Fast Table-Update Scheme and Implementation of a Trie-based Scheme for Fast IP Lookup

Roberto Rojas-Cessa, Lakshmi Ramesh, Ziqian Dong, and Lin Cai
Department of Electrical and Computer Engineering,
New Jersey Institute of Technology,
University Heights, Newark, NJ 07102, USA.

***Abstract*—As data rates in the Internet increase, routers are required to perform Internet Protocol (IP) address lookup in shorter resolution times. IP address lookup involves finding the longest matching prefix from the list of prefixes that matches the destination address of a packet. In this paper, we discuss the implementation of our proposed parallel-search IP lookup scheme and introduce a mechanism for fast table update. Our scheme is uses random access memory (RAM) to store a forwarding table in a trie form. Our proposed algorithm has the options to perform lookup in either two or three memory-access times depending on the memory savings desired. The proposed update mechanism has an execution complexity of $O(1)$, which improves the complexity of $O(N)$ in the original scheme, where N is the number of groups indicating prefix values.**

***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 required memory amount low 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.

An alternative to the TCAM approach is a trie-based scheme that uses random access memory (RAM). In a trie-based scheme, a binary tree represents all combinations existing in the forwarding table of a router. In this approach, the worst-case search 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 small memory and up to 3 memory-access times.

In this paper, we describe the implementation of our previously proposed trie-based IP-lookup scheme [6], [7], which performs parallel search of the matching longest prefix, and propose a new table update scheme to reduce the number of memory access times that can be triggered by prefix modifications. To reduce the search complexity, the parallel-search lookup scheme uses controlled prefix expansion [5]. This lookup scheme finds the longest prefix match in a maximum of two memory-access times. With a small modification on the data structure, the parallel-search scheme can compact memory further at the cost of performing the lookup process in three memory access times. With the new table update scheme, we reduce the memory access complexity from $O(N)$, where N is the number of memory blocks at the reference prefix level, to $O(1)$. The presented algorithm is flexible for routing tables with diverse prefix length distributions.

The remainder of the paper is organized as follows. Section II describes the data structures used and the components of the proposed scheme. Section III describes the lookup procedure of our scheme. Section IV describes the implementation of this scheme. Section V introduces our proposed table update mechanism. Section VI discusses the complexity and performance. Section VII 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 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 forwarding table using CIDR. Figure 1 shows the CIDR entries of the example table presented 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 bottom. Our scheme considers that each level can be searched in parallel. To decrease the number of parallel searches, the number of levels (with prefixes) is minimized, into a small number of target levels. To minimize the number

Roberto Rojas-Cessa is the corresponding author. Email: rrojas@njit.edu.

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
EXAMPLE OF A FORWARDING TABLE.

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

TABLE II
EXPANDED FORWARDING TABLE.

of levels, we use controlled prefix expansion [5]. The target levels can be selected by using the most populated levels of an actual 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 our example to describe the proposed scheme without losing generality.

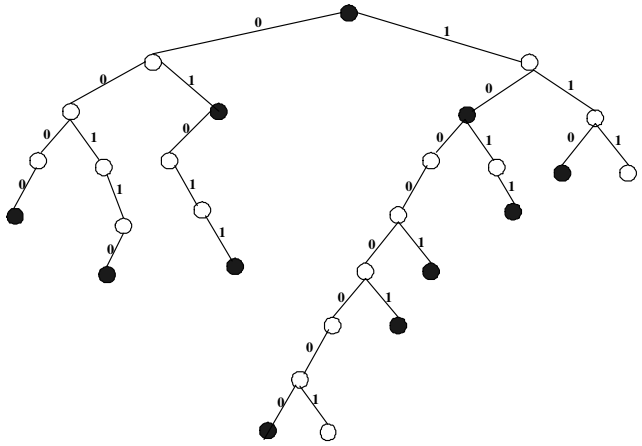

Fig. 1. Binary tree representing the forwarding table.

Figure 2 shows the prefix expansion for levels 2, 5, and 8, as target levels in our example and Table II shows the resulting expanded prefixes that result from Table I or Figure 1.

To reduce the number of distinct prefix lengths to localize the lookup search levels, the original set of prefixes with length X is expanded to a set of predefined prefixes with length Y, where $Y < X$.

In order to do the prefix expansion, we need to select all the prefixes whose lengths do not match the allowed 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, respectively. Both expanded prefixes inherit the next hop number of the original prefix. This can be seen from the repeating next hop pointer values in column 3 of this table. When one of the expanded prefixes is the same as one of the existing original prefixes, the existing entry prevails to avoid duplicate entries in the routing table. Therefore, the original prefix overrides the expanded prefix and only the original prefix is stored.

This algorithm states that the expansion lengths must be chosen in such a way that the majority of the prefixes are at that length such that a small number of original prefixes are to be expanded. Using the contents of actual routing tables [8], we found that a large number of the prefixes are found between levels 16 and 24. Considering that population, we selected tree levels (or prefix lengths) 8, 16, 24, and 32 as the target levels in our scheme. This selection can be changed according to the prefix-length distribution of routing tables.

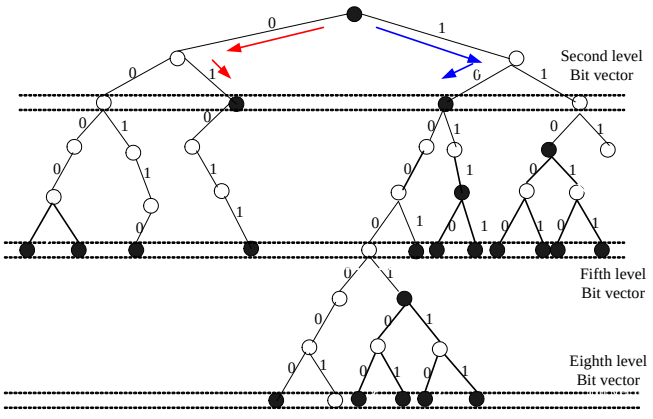

Fig. 2. Bit vectors and stored prefixes in extended-prefix tree.

A. Data Structures at Target Tree 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 the right-most bit corresponds to the decimal 2^{level}-1, where $level$ is the level number. The bitmaps of levels 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. The bitmaps for levels 24 and 32 are called bit segments as only partial bit vectors containing one or more prefixes are used. The data structure in each level carries the following information:

Level-8 bit vector indicates the existence of all the prefixes between levels 1 and 7, which are expanded to level 8, including level-8 prefixes. The level-8 bit vector, denoted as *prefixval8*, has 256 bits. Each bit represents a 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.

Level-16 bit vector indicates the existence of all the prefixes between levels 9 and 15, which are expanded to level 16, including 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 shares each 16-bit combination indexed by *prefixval16*, and *childval32*, which indicates whether there is one or more prefixes with a length between levels 25 and 32 that shares 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. Therefore, *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 these three offset fields is 16 bits each. Figure 3 depicts the data structure for level 16.

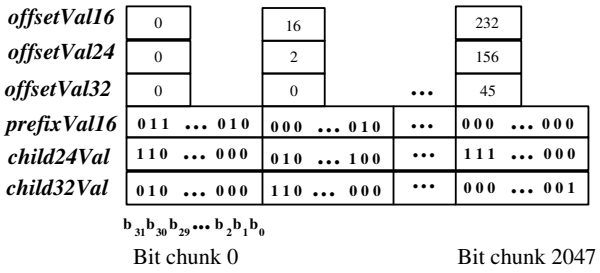

Fig. 3. Bit vector at level 16.

Level-24 bit segment carries the 256-bit intervals of the level-24 bitmap that correspond to the subtrees rooted by *prefix16*, which have one or more stored prefixes located between levels 17 and 24. Those intervals are stored in a pseudo-continuous way to reduce memory amount. This bit segment is denoted as *prefixval24*. The sum of *offsetval24* and the number of ones to the left the *childval24* bit in the chunk is used to find the corresponding interval at level 24.

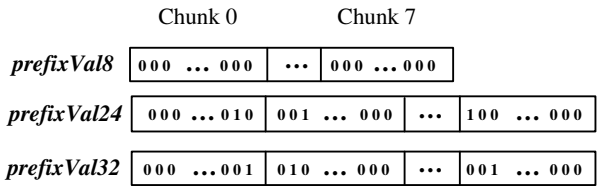

Fig. 4. Bit vectors at level 8, 24, and 32.

case	$prefixval16$	$child24val$	$child32val$	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
REFERENCE TABLE OF FLAG BITS ON BIT VECTORS TO DETERMINE THE LONGEST MATCH.

Level-32 bit segment carries those 2^{16}-bit intervals at level 32, which correspond to the subtrees rooted by *prefix16*, with one or more stored prefixes between levels 25 and 32. This segment bitmap, denoted as *prefixval32*, is used in the same way as level-24 bit segment.

The **next-hop information** for prefixes in each level is stored in several tables, one table per level, called $tablenextY$, where $Y = \{16, 24, 32\}$. These tables store the next-hop information for each bit in the bit vectors and bit segments.

III. 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}, ... , x_{16}. In addition, *prefixval8* and *tablenext8* are accessed, however, with bits x_{31}, ..., x_{24}.

During the second memory-access time, the following fields are accessed: *prefixval24*, using x_{23}, ..., 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. The same is done for *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 Table III. Note that a match at level 16 is confirmed after the first memory-access time. The retrieved next-hop values 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 $Y = \{8, 16, 24, 32\}$.

IV. Implementation

The bit vectors and bit segments are stored in a memory block per level. Figure 5 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 levels 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 levels 24 and 32, respectively. In the remainder of this paper, we assume

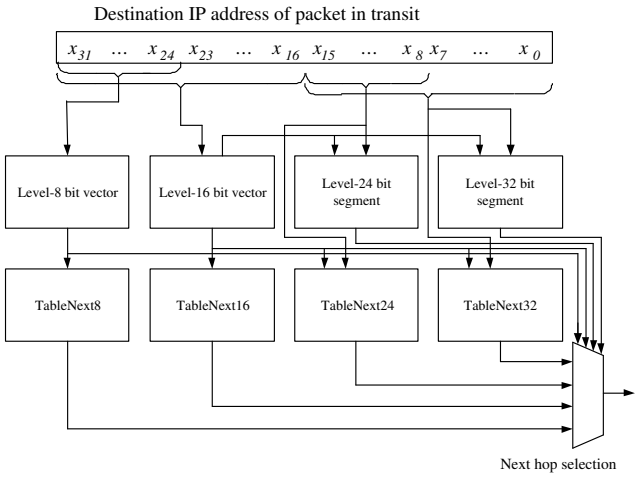

Fig. 5. Blocks of memory tables for parallel search IP lookup.

that tables, bit vectors, and bit segments are stored in separate memory blocks.

V. Table update

In the previously proposed scheme, prefixes are denoted by single bits. Offset values at level 16, used for search on bitmaps at other levels, keep a cumulative count of *prefixvalue* bits. If there is a modification in the routing table that impacts the forwarding table, such as prefix addition or removal, the offset values need to be updated. The worse case scenario occurs when the first prefix bit is affected, then all offset values in level 16 need to be modified accordingly. This makes the update complexity be described as $O(N)$, where N is the number of offset values.

Our new table update mechanism removes the cumulative property of the previous scheme for storing the offset values. Instead of using an offset value that depends on the existing prefixes to calculate the position of the bit maps of other levels, the new scheme assigns a value related to a position in memory in an independent and arbitrary fashion. Figure 6 shows an example of the new data structure at level 16. In the new scheme, the data structure at level 16 is $prefixval16$, and instead of $offsetval24$ and $offsetval32$, the new values $indexval24$ and $indexval32$ are used.

Considering that memory for storing $prefixval24$ and $prefixval32$ holds segments of the bit vectors at these levels and that these segments may not be contiguous, memory is partitioned into blocks (holding a set of prefixval bits) and their location can be arbitrarily assigned. Therefore, $Indexval24$ holds the memory location (or a reference number) for the block of $prefixval24$ bits associated with $prefixval6$ as described in the original scheme. This value also indicates if there is one or more prefixes in this interval, and indicating with 0 the lack of prefixes. $Indexval32$ does similar function as $Indexval24$ but for prefixes at level 32.

Since memory needs to be allocated in blocks and modifications may occur to different chunks (also denoted as $indexval$), there is a list of possible blocks that can be used. To assign idle blocks, a queue with first-in first-out (FIFO) service is used. The FIFO for blocks of levels 24 and 32 are called $SeedFIFO24$ and $SeedFIFO32$, respectively. and This every time a new segment comes for an addition to $childval$ in levels 24 or 32, a new block number is provided by this FIFO. If some segment are released from memory (i.e., the only $childval$ bit is reset), the memory block is released and considered as idle. This block number goes back to the FIFO.

This new scheme has a complexity for updating a forwarding table of $O(1)$. However, the associated cost is more memory usage. Nevertheless, we consider that the number of memory accesses has higher cost than memory amount.

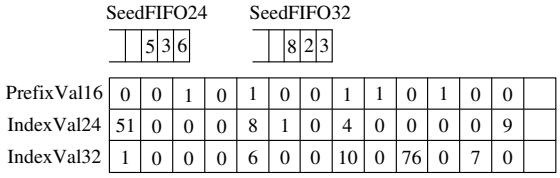

Fig. 6. Proposed data structure at level 16 for fast table update.

VI. 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.

We used an actual routing table (AS65000, August 1, 2007) [8], with 82835 active entries and an average prefix length of 22 to estimate the memory usage. The original scheme (without fast table update) uses 1277904 bits at level 16, 25825280 bits are level 24, and 10485760 bits at level 32. The total amount of memory is 4699130 bytes, with up to two memory accesses.

The memory used with the fast table update mechanism increases the memory amount to 3159728 bits for level 16, the memory at levels 24 and 32 remain the same as the only affected fields are the original $offsetvals$ parameters that as substituted by $indexval$. The total memory amount used is then 4934358 bytes, which is slightly higher than the original scheme.

In the original scheme, a modified prefixval bit can make the $offsetval$ value change. Since the value $offsetval(N+1)$, where n is an arbitrary chunk number, depends the sum of $offsetval(N)$ plus the number of ones in the chunk N or $offsetval(N)$, then all values of the $offsetval$s after the modified chunk n are affected. This gives an update complexity of $O(N)$. The scheme proposed here, instead of

counting the $childval$ bits, it stores a memory location for each segment block at levels 24 and 32, therefore, $indexval(N+1)$ has an independent value to $indexval(N)$. Therefore, the update of $indexval(N)$ does not affect any other $indexval$ field. This new scheme has a complexity of $O(1)$.

VII. Conclusions

We proposed a scheme for minimum memory access time in table updates. A new table update affects needs a single memory access time. This novel table update mechanism is used in our previously proposed trie-based IP lookup algorithm that performs parallel search for the longest prefix. The IP lookup scheme finds the longest match in up to two memory-access times without optimizing memory use for next hop information, and up to three memory access times with compact next-hop information. The new table update scheme has no impact in the table update search and optimizes the table updates.

We also discussed some of the issues for the implementation of the IP lookup scheme and the proposed table update scheme.

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," 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," IEEE INFOCOM '99, 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] R. Rojas-Cessa, L. Ramesh, Z. Dong, L. Cai, and N. Ansari, "Fast Parallel-Search Trie-Based IP Lookup Scheme," Proc. IEEE Globecom 2007, 5 pages, Washington, DC, 26-30 November, 2007.

[7] R. Rojas-Cessa, L. Ramesh, Z. Dong, B. D'Alessandro, and N. Ansari, "Implementation of a Parallel-Search Trie-Based Scheme for Fast IP Lookup, Proc. IASTED International Conference on Communication Systems, Networks and Applications (CSNA 2007), 3 pages, Beijing, China, October 2007.

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

[9] P. Gupta, S. Lin, and N. McKeown, "Routing Lookups in Hardware at Memory Access Speeds," IEEE Infocom 1998, vol. 3. pp. 1240-1247, 1998.

[10] 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.

[11] T. Wolf and J.S. Turner, "Design issues for high-performance active routers," IEEE J. on Select. Areas on Commun., Vol. 19, Issue 3, pp:404 - 409, March 2001.

[12] T. Wolf, S. You, and R. Ramaswamy, "Transparent TCP acceleration through network processing," Proc. IEEE Globecom 2005, Vol. 2, 5 pp, November 2005.