An On-Chip IP Address Lookup Algorithm

Xuehong Sun and Yiqiang Q. Zhao, *Member*, *IEEE*

Abstract—This paper proposes a new data compression algorithm to store the routing table in a tree structure using very little memory. This data structure is tailored to a hardware design reference model presented in this paper. By exploiting the low memory access latency and high bandwidth of on-chip memory, high-speed packet forwarding can be achieved using this data structure. With the addition of pipeline in the hardware, IP address lookup can only be limited by the memory access speed. The algorithm is also flexible for different implementation. Experimental analysis shows that, given the memory width of 144 bits, our algorithm needs only 400kb memory for storing a 20k entries IPv4 routing table and five memory accesses for a search. For a 1M entries IPv4 routing table, 9Mb memory and seven memory accesses are needed. With memory width of 1,068 bits, we estimate that we need 100Mb memory and six memory accesses for a routing table with 1M IPv6 prefixes.

Index Terms—Algorithms, hardware, tree data structures, range search, IP address lookup, on-chip memory.

1 Introduction

The Internet system consists of Internet nodes and transmission media which connect Internet nodes to form networks. Transmission media are responsible for transferring data and Internet nodes are responsible for processing data. In today's networks, optical fibers are used as transmission media. Optical transmission systems provide high bandwidth. It can transmit data in several gigabits (OC48=2.4Gb/s and OC192=10Gb/s are common and OC768c=40Gb/s is the goal of the near future) per second per fiber channel. Dense Wavelength Division Multiplexing (DWDM) [11] technology can accommodate about 100 channels (2004) and possibly more in the future in one strand of fiber. This amounts to terabits per second transmission speed on optical fiber. In order to keep pace with this speed, the Internet nodes need to achieve the same speed of processing packets. The Internet nodes implement the functions incurred by the Internet system. The four main tasks of the Internet nodes are IP address lookup (and/or packet classification), packet modification, queue/policy management, and packet switching.

Given the smallest packet size of 40 bytes (worst case), in order to achieve 40 Gigabits per second (OC768) wire speed, the router needs to lookup packets at a speed of 125 million packets per second. This, together with other needs in processing, amounts to less than 8ns per packet lookup. Nowadays, one access to on-chip memory takes 1-5ns for SRAM and about 10ns for DRAM. One access to off-chip memory takes 10-20ns for SRAM and 60-100ns for DRAM. This figure shows that the development of high-speed IP lookup algorithms which can be implemented on chip is in great demand. It also shows that it is very difficult for serial algorithms to achieve the ideal wire speed. Developing algorithms that integrate parallel or pipeline mechanisms into hardware seems a must for future Internet Protocol (IP) address lookup.

Industry responded to the aforementioned demand by offloading the processing tasks to coprocessors [20]. A coprocessor is a system on a single chip to perform a single task. Previously, the general purpose processor was used in the Internet nodes. Recently, Network Processors [23] are gaining popularity in processing Internet data. The trend is to use a chip to exclusively perform the IP lookup task. As mentioned above, one of the advantages of an on-chip system is that the on-chip memory access latency is very low. Another advantages of on-chip systems is larger bus width to on-chip memory than that to off-chip memory. The number of pins of the chip is smaller if the memory goes on chip rather than off chip. The memos width to on-DRAM can be more than 1,000. The number of pins for off-chip memory would increase a lot with the same memory width.

The restriction of an on-chip system is that the memory cannot be large. Reference [19] shows that, for the embedded DRAM, the maximum macro capacity/size is 72.95 Mb/31.80mm^2 with random access time of 9.0ns using the Cu-08 process; for the embedded SRAM, the maximum macro capacity is 1 Mb with random access time of 1.25ns using the Cu-11 process.

We develop an IP address lookup algorithm which uses a small amount of memory. With this merit, the algorithm can be implemented in one single chip. Our approach is to convert the longest prefix match problem into a range search problem [9], [16], [10] and use a tree structure to do the search. Our main contribution is the development of a novel prefix compression algorithm for compactly storing IP address lookup table. The following techniques are used in the compression algorithm: 1) We compress the keys in a tree node, 2) we use a shared pointer in a tree node, and 3) we use a bottom-up process from the leaf to the root scheme to build the tree.

The rest of the paper is organized as follows: In Section 2, IPv4 and IPv6 address architectures are introduced. Section 3 gives a hardware design reference model for our analysis. In Section 4, we give the concepts and definitions

- *X. Sun is with the Canadian Space Agency and can be reached at 311 200 De Gaspe, Verdun, Quebec Canada H3E 1E6. E-mail: xsun@math.carleton.ca.*
- *Y.Q. Zhao is with the School of Mathematics and Statistics, Carleton University, 1125 Colonel By Drive, Ottawa, Ontario Canada K1S 5B6. E-mail: zhao@math.carleton.ca.*

Manuscript received 5 Mar. 2004; revised 3 Dec. 2004; accepted 26 Jan. 2005; published online 16 May 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0075-0304.

related to the range search problem. Details of our new algorithm are presented in Section 5. Results from an experimental study are presented in Section 6. In Section 7, we highlight comparison results with some existing algorithms. Concluding remarks are made in Section 8.

2 IP Address Lookup Problem

Internet Protocol (IP) defines a mechanism to forward Internet packets. Each packet has an IP destination address. In an Internet node (Internet router), there is an IP address lookup table (forwarding table) which associates any IP destination address with an output port number (or next-hop address). When a packet comes in, the router extracts the IP destination field and uses the IP destination address to look up the table to get the output port number for this packet. The IP address lookup problem is to study how to construct a data structure to accommodate the routing table so that we can find the output port number quickly.

Since the IP addresses in a lookup table have special structures, the IP address lookup problem can use techniques that are different from that used to solve general table lookup problems by exploiting the special structures of the IP addresses. Nowadays, IPv4 address is used. IPv6 address could be adopted in the future. We next introduce these two address architectures.

2.1 IPv4 Address

An IPv4 address is 32 bits long. It can be represented in dotted-decimal notation: 32 bits are divided into four groups of 8 bits with each group represented as decimal and separated by a dot. For example, 134.117.87.15 is the IP address for a computer at Carleton University, Canada. (Sometimes we use a binary or decimal representation of an IP address for other purposes.) An IP address is partitioned into two parts: a constituent network prefix (hereafter we call it prefix) and a host number on that network. The Classless Inter-Domain Routing (CIDR) [12] uses a notation to explicitly mention the bit length for the prefix. Its form is "IP address/prefix length." For example, 134.117.87.15/24 represents that 134.117.87 is for the network and 15 is for the host. 134.117.87.15/22 represents that 134.117.84 is the network and 3.15 is the host. (We need some calculation here. Essentially, we have 22 bits as prefix and 10 bits as host. So, 134.117.87.15 (10000110 01110101 01010111 00001111) is divided into two parts: 10000110 01110101 010101* (134.117.84) and 11 00001111(3.15)). Sometimes, we use a mask to represent the network part. For example, 134.117.87.15/255.255.255 is equal to 134.117.87.15/24 since the binary form of 255.255.255 is 24 bits of 1s (note that the binary form of 255 is 11111111). 134.117.87.15/255.255.253 is equal to 134.117.87.15/22 since the binary form of 255.255.253 is 22 bits of 1s.

We can look at the prefix from another perspective. The IPv4 address space is the set of integers from 0 to $2^{32}-1$ inclusive. The prefix represents a subset in the IPv4 address space. For example, 10000110 01110101 010101* (134.117.84) represents the integers between 2255836160 and 2255837183 inclusive. We will define the conversion in a later section. The longer the prefix is, the smaller the subset is. For example, 10000110 01110101 010101* (length 22) has $2^{10} = 1,024$ IP addresses in it; 10000110 01110101 01010111* (length 24) has only $2^8 = 256$ IP addresses in it. We can also see that, if an address is in 10000110 01110101 01010111*, it is also in 10000110 01110101 010101*. We say 10000110 01110101 01010111* is more specific than 10000110 01110101 010101*. IP address lookup is to find the most specific prefix that matches an IP address. It is also called longest prefix match (because the longer the prefix is, the more specific it is).

2.2 IPv6 Address

The research of next-generation Internet protocol (IP) IPv6 [2] was triggered by solving the IPv4 address space exhaustion problem, among other things. In IPv6, the IPv6 addressing architecture [6] is used.

An IPv6 address is 128 bits long. An IPv6 address is represented as text strings in the form of x:x:x:x:x:x:x:x, where the xs are the hexadecimal values of the eight 16-bit pieces of the address. For example, "FE0C:BC98:7654:A210:FEDC:B098:7054:3A10." Another example is "2070:0:0:0:8:80:200C:17A." Note that it is not necessary to write the leading zeros in an individual field. We can use "::" to indicate multiple groups of 16-bits of zeros and to compress the leading and/or trailing zeros in an address, but it can only appear once in an address. For example, "2070:0:0:0:8:80:200C:17A" may be represented as "2070::8:80:200C:17A." "0:0:0:0:1:0:0:0" as "::1:0:0:0" or "0:0:0:0:1::" but not as "::1::." When dealing with a mixed environment of IPv4 and IPv6 nodes, the form of x:x:x:x:x:x:d.d.d.d is used, where the xs are the hexadecimal values of the six high-order 16-bit pieces of the address and the ds are the decimal values of the four low-order 8-bit pieces of the address (standard IPv4 representation). For example, "0:0:0:0:0:ABCD:129.140.50.38" or, in compressed form, "::ABCD:129.140.50.38."

A form similar to the CIDR notation for IPv4 addresses is used to indicate the network part of an IPv6 address. For example, "1200:0:0:CD30:1A3:4567:8AAB:CDEF/60" represents the first 60 bits are network part and the other 68 bits are host part. It also represents a 60 bits length prefix.

In the following sections, we use IPv4 addresses as an example to explain the concept for purpose of simplicity.

3 A Hardware Design Reference Model

Fig. 1 is a reference model for the hardware design. The left part is a chip and the right part is external SRAM memory. The IP destination address enters the chip as a key for looking up the next hop information. The output of the chip is an index to the external SRAM where the port information can be found. The chip is an ASIC that consists of a memory system and an ALU part. The memory system uses on-chip SRAM. Using an IBM blue logic Cu-08 ASIC process, the I/O width of the SRAM to the control logic unit can be as wide as 144 bits. The memory access time can be as low as 1.25ns. The size of the on-chip memory is about 1 Mbits. Assuming there are 144 bits in each row, the chip has less than 2^{13} rows altogether. This means 13 bits is enough to index into any row of the memory. The ALU receives keys from outside and produces outputs to the outside. It may access the memory system and performs some simple logic and arithmetic operations. According to different design goals, the chip can be configured or programmed. In fact, this reference model can be modified to be tailored to different situations.

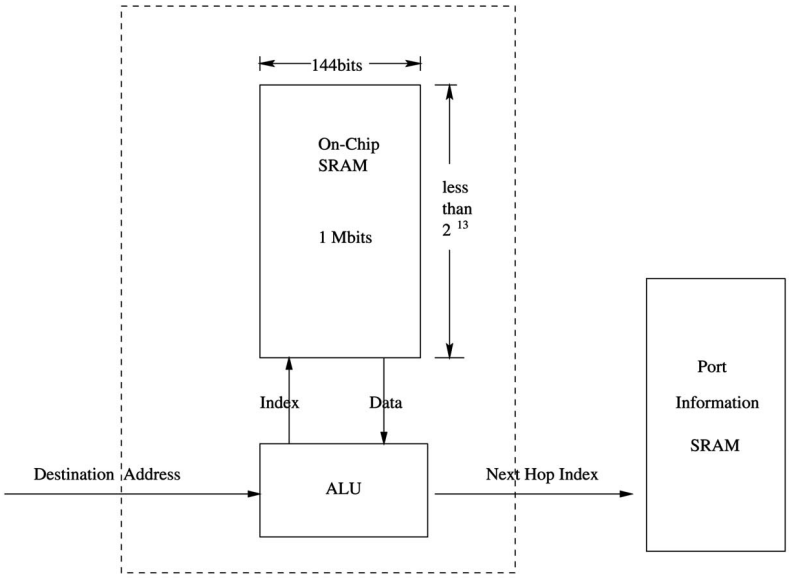


Fig. 1. Hardware design reference model.

The reason to choose SRAM instead of DRAM as the on-chip memory is that the SRAM access time is several times faster than the DRAM. The on-chip SRAM memory size can be made more than 100M bits large, which is well suitable for any IP lookup task.

4 Convert Longest Prefix Match to Range Search Problem

In this section, we will give some more definitions related to converting a prefix to a range of addresses. We use a small artificial routing table in Table 1 as an example. We first give the following definition.

Definition. *A prefix P represents address(es) in a range. When an address is expressed as an integer, the prefix P can be represented as a set of consecutive integer(s), expressed as $[b, e)$, where b and e are integers and $[b, e) = \{x : b \leq x < e, x$ is an integer$\}$. $[b, e)$ is defined as the* range *of the prefix P. b and e are defined as the* left endpoint *and* right endpoint *of the prefix P, respectively, or* endpoints *of the prefix P.*

For example, for the 6-bit-length addresses, the prefix 001* represents the addresses between 001000 and 001111 inclusive (in decimal form, between 8 and 15 inclusive). $[8, 16)$ is the *range* of the prefix 001*. 8 and 16 are the *left endpoint* and *right endpoint* of the prefix 001*, respectively. Readers may notice that the definition of right endpoint is different from that in the literature (e.g., [13]). With our definition, each endpoint can have at least as many trailing zeros as the length of the host part of the IP address. As in the example given above, 16 has four trailing zeros in its binary form, while 15 has no trailing zeros in its binary form. In real-life forwarding tables, most of the prefixes have a length of 24 bits. Thus, most of the endpoints have at least 8 bits of trailing zeros. We will exploit this property using the tree structure in the following section.

TABLE 1
A Mini Routing (Maximum Prefix Length = 6)

prefix	port	prefix	port
0*	A	10*	C
00010*	B	10001*	A
001*	C	1001*	A
01*	C	1011*	D
011000	D	11*	D
0111*	A	110*	A
1*	B	1101*	B

Two distinct prefixes may share at most one endpoint. For example, in Table 1, prefix 001* has endpoints 8 and 16 and prefix 01* has endpoints 16 and 32. They share the same endpoint 16. Since each prefix can be converted to two endpoints, N prefixes can be converted to at most $2N$ different endpoints. Fig. 2 is the mapping of prefixes in Table 1 to endpoints and ranges. Notice that 14 prefixes produce 17 endpoints in this example.

If two consecutive ranges have the same port, the shared endpoint can be eliminated. Hence, the two ranges can be merged into one range. For example, since range $[34, 36)$ and range $[36, 40)$ both map to port A, we can merge them into the range $[34, 40)$. We can assign a unique port to each range according to the rule of longest prefix match. We can use an endpoint to represent the range to its right and, thus, assign the port of the range to the left endpoint. For example, let a be an endpoint and b its successor. If port A is assigned to a, it means any address that is in $[a, b)$ is mapped to port A. The algorithm to convert the routing table into (endpoint, port) tuples is given in the Appendix. Notice that, for simplicity, this algorithm does not do the possible merge of endpoints mentioned above. In fact, this can be easily done by using a variable to record the port assigned to the previous endpoint. Whenever there is a port assignment, we check the port with the recorded port. If they are equal, this endpoint will be eliminated; otherwise, assign the port as usual. Fig. 3 is the result of the conversion

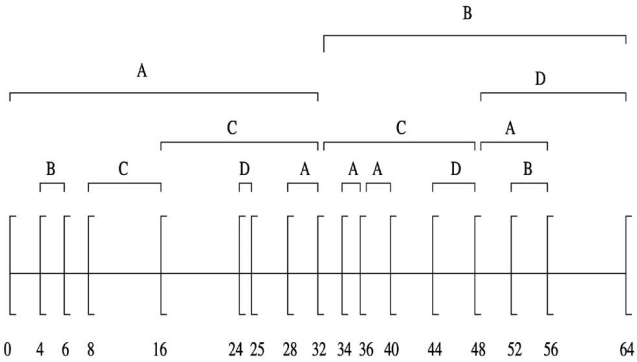


Fig. 2. Endpoints converted from prefixes and the corresponding ranges and ports.

of the routing table in Table 1 into (endpoint, port) tuples. With the merging of ports, the number of endpoints is reduced from 17 to 15.

The set of endpoints from Fig. 3 is {0, 4, 6, 8, 16, 24, 25, 28, 32, 34, 36, 40, 44, 48, 52, 56, 64}. Usually, 0 and the maximum value 64 can be eliminated. With the above process, the IP address lookup problem is converted into the *predecessor finding* problem in a priority queue [4]. The *predecessor finding* problem is defined as follows: Given a set of sorted integers S and an integer i, find an integer e in S such that e is the biggest one which is less than or equal to i. One of the solutions is to use a tree structure to tackle the predecessor finding problem. Fig. 4 is an example of the tree structure created from Fig. 3. We start the search from the root, then go through internal nodes until we reach a leaf. The leaf has a pointer which leads to the corresponding port information. For example, in order to look up 39, we search in the root and find that 34 is the the biggest one which is less than or equal to 39. Following the pointer, we search the third node in the next stage. We find 39 is less than all the stored values. It points to port A.

The data structure we propose is essentially a tree structure, too. However, the details and the techniques are different from that in the literature (e.g., [9], [16], [10]). In order to be convenient for explanation, we divide the tree into levels. The root is level one. The next stage nodes are defined as level two and so on. The leaves are the last level. There are two levels in Fig. 4.

5 Put the Tree in Memory

In this section, we will explain how to put the range tree structure into memory. This is considered the main contribution in this algorithm. We will discuss how to compress the endpoints, how to use the shared pointer in a node, and how to use a bottom up approach to build a range tree in the memory.

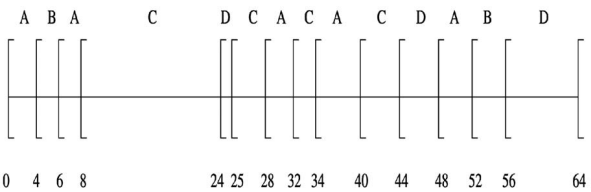


Fig. 3. Endpoints with the corresponding ports obtained by the longest-prefix match rule.

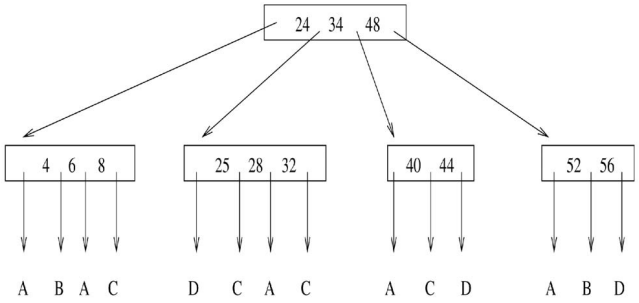


Fig. 4. A tree structure.

5.1 Compressed Endpoints and Shared Pointers

Let us take a close look at the endpoints of IPv4 addresses from Table 2. This is a set of consecutive endpoints that are taken from a routing table. We can find that all the endpoints have the same head bits "110000001110110" (15 bits) and each has at least eight trailing zeros. This is the source of compression. We can compress the common leading bits and trailing zeros. The more common leading bits and trailing zeros there are, the better the endpoints are compressed. The number of common leading bits is related to the number of endpoints in a forwarding table. Intuitively, the larger the number of endpoints is in a fixed address space, the larger the number of common leading bits tends to be since the endpoint space is fixed and the endpoints tend to cluster together when the number of endpoints is large. Human being assigned numbers such as IP prefixes tend to be correlated. For example, consecutive numbers tend to be assigned. So, endpoints from real-life IP tables tend to have a large number of common leading bits. We will verify this by experimental study.

Assume the endpoints in Table 2 are all the endpoints stored in a same node of the tree structure. We create a data structure, shown in Fig. 5, to represent the node. In this data structure, we use two fields to indicate the number of common bits to skip and the number of trailing zeros that each endpoint at least has. So, we only need to store the middle bits of each endpoint as a key for the endpoint. In this example, we only need to store a nine-bit key for each endpoint instead of an endpoint of 32 bits. Referring to the hardware design model in Fig. 1, there are 144 bits in each memory row. Each row is used to store a node of the tree. With the endpoints compressed, more endpoints can be stored in a node using 144 bits. Fig. 5 is the data structure for the 144 bits. We explain it in detail in the following: The

TABLE 2
Real-Life IPv4 Endpoints

11000000 11101100 01000000 00000000
11000000 11101101 00100000 00000000
11000000 11101101 01000000 00000000
11000000 11101101 01110010 00000000
11000000 11101101 01110011 00000000
11000000 11101101 01110100 00000000

internal node or leaf bit	number of keys	number of skip bits	number of trailing zeros	key1 key2 key3	next tree pointer

Fig. 5. The row data structure for a node of the tree.

data structure has six fields. The first field is a one-bit field to indicate whether the node is an internal node or a leaf node. The second field is the number of endpoints stored in this node. If the number of endpoints stored in any node does not exceed 16, then 4 bits are needed for this field. The third field records the number of head bits to skip in the IP address (endpoint) and the fourth field is the number of zeros to ignore at the trail of the IP address (endpoint). In general, for IPv4 addresses, these two fields need at most five bits each. For IPv6 addresses, these two fields need at most seven bits each. We will further discuss how to reduce the number of bits for these fields in later sections. The next field is for the keys. The last field is the pointer indexing to next level node of the tree structure. We use 20 bits in this field for supporting 500k entry forwarding tables. This is because each entry can produce at most two endpoints and each endpoint has its corresponding port information. Thus, the external SRAM which stores port information can have as many as 1M entries in the worst case, which can be indexed by 20 bits. To sum up, for IPv4, the first four fields and the last field use 1 + 4 + 5 + 5 + 20 = 35 bits. The leftover 109 bits are used to store as many keys as possible. For IPv6, the first four fields and the last field use 1 + 4 + 7 + 7 + 20 = 39 bits. The leftover 105 bits are used to store as many keys as possible. The order of the fields is not important. This is the basic structure, but it can be modified for different variants. For example, the first field may be removed if possible. The pointer field must be at least 22 bits long if we want to support 2M entry forwarding tables. We will mention later that the pointer field in a different level node may have different bits. The 144 bits can be varied.

We may notice that each node has only one pointer rather than many pointers, as in the case shown in Fig. 4. This is where our shared pointer goes. This scheme saves memory tremendously. As defined in the previous section, the tree is divided into levels. The nodes in each level are stored in ordered consecutive rows, such as in Fig. 6. In this way, all the endpoints in a node can share one pointer. The exact pointer of a subtree can be determined by the position of the corresponding endpoint. For example, assume a node with sorted endpoints p_1, p_2, p_3, p_4. Traditionally, we would need five pointers. Here, we only need the pointer that points to the node whose endpoints are smaller than p_1. Since other nodes are stored in ordered consecutive rows, we can find the node of a search by knowing the position of the destination address in the searched node. For example, if the searched destination address is greater than or equal to p_3 but less than p_4, we can find the node by adding 3 to the stored pointer.

The search of the keys in a node is carried out in the registers (or using logic circuit). It is very fast compared to memory access. Also, the time can be overlapped with memory access. We can also use a binary search in the keys of a node to speed up the search in the registers.

Two functions are needed for endpoints compression: One is for calculating the number of the common leading bits of a group of endpoints and another is for calculating the common trailing zeros of the group. For calculating the number of common leading bits of the group, we only need to calculate the number of common leading bits of the smallest and the biggest endpoints in the group. The number of common trailing zeros can be calculated iteratively as follows: The trailing zeros of the first endpoint are counted and recorded as the candidate number of common trailing zeros. Then, the trailing zeros of the next endpoint are counted. If the number of the trailing zeros of the next endpoint is smaller than the candidate number, the candidate number is changed to the smaller one; otherwise, the candidate number is unchanged. This operation is performed for the left endpoints until we exhaust all the endpoints. Using this approach to find the number of common trailing zeros of n endpoints, $O(n)$ operations are needed. $O(2)$ operations are needed to find the number of common leading bits of any sorted group. This can also be done by ORing together all the bits of the endpoint and then counting trailing zeros (which is probably more efficient, both in hardware and software).

5.2 Build the Tree from the Bottom Up

Given a set of sorted endpoints, we next discuss how to create a tree structure from it. We adopt a bottom-up approach. First, we assign endpoints to leaf nodes and then to the next highest level until the root level. Beginning with the smallest value endpoint, we try to store as many endpoints to a node as possible. We use IPv4 for the explanation in the following. From the analysis of the previous subsection, we know that there are 109 bits to store the compressed keys. Since each endpoint is 32 bits long, we can store at least three endpoints ($3 * 32 = 96 < 109$) even without key compression. Therefore, we initially select the first four endpoints as a group and calculate the total bits of the compressed keys. If the total bits is bigger than 109, then we cannot store these four keys. Instead, we store the first

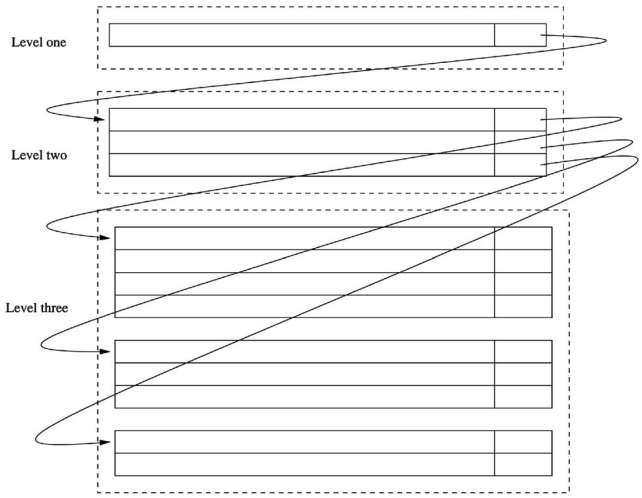


Fig. 6. Tree in memory.

three keys. If the total bits is equal to 109, we store these four keys. Otherwise, if the total bits is smaller than 109, we have the potential to store more keys. Thus, we need to probe further. We add the next endpoint to the group and repeat the above procedure. In this way, we can learn how many keys can be stored in the 109 bits.

Several variants are proposed in the following to describe the details of the tree creation. Specifically, they differ in how to calculate the compressed keys and select endpoints to store in the higher level nodes. They represent some trade-offs between the memory size and the number of memory accesses.

5.2.1 Variant One

Let $\{e_1, e_2, e_3, \cdots, e_n\}$ be the set of endpoints to be stored in a tree structure. Assume that the first four endpoints, $\{e_1, e_2, e_3, e_4\}$, are stored in the first leaf node, then the endpoint $\{e_5\}$ will be stored in the next higher level node. Assume the next five endpoints, $\{e_6, e_7, e_8, e_9, e_{10}\}$, are stored in the second leaf node, then the endpoint $\{e_{11}\}$ will be stored in the next higher level node and so on.

For this scheme, the endpoint(s) in the next higher level node must be involved in the leaf nodes to calculate the compressed keys. Specifically, in the aforementioned example, two endpoints, $\{e_1\}$ and $\{e_5\}$, are involved to find the the common leading bits of the first leaf node; $\{e_1, e_2, e_3, e_4\}$ are used to find the common trailing zeros of the first leaf node. Two endpoints, $\{e_5\}$ and $\{e_{11}\}$, are involved to find the common leading bits of the second leaf node; $\{e_6, e_7, e_8, e_9, e_{10}\}$ are used to find the common trailing zeros of the second leaf node; and so on. Next, we will explain why the higher level endpoints will be involved in the calculation of compressed keys using a concrete example.

For example, in Table 3 (the blank between bits is for convenience of reading), the first seven endpoints are stored in the first leaf node. The next endpoint, "10000110 11101111 00001101 10000000," will be stored in a higher level node. The next four endpoints following "10000110 11101111 00001101 10000000" will be stored in the next leaf node. "11111000 11110000 00010000 11000000" will be stored in a higher level node and so on.

The common leading bits of the first leaf node are "10000" instead of "1000000011." The number of common trailing zeros is eight. The common leading bits of the second leaf node are "1" (because "10000110 11101111 00001101 10000000" and "11111000 11110000 00010000 11000000" are involved to calculate the common leading bits) instead of "1101." The number of common trailing zeros is eight.

The reason is as follows: Let us assume that we are searching endpoint "10011111 11111111 11111111 00000000." This endpoint is greater than the endpoint "10000110 11101111 00001101 10000000" and less than the first endpoint in the second group. If we took "1101" as the leading bits of the second group, i.e., we skipped four bits, we would mistake "10011111 11111111 11111111 00000000" as greater than the last endpoint in the second group.

After constructing the leaf nodes, we proceed to the next level using the same method. The number of endpoints in this level are reduced to approximately N/k, where N is the number of endpoints in the leaf level and k is the average number of endpoints in a leaf node.

TABLE 3
Calculating the "Common Leading Bits"

10000000 11001000 01000000 00000000
10000000 11001001 00100000 00000000
10000000 11101101 01000000 00000000
10000000 11101101 01110010 00000000
10000000 11101101 01110011 00000000
10000000 11101101 01110100 00000000
10000000 11101101 01111101 00000000
10000110 11101111 00001101 10000000
11010110 11101111 00001110 00000000
11010110 11101111 00100111 00000000
11011000 11101111 00101000 00000000
11011000 11101111 00110000 00000000
11111000 11110000 00010000 11000000

As we can see, the cost for constructing the tree structure is not high. Let N be the total number of endpoints. Sorting the endpoints may take $O(N \log N)$ time. However, after the first sorting, we can incrementally add or delete an endpoint, which takes only $O(N)$ time using binary search. Assigning ports to endpoints takes $O(N)$ time. Creating the nodes takes $O(N)$ time. Putting them together, we need $O(N)$ time to create the tree structure.

The preferred architecture for a router is to use two banks of memory for IP packet forwarding. One bank is for updating and the other is for searching. One advantage of this architecture is that the updating will not interfere with the searching. With this architecture, an update can be performed in less than one second. This architecture is even comparable to some dynamic data structures in terms of memory utilization. For example, the memory utilization of a basic B-tree [1] is 50 percent in the worst case, which is the same as the two-banks-of-memory architecture. For a very fast update, say more than 10k updates per second, we need new techniques. This will be studied in our future work.

A word for IPv6: If the endpoints of an IPv6 address cannot be compressed, we cannot store even one address in a row. This is because an IPv6 address is 128 bits long and the key field of our data structure is only 105 bits. In this case, we may allow two rows to store the key with a minor modification of the data structure. IPv6 addresses may have consecutive zeros in the middle rather than in the trail. We may compress these zeros. However, we do not have enough IPv6 routing tables for the experiment, and we will not go into details. Nonetheless, experiments on small IPv6 routing tables are presented in this paper for the unmodified data structure.

5.2.2 Variant Two

The essential difference between variants one and two is that a new endpoint is created to store in the higher level node rather than an existing endpoint. We describe the algorithm first, then explain the reason to do so.

We first explain how to create the new endpoints for the high levels. Let $\{e_1, e_2, e_3, \cdots, e_n\}$ be the set of endpoints to be stored in a tree structure. Assume the first four endpoints, $\{e_1, e_2, e_3, e_4\}$, are stored in the first leaf node and the next five endpoints, $\{e_5, e_6, e_7, e_8, e_9\}$, are stored in the second leaf node and the next four endpoints, $\{e_{10}, e_{11}, e_{12}, e_{13}\}$, are stored in the third leaf node and so on. The first endpoint to be stored in the higher level node is simply the common leading bits of $\{e_1, e_2, e_3, e_4\}$ padded with trailing zeros to form a 32 bit endpoint $\overline{e}_1$. This new endpoint will be stored in the higher level node. The second endpoint is created according to e_4, e_5, and the number of common leading bits of $\{e_5, e_6, e_7, e_8, e_9\}$. Let n_1 be the number of common leading bits of e_4 and e_5; let n_2 be the number of common leading bits of $\{e_5, e_6, e_7, e_8, e_9\}$. Let $n_3 = max\{n_1 + 1, n_2\}$. Truncate the n_3 most significant bits of e_5 and padded with trailing zeros to form a 32 bit endpoint, $\overline{e}_2$. This procedure continues for all the leaf nodes left. When the endpoints for the higher level nodes are created, we can use the procedure recursively to create the tree structure.

We may notice that, when creating a new endpoint, essentially we make all the *unneeded* trailing bits zero. By *unneeded* trailing bits, we mean that these bits are not needed for a search. For example, assume that we need to search a key k with the two endpoints 10010101 and 11110001. In this case, we can compare the key k with a newly created endpoint 11000000. If $k > 11000000$, we search the 11110001 branch; otherwise, we search the 10010101 branch. The benefit is that 11000000 has more trailing zeros. The closer the endpoints are to the root, the more trailing zeros they have. We know that the trailing zeros help to compact more endpoints in the same memory. However, this does not mean that the total memory is reduced (explained later). This can help when the table is sparse or the prefix is long, such as IPv6. For dense IPv4, this gain does not compensate for the price of creating new endpoints.

With this scheme, the search procedure needs to be modified. For searching an endpoint e_0 in a node, the number of skip bits in the data structure is used to truncate the most significant bits of e_0 to form a search key k_0 which is padded with trailing zeros. The biggest key, k_i, in the node which is less than or equal to k_0 is found. This leads us to search in the next lower level node (root node) in subtree t_i. If $k_0 = k_i$, a search of the keys in the root node of this subtree is needed as usual. If $k_0 > k_i$, the endpoint e_0 is bigger than all the keys in this root node, thus a search is not needed.

In general, variant one uses less memory than variant two and variant two tends to need less memory accesses than variant one. We can explain this intuitively. Since the memory usage is dominated by the number of leaf nodes, assuming a leaf node can store five endpoints on average, variant one uses about $1/6$ less nodes than variant two. On the other hand, for variant two, when we approach the root, the endpoints in the nodes tends to have a larger number of trailing zeros and, therefore, the number of endpoints stored in a node tends to be larger. Thus, fewer memory accesses are needed. We believe that, for an IP table dominated by long prefixes, such as 32 length for IPv4 or 128 for IPv6, variant two is a better choice. If an IP table is dominated by short prefixes, variant one is a better choice.

5.2.3 Variant Three

To compromise between variant one and two, we may combine them to form a new variant. We may use variant one first. When we are in a level of the tree approaching the root, we may change to variant two. An experimental study shows that this is a good compromise of the two variants. In the experiment, variant one is used for leaf nodes. Variant two is applied to all higher level nodes.

5.2.4 Variant Four

Variant four is to use the front end array similarly to [9]. This essentially divides the endpoints into groups according to their leading bits. In each group, a respective tree structure is created using techniques described in this paper. The number of bits to choose to divide the table is a parameter to be optimized by the particular table. On one hand, the longer the number of bits is, potentially, the smaller the number of memory accesses is needed; on the other hand, the longer the number of bits is, the more groups will have empty or a small number of endpoints in it and, therefore, waste the memory. It is a trade-off between memory size and the number of memory accesses. The memory size is the largest among the four variants and the number of memory accesses is the smallest among the four variants.

As mentioned above, when the number of empty groups is large, a large amount of memory is wasted. To remedy the drawback in this situation, we can modify the algorithm. Instead of using the front end array, we may use a tree structure. The idea is that we truncate a number of leading bits of all the endpoints to form a set of new endpoints (with duplicates removed). With the set of new endpoints, variant one or two is used to create a search tree. The search result is a pointer to a subtree instead of a port entry. The subtree is created using variant one or two. In fact, this is a layered approach which can be applied recursively. It is effective for long bit endpoints such as IPv6 and sparse tables. In general, the cost of this is a bit of memory inefficiency.

For an extreme example, in Table 4, we are better off to divide the table into two obvious new tables and use 10000000 and 11111111 for a front search to lead to one of the tables.

5.3 Optimizations

Several suggestions for optimization for the data structure are provided in this section.

From Fig. 6, we can see the level one (root) always occupies one row of memory. The level two always starts from the next row. So, the pointer in the root always points to its next row and, therefore, is not necessary. This saves 20 bits for the root so that we can store potentially more endpoints in the root. Provided that the maximum number of keys stored in a node is 16, the number of nodes in the third level will be less than $17^2 = 289$, hence the pointer field in the level two needs 9 bits instead of 20 bits, and so on.

The next optimization step is to put the root in the registers. In practice, the number of endpoints in the root could be small. In that case, we can store these endpoints in the registers, thus saving one memory access.

Another optimization factor is for the skip bits field. We can define this field as the additional bits to skip relative to the previous level. For example, if we skip 2 bits in the first level and 7 bits in the second level, we only need to store

TABLE 4
An Example

10000000 00001000 01000000 00000000
10000000 10001001 00100000 00000000
10000000 11101101 01000000 00000000
10000000 11101101 01110010 00000000
10000000 11101101 01110011 00000000
10000000 11101101 01110100 00000000
10000000 11101101 01111101 00000000
11111111 00001111 00001101 10000000
11111111 00101111 00001110 00000000
11111111 10001111 00100111 00000000
11111111 10101111 00101000 00000000
11111111 11101111 00110000 00000000
11111111 11110000 00010000 11000000

$7 - 2 = 5$ as additional bits to skip. From the experiments, 3 bits is enough for IPv4.

6 Experimental Study

We download IPv4 routing tables from [21], [18] and IPv6 routing tables from [22] for the experiments. They are the basis for our experiments study. We plan three groups of the experiments. One is to use the original tables. Another is to create new tables by expanding original tables. The third one is to generate random tables. Results are analyzed in the following subsections.

6.1 Port Merge

If two consecutive intervals have the same port, the two consecutive intervals can be merged into one interval. This reduces the number of endpoints. Table 5 shows the port merge effect for IPv4 addresses. The *nonmerge rate* is equal to the number of endpoints with merge divided by the number of endpoints without merge. From the table, we can see that the merge effect is very significant. For example, for pacbell, the nonmerge rate is 57 percent. This implies that we can save about 43 percent memory if the merge effect is taken into account. We also conduct experiments by randomly assigning port numbers to prefix. The merge effect is not significant. This indicates that, in real-life tables, the port assignment has some correlation between neighboring intervals. By observing the port numbers in a real-life forwarding table, we can see the port numbers tend to be clustered. For IPv6, due to the lack of large real-life tables, we cannot analyze the effect. In the following experiments, we will not perform port merge for the set of endpoints.

6.2 Comparisons on Variants

Table 6 gives a general picture of the performance of the four variants. It shows the memory requirement and the number of memory accesses for the original tables under different variants. Columns with headings one, two, and three correspond to variants one, two, and three. The column with heading four is the result of variant four with 6 as the number of the front end bits. The column with heading five is the result of variant four with 14 as the number of the front end bits. We can see variant one uses the smallest amount of memory, but the number of memory accesses is the highest. Variant three is well balanced. Other experiments we conducted show similar results with one exception, that, when a table is dominated by short prefixes, variant one shows the best performance for both memory size and accesses. In order to avoid swamping pages due to huge experimental results, we choose variant three for the following experiments.

TABLE 5
Port Merge Effect

name	entries	endpoints without merge	endpoints with merge	non-merge rate
aads	17,487	27,250	16,807	62%
mae-east	18,661	29,871	19,410	65%
mae-west	29,608	45,826	27,792	61%
pacbell	24,193	37,771	21,389	57%
paix	15,332	24,126	16,836	70%

6.3 Results Using Real-Life Tables

This section shows the results using real-life tables. We first describe the characteristics of the forwarding tables. The seven IPv4 tables are dominated by 24 bit prefixes of more than 50 percent. The next largest number of prefixes are 23, 22, 19, 32, and 16 bits. The IPv6 tables are dominated by 48, 35, 28, and 24 bits of length. Some tables have as high as 70 percent 48 bit prefixes.

Table 7 is the result of IPv4 using variant three. The first column is the name for network access points (NAPs). The second column is the number of entries (prefixes) in the routing tables. The third column is the *ME ratio*. The ME ratio is defined as the number from dividing the memory requirement in bits by the number of prefixes. It measures the average number of bits needed for one prefix. The table shows that about 22 bits are needed for a prefix. Note that, if the port merge is taken into account, fewer bits are needed.

The *Arity* of a node is defined as the number of subtrees of the node. The average arity measures the average number of subtrees of all the nodes in the tree. The last column is the average arity from each routing table. From the average arity, we can roughly estimate the performance of a routing table with a different size. For example, assuming the average arity of 8, a routing table with 1M prefixes will need roughly $\log_8 1000000 \approx 7$ memory accesses.

The above results are obtained without using any optimizations mentioned in the previous section. We should mention that our data structure is also suitable for a pipeline implementation. In that case, the lookup speed is only restricted by the memory access latency.

Table 8 is the result of IPv6 using variant three. We use 168 bits memory width so that the key field is 128 bits long, which can accommodate at least one IPv6 endpoint. The low ME ratio (about 30) for t3 and t4 is probably due to a high percentage (above 70 percent) of length 48 prefixes. For other tables, no prefix has a percentage above 40 percent.

TABLE 6
Experiments with IPv4 Using 144 Bits Memory Width

name	memory (bits)					number of accesses				
	one	two	three	four/6	four/14	one	two	three	four/6	four/14
aads	400,896	423,360	403,776	435,744	2,677,248	6	5	5	4	3
mae-east	421,344	440,064	423,792	454,752	2,707,776	6	5	5	5	3
mae-west	644,256	672,912	648,864	690,768	2,929,824	6	5	5	5	3
pacbell	541,872	565,632	545,760	580,176	2,818,224	6	5	5	4	3
paix	359,712	379,152	362,016	391,824	2,649,600	6	5	5	5	3
Telstra	3,362,503	3,600,621	3,398,598	3,722,562	18,335,588	6	5	5	5	3
Reach	2,858,721	3,021,362	2,888,647	3,129,345	13,578,174	6	5	5	5	3

6.4 Results Using the Expanded IPv4 Tables

This section provides results for tables expanded from real-life tables. The six real-life IPv4 tables are combined and the duplicates are removed to form a large table as a basis. The size of the resulting table is 50,449. We can expand the basis table to a larger size table. The method of expanding the table is to pick a prefix of length 24 and expand it to $2^8 = 256$ prefixes of length 32. Given a size of a table we want to expand and the total number of prefixes of length 24, we can roughly figure what percentage of the prefixes of length 24 needs to be expanded. Then, we randomly select this percentage of prefixes of length 24 to expand. Using this method, we generated five tables of sizes of 66,004, 128,479, 258,784, 591,050, and 939,889. We have done experiments with different memory widths. We make the length of the key field in a node 32, 64, 128, 256, 512, or 1,024 bits. Assuming that the total length of the other fields is 36 bits for IPv4, the memory width comes to 68, 100, 164, 292, 548, or 1,060 bits, respectively. Table 9 shows the memory size as a function of table size and memory width. The result is also plotted in Fig. 7. From the results, we can see that 164 bits and 292 bits memory width are better choices than other memory widths for this set of tables. We can also see that the memory requirement increases proportionally to the increase of the number of prefixes, but at a small speed. For example, using a 164 bit memory width, the ME ratio is about 19.4 for tables of size around 50k; the ME ratio is only about 9.5 for tables of size around 1M.

TABLE 7
Results of IPv4 Using 144 Bits Memory Width

name	entries	ME ratio	average arity
aads	17,487	23.09	9.7
mae-east	18,661	22.71	10.2
mae-west	29,608	21.92	10.2
pacbell	24,193	22.56	10.0
paix	15,332	23.61	9.6
Telstra	161,684	21.02	10.7
Reach	132,446	21.81	10.2

The number of memory accesses is shown in Table 10. We can see it is not sensitive to the size of the tables. It indicates that our algorithm scales well to the size of the forwarding table.

6.5 Results Using the Expanded IPv6 Tables

Since we do not have large IPv6 tables, we randomly generate the tables and then expand them. The method is as follows: First, we randomly generate a table of size 10k with about 80 percent of prefix length 48 and lengths from 12 to 64 are uniformly distributed. A second table of size 10k is generated in the same way with about 80 percent of prefix length 64 instead of 48. Then, the two tables are combined with duplicates removed. This is the basis table for expansion which has about 40 percent of prefixes of length 48 and 64 each. When expanding, a prefix with length 64 is randomly selected and expanded to a number of prefixes of length 128. The exact number is randomly chosen between 128 and 1,024. The bits that expand from bit 65 to bit 128 are also randomly generated. Expansion continues until we reach the desired size of the table.

Using this scheme, we generated seven tables of sizes of 20,000, 31,284, 48,210, 144,124, 223,112, 530,601, and 1,208,666. The memory widths for this set of experiments are 168, 296, 552, and 1,064, respectively. Assuming that the total length of other fields is 40 bits for IPv6, the length of the key field in a node is 128, 256, 512, and 1,024, respectively. Shorter memory widths cannot accommodate a single IPv6 endpoint in the worst case and modification needs to be done.

TABLE 8
Results of IPv6 Using 168 Bits Memory Width

name	entries	memory (bits)	ME ratio	memory accesses
t1	58	3696	63.72	4
t2	181	11760	64.97	4
t3	534	17304	32.40	4
t4	488	15456	31.67	4
t5	248	15792	63.68	4
t6	284	16128	56.79	4

TABLE 9
Memory Size for IPv4 as a Function of Table Size and Memory Width

memory width (bits)	table size (number of prefixes)					
	50449	66004	128479	258784	591050	939889
68	1411000	1613300	2415700	4084828	8292736	12685196
100	1090300	1251300	1895400	3220800	6558500	10028200
164	978752	1123236	1687396	2860816	5822820	8903396
292	953088	1099964	1681044	2868608	5842920	8955640
548	987496	1144772	1755792	3028796	6201716	9477112
1060	1038800	1208400	1876200	3294480	6954660	10767480

Table 11 shows the memory size as a function of table size and memory width. The result is plotted in Fig. 8. From the results, we can see 1,064 bits memory width is the best choice among all memory widths for this set of tables. We can also see that the memory requirement increases proportionally to the increase of the number of prefixes. The ME ratio does not change much when the number of prefixes increases. For example, the ME ratio is 115.8 for tables of size of around 20k and 125.6 for tables of size of around 1M if the memory width is 1,064. All ME ratios are rather high compared with the small real-life tables. The reason for this is that the endpoints in real-life tables are correlated. The endpoints in randomly generated tables are not. For example, when a 64 bit prefix expands to 128, the trailing 64 bits are randomly generated. If we select 128 to 1,024 entries from a space of size of 2^{64}, it is rare that any two of them correlate. (We borrow the word *correlate* to roughly express the following idea: For example, we can say 11111100 and 11111111 correlate, but 10100100 and 11111111 not so much).

The number of memory accesses is shown in Table 12. Similarly to IPv4, it is not sensitive to the size of the tables.

7 Previous Work

Papers on address lookup algorithms are abound in the literature. It is not possible to mention all of them. We only compare with those that are similar to our approach. We also list papers that use a small amount of memory and omit those that use a large amount of memory. Surveys on address lookup algorithms were given in [14], [13]. Performance measurements for some of the algorithms are highlighted and compared as follows: Multiway search [9], [16] is the most similar approach to our method. However,

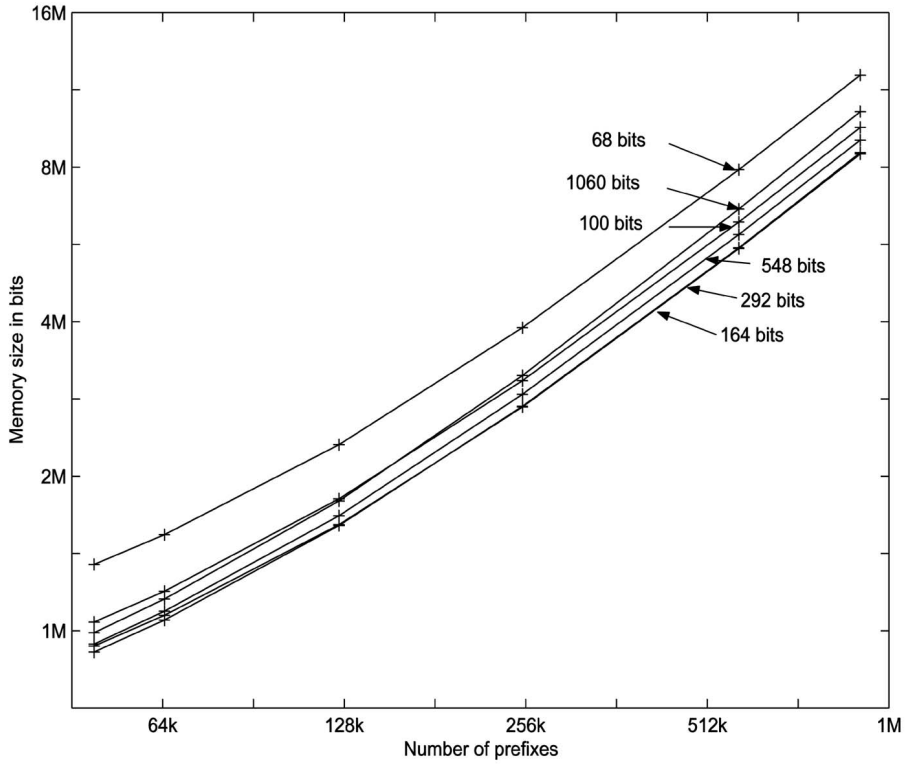


Fig. 7. Memory size for IPv4 as a function of table size with different memory widths.

TABLE 10
Number of Memory Accesses for IPv4
as a Function of Table Size and Memory Width

memory width	table size (number of prefixes)					
	50449	66004	128479	258784	591050	939889
68	9	9	10	10	10	11
100	6	7	7	8	8	8
164	5	5	6	6	6	6
292	4	5	5	5	5	5
548	4	4	4	4	4	4
1060	3	3	4	4	4	4

due to the lack of techniques to compress and optimize the data structures, larger memory requirements than ours are reported and the number of memory accesses is comparable to ours. Using a 32 byte cache line (which equals 256 bits memory width), a 6-way search can be done. For a routing table with over 32,000 entries, 5.6M bits memory is needed. In the worst case, four memory accesses are needed. If 256 bit memory width is used, we can report much better results than those in the previous section. Even with 144 bit memory width, we reported more favorable results than those from [9].

Reference [17] proposed an algorithm which requires a worst case of $\log W$ hash lookups, where W is the length of the address in bits. Hence, at most five hash lookups for IPv4 and at most seven hash lookups for IPv6 are needed. However, data from [9] showed that this algorithm needs 1,600K bytes for an IPv4 routing table with roughly 40k entries. In addition to that, the building of the data structure is not fast.

Reference [7] uses LC-trie for storing the table. They reported that about 4M bits are needed for the LC-trie. According to the paper, they need at least 100 bits for each prefix.

Reference [8] presented an IP address lookup scheme and a hardware architecture. The routing table is meant to put in off-chip memory, for they need 450-470k bytes memory for a routing table with 40,000 entries. They

TABLE 11
Memory Size for IPv6 as a Function of Table Size and Memory Width

memory width (bits)	table size (number of prefixes)						
	20000	31284	48210	144124	223112	530601	1208666
168	3078264	5032440	7962696	24559080	38229576	91463568	208804848
296	2644760	4343800	6890880	21319400	33197880	79448472	181433200
552	2390712	3881112	6115056	18782904	29196936	69782184	159235992
1064	2316328	3736768	5866896	17931592	27878928	66544688	151784920

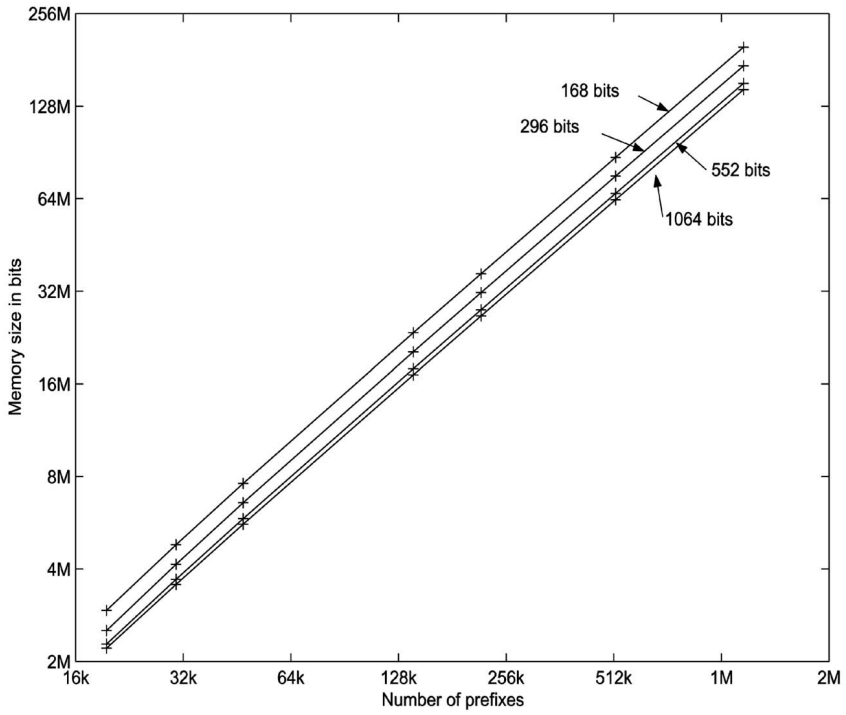


Fig. 8. Memory size for IPv6 as a function of table size with different memory widths.

TABLE 12
Number of Memory Accesses for IPv6 as a Function of Table Size and Memory Width

memory	table size (number of prefixes)						
width (bits)	20000	31284	48210	144124	223112	530601	1208666
168	6	7	8	8	8	9	9
296	5	7	7	7	7	8	8
552	5	5	6	6	6	7	7
1064	4	4	5	5	5	6	6

reported one to three memory accesses. They compared their scheme favorably with those proposed in [3] and [5].

Reference [15] was also aimed at hardware implementation. They used a compact stride multibit trie data structure to search the prefixes. They reported 4.3MB memory for a forwarding table of 104,547 entries. The algorithm allows for incremental updates, but is not scalable to IPv6.

Although [10] achieves $O(\log n)$ complexity for both update and search, the memory required is much larger than ours.

Table 13 shows roughly the bits consumed for each IPv4 prefix in the above-mentioned papers. These figures are derived from the experimental reports from each paper. They just provide a general picture, rather than an accurate measure, because data used for experiments, data structures, and resource assumption, etc., are different for each paper. For example, reference [15] used 19 bits for the next hop index field and we use 22 bits.

There are several papers, e.g., [9], [17], that deal with IPv6 address lookup. They have different merits from different perspective views of performance. Our algorithm has the advantage of using the lowest memory and the search speed is comparable to others.

8 Conclusion

We developed a novel algorithm which is tailored to hardware technology. The distinguishing merit of our algorithm is that it has a very small memory requirement. With this merit, a routing table can be put into a single chip, thus, the memory latency to access the routing table can be reduced. With the pipeline implementation of our algorithm, the IP address lookup speed can only be limited by the memory access technology.

Experiment analysis shows that, given the memory width of 144 bits, our algorithm needs only 400kb memory for storing a 20k entry IPv4 routing table and five memory accesses for a search. For a 1M entry IPv4 routing table, 9Mb memory and seven memory accesses are needed. With a memory width of 1,068 bits, we estimate that we need 100Mb of memory and six memory accesses for a routing table with 1M IPv6 prefixes.

TABLE 13
Bits Consumed for Each IPv4 Prefix

papers	[7]	[9]	[17]	[8]	[15]	this paper
ME ratio	100	175	320	90	329	15

Real-life tables have some structures. Modifying our algorithm to exploit these structures can achieve better performance.

Appendix

An Algorithm for Converting the Prefixes to Endpoints

Step 1: Sort the prefixes.

The host part of a prefix is filled with 0s and the prefix is treated as an integer. Prefixes are sorted according to the value of the integer. The smaller value prefix is sorted in the front. If two prefixes have the same value, the one whose length is smaller is sorted in the front.

Step 2: Assign ports to endpoints.

We have a stack. Let "max" be the maximum integer (endpoint) in the address space, "def" the default port for the address space, "M" the variable for the endpoint, and "N" the variable for the port.

```
N=def;
M=max;
push N;
push M;
For each prefix P=[b, e) with port p
in the sorted routing table {
  pop M;
  If (b<M) {
    assign p to b;
    push M; (push back M.)
    push p;
    push e;
  } else if (b=M) {
    assign p to b;
    while (b=M) {
      pop N;
      pop M;
    }
    push M; (M>b so push back.)
    push p;
    push e;
  } else if (b>M) {
    oldM=M;
    while (b>M) {
      pop N;
      if (oldM not = M) {
      (assign oldM'port to M.)
```

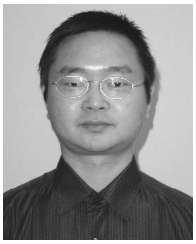
```
        pop M;
        pop N;  (get the next port.)
        assign port N to oldM;
        push N;
        push M; (push back N, M.)
      }
      oldM=M;
      pop M;
    }
    while (b=M) {
      pop N;
      pop M;
    }
    push M; (M>b so push back.)
    push p;
    push e;
}
```

ACKNOWLEDGMENTS

The authors would like to acknowledge Greg Soprovich of SiberCore Technologies for providing comments and suggestions for most of the experimental scenarios. A patent was filed based on this work.

REFERENCES

[1] R. Bayer and E. McCreight, "Organization and Maintenance of Large Ordered Indexes," *Acta Informatica,* vol. 1, no. 3, pp. 173-189, Sept. 1972.
[2] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," RFC 2460, Dec. 1998.
[3] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small Forwarding Tables for Fast Routing Lookups," *Proc. ACM SIGCOMM,* pp. 3-14, Sept. 1997.
[4] P. van Emde Boas, R. Kaas, and E. Zijlstra, "Design and Implementation of an Efficient Priority Queue," *Math. Systems Theory,* vol. 10, pp. 99-127, 1977.
[5] P. Gupta, S. Lin, and N. McKeown, "Routing Lookups in Hardware at Memory Access Speeds," *Proc. Infocom,* Apr. 1998.
[6] R. Hinden and S. Deering, "Internet Protocol Version 6 (IPv6) Addressing Architecture," RFC 3513, Apr. 2003.
[7] S. Nilsson and G. Karlsson, "IP Address Lookup Using LC-Tries," *IEEE J. Selected Areas in Comm.,* vol. 17, no. 6, pp. 1083-1092, June 1999.
[8] N.-F. Huang and S.-M. Zhao, "A Novel IP-Routing Lookup Scheme and Hardware Architecture for Multigigabit Switching Routers," *IEEE J. Selected Areas in Comm.,* vol. 17, no. 6, pp. 1093-1104, June 1999.
[9] B. Lampson, V. Srinivasan, and G. Varghese, "IP Lookups Using Multiway and Multicolumn Search," *IEEE/ACM Trans. Networking,* vol. 7, pp. 324-334, 1999.
[10] H. Lu and S. Sahni, "O(log n) Dynamic Router-Tables for Ranges," *Proc. IEEE Symp. Computers and Comm.,* pp. 91-96, 2003.
[11] R. Ramaswami and K.N. Sivarajan, *Optical Networks: A Practical Perspective.* San Francisco: Morgan Kaufmann, 1998.
[12] Y. Rekhter and T. Li, "An Architecture for IP Address Allocation with CIDR," RFC 1518, Sept. 1993.
[13] M.A. Ruiz-Sanchez, E.W. Biersack, and W. Dabbous, "Survey and Taxonomy of IP Address Lookup Algorithms," *IEEE Network,* vol. 15, no. 2, pp. 8-23, Mar./Apr. 2001.
[14] S. Sahni, K. Kim, and H. Lu, "Data Structures for One-Dimensional Packet Classification Using Most-Specific-Rule Matching," *Int'l J. Foundations of Computer Science,* vol. 14, no. 3, pp. 337-358, 2003.
[15] K. Seppanen, "Novel IP Address Lookup Algorithm for Inexpensive Hardware Implementation," *WSEAS Trans. Comm.,* vol. 1, no. 1, pp. 76-84, 2002.
[16] S. Suri, G. Varghese, and P. Warkhede, "Multiway Range Trees: Scalable IP Lookup with Fast Updates," *Proc. GLOBECOM,* 2001.
[17] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High Speed IP Routing Lookups," *Proc. ACM SIGCOMM,* pp. 25-36, Sept. 1997.
[18] http://bgp.potaroo.net/, 2003.
[19] http://www-3.ibm.com/chips/products/asics/products/ememory.html, 2003.
[20] http://www.linleygroup.com/, 2005.
[21] http://www.merit.edu/ipma/routing_table/, 2003.
[22] http://www.mcvax.org/~jhma/routing/ipv6/, 2005.
[23] http://www.npforum.org/org/, 2005.



Xuehong Sun received the PhD degree from Carleton University, Canada. He is a visiting fellow at the Canadian Space Agency and a member of AIAA. He is a cofounder of Sourithm Corp., which commercializes technologies through advanced algorithms for Internet Protocol (IP) address lookup and packet classification in Internet routers.

Yiqiang Q. Zhao (M'01) received the PhD degree from the University of Saskatchewan in 1990. After a two-year appointment as a postdoctoral fellow sponsored by the Canadian Institute for Telecommunications Research (CITR) at Queen's University, he joined the Department of Mathematics and Statistics of the University of Winnipeg as an assistant professor in 1992 and became an associate professor in 1996. In 2000, he moved to Carleton University, where he is now a full professor and the director of the School of Mathematics and Statistics. His research interests are in applied probability and stochastic processes, with particular emphasis on computer and telecommunication network and inventory control applications. He has published approximately 50 papers in refereed journals, delivered approximately 50 talks at conferences, and been invited more than 30 times to speak to seminars/colloquia or workshops. He has also had considerable experience interacting with industry. He has been the recipient of a number of grants from the Natural Sciences and Engineering Research Council of Canada (NSERC) and industries. He is currently on the editorial board of *Operations Research Letters*, *Queueing Systems*, *Stochastic Models*, and the *Journal of Probability and Statistical Science*. He is a member of the IEEE.