

# IP-Address Lookup Using LC-Tries

Stefan Nilsson  
Department of Computer Science  
P.O. Box 1100  
FIN-02015 Helsinki University of Technology  
`sni@cs.hut.fi`

and

Gunnar Karlsson  
Department of Teleinformatics  
Royal Institute of Technology  
SE-164 40 Kista  
`gk@it.kth.se`

July 2, 1998

## 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 multi-Gb/s routers for the Internet, without having to rely on multi-layer 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 version 6.

We present the basic structure, as well as an adaptive version that roughly doubles the number of lookups per second. More general classifications of packets that are needed for link sharing, quality of service provisioning and for multicast and multipath routing are also discussed. Our experimental results compare favorably with those reported previously in the research literature.

# 1 Introduction

Address lookup is one of the fundamental functions of a router, along with buffering, scheduling and switching of packets. The look-up is a particularly critical function for building routers that should support multi-Gb/s links. The lookup warrants a data structure for the organization of the routing information that can be quickly 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 internet protocol uses the longest matching prefix for determining which entry in a routing table should be used for a given packet address. This means in principle that the prefixes are compared bit by bit to the given address and that the routing information associated with the longest of the matching prefixes should be used to forward the packet. The prefixes may thereby 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 the work we present in this paper.

This paper presents the *LC-trie* as a suitable data structure for such efficient fast address look-up in software. Our implementation can process a million addresses per second on a standard 133 MHz Pentium personal computer, which is sufficient to match a Gb/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 look-ups per second.

Following this introduction, we review the procedure of address lookup for IP in Section 2. In Section 3 our solution is compared to those previously published. Section 4 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 5. 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 6 and 7. Section 8 outlines more general types of packet classifications for link sharing, flow-based routing, multicast and multipath routing. A summary of this study closes the paper.

## 2 Address Lookup for the Internet Protocol

A packet header for version 4 of the Internet Protocol 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 based on the network identifier solely. Originally the network identifier had a predetermined length, indicated by a prefix in the first address bits which specified the address class: 0 indicated class A with 8-bit network identifiers, 10 class B with 16 bits, and 110 class C with 24-bit identifiers (1110 is class D addresses which are used for multicast). This class-based structure is, however, outmoded by the introduction of the classless inter-domain routing (CIDR) [10]. 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.

Although the class-based address structure has been abolished, it is still visible in routing table entries. Figure 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, a 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 number of address look-ups per second.

Most routers have a default route which is given by a prefix of length zero and therefore matches all addresses. The default route is consequently used 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 worst cases.

Although the address structure for IP version 6 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) [14]. 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 version 6 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 which lists the next-hop addresses in use.

### 3 Related Works

There has been a remarkable interest in the organization of routing tables during the last years. The proposals include both hardware and software solutions. A hardware design is proposed by Moestedt and Sjödin which uses a pipeline structure that allows one address lookup per memory cycle [16]. Gupta, *et al.* propose a similar structure [12]. (An earlier hardware solution may be found in [23].)

Most new structures for fast address lookup are however 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 IP version 6. A main idea of their work is to quantify the prefix lengths to levels of 16, 24 and 32 bits and expanding each prefix in the table to the next higher level. Rather than expanding the prefixes to a few arbitrarily chosen levels, we use

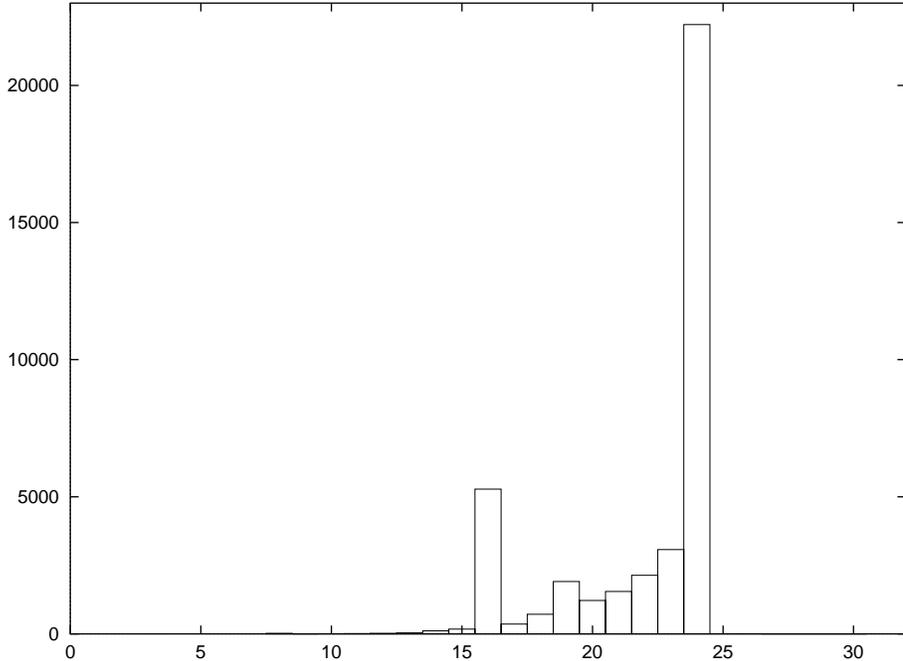


Figure 1: Distribution of prefix lengths for the Mae East core router on December 2, 1997.

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 however served as the inspiration for ours.

Srinivasan and Varghese [21] 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.* [22] 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 Figure 1). The algorithm exploits this fact and the search starts with the most probable prefix length. For the 128 bit addresses required by IP version 6 this approach may require as many as 7 hash-table lookups, each of which might in turn require several memory accesses.

Lampson, Srinivasan, and Varghese [15] present a solution based on binary search. The idea is to use two copies of each entry, one copy is padded with ones and the other one with zeroes. 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, Karjoth, and Nassehi [7] also uses a trie structure. One of their concerns is to allow fully dynamic updates. This results in a large space overhead and less than optimum performance. In fact, the nodes of the trie structure contains five pointers and one index. Finally, the

| 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 |

Figure 2: Binary strings to be stored in a trie structure.

implementation in FreeBSD is based on a path-compressed trie structure [20].

It is worth noting that our solution is fully public and source code is freely available.

## 4 Level-Compressed Tries

The *trie* [9] 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 Figure 2 correspond to the trie in Figure 3a. 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 to 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*, 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, the *skip value*, in each node that indicates how many bits that have been skipped on the path. A path-compressed binary trie is sometimes referred to as a Patricia tree [11]. The path-compressed version of the trie in Figure 3a is shown in Figure 3b. 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, 19]. 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.

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

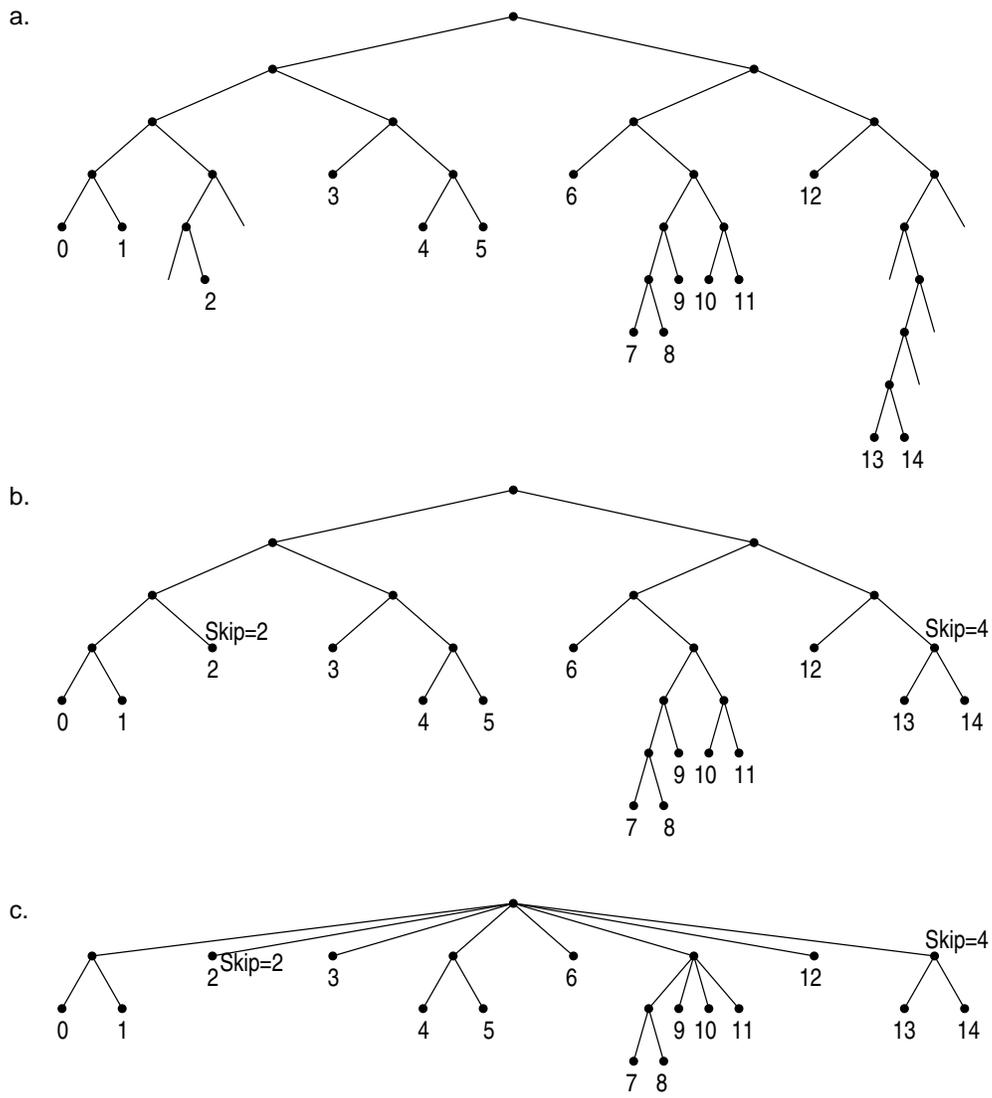


Figure 3: a. Binary trie, b. Path-compressed trie, c. LC-trie.

|    | 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       |

Figure 4: Array representation of the LC-trie in Figure 3c.

$2^i$ ; this replacement is performed recursively on each subtrie. The level-compressed version, the *LC-trie*, of the trie in Figure 3b is shown in Figure 3c.

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 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.

## 4.1 Representation

If we want to achieve the efficiency promised by these theoretical bounds, it is of course 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 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 5 bits represent the *branching factor*, the number of descendants of the node. This number is always a power of 2 and hence, using 5 bits, the maximum branching factor that can be represented is  $2^{31}$ . The next 7 bits represent the skip value. In this way, we can represent values in the range from 0 to 127, which is sufficient for IP version 6 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

```

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;

```

Figure 5: Pseudo code for find operation in an LC-trie. The trie is represented by a vector as shown in Figure 4.

largest address tables today contain about 50,000 entries). Figure 4 shows the array representation of the LC-trie Figure 3c, 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, 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 Figure 5. Let  $s$  be the string searched for and let  $\text{EXTRACT}(p, b, s)$  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 number 0. We see that the branching value is 3 and skip value is 0 and therefore we extract the first three bits from the search string. These 3 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 2 and the skip value is 0 and therefore we extract the next two bits. They have the value 2. Adding 2 to the pointer we arrive at position 15. At this node the branching value is 0, 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 5 bits of the search string with the first 5 bits of a value stored in the base vector in the position indicated by the pointer (10) in the leaf. In fact, our table (Figure 2) contains a prefix 10110 matching the string and the search was successful.

## 4.2 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 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 pseudo code in Figure 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 below), create an internal node and, finally, the procedure is called recursively to build the subtrees.

To compute the skip value, 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 4 prefixes of length two are present (there are at least two branches, since the number of strings are at least two and the entries are unique). If these prefixes are present we continue the search, by examining if all 8 prefixes of length 3 are present. Continuing this search we will eventually find a prefix that is missing and the branching factor has been 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 pseudo code 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 caching 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 at 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.

### 4.3 Further Optimizations

In this section we present a simple optimization that reduces the depth of the trie drastically. The idea is to use a weaker criterion for computing the branching factor. As presented above 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 two strings, and hence we always use the branching factor 2 in this case.)

```

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;
    }
}

```

Figure 6: Pseudo code 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 is disregarded. The trie is represented by a vector (**trie**) and the first free position in this vector is given by **pos**.

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.

## 5 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-hops 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 IP version 6. Each entry also contains two pointers: one pointer into the next-hop table and one pointer into the prefix table. The search routine follow 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 need not 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 2 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 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 addition 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 look-up: 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 IP version 6; 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 gain was thought too insignificant. Most would yield small savings in memory which, as our results show, is not greatly influencing the execution speed. However, we briefly describe these ideas here 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 pointers values in 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 account 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.

## 6 Experiments

Source code and data is available at URL <http://www.cs.hut.fi/~sni>. The measurements were performed on two different machines: a SUN Ultra Sparc II with two 296-MHz processors and 512 MB of RAM, and a personal computer with a 133-MHz Pentium processor and 32 MB 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 (URL <http://www.merit.edu/ipma>). We have used the routing tables for Mae East and Mae West from the 30th of October, 1997, and AADS and Pac Bell from the 24th of August, 1997. We did not have access to the actual traffic being routed according to these tables and therefore the traffic is

| Site     | Routing<br>Entries | Next-<br>Hops | Number of Entries |        |        | Av. depth<br>(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 |

Table 1: Results for a structure with 16 bits used for branching at root and a fill factor of 0.5. The speed is measured in million lookups per second.

simulated: We simply use a random permutations of all entries in a routing table. The entries were extended to 32 bits numbers by adding zeroes (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 the Finnish University and Research Network (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.

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.

## 7 Discussion of Results

Table 1 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 per second. 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 per second 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 look-up operations per second on a 133 MHz Pentium personal computer, and more than 2 million per second on a more powerful SUN Sparc Ultra II workstation. These numbers pertain to the US core routers and randomly generated traffic with no dependencies in the destination addresses.

It is interesting to note that using actual traffic traces from FUNET (the Finnish University and Research Network) the look-up time is about twice as fast, with 5 million lookups per second 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

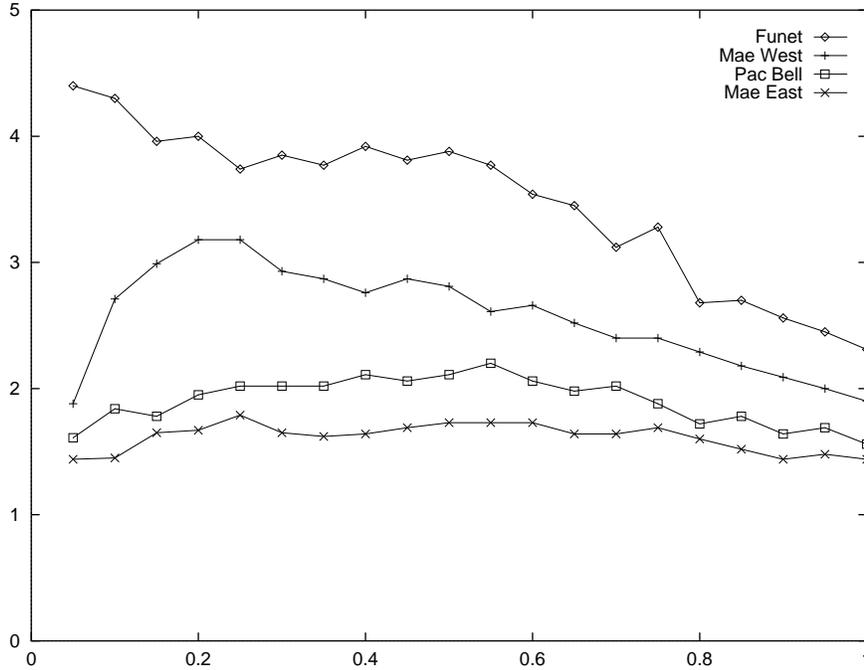


Figure 7: Look-up speed in million lookups per second as a function of the fill factor.

structure. In our experiments the maximum depth of the LC-trie is at most 5, while the average depth is just below 2.

In Figure 7 the number of lookups per second 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 which, 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 2 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 fill factor 1. The main drawback being 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, 15, 22], since their source codes are not publically 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 that the performance of these algorithms and that of 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

| 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          |

Table 2: The effect of changing the fill factor and branch at root for the FUNET routing table. The speed is measured in million lookups per second and the size of the tree is given in kwords.

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 IP version 6 the prefix matching problem can no longer be solved by simple bucketing. The LC-trie adapts nicely also to this situation. In fact, no changes are needed to the trie structure to accommodate the 128 bit strings of IP version 6. 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 is 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 also for the next generation of Internet routers.

## 8 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 could, however, be based on more fields than the destination address: for instance, 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.

### 8.1 Link sharing and quality of service provisioning

Several organizations or persons may share a given link and may require some form of shielding from each others potential overload. This can be provided by packet scheduling for the link [8]. 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, in terms of address space, is not fixed, 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. The next-hop vector is however indexed by two pointer, one from each lookup. This allows different routes for two source organizations for one and the same destination, as needed for instance to distinguish traffic to separate service providers.

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 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 for instance in the asynchronous transfer mode, could be used also for the internet protocol to simplify traffic management.)

## 8.2 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 length 32 bits. The path and level compression works equally well as for 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 [13]. The particular port assigned to a given packet is given by a function that aims at distributing the load over the ports in pre-assigned 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.

## 9 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 look-up. Lookups for a core router can be executed at up to 5 million lookups per second which corresponds to an average throughput of 10 Gb/s.

The address lookup system can readily be expanded to provide classification for link sharing and quality of service 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 [17], are not valid. In many cases, for instance in [5], the trie implementations cited simply do not reflect the state of the art.

## Acknowledgment

This work has in parts been presented at the IFIP Broadband Communication conference, Stuttgart, Germany, April 1998 [18].

We thank Erja Kinnunen and Pekka Kytöläakso of the Center for Scientific Computing at Helsinki University of Technology for providing the FUNET routing table and associated packet traces.

This research was done when G. Karlsson was visiting professor at the Telecommunication Software and Multimedia Laboratory at the Helsinki University of Technology. This support is gratefully acknowledged.

## References

- [1] A. Andersson and S. Nilsson. Improved behaviour of tries by adaptive branching. *Information Processing Letters*, 46(6):295–300, 1993.
- [2] A. Andersson and S. Nilsson. Faster searching in tries and quadtrees – an analysis of level compression. In *Proceedings of the Second Annual European Symposium on Algorithms*, pages 82–93, 1994. LNCS 855.
- [3] A. Andersson and S. Nilsson. Implementing radixsort. *The ACM Journal of Experimental Algorithmics*, 1998. to appear.

- [4] J. L. Bentley and M. D. McIlroy. Engineering a sort function. *Software – Practice and Experience*, 23(11):1249–1265, 1993.
- [5] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. *ACM Computer Communication Review*, 27(4):3–14, October 1997.
- [6] L. Devroye. A note on the average depth of tries. *Computing*, 28(4):367–371, 1982.
- [7] W. Doeringer, G. Karjoth, and M. Nassehi. Routing on longest-matching prefixes. *IEEE/ACM Transactions on Networking*, 4(1):86–97, February 1996.
- [8] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4):365–386, August 1995.
- [9] E. Fredkin. Trie memory. *Communications of the ACM*, 3:490–500, 1960.
- [10] V. Fuller, T. Li, J. Yu, and K. Varadhan. Classless inter-domain routing (CIDR): an address assignment and aggregation strategy. Request for Comments: 1519, September 1993.
- [11] G. H. Gonnet and R. A. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, second edition, 1991.
- [12] P. Gupta, S. Lin, and N. McKeown. Routing lookups in hardware at memory access speeds. In *Proceedings of Infocom’98, San Francisco*, April 1998.
- [13] E. Gustafsson and G. Karlsson. A literature survey on traffic dispersion. *IEEE Network*, 11(2):265–270, 1997.
- [14] R. Hinden and S. Deering. IP version 6 addressing architecture. Request for Comments: 1884, December 1995.
- [15] B. Lampson, V. Srinivasan, and G. Varghese. IP lookups using multiway and multicolumn search. In *Proceedings of IEEE Infocom’98, San Fransisco*, 1998.
- [16] A. Moestedt and P. Sjödin. IP address lookup in hardware for high-speed routing. In *Proceedings of Hot Interconnects VI, Stanford*, August 1998.
- [17] P. Newman, G. Minshall, T. Lyon, and L. Huston. IP switching and gigabit routers. *IEEE Communications Magazine*, 35(1):64–69, January 1997.
- [18] S. Nilsson and G. Karlsson. Fast address lookup for internet routers. In P. Kühn and R. Ulrich, editors, *Proceedings of Broadband Communications: The Future of Telecommunications*, pages 11–22. Chapman & Hall, 1998.
- [19] B. Rais, P. Jacquet, and W. Szpankowski. Limiting distribution for the depth in Patricia tries. *SIAM Journal on Discrete Mathematics*, 6(2):197–213, 1993.
- [20] K. Sklower. A tree-based packet routing table for Berkeley Unix. In *Proceedings of the 1991 Winter USENIX Conference, Dallas*, pages 93–99, 1991.
- [21] V. Srinivasan and G. Varghese. Faster IP lookups using controlled prefix expansion. *Proceedings of SIGMETRICS 98, Madison*, pages 1–10, 1998.
- [22] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed IP routing lookups. *ACM Computer Communication Review*, 27(4):25–36, October 1997.

- [23] C. A. Zukowski and T. Pei. Putting routing tables into silicon. *IEEE Network*, pages 42–50, January 1992.