Twin-bit based Fast IP Lookup and Update Algorithm

Ching-Lung Chang, Cheng-Che Hsu
Institute of Computer Science,
National Yunlin University of Science and Technology,
Taiwan, R.O.C.
chang@yuntech.edu.tw

ABSTRACT

Success of the Internet and the increased use of broadband in homes have caused a gradual shift in traffic on the Internet from data to multimedia communications. Traffic on the Internet traffic is increasing daily, while advances in communication technologies have allowed the Ethernet speeds to rise from 10 Mbps to 100 Mbps, and now to 10Gbps. The IP address lookup time in gigabit networks is a bottleneck for a router, which needs to find the longest prefix matching for the address. This study proposes a Twin-bit based IP address lookup and update algorithm, based on tree structures, called Fast Twin-bit Tree (FTBT). FTBT can effectively reduce the number of memory access times, and provide fast routing table update. Performance evaluation results reveal that the proposed algorithm can lookup an address among 78504 routing entries in six memory accesses on average.

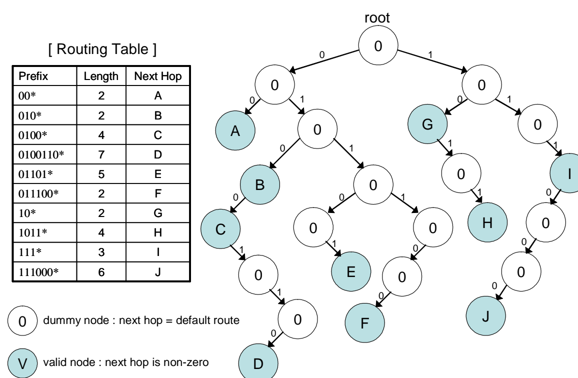
1: INTRODUCTIONS

Internet traffic has been growing rapidly due to the wide acceptance and success of the Internet while advances in communication technology has increased Ethernet speeds from 10 Mbps to 100 Mbps, and now to 10Gbps. Designing a high-performance router to increase the packet processing speed to the 10Gbps is an most important issue. A key design issue for a gigabit router is the IP address lookup scheme. The CIDR (Classless Inter-Domain Routing) [1], which removes the restriction of IP class, makes IP address efficient and flexible to use. In CIDR, the IP address lookup is a bottleneck for a gigabit router, which has to find the Longest Prefix Matching (LPM) for the address.

Since the speed of IP Lookup affects the router performance, many schemes have been developed to solve the classless IP lookup problem. Some such schemes (such as CAM-based and Hashing-based) are hardware-based, while others are more suitable for software solution (e.g. Tree-based schemes). Since software-based IP lookup is more suitable for applying to the large scale network environment, this study presents a tree-based IP lookup scheme.

P. Gupta, et al. [2] used an IP address to index the routing table directly. Their scheme can perform lookup in only one memory access, but has a large memory requirement. Some hardware schemes use Content Addressable Memory (CAM) [3]-[4] to increase the IP Lookup speed. The disadvantages of CAM-based lookup scheme are the high cost and heavy power consumption. Another lookup scheme uses hashing [5] to realize the IP address table lookup. Nevertheless, the hashing function has the collision problem, in which the numbers of IP address are indexed to the same entry of the hashing table. Authors of [6] proposed a multi-hashing function to reduce the collision problem.

Unlike the hardware-based, most software-based IP lookup solutions are based on the tree structure. The solution of [7] is a typical IP lookup operation with a tree data structure. Figure 1 illustrates the mapping from the routing table to the binary tree, where the left leaf represents the bit “0”, and right leaf represents the bit “1”. The node value indicates the next hop port number. A node value of zero (called as a *dummy* node) does not represent a routing entry. For example, if the output port of IP address “10110000” is looked up, then this address can be applied to trace the tree, and thus derive the LPM result. Finding the longest prefix matching involves looking up the tree until a leaf node is reached. In this example, the lookup result is H. Although the binary tree is a simple method, it requires 32 memory accesses (the highest level in the tree) to find the LPM in the worst cast.



Prefix	Length	Next Hop
00*	2	A
010*	2	B
0100*	4	C
0100110*	7	D
01101*	5	E
011100*	2	F
10*	2	G
1011*	4	H
111*	3	I
111000*	6	J

Figure 1. An example of binary Tree

To reduce the height of the IP lookup tree, Patricia [8] proposed a compressed tree structure, which removes

partial unnecessary dummy nodes, but which needs more memory storage to record the skip information. Berger proposed a prefix tree structure [9], which removes all of dummy nodes and has the least memory storage in IP address lookup. In contrast to the previous concepts, the Wu proposed LPFST scheme [10], which utilizes the heap concept to reduce the table lookup time. LPFST endeavors to place the longest prefix on the upper level of the tree, and thus has the fastest lookup time. However, LPFST incurs more memory access in routing entry update operations.

As discussed above up until now, all of the tree-based IP lookup schemes perform the lookup operation bit by bit. The complexity of the IP lookup and routing table update operation is $O(w)$, where w is the prefix length. In contrast with the current tree-based scheme, the proposed Fast Twin-bit Tree (FTBT) IP lookup algorithm is based on a Twin-bit based tree structure. Each node in this tree structure represents two bits of an IP address. The FTBT algorithm can reduce the order of the memory access in an IP lookup and routing entry update operation to $O(\frac{w-8}{2})$.

The rest of this study is organized as follows. Section 2 first defines the Twin-bit based tree's node and data structure. Section 3 introduces the proposed Fast Twin-bit Tree (FTBT) algorithm. Section 4 compares the performance of the FTBT IP lookup with other IP lookup schemes. Conclusions are finally in Section 5.

2: TWIN-BIT BASED TREE NODE

In contrast with the other tree-based IP lookup schemes [7]-[10], the proposed scheme adopts two bits to define a tree node to reduce the height of the lookup tree, and thus can minimize the number of memory accesses in an IP address lookup.

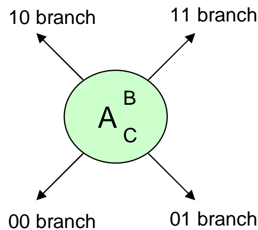
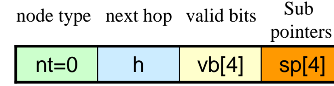
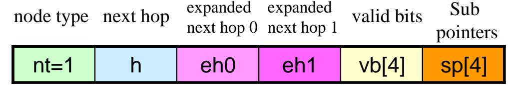


Figure 2. FTBT Tree Node

Figure 2 shows the FTBT tree node, where A is the output port number of the prefix value *Curr_Prefix*, and B and C represent the output port numbers of the prefix value *Curr_prefix* $\times 2^1$ and *Curr_prefix* $\times 2^1 + 1$, respectively. This study refers to B and C as **expanded next hop0** and **expanded next hop1**, respectively. When the length of the lookup prefix is more than the 1-bit length of *Curr_prefix*, the lookup operation visits one of the sub pointers, *00 branch, 01 branch, 10 branch*, or *11 branch*, to find the associated output port number. For example if the prefix value is *Curr_prefix* $\times 2^2$, then the lookup operation visits the *00 branch* to find the associated output port number.



(a) *node type 0*



(b) *node type 1*

Figure 3. FTBT Node Formats

The FTBT node structure is defined in Fig. 3. Each node is in one of two node formats. All fields are defined below.

Node type (nt): 0 or 1. Stands for the node type 0 or node type 1.
Next hop (h): the output port number of *curr_prefix*.
Valid bits (vb[4]): indicates whether the field of the Sub pointers have valid values.
Sub pointers (sp[4]): a pointer that points to the *00 branch*, *01 branch*, *10 branch*, or *11 branch.*
Expanded next hop 0 (eh0): the output port number of the prefix value *curr_prefix* $\times 2^1$.
Expanded next hop 1 (eh1): the output port number of the prefix value *curr_prefix* $\times 2^1+1$.

The node type is 0 when the routing table contains no routing entry mapped to the *eh0* field or mapped to the *eh1* field.

3: FAST TWIN-BIT TREE (FTBT) IP LOOKUP ALGORITHM

Some terms used in the FTBT algorithm are defined below.

- ***PosVal(prefix A, i, j)***: returns the value of the bits i to j of *prefix A*, bit 0 is the MSB of *prefix A*. For example: PosVal(1101*, 0, 2) = 110.
- ***Prefix match***: consider two prefixes, $A = a_0a_1a_2...a_n$ and $B = b_0b_1b_2b_3...b_m$. If $n<m$ and $A =$ PosVal(B, 0, n), then prefixes A and B match.

In real operations, the prefix length in the routing table is greater than 8. A range table (RTB) is created to reduce the number of memory accesses in the lookup tree search. The RTB has 2^8 entries, each with a pointer to the root node of the FTBT tree. Figure 4 shows the routing table and the associated FTBT tree. In this example, the first eight bits of prefixes are used as the index of the range table. Thus, a large routing table is divided into

multiple FTBT trees. In FTBT tree construction, if an FTBT node with no entry in the routing map is called a pseudo-node. The next-hop field of a pseudo node always has a value of zero.

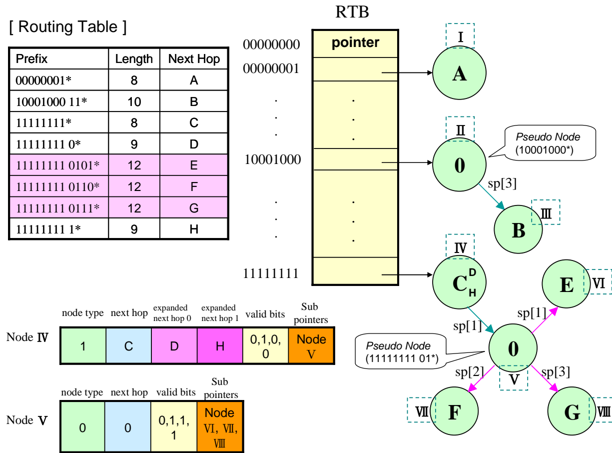


Figure 4. FTBT Example

```
Function Construction (routing table)
{ /* P = {P0, P1,..., Pn-1} are the routing table prefixes ( unsorted ).
    li and hi are the length and the next hop of a route prefix Pi respectively.

    RTB is the Range Table which has 2^8 entries, each entry has a
    pointer to map a corresponding node. */
   while (more entry in routing table)
   {
      read one entry ( Pi , li , hi ) from routing table;
      gPrefix = Pi ; gLength = li ; gNexthop = hi ;   //global variables
      index = PosVal ( gPrefixes , 0 , 7 );

      NewNode = malloc ( Node(0) ); //memory allocated node type 0
      /* NewNode.nt = 0 , NewNode.h = gNexthop ,
         NewNode.vb = 0 , NewNode.sp = NULL    */

      if ( RTB[index] ≡ NULL )
      {
          if ( gLength ≡ 8 )
          {   NewNode.h = gNexthop;
              RTB[index] = NewNode;
          }
          else
          {    PseudoNode = malloc ( Node(0) );
               /* PseudoNode.nt = 0 , PseudoNode.h = 0 ,
                  PseudoNode.vb = 0 , PseudoNode.sp = NULL   */
               RTB[index] = PseudoNode;
               Insertion (NewNode , RTB[index] , 0 ); }
      }else
          Insertion (NewNode , RTB[index] , 0 );
   }// end while
} // end Function
```

Figure 5. FTBT Construction Algorithm

Figure 5 illustrate the FTBT tree construction algorithm. Initially, the whole RTB points to NULL values. In the CIDR, a routing entry is represented by the (p, l, h) format, where p denotes the route prefix, l is the length of prefix p, and h is the output port number of prefix p. The proposed algorithm does not have to sort the routing table before constructing the FTBT tree.

First, the first 8-bit value of the prefix p is extracted and used as the index of RTB. Thus, the first 8-bit value of prefix 8 determines the FTBT tree related to the prefix p. If the associated RTB bank points to the null address, then the associated FTBT tree is empty, and a new FTBT tree has to be constructed. Otherwise, the *Insertion()* function is called to insert the routing entry into the associated FTBT tree. If a tree node in a constructed FTBT tree has no correspondence routing entry, then a pseudo node is inserted.

```
Function Insertion (a, b, Level)
{
      px_remain = gLength - ( 8+Level*2 );
      case 1: px_remain > 2
      {   k=PosVal ( gPrefix, 8+Level*2 , 8+Level*2+1 ); //2-bit compared
          if (b.vb[k] ≡ 0 )
          {  b.vb[k] = 1;  b.sp[k] = a; // node a is pseudo node
             NewNode = malloc ( Node(0) ); // create a new node
             Insertion ( NewNode, b.sp[k], ++Level ); // link to sub pointer
          }
          else Insertion ( a, b.sp[k], ++Level ); // link to sub pointer
          return;
      }
      case 2: px_remain ≡ 1 // 1-bit difference between prefixes
      {                                           node  next
                                                  type  hop   eh0  eh1
           if ( b.nt ≡ 0 )
               transfer b to node type 1;  // b = Node( 1 , hb , 0 , 0 )

           if ( PosVal (gPrefix, gLength-1, gLength-1 ) ≡ 0 )
               b.eh0=gNexthop;
           else
               b.eh1=gNexthop;
           delete a; // release node a
           return;
      }
      case 3: px_remain ≡ 2 // 2-bit difference between prefixes
      {
          k=PosVal ( gPrefix ,8+Level*2,8+Level*2+1 );
          if ( b.vb[k] ≡ 0 )
          { b.vb[k] = 1;  a.h = gNexthop;
            b.sp[k] = a; // link to sub pointer
          }
          else
          { b.sp[k].h = gNexthop; // update next hop of node b
            delete a; // release node a }
          return;
      }
} // end Function
```

Figure 6. FTBT Insertion Algorithm

Figure 6 shows the FTBT node insertion algorithm *Insertion*(a, b, *level*), where variable a represents the desired inserted node, variable b represents insertion location, and *level* is the level of node b in the tree. The root node is at level zero. Since one FTBT node represents two-bit length of prefix, the prefix length of the current node is given by 8+level*2.

In the insertion algorithm, the *px_remain* parameter denotes the number of bits in the prefix that have not yet been processed, up to the current level of the FTBT tree. If *px_remain* is greater than 2-bit, then the routing entry a must be inserted in the b-node's branch according to the value k in Fig. 6. If *px_remain* = 1, then the routing entry is inserted in *eh0* or *eh1*, depending on whether the remained value is 0 or 1. Finally, If *px_remain* = 2, then the routing entry is also inserted in the branch of node b based on the value k in Fig. 6.

```
Function Lookup (IP, x, Level)
{
        if ( x.h≠0 ) next_hop = x.h;

        if ( PosVal ( IP, 8+Level*2, 8+Level*2 )≡0
                & x.nt≡1 & x.eh0≠0 )
                    next_hop = x.eh0;

        if ( PosVal ( IP, 8+Level*2, 8+Level*2 )≡1
                & x.nt≡1 & x.eh1≠0 )
                    next_hop = x.eh1;

        k = PosVal ( IP, 8+Level*2, 8+Level*2+1 )
        if ( x.vb[k] = = 1 ) // link to sub pointer
           Lookup (IP, x.sp[k], ++Level )

        return next_hop;
} // end Function
```

Figure 7. FTBT Lookup Algorithm

Figure 7 shows the lookup algorithm *Lookup*(*IP*, *x*, *Level*), where *IP* is the destination address (DA) of the incoming packet, and *x* is the current node of the FTBT tree.

Using the FTBT tree in Fig. 4, the IP address “11111111010100…0” is now looked up. The first 8-bit value of the IP address is extracted as the index of the RTB. A root node whose next hop is C is then obtained. To reach the LPM (longest prefix matching), the next hop E is finally obtained.

The routing table update algorithm in FTBT is implemented by the remove algorithm, *Remove*(*px*, *x*, *Level*) as depicted in Fig. 8 to avoid reconstructing the FTBT tree. As shown in Fig. 8 shows, *px* is a prefix value of the desired removal routing entry, *x* is the current traced location in FTBT tree, and *x’s parameters* denotes information related to *x*, such as whether *x* has an expanded next hop or sub-pointer to another branch.

For instance, in Fig. 9, if consider the removal of a routing entry with prefix value “10001111 0001* and a prefix length of 12. The root node is determined via RTB as shown in Fig. 9 (a), and then the FTBT tree is searched until the current location *x* is node K, as in Fig. 9 (b). Then the *x.h* is then set to zero. Furthermore, the *x’s parameters* must be checked to decide whether node K can be removed directly. When the node is removed, *x* returns to the previous node, as shown in Fig. 9 (c). The *x’s parameters* is checked continually until the current location x stops at node J, as shown in Fig. 9(d).

```
Function Remove (px, x, Level)
{
     px_remain = gLength - ( 8+Level*2 );
     case 1: px_remain ≧ 2
     {
          k=PosVal( px, 8+Level*2 , 8+Level*2+1 );
          if (x.vb[k]≡1 )
          {
             Remove (px, x.sp[k], ++Level)
             x.vb[k] = 0;
             if ( x’s parameters≡0 )
             {  delete x; // release node x
                return; }
          }
          return 0; // Exit Function
     }
     case 2: px_remain ≡ 1
     {
          if ( PosVal(px, gLength-1, gLength-1 )≡0 )
               x.eh0 = 0;
          else
               x.eh1 = 0;
          if ( x’s parameters≡0 )
          {  delete x; // release node x
               return;
          }else if ( x.eh0 & x.eh1≡0 )
               transfer x to node type 0; // x = Node( 0 , h_x )
                                                   (node type ↓  next hop ↓)
          return 0; // Exit Function
     }
     case 3: px_remain ≡ 0
     {
          x.h = 0;
          if ( x’s parameters≡0 )
          {  delete x; // release node x
               return;
          }else
               return 0; // Exit Function
     }
} // end Function
```

Figure 8. FTBT Remove Algorithm

[Routing Table]

Prefix	Length	Next Hop
10001111*	8	J
10001111 0001*	12	K

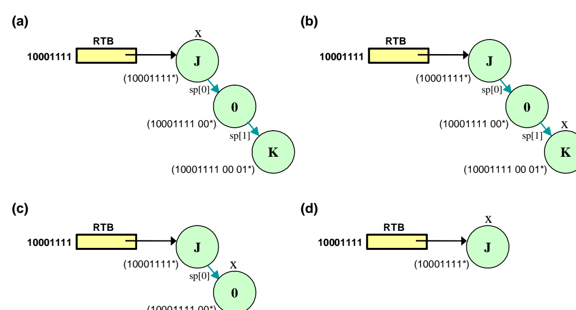


Figure 9. FTBT Remove Example1

Figure 10 illustrated another example of routing entry removal. In this case, two routing entries are matched and the prefix length differs by one-bit. Figure 10 (a) reveals that the root node must be found via RTB remove the prefix value “11110000 1*”. In this case, *px_remain* = 1. Since the 9th bit of the prefix value is “1”, the *x.ehl* of the current location must be removed. Once the *x.ehl* field is clear, the current node *x's parameters* is checked, and the node *x* is changed to node type 0. The associated information of node *x* at this time is shown in Fig. 10 (b).

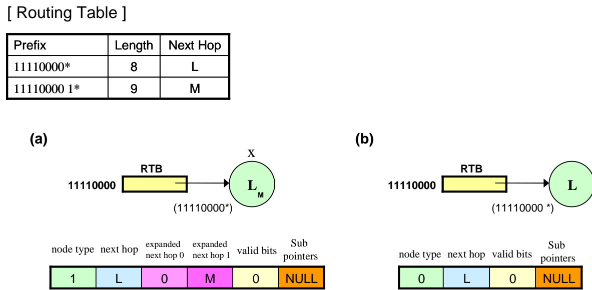


Figure 10. FTBT Remove Example2

4: PERFORMANCE ANALYSIS

Performance of the proposed FTBT scheme is evaluated. The number of total nodes, number of dummy nodes, maximum memory access times, average memory access times and required memory size of the proposed FTBT scheme are simulated and compared with Tree [7], Patricia [8], Prefix Tree [9], LPFST [10] based on the AS3303 routing table, which has 78504 routing entries [11].

Table I: Performance Comparison

	Tree	Patricia	Prefix Tree	LPFST	FTBT
total nodes	201181	147846	78470	75682	109248
dummy nodes	122677	69342	0	0	30744
Max. memory access	28	25	25	24	9
Avg. memory access	21.7	19.9	19.7	17.6	6.2
memory size(KB)	982.3	1588.1	766.3	815.7	1059.5

Table I presents the simulation results, which indicate the Tree [7], Patricia [8], and proposed FTBT schemes require extra memory space to maintain the dummy nodes. Because an FTBT node is two bits long, and to reduce the routing table update complexity (i.e., require extra dummy nodes), FTBT has a larger memory requirement than other schemes, as shown in Table I.

Comparing the average and maximum memory access times reveals that the proposed FTBT performs better than the other schemes. The use of the range table, and the two-bit node representation, both significantly reduce the height of the FTBT tree. For N number of prefix table, the proposed FTBT can reduce the number of memory access times from $O(w)$ to $O(\frac{w-8}{2})$ with a range table size of 2^8. As shown in Table I, the average number of memory access times for one IP address lookup in a routing table with 78504 entries environment is six. FTBT has a significant performance improvement, but has a larger memory size requirement than other schemes.

5: CONCLUSION

This study has proposed a practical software-based scheme called Fast Twin-bit Tree (FTBT) for IP address lookup. The proposed scheme adopts three routing entries to represent one FTBT node, thus reducing the tree height effectively. A range table is adopted to divide a large routing table into a multiple FTBT tree, and to reduce the search space. The height of each FTBT tree in the IPv4 environment is only 12 in the worst cast.

The FTBT IP lookup algorithm can effectively reduce the number of memory access times and provide fast routing table update operations. Performance simulation reveals that the FTBT algorithm requires an average of only six memory accesses when applied to a routing table with 78504 entries.

REFERENCES

[1] Y. Rekhter and T. Li, “An Architecture for IP Address Allocation. with CIDR,” RFC 1518, 1993.

[2] P.Gupta, S. Lin, and N. McKeown, “Routing Lookups in Hardware at Memory Access Speeds,” *in Proc. IEEE HPSR2004*, pp. 1240-1247, 1998.

[3] A. McAuley and P. Francis, “Fast routing table lookup using CAMs,” *in Proc. IEEE INFOCOM*, pp. 1382-1391, 1993.

[4] Ravikumar, V.C.; Mahapatra, R.N.; “TCAM architecture for IP lookup using prefix properties,” *IEEE Journal on Micro*, pp 60-69, 2004.

[5] D. Yu, B. C. Smith, and B. Wei, “Forwarding engine for fast routing lookups and updates,” *in Proc. IEEE GLOBECOM*, pp. 1556-1564, 1999.

[6] H. Lim and Y. Jung, “A Parallel Multiple Hashing Architecture for IP Address Lookup”, *in Proc. IEEE HPSR2004*, pp. 91-95, 2004.

[7] Sklower, K., "A Tree-Based Routing Table for Berkeley Unix," in *Proc. USENIX Conf.*, pp. 93-99., 1991.

[8] D.Morrison, "PATRICIA- Practical Algorithm To Retrieve Information Coded in Alphanumeric," *Journal of the ACM*, pp.514-534, 1968.

[9] M. Berger, "IP lookup with low memory requirement and fast update," *IEEE Conf. on High Performance Switching and Routing*, pp. 287-291, 2003.

[10] Lih-Chyau W., Kuo-Ming C., Tzong-Jye L., "A Longest Prefix First Search Tree for IP Lookup," *in Proc. IEEE INFOCOM*, pp. 989-993, 2005.

[11] BGP Table, http://bgp.potaroo.net/

[12] The Routing Arbiter Project, Internet routing and network statistic, http://www.re.net/statistics/

[13] Huang, N.-F., and Zhao, S.-M., "A Novel IP-Routing Lookup Scheme and Hardware Architecture for Multigigabit Switching Routers," *IEEE Journal on Selected Areas in Communications*, pp. 1093-1104, 1999.

[14] Yazdani, N., Min, and P.S., "Fast and scalable schemes for the IP address lookup problem, "*IEEE Conf. on High Performance Switching and Routing*, pp.83-92, 2000.

[15] Sharma, S., Panigrahy, R., "Sorting and Searching using Ternary CAMs, " *IEEE Conf. on High Performance Interconnects,* pp.101-106, 2002.

[16] Chang, R.C., Lim, B.H., "Efficient IP routing table lookup scheme, " *in Proc. IEE on Communications*, pp. 77-82, 2002.

[17] Zhen Xu, Damm, G., Lambadaris, I., Zhao and Y.Q., "IP packet forwarding based on comb extraction scheme," *in Proc. IEEE INFOCOM*, pp. 1065-1069, 2004.