Flexible IP Lookup Algorithm with Fast Update*

SUMMARY Many algorithms have been introduced to obtain giga-bit routing performance by reducing searching time. As most of them, however, have not considered the importance of update time and memory requirement seriously, they couldn't work well in real networks. We propose a flexible and fast IP lookup algorithm, named FFILA, considering these factors and compare the performance of our scheme with that of the conventional scheme of Patricia trie.

key words: routing table lookup algorithm, longest prefix matching algorithm, routing table update

1. Introduction

Many algorithms that show improved performance over Patricia trie have been introduced [2]. However, most of them show poor performance in terms of updating time and require large routing tables. Backbone routers, in these days, should be able to handle 2 million searches per second to have 1 Gbps performance. Additionally they need to process many updating packets when the network is in the unstable node. Therefore it is very important for backbone routers to update their tables so fast, at least one thousand updates per second. In addition to fast search and updating, routers need to maintain the routing table as small as possible. Backbone routers normally have tens of thousand routing entries, and some have more than 300,000 entries. So, to achieve good scalability, it is crucial for an algorithm to consume small memory to maintain and handle its routing table.

2. Proposed Algorithm

Giga-bit IP table lookup algorithms can be classified as three categories; level compression based [4], caching based [5], and hash based [1]. These algorithms search their tables a few times faster than Patricia trie does. However, they fail to update the lookup table faster than Patricia trie, and sometimes show even slower performance. Hash based algorithms usually require very large routing table image, which causes a serious problem in scalability.

We introduce a new IP lookup algorithm that can show giga-bit performance. Routers working correctly in real networks should be able to update a routing entry at least in 1 msec. In addition they should have memory requirement as small as possible to achieve scalability. Our proposed algorithm can search and update an entry so fast by small memory. It uses the properties of IP address classes to improve the performance.

2.1 Hash

Hash can search faster than any other lookup algorithms in exact prefix matching. An IP address consists of two parts, i.e., netmaskid and hostid. As the length of a netmask id is more than or equal to 8 bits, we can take advantage of the 8-bit hash by using it at the header of Patricia trie without having duplicated pointers.

2.2 Patricia trie without Backtracking

Backtracking occurs during the search because Patricia trie can insert bit masks such as 0xff00ffff that are not used in real networks. As this deteriorates the Patricia trie performance very much, we can obtain remarkable performance improvement by avoiding it. We can simply avoid backtracking by removing the support of unnecessary bit masks in Patricia trie that compares particular bits for searching. Our modified Patricia trie tries to compare all the bits in a given range, like from bit 0 to the bit where the original Patricia trie takes comparison.

3. Modified Patricia trie with Small Routing Table

An entry in FreeBSD version of Patricia trie consists of a radix node and a radix mask node. The radix node is a node structure of Patricia trie and the radix mask node is a structure containing bit masking information. The sizes of the radix node and the mask node are 24 bytes and 16 bytes, respectively. The larger the node size is, the more memory and time the algorithm consumes in allocating and de-allocating the resource. So we redesign Patricia trie by compacting the node structure, that is, the radix mask node is removed while the radix node has its own mask. Bit masks and other information for the bit test are removed, too. It is possible to get bit masks and other information by using the value that represents the bit position for the test because the original radix node and the radix mask node have variables that will be removed or derived by other variables. For example, the bit mask of Patricia trie is not the bit mask of IP address but the bit mask for socket address structure. It needs only 5 bits and 8 bytes for the bit masks of IP address and socket.

---

*Manuscript received September 22, 2003.
*The author is with Samsung SECUI.COM Corporation Laboratory, Korea.
††The author is with the School of Computer and Electrical Engineering, Seoul National University, Seoul, Korea.
*Part of this paper was presented in IEEE ICC, Helsinki, 2001.
struct radix_node {
    struct radix_mask *m_mask;
    struct radix_node *m_p;
    short m_b;
    char m_bmask;
    u_char m_flags;
    union {
        struct {
            caddr_t m_Key;
            caddr_t m_Mask;
            struct radix_node *m_Dupedkey;
            }m_leaf;
        struct {
            int m_Off;
            struct radix_node *m_L;
            struct radix_node *m_R;
            }m_node;
        }m_w;
    };
};

struct improved_radix_node {
    struct radix_node *parent;
    unsigned int maskedkey;
    char comparebit;
    struct radix_node *left;
    struct radix_node *right;
    };

Fig. 1 The original radix node (left) and our proposed node structures.

address structure, respectively. Accordingly we can reduce
the size of the radix node to 17 bytes without using the radix
mask node by imbedding the bit mask information into the
radix node. Figure 1 shows the original radix node and our
new radix node structures.

An insertion in Patricia trie requires two Patricia nodes,
named innernode and leafnode. Innernode is a node to
make path to the leafnode and it has node route information.
Leafnode is a node contains route information. However
sometimes an insertion of innernode is not necessary if
a node is inserted at a leaf of Patricia trie. To explain
this feature we consider an example. Let's assume Innernode A
has two leafnodes C and D. If we try to add leafnode E next
to leafnode D, we should add new innernode F, and then
D and E should be the children of innernode F because the
innernode can't have route information for original Patricia
trie. It means we need two nodes to insert a leafnode. However
if we try to add node E next to node D, our algorithm
simply adds E as a child of D. So we need only one more
node because we use only one type of radix node which can
have route information regardless of whether it is used as an
innernode or not.

In this way, we can make Patricia trie require less mem-
ory. For example, let's assume that Patricia trie has a form
of a balanced binary tree with full nodes of which depth is
k. The modified Patricia trie can have $3 \cdot 2^{k-1} - 1$ nodes at
best whereas the original one has $2^{k-1} - 1$ nodes. So we can
reduce the number of nodes by 25% as their ratio is 3/4.

4. Simulation and Analysis

We tested the original Patricia trie code of FreeBSD 4.0 and
our algorithm. The routing table entries were generated by
routing the table dump, MAE-EST (exchange point operated
by MCI WorldCom). It was taken on June 21, 2000 [3] and
there were 34,765 entries in the routing table.

4.1 Routing Table Size

Figure 2 shows tendency of the linear increase in the rout-
ing table sizes in the two algorithms. Our proposed algo-
rithm consumes memory of 33% less than Patricia trie on
the average. This benefit mainly comes from the node size
difference, as our algorithm uses the reduced node size of
17 bytes, resulting in 30% smaller than that of Patricia trie.
As our algorithm requires a smaller number of nodes than
Patricia trie, its table size requirement naturally becomes
smaller. Lastly our algorithm has the advantage of not using
any radix mask node whereas Patricia trie needs additional
ones.

4.2 Search Speed

The search algorithm is a critical component in router algo-
rithms because it heavily affects overall router performance.
To support the giga-bit performance, routers should be able
to handle at least 2 \cdot 10^9 packets per second. Figure 3 shows
that our algorithm runs nearly 3 times faster than Patricia
trie. The average time to search an entry is 0.45 msec given
that the number of entries is 34765. This algorithm can show
at least throughput of 1.1 Gbps with general computer archi-
ecture. If it is implemented on top of a dedicated hardware
for optimal performance, it can produce higher throughput.

4.3 Table Updating Speed

We compared the insertion speed of the two algorithms by
inserting all entries in the dump data into the empty routing
table. We measured the insertion time according to the num-

ber of entries as shown in Fig. 4(a). Our algorithm is faster by 36% than Patricia trie. We tested the deletion speed of the two algorithms by removing all these entries from the routing table. We measured the deletion time in accordance with the number of entries. Our algorithm shows the performance of 34% faster than Patricia trie in terms of the speed of deleting all the entries.

This is mainly because the search algorithm of FFILA runs faster than that of Patricia trie. The other reason is that our algorithm uses 30% smaller sized node than Patricia trie, resulting in less time requirement for memory allocation and initialization.

5. Conclusion

Our proposed FFILA can search nearly 3 times faster than Patricia trie, which is fast enough to achieve gigabit performance. It also updates the routing table faster and consumes less memory than Patricia trie algorithm does. Because of its fast update and small memory requirement, it would be a good choice to combine FFILA with some other implementation techniques such as B-tree and a node pool. As we didn’t assume any platform specific technique like caching for better performance, further improvement can be expected by adopting such a technique.

Acknowledgments

This research was supported by University IT Research Center Project.

References