Survey and Taxonomy of IP Address Lookup Algorithms

Miguel Á. Ruiz-Sánchez, INRIA Sophia Antipolis, Universidad Autónoma Metropolitana
Ernst W. Biersack, Institut Eurécom Sophia Antipolis
Walid Dabbous, INRIA Sophia Antipolis

Abstract

Due to the rapid growth of traffic in the Internet, backbone links of several gigabits per second are commonly deployed. To handle gigabit-per-second traffic rates, the backbone routers must be able to forward millions of packets per second on each of their ports. Fast IP address lookup in the routers, which uses the packet's destination address to determine for each packet the next hop, is therefore crucial to achieve the packet forwarding rates required. IP address lookup is difficult because it requires a longest matching prefix search. In the last couple of years, various algorithms for high-performance IP address lookup have been proposed. We present a survey of state-of-the-art IP address lookup algorithms and compare their performance in terms of lookup speed, scalability, and update overhead.

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. More exactly, the forwarding decision consists of 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*. Once the forwarding information is retrieved, the router can transfer the packet from the incoming link to the appropriate outgoing link, in a process called *switching*.

The exponential growth of the Internet has stressed its routing system. While the data rates of links have kept pace with the increasing traffic, it has been difficult for the packet processing capacity of routers to keep up with these increased data rates. Specifically, the address lookup operation is a major bottleneck in the forwarding performance of today's routers. This article presents a survey of the latest algorithms for efficient IP address lookup. We start by tracing the evolution of the IP addressing architecture. The addressing architecture is of fundamental importance to the routing architecture, and reviewing it will help us to understand the address lookup problem.

The Classful Addressing Scheme

In IPv4, IP addresses are 32 bits long and, when broken up into 4 groups of 8 bits, are normally represented as four decimal numbers separated by dots. For example, the address 10000010_01010110_00010000_01000010 corresponds in dotted-decimal notation to 130.86.16.66.

One of the fundamental objectives of the Internet Protocol is to interconnect networks, so routing on a network basis was a natural choice (rather than routing on a host basis). Thus, the IP address scheme initially used a simple two-level hierarchy, with networks at the top level and hosts at the bottom level. This hierarchy is reflected in the fact that an IP address consists of two parts, a network part and a host part. The network part identifies the network to which a host is attached, and thus all hosts attached to the same network agree in the network part of their IP addresses.

Since the network part corresponds to the first bits of the IP address, it is called the *address prefix*. We will write prefixes as bit strings of up to 32 bits in IPv4 followed by a *. For example, the prefix 1000001001010110* represents all the 2^{16} addresses that begin with the bit pattern 1000001001010110. 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.

With a two-level hierarchy, IP routers forwarded packets based only on the network part, until packets reached the destination network. As a result, a forwarding table only needed to store a single entry to forward packets to all the hosts attached to the same network. This technique is called *address aggregation* and allows using prefixes to represent a group of addresses. Each entry in a forwarding table contains a prefix, as can be seen in Table 1. Thus, finding the forwarding infor-

Destination address prefix	Next-hop	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

Table 1. *A forwarding table.*

0890-8044/01/$10.00 © 2001 IEEE

mation requires searching for the prefix in the forwarding table that matches the corresponding bits of the destination address.

The addressing architecture specifies how the allocation of addresses is performed; that is, it defines how to partition the total IP address space of 2^{32} addresses — specifically, how many network addresses will be allowed and what size each of them should be. When Internet addressing was initially designed, a rather simple address allocation scheme was defined, which is known today as the *classful addressing scheme*. Basically, three different sizes of networks were defined in this scheme, identified by a class name: A, B, or C (Fig. 1). Network size was determined by the number of bits used to represent the network and host parts. Thus, networks of class A, B, or C consisted of an 8, 16, or 24-bit network part and a corresponding 24, 16, or 8-bit host part.

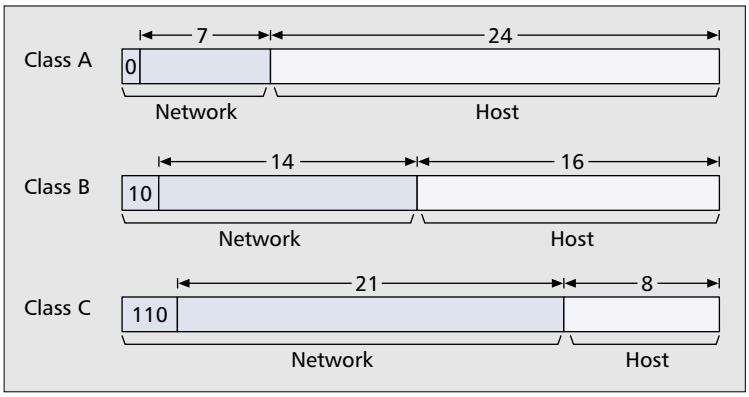


Figure 1. *Classful addresses.*

With this scheme there were very few class A networks, and their addressing space represented 50 percent of the total IPv4 address space (2^{31} addresses out of a total of 2^{32}). There were 16,384 (2^{14}) class B networks with a maximum of 65,534 hosts/network, and 2,097,152 (2^{21}) class C networks with up to 256 hosts. This allocation scheme worked well in the early days of the Internet. However, the continuous growth of the number of hosts and networks has made apparent two problems with the classful addressing architecture. First, with only three different network sizes from which to choose, the address space was not used efficiently and the IP address space was getting exhausted very rapidly, even though only a small fraction of the addresses allocated were actually in use. Second, although the state information stored in the forwarding tables did not grow in proportion to the number of hosts, it still grew in proportion to the number of networks. This was especially important in the backbone routers, which must maintain an entry in the forwarding table for every allocated network address. As a result, the forwarding tables in the backbone routers grew very rapidly. The growth of the forwarding tables resulted in higher lookup times and higher memory requirements in the routers, and threatened to impact their forwarding capacity.

The CIDR Addressing Scheme

To allow more efficient use of the IP address space and to slow down the growth of the backbone forwarding tables, a new scheme called *classless interdomain routing* (CIDR) was introduced.

Remember that in the classful address scheme, only three different prefix lengths are allowed: 8, 16, and 24, corresponding to classes A, B and C, respectively (Fig. 1). CIDR uses the IP address space more efficiently by allowing finer granularity in the prefix lengths. With CIDR, prefixes can be of arbitrary length rather than constraining them to be 8, 16, or 24 bits long.

To address the problem of forwarding table explosion, CIDR allows address aggregation at several levels. The idea is that the allocation of addresses has a topological significance. Then we can recursively aggregate addresses at various points within the hierarchy of the Internet's topology. As a result, backbone routers maintain forwarding information not at the network level, but at the level of arbitrary aggregates of networks. Thus, recursive address aggregation reduces the number of entries in the forwarding table of backbone routers.

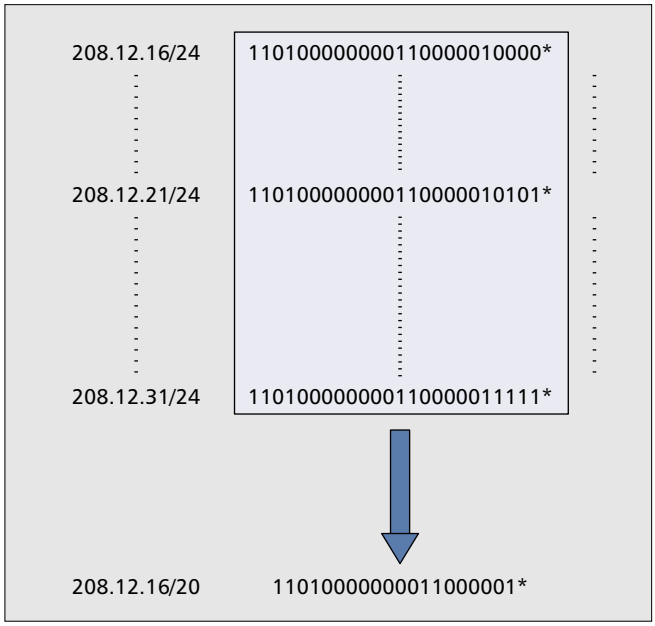


Figure 2. *Prefix aggregation.*

To understand how this works, consider the networks represented by the network numbers from 208.12.16/24 through 208.12.31/24 (Figs. 2 and 3). Suppose that in a router all these network addresses are reachable through the same service provider. From the binary representation we can see that the leftmost 20 bits of all the addresses in this range are the same (11010000 00001100 0001). Thus, we can aggregate these 16 networks into one "supernetwork" represented by the 20-bit prefix, which in decimal notation gives 208.12.16/20. Note that 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*).

While a great deal of aggregation can be achieved if addresses are care-

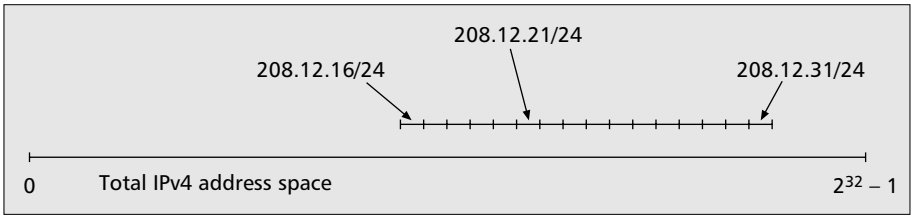
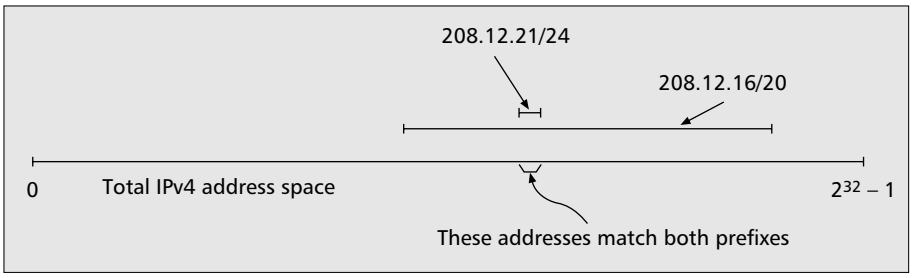
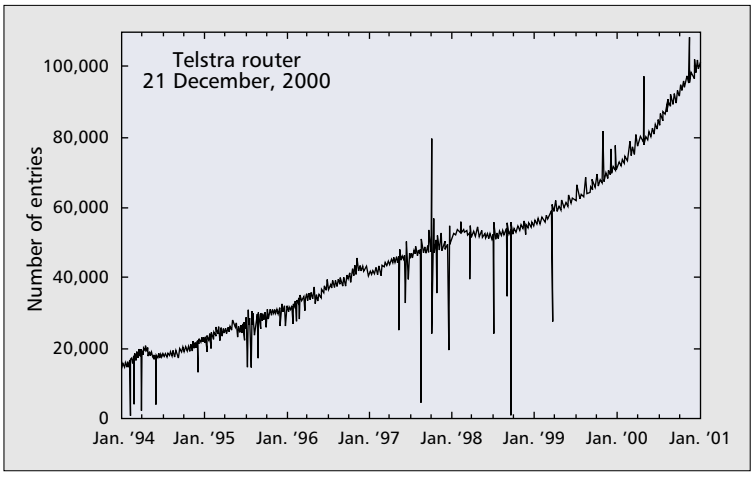


Figure 3. *Prefix ranges.*

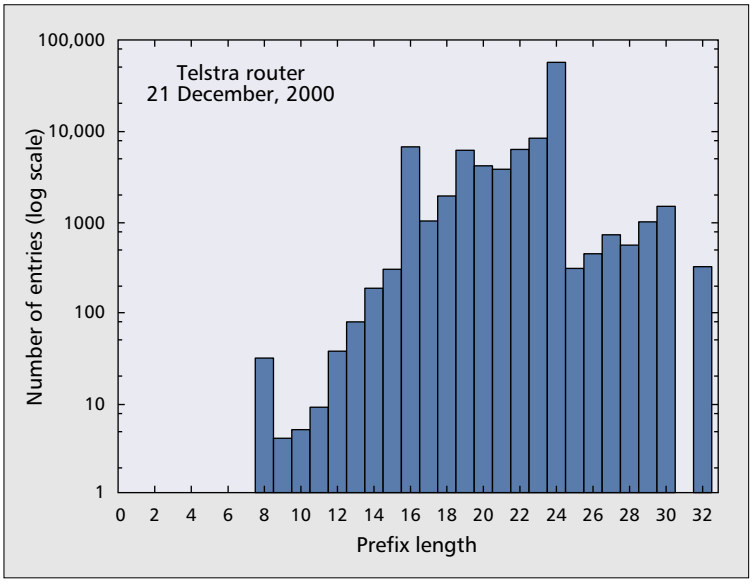


■ Figure 4. *An exception prefix.*

fully assigned, in some situations a few networks can interfere with the process of aggregation. For example, suppose now that a customer owning the network 208.12.21/24 changes its service provider and 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 (Fig. 3). We cannot perform aggregation as before, and instead of only one entry, 16 entries need to be stored in the forwarding table. One solution that can be used in this situation is aggregating in spite of the exception networks and additionally storing entries for the exception networks. In our example, this will result in only two entries in the forwarding table: 208.12.16/20 and 208.12.21/24 (Fig. 4 and Table 1). Note, however, that now some addresses will match both entries because prefixes overlap. In order to always make the correct forwarding decision, routers need to do more than to 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.



■ Figure 5. *Table growth of a typical backbone router.*



■ Figure 6. *Prefix length distribution of a typical backbone router.*

Difficulty of the Longest Matching Prefix Search

In the classful addressing architecture, the length of the prefixes was coded in the most significant bits of an IP address (Fig. 1), and the address lookup was a relatively simple operation: Prefixes in the forwarding table were organized in three separate tables, one for each of the three allowed lengths. The lookup operation amounted to finding an exact prefix match in the appropriate table. The search for an exact match could be performed using standard algorithms based on hashing or binary search.

While CIDR allows the size of the forwarding tables to be reduced, the address lookup problem now becomes more complex. With CIDR, the destination prefixes in the forwarding tables have arbitrary lengths and no longer correspond to the network part since they are the result of an arbitrary number of network aggregations. Therefore, when using CIDR, the search in a forwarding table can no longer be performed by exact matching because the length of the prefix cannot be derived from the address itself. As a result, determining the longest matching prefix involves not only comparing the bit pattern itself, but also finding the appropriate length. Therefore, we talk about searching in two dimensions: value and length. The search methods we will review try to reduce the search space at each step in both of these dimensions. In what follows we will use N to denote the number of prefixes in a forwarding table and W to indicate the maximum length of prefixes, which typically is also the length of the IP addresses.

Requirements on Address Lookup Algorithms

It is important to briefly review the characteristics of today's routing environment to derive adequate requirements and metrics for the address lookup algorithms we will survey.

As we have seen, using address prefixes is a simple method to represent groups of contiguous addresses. Address prefixes allow aggregation of forwarding information and hence support the growth of the Internet. Figure 5 shows the growth of a typical backbone router table. We can observe three phases of table growth: before the introduction of CIDR, growth was exponential (partly visible in early 1994). From mid-1994 to mid-1998, growth slowed down and was nearly linear. From mid-1998 to now growth is again exponential. Since the number of entries in router tables is still growing, it is important that search methods drastically reduce the search space at each step. Algorithms must be scalable with respect to the number of prefixes.

Another characteristic of the routing environment

is that a forwarding table needs to be updated dynamically to reflect route changes. In fact, instabilities in the backbone routing protocols can fairly frequently change the entries in a forwarding table. Labovitz [1] found that backbone routers may receive bursts of route changes at rates exceeding several hundred prefix updates per second. He also found that, on average, route changes occur 100 times/s. Thus, update operations must be performed in 10 ms or less.

The prefix length distribution in the forwarding tables can be used as a metric of the quality of the Internet hierarchy and address aggregation. Shorter prefixes represent a greater degree of aggregation. Thus, a decrease in average prefix length would indicate improved aggregation and hierarchy in the Internet. In Fig. 6 we can see that the historical class C with its 24-bit prefix length still dominates the number of entries in the forwarding table (note that the scale is logarithmic). A recent study shows that the number of exceptions in the address aggregation is growing. More precisely, Huston [2] found that currently 40 percent of the entries of a typical backbone forwarding table are prefix exceptions.

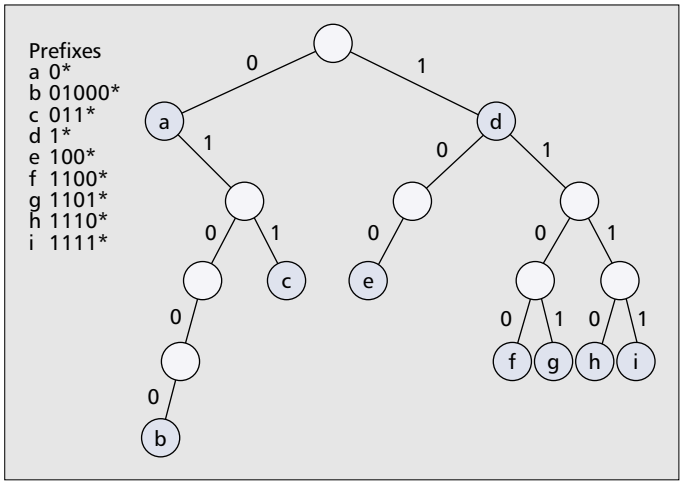


Figure 7. *A binary trie for a set of prefixes.*

The Classical Solution

A Binary Trie

A natural way to represent prefixes is using a trie. A trie is a tree-based data structure allowing the organization of prefixes on a digital basis by using the bits of prefixes to direct the branching. Figure 7 shows a binary trie (each node has at most two children) representing a set of prefixes of a forwarding table.

In a trie, a node on level l represents the set of all addresses that begin with the sequence of *l* bits consisting of the string of bits labeling the path from the root to that node. For example, node c in Fig. 7 is at level 3 and represents all addresses beginning with the sequence 011. The nodes that correspond to prefixes are shown in a darker shade; these nodes will contain the forwarding information or a pointer to it. Note also that prefixes are not only located at leaves but also at some internal nodes. This situation arises because of exceptions in the aggregation process. For example, in Fig. 7 the prefixes b and c represent exceptions to prefix a. Figure 8 illustrates this situation better. The trie shows the total address space, assuming 5-bit long addresses. Each leaf represents one possible address. We can see that the address spaces covered by prefixes b and c overlap with the address space covered by prefix a. Thus, prefixes b and c represent exceptions to prefix a and refer to specific subintervals of the address interval covered by prefix a. In the trie in Fig. 7, this is reflected by the fact that prefixes b and c are descendants of prefix a; in other words, prefix a is itself a prefix of b and c. As a result, some addresses will match several prefixes. For example, addresses beginning with 011 will match both prefixes c and a. Nevertheless, prefix c must be preferred because it is more specific (longest match rule).

Tries allow finding, in a straightforward way, the longest prefix that matches a given destination address. The search in a trie is guided by the bits of the destination address. At each node, the search proceeds to the left or right according to the sequential inspection of the address bits. While traversing the trie, every time we visit a node marked as prefix (i.e., a dark node) we remember this prefix as the longest match found so far. The search ends when there are no more branches to take, and the longest or best matching prefix will be the last prefix remembered. For instance, if we search the best matching prefix (BMP) for an address beginning with the bit pattern 10110 we start at the root in Fig. 7. Since the first bit of the address is 1 we move to the right, to the node marked as prefix d, and we remember d as the BMP found so far. Then we move to the left since the second address bit is 0; this time the node is not marked as a prefix, so d is still the BMP found so far. Next, the third address bit is 1, but at this point there is no branch labeled 1, so the search ends and the last remembered BMP (prefix d) is the longest matching prefix.

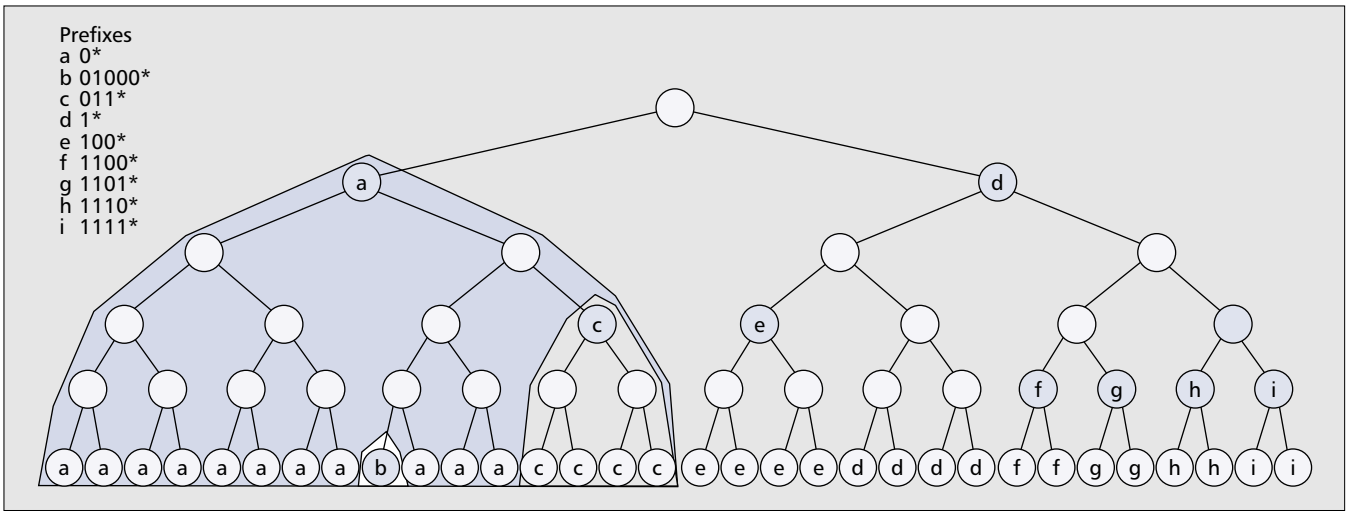
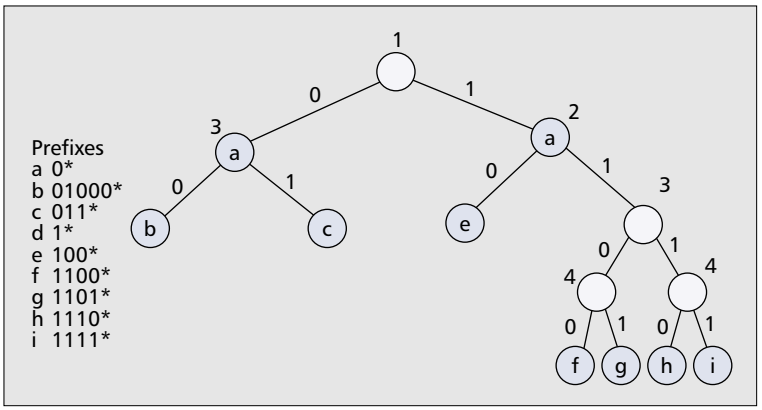


Figure 8. *An address space.*



■ Figure 9. *A path-compressed trie.*

In fact, what we are doing is a sequential prefix search by length, trying at each step to find a better match. We begin by looking in the set of length-1 prefixes, which are located at the first level in the trie, then in the set of length-2, located at the second level, and so on. Moreover, using a trie has the advantage that while stepping through the trie, the search space is reduced hierarchically. At each step, the set of potential prefixes is reduced, and the search ends when this set is reduced to one.

Update operations are also straightforward to implement in binary tries. Inserting a prefix begins by doing a search. When arriving at a node with no branch to take, we can insert the necessary nodes. Deleting a prefix starts again by a search, unmarking the node as prefix and, if necessary deleting unused node (i.e., leave nodes not marked as prefixes). Note finally that since the bit strings of prefixes are represented by the structure of the trie, the nodes marked as prefixes do not need to store the bit strings themselves.

Path-Compressed Tries

While binary tries allow the representation of arbitrary-length prefixes, they have the characteristic that long sequences of one-child nodes may exist (see prefix b in Fig. 7). Since these bits need to be inspected, even though no actual branching decision is made, search time can be longer than necessary in some cases. Also, one-child nodes consume additional memory. In an attempt to improve time and space performance, a technique called *path compression* can be used. Path compression consists of collapsing one-way branch nodes. When one-way branch nodes are removed from a trie, additional information must be kept in remaining nodes so that a search operation can be performed correctly.

There are many ways to exploit the path compression technique; perhaps the simplest to explain is illustrated in Fig. 9, corresponding to the binary trie in Fig. 7. Note that the two nodes preceding b now have been removed. Note also that since prefix a was located at a one-child node, it has been moved to the nearest descendant that is not a one-child node. Since in a path to be compressed several one-child nodes may contain prefixes, in general, a list of prefixes must be maintained in some of the nodes. Because one-way branch nodes are now removed, we can jump directly to the bit where a significant decision is to be made, bypassing the bit inspection of some bits. As a result, a bit number field must be kept now to indicate which bit is the next bit to inspect. In Fig. 9 these bit numbers are shown next to the nodes. Moreover, the bit strings of prefixes must be explicitly stored. A search in this kind of path-compressed trie is as follows. The algorithm performs, as usual, a descent in the trie under the guidance of the address bits, but this time only inspecting bit positions indicated by the bit-number field in the nodes traversed. When a node marked as a prefix is encountered, a comparison with the actual prefix value is performed. This is necessary since during the descent in the trie we may skip some bits. If a match is found, we proceed traversing the trie and keep the prefix as the BMP so far. The search ends when a leaf is encountered or a mismatch found. As usual, the BMP will be the last matching prefix encountered. For instance, if we look for the BMP of an address beginning with the bit pattern 010110 in the path-compressed trie shown in Fig. 9, we proceed as follows. We start at the root node and, since its bit number is 1, we inspect the first bit of the address. The first bit is 0, so we go to the left. Since the node is marked as a prefix, we compare prefix a with the corresponding part of the address (0). Since they match, we proceed and keep a as the BMP so far. Since the node's bit number is 3, we skip the second bit of the address and inspect the third one. This bit is 0, so we go to the left. Again, we check whether the prefix b matches the corresponding part of the address (01011). Since they do not match, the search stops, and the last remembered BMP (prefix a) is the correct BMP.

Path compression was first proposed in a scheme called PATRICIA [3], but this scheme does not support longest prefix matching. Sklower proposed a scheme with modifications for longest prefix matching in [4]. In fact, this variant was originally designed to support not only prefixes but also more general noncontiguous masks. Since this feature was really never used, current implementations differ somewhat from Sklower's original scheme. For example, the BSD version of the path-compressed trie (referred to as a *BSD trie*) is essentially the same as that just described. The basic difference is that in the BSD scheme, the trie is first traversed without checking the prefixes at internal nodes. Once at a leaf, the traversed path is backtracked in search of the longest matching prefix. At each node with a prefix or list of prefixes, a comparison is performed to check for a match. The search ends when a match is found. Comparison operations are not made on the downward path in the hope that not many exception prefixes exist. Note that with this scheme, in the worst case the path is completely traversed two times. In the case of Sklower's original scheme, the backtrack phase also needs to do recursive descents of the trie because noncontiguous masks are allowed.

Until recently, the longest matching prefix problem was addressed by using data structures based on path-compressed tries such as the BSD trie. Path compression makes a lot of sense when the binary trie is sparsely populated; but when the number of prefixes increases and the trie gets denser, using path compression has little benefit. Moreover, the principal disadvantage of path-compressed tries, as well as binary tries in general, is that a search needs to do many memory accesses, in the worst case 32 for IPv4 addresses. For example, for a typical backbone router [5] with 47,113 prefixes, the BSD version for a path-compressed trie creates 93,304 nodes. The maximal height is 26, while the average height is almost 20. For the same prefixes, a simple binary trie (with one-child nodes) has a maximal height of 30 and an average height of almost 22. As we can see, the heights of both tries are very similar, and the BSD trie may perform additional comparison operations when backtracking is needed.

New IP Lookup Algorithms

We have seen that the difficulty with the longest prefix matching operation is its dual dimensions: length and value. The new schemes for fast IP lookups differ in the dimension to search

and whether this search is linear or binary. In the following, we present a classification of the different IP lookup schemes.

A Taxonomy of IP Lookup Algorithms

Search on values approaches: Sequential search on values is the simplest method to find the BMP. The data structure needed is just an array with unordered prefixes. The search algorithm is very simple. It goes through all the entries comparing the prefix with the corresponding bits of a given address. When a match is found, we keep the longest match so far and continue. At the end, the last prefix remembered is the BMP. The problem with this approach is that the search space is reduced only by one prefix at each step. Clearly, the search complexity in time for this scheme is a function of the number of prefixes $O(N)$, and hence the scheme is not scalable. With the search on value approach, we get rid of the length dimension because of the exhaustive search. It is clear that a binary search on values would be better, and we will see in a later section how this can be done.

Search on length approaches: Another possibility is to base the search on the length dimension and use linear or binary search. Two possible ways to organize the prefixes for search on length exist. In fact, we have already seen linear search on length, which is performed on a trie. Tries allow to check the prefixes of length i at step i. Moreover, prefixes in a trie are organized in such a way that stepping through the trie reduces the set of possible prefixes. As we will see in a later section, one optimization of this scheme is using multibit tries. Multibit tries still do linear search on length, but inspect several bits simultaneously at each step.

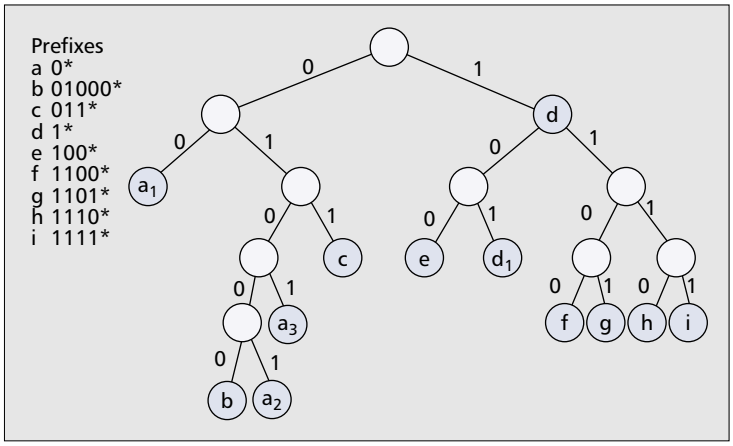
The other possible way to organize the prefixes to allow a search on length is to use a different table for each possible length. Then linear search on length can be made by doing at each step a search on a particular table using hashing, for instance. We will see how Waldvogel *et al.* [6] use hash tables to do binary search on length.

In addition to the algorithm-data structure aspect, various approaches use different techniques, such as transformation of the prefix set, compression of redundant information to reduce the memory requirements, application of optimization techniques, and exploitation of the memory hierarchy in computers. We introduce each of these aspects briefly in the following subsection and then discuss the new lookup schemes in detail according to the algorithm-data structure aspect in the next sections.

Auxiliary Techniques

Prefix Transformation — Forwarding information is specified with prefixes that represent ranges of addresses. Although the set of prefixes used is usually determined by the information gathered by the routing protocols, the same forwarding information can be expressed with different sets of prefixes. Various transformations are possible according to special needs, but one of the most common prefix transformation techniques is *prefix expansion*. Expanding a prefix means transforming one prefix into several longer and more specific prefixes that cover the same range of addresses. As an example, the range of addresses covered by prefix 1* can also be specified with the two prefixes 10*, 11*; or with the four prefixes 100*, 101*, 110*, 111*. If we do prefix expansion appropriately, we can get a set of prefixes that has fewer different lengths, which can be used to make a faster search, as we will show later.

We have seen that prefixes can overlap (Fig. 4). In a trie, when two prefixes overlap, one of them is itself a prefix of the other (Figs. 7 and 8). Since prefixes represent intervals of contiguous addresses, when two prefixes overlap it means that one interval of addresses contains another interval of addresses (Figs. 4 and 8). In fact, that is why an address can be matched to several prefixes. If several prefixes match, the longest prefix match rule is used in order to find the most specific forwarding information. 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. For example, Fig. 10 shows the disjoint-prefix binary trie that corresponds to the trie in Fig. 7. Prefixes a_1, a_2, and a_3 have inherited the forwarding information of the original prefix a, which now has been suppressed. Prefix d_1 has been obtained in a similar way. Since prefixes at internal nodes are expanded or pushed down to the leaves of the trie, this technique is called *leaf pushing* by Srinivasan *et al.* [7]. Figure 11 shows the disjoint intervals of addresses that correspond to the disjoint-prefix binary trie of Fig. 10.

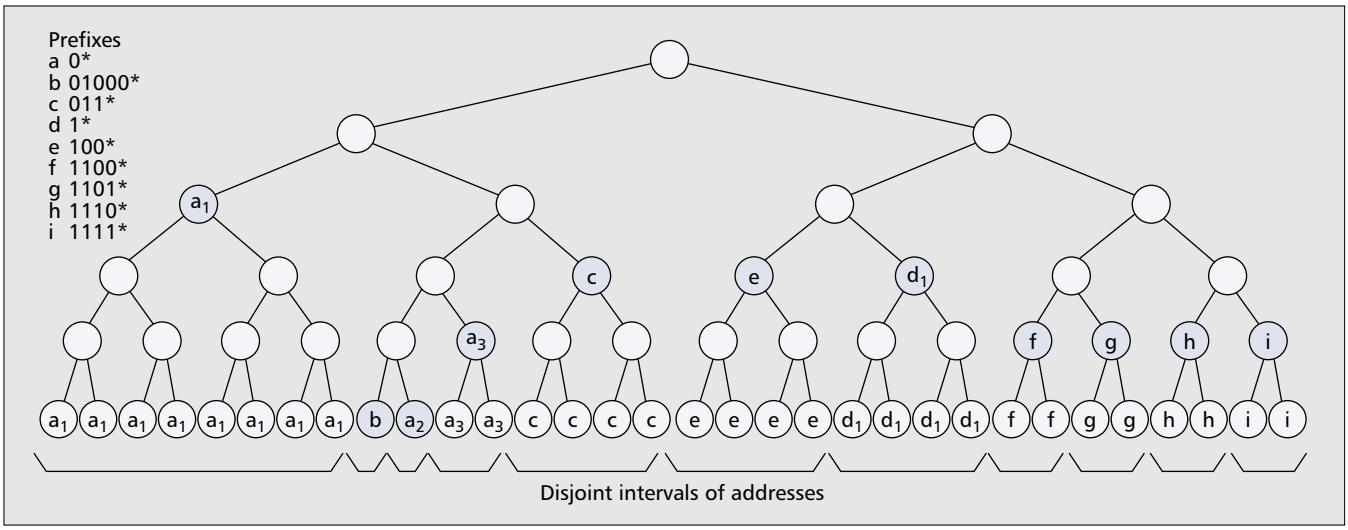


■ Figure 10. *A disjoint-prefix binary trie.*

Compression Techniques — Data compression attempts to remove redundancy from the encoding. The idea to use compression comes from the fact that expanding the prefixes increases information redundancy. Compression should be done such that memory consumption is decreased, and retrieving the information from the compressed structure can be done easily and with a minimum number of memory accesses. Run-length encoding is a very simple compression technique that replaces consecutive occurrences of a given symbol with only one occurrence plus a count of how many times the symbol occurs. This technique is well adapted to our problem because prefixes represent intervals of contiguous addresses that have the same forwarding information.

Application of Optimization Techniques — There are more than one way to transform the set of prefixes. Optimization allows us to define some constraints and find the right set of prefixes to satisfy those constraints. Normally we want to minimize the amount of memory consumed.

Memory Hierarchy in Computers — One of the characteristics of today's computers is the difference in speed between processor and memory, and also between memories of different hierarchies (cache, RAM, disk). Retrieving information from memory is expensive, so small data structures are desirable



■ Figure 11. *An expanded disjoint-prefix binary trie.*

because they make it more likely for the forwarding table to fit into the faster cache memory. Furthermore, the number of memory accesses must be minimized to make searching faster.

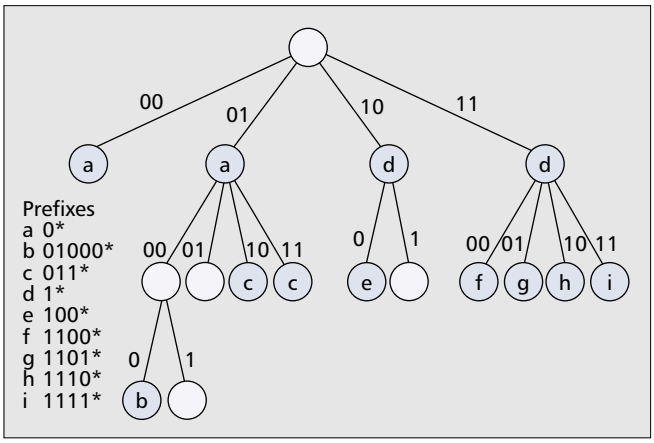
New algorithms to the longest prefix matching problem use one or several of the aspects just outlined. We will survey the different algorithms by classifying them according to the algorithm-data structure aspect, and discuss other aspects as well. It is worth mentioning that organizing the prefixes in different ways allows for different trade-offs between the search and update costs, as well as memory consumption. We discuss these trade-offs when we explain the different schemes. We now present in detail some of the most efficient algorithms for IP address lookup.

Search on Prefix Lengths Using Multibit Tries

The Basic Scheme

Binary tries provide an easy way to handle arbitrary length prefixes. Lookup and update operations are straightforward. Nevertheless, the search in a binary trie can be rather slow because we inspect one bit at a time and in the worst case 32 memory accesses are needed for an IPv4 address.

One way to speedup the search operation is to inspect not just one bit a time but *several bits simultaneously*. For instance, if we inspect 4 bits at a time we would need only 8 memory accesses in the worst case for an IPv4 address. The number of bits to be inspected per step is called *stride* and can be constant or variable. A trie structure that allows the inspection of bits in strides of several bits is called a *multibit* trie. Thus, a multibit trie is a trie where each node has 2^k children, where k is the stride.



■ Figure 12. *A variable-stride multibit trie.*

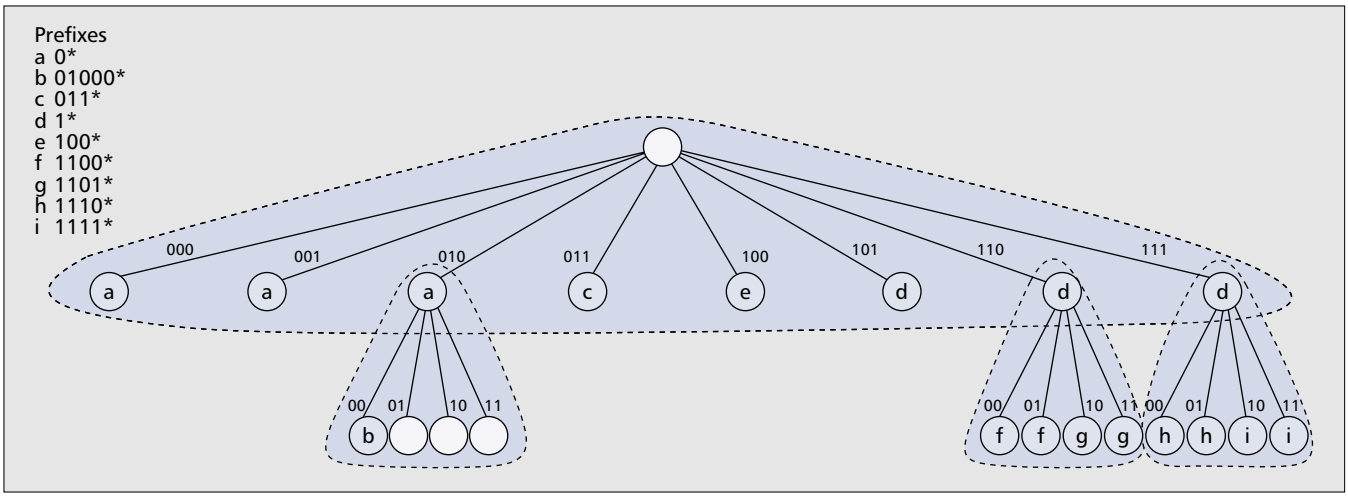
Since multibit tries allow the data structure to be traversed in strides of several bits at a time, they cannot support arbitrary prefix lengths. To use a given multibit trie, the prefix set must be transformed into an equivalent set with the prefix lengths allowed by the new structure. For instance, a multibit trie corresponding to our example from Fig. 7 is shown in Fig. 12. We see that a first stride of 2 bits is used, so prefixes of length 1 are not allowed, and we need to expand prefixes a and d to produce four equivalent prefixes of length 2. In the same figure it is shown how prefix c has been expanded to length 4. Note that the height of the trie has decreased, and so has the number of memory accesses when doing a search. Figure 13 shows a different multibit trie for our example. We can see again that prefixes a and d have been expanded, but now to length 3. However, two of the prefixes produced by expansion already exist (prefixes c and e). We must preserve the forwarding information of prefixes c and e since their forwarding information is more specific than that of the expanded prefix. Thus, expansion of prefixes a and d finally results in six prefixes, not eight. In general, when an expanded prefix collides with an existing longer prefix, forwarding information of the existing prefix must be preserved to respect the longest matching rule.

Searching in a multibit trie is essentially the same as in a binary trie. To find the BMP of a given address consists of successively looking for longer prefixes that match. The multibit trie is traversed, and each time a prefix is found at a node, it is remembered as the new BMP seen so far. At the end, the last BMP found is the correct BMP for the given address. Multibit tries still do linear search on lengths as do binary tries, but the search is faster because the trie is traversed using larger strides.

In a multibit trie, if all nodes at the same level have the same stride size, we say that it is a *fixed* stride; otherwise, it is a *variable* stride. We can choose multibit tries with fixed or variable strides. Fixed strides are simpler to implement than variable strides, but in general waste more memory. Figure 13 is an example of a fixed-stride multibit trie, Fig. 12 a variable-stride multibit trie.

Choice of Strides

Choosing the strides requires a trade-off between search speed and memory consumption. In the extreme case, we could make a trie with a single level (i.e., a one-level trie with a 32-bit stride for IPv4). Search would take in this case just one access, but we would need a huge amount of memory to store 2^{32} entries.



■ Figure 13. *A fixed-stride multibit trie.*

One natural way to choose strides and control memory consumption is to let the structure of the binary trie determine this choice. For example, if we look at Fig. 7, we observe that the subtrie with its root the right child of node d is a full subtrie of two levels (a full binary subtrie is a subtrie where each level has the maximum number of nodes). We can replace this full binary subtrie with a one-level multibit subtrie. The stride of the multibit subtrie is simply the number of levels of the substituted full binary subtrie, two in our example. In fact, this transformation was already made in Fig. 12. This transformation is straightforward, but since it is the only transformation we can do in Fig. 7, it has a limited benefit. We will see later how to replace, in a controlled way, binary subtries that are unnecessary full subtries. The height of the multibit trie will be reduced while controlling memory consumption. We will also see how optimization techniques can be used to choose the strides.

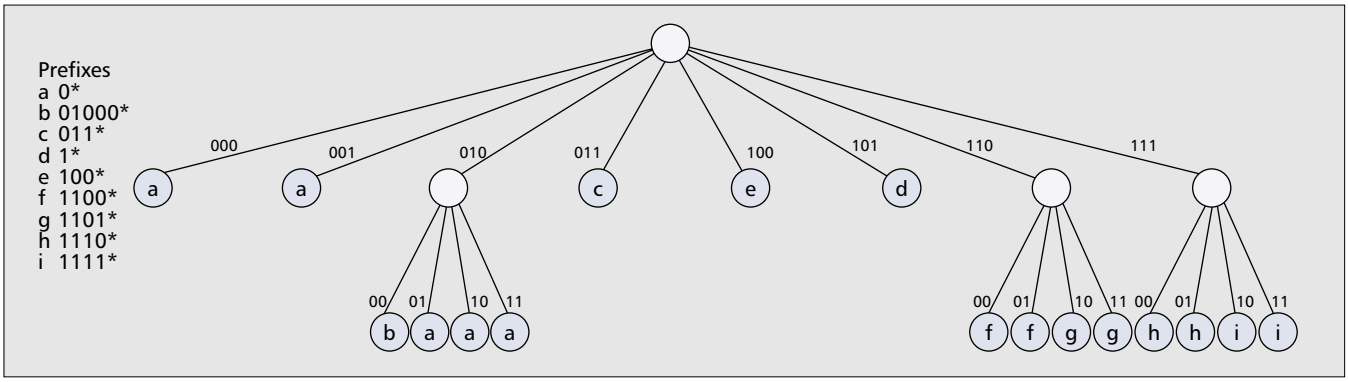
Updating Multibit Tries

Size of strides also determines update time bounds. A multibit trie can be viewed as a tree of one-level subtries. For instance, in Fig. 13 we have one subtrie at the first level and three subtries at the second level. When we do prefix expansion in a subtrie, what we actually do is compute for each node of the subtrie its *local BMP*. The BMP is local because it is computed from a subset of the total of prefixes. For instance, in the subtrie at the first level we are only concerned with finding for each node the BMP among prefixes a, c, d, e. In the leftmost subtrie at the second level the BMP for each node will be selected from only prefix b. In the second subtrie at the second level, the BMP is selected for each node among prefixes f and g, and the rightmost subtrie is concerned only with prefixes h and i. Some nodes may be empty, indicating that there are no BMPs for these nodes among the prefixes corresponding to this subtrie. As a result, multibit tries divide the problem of finding the BMP into small problems in which local BMPs are selected among a subset of prefixes. Hence, when looking for the BMP of a given address, we traverse the tree and remember the last local BMP as we go through it.

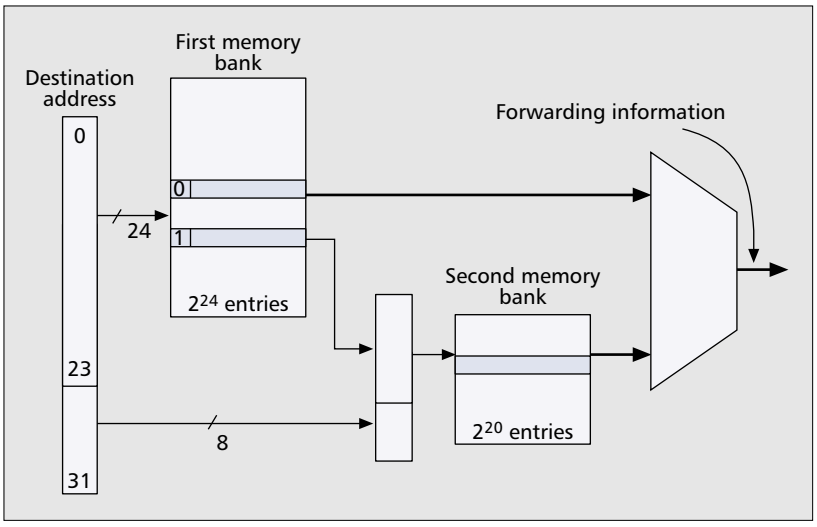
It is worth noting that the BMPs computed at each subtrie are independent of the BMPs computed at other subtries. The advantage of this scheme is that inserting or deleting a prefix only needs to update one of the subtries. Prefix update is completely local. In particular, if the prefix is or will be stored in a subtrie with a stride of k bits, the update needs to modify at most 2^{k-1} nodes (a prefix populates at most the half of the nodes in a subtrie). Thus, choosing appropriate stride values allows the update time to be bounded.

Local BMPs allow incremental updates, but require that internal nodes, besides leaves, store prefixes; thus, memory consumption is incremented. As we know, we can avoid prefixes at internal nodes if we use a set of disjoint prefixes. We can obtain a multibit trie with disjoint prefixes if we expand prefixes at internal nodes of the multibit trie down to its leaves (leaf pushing). Figure 14 shows the result of this process when applied to the multibit trie in Fig. 13. Nevertheless, note that now, in the general case, a prefix can be theoretically expanded to several subtries at all levels. Clearly, with this approach, the BMPs computed at each subtrie are no longer local; thus, updates will suffer longer worst case times.

As we can see, a multibit trie with several levels allows, by varying stride k, an interesting trade-off between search time,



■ Figure 14. *A disjoint-prefix multibit trie.*



■ Figure 15. *The hardware scheme of [8].*

memory consumption, and update time. The length of the path can be controlled to reduce search time. Choosing larger strides will make faster searches, but more memory will be needed, and updates will require more entries to be modified because of expansion.

As we have seen, incremental updates are possible with multibit tries if we do not use leaf pushing. However, inserting and deleting operations are slightly more complicated than with binary tries because of prefix transformation. Inserting one prefix means finding the appropriate subtrie, doing an expansion, and inserting each of the resulting prefixes. Deleting is still more complicated because it means deleting the expanded prefixes and, more important, updating the entries with the next BMP. The problem is that original prefixes are not actually stored in the trie. To see this better, suppose we insert prefixes 101*, 110* and 111* in the multibit trie in Fig. 13. Clearly, prefix d will disappear; and if later we delete prefix 101*, for instance, there will be no way to find the new BMP (d) for node 101. Thus, update operations need an additional structure for managing original prefixes.

Multibit Tries in Hardware

The basic scheme of Gupta *et al.* [8] uses a two-level multibit trie with fixed strides similar to the one in Fig. 14. However, the first level corresponds to a stride of 24 bits and the second level to a stride of 8 bits. One key observation in this scheme is that in a typical backbone router, most of the entries have prefixes of length 24 bits or less (Fig. 6, with logarithmic scale on the y axis). As a result, using a first stride of 24 bits allows the BMP to be found in one memory access for the majority of cases. Also, since few prefixes have a length longer than 24, there will be only a small number of subtries at the second level. In order to save memory, internal nodes are not allowed to store prefixes. Hence, should a prefix correspond to an internal node, it will be expanded to the second level (leaf pushing). This process results in a multibit trie with disjoint expanded prefixes similar to the one in Fig. 14 for the example in Fig. 13. The first level of the multibit trie has 2^{24} nodes and is implemented as a table with the same number of entries. An entry in the first level contains either the forwarding information or a pointer to the corresponding subtrie at the second level. Entries in the first table need 2 bytes to store a pointer; hence, a memory bank of 32 Mbytes is used to store 2^{24} entries. Actually, the pointers use 15 bits because the first bit of an entry indicates if the information stored is the forwarding information or a pointer to a second-level subtrie. The number of subtries at the second level depends on the number of prefixes longer than 24 bits. In the worst case each of these prefixes will need a different subtrie at the second level. Since the stride for the second level is 8 bits, a subtrie at the second level has $2^8 = 256$ leaves. The second-level subtries are stored in a second memory bank. The size of this second memory bank depends on the expected worst case prefix length distribution. In the MaeEast table [5] we examined on August 16, 1999, only 96 prefixes were longer than 24 bits. For example, for a memory bank of 2^{20} entries of 1 byte each (i.e., a memory bank of 1 Mbyte), the design supports a maximum of $2^{12} = 4096$ subtries at the second level.

In Fig. 15 we can see how the decoding of a destination address is done to find the corresponding forwarding information. The first 24 bits of the destination address are used to index into the first memory bank (the first level of the multibit trie). If the first bit of the entry is 0, the entry contains the forwarding information; otherwise, the forwarding information must be looked up in the second memory bank (the second level of the multibit trie). In that case, we concatenate the last 8 bits of the destination address with the pointer just found in the first table. The result is used as an index to look up the forwarding information in the second memory bank.

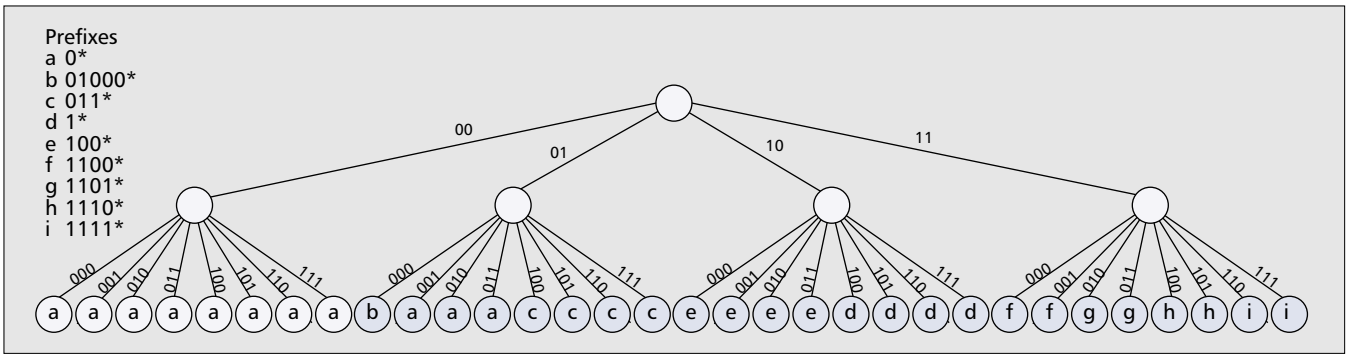
The advantage of this simple scheme is that the lookup requires a maximum of two memory accesses. Moreover, since it is a hardware approach, the memory accesses can be pipelined or parallelized. As a result the lookup operation takes practically one memory access time. Nevertheless, since the first stride is 24 bits and leaf pushing is used, updates may take a long time in some cases.

Multibit Tries with the Path Compression Technique

Nilsson *et al.* [9] recursively transform a binary trie with prefixes into a multibit trie. Starting at the root, they replace the largest full binary subtrie with a corresponding one-level multibit subtrie. This process is repeated recursively with the children of the multibit subtrie obtained. Additionally, one-child paths are compressed. Since we replace at each step a binary subtrie of several levels with a multibit trie of one level, the process can be viewed as a compression of the levels of the original binary trie. *Level-compressed* (LC) is the name given by Nilsson to these multibit tries. Nevertheless, letting the structure of the binary trie strictly determine the choice of strides does not allow control of the height of the resulting multibit trie. One way to further reduce the height of the multibit trie is to let the structure of the trie only guide, not determine, the choice of strides. In other words, we will replace nearly full binary subtries with a multibit subtrie (i.e., binary subtries where only few nodes are missing).

Nilsson proposes replacing a nearly full binary subtrie with a multibit subtrie of stride k if the nearly full binary subtrie has a sufficient fraction of the 2^k nodes at level k, where a sufficient fraction of nodes is defined using a single parameter called *fill factor* x, with $0 < x \leq 1$. For instance, in Fig. 7, if the fill factor is 0.5, the fraction of nodes at the fourth level is not enough to choose a stride of 4, since only 5 of the 16 possible nodes are present. Instead, there are enough nodes at the third level (5 of the 8 possible nodes) for a multibit subtrie of stride 3.

In order to save memory space, all the nodes of the LC trie are stored in a single array: first the root, then all the nodes at the second level, then all the nodes at the third level, and so



■ Figure 16. *A two-level fully expanded multibit trie.*

on. Moreover, internal nodes are not allowed to store prefixes. Instead, each leaf has a linear list with prefixes, in case the path to the leaf should have one or several prefixes (less specific prefixes). As a result, a search in an LC trie proceeds as follows. The LC trie is traversed as is the basic multibit trie. Nevertheless, since path compression is used, an explicit comparison must be performed when arriving at a leaf. In case of mismatch, a search of the list of prefixes must be performed (less specific prefixes, i.e., prefixes in internal nodes in the original binary trie).

Since the LC trie is implemented using a single array of consecutive memory locations and a list of prefixes must be maintained at leaves, incremental updates are very difficult.

Multibit Tries and Optimization Techniques

One easy way to bound worst-case search times is to define fixed strides that yield a well-defined height for the multibit trie. The problem is that, in general, memory consumption will be large, as seen earlier.

On the other hand, we can minimize the memory consumption by letting the prefix distribution strictly determine the choice of strides. Unfortunately, the height of the resulting multibit trie cannot be controlled and depends exclusively on the specific prefix distribution. We saw in the last section that Nilsson uses the fill factor as a parameter to control the influence of the prefix distribution on stride choice, and so influences somewhat the height of the resulting multibit trie. Since prefix distribution still guides stride choice, memory consumption is still controlled. Nevertheless, the use of the fill factor is simply a reasonable heuristic and, more important, does not allow a guarantee on worst-case height.

Srinivasan *et al.* [7] use dynamic programming to determine, for a given prefix distribution, the optimal strides that minimize memory consumption and guarantee a worst-case number of memory accesses. The authors give a method to find the optimal strides for the two types of multibit tries: fixed stride and variable stride.

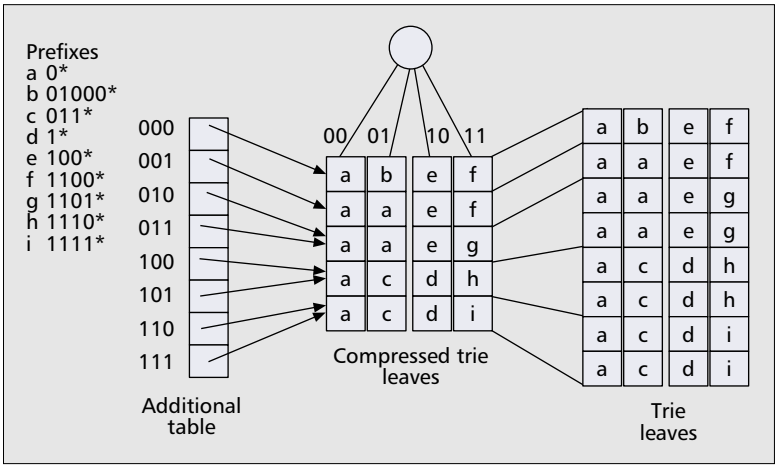
Another way to minimize lookup time is to take into account, on one hand, the hierarchical structure of the memory in a system and, on the other, the probability distribution of the usage of prefixes (which is traffic-dependent). Cheung *et al.* [10] give methods to minimize the average lookup time per prefix for this case. They suppose a system having three types of hierarchical memories with different access times and sizes.

Using optimization techniques makes sense if the entries of the forwarding table do not change at all or change very little, but this is rarely the case for backbone routers. Inserting and deleting prefixes degrades the improvement due to optimization, and rebuilding the structure may be necessary.

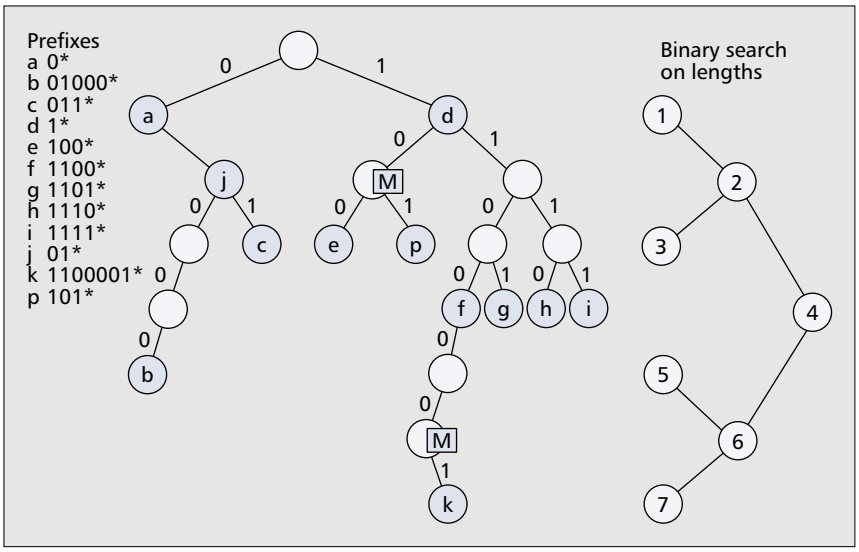
Multibit Tries and Compression

Expansion creates several prefixes that all inherit the forwarding information of the original prefix. Thus, if we use multibit tries with large strides, we will have a great number of contiguous nodes with the same BMP. We can use this fact and compress the redundant information, which will allow saving memory and make the search operation faster because of the small height of the trie.

One example of this approach is the full expansion/compression scheme proposed by Crescenzi *et al.* [11]. We will illustrate their method with a small example where we do a maximal expansion supposing 5-bit addresses and use a two-level multibit trie. The first level uses a stride of 2 bits, the second level a stride of 3 bits, as shown in Fig. 16. The idea is to compress each of the subtries at the second level. In Fig. 17 we can see how the leaves of each second-level subtrie have been placed vertically. Each column corresponds to one of the second-level subtries. The goal is to compress the repeated occurrences of the BMPs. Nevertheless, the compression is done in such a way that at each step the number of compressed symbols is the same for each column. With this strategy the compression is not optimal for all columns, but since the compression is made in a synchronized way for all the columns, accessing any of the compressed subtries can be made with one common additional table of pointers, as shown in Fig. 17. To find the BMP of a given address we traverse the first level of the multibit trie as usual; that is, the first 2 bits of the address are used to choose the correct subtrie at the second level. Then the last 3 bits of the address are used to find the pointer in the additional table. With this pointer we can readily find the BMP in the compressed subtrie. For example, searching for the address 10110 will guide us to the third subtrie (column) in the compressed structure; and using the



■ Figure 17. *A full expansion parallel compression scheme.*



■ Figure 18. *Binary search on prefix lengths.*

pointer contained in the entry 110 of the additional table, we will find d as the best matching prefix.

In the actual scheme proposed by Crescenzi, prefixes are expanded to 32 bits. A multibit trie of two levels is also used, but the stride of the first and second levels is 16 bits. It is worth noting that even though compression is done, the resulting structure is not small enough to fit in the cache memory. Nevertheless, because of the way to access the information, search always takes only three memory accesses. The reported memory size for a typical backbone forwarding table is 1.2 Mbytes.

Another scheme that combines multibit tries with the compression idea has been dubbed the *Lulea algorithm* [12]. In this scheme, a multibit trie with fixed stride lengths is used. The strides are 16,8,8, for the first, second, and third level respectively, which gives a trie of height 3. In order to do efficient compression, the Lulea scheme must use a set of disjoint prefixes; hence, the Lulea scheme first transforms the set of prefixes into a disjoint-prefix set. Then the prefixes are expanded in order to meet the stride constraints of the multibit trie. Additionally, in order to save memory, prefixes are not allowed at internal nodes of the multibit trie ; thus, leaf pushing is used.

Again, the idea is to compress the prefix information in the subtries by suppressing repeated occurrences of consecutive BMPs. Nevertheless, contrary to the last scheme, each subtrie is compressed independent of the others. Once a subtrie is compressed, a clever decoding mechanism allows the access to the BMPs. Due to lack of space we do not give the details of the decoding mechanism.

While the trie height in the Lulea scheme is 3, actually more than three memory references are needed because of the decoding required to access the compressed data structure. Searching at each level of the multibit trie needs, in general, four memory references. This means that in the worst case 12 memory references are needed for IPv4. The advantage of the Lulea scheme, however, is that these references are almost always to the cache memory because the whole data structure is very small. For instance, for a forwarding table containing 32,732 prefixes the reported size of the data structure is 160 kbytes.

Schemes using multibit tries and compression give very fast search times. However, compression and the leaf pushing technique used do not allow incremental updates. Rebuilding the whole structure is the only solution.

A different scheme using compression is the Full Tree Bit Map by Eatherton [13]. Leaf pushing is avoided, so incremental updates are allowed.

Binary Search on Prefix Lengths

The problem with arbitrary prefix lengths is that we do not know how many bits of the destination address should be taken into account when compared with the prefix values. Tries allow a sequential search on the length dimension: first we look in the set of prefixes of length 1, then in the set of length 2 prefixes, and so on. Moreover, at each step the search space is reduced because of the prefix organization in the trie.

Another approach to sequential search on lengths without using a trie is organizing the prefixes in different tables according to their lengths. In this case, a hashing technique can be used to search in each of these tables. Since we look for the longest match, we begin the search in the table holding the longest prefixes; the search ends as soon as a match is found in one of these tables. Nevertheless, the number of tables equals the number of different prefix lengths. If W is the address length (32 for IPv4), the time complexity of the search operation is $O(W)$ assuming a perfect hash function, which is the same as for a trie.

In order to reduce the search time, a binary search on lengths was proposed by Waldvogel *et al.* [6]. In a binary search, we reduce the search space in each step by half. On which half to continue the search depends on the result of a comparison. However, an ordering relation needs to be established before being able to make comparisons and proceed to search in a direction according to the result. Comparisons are usually done using key values, but our problem is different since we do binary search on lengths. We are restricted to checking whether a match exists at a given length. Using a match to decide what to do next is possible: if a match is found, we can reduce the search space to only longer lengths. Unfortunately, if no match is found, we cannot be sure that the search should proceed in the direction of shorter lengths, because the BMP could be of longer length as well. Waldvogel *et al.* insert extra prefixes of adequate length, called *markers*, to be sure that, when no match is found, the search must proceed necessarily in the direction of shorter prefixes.

To illustrate this approach consider the prefixes shown in Fig. 18. In the trie we can observe the levels at which the prefixes are located. At the right, a binary search tree shows the levels or lengths that are searched at each step of the binary search on lengths algorithm. Note that the trie is only shown to understand the relationship between markers and prefixes, but the algorithm does not use a trie data structure. Instead, for each level in the trie, a hash table is used to store the prefixes. For example, if we search for the BMP of the address 11000010, we begin by searching the table corresponding to length 4; a match will be found because of prefix f, and the search proceeds in the half of longer prefixes. Then we search at length 6, where the marker 110000* has been placed. Since a match is found, the search proceeds to length 7 and finds prefix k as the BMP. Note that without the marker at level 6, the search procedure would fail to find prefix k as the BMP. In general, for each prefix entry a series of markers are needed to guide the search. Since a binary search only checks a maximum of $\log_2 W$ levels, each entry will generate a maximum of $\log_2 W$ markers. In fact, the number of markers required will be much smaller for two reasons: no marker will be inserted if the corresponding prefix entry already exists (prefix f in Fig. 18), and a single marker can be used to guide

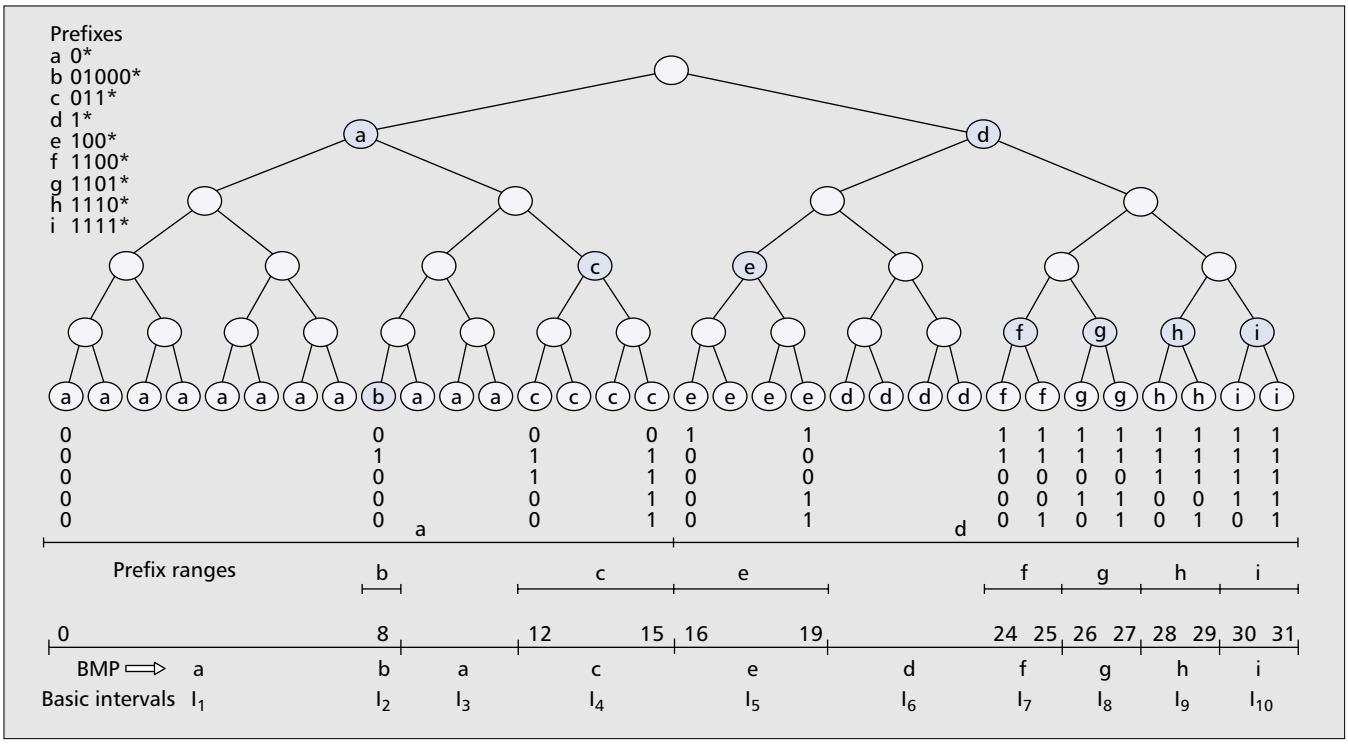


Figure 19. *Binary range search.*

the search for several prefixes (e.g., prefixes e and p, which use the same marker at level 2). However, for the very same reasons, the search may be directed toward longer prefixes, although no longer prefix will match. For example, suppose we search for the BMP for address 11000001. We begin at level 4 and find a match with prefix f, so we proceed to length 6, where we find again a match with the marker, so we proceed to level 7. However, at level 7 no match will be found because the marker has guided us in a bad direction. While markers provide valid hints in some cases, they can mislead in others. To avoid backtracking when being misled, Waldvogel uses precomputation of the BMP for each marker. In our example, the marker at level 6 will have f as the precomputed BMP. Thus, as we search, we keep track of the precomputed BMP so far, and then in case of failure we always have the last BMP. The markers and precomputed BMP values increase the memory required. Additionally, the update operations become difficult because of the several different values that must be updated.

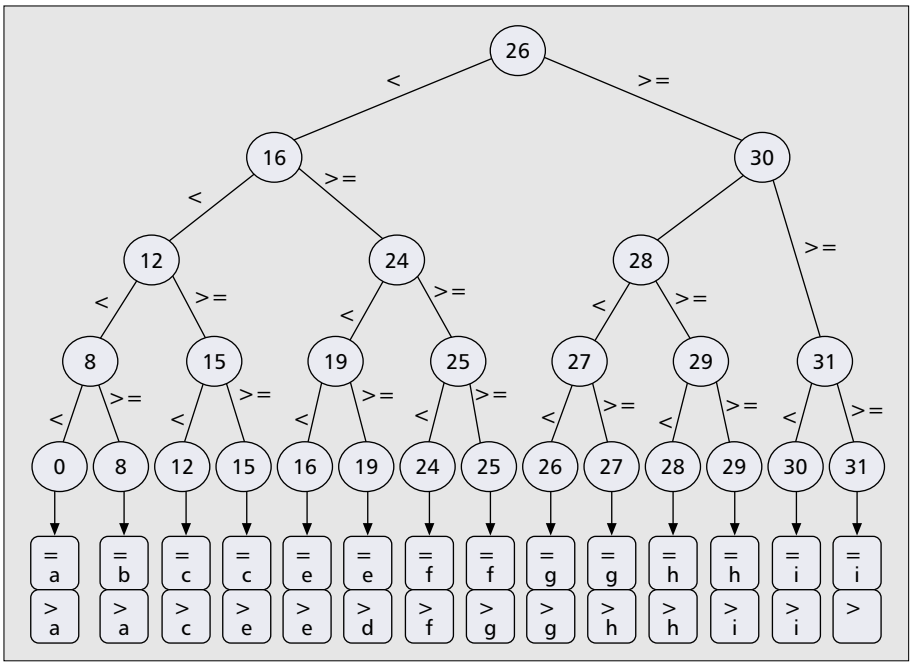
Prefix Range Search

A search on values only, to find the longest matching prefix, is possible if we can get rid of the length dimension. One way of doing so is to transform the prefixes to a unique length. Since prefixes are of arbitrary lengths, we need to do a full expansion, transforming all prefixes to 32-bit-length prefixes in the case of IPv4. While a binary search on values could be done now, this approach needs a huge amount of memory. Fortunately, it is not necessary to store all of the 2^{32} entries. Since a full expansion has been done, information redundancy exists.

A prefix represents an aggregation of contiguous addresses; in other words, a prefix determines a well-defined range of addresses. For example, supposing 5-bit-length addresses, prefix a = 0* defines the range of addresses [0,15]. So why not simple store the range *endpoints* instead of every single address? The BMP of the endpoints is, in theory, the same for all the addresses in the interval; and search of the BMP for a given address would be reduced to finding any of the endpoints of the corresponding interval (e.g., the predecessor, which is the greatest endpoint smaller than or equal to a given address). The BMP problem would be readily solved, because finding the predecessor of a given address can be performed with a classical binary search method. Unfortunately, this approach may not work because prefix ranges may overlap (i.e., prefix ranges may be included in other prefix ranges; Fig. 4). For example, Fig. 19 shows the full expansion of prefixes assuming 5-bit-length addresses. The same figure shows the endpoints of the different prefix ranges, in binary as well as decimal form. There we can see that the predecessor of address value 9, for instance, is endpoint value 8; nevertheless, the BMP of address 9 is not associated with endpoint 8 (b), but with endpoint 0 (a) instead. Clearly, the fact that a range may be contained in another range does not allow this approach to work. One solution is to avoid interval overlap. In fact, by observing the endpoints we can see that these values divide the total address space into disjoint basic intervals.

In a basic interval, every address actually has the same BMP. Figure 19 shows the BMP for each basic interval of our example. Note that for each basic interval, its BMP is the BMP of the shortest prefix range enclosing the basic interval. The BMP of a given address can now be found by using the endpoints of the basic intervals. Nevertheless, we can observe in Fig. 19 that some basic intervals do not have explicit endpoints (e.g., I_3 and I_6). In these cases, we can associate the basic interval with the closer endpoint to its left. As a result, some endpoints need to be associated to two basic intervals, and thus endpoints must maintain in general two BMPs, one for the interval they belong to and one for the potential next basic interval. For instance, endpoint value 8 will be associated with basic intervals I2 and I3, and must maintain BMPs b and a.

Figure 20 shows the search tree indicating the steps of the binary search algorithm. The leaves correspond to the endpoints, which store the two BMPs (= and >). For example, if we search the BMP for address 10110 (22), we begin comparing the address with key 26; since 22 is smaller than 26, we take the left branch in the search tree. Then we compare 22 with key 16 and go to the right; then at node 24 we go to the left arriving at



■ Figure 20. *A basic range search tree.*

node 19; and finally, we go to the right and arrive at the leaf with key 19. Because the address (22) is greater than 19, the BMP is the value associated with > (i.e., d).

As for traditional binary search, the implementation of this scheme can be made by explicitly building the binary search tree. Moreover, instead of a binary search tree, a multiway search tree can be used to reduce the height of the tree and thus make the search faster. The idea is similar to the use of multibit tries instead of binary tries. In a multiway search tree, internal nodes have k branches and $k - 1$ keys; this is especially attractive if an entire node fits into a single cache line because search in the node will be negligible compared to normal memory accesses.

As previously mentioned, the BMP for each basic interval needs to be precomputed by finding the shortest range (longest prefix) enclosing the basic interval. The problem with this approach, which was proposed by Lampson *et al.* [14], is that inserting or deleting a single prefix may require recomputing the BMP for many basic intervals. In general, every prefix range spans several basic intervals. The more basic intervals a prefix range covers, the higher the number of BMPs to potentially recompute. In fact, in the worst case we would need to update the BMP for N basic intervals, N as usual being the number of prefixes. This is the case when all $2N$ endpoints are different and one prefix contains all the other prefixes.

One idea to reduce the number of intervals covered by a prefix range is to use larger, yet still disjoint, intervals. The leaves of the tree in Fig. 20 correspond to basic intervals. A crucial observation is that internal nodes correspond to intervals that are the union of basic intervals (Fig. 21). Also, all the nodes at a given level form a set of disjoint intervals. For example, at the second level the nodes marked 12, 24, and 28 correspond to the intervals [0,15], [16,25], and [26,29], respectively. So why store BMPs only at leaves? For instance, if we store a at the node marked 12 in the second level, we will not need to store a at leaves, and update performance will be better. In other words, instead of decomposing prefix ranges into basic intervals, we decompose prefix ranges into disjoint intervals as large as possible. Figure 21 shows how prefixes can be stored using this idea. Search operation is almost the same, except that now it needs to keep track of the BMP encountered when traversing the path to the leaves. We can compare the basic scheme to using leaf pushing and the new method to not doing so. Again, we can see that pushing information to leaves makes update difficult, because the number of entries to modify grows. The multiway range tree approach [15] presents and develops this idea to allow incremental updates.

Comparison and Measurements of Schemes

Each of the schemes presented has its strengths and weaknesses. In this section, we compare the different schemes and discuss the important metrics to evaluate these schemes. The ideal scheme would be one with fast searching, fast dynamic updates, and a small memory requirement. The schemes presented make different tradeoffs between these aspects. The most important metric is obviously lookup time, but update time must also be taken into account, as well as memory requirements. Scalability is also another important issue, with respect to both the number and length of prefixes.

Complexity Analysis

The complexity of the different schemes is compared in Table 2. The next sections carry out detailed comparison.

Tries — In binary tries we potentially traverse a number of nodes equal to the length of addresses. Therefore, the search complexity is $O(W)$. Update operations are readily made and basically need a search, so update complexity is also $O(W)$. Since inserting a prefix potentially creates W successive nodes (along the path that represents the prefix), the memory consumption for a set of N prefixes has

Scheme	Worst case lookup	Update	Memory
Binary trie	$O(W)$	$O(W)$	$O(NW)$
Path-compressed tries	$O(W)$	$O(W)$	$O(N)$
k-stride multibit trie	$O(W/k)$	$O(W/k + 2^k)$	$O(2^kNW/k)$
LC trie	$O(W/k)$	–	$O(2^kNW/k)$
Lulea trie	$O(W/k)$	–	$O(2^kNW/k)$
Full expansion/compression	3	–	$O(2^k + N^2)$
Binary search on prefix lengths	$O(\log_2 W)$	$O(N\log_2 W)$	$O(\log_2 W)$
Binary range search	$O(\log_2 N)$	$O(N)$	$O(N)$
Multiway range search	$O(\log_2 N)$	$O(N)$	$O(N)$
Multiway range trees	$O(\log_2 N)$	$O(k\log_k N)$	$O(N\log_k N)$

■ Table 2. *Complexity comparison.*

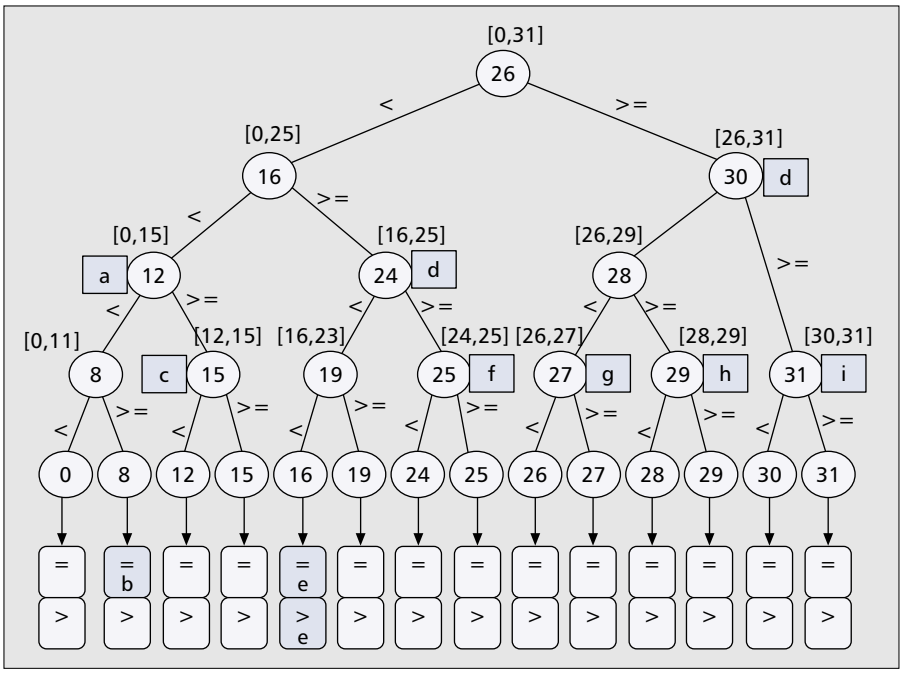


Figure 21. *A range search tree.*

complexity $O(NW)$. Note that this upper bound is not tight, since some nodes are, in fact, shared along the prefix paths. Path compression reduces the height of a sparse binary trie, but when the prefix distribution in a trie gets denser, height reduction is less effective. Hence, complexity of search and update operations in path-compressed tries is the same as in classical binary tries. Path-compressed tries are full binary tries. Full binary tries with N leaves have $N - 1$ internal nodes. Hence, space complexity for path compressed tries is $O(N)$.

Multibit tries still do linear search on lengths, but since the trie is traversed in larger strides the search is faster. If search is done in strides of k bits, the complexity of the lookup operation is $O(W/k)$. As we have seen, updates require a search and will modify a maximum of 2^{k-1} entries (if leaf pushing is not used). Update complexity is thus $O(W/k + 2^k)$ where k is the maximum stride size in bits in the multibit trie. Memory consumption increases exponentially with k: each prefix entry may need potentially an entire path of length W/k, and paths consist of one-level subtries of size 2^k. Hence, memory used has complexity $(O(2^kNW/k)$.

Since the Lulea and full expansion/compression schemes use compressed multibit tries together with leaf pushing, incremental updates are difficult if not impossible, and we have not indicated update complexity for these schemes. The LC trie scheme uses an array layout and must maintain lists of less specific prefixes. Hence, incremental updates are also very difficult.

Binary Search on Lengths — For a binary search on lengths, the complexity of the lookup operation is logarithmic in the prefix length. Notice that the lookup operation is independent of the number of entries. Nevertheless, updates are complicated due to the use of markers. As we have seen, in the worst case $\log_2 W$ markers are necessary per prefix; hence, memory consumption has complexity $O(N\log_2 W)$. For the scheme to work, we need to precompute the BMP of every marker. This precomputed BMP is a function of the entries being prefixes of the marker; specifically, the BMP is the longest. When one of these prefix entries is deleted or a new one is added, the precomputed BMP may change for many of the markers that are longer than the new (or deleted) prefix entry. Thus, the marker update complexity is $O(N\log_2 W)$ since theoretically an entry may potentially be prefix of $N - 1$ longer entries, each having potentially $\log_2 W$ markers to update.

Range Search — The range search approach gets rid of the length dimension of prefixes and performs a search based on the endpoints delimiting disjoint basic intervals of addresses. The number of basic intervals depends on the covering relationship between the prefix ranges, but in the worst case it is equal to $2N$. Since a binary or multiway search is performed, the complexity of the lookup operation is $O(\log_2 N)$ or $O(\log_k N)$, respectively, where k is the number of branches at each node of the search tree. Remember that the BMP must be precomputed for each basic interval, and in the worst case an update needs to recompute the BMP of N basic intervals. The update complexity is thus $O(N)$. Since the range search scheme needs to store the endpoints, the memory requirement has complexity $O(N)$.

We previously mentioned that by using intervals made of unions of the basic intervals, the approach of [15] allows better update performance. In fact, the update complexity is $O(k\log_k N)$, where k is the number of branches at each node of the multiway search tree.

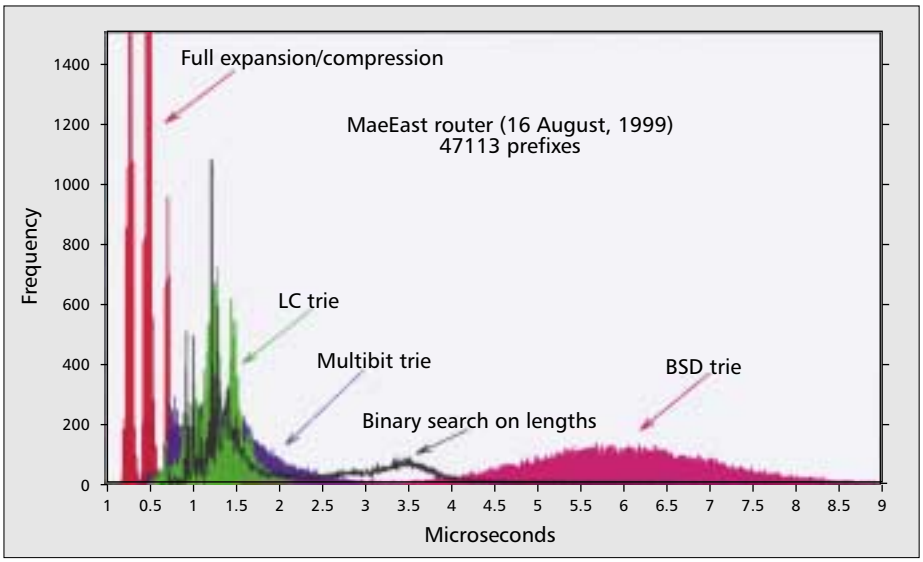
Scalability and IPv6 — An important issue in the Internet is scalability. Two aspects are important: the number of entries and the prefix length. The last aspect is especially important because of the next generation of IP (IPv6), which uses 128-bit addresses. Multibit tries improve lookup speed with respect to binary tries, but only by a constant factor on the length dimension. Hence, multibit tries scale badly to longer addresses. Binary search on lengths has a logarithmic complexity with respect to the prefix length, and its scalability property is very good. The range search approaches have logarithmic lookup complexity with respect to the number of entries but independent, in principle, of prefix length. Thus, if the number of entries does not grow excessively, the range search approach is scalable for IPv6.

Measured Lookup Time

While the complexity metrics of the different schemes described above are an important aspect for comparison, it is equally important to measure the performance of these schemes under "real conditions." We now show the results of a performance comparison made using a common platform and a prefix database of a typical backbone router [5].

Our platform consists of a Pentium-Pro-based computer with a clock speed of 200 MHz. The size of memory cache L2 is 512 kbytes. All programs are coded in C and were executed under the Linux operating system. The code for the path-compressed trie (BSD trie) was extracted from the FreeBSD implementation, the code for the multibit trie was implemented by us [16], and the code for the other schemes was obtained from the corresponding authors.

While prefix databases in backbone routers are publicly available, this is not the case for traffic traces. Indeed, traffic statistics depend on the location of the router. Thus, what we have done to measure the performance of the lookup opera-



■ Figure 22. *Lookup time distributions of several lookup mechanisms.*

Scheme	10th percentile	50th percentile (median)	99th percentile
BSD trie	4.63	5.95	8.92
Multibit trie	0.82	1.33	2.99
LC trie	0.95	1.28	1.98
Full expansion/compression	0.26	0.48	0.84
Binary search on prefix lengths	1.09	1.58	7.08

■ Table 3. *Percentiles of lookup times (μs).*

tion is to consider that every prefix has the same probability of being accessed. In other words, the traffic per prefix is supposed to be the same for all prefixes. Although a knowledge of the access probabilities of the forwarding table entries would allow better evaluation of the average lookup time, assuming constant traffic per prefix still allows us to measure important characteristics, such as the worst-case lookup time. In order to reduce the effects of cache locality we used a random permutation of all entries in the forwarding table (extended to 32 bits by adding zeroes). Figure 22 shows the distributions of the lookup operation for five different schemes. The lookup time variability for the five different schemes is summarized in Table 3.

Lookup time measured for the BSD trie scheme reflects the dependence on the prefix length distribution. We can observe a large variance between time for short prefixes and time for long prefixes because of the high height of the BSD trie. On the contrary, the full expansion/compression scheme always needs exactly three memory accesses. This scheme has the best performance for the lookup operation in our experiment. Small variations should be due to cache misses as well as background operating system tasks.

As we know, lookup times for multibit tries can be tuned by choosing different strides. We have measured the lookup time for the LC trie scheme, which uses an array layout and the path compression technique. We have also measured the lookup time for a multibit trie implemented with a linked tree structure and without path compression [16]. Both are variable-stride multibit tries that use the distribution of prefixes to guide the choice of strides. Additionally, the fill factor was chosen such that a stride of k bits is used if at least 50 percent of the total possible nodes at level k exist (discussed earlier). Even with this simple strategy to build multibit tries, lookup times are much better than for the BSD trie. Table 4 shows the statistics of the BSD trie and multibit tries, which explains the performance observed. The statistics for the corresponding binary trie are also shown. Notice that the height values of the BSD trie are very close to values for the binary trie. Hence, a path compression technique used alone, as in the BSD trie, has almost no benefit for a typical backbone router. Path compression in the multibit trie LC makes the maximum height much smaller than in the "pure" multibit trie. Nevertheless, the average height is only one level smaller than for the pure multibit trie. Moreover, since the LC trie needs to do extra comparisons in some cases, the gain in lookup performance is not very significant.

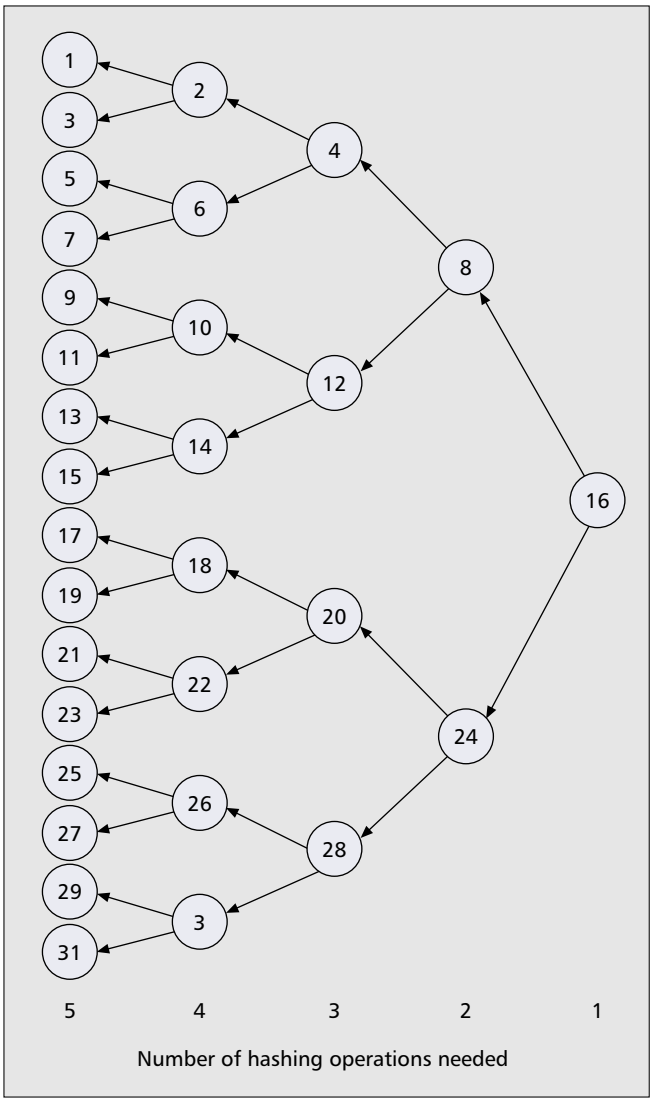
Scheme	Average height	Maximum height
Binary trie	21.84	30
BSD trie	19.95	26
LC trie	1.81	5
Multibit trie	2.76	12

■ Table 4. *Trie statistics for the MaeEast router (16 August, 1999).*

The binary search on lengths scheme also shows better performance than the BSD trie scheme. However, the lookup time has a large variance. As we can see in Fig. 23, different prefix lengths need a different number of hashing operations. We can distinguish five different groups, which need from one to five hashing operations. Since hashing operations are not basic operations, the difference, between a search that needs five hashes and one that needs only one hash can be significant. For example, lookup times of about 3.5 μs correspond to prefix lengths that need five hash operations.

Summary

To avoid running out of available IP addresses and reduce the amount of information exchanged by the routing protocols, a new address allocation scheme, CIDR, was introduced. CIDR promotes hierarchical aggregation of addresses and leads to relatively small forwarding tables, but requires a longest prefix matching operation. Longest prefix matching is more complex than exact matching. The lookup schemes we have surveyed manipulate prefixes by doing controlled disaggregation in order to provide faster search. Since original prefixes are usually transformed into several prefixes, to add, delete or change a single prefix requires updating several entries, and in the worst case the entire data structure needs to be rebuilt. Thus, in general a trade-off between lookup time and incremental update time needs to be made. We provide a framework and classify the schemes according to the algorithm-data structure aspect. We have seen that the difficulty with the longest prefix matching operation is its dual dimension: length and value. Furthermore, we describe how classical search techniques have been adapted to



■ Figure 23. *Standard binary search on lengths for IPv4.*

solve the longest prefix matching problem. Finally, we compare the different algorithms in terms of their complexity and measured execution time on a common platform. The longest prefix matching problem is important by itself; moreover, solutions to this problem can be used as a building block for the more general problem of packet classification [17].

Acknowledgments

The authors are grateful to M. Waldvogel, B. Lampson, V. Srinivasan, G. Varghese, P. Crescenzi, L. Dardini, R. Grossi, S. Nilsson, and G. Karlsson, who made available their code. It not only allowed us to perform comparative measurements, but also provided valuable insights into the solutions to the longest prefix matching problem.

We would also like to thank the anonymous reviewers for their helpful suggestions on the organization of the article.

References

[1] C. Labovitz, "Scalability of the Internet Backbone Routing Infrastructure," Ph.D. thesis, Univ. of Michigan, 1999.
[2] G. Huston, "Tracking the Internet's BGP Table," presentation slides: http://www.telstra.net/gih/prestns/ietf/bgptable.pdf, Dec. 2000.
[3] D. R. Morrison, "PATRICIA — Practical Algorithm to Retrieve Information Coded in Alphanumeric," *J. ACM*, vol. 15, no. 4, Oct. 1968, pp. 514–34.
[4] K. Sklower, "A Tree–Based Packet Routing Table for Berkeley Unix," *Proc. 1991 Winter Usenix Conf.*, 1991, pp. 93–99.
[5] Prefix database MaeEast, The Internet Performance Measurement and Analysis (IPMA) project, data available at http://www.merit.edu/ipma/routing_table/, 16 August, 1999.
[6] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High Speed IP Routing Lookups," *Proc. ACM SIGCOMM '97*, Sept. 1997, pp. 25–36.
[7] V. Srinivasan and G. Varghese, "Fast Address Lookups using Controlled Prefix Expansion," *Proc. ACM Sigmetrics '98*, June 1998, pp. 1–11.
[8] P. Gupta, S. Lin, and N. McKeown, "Routing Lookups in Hardware at Memory Access Speeds," *Proc. IEEE INFOCOM '98*, Apr. 1998, pp. 1240–47.
[9] S. Nilsson and G. Karlsson "IP-Address Lookup Using LC-Tries," *IEEE JSAC*, June 1999, vol. 17, no. 6, pp. 1083–92.
[10] G. Cheung and S. McCanne, "Optimal Routing Table Design for IP Address Lookups Under Memory Constraints," *Proc. IEEE INFOCOM '99*, Mar. 1999, pp. 1437–44.
[11] P. Crescenzi, L. Dardini, and R. Grossi, "IP Address Lookup Made Fast and Simple," *7th Annual Euro. Symp. Algorithms*; also, tech. rep. TR-99-01 Univ. di Pisa.
[12] M. Degermark *et al.*, "Small Forwarding Tables for Fast Routing Lookups," *Proc. ACM SIGCOMM '97*, Sept. 1997, pp. 3–14.
[13] W. Eatherton, "Full Tree Bit Map," Master's thesis, Washington Univ., 1999.
[14] B. Lampson, V. Srinivasan, and G. Varghese, "IP Lookups Using Multiway and Multicolumn Search," *Proc. IEEE INFOCOM '98*, Apr. 1998, pp. 1248–56.
[15] S. Suri, G. Varghese, and P. R. Warkhede, "Multiway Range Trees: Scalable IP Lookup with Fast Updates," Tech. rep. 99-28, Washington Univ., 1999.
[16] M. Á. Ruiz-Sánchez and W. Dabbous, "Un Mécanisme Optimisé de Recherche de route IP," *Proc. CFIP 2000*, Oct. 2000, pp. 217–32.
[17] A. Feldmann and S. Muthukrishnan, "Tradeoffs for Packet Classification," *Proc. IEEE INFOCOM 2000*, Mar. 2000, pp. 1193–1202.

Biographies

MIGUEL ÁNGEL RUIZ SANCHEZ (mruiz@sophia.inria.fr) graduated in electronic engineering from the Universidad Autónoma Metropolitana, Mexico City. In 1995 he joined the Electrical Engineering Department of the Universidad Autónoma Metropolitana-Iztapalapa in Mexico City, where he is associate professor. He was awarded his DEA (French equivalent of the Master's degree) in networking and distributed systems from the University of Nice Sophia Antipolis, France, in 1999. He is currently working toward a Ph.D. degree in computer science within the PLANETE team at INRIA Sophia Antipolis, France. His current research interests include forwarding and buffer management in routers.

ERNST BIERSACK [M '88] (erbi@eurecom.fr) received his M.S. and Ph.D. degrees in computer science from the Technische Universitaet München, Munich, Germany. Since March 1992 he has been a professor in telecommunications at Institut Eurecom, Sophia Antipolis, France. For his work on synchronization in video servers he received in 1996 (together with W. Geyer) the Outstanding Paper Award of the IEEE Conference on Multimedia Computing and Systems. For his work on reliable multicast he received (together with J. Nonnenmacher and D. Towsley) the 1999 W. R. Bennet Price of the IEEE for the best original paper published 1998 in the *ACM/IEEE Transactions on Networking*. He is currently an associate editor of *IEEE Network*, *ACM/IEEE Transactions on Networking*, and *ACM Multimedia Systems*.

WALID DABBOUS (dabbous@sophia.inria.fr) graduated from the Faculty of Engineering of the Lebanese University in Beirut in 1986 (Electrical Engineering Department). He obtained his DEA and Doctorat d'Université from the University of Paris XI in 1987 and 1991, respectively. He joined the RODEO team within INRIA in 1987. He is a staff researcher at INRIA since 1991, and leader of the Planete team since 2001. His main research topics are high-performance communication protocols, congestion control, reliable multicast protocols, audio and videoconferencing over the Internet, efficient and flexible protocol architecture design, and the integration of new transmission media such as satellite links in the Internet. He is co-chair of the UDLR working group at the IETF.