Efficient Hardware Architecture for Fast IP Address Lookup

Derek Pao[1], Cutson Liu[1], Angus Wu[2], Lawrence Yeung[3] and K. S. Chan[3]*

[1]Dept. of Computer Engineering & Information Technology, City University of Hong Kong, Kowloon, Hong Kong
[2]Dept. of Electronic Engineering, City University of Hong Kong, Kowloon, Hong Kong
[3]Dept. of Electrical and Electronic Engineering, University of Hong Kong, Hong Kong

[*]Corresponding author. Email: d.pao@cityu.edu.hk

Abstract — A multigigabit IP router may receive several millions packets per second from each input link. For each packet, the router needs to find the longest matching prefix in the forwarding table in order to determine the packet's next-hop. In this paper, we present an efficient hardware solution for the IP address lookup problem. We model the address lookup problem as a searching problem on a binary-trie. The binary-trie is partitioned into four levels of fixed size 255-node subtrees. We employ a hierarchical indexing structure to facilitate direct access to subtrees in a given level. It is estimated that a forwarding table with 40K prefixes will consume 2.5Mbytes of memory. The searching is implemented using a hardware pipeline with a minimum cycle of 12.5ns if the memory modules are implemented using SRAM. A distinguishing feature of our design is that forwarding table entries are not replicated in the data structure. Hence, table updates can be done in constant time with only a few memory accesses.

1. INTRODUCTION

In IPv4, the destination address of an IP packet is 32 bits long. There are two addressing schemes in use, namely the classful addressing scheme and the classless interdomain routing (CIDR) scheme. The classful addressing scheme uses a simple two-level hierarchy. The 32-bit IP address is broken down into the network address part and the host address part. IP routers forward packets based only on the network address until the packets reach the destined network. Typically, an entry in the forwarding table stores the address prefix (e.g. the network address) and the routing information (i.e. the next-hop router and output interface). Three different network sizes are defined in the classful addressing scheme, namely classes A, B, and C. The addresses of class A networks are 8 bits long and start with the prefix '0'. The addresses of class B networks are 16 bits long and start with the prefix'10'. The addresses of class C networks are 24 bits long and start with the prefix '110'. The length of the network address can be determined based on the value of the destination address. The address lookup operation amounts to finding an exact prefix match in the forwarding table.

In order to allow a more efficient use of the IP address space and avoid the problem of forwarding table explosion, arbitrary length prefixes are allowed in the CIDR scheme. With CIDR, the address lookup operation is more difficult. It amounts to finding the longest address prefix in the forwarding table that matches the destination address. Several software solutions to the IP address lookup problem have been published in the literature [1-3,7,9-11]. These methods employ sophisticated data structures and rely heavily on the on-chip cache memory of the CPU. The average time to perform one address lookup ranges from 0.5 to 6 micro second when executed on a 200 MHz Pentium-Pro-based computer [8]. Although the processing time can be shortened by using a more powerful CPU, the software approaches may not scale with the explosive growth of the internet in terms of the data rate and the size of the forwarding table.

Gupta et al proposed a simple hardware table lookup approach in [4]. Their design uses 2-level lookup tables. The first level lookup table (called *TBL24*) has 2^{24} entries and the second level lookup table (called *TBLlong*) is consisted of 32K 256-entry segments. Let A and B be two strings of 0's and 1's, and the length of A is less than or equal to the length of B. A *encloses* B, or A is B's *enclosure*, if A is a prefix of B. Let F be the set of prefixes in the forwarding table, and $A \in F$. A is the *inner enclosure* of B with respect to F if A is the longest prefix in F that encloses B. Conceptually, the d-th entry in the lookup table (for both *TBL24* and *TBLlong*) stores the next-hop of the *inner enclosure* of d with respective to F. Let d be a 24-bit address, and p is the *inner enclosure* of d with respective to F. If d does not enclose any prefix q in F (where q has > 24 bits), then the d-th entry in *TBL24* stores the next-hop of p. If d encloses some prefix q in F (where q has > 24 bits), then the d-th entry in *TBL24* stores the address of a 256-entry segment of *TBLlong*. The IP address lookup operation is divided into two steps. First, bits 1 to 24 of the destination address are used to index into *TBL24*. If a next-hop identifier is returned, then the lookup process is done. If the *TBL24* entry stores the address of a *TBLlong* segment, bits 25 to 32 of the destination address are then used to index into the *TBLlong* segment to obtain the required next-hop identifier. Assume each next-hop

0-7803-7476-2/02/$17.00 (c) 2002 IEEE.

identifier is 8 bits long. The overall memory requirement is about 33M bytes. Using the pipelined processing technique, this approach can achieve a throughput of one address lookup per memory access time. By using an additional intermediate length table and break down the 32-bit address into three portions of 21, 3 and 8 bits, respectively, the amount of memory can be reduced to 9M bytes. The advantage of this approach is its simplicity. However, it has two drawbacks. First, the next-hop identifier associated with a prefix may be replicated many times. For example, the next-hop identifier associated with a 8-bit prefix may be stored in up to 2^{16} entries in *TBL24* and also in one or more *TBLlong* segments. This high degree of replication will make table updates difficult. Second, this approach is not applicable to IPv6 with 128 bits destination addresses.

Huang and Zhao proposed an encoding and compression scheme to reduce the memory requirement [5]. In their design, the 32-bit address space is divided into 64K segments (called the next hop array, NHA). The size of a NHA depends on the length of the longest prefix that shares the same 16-bit prefix. Essentially each entry in the NHA stores the next-hop of its inner enclosure. An encoding scheme was proposed to compress the NHAs to save memory. The amount of memory saved depends on the distribution of the prefixes. The memory requirement for Huang and Zhao's approach is very sensitive to the number of long prefixes in the forwarding table, and is difficult to estimate. For every 32-bit prefix, a 64K-entry NHA (which consumes 16K bytes memory) may be required. In a hardware implementation, provision of a safety margin is necessary. Today's backbone routers have, on the average, one to two hundreds long prefixes. Suppose the system is required to support at least 1K 32-bit prefixes, then the amount of memory required is at least 16M bytes. Forwarding table updates are even more difficult in Huang and Zhao's design. Insertion and deletion to the forwarding table may affect the size of the NHA. Hence, the memory management unit may need to allocate new segment of larger (or smaller) size and de-allocate the old segment. The allocation and de-allocation of variable sized segments is very difficult to implement and periodic memory compaction is inevitable.

In this paper, we present a novel hardware architecture for the IP address lookup problem. We limit the discussion to IPv4. The address lookup problem is modeled as a searching problem on a binary-trie. The complete binary-trie is partitioned into non-overlapping subtrees of 255 nodes. Each subtree is represented using a bit-vector and can be searched in parallel. We employ both paralleling processing and pipelining to maximize the throughput. If the data structures are stored in SRAM, we can achieve a cycle time of 12.5ns. Since the forwarding table entries are not replicated, forwarding table updates can be done in constant time and only involved a few memory accesses. In section 2, we'll present the circuit for searching a 8-level 255-nodes binary-trie in parallel. We'll use the circuit as a building block and describe the pipeline architecture in section 3. Section 4 is the concluding remarks.

2. SEARHING A BINARY-TRIE in PARALLEL

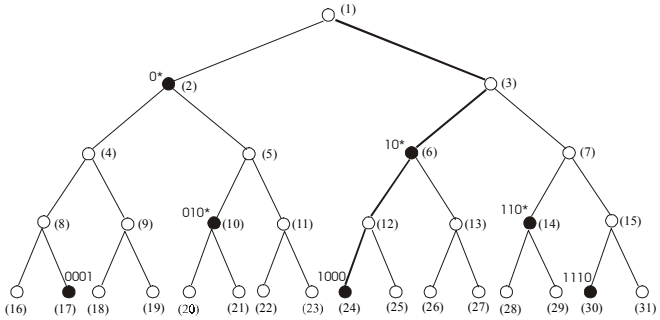

Figure 1. A binary-trie for a 4-bit address space.

The problem of finding the longest matching prefix can be modeled as the searching problem on a binary-trie. In a binary-trie, a string of 0's and 1's is represented by the path from the root to the corresponding node in the binary tree. We adopt the convention of taking a left turn for the value 0, and a right turn for the value 1. Figure 1 depicts a binary-trie for a 4-bit address space. The presence of a prefix is represented by a black dot in the Figure. To find the longest matching prefix for a given input address, we search along a simple path from the root to the leaf that corresponds to the input address. The search path for the input address 1000 is highlighted in the Figure. We label the nodes of the binary-trie in level order. The root is assigned the label 1. The left child of node i is assigned the label $2i$, and the right child of node i is assigned the label $2i+1$. A k-bit binary-trie can be represented by a bit-vector with $2^{k+1}-1$ bits, called the *tree-vector*. We number the bits in the *tree-vector* from left to right. Bit i of the *tree-vector* is equal to 1 if the prefix that corresponds to node i of the binary-trie is present in the forwarding table, otherwise bit i is equal to 0. Similarly, the search path can be represented by a bit-vector called the *mask-vector*. In the example of Figure 1, both the *tree-vector* and the *mask-vector* have 31 bits. To search for the longest matching prefix for the address 1000, we need to perform a bit-wise AND operation of the *tree-vector* with a *mask-vector* with bits 1, 3, 6, 12, and 24 equal to 1. The longest matching prefix, if any, is given by the bit number of the rightmost '1' in the *result-vector* of the AND operation. Accompanying the *tree-vector* is a *routing-vector* that stores the next-hop identifier. The next-hop identifier that corresponds to the rightmost '1' in the *result-vector* is used to forward the packet. In the example shown in Figure 2, the packet with destination address 1000 will be sent according to the information stored in the next-hop identifier number 9.

0-7803-7476-2/02/$17.00 (c) 2002 IEEE.

tree-vector corresponds to the binary-trie of Figure 1

0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

routing-vector for the binary-trie of Figure 1

-	5	-	-	-	8	-	-	-	6	-	-	-	3	-	-	4	-	-	-	-	-	-	9	-	-	-	-	-	2	-

mask-vector for the destination address 1000

1	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

Result vector = tree-vector AND *mask-vector*

0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

Figure 2. Example of the parallel search operation on a binary-trie.

To find the best matching prefix in a subtree, we need to perform the following operations: (i) read the *tree-vector* and the *mask-vector* from memory, (ii) perform a bit-wise AND operation of the *mask-vector* and the *tree-vector* to obtain the *result-vector* and find the position of the rightmost '1' in the *result-vector*, and (iii) read the next-hop identifier from the *routing-vector*, if required. The above 3 operations are carried out using a pipelined architecture. Details will be given in section 3. Since steps (i) and (iii) only involve memory accesses, the processing time of step (ii) will be our major concern in the circuit design. Since the bit-wise AND operation only involves 1 single gate delay, the most critical part of the computation will be the task to find the position of the rightmost '1' in the *result-vector*. In our design, we assume the subtree has 8 levels and 255 nodes. With the current CMOS technology, we are able to implement a digital circuit called the *RMB locator circuit* to find the position of the right most '1' in the *result-vector* in about 12.5ns. We shall use this as a building block in our pipelined architecture presented in section 3.

2.1 Design of the *RMB locator*

We observe that the position of the rightmost '1' can be determined easily once we know the level of the longest matching prefix in the subtree. We assume the root is at level 0. Consider the example in Figure 1, the position of the rightmost '1' corresponding to the search path "1000" will be 6 and 24 if the longest matching prefix is found on level 2 and level 4, respectively. The level number of the rightmost '1' in the *result-vector* can be determined using simple OR-gates and a 8-to-3 priority encoder as shown in Figure 3. Once we know the level number of the rightmost '1', the bit position can be found by a simple table lookup. The lookup table has 128 rows and 8 columns where each row stores the indexes of the nodes along the corresponding search path. When the *result-vector* is all zeros, a *no-match* signal will be generated and the bit position obtained from the table will be discarded.

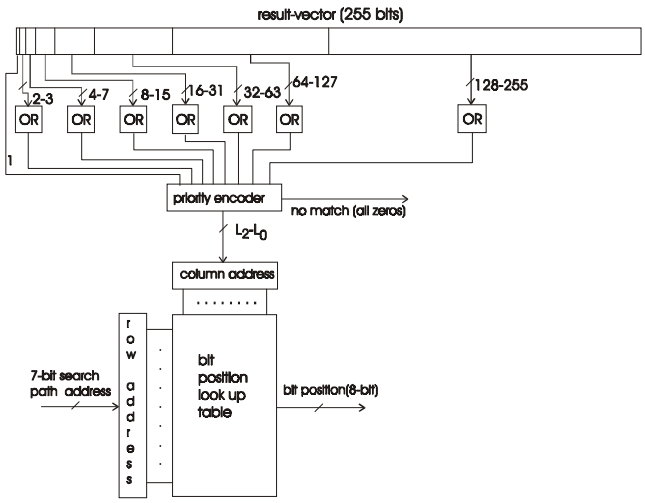

Figure 3. Organization of the *RMB locator*.

The circuit is designed using the AMD C5N 0.5 μm technology device model. All the devices are implemented with minimum size without device sizing to optimize the timing of the system. The propagation delay for determining the level number of the rightmost '1' is found to be 2.5ns by circuit simulation. The size of the bit position lookup table is 1Kbytes and it is implemented using SRAM. The SRAM access time is assumed to be 10ns. Hence, if the *tree-vectors* and *routing-vectors* are also implemented using SRAM, then we can achieve a cycle time of 12.5ns. If the *tree-vectors* and *routing-vectors* are implemented using DRAM, then the cycle time will be about 50ns.

3. HARDWARE ARCHITECTURE

In IPv4, the destination address of an IP packet has 32 bits. We use a hierarchical approach to find the longest matching prefix for a 32-bit input address. Figure 4 shows the typical distribution of prefix lengths in a backbone router [6]. Because of the hierarchical structure of IP address allocation, over 99% of the prefixes are shorter than 25 bits.

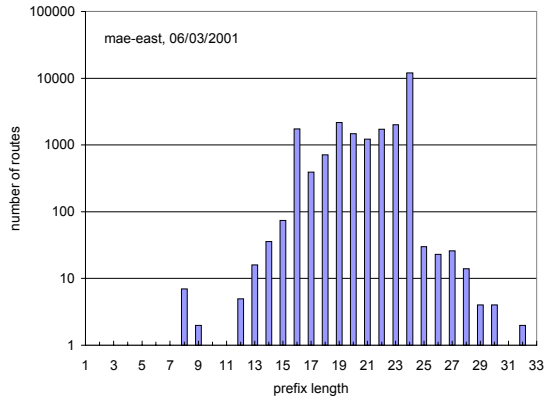

Figure 4. Typical distribution of prefix lengths in a backbone router

Logically the binary-trie for the complete 32-bit address space has 33 levels and $2^{33}-1$ nodes. Each node in the binary-trie corresponds to a distinct prefix. A node will be assigned the value '1' if the corresponding prefix is present in the forwarding table; otherwise, the node is assigned the value '0'. We partition the entire binary-trie into non-overlapping blocks of 255 nodes as shown in Figure 5. Each block corresponds to a 255-node subtree in the binary-trie and is represented using a 255-bit *tree-vector*. Physically, we only store the *tree-vectors* that have at least one bit equal to 1. The entire binary-trie can be partitioned into 4 levels of subtrees, namely level 0 to level 3. For each level of subtrees, we build separate index tables called the *index blocks* (*IBs*) to facilitate direct access to subtrees in a given level. Based on the distribution of prefix lengths in real life backbone routers, we may need to store several thousands level 2 *tree-vectors* and about a hundred level 3 *tree-vectors*.

The proposed hardware architecture will incorporate parallel processing and pipelining to maximize the throughput. First, we search the level 0 to level 2 subtrees in parallel as shown in Figure 6, i.e. to find the best matching prefix with 1 to 8 bits, 9 to16 bits and 17 to 24 bits in parallel. To match the packet's destination address with prefixes of 1-8 bits, we only need to decode the first bit of the destination address, and then use the circuit described in section 2.1 to search one of the two level 0 *tree-vectors*. The search path is determined by the value of bits 2-8. Since there are only 128 search paths, the 128 *mask-vectors* as well as the two level 0 *tree-vectors* can be stored in a fast register file. Similarly, to match the destination address with prefixes of 9 to 16 bits, we use the first 9 bits of the destination address to index into the level 1 *index block* IB1. The entry in IB1 will give the address of the corresponding level 1 *tree-vector* if it exists; otherwise the index block entry will be equal to *null*. We use another circuit to search the level 1 *tree-vector* and the search path is determined by the value of bits 10-16. The same operation is applied to search the level 2 *tree-vectors*. Since the level 3 *index block* entries are very sparsely populated, we divided the IB3 into 128K disjoint 256-entry segments. Only non-empty segments, i.e. segments with at least one entry not equal to *null*, will be stored. The starting address of the IB3 segment is stored in the corresponding IB2 entry. Hence, each IB2 entry will have two fields, namely the address of the level 2 *tree-vector*, and the address of the IB3 segment, *nextIB*. When we look up IB2 using bits 1-17 of the destination address and find *nextIB* is not equal to null, we extract bits 18-25 of the destination address and use them to index into the IB3 segment to locate the level 3 *tree-vector*.

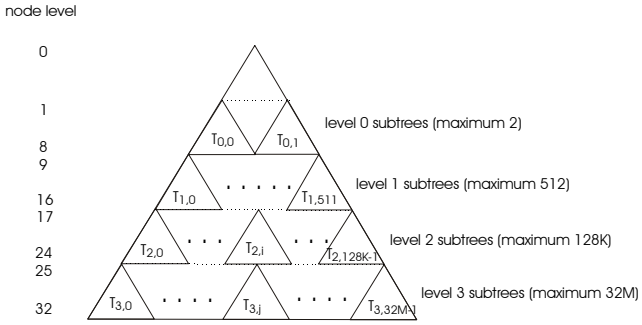

Figure 5. Partitioning of the binary-trie into non-overlapping 255-node subtrees.

To allow pipelined processing, the level 1 *tree-vectors* and *routing-vectors*, level 2 *tree-vectors* and *routing-vectors*, level 3 *tree-vectors* and *routing-vectors*, level 2 index block and level 3 index block are placed in separate memory modules. We assume that all the 512 level 1 *tree-vectors* are present, hence, IB1 can be replaced by the address decoder of the memory module of the level 1 *tree-vectors*. The computation can be implemented using a five-stage pipeline as shown in Figure 7. The computation time of each stage depends mainly on the access time of the memory modules of the *tree-vectors*, *routing-vectors* and index blocks. If SRAMs are used, then the pipeline cycle time can be as

small as 12.5ns. If DRAMs are used, the pipeline cycle time will be about 50ns.

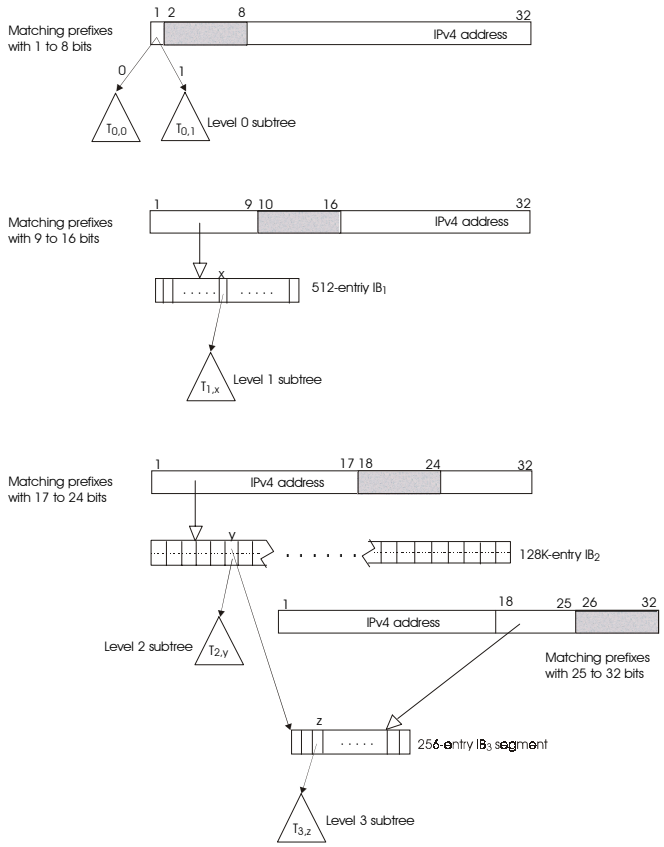

Figure 6. The hardware indexing structure

3.1 Memory Requirement

In the proposed architecture, the memory required for the level 2 and level 3 tree-vectors, and the level 3 index block segments depend on the distribution of the prefixes. Let's define 2^{17} buckets labeled from 0 to $2^{17}-1$ and distribute the prefixes that have 17 to 24 bits by the value of bits 1-17. The number of level 2 tree-vectors will be given by the number of non-empty buckets. Similarly by grouping prefixes with 25 to 32 bits by the first 17 bits will give us the number of IB3 segments, and grouping prefixes with 25 to 32 bits by the first 25 bits will give us the number of level 3 *tree-vectors*. Table 1 shows the grouping of the forwarding table prefixes of typical backbone routers [6].

Assume 8 bits are sufficient to designate the next-hop, then the total memory for a pair of *tree-vector* and *routing-vector* will be 255×9 bits. In the classful addressing scheme, only the Class C addresses are longer than 16 bits. Since Class C addresses always start with the prefix '110' and we build the level 2 index block using the first 17 bits, all Class C network addresses will be included in no more than 16K distinct level 2 *tree-vectors*. Suppose the hardware supports 16K level 2 *tree-vectors*, 4K level 3 *tree-vectors* and 4K level 3 index block segments. The total memory for storing the 4 levels of *tree-vectors* and *routing-vectors* is equal to (512 + 16K + 4K)×255×9 bits = 5.74M bytes. An entry of the level 2 index block is consisted of an 1-bit empty flag, a 14-bit L2 *tree-vector* address and a 12-bit level 3 IB segment address. An entry of the level 3 index block has an 1-bit empty flag and a 12-bit level 3 *tree-vector* address. Hence, the total memory for the level 2 and level 3 index blocks is equal to 128K × 22 bits + 4K × 256 × 13 bits = 2.05M bytes. The overall memory required is about 7.8M bytes. In Table 1, we also show the actual amount of memory consumed by the five forwarding tables.

stage	computation
1	• search one of the two level 0 tree-vectors • read the value of the level 1 tree-vector • access IB2 to obtain the address of the level 2 tree-vector and the IB3 segment, if exists
2	• read the routing information associated with the search result of stage 1, if required • search the level 1 tree-vector obtained in stage 1; and if successful, over-write the search result of stage 1 • read the value of the level 2 tree-vector, if exists • access the IB3 segment, if exists
3	• read the routing information associated with the search result of stage 2, if required • search the level 2 tree-vector, if exist; and if successful, over-write the search result of stage 2 • read the value of the level 3 tree-vector, if exists
4	• read the routing information associated with the search result of stage 3, if required • search the level 3 tree-vector, if exist; and if successful, over-write the search result of stage 3
5	• read the routing information associated with the search result of stage 4, if required • send the final result to the output interface

Figure 7. Organization of the pipeline.

Router	total no. of prefixes	no. of prefixes with ≥ 25 bits	no. of L2 tree-vectors	no. of IB3 segments	no. of L3 tree-vectors	memory consumed (Mbyte)
aads	29831	102	5690	33	70	2.15
mae-east	23729	103	5110	37	74	1.99
mea-west	34230	94	6060	34	65	2.25
pacbell	41811	139	6847	56	96	2.5
paix	16445	155	3710	61	101	1.63

Table 1. Grouping of prefixes in backbone routers (based on data collected on March 6, 2001)

3.2 Forwarding Table Updates

In our design, forwarding table entries are not replicated. Updating of the forwarding table is relatively easy. We consider three types of update operations, (i) modifying the next-hop of an existing entry, (ii) inserting a new entry, and (iii) deleting an existing entry. Since the costs for searching and memory allocation/de-allocation of fixed size segments are bounded, we only consider the number of memory- write operation required in updating the data structure in the following discussion. To modify the next-hop of an existing entry is the easiest. It only involves changing the corresponding routing-vector entry. The number of memory-write operation required is only one. To insert a new entry to the forwarding table, there can be two possibilities. If the pair of tree-vector and routing-vector to host the new entry already exists, then we only need to set the corresponding bit in the tree-vector and write the next-hop identifier to the routing-vector. In this case, the number of memory- write operations required is two. If the pair of tree-vector and routing-vector for the new entry does not exist, then we need to request the memory management unit to allocate the memory for the corresponding tree-vector, routing-vector and index segment, if necessary. In the worst case, the new prefix has 25 bits or more. We may need to create the IB3 segment in addition to the tree-vector and routing-vector and update the IB2 entry. In this case, the number of memory-write operations is four. To delete an existing entry, we only need to clear the corresponding bit in the tree-vector that hosts the prefix to be removed. If after the removal, the tree-vector becomes empty, then we should de-allocate the tree-vector and routing-vector and update the index block. In the worst case, the prefix to be removed is in the level 3 tree-vector and the IB3 segment becomes empty after the removal. In this case, we need to de-allocate the IB3 segment and update the corresponding IB2 entry. Three memory-write operations are required.

4. CONLULDING REMARKS

In this paper, we present a novel hardware architecture for fast IP address lookup. We model the address lookup problem as a searching problem on a binary-trie. The binary-trie can be represented very efficiently. A node in the binary-trie only requires 9 bits of memory. The complete binary-trie is partitioned into four levels of fixed size, 255-node, non-overlapping subtrees. The maximum number of subtrees in the four levels are 2, 512, 128K and 32M, respectively. The storage for the level 2 and level 3 subtrees are allocated on demand, i.e. only non-empty subtrees are included in the data structure. The total amount of memory required for a reasonable implementation is about 8M bytes. The whole IP address lookup engine can be implemented in a single VLSI chip. The searching time is speeded up by using the parallel searching approach and a hierarchical indexing structure. Together with a pipelined architecture, we can achieve a throughput of 80 millions and 20 millions lookup per second if the major memory modules are implemented using SRAM and DRAM, respectively.

Previous hardware designs can also achieve a throughput of one lookup per memory access [4,5]. However, those designs require high degree of replication of forwarding table entries. This makes the forwarding table updates difficult and limits their applicability to IPv4 only. A distinguishing feature of our design is that the forwarding table entries are not replicated in the data structure. Hence, the forwarding table can be updated in constant time.

Currently, we are investigating the application of our design to IPv6. If this approach is extended to IPv6, we may have 14 levels of 255-node subtrees. The memory requirement will be a major concern. With the exponential growth in the address space from 32-bit to 128-bit, the density of the prefixes in the binary tire will be reduced substantially. Most of the subtrees and the IB segments will be very sparsely populated. Currently, we are looking into three different approaches to reduce memory requirement:

(i) use the path compression technique to skip the intermediate levels of single path routes in the binary trie;
(ii) compact the *routing-vectors*; and
(iii) compact the IB segments.

In the present design, we use the position of the rightmost bit in the *result-vector* to index into the *routing-vector*. If we can find the number of 1's in the *tree-vector* from the left up to the position of the rightmost 1 in the *result-vector*, we can throw away the null entries in the *routing-vector*. Given the fact that we can find the position of the rightmost bit in 12.5ns, we are confident that the all the required computation can be done within the DRAM access time.

References

1. A. Broder and M. Mitzenmacher, “Using Multiple Hash Functions to Improve IP Lookups”, IEEE INFOCOM, pp. 1454-1463, 2001.
2. Tzi-cker Chiueh and P. Pradhan, “High-Performance IP Routing Table Lookup Using CPU Caching”, IEEE INFOCOM, pp. 1421-1428, 1999.
3. W. Doeringer, G. Karjoth and M. Nassehi, “Routing on Longest-Matching Prefixes”, IEEE/ACM Transactions on Networking, Vol. 4, No. 1, pp. 86-97, Feb. 1996.
4. P. Gupta, S. Lin and N. McKeown, “Routing Lookups in Hardware at Memory Access Speeds”, IEEE INFOCOM, pp. 1240-1247, 1998.
5. Nen-Fu Huang and Shi-Ming Zhao, “A Novel IP-Routing Lookup Scheme and Hardware Architecture for Multigigabit Switching Routers”, IEEE Journal on Selected Areas in Communications, Vol. 17, No. 6, pp. 1093-1104, June 1999.
6. Internet Performance Measurement and Analysis Project, University of Michigan and Merit Network, URL: http://www.merit.edu/ipma
7. S. Nilsson and G. Karlsson, “IP-Address Lookup Using LC-Tries”, IEEE Journal on Selected Areas in Communications, Vol. 17, No. 6, pp. 1083-1092, June 1999.
8. M. A. Ruiz-Sanchez, E. W. Biersack and W. Dabbous, “Survey and Taxonomy of IP Address Lookup Algorithms”, IEEE Network, pp. 8-23, March/April 2001.
9. Henry H. Y. Tzeng and Tony Przygienda, “On Fast Address-Lookup Algorithms”, IEEE Journal on Selected Areas in Communications, Vol. 17, No. 6, pp. 1067-1082, June 1999.
10. M. Waldvogel, “Multi-Dimensional Prefix Matching Using Line Search”, IEEE Conf. on Local Computer Networks, pp. 200-207, 2000.
11. N. Yazdani and P. S. Min, “Fast and Scalable Schemes for the IP Address Lookup Problem”, IEEE Conf. on High Performance Switching and Routing, pp. 83-92, 2000.

0-7803-7476-2/02/$17.00 (c) 2002 IEEE.