A Small IP Forwarding Table Using Hashing

Yeim-Kuan Chang and Wen-Hsin Cheng

Department of Computer Science and Information Engineering
National Cheng Kung University
Tainan, Taiwan R.O.C.
ykchang@mail.ncku.edu.tw

Abstract—**As the demand for high bandwidth on the Internet increases, it is required to build next generation routers with the capability of forwarding multiple millions of packets per second. Reducing the required memory size of the forwarding table is a possible solution since small forward table can be integrated into the application specific integrated circuit (ASIC). In this paper a hash technique is developed to make the IP forwarding table as small as possible. The experiments show that the required memory size of the proposed scheme is smaller than other existing schemes for a large routing table.**

Keywords—**Hash table, IP lookup, and binary trie.**

I. INTRODUCTION

The exponential traffic rate due to the advent of World Wide Web (WWW) demands for high bandwidth on the Internet. Backbone routers with gigabit links, such as OC-192, 10 Gigabits and OC-768, 40 Gigabits, are not uncommon. Among all the tasks performed by the routers, the packet forwarding is the most critical one that must be able to keep up with the link speed and router bandwidth. These backbone routers have to forward millions of packets per second at each port.

The classless IP subnet scheme called Classless Inter-Domain Routing (CIDR) evolves from the scarcity of IPv4 addresses. With CIDR, the routing entry (or called prefix) in a routing table could have arbitrary length ranging from 1 to 32 bits, instead of 8, 16, 24 bits in Classful Address scheme. Therefore, the IP lookup problem becomes a longest prefix matching problem, also called Best Matching Prefix (BMP) problem because of there may be more than one prefix that matches the target IP address. Figure 1 shows the distribution of prefix lengths for a typical large routing table available on the Internet [12]. This routing table will be used in the performance experiments in this paper.

The routing table in a router that is used to lookup an IP address stores an array of entries, each consisting of a network address that is the prefix of a group of IP addresses and the corresponding next 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.

A large variety of routing lookup algorithms were classified and their worst-case complexities of lookup latency, update time, and storage usage are compared in

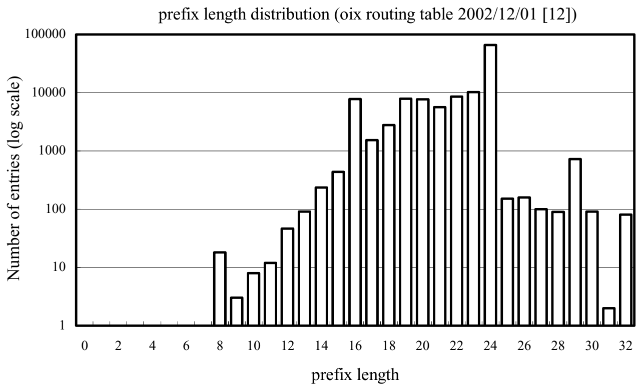

Figure 1: Distribution of prefix lengths for oix routing table.

[1]. Among them, a category of algorithms is based on a trie/tree structure. The binary trie is the basic data structure used in most of IP lookup algorithms. The binary trie is in fact a binary search tree using the bit value (0 or 1) to guide the search to the left or the right part of the tree. However, the binary tree structure must be implemented using linked list data structure. Each trie node has the left and right pointers pointing to its left and right sub-tree, respectively.

Based upon this primitive trie structure, a set of prefix compression and transformation techniques are used to either make the whole data structure small enough to fit in a cache [9], or to transform the set of original prefixes to a different one in order to speed up the tree traversal procedure [4, 13]. In [7], the binary search on prefix lengths is proposed. The worst-case lookup complexity is the best among all the existing schemes, however, with the assumption that we can use perfect hash tables to lookup a possible prefix match on the set of prefixes in one step. The hardware based lookup algorithms using multi-bit trie proposed in [3, 11] is in fact a variation of the prefix transformation techniques. The extreme case is a 32-bit extended trie that 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.

In [9], a small forwarding table (SFT) is proposed for IP lookups. The SFT scheme is based on a run length encoding technique to encode an array of disjoint prefixes. The array of disjoint prefixes is constructed by converting the binary trie to a complete binary tree. The detailed data structure can be illustrated in Figure 2,

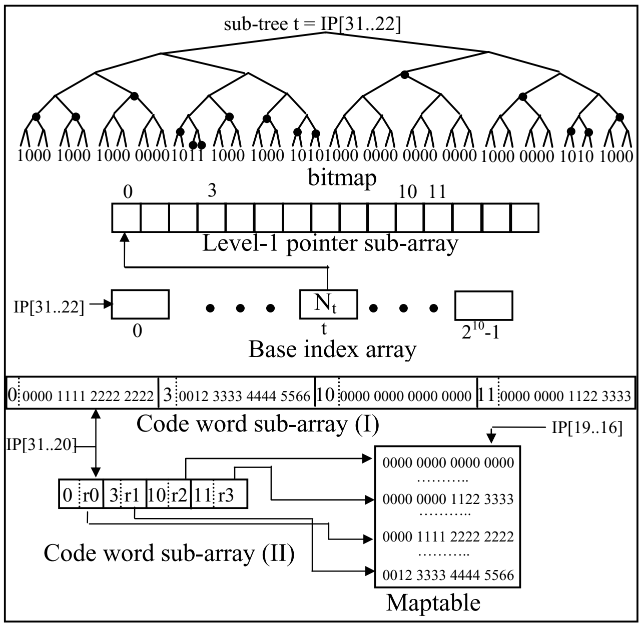

Figure 2: Data structure for SFT forwarding table.

where a 6-bit sub-trie is shown. The level-1 pointer sub-array corresponds to the fifteen 1's (denoted as heads) in the bitmap shown in the figure. Out of these 15 heads, 13 heads record the prefixes of length less than or equal to 16 and two heads (the fourth and fifth) indicate there exist sub-tries underneath them. The index N_t of the level-1 pointer sub-array for this 6-bit sub-trie is stored in the t^{th} position of the base index array. The pointer sub-array starting at N_t contains 15 pointers corresponding to this 6-bit sub-trie. The code word array (I) which is the run-length encoding of the bitmap illustrates how the maptable is built. Four parts of 16 numbers in the code word array (I) shown in a smaller font indicate the relative positions of the corresponding heads in the pointer sub-array of this 6-level sub-trie. Since the possible combinations of 16-number lists is limited to 678 and the patterns of the 16-number lists repeat, all the 16-number lists are sorted and stored in the *maptable*, a two-dimensional array. Thus the code word array (I) is rewritten to be (II). Although the memory usage is reduced because of maptable, one more memory access to the maptable is also incurred.

Based the data structure shown in Figure 2, the lookup operations work as follows. Firstly, IP[31..22] is used to index the base index array and N_t is obtained. N_t is the starting index of level-1 pointer sub-array for the current 6-bit trie. Next, IP[31..20] is used to index the code word array (II) to get a pair of numbers which may be 0/r0, 3/r1, 10/r2, or 11/r3 in this example. For example, if IP[21..20] = 01 then the code word with 3/r1 will be used. The index r1 is for maptable. Thus, through r1, we get a 16-number list of 0012333344445566 each of which is referenced by IP[19..16]. In summary, if IP[21..16] is 010011, the index of level-1 pointer array will be $N_t + 3 + 2$. The element of the level-1 pointer array at index $N_t + 3 + 2$ will be referenced to see if a match is found.

Notice that only the first 16 levels of the complete binary trie are examined. We need the structure for the next 16 levels. The structure of the next 16 levels is similar, but a little modification is made with the memory reduction in mind, where sparse, dense, and very dense subtrie of 8 levels are distinguished.

```
GNode{
   Base ptr: index of blocks (20 bits);
   Left, Right: index of LNodes (k+1)*2 bits;
   N: number of LNodes (k+1 bits);
   M: number of GNodes (k bits);
   P: port number (8 bits);
}
LNode{
   Left, Right: index of LNodes or GNodes (k+1 bits each);
   flagL, flagR: for indicating if Left and Right point to a
                 Gnode or a LNode (one bit each);
   P: port number (8 bits);
}
```

Figure 3: Data structure for global and local nodes in the binary trie.

In [3], another run-length encoding scheme is proposed. To reduce the number of memory accesses, a straightforward 16-bit segmentation table is utilized to cover the first 16 levels of the trie. The next 16 levels of the trie are constructed by a compressed next hop array and a compressed bit map. The advantage of the proposed scheme is the maximum number of memory accesses that is 3. However, the memory usage is still high for a large routing table.

In this paper, we shall propose a new method that employs a novel hashing technique to avoid wasting the unused space in the multi-bit trie. For 4-bit sub-tries, a near-minimal hash function can be constructed. The 4-bit hashing tables are used as the building blocks to build the whole routing table recursively. We will show that the size of the proposed routing table is the smallest among all the existing schemes.

The rest of the paper is as follows. Section 2 first shows a simple mechanism to reduce the memory usage of the binary trie. Then the basic idea of the hash function is illustrated. Based on the hash function, a 8-8-8-8 hierarchical routing table is proposed. Performance comparisons using real routing tables are presented in Section 3. Finally, a concluding remark is given in the last section.

II. PROPOSED DATA STRUCTURE

The binary trie is the simplest data structure for the IP lookup problem. The primary disadvantage is its worst case search time for an IP lookup which is 32 and 128 memory references for IPv4 and IPv6, respectively. As indicated in [14], the more controlled expanded levels are used, the less memory is required for the routing table. Therefore, we conjecture that the binary trie can be optimized in term of memory storage. By carefully inspecting how the node space in the binary trie is utilized, it is easy to know that the reason for high memory usage in binary trie is the space for the pointers. In short, the space overhead for storing left and right pointers is large.

A naïve node implementation in the binary trie is to use memory addresses as pointers. In a 32-bit address space, these pointers in the trie node are of 32 bits. Therefore, a trie node needs eight bytes for left and right pointers plus one byte for next port number, a total of 9 bytes. As in LC trie, using local indices of a large pre-allocated node array may reduce the pointer size.

Consider an example routing table of more than 100K entries we use in this paper, there are around 370K trie nodes if these routing entries are organized in a binary trie. Therefore, if all the trie nodes are organized in an array, we can use the node indices of the array as pointers. In other words, the trie nodes are physically stored in a sequential array but are logically structured in the binary trie. Assuming there are no more than one million trie nodes, 20 bits are sufficient for a pointer in the trie node. Thus, 40 bits are required for two pointers plus 8 bits for the next port number. A total of 6 bytes is required for a trie node. By a simple calculation for a binary trie of 370K nodes, the total memory requirement is around 2.2 Mbytes.

k	# of global nodes	# of local nodes	# of nonprefix node	Memory usage (Kbyes)
1	347771	0	0	1,244
2	104502	243269	132742	963
3	64270	283501	172974	945
4	37975	309796	199269	951
5	36642	311129	200602	1,002
6	19738	328033	217506	1,029
7	41243	306528	196001	1,204
8	7758	340013	229486	1,153
Original binary trie with 20-bit pointers				2,038

Table 1: Memory usage for growing subtrie of k levels.

We can see that there are still two 20-bit pointers needed in a node that are in fact the global pointers. The node size can be reduced by using only one global pointer. If both left and right children exist, we restrict them to locate next to each other in the sequential array. Thus additional two bits are used to indicate whether left and right children exist. The node size is thus reduced from 48 bits to 30 bits. Notice that the number of trie nodes remains the same. Thus, for the binary trie of 370K trie nodes, the total memory requirement is now 1.4 Mbytes. Notice that the LC trie [6] has used this idea. We do not compare the proposed scheme with LC trie until Section 3.

We generalize the above idea as follows. The nodes in a subtree of *k* levels are grouped into a block such that accessing any node in the block is based on the local index of the node inside the block. Figure 3 shows the generic data structure for global and local nodes (GNode and LNode). The global node and local node take 31+4k bits and 12+2k bits, respectively. The number N is of size one bit more than M because every node in the block can be local node and only the nodes in the bottom of k levels can be global node. The number of local nodes increases when k increases. What is the good choice for k in order to have minimum memory requirement for building the binary trie? It depends on how the binary trie is structured. The following table

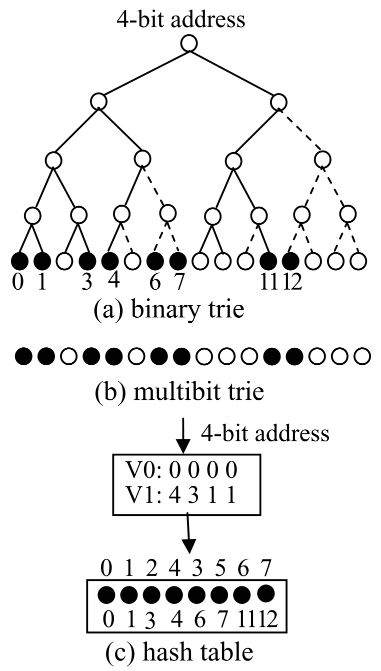

Figure 4: Binary trie, multibit trie, and hash table.

shows the memory requirements with different k using a large oix table [12]. We can see that the best value for k is 3. In fact, k = 2 or 4 are also very good. The table also shows the number of internal nodes that are not prefixes, called non-prefix nodes. These non-prefix nodes are the main cause for high memory usage. In next section, we will introduce a hashing technique that tries to remove these non-prefix nodes to save space. Although multibit techniques also remove the non-prefix nodes, they are in fact a special case of our hashing technique. The main drawback of the multibit techniques is that there are a lot of used memory slots in the multibit array.

Hash function. Assume that there is a set of *m* keys, *S = {k₀, …, k_{m-1}}*; each key is an n-bit integer. We like to find a mapping called the *perfect hash function* such that each key is mapped into a unique number in range 0 to *H_Size – 1*. If m = H_Size, this perfect hash function is minimal. Finding a perfect hash function is easy if we can support a memory array of size 2^n elements. The hashed number of a key is equal to the value of the key. However, there will be $2^n - m$ unused elements. When 2^n is much greater than m, it is a large waste of memory space. Finding a minimal perfect hash function is difficult. In this section, we will propose a mechanism that allows us to find a near-minimal perfect hash function. The form of the proposed hash function, **H**, for an n-bit number ($b_{n-1} \ldots b_0$) is formulated as follows.

$$H(b_{n-1} \ldots b_0) = \sum_{i=0}^{i=n-1} \left| V_{b_i}[i] \right|, \qquad (1)$$

The absolute value of x is denoted as $|x|$. V_0 and V_1 are two pre-computed arrays of size n in which either of the elements $V_0[i]$ or $V_1[i]$ is zero and the values of the non-zero elements are in the range of – MinSize to

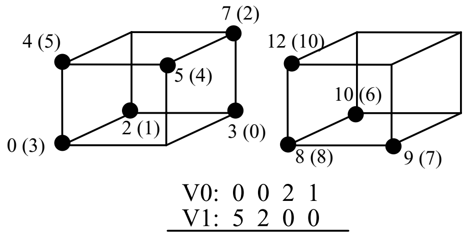

Figure 5: Hash tables of 4-bit addresses.

MinSize for *MinSize* = min(H_Size – 1, 2^{n-1}).Consider a list of eight 4-bit numbers, 0000, 0001, 0011, 0100, 0110, 0111, 1011, and 1100. We can construct arrays V_0 and V_1 as V_0 = [0,0,0,0] and V_1 = [4,3,1,1]. Based on the above hash function, we have the following results: H(0000) = 0, H(0001) = 1, H(0010) = 2, H(0011) = 3, H(0100) = 4, H(1000) = 5, H(1001) = 6, H(1011) = 7. Compared to multibit trie, the hashing technique can be illustrated in Figure 4. The 4-bit trie from an original binary trie is shown in Figure 4(b). Eight slots in the multibit array are unused. The trie using hash table is shown in Figure 4(c). The index of the prefix array can be computed by using the pre-computed hash tables. The details of computing arrays V_0 and V_1 are given as follows.

Computing V_0[n-1 … 0] and V_1[n-1 … 0]: The construction algorithm for arrays V_0 and V_1 is based on a form of exhaustive search. Briefly, for each cell in arrays V_0 and V_1, a number in range – MinSize to MinSize is tried one at a time until the hash values of keys are unique. The construction algorithm involves three steps that are described as follows.

Step 1: sort the keys based on the frequencies of the occurrences of 0 or 1 starting from dimension 0 to n – 1. If the number of 1's is the same as that of 0's the order of the keys may keep unchanged. Now assume the keys are in the order of k_0, …, k_{m-1} after sorting. In the next two steps the keys are processed in this order.

Step 2: compute the cells in arrays V_0 and V_1 that the current key controls. The key, $b_{n-1}…b_0$, has the control on $V_{bi}[i]$ if $V_{bi}[i]$ is not yet controlled by one of preceding keys for i = n-1 to 0. For example, assume the first two keys in a 4-bit address space are 0000 and 0011. The first key control $V_0[3]$, $V_0[2]$, $V_0[1]$, $V_0[0]$. However, the second key only control $V_1[1]$, $V_1[0]$ because $V_0[3]$ and $V_0[2]$ are already controlled by the first key.

Step 3: Use the following rules to assign a number in range –MinSize to MinSize to each cell controlled by the current key. If the hash value, $H(b_{n-1}…b_0)$, of the key is taken by preceding keys or larger than MinSize or smaller than –MinSize, every cell must be re-assigned a new number. If no number can be found after exhausting all the possible numbers for the cells controlled by the key, we will backtrack to the previous key by re-assigning it a new number and continues the same procedure.

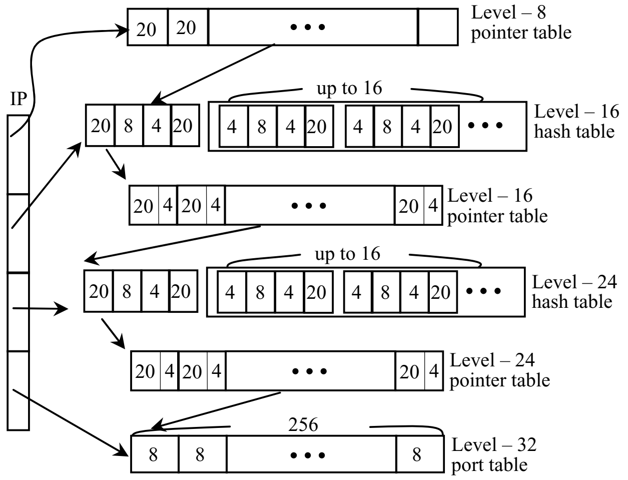

Figure 6: Data structure for the proposed 8-8-8-8 table.

Example: We use the keys in Figure 4 to demonstrate how the hash table is constructed. The numbers of 0's are then same as that of 1's at the bit positions of 0, 1, and 2. The order of the keys depends only on bit 3 and thus is in the order of 0000, 0001, 0011, 0100, 0110, 0111, 1011, and 1100. The construction of the hash table starts with the key 0000 which controls $V_0[3]$, $V_0[2]$, $V_0[1]$, $V_0[0]$ and we have $V_0[3]$=0, $V_0[2]$=0, $V_0[1]$=0, and $V_0[0]$=0. Next we consider key 0001 which controls only $V_1[0]$. To make current hash table minimal, $V_1[0]$ is set to 1. We have H(0000) = 0, H(0001) = 1. Now, the third key is 0011 that controls only $V_1[1]$ set to 1. Therefore, we have H(0011) = 2. By doing the same construction process, we have $V_1[2]$=3 because of key 0100 and H(0100)=3. It is lucky we have H(0110)=4 and H(0111)=5. Finally, we set $V_1[3]$=4 and have H(1011)=6 and H(1100)=7.

Analysis of the hash table. Since building a hash table uses the exhaustive search, it is not feasible for a large n. In this paper, we select the hash table of size n = 4 as the building block for creating a large routing table. We use the exhaustive search to check whether finding a minimal perfect hash function is possible for a set of N 4-bit numbers, where N = 1 to 16. We find out that the minimal perfect hash function exists for all cases except some rare cases when N = 10 or 11. The perfect hash function with the minimal hash size increased by one exists for these rare cases when N = 10 or 11. Figure 5 illustrates one of these rare cases whose minimal perfect hash functions do not exist. The perfect hash function of size 11 is shown for a list of 10 numbers. Value nine is the hashed value that is not used by any of these 10 numbers.

The 8-8-8-8 Routing table. The proposed data structure for the routing table will be based on the 4-bit hash tables as the building blocks. The data structure of the routing table is organized in a 4-level 8-8-8-8 hierarchy shown in Figure 6. Many other IP structures use a 16-bit segmentation table as the front-end lookup array [8, 11, 15]. The advantage of the 16-bit front-end lookup array is that only one memory reference is needed to reach half of the depth compared to the binary trie. However, assuming each element takes 24 bits, the 16-bit segmentation table needs 192 Kbytes. We can see that many elements in the table are unused or the pointers in the elements are unused.

Therefore, instead of the 16-bit table, we use an 8-bit pointer table as the front-end lookup array to avoid wasting memory. The reason we do not use hashing for this front-end array is we trade a little more space for access latency. Since this front-end 8-bit table is the only first level table, the memory wasted for unused slots is small and thus acceptable. Each element of the 8-bit pointer table is a 20-bit pointer pointing to the hash table of the next level subtrie or the port number if only prefix of length 8 exists. The hash table of the next level subtrie uses a data structure called *format H* block consisting of hash tables of 4-bit addresses that recursively hashes up to 16 hash tables of 4-bit addresses. The first part of *format H* block consists of a 20-bit global pointer pointing to the level-16 pointer table, an 8-bit default port number, a 4-bit number (denoted as *hash_cnt*) to record the number of sub-hash tables and a 20-bit hash table. The 20-bit hash table records $V_0[3\text{-}0]$ and $V_1[3\text{-}0]$. Since there are four non-zero numbers each of which needs 4 bits and one more bit to record either $V_0[i]$ or $V_1[i]$ is zero. The second part is an array of *hash_cnt* sub-hash tables each of which consists of a 4-bit prefix string, an 8-bit local index, a 4-bit counter to record the number of keys in the corresponding sub-hash table, and a 20-bit sub-hash table. The data structure in level 24 is the same as that in level 16. The level-16 and level-24 pointer tables consist of 4-bit prefix and 20-bit global pointer. We use a different data structure in level 32 since not so many routing entries have their lengths longer than 24. The level 32 structure is just an array of 256 port numbers.

Building and updating the 8-8-8-8 routing table. The proposed 8-8-8-8 routing table is similar to the 4-bit trie except the some internal nodes in each 4-bit trie are replaced by the recursive hash tables of 4-bit addresses. We propose to first build the 4-bit trie and then compute the corresponding hash tables and the associated pointer tables. When a prefix is deleted from or added in the routing table, the 4-bit trie is then updated and thus the corresponding part of the proposed 8-8-8-8 routing table can be changed accordingly. In order to avoid the worst-case computation time for some combination of 4-bit numbers as shown in Table 3, we pre-compute all the 64K (2^{16}) possible 4-bit hash tables which account for 128K bytes since each 4-bit hash table needs 16 bits. Now computing a 4-bit hash table becomes one memory reference to the corresponding 16 bits in the array of 64K entries. Thus, the updating process when deleting or inserting a prefix is as fast as the 4-bit tire with additional four pre-computed 4-bit hash tables needed to be modified, at most.

IP lookup. The IP lookup process is also simple. The level-8 pointer table is first referenced using the most significant 8 bits of the IP address. Then the next 8 bits of the IP address and the level-16 hash tables are used to compute the index of level-16 pointer table. The same process is performed for level-24 hash table. After reaching the level-24 pointer array, the last 8 bits of the IP address is used to reference the bottom port table, if needed. Based on the distribution of prefix lengths, 99.9% of the routing entries have the prefix length less than or equal to 24. Thus, the number of the memory references is from 1 to 8. The hash tables on level-16 are used to compute the index of level-16 pointer array for accessing the level-24 hashing tables if needed. The operations are similar for level-24 hash tables and pointer table. Notice that we have counted twice of the memory references in accessing the level-16 and level-24 hash tables because accessing two hash tables of 4-bit addresses is needed.

Optimization. The proposed data structure can be further optimized as follows. The level-8 pointer array can be combined with the first part of the level-16 hash table, the 20-bit pointer, the default port, the 4-bit sub-hash table count, and the 16-bit hash table. Similarly, the level-16 pointer table can be combined with the first part of level-24 hash table. Additionally, the level-24 pointer table can be combined with level-32 port table. The wasted memory is small because only a few prefixes are of length longer than 24. With these optimizations, the total number of memory accesses becomes 1 to 5.

Computing the hash tables of addresses more than 4 bits is a very time-consuming process. Also, the existence of the minimal perfect hash can not be known in advance. We have tried to use the hash table of 8-bit addresses directly for level-16 and level-24 hash table. The advantage is as follows. The size of the hash table of 8-bit addresses is 9 bytes (8×8 bits for V_0 and V_1, 8 bits for prefix count in the 8-bit trie). The memory reduction will be much larger when the number of prefixes in a 8-bit trie is small. However, the obvious drawback is the time-consuming computation of the hash table.

Lookup Scheme	Memory (KB)	# of memory accesses (Min/Max)
Original table	662.7	-
LC Trie	1,036.1	1/5
SFT	649.4	2/12
FIPRT	2,686	1/3
Proposed 8-8-8-8	533.5	1/8
Proposed 8-8-8-8 (optimized)	572.8	1/5

Table 2: Memory required to supporting 120,635 entries (oix routing table).

Build time for hash table of 4-bit addresses	Worst case	20 ms
	Average case	358 us
IP lookup time	Worst case	1 us
	Average case	0.6 us

Table 3: times required for building the hash tables and the IP lookups.

III. Performance Analysis

In this section, we will analyze the performance of the proposed schemes. Then a routing table from University of Oregon Route Views Archive Project [12] is also included in our analysis. This routing table contains 120,635 entries. The length distribution of the routing table is illustrated in Figure 1.

For the purpose of comparisons, three lookup data structures, the LC trie [6], the small forwarding table lookup scheme (SFT) proposed by W. Degermark, et al., [9] and a fast IP routing table lookup scheme (denoted as FIPRT) proposed by Huang, et al., [3] are also investigated.

The performance comparisons in terms of number of memory references and memory usage are shown in Table 2. The size of the memory required for the original routing table is computed based on the assumption of length format of the routing entry. In other words, each original routing entry records a 32-bit address, a 5-bit length, and an 8-bit port number. For the number of memory references in the FIPRT scheme is one to three, which is the best among the schemes. However, the memory needed for the whole routing table is 2.6Mbytes which is the largest in all the schemes. Although, in their original paper, the authors claimed that the routing table using their proposed data structure can fit into the SRAM, it is only for a small routing table (40,000 entries). The size of the elements in the level-1 pointer array in SFT is assumed to be three bytes. The total number of bytes required for the routing table in the SFT scheme is 649.4Kbytes. The proposed routing table is the smallest one (533.5Kbytes) and can be put in SRAM of size 640Kbytes.

We also conduct the experiments to measure the hash table build times and the lookup times for various IPs. Table 3 shows the measured times using Intel Pentium IV with 1.7G Hz clock rate and 768MB memory. We can see that the average lookup time is 0.6 micro-second that is fast. The main delay in lookup is because of the latency for the memory accesses and the slow hash function computing process. If we can fit the entire routing table into the fast cache memory, the IP lookup time can be further reduced.

The hash tables of 8-bit addresses in level-16 and level-24 are computed, using the same large routing table as before. There are around 12,000 8-bit segments in this routing table. Sixty-one percent of these 8-bit segments have the minimal perfect hash tables. If we increase the hash table size by 10, all except 29 segments have the perfect hash tables. For these 29 segments, the only solution is to use the hash table of size 256.

IV. CONCLUSIONS

In this paper, we introduced a new hash table to compact the routing table. Hash tables of 4-bit addresses were used as the building blocks to construct the hash table of 8-bit addresses which in turn were used to build the 8-8-8-8 hierarchical routing table.

We also conducted experiments to measure the size of required memory and showed that the proposed data structure of the routing table is the smallest, using a real routing table containing 120,635 routing entries.

REFERENCES

[1] M. A. Ruiz-Sanchez, Ernst W. Biersack, and Walid Dabbous, “Survey and taxonomy of IP address lookup algorithms”, IEEE Network Magazine, 15(2):8--23, March/April 2001.

[2] Yazdani, N. and Min, P.S. “Fast and scalable schemes for the IP address lookup problem”, Proceedings of IEEE High Performance Switching and routing 2000.

[3] N. F. Huang, S. M. Zhao, J. Y. Pan, and C. A. Su, "A fast IP routing lookup scheme for gigabit switching routers," in Proc. INFOCOM 99, March 1999.

[4] Butler Lampson, Venkatachary Srinivasan and George Varghese, “IP lookups using multiway and multicolumn search", IEEE/ACM Transactions on Networking, Volume 3, Number 3, Pages 324-334, 1999.

[5] Geoff Huston, “Analysis of the Internet's BGP routing table”, Internet Protocol Journal, 4(1), March 2001.

[6] S. Nilsson and G. Karlsson, “Fast address lookup for Internet routers”, Proc. IFIP 4th International Conference on Broadband Communications (BC'98), pp.11-22, 1998.

[7] M. Waldvogel, G. Varghese, J. Turner and B. Plattner. “Scalable high-speed IP routing lookups,” Proceedings of ACM Sigcomm, pp.25-36, October 1997.

[8] T. Chiueh et al, “High Performance IP Routing Table Lookup Using CPU Caching”, in Proc. INFOCOM 99, March 1999.

[9] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. “Small Forwarding Tables for Fast Routing Lookups.” ACM SIGCOMM, Palais des Festivals, Cannes, France, pp. 3-14, 1997.

[10]K. Sklower, A Tree-based Packet Routing Table for Berkeley Unix, Proc of 1991 Winter Usenix Conf, 1991, pp.93-99

[11]P. Gupta, S. Lin, N. McKeown, “Routing Lookups in Hardware at Memory Access Speeds”, in Proc. INFOCOM 99, March 1999.

[12]David Meyer, “University of Oregon Route Views Archive Project: oix-damp-snapshot-2002-12-01-0000.dat.gz “ at http://archive.routeviews.org/

[13]S. Suri, G. Varghese, and P.R. Warkhede, “Multiway Range Trees: Scalable IP Lookup with Fast Updates”, Global Com 2001.

[14]Srinivasan, V., and Varghese, G. “Fast address lookups using controlled prefix expansion”, ACM Trans. on Computer Systems 17, 1,1--40, Feb. 1999.