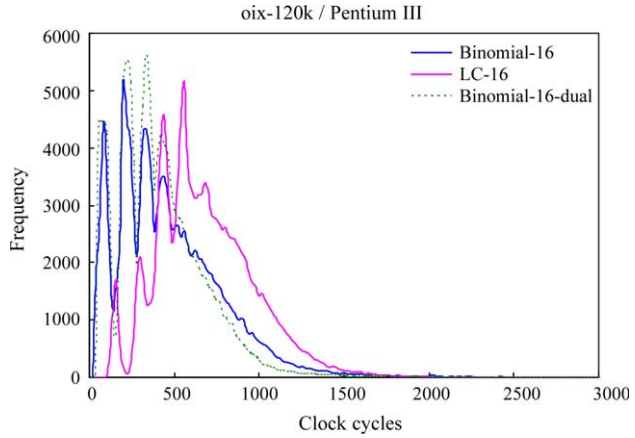

Fig. 14. The binomial spanning trees using one root and dual roots and the LC trie with a 16-level segmentation for oix routing table on Pentium III.

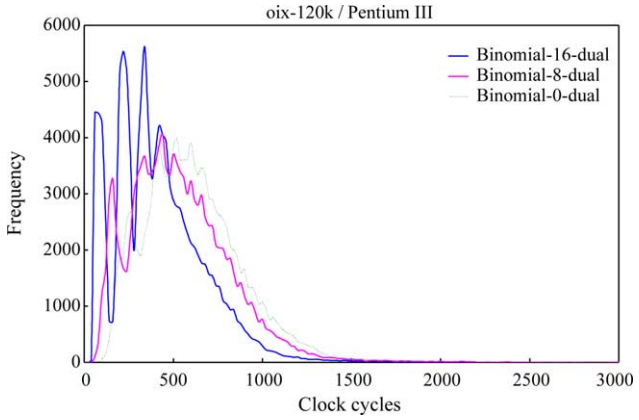

Fig. 15. The binomial spanning trees using dual roots with 16-level, 8-level, and 0-level segmentations for oix routing table on Pentium III.

Table 3
Average cycle counts of the binomial spanning trees, the binary tries, and the LC tries with 0-level, 8-level, and 16-level segmentations for oix and funet routing tables on the Pentium IV processor

Scheme	Clock cycles	
	oix	funet
Binomial-x-0	1467	1165
Binomial-x-0-d	1362	1064
Binomial-x-8	1287	875
Binomial-x-8-d	1140	845
Binomial-x-16	827	528
Binomial-x-16-d	682	483
LC-trie-x-0	1208	587
LC-trie-x-8	905	578
LC-trie-x-16	879	557
Binary-trie-x-0	1802	1357
Binary-trie-x-8	1989	1553
Binary-trie-x-16	1395	1094

the curves for these two binomial spanning trees are similar except curves on the right side. There are more hits on longer cycles in the binomial spanning tree with only one root than that with dual roots. This improvement is because lookups with longer cycles in the binomial spanning tree with one root is transferred to tree rooted at the diagonally opposite root. The lookups with shorter cycles are not affected.

The results of the same experiments for funet routing table are depicted in Figs. 11 and 12. The difference between these two routing tables is not significant. The binomial spanning tree still performs better than other schemes.

We also conducted experiments on the 1 G Pentium III CPU with 16 KB L1 and 256 KB L2 caches. We only show the results for the oix table here. As shown in Fig. 13 with an 8-level segmentation, the binomial spanning trees using one root and dual roots have more hits on shorter cycles than the LC trie. The LC trie has the same peaks as previous experiments on Pentium IV. The peaks in the curve of the binomial spanning trees are not as prominent as the LC trie. Fig. 14 shows the similar results as Fig. 13 using a 16-level segmentation. The proposed binomial spanning trees perform consistently better than the LC counterpart. Fig. 15 compares the performance for the binomial schemes using dual roots with 0-level, 8-level, and 16-level segmentations. Obviously, the binomial scheme using dual roots with a 16-level segmentation performs the best as demonstrated by more hits at the shorter clock cycles. In order to summarize the performance for all the schemes run on Pentium IV processor, we calculate the average clock cycles and show the results in Table 3. Again, the binomial spanning tree with dual roots performs better than any other scheme.

5. Conclusions

In this paper, we introduced a new method for the IP lookups based on the binomial spanning tree. By mapping the prefixes of different lengths on the vertices of an n-cube, we can construct the binomial spanning tree using simple tree construction and update procedures. The fundamental binomial spanning tree can be optimized by using the path compression, level compression, *k*-level segmentation, and the property of diagonally opposite nodes. By analyses and computer simulations, we show that the proposed scheme based on the binomial spanning tree performs the best in terms of memory consumption and lookup time.

References

[1] M. Akhbarizadeh, M. Nourani, IP routing based on partitioned lookup table, in: Proceedings of the IEEE International Conference on Communications (ICC), April 2002, pp. 2263–2267.
[2] T. Chiueh, P. Pradhan, High performance IP routing table lookup using CPU caching, in: Proceedings of INFOCOM 99, March 1999, pp. 1421–1428.
[3] M. Degermark, A. Brodnik, S. Carlsson, S. Pink, Small forwarding tables for fast routing lookups, in: Proceedings of ACM SIGCOMM, October 1997, pp. 3–14.
[4] E. Fredkin, Trie, memory, Communications of the ACM 1960; 490–500.
[5] P. Gupta, S. Lin, N. McKeown, Routing lookups in hardware at memory access speeds, in: Proceedings of INFOCOM 99, March 1999, pp. 1240–1247.
[6] N.F. Huang, S.M. Zhao, J.Y. Pan, C.A. Su, A fast IP routing lookup scheme for gigabit switching routers, in: Proceedings, INFOCOM 99, March 1999.
[7] G. Huston, Analysis of the Internet's BGP routing table, Internet Protocol Journal 4 (1) (2001).
[8] Intel IXA SDK Documentation Library, L3 Forwarder microACE, 2001 in http://www.capsl.udel.edu/~fchen/projects/np/docs/manual/
[9] S.L. Johnsson, C.-T. Ho, Optimum broadcasting and personalized communication in hypercubes, IEEE Transactions on Computers 38 (9) (1989) 1249–1268.
[10] B. Lampson, V. Srinivasan, G. Varghese, IP lookups using multiway and multicolumn search, IEEE/ACM Transactions on Networking 3 (3) (1999) 324–334.

[11] D. Meyer, University of Oregon Route Views Archive Project: oix-damp-snapshot-2002- 12-01-0000.dat.gz, at http://archive.routeviews.org/
[12] D. Morrison, PATRICIA—practical algorithm to retreive information coded in alfanumeric, Journal of the ACM 15 (4) (1968) 514–534.
[13] S. Nilsson, Level-Compressed Trie Structures, Licentiate Thesis, Lund University, Sweden, 1995.
[14] S. Nilsson, G. Karlsson, IP-address lookup using LC-tries, IEEE Journal on selected Areas in Communications 17 (6) (1999) 1083–1092.
[15] M.A. Ruiz-Sanchez, E.W. Biersack, W. Dabbous, Survey and taxonomy of IP address lookup algorithms, IEEE Network Magazine 15 (2) (2001) 8–23.
[16] K. Sklower, A tree-based packet routing table for Berkeley Unix, Proceedings of Winter Usenix Conference, 1991, pp. 93–99.
[17] V. Srinivasan, G. Varghese, Fast address lookups using controlled prefix expansion, ACM Transactions on Computer Systems 17 (1) (1999) 1–40.
[18] M. Waldvogel, G. Varghese, J. Turner, B. Plattner, Scalable high-speed IP routing lookups, in: Proceedings of ACM SIGCOMM, October 1997, pp. 25–36.