PAPER

A Small and Fast IP Forwarding Table Using Hashing*

Yeim-Kuan CHANG[†a)], ***Nonmember***

SUMMARY Building next generation routers with the capability of forwarding multiple millions of packets per second is required for the increasing demand for high bandwidth on the Internet. Reducing the required memory size of the forwarding table is a possible solution since small forwarding table can be integrated into the application specific integrated circuit (ASIC). In this paper a hash technique is developed to reduce the size of the IP forwarding table. The proposed data structure is a compressed 8-8-8-8 multibit trie that is based on hash tables of 4-bit addresses. Two optimization techniques are also proposed to further improve the performance of the proposed schemes. Our experimental results show that the proposed hashing-based schemes are better than the Small Forwarding Table scheme [6] both in memory size and lookup latency.

key words: *hash table, IP lookup, binary trie*

1. Introduction

The exponential traffic rate due to the advent of World Wide Web (WWW) demands for high bandwidth on the Internet [4]. Backbone routers with gigabit links such as OC-192 at 10 Gb/s and OC-768 at 40 Gb/s are common. Among all the tasks performed by the routers, the IP lookup 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 IP lookup problem becomes a longest prefix matching (LPM) problem because of there may be more than one prefix that matches the target IP address. The binary trie is the basic data structure used in most of IP lookup algorithms. Based upon this primitive trie structure, a set of prefix compression and transformation techniques are developed. The path compression technique [7], [12] improves time and space performance by removing the nodes that have only one child [2]. The prefix expansion, one of the most common transformation techniques allows us to transform one prefix into many longer or more specific prefixes that cover the same range of addresses. One way to use the prefix expansion technique is to remove the enclosure property and thus make a set of prefixes disjoint, or non-overlapping. Enclosure property means one prefix covers the addresses of another prefix. Making prefixes disjoint can remove many complex tasks when performing lookup operations. Another way to use the prefix expansion technique is to transform a binary trie into a k-bit trie, where k is called the stride. A k-bit trie can speedup the search performance by inspecting not just one bit but k bits at a time in an IP lookup operation. The stride size can be varied in any trie nodes at different levels. Therefore, if all nodes at the same level have the same stride size, we say that it is a fixed stride; otherwise, it is a variable stride.

Various techniques were developed to improve the performance of the multibit trie data structure [5], [10], [11]. The basic hardware based scheme proposed by Gupta et al. [8] uses a two-level multibit trie with fixed strides. The small forwarding table (SFT) scheme [6] was proposed to reduce space consumption. Nilsson et al. [5] proposed a scheme called Level-compressed (LC) that recursively transforms binary tries with prefixes into multibit tries. A large variety of routing lookup algorithms can be found in the survey paper by Ruiz-Sanchez et al., where the worst-case complexities of most IP lookup schemes in lookup latency, update time, and storage usage are compared [1].

In this paper, we shall propose a new method that employs a novel hashing technique to avoid wasting the unused space in the multibit trie. For 4-bit subtries, 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 organized as follows. We summarize the existing IP lookup schemes related to the proposed data structure in section 2. We also conduct performance simulations to demonstrate their performance differences. In Section 3, the basic idea of the proposed hash table of 4-bit addresses is first illustrated. Then an 8-8-8-8 hierarchical routing table is proposed. Two optimization techniques are also developed to further reduce the size of the proposed 8-8-8-8 data structure. Performance comparisons using real routing tables are presented in Section 4. Finally, a concluding remark is given in the last section.

2. Related IP Lookups and Their Performance

In this section,we shall perform computer simulations to obtain the memory sizes and lookup times for the related lookup schemes that are closely related to the designs for minimizing the memory consumption of the routing tables. For completeness, the basic binary trie, BSD trie, and the binary search on ranges are also included in the performance

Manuscript received May 21, 2004.

[†]The author is with the Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan, Taiwan, Republic of China.

a) E-mail: ykchang@mail.ncku.edu.tw

*This work was supported in part by the National Science Council, Republic of China, under Grant NSC-93-2213-E-006-085.

comparisons.

The data structures designed to minimize the amount of memory required for the routing tables are mostly based on the run-length encoding and the technique that uses local indices of pre-allocated arrays. Basically, in the run-length encoding, an array of keys (possibly repeated) is encoded by a bitmap and an array of non-repeated keys. The index of a bit position in the bitmap represents the index of the corresponding key in the original array. The number of one's in the bitmap is equal to the size of the array of non-repeated keys. Thus, the operation of searching for a key in the *i*th position of the original array now becomes a 2-step process. The first step is to count the number of one's (say *n*) ahead of *i*th position of the bitmap, including the *i*th position. The second step is to locate the *n*th position of the array of non-repeated keys. Two existing schemes using the run-length encoding are the small forwarding table [6] and the compressed 16-x scheme [2].

Performance Evaluation:

Instead of comparing the theoretical performance complexities [2], [6], [13], [14], trace-driven simulations are conducted to show the performance of the existing schemes. We measure the amount of memory required for building the routing tables and the lookup latencies. To demonstrate the performance differences between the existing schemes, the experiments are conducted on a large routing table (Oix-120k) containing 120,635 routing entries obtained from [9].

Since it is almost impossible to obtain the actual IP traffic being routed through the router at the time when the routing table trace is logged, we simulated the IP traffic by taking the IP's from the original prefixes and randomizing them. The same method was also used in [1] and [5].

The experiments for measuring lookup times are conducted on the Intel Pentium platform with a 2.4G Pentium IV processor and 8 KB L1 and 256 KB L2 caches. To have a fair comparison, all the schemes employ a 16-bit segmentation table similar to the 16-16 scheme. The clock cycles are measured by using the special instruction, rdtsc (read time stamp counter), provided by Intel Pentium processor. We assume each entry in the 16-bit segmentation table is of 4 bytes. Therefore, the 16-bit segmentation table consumes 256 kbytes. Notice that the source codes for the LC trie are obtained from the web site provided by the original authors [5]. In the experiments for the LC trie, we use the default branch factor at the root node which is 16. This means the default LC trie also has a 16-bit front-end segmentation table. The fill factor is set to 0.5. Since we do not have a chance to get the source codes from the original authors of the other existing schemes, we implement them using the C programming language. All the programs are compiled by gcc compiler and optimized by the O4 option.

In Table 1, the detailed statistics are included in order to illustrate the subtle differences between these schemes. The assumptions about memory sizes of the basic data units used in the corresponding schemes are also included in the parentheses. The original table shown in the last row of Table 1 contains 120,635 prefixes. The size of the memory required for the original routing table is computed based on the length format with a 32-bit address, a 5-bit length, and an 8-bit port. Therefore, the amount of memory required for the original table is 662.7 kbytes. The lookup time variability for the seven different schemes is summarized in Table 2 by showing the results of the 10th percentile, 50th percentile (median), 90th percentile, and the average.

The multiway search on range scheme [3] needs the largest amount of memory because pointer overhead in each block is high and many blocks are partially filled with prefixes. However, since the multiway search on range scheme takes the advantage of L1 cache lines, its lookup latency performs very well. The amount of memory required for the binary trie, BSD trie, and LC trie is only better than the multiway search on range scheme. The fill factor of 0.5 is the main reason why the LC trie consumes more memory space than expected. As most people expected, the BSD trie needs less amount of memory than the binary trie. The difference in lookup latency between the binary trie and BSD trie is not significant. The lookup latencies of the binary trie and the BSD trie are worse than other schemes. Since it is too complicated to show the statistics of levels 2 and 3 for SFT separately, all the information of levels 2 and 3 are combined. For example, there are 2,765 sparse segments in level 2 and level 3 that contain no more than eight prefixes. Based on the proposed formats in SFT, a sparse, a dense, and a very dense segment containing k prefixes need 24, 34 + 2k, and 40 + 2k bytes, respectively. The SFT scheme

Table 1 Memory required for table Oix-120k.

Scheme	Segmentation	Statistics	Memory size
Binary trie	16-bit segmentation table 65535 entries (4 byte each)	# of nodes: 320,478 (7 byte each)	2,447 KB
BSD trie		# of nodes: 222,334 (8 byte each)	1,993 KB
Binary range		# of prefixes: 112,286 (4 byte each)	1,104 KB
Multiway range		# of blocks: 71,034 (64 byte each)	4,695 KB
Compressed 16-x		Base array: 427 KB Compressed Bit-map: 427 KB CNHA: 37.9 KB	1,147 KB
LC trie	Branch factor:16 Fill factor:0.5	# of nodes: 259,371 (4 byte each) Base Vector: 110,679 (16 byte each) Prefix Vector: 9,927 (12 byte each) Next Hop Vector:255 (4 byte each)	2,859 KB
SFT	# of level-1 pointers: 13,317 # of level-2 pointers: 461 (4 bytes each)	# of segments (avg # of prefixes per segment) Sparse:2,765(2.9) Dense:4,300(25.5) Very dense:586(91.7) Maptable: 5.3K Binary array: 2K Code Word array: 8K	649.9 KB
Original table	N/A	# of prefixes: 120,635(45 bits each)	662.7 KB

Table 2 Average IP lookup times for table Oix-120k.

Scheme	# of memory access (min/max)	10th percentile (cycles)	50th percentile (cycles)	90th percentile (cycles)	Average lookup time (cycles)	Average lookup time (μs)
Binary trie	8/32	432	1065	1548	1140	0.470
BSD trie	8/26	490	1006	1452	1077	0.444
Binary range	1/6	264	680	1022	648	0.267
Multiway range	1/4	226	541	861	568	0.234
Compressed 16-x	1/3	230	434	830	409	0.169
LC trie	1/5	455	777	1070	757	0.312
SFT	1/12	378	657	892	503	0.207

Table 3 Routing tables considered in the paper.

	Funet-40k	Oix-80k	Oix-120k	Oix-150k
# of prefixes	41,709	89,088	120,635	151,511
Length distribution	8-30-32	6-32	8-32	8-32
Date	1997-10-30	2003-12-28	2002-12-01	2004-2-1

needs 649.9 kbytes of memory that is the smallest among all the existing schemes. The lookup latency of the SFT scheme also performs very well.

The compressed 16-x scheme performs better than SFT in term of lookup latency. However, the compressed 16-x scheme needs more than one Mbytes of memory. To have a complete understanding of the amount of memory required for the compressed 16-x scheme, we also conduct experiments for tables of various sizes. Table 3 shows the general information of the routing tables considered. We observe that the size of required memory is not proportional to the number of the prefixes in the original table. The amount of memory required for Oix-80k is 1838 kbytes which is even larger than that for Oix-120k. For Oix-150k, 3,182 kbytes of memory are needed. This disproportion in the amount of required memory is mainly caused by the number of segments that contain prefixes of lengths longer than 24. As a special case, it can be easily computed that a segment containing at least one prefix of length 32 needs 16 kbytes for the base array and the bitmap. Based on our simulation results, Oix-80k and Oix-120k tables have 64 and 18 segments that have at least one prefix of length 32, respectively. Therefore, Oix-80k consumes more memory than Oix-120k. In short, the more prefixes of lengths 25 to 32 exists in a routing table, the larger the memory is required for the compressed 16-x scheme. Based on this observation, it is hard to predict the amount of memory required for the compressed 16-x scheme without knowing the length distribution of the routing table.

Besides the compressed 16-x scheme, it seems that SFT is the best approach that a router can adopt because it has a very good performance both in memory size and lookup latency. However, its update can not be done without re-building entire data structure for the first 16 levels of the trie. The entire data structure for the first 16 levels is re-built when a prefix is inserted into one of the unused segment or all the prefixes in a segment are deleted. The re-building process that modifies code word array, base index array, and level-1 pointer array is very time-consuming. Since all the existing schemes, except SFT, use a 16-bit segmentation table, the update process is limited to a single segment if inserted or deleted prefix is longer than 16 bits. On the other hand, a number of fixed positions in the segmentation table need to be updated if the inserted or deleted prefixes are shorter than 16. Therefore, it is still desirable to have a new lookup scheme that has a better performance in memory and lookup time than SFT and allows a similar incremental update to the schemes with a 16-bit segmentation table.

3. The Proposed Scheme

In this section, a hashing technique is developed to remove the unused pointers in a multibit trie. Based on this hashing technique for 4-bit addresses, we will propose an 8-8-8-8 hierarchical data structure and show by experiments that the proposed schemes in terms of memory consumption and lookup latency perform very well compared with the existing schemes.

3.1 Hash Function

Assume that there is a set of m keys, S = $\{k_0, \ldots, k_{m-1}\}$; each key is an n-bit binary number. We like to find a mapping called the perfect hash function such that each key is mapped into a unique number in the range from 0 to $H\_Size - 1$, where H_Size is the size of the hash table. 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. We propose a mechanism that allows us to find a near-minimal perfect hash function. The proposed hash function, H, for an n-bit number $(b_{n-1} \ldots b_0)$ is formulated as follows.

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

The absolute value of x is denoted as |x|. V_0 and V_1 are two pre-computed arrays of size n. At least one of the elements V_0[i] and V_1[i] is zero and the other is in the range from $-2^{n-1} + 1$ to 2^{n-1}.

Consider a list of eight 4-bit numbers shown in Figure 1. We have V_0 = [0,0,0,0] and V_1 = [4,3,1,1]. Eight slots in the 4-bit trie array are unused. The trie using hash table is shown in Figure 1(c). The index for each prefix can be computed by using the pre-computed hash table. The construction algorithm for arrays V_0 and V_1 is based on a form of exhaustive search. Briefly, for each one out of $2n$ cells in V_0 and V_1, a number in the range from $-2^{n-1} + 1$ to 2^{n-1} is tried one at a time until the hashed values of all the 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's or 1's 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 keeps unchanged. Now assume the keys are in the order of $k_0, \ldots, k_{m-1}$ 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, $b_{n-1} \ldots b_0$, has the control on V_{bi}[i] if V_{bi}[i] is not yet controlled by one of the preceding keys for $i = n - 1$ to 0. For example, assume the first two

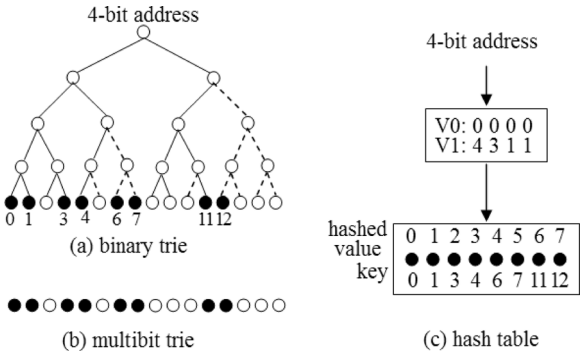

Fig. 1 The binary trie, 4-bit trie, and 4-bit hash table.

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 controls $V_1[1]$ and $V_1[0]$ 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 the range from $-2^{n-1}+1$ to 2^{n-1} to each cell controlled by the current key. If the hashed value, $H(b_{n-1}\ldots b_0)$, of the current key is already taken by its preceding keys or larger than H_Size-1, all the cells controlled by the current key must be re-assigned new numbers. If no number can be used after exhausting all the possible cells controlled by the current key, we will backtrack to the previous key and set it to be the current key again. We then re-assign the cells controlled by the new current key different numbers and continue the same procedure. If the hashed value of the current key is in the range from 0 to H_Size-1 and is not taken by its preceding keys, the next key will be tried and the same procedure goes on.

Example 1: We use the eight 4-bit keys in Figure 1 to demonstrate how the hash table is constructed. The numbers of 0's are the 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 and H(0001) = 1. Now, the third key 0011 controls only $V_1[1]$ which is 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. Fortunately, 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 4-bit hash table**:*

Since building an n-bit hash table uses an 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 large routing tables. 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. Figure 2 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. The hashed values are shown in the parentheses. Value nine is the only unused hashed value for these 10 keys.

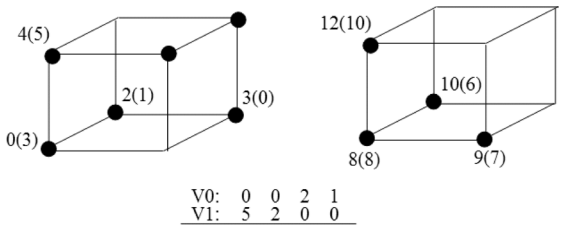

Fig. 2 Hash tables of 4-bit addresses.

***Additional data structure**:*

In order to make the proposed hash technique work for the routing table lookup, two additional data are needed. One is the total count of the hashed keys and the other is the hashed key itself. Consider again the hash table consisting of eight keys in Figure 1. We have illustrated how to search for the keys that are in the hash table. However, for the keys that are not in the hash table, we need a mechanism to avoid fault hits. For example, the hashed value of key 1111 is nine which exceeds the capacity of the hash table. Therefore, the total count of the hashed keys can be used to handle this case. To be more specific, if the hashed value of a key exceeds the total number of the keys minus one, the key must not be in the hash table. Consider another case where the incoming key is 0010. The hashed value is one which is the same as that of key 0001. Therefore, to avoid an incorrect result, the key itself, 0001, must also be stored along with other routing information. In other words, even the hashed value does not exceed the capacity of the hash table we still need to check if the key matches the value stored in the corresponding cell of the hash table.

3.2 The Proposed Data Structure

The proposed data structure for the routing table uses 4-bit hash tables as the building blocks. It is organized as a 4-level 8-8-8-8 hierarchy and is shown in Figure 3. Instead of a 16-bit front-end segmentation table, we use an 8-bit pointer table as the front-end lookup array to avoid wasting too much memory. The reason why we do not use hashing for this front-end array is that we trade a little more space for smaller 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 stores either a 20-bit pointer pointing to the hash table of the second level subtrie or a port number if only prefix of length 8 exists.

The hash table of the second level subtrie uses a data structure called *format H* block. Each *format H* block consists of a 4-bit hash table that recursively hashes at most 16

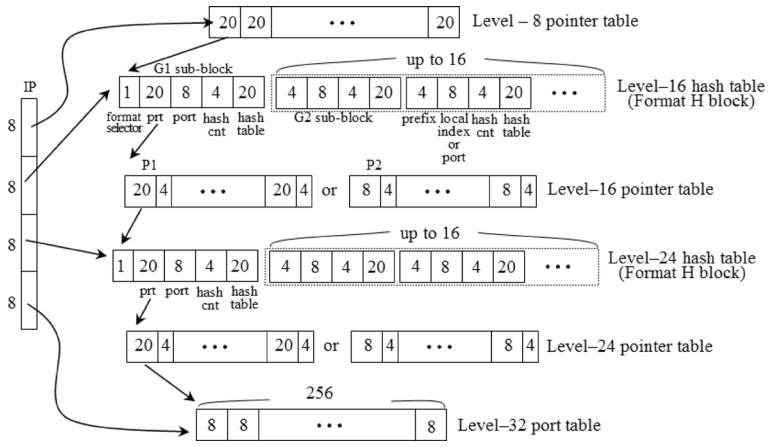

Fig. 3 Data structure for the proposed 8-8-8-8 table.

4-bit hash tables. The first part of *format H* block is called the *format G1* sub-block that consists of a 1-bit format selector for the level-16 pointer table (described later), a 20-bit global pointer pointing to the level-16 pointer table, an 8-bit default port number, a 4-bit count to record the total number of next-level hash tables and a 20-bit hash table. The 20-bit hash table records $V_0[3\ldots0]$ and $V_1[3\ldots0]$. Based on the proposed hashing technique, at least one of V_0[i] or V_1[i] must be zero, for i = 3 . . . 0. Therefore, one bit is sufficient to record whether V_0[i] or V_1[i] is zero and four bits are needed to store the other value. In summary, the 20-bit hash table consists of a 4-bit number called bit-array that records whether V_0[i] or V_1[i] is zero and four 4-bit numbers called V_array that record the other four values of V_0[i] and V_1[i] that may be zero or non-zero. The 20-bit hash table (bit_array, V_array) is represented in the numerical form of ($b_3b_2b_1b_0$,V[3],V[2],V[1],V[0]). The second part of the *format H* block is an array of *format G2* sub-blocks. Each *format G2* sub-block consists of a 4-bit prefix string, a port number or an 8-bit local index to the level-16 pointer array, a 4-bit count to record the number of keys in the corresponding sub-hash table, and a 20-bit sub-hash table. Notice that when the trie stops at prefix length 12, a port number instead of local index to the pointer array is recorded. Also the fields for key count and sub-hash table are set to zero.

The data structure in level 24 is the same as that in level-16. The level-16 and level-24 pointer tables are arrays of either format P1 pointers consisting of 4-bit prefixes and 20-bit pointers or format P2 pointers consisting of 8-bit port numbers if no subtrie exists beneath this level. The 1-bit format selector in G1 sub-block is used to determine whether format P1 or P2 is used. We use a different data structure in level 32 since most routing entries have their lengths shorter than or equal to 24. The level-32 structure is organized as an array of 256 port numbers.

Build and update:

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, we pre-compute all the 64K (2^{16}) possible 4-bit hash tables which account for 160 kbytes since each 4-bit hash table needs 20 bits. Notice that the number of distinct hash tables is much less than 64K which will be shown in the sub-section of optimization. Now computing a 4-bit hash table becomes one memory reference to the corresponding 20 bits in the array of 64K entries. Thus, the updating process for deleting or inserting a prefix in the proposed 8-8-8-8 data structure is as fast as that of updating a 4-bit trie plus accessing at most four additional pre-computed 4-bit hash tables and rearranging the corresponding pointers.

IP lookup:

The IP lookup process based on the proposed 8-8-8-8 hierarchical structure is simple and is described as follows. The level-8 pointer table is first referenced by using the most significant 8 bits (bits 31 to 24) of the IP address. Then the next 8 bits (bits 23 to 16) of the IP address and the level-16 hash tables are used to compute the index of level-16 pointer table. The hashed value is first computed with bits 23 to 20 of the IP address and the 20-bit hash table in the *format G1* sub-block of the level-16 hash table. If the hashed value is larger than the 4-bit count of the *format G1* sub-block then the 8-bit port number of the *format G1* sub-block is returned. Otherwise, the hashed value is used as the index for reference the corresponding *format G2* sub-block. The 4-bit prefix of the corresponding G2 sub-block is matched against bits 24 to 20 of the IP address. If match is not found the default port number of the *format G1* sub-block is again returned. If match is found and the key count field is zero, then the port number in the G2 sub-lock is returned. Otherwise, bits 19 to 16 of the IP address, the 4-bit count and, the 20-bit hash table of the corresponding *format G2* sub-block are used to check if there is a matched element in the level-16 pointer array. The format selector in the G1 sub-block

determines whether format P1 or P2 pointer table is used. If a matched element in level-16 pointer array is not found the default port number in G1 sub-block is returned. Otherwise, the same process is performed for level-24 hash table and pointer 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. The worst-case number of the memory references for a lookup is from 1 to 8. Two memory references are needed when accessing the level-16 or level-24 hash tables. In other words, at most four 4-bit hash tables are checked.

3.3 Optimizations

Two optimization techniques are developed to further improve the performance of the proposed scheme. The first technique called *level-compression (LC)* approach is designed to reduce the number of memory references. The second technique called *maptable* approach is designed to further reduce the memory consumption.

The level-compression approach is developed because of the following observation. There is only one *format G1* sub-block in each level-16 hash table. Therefore, the front-end level-8 pointer table can be combined with the *format G1* sub-blocks. Similarly, the level-16 pointer table can be combined with the *format G1* sub-blocks of the level-24 hash tables. 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 the LC optimization, the total number of memory accesses for an IP lookup operation becomes 1 to 5.

In order to further reduce the memory usage, the second optimization technique called the *maptable* approach uses an approach similar to the maptable of SFT [6]. Before we describe the detailed design, we shall first illustrate our idea by considering the example in Figure 1. The hash table of the eight keys is represented numerically in Figure 4(a) as (1111,4,3,1,1). This hash table is not only applicable to the list with the eight keys shown in Figure 1, but also applicable to the other lists with various numbers of keys. For example, the hash table (1111,4,3,1,1) can also be applied to the list with 0, 2, 3, 4, and 8, the list with 0, 2, 3, 4, 5, 10, 11, 12, 14, and 15, and so on. The complete list of hashed values and the corresponding keys are illustrated in Figure 4(b). In general, finding the list of keys that can be applied to the hash table is easy. We first select a list of consecutive non-duplicated hashed values starting from 0 to $cnt - 1$, where cnt is the number of the keys in the list. Then the corresponding keys to the selected hashed values form the list of keys that is applied to the hash table. If the capacity of the hash table that is allowed to be more than the number of the keys, the number of the applicable lists may be even larger. The following example shows that the hash table that is minimal and prefect for a list of 11 keys can also be applied to another list containing 10 keys.

Example 2: The minimal perfect hash table of keys, 0, 1, 2, 4, 5, 6, 8, 11, 12, 13, and 15 is (0011,8,2,-4,-1). By an exhaustive search, we have not found a minimal prefect hash table for the list of keys 0, 1, 2, 4, 5, 6, 8, 11, 12, and 15. If we allow the hash size to be one more than the number of keys, the hash table (0011,8,2,-4,-1) can be reused for the latter list.

It is obvious that there are 2^{16} combinations of zero to sixteen 4-bit numbers. We used an exhaustive search to find all the distinct V-array's that are applicable to all 2^{16} combinations of 4-bit numbers. We found out that there are only 217 distinct V_array's. Notice that we do not further reduce the number of V_array's although it is possible. Therefore, V_array can be replaced by an 8-bit number that is the index to the array of pre-computed 217 V_array's. The bit-array remains the same. This technique reduces the size of a 20-bit hash table from 20 to 12 bits with only a small overhead of 217×16 bits = 434 bytes used for the pre-computed array of 217 V_array's.

Although computing the hashed value by using the pre-computed hash table and the corresponding 4-bit IP address is simple, we propose another enhanced technique to avoid the computations for hashed values. There are sixteen possible values for 4 bits. Therefore, we can pre-compute the hashed values of a hash table. This approach allows us to only record sixteen 4-bit hash values instead of the V_array itself. In other words, we create a 2-dimensional array called maptable in which the first dimension is of size 217 and the second dimension is of size 16. Each element of maptable is 4 bits because the hashed value is in the range of 0 to 15. Assuming the recorded 8-bit V_array index in the *format G1/G2* sub-block is i and the 4-bit address is $A = a_3a_2a_1a_0$, the lookup process becomes a two-step process. Assume the current 4-bit bit-array is $B = b_3b_2b_1b_0$. The first step requires the exclusive-or (XOR) operation on A and B. Assume A XOR B = C. Then, the second step is a search for the Cth element of the ith row in maptable. The 2-D mapt-

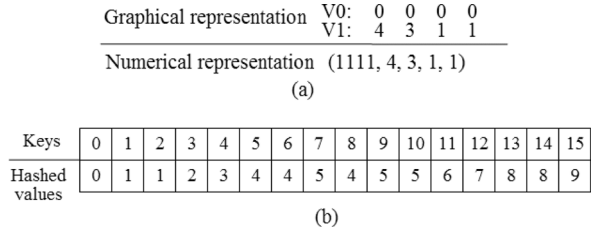

Fig. 4 Illustrations of lists containing various numbers of keys hashed to a 4-bit hash table.

Table 4 The number of memory accesses for Oix-120k.

Lookup schemes	# of Memory accesses (min/max)
SFT	2/12
Original 8-8-8-8	1/4
Proposed 8-8-8-8	1/8
Proposed 8-8-8-8 with LC	1/5
Proposed 8-8-8-8 with maptable	1/12
Proposed 8-8-8-8 with LC and maptable	1/9

able is of size $217 \times 16 \times 4$ bits = 1736 bytes.

By using the maptable optimization, the total number of memory accesses for a lookup becomes 1 to 12. This is because obtaining a hashed valued takes one memory access to the maptable and there are at most four hashed values to complete a lookup. The proposed 8-8-8-8 scheme can be optimized by the two techniques simultaneously, which results in 1 to 9 memory accesses. We summarize the maximum and minimum numbers of memory accesses that a lookup operation takes for the proposed 8-8-8-8 schemes in Table 4, along with the results for the SFT scheme.

4. Performance Evaluation

In this section, we use the same simulation setup in section 2 to evaluate the performance of the proposed schemes in term of amounts of memory and lookup latencies required for the proposed schemes. As demonstrated by the previous performance results, the SFT scheme is in general better than other existing lookup schemes. Therefore, only the SFT scheme is compared with the proposed schemes in this section.

We first show the memory sizes required for the routing tables in Table 5. The sizes of the routing tables for all the proposed 8-8-8-8 schemes are smaller than SFT. We believe that maptable will reside in the L1 cache most of time since maptable array is only less than 2 kbytes. Therefore, the average lookup latency will perform much better than what the worst-case number of memory references indicates.

Figure 5 shows the distribution of lookup latencies using oix-120k routing table. The lookup latencies for the proposed scheme with only LC optimization and that with both LC and maptable are shown. We do not show the results for the scheme with only maptable optimization and that without any optimization because they have no significant difference compared with the two results shown in Figure 5. We can see that the proposed schemes perform better than SFT. The performance of the proposed scheme with both LC and maptable is only a little better than the scheme with only LC optimization. The highest peak of the proposed schemes is from the accesses to the prefixes of length 24 because the accesses to prefixes of length 24 account for more than 60total accesses. The leftmost and rightmost peaks of the proposed schemes are for the accesses to the prefixes of length less than and larger than 24, respectively. The performance of the SFT scheme is the worst. Although the curve of the SFT scheme also has three prominent peaks, it has a different characteristic from the proposed schemes. The differences are described as follows. Since all the prefixes of length less than 16 are extended to length 16, the leftmost peak is mainly caused by the accesses to the prefixes of length equal to 16. The other two peaks are caused by the accesses to the prefixes of length 24. Theses two peaks correspond to the two different access patterns for the sparse chunks and non-sparse chunks (dense and very dense chunks).

We also obtain the performance curves for the routing table Oix-150k, Oix-80k, and Funet-40k. Their curves have similar shapes to that of Oix-120k. Therefore, we only show the lookup time distribution for funet-40k table in Figure 6. The performance of the proposed schemes in lookup latency is better than SFT for all the tables of various sizes.

Table 6 summarizes the average lookup times for these four tables. We can see that there is no big difference in the average lookup latencies among all the three schemes for the smallest funet-40k table. However, the average lookup times of the proposed schemes become better than SFT for

Table 5 Memory required for the proposed schemes.

Routing table (kbytes)	Funet-40k	Oix-80k	Oix-120k	Oix-150k
SFT	274.5	538.2	649.9	800.6
Proposed 8-8-8-8	174.7	415.9	533.5	707.1
Proposed 8-8-8-8 with LC	195.4	441.5	572.8	726.3
Proposed 8-8-8-8 with maptable	164.2	392.1	503.4	666.9
Proposed 8-8-8-8 with LC and maptable	184.7	417.7	541.7	688.2

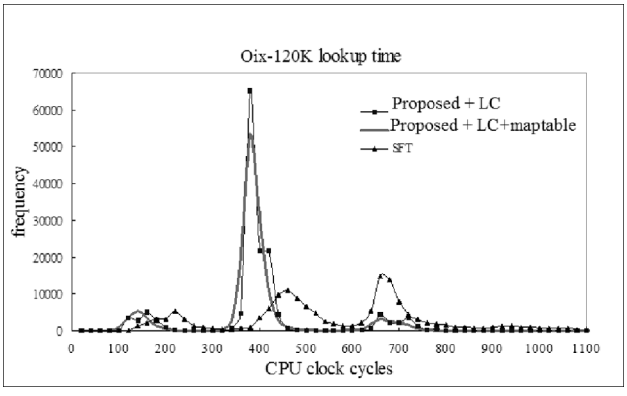

Fig. 5 IP lookup latency for table Oix-120k.

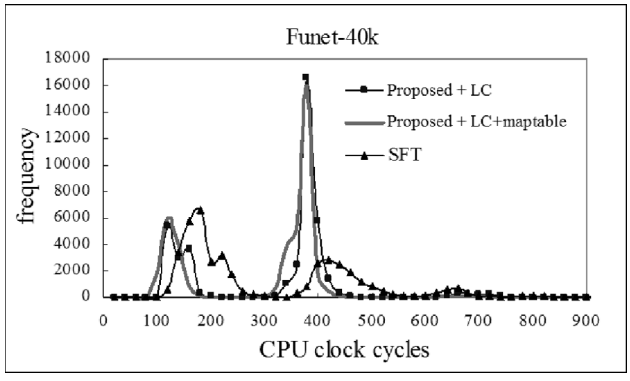

Fig. 6 IP lookup latency for table Funet-40k.

Table 6 Average lookup latencies in clock cycles.

Routing table	Funet-40k	Oix-80k	Oix-120k	Oix-150k
SFT	292	452	507	546
Proposed 8-8-8-8 with LC	300	350	401	423
Proposed 8-8-8-8 with LC and maptable	286	348	392	411

larger tables.

5. Conclusions

In this paper, we conducted trace-driven simulations to evaluate the existing lookup schemes designed for routing table reduction. We found out that the compressed 16-x scheme is the fastest one and the SFT scheme is the smallest one. To further reduce the table size, we introduced a new hashing technique that can compact the routing table and produce a smaller data structure than SFT. Hash tables of 4-bit addresses are the basic building blocks to construct hash tables of 8-bit addresses which in turn were used to build the proposed 8-8-8-8 routing table. The basic hash table is further optimized in term of access latency and memory size by the level compression and maptable techniques. The experiments showed that the proposed data structure is smaller and faster than the SFT scheme.

References

[1] M.A. Ruiz-Sanchez, E.W. Biersack, and W. Dabbous, "Survey and taxonomy of IP address lookup algorithms," IEEE Netw. Mag., vol.15, no.2, pp.8–23, March/April 2001.
[2] N.F. Huang, S.M. Zhao, J.Y. Pan, and C.A. Su, "A fast IP routing lookup scheme for gigabit switching routers," Proc. INFOCOM 99, pp.1429–1436, March 1999.
[3] B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search," IEEE/ACM Trans. Netw., vol.3, no.3, pp.324–334, 1999.
[4] G. Huston, "Analysis of the Internet's BGP routing table," Internet Protocol Journal, vol.4, no.1, pp.2–15, March 2001.
[5] S. Nilsson and G. Karlsson, "IP-address lookup using LC-tries," IEEE J. Sel. Areas Commun., vol.17, no.6, pp.1083–1092, June 1999.
[6] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," ACM SIGCOMM, Palais des Festivals, pp.3–14, Cannes, France, 1997.
[7] K. Sklower, "A tree-based packet routing table for berkeley unix," Proc. 1991 Winter Usenix Conf, pp.93–99, 1991.
[8] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," Proc. INFOCOM 99, pp.1240–1247, March 1999.
[9] D. Meyer, "University of oregon route views archive project: Oix-damp-snapshot-2002-12-01-0000.dat.gz," http://archive.routeviews.org/
[10] V. Srinivasan and G. Varghese, "Fast address lookups using controlled prefix expansion," ACM Trans. Comput. Syst., vol.17, no.1, pp.1–40, Feb. 1999.
[11] S. Sahni and K.S. Kim, "Efficient construction of multibit tries for IP lookup," IEEE/ACM Trans. Netw., vol.11, no.4, pp.650–662, 2003.
[12] D. Morrison, "PATRICIA-Practical algorithm to retreive information coded in alfanumeric," J. ACM, vol.15, no.4, pp.514–534, Oct. 1968.
[13] A. Andersson and S. Nilsson, "Improved behaviour of tries by adaptive branching," Inf. Process. Lett., vol.46, no.6, pp.295–300, 1993.
[14] P. Crescenzi, L. Dardini, and R. Grossi, "IP address lookup made fast and simple," 7th Annual Euro. Symp. Algorithms; also, Technical Report TR-99-01 Univ. di Pisa, 1999.

Yeim-Kuan Chang received the M.Sc. degree in computer science from University of Houston at Clear Lake in 1990 and the PhD degree in computer science from Texas A&M University, College Station, Texas, in 1995. Dr. Chang is currently an Assistant Professor in the Department of Computer Science and Information Engineering at National Cheng Kung University, Tainan, Taiwan, Republic of China. His research interests are in the areas of computer architecture, parallel processing, Internet technology, and computer networking.