Flexible and Fast IP Lookup Algorithm

Wooguil Pak
Secni.com Corporation
Yeoksam-dong, Kangnam-gu
Seoul, Korea 135-080
TEL:+82-2-3458-1972
FAX:+82-2-3458-1999
wgpak@secni.com

Saewoong Bahk
School of Electrical Engineering
Seoul National University
Seoul, Korea 151-742
TEL:+82-2-880-8434
FAX:+82-2-880-8198
ssbahk@netlab.snu.ac.kr

Abstract — In this paper, we introduce a fast IP table lookup algorithm that improves table updating time as well as IP address searching time. Because routers with Patricia trie can’t support gigabit performance, many algorithms to support gigabit routing performance by reducing searching time have been introduced. Most of them, however, did not considerably count the importance of updating time. As a network often falls into unstable states, a router may generate and receive hundreds of update request messages per second. So the router should be able to update its routing table at least 1000 times per second to appropriately run in real networks.

We consider updating time as much important factor as searching time in proposing a flexible and fast IP lookup algorithm (FFILA) in this paper. Our scheme searches the table about 3 times faster than Patricia trie. It also shows improved performance in updating time by at least 30% when compared with Patricia trie. Also as many backbone routers today have over 100,000 routing table entries and its number is still increasing due to the growth in the network size, the memory requirement for the lookup algorithm becomes more important. An additional advantage of our algorithm is in its small memory requirement, which is good to overcome the scalability problem.

I. INTRODUCTION

Recently the Internet traffic has been doubled every three months. As the network traffic increases, the network becomes faster, and its capacity is increased. Today, an hot issue in the Internet is fast IP table lookup in backbone routers. As conventional routers use the longest prefix matching algorithm, they are too slow to support gigabit performance. The longest prefix matching algorithm selects the best matched entry among the entries matching searching conditions. Hence, this matching algorithm is slower and more complicated than the exact prefix matching algorithm.

In the early days of Internet, IP addresses have been classified into five classes. Each class has its own range of network and host address fields. So when a router receives a packet, it reads the packet’s network id only and forwards it easily to the interface matching the prefix. However as the network size has been growing up, IP addresses have started to run out. Therefore CIDR and Subnetting have been proposed and widely adopted to solve the IP address space scarce problem[5]. CIDR and Subnetting use network masks with arbitrary lengths. As there is no conventional IP address classes any more, the router suffers some problem in determining how many bits it needs to check for network address matching. For this reason, the longest prefix matching algorithm such as Patricia trie has been proposed.

Patricia trie is a widely deployed algorithm for searching entries with varying network masks and it is implemented in NET/3 and many other systems such as FreeBSD, Linux etc. While Patricia trie runs very fast when an entry satisfying searching conditions exists, it suffers slow searching time when there is no such entry. This is because it has backtracking. Many algorithms that show improved performance over Patricia trie have been introduced[2]. Usually they search a few times faster than Patricia trie. However, most of them show poor performance in updating time and have large routing tables.

Backbone routers, in these days, should be able to handle 2 million searches per second to have 1Gps performance. Also there exists a possibility for many updating packets to be generated, especially when the network is in unstable mode. Therefore it is very important that backbone routers can update their table so fast, at least one thousand updates per second. In addition to fast search and updating, routers also 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 critical for the algorithm to consume small memory in maintaining and handling the routing table.

We first explain the weaknesses of existing algorithms in Section II. Then we propose a fast and flexible IP lookup algorithm (FFILA) that can show gigabit routing performance in real network environments, which has very low updating time and uses small table size. And we present simulation results and analysis in Section III. Some implementation issues to achieve better performance are considered and the conclusion follows.

II. EXISTING ALGORITHMS

Most IP table lookup algorithms can be classified as three categories; level compression based, caching based, and hash based. These algorithms can search a few times faster than Patricia trie does. However, they fail to update the lookup table faster than Patricia trie, and show sometimes even slower.

A. Level compression based lookup

Algorithms in this category use n-bit B-tree or hash[6]. Because they compare n bits simultaneously rather than 1 bit when traversing the tree, the height of the tree tends to be kept very small, which results in very fast search. But, their performance is still affected by memory usage very much. In case of 4-bit B-tree, a parent node can have up to 2^n children nodes. If the number of children nodes is less than 16, there will be some empty nodes, which result in duplicated pointers that make table update very hard. Due to these reasons, the algorithms are apt to have very large routing table image, which causes slow update.

B. Caching based lookup

In this category, minimal memory usage is achieved by using caching strategies. These algorithms cache routing entries in processor caches that allow fast access and update. However, they are limited in updating rates and suffer from backtracking. The backtracking problem becomes more serious as the number of cached entries increases.

C. Hash based lookup

Hash based lookup algorithms use hash functions to map IP addresses to hash values and then to store routing entries in a hash table. These algorithms can search a few times faster than Patricia trie does. However, they fail to update the lookup table faster than Patricia trie, and show sometimes even slower.
B. Caching based lookup

As algorithms in this category use caching, search time tends to be short[8][7]. But there are some problems when they are used in backbone routers. The first problem is that the caching is not useful in large networks. The temporal correlation of destination addresses of traffic is very important for the efficient operation of cache based algorithms, and this property called locality affects cache performance very much. However, as networks of today become very large in size, there is not much of locality. Therefore, using caching can’t be a good approach to speed up searching in backbone routers.

The other problem is caused by the memory size for caching in large networks. So routers sometimes compresses routing table image, which results in the reduced memory requirement. Compressed image means that the router can load more entries in the same sized cache memory. But the compressed routing table makes updating very hard and slow. Each entry is correlated with the entire table. To update an entry, the router should update the table that is not compressed, and then compress it again. Therefore it takes much time for the router to process each update request.

C. Hash based lookup

Hash based lookup algorithms use hash functions to increase their searching velocity[1]. An entry can found nearly in \( O(1) \) time by using hash. However, when hash is used for IP table lookup, it demands a quite large memory because it fits the exact prefix matching, not the longest prefix matching. So the algorithm should extend the network id length of an entry to be the same as the length of the hash key. For example, when we insert new entry, 5.0.0.0 whose network id length is 8 to empty 10bit-hash, this algorithm should insert 5.0.0.0, 5.0.0.0, 5.128.0.0, and 5.192.0.0 entries for extending network id length.

III. PROPOSED ALGORITHM

We introduce a new IP lookup algorithm that can support giga-bit performance. Existing algorithms have mainly concentrated only on improving the search speed. But routers to work correctly in real networks, they should be able to update an entry at least in \( 1 \) \( \mu \)sec. In addition they should have memory requirement as small as possible to achieve scalability. Our proposed algorithm can search and update very fast with small memory requirement and also use the properties of IP addresses to improve the performance.

A. Hash

Hash can search faster than any other lookup algorithms in exact prefix matching. But this algorithm can’t be used in longest prefix matching easily because of many duplicated pointers that are generated if the lengths of addresses to be inserted are smaller than the lengths of the hash key. Hence, we use the properties of IP addresses. To be more precise, IP addresses consist of two parts, i.e., netmask id and host id. The length of a netmask id is more than or equal to 8bits. Thus if we use the 8-bit hash at the head of Patricia trie, we can get the merit of hash without having duplicated pointers. By doing so, we can maintain the wasted memory at 0.

B. Patricia trie without Backtracking

As Patricia trie searches an entry by distinguishing it from other entries with minimal number of bits' comparison, it can find the right one very fast. As shown in Fig. 1, it takes 24 comparisons to find an entry labeled 5.0.0.1 in the general binary tree while just two comparisons are required in Patricia trie.

However the performance of Patricia trie is deteriorated by backtracking occurred during the search. So by avoiding backtracking, we can obtain remarkable performance improvement. Backtracking occurs because Patricia trie can insert bit masks such as 0xff000ff which are not used in real networks. In other words, when we traverse Patricia trie and reach the leaf node, we are not sure that the node is the one that we try to find. We can simply avoid backtracking in Patricia trie by removing the support of the unnecessary bit masks. Instead of comparing particular bits, the modified Patricia tries to compare all bits in a given range, like from bit 0 to the bit position where the original Patricia trie takes comparison.

For examples in Fig 1, Patricia trie compares the bits of the positions at 1 and 24 to find an entry labeled 5.0.0.1. But the modified Patricia trie compares the bit at 1 to distinguish 5.0.0.0 and 128.0.0.0 and the bits from 1 to 24 to distinguish 5.0.0.0 and 5.0.0.1 as shown in Fig 2.

![Fig. 1. Patricia trie](image1)

![Fig. 2. Patricia trie without Backtracking](image2)
C. Modified Patricia trie with small routing tables

An entry in FreeBSD version of Patricia trie consists of radix node and radix mask node. The sizes of the radix node and mask are 24bytes and 16bytes respectively.

```c
struct radix_node {
    struct radix_mask *rn_mklist;
    struct radix_node *rn_p;
    short rn_bi;
    char rn_bmask;
    u_char rn_flags;
    union {
        struct {
            caddr_t rn_Key;
            caddr_t rn_Mask;
            struct radix_node *rn_Dupedkey;
        } rn_leaf;
        struct {
            int    rn_Off;
            struct radix_node *rn_L;
            struct radix_node *rn_R;
        } rn_node;
    }rn_u;
};
```

Fig. 3. Patricia Trie Node Structure

```c
struct radix_node {
    struct radix_node *parent;
    unsigned int maskedkey[1];
    char comparebit;
    struct radix_node *left;
    struct radix_node *right;
};
```

Fig. 4. Modified Patricia Trie Node Structure

Larger sized node indicates that this algorithm consumes more memory and more time to allocate and to de-allocate it. So we redesigned Patricia trie to compact the node structure. The radix mask node is removed and each radix node has its own mask. Bitmasks and other information for the bit test are removed, too. It is possible to get bitmasks and other information by using the value that represents the bit position for bit test. In this way we could reduce the radix node’s size to 17bytes without using radix mask node.

Insertion in Patricia trie requires two Patricia nodes named innernode and leafnode. However sometimes innernode is not necessary for insertion if a node is inserted at the leaf of Patricia trie. Therefore we modified the insertion part of Patricia trie so that it needs only one node at first. When another node is necessary, the algorithm allocates a new node. In this way, we can make Patricia tries require less memory. For example, let’s assume that Patricia trie has a form of a balanced binary tree with full nodes which depth is k. The modified Patricia trie can have $3 \cdot 2^{k-1} - 1$ nodes in the best case whereas the original Patricia trie have $2^{k+1} - 1$ nodes. In case of $k = 14$, the original Patricia trie will have 32,767 nodes while the modified one has 24,575 nodes. So we can reduce the number of nodes by 8092, which is about reduction of 25%.

IV. Implementation

We modified FreeBSD 4.0 version of Patricia trie to implement the proposed algorithm. This Patricia trie code can be used for many domains such as OSI, IP, Routing, Unix, and XNS without modification. So we consider IP domain separately from other domains to develop an optimal IP table lookup algorithm. This is because as Patricia trie has been in place for general purpose, it could not have been optimized for a particular protocol like IP. We modified Patricia trie only for IP domain. We changed codes according to the modified node structure for IP domain, but left the other domains as before. Fig. 5 explains the overall routing table structure which has been optimized for IP domain.

```c
Fig. 5. Routing Table
```

V. Simulation and Analysis

To measure the performance accurately, we included test codes to run the algorithm in the user mode not in the kernel mode. We tested original Patricia trie code of FreeBSD 4.0 and our new algorithm with Intel Pentium III 530Mhz computer. The routing table entries were generated by routing table dump, MAE-EST\(^1\) which was taken on June 21, 2000 [4]. There have been found 34,765 entries in the routing table.

A. Routing Table Size

We measured the routing table size by the number of entries in it. Fig. 6 shows the tendency of linear increase in routing table sizes for both algorithms. Our proposed algorithm consumes memory 33% less than Patricia trie on the average. This result mainly came from the node size difference as our algorithm requires the node size of 17Bytes which is smaller by 30% than that of Patricia trie. Also our algorithm uses 8bit hash, which results in smaller tree height than Patricia trie. Therefore its memory requirement is becoming lower. As shown in the previous section, our modified algorithm has smaller number of nodes than Patricia trie, which results in the reduced table size too. Lastly our algorithm doesn’t need any radix mask whereas Patricia trie needs additional radix masks.

\(^1\)Exchange point operated by 3C1 WorldCom
We compared the insertion speed of the algorithms for all entries in routing table dump data to be inserted into the empty routing table. We measured the insertion time according to the increase of the number of entries as shown in Fig.7.

Our algorithm is 36% faster than Patricia trie. To insert new entry, it searches the entry to check whether the entry to be inserted exists in the table. So this is mainly because the searching algorithm of FFILA runs faster than that of Patricia. The other reason is that as our algorithm uses 30% smaller sized node than Patricia trie, it requires less time in memory allocation and initialization.

D. Deletion Speed

We tested deletion speed of the algorithms for routing table dump data by removing all entries in the routing table. In simulations we measured the deletion time for the various number of entries. Our algorithm shows performance of 34% faster than Patricia trie in the deletion of all entries.

Fig. 10 shows the performance comparison results for the entry size of 34,765.

<table>
<thead>
<tr>
<th></th>
<th>Insertion Time (µsec)</th>
<th>Search Time (µsec)</th>
<th>Deletion Time (µsec)</th>
<th>Memory Usage (KBytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFILA</td>
<td>3.0</td>
<td>0.45</td>
<td>2.1</td>
<td>1245</td>
</tr>
<tr>
<td>Patricia Trie</td>
<td>4.7</td>
<td>1.31</td>
<td>3.2</td>
<td>1700</td>
</tr>
<tr>
<td>Ratio</td>
<td>1.56</td>
<td>2.91</td>
<td>1.52</td>
<td>1.48</td>
</tr>
</tbody>
</table>

VI. IMPLEMENTATION IMPROVEMENTS

Until now, our proposed algorithm has not assumed any particular platform for implementation. In this section we consider
some implementation ideas for the proposed algorithm to produce much improved performance. The proposed algorithm will run on top of a platform with the following specific techniques.

A. Hash

It is useful to use hash to improve search performance. However, hash has a tendency to consume large memory and it is hard to update. A large hash key size, \( n \), improves search speed while a small \( n \) reduces memory requirement with improved update performance. Therefore \( n \) is a very significant design factor. To remove the drawback of hash, we may use dynamic hash in our algorithm. The routing table consists of FFILA at first. However when predefined conditions are satisfied with the modified Patricia trie nodes are converted into hash nodes. Generally the conditions are related with the number of children or amount of free memory. \( n \) can be an arbitrary number determined by conditions like in Fig. 11. This is very similar to level compression algorithms given in [6] except using modified Patricia trie that introduced here.

![Diagram of Trie Node with 2Bit Hash](image)

**Fig. 11. New Algorithm With Dynamic Hash**

B. Caching

In some caching solutions, the entire table is at once loaded into cache memory to reduce memory access costs. If the algorithm maintains a small routing table, it can work with small cache memory. Because the cost of cache memory is very high, the required memory size is very critical. Our algorithm can be combined with caching more appropriately than Patricia trie because it needs smaller routing table than Patricia trie does.

C. Node Pool

A node pool can be also used to improve performance. If a node should be deleted, it will be removed from trie. However, it will be saved for future use in the node pool instead of being destroyed completely. If a new node is needed, it will be picked up from the node pool instead of being created again. In this way the algorithm can reduce the number of memory allocation and deletion. Also it can reduce allocation and de-allocation system calls, which results in improved performance.

VII. CONCLUSIONS

Most of existing routing table lookup algorithms have some weakness when used in gigabit routers even though they can search faster than Patricia trie. Some of them have very large updating time that hinders from being used in real networks, and the others waste very large memory, which causes a serious problem in scalability. In this paper, we introduced a modified Patricia trie called FFILA that uses 8-bit hash. Our proposed algorithm can search nearly 3 times faster than Patricia trie, which is fast enough to have gigabit performance. It can update the routing table at least 30% faster than Patricia trie. In addition, the algorithm consumes less memory by 33%. Because of its fast update and small memory requirement, combining FFILA with some other implementation techniques such as hash and caching would be a good choice. As we didn’t use any platform related techniques to improve performance, further improvements can be possibly made by adopting such techniques.

REFERENCES