Abstract - Networks are expanding very fast and the number of clients is increasing dramatically, this causes the router forwarding table to become very large and present more demand on faster router operations. In this paper, we address the problem of packet forwarding in the routers aiming to increase the speed of address lookup and minimize the memory required for storing the forwarding table. We propose a new algorithm that makes use of two-level indexing and B-trees. We test the approach and compare it to other famous IP lookup approaches. The preliminary simulations show 20% less memory requirements and the lookup speed scaling linearly with increasing table size.

Keywords: IP Lookup, B-Tree, Longest prefix match

1 Introduction

Routers perform a significant task of forwarding the incoming packets to the appropriate outgoing port to eventually deliver it to the destination. To be able to do that routers are equipped with lookup tables, called forwarding or routing tables, that match IP addresses to outgoing ports. Although this seems like a simple equality assignment operation, the growing sizes of forwarding tables’ poses difficulty and causes delay in network traffic.

IPv4 addresses according to Classless Inter-Domain Routing (CIDR) are composed of 32 bit address and a network mask. The net mask determines the significant IP prefix. For example a Hex IP=10.15.45.91 and net mask of FF.FF.FF.00 means a range of IPs from 10.15.45.00 to 10.15.45.FF. The whole range belongs to one large network and is forwarded to one outgoing port. Entries in the forwarding table consist of IP prefixes and the corresponding port numbers, in the previous example the table entry might look like this 10.15.45.* and port 5.

Aggregation is used to reduce the table sizes. If all other subnets belonging to the same network have similar forwarding information then they are aggregated in one entry as opposed to having a single entry per subnet. The table mentioned earlier might have another entry: 10.15.* and port 3. The two entries are interpreted as follows: addresses that begin with 10.15 are forwarded to port 3 with the exception of those that start with 10.15.45 who will be forwarded to port 5. This establishes the need for the longest prefix match (LPM), in which the entries in the forwarding table are searched for the best match representing the longest bit wise match with the destination address of the incoming packet [1]. However, searching a huge database for LPM is time consuming if performed linearly. Therefore, many researchers addressed this problem and tried to optimize the search problem.

Almost all proposed approaches fall under one of two categories, thumb indexing and binary based approaches. In thumb indexing the prefix is checked bit by bit to direct the search to a specific part of the data space (forwarding table). These approaches perform well depending on the size of the forwarding table. In binary based approaches, studies depend on performing a binary search either on the prefixes stored in the table or the prefixes length to find the LPM. These approaches work better with longer IP addresses such as IPv6 [2].

Our proposed approach combines both thumb indexing and binary based approaches to get a faster lookup in the router with minimal memory requirements. The search algorithm consists of two levels of indexing applied to the forwarding table to narrow down the search space and reduce the memory requirements by reducing the size of the trees. The resulting tables are stored in a Balanced Tree (B-tree) structure to provide faster binary search. The choice of the indexes is very important to better partition forwarding table. In our algorithm we use the first k bits of the address as the first level index and the number of ones in the address as the second level index. In this paper we study the value of k to be used and provide simulation results that shows the optimal values of k.

The rest of this paper is organized as follows. Section 2 briefly surveys the related work advantages and disadvantages. Hardware algorithms and implementations are left out. Section 3 provides a brief description of B-tree structures. Section 4 presents the proposed approach in details. Section 5 provides the preliminary simulation results of the new algorithm and compares to related work. Finally, Section 6 provides the conclusions and directions to complete this short paper.

2 Related Work

There is a wealth of literature and research that addresses the IP lookup problem. We list few sample examples in historical order starting from sequential search to
multi-way tries and hashing. Sequential search is the trivial and simple approach to solve the LPM problem where exact matching is performed for all prefix lengths starting with the maximum length. This is very expensive and requires at worst case 32 comparisons per entry. The prefixes are stored in an array and search is performed on the array by comparing the bits of the incoming packet (destination) IP prefix with the prefixes stored in the array. When a match is found the algorithm remembers this match and moves on in the search until the end of the array is reached, the last match remembered is the LPM [3].

Binary trie is a classical solution for the LPM problem; it is a tree based data structure in which the bits of prefixes in the forwarding table are organized. Each level in the tree presents a bit in the IP address, and the branching of the tree depends on the value of the next bit (0 or 1). Figure 1 shows a small forwarding table and the corresponding binary trie. The edges represent IP bits and the nodes store outgoing ports. If a router receives a packet with prefix 10001*, it traverses the binary trie by taking the following branches: 1, 0, 0. The search finds that the longest prefix match at node d and consequently the router forwards the packet through the port stored at node d. The internal nodes that are not destination nodes are called dummy nodes (shown as empty white circles), which are considered as a waste of memory storage [3,4]. The implementation of this approach requires a high number of memory accesses for each lookup therefore increasing the lookup time. Tries with a lot of nodes have a large storage requirement.

An improvement over the binary trie is path compressed trees (often called binary prefix tree). It eliminates the dummy nodes from the tree structure which reduces the memory space needed to store the tree and significantly decreases the length of the tree. Unfortunately, the original problem of high number of memory accesses still exists in path-compressed tries especially when the tree becomes larger [2, 5].

The basic multi-way trie is similar to binary trie except that the multi-way trie allows the branching based on multiple bits of the prefix rather than a single bit. Commonly two bits are used to decide which of the four outgoing branches to follow. This efficiently decreases the number of memory accesses needed to perform the lookup. The main problem with this approach is that it cannot support arbitrary prefix lengths. Odd length prefixes cannot be stored because the branching depends on 2 bits at a time such as the prefixes 1*, 0*, 101* or 10110* [5,6]. Sahni et al. proposed an efficient dynamic programming algorithm to construct multi-bit trie which overcomes the trie branching problems in the original approach [7].

Waldvogel et al. used hashing to minimize the search time, they chose to use the prefix length as a hash function to partition the huge forwarding table to a smaller hash tables as shown in Figure 2. Then they performed linear search on the smaller tables which still resulted in a large number of memory accesses [3].

Warkhede et al. proposed to work with the IP addresses as decimal values and arrange these values into ranges. Many IP addresses could belong to the same range; the approach depends on the prefix length to distinguish similar IP decimal values for different prefix lengths [8]. Warkhede et al. applied the IP range on multi-way tries, in which the nodes of the trie are classified into different ranges. The performance of this scheme was competitive compared to that of multi-way tries for IPv4.

Leu et al. used the idea of ranges and proposed a data structure that connects overlapped addresses and classifies them into levels where the lower levels represent IP with smaller numeric value [9]. This approach was designed for longer IP addresses such as 128 bit IPv6. The speed of the proposed algorithm is not related to the length of IP addresses. However the speed will suffer as the number of prefixes increases. In general trie based approaches are slow for large forwarding table sizes and worst case memory increases exponentially.
3 B-Trees

Balanced trees were originally designed to provide a fast disk access for secondary storage devices. Only a small portion of data is kept in the faster primary storage (RAM) because of its small size. B-trees reduce the number of secondary storage accesses when part of the tree is kept on the secondary storage. At each access, it tries to read as much data as possible. Each node can hold four fields, the number of keys currently stored in the tree \( n[k] \), the keys themselves, a Boolean value indicating if it is a leaf or not, and pointers to the children nodes \( n[k]+1 \).

B-Tree keeps the entries sorted and allows insertion, deletion and searching in logarithmic time complexity. Non leaf nodes children belong to the same range and the number of memory accesses needed is related to the height of the tree. In the best case the height of a B-Tree is \( \log M \), while the worst case height is \( \log M/2 \), where \( M \) is the number of children the node can have \([10]\).

The choice of the properties or characteristics to be used to partition the forwarding table is the key factor for faster search. The index must apply to all entries and it has to uniquely identify the different entries of the table. The first level index is the first \( k \) bits of the destination address; this choice is applicable for all entries and will evenly partition the address space. The number of all possible entries in the index table will be \( 2^k \) bits, this also means that we divide the forwarding table to \( 2^k \) smaller tables (sub-lists) indexed by the first \( k \) bits. To further reduce the search space, we use the number of ones in the IP address as a second index to partition the sub-lists resulting from applying the first level index. The IP addresses are stored in their decimal value to minimize the memory usage and provide faster search.

Figure 4 shows an example of the proposed approach using 3 bits of the address as the first level index and assuming a packet with destination address 101000...0 arriving at the router. First, the router will extract the first 3 bits and use them as first level index (101) for the first index table. If not found the packet is forwarded to the default port. If found, which is the case in this example, the router will calculate the number of ones in the destination IP, which in this case is 2, and use it to index the second level. The IP forwarding problem is narrowed down to searching the chosen B-Tree. The tree is searched for the destination address, if found the port number is returned and the packet is forwarded. If it is not found the right most bit is changed to 0, and a new binary search is conducted. For each miss the right most bit corresponding to the number of searches performed on this IP is modified to 0 and a new search is conducted.

4 Two-level indexing and B-Trees

The approach is to combine the best of features of the previous work in one simple and fast implementation. We apply two level indexing, decimal storage of IPs, B-trees and Binary search to the IP lookup problem. Figure 3 presents the proposed data structure. We use two-level indexing with two tables. The first table holds the values of the first index with pointers to the appropriate second level index tables. The second level index tables hold pointers to the B-trees.

The choice of the properties or characteristics to be used to partition the forwarding table is the key factor for faster search. The index must apply to all entries and it has to uniquely identify the different entries of the table. The first level index is the first \( k \) bits of the destination address; this choice is applicable for all entries and will evenly partition the address space. The number of all possible entries in the index table will be \( 2^k \) bits, this also means that we divide the forwarding table to \( 2^k \) smaller tables (sub-lists) indexed by the first \( k \) bits. To further reduce the search space, we use the number of ones in the IP address as a second index to partition the sub-lists resulting from applying the first level index. The IP addresses are stored in their decimal value to minimize the memory usage and provide faster search.

Figure 4 shows an example of the proposed approach using 3 bits of the address as the first level index and assuming a packet with destination address 101000...0 arriving at the router. First, the router will extract the first 3 bits and use them as first level index (101) for the first index table. If not found the packet is forwarded to the default port. If found, which is the case in this example, the router will calculate the number of ones in the destination IP, which in this case is 2, and use it to index the second level. The IP forwarding problem is narrowed down to searching the chosen B-Tree. The tree is searched for the destination address, if found the port number is returned and the packet is forwarded. If it is not found the right most bit is changed to 0, and a new binary search is conducted. For each miss the right most bit corresponding to the number of searches performed on this IP is modified to 0 and a new search is conducted.

5 Preliminary Results

We implement the approach in C++ using Microsoft Visual C++ 2008 Express Edition and tested it using random IPv4 addresses. The test was performed on an Intel Core 2 Duo processor with 2.5 GHz speed and 4 GB of memory. To test the amount of memory needed for our data structure, we fixed the first level index to 3 bits, we calculated the memory required for a typical forwarding table size of 40,000 IP addresses to be stored in the suggested data structure. Figure 5 compares our approach with binary prefix tree and multi-way tree approaches. The results of those approaches are taken form \([6]\). As shown in the figure, our proposed approach provides a lower memory requirement. This is mainly because
of the dummy nodes overhead in the binary and multi-way trees and the overhead introduced by the index tables is negligible.

Next we simulate the memory requirements versus the number of bits in the first level index ($k$). Using a 300 entry table, we test the amount of memory needed for different values of $k$. The result are shown in Figure 6, the memory required is increasing exponentially as we increase the index size, which is expected because of the size of the $2^k$ index table. The memory overhead caused by values of $k$ between two and five is acceptable.

Finally we consider the number of memory accesses for different values of index size (2-5 resulted in minimal memory overhead). We simulate the minimum, maximum and average number of memory access needed for different $k$ values. Figure 7 shows that the number of memory accesses is increasing linearly as we increase the size of the index bits, which still scales better than multi-way tries.

![Figure 5. Memory requirements compared to related work.](image1)

![Figure 6. Memory requirements vs index level 1 size ($k$).](image2)

### 6 Conclusions

In this paper we suggest a new data structure and search algorithm to reduce the time to search the ever growing forwarding tables in edge routers.

We compared our approach with other related work in terms of speed and memory requirements. We achieved 20% reduction in memory requirements compared to the closet approach. The lookup speed represented by the number of required memory accesses is significantly lower and scales almost linearly with the table size.

This is just a preliminary examination of the new algorithm which shows promising results. To better compare against related work, we are working on testing the approach with real data from Internet backbone routers such as Mae-East network access point (NAP), Mae-West Paix, Pb…etc. Update times need to be measured and quantified and finally scaling for larger IP sizes such as IPv6 are to be studied.

### 7 References


