The CPBT: A Method for Searching the Prefixes Using Coded Prefixes in B-Tree

Mohammad Behdadfar and Hossein Saidi

Department of Electrical and Computer Engineering, Isfahan University of Technology
behdadfar@ec.iut.ac.ir, hsaidi@cc.iut.ac.ir

Abstract. Due to the increasing size of IP routing table and the growing rate of their lookups, many algorithms are introduced to achieve the required speed in table search and update or optimizing the required memory size. Many of these algorithms are suitable for IPv4 but cannot be used for IPv6. Nevertheless new efforts are going on to fit the available algorithms and techniques to the IPv6 128 bits addresses and prefixes. The challenging issues in the design of these prefix search structures are minimizing the worst case memory access, processing and pre-processing time for search and update procedures, e.g. Binary Tries have a worst case performance of O(32) memory accesses for IPv4 and O(128) for IPv6. Other compressed Tries have better worst case lookup performance however their update performance is degraded. One of the proposed schemes to make the lookup algorithm independent of IP address length is to use a binary search tree and keep the prefix endpoints within its nodes. Some new methods are introduced based on this idea such as "Prefixes in B-Tree", PIBT. But, they need up to 2n nodes for n prefixes. This paper proposes "Coded Prefixes in B-Tree", the CPBT. In which, prefixes are coded to make them scalar and therefore suitable as ordinary B-tree keys, without the need for additional node complexity which PIBT requires. We will finally compare the CPBT method with the recent PIBT method and show that although both of them have the same search complexity; this scheme has much better storage requirements and about half of the memory accesses during the update procedures.

Keywords: lookup**,** insert, delete, LMP, LPM**,** SMV.

1 Introduction

Because of the exponential growth of routing table size in IP routers, an addressing method named classless addressing was proposed in 1993. This addressing scheme didn't use the previous fixed length classes of IP address space and regardless of number of the bits; it used address prefixes with variable lengths up to 32 bits. This addressing mechanism caused new design considerations for routing table search and

A. Das et al. (Eds.): NETWORKING 2008, LNCS 4982, pp. 562–573, 2008.

update procedures. Route decision and forwarding of IP packets to their next hop routers based on the LPM[1] is called CIDR or the "Classless Inter Domain Routing".

Up to now, several algorithms are proposed for finding the LMP[2] of a database with an input address. The most popular known methods are PATRICIA [1], binary and compressed Tries [2,3], Gupta [4], CAM based algorithms [5], range based algorithms [6] and some other derivatives of these algorithms. The main problem of Trie based algorithms is their worst case search and lookup performance $O(W)$, with W considered as IP address length which is 32 for IPv4 and 128 for IPv6. Of course, for hardware based online lookup systems, the worst case lookup time and memory access would be the important factor during a burst of packet arrivals. CAM based algorithms are also facing the issues of their huge power consumption and small number of entries in the CAM modules. Range based algorithms construct their structure based on the range related to each prefix. For example if we consider W=5 and we have two prefixes 1* and 100*, their ranges would be {16, 31} and {16, 19}. Range based algorithms store prefix endpoints in their data structure [6], e.g. for 100* endpoints are 16 and 19 and in a range based algorithm, the information of both endpoints should be stored in the database. [6] proposed a range based algorithm which can use a binary tree for storing endpoints. In the worst case for n prefixes there exists 2n endpoints [6] and all of the 2n endpoints need to be stored in the tree. Because of unbalanced structure of binary search tree, the height of the tree will not be controllable in the worst case and this would result to bad search and update memory access performance. In [7] Yazdani and Min proposed a scheme and tried to improve the average lookup and update times however the worst case times didn't improve. To balance the tree and improve the worst case search operation, in [8], we proposed a new coding algorithm for the interpretation and storing the prefixes as numbers in the B-tree and using the scalar comparison during the search for the prefixes. Later, Lu and Sahni [9] have also used PIBT in a B-tree structure, using another method of separating prefixes into endpoints and store these endpoints using some vectors to facilitate the search and update procedures. Because of guaranteed maximum height of the B-tree structure, for both [8] and [9] the worst case memory access of search and update procedures will be limited to the height of the tree in comparison to the worst cases memory access in unbalanced trees. [10] has also used the B-tree but only to store disjoint IPv6 prefixes.

Using our idea in [8], with modifications to its search and update operations, here we introduce the CPBT; "Coded Prefixes in B-Tree", which simplifies the search and update operations and reduces the memory requirement by a factor of 4 in comparison to PIBT.

In what follows, the review of PIBT [9] is presented in section 2. In Section 3, we present the main idea of the proposed scheme as it is discussed in [8]. In section 4, the CPBT algorithm is described which modifies the main idea of [8]. Here it is described that without storing the actual encoded prefixes, the idea of the encoded prefixes might be used to compare prefixes during the search and update procedures. Additionally, enhanced version of the search and update procedures for this scheme is presented which reduces the update complexity of the algorithm. Finally in section 5, the memory requirements and update complexity of the CPBT are compared with those of the PIBT.

[1] Longest Prefix Matching.
[2] Longest Matching Prefix.

2 PIBT Review

Consider each node of a simple B-tree with "*t*" keys. The PIBT structure is made by the endpoints and some additional vectors called "Interval Vector" and "Equal Vector". Therefore for each node with *t* keys in the tree, three sets of parameters exist: *t* key values, *(t+1)* interval vectors, and *(t+1)* equal vectors i.e. *t*W + (t+1)*W*2* bits. It also separates each prefix into two end points and uses a key for each of them. Therefore for *n* prefixes it may use up to *2n* keys or 6 set of parameters.

An interval int(*x*) from the address space is allocated for each node. For the "Root" node, this interval, int(root) would be $[0, 2^w-1]$. Assume that *X* is a B-tree node with *t* keys and the information stored in the *X* has the following format:

$t, child_0, (key_1, child_1), (key_2, child_2), \ldots (key_t, child_t)$

Consider that $child_i$ is a pointer to *X*'s i'th sub-tree and assume that $key_1<key_2<\ldots<key_t$. Also assume that $key_0=start(int(x))$ and $key_{t+1}=finish(int(x))$. Then, using the previous definitions, for the $child_i$ subtree we have:

$int_i(X)=int(child_i)=[key_i,key_{i+1}]\ 0\le i\le t.$

In PIBT algorithm, each prefix specifies two end points called "key"s, then the set of all endpoints or keys are ordered and stored in the B-tree. Node *X* of a PIBT includes "*t+1*" *W*-bit vectors called $X.interval_i$. The bit *j* of this vector, $X.interval_i[j]$ is '1' if and only if a prefix with length "j" exists which its range contains $int_i(x)$. This rule for intervals is called "prefix allocation rule". In addition to $X.interval_i$ another vector named $X.equal_i$ exists in the structure. The bit "$X.equal_i[j]$ " is '1' if and only if a prefix with length "j" exists which has a start or end point equal to $key_i(X)$.

The search procedure is as follows: PIBT-search algorithm first makes a *W*-bit vector named *Match-Vector* with all bits set to '0'. If the router table has any prefix with length "j", with start or end point equal to the destination address using the *Equal Vectors*, then the *Match-Vector[j]* bit will be set to '1'. Otherwise the corresponding *Interval Vectors* of the current searching node are checked and if a match with length *j* is found, then the bit *Match-Vector[j]* will be set to '1', else this bit is not changed. After completion of the search in a node, the procedure goes through one of the child nodes. Insert and delete algorithms are similar to B-tree insert and delete procedures with the additional tasks of updating 2 sets of additional vectors: equal vectors and interval vectors, plus some modifications in the B-tree Split-child and node merge procedures. The node access complexity of this algorithm is $O(log_m(n))$ for the search, insert and delete, respectively [9].

3 Proposed Encoding of Prefixes, the Main Idea

In [8], we proposed a method of storing prefixes in a routing table in which prefixes look like numbers but not ranges. Consider a prefix *p* with *k* bits and *W* bit addresses. Each bit of prefix *p*, would be coded into a 2 bit value as follows: for each bit '1', 2 bits '10', for each bit '0' two bits '01' and for each blank place, 2 bits '00' are substituted. The number of blank places is '*W-k*'. For example if *W*=5:

0* → 01 00 00 00 00
010* → 01 10 01 00 00
001* → 01 01 10 00 00
0010* → 01 01 10 01 00
0001* → 01 01 01 10 00
00001* → 01 01 01 01 10

With the above definition, the prefixes can be compared like numbers and we'll have:

0* < 00001* < 0001* < 001* < 0010* < 010*

3.1 Using Proposed Encoding of Prefixes to Search for the Prefixes

As an example consider the B-tree of prefixes shown in figure 1 and assume the objective is to find prefix "00100*". Start from the root. Since "01*>00100*", traverse the tree from "01*" to C0. The key "00100*" will be found in C0. "A" shows the direction of the search. With the above procedure we can find any given prefix with a simple search, but the challenge would be finding Longest Matching Prefix (LMP) of an input destination address.

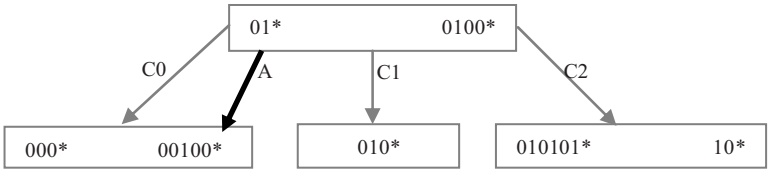

Fig. 1. Example B-tree for storing prefixes

Now, assume W=6 and an input destination address of "d=010100". The objective is to find LMP of d in the tree. Start from the root. Obviously, "d>01*" and "d>0100*" and also d matches with "01*". Therefore we should save "01*" as the LMP up to now and go through C2. In C2 there isn't any key matching with d. Therefore the LMP result for d would be "01*", however it can be seen from the figure that there is another key, "010*", in C1 that matches with d and its length is longer than "01*". So, this search algorithm for finding the LMP of an input destination address might miss the correct answer. This type of storing prefixes will be only useful when the prefixes are completely disjoint i.e. there aren't any 2 prefixes in the database while one is the prefix of the other. Our solution to the problem depends on a lemma, summarized here and proved in [8].

Lemma 1. Given a destination address, "d", with W bits, let's assume a prefix, "c" with j bits, which is located in a node of the B-tree and is currently being checked. Also assume another prefix, "k", with ℓ bits exists in the tree and is not found in the search yet.

a) If
- k is a prefix of c and d,
- $d>c$,
- k, c are not stored in one common node

Then k will not be in the search path and it will be missed.

b) The prefix k which is a prefix of d but is not a prefix of c, would be in the search path at this node.

3.2 A Method for Finding the LMP of a Given Destination Address

As described earlier, storing raw prefixes in the B-tree will cause some of them to fall out of the search path. To solve this problem, we introduce an additional vector stored with each key, called "Shorter Matching Vector" or SMV. This vector is W bits long. Consider a key "k" in the database and its SMV, if bit "i" of SMV is set to 1, it means that a prefix with length i exists in the tree which is a prefix of "k". From the definition of the SMV and Lemma 1, it is clear that if it can be updated correctly, it will prevent those mentioned prefixes to fall out of the search path [8].

4 CPBT

Here, the CPBT algorithm is described. CPBT is a modified version of the main idea in [8]. It is shown that without storing the actual encoded prefixes, the idea of the encoded prefixes might be used to compare prefixes during the search and update procedures. Additionally, an enhanced version of the search and update procedures for this scheme is presented which reduces the update complexity of the algorithm. Details of modifications are described in the following sub-sections.

4.1 Modification of the Main Idea of [8]

Instead of storing the encoded prefixes as it has been described in section 3, the prefixes might be stored as W bit vectors. For example, if W=6 and a prefix p=110*, a W bit vector '110000' should be stored instead of p. Additionally a 6 bits SMV is stored with this prefix, and its third bit (the prefix length) is set to one and its 4^{th}, 5^{th} and 6^{th} bits are set to zero. If this SMV key is read during the search algorithm, the rightmost 1 will identify the actual prefix and its length. In this example, if the SMV is "01**1**000", it means that the prefix is 110* (the rightmost 1, is the bolded bit in 01**1**000). Therefore, it is not needed to keep the prefix length as a separate entity.

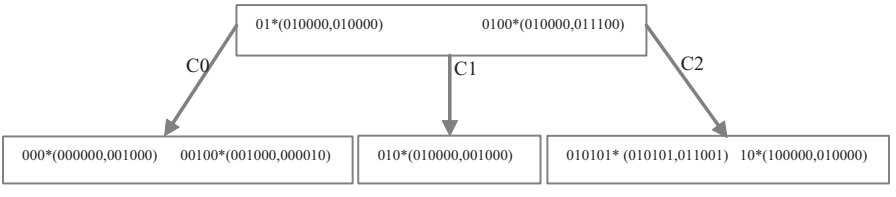

Fig. 2. Prefixes and their shorter matching vectors in the proposed structure in the form of prefix*(prefix, SMV)

Additionally, although keys are saved as *W* bit vectors, in comparing two keys, the same notion of "*encoding the prefixes*" to the "*numbers*" which was described earlier will be used. With the above modifications and using the pair entry of *(prefix, SMV)* and with the assumption of *W*=6, figure 2 shows the new tree compared to figure 1.

As an example, let's assume *W*=6 and an input destination address of "*d*=010100" is given to find the prefixes for it. Starting from the root and checking the left key of the root we find that 01* exists in the database. Then checking the right key of the root node we find that *d*>"right key" of the root and its SMV also shows that up to now the LMP of *d* is 010*. Since *d*>"right key", we should go through C2 but in C2 we will not find another match, so the LMP of *d* will be 010* and the search procedure is completed.

Next objective is to find a solution for updating SMVs at the time a prefix is inserted or deleted. In the next sub sections we will introduce methods for the update and search procedures. The proof of the correctness of these methods was omitted due to space limitations.

First of all, consider the following assumptions:

- The length of a prefix *p* is shown by *len(p)*.
- The i^{th} key in a node "*N*" can be showed by *N(i)*.
- The length of the IP Address is assumed to be: *W*
- The SMV vector of a prefix *p* is shown by SMV_p.
- Consider two prefixes, *p* and *x*. if *p* is a prefix of *x*, we use the following notation: $p \rightarrow x$ and if *p* is not a prefix of *x* we use this notation: $p \nrightarrow x$.
- If $p \rightarrow x$, the bit of SMV_x which corresponds to *p* is shown by $SMV_x(len(p))$.
- A prefix of *x* with length of *k* is shown by $len_k(x)$
- If the SMV of a prefix *x* overwrites the SMV of a prefix *y*, *x* is called the Master and *y* is called the Slave.

4.2 Insertion

Insertion of a new prefix is similar to B-tree insert of [8] and [11]. However updating the SMV is simpler than [8]. In general it may seem that updating a node key may result to update all its children keys, but, this is not true.

The Insert procedure is described as follows:

- The prefixes should be compared according to the concept of the coding introduced in section 3. Initially, the new adding prefix is compared with all the keys in the root to find its location among them. This comparison identifies the child and sub tree which this adding prefix would be added to.
- On the search path of inserting a prefix *p*, in the first node which contains any prefix *x* with the property of $p \rightarrow x$, the $SMV_x(len(p))$ will be updated to '1' for all the keys with the above property.
- If node splitting is required, the key "*s*" which is removed from the child node and added to the parent node will be the Slave in setting SMV vectors. It means that the keys of the father node and their SMV vectors are the Master and would update the SMV vector of *s*; however the SMV vector of *s* would not update them. It means that we assume that each father node's info will overwrite its child nodes info in search and update procedures.

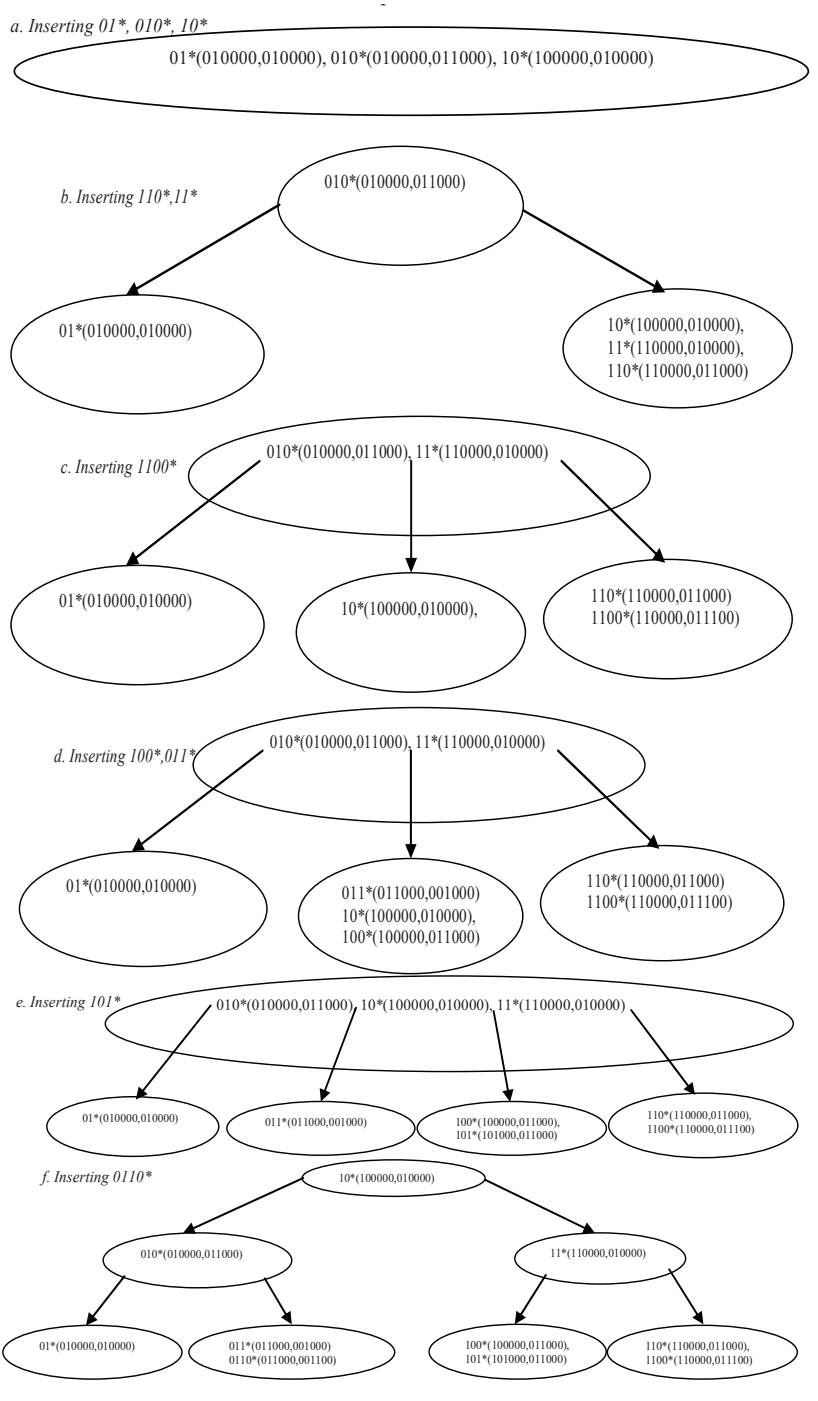

Fig. 3. An Example for insertion of prefixes and their shorter matching vectors in the form of prefix*(prefix, SMV)

- At last, after storing p in a node, for example node S, we will set the SMV vectors such that:
  - For each prefix x in node S, if $p \rightarrow x$ then $SMV_x(len(p))=1$.
  - For each prefix x in node S, if there is a length k such that the bit k of its SMV is already set to 1 (or $SMV_x(k)=1$) and if $len_k(x) \rightarrow p$ then it will set the $SMV_p(k)=1$.

As an example for insertion, consider a tree in which the following prefixes are added with the order of: *01*, 010*, 10*, 110*, 11*, 1100*, 100*, 011*, 101*, 0110**. Figure 3 shows the results of the insertion process.

As it is shown in figure 3, although 01* is a prefix of 011* and 0110*, the SMV vectors of 011* and 0110* are not updated for 01*. This shows that it is not necessary to update all of the SMV keys related to a prefix p in this new version of the algorithm.

Since it is needed to traverse the tree height for an insertion process, the memory access complexity will be $O(log_m(n))$ with n the total number of prefixes and m the order of the B-tree.

4.3 Deletion

The method which was introduced in [8] for delete procedure, has a complexity issue. If a prefix "p" needs to be removed from the database, all other prefixes that p is also a prefix for them must be checked and their SMV vectors should be updated. It means that in their SMV vectors, the bit which is related to p should be set to zero. As this method is time consuming, we introduce a new method that improves the delete performance.

The delete procedure for a prefix p is similar to the delete procedure of a B-tree [11] with additional updating of the SMV vectors. Therefore, the delete procedure and updating the SMV vectors are described as follows:

- Using the B-tree delete procedure [11], it needs to find the location of the deleting prefix. The prefixes should be compared according to the concept of the coding introduced in section 3. Initially, a deleting prefix "p" is compared with all the keys in the root to find its location among them. This comparison identifies the child that the path to the location of p will go through it.
- During the search for the prefix, some other prefixes which p is also a prefix of them may be found. If any of these prefixes is found in the search path, its SMV will be updated immediately. Otherwise, it means that either there is no prefix in the database which p is a prefix of it, or these prefixes are allocated after p in the search path. The regular delete operation of p checks the path for recursive delete [11] to either the predecessor or the successor of p. Therefore, if p is found in an internal node, it is necessary to search the path to the successor of p in the database. During the search to the successor, if other prefixes are found which p is a prefix of them, then its corresponding bit in their SMV vectors will be set to zero.
- In merge procedure [11], the key which is deleted from the father node and stored as a median key in a merged node, is the Master and will modify the SMV vectors of the other keys in the new merged node.

- In the Borrow key procedure [11], the key which is borrowed from a child node *x* to a father node F, is the Slave and the key which is deleted from the father node F and stored in the sibling of *x*, e.g. *y*; is the Master.

The above proposed function for delete operation may not update the SMV of all the prefixes which *x* is a prefix of them, but it can be proved that if there are any of them in the tree, then at least the one which is the closest to the root of the tree will be updated.

As an example of delete procedure consider the tree of figure 3. If 10* is deleted from this tree, then one of the trees shown in figures 4 and 5 would be resulted depending on the search order of its predecessor and successor.

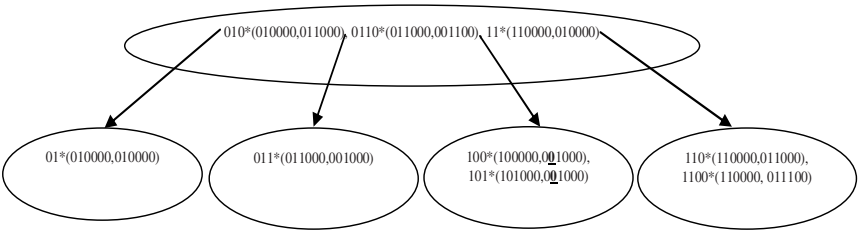

Fig. 4. An Example for delete of prefixes- prefix 10* is deleted from the tree of figure 3, checking predecessor first

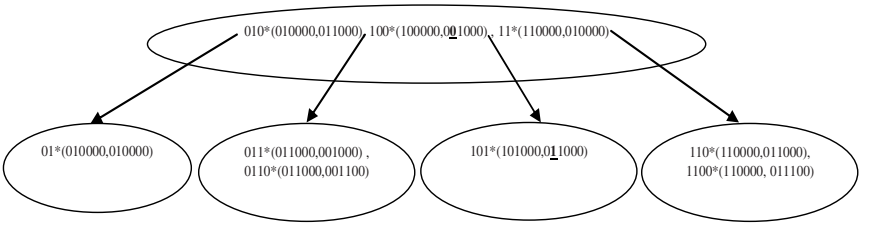

Fig. 5. An Example for delete of prefixes- prefix 10* is deleted from the tree of figure 3, checking successor first

In the first method which is used in figure 4, the predecessor of 10* replaces it. As it is depicted in this figure, the path to the successor prefix is checked as well. Then the corresponding SMV bits of 100* and 101* are converted to zero which are bolded in the figure.

In the second method which is used in figure 5, the successor of 10* i.e. 100* replaces it and its corresponding SMV bit is set to zero (the bolded bit). As it is shown in this figure, it is not necessary to modify the corresponding SMV bit of 101*. During future operations, if there is a need to move the 101* and its SMV to its parent location, its SMV vector would be updated based on its parent SMV vectors.

Similar to the B-tree delete operation, the memory access complexity of this delete strategy will be $O(\log_m(n))$, where n is the number of prefixes and m is the order of the B-tree.

4.4 Search of LMP for a Given Address

Now, the CPBT search procedure for finding the LMP of a given address might be revised as follows. Again, the search procedure is similar to the search procedure of a B-tree but the prefixes should be compared according to the concept of the coding introduced in section 3. Consider an input address d for the search. An SMV called SMV_d is defined to store the information about the prefixes of d. Initially all of its bits are set to *. The search is started at the root node. For any stored prefix x in the root node if $len_i(x)$ is a prefix of d, the corresponding bit in the $SMV_d(i)$ will be set to "0" or "1" based on the $SMV_x(i)$ and it will not change during the search operation anymore. This procedure will be repeated in the next nodes of the search path to update the remaining * bits. After completely traversing the tree, the search algorithm is finished and SMV_d identifies all of the available prefixes of d in the tree. At this time, the rightmost "1" of SMV_d represents the LMP of d.

As an example of the search procedure of a given address, consider $W=6$ and $d=100100$. The search procedure of d in the tree of figure 5, first looks up the root and finds $100*$ as an existing prefix. It also finds out that $1*$ and $10*$ do not exist in the tree. By continuing the search, the path should go through the third child of the root. But, the corresponding SMV bit of $10*$ is set to one in the prefix $101*$ of the third child. As the corresponding SMV bit of 10* is already set to zero in the root node, the recent result cannot overwrite it. Therefore the LMP of d will be $100*$.

Similar to the B-tree search operation, the search time is propotional to the height of the tree which is less than $log_m((n+1)/2)$ with n the number of prefixes and m the order of the B-tree.

4.5 The Final Theorem

The above insert, delete and search procedures result in a final theorem:

Theorem- Assume that a B-tree of prefixes is constructed with the definitions of insertion and deletion of a prefix that was proposed in sections 4.2 and 4.3. For an arbitrary destination address d, each prefix p that has the property of $p \rightarrow d$, will be found in the B-tree using the search algorithm proposed in section 4.4.
The proof of this theorem has been removed due to the space limitation.

5 The Comparison of the Proposed Algorithm and PIBT

As it was explained, almost all of the range based algorithms proposed up to now, need to store two end points for each prefix or at least they need to store more than a key. One of the latest proposed range based algorithms is PIBT which stores prefixes end points in a B-tree. PIBT needs to store two keys or endpoints for each prefix and also requires storing about two W bit vectors for each endpoint. In the proposed scheme of CPBT it is not needed to store the prefixes end points. It is just enough to store the prefixes themselves and use the proposed coding definition. Therefore, the number of keys in the tree will be half of the number of keys of PIBT. On the other hand we do not need to store 2 additional vectors for each key and only one SMV vector per key is sufficient. This reduces the required vector bits to 1/4. It means that

it requires 32 bits for IPv4 and 128 bits for IPv6, comparing to128 and 512 bits for PIBT, which is a good advantage for long IPv4 and IPv6 prefixes. Of course, other required memories to keep the pointers are ignored.

In PIBT, the update of additional interval vectors and the update of the keys will be performed separately. But in the proposed architecture, the update of a prefix and SMV vectors could be done simultaneously. In other words, the SMV vectors will be updated during the insert or delete procedure of a prefix and this means less number of memory accesses for each update.

Another advantage of this algorithm compared to PIBT is that we need to insert or delete only one prefix in each update process, however in the case of PIBT, it may require to insert or delete both two end points of a prefix which means traversing the tree twice.

As it is shown, the height of tree, h, will be the height of a B-tree, $h \leq log_m((n+1)/2)$ with n the number of the prefixes and m the order of the B-tree. Since the tree is balanced, the proposed algorithm in this paper has a guaranteed worst case search and update time for any prefix length distribution. Also, as a good property of this proposed algorithm, it doesn't need any pre-processing to be performed.

6 Conclusions

This paper introduced CPBT, a new algorithm for the prefix matching problem and also finding the Longest Matching Prefix or LMP. It treats prefixes as numbers but not ranges during the search procedures. This type of coding of prefixes has the advantage of removing the dependency to the prefix length during the search and update time and also in memory usage for the IPv4 or IPv6. Therefore, it is equally suitable for both IPv4 and IPv6. It is also possible to apply this scheme to the method in [10] for its extension to the IPv6. This scheme is compared with PIBT [9] and it is shown that although both of them have the same search complexity; this scheme has much better storage requirements and about half of the memory accesses during the update procedures.

References

1. Morrison, D.R.: PATRICIA Practical algorithm to retrieve information coded in alphanumeric. Journal of the ACM 15(14), 514–534 (1968)
2. Sirinivasan, V., Varghes, G.: Faster IP lookup using controlled prefix expansion. ACM Transactions on computer systems 17(1), 1–40 (1999)
3. Nilson, S., Karlsson, G.: IP address lookup using LC-tries. IEEE Journal of selected Areas in communications 17(6), 1083–1092 (1999)
4. Gupta, P., lin, S., McKeown, N.: Routing lookups in Hardware at Memory Access Speeds. In: Proceedings of IEEE Infocom vol. 3, pp. 1240–1247 (1998)
5. Shad, D., Gupta, P.: Fast Incremental updates on ternary-CAMs for routing lookups and Packet classification. In: Proceedings of Hot Interconnects VIII (2000), IEEE Micro. (2001)

6. Lampson, B., Srinivasan, V., Varghese, G.: IP lookups using multiway and multicolumn search. In: Proceedings of IEEE Infocom vol. 3, pp. 1248–1256 (1998)
7. Yazdani, N., Min, P.: Prefix Trees: new Efficient Data Structures for Matching Strings of Different length. In: Ideas 2001, pp. 76–85 (2001)
8. Behdadfar, M.: Review and Improvement of Longest Matching Prefix Problem in IP Network, MSC thesis, Isfahan University of Technology (2002)
9. Lu, H., Sahni, S.: A B-Tree Dynamic Router-Table Design. IEEE transactions on computers 54(7), 813–824 (2005)
10. Sun, Q., Zhaho, X., Huang, X., Jiang, W., Ma, Y.: A Scalable Exact Matching in Balance Tree Scheme for IPv6 Lookup. In: ACM SIGCOMM 2007 data communication festival, IPv6 2007 (2007)
11. Cormen, T., Leiserson, C., Rivest, R.: Introduction to Algorithms. Hill Book Company. McGraw-Hill, New York (1999)