IP lookup with low memory requirement and fast update

Berger, Michael Stübert

Published in:
Workshop on High Performance Switching and Routing, 2003, HPSR.

Link to article, DOI:
10.1109/HPSR.2003.1226720

Publication date:
2003

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):
Berger, M. S. (2003). IP lookup with low memory requirement and fast update. In Workshop on High Performance Switching and Routing, 2003, HPSR. IEEE. DOI: 10.1109/HPSR.2003.1226720

IP lookup with low memory requirement and fast update

Michael Berger
COM, Technical University Of Denmark
Building 345v, DK-2800 Kgs. Lyngby, Denmark
Phone: +45 45 25 38 53 Fax: +45 45 93 65 81
E-mail: msb@com.dtu.dk

Abstract—This paper presents an IP address lookup algorithm with low memory requirement and fast updates. The scheme, which is denoted prefix-tree, uses a combination of a trie and a tree search, which is efficient in memory usage because the tree contains exactly one node for each prefix in the routing table. The time complexity for update operations is low for prefix-tree. The lookup operation for the basic binary prefix-tree may require that a high number of nodes be traversed. This paper presents improvements to decrease lookup time, including shortcut tables and multi-bit trees. The prefix-tree is compared to a trie and a path compressed trie using prefixes from a real routing table.

Index Terms—Longest prefix match, IP lookup, trie, tree

A. Introduction

IP lookup algorithms have been studied extensively in the literature. The introduction of Classless Inter Domain Routing (CIDR) has reduced the size of forwarding tables, but the lookup procedure is more complex because exact matching is replaced by longest prefix matching [2][3].

A trie structure [4] is a convenient way to represent the prefixes in the forwarding table. The lookup and update procedures are simple, but the lookup procedure may traverse up to B nodes, if B is the number of address bits. A trie will contain a high number of intermediate nodes that does not contain any prefixes. Removing any intermediate nodes with only one child node can reduce the number of intermediate nodes. The resulting trie is called a path compressed trie or Patricia trie. Another approach is Level Compressed (LC) tries [5]. They can increase lookup speed with the cost of a more time-consuming update procedure. The time complexity of the update procedure is important because frequent routing updates may occur [6].

The lookup speed can be improved by having balanced trees, e.g. range search trees. If N is the number of prefixes, the time complexity of lookup is O(log(N)). The main drawback is the update time; to keep the tree balanced it is usually necessary to re-construct the whole tree.

This paper presents a new tree structure for storing the IP forwarding table. It is based on an exact match VPI/VCI search algorithm developed in [1]. Minor modifications have been introduced to support variable length prefixes. The prefix-tree structure uses a combination between a trie and a tree; in each node a comparison is performed similar to a tree. Branches are performed as in tries based on the address bits. The algorithm is described further in section B. Further improvements that can speed up the lookup operation is given in section C. The performance of the proposed algorithms is compared to trie based approaches in section D. Finally, in section E, concluding remarks are given.

B. Algorithm

The IP lookup algorithm organizes the IP prefixes in a tree structure. Each node in the tree contains exactly one prefix, so the size of the tree is equal to the size of the routing table. Consider the routing table given in Table 1. It contains seven different prefixes belonging to seven different routes.

Table 1: Routing table example

Prefix	Route
01*	R1
11*	R2
0*	R3
001*	R4
10*	R5
1000*	R6
1010*	R7

Figure 1 shows the tree structure for the routing table given in Table 1. Each node contains a prefix and two pointers pointing to successive tree nodes. The prefix size must be greater than or equal to the level where the prefix is located. E.g. the prefix size of '11*' is two, and the level is 1. in Figure 1. The remainder of this section explains the search, insert and delete operations by examples followed by a pseudo code description.

1) *Lookup operation*

The example below explains the lookup operation. Assume that "10101010" is used as input to the lookup tree. The lookup procedure works as follows: The first step is to compare the address with the prefix in the root node. The root node prefix does not match the address, so the result of the comparison is not stored. The first bit in the address is then used to determine the next node in the tree. The first bit is a '1' so the next visited node is '11*'. Again, there is no match between the prefix '11*' and the address "10101010". The search procedure is continued using the

0-7803-7710-9/03/$17.00 © 2003 IEEE.

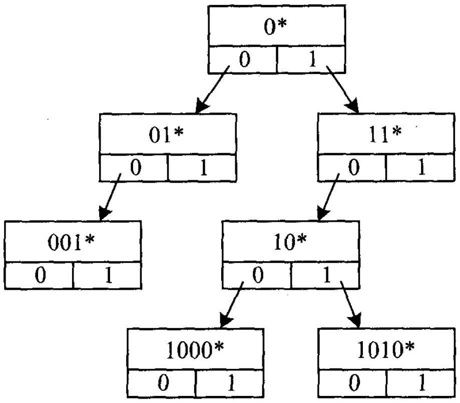

Figure 1: Prefix-tree

next bit in the address, which is a '0'. This leads to the tree node containing prefix '10*'. Now there is a match between the prefix and the address. Since this is the longest prefix match until now, the R5 route is stored. The next address bit is '1' which implies that the next visited node is '1010*'. Again there is a match between the address and the prefix. This is the longest prefix match until now so the route R7 is stored and replaces R5. Since '1010*' is a leaf node, the lookup operation has finished. The longest prefix match is route R7.

The time complexity of the lookup operation is O(B) where B is the number of address bits. The pseudo code for the lookup operation is shown below. **Ref** means that the variable is called by reference. 'pmax' is initially set to 0, and a match is found if 'pmax' is larger than 0 when the call returns. The procedure call is as follows: *lookup* (addr, 0, root, &pmax, &route)

```
lookup(addr,level,node,ref pmax,ref route) {

 if (node == NULL)
  return

 p = prefix_match_size(addr,prefix(node))

 if (p > pmax) {
  pmax = p
  route = route(node)
 }

 if (addr[level] == '1')
  lookup(addr,level+1,right(node),pmax,route)
 else
  lookup(addr,level+1,left(node),pmax,route)
}
```

2) Insert Operation

Now the insert procedure will be illustrated by an example. The resulting tree is shown in Figure 2. It is assumed that a new route R8 with prefix '1*' will be added to the tree. The first step is to examine the root node. The level of the root node is 0, which is smaller than the prefix length of 1. for route R8. The first bit of the new prefix is used to determine the next node in tree, which is '11*'. Now the level is 1., which is equal to the prefix length of R8. The new node must then be inserted at this location since the level in the tree must never be higher than the prefix length. The second prefix bit of the old prefix '11*' is 1., and the old node is therefore inserted to the right of '1*'.

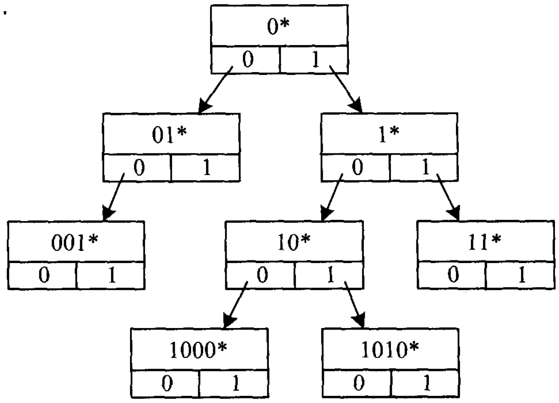

Figure 2: Prefix-tree after addition of '1*' prefix.

The time complexity of the insert procedure is O(B). The pseudo code is shown below. The procedure call is as follows: *insert*(prefix,route,0,&root).

```
insert(prefix,route,level,ref node) {

 if (node == NULL) {
  node = new node(prefix,route)
  return
 }

 if (length(prefix) == level) {

  p1 = prefix(node)
  r1 = route(node)
  prefix(node) = prefix
  route(node) = route

  if (p1[level] == '1')
   insert(p1,r1,level+1,right(node))
  else
   insert(p1,r1,level+1,left(node))

 }
 else {
  if (prefix[level] == '1')
   insert(prefix,route,level+1,right(node))
  else
   insert(prefix,route,level+1,left(node))
 }
}
```

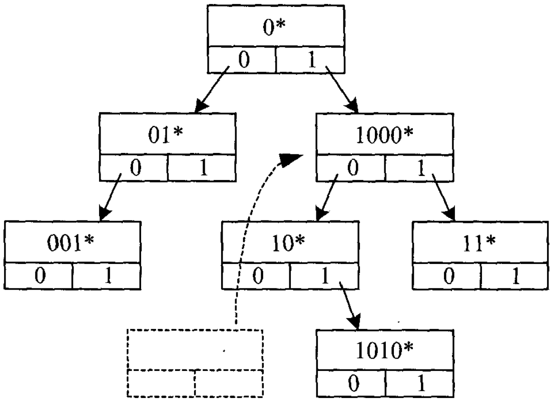

Figure 3: Prefix-tree after removal of '1*' prefix.

3) *Delete Operation*

The following example shows how a node is removed. The remove operation is shown in Figure 3. The selected node is '1*' which was just inserted in the previous example. A node without child nodes can easily be removed. If the node has one or two child nodes it is necessary to reconstruct the tree. This can be done by finding a leaf node in one of the child trees, and then insert this node at the location, which was just removed. The selected leaf node is '1000*' which is now inserted as the right child of the root node.

The time complexity of the delete procedure is O(B). The pseudo code is shown below. The procedure call is as follows: *delete*(prefix,&root,0).

```
delete(prefix,ref node,level) {

  if (prefix == prefix(node)) {

    leaf_node = find_leaf_node(node)
    if (leaf_node == NULL) {
      delete(node)
      node = NULL
      return
    }
    else {
      right(leaf_node) = right(node)
      left(leaf_node) = left(node)
      delete(node)
      node = leaf_node
      return
    }
  }
  else {
    if (prefix[level] = '1')
      delete(prefix,right(node),level+1)
    else
      delete(prefix,left(node),level+1)
  }
}
```

C. Improvements

The following section describes a number of improvements that can speed up the lookup operation. The first approach shown in Figure 4 uses an additional level field in each tree node. The lèvel value determines the next bit in the address that is used for searching. A similar technique is utilized for path-compressed tries.

Figure 4a shows four prefixes stored in a prefix-tree. The resulting level-tree is shown in Figure 4b. Note that the root node has level = 2 which means that bit number 2 (third bit from the left) in the address determines the child node. In this example the level-tree has fewer levels (3) compared to the prefix-tree (4) however, in general the worst-case tree height is not reduced by the level-tree.

Introducing a lookup table that provides shortcuts into the tree can reduce the maximum number of levels. This is suggested for tries in [7], but can be exploited for prefix-trees as well. Assume that the number of bits in the address is B. The basic tree in Figure 1 has at maximum B levels. An example is shown in Figure 5 with B = 8 bits.

The first B/2 bits of the address are used as index to the lookup table. If the entry contains a valid pointer, it will point to the sub-tree that will be traversed. If the entry does not contain any valid pointer, then the basic tree will be used. The basic tree contains prefixes between 0 and B/2. With the introduction of a shortcut table, the maximum search depth is reduced to B/2. IPv4 prefixes requires a lookup table of size 16 bits giving 65536 locations.

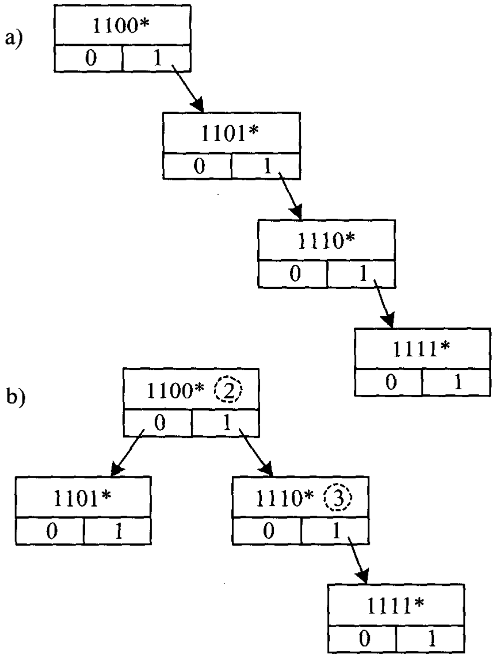

Figure 4: a) Prefix-tree, b) Level-tree

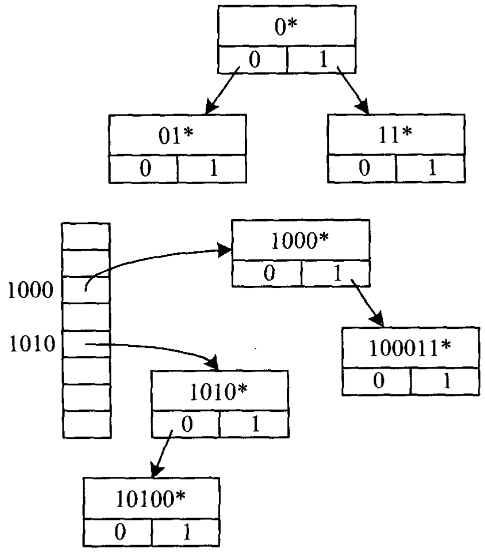

Figure 5: Prefix-tree with shortcut table

The lookup time can be further improved by increasing the number of lookup tables. In case of Ipv4 the address space of 32 bits can be subdivided into 4 regions with 8 bit in each. Having three lookup tables, the first table will examine the first 8 address bits, the second table will use the first 16 bits and the third table will use the first 24 bits. The maximum depth of each sub-tree will then become 8 levels. The main drawback is the size of the third lookup table with 24 address bits. However, the number of prefixes with length above 24 is very limited according to Table 2, and it is therefore more efficient to store the prefixes in a CAM (Content Addressable Memory).

Search speed can be further improved by increasing the number of leaf nodes. With four leaf nodes it is necessary to examine two address bits to find the next node in the tree. Each node must now store more than one prefix. E.g. the prefixes '*', '0*' and '1*' must be stored in the same node because nodes at a higher level will contain at least two bits more in the prefixes. Figure 6a shows a new node. The size of the new node is at least twice as big because it contains 3 prefixes and 4 pointers. If he node is too big for a single memory line, it can easily be split into two consecutive memory lines. The content of the two memory lines is shown in Figure 6b. The first bit among the two bits used for searching is then used to determine the memory line. Note that the prefix '*' must be repeated in both lines because it matches addresses starting with both '1' and '0'.

A multi-bit node may contain between 1 and 3 prefixes, so the total number of prefixes that can be stored is not directly given by the total amount of memory measured in number of nodes, which is the case for the basic prefix tree. The worst case can however be determined. It occurs when the tree is balanced and when all leaf nodes contain only one prefix. Assume that the highest level in the tree contains l nodes, and that the total number of nodes above that level is m. It can be shown that $m=3l+1$ for a four child node. The total number of prefixes is $p=3l+m$, and the total number of nodes is $n=l+m$, thus $p=2.5n-0.5$. The smallest number of prefixes that always can be stored for a given memory space n is thus the integer part of $2.5n-0.5$.

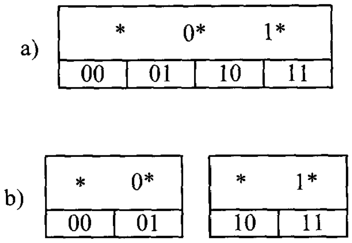

Figure 6: Multi-bit node

Using both three shortcut-lookup tables and larger 2 level nodes the lookup operation can be performed in 4-5 memory cycles. Using synchronous SRAM with 10 ns access time, the lookup time will be in the range 40-50 ns. This is sufficient for 10 Gigabit Ethernet with minimum packet duration of 51.2 ns.

D. PERFORMANCE

This section compares the performance in terms of tree size of the prefix tree and level tree with a basic trie and a path compressed trie (patricia). The routing table is from the Mae-West router 03/15/02 [8]. It contains 29587 prefixes. Table 2 shows the number of prefixes at each level of the trie/tree. The prefix distribution is depicted graphically in Figure 7.

Table 2: Prefix distribution

LEVEL	TRIE	PATRICIA	PREFIX TREE	LEVEL TREE
0	0	0	1	1
1	0	0	2	2
2	0	0	4	4
3	0	1	7	7
4	0	0	11	14
5	0	1	20	28
6	0	9	32	53
7	0	15	52	95
8	10	19	92	162
9	3	40	165	286
10	311	49	295	526
11	4	114	535	958
12	14	377	917	1575
13	37	930	1403	2366
14	68	1735	1902	3129
15	128	2568	2430	3801

16	2339	3262	2932	4234
17	531	4030	3424	4175
18	887	4397	3638	3501
19	2248	4309	3426	2550
20	2041	3487	2893	1451
21	1399	2568	2324	552
22	1971	1334	1679	106
23	2253	337	1057	11
24	15627	4	346	0
25	15	1	0	0
26	2	0	0	0
27	2	0	0	0
28	1	0	0	0
29	1	0	0	0
30	2	0	0	0
31	0	0	0	0
32	1	0	0	0

Table 3 shows the total number of nodes and the number of dummy nodes. A dummy node is an intermediate node that does not contain any routing information. The average and maximum height does not differ much for the patricia trie and prefix tree, however the number of nodes is almost twice as high for the patricia trie. Note that the Level-tree gives a small reduction in the average and maximum height compared to the prefix-tree. However, it is expected that the advantage of level-tree is larger for a more hierarchically organised prefix database e.g. in combination with IPv6.

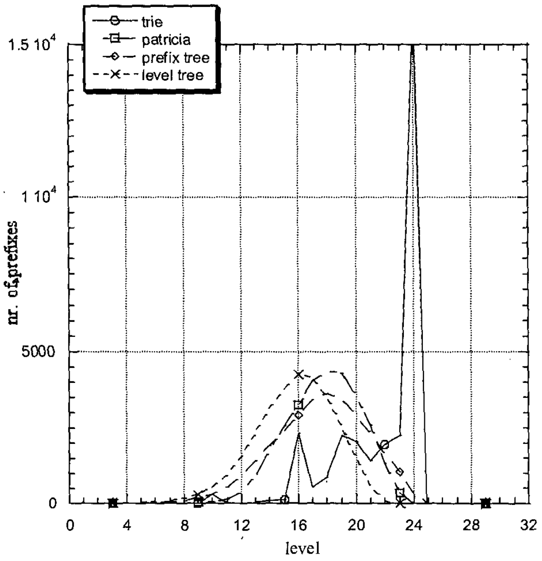

Figure 7: Prefix distribution

Table 3

NODE	TRIE	PATRICIA	PREFIX TREE	LEVEL TREE
Total	111747	56295	29587	29587
Dummy	82160	26708	0	0
Av. height	22.0	17.7	17.4	15.7
Max height	32	25	24	23

E. CONCLUSION .

Trie-based IP lookup schemes have several advantages including simple procedures for update and lookup. The time complexity of the lookup operation is however proportional to the number of address-bits so a basic trie structure is not scalable to gigabit speed. Using multi-bit tries and shortcut lookup tables can increase the lookup speed. In terms of memory usage, the main drawback of the trie approach is the high number of intermediate nodes. Even for the patricia trie the number of intermediate nodes is almost as high as the number of entries in the forwarding table.

The proposed prefix-tree overcomes this drawback by combining a tree and a trie such that each node contains exactly one prefix. The memory requirement for the prefix-tree is therefore exactly given by the size of the forwarding table. This paper also proposes the level-tree that has slightly better performance than the prefix tree, but in order to increase the lookup speed significantly, shortcut lookup tables and multi-bit nodes can be introduced. Using both three shortcut-lookup tables and larger two-bit nodes the lookup operation can be performed in 4-5 memory cycles.

F. REFERENCES

[1] Andreas Magnussen, "ATM Switching Systems", *COM, Technical University of Denmark,* Ph.D Thesis.

[2] M. Á. Ruiz-Sánches, E. W. Biersack, W.Dabbous, "Survey and Taxonomy of IP Address Lookup Algorithms", *IEEE Network,* March/April 2001

[3] M. Zitterbart, "High-performance routing-table lookup", *Phil. Trans.,* Royal Soc. London A (2000) 258 p 2217-2231.

[4] R. Sedgewick, "Algorithms in C++", *Addison Wesley,* 1990

[5] S. Nilsson, G. Karlsson, "IP-address Lookup Using LC-tries", *IEEE JSAC,* June 1999, vol. 17 no.6, p 1083-92

[6] C. Labovitz, G. malan, F.Jahanian, "Internet routing instability", *ACM SIGCOMM '97,* September 1997

[7] M. Uga, K. Shiomoto, "A Fast and Compact Longest Prefix Lookup Method using Pointer Cache for very long Network Address", *proc. IEEE ICCCN 1999,* Boston MA, Oct 1999.

[8] Mae-West routing database, "The Internet performance Measurement and Analysis (IPMA) project", *http: //www.merit.edu/ipma/routing_table/,* 10 Oct. 2002