Dynamic Routing Tables Using Simple Balanced Search Trees

Y.-K. Chang and Y.-C. Lin
Department of Computer Science and Information Engineering
National Cheng Kung University
Tainan, Taiwan R.O.C.
ykchang@mail.ncku.edu.tw

Abstract. Various schemes for high-performance IP address lookups have been proposed recently. Pre-computations are usually used by the special designed IP address lookup algorithms for better performance in terms of lookup speed and memory requirement. However, the disadvantage of the pre-computation based schemes is that when a single prefix is added or deleted, the entire data structure may need to be rebuilt. Rebuilding the entire data structure seriously affects the update performance of a backbone router and thus not suitable for dynamic routing tables.

In this paper, we develop a new dynamic routing table algorithm. The proposed data structure is a collection of balanced binary search trees. The search, insertion, and deletion operations can be finished in O(log N) time, where N is the number of prefixes in a routing table. Comparing with the best existing dynamic routing table algorithm which is PBOB (prefix binary tree on binary tree), our experiment results using real routing tables show that the proposed scheme performs better in all aspect.

Keywords：*IP address lookup, dynamic routing table, fast update, precomputation*

1. Introduction

To handle gigabit-per-second traffic rates, the current backbone routers must be able to forward millions of packets per second at each port. The IP address lookup in the routers is the most critical task to perform and must have the capability of forwarding millions of packets per second. When a router receives a packet, the destination address in the packet's header is used to lookup the routing table. There may be more than one route entries in the routing table that match the destination address. Therefore, it may require some comparisons with every route entries to determine which one is the longest matching. The longest route from all the matched entries is called the longest prefix match (LPM). The IP address lookup problem becomes a longest prefix matching problem.

To design a good IP address lookup scheme, we should consider several key requirements: lookup speed, storage requirement, update time and scalability. The update process is the focus of this paper. Currently, the Internet has a peak of a few

hundred BGP updates per second. Thus, the address lookup schemes with fast update time are desirable to avoid routing instabilities. These updates should interfere little with normal address lookup operation.

Various algorithms for high-performance IP address lookup have been proposed. In the survey paper [10], a large variety of routing lookup algorithms are classified and their complexities of worst case lookup, update, and memory references are compared. Despite the intense research that has been conducted in recent years, there should be a balance between lookup speed, memory requirement, update, and scalability for a good IP address lookup scheme. The pre-computation [1], [3], [4], [8], [11] perform a lot of pre-computation and thus improve the performance of the lookup speed and memory requirement. However, a disadvantage of the pre-computation is that when a single prefix is added or deleted, the entire data structure may need to be rebuilt. Rebuilding the routing tables seriously affects the update performance of a backbone router. Thus, the schemes based on pre-computation are not suitable for dynamic routing tables. On the other hand, schemes based on the trie data structure like binary trie, multi-bit trie and Patricia trie [9] do not use pre-computation; however, their performances grow linearly with the address length, and thus the scalability of these schemes is not good when switching to IPv6 or large routing table.

Although schemes like [5], [6], [12] develop a search tree data structure that is suitable for the representation of dynamic routing tables, the complex data structure leads to the memory requirement expanded and reduce the performance of lookup. Sahni and Kim [4] developed a data structure, called a collection of red-black tree (CRBT), that supports three operations for dynamic routing table (longest prefix match, prefix insert, prefix delete) in O(log N) time each. In [5], Lu and Sahni developed a data structure called BOB (binary tree on binary tree) for dynamic routing tables. Based on the BOB, data structures PBOB (prefix BOB) and LMPBOB (longest matching prefix BOB) are also proposed for highest-priority prefix matching and longest-matching prefix. On practical routing tables, LMPBOB and PBOB permit longest prefix matching in O(W) and O(log N). For the insertion and delete operations, they both take O(log N) time. Suri et al. [12] have proposed a B-tree data structure called multiway range tree. This scheme achieves the optimal lookup time of binary search, but also can be updated in logarithmic time when a prefix is inserted or deleted.

In this paper, we develop a data structure based on a collection of independent balanced search trees. Unlike the augmented data structures proposed in the literature, the proposed scheme can be implemented with any balanced tree algorithm without any modification. As a result, the proposed data structure is simple and has a better performance than PBOB we compared.

The rest of the paper is organized as follows. Section 2 presents a simple analysis for the routing tables. Section 3 illustrates proposed scheme based on the analysis in section 2 and the detailed algorithms. The results of performance comparisons using real routing tables are presented in section 4. Finally, a concluding remark is given in the last section.

Table 1: Data structure analysis for MSPT.

Database (year-mouth)	AS6447 (2000-4)	AS6447 (2002-4)	AS6447 (2005-4)
number of prefixes	79530	124798	163535
Level-1 prefixes	73891(92.9%)	114745 (91.9%)	150245 (91.9%)
Level-2 prefixes	4874 (6.1%)	8496 (6.8%)	11135 (6.8%)
Level-3 prefixes	642 (0.8%)	1290 (1%)	1775 (1.1%)
Level-4 prefixes	104 (0.1%)	235 (0.2%)	329 (0.2%)
Level-5 prefixes	17	29	45
Level-6 prefixes	2	3	6

2. Analysis of Covering and Covered Prefixes

The Border Gateway Protocol (BGP) is the de facto standard inter-domain routing protocol in the Internet. BGP provides loop-free inter-domain routing between autonomous systems, each consisting of a set of routers that operate under the same administration. The address space represented by an advertised BGP prefix may be a sub-block of another existing prefix. The former is called a *covered* prefix and the latter a *covering* prefix. For example, the address block 140.116.82.0/24 is covered by another address block 140.116.0.0/16.

We analyzed three BGP routing tables obtained from [1], and obtained the detailed statistics for the enclosure relationship between the covered and covering prefixes. Theoretically, one prefix may be covered by at most 31 prefixes for IPv4. The prefix and the ones that cover it form an enclosure chain. Therefore, the theoretical worse-case enclosure chain size is 32 for IPv4. However, our analysis shows that the chain size is 6 for all the tables we examined. Figure 1 shows the enclosure relationship between covering and covered prefixes for an example routing table that has the enclosure chain size of 5. Contrary to the definition in [7], we define the numbering of the prefixes in a bottom-up manner. For example, the prefixes that do not cover any other prefix in the routing table are called the *level-1* prefixes. We further show the number of prefixes in each level for all the three routing tables in Table 1. The level-1 prefixes account for about 92% ~ 93% of the prefixes in a routing table. The level-2 prefixes account for about 6% ~ 7% of the prefixes. The prefixes in other levels only account for less than 1% of the total prefixes. Based on the above analysis, it is straightforward to design dynamic routing lookup algorithm with a O(log N) complexity for search and update operations.

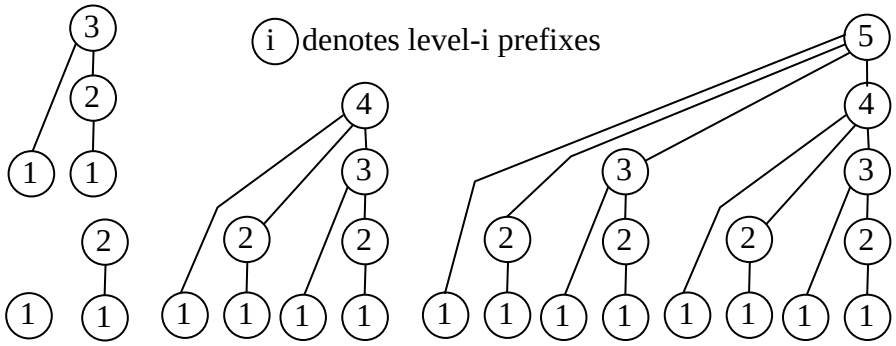

Figure 1: Enclosure relationship between covering and covered prefixes.

3 The proposed scheme

It is easy to see that all the prefixes in one level are disjoint. Moreover, the prefixes in level-i are more specific than the ones in level-(i+1). Therefore, assuming there are at most *s* levels in the routing table, we can build s independent data structures for the lookup problem by obeying the *tree level constraint* as follows.

Tree Level Constraint: Based on the enclosure relationship between prefixes, the level-i prefixes are stored in the level-i data structure.

```
Algorithm Search(d, root[], s) { // d is the destination address, s is the number
            of trees
      For (i = 1; i <= s; i++) {
            x = root[i];
            While x ≠ Null {
                  If x.prefix ⊇ d Then Return x.prefix; //A⊇B denotes A covers B
                  Else
                        If d < x Then x = x.LeftChild;
                        Else x = x.RightChild;
            } //End-While
      } //End-For
      Return default_prefix
}
```

Figure 2: Algorithm to find longest prefix match.

```
Algorithm Insert (P, root[], s) {
    For (i = 1; i <= s; i++) {
        x = root[i];
        While ( x ≠ null ) {
            If ( P = x.prefix ) Return;
            Else If ( P ⊆ x.prefix ) {    /* x.prefix encloses prefix P */
            Q = x.prefix; x.prefix = P; P = Q; break;
            } Else If (x.prefix ⊆ P ) break;
            If ( P > x.prefix )
                If (x.RightChild = NULL) {
                    new_node = Create_A_Node(P);
                    x.RightChild = new_node;
                    BST_Balancing(root[i]); Return;
                } Else x = x.RightChild;
            Else
                If (x.LeftChild = NULL) {
                    new_node = Create_A_Node(P);
                    x.LeftChild = new_node;
                    BST_Balancing(root[i]); Return;
                } Else x = x.LeftChild;
        } //End-While
    }//End-For
    // The level is incremented by one
    s = s + 1;
    root[s] = Create_A_Node(P);
}
```

Figure 3: Algorithm to insert a prefix

If the enclosure relationship between prefixes is changed because of insertion or deletion, the locations of some of the prefixes must also be adjusted in order to follow the tree level constraint. In this paper, we decide to use s balanced binary search trees. Other data structures will be considered in the future. If the matched prefix can be found in the level-1 tree, then it must be the longest prefix match and the search process is done. Otherwise, higher level trees are searched until the level-s tree. We know that the number of levels is a constant and each level is implemented as a balanced binary search tree. Therefore, the total complexity must be O(log N) for a routing table of N prefixes. Figure 2 shows the search algorithm Search(d, root[], s), where parameter d is the destination address and there are s balanced trees.

The insertion of a prefix P is done by performing tree traversals from the level-1 tree to the level-s tree. The main task when traversing the trees is to check if there exists a prefix that covers P or is covered by P. If no such prefix is found, then P is disjoint from all the prefixes in level-1 tree. And thus, P is inserted as a leaf node in the level-1 tree. A possible rotation is needed after P is inserted. However, if a prefix Q in the level-1 tree is found to cover P, then Q is replaced by P and the process of inserting Q in the level-2 tree is performed. If a prefix Q is found to be covered by P, the process of inserting P in level-2 tree is performed.

Table 2: Performance statistics

(a) Memory requirement in KB.

schemes	AS6447 (79,560)	AS6447 (124,824)	AS6447 (163,574)
PBOB	1,525	2,374	3,101
Proposed scheme	1,330	2,087	2,734

(b) Average search time in microseconds.

schemes	AS6447 (79,560)	AS6447 (124,824)	AS6447 (163,574)
PBOB	1.02	1.37	1.57
Proposes scheme	0.65	0.79	0.88

(c) Average insertion time (in Microsecond).

schemes	AS6447 (79,560)	AS6447 (124,824)	AS6447 (163,574)
PBOB	0.90	0.89	1.01
Proposed scheme	0.71	0.75	0.76

(d) Average deletion time (in Microsecond) for IPv4.

schemes	AS6447 (79,560)	AS6447 (124,824)	AS6447 (163,574)
PBOB	0.57	0.57	0.64
Proposed scheme	0.47	0.48	0.49

Figure 3 shows the insertion algorithm *Insert*(*P*, *root, s*) that inserts a prefix *P* in the balanced binary search trees rooted at *root[1..s]*. If P covers all the prefixes in the routing table, a new tree at level s+1 will be generated. After P is inserted in one of the balanced trees, a possible balancing operation (function *BST_Balancing()*) must be performed.

The deletion process is done by first searching the prefix D to be deleted from one of s balanced search trees, say level-i tree. There may be a prefix Q that covers D in level-(i+1). If prefix D is the only prefix that Q covers, then the tree level constraint is violated. Therefore, in this case, prefix Q must be moved to level-i. The violation of tree level constraint may cause a chain effect to higher level trees. On the other hand, if prefix D is not the only one covered by Q, then anything other than deleting prefix D is not required.

The process of checking whether or not prefix D is the only prefix covered by

prefix Q may have a strong impact on the overall process time for deletion. Therefore, we propose an efficient scheme to minimize the time taken for this process. This scheme only checks if prefix Q covers the prefixes y and z that are the smallest prefix in D's right subtree and the largest prefix in D's right subtree, respectively. Why is no other prefix needed for examination? The reason is as follows. If there is another prefix w that is also covered prefix Q, then Q must also cover y or z because y or z sits between P and w. Also notice that it is possible that the immediate parent Q of prefix P in level-i is in level-(i+2). In this case, P can be deleted directly. Prefix Q is kept in the leve-(i+2) tree because prefix Q must also cover another prefix in level-(i+1). Figure 4 shows the details of the deletion algorithm.

```
Algorithm Delete(P, root[], s) { // d is the destination address, s is the
            number of trees, z = y = ϕ; // y and z are the successor and
            predecessor of node x containing prefix P
      For (i = 1; i <= s; i++) {
            x = root[i];
            While x ≠ Null {
                  If x.prefix = P Then
                        Node = Search_a_Tree_for_enclosure(P, root[i+1]);
                        If Node = ϕ Then
                              BST_Delete(P, root[i]); Return;
                        Else
                              If x.RightChild ≠ ϕ Then
                                    y = Smallest_Prefix(x.RightChild);
                              If y ≠ ϕ and Node.prefix ⊇ y.prefix Then
                                    BST_Delete(P, root[i]); Return
                              If x.LeftChild ≠ ϕ Then
                                    z = Largest_Prefix(x.LeftChild);
                              If z ≠ ϕ and Node.prefix ⊇ z.prefix Then
                                    BST_Delete(P, root[i]); Return

                              x.prefix = node.prefix; P = node.prefix;
                              If i = s Then BST_Delete(P, root[i]); Return
                              Else Break;
                  If P ⊇ x.prefix Then Break; // Break inner loop
                  If P ⊆ x.prefix Then Return ; // P does not exist
                  If P < x Then
                        y = x;
                        x = x.LeftChild;
                  Else
                        z = x;
                        x = x.RightChild;
            } //End-While
            Return ; // P does not exist
      } //End-For
}
```

Figure 4: Algorithm to delete a prefix.

4 Performance Evaluations

In this section, we present the performance results for IPv4 routing tables. Three BGP tables of different sizes obtained from [1] are used in our experiments. These BGP routing tables reflect the realistic sizes of the routing tables in the backbone routers currently deployed on the Internet. We compare the proposed algorithm with the prefix binary tree on the binary tree structure (PBOP) [5]. We only choose PBOB for comparisons because other schemes also proposed by the authors in [7] do not perform better than PBOB. The performance experiments are implemented in C language on a Linux Redhat platform with a 2.4G Pentium IV processor containing 8KB L1, 256KB L2 caches and 768MB main memory. GNU gcc-3.2.2 compiler with optimization level –O4 is used.

Table 2(a) shows the amount of memory used by each of the tested schemes. We can see that the proposed scheme performs better than PBOB, about 15% less memory than the PBOB structure. This result can be attributed to that the node structure is simpler than that in PBOB. In PBOB, less than 1 percent of range sets in the constructed PBOB are empty. It needs additional memory to store those nonempty range sets (each nonempty range set is constructed by an array structure with six entries). To measure the lookup times, we first use an array *A* to store the address parts of all prefixes in a routing table and then randomize them to obtain the input address sequence. The time required to determine all the LPMs is measured and averaged over the number of addresses in *A*. The experiment is repeated 100 times, and the mean of these average times is computed. These mean times are reported in Table 2(b). Although the worst case search time may be worse than that in PBOB because all the balanced binary trees must be searched, the average time is better than PBOB. This is because most of the search result can be determined in the level-1 tree. For the average update (insert/delete) time, we start by randomly selecting 5% of the prefixes from the routing tables. The remaining prefixes are used to build the desired data structures (PBOB and the proposed balanced binary search trees). After the desired data structure is constructed, the selected prefixes are inserted. Once the selected insertions are done, they are removed from the routing tables. The total elapsed insertion and deletion times are averaged to get the average insertion and deletion times. This experiment is repeated 10 times and the mean of the average times is reported. The deletion times for PBOB are obtained by the implementation with the optimized version of the deletion algorithm proposed in [7]. In other words, the empty nodes in PBOB are not removed if they have two children nodes. However, in the proposed scheme, we implement the complete deletion procedure such that as long as the prefixes are deleted, the corresponding nodes in one of the balanced tree are removed and the required rotations are also performed. Even with this implementation difference, the proposed scheme still performs better than PBOB.

5 Conclusions

We have developed a dynamic routing table data structure based on the enclosure relationship structure. At most 6 independent balanced binary search trees are needed for the routing tables currently available in backbone routers. Since the first two trees account for 97%-99% prefixes in the routing table, the average performance of the lookup, insertion, and deletion are very well. Since the number of the balanced tree is

constant, the search, insertion, and deletion operations can be finished in O(log N) time, where N is the number of prefixes. Our experiment results showed that the proposed scheme performs better than PBOB, the best dynamic routing table algorithm, in terms of lookup, insertion, and deletion.

References

[1] BGP Routing Table Analysis Reports, http://bgp.potaroo.net/.

[2] A. Brodnik, S. Carlsson, M. Degermark, S. Pink, "Small Forwarding Tables for Fast Routing Lookups," ACM SIGCOMM, pp. 3-14, Sept. 1997.

[3] N. F. Huang, S. M. Zhao, J. Y. Pan, and C. A. Su, "A Fast IP Routing Lookup Scheme for Gigabit Switching Routers," in Proc. INFOCOM, pp. 1429-1436, Mar. 1999.

[4] K. Kim, S Sahni, "An O(logn) Dynamic Router-Table Design," IEEE Transactions on Computers, pp. 351-363, Mar. 2004.

[5] B. Lampson, V. Srinivasan and G. Varghese, "IP Lookups Using Multiway and Multicolumn Search," IEEE/ACM Transactions on Networking, Vol. 3, No. 3, pp. 324-334, Jun.1999.

[6] H. Lu, S. Sahni, "Enhanced Interval Tree for Dynamic IP Router-Tables," IEEE Transactions on Computers, pp. 1615-1628, Dec. 2004.

[7] X. Meng, Z. Xu, B. Zhang, G.. Huston, S. Lu, L. Zhang, "IPv4 Address Allocation and the BGP Routing Table Evolution," ACM SIGCOMM, pp. 71-80, Jan. 2005.

[8] S. Nilsson and G. Karlsson "IP-Address Lookup Using LC-trie," IEEE Journal on selected Areas in Communications, 17(6):1083-1092, June 1999.

[9] K. Sklower, "A Tree-based Packet Routing Table for Berkeley Unix," Proc. Winter Usenix Conf, pp. 93-99, 1991.

[10] M. A. Ruiz-Sanchez, Ernst W. Biersack, and Walid Dabbous, "Survey and taxonomy of IP address lookup algorithms," IEEE Network Magazine, 15(2):8--23, March/April 2001.

[11] M. Waldvogel, G. Varghese, J. Turner and B. Plattner, "Scalable High-Speed IP Routing Lookups," ACM SIGCOMM, pp. 25-36, Sept. 1997.

[12] P. Warkhede, S. Suri, G.. Varghese, "Multiway Range Trees: Scalable IP Lookup with Fast Updates," The International Journal of Computer and Telecommunications Networking, pp. 289-303, Feb. 2004.