Parallel-Search Trie-based Scheme for Fast IP Lookup

Roberto Rojas-Cessa, Lakshmi Ramesh, Ziqian Dong, Lin Cai, and Nirwan Ansari
Department of Electrical and Computer Engineering,
New Jersey Institute of Technology,
Newark, NJ 07102.
Email: {rrojas, lr9, zd2, lc76, ansari}@njit.edu.

***Abstract*—As data rates in the Internet increase, the Internet Protocol (IP) address lookup is required to be resolved in shorter resolution times. IP address lookup involves finding the longest matching prefix from a database of prefixes that better matches the destination address of a packet. The fastest IP-address lookup solutions are based on ternary content addressable memories (TCAMs), which can resolve the IP lookup in one memory-access time. However, TCAMs have a high power consumption and large complexity that may limit their scalability and storage capacity. An alternative is to use random access memory (RAM) that stores a forwarding table in a trie form. Proposed trie-based solutions for IP lookup require three or more memory-access times in the worst-case scenario. This makes them unattractive despite their reduced power consumption. In this paper, we propose a flexible and fast trie-based IP-lookup algorithm where parallel searching is performed. This algorithm performs lookup in two memory-access times whith a feasible amount of memory or three memory access times with reduced memory.**

***Index Terms*—Trie search, parallel search, prefix expansion, hashing, RAM based**

I. Introduction

Classless inter-domain routing (CIDR) allows Internet routers to store a large number of Internet addresses compactly. While reducing the number of entries in the forwarding table, CIDR increases the complexity of the address-lookup procedure because the longest prefix match is sought rather than the exact prefix match. An efficient IP-lookup algorithm: 1) performs a small number of memory accesses, if not one, for a single lookup, and 2) uses a feasible amount of memory to store the prefix information. Because of long memory-access times and slow advances in improving memory speed, we consider that reducing the number of memory-access times is critical in keeping up with the ever-increasing data link rates. Furthermore, it is required to keep the memory amount within a feasible amount for an algorithm to be practical.

The fastest IP-lookup engines are based on ternary content addressable memory (TCAM). Basically, in a TCAM-based IP-lookup engine, the packet destination address is compared to all entries in every memory location. Therefore, it is possible to retrieve the longest prefix match in a single TCAM memory-access time. However, this performance is achieved at the cost of having high power consumption and complex circuitry surrounding the memory cells.

A known alternative is a trie-based scheme that uses random access memory (RAM). In the basic trie-based scheme, a binary tree represents all combinations existing in the forwarding table. In this scheme, the worst-case time takes up to 32 memory-access times to find the longest prefix match for IPv4, as described in PATRICIA trees [1]. Other improved schemes are presented in [2], which uses small forwarding tables at the expense of requiring up to 12 memory-access times, in [3], which uses 4-bit strides, requiring up to 8 memory-access times, and in [4], using a small amount of memory and up to 3 memory-access times.

In this paper, we propose a trie-based IP-lookup scheme, which performs parallel search of the matching longest prefix. To reduce the search complexity, the proposed scheme uses controlled prefix expansion [5]. Our scheme uses independent memories for allowing parallel access, and finds the longest prefix match in two memory-access times with a feasible amount of memory or three access times with a reduced amount of memory. The presented algorithm is flexible for routing tables with diverse prefix length distributions. This scheme presents high scalability since memories are assigned separately per prefix length.

The remainder of the paper is organized as follows. Section II describes the proposed scheme, the data structures and components used in it. Section III describes the implementation of our proposed scheme. Section IV describes the lookup procedure. Section V discusses the complexity and performance. Section VI presents our conclusions.

II. Parallel-Search Trie-based Scheme

The proposed scheme is based on performing parallel access to independent memory blocks, where each block stores the entries existing for each group of a given prefix length. In this paper, we refer to a prefix length as a tree level, i.e., there are up to 32 levels for IPv4 prefixes. Table I shows an example of the contents of a routing table using CIDR as prefixes and Figure 1 shows these entries in a binary-tree structure. In this case, the tree has eight levels, where level one is indicated by the first node below the root, and level eight at the lowest nodes of the tree. Since each level can be provisioned independently of the existence of a prefix node, each level

This work was supported in part by National Science Foundation under Grant Award 0435250.

Prefix	Next Hop
01*	21
10*	28
110*	9
1011*	1
0000*	68
01011*	51
00110*	3
10001*	6
100001*	33
10000000*	54

TABLE I
EXPANDED FORWARDING TABLE.

Original Prefix	Expanded Prefix	Next Hop Pointer	Length
01*	01*	21	2
10*	10*	28	
0000*	00000*	68	5
	00001*	68	
1011*	10110*	1	
	10111*	1	
110*	11000*	9	
	11001*	9	
	11010*	9	
	11011*	9	
00110*	00110*	3	
01011*	01011*	51	
10001*	10001*	6	
100001*	10000100*	33	8
	10000101*	33	
	10000110*	33	
	10000111*	33	
10000000*	10000000*	54	

TABLE II
EXPANDED FORWARDING TABLE.

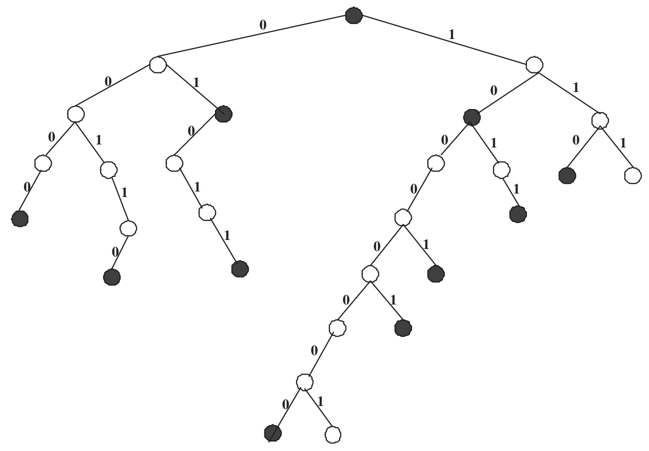
Fig. 1. Binary tree representing the forwarding table.

can be searched in parallel. However, this would require a large amount of memory. To decrease the memory amount, the number of levels (with prefixes) is minimized, into a small number of target levels. To minimize the number of levels, we use *controlled prefix expansion* [5]. The target levels can be selected by using the most populated levels of a forwarding table while considering the memory amount required by each level. Once the levels are selected, the existing prefixes in the removed levels are expanded to the immediate-longer target level. For the sake of brevity, we use an example to describe our scheme, without losing generality.

Table II shows the expanded prefixes that result from Figure 1. Figure 2 shows the expanded prefixes in a tree structure for levels 2, 5, and 8, as target levels.

In order to do the prefix expansion, we select all the prefixes whose lengths ar other than the target set of prefix lengths and expand them to the next allowable prefix length. For example, from Table II, the original prefix 1011* (P4) is of length four, which is not a predefined prefix length. This prefix is extended to the closest allowable predefined prefix length which is length five. This results in two prefixes of length five: 10110* and 10111*, which are expansions of P4 by adding a 0 and a 1 to the least significant bit of the expanded prefixes respectively. Both expanded prefixes inherit the next hop number of the original prefix. In addition to the procedure mentioned above, the condition where one of the expanded prefixes is the same as a prefix with original length equal to the selected level is also considered. In this case, the original prefix overrides the expanded prefix, meaning that only the original prefix is stored along with its port number.

To select the target levels in our scheme, we looked into actual routing tables in [6] and observed that a large number of the prefixes are found between levels 16 and 24. Considering that population, we selected levels 8, 16, 24, and 32 as the target levels in our scheme. This selection can be changed according to the prefix-length distribution of a given routing table where this scheme is applied.

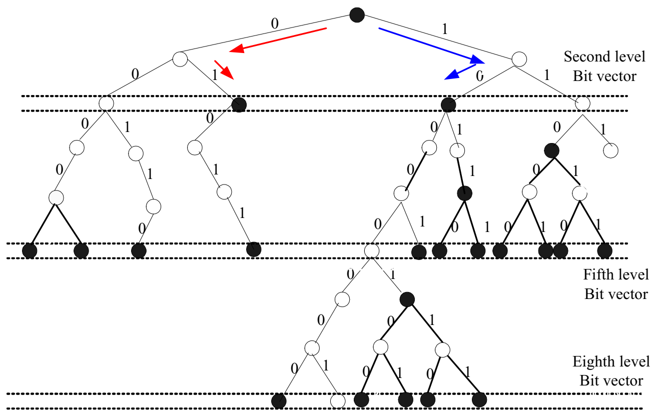

Fig. 2. Bit vectors and stored prefixes in extended-prefix tree.

A. Data Structures at Target Levels

The combined contents of the target levels must contain all the existing prefixes in the original forwarding table. The set of all possible nodes at each level are represented with bitmaps, where each bit position represents a binary combination corresponding to the bits indicated by the prefix length. In a bitmap, a bit with value of 1 indicates the presence of a stored prefix, and a 0 denotes the absence of it. The left-most bit of the bitmap corresponds to the decimal 0, and the right-most bit corresponds to the decimal 2^{level}-1, where $level$ is the level number. The bitmaps of level 8 and 16 are called bit vectors as 2^8 and 2^{16} bits are used, respectively, independently of the existence of prefixes for each bit in the bitmap. The bitmaps for level 24 and 32 are called bit segments as only partial bit vectors containing one or more prefixes are used. Figure 3 shows a general representation of binary tree of the expanded forwarding table with bit vectors and segments. The data structure in each level carries the following information:

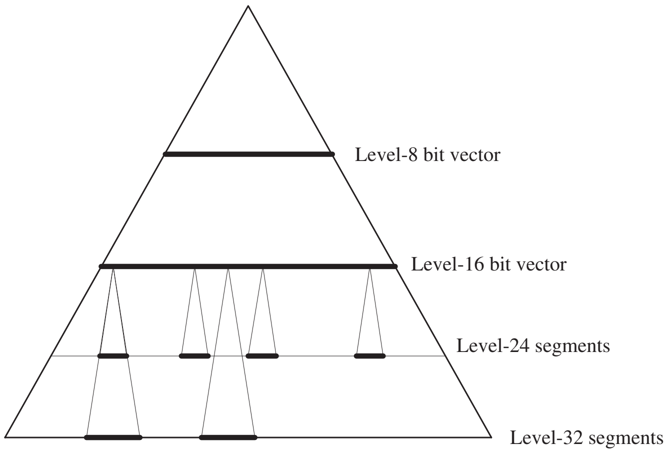

Fig. 3. General representation of binary tree of the expanded forwarding table with bit vectors and segments.

The **level-8 bit vector** uses one bit for each node at this level. It includes all the prefixes between level 1 and 7, which are expanded to level 8, and level-8 prefixes. The level-8 bit vector, denoted as *prefixVal8*, has 256 bits. Each bit represents an 8-bit prefix at this level. Note that, since this level contains a small number of bits, the bit vector can be stored in a memory block together with the next-hop information.

A **level-16 bit vector** includes all nodes at level 16. Each node indicates the existence of all the prefixes between level 9 and 15, which are expanded to level 16, and level-16 prefixes. The level-16 bit vector, denoted as *prefixVal16*, has 2^{16} bits. In addition to the *prefixVal16* bitmap, there are two other bitmaps at this level: *childVal24*, which indicates whether there is one or more prefixes of length between 17 and 24 that share each 16-bit combination indexed by *prefixVal16*, and *childVal32*, which indicates whether there is one or more prefixes with a length between level 25 and 32 that share each 16-bit combination indexed by *prefixVal16*.

Furthermore, the level-16 bit vectors are physically divided into 32-bit chunks. For every 32-bit chunk, there is an offset value. *offsetVal16* is the offset value for *prefixVal16*, *offsetVal24* is the offset value for *childVal24*, and *offsetVal32* is the offset value for *childVal32*. The offset value of bit-chunk of bit n_{16}, where n_{16} is the bit at level 16 in *prefixVal16*, *childVal24*, or *childVal32*, stores the total number of ones accumulated from all previous chunks. The size of each chunk of these three offset fields is 16 bits. Figure 4 depicts the data structure for level 16 with all bitmaps.

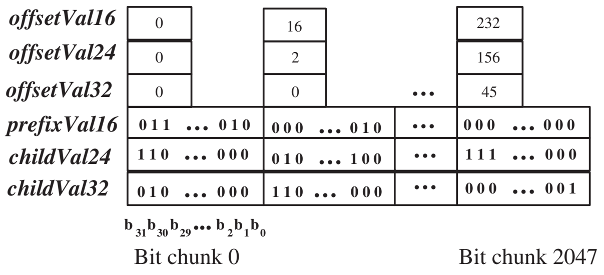

Fig. 4. Bit vector at level 16.

A **level-24 bit segment** carries 256-bit intervals of the level-24 bitmap that correspond to the subtrees rooted by *prefix16*, that have one or more stored prefixes located between level 17 and 24. These intervals are stored in a pseudo-continuous way to reduce memory use. This bit segment is denoted as *prefixVal24*. The sum of *offsetVal24* and the number of ones to the left of the *childVal24* bit in the chunk is used to find the corresponding interval at level 24. The data structure described so far is enough to find the longest match if each prefixVal bit gets assigned a port number directly. However, although this data structure allows to complete the lookup in two memory access times, it is not economical. To reduce the amount of memory to store the next hop information, we use 256-bit intervals, called *portInterval24*, to indicate which nodes share port numbers, and 24-bit fields, called *offsetPort24*, in another bitmap to indicate the an offset value of the memory location of the next hop information for prefixes at level 24. The value of *offsetport24* indicates the number of prefixes that has next hop information in the previous segments of level 24. The value of *portInterval24* indicates the existence of the next hop information that is duplicated to reduce the number of port number memory. The memory location of the next hop information is calculated by adding the decimal value of *offsetPort24* with the ones of *portInterval24* at and the left of the matching bit. Figure 5 depicts the data structure for level 24 with all bitmaps.

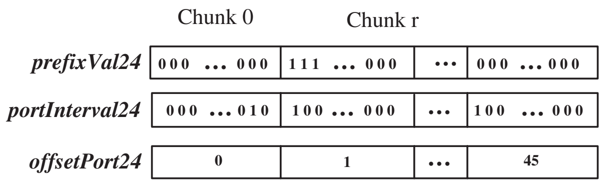

Fig. 5. Bit vectors at level 24.

A **level-32 bit segment** carries 2^{16}-bit intervals at level 32, which correspond to the subtrees, rooted by *prefix16*, with one or more stored prefixes between level 25 and 32. This segment bitmap, denoted as *prefixVal32*, is used in the same way as level-24 bit segment. The 2^{16}-bit *portInterval32* and 32-bit *offsetPort32* fields are used to indicate the memory location of the next hop information for prefixes at level 32. The memory location for next hop information is calculated by adding the decimal value of *offsetPort32* with the ones of *portInterval32* at and the left of the prefix match point.

The **next-hop information** for prefixes in each level is stored in several tables, one table per level, called $tableNextY$, where $Y = \{8, 16, 24, 32\}$ for the case of reduction of port number memory. In general, we use two schemes in storing the next-hop information in memory. One is to associate every bit within a segment with a 15-bit next-hop information without considering whether the prefix exists. The memory amount required is $T \times 2^{Y-16} \times 15$, where T is the number of segments in level Y, $Y = 24, 32$. Another scheme is to store the next-hop information for the positions where prefix exists. We use *portInervalY* and *offsetPortY*, where $Y = 24, 32$ for level 24 and level 32 to find the location of the next-hop information as explained previously. The memory amount needed in this scheme is $S \times 15 + 2 \times T \times 2^{Y-16} + Y \times T$, where S is the number of expanded prefixes between level 16 and 24 for $Y = 24$ and between level 25 and 32 for $Y = 32$. This scheme may require one extra memory-access time if the longest prefix match is found on level 24 or 32 to find the memory location for the next-hop information. Each port number is cosidered to be stored in 15-bit fields for large routers.

III. Implementation

To allow a parallel IP lookup process at the target levels, the implementation of our scheme uses separate memory blocks. Specialized hardware can be used to execute the search and match process, and the building of the tree structure with prefix data. The specialized hardware are blocks where address comparisons and other processes are performed, and a control logic to sequence the memory access and data flow.

Since the dominant term that determines the maximum processing time is the memory access, the following memory partitioning is used. The bit vectors and bit segments are stored in a memory block per level. Figure 6 shows the memory blocks for each bitmap and next-hop tables. The table for level 8 (*tableNext8*) stores the next-hop information. Therefore, *prefixVal8* and *tableNext8* can be accessed at the same time. Tables for level 16, 24, and 32 have a number of locations proportional to the number of entries for level 16, and to a number of intervals, with 256 and 65536 entries per interval, for level 24 and 32, respectively.

IV. Search Procedure

Consider the destination address x of a packet in transit, which can be represented in binary as $x_{31}, ..., x_0$, where x_{31} is the most significant bit. During the first memory-access time, the following fields are accessed: *prefixVal16*, *childVal24*, *childVal32*, *offsetVal16*, *offsetVal24*, and *offsetVal32* with bits $x_{31}, \ldots, x_{16}$. In addition, *prefixVal8* and *tableNext8* are accessed, however, with bits $x_{31}, \ldots, x_{24}$.

During the second memory-access time, the following fields are accessed: *prefixVal24*, using $x_{23}, \ldots, x_{16}$, of the interval indicated by the value stored in *offsetVal24* plus the number of ones on the left of bit *childVal24* in the bit chunk, if the bit corresponding to the root *childVal24* was found to be set in the first memory-access time. The same is done for

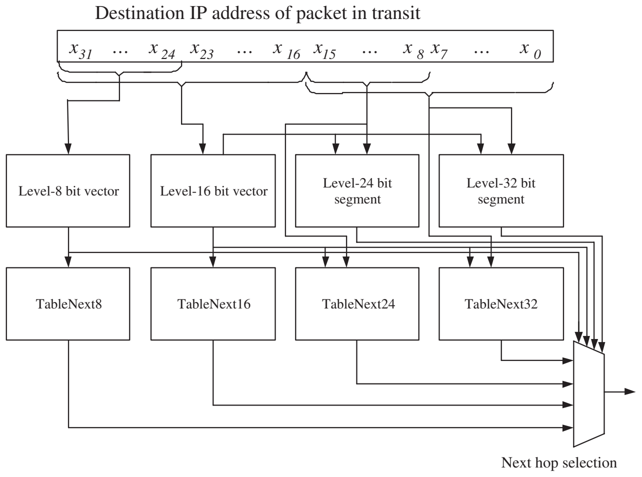

Fig. 6. Separate blocks of memory for independent access.

prefixVal32, using *childVal32* and *offsetVal32*. At the same time, *tableNext16*, *tableNext24*, and *tableNext32* are accessed.

During the second memory-access time, the combined results of the fields *prefixVal8, prefixVal16, childVal24*, and *childVal32* are considered to determine which level has a possible matching prefix, or candidate levels, according to the values as shown by Table III. Note that a match at level 16 is confirmed after the first memory-access time. The retrieved next-hop information from those *tableNextY* that are considered candidates are kept.

After the second memory-access time, the next-hop information of the longest prefix value is selected according to the result of *prefixValueY*, where $Y = \{8, 16, 24, 32\}$. If memory reduction scheme is used in storing the next-hop information, one more memory-access time may be needed to find the location for the next-hop information according to the values of *portIntervalY* and *offsetPortY*.

A. Determining the Longest Prefix Matching

The search starts by looking at the bit indicated by x_{31}, ..., x_{16}, or bit I which is the decimal value of binary (x_{31}, ..., x_{16}) in level 16, or $prefixVal16$. According to Table III, the remaining of the search procedure follows according to the case number based on the values of $prefixVal8$, $childVal24$, and $childVal32$.

Case 1. Only the segment $prefixVal8$ in level 8 is matched. The bit of the variables shown in this case of Table III, other than $prefixVal8$ are 0. The bit I of level 8, denoted by the binary equivalent of bits x_{31}, ..., x_{24} is checked. If bit I of $prefixVal8$ is 1, the location of the next hop in the $tableNext8$ is indexed by the sum of ones to the left of node I of level 8. The next hop can be retrieved from the table. If bit I is 0, the default next hop value is used.

Case 2. Search for a prefix in level 8 is performed as indicated in Case 1. As the other possibility is to find a match in level 32, the pointer p to the memory location $mem32$, where $prefixVal32$ is stored, is indicated by $offsetVal32$ plus the number of ones in the $childVal32$ chunk found to the

case	$prefixVal16$	$childVal24$	$childVal32$	match level
1	0	0	0	8
2	0	0	1	8, 32
3	0	1	0	8, 24
4	0	1	1	8, 24 and 32
5	1	0	0	16$*$
6	1	0	1	16$*$ and 32
7	1	1	0	16$*$ and 24
8	1	1	1	16$*$, 24, and 32

TABLE III
POSSIBILITIES OF MATCHING PREFIXVAL BITS AT DIFFERENT LEVELS.

left of bit I. At the same time, the next 16 bits from address x, say x_{15}, ..., x_0, whose binary equivalent is denoted as j, indicate the position of next hop location in $tableNext32$ for level 32. If the j^{th} bit of $prefixVal32$ is 1, then a match is found and the next hop for level 32 is used. If the value at the j^{th} bit is 0 and there is a prefix match found at the 8^{th} level, then the search for match in level 8 is used. If no match is achieved, the default next hop is used instead.

Case 3. Search for a prefix in level 8 is performed as indicated in Case 1. The other possibility is a match in level 24. In this case, the pointer p to the memory storing $prefixVal24$ is indicated by $offsetVal24$ plus the number of 1s in the $childVal24$ chunk found to the left of bit I. At the same time, the following 8 bits of x, x_{15}, ..., x_8, whose binary equivalent is denoted as j, are used to indicate the position of the next hop entry number in $tableNext24$ for the segment of level 24. If the value of the j^{th} bit of $prefixVal24$ is 1, then a match is found and the next hop for level 24 is used. If the value at the j^{th} bit is 0 and if the matching prefix found at the 8^{th} level is matched, the next hop information for level 8 is extracted from $tableNext8$. Otherwise, the default next hop information is used instead.

Case 4. The search in levels 8, 24, and 32 are performed as indicated in Cases 1, 2, and 3.

Case 5. In this case, as the match in level 16 is found and no possible match in levels 24 and 32, the only remaining procedure is to find the next hop information. To find the pointer to the location where the next hop information is stored, $offsetVal16$ of the chunk where I is located is added to the number of ones at and to the left of bit I within the chunk. The resulted value is the offset value used in the portion of $tableNext16$ corresponding to prefix of length 16 indicating the next hop information.

Case 6. A search for a prefix in level 32 is performed with similar procedure as for Case 2. If no match in achieved at that level, the prefix match in level 16, as mentioned in Case 5, is performed.

Case 7. A search for level 24 is performed according to the procedure mentioned in Case 3. If no match is achieved at that level, the prefix match in level 16 is used.

Case 8. In this case, levels 24 and 32 levels are considered. The longest prefix in levels 32, 24, or 16 is used.

V. COMPLEXITY AND PERFORMANCE

Matchings at level 8 are resolved in a single memory-access time, and matching at levels 16, 24, and 32 are resolved in two memory-access times. Using the memory reduction scheme for the next-hop information may require up three memory-access times.

We tested our scheme with a recent routing table AS65000 (August 1, 2007) [6], which has 82835 entries and average prefix length of 22. The number of segments for level 24 is 6305 and 82 for level 32. The total amount of memory required by this scheme is 1.6 Mbytes with reduction of memory for next hop information and 10 Mbytes without memory reduction for next hop information.

VI. CONCLUSIONS

We proposed a trie-based IP lookup algorithm that performs parallel search for the longest prefix. Controlled prefix expansion is used to reduce the number of different prefix lengths or levels, and separate memory blocks to reduce the number of memory-access times. As a result, the proposed scheme finds the longest match in up to two memory-access times or three memory-access times if memory reduction scheme is used in storing the next hop information.

As an example, we selected four prefix levels (prefix lengths): 8, 16, 24, and 32. The algorithm searches for a match in level 16 and 8 at the first memory access. Then it verifies possible matches at levels 24 and 32, and retrieves all possible next hops, one per level, in the second memory access. If matches are achieved at different levels, the match belonging to the longest prefix is selected. This results in a maximum of two memory-access times for the lookup process. We introduced memory reduction scheme for next hop information which reduces the total amount of memory by at least six times in our example. However, this memory reduction mechanism may add one more memory-access time.

This scheme uses a feasible amount of memory for large forwarding tables that make a heavy use of CIDR. An additional advantage of using separate memories is the feasibility for pipelining the search process, thus achieving address lookup in a single memory-access time.

REFERENCES

[1] D. R. Morrison, "PATRICIA - Practical Algorithm to Retrieve Information Coded In Alphanumeric," Journal of the ACM, 15(4), pp. 514-534, October 1968.

[2] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," Proc. ACM SIGCOMM, pp.3-14, 1997.

[3] D. E. Taylor, J. S. Turner, J. W. Lockwood, T. S. Sproull, and D. B. Parlour, "Scalable IP Lookup for Internet Routers," IEEE J. of Select. Areas in Commun., Vol. 21, Issue 4 , pp. 522-534, May 2003.

[4] N-F. Huang, S-M. Zhao, J-Y. Pan, and C-A. Su, "A Fast IP Routing Lookup Scheme for Gigabit Switching Routers," Proc. IEEE Infocom 1999, Vol. 3, pp. 1429-1436, March 1999.

[5] V. Srinivasan and G. Varghese, "Faster IP lookups using controlled prefix expansion," ACM Trans. Comput. Syst., pp. 1-40, Feb. 1999.

[6] BGP Table Data, http://bgp.potaroo.net.

[7] H. Lim, J-H Seo, and Y. Jung, "High Speed IP Address Lookup Architecture using Hashing," IEEE Commun. Letters, vol. 7, No. 10. October 2003.