Survey and Taxonomy of IP Address Lookup Algorithms

Miguel Á. Ruiz-Sánchez[†], Ernst W. Biersack[‡], Walid Dabbous[†]

mruiz@sophia.inria.fr erbi@eurecom.fr dabbous@sophia.inria.fr

January 2, 2001

[†]INRIA Sophia Antipolis
2004, route des Lucioles, BP 93
06902 Sophia Antipolis
France

[‡]Institut Eurécom
06904 Sophia Antipolis
France

Abstract

Due to the rapid growth of traffic in the Internet, backbone links of several Gigabit/sec are commonly deployed. To handle Gigabit/sec 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 packets 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.

1 Introduction

The primary role of routers is to forward packets towards their final destination. To this purpose, a router must decide for each incoming packet where to send it next. More exactly, the forwarding decision consists in 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 paper 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.

1.1 The Classful Addressing Scheme

In IP version 4, IP addresses are 32 bit 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 the 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. So, finding the forwarding information requires to search for the prefix in the forwarding table that matches the corresponding bits of the destination address.

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

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 of what size each of them should be. When the 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: class A, B, and C (see figure 1). Size of networks was determined by the number of bits used to represent the network part and the host part. Thus networks of class A, B or C consisted in an 8, 16 or 24-bit network part and a corresponding 24, 16 or 8-bit host part.

With this scheme there were very few class A networks and their addressing space represented 50% 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 per network and 2,097,152 (2^{21}) class C networks with up to 256 hosts. This

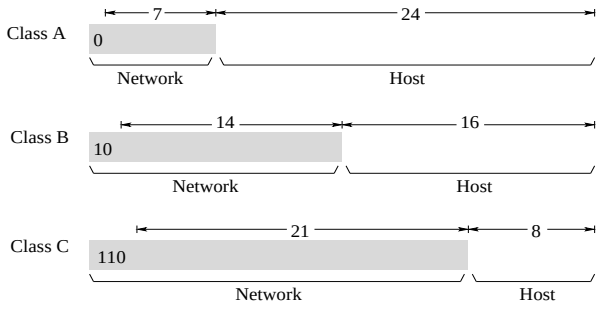

Figure 1: Classful Addresses

allocation scheme worked well in the early days of the Internet. However, the continuous growth of the number of hosts and networks have made apparent two problems with the classful addressing architecture: the rapid depletion of the IP address space and the growth in the size of the forwarding tables.

In effect, with only three different network sizes 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.

The second problem arose because even though 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 were growing 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.

1.2 The CIDR Addressing Scheme

To allow for a more efficient use of the IP address space and to slow down the growth of the backbone forwarding tables, a new scheme called Classless Inter-domain Routing or CIDR was introduced.

Remember, that in the classful address scheme, only 3 different prefix lengths are allowed: 8,16 and 24 corresponding to the classes A, B and C, respectively (see figure 1). CIDR makes more efficient use of the IP address space by allowing a finer granularity in the prefix lengths. In effect, 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.

To understand how this works, consider the networks represented by the network numbers from 208.12.16/24 through 208.12.31/24 (see figures 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 to prefixes of different lengths, for instance *208.12.16./20* (11010000 00001100 0001*) is different from

208.12.16./22 (11010000 00001100 000100*).

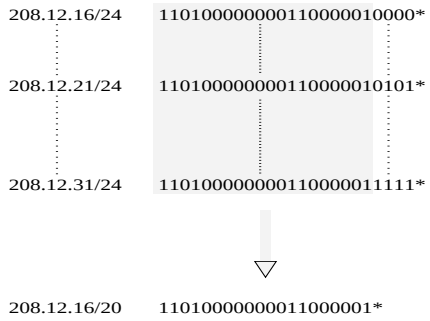

Figure 2: Prefix aggregation

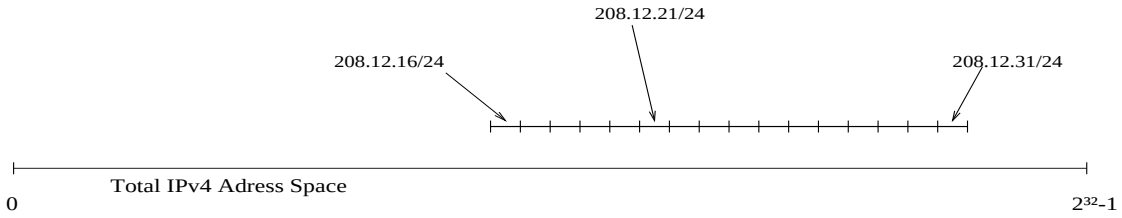

Figure 3: Prefix Ranges

While a great deal of aggregation can be achieved if addresses are carefully assigned, in some situations, a few networks can interfere with the process of aggregation. For example, suppose now that customer owing 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 (see figure 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, see figure 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. In effect, because of exceptions in the aggregations, a router must find the most specific match. The most specific match is the **longest matching prefix**. In summary, the address lookup problem in routers requires to search the forwarding table for the longest prefix that matches the destination address of a packet.

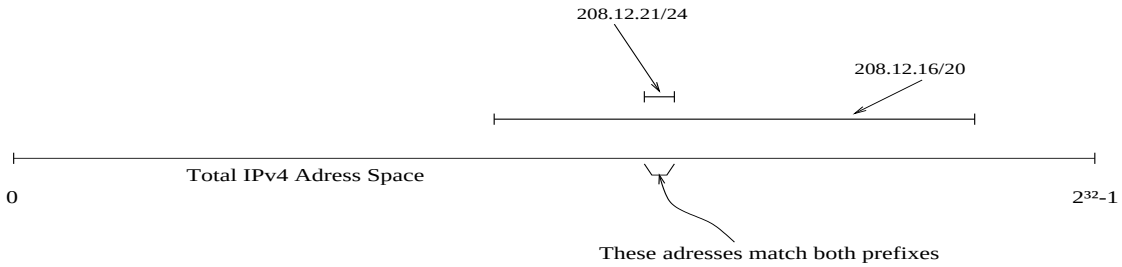

Figure 4: Exception prefix

1.3 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 (see figure 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 find 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 to reduce the size of the forwarding tables, the address lookup problem now becomes more complex. With CIDR, the destination prefixes in the forwarding tables have arbitrary lengths and do not correspond any more 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 cannot be performed any longer 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 to compare the bit pattern itself but also to find the appropriate length. Therefore, we talk about searching in two dimensions, the value dimension and the length dimension. 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 is typically also the length of the IP addresses.

1.4 Requirements on Address Lookup Algorithms

It is important to briefly resume the characteristics of the today's routing environment to derive the adequate requirements and metrics for the address lookup algorithms that 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 in early 1994 growth was exponential. From the mid of 1994 to the mid of 1998, growth slowed down and is nearly linear. From the mid of 1998 up to now growth is again exponential. Since the number of entries in router tables still grows, it is important that search methods reduce drastically 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 effect, instabilities in the backbone routing protocols can change fairly frequently the entries in a forwarding table. Labovitz [8] found that backbone routers may receive bursts of route changes at rates exceeding several hundred prefix updates per second. He also found that, in average, route changes occur one hundred times per second. Thus, the address lookup algorithms must be able to perform updates in 10 msec 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 the average prefix length would indicate improved aggregation and hierarchy in the Internet. In figure 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 scale is logarithmic). A recent study shows that the number of exceptions in the address aggregation is growing. More precisely, Huston [7] found that currently 40 % of the entries of a typical backbone forwarding table are prefix exceptions.

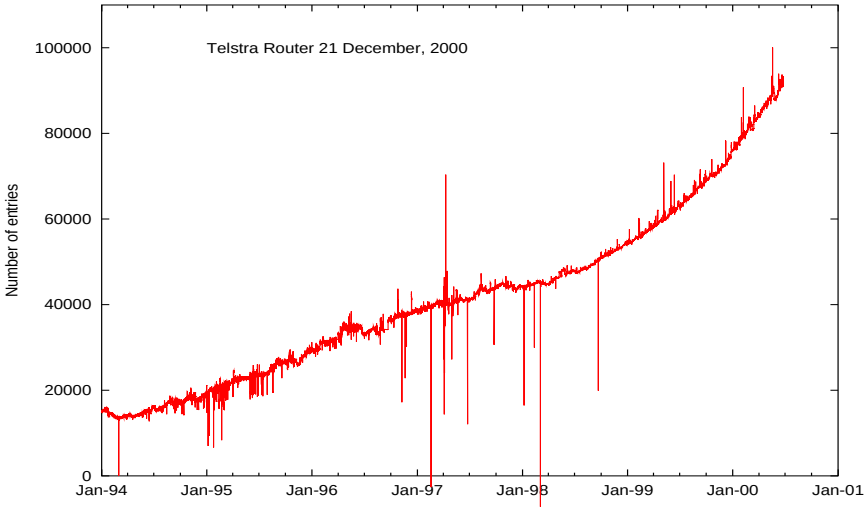

Figure 5: Table growth of a typical backbone router

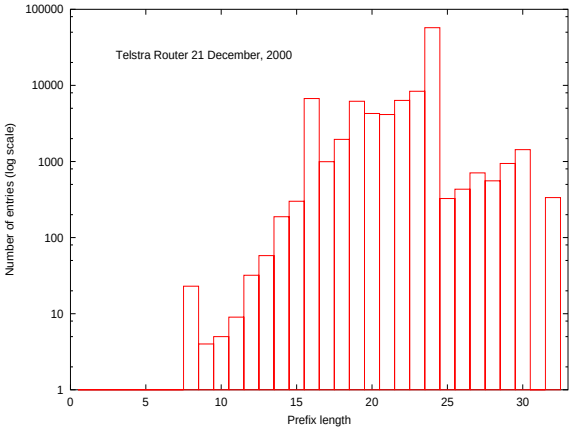

Figure 6: Prefix length distribution of a typical backbone router

2 Classical Solution

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.

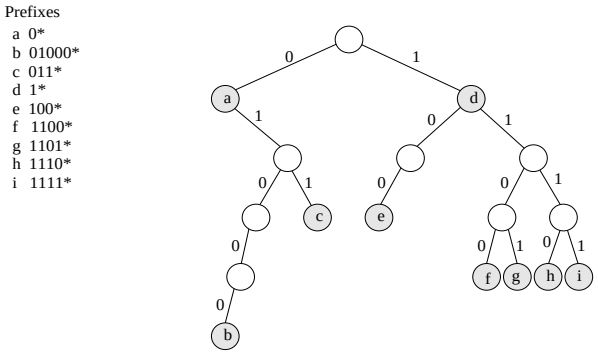

Figure 7: Binary trie for a set of prefixes.

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 figure 7 is at level *3* and represents all addresses beginning with the sequence *011*. The nodes that correspond to prefixes are shown in dark color and these nodes will contain the forwarding information or a pointer to the forwarding information. 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 figure 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 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 figure 7, this is reflected by the fact that prefixes *b* and *c* are descendants of prefix *a*, or 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, prefix *c* and prefix *a*. Nevertheless, prefix *c* must be preferred because it is more specific (longest match rule).

Tries allow in a straightforward way to find 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 to the right according to the sequential inspection of the address bits. While traversing the trie, every time we visit a node marked as prefix we remember it as the longest match found so far. The search ends, when there is no more branch 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 figure 7. Since the first bit of the address is *1* we move to the right, to the node marked with 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 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 search ends and the last remembered BMP (prefix *d*) is the longest

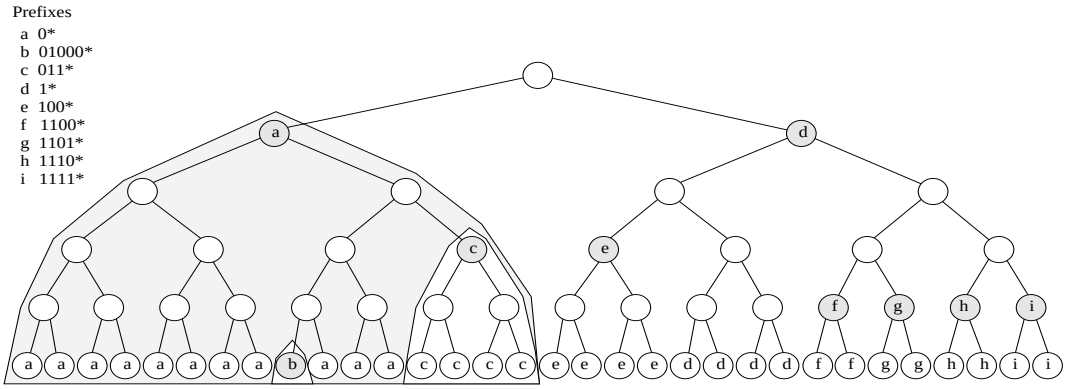

Figure 8: Address space

matching prefix.

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 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 nodes, 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 figure 7). Since these bits need to be inspected, even though no actual branching decision is made, search time can be longer than necessary for 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 in collapsing one-way branch nodes. When one-way branch nodes are removed from a trie, additional information must be kept in nodes, so that search operation can be performed correctly.

There are many ways to exploit the path-compression technique; perhaps the simplest to explain is illustrated in figure 9, corresponding to the binary trie in figure 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 not being a one-child node. As several one-child nodes may contain prefixes, in general, a linear 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 figure 9 these bit numbers are shown next to the nodes. A search in this kind of path-compressed tries 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

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. Search ends when a leaf is encountered. 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 figure 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 prefix we compare the 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, search stops and the last remembered BMP (prefix *a*) is the correct BMP.

Path-compression was first proposed for non-prefix applications and the resulting data structure was called Patricia trie (practical algorithm to retrieve information coded in alphanumeric). A variant supporting prefixes was published by Sklower [11]. In fact, this variant was originally designed not only to support prefixes but more general non-contiguous masks. Since this feature was really never used, current implementations differ somehow from the Sklower's original scheme. For example, the BSD version of the path-compressed trie (referred to as BSD trie) is essentially the same as we have just described. The difference is that the trie is traversed until a leaf is encountered, without checking the actual prefix values stored in 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 a list of prefixes, a comparison is performed to check for a match. 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 the original Sklower's scheme the backtrack phase also needs to do recursive descents of the trie because non-contiguous masks are allowed.

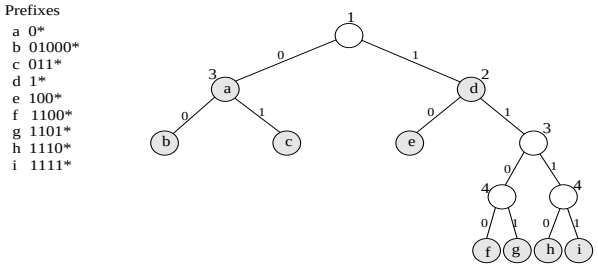

Figure 9: A path-compressed trie

Until recently, the longest matching prefix problem has been solved by using data structures based on path-compressed tries, like the BSD trie. Path-compression makes much 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 [16] with 47113 prefixes, the BSD version for a path-compressed trie creates 93304 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.

3 New IP Lookup Algorithms

We have seen that the difficulty with the longest prefix matching operation is its dual dimension: length and value. The new schemes for fast IP lookups differ in the dimension to search and whether this search is a linear or a binary search. In the following, we present a classification of the different IP lookup schemes.

3.1 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 section 6 how this can be done.

Search on lengths approaches: Another possibility is to base the search on the length dimension and to use linear search or binary search. Two possible ways of organizing the prefixes for search on lengths exist. In fact we have already seen linear search on lengths, which is performed on a trie. Tries allow at step *i* to check the prefixes of length *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 section 4, one optimization to this scheme consists in using multibit tries. Multibit tries still do linear search on lengths, but inspect several bits simultaneously at each step.

The other possible way of organizing the prefixes that allows a search on lengths is to use a different table for each possible length. Then, linear search on lengths can be made by doing at each step a search on a particular table using hashing, for instance. We will see in section 5 how Waldvogel et al. [15] use hash tables to do binary search on lengths.

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.

3.2 Auxiliary Techniques

Prefix transformation: Forwarding information is specified with prefixes that represent ranges of addresses. Although the set of prefixes to use 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 also, 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 (see figure 4). In a trie, when two prefixes overlap, one of them is itself a prefix of the other, see figures 7 and 8. Since prefixes represent intervals of contiguous addresses, when two prefixes overlap this means that one interval of addresses contains another interval of addresses, see figure 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 to 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 prefix of another one. 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, figure 10 shows the disjoint-prefix binary trie that corresponds to the trie in figure 7. Prefixes a_1, a_2, 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 has been called **leaf pushing** by Srinivasan et al. [12]. Figure 11 shows the disjoint intervals of addresses that correspond to the disjoint-prefix binary trie of figure 10.

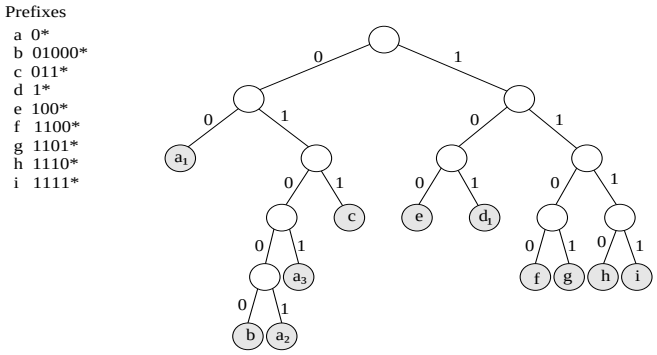

Figure 10: Disjoint-prefix binary trie

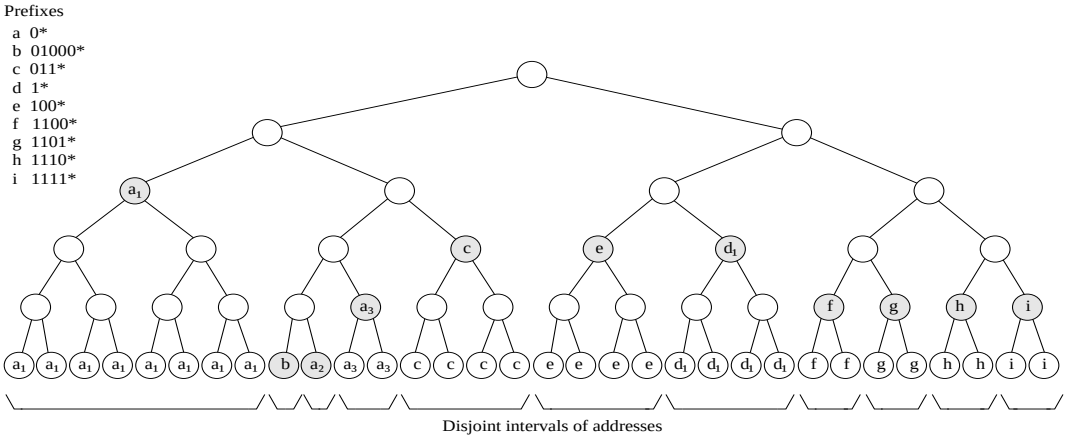

Figure 11: Expanded disjoint-prefix binary trie

Compression techniques: Data compression tries 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 in such a way that memory consumption is decreased and that 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 that 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 is more than one way to transform the set of prefixes. Optimization allows to define some constraints and to find the right set of prefixes satisfying 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 because they make it more likely that the forwarding table fits into the faster cache memory. Furthermore, the number of memory accesses must be minimized to make search 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 we will discuss other aspects as well. It is worth to mention that organizing the prefixes in different ways allows for different tradeoffs between the search cost and the update cost, as well as memory consumption. We discuss these tradeoffs when we explain the different schemes. We now present in detail some of the most efficient algorithms for IP address lookup.

4 Search on Prefix Lengths using Multibit Tries

4.1 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 **multibit** trie. Thus, a multibit trie is a trie where each node has 2^k children, where k is the stride.

Since multibit tries allow to traverse the data structure 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 figure 7 is shown in figure 12. We see that a first stride of two bits is used, so prefixes of length one are not allowed, and we need to expand prefixes *a* and *d* to produce four equivalent prefixes of length two. 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 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 three. 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 the one of the expanded prefix. Thus, expansion of prefixes *a* and *d* finally results in six prefixes and 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.

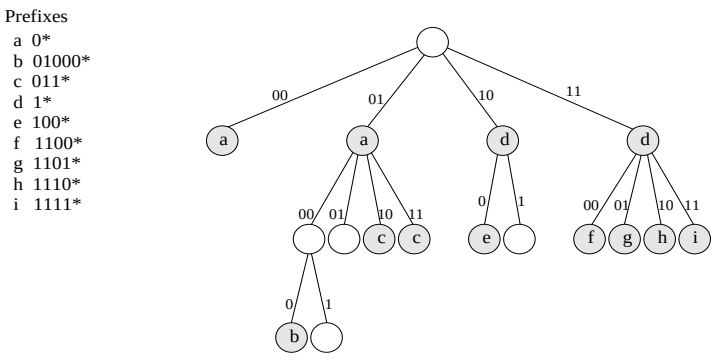

Figure 12: A variable stride multibit trie

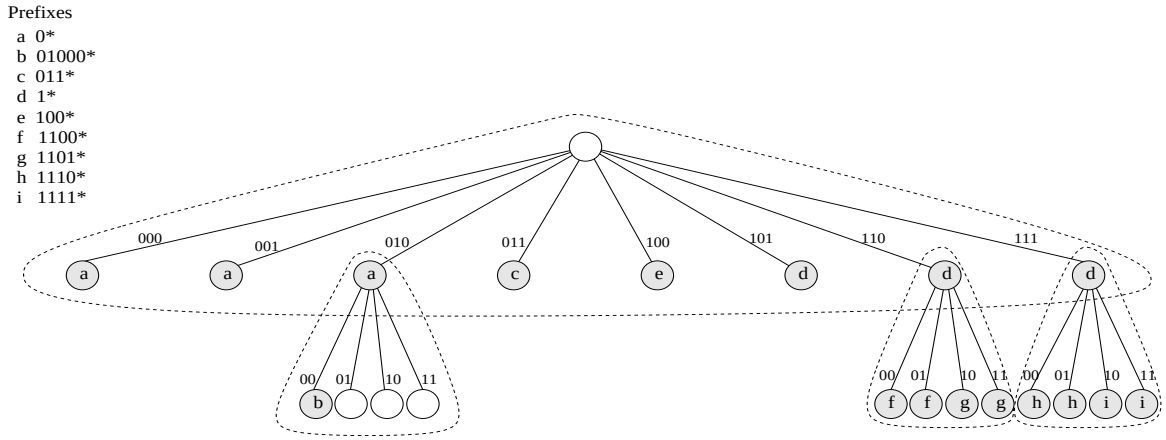

Figure 13: A fixed stride multibit trie

Searching in a multibit trie is essentially the same as in a binary trie. To find the BMP of a given address consists in 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 strides 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, while figure 12 shows a variable stride multibit trie.

4.2 Choice of Strides

Choosing the strides requires to make a tradeoff between search speed and memory consumption. In the extreme case, we could make a trie with a single level, that is 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.

One natural way to choose strides and control the memory consumption is to let the structure of the binary trie determine this choice. For example, if we look at figure 7, we can observe that the subtrie having as 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 has been already made in figure 12. This transformation is straightforward, but as it is the only transformation we can do in figure 7, it has a limited benefit. We will see later how to replace, in a controlled way, binary subtries that are not necessary full subtries . Height of the multibit trie will be reduced while controlling memory consumption. We will see also, how optimization techniques can be used to choose the strides.

4.3 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 figure 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 to find for each node the BMP among the prefixes *a*,*c*,*d*,*e*. In the leftmost subtrie at the second level the BMP for each node will be selected from the only prefix *b*. In the second subtrie at the second level, the BMP is selected for each node among the prefixes *f*,*g*, and the rightmost subtrie is concerned only with prefixes *h*,*i*. Some nodes may be empty indicating that there are no BMP 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 to note 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 needs only 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 to bound the update time.

Local BMPs allow incremental updates but require that internal nodes, besides leaves, store prefixes and 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 figure 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 not any more local and thus updates will suffer of longer worst case times.

As we can see, a multibit trie with several levels allows, by varying the stride *k*, an interesting tradeoff between search time, memory consumption, and update time. The length of the path can be controlled to reduce the search time. Choosing larger strides will make faster searches but more memory will be needed and updates will require to modify more entries 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 the 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

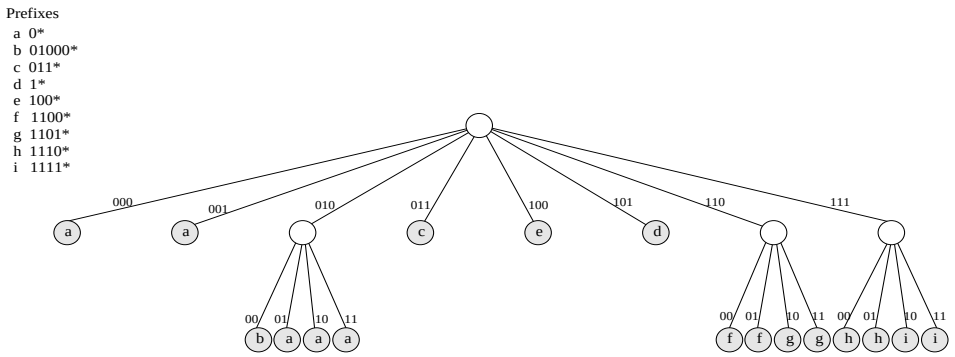

Figure 14: A disjoint-prefix multibit trie

and more importantly updating the entries with the next best matching prefix. 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 figure 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.

4.4 Multibit Tries in Hardware

The basic scheme of Gupta et al. [6] uses a 2 level multibit trie with fixed strides similar to the one shown in figure 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 very few entries have prefixes longer than 24 bits (see figure 6). As a result, using a first stride of 24 bits allows to find the BMP in one memory access for the majority of the cases. Also since very 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 illustrated in figure 14, for the example in figure 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 two 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 for this second memory bank depends on the expected worst case prefix length distribution. In the MaeEast table we examined in 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, that is a memory bank of 1 Mbyte. The design supports a maximum of 2^{12}=4096 subtries at the second level.

In figure 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 (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 (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 lookup in the second memory bank the forwarding information.

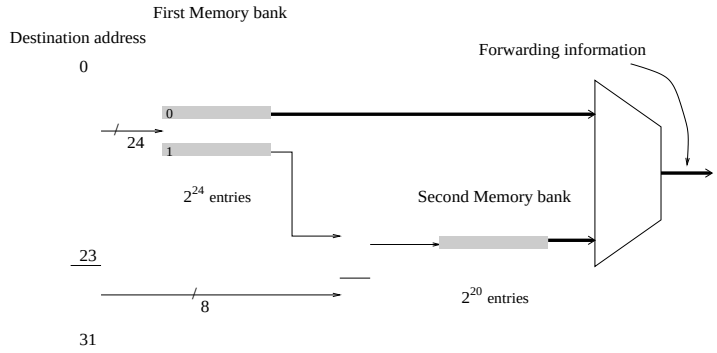

Figure 15: Gupta et al. hardware scheme

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 of 24 bits and leaf pushing is used, updates may take long time for some cases.

4.5 Multibit tries with Path-compression Technique

Nilsson et al. [10] recursively transform a binary trie with prefixes into a multibit trie: Starting at the root, we 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. LC (level-compressed) trie 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 to control 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 and not determine the choice of strides. In other words, we will replace nearly full binary subtries with a multibit subtrie, that is binary subtries where only few nodes are missing.

Nilsson proposes to replace 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 sufficient fraction of nodes is defined by using a single parameter called *fill factor x*, with $0 < x \leq 1$. For instance, in figure 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 nodes at third level, etc. Also, 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 ones. As a result, a search in an LC trie proceeds as follows: The LC trie is traversed like in the basic multibit trie. Nevertheless, since path compression is used, a explicit comparison must be performed when arriving at a leaf. In case of mismatch, a search in 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.

4.6 Multibit Tries and Optimization Techniques

One easy way to bound worst-case search times is by defining fixed strides that yield a well defined height for the multibit trie. Nevertheless the problem is that in general, memory consumption will be large, see section 4.4.

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 parameter to control the influence of the prefix distribution in the stride choice and so influences somehow the height of the resulting multibit trie. Since the prefix distribution still guides the stride choice, memory consumption is still controlled. Nevertheless, the use of the fill factor is simply a reasonable heuristic and more importantly it does not allow to guarantee a worst-case height.

Srinivasan et al. [12] use dynamic programming to determine, for a given prefix distribution, the optimal strides that minimize the 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 of minimize the lookup time is by taking 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. [1] 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.

4.7 Multibit Tries and Compression

Doing 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 to save memory and to 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. [2]. We will illustrate their method with a small example where we do a maximal expansion supposing 5-bit addresses and using a two level multibit trie. The first level uses a stride of 2 bits and the second level a stride of 3 bits, as it is shown in figure 16. The idea is to compress each of the subtries at the second level. In figure 17 we can see how the leaves of each second level subtries have been placed in a vertical fashion. 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 it is shown in figure 17. To find the BMP of a given address we traverse the first level of the multibit trie as usual, that is the first two bits of the address are used to

choose the correct subtrie at the second level. Then, the last three 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 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 level and second level is 16 bits. It is worth to note 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 takes always only three memory accesses. The reported memory size for a typical backbone router table is 1.2 Mbytes.

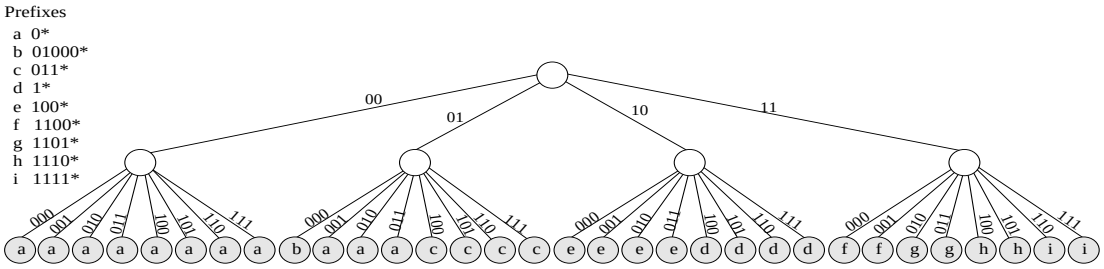

Figure 16: A two level full expanded multibit trie

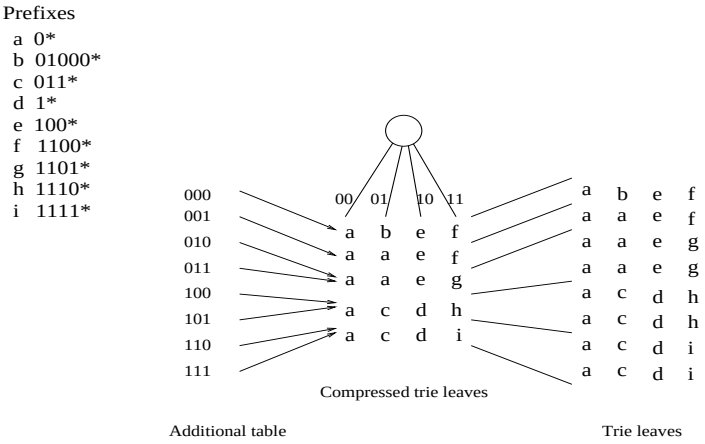

Figure 17: Full expansion parallel compression scheme

Another scheme that combines multibit tries with the compression idea has been dubbed the Lulea algorithm [3]. 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 an 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 and thus leaf pushing is used.

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

While the trie height in the Lulea scheme is 3, actually more than 3 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, 4 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 32732 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 [4]. Leaf pushing is avoided and so incremental updates are allowed.

5 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 by 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 and search ends as soon as a match is found in one of these tables. Nevertheless, the number of tables is equal to the number of different prefix lengths. If W is the addresses length, which is 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. [15]. In a binary search, we reduce the search space in each step by half. 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 the 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 check whether at a given length, a match exists. 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 best matching prefix 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 the figure 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 the BMP for the address 11000010, we begin by searching the table corresponding to length 4, a match will be found because of the 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 the 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

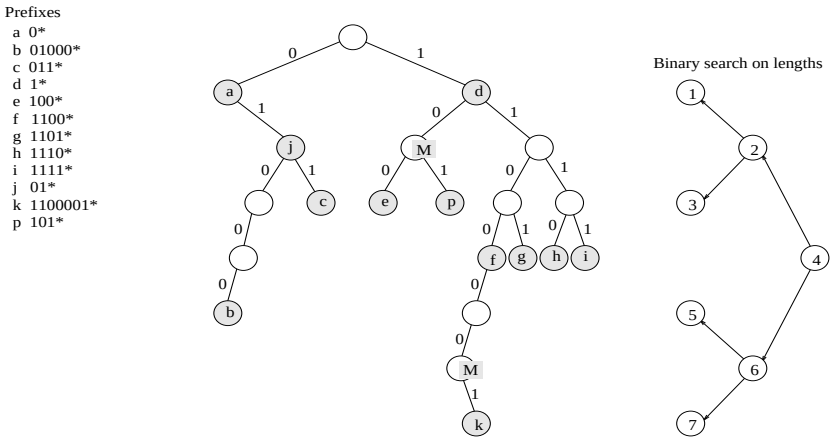

Figure 18: Binary search on prefix lengths

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 figure 18), and a single marker can be used to guide the search for several prefixes, see for example prefixes *e* and *p* which use the same marker at level 2. However, for the very same reasons, the search may be directed towards longer prefixes although no longer prefix will match. For example, suppose we search the BMP for address 11000001. We begin at level 4 and find a match with the 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 the bad direction. While markers provide valid hints in some cases, they can mislead in other cases. To avoid backtracking when being mislead, 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 best matching prefix. The markers and the precomputed BMP values increase the memory required. Additionally, the update operations become difficult because of the several different values that must be updated.

6 Prefix Range Search

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 this is by transforming the prefixes to a unique length. As 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. As a full expansion has been done, information redundancy exists.

In effect, 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 find any of the endpoints of the corresponding interval. For instance, 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, that is prefix ranges may be included in other prefix ranges, see figure 4. For example, figure 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 the address value 9, for instance, is the endpoint value 8; nevertheless the BMP of the address 9 is not the one associated to endpoint 8 (*b*), but the one associated to 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 has actually 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 figure 19 that some basic intervals do not have explicit endpoints (for example I3 and I6). 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, the endpoint value 8 will be associated to basic intervals I_2 and I_3, and must maintain BMP *b* and *a*.

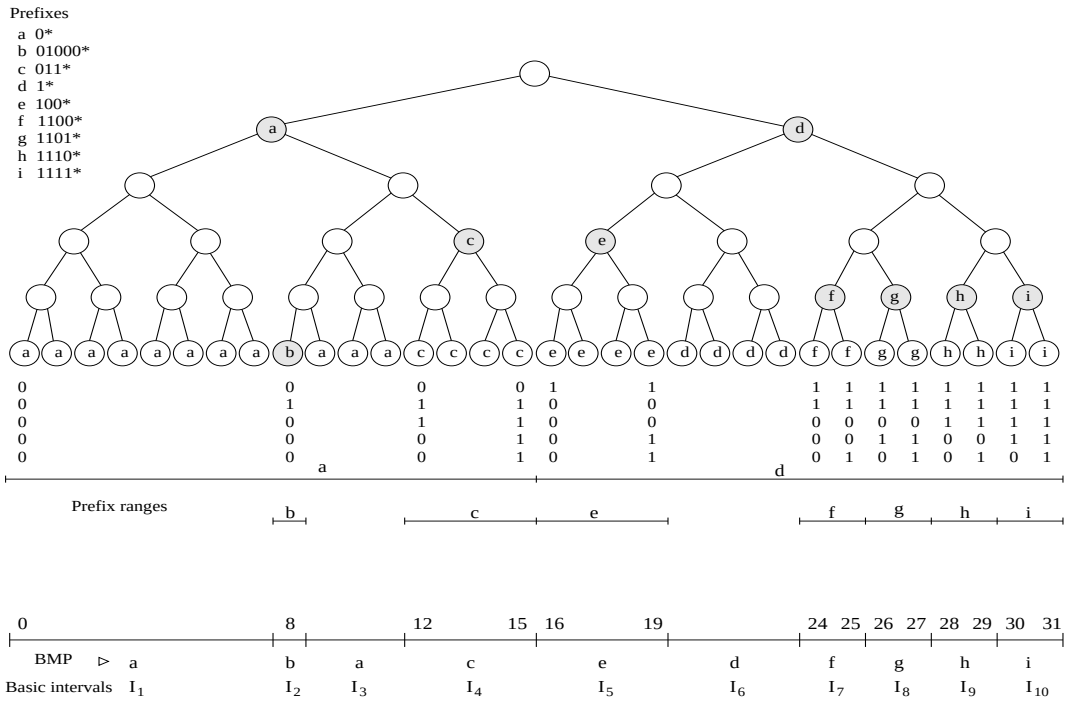

Figure 19: Binary range search

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 the address 10110 (22) we begin comparing the address with the key 26, as 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 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 ">", that is *d*.

As with 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 specially 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 we have 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. [9], is that inserting or deleting a single prefix may require to recompute the BMP for many basic intervals. In general, every prefix range spans several basic intervals. The more basic intervals a

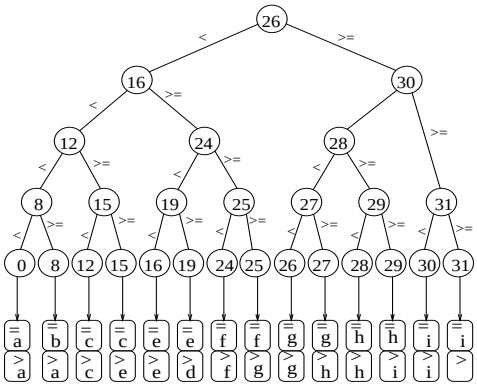

Figure 20: Basic Range search tree

prefix range cover, 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 the $2N$ endpoints are all 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 figure 20 correspond to basic intervals. A crucial observation is that internal nodes correspond to intervals that are the union of basic intervals, see figure21. 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 would be better. In other words, instead of decomposing prefix ranges into basic intervals, we decompose prefix ranges into disjoint intervals as larger as possible. Figure 21 shows how prefixes can be stored using this idea. Search operation is almost the same except that now 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 while the new method does not. 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 [14] presents and develops this idea to allow incremental updates.

7 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

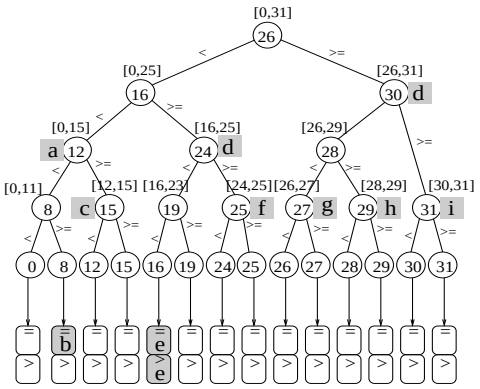

Figure 21: Range search tree

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 the lookup time, but update time must also be taken into account, as well as the memory requirements. Scalability is also another important issue, both, with respect to the number of prefixes and the length of prefixes.

7.1 Complexity Analysis

A comparison of the lookup complexity for the different schemes is shown in table 2. The next sections carry out a detailed comparison.

7.1.1 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 need only a search, so its complexity is also $O(W)$. The memory consumption is $O(NW)$ because each entry may need potentially an entire path of length W.

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 $O(\frac{W}{k})$. 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 $O(\frac{W}{k}+2^k)$ 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 $\frac{W}{k}$ and paths consist in one-level subtries of size 2^k. Hence memory used has complexity $O(2^k N \frac{W}{k})$.

Since the Lulea and the Full expansion/Compression schemes use compressed multibit tries together with the leaf pushing technique, 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 together with the path compression technique, hence incremental updates are also very difficult.

7.1.2 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 the

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 function of the entries being prefixes of the marker, specifically the BMP is the longest of them. 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 of one having potentially $\log_2 W$ markers to update.

7.1.3 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 a 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 have previously mentioned that by using intervals made of unions of the basic intervals, the approach of [14] allows a 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.

7.1.4 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 specially important because of the next generation of IP (IPv6) that uses 128 bit addresses. Multibit tries improve the 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 the prefix length. Thus, if the number of entries does not grow excessively, the range search approach is scalable for IPv6.

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	$O(\frac{W}{k})$	$O(\frac{W}{k}+2^k)$	$O(2^k N\frac{W}{k})$
LC trie	$O(\frac{W}{k})$	-	$O(2^k N\frac{W}{k})$
Lulea trie	$O(\frac{W}{k})$	-	$O(2^k N\frac{W}{k})$
Full expansion/compression	3	-	$O(2^k+N^2)$
Binary search on prefix lengths	$O(\log_2 W)$	$O(N\log_2 W)$	$O(N\log_2 W)$
Binary range search	$O(\log_2 N)$	$O(N)$	$O(N)$
Multiway range search	$O(\log_k N)$	$O(N)$	$O(N)$
Multiway range trees	$O(\log_k N)$	$O(k\log_k N)$	$O(Nk\log_k N)$

Table 2: Lookup time complexity

7.2 Measured Lookup Time

While the complexity metrics of the different schemes described in the last section 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 data-base of a typical backbone router [16].

Our platform consists in a Pentium Pro based computer with a clock speed of 200 MHz. The size of the 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 [13], the code for the Multibit trie was implemented by us, and the code for the other schemes were obtained from the corresponding authors.

While prefix data bases in backbone routers are publically 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 operation 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 a better evaluation of the average lookup time, assuming constant traffic per prefix still allows us to measure important characteristics, like 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 5 different schemes. The lookup time variability for the 5 different schemes is summarized in table 3.

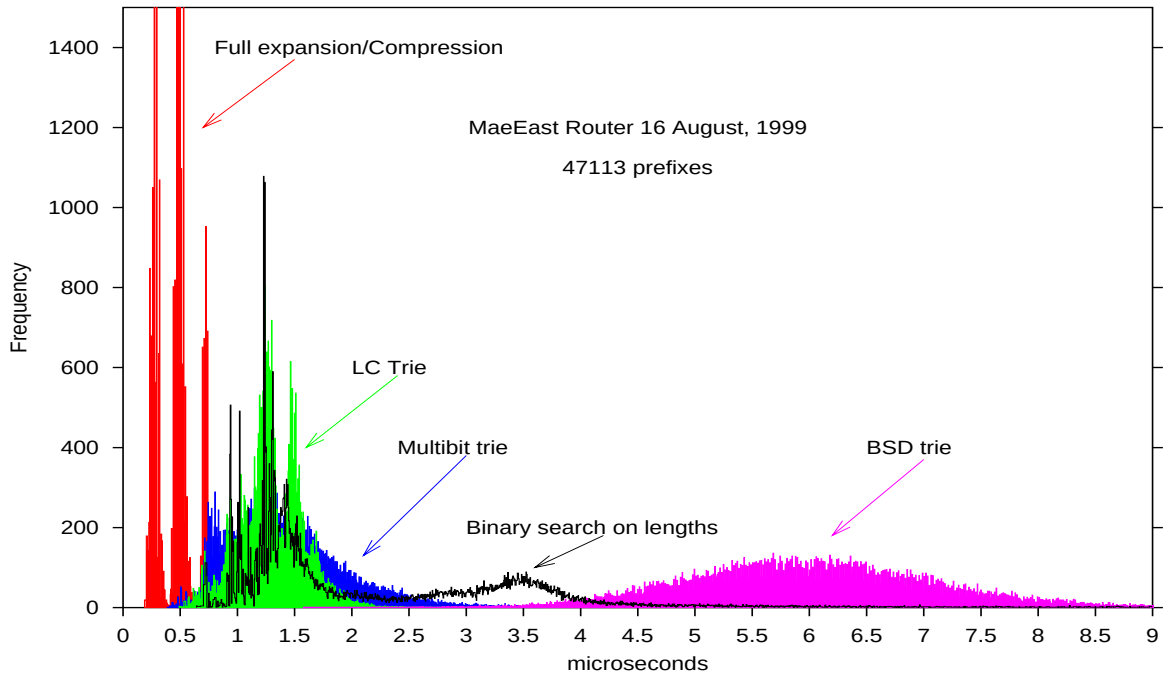

Figure 22: Lookup time distributions of several lookup mechanisms

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 3 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.

Scheme	10-percentile	50-percentile (Median)	99-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 the lookup times (μseconds)

As we know, lookup times for the 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 the path compression technique. Both of them are variable stride multibit tries that use the distribution of prefixes to guide the choice of strides. Additionally, the fill factor was chosen such as a stride of k bits is used if at least 50% of the total possible nodes at level k exist (see section 4.5). Even with this simple strategy to build the 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 is the case of 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, as 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 a better performance than the BSD trie scheme. However, the lookup time has a large variance. As we can see in figure 23, different prefix lengths need a different number of hashing operations. We can distinguish 5 different groups, which need from one to 5 hashing operations. As hashing operations are not basic operations the difference, between a search that needs 5 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 5 hash operations. In the MaeEast prefix table, a total of 10248 prefixes require 5 hashing operations.

8 Conclusions

To avoid running out of available IP addresses and to 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.

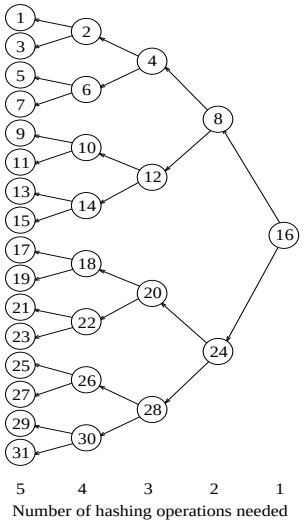

Figure 23: Standard binary search on lengths for IPv4

Longest prefix matching is more complex than an exact matching operation. The lookup schemes we have surveyed manipulate prefixes by doing controlled disaggregation in order to provide faster search. As original prefixes are usually transformed into several prefixes, to add, delete or change one single prefix requires to update several entries and in the worst case the entire data structure needs to be rebuilt. Thus, in general a tradeoff between lookup time and incremental update time needs to be made. We have provided a framework and classified 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 described how classical search techniques have been adapted to solve the longest prefix matching problem. Finally, we have compared 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 [5].

9 Acknowledgements

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 allowed us not only to perform comparative measurements but also provided valuable insights into the solutions to the longest prefix matching problem.

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

References

[1] G. Cheung and S. McCanne, “Optimal Routing Table Design for IP Address Lookups Under Memory Constraints,” Proceedings of IEEE INFOCOM’99, pp. 1437-1444, March 1999

[2] P. Crescenzi, L. Dardini, R. Grossi, “IP Address Lookup Made Fast and Simple,” 7th Annual European Symposium on Algorithms ESA’99, also as Technical Report TR-99-01 Università di Pisa.

[3] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, “Small Forwarding Tables for Fast Routing Lookups,” Proceedings of ACM SIGCOMM’97, pp. 3-14, September 1997.

[4] W. Eatherton, “Full Tree Bit Map,” Master’s Thesis, Washington University, 1999.

[5] A. Feldmann and S. Muthukrishnan, “Tradeoffs for Packet Classification,” Proceedings of IEEE INFOCOM 2000, pp 1193-1202, March 2000.

[6] P. Gupta, S. Lin, N. McKeown, “Routing Lookups in Hardware at Memory Access Speeds,” in Proceedings of IEEE INFOCOM’98, pp. 1240-1247, April 1998.

[7] G. Huston, “Tracking the Internet’s BGP Table,” presentation slides available at: http://www.telstra.net/gih/prestns/ietf/bgptable.pdf, December 2000.

[8] C. Labovitz, “Scalability of the Internet Backbone Routing Infrastructure,” Ph. D. Thesis, University of Michigan, 1999.

[9] B. Lampson, V. Srinivasan, and G. Varghese, “IP Lookups using Multi-way and Multicolumn Search,” in Proceedings of IEEE INFOCOM’98, pp. 1248-1256, April 1998.

[10] S. Nilsson and G. Karlsson “IP-Address Lookup Using LC-Tries,” IEEE Journal on Selected Areas in Communications June 1999, Vol. 17, Number 6, pp. 1083-1092.

[11] K. Sklower, “A tree -based packet routing table for Berkeley Unix,” Proceedings of the 1991 Winter Usenix Conference, pages 93-99, 1991.

[12] V. Srinivasan and G. Varghese, “Fast Address Lookups using Controlled Prefix Expansion,” Proceedings of ACM Sigmetrics’98, pp. 1-11, June 1998.

[13] W. Richard Stevens and Gary R. Wright. TCP/IP Illustrated, Vol. 2 The Implementation. Addison-Wesley, 1995.

[14] S. Suri, G. Varghese, and P. R. Warkhede, “Multiway Range Trees: Scalable IP Lookup with Fast Updates,” Technical Report 99-28, Washington University, 1999.

[15] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, “Scalable High Speed IP Routing Lookups,” Proceedings of ACM SIGCOMM’97, pp. 25-36, September 1997.

[16] Prefix database MaeEast, The Internet Performance Measurement and Analysis (IPMA) project, data available at: http://www.merit.edu/ipma/routing_table/, 16 August, 1999.