CHAPTER 2

IP ADDRESS LOOKUP

2.1 OVERVIEW

The primary role of routers is to forward packets toward their final destinations. To this purpose, a router must decide for each incoming packet where to send it next, that is, finding the address of the next-hop router as well as the egress port through which the packet should be sent. This forwarding information is stored in a forwarding table that the router computes based on the information gathered by routing protocols. To consult the forwarding table, the router uses the packet's destination address as a key – this operation is called *address lookup* [1]. Once the forwarding information is retrieved, the router can transfer the packet from the incoming link to the appropriate outgoing link.

Classful Addressing Scheme. IPv4 IP addresses are 32 bits in length and are divided into 4 octets. Each octet has 8 bits that are separated by dots. For example, the address 10000010 01010110 00010000 01000010 corresponds in dotted-decimal notation to 130.86.16.66. The bits in an IP address are ordered as shown in Figure 2.1, where the 1st bit is the most significant bit (MSB) that lies in the leftmost position. The 32nd bit is the least significant bit (LSB) and it lies in the rightmost position.

The IP address consists of two parts. The first part contains the IP addresses for networks and the second part contains the IP addresses for hosts. The network part corresponds to the first bits of the IP address, called the *address prefix*. We will write prefixes as bit strings of up to 32 bits in IPv4 followed by an asterisk(*). For example, the prefix 10000010 01010110* represents all the 2^{16} addresses that begin with the bit pattern 10000010 01010110. Alternatively, prefixes can be indicated using the dotted-decimal notation, so the same prefix can be written as 130.86/16, where the number after the slash indicates the length of the prefix.

High Performance Switches and Routers, by H. Jonathan Chao and Bin Liu
Copyright © 2007 John Wiley & Sons, Inc.

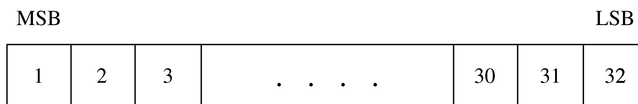


Figure 2.1 IP address bit positions.

Since routing occurs at the network level to locate the destination network, routers only forward packets based on network level IP addresses. Thus, all hosts attached to the network can be stored in the router's forwarding table by a single network IP address, known as *address aggregation*. A group of addresses are represented by prefixes. An example of a router's forwarding table is shown in Table 2.1. Each entry in the forwarding table contains a prefix, next-hop IP address, and output interface number. The forwarding information is located by searching for the prefix of the destination address.

The Internet addressing architecture was first designed using an allocation scheme known as *classful addressing*. Classful addressing defines three different sized networks of classes: A, B, or C (Fig. 2.2). The classes are based on the amount of IP addresses contained in the network partition. With the IPv4 address space of 32 bits, Class A has a network size of 8 bits and a host size of 24 bits. Class B has a network size of 16 bits and a host size of 16 bits. Class C has a network size of 24 bits and a host size of 8 bits. Class D is for multicasting applications.

The classful addressing scheme created very few class A networks. Their address space contains 50 percent of the total IPv4 address space (2^{31} addresses out of a total of 2^{32}). Class B address space contains 16,384 (2^{14}) networks with up to 65,534 hosts per network. Class C address space contains 2,097,152 (2^{21}) networks with up to 256 hosts per network.

Classless Inter-Domain Routing (CIDR) Addressing Scheme. The evolution and growth of the Internet in recent years has proven that the classful address scheme is inflexible and wasteful. For most organizations, class C is too small while class B is too large. The three choices resulted in address space being exhausted very rapidly, even though only a small fraction of the addresses allocated were actually in use. The lack of a network class of a size that is appropriate for mid-sized organizations results in the exhaustion of the class B network address space. In order to use the address space efficiently, bundles of class C addresses were given out instead of class B addresses. This causes a massive growth of forwarding table entries.

CIDR [2] was introduced to remedy the inefficiencies of classful addressing. The Internet Engineering Task Force (IETF) began implementing CIDR in the early 1990s [2, 3]. With CIDR, IP address space is better conserved through arbitrary aggregation of network

TABLE 2.1 Router's Forwarding Table Structure [1]

Destination Address Prefix	Next Hop IP Address	Output Interface
24.40.32/20	192.41.177.148	2
130.86/16	192.41.177.181	6
208.12.16/20	192.41.177.241	4
208.12.21/24	192.41.177.196	1
167.24.103/24	192.41.177.3	4

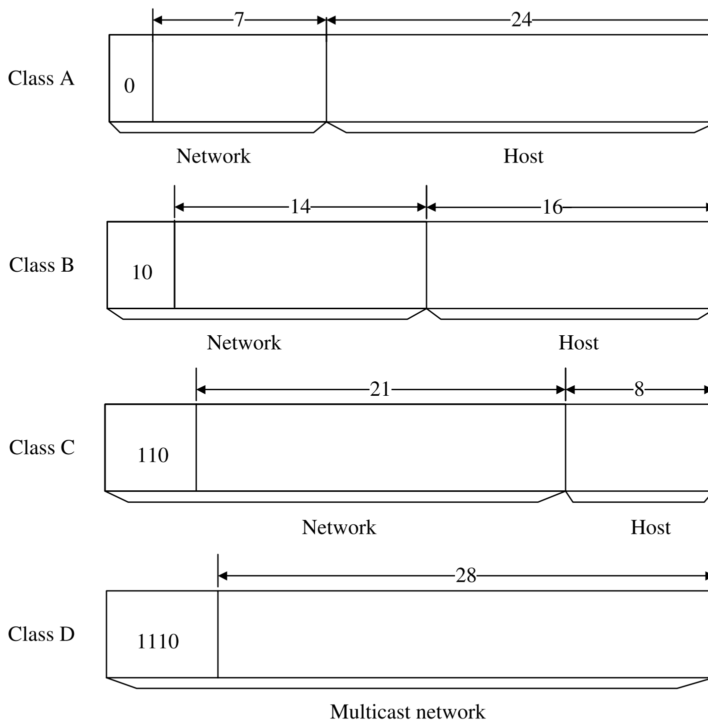
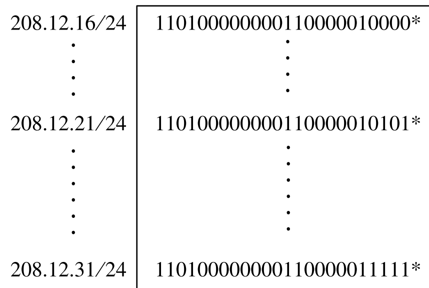


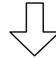
Figure 2.2 Classful addresses [1].

addresses rather than being limited to 8, 16, or 24 bits in length for the network part. This type of granularity provides an organization with more precise matches for IP address space requirements. The growth of forwarding table entries is also slowed by allowing address aggregation to occur at several levels within the heirarchy of the Internet's topology. Backbone routers can now maintain the forwarding information at the level of the arbitrary aggregates of networks, instead of at the network level only.

For example, consider the networks represented by the network numbers from 208.12.16/24 through 208.12.31/24 (see Fig. 2.3) and in a router all these network addresses are reachable through the same service provider. The leftmost 20 bits of all the addresses in this range are the same (11010000 00001100 0001). Thus, these 16 networks can be aggregated into one 'supernetwork' represented by the 20-bit prefix, which in decimal notation gives 208.12.16/20. Indicating the prefix length is necessary in decimal notation, because the same value may be associated with prefixes of different lengths; for instance, 208.12.16/20 (11010000 00001100 0001*) is different from 208.12.16/22 (11010000 00001100 000100*).

Address aggregation does not reduce entries in the router's forwarding table for all cases. Consider the scenario where a customer owns the network 208.12.21/24 and changes its service provider, but does not want to renumber its network. Now, all the networks from 208.12.16/24 through 208.12.31/24 can be reached through the same service provider, except for the network 208.12.21/24 (see Fig. 2.4). We cannot perform aggregation as before, and instead of only one entry, 16 entries need to be stored in the forwarding table. One solution is aggregating in spite of the exception networks and additionally storing





208.12.16/20 11010000000011000001*

Figure 2.3 Prefix aggregation [1].

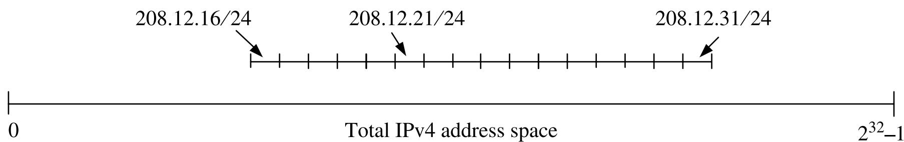


Figure 2.4 Prefix ranges [1].

entries for the exception networks. In this example, this will result in only two entries in the forwarding table: 208.12.16/20 and 208.12.21/24 (see Fig. 2.5 and Table 2.1). However, now some addresses will match both entries because of the prefixes overlap. In order to always make the correct forwarding decision, routers need to do more than search for a prefix that matches. Since exceptions in the aggregations may exist, a router must find the most specific match, which is the longest matching prefix. In summary, the address lookup problem in routers requires searching the forwarding table for the longest prefix that matches the destination address of a packet.

Obviously, the longest prefix match is harder than the exact match used for class-based addressing because the destination address of an arriving packet does not carry with it the information to determine the length of the longest matching prefix. Hence, we need to search among the space of all prefix lengths, as well as the space of all prefixes of a given length. Many algorithms have been proposed in recent years regarding the longest prefix match. This chapter provides a survey of these techniques. But before that, we introduce some performance metrics [4] for the comparison of these lookup algorithms.

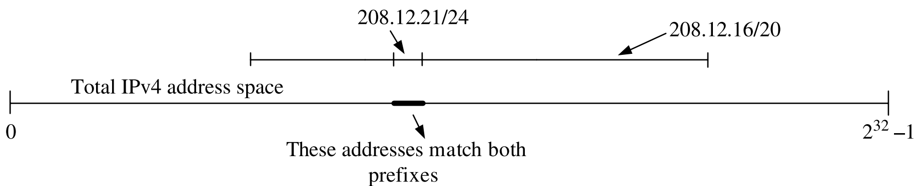


Figure 2.5 Exception prefix [1].

Lookup Speed. The explosive growth of link bandwidth requires faster IP lookups. For example, links running at 10 Gbps can carry 31.25 million packets per second (mpps) (assuming minimum sized 40-byte IP packets).

Storage Requirement. Small storage means fast memory access speed and low power consumption, which are important for cache-based software algorithms and SRAM (static RAM)-based hardware algorithms.

Update Time. Currently, the Internet has a peak of a few hundred BGP (Border Gateway Protocol) updates per second. Thus, a certain algorithm should be able to perform 1k updates per second to avoid routing instabilities. These updates should interfere little with normal lookup operations.

Scalability. It is expected that the size of forwarding tables will increase at a speed of 25k entries per year. Hence, there will be around 250 k entries for the next five years. The ability of an algorithm to handle large forwarding tables is required.

Flexibility in Implementation. Most current lookup algorithms can be implemented in either software or hardware. Some of them have the flexibility of being implemented in different ways, such as ASIC, a network processor, or a generic processor.

2.2 TRIE-BASED ALGORITHMS

2.2.1 Binary Trie

A trie structure is a multi-way tree where each node contains zero or more pointers to point its child nodes, allowing the organization of prefixes on a digital basis by using the bits of prefixes to direct the branching. In the binary trie (or 1-bit trie) structure [5], each node contains two pointers, the 0-pointer and the 1-pointer.

Data Structure. A node X at the level h of the trie represents the set of all addresses that begin with the sequence of h bits consisting of the string of bits labeling the path from the root to that node. Depending on the value of the $(h + 1)$th bit, 0 or 1, each pointer of the node X points to the corresponding subtree (if it exists), which represents the set of all route prefixes that have the same $(h + 1)$ bits as their first bits. An example data structure of each node (i.e., the entry in a memory) is shown in Figure 2.6, including the next hop address (if it is a prefix node), a left pointer pointing to the left node location (with an address bit 0) and a right pointer pointing to the right node location (with an address bit 1).

A prefix database is defined as a collection of all prefixes in a forwarding table. A prefix database example is shown in Figure 2.6 [6], where the prefixes are arranged in an ascending order of their lengths for easy illustration.

To add a route prefix, say 10111*, simply follow the pointer to where 10111 would be in the trie. If no pointer exists for that prefix, it should be added. If the node for the prefix already exists, it needs to be marked with a label as being in the forwarding table (for example, P_i). The nodes in gray are prefix nodes. When deleting a route prefix that has no children, the node and the pointer pointing to it are deleted and the parent node is examined. If the parent node has another child or it is a gray node, it is left alone. Otherwise, that node is also deleted and its parent node is examined. The deletion process is repeated up to the trie until a node that has another child or a gray node is found.

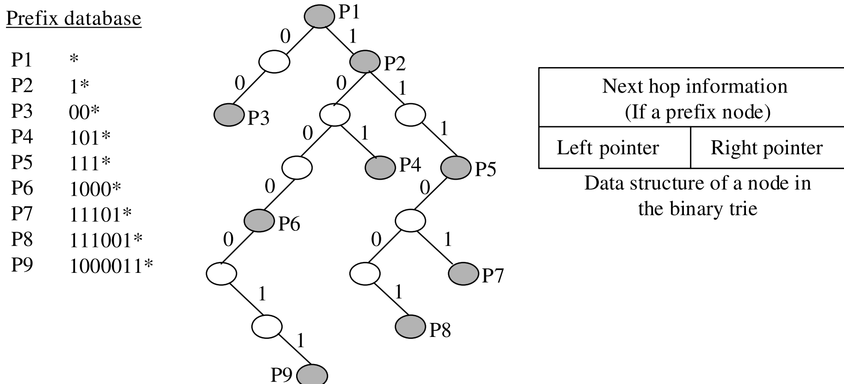


Figure 2.6 Data structure of a 1-bit binary trie.

Route Lookup. Each IP lookup starts at the root node of the trie. Based on the value of each bit of the destination address of the packet, the lookup algorithm determines whether the left or the right node is to be visited. The next hop of the longer matching prefix found along the path is maintained while the trie is traversed. An example is shown in Figure 2.6. Suppose that a destination address 11101000 is given. The IP lookup starts at the root, traverses the path indicated by the destination address, and remembers the last time a gray node was visited. The first bit of 11101000 is 1, so we go to the right and get to the node 1*, which is a gray node, the longest prefix match so far. The 2nd–5th bits of the key are 1, 1, 0, and 1, respectively. So, we turn right, right, left, and right in sequence, and come to a leaf node P7. It is a prefix node and its associated next hop information is returned.

Performance. The drawback of using the binary trie structure for IP route lookup is that the number of memory accesses in the worst case is 32 for IPv4. To add a prefix to the trie, in the worst case it needs to add 32 nodes. In this case, the storing complexity is $32N \cdot S$, where N denotes the number of prefixes in the forwarding table and S denotes the memory space required for each node. In summary, the lookup complexity is $O(W)$, so is the update complexity, where W is the maximum length of the prefix. The storage complexity is $O(NW)$.

Variants of Binary Tries. The 1-bit binary trie in Figure 2.6 can be expanded to a complete trie, where every bottom leaf node is a prefix. There will be 128 leaf nodes. The data structure will be a memory with 128 entries. Each stores a prefix and can be directly accessed by a memory lookup using the seven bits of the destination address. One drawback is that the memory size becomes too big to be practical when the address has 32 bits, requiring a memory with 2^{32} entries.

One way to avoid the use of the longest prefix match rule and still find the most specific forwarding information is to transform a given set of prefixes into a set of disjoint prefixes. Disjoint prefixes do not overlap, and thus no address prefix is itself a prefix of another. A trie representing a set of disjoint prefixes will have prefixes at the leaves but not at internal nodes. To obtain a disjoint-prefix binary trie, we simply add leaves to nodes that have only one child. These new leaves are new prefixes that inherit the forwarding information of the closest ancestor marked as a prefix. Finally, internal nodes marked as prefixes are unmarked.

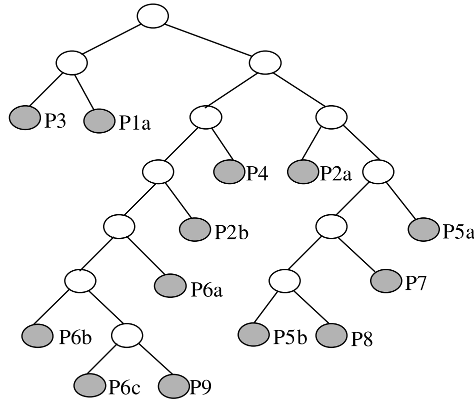


Figure 2.7 Disjoint-prefix binary trie.

For example, Figure 2.7 shows the disjoint-prefix binary trie that corresponds to the trie in Figure 2.6. Prefixes P2a and P2b have inherited the forwarding information of the original prefix P2, similar to other nodes such as P1a, P5b, P6a, P6b, and P6c. Since prefixes at internal nodes are expanded or pushed down to the leaves of the trie, this technique is called 'leaf pushing' by Srinivasan and Varghese [7].

2.2.2 Path-Compressed Trie

Path compression technique was first adopted in the Patricia trees [8]. A path-compressed trie is based on the observation that each internal node of the trie that does not contain a route prefix and has only one child node can be removed in order to shorten the path from the root node.

Data Structure. By removing some internal nodes, the technique requires a mechanism to record which nodes are missing. A simple mechanism is to store in each node:

- A number, the skip value, that indicates how many bits have been skipped on the path.
- A variable-length bit-string, segment, that indicates the missing bit-string from the last skip operation.

The path-compressed version of the binary trie in Figure 2.6 is shown in Figure 2.8. The node structure has two more fields – skip value and segment. Note that some gray nodes have a skip value = 1 or >1. For instance, for node P9, its skip value = 2 and the segment is '11'. As compared to P9 in Figure 2.6, the P9 node in Figure 2.8 moved up the level by 2 and missed the examination of two bits '11'. Therefore, when we traverse the trie in Figure 2.8 and reach P9, the immediate two bits of the key need to be checked with the 2-bit segment.

Route Lookup. Suppose that a destination address 11101000 (i.e., the key) is given. The route lookup starts at the root and traverses the path based on the destination address bits. It also records the last gray node that was visited. The first bit of 11101000 is 1, so we go to the right and get to the prefix node P2. As the second bit of the key is 1, we go right again,

Prefix database

P1	*
P2	1*
P3	00*
P4	101*
P5	111*
P6	1000*
P7	11101*
P8	111001*
P9	1000011*

Next hop information (If a prefix node)	
Skip value	Segment
Left pointer	Right pointer

Data structure of a node in the path-compressed trie

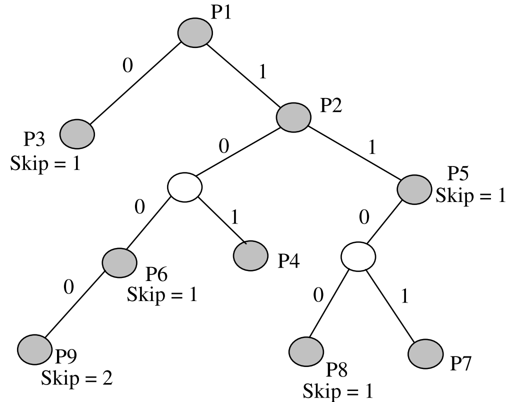


Figure 2.8 Path-compressed trie example.

and reach node P5. This node has a skip value of 1, meaning that a node is skipped on the path. We then use the 3rd bits of the key to compare with the segment field '1' (to verify we have arrived at the correct node in the path). If a match is found, it indicates that we have arrived at P5 correctly. As the 4th bit of the key is 0, we turn left; no skip value is found so we move on. With the 5th bit a 1, we again turn right and get to node P7. Here, we reach a leaf and no skip value is found. So, the lookup stops here, and the next hop information corresponding to P7 is returned.

Performance. Path compression reduces the height of a sparse binary trie. When the tree is full and there is no compression possible, a path-compressed trie looks the same as a regular binary trie. Thus, its lookup and update complexity (the worst case) is the same as a binary trie, $O(W)$. Considering a path-compressed trie as a full binary trie with N leaves, there can be $N - 1$ internal nodes between the root and each leaf node (including the root node), as shown in Figure 2.9. Since the path can be significantly compressed to reduce the internal nodes, the space complexity becomes $O(N)$, independent of W.

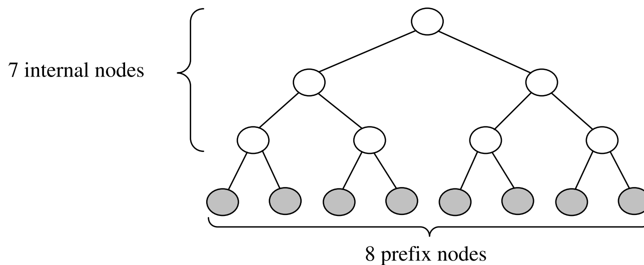


Figure 2.9 Example of path-compressed trie with N leaves.

2.2.3 Multi-Bit Trie

The lookup performance can be improved by using a multi-bit trie structure [7]. The multi-bit trie structure examines several bits at a time, called the lookup stride, while the standard binary trie only examines one bit at a time.

Data Structure. A multi-bit trie example is shown in Figure 2.10. Its prefix database is the same as the one in Figure 2.6. Suppose we examine the destination address three bits at a time, that is, the lookup stride is 3. Then a problem arises for the prefixes like P2 = 1* that do not fit evenly in multiples of three bits. One solution is to expand a prefix like 1* into all possible 3 bit extensions (100, 101, 110, and 111). However, prefixes like P4 = 101 and P5 = 111 are selected because they have longer length matches than those of expanded prefixes of P2. In other words, prefixes whose lengths do not fit into the stride length are expanded into a number of prefixes that fit into the stride length. However, expanded prefixes that collide with an existing longer length prefix are discarded.

Figure 2.10 shows an expanded trie with a fixed stride length of 3 (i.e., each trie node examines three bits). Notice that each trie node has a record of eight entries and each has two fields: one for the next hop information of a prefix and one for a pointer that points to the next child node. Consider, for example, entry 100 in the root node. It contains a prefix (P2 = 100) and a pointer to the trie node containing P6. The P6 pointer also points to another trie containing P9.

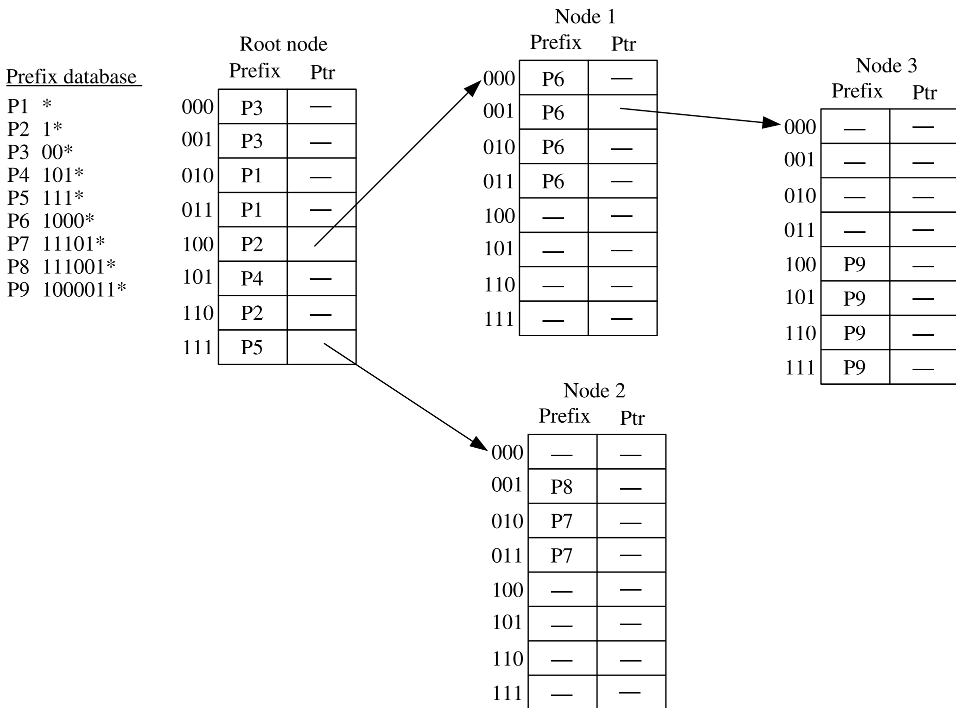


Figure 2.10 Multi-bit trie structure example.

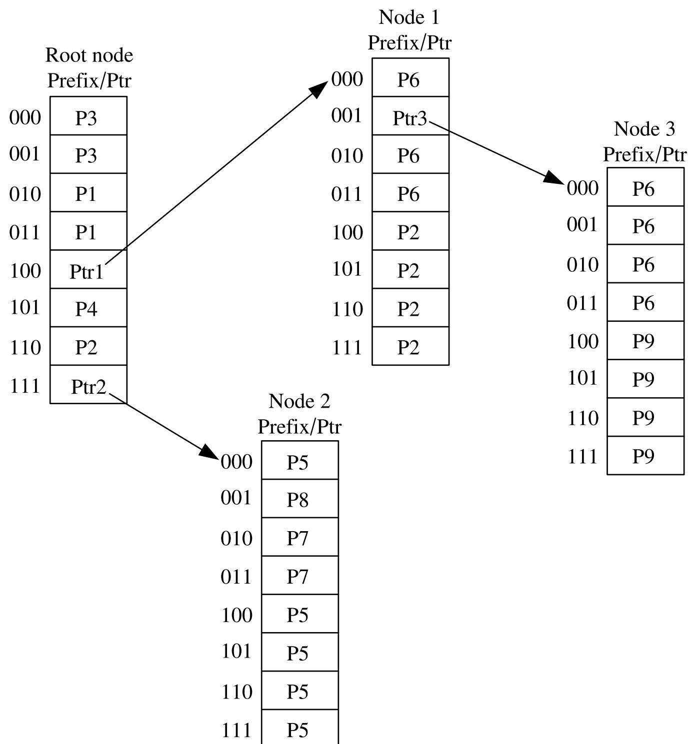


Figure 2.11 Multi-bit trie structure example with each entry a prefix or a pointer to save memory space.

Route Lookup. Let us use the example in Figure 2.10 and again suppose that the destination address is 11101000. The IP lookup starts at the root and traverses the path using three address bits at a time while remembering the last prefix that was visited. The first three bits of the key 111 are used as the offset address in the root node to find if the corresponding entry contains a prefix (in this case, it contains P5). We then follow the pointer and move to the next node. Then the 4–6th bits of the key 010 are used as an offset in the second node. The corresponding entry contains P7's next hop information and a pointer pointing to a NULL address, indicating the end of the lookup.

Performance. The advantage of the k-bit trie structure is that it improves the lookup by k times. The disadvantage is that a large memory space is required. One way to reduce the memory space is to use a scheme called 'leaf pushing' described in Section 2.2.1. Leaf pushing can cut the memory requirements of the expanded trie in half by making each node's entry contain either a pointer or next hop information but not both (as shown in Figure 2.11 versus the one shown in Figure 2.10). The trick is to push down the next hop information to the leaf nodes of the trie. The leaf nodes only have the next hop information, since they do not have a pointer. For instance, P6's next hop information at the entry 001 of the top middle node is pushed down to the vacant locations of its child node (i.e., the right most node).

The lookup is performed in strides of k bits. The lookup complexity is the number of bits in the prefix divided by k bits, $O(W/k)$. For example, if W is 32 and k is 4, then 8 lookups in the worst case are required to access that node. An update requires a search through W/k lookup iterations plus access to each child node (2^k). The update complexity is $O(W/k + 2^k)$. In the worst case, each prefix would need an entire path of length (W/k) and each node would have 2^k entries. The space complexity would then be $O((2^k * N * W)/k)$.

2.2.4 Level Compression Trie

The path-compressed trie in Section 2.2.2 is an effective way to compress a trie when nodes are sparsely populated. Nilsson and Karlsson [9] have proposed a technique called level compression trie (LC-trie), to compress the trie where nodes are densely populated. The LC-trie actually combines the path-compression and multi-bit trie concepts to optimize a binary trie structure.

Data Structure. The fixed-stride multi-bit trie (in Section 2.2.3) can improve lookup performance, but it may incur redundant storage. However, from an angle of local scopes, if we only perform multi-bit lookup wherever the sub-trie structures are full sub-trees, then no redundant storage is needed in those nodes. The construction of the LC-trie is, in fact, a process of transforming a path-compressed trie to a multi-bit path-compressed trie. The process is to find the full sub-trees of a path-compressed trie, and transform them into multi-bit lookup nodes. Therefore, information stored in each node of the LC-trie includes those that are needed for both the path-compressed trie and the multi-bit trie lookup.

The LC-trie algorithm starts with a disjoint-prefix binary trie as described in Section 2.2.1, where only leaf nodes contain prefixes. Figure 2.12 shows three different tries: (*a*) 1-bit binary trie with prefixes at leaf nodes; (*b*) path-compressed trie; and (*c*) LC-trie. The LC-trie needs only three levels instead of the six required in the path-compressed trie. Furthermore, Figure 2.12 shows a straight-forward transformation from path-compressed trie to LC-trie, as shown in the dashed boxes in Figures 2.12*b* and *c*, where the first three levels of the path-compressed trie that form a full sub-trie are converted to a single-level 3-bit sub-trie in the LC-trie.

Route Lookup. Let us again use the 8-bit destination address 11100000 as an example to explain the lookup process. The lookup starts at the root node of the LC-trie in Figure 2.12*c*. In this node, the multi-bit and path-compression trie lookup information shows 'stride = 3' and 'skip = 0'. 'Stride = 3' indicates that there would be $2^3 = 8$ branches in this node and that we should use (the next) 3 bits of the address to find the corresponding branch. 'Skip = 0' indicates that no path compression is involved here. The first three bits of the address 111 are inspected, and we should take the $(7 + 1) = 8$th branch of this node. Then, at the branched node, we have 'stride = 1' and 'skip = 3'. 'Stride = 1' indicates that there are only $2^1 = 2$ branches in this node. 'Skip = 3' indicates that the following path of the trie is path-compressed, and we should compare with the stored segment, and if it matches then skip the next three bits of the IP address. We skip the next three bits 000 and examine the fourth bit, 0. The bit 0 indicates that we should take the left branch of this node, and then we come to node 13. On finding that node 13 is a leaf node, the lookup stops here, and the next hop information corresponding to node 13 is returned.

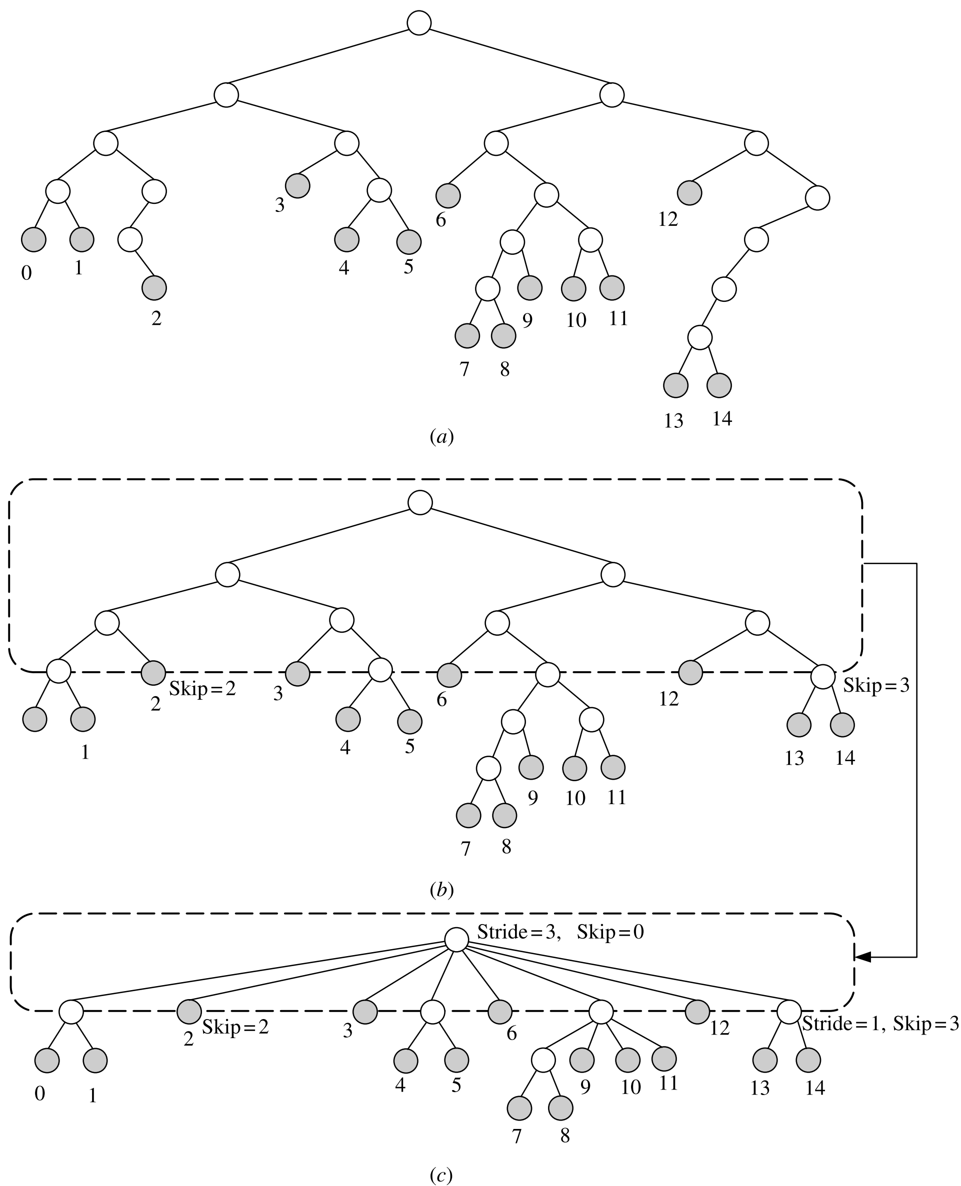


Figure 2.12 (*a*) One-bit trie; (*b*) Path-compressed trie; (*c*) LC-trie.

Performance. Instead of only transforming strictly full sub-tries in the path-compressed trie, we could transform nearly full sub-tries as well. An optimization proposed by Nilsson and Karlsson [9] suggests this improvement under the control of a fill factor x, where $0 < x < 1$. The transformation could be performed on this sub-trie when a k-level sub-trie has more than $2^k \cdot x$ leaf nodes available.

Less than 1-Mbyte memory is required and 2 Mpackets/s (assuming average packet size of 250 bytes) search speed is achieved when running on a SUN Sparc Ultra II

workstation [9]. This is for an LC-trie built with a fill factor of 0.5 and using 16 bits for branching at the root and a forwarding table with around 38,000 entries. However, using a node array to store the LC-trie makes incremental updates very difficult.

An LC-trie searches in strides of k bits, and thus the lookup complexity of a k-stride LC-trie is $O(W/k)$. To update a particular node, we would have to go through W/k lookups and then access each child of the node (2^k). Thus, the update complexity is $O(W/k + 2^k)$. The memory consumption increases exponentially as the stride size (k) increases. In the worst case, each prefix would need an entire path of length (W/k) and each node has 2^k entries. The space complexity would then be $O((2^k * N * W)/k)$.

2.2.5 Lulea Algorithm

Degermark et al. [10] have proposed a data structure that can represent large forwarding tables in a very compact form. This solution is small enough to fit entirely in the Level 1/Level 2 cache. This provides an advantage in that this fast IP route-lookup algorithm can be implemented in software running on general-purpose microprocessors at high speeds.

The key idea of the Lulea scheme is to replace all consecutive elements in a trie node that have the same value with a single value, and use a bitmap to indicate which elements have been omitted. This can significantly reduce the number of elements in a trie node and save memory.

Basically, the Lulea trie is a multi-bit trie that uses bitmap compression. We begin with the corresponding multi-bit 'leaf-pushing' trie structure given in Section 2.2.3. For instance, consider the root node in Figure 2.11. Before bitmap compression, the root node has the sequence of values (P3, P3, P1, P1, ptr1, P4, P2, ptr2), where ptr1 is a pointer to the trie node containing P6 and ptr2 is a pointer to the trie node containing P7. If we replace each string of consecutive values by the first value in the sequence, we get P3, −, P1, −, ptr1, P4, P2, ptr2. Notice the two redundant values have been removed. We can now get rid of the original trie node and replace it with a bitmap 10101111 where the '0's indicate the removed position and a compressed list (P3, P1, ptr1, P4, P2, ptr2). The result of doing bitmap compression for all four trie nodes is shown in Figure 2.13.

Data Structure. The multi-bit trie in the Lulea algorithm is a disjoint-prefix trie as described in Section 2.2.1. As shown in Figure 2.14, level one of the data structure covers the trie down to depth 16, level two covers depths 17 to 24, and level three depths 25 to 32. A level-two chunk describes parts of the trie that are deeper than 16. Similarly, chunks at level three describe parts of the trie that are deeper than 24. The result of searching a level of the data structure is either an index into the next-hop information or an index into an array of chunks for the next level.

The first level is essentially a complete trie with 1–64k children nodes, which covers the trie down to depth 16. Imagine a cut through the trie at depth 16. Figure 2.15 shows an example of partial cut. The cut is stored in a bit-vector, with one bit per possible node at depth 16. Thus, 2^{16} bits = 64 kbits = 8 kbytes are required for this. The upper 16 bits of the address are used as an index into the bit-vector to find the bit corresponding to the initial part of an IP address.

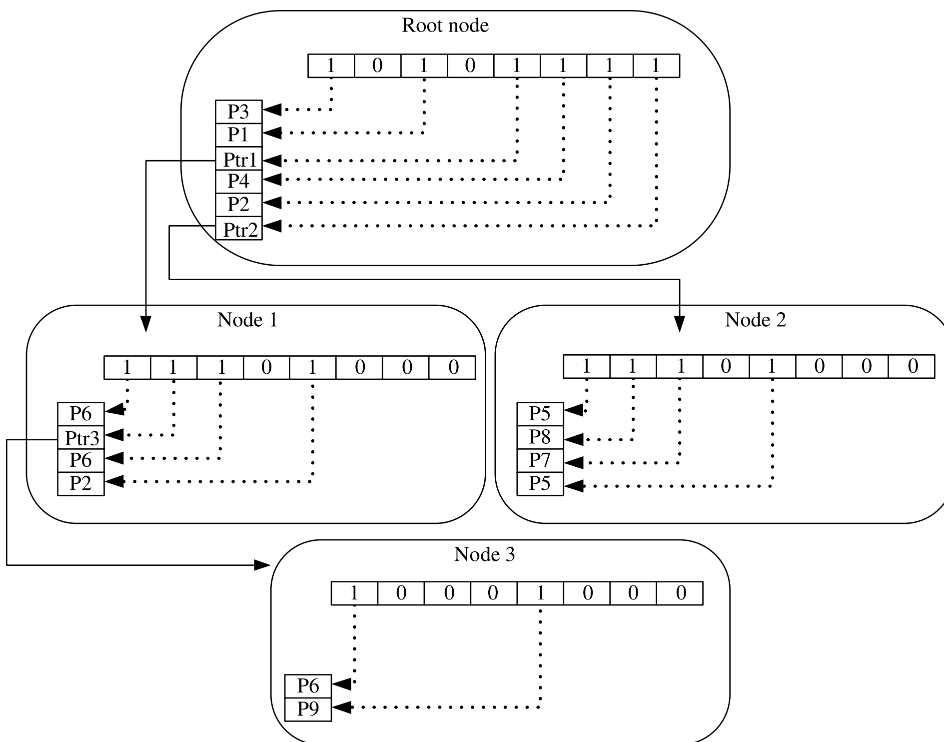


Figure 2.13 Example of a Lulea trie.

As shown in Figure 2.15, a bit in the bit-vector can be set to:

- One representing that a prefix tree continues below the cut. They are called a root head (bits 6, 12, and 13 in Fig. 2.15).
- One representing a prefix at depth 16 or less. For the latter case, only the lowest bit in the interval covered by that prefix is set. They are called a genuine head (bits 0, 4, 7, 8, 14, and 15 in Fig. 2.15).
- Zero, which means that this value is a member of a range covered by a prefix at a depth less than 16 (bits 1, 2, 3, 5, 9, 10, and 11 in Fig. 2.15).

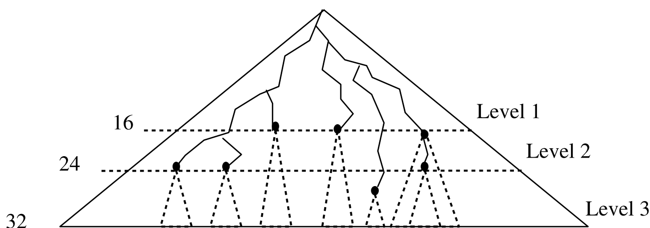


Figure 2.14 Three levels of the data structure [10].

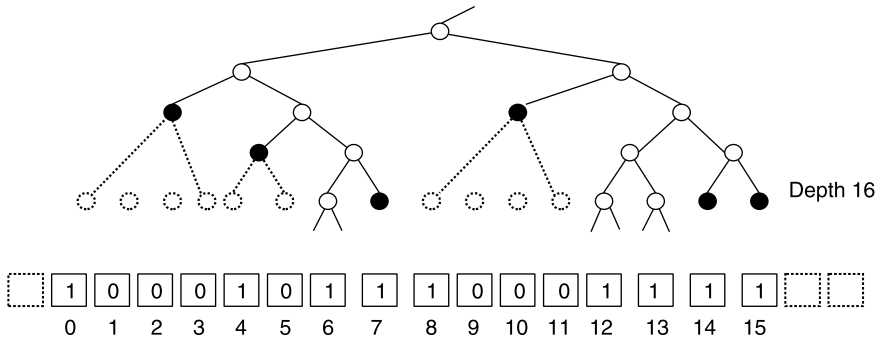


Figure 2.15 Part of cut with corresponding bit-vector [10].

A pointer to the next-hop information is stored for genuine heads. Members behind the genuine head use the same index. A pointer to the level two chunk that represents the corresponding sub-trie is stored in the root heads.

The head information is encoded in 16-bit pointers stored in an array. Two bits of the pointer encode what kind of pointer it is. The 14 remaining bits are either indices into the next-hop information or into an array containing level two chunks. Note that there are as many pointers associated with a bit-mask as its number of set bits.

Finding Pointer Groups. The bit-vector is divided into bit-masks of length 16 and there are $2^{12} = 4096$ of those. Figure 2.16 is an illustration of how the data structure for finding pointers corresponds to the bit-masks. The data structure consists of an array of code words of all bit-masks and an array of base indices of one per four code words. The code words consist of a 6-bit offset $(0, 3, 10, 11, \ldots)$ and a 10-bit value $(r1, r2, \ldots)$.

The first bit-mask in Figure 2.16 has three set bits. The second code word thus has an offset of three because three pointers must be skipped over to find the first pointer associated with that bit-mask. The second bit-mask has seven set bits and consequently the offset in the third code word is $3 + 7 = 10$.

After four code words, the offset value might be too large to represent with six bits. Therefore, a base index is used together with the offset to find a group of pointers. There can be at most 64k pointers in a level of the data structure, so the base indices need to be at most 16 bits ($2^{16} = 64$k). In Figure 2.16, the second base index is 13 because there are 13 set bits in the first four bit-masks. This explains how a group of pointers is located. The first 12 bits of the IP address are an index to the proper code words. The first 10 bits are an index to the array of base indices.

How to find the correct pointer in the group of pointers will now be explained. This is what the 10-bit value is for $(r1, r2, \ldots$ in Fig. 2.16). The value is an index into a table that maps bit-numbers in the IP address to pointer offsets. Since bit-masks are generated from a complete trie, not all combinations of the 16 bits are possible. As shown by Degermark et al. [10], the number of possible bit-masks with length 16 is only 678. They are all stored in a table, called `maptable`, as shown in Figure 2.17. An index into a table with an entry for each bit-mask only needs 10 bits. The content of each entry of the `maptable` is a bit-mask of 4-bit offsets. The offset specifies how many pointers to skip over to find the wanted one, so it is equal to the number of set bits smaller than the bit index. For instance, if the 2nd 4-bit mask in Figure 2.16 is chosen and the low 4 of high 16 bits of IP address is

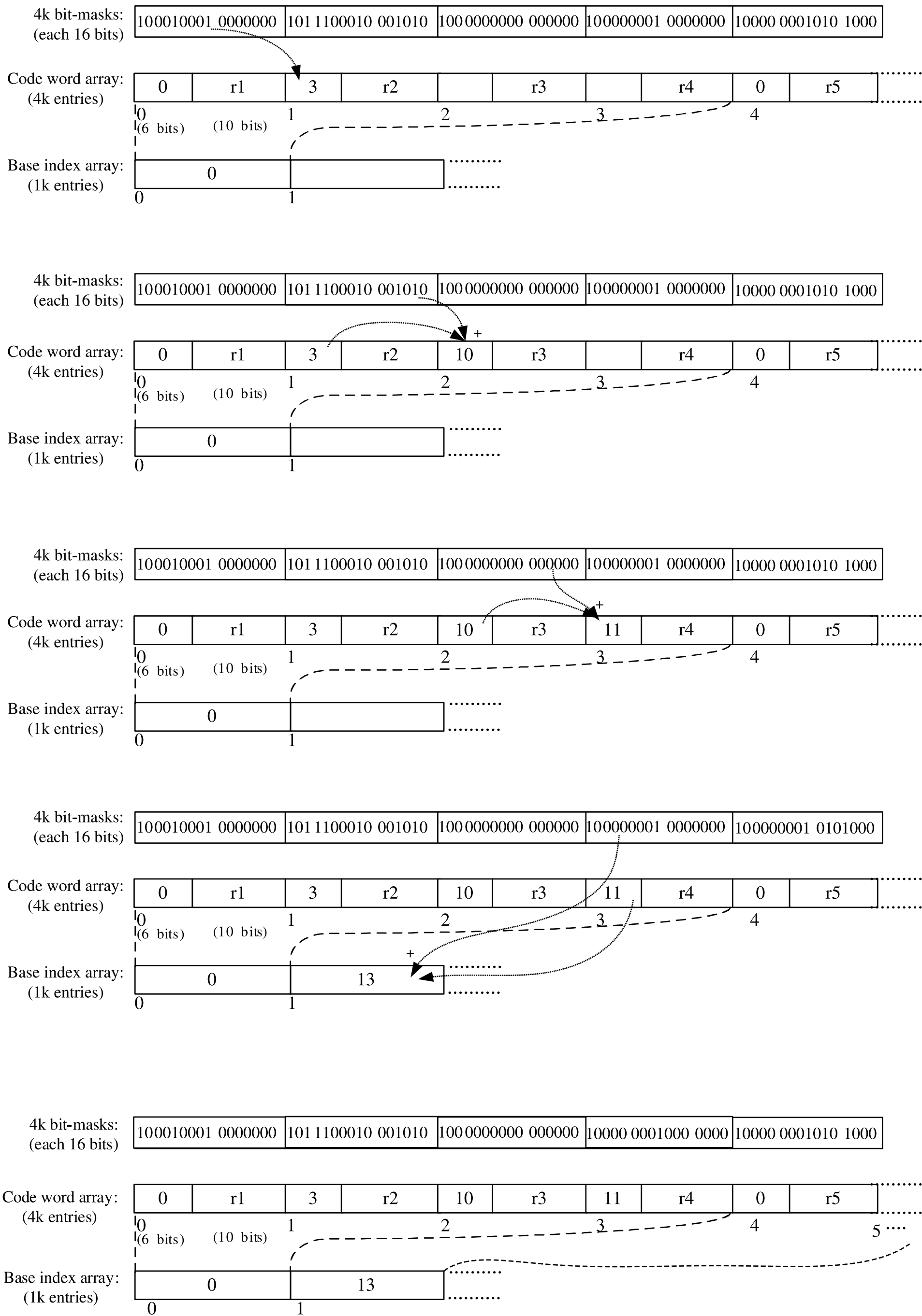


Figure 2.16 Bit-masks versus code words and base indices.

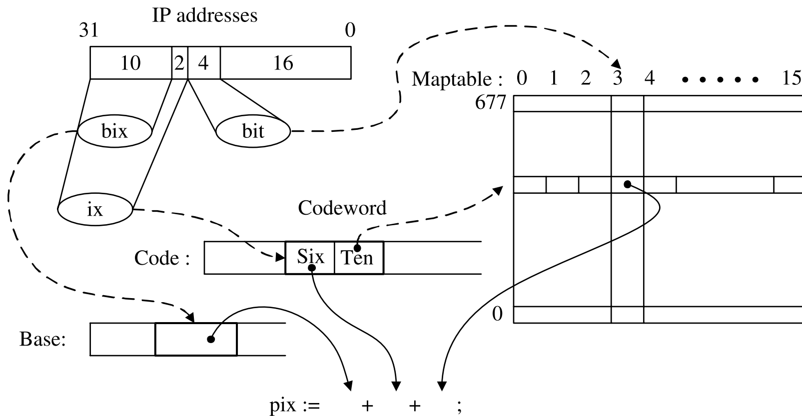


Figure 2.17 Finding the pointer index [10].

'1010', the 4-bit offset in the `Maptable` will be 5, counting five '1' from the left to the bit location of '1010'.

Route Lookup. The following steps are required to search the first level of the data structure. The array of code words is called `code` and the array of base addresses is called `base`. Figure 2.17 illustrates the procedure. The variables are defined as below:

`ix` := high 12 bits of IP address
`bit` := low 4 of high 16 bits of IP address
`code word` := code[ix]
`ten` := ten bits from codeword
`six` := six bits from codeword
`bix` := high 10 bits of IP address
`pix` := base[bix] + six + maptable[ten][bit]
`pointer` := level1_pointers[pix]

The index of the code word (`ix`), the index of the base index (`bix`), and the bit number (`bit`) are first extracted from the IP address. Then the `code word` is retrieved and its two parts are extracted into `ten` and `six`. The pointer index (`pix`) is then obtained by adding the base index, the 6-bit offset `six`, and the pointer offset obtained by retrieving column `bit` from row `ten` of `maptable`. After the `pointer` is retrieved from the pointer array, it will be examined to determine if the next-hop information has been found or if the search should continue on to the next level.

Performance. The Lulea algrorithm provides a very compact data structure and fast lookups. The data structure has 150–160 kbytes for the largest forwarding tables with 40,000 routing entries, which is small enough to fit in the cache of a conventional general-purpose processor. A 200 MHz Pentium Pro or a 333 MHz Alpha 21164 with the table in the cache can perform a few million IP lookups per second without special hardware and no traffic locality is assumed. Lulea does not support incremental updates because of the algorithm's tight coupling property. In many cases, the whole table should be reconstructed. Thus, routing protocols that require frequent updates make this algorithm unsuitable.

The Lulea trie uses the k-bit stride multi-bit method. The lookup complexity is the same as a multi-bit trie, $O(W/k)$. The bitmap compression technique applied to the multi-bit trie makes it almost impossible to perform incremental updates. The data structure may need to be completely rebuilt. The memory consumption is the same as the k-bit stride multi-bit trie. Thus, the space complexity is $O((2^k * N * W)/k)$.

2.2.6 Tree Bitmap Algorithm

Eatherton et al. [11] have proposed a data structure of lookup scheme based on multi-bit expanded tries without any leaf pushing and bitmap compression, called Tree Bitmap. The lookup scheme simultaneously meets three criteria: scalability in terms of lookup speed and table size, the capability of high-speed updates, and feasibility in size to fit in a Level 3 forwarding engine or packet processor with low overhead. Tree Bitmap has flexibility to adapt to the next generations of memory technology. To the best of our knowledge, Cisco CRS-1 uses this lookup scheme.

Data Structure. In the Tree Bitmap algorithm, a trie node, as shown in Figure 2.18, is fixed in size, containing a head pointer to the block of child nodes, an extending paths bitmap, an internal next hop information bitmap, and a pointer to the external result array of next hop information associated with prefixes stored in the node. Since update time is bounded by the size of a trie node, Tree Bitmap uses trie nodes of no more than eight strides. By taking advantage of modern burst-based memory technologies, Tree Bitmap keeps the trie nodes as small as possible to be within the optimal memory burst size. Thus, all information about the node being examined can be fetched into the processor in one memory reference and processed to find the longest prefix match.

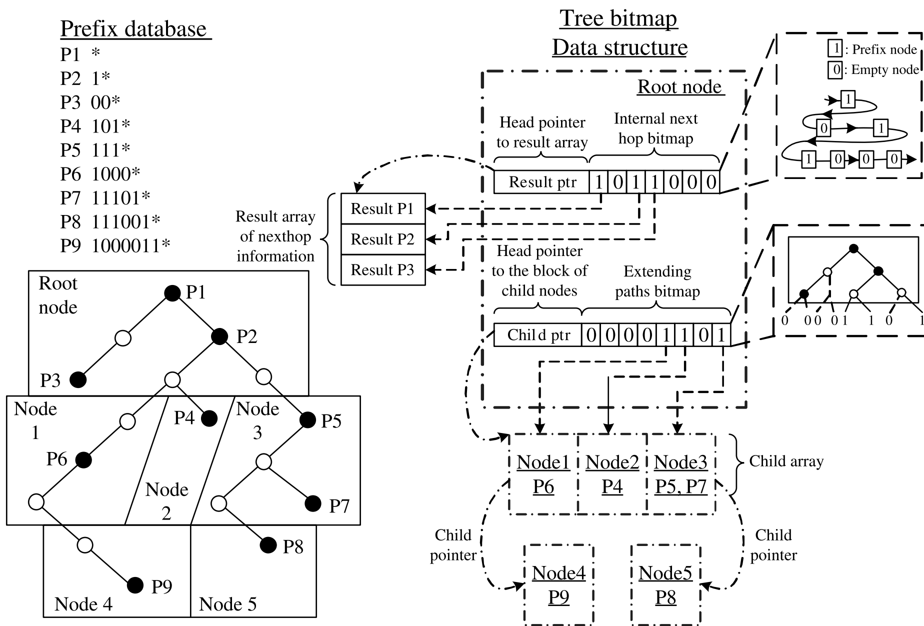


Figure 2.18 Data structure of Tree Bitmap.

A multi-bit node could be seen as a unibit node of multiple levels. It has to have two functions: it has to point to all its child nodes and it has to point to next hop information for prefix nodes within its internal structure. Tree Bitmap uses two bitmaps to implement these two functions individually. Tree Bitmap uses different multi-bit node grouping schemes from the multi-bit trie described in Section 2.2.3. As shown in Figure 2.18, all length 3 prefixes are pushed and stored along with the length 0 prefixes in the next node down. For example, P5 is pushed down to Node 3 in Level 2; P4 is pushed down to Node 2, which is created to store P4. However, in the multi-bit trie of Figure 2.10, P4 and P5 are stored in the root node.

All child nodes of a given trie node are stored contiguously. In Figure 2.18, all child nodes of the root node, Node 1, Node 2, and Node 3, are contiguous in memory. Thus, only one head child pointer is needed in the root node to point to all its children with the help of the extending paths bitmap, which contains a bit for all possible 2^r child nodes (r: stride size) and is used to calculate the offset from the head pointer. In Figure 2.18, of the eight possible leaves, only the fifth, the sixth, and the eighth leaf nodes have pointers to children. Thus, the extending paths bitmap is 00001101 by setting the corresponding bit positions from the left to be '1'.

The next hop information associated with the internal prefixes of each trie node is stored in a separate array associated with the trie node. Only the head pointer to the array is necessarily kept in the trie node, together with an internal next-hop information bitmap, which is used to record every prefix stored within the node and to calculate the offset from the head pointer. In a r-bit trie, the first bit of the bitmap is associated with prefix of length 0. The two following bits are associated with prefixes of length 1, the four following bits are associated with prefixes of length 2, . . . and 2^{r-1} following bits are associated with prefixes of length $r - 1$ for a total of $2^r - 1$ bits. In Figure 2.18, the internal bitmap of the root node is obtained by traversing through the 3-level unibit trie nodes from top to bottom and from left to right, and replacing the prefix nodes with 1s and non-prefix nodes with 0s.

Route Lookup. Figure 2.19 illustrates how to search for the longest prefix match of '11101100' in the Tree Bitmap data structure. According to the stride of the root node, three in the example, the first three bits of '11101100', '111', are used as an index to look up the extending path bitmap, resulting in a '1' (Step 1). This means there is a valid child pointer. The number of 1s is counted to the left in the bitmap to compute the offset, $I = 3$ (Step 2). Since the head pointer to the block of child nodes, H, is known as well as the size of each child node, S, the pointer to the child node can be easily calculated as $H + (I \times S)$, which points to Child Node 3 (Step 3).

Then, the internal next hop information bitmap is checked to see if there is a prefix match (Step 4). It uses a completely different calculation from Lulea. As mentioned earlier, in a r-bit trie, Tree Bitmap pushes length r prefixes down to be stored as length 0 in the next level. Thus, the right most bit of '111' is removed at first, resulting in '11*'. From the previous example about how to construct an internal bitmap, it is easy to conceive that '11*' corresponds to the seventh bit of the internal bitmap, where '0' is found. No prefix is found (Step 5). One more right most bit of '11*' is removed, resulting in '1*'. The corresponding third bit of the internal bitmap is checked. '1' is found, meaning there is a prefix found (Step 6). The number of '1's before the matched prefix at Step 6 is 1. It indicates the offset of the prefix match in the result array. The pointer to the matched prefix is calculated by using the head pointer to the result array and the offset, resulting in P2, which is stored as

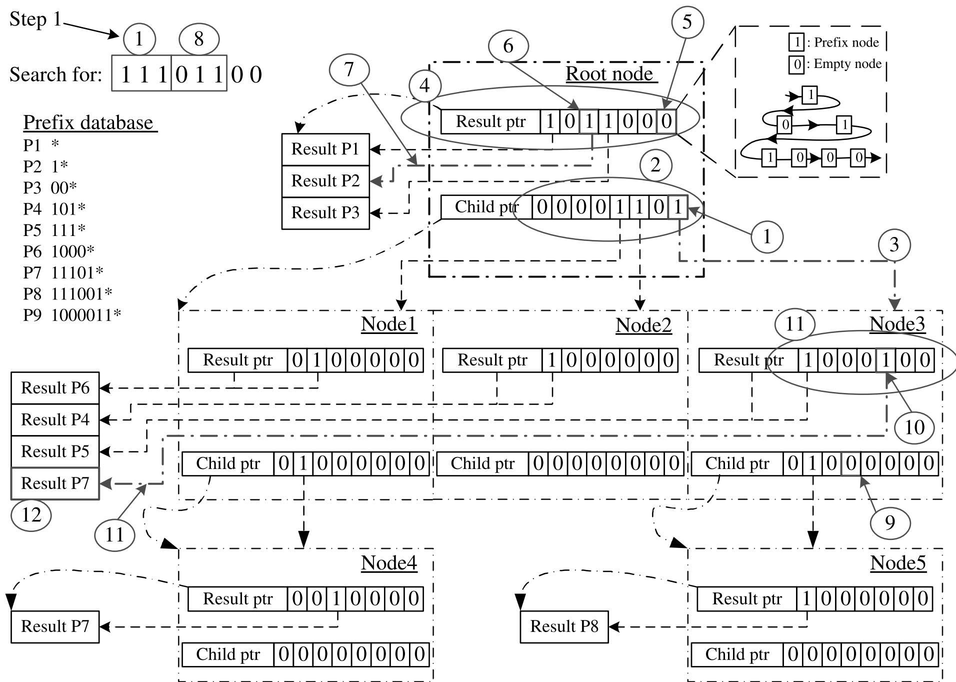


Figure 2.19 Example of route lookup.

prefix match (Step 7). The information of Child Node 3 is loaded from memory and another iteration begins.

The next 3-bit chunk from ‘11101100’, ‘011’, is extracted (Step 8). No extending path with index ’011’ is found in Node 3 (Step 9). The right most bit of ‘011’ is removed, resulting in ‘01*’. The fifth bit of the internal bitmap is found to be ‘1’, indicating a prefix found (Step 10). The number of 1s to the left of the matched prefix is 1. The prefix match is P7 (Step 11). The prefix match found in the lowest level is the longest one. When there are no more child nodes, the search stops. The next hop information of P7 is retrieved from the result array (Step 12).

Performance. Similar to the Lulea algorithm, described in Section 2.2.5, Tree Bitmap uses multi-bit tries and bitmaps to compress wasted storage in trie nodes and achieve fast lookup speeds. However, Tree Bitmap uses a completely different encoding scheme that relies on two bitmaps per node to avoid leaf pushing, which makes update inherently slow as in the Lulea algorithm. Tree Bitmap allows not only fast searches comparable to the Lulea algorithm, but also fast update.

Tree Bitmap takes advantage of memory technology and processing power. It performs complex processing in one memory access per trie node, as opposed to three memory references required by the Lulea algorithm. It trades off algorithm complexity for less memory access. Tree Bitmap is tunable over a wide range of architectures.

2.2.7 Tree-Based Pipelined Search

Most tree-based solutions for network searches can be regarded as some form of tree traversal, where the search starts at the root node, traverses various levels of the tree, and typically ends at a leaf node. The tree-based solution has optimal memory utilization but needs multiple times memory access per packets according to the depth of searching tree. Tree-based pipelined searching methods are then introduced to avoid the bottleneck in memory access. It allows different levels of the tree to be partitioned onto private memories associated with the processing elements. Tree-based searches are pipelined across a number of stages to achieve high throughput, but this results in unevenly distributed memory. To address this imbalance, conventional approaches use either complex dynamic memory allocation schemes (dramatically increasing the hardware complexity) or over-provision each of the pipeline stages (resulting in memory waste). The use of large, poorly utilized memory modules results in high system cost and high memory latencies, which can have a dramatic effect on the speed of each stage of the pipelined computation, and thus on the throughput of the entire architecture.

Balance Memory Distribution in a Pipelined Search Architecture. A novel architecture for a network search processor, which provides both high execution throughput and balanced memory distribution by dividing the searching tree into subtrees and allocating each subtree separately has recently been described [12]. This method allows searches to begin at any pipeline stage to balance the separated memory rather than the prior pipelined network search algorithms, which require all searches to start from the first pipeline stage (the root node of searching tree), going next to the second, and so on.

Figure 2.20 shows a tree-based search structure. To keep the explanation simple, let us assume that the tree has four subtrees, called $S_1, \ldots, S_4$. The separation of the subtree can be determined by using a hash function based on information in the packet header. For IP lookups the hash function is made up of a set of variable length IP prefixes. For packet classification, the hash function may use some of the most significant bits in two or three different fields of the packet header. Furthermore, the search structure is implemented on a four-stage pipeline, called $P_1, \ldots, P_4$, corresponding to the depth of each subtree with four levels. Each level in the subtree can handle multiple bits lookup, for example, by 'Tree Bitmap' scheme [11] in Section 2.2.6. The first level of the subtree S_1, called S_1^1, is stored

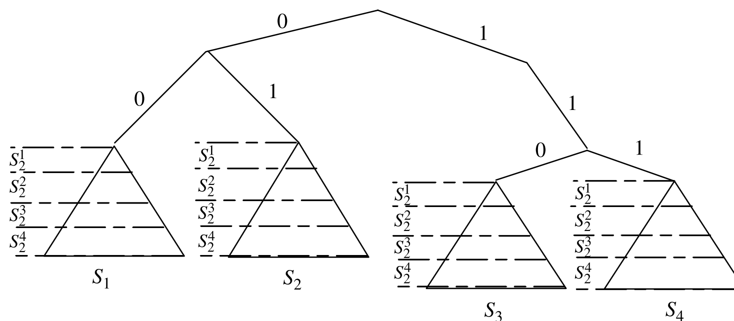


Figure 2.20 Example of a basic tree-based search structure. The tree is split into four subtrees $S_1, \ldots, S_4$. Each subtree has up to four levels. We call S_i^j the level j into the subtree S_i [12].

and processed by the pipeline stage P_1. The second level S_1^2 is stored and processed by the pipeline stage P_2, and so on. The second subtree is processed starting with pipeline stage P_2, S_2^1 on P_2, S_2^2 on P_3, S_2^3 on P_4 and S_2^4 on P_1, respectively. Similarly, the third subtree S_3 starts on stage P_3, while the fourth subtree S_4 starts on pipeline stage P_4. This allocation scheme tries to balance the load on each of the pipeline stages. By doing so, the pipeline allocates nearly equal amounts of memory to each stage.

In practice, the number of subtrees should be more than or equal to the number of pipeline stages (processing elements), thus implying multiple subtrees may have the same start node. The number of the maximum depth of each subtree should be less than or equal to the number of pipeline stages. However, this tree-based pipelined search architecture with balanced memory distribution by allowing search tasks to start execution from any pipeline stage impacts the throughput of the system. This is because of potential conflicts in memory access between the new tasks and the ones that are in execution.

Avoid Memory Access Conflicts in the Pipelined Search Architecture. A random ring pipeline architecture with two data paths is described and shown in Figure 2.21 to eliminate the possible conflicts. All tasks are inserted at the first pipeline stage and traverse the pipeline twice, irrespective of their starting stage (for execution) in the pipeline. Each pipeline stage accommodates two data paths (virtual data paths – they can share the same physical wires). The first data path (represented by the top lines) is active during the odd clock cycles and it is used for a first traversal of the pipeline. The second data path is traversed during even cycles and allows the task to continue its execution on the pipeline stages that are left. Once a task finishes executing, its results are propagated to the output through the final stage. Each pipeline stage works at a frequency $f = 2 \times F$ where F is the maximum throughput of the input.

For example, consider the four-stage pipeline in Figure 2.21. A task that must start executing in pipeline stage 3 is inserted in pipeline stage 1. It traverses the pipeline only in the odd cycles until it reaches stage 3 where it starts executing. Its results are forwarded to pipeline stage 4 also during an odd cycle. However, the results of the execution on stage 4 are moved forward to pipeline stage 1 for execution during the next even cycle. The task finishes its execution on pipeline stage 2. The final results are moved to the output via pipeline stages 3 and 4 during even cycles.

This solution guarantees the following features: (1) an output rate equal to the input rate; (2) all the tasks exit in order; and (3) all the tasks have a constant latency through the pipeline equal to $M \times (1/F)$ where M is the total number of pipeline stages.

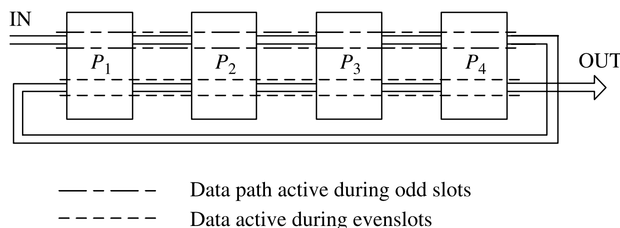


Figure 2.21 Random ring pipeline architecture with two data paths: first path is active during the odd clock cycles, used during the first traversal of the pipeline; second path is active during the even clock cycles to allow a second traversal of the pipeline [12].

2.2.8 Binary Search on Prefix Lengths

The idea of this algorithm is that the longest prefix matching operation can be decomposed into a series of exact matching operations, each performed on the prefixes with the same length. This decomposition can be viewed as a linear search of the space of $1, 2, \ldots, W$ prefix lengths. An algorithm that performs binary searches on this space was proposed by Waldvogel et al. [13].

Data Structure. The prefixes of a forwarding table are stored by length respectively, say sub-table $H_1, H_2, \ldots, H_w$. More specifically, a sub-table (say H_i) stores all the prefixes with a length of i. To reduce the exact matching time during route lookup, this algorithm uses hashing for the exact matching operation among prefixes of the same length (in the same sub-table). In other words, each sub-table uses a different hash function to hash all its prefixes. Each prefix will be associated with a hash value. If there are more than one prefix hashed to the same value, it is called 'hash collision'. They are linked together for one by one exact matching during route lookup.

Route Lookup. Given a destination address, a linear search on the space of prefix lengths requires probing each of the W hash tables, $H_1, H_2, \ldots, H_w$. It requires W hash operations and W hashed memory accesses. To shoot the probing process, the algorithm first probes $H_{w/2}$. If a node is found in this hash table, there is no need to probe tables $H_1, H_2, \ldots, H_{w/2-1}$. This is due to the requirement of longest prefix match. If no node is found, hash table $H_{w/2+1}, \ldots, H_w$ need not be probed. The remaining hash tables are probed again in a binary search manner.

Figure 2.22 illustrates how the algorithm works. The table on top of Figure 2.22 gives the five example prefixes in the forwarding table. There are three different lengths, 8, 16, and 24, of these prefixes, so three hash tables H_8, H_{16}, and H_{24} are constructed in Figure 2.22. In addition to five prefixes in the three sub-tables, there is an entry $\langle 90.2 \rangle$ in H_{16}. It is called a marker because it is not a prefix in the forwarding table but an internal node to help determine the direction of the next branch, either going to the lower part or the higher part. Given an IP address $\langle 90.1.20.3 \rangle$, we start the lookup from H_{16} (since $W = 32$ for IPv4 and $W/2 = 16$) and find a match at the entry $\langle 90.1 \rangle$. Then, we process to H_{24} and another match is found at the entry $\langle 90.1.20 \rangle$. The lookup terminates here because no more hash tables are available to be searched. The prefix $\langle 90.1.20 \rangle$ is returned as the result. Figure 2.23 illustrates an example of the binary search on prefix lengths.

Performance. The algorithm requires $O(\log_2 W)$ hashed memory accesses for one lookup operation, taking no account of the hash collision. So does the update complexity. This data structure has storage complexity of $O(NW)$ since there could be up to W markers for a prefix-each internal node in the trie on the path from the root node to the prefix. However, not all the markers need to be kept. Only the $\log_2 W$ markers that would be probed by the binary search algorithm need be stored in the corresponding hash tables. For instance, an IPv4 prefix of length 22 needs markers only for prefix lengths 16 and 20. This decreases the storage complexity to $O(N \log_2 W)$.

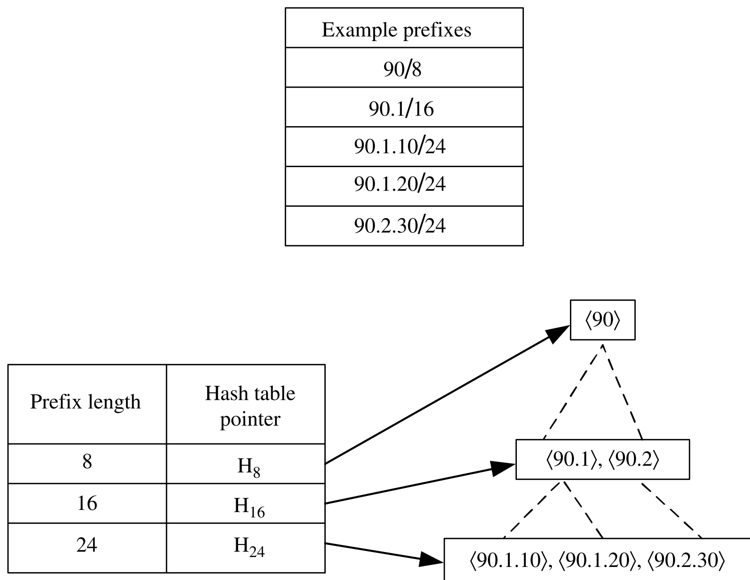


Figure 2.22 Binary search example on prefix lengths.

2.2.9 Binary Search on Prefix Range

Lampson et al. [14] showed how to apply a binary search to perform longest prefix matching lookup. This algorithmic approach to IP lookups views a prefix database as a set of intermingled ranges, where each prefix is expanded to two endpoints in a number line.

Data Structure. The left column of Figure 2.24*a* shows a sample prefix database with an assumption of 6-bit addresses. To apply a prefix-range-based binary search on this set, two endpoints are generated for each prefix by padding 0s and 1s, respectively, as shown in the right column of Figure 2.24*a*. These endpoints are mapped to a number line as shown in 2.24*b*. A table containing six endpoints based on these expanded prefixes is shown in Figure 2.24*c*. Each of the endpoints (excluding the last one) actually denotes the left boundary of a specific prefix range (total of five prefix ranges in the example). Another two fields (the '=' and the '>' fields) are created for each endpoint. This indicates the corresponding longest matching prefix when the destination address falls within the prefix range ('>') following the boundary or just falls on the (left) boundary ('=') of the prefix range, as shown in the right column of Figure 2.24*c*. These are all pre-computed.

Route Lookup. The lookup process of a given destination address is actually the process of a linear search for the nearest left boundary to the destination address in the forwarding table (since we only keep the '=' and '>' fields, but not '<', we only find the left but not right boundary). Let us use the prefix database in Figure 2.24 as an example and assume the destination address to be 101011. Then lookup the address in the forwarding table is actually to find which of the six entries is the nearest left boundary to 101011. For instance, we can see that the key falls into the range of P2 by mapping the given key to the number line. Binary search is a very efficient way to search a linear space. The first time, the key (destination address) 101011 is compared with the $\lceil 6/2 \rceil =$ third entry 101111. Then a '<'

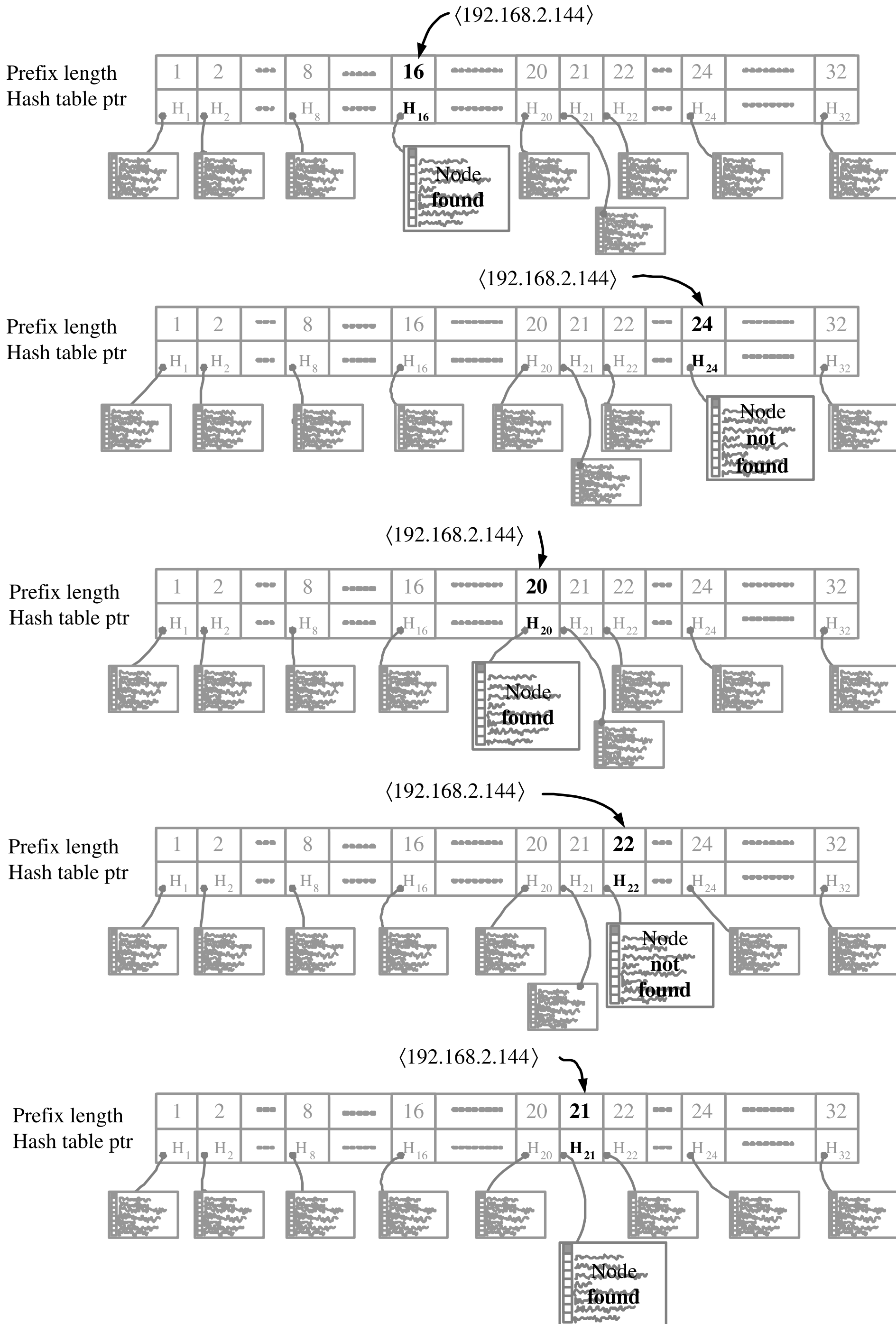
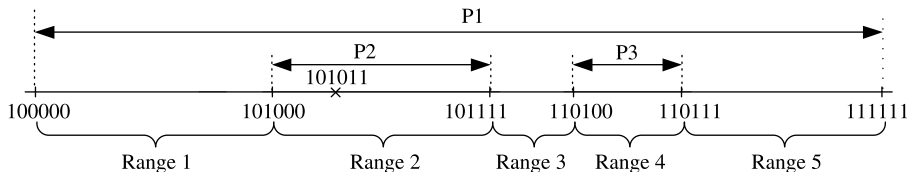


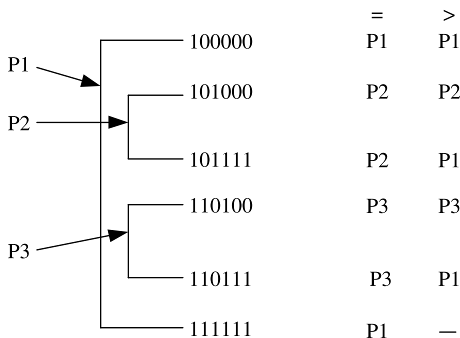
Figure 2.23 Example of the binary search on prefix lengths.

Prefix database	Range
P1: 1*	100000 – 111111
P2: 101*	101000 – 101111
P3: 1101*	110100 – 110111

(*a*)



(*b*)



(*c*)

Figure 2.24 Example for binary search on prefix range.

is returned (meaning that the entries behind the third one need not be probed), so a second search of those ahead of third entry is needed. At the second time, the key is compared with the $\lceil 3/2 \rceil = 2$nd entry, and a '>' is returned. There is no entry between the second and third ones causing the search to stop there. P2 in the '>' field of the second entry is the result.

Performance. There should be $2N$ segment points for a prefix database size of N when each prefix generates two endpoints. If a k-way search is used, the search time in the worst case will be $\log_k 2N$. Once a prefix is added or deleted, the range sequence is changed and the content of N memory locations storing the original N ranges need to be updated. The update complexity and memory space are both $O(N)$. It has been reported that by using a 200-MHz Pentium Pro-based machine and a practical forwarding table with over 32,000 route entries, the worst-case time of 490 ns and an average time of 100 ns for IP route lookups were obtained [14]. Only a 0.7-Mbyte memory was used. A drawback of this algorithm is that it does not support incremental updates.

2.3 HARDWARE-BASED SCHEMES

2.3.1 DIR-24-8-BASIC Scheme

Gupta et al. [15] proposed a route lookup mechanism that can achieve one route lookup every memory access when implemented in a pipeline fashion in hardware. It is called the DIR-24-8-BASIC scheme. This corresponds to approximately 100 Mlookups/sec with the 10-ns SDRAM technology.

Data Structure. The DIR-24-8-BASIC has two level searches as shown in Figure 2.25. The first-level search uses the first 24 bits and the second-level search (if necessary) uses the combination of index and the remaining 8 bits. The scheme makes use of the two tables shown in Figure 2.26, both stored in SDRAM. The first table (called TBL24) stores all possible route pre-fixes that are up to, and including, 24 bits long. This table has 2^{24} entries, addressed from 0.0.0 to 255.255.255. Each entry in TBL24 has the format shown in Figure 2.27. The second table (TBL*long*) stores all route prefixes in the forwarding table that are longer than 24 bits.

Assume for example that we wish to store a prefix *X* in an otherwise empty forwarding table. If *X* is less than or equal to 24 bits long, it needs only be stored in TBL24: the first bit of the entry is set to zero to indicate that the remaining 15 bits designate the next hop information. If, on the other hand, the prefix *X* is longer than 24 bits, we then use the entry

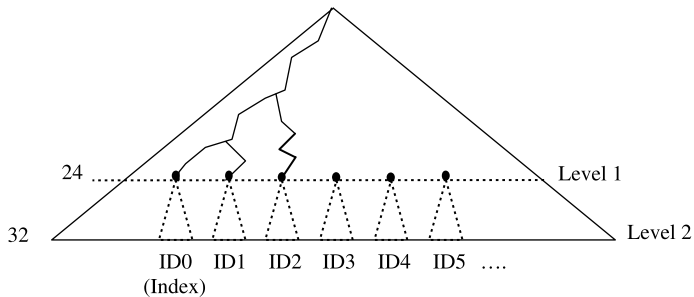


Figure 2.25 Two levels of the data structure.

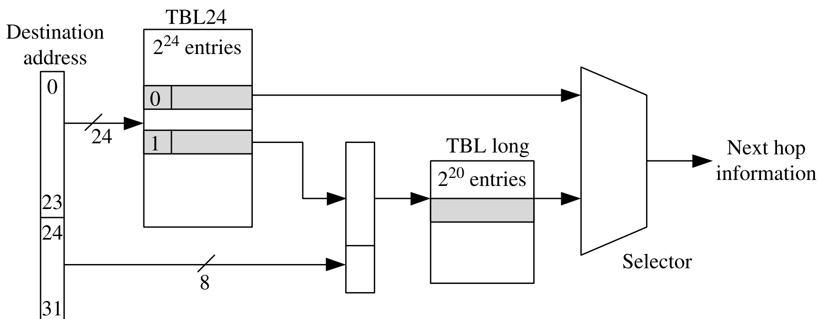


Figure 2.26 DIR-24-8 BASIC architecture.

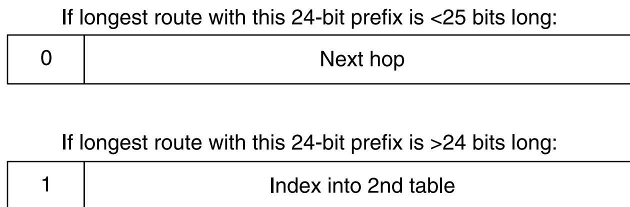


Figure 2.27 TBL24 entry format.

in TBL24 addressed by the first 24 bits of X. We set the first bit of the entry to one to indicate that the remaining 15 bits contain a pointer to a set of entries in TBL*long*.

In effect, route prefixes shorter than 24 bits are expanded. For example, the route prefix 128.23/16 will have entries associated with it in TBL24, ranging from the memory address 128.23.0 through 128.23.255. All 256 entries will have exactly the same contents (the next hop corresponding to the routing prefix 128.23/16). With the cost of a large memory, we can find the next hop information within one memory access.

TBL*long* contains all route prefixes that are longer than 24 bits. Each 24-bit prefix that has at least one route longer than 24 bits is allocated $2^8 = 256$ entries in TBL*long*. Each entry in TBL*long* corresponds to one of the 256 possible longer prefixes that share the single 24-bit prefix in TBL24. It needs to be only 1 byte wide if we assume that there are fewer than 255 next-hop routers, this assumption could be relaxed if the memory was wider than 1 byte because we simply store the next-hop in each entry of the second table.

When a destination address is presented to the route lookup mechanism, the following steps are taken: (1) We perform a single memory read yielding 2 bytes using the first 24 bits of the address as an index to the first table TBL24; (2) If the first bit equals zero, then the remaining 15 bits describe the next hop information; (3) Otherwise (if the first bit is one), we multiply the remaining 15 bits by 256, add the product to the last 8 bits of the original destination address (achieved by shifting and concatenation), and use this value as a direct index into TBL*long*, which contains the next hop information.

Consider the following example, we can see how prefixes, 10.54/16 (A), 10.54.34/24 (B), 10.54.34.192/26 (C), are stored in the two tables as shown in Figure 2.28. The first route includes entries in TBL24 that correspond to the 24-bit prefixes from 10.54.0 to 10.54.255 (except for 10.54.34). The second and third routes require that the second table be used. This is because both of them have the same first 24 bits and one of them is more than 24 bits long. In TBL24, we insert a one followed by an index (in the example, the index equals 123) into the entry corresponding to the 10.54.34 prefix. In the second table, we allocate 256 entries starting with memory location 123×256. Most of these locations are filled in with the next hop information corresponding to 10.54.34 route (B), and 64 of them [those from $(123 \times 256) + 192$ to $(123 \times 256) + 255$] are filled in with the next hop corresponding to the 10.54.34.192 route (C).

Performance. The advantages associated with the basic DIR-24-8-BASIC scheme include: (1) Generally, two memory accesses are required. These accesses are in separate memories that allow the scheme to be pipelined; (2) This infrastructure will support an unlimited number of routes, except for the limit on the number of distinct 24-bit prefixed routes with length greater than 24 bits; (3) The total cost of memory in this scheme is the

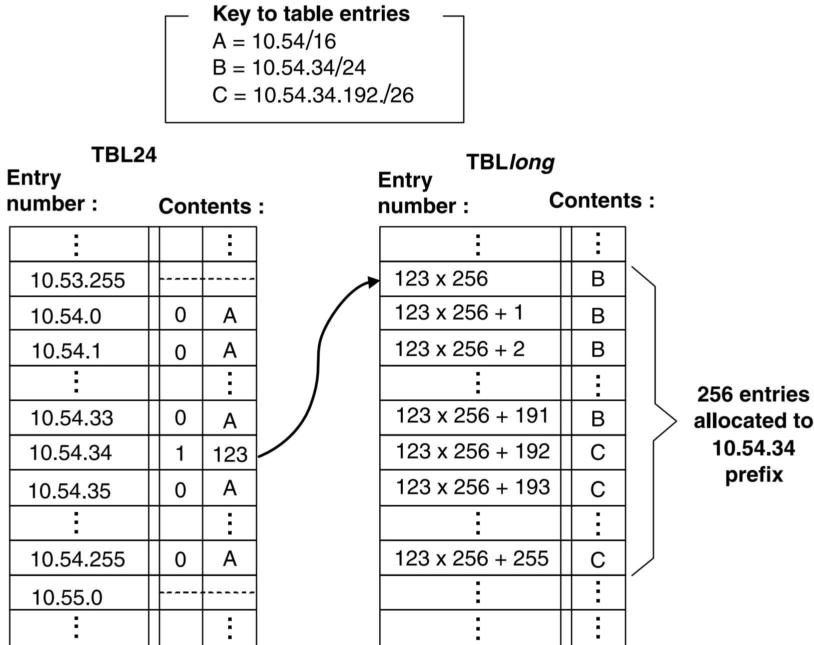


Figure 2.28 Example of two tables containing three routes.

cost of 33 Mb of SDRAM. No exotic memory architectures are required; and (4) The design is well-suited for hardware implementation.

The disadvantages are: (1) Memory is used inefficiently and (2) Insertion and deletion of routes from this table may require many memory accesses.

2.3.2 DIR-Based Scheme with Bitmap Compression (BC-16-16)

Huang [16] proposed a route lookup scheme combining the concepts of Lulea algorithms' bitmap compression (Section 2.2.5) and DIR-24-8's direct lookup. The most straightforward way to implement a lookup scheme is to have a forwarding table in which an entry is designated for each 32-bit IP address, as depicted in Figure 2.29. This design needs only one memory access for each IP route lookup, but the size of the forwarding table, next-hop array (NHA), is huge (2^{32} bytes $= 4\,$GB).

An indirect lookup can be employed to reduce the memory size (see Fig. 2.30). Each IP address is split into two parts: (a) segment (the higher 16 bits) and (b) offset (the lower 16 bits). The segmentation table has 64K entries (2^{16}) and each entry (32 bits) records either the next hop information (port number, if value $\leq$255) or a pointer (if value $>$255) pointing to the associated NHA. Each NHA consists of 64K entries (2^{16}) and each entry (8 bits) records the next hop (port number) of the destination IP address. For a destination IP address *a.b.x.y*, the *a.b* is used as the index of the segmentation table and the *x.y* is employed as the index of the associated NHA, if necessary. For a segment *a.b*, if the length of the longest prefix belonging to this segment is less than or equal to 16, then the corresponding entries of the segmentation table store the output port directly, and the associated NHA is not necessary. On the other hand, if the length of the longest prefix belonging to this

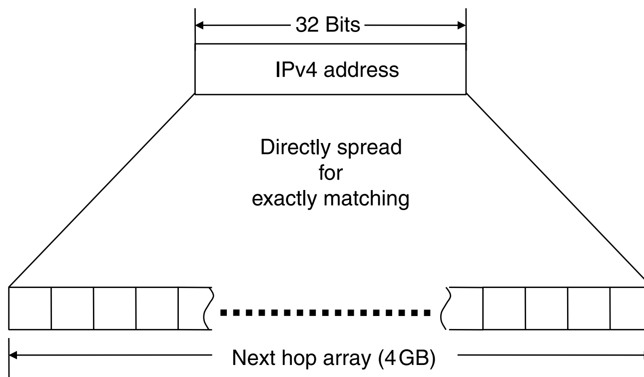


Figure 2.29 Direct lookup mechanism.

segment is greater than 16, then an associated 64 KB NHA is required. In this design, a maximum of two memory accesses are needed for an IP route lookup.

Although the indirect-lookup scheme furnishes a fast lookup (up to two memory accesses), it does not consider the distribution of the prefixes belonging to a segment. A 64 KB NHA is required as long as the length of the longest prefix belonging to this segment is greater than 16. The size of the associated NHA can be reduced further by considering the distribution of the prefixes within a segment. The IP address is still partitioned into segment (16 bits) and offset ($\leq$16 bits).

Data Structure. Figure 2.31 shows Huang's lookup mechanism. Four tables are used, the 2^{16}-entry Segment Table (Seg Table), the Code Word Array (CWA), the Compressed Next Hop Array (CNHA), and the Next Hop Array (NHA).

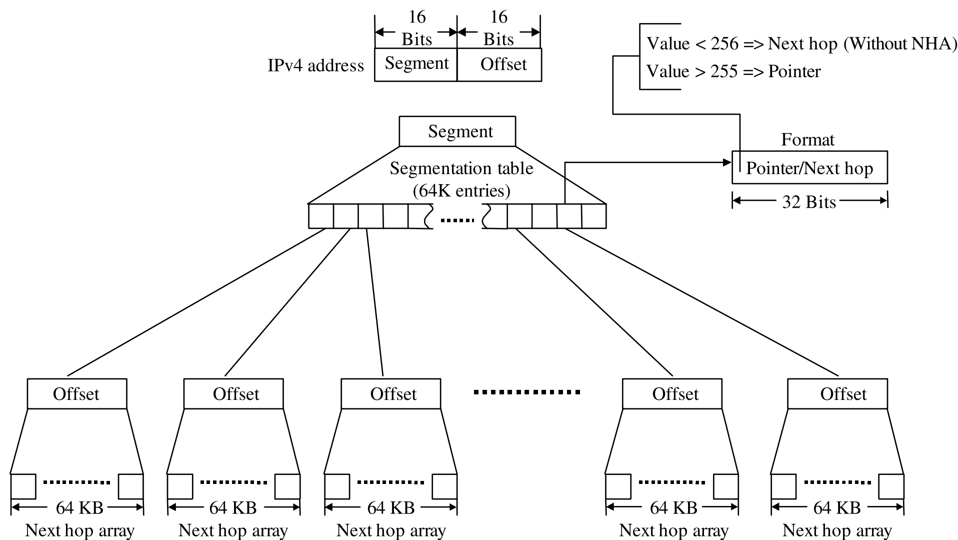


Figure 2.30 Indirect-lookup mechanism.

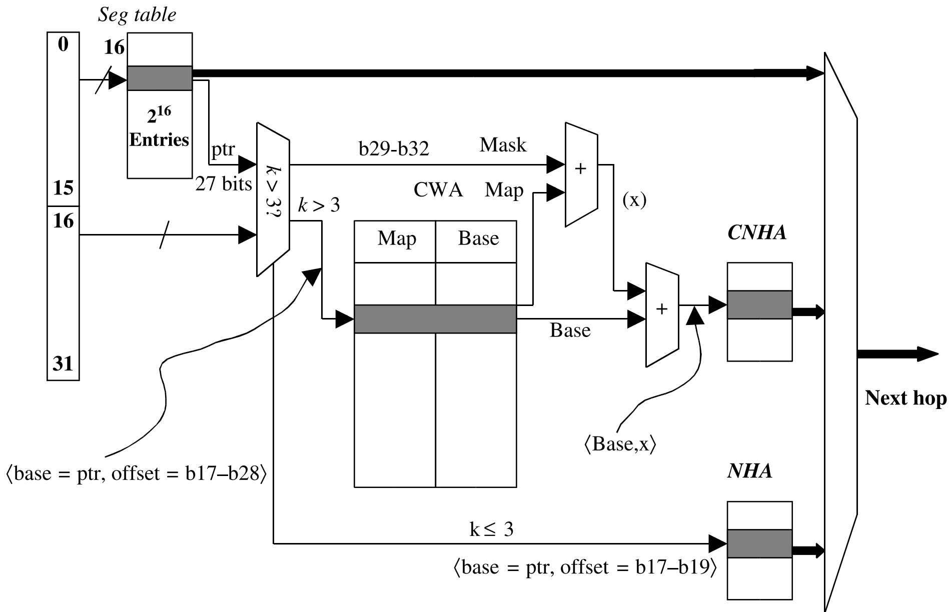


Figure 2.31 Multi-bit trie algorithm with bitmap compression technique in BC-16-16.

Similar to the DIR-24-8, the segment table in BC-16-16 is with the size of 64 KB entries, and each one (32 bits) is divided into three fields: *F* (Flag, 1 bit), pointer/next hop (27 bits) and offset length (4 bits) (as shown in Fig. 2.32). The first field indicates if the second field contains a pointer to the next level table structure or a next hop index; the second field records either the next hop (port number, if value $\leq$255) of the routing, or a pointer (if value $>$255) pointing to the associated NHA or a second level table structure (CWA-CNHA). The offset length field shows the size of NHA table, in terms of the power of 2. If the size is not greater than eight (i.e., field value $k \leq 3$), the pointer will point to an entrance of a common NHA (a table containing a series of next hop index). If the size is greater than eight (i.e., field value $>$3), the pointer will point to a CWA. We can decode the CNHA and find out the next hop by searching the CWA. The frameworks of the CWA and the CNHA are shown in Figures 2.33 and 2.34, respectively.

F (1 bit)	*Pointer/Next hop (27 bits)*	*Offset length k (4 bits)*

Figure 2.32 Segment table entry format.

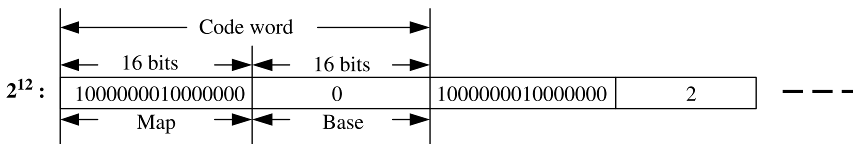


Figure 2.33 Code word array (CWA).

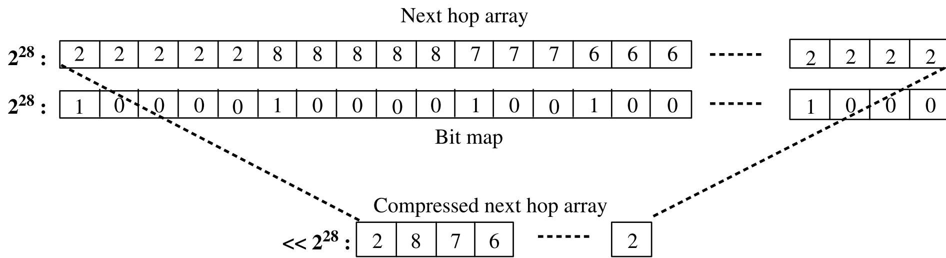


Figure 2.34 Compressed next hop array (CNHA).

Each `Code Word` in the CWA is composed of 2 parts: `Map` and `Base`. The `(Bit) Map` indicates the distribution of the next hops, while `Base` is the summing-up of the '1's in the previous `Maps`. Each bit in the `Maps` corresponds to an entry in the original NHA, and the '1's in the bitmap indicate the beginning of a prefix area. The NHA with 2^{28} entries is compressed into the CNHA as shown in Figure 2.34. By counting the '1's in the bitmap, we can find out the next hop in the CNHA.

Route Lookup. The first 16 bits (segment) of the incoming IP address are used to lookup the Seg Table. If the most significant bit (F field) is a '0', indicating the following field contains a next hop index (output port number), then the lookup process stops and the next hop index is returned. Otherwise, if the F field is a '1', then we inspect the offset length field. If the offset length ≤ 3, we use the pointer field as the base address, and the $(16 + 1)$th to $(16 +$ offset length$)$th bits of the IP address as the offset address to lookup the NHA table, and return the associated result. If offset length >3, we should then lookup the CWA table with the following steps: (1) Use the pointer field as the base address, and the 17th to 28th bits as the offset address to find the corresponding Code Word in the CWA table, and get the corresponding `Map` and `Base`; (2) Define the 29th to 32nd bits of the IP address to be p, and calculate the number of '1's in the most significant p bits in the `Map` field of the Code Word, say x; (3) Use `Base` as the base address and x as the offset address to lookup the CNHA table, and return the associated result (next hop index).

Performance. The basic idea of this lookup scheme is derived from the Lulea algorithm in Section 2.2.5, which uses bitmap code to represent part of the trie and significantly reduce the memory requirement. The main difference between BC-16-16 and the Lulea algorithm is that the former is hardware-based and the latter is software-based. The first-level lookup of the BC-16-16 uses direct 16-bit address lookup while the Lulea scheme uses bitmap code to look up a pointer to the next level data structure.

BC-16-16 needs only a tiny amount of SRAM and can be easily implemented in hardware. Based on the data obtained from [17], a large forwarding table with 40,000 routing entries can be compacted to a forwarding table of 450–470 kbytes. Most of the address lookups can be done by one memory access. In the worst case, the number of memory accesses for a lookup is three. When implemented in a pipeline in hardware, the proposed mechanism can achieve one route lookup every memory access. This mechanism furnishes approximately 100 M route lookups per second with current 10 ns SRAM.

However, this algorithm does not support incremental updates. When the CWA table needs to be updated, the whole second level table, including the associated CWA and CNHA should be reconstructed.

2.3.3 Ternary CAM for Route Lookup

Basic TCAM Scheme. CAM is a specialized matching memory that performs parallel comparison. The CAM outputs the location (or address) where a match is found. Conventional CAM can only perform exact matching, when presenting a parallel word to the input, and cannot be applied to CIDR 1P route lookup. A ternary CAM (TCAM) stores each entry with a (val, mask) pair, where val and mask are both W-bit numbers. For example, if $W = 6$, a prefix 110* is stored as the pair (110000, 111000). Each TCAM element matches a given input key by checking if those bits of val for which the mask bit is 1 match those in the key.

The logical structure of a TCAM device is shown in Figure 2.35. Whenever a matching operation is triggered, the 32-bit destination IP address will be compared with all the TCAM entries bit by bit, respectively and simultaneously. Since there may be multiple matches found at the same time, priority resolution should be used to select a match with the highest priority as the output. Many commercial TCAM products are order-based, such that the priority is determined by the memory location. The lower the location, the higher is the priority. Namely the ith TCAM has higher priority than the jth TCAM, if $i < j$. The priority arbitration unit selects the highest priority matched output from the TCAMs. For instance, an IP address 192.168.0.177 fed to the TCAM in Figure 2.35 results in four matches at the locations of 1, 1003, 1007, and 65535. The location of 1 is selected.

The forwarding table is stored in the TCAM in decreasing order of prefix lengths, so that the longest prefix is selected by the priority encoder. As shown in Figure 2.35, the group of 32-bit prefixes are at the bottom of the TCAM. Note that there are some empty spaces

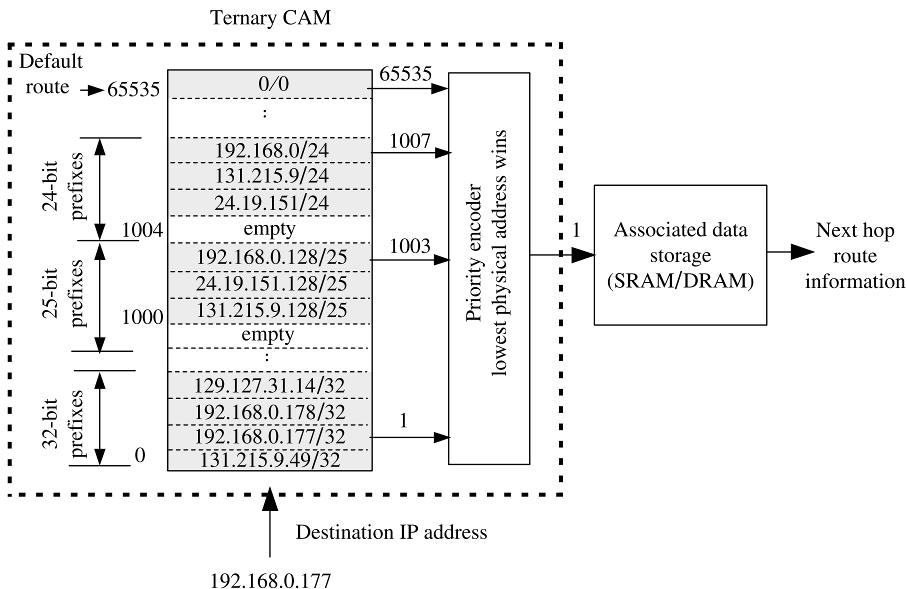


Figure 2.35 Logic structure of a TCAM device.

in some groups reserved for adding prefixes in the future. The default route is located at the very top of the TCAM chip. Its mask value is 0, which guarantees that it will match with any input IP address. Only when there is no match from all the locations below it, will it be selected by the priority arbitrator. The output from the TCAM is then used to access RAM, in which the next hop information is stored in the same location as the prefix in the TCAM.

Performance. TCAM returns the result of the longest matching prefix lookup within only one memory access, which is independent of the width of the search key. And the implementation of TCAM-based lookup schemes are commonly used because they are much simpler than that of the trie-based algorithms. The commercially available TCAM chips [18, 19] can integrate 18 M-bit (configurable to 256 k $\times$ 36-bit entries) into a single chip working at 133 MHz, which means it can perform up to 133 million lookups per second. However, the TCAM approach has the disadvantage of high cost-to-density ratio and high-power consumption (10–15 Watts/chip) [20].

2.3.4 Two Algorithms for Reducing TCAM Entries

TCAM-based IP route lookup is fast and easy to implement, but it also has high system cost and power consumption. Liu et al. [21] proposed two simple schemes to circumvent the above drawbacks. The TCAM size can be reduced by minimizing the amount of redundancy that exists among the prefixes in the forwarding table. Reducing the TCAM size will save on cost and power consumption.

Prefix Pruning. Figure 2.36 shows a situation where a part of the prefix trie is given. Assume that P1 and P2 have the same forwarding information. Then, P2 is actually a redundant prefix because deleting P2 will not affect the lookup result in any way. The direct ancestor of P2 (i.e., P1) on the prefix trie contains the same forwarding information (to port 2) with P2. For a longest matching prefix lookup that should terminate at P2 originally, it will terminate at P1 if P2 is pruned.

Many real-life IP forwarding tables have a substantial number of redundant prefixes. Pruning the redundant prefixes before storing them into TCAM will significantly reduce the TCAM requirement. For instance, an original prefix table in Figure 2.37 with seven prefixes can be reduced to a table with only four prefixes. It is reported that ~20–30 percent of the prefixes are redundant. This equates to ~20–30 percent TCAM space that can be saved by using the prefix pruning technique [21].

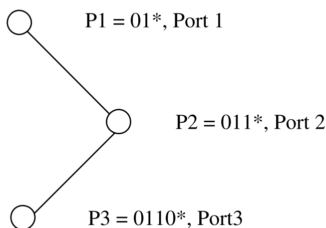
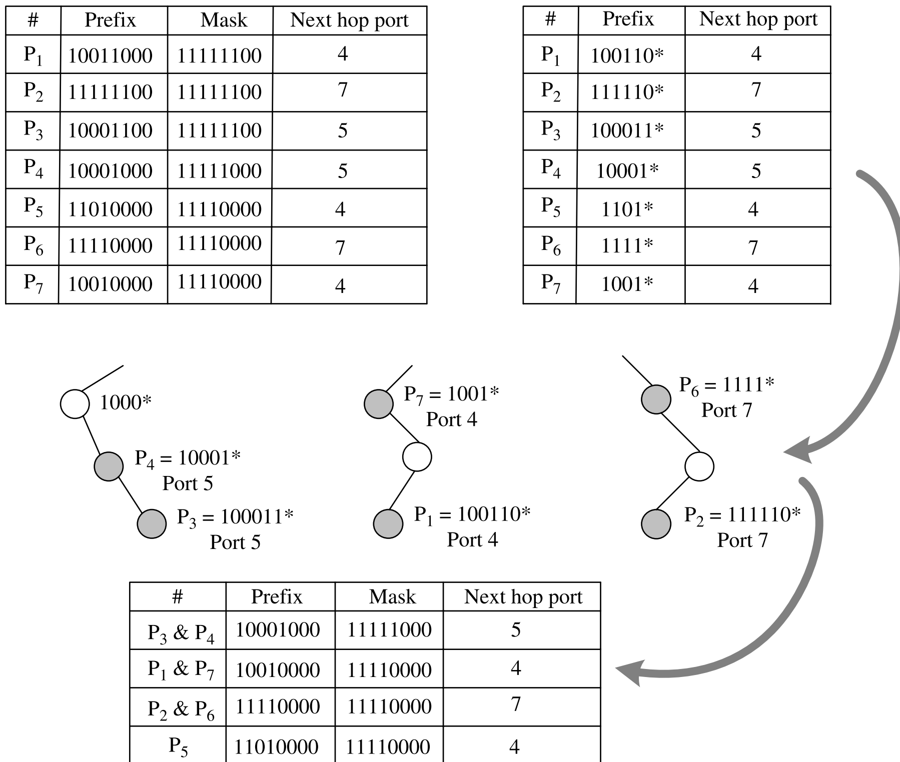


Figure 2.36 Pruning example.



#	Prefix	Mask	Next hop port
P_1	10011000	11111100	4
P_2	11111100	11111100	7
P_3	10001100	11111100	5
P_4	10001000	11111000	5
P_5	11010000	11110000	4
P_6	11110000	11110000	7
P_7	10010000	11110000	4

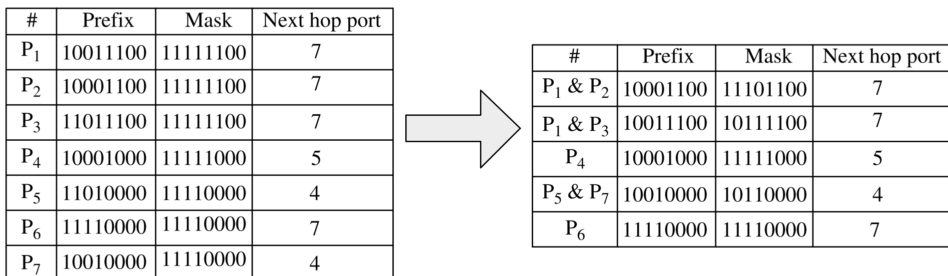
#	Prefix	Next hop port
P_1	100110*	4
P_2	111110*	7
P_3	100011*	5
P_4	10001*	5
P_5	1101*	4
P_6	1111*	7
P_7	1001*	4

#	Prefix	Mask	Next hop port
P_3 & P_4	10001000	11111000	5
P_1 & P_7	10010000	11110000	4
P_2 & P_6	11110000	11110000	7
P_5	11010000	11110000	4

Figure 2.37 Compacted table using prefix pruning.

Mask Extension. TCAM is used for ternary matching, and not just prefix matching. The main difference between these two kinds of matching is that prefix matching needs the mask to be all '1's in the most significant bit and all '0' in the rest less significant bit. Ternary matching uses free types of mask. The main idea of mask extension is to extend the prefix-match-kind of masks to the ternary-match-kind of masks in the TCAM-based route lookup scheme. In this case, some TCAM entries in the prefix-match-kind can be represented by only one ternary-match-kind entry. For instance, the seven entries in an original prefix table can be fully represented by the five entries in the mask extended table (Fig. 2.38), when ternary-match-kind of masks are used. It is reported that nearly 20–30 percent of the original TCAM entries can be further saved if the mask extension technique is adopted [21].

Performance. Real-life forwarding tables can be represented by much fewer TCAM (ternary match) entries, typically 50–60 percent of the original size by using the above two techniques. Prefix pruning would cause no change in prefix update complexity. Mask extension increases update complexity. Many prefixes are associated with others after mask extension, which results in the obstacles of performing incremental updates.



#	Prefix	Mask	Next hop port
P_1	10011100	11111100	7
P_2	10001100	11111100	7
P_3	11011100	11111100	7
P_4	10001000	11111000	5
P_5	11010000	11110000	4
P_6	11110000	11110000	7
P_7	10010000	11110000	4

#	Prefix	Mask	Next hop port
P_1 & P_2	10001100	11101100	7
P_1 & P_3	10011100	10111100	7
P_4	10001000	11111000	5
P_5 & P_7	10010000	10110000	4
P_6	11110000	11110000	7

Figure 2.38 Compacted table using mask extension.

2.3.5 Reducing TCAM Power – CoolCAMs

TCAMs are fast and simple to manage, but they suffer from high power consumption. Minimizing the power budget for TCAM-based forwarding engines is important to make them economically viable. Zane et al. [22] proposed some algorithms to make TCAM-based forwarding tables more power efficient. Present-day TCAM provides power-saving mechanisms by selectively addressing small portions of the TCAM, called blocks. A block is a contiguous, fixed-size chunk of TCAM entries, usually much smaller than the size of the entire TCAM. The key idea for the CoolCAMs architecture is to split the entire forwarding table into multiple sub-tables or *buckets*, where each bucket is laid out over one or more TCAM blocks. Two different table-splitting schemes have been proposed: *Bit-Selection Architecture* and *Trie-Based Table Partitioning* [22].

Bit-Selection Architecture. Figure 2.39 shows a fixed set of bits (labeled as *suitable ID*) of the input IP address used to hash to one of the buckets (the shaded area in the *data TCAM*). Then, the IP address is compared to all the entries within the selected buckets and the index of the matched entry is used to address the associated SRAM to get the next hop information. The bit-selection logic in front of the TCAM is a set of muxes that can be programmed to extract the hashing bits from the destination address and use them to index to the appropriate TCAM bucket. For example, in Figure 2.39, each of the three 32 : 1 muxes uses a 5-bit value ($c0$, $c1$, and $c2$) to pick one bit from the incoming 32-bit address. The set of hashing bits can be changed over time by reprogramming the selectors of the muxes.

Because only a very small percentage of the prefixes in the core forwarding tables (less than 2 percent [22]) are either very short (<16 bits) or very long (>24 bits), they are grouped into the minimum possible number of TCAM blocks and will be searched for every lookup. The remaining 98 percent of the prefixes that are 16 to 24 bits long, called *split set*, are partitioned into buckets.

Assume that the total number of buckets $K = 2^k$ is a power of 2. Then the bit selection logic extracts a set of k hashing bits from the destination address and selects a bucket to be searched. As in the example of Figure 2.39, the total number of buckets is $K = 8$ and the number of hashing bits is $k = 3$. The hashing bits should be chosen from the first 16 bits, which is the minimum length of a prefix in the split set. On the other hand, if k' of the hashing bits are in bit positions larger than the length of a prefix, this prefix needs to be replicated in $2^{k'}$ buckets. For example, if we choose $k' = 2$ bits from bit positions 19 and

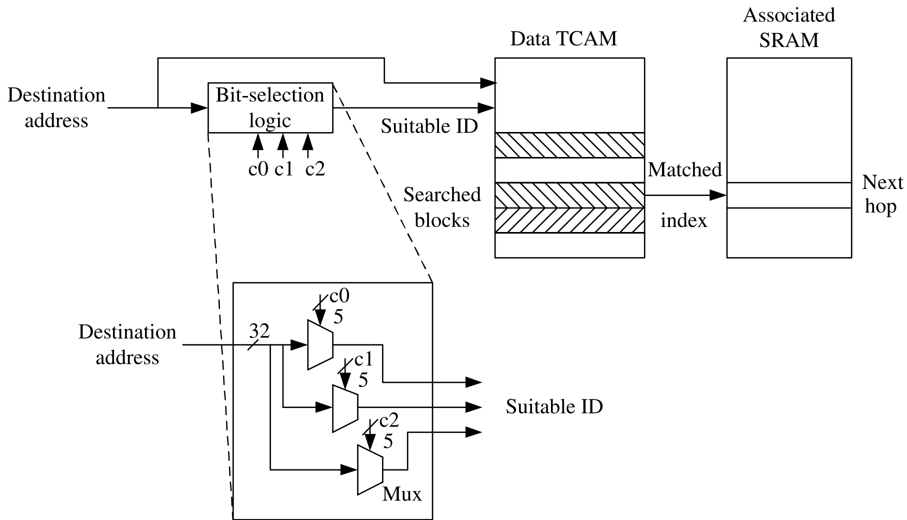


Figure 2.39 Forwarding engine architecture for using bit selection to reduce power consumption. The three hashing bits are selected from the 32-bit destination address by setting the appropriate 5-bit values of $c0$, $c1$, and $c2$.

20 as hashing bits, then for a 18-bit prefix, it should be extended to $2^{k'}$ (or 4) 20-bit prefixes and stored in the four corresponding buckets.

Trie-Based Table Partitioning. This approach uses a prefix trie (i.e., the 1-bit trie as shown in Fig. 2.6) to get the ID of the proper TCAM bucket. Figure 2.40 illustrates the prefix trie contained in a small-sized TCAM called the *index TCAM*. Each input is first fully searched in the index TCAM and then addressed into an *index SRAM* which contains the ID of the TCAM bucket.

Trie-based table partitioning works in two steps. In the first step, a binary routing trie is constructed from a given forwarding table. In the second step, subtrees or collections of subtrees of the 1-bit trie are successively carved out and mapped into individual TCAM buckets. The second step is called the *partitioning step*. There are two different partitioning schemes and are described below. Figure 2.41 shows a 1-bit trie that will be used as an example in both schemes. Here, the number of forwarding table prefixes in the subtree rooted at a node v is defined as the *count* of v. For any node u, the prefix of the lowest

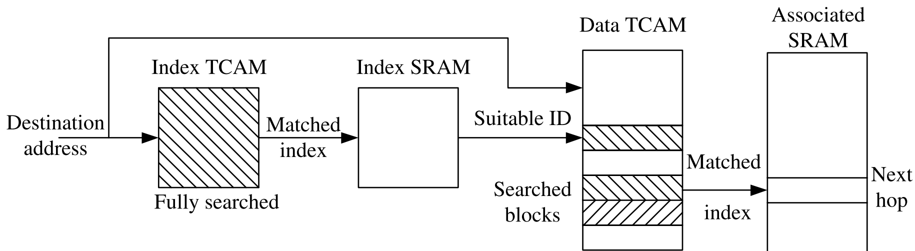


Figure 2.40 Forwarding engine architecture for the trie-based power reduction schemes.

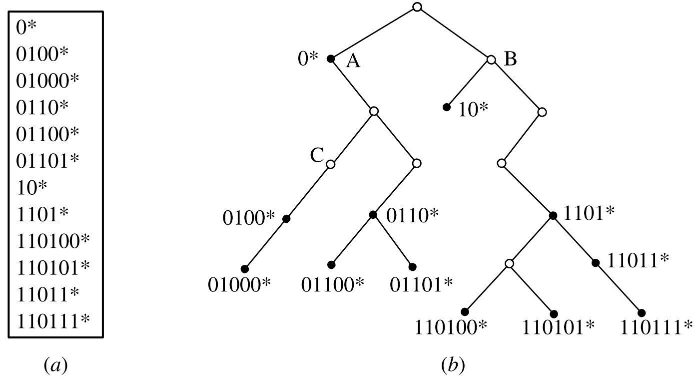


Figure 2.41 (*a*) Example of forwarding table; (*b*) corresponding 1-bit trie.

common ancestor of u (including u itself) that is in the forwarding table is called the *covering prefix* of u. For example, both nodes A and C in Figure 2.41*b* have the covering prefix A. If there are no nodes in the path of a certain node to the root whose prefix is in the forwarding table, the covering prefix of this node is nil, such as B.

Subtree Splitting. Let N denote the number of prefixes in a forwarding table and b the maximum number of prefixes in a TCAM bucket. This algorithm produces a set of $K \in [\lceil N/b \rceil, \lceil 2N/b \rceil]$ TCAM buckets, each with a size in the range $[\lceil b/2 \rceil, b]$ (except possibly the last bucket, whose size is in the range $[1, b]$), and an index TCAM of size K. During the partitioning step, the entire trie is traversed in post-order looking for a *carving node*, which is a node v whose count is at least $\lceil b/2 \rceil$ and whose parent exists and has a count greater than b. Every time a carving node v is encountered (not necessary a prefix), the entire subtree rooted at v is carved out and the prefixes in the subtree are placed into a separate TCAM bucket. Next, the prefix of v is placed in the index TCAM and the covering prefix of v is added to the TCAM bucket. This ensures a correct result is returned when an input address that matches an entry in the index TCAM has no matching prefix in the corresponding subtree. Finally, the counts of all the ancestors of v are decreased by the count of v. When there are no more carving nodes left in the trie, the remaining prefixes (if any) are put in a new TCAM bucket with an index entry of an asterisk ($*$) in the index TCAM.

Figure 2.42 shows how subtrees are carved out of the 1-bit trie from Figure 2.41. The number at each node u denotes the current value of $count(u)$. The arrows show the path along with $count(u)$ is updated in each iteration, while the circle denotes the subtree that is carved. Table 2.2 shows the final results.

Post-order Splitting. This algorithm partitions the forwarding table into buckets that each contain exactly b prefixes (except possibly the last bucket). The algorithm traverses the 1-bit trie in post-order and successively carves out subtree collections that form a bucket. If a node v is encountered such that the count of v is b, a new TCAM bucket is created, the prefix of v is put in the index TCAM and the covering prefix of v is put in the TCAM bucket. If $count(v)$ is x such that $x < b$ and the count of v's parent is $>b$, then a recursive carving procedure is performed. Denote the node next to v in post-order traversal as u. Then the subtree rooted at u is traversed in post-order, and the algorithm attempts to carve out a

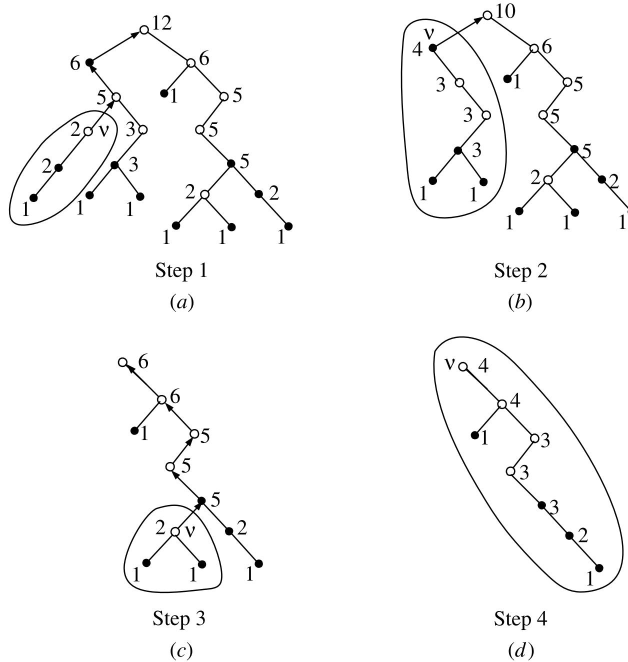


Figure 2.42 Four iterations of the subtree-split algorithm (with parameter b set to 4) applied to the 1-bit trie from Figure 2.41.

TABLE 2.2 Four Resulting Buckets from the Subtree-Split Algorithm

Index	Bucket Prefixes	Bucket Size	Covering Prefix
010*	0100*, 01000*	2	0*
0*	0*, 0110*, 01100*, 01101*	4	0*
11010*	110100*, 110101*	2	1101*
*	10*, 1101*, 11011*, 110111*	4	—

subtree of size $b - x$ from it. In addition, the x entries are put into the current TCAM bucket (a new one is created if necessary), and the prefix of v is added to the index TCAM and made to point to the current TCAM bucket. Finally, when no more subtrees can be carved out in this fashion, any remaining prefixes less than b in number are put in a new TCAM bucket. An asterisk ($*$) entry in the index TCAM points to the last bucket. Figure 2.43 shows a sample execution of the algorithm with $b = 4$ and Table 2.3 lists the final result.

Performance. The complexity for the post-order traversal is $O(N)$. Updating the counts of nodes all the way to root when a subtree is carved out gives a complexity of $O(NW/b)$. This is where W is the maximum prefix length and $O(N/b)$ is the number of subtrees carved out. The total work for laying out the forwarding table in the TCAM buckets is $O(N)$. This makes the total complexity for subtree-split $O(N + NW/b)$. It can be proved that the total

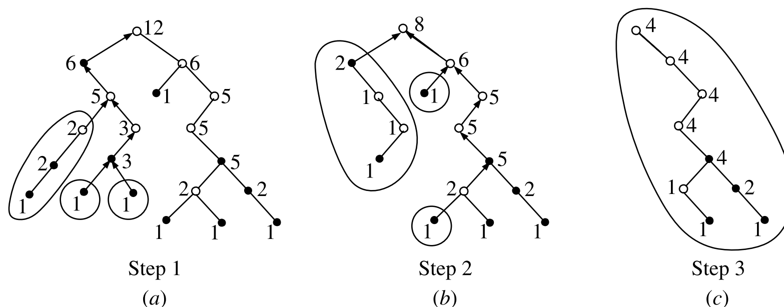


Figure 2.43 Three iterations of the post-order split algorithm (with parameter b set to 4) applied to the 1-bit trie from Figure 2.41.

TABLE 2.3 Three Resulting Buckets from the Post-Order Split Algorithm

i	$Index_i$	Bucket Prefixes	Size	Covering Prefix
1	010*, 01100*, 01101*	0100*, 01000*, 01100*, 01101*	4	0*, 01100*, 01101*
2	0*, 10*, 110100*	0*, 0110*, 10*, 110100*	4	0*, 10*, 110100*
3	1*	110101*, 1101*, 11011*, 110111*	4	—

running time for post-order split is also $O(N + NW/b)$. The drawback of subtree-split is that the smallest and largest bucket sizes vary by as much as a factor of 2.

This method requires the entire index TCAM to be searched every time. The algorithm has to ensure that the index TCAM is small enough as compared to the data TCAM and does not contribute significantly to the power budget. The size of index TCAM for subtree-split is exactly the number of buckets in data TCAM. Post-order split adds at most $W + 1$ entries to the index TCAM and W covering prefixes to the bucket in the data TCAM because the maximum number of times for carve-exact procedure is $W + 1$, which equals the number of prefixes added to the index TCAM for any given TCAM bucket.

Experimental results with real-life prefix database show that all three methods can dramatically reduce power consumption in practice. Even with only eight buckets, the bit-selection algorithm results in reduction factors of about 7.55; and subtree-split and post-order split of 6.09 and 7.95, respectively.

2.3.6 TCAM-Based Distributed Parallel Lookup

Zheng et al. [23] proposed an ultra-high throughput and power efficient TCAM-based lookup scheme. The throughput of TCAM-based lookup is determined by the TCAM access speed. The memory access rate increases ~7 percent per year. This is far behind the optical transmission increase rate that roughly doubles every year [24]. By using the chip-level-parallelism technique, we can improve the lookup throughput and so meeting the needs of the next generation routers. However, if one just duplicates a forwarding table into multiple copies and stores each copy into a TCAM device, the cost and power consumption will be prohibitively high.

By analyzing several real-life forwarding tables, it has been observed that the prefixes in a forwarding table can be approximately evenly partitioned into groups, called ID groups.

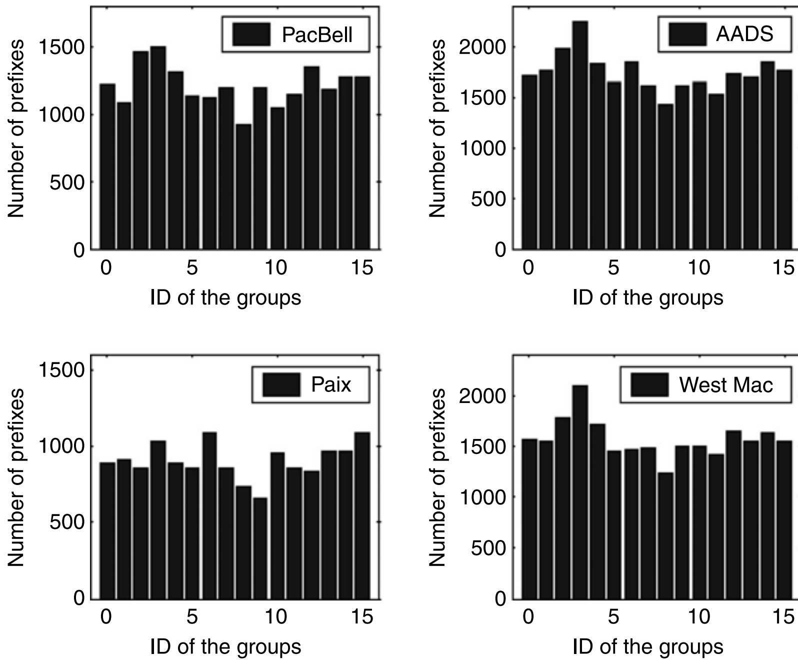


Figure 2.44 Prefixes distribution among ID groups in four real-life forwarding tables.

The partition is based on a few certain bits of the prefixes. Figure 2.44 illustrates 16 ID groups created by using the 10–13th bits of the prefixes in the forwarding table of the routers at four popular sites. Note that using more significant bits of the IP address may not obtain uniform classification, while using less significant bits may need to expand many short prefixes to long ones. Using the 10–13th bits of the IP addresses as their ID bits seems to be a good choice.

Data Structure. Suppose that there are K (e.g., $K = 4$) TCAM chips and each TCAM chip has n (e.g., $n = 5$) partitions. Each ID group is approximately equal to $N/16$ prefixes, where N denotes the total number of prefixes in the forwarding table. These ID groups are then distributed to K TCAM chips (each ID group is stored in a TCAM). The goal is to have incoming packets that belong to different ID groups access the TCAM chips as evenly as possible. More TCAM chips working in parallel results in a higher lookup throughput.

In order to further balance the lookups among the TCAM chips, some of the ID groups with large traffic load ratio may be stored in multiple TCAM chips. For example, ID group 0, 4 and 11 are stored twice in the example in Figure 2.45. Therefore, the lookups for these three ID groups may be shared between the TCAM chips that have copies of them.

Route Lookup. An implementation architecture for the distributed parallel lookup scheme is shown in Figure 2.46. For a given IP address to be searched, firstly, 4 bits (10–13th bits) are extracted to determine which TCAM chips contain the ID group that matches the IP address. This is implemented by the Indexing Logic. The output will be multiple TCAM candidates because of the redundant storage. A Priority Selector will pick

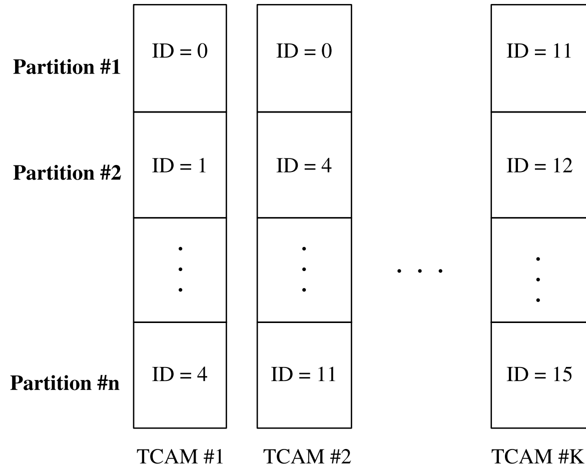


Figure 2.45 Example of the TCAM organization.

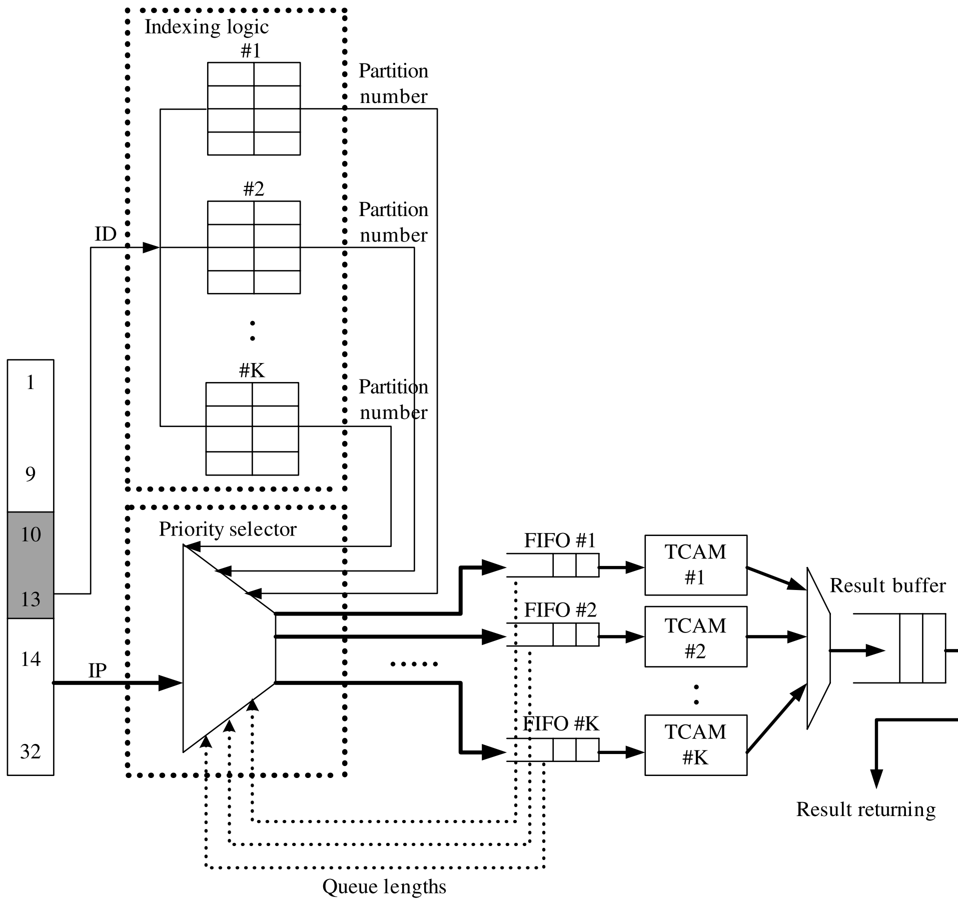


Figure 2.46 Implementation of the parallel lookup scheme.

a TCAM chip that is the least busy among the candidates (e.g., by comparing the first-in first-out (FIFO) queue lengths of the TCAM chips).

During the TCAM access cycle, each of the K TCAM chips fetch an assigned IP address from the FIFO queue, and performs a longest prefix match lookup independently. At the output ports of the TCAM chips, an Ordering Logic reads out the lookup results and returns them in their original sequence according to the time stamps attached to them.

Performance. The lookup throughput can be significantly improved with chip level parallelism. The proposed scheme can achieve a peak lookup throughput of 533 million packets per second (mpps) using four 133 MHz TCAM chips and having 25 percent more TCAM entries than the original forwarding table [24]. This performance can readily support a line rate of 160 Gbps. The disadvantage is that route update can be quite complicated.

2.4 IPV6 LOOKUP

2.4.1 Characteristics of IPv6 Lookup

Internet Protocol Version 6 (IPv6) is one of the key supporting technologies of the next generation network. According to the Internet Architecture Board (IAB), a unicast IPv6 address consists of two parts: a 64-bit network/sub-network ID followed by a 64-bit host ID. To enable a smooth transition from IPv4 to IPv6, the *IPv4-mapped address* and *IPv4-compatible address* formats were introduced. The former is defined by attaching the 32-bit IPv4 address to a special 96-bit pattern of all zeros. The IPv4-mapped address starts with 80 bits of zeros and 16 bits of ones, followed by the 32-bit IPv4 address.

The IAB and IESG [25] recommend that, in general, an address block with a 48-bit prefix be allocated to a subscriber. Very large subscribers could receive a 47-bit prefix or slightly shorter prefix, or multiple 48-bit prefixes. A 64-bit prefix may be allocated when it is known that one, and only one, subnet is needed; and a 128-bit prefix is allocated when it is absolutely known that one, and only one, device is connecting to the network. It is also recommended that mobile devices be allocated 64-bit prefixes.

From related recommendations in request for comments (RFC) and Réseaux IP Européens (RIPE) documents, the following important characteristics can be obtained: (1) It is obvious but important to note that there is no prefix with lengths between 64 bits and 128 bits (excluding 64 bits and 128 bits); (2) The majority of the prefixes should be the '/48s' and '/64s' the secondary majority. Other prefixes would be distinctly fewer than the '/48s' and '/64s'; (3) Specifically, the number of '/128s' should be tiny, which would be similar to the ratio of the '/32s' in the case of IPv4. Figure 2.47 shows the prefix length distribution of an IPv6 routing table reported in [26]. In this routing table, there are a total of 567 prefixes where only one prefix is longer than 64 bits.

2.4.2 A Folded Method for Saving TCAM Storage

Because of the high lookup rate and simplicity of table management, TCAM is currently the most popular solution for address lookup and packet classification. Commercial TCAM

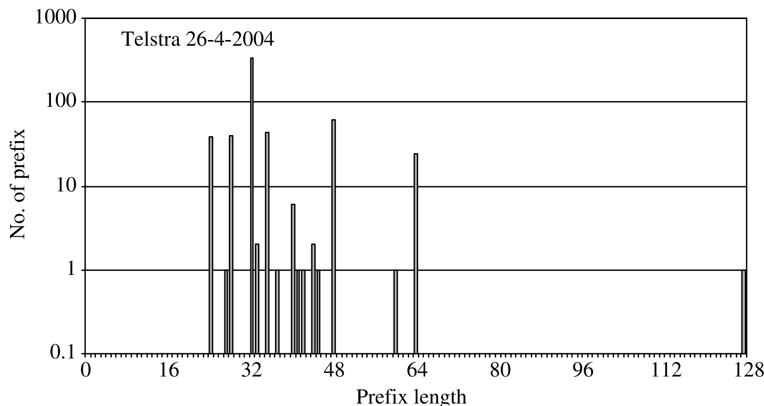


Figure 2.47 Prefix length distribution of an IPv6 routing table.

devices available today support configurable word-lengths of 36-, 72-, 144-, 288-, and 576-bits, and there are 72 input/output (I/O) pins for inputting the search key [27]. Double data rate I/O is supported, so there will be no extra overhead in handling search keys that are up to 144 bits in length. In a straightforward implementation of a route lookup engine using TCAM, a word-length of 144-bits will be selected. However, as mentioned in Section 2.4.1, the majority of IPv6 prefixes are no longer than 64 bits. Hence, over 50 percent of a TCAM word will store the 'don't care' value. Pao [28] has recently presented a simple but effective approach to improve the space efficiency by over 40 percent.

In his approach [28], the TCAM is configured with 72-bit words. The route prefixes are divided into two groups, G_S and G_L. G_S contains all the prefixes with no more than 72 bits, and G_L contains all the prefixes with more than 72 bits. The TCAM blocks are divided into two partitions, P_S and P_L. Prefixes in G_S are stored in P_S. Prefixes in G_L are grouped by the value of the first 72 bits. Routes a and b are put in the same subgroup if a and b share a common 72-bit prefix M. Assume the number of subgroups of long prefixes is less than 64k. Each subgroup is assigned a distinct 16-bit tag T. The common prefix M of a subgroup is inserted into the partition P_S of the TCAM to serve as a *marker*. An entry in the routing table is a six-tuple $\langle$*prefix, p, m, v, next-hop, T*$\rangle$, where the flags p and m indicate whether the entry is a prefix, a marker, or both. If the next-hop value is valid, then $v = 1$; otherwise, $v = 0$. For a marker, the next-hop field records the next hop of the longest prefix in G_S that matches M, if any. Let a be a prefix with l bits where $l > 72$. The bits are numbered from 1 to l starting from the left. We define the suffix of a as the substring consisting of bits 73 to l. The first 72 bits of a will be stored in partition P_S, and the suffix (concatenated to the tag T) is stored in partition P_L. The entries in the two partitions are ordered by their lengths. An example of the two-level routing table organization is shown in Figure 2.48, where a full-length address has 12 bits and a tag has 4 bits. A wildcard entry is inserted into P_S to represent the default route, and a wildcard entry with $v = 0$ is inserted into P_L to serve as a sentinel to simplify the handling of the boundary condition.

Address lookup is a two-step process. First, the most significant 72 bits of the destination address A is extracted and the partition P_S is searched. If the best matching entry found is not a marker, then the packet is forwarded to the next-hop value returned by the address lookup engine. If the best matching entry is a marker, then the 56-bit suffix of A is extracted and concatenated with the 16-bit tag of the marker to form a key to search the partition

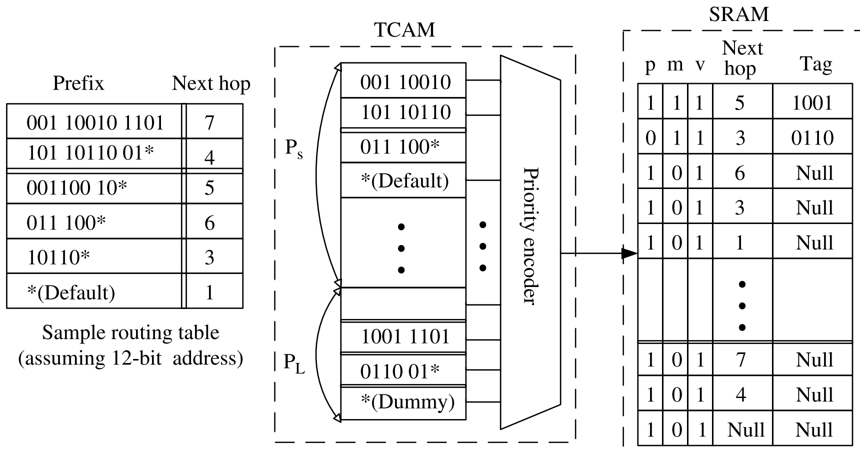


Figure 2.48 Two-level routing table organization in TCAM.

P_L. If the search result is invalid, that is, $v = 0$, then the packet will be forwarded to the next-hop value found in step 1; otherwise, the packet will be forwarded using the next-hop found in step 2.

If there are more than 64k subgroups of long prefixes, one can further divide P_L into two or more partitions. In addition to the tag value, a partition ID is associated with the marker. When a partition is searched, the other partitions are disabled. Hence, each partition can support 64k subgroups of long prefixes, that is, a tag value only has local context within a partition.

Performance. Let the total number of prefixes be N and the fraction of prefixes with no more than 72 bits be S. Hence, the number of prefixes in group G_S is SN and the number of prefixes in G_L is $(1 - S)N$. In the basic scheme, a 144-bit word is used to store a prefix regardless of the actual length of the prefix. The total TCAM space used is $144N$ bits. For the folded approach, let the average size of a subgroup of long prefixes in G_L be β. The number of markers required is equal to $(1 - S)N/\beta$. The total TCAM space required is $72(SN + (1 - S)N/\beta) + 72(1 - S)N = 72N(1 + (1 - S)/\beta)$ bits. Let R_s be the ratio of the space used by the folded scheme over the space used by the basic scheme. We have $R_s = (1 + (1 - S)/\beta)/2$. Table 2.4 lists the values of R_s for various combinations of S and β. One can expect a space saving of more than 40 percent using the two-level routing table organization.

2.4.3 IPv6 Lookup via Variable-Stride Path and Bitmap Compression

By considering the discrepancies among different parts of the prefix trie, and taking advantage of the wide-word memory architecture, Zheng et al. [29] proposed a scalable IPv6 route lookup scheme by combining the techniques of path compression and Lulea bitmap compression. In addition, variable-stride was introduced to enhance the efficiency of compression.

The viability and effectiveness of the combination of path and bitmap compression are based on the facts that the prefix density of IPv6 is much smaller than that of IPv4. Hence, the

TABLE 2.4 Values of R_S for Different Combinations of S and β

	β				
S	1.0	1.5	2.0	2.5	3.0
0.6	0.7	0.63	0.6	0.58	0.57
0.7	0.65	0.6	0.575	0.56	0.55
0.8	0.6	0.57	0.55	0.54	0.53
0.9	0.55	0.53	0.525	0.52	0.52

IPv6 prefix trie should be more compressible (for both bitmap and path compression). The efficiency of introducing variable-stride lies in the fact that the bitmap compression ratios vary distinctly among different parts of the prefix trie. Thus, a fixed compression stride, such as cuttings in depth 16 and 24 in Lulea (see Fig. 2.14), does not provide efficiency in route table lookup, storage, and update costs due to the uneven distribution of prefixes.

Data Structure. A prefix trie is partitioned into sub-tries with variable heights, as shown in Figure 2.49. A subtrie is defined to be a full binary trie that is carved out from a prefix trie. It can be specified by a 2-tuple (*root*, *SubTrieHeight*), where *root* is the root node of a subtrie, and *SubTrieHeight* specifies the height/stride of this subtrie. The nodes on the bottom of the subtrie are called edge nodes and each subtrie should have $2^{SubTrieHeight}$ edge nodes.

Each trie node carries next hop IP address (NIP) or trie structure information. We use a data structure called pointer array (PA) to hold the route information carried by the edge node of a subtrie, with each pointer representing the memory location of either the NIP or the next level subtrie structure, as shown in Figure 2.50.

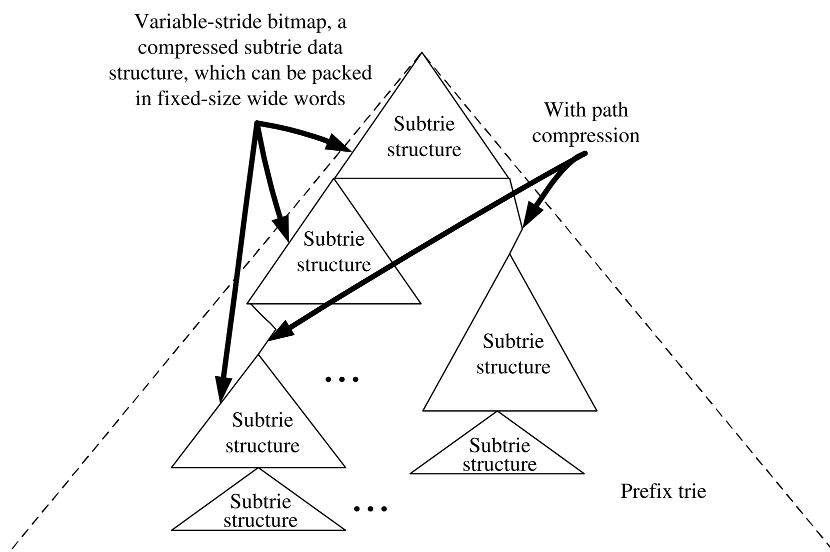


Figure 2.49 Demonstration of the subtrie hierarchy.

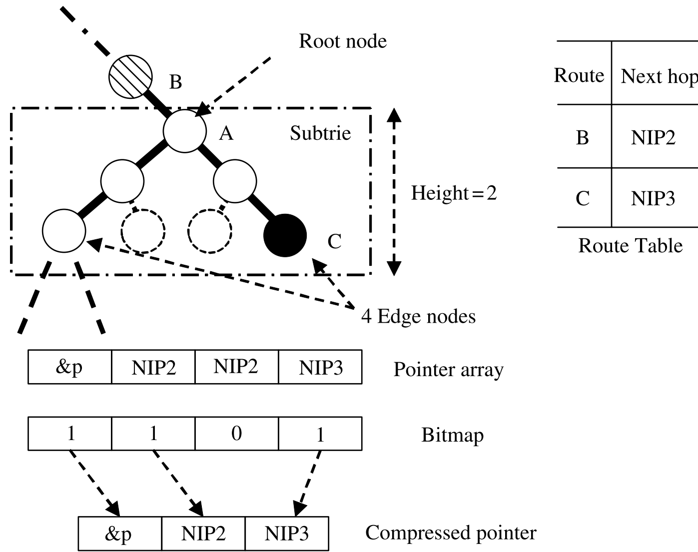


Figure 2.50 PA and corresponding compressed PA.

As depicted in Figure 2.50, Node A is the root of a two-layer subtrie, which consequently contains four edge nodes and they are associated with a PA of four pointers, each of which contains the memory location of either the NIP or the next subtrie structure. Node B carries the route information with NIP2, which descends to the root of the subtrie. Hence, two successive pointers (for the second- and third-edge nodes) within in the PA contain the same value (NIP2). In order to reduce the storage requirement, we can only keep such

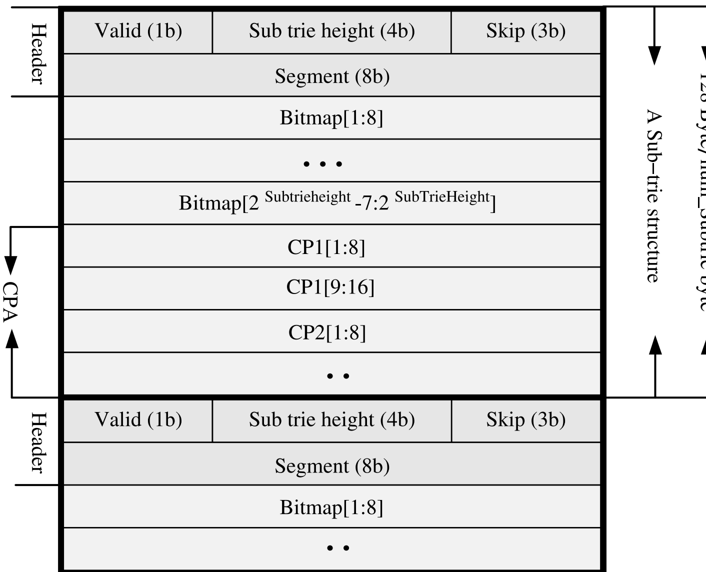
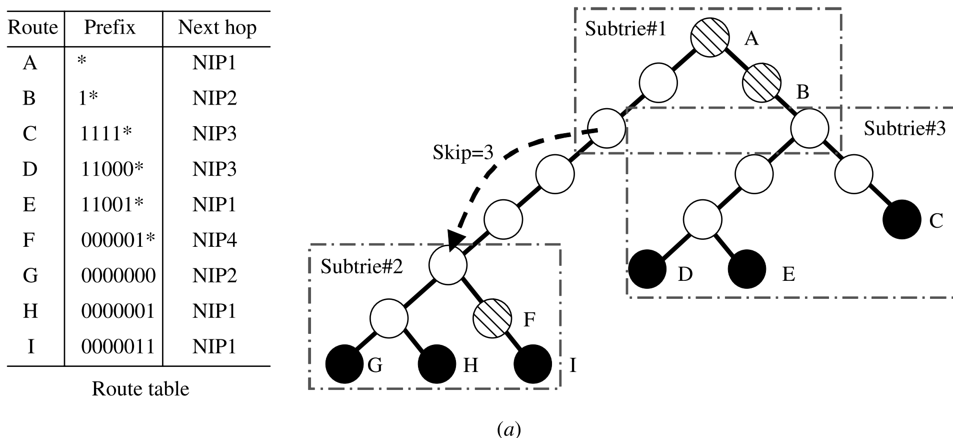


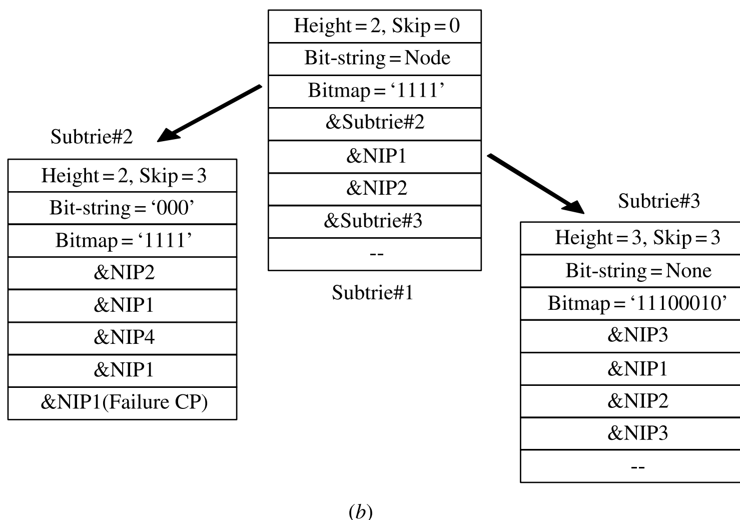
Figure 2.51 Data structure of a word frame.



Route	Prefix	Next hop
A	*	NIP1
B	1*	NIP2
C	1111*	NIP3
D	11000*	NIP3
E	11001*	NIP1
F	000001*	NIP4
G	0000000	NIP2
H	0000001	NIP1
I	0000011	NIP1

Route table

(*a*)



(*b*)

Figure 2.52 Example of variable-strike trie with path and bitmap compression trie. (*a*) Forwarding table and corresponding binary trie; (*b*) Corresponding data structure of example in (*a*).

information in a compressed pointer array (CPA) through introducing a bitmap to indicate which value can be omitted. As illustrated in the lower part of Figure 2.50, NIP2 appears only once in the CPA, but twice in the PA. Bitmap stores the compressed information. Such an idea/technique is called bitmap compression, which has been discussed in detail in Section 2.2.2.

The subtries are packed in fixed-size wide words with compressed information, if any. One wide word, called word frame, may contain one big subtrie when its compressing potential is high; otherwise, several subtries are enclosed in one wide word. An example of the word frame data structure is depicted in Figure 2.51.

Route Lookup. Figure 2.52*a* gives an example of a forwarding table and the corresponding binary trie. Through path-compression, some internal nodes are omitted. The heights

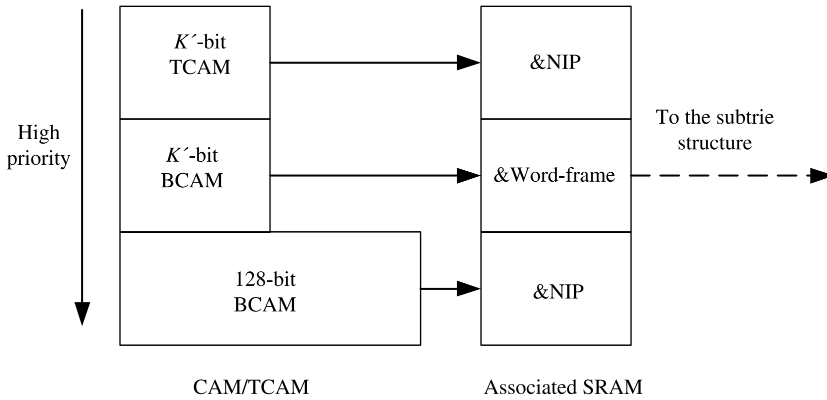


Figure 2.53 CAM organization for lookup.

of the three subtries are 2, 2 and 3, respectively. Figure 2.52*b* illustrates the corresponding data structure of the example. Note that the 'skip' value of subtrie #2 is 3 (>0), so the corresponding word frame contains a failure CP.

The lookup process starts with the destination IP address as the search key and a compressed pointer to the root of the prefix trie (as subtrie #1 in Fig. 2.52*b*). The lookup proceeds by using the key and the pointer to find the successive word frames and get the corresponding pointers iteratively. When the NIP containing the next IP address is encountered, the lookup process terminates.

A delicate CAM lookup mechanism is also introduced based on the following two observations: (1) the number of prefixes with the shortest length (say k_{min}) or with the length that is slightly longer than k_{min} (say k') are tiny; (2) the number of 128-bit prefixes is also very small. Therefore, (1) a k'-bit wide TCAM is used to store the prefixes no longer than k'-bit; (2) a k'-bit wide BCAM (binary CAM) is used for the prefixes between k'-bit and 64-bit; and (3) a 128-bit wide BCAM is used to store the 128-bit prefixes. The architecture is shown in Figure 2.53.

An incoming search key will be first sent to the mixed-CAM for a match. The matching result with the highest priority is returned, along with the pointer to a word frame containing either the associated next-hop IP address (for prefixes less than k' bits or exactly 128 bits) or the subtrie structure whose root node exactly matches the first k'th bits of the key.

Performance. The experimental results show that for an IPv6 forwarding table containing over 130k prefixes, generated by an IPv6 generator [29], the scheme can perform 22 million lookups per second even in the worst case with only 440 kbytes of SRAM and no more than 3 kbytes of TCAM. This means that it can support 10 Gbps route lookup for back-to-back 60-byte packets using on-chip memories.

2.5 COMPARISON

Table 2.5 summarizes several IP route lookup schemes with the asymptotic complexity for worst case lookup, storage, and update [1]. They are categorized into five classes, Class I–V, based on data structure and lookup mechanism. Class I includes the basic 1-bit trie and

TABLE 2.5 Complexity Comparison of Different Route Lookup Schemes

Case	Scheme	Worst-case Lookup	Storage	Update
I	1-bit trie	$O(W)$	$O(NW)$	$O(W)$
	PC-trie	$O(W)$	$O(N)$	$O(W)$
II	k-bit trie	$O(W/k)$	$O(2^k NW/k)$	$O(W/k + 2^k)$
	LC-trie	$O(W/k)$	$O(2^k NW/k)$	$O(W/k + 2^k)$
	Lulea	$O(W/k)$	$O(2^k NW/k)$	$O(W/k + 2^k)$
III	Binary search on prefix lengths	$O(\log_2 W)$	$O(N \log_2 W)$	$O(\log_2 W)$
IV	k-way range search	$O(\log_k N)$	$O(N)$	$O(N)$
V	TCAM	$O(1)$	$O(N)$	—

the path-compressed trie structures. The latter improves the average lookup performance and storage complexity. But its worst-case lookup performance is still $O(W)$, because there may be some paths from the root to the leaves that are W bits long. The path-compressed trie has the property of keeping the total number of nodes, both leaf nodes and internal nodes, below $2N$, resulting in the storage complexity of $O(N)$.

The schemes in Class II are based on the multi-bit trie, which is a very powerful and generalized structure. Besides the four schemes listed in Table 2.5, the DIR-24-8-BASIC and BC-16-16 also belong to this class. The worst case lookup is in $O(W/k)$ because a chunk of k bits of the destination address is inspected at a time, as opposed to one bit at a time in Class I. The Lulea algorithm has the same order of storage complexity as the others, but with a smaller constant factor, making it attractive in implementation.

The schemes in Classes I and II perform the lookup by linearly traversing the trie levels from top to bottom. The algorithm in Class III deploys a binary search over trie levels and performs hash operations at each level. Although the algorithm in Class III seems to have a small lookup complexity, it is based on the assumption of perfect hashing (no hash collision and thus one hash operation at each level). In reality, the time complexities of searches and updates over the hash tables are non-deterministic and can be quite large.

The algorithm in Class IV solves the IP lookup problem by treating each prefix as a range of the address space. The ranges split the entire address space into multiple intervals, each of which is associated with a unique prefix. Then a two-way (binary) or k-way search is deployed to determine which interval the prefix belongs to. Its lookup performance is $O(\log_k N)$ in the worst case.

TCAM is specialized hardware that completes the lookup in a constant time by simultaneously comparing the IP address with all prefixes in the table. We left '–' in the update complexity column due to its dependence on the data management schemes used for the TCAM.

Making a choice among these IP lookup schemes depends on the speed and storage requirements. For instance, in a low-end or a medium router, packet processing and forwarding are usually handled by general purpose processors. The trie-based schemes, such as LC-trie, Lulea algorithm, and binary search on trie levels are good candidates in these conditions. Typically, low-end or medium routers perform packet processing and forwarding with generic processors allowing for the algorithms to be easily implemented in software. However, for a core/backbone router, where the link speed is high and the forwarding table is large, the lookup time becomes more stringent. Assuming 40-byte packets are back-to-back

at a 10 Gbps (OC-192) link, the lookup time is only 32 ns. In this case, the hardware-based algorithms, such as DIR-24-8-BASIC, BC-16-16, and TCAM, become more feasible.

REFERENCES

[1] M. 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. 2001).

[2] V. Fuller, T. Li, J. Yu, and K. Varadhan, "Classless inter-domain routing (CIDR): an address assignment and aggregation strategy," RFC 1519 (Proposed Standard), Sept. 1993. [Online]. Available at: http://www.ietf.org/rfc/rfc1519.txt

[3] Y. Rekhter and T. Li, "An architecture for IP address allocation with CIDR," RFC 1518 (Proposed Standard), Sept. 1993. [Online]. Available at: http://www.ietf.org/rfc/rfc1518.txt

[4] P. Gupta, "Routing lookups and packet classifications: theory and practice," in *Proc. HOT Interconnects* 8, Stanford, California (Aug. 2000).

[5] D. Knuth, *Fundamental Algorithms Vol. 3: Sorting and Searching*. Addison-Wesley, Massachusetts, 1973.

[6] W. Eatherton, "Hardware-based internet protocol prefix lookups," M. S. Thesis, Washington University, St. Louis, Missouri (May 1999).

[7] V. Srinivasan and G. Varghese, "Faster IP lookups using controlled prefix expansion," in *Proc. ACM SIGMATICS*, Madison, Wisconsin, pp. 1–10 (June 1998).

[8] D. R. Morrison, "PATRICIA - Practical algorithm to retrieve information coded in alfanumeric," *IEEE/ACM Transactions on Networking*, vol. 17, no. 1, pp. 1093–1102 (Oct. 1968).

[9] S. Nilsson and G. Karlsson, "IP-Address lookup using LC-tries," *IEEE Journal on Selected Areas in Communications*, vol. 17, pp. 1083–1092 (June 1999).

[10] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," in *Proc. ACM SIGCOMM*, Cannes, France, pp. 3–14 (Sept. 1997).

[11] W. Eatherton, G. Varghese, and Z. Dittia, "Tree Bitmap: hardware/software IP lookups with incremental updates," *ACM SIGCOMM Computer Communications Review*, vol. 34, no. 2, pp. 97–122 (Apr. 2004).

[12] F. Baboescu, D. M. Tullsen, G. Rosu, and S. Singh, "A tree based router search engine architecture with single port memories," *ACM SIGARCH Computer Architecture News*, vol. 33, pp. 123–133 (May 2005).

[13] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high-speed IP routing lookups," in *Proc. ACM SIGCOMM*, Cannes, France, pp. 25–36 (Sept. 1997).

[14] B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search," in *Proc. IEEE INFOCOM'98* San Francisco, California, vol. 3, pp. 1248–1256 (Apr. 1998).

[15] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *Proc. IEEE INFOCOM'98*, San Francisco, California, vol. 3, pp. 1240–1247 (Apr. 1998).

[16] N. Huang, S. Zhao, J. Pan, and C. Su, "A fast IP routing lookup scheme for gigabit switching routers," in *Proc. IEEE INFOCOM'99*, New York, pp. 1429–1436 (Mar. 1999).

[17] *IDT75P52100 Network Search Engine*, IDT, June 2003. [Online]. Available at: http://www.idt.com

[18] *CYNSE10512 Network Search Engine*, CYPRESS, Nov. 2002. [Online]. Available at: http://www.cypress.com

[19] *Ultra9M – Datasheet from SiberCore Technologies*. [Online]. Available at: http://www.sibercore.com

[20] P. Gupta, “Algorithmic search solutions: features and benefits,” in *Proc. NPC-West 2003*, San Jose, California (Oct. 2003).

[21] H. Liu, “Routing table compaction in ternary CAM,” *IEEE Micro*, vol. 22, no. 1, pp. 58–64 (Jan. 2002).

[22] F. Zane, G. Narlikar, and A. Basu, “Cool CAMs: power-efficient TCAMs for forwarding engines,” in *Proc. IEEE INFOCOM’03*, San Francisco, California, pp. 42–52 (Apr. 2003).

[23] K. Zheng, C. C. Hu, H. B. Lu, and B. Liu, “An ultra high throughput and power efficient TCAM-based IP lookup engine,” in *Proc. IEEE INFOCOM’04*, Hong Kong, vol. 3, pp. 1984–1994 (Mar. 2004).

[24] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed., Morgan-Kaufmann, San Francisco, California, 1995.

[25] X. Zhang, B. Liu, W. Li, Y. Xi, D. Bermingham, and X. Wang, “IPv6-oriented 4 OC-768 packet classification with deriving-merging partition and field-variable encoding algorithm,” in *Proc. IEEE INFOCOM’06*, Barcelona, Spain (Apr. 2006).

[26] “Route-view v6 database.” [Online]. Available at: http://archive.routeviews.org/routeviews6/bgpdata/

[27] Netlogic Microsystems. [Online]. Available at: http://www.netlogicmicro.com

[28] D. Pao, “TCAM organization for IPv6 address lookup,” in *Proc. IEEE Int. Conf. on Advanced Communications Technology*, Phoenix Park, South Korea, vol. 1, pp. 26–31 (Feb. 2005).

[29] K. Zheng, Z. Liu, and B. Liu, “A scalable IPv6 route lookup scheme via dynamic variable-stride bitmap compression and path compression,” Computer Communication, vol. 29, no. 16, pp. 3037–3050 (Oct. 2006).