A Scalable Exact matching in Balance Tree Scheme for IPv6 Lookup

Qiong Sun[1] Xiaoyu Zhao[2] Xiaohong Huang[1] Wenjian Jiang[2] Yan Ma[1]

1. Beijing University of Posts and Telecommunications, Beijing, 100876, P.R.China

2. France Telecom Research and Development Beijing, Beijing 100080, China

1. {sunq, huangxh, mayan}@buptnet.edu.cn

2. {xiaoyu.zhao, wenjian.jiang}@orange-ftgroup.com

ABSTRACT

Recently, the significantly increased IPv6 address length has posed a greater challenge on wire-speed router for IP lookup. As a result, even the most efficient IPv4 lookup scheme can not meet the demand in IPv6.

In this paper, we make a thorough study of real world IPv4/IPv6 routing tables and find out the useful characteristic of leaf nodes for the first time. The leaf nodes can be arranged in a single balance tree and thus change the LPM (longest prefix matching) model to exact matching one in routing lookup. This exact matching model can not only reduce the number of searching keys, but can also reduce the memory cost and support fast update. What's more, the searching procedure can stop immediately when meeting a match.

The balance tree in our scheme is a general concept. Here, we implement with three typical trees: B-tree, red black tree and avl tree and make a detailed comparison from every aspect. The experimental results show that its average lookup speed and memory cost is less than one third of the newly proposed range-based algorithm PIBT[1]. And among these three balance tree schemes, avl tree has the best lookup speed and memory consumption while B-tree scheme has the least update time.

Categories and Subject Descriptors

C.2.6 [**Internetworking**]: Routers.

General Terms

Algorithms, Performance, Design.

Keywords

lookup; balance tree; exact matching

1. INTRODUCTION

Internet Protocol Version 6 (IPv6) can provide extremely large pool of address space for the Internet, but it also poses a great challenge on the data path functions of IP address lookup in a router. Currently, most IPv4 routing lookup algorithms are length sensitive and their performance will descend dramatically with the longer address length. As a result, how to handle these larger routing tables and wider address length efficiently has become a major problem in the design of lookup engine.

In general, the metrics taken into consideration for a lookup algorithm are usually lookup speed, memory cost and update time. Lookup speed is obviously the most significant one with the development of fiber-optic technology. At the same time, the memory consumption and the update speed is also very crucial for a backbone router. We should keep in mind that lookup function is just one part of a router and it will share memory bandwidth and memory bits with other features, like VPN, MPLS, IPv4/IPv6, etc. Therefore, the memory cost for a lookup algorithm is worth taking into account. Besides, not only does the instabilities in backbone protocols can cause frequent prefix insertion or deletion(more than 1000 per second[2]), but also the multicast forwarding support requires route entries to be added as part of the forwarding process. Therefore, static lookup scheme is not quite useful in practice.

The IP lookup algorithms proposed so far can be broadly divided into trie-based, range-based and hash-based and they can implement on software or hardware or both. Hardware solution like TCAM, ASIC can search the contents in parallel while software solutions can benefit from low cost, flexible and scalability.

- trie-based algorithm

The original binary trie organize the prefixes with the bits of prefixes to direct the branching. However, since it’s worst case memory accesses can be so high that many other techniques have been integrated to reduce the height of trie (like Patricia[3], multibit trie[4], LC trie[5],etc), but the usage of these techniques often lead either to hard-updating or memory expansion. Although some of them are very attractive in IPv4, their worst case performance grows linearly with address length, which makes them not so absorbing in IPv6.

All these algorithms mentioned above search shorter prefixes first and the searching procedure can not be stopped until to the leaf nodes in the trie. The latest longest prefix first search tree (LPFST[6]) is firstly constructed with longer prefixes on the upper level and shorter prefixes on the lower level. And there is no memory waste in the tree. This scheme scales well when the number of prefixes is not large.

- range-based algorithm

As a prefix by its nature can be represented by the start point and end point, some algorithms have been introduced to search with these endpoints. In [7], it uses binary search on these endpoints but the precomputation made it really hard to update. In

IPv6'07, August 31, 2007, Kyoto, Japan.

[8], a new algorithm is proposed to improve the performance by using B-tree to prevent precomputation but every prefix may be stored in several nodes. Then in [1], a new algorithm is again proposed to improve the performance by storing exactly one node for each prefix. All these range-based algorithms can achieve the worst case lookup performance of O(logN) (N is the number of prefixes in a routing table) with no sensitive to key length, so they are rather attractive in IPv6. Furthermore, although a single 128-bit address still needs four memory accesses with 32-bit wide machine, the performance of range-based algorithm will be enhanced greatly with wider machine coming soon. Therefore, range-based algorithms will be more attractive in the future.

However, all the ranges in the algorithms mentioned above are not disjoint prefixes and they have to cope with overlapping situation either by some additional vectors (equal vectors and interval vectors) in [1] or percomputation in [7]. These methods not only waste a lot of memory but also almost double the number of keys needed to insert or delete because every prefix needs two vectors in [1]. As a result, how to make use of the original advantage of range-based balance tree is extremely important.

- hash-based algorithm

Hash has an outstanding feature of nearly O(1) run-time and is a wildly used technique in IP lookup [9, 10, 11]. So far many network processors have also build-in hash unit in it. Waldvogel in [11] proposes a binary search on length-based hash tables which can have the worst cast memory accesses of O(logW), where W is the number of distinct prefix lengths in the routing table. Nevertheless, it includes a lot of pre-computation leading to difficulty-update.

In this paper, we develop a novel IPv6 lookup scheme which can achieve all the features mentioned earlier. The main contributions of this paper are fourfold. First, we introduce a series of concepts and lemmas which connect trie-based aspect and range-based aspect in a new explicit view of IP lookup problem. Second, we analyze real-world IPv4/IPv6 BGP routing tables using trie technology and discover that the leaf nodes prefixes in a trie takes up to more than 90 percent of total prefixes. These prefixes are disjoint ones which can be organized in a sole data structure. And third, we put these leaf-node prefixes in a balance tree which can use the original outstanding feature of range-based balance tree and the rest prefixes in LPFST. In our range-based balance tree, the commonly searching model (most specific range) has turned to be an exact matching one. So the searching process can stop immediately when meeting a match. As a result, there is no memory waste and no precomputation needed. Moreover, the insertion or deletion does not need doubled workload for every prefix anymore. Finally, we also make a deep comparison with three typical balance trees: btree, red black tree and avl tree. Although all these balance trees have the same lookup and update performance of O(logN), their average performance in reality are quite different. The conclusion is very useful for future algorithm design.

The rest of the paper is organized as follows. Section 2 presents several trie-associated and range-associated concepts and terms. In section 3, we present the informative observations on real world IPv4/IPv6 BGP tables. In section 4, we discuss about our main data structure and the detailed way of searching, inserting and deleting. In section 5, we present several improvements for our scheme. In section 6, we present our experimental result of our three schemes together with PIBT, patrica, LPFST schemes. Finally, a conclusion is drawn in section 7.

2. DEFINITIONS AND LEMMAS FOR IP LOOKUP

➢ *Definition: address/mask-length start/end point*

With the introduction of CIDR, an entry in a routing table represents a continuous set of addresses and there are two ways to denote the set: address/mask-length and start/end point.

For example, assuming the address length is eight and there are four continuous addresses: 01000100, 01000101, 01000110 and 01000111. These addresses can be represented by 010001**/6 and [01000100, 01000111].

➢ *Definition: trie,genuine node, inner node, leaf node*

Trie is a binary search tree used for storing routing prefixes. A prefix of the routing table defines a path in the trie which begins from the root and ends in some node. The genuine node is the node corresponding to true entry while the valid node is just a path node. Here, every leaf node in the trie must be a *genuine node* and we define the *inner node* as the genuine internal node.

Every genuine node in trie (in darker color) can be represented by both address/mask-length and start/end point ways (as illustrated in table 1).

From range aspect, *leaf node* is the most specific range in one branch. The deeper the node is, the more specific range it will be. In addition, any ranges of the two prefixes located on separate branches would not overlap with each other, e.g, prefix A and prefix D in Figure.1.

No	Prefix	Next Hop
A	00*	N1
B	01*	N2
C	11*	N3
D	0100*	N4
E	0101*	N5
F	1011*	N6
G	010101*	N7

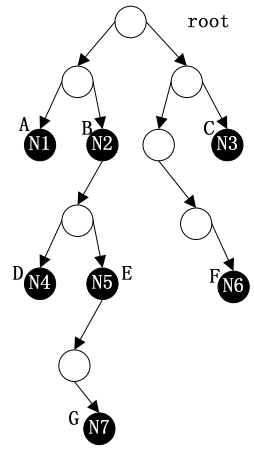

Figure 1. Example of a binary search trie

Table 1. Two Ways to Represent Ranges

Prefix No	addree/mask-length	start/end point
A	00*/2	[00000000, 00111111]
B	01*/2	[01000000, 01111111]
C	11*/2	[11000000, 11111111]
D	0100*/4	[01000000, 01001111
E	0101*/4	[01010000, 01011111]
F	1011*/4	[10110000, 10111111]
G	010101*/6	[01010100, 01010111]

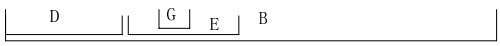

Figure 2. Relationship of The Sub-tree Under Prefix B

- Definition: contain, contained, disjoint

For IP prefixes, there are three range relationships: contain, contained and disjoint. Suppose there are two prefixes: X and Y.
If left(X) ⩽ left(Y) ∩ right(X) ⩾ right(Y), X contain Y.

If left(X) ⩾ left(Y) ∩ right(X) ⩽ right(Y), X is contained by Y

If left(X) > right(Y) U right(X) < left(Y),X and Y are disjoint.

Figure. 2 depicts these range relationship of the sub-tree under prefix B. We can see that prefix E is contained in prefix B and it contain prefix G. Prefix D and prefix E are disjoint prefixes.

- *Definition: level*

From trie aspect, we can determine the contain and contained relationship for each genuine node. The level is defined as the number of contained nodes along its corresponding path. Prefixes in the same level are also disjoint prefixes.

For example, prefix A, B, F, C are at level 0 as they have be contained by no prefix. Prefix D, E are at level one as they can be contained by one prefix in its branch.

- *Lemma1:* All the leaf nodes in trie are disjoint prefixes.

  Proof: omit

- *Lemma2:* If there are disjoint prefixes X and Y and another prefix Z which can be contained by X or Y, then at most one prefix can contain prefix Z.

Proof: If prefix X and Y can both contain prefix Z, then the range of X and Y must have the common part, which is the contradiction that prefix X and Y are disjoint prefixes.

- *Lemma3*: If there is a disjoint prefix X and another prefix Z which can be contained by X, then X is the prefix whose left point is the largest one among those disjoint prefixes whose left point is smaller than Z.

Proof: If there is another disjoint prefix Y whose left point is larger than X, then the left point of Y is no less than the right point of X. As a result, prefix Y is not among disjoints prefixes whose left point is smaller than Z.

- *Lemma4:* If prefix X is the longest prefix match (LPM) of prefix Y, then the range of X is the most specific range along branch of Y.

Proof: If there is prefix Z other than Y whose range is more specific than X along the branch of Y, then the length of Z must be longer than Y, which is a contradiction to the definition of LPM.

- *Lemma4:* When deleting a leaf node in a trie, there will be at most one new leaf node, which must be the longest prefix match (LPM) of the leaf node.

Proof: When deleting a leaf node in a trie, there may no new leaf nodes at all (take Figure.1 for example, when deleting G, E will become a new leaf node while deleting F or D, there will be no new leaf node coming out.).

If there are new leaf nodes existing, there will be at most one node. It is because that if there are more than one prefixes, these prefixes must all contain the leaf node before deleting. Then this is the contradiction with lemma2. As proofed in lemma3, LPM is the most specific range along the branch of the leaf node after deleting. As a result, it will become a new left node.

3. FEATURES OF REAL-WORLD ROUTING TABLE

IPv6 is just emerging recently, so the features of IPv6 BGP routing table is not enough to forecast the future. But as IPv6 will share some common features due to the allocation policies, routing practices, and the evolution of the Internet, the future IPv6 routing table is approximately predictable. Not only does the overall network topological distribution, which affects routing, is expected to be intact of the Internet will continue, but also the business relationships among IP prefix providers and the customers remain the same. As a result, we can predict the feature of further IPv6 routing table based on the new policy of IPv6 and the existing features of IPv4.

3.1 Existing ways to analyze a routing table

Here, we seek to discover a new feature of the disjoint prefixes in routing table. Previously, many people have analyzed the routing table from length distribution aspect, level distribution aspect, etc. Either prefixes with the same length or prefixes in the same level are disjoint prefixes, but they have the following problems:

- *Prefixes with the same length*

If we search the prefixes in each length in sequence, the performance will descend linearly to at most 128 in IPv6. If we search the prefixes binary in the length just as [11] did, we will need a lot of percomputation, which will make updating impossible.

- *Prefixes with the same level*

Accordingly, to search the prefixes with each level is much more attractive because the number of levels is not large (only 6 or 7 currently). But the hierarchy relationship may change when updating, which may result in transferring a lot of prefixes from one level to another. In this case, the worst case performance is O(NlogN) for update.

3.2 IPv4 and IPv6 address allocation policy comparison

In IPv4 and IPv6, they both use hierarchical allocation policy. The Internet Assigned Numbers Authority (IANA) is at the highest level and it delegates blocks of addresses to Regional Internet Registries (RIRs). These, in turn, allocate portions of their address blocks to Local Internet Registries (LIRs). With few exceptions, LIRs correspond to ISPs. In turn, LIRs assign parts of their address space to end-users.

We can see clearly from these two policies that IPv6 has fixed length allocation to get better aggregation effect. But the fixed length policy is somewhat like the policy before CIDR except that

it can aggregate after assigned. And it is well known that multi-homing and load balancing which has been the fastest growing factor in IPv4 routing table will avoid aggregation also in IPv6. As a result, we can predict that in future, 1) /48 in IPv6 will be in majority just as /24 in IPv6, followed by /64 in the second place 2) /32 will be the third major prefixes but the number will be distinctly less than /48 and /64.

3.3 IPv4 leaf nodes analyze

In this paper, we use trie technology to arrange the prefixes in routing table. We account the proportion of leaf nodes in total nodes (true nodes). These BGP tables are all from backbone routers with AS number of as4637, as1221 and as6447 on 2006, Jan. 02. The results are almost the same: in all these BGP tables, the leaf nodes are beyond proportion of 90% of total prefixes. Table2 shows the results.

Table 2. IPv4 leaf nodes percentage

As No.	leaf nodes	total number	leaf percentage
as4637	145093	158785	0.9147
as1221	146680	160434	0.9142
as6447	168557	184252	0.9148

3.4 IPv6 leaf nodes analysis

For IPv6 routing table, we also use trie technology to arrange the prefixes. The proportion of leaf-node number and total number are shown in table3.

Table 3. IPv6 leaf nodes percentage

As No.	leaf nodes	total number	leaf percentage
Japan	587	606	0.9639
potaroo	612	682	0.8988
London	622	660	0.9424
usa	627	665	0.9428
va	624	665	0.9383
tilab	612	652	0.9386

Although the current IPv6 BGP routing table only have hundreds of routing entries, the leaf node percentage in IPv6 similar with the result in IPv4.

4. BLANCE TREE WITH LPFST (BTLPT)

In this section, we discuss about our proposed scheme in detail. We first show the data structure of BTLPT and the reason we choose these data structures. Then we explicitly discuss about the lookup procedure, inserting procedure and updating procedure.

4.1 Data structure of BTLPT

As pointed before, we have found out that there is a majority of leaf nodes in real world routing table. The main motivation of introducing leaf nodes is as follows. First, all the leaf nodes are disjoint prefixes, so the LPM IP lookup model can be changed to an exact matching model in these leaf nodes. Second, all the leaf nodes are the most specific range in one branch and it matched perfectly with LPM motivation. As a result, the searching procedure can stop immediately when meeting a match Third,

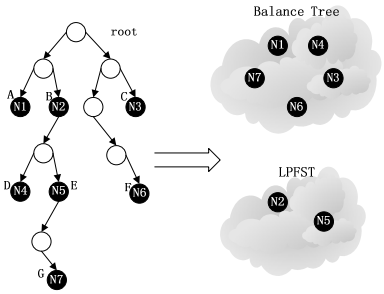

Figure 3. The ordinary trie based

unlikely to the inner nodes, every leaf can only be overlapped by at most one contained inner node while every inner node can overlap several leaf nodes, thus at most one node will be affected when updating and it will be updatable between these two parts. With these considerations, we make all the prefixes into two parts: **one part is leaf nodes** and **the other one is inner nodes.**

The following are the guidelines to choose data structures for these two parts of prefixes. First, these data structures should scale well in IPv6, especially the one for leaf nodes. Second, these data structures should consume no memory waste. Third, these data structure should support incremental update.

Based on these guidelines, we select balance tree to arrange leaf nodes and LPFST to arrange inner nodes. For one thing, balance tree is a very attractive data structure in IPv6 because of its O(logN) lookup performance and update ability. Moreover, it can gain much better performance when the prefixes in it are disjoint. Secondly, LPFST is also an absorbing data structure which can also stop immediately when meeting a match. Finally, both these two data structure are memory efficiency with no valid nodes.

As the leaf nodes are disjoint prefixes, we arrange all these leaf nodes in a balance tree and the rest of inner nodes in LPFST illustrated in Figure.3.

We should keep in mind that these two data structures: balance tree and LPFST are totally separately. And the balance tree here is just a general concept which can be fit to any balance tree like B-tree, red black tree, avl tree, etc. As the number of leaf nodes greatly surpasses the inner nodes, the searching procedure will stop mostly in balance tree and its performance will play a major role in the overall performance.

4.2 Lookup procedure in BTLPT

The lookup procedure in BTLPT is rather simple. As the leaf nodes are the most specific range in one branch, we should first do our searching in the balance tree of BTLPT. If there is a match in the balance tree, the searching procedure can just stop. Otherwise, we continue to search the LPFST for inner nodes. For example, there is a prefix of 01010100. So we first search in the balance tree and find that it locates in the key of prefix G[01010100, 01010111], so the searching procedure can just stop immediately. If we search the prefix of 01010000, then it matches no prefixes in the balance tree. Hence we have to search in LPFST. The pseudocode of Lookup(x) is as Figure.4.

```
Function Lookup(x) //lookup function of BTLPT
{ next_hop ← default next-hop;//initialize the next hop result
  if lookup_balance(x, &next_hop) ≡ match
          return next_hop; // if it matches a leaf node, stop
  else
          lookup_LPFST(x, &next_hop)
          return next_hop;}// else, search the inner nodes
```

Figure 4. Lookup algorithm for BTLPT

The pseudocode Lookup_balance function depends on the different data structures of balance tree and it returns a bool statment indicating whether there is a match in the balance tree. Also, it will change the next_hop when meeting a match.

4.3 Insert procedure in BTLPT

Insert procedure in BTLPT is a little more complicated. The main problem is that insertion may cause an initial leaf node become an inner node. Thus the initial leaf node should be deleted in the balance tree and then inserted into LPFST.

For example, if we have inserted A, B and C sequentially. At this time all of these three prefixes are leaf nodes and they are inserted in balance tree. Then when we want to insert D subsequently, we first search the balance tree and find that B can contain D which means inserting D has made B be an inner node. As a result, we have to delete B in balance and insert it in LPFST.

The pseudocode of Insert(x, length) is as Figure.5. Here, the search_balance _stat function is to find out whether there is prefix in the balance tree that can contain the inserting prefix or there is prefix in the balance tree that can be contained by the inserting prefix. Since the nodes in balance tree are leaf disjoint nodes, there is only one node will change from leaf node to inner node when inserting. And the relationship of contain and contained can be ascertained by the contained prefix in balance tree of the inserting prefix. As either insertion or lookup in balance tree all have the worst case performance of O(logN), the overall performance is also pleasant.

4.4 Delete procedure in BTLPT

The main problem in deleting procedure is that an inner node may turn to be a leaf node. So we should not only determine

```
Function Insert(x) //insert function for BTLPT
{ contain = contained = root;
  stat = search_balance_stat(x, contain, contained);
  if(stat ≡ INNER_NODE) //if it is a inner node
     insert_LPFST(x); // insert in the LPFST
  else if((stat ≡ LEAF_NODE) && (contain != root)){
          //If there is leaf node contain the inserting node
     delete_balance(contain); //delete the contain node
     insert_balance(x); //insert the node to balance tree
     range_trie(contain, contain_trie);//change contain node
                    //from range format to address/mask format
     insert_LPFST(contain_trie)}//insert the contain node
  else if((stat ≡ LEAF_NODE) && (contain ≡ root))
          insert_balance(x);}//if it is a new leaf node
```

Figure 5. Insert algorithm for BTLPT

```
Function Delete(x)
{ delete_node = lpm = NULL;
  delete_node = search_balance(x); // check if it is a leaf node
  if(delete_node){ //if it is a leaf node
    search_lpfstlpm(x, &lpm); //find the LPM of deleting node
    stat = search_balance_stat(lpm, contain, contained); //find
                  //out whether there is new leaf node coming out
    delete_balance(x); //delete the leaf node in balance tree
    if(contained != x){ //if there is new leaf node
      insert_balance(lpm); //insert the new leaf node
      delete_LPFST(lpm);} //delete the new leaf node
  else
    delete  LPFST(x);}//if it is an inner node
```

Figure 6. Delete algorithm for BTLPT

whether there is a new leaf node but also delete the newly leaf node from LPFST. And then the new node is inserted into balance tree when existing.

Take deleting D for example, we first find out that B is the LPM of D and there is G which is contained by B. So there is no new leaf node coming out. However, if we want to delete G, as the LPM of G is D and there is no leaf node other than G is contained by D, then D become a new leaf node. However, if the deleting node is inner node, there will be no change on the relationship of leaf node and inner node. The pseudocode of delete(x, length) is as Figure.6:

We should keep in mind that there will be at most one leaf node coming out when deleting the leaf node. The worst case performance in balance tree is O(logN). As a result, the deleting procedure will at most affect only one inner node to change its state.

5. IMPROVEMENT

5.1 Two types of keys

In real world routing table, there is very few prefixes of length between 64 to 128, it is really a memory waste to store next 8 bytes for every key in the node. So we divide every key into two types: one for the length less than 64 and the other for the length longer than 64.

Just as shown in the above Figure7, type0 is for the prefix shorter than 64 and type1 is for the prefix longer than 64. As a result, the memory cost for most of prefixes will save by almost half in practice.

5.2 Two formats of keys

As discussed in section3, each range can be represented by the following two formats:

[start point, end point] and [start point/end point, length]

As every start point/end point has 128 bit long (or at least 64 bit long with the improvement above), it also consume much more memory to start two points for every node. This is one of the reasons that make is less attractive for range-based algorithm.

With disjoint prefix, we can use the second format. As a result,

type0	S point	End point	nh

type1	Start point	End point	nh

Figure 7. Improvement for two types of keys

type0	S point	len	nh

type1	Start point	len	nh

Figure 8. Improvement for two formats of keys

the type of key can be turned to the following format in Figure.8.

6. PERFORMANCE EVALUATION

We implement the above data structure and algorithm with standard C, and simulate the actual process of IPv6 router lookup under Linux on an Intel Pentium4 PC with CPU of 2.4GHz and 512MB RAM with real world IPv6 routing table and synthetic one million traffic.

Usually, a routing lookup algorithm can be evaluated by its lookup speed, memory cost, and update speed. Figure9 to Figure11 gives the measured average lookup time, memory cost and average update time for BTLPT with avl tree, red black tree, B-tree , together with LPFST, patrica and PIBT. We can see from the result that BTLPT has achieved much better performance than the latest range-based algorithm PIBT. As the number of search keys has reduced to only half of the PIBT, its average lookup speed and memory cost has been cut to only half of PIBT. Its average lookup time and update time is also the lowest compared to the best trie-based algorithm, i.e., LPFST and patrica, so far.

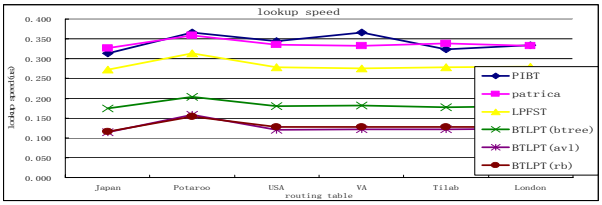

Figure 9. Lookup performance comparison

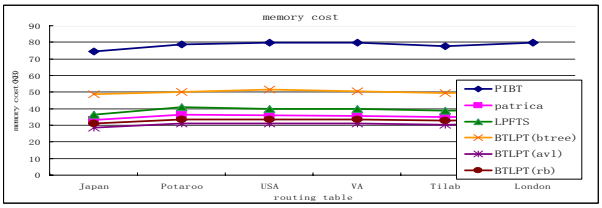

Figure 10. memory cost comparison

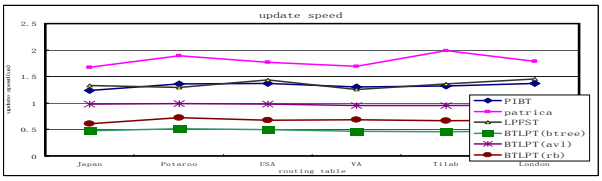

Figure 11. update performance comparison

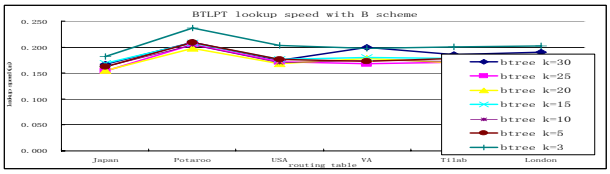

Figure 12. different key number lookup performance comparison for B-tree scheme

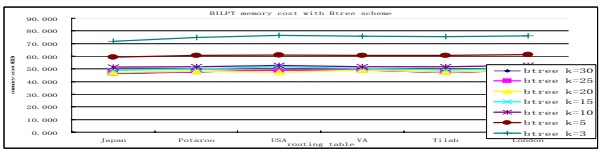

Figure 13. different key number memory consumption comparison for B-tree scheme

For BTLPT scheme itself, we can see that different balance trees can have different performance. Avl tree is a most strict balance tree whose heights of left subtree and right subtree differs at most by one. As a result, its average lookup performance is best among these three schemes. Red-black tree is a less strict balance tree and accordingly it can have much better update performance than avl tree. B-tree is a multiway tree with a handful of keys in each node. As a result, the number of keys in B-tree can affect B-tree scheme greatly. From Figure12 and Figure13, we can see that B-tree scheme can have best average lookup performance with key number of 15 while it can have the best average memory cost with key number of 25. This is a tradeoff for lookup and memory consumption.

7. CONCLUSION

The proposed BTLPT algorithm firstly collects all the majority leaf nodes in a single balance tree. With the useful characteristic of disjoint leaf nodes, BTLPT has gained much better performance than the latest range-based algorithm. In addition, we implement three typical balance tree on BTLPT scheme and the performance comparison is also given.

8. REFERENCES

[1] H.Lu, S.Sahni, “A B-Tree Dynamic Router-Table Design”, IEEE Trans.Computers,vol.54,pp.813-823, 2005

[2] C. Labovits, G. Malan, and F. Jahanian, “Internet routing instability” in SIGCOM’97, pp.115-126, Sep. 1997

[3] D.R. Morrison, ”PATRICIA – Practical Algorithm to Retrieve Information Coded in Alphanumeric” J.ACM, Vol.15, pp.514-534, Oct. 1968.

[4] P. Gupta, S. Lin, and N. McKeown, “Routing Lookups in Hardware at Memory Access Speeds” IEEE INFOCOM’98, pp.1240-1247, Apr.1998.

[5] S. Nilsson and G. Karlsson, “IP-Address Lookup Using LC-Trie” IEEE J. on Sel. Area in Comm, Vol.17, pp.1083-1092, June.1999

[6] L.C. Wnn, K.M. Chen and T.J. Liu, “A Longest Prefix First Search Tree for IP Lookup” in ICC’05, pp..989 – 993, May.16-20, 2005.

[7] B.Lampson, V.Srinivasan, and G.Varghese, “IP Lookups Using Multiway and Multicolumn Search,” Proc. IEEE INFOCOM’98, Apr, 1999,pp.1248-1256

[8] P.R.Warkhede, S.Suri, and G.Varghese, “Multiway Range Trees: Scalable IP Lookup with Fast Updates”, Computer Networks, vol.44, no.3,pp.289-303

[9] A. Broder and M. Mitzenmacher, ”Using Multiple Hash Funtions to Improve IP Lookups”, IEEE INFOCOM’01, pp.1454-1463, Apr.2001

[10] P.A. Yilmaz, A. Belenkiv, N. Uzun, N. Gogate and M. Toy,”A Trie-based Algorithm for IP Lookup” , IEEE GLOBECOM’00, pp.593-598, 27 Nov.-1 Dec

[11] M. Waldvogel, G. Varghese, J. Turner and B.A. Plattner “Scalable High Speed IP Routing Lookups”, ACM SIGCOMM’97, pp.25-36, Sept.1997