IP-Address Lookup Using LC-Tries

Stefan Nilsson and Gunnar Karlsson, *Senior Member, IEEE*

Abstract— **There has recently been a notable interest in the organization of routing information to enable fast lookup of IP addresses. The interest is primarily motivated by the goal of building multigigabit routers for the Internet, without having to rely on multilayer switching techniques. We address this problem by using an *LC-trie*, a trie structure with combined path and level compression. This data structure enables us to build efficient, compact, and easily searchable implementations of an IP-routing table. The structure can store both unicast and multicast addresses with the same average search times. The search depth increases as $\Theta(\log\log n)$ with the number of entries in the table for a large class of distributions, and it is independent of the length of the addresses. A node in the trie can be coded with four bytes. Only the size of the base vector, which contains the search strings, grows linearly with the length of the addresses when extended from 4 to 16 bytes, as mandated by the shift from IP version 4 to IP version 6. We present the basic structure as well as an adaptive version that roughly doubles the number of lookups/s. More general classifications of packets that are needed for link sharing, quality-of-service provisioning, and multicast and multipath routing are also discussed. Our experimental results compare favorably with those reported previously in the research literature.**

Index Terms—**IP address lookup, LC-trie, routing, search.**

I. Introduction

ADDRESS lookup is one of the fundamental functions of a router, along with the buffering, scheduling, and switching of packets. The lookup is a particularly critical function for building routers that should support multigigabit links. The lookup warrants a data structure for the organization of the routing information that quickly can be searched. In this study, we are primarily concerned with the choice of the data structure and its adaptation to the particular distribution of routing data. We evaluate the performance experimentally for a software implementation of an address-lookup system.

The IP uses the longest-matching prefix for determining which entry in a routing table should be used for a given packet address. In principle, this means that the prefixes are compared bit-by-bit to the given address, and the routing information associated with the longest of the matching prefixes should be used to forward the packet. The prefixes may be seen as branches of a binary tree with the routing information attached as leaves. When implemented in this way, there is one comparison and one memory lookup needed for each branching point in the tree. However, comparing strings of lengths equal or less than the machine word has basically a fixed cost, and it is therefore more efficient to compare more bits at a time in order to reduce the number of comparisons and memory accesses. This is the basic idea of this paper.

This paper presents the *LC-trie* as a suitable data structure for such efficient fast-address lookups in software. Our implementation can process 1 million addresses/s on a standard 133 MHz Pentium personal computer, which is sufficient to match a Gbit/s link (assuming an average packet length of 250 bytes). The performance scales nicely to fully exploit faster memory and processor clock rates, which is illustrated by the fact that a SUN Sparc Ultra II workstation can perform 5 million lookups/s.

In Section II, we review the procedure of address lookup for IP. In Section III, our solution is compared to those previously published. Section IV gives the needed background on the structure and properties of the level-compressed tries, along with the chosen data structure and the C-code. This structure is applied to the organization of IP-routing tables in Section V. The results for routing tables from the Internet's core routers and from a router in the Finnish University and Research Network (FUNET) are presented and discussed in Sections VI and VII. Section VIII outlines more general types of packet classifications for link sharing, flow-based routing, and multicast and multipath routing. A summary of this study closes the paper.

II. Address Lookup for the IP

A packet header for version 4 of the IP is 20 bytes long in its most basic form. Among other fields, it contains the addresses of the packet's source and destination. An IP address consists of a network identifier and a host identifier. Routing is solely based on the network identifier. Originally, the network identifier had a predetermined length, indicated by a prefix in the first address bits that specified the address class: 0 indicated class A with 8-bit network identifiers, 10 indicated class B with 16 bits, and 110 indicated class C with 24-bit identifiers (1110 is a class D address, which is used for multicast). This class-based structure is outmoded, however, by the introduction of classless interdomain routing (CIDR) [11]. An IP address can now be split into network and host identifiers at any point. The network identifier is the prefix that is stored in the routing table, and it is not necessarily the same for one and the same address in all routers. For instance, the address 222.21.67.68 gives the network identifier 222.21.64 with a prefix length of 18 bits and identifier 222.16 with a length of 12 bits. An address that would match both these two prefixes in a router should consequently be routed according to the information kept for the 18-bit prefix.

Manuscript received July 2, 1998; revised February 3, 1999.

S. Nilsson is with the Department of Computer Science, Royal Institute of Technology, Stockholm SE-100 44 Sweden (e-mail: snilsson@nada.kth.se).

G. Karlsson is with the Department of Teleinformatics, Royal Institute of Technology, Kista SE-164 40 Sweden (e-mail: gk@it.kth.se).

Publisher Item Identifier S 0733-8716(99)04484-4.

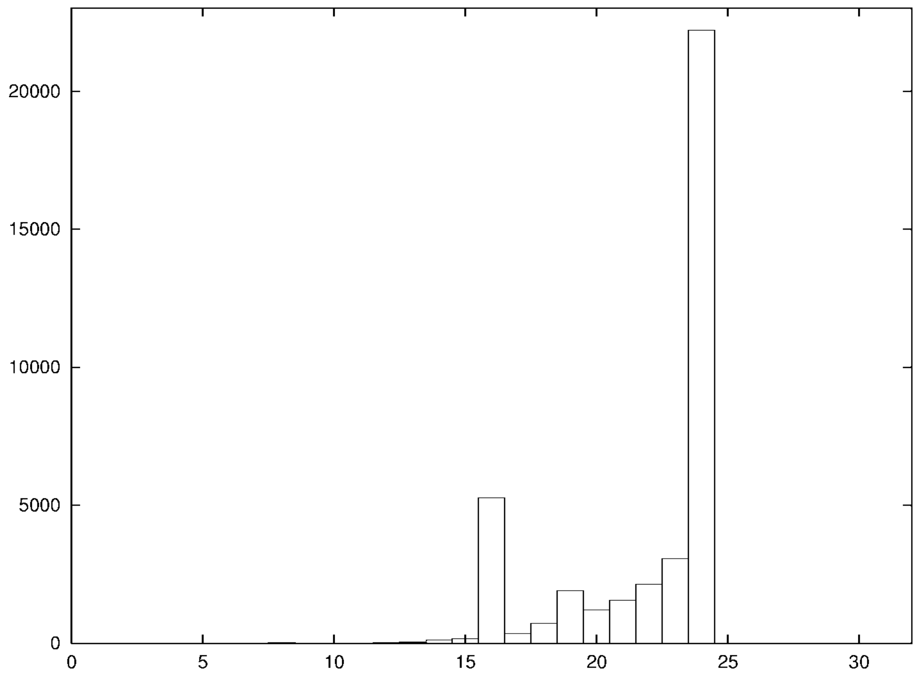

Fig. 1. Distribution of prefix lengths for the Mae East core router on December 2, 1997.

Although the class-based address structure has been abolished, it is still visible in routing table entries. Fig. 1 shows the lengths from the Mae East core router. There are pronounced peaks at length 24, and to a lesser degree, at length 16 (formerly class C and class B addresses, respectively). Prefixes of lengths below 16 bits are rare and could be extended to 16 bits. Thus, the first comparison of an address to stored prefixes could be based on the first 16 bits of the address, which would reduce the depth of the search tree considerably. As the redistribution of IP addresses is proceeding, we might expect a smoother distribution without such pronounced peaks. In our study, we show results for both a scheme with fixed 16-bit comparison at the first level in the tree and a fully adaptive scheme. The results are comparable in terms of the number of address lookups/s.

Most routers have a default route that is given by a prefix of length zero and therefore matches all addresses. The default route is used consequently if no other prefix matches. The core routers in the Internet are required to recognize all network identifiers and cannot resort to using a default route. The consequence is that their address tables tend to be larger than those in other routers. We have used tables from the core routers in our evaluation to ensure that we test with realistic data.

Although the address structure for IP version 6 (IPv6) is not fully decided even for unicast addresses, there are suggestions to keep the variable-length network identifiers (or subnetwork identifiers, as they are called here) [15]. Thus, a subnetwork can be identified by some m bits in a router, while the remaining $(128-m)$ bits form the interface identifier (replacing the host identifier of version 4) and are ignored. Our data structure is designed to easily handle IPv6 addresses, albeit with a corresponding increase in storage.

The result of an IP-address lookup in a router is the port number and next-hop address that should be used for the packet. The next-hop address is used to find the physical-link address (e.g., Ethernet address) for the next downstream router when the interconnection is via a shared-medium network. The next-hop information is consequently not needed for point-to-point links, and the corresponding routing-table entry would only contain the output port number. Even when next-hop addresses are needed, there are usually fewer distinct such addresses than there are entries in the routing table. The table can therefore contain a pointer to an array that lists the next-hop addresses in use.

III. Related Works

There has been a remarkable interest in the organization of routing tables during the last few years. The proposals include both hardware and software solutions. A hardware design is proposed by Moestedt and Sjödin using a pipeline structure that allows one address lookup per memory cycle [17]. Gupta *et al*. propose a similar structure [13]. (An earlier hardware solution may be found in [24].)

However, most new structures for fast address lookup are based on software-based searches. Degermark *et al*. [5] use a trie-like data structure. A central concern in their work is the size of the trie to ensure that it fits in a processor's on-chip cache memory. As a consequence, the structure will hardly scale to the longer addresses of IPv6. A main idea of their work is to quantify the prefix lengths to levels of 16, 24, and 32 bits and expand each prefix in the table to the next higher level. Rather than expanding the prefixes to some predefined levels, we use level-compression to reduce the size of the trie. Thus, we obtain similar sizes in our simulations with a more general structure, without assumptions about the address structure. This work has served as the inspiration for ours, however.

Srinivasan and Varghese [22] present a data structure akin to ours. It is also a binary trie structure, and it allows for multiway branching. By using a standard trie representation with arrays of children pointers, insertions and deletions of prefixes are supported. To minimize the size of the initial trie (before updates), dynamic programming is used.

Waldvogel *et al.* [23] take a different approach and store the entries in a hash-table. A lookup is performed as a binary search over the different possible lengths of the prefixes, and one hash-table lookup is performed at each step of this search. This is potentially expensive, but works well in practice for a number of routing tables, since the lengths of the entries are typically unevenly distributed (see Fig. 1). The algorithm exploits this fact and the search starts with the most probable prefix length. For the 128-bit addresses required by IPv6, this approach may require as many as seven hash-table lookups, each of which might in turn require several memory accesses.

Lampson *et al.* [16] present a solution based on a binary search. The idea is to use two copies of each entry; one copy is padded with ones and the other copy with zeros. With some additional precomputation, it is possible to perform a prefix match by doing a binary search in a sorted array containing these extended prefixes.

The earlier work on prefix matching by Doeringer *et al.* [7] also uses a trie structure. One of their concerns is to allow fully dynamic updates. This results in a large space overhead and a less than optimum performance. In fact, the nodes of the trie structure contain five pointers and one index. Also, the implementation in FreeBSD is based on a path-compressed trie structure [21]. The paper by Partridge *et al.* [8] also discusses the issue of IP-packet forwarding.

An earlier version of this work appeared in [19].

IV. Level-Compressed Tries

The trie [10] is a general-purpose data structure for storing strings. The idea is very simple: each string is represented by a leaf in a tree structure, and the value of the string corresponds to the path from the root of the tree to the leaf. Consider a small example. The binary strings in Fig. 2 correspond to the trie in Fig. 3(a). In particular, the string `010` corresponds to the path starting at the root and ending in leaf number 3: first a left-turn (`0`), then a right-turn (`1`), and finally a turn to the left (`0`). For simplicity, we will assume that the set of strings to be stored in a trie is prefix free; no string may be a proper prefix of another string. We postpone the discussion of how to represent prefixes until the next section.

This simple structure is not very efficient. The number of nodes may be large and the average depth (the average length of a path from the root to a leaf) may be long. The traditional technique to overcome this problem is to use *path compression* when each internal node with only one child is removed. Of course, we have to somehow record which nodes are missing. A simple technique is to store a number in each node, called the *skip value,* which indicates how many bits have been skipped on the path. A path-compressed binary trie is sometimes referred to as a Patricia tree [12]. The path-compressed version of the trie in Fig. 3(a) is shown in Fig. 3(b). The total number of nodes in a path-compressed binary trie is exactly $2n-1$, where n is the number of leaves in the trie. The statistical properties of this trie structure are very well understood [6], [20]. For a large class of distributions, path compression does not give an asymptotic reduction of the average depth. Even so, path compression is very important in practice, since it often gives a significant overall size reduction.

nbr	string
0	0000
1	0001
2	00101
3	010
4	0110
5	0111
6	100
7	101000
8	101001
9	10101
10	10110
11	10111
12	110
13	11101000
14	11101001

Fig. 2. Binary strings to be stored in a trie structure.

One might think of path compression as a way to compress the parts of the trie that are sparsely populated. *Level compression* [1] is a recently introduced technique for compressing parts of the trie that are densely populated. The idea is to replace the i highest complete levels of the binary trie with a single node of degree 2^i; this replacement is performed recursively on each subtrie. The level-compressed version of the trie in Fig. 3(b), the *LC-trie,* is shown in Fig. 3(c).

For an independent random sample with a density function that is bounded from above and below, the expected average depth of an LC-trie is $\Theta(\log^* n)$, where $\log^* n$ is the iterated logarithm function, $\log^* n = 1 + \log^*(\log n)$, if $n > 1$, and $\log^* n = 0$, otherwise. For data from a Bernoulli-type process with character probabilities that are not all equal, the expected average depth is $\Theta(\log \log n)$ [2]. Uncompressed tries and path-compressed tries both have expected average depth $\Theta(\log n)$ for these distributions.

A. Representation

If we want to achieve the efficiency promised by these theoretical bounds, it is important to represent the trie efficiently. The standard implementation of a trie, where a set of children pointers are stored at each internal node, is not a good solution since it has a large space overhead. This may be one explanation for why trie structures have traditionally been considered to require much memory.

A space-efficient alternative is to store the children of a node in consecutive memory locations. In this way, only a pointer to the leftmost child is needed. In fact, the nodes may be stored in an array, and each node can be represented by a single word. In our implementation, the first five bits represent the *branching factor,* the number of descendants of the node. This number is always a power of two, and hence using five bits, the maximum branching factor that can be represented

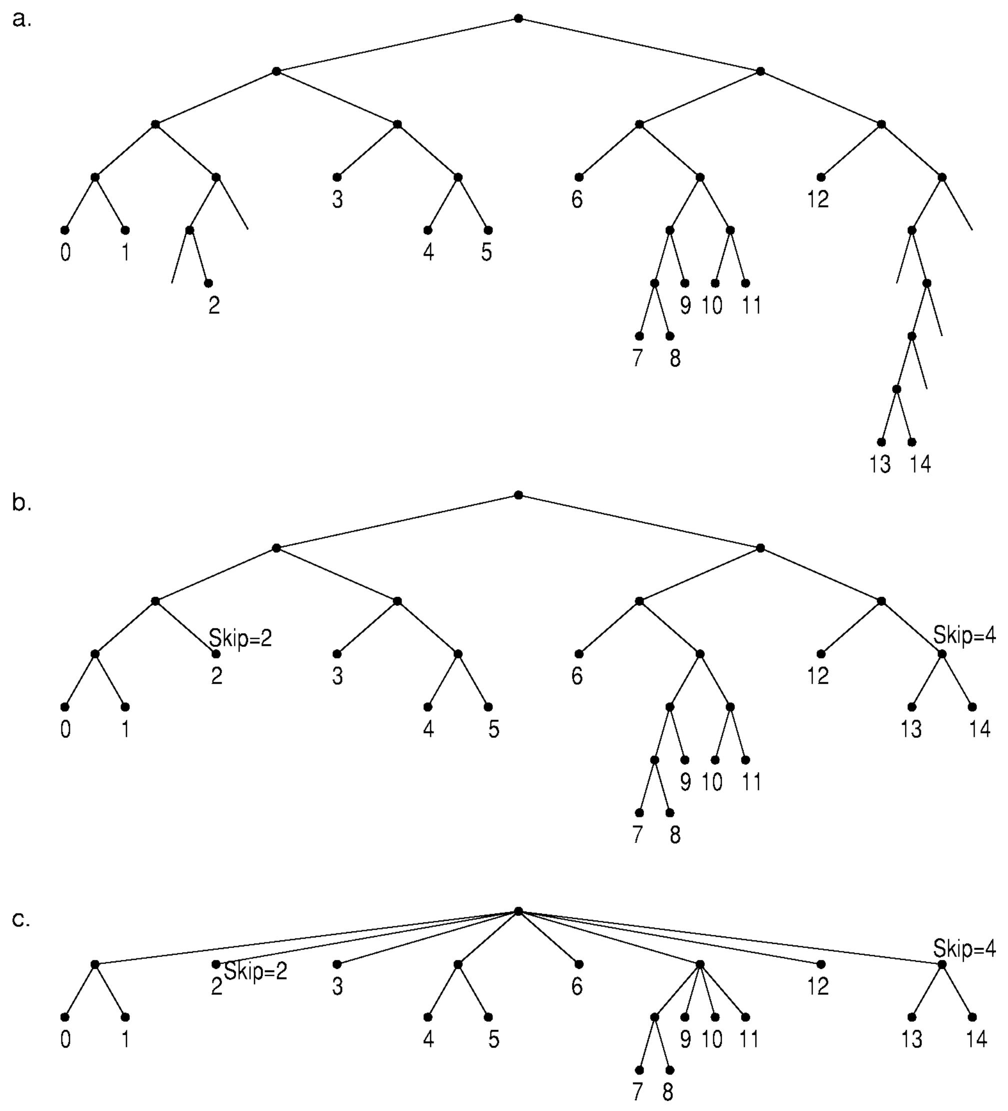

Fig. 3. (a) Binary trie. (b) Path-compressed trie. (c) LC-trie.

is 2^{31}. The next seven bits represent the skip value. In this way, we can represent values in the range 0–127, which is sufficient for IPv6 addresses. This leaves 20 bits for the pointer to the leftmost child, and hence using this very compact 32-bit representation, we can store at least $2^{19} = 524\,288$ strings (note that the largest address tables today contain about 50 000 entries). Fig. 4 shows the array representation of the LC-trie in Fig. 3(c); each entry represents a node. The nodes are numbered in breadth-first order starting at the root. The number in the branch column indicates the number of bits used for branching at each node. A value $k \geq 1$ indicates that the node has 2^k children. The value $k = 0$ indicates that the node is a leaf. The next column contains the skip value, which is the number of bits that can be skipped during a search operation. The value in the pointer column has two different interpretations. For an internal node, it is used as a pointer to the leftmost child; for a leaf, it is used as a pointer to a base vector containing the complete strings.

The search algorithm can be implemented very efficiently, as shown in Fig. 5. Let s be the string searched for, and let

	branch	skip	pointer
0	3	0	1
1	1	0	9
2	0	2	2
3	0	0	3
4	1	0	11
5	0	0	6
6	2	0	13
7	0	0	12
8	1	4	17
9	0	0	0
10	0	0	1
11	0	0	4
12	0	0	5
13	1	0	19
14	0	0	9
15	0	0	10
16	0	0	11
17	0	0	13
18	0	0	14
19	0	0	7
20	0	0	8

Fig. 4. Array representation of the LC-trie in Fig. 3(c).

```
node = trie[0];
pos = node.skip;
branch = node.branch;
adr = node.adr;
while (branch != 0) {
   node = trie[adr + EXTRACT(pos, branch, s)];
   pos = pos + branch + node.skip;
   branch = node.branch;
   adr = node.adr;
}
return adr;
```

Fig. 5. Pseudocode for the find operation in an LC-trie. The trie is represented by a vector, as shown in Fig. 4.

EXTRACT(p, b, s) be a function that returns the number given by the b bits starting at position p in the string s. We denote the array representing the tree by T. The root is stored in T[0].

Note that the address returned only indicates a possible hit; the bits that have been skipped during the search may not match. Therefore, we need to store the values of the strings separately and perform one additional comparison to check whether the search actually was successful.

As an example, we search for the string 10110111. We start at the root, node 0. We see that the branching value is three, and the skip value is zero, and therefore we extract the first three bits from the search string. These five bits have the value 5, which is added to the pointer, leading to position 6 in the array. At this node the branching value is two, and the skip value is zero, and therefore we extract the next two bits. They have the value 2. Adding two to the pointer, we arrive at position 15. At this node the branching value is zero, which implies that it is a leaf. The pointer value 5 gives the position of the string in the base vector. Observe that it is necessary to check whether this constitutes a true hit. We need to compare the first five bits of the search string with the first five bits of a value stored in the base vector in the position indicated by the pointer (10) in the leaf. In fact, our table (Fig. 2) contains a prefix 10 110 matching the string, and the search was successful.

B. Building the Trie

Building the trie is straightforward. The first step is to sort the base vector. A standard comparison-based sorting algorithm, such as quick sort [4], is typically sufficient. If a higher speed is required, a radix-sorting algorithm [3] may be utilized.

Given the sorted base vector, it is easy to make a top-down construction of the LC-trie, as demonstrated by the pseudocode in Fig. 6. This recursive procedure builds an LC-trie covering a subinterval of the base vector. This subinterval is identified by its first member (first), the number of strings (n), and the length of a common prefix (pre). Furthermore, the first free position (pos) in the vector that holds the representation of the trie is passed as an argument to the procedure. There are two cases. If the interval contains only one string, we simply create a leaf; otherwise, we compute the skip and branch values (the details are discussed to follow), create an internal node, and finally, the procedure is called recursively to build the subtries.

To compute the skip value, which is the longest common prefix of the strings in the interval, we only need to inspect the first and last string. If they have a common prefix of length k, all the strings in between must also have the same prefix since the strings are sorted.

The branching factor can also be computed in a straightforward manner. Disregarding the common prefix, start by checking if all four prefixes of length two are present (there are at least two branches since the number of strings is at least two, and the entries are unique). If these prefixes are present, we continue the search by examining if all eight prefixes of length 3 are present. Continuing this search, we will eventually find a prefix that is missing, and the branching factor will be established.

Allocating memory in the vector representing the trie can be done by simply keeping track of the first free position in this vector. (In the pseudocode, it is assumed that memory has been allocated for the root node before the procedure is invoked.) A more elaborate memory-allocation scheme may be substituted. For example, in some cases it may be worthwhile to use a memory layout that optimizes the cashing behavior of the search algorithm for a particular machine architecture.

To estimate the time complexity, we observe that the recursive procedure is invoked once for each node of the trie. Computing the skip value takes constant time. The branching factor of an internal node is computed in time proportional to the number of strings in the interval. Similarly, the increment statement of the inner loop is executed once for every string in the interval. This means that each level of the trie can be built in $O(n)$ time, where n is the total number of strings. Hence, the worst-case time complexity is $O(nh) + S$, where h is the height of the trie, and S is the time to sort the strings. Using a comparison-based sorting algorithm and assuming that the strings are of constant length, $S = O(n \log n)$. With radix sorting, we can achieve $S = O(n)$ under these assumptions.

C. Further Optimizations

In this section, we present a simple optimization that drastically reduces the depth of the trie. The idea is to use a weaker criterion for computing the branching factor. As presented earlier, a node can have a branching factor 2^k only if all prefixes of length k are present. This means that a few missing prefixes might have a considerable negative influence on the efficiency of the level compression. To avoid this, it is natural to require only that a fraction of the prefixes are present. In this way, we get a tradeoff between space and time. Using larger branching factors will decrease the depth of the trie, but it will also introduce superfluous empty leaves into the trie. In practice, however, this scheme gives substantial time improvements with only moderate increases in space.

In our implementation, we have implemented this by using a *fill factor* x, $0 < x \leq 1$. When computing the branching factor for a node covering k strings, we use the highest branching that produces at most $\lceil k(1-x) \rceil$ empty leaves. (There is no point in adding superfluous leaves to a node covering only

```
build(int first, int n, int pre, int pos)
{
   if (n == 1) {
      trie[pos] = {0, 0, first};
      return;
   }

   skip = computeSkip(pre, first, n);
   branch = computeBranch(pre, first, n, skip);
   adr = allocateMemory(2^branch);
   trie[pos] = {branch, skip, adr};

   p = first;
   for bitpat = 0 to 2^branch - 1 {
      k = 0;
      while (EXTRACT(pre + skip, branch, base[p + k]) == bitpat)
         k = k + 1;
      build(p, k, pre + skip, adr + bitpat);
      p = p + k;
   }
}
```

Fig. 6. Pseudocode for building an LC-trie given a sorted base vector. The procedure builds a trie that covers a subrange of the base vector (`base`) starting at position `first` and consisting of `n` elements. The first `pre` bits of each string are disregarded. The trie is represented by a vector (`trie`), and the first free position in this vector is given by `pos`.

two strings, and hence we always use the branching factor 2 in this case.)

In particular, we observe that the branching factor at the root of the trie has a large influence on overall behavior. Using a large branching factor at the root affects the path for all strings in the trie. Hence, it is particularly advantageous to use a large branching factor for the root. Therefore, we have included an option to fix a branching factor at the root, independently of the fill factor.

V. Routing Table

The routing table consists of four parts. At the heart of the data structure, we have an LC-trie implemented, as discussed in the previous section. The leaves of this trie contain pointers into a *base vector,* where the complete strings are stored. Furthermore, we have a *next-hop table,* an array containing all possible next-hop addresses, and a special *prefix vector,* which contains information about strings that are proper prefixes of other strings. This is needed because internal nodes of the LC-trie do not contain pointers to the base vector.

The base vector is typically the largest of these structures. Each entry contains a string. In the current implementation, it occupies 32 bits, but it can of course easily be extended to the 128 bits required in IPv6. Each entry also contains two pointers: one pointer into the next-hop table and one pointer into the prefix table. The search routine follows the next-hop pointer if the search was successful. If not, the search routine tries to match a prefix of the string with the entries in the prefix table. The prefix pointer has the special value -1 if no prefix of the string is present.

The prefix table is also very simple. Each entry contains a number that indicates the length of the prefix. The actual value does not need to be explicitly stored since it is always a proper prefix of the corresponding value in the base vector. As in the base vector, each entry also contains two pointers: one pointer into the next-hop table and one pointer into the prefix table. The prefix pointer is needed since it might happen that a path in the trie contains more than one prefix.

The main part of the search is spent within the trie. In our experiments, the average depth of the trie is typically less than two, and one memory lookup is performed for each node traversed. The second step is to access the base vector. This accounts for one additional memory lookup. If the string is found at this point, then one final lookup in the next-hop table is made. This memory access will be fast since the next-hop table is typically very small—in our experiments, less than 60 entries.

Finally, if the string searched for does not match the string in the base vector, an additional lookup in the prefix vector will have to be made. Also, this vector is typically very small—in our experiments, it contains less than 2000 entries, and it is rarely accessed more than once per lookup: in all the routing tables that we have examined, we have found only a few multiple prefixes. Conceptually, a prefix corresponds to an exception in the address space. Each entry in the routing table defines a set of addresses that share the same routing-table entry. In such an address set, a longer match corresponds to a subset of addresses that should be routed differently. We expect this special case to be less frequent with IPv6; the address space is so much larger that it should be possible to allocate the addresses in a strictly hierarchical fashion.

There are several minor ideas that have been considered but were never included in the implementation; the potential

TABLE I
RESULTS FOR A STRUCTURE WITH 16 BITS USED FOR BRANCHING AT THE ROOT AND A FILL FACTOR OF 0.5. THE SPEED IS MEASURED IN MILLION LOOKUPS/S

Site	Routing Entries	Next-Hops	Number of Entries			Av. depth (Max depth)	Lookups	
			Trie	Base	Prefix		Sparc	PC
Mae East	38 367	59	114 319	36 859	1508	1.66 (5)	2.0	0.6
Mae West	15 022	57	81 817	14 621	401	1.29 (5)	3.8	0.8
AADS	20 299	19	91 149	19 846	453	1.42 (5)	3.2	0.7
Pac Bell	20 611	3	91 871	20 171	440	1.43 (5)	2.6	0.7
FUNET	41 578	20	128 865	39 765	1813	1.73 (5)	5.0	1.2

gain was thought to be too insignificant. Most would yield small savings in memory, which as our results show, does not greatly influence the execution speed. However, we briefly describe these ideas in this section for completeness. First, the size of the nodes in the search trie could be reduced by allowing either skipping or branching but not both. Since the skip value is the larger of the two, we could save four out of the five bits given to the branching factor and use one bit to distinguish the two uses of the field. Second, a range of the pointer values in the trie could index the next-hop vector directly. It would be used when a path has been followed without skipping any bits in the address, and thus a comparison with the string stored in the base vector is not required. One memory access would thus be saved for these addresses. Third, the base vector entries could be shrunk by only storing the patterns that should match the bits skipped in the path compression, which typically accounts for a smaller part of the full string. Also, the two pointers to the next-hop and prefix vectors could be reduced to one and two bytes, respectively. Fourth, we could apply different filling factors in different parts of the trie for the relaxed level compression to allow a more fine-grained tuning between search depth and trie size. However, we only fix the branching at the root to 16 bits, independently of the fill factor.

VI. EXPERIMENTS

Our solution is fully public, as well as our source code and data.[1] The measurements were performed on two different machines: a SUN Ultra Sparc II with two 296-MHz processors and 512 megabytes of RAM and a personal computer with a 133-MHz Pentium processor and 32 megabytes of RAM. The programs are written in the C programming language and have been compiled with the `gcc` compiler using optimization level `−O4`. We used routing tables provided by the Internet performance measurement and analysis project.[2] We have used the routing tables for Mae East and Mae West from October 30, 1997 and AADS and Pac Bell from August 24, 1997. We did not have access to the actual traffic being routed according to these tables, and therefore the traffic is simulated: we simply use random permutations of all entries in a routing table. The entries were extended to 32-bit numbers by adding zeros (this should not affect the measurements since these bits are never inspected by the search routine). We have also tested our algorithm on a routing table with recorded traces of the actual packet destinations. The router is part of FUNET. Real traffic gives better results than runs of randomly generated destinations and owes to dependencies in the destination addresses. The time measurements have been performed on sequences of lookup operations, where each lookup includes fetching the address from an array, performing the routing table lookup, accessing the nexthop table, and assigning the result to a volatile variable.

[1] See http://www.nada.kth.se/∼snilsson.

[2] See http://www.merit.edu/ipma.

Some of the entries in the routing tables contain multiple next-hops. In this case, the first one listed was selected as the next-hop address for the routing table, since we only considered one next-hop address per entry in the routing table. There were also a few entries in the routing tables that did not contain a corresponding next-hop address. These entries were routed to a special next-hop address different from the ones found in the routing table.

VII. DISCUSSION OF RESULTS

Table I shows some measurements for an LC-trie with fill factor 0.50 and using 16 bits for branching at the root. It shows the number of entries in the routing table, the number of next-hop addresses, the size of our data structure, the average of the trie, and the number of lookups measured in million lookups/s. In our current implementation, an entry in the trie occupies 4 bytes, while the entries in the base vector and the prefix vector each occupy 16 and 12 bytes, respectively. Hence, the size of the LC-trie is less than 500 kB. The average throughput corresponding to the number of lookups/s is found by multiplying it with the average packet size, which currently is around 250 bytes. In effect, the structure can sustain over half a million complete lookup operations/s on a 133 MHz Pentium personal computer, and more than 2 M/s on a more powerful SUN Sparc Ultra II workstation. These numbers pertain to the U.S. core routers and randomly generated traffic with no dependencies in the destination addresses.

It is interesting to note that when using actual traffic traces from FUNET, the lookup time is about twice as fast (with 5 million lookups/s on the SUN workstation), even though this trie is both larger and deeper than the others. This can be explained by locality in the traffic traces. Several lookups close to each other in the trie will be considerably faster due to the native caching scheme used by the machine.

The worst-case lookup time is bounded by the maximal path through the search structure. In our experiments, the maximum depth of the LC-trie is at most five, while the average depth is just below two.

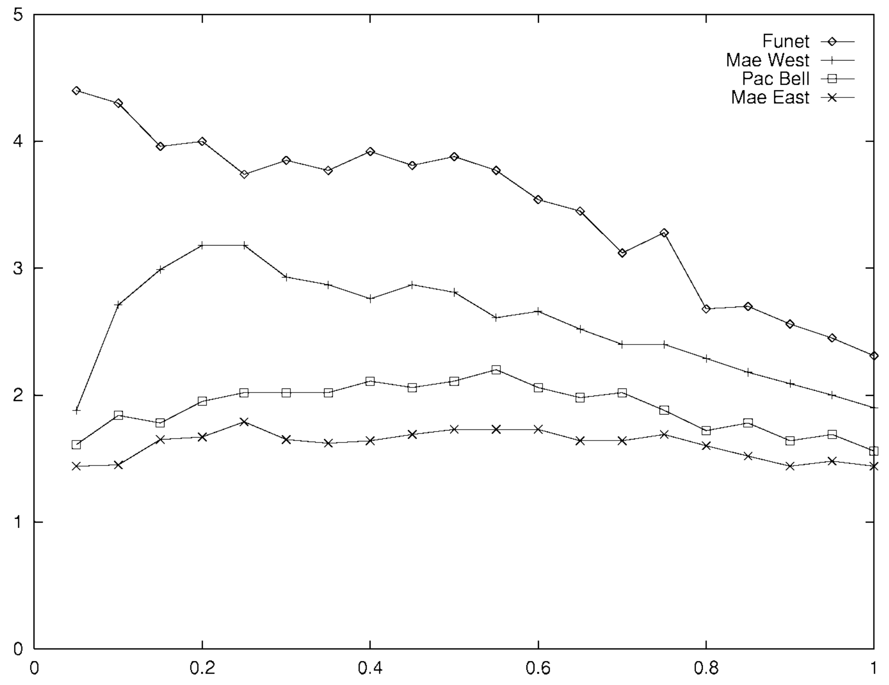

Fig. 7. Lookup speed in million lookups/s as a function of the fill factor.

TABLE II
THE EFFECT OF CHANGING THE FILL FACTOR AND BRANCH AT ROOT FOR THE FUNET ROUTING TABLE. THE SPEED IS MEASURED IN MILLION LOOKUPS/S AND THE SIZE OF THE TREE IS GIVING IN KWORDS

Fill Factor	Lookups (Sim. traffic)	Av. depth (Max depth)	Branching at root	Size of tree
No LC	1.2 (0.8)	19.5 (25)	2	80
1.00	2.5 (1.4)	8.31 (14)	4	64
0.10	4.5 (1.6)	1.81 (4)	65536	304
0.20	4.3 (1.6)	2.36 (4)	4096	159
0.30	4.2 (1.7)	2.71 (5)	256	110
0.40	4.4 (1.9)	2.93 (6)	128	86
0.50	4.3 (1.8)	3.36 (7)	64	78
0.25	4.8 (2.2)	1.73 (3)	65536 fix	171
0.50	5.0 (2.1)	1.73 (5)	65536 fix	129
0.75	4.4 (1.9)	1.82 (6)	65536 fix	118
1.00	4.3 (1.7)	2.10 (9)	65536 fix	117

In Fig. 7, the number of lookups/s is plotted as a function of the fill factor. As expected, the throughput increases as the fill factor decreases, but only to a point. Using a very small fill factor creates a large data structure that in turn puts heavier demands on the caching mechanism of the underlying machine, which normally results in slower lookups. However, using actual traffic traces from FUNET, the caching mechanism manages to handle even very large data structures with little slowdown. In fact, we get the fastest lookup times using the fill factor 0.05, even though the trie occupies 2.3 megabytes of memory.

Table II gives a more detailed picture of how the fill factor influences the data structure. Note that we achieve dramatic reductions in both average and maximal depth with only a small increase in space.

The single most important optimization is to use a large branching factor at the root. Using 16 bits for branching at the root yields a very efficient data structure, even with a fill factor of one. The main drawback is that the maximum search path is large in this case.

We have not been able to do an experimental comparison of our data structure with other state-of-the-art software implementations [5], [16], [23] since their source codes are not publicly accessible. However, an inspection of the algorithms shows that all of them have simple search routines and perform only a small number of memory accesses during a typical search operation. Hence, we believe the performance of these algorithms and ours will turn out to be quite similar. In fact, all of the algorithms start out by using a technique often referred to as bucketing. The 16 first bits or so of the search string are extracted, and this number is used as an address into the search structure. (The address space is divided into 2^{16} "buckets.") For current address spaces, this initial bucketing step alone almost solves the routing problem. This is what we do by fixing the branching at the root to 16 bits independently of the fill factor.

However, with a more crowded address space and the introduction of IPv6, the prefix-matching problem can no longer be solved by simple bucketing. The LC-trie also adapts nicely to this situation. In fact, no changes are needed to the trie structure to accommodate the 128-bit strings of IPv6. The same code can be used. Only the size of the base vector, which is accessed only once, will grow. Also, note that the theoretical bounds for the trie show that the average depth, which is $O(\log \log n)$ for a large class of distributions, does not grow as a function of the length of the strings, but only as a function of the number of strings. Furthermore, our experiments show that the lookup times for actual traffic traces are fast, even in very large routing tables. The native caching mechanisms work well. Hence, we expect the LC-trie to be a competitive data structure for the next generation of Internet routers also.

VIII. Augmented Classification

The hitherto presented structure has been aimed at classifying packets based on their destination addresses. The result of the classification is simply the next-hop information. The classification, however, could be based on more fields than the destination address, e.g., the source address, the type of service, and other protocol fields, as well as external variables.

In this section, we will show how the previous structure could be augmented to support more complex forms of classifications.

A. Link Sharing and Quality-of-Service Provisioning

Several organizations or persons may share a given link and require some form of shielding from each other's potential overload. This can be provided by packet scheduling for the link [9]. The related classification is therefore to separate packets from different organizations. This can be done based on the IP-source address. Since the size of organizations is not fixed in terms of address space, it turns out that a prefix matching is the most appropriate means of distinguishing organizations. Also, we have to deal with the problem that a short prefix is a proper prefix of another. An organization is consequently identified by a set of prefixes of various lengths.

The lookup of the source and the destination addresses may be performed in parallel or sequentially. The same structure with an LC-trie, a base vector, and a prefix vector is used for both addresses. However, the next-hop vector is indexed by two pointers, one from each lookup. This allows different routes for two source organizations for one and the same destination, as needed to distinguish traffic to separate service providers, for instance.

This procedure for classification can also be used to distinguish individual traffic flows or aggregates of flows. A flow is a partition of traffic that should receive some *a priori* established quality-of-service (QoS) in terms of delay and packet-loss probability. In the simplest form, a flow is identified by a source-destination address pair. This can be supported by the link-sharing structure. Finer classification can be made by considering the type of service field or the protocol type (TCP or UDP) and port addresses included in the packet payload. Since these are fields of fixed length, they could be reduced by an appropriate hash function. The two pointers from the source and destination address lookups, together with the hash, will index the next-hop vector. The result is a port number and a priority value or weight to be used in the packet scheduling.

The differentiated-services architecture simplifies the classification somewhat since the per-hop behavior replaces the hash of the protocol fields as a pointer into the next hop vector. The source-address lookup is not strictly necessary but simplifies traffic management since a behavior aggregate can be split up over several routes without dispersing packets in a flow over the routes. (This means that the concept of a virtual path, used in ATM for instance, could also be used for the IP to simplify traffic management.)

B. Multicast and Multipath Routing

Multicast addresses are not regular IP addresses that can be divided into network and host identifiers; a multicast address solely points out a particular multicast group. Multicast addresses may simply be included in the search trie as prefixes of a length of 32 bits. The path and level compression works equally well as the variable-length prefixes (in fact, the compression may even work better if the multicast addresses are chosen completely at random from the address range). Since they are prefix free, the pointer in the base vector to the prefix table is redundant and can be used for another purpose. There are several output ports associated with a multicast address, and we use this pointer field to index the last of the next-hop entries, starting from the pointer given in the other field of the base vector. The change to the structure is that a pointer to the prefix table is identified by the most significant bit set; when reset, it denotes a pointer into the next-hop vector instead.

Multipath routing means that packets for a particular destination may be forwarded to any one of a group of ports [14]. The particular port assigned to a given packet is given by a function that aims at distributing the load over the ports in preassigned proportions. It is analogous to multicast routing since a set of next-hop entries belong to a leaf in the trie. The difference is that only one entry is used at a time. The next-hop vector is consequently indexed by a pointer from the base vector (or occasionally from the prefix vector) and an external variable from the load-distribution function.

IX. Summary

We have demonstrated how IP-routing tables can be succinctly represented and efficiently searched by structuring them as level-compressed tries. Our data structure is perfectly general and not based on any ad hoc assumptions about the distribution of the prefix lengths in routing tables. The only assumption is that an address is a binary string. Increasing the length of this string from 32 to 128 does not affect the main data structure at all: the nodes still fit within one 32-bit machine word, and the size of the entries in the other tables simply need to be extended appropriately. Even though the data structure does not make explicit assumptions about the distribution of the address, it does adapt gracefully: path compression compacts the sparse parts of the trie, and level compression packs the dense parts.

The average depth of the trie grows very slowly. This is in accordance with theoretical results. Recall that the average depth of an LC-trie is $O(\log\log n)$ for a large class of distributions. Actually, our experiments show that in some cases the average depth is smaller for a larger table. This can be explained by the fact that a larger table might be more densely populated, and hence the level compression will be more efficient. The inner loop of the search algorithm is very tight; it contains only one addressing operation and a few very basic operations, such as shift and addition. Furthermore, the trie can be stored very compactly, using only one 32-bit machine word per node. The base vector is larger, but is only accessed once per lookup. Lookups for a core router can be executed at up to 5 million lookups/s, which corresponds to an average throughput of 10 Gbit/s.

The address-lookup system can readily be expanded to provide classification for link sharing and QoS routing, and it can incorporate multiple next-hops as needed for multicast and multipath routing. The results show that the routinely made statements about possible processing speeds for IP addresses, such as those put forward in [18], are not valid. In many cases, in [5] for instance, the trie implementations cited simply do not reflect the state of the art.

Acknowledgment

The authors wish to thank E. Kinnunen and P. Kytölaakso at the Center for Scientific Computing, Helsinki University of Technology, for providing the FUNET routing table and associated packet traces. The support of the Telecommunication Software and Multimedia Laboratory, Helsinki University of Technology, is gratefully acknowledged.

References

[1] A. Andersson and S. Nilsson, "Improved behavior of tries by adaptive branching," *Inform. Processing Lett.,* vol. 46, no. 6, pp. 295–300, 1993.

[2] ———, "Faster searching in tries and quadtrees—An analysis of level compression," in *Proc. 2nd Ann. European Symp. Algorithms,* 1994, pp. 82–93.

[3] ———, "Implementing radixsort," *ACM J. Experimental Algorithmics,* to be published.

[4] J. L. Bentley and M. D. McIlroy, "Engineering a sort function," *Software—Practice and Experience,* vol. 23, no. 11, pp. 1249–1265, 1993.

[5] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," *ACM Comput. Commun. Rev.,* vol. 27, pp. 3–14, Oct. 1997.

[6] L. Devroye, "A note on the average depth of tries," *Computing,* vol. 28, no. 4, pp. 367–371, 1992.

[7] W. Doeringer, G. Karjoth, and M. Nassehi, "Routing on longest-matching prefixes," *IEEE/ACM Trans. Networking,* vol. 4, pp. 86–97, Feb. 1996.

[8] C. Partridge, P. P. Carvey, E. Burgess, I. Castineyra, T. Clarke, L. Graham, M. Hathaway, P. Herman, A. King, S. Kohalmi, T. Ma, J. McCallen, T. Mendez, W. C. Milliken, R. Pettyjohn, J. Rokosz, J. Seeger, M. Sollins, S. Storch, B. Tober, G. D. Troxel, D. Waitzman, and S. Winterble, "A 50-Gb/s IP router," *IEEE/ACM Trans. Networking,* vol. 6, pp. 237–248, June 1998.

[9] S. Floyd and V. Jacobson, "Link-sharing and resource management models for packet networks," *IEEE/ACM Trans. Networking,* vol. 3, pp. 365–386, Aug. 1995.

[10] E. Fredkin, "Trie memory," *Commun. ACM,* vol. 3, pp. 490–500, 1960.

[11] V. Fuller, T. Li, J. Yu, and K. Varadhan, "Classless inter-domain routing (CIDR): An address assignment and aggregation strategy," RFC 1519, Sept. 1993.

[12] G. H. Gonnet and R. A. Baeza-Yates, *Handbook of Algorithms and Data Structures,* 2nd ed. Reading, MA: Addison-Wesley, 1991.

[13] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *Proc. Infocom'98,* San Francisco, CA.

[14] E. Gustafsson and G. Karlsson, "A literature survey on traffic dispersion," *IEEE Network,* vol. 11, no. 2, pp. 265–270, 1997.

[15] R. Hinden and S. Deering, "IP version 6 addressing architecture," RFC 1884, Dec. 1995.

[16] B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search," in *Proc. IEEE Infocom'98,* San Francisco, CA, 1998, pp. 1248–1256.

[17] A. Moestedt and P. Sjödin, "IP address lookup in hardware for high-speed routing," in *Proc. Hot Interconnects VI,* Stanford, Aug. 1998.

[18] P. Newman, G. Minshall, T. Lyon, and L. Huston, "IP switching and gigabit routers," *IEEE Commun. Magazine,* vol. 35, pp. 64–69, Jan. 1997.

[19] S. Nilsson and G. Karlsson, "Fast address lookup for internet routers," in *Proceedings of Broadband Communications: The Future of Telecommunications,* P. Kühn and R. Ulrich, Eds. London: Chapman & Hall, 1998, pp. 11–22.

[20] B. Rais, P. Jacquet, and W. Szpankowski, "Limiting distribution for the depth in Patricia tries," *SIAM J. Discrete Math.,* vol. 6, no. 2, pp. 197–213, 1993.

[21] K. Sklower, "A tree-based packet routing table for Berkeley Unix," in *Proc. 1991 Winter USENIX Conf.,* Dallas, TX, pp. 93–99.

[22] V. Srinivasan and G. Varghese, "Faster IP lookups using controlled prefix expansion," in *Proc. SIGMETRICS 98,* Madison, WI, pp. 1–10.

[23] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP routing lookups," *ACM Comput. Commun. Rev.,* vol. 27, pp. 25–36, Oct. 1997.

[24] C. A. Zukowski and T. Pei, "Putting routing tables into silicon," *IEEE Network,* pp. 42–50, Jan. 1992.

Stefan Nilsson received the M.Sc. and Ph.D. degrees from Lund University, Lund, Sweden.

He is currently a Lecturer in the Department of Computer Science, Royal Institute of Technology (KTH), Stockholm, Sweden. He was previously with the Helsinki University of Technology, Finland. His research interests include data structures and experimental algorithmics.

Gunnar Karlsson (S'85–M'89–SM'99) received the M.Sc. degree from Chalmers University of Technology, Gothenburg, Sweden, in 1983 and the Ph.D. degree from Columbia University, New York, in 1989.

Since March 1998, he has been a Professor in the Department of Teleinformatics, Royal Institute of Technology (KTH), Stockholm, Sweden. He previously worked at the IBM Zurich Research Laboratory, from 1989 to 1992, and at the Swedish Institute of Computer Science (SICS), from 1992 to 1998. His research interests lie within the general field of multimedia networking, with special attention to video transfer issues, quality-of-service provisioning, and switch architectures.