Scalar Prefix Search: A New Route Lookup Algorithm for Next Generation Internet

Mohammad Behdadfar Hossein Saidi Hamid Alaei Babak Samari
Department of Electrical and Computer Engineering
Isfahan University of Technology
Isfahan, Iran
behdadfar@ec.iut.ac.ir hsaidi@cc.iut.ac.ir h.alaei@ec.iut.ac.ir b.samari@ec.iut.ac.ir

Abstract*- Currently, the increasing rate of routing lookups in Internet routers, the large number of prefixes and also the transition from IPV4 to IPV6, have caused Internet designers to propose new lookup algorithms and try to reduce the memory cost and the prefix search and update procedures times. Recently, some new algorithms are proposed trying to store the prefixes in a balanced tree to reduce the worst case prefix search and update times. These algorithms improve the search and update times compared to previous range based trees. In this paper it is shown that there is no need to treat the prefixes as ranges. It is only required to compare them like scalar values using a predefined rule. The method "Scalar Prefix Search" which is presented here, is built on this concept and combining it with the proposed store and search methods, interprets each prefix as a number without any encoding, the need to convert it to the prefix end points or to use the Trie based algorithms whose performance completely depends on IP address length. This method can be applied to many different tree structures. It is implemented using the Binary Search Tree and some other balanced trees such as RB-tree, AVL-tree and B-tree for both IPV4 and IPV6 prefixes. Comparison results show better lookup and update performance or superior storage requirements for Scalar Prefix Search in both average and worst cases, against current solutions like PIBT [1] and LPFST [2].

Keywords-Prefix; Scalar; Match; search; update; Lookup

I. Introduction

Since the conventional class-based IP addressing and routing [3] caused some problems like depletion of IP address space, a new routing method was introduced in 1993, called "Classless Inter Domain Routing" or CIDR [4]. This addressing and routing strategy doesn't use the original classes of IP address like class A, B or C. In CIDR, IP prefixes with variable lengths are used instead of these address classes. For example if two prefixes P_1, P_2 match with the destination address *d*, the one which has the greater length will be chosen and its corresponding next hop, will determine the outgoing port. The algorithms which find this longest prefix are called longest prefix matching algorithms. Many algorithms and data structures have been proposed since the introduction of CIDR. One of the first structures which were introduced was Trie. It is a simple tree that each of its nodes is representative of a string. The left child of this node adds a '0' to the string and the right child, adds a '1'. Fig.1 is a simple Trie with four prefixes: 01*, 010*, 011* and 0111*.

For *w* bit IP addresses, this structure has a main drawback which is its worst case memory access complexity for search and update procedures that is *O(w)*. After introduction of Trie, some improvements like LC-Trie were introduced to improve its memory access [5], however the worst case memory access complexity remained *O(w)*. After that, some algorithms were proposed to decrease the worst case height of the Trie. These algorithms were named Multi-bit Trie [6]. In Multi-bit Tries, the processing is done using more than one bit in each node. Therefore, the worst case height of the tree is decreased. However, the update time is increased and the memory is wasted [7]. In Multi-bit Tries the complexity still depends on IP address length. To solve this problem, range based algorithms were introduced. These algorithms define a range for each prefix and use the end points of this range. For example, if *w=4* and *p=10**, the range of *p* will be 1000~1011. Therefore these algorithms use both end points for each prefix and implement a binary search tree to store the end points in which the complexity of the lookup will be *O(log n)*. However updates will be inefficient and in the worst case, the tree should be constructed again [8]. Recently, a new range based algorithm is proposed that implements prefix end points in a B-tree (a kind of Balanced tree), called PIBT [1]. An advantage of PIBT is concurrent improving of the worst case lookup and update complexity. However it uses about 6 *w*-bit vectors for each prefix. Another algorithm is also introduced called BTLPT [9]. The authors of this algorithm have proved that the prefixes in the leaf nodes of the Trie are disjoint. Using this property, BTLPT has used two separate trees. One is a balanced tree to store the disjoint prefixes of leaf nodes and the other is a LPFST to store the remaining prefixes. The worst case performance of LPFST depends on *w* [2]. Therefore, the same would be true for BTLPT.

In contrast to the Trie based and range based algorithms, the method which is proposed in this paper, treats each prefix as a number. Importantly this makes the algorithm search and update times independent of IP address length. This property makes it superior for 32 bits IPV4 addresses and especially for 128 bits IPV6 addresses. The concept of comparing the prefixes as scalar values has been introduced in [10] using encoded representation of prefixes. But here extending the concept without the encoding scheme and proposing the store and search methods, it is shown how this concept can be implemented on many different trees. It is worth mentioning that the correctness of Scalar Prefix search is proved completely by authors. However, many parts of the proofs are removed because of the space limitation. In this paper, this Scalar Prefix Search algorithm is applied to BST (Binary Search Tree), B-tree, RB-tree (Red Black Tree) and AVL-tree. Finally, it is shown that the performance is better than current solutions like PIBT.

II. Scalar Prefixes

Scalar Prefixes refers to the concept of treating each prefix as a number. The method of comparison is similar to [10] and is described as it follows.

Assuming that the length of each IP Address is *w* and *len(p)* shows the length of a prefix *p*, in comparing two prefixes:

If *len(p)=len(q)* then *p* and *q* will be compared like two ordinary scalars.

Otherwise if they do not have equal lengths and if *len(p)>len(q)*, then two new scalars (binary numbers) using the first *len(q)* bits of each of *p* and *q* shown by *p[0:len(q)-1]* and *q[0:len(q)-1]* should be compared.

Then if $p[0{:}len(q)-1] \geq q[0{:}len(q)-1]$, *p* is greater than *q*, otherwise *q* is greater than *p*.

For example, using the above method and with four prefixes P_1=01*, P_2=010*, P_3=011* and P_4=0111*, the comparison will result to:

$P_1 < P_2 < P_3 < P_4$

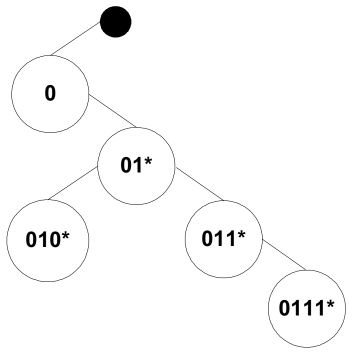

Figure 1. Trie Structure

III. Scalar Prefix Search Procedure

To describe the Scalar Prefix Search, here this method has been applied to a Binary Search Tree (SP-BST).

To describe the algorithm, special Insertion, Search and delete procedures are defined for the tree which is different from Binary Search Tree.

Please note that although each prefix would be stored in the form of two vectors called "Match Vector" and "key", the procedure may simply mentioned as storing the prefix instead of storing the key or Match Vector.

First let's define some notations:

- *Key*:
  -For each prefix *p*, we define a *w* bit *key* and this *key* is shown as *key(p)*. It will be inserted in the tree instead of the original prefix.
  -If *len(p)=k,* and *k<w*, for a *k*-bit prefix *p, key(p)* would be defined as:
  key(p)="p(0)p(1)p(2)…p(k-1)000…0" (where the number of zeros is *w-k,* e.g. if *w=4* and *p=101**, *key(p)="1010"*).
- *Prefix symbol* $\rightarrow$:
  -The notation $p \rightarrow q$ shows that *p* is a prefix of *q*.
  -The notation $p! \rightarrow q$ shows that *p* is not a prefix of *q*.
  -If $p! \rightarrow q$ and $q! \rightarrow p$, *p* and *q* are called disjoint prefixes.
- *Pref:*
  -A prefix of *p* with length of *k* is shown by $pref_k(p)$.
- *Longest Matching Prefix:*
  -The longest matching prefix of a string "*S*" is denoted as *lmp(S).*
- *Match Vector*:
  -For a key "*k*", a *w* bit "Match Vector" is also defined and abbreviated with "*k.mv*".
  -The *i*th bit of *k.mv* is called *k.mv(i)*. If *k.mv(i)=1*, it means that a prefix *q* with length *i+1*or *len(q)=i+1 or* $q=pref_{i+1}(k)$ exists in the database and $q \rightarrow k$. Please note that the bits numbers in the Match Vector starts form "0".
- *height*:
  -If *x* is a node of the tree, *height(x)* is defined as the length of the path from the root of the tree to that node, e.g. *height(root)* is zero, the height of each child of the root is "*one*" and so on.
- *Max-length Prefix*:
  -The longest prefix of each key in a node is called the "Max-length Prefix" of the key and is abbreviated by *MP(key)*. The largest *i* so that *key.mv(i)=1*, defines that the length of *MP(key)* is *i+1.*
- *Others()*:
  -If *MP(k)* is the Max-length prefix of *k,* then the set of all other prefixes of the key *k* whose corresponding bits in *k.mv* are set to *one*, is called *Others(k).*
- *Predecessor and Successor of a key*:
  -The "Predecessor key" of a node key in a tree, is the largest key in the left sub-tree of this key and the

"Successor key" of a node key in a tree, is the smallest key in the right sub-tree of this key.

Using these definitions, prefixes can be stored in any search tree including Binary Search Tree or BST.

A. *Insert Procedure* for SP-BST

To insert a "*newPrefix*", or to determine the insertion path, the algorithm starts from the "root node". Visiting any node in the insertion path in which a key *r* is stored and to make a decision on insertion or continuing on the insertion path, the "*newPrefix*" is compared with *r*.

If the "*newPrefix*" is a prefix of the Max-length Prefix of *r*, then the corresponding bit in "match vector" of *r* would be set to *one*, and algorithm returns. In another word:

If $"newPrefix" \rightarrow MP(r)$, then $r.mv(len(newPrefix)-1)=1$.

But if the Max-length Prefix of *r* is the prefix of the "*newPrefix*", then the corresponding bit with the length of *len(newPrefix)* in the "match vector" of *r* would be set to 1 and the *key(new prefix)* is stored as *r* and algorithms returns or:

If $MP(r) \rightarrow "newPrefix"$, then $r.mv(len(newPrefix)-1)=1$ and $r= key(new\ prefix)$.

Else if *MP(r)* and new prefix are disjoint, based on the result of comparison, the insertion procedure selects the next node to go through. If $"newPrefix"<MP(r)$, then the procedure goes to the left child of *r*, else it goes through the right child.

This procedure will continue till it will be terminated in a node or it would reach a leaf node but not terminated. Then a right or left child will be created based on the procedure above and the prefix will be inserted in the new node.

For an example of insert procedure with $w=4$, consider the following prefixes with their order of arrivals:

$p_1=0^*, p_2=01^*, p_3=000^*, p_4=001^*, p_5=10^*, p_6=00^*$

Based on the scalar comparison method mentioned above it is clear that: $p_1<p_6<p_3<p_4<p_2<p_5$

First of all, to insert p_1, $(key(p_1)=0000, p_1.mv=\underline{1000})$ will be inserted in the root node (Fig.2.a). Therefore, if the root node key is named k_r, $(k_r.mv,k_r)=(\underline{1000},0000)$. After that, to insert p_2, the procedure checks the root node. Since $MP(0000) \rightarrow p_2$, $key(p_2)$ will be stored in the root node instead of k_r. As $len(p_2)=2$, $k_r.mv(1)$ will be set to *one*. Therefore $(k_r.mv,k_r)=(\underline{1100},0100)$. To insert $p_3=000^*$, since 000^* and $MP(root)=p_2$, are disjoint, 000^* will be stored in a new leaf node. The same will be true for insertion of $p_4=001^*$, $p_5=10^*$ and $p_6=00^*$.

B. *Search Procedure* for SP-BST

The search procedure for the Longest Matching Prefix of the address *d* is started from the root and maybe finished in a leaf or non-leaf node. Consider a match vector *d.mv* for *d*. In each node *n* that is being searched, let's consider key_n as the key which is stored in *n*.

If $MP(key_n) \rightarrow d$, then $MP(key_n)$ will be the *lmp(d)* and the procedure will be terminated.

Otherwise, if some other prefixes of key_n match with *d*, the corresponding bit in *d.mv* will be set to *one.*

Then, if $d> key_n$, the procedure goes through the right child of *n*, otherwise it goes through its left child and starts with that node.

For an example of the search procedure, consider $d=0101$. As it is depicted in Fig.2.f, the procedure first searches the Root node. Since the key of the root node $k=0100$, $MP(k)=01^*$ and $MP(k) \rightarrow d$, we have $lmp(d)=01^*$.

As another example, consider $d=0001$. Checking the Root node key $k=0100$, results that $MP(k)! \rightarrow d$. However, checking *k.mv*, the procedure finds 0^* which is a prefix of *d*. Then, since $0001<0100$ i.e. $d<k$, it goes through the left child *A*. In node *A*, $j=0000$ is stored as the key and its main prefix is 000^* i.e. $MP(j)=000^*$. Since $MP(j) \rightarrow d$, then $lmp(d)=000^*$ and the algorithm doesn't need to go through the remaining tree levels e.g. node *C*.

C. *Delete Procedure* for SP-BST

Since the delete procedure sometimes modifies the tree structure, it should be done in a way that the properties of the tree will remain intact after a deletion.

To delete a prefix "*delPrefix*", the procedure would be started from the root and would traverse the path as if it is looking for *delPrefix*.

In this path, assume that the algorithm reaches a node *n* with a key key_n and with *p* as its maximum length prefix, $p=MP(key_n)$. If *delPrefix* is a prefix of *p* but it is not the *p* itself, it only resets (set to *zero*) the corresponding bit in the match vector and returns, or:

if $delPrefix \rightarrow p$ and $delPrefix \neq p$, then $p.mv(len(delPrefix)-1)=0$.

If the algorithm reaches *n* where *p* is the only prefix of key_n, then the deletion will be the same as the deletion in Binary Search Tree.

If $delPrefix=p$ and key_n has some other prefixes in addition to *p*, the corresponding bit in match vector, $p.mv(len(delPrefix)-1)$ will be set to zero and node *n*'s successor (or predecessor) node will be checked. If all of the prefixes of $others(key_n)$ can be stored in *n*'s successor (or predecessor), they will be added to *n*'s successor (or predecessor) and this node will be pulled up. In pulling up the successor (or predecessor), for every node *m* with a key key_m in the path, if there is a prefix of key_m which can be stored in the successor (or predecessor), it will be moved from $key_m.mv$ to the successor (or predecessor).

D. *Properties*

As it was explained in search and insert procedure and also from the example of Fig.2, SP-BST has some properties which are listed below:

a. The Max-length prefixes of all of the node keys in the tree are disjoint. For example, in Fig.2.f, the Max-length prefixes of the nodes are: *01**, *000**, *10** and *001**. These prefixes are disjoint. This property is a result of *Lemma 1* which will be explained in the Appendix.

b. In Scalar Prefix Search, any time the search for address d reaches a key k that its Max-length prefix is a prefix of d or if $p=MP(k)$ and $p \rightarrow d$, then p will be the $lmp(d)$ and therefore the search will be terminated (This is the result of a lemma which is not discussed here).

c. A prefix is stored in the match vector of only one key in the tree. This is a result of *Lemma 2* which is stated in the Appendix. For Example, in Fig.2.f, the prefix *0** is prefix of three keys: 0100, 0000 and 0010, however *0** is stored only in the match vector of 0100 in the root node.

d. If p *is a prefix of all* $k_1, k_2, k_3, \ldots, k_n$ and j is the index of the key of the node with the least height among $k_1, k_2, k_3, \ldots, k_n$, then the prefix p would be stored only in the match vector of k_j and then, $k_j.mv(len(p)-1)=1$.

e. Assume that k is a node key at a node X of a Scalar Prefix Search Tree with $p=MP(k)$. Also assume that Y is a node in a sub-tree of X, therefore its height is greater than the height of X. If l is a key stored in node Y, where $p_l=MP(l)$ and also p_c with length of i is a common prefix for p_l and p (it means that $p_c=pref_i(p_l)=pref_i(p)$), then the existence of p_c in the tree should be indicated by $k.mv(i-1)$ in contrast to $l.mv(i-1)$. For this property, k which is located in a smaller height node (X), is called the "Master key" and l which is located in a larger height node (Y), is called the "Slave key". Also, X is called the "Master node" and Y is called the "Slave node". Therefore each node will be the Master node for all the nodes in its sub-trees. This property is the result of a lemma which is omitted because of the space limitation.

According to the above properties, up to w prefixes can be stored in a key, therefore if n_p is the number of prefixes and n_k is the number of the node keys in the tree, then always $n_k \leq n_p$. The equality holds only when all of the prefixes are disjoint.

The pseudo code for insert procedure in SP-BST is as depicted in Fig.3. Based on the insert procedure, the pseudo code for search procedure which returns the length of the $lmp(d)$ in SP-BST is shown in Fig.4. The delete procedure is depicted in Fig.5.

The SP-BST has many advantages compared to Trie based and range based algorithms. A node of SP-BST may contain up to w prefixes. Therefore, the average height of the tree is reduced. On the other hand, since all of these prefixes are stored in one key and also this tree doesn't need to store both of the prefix end points, the average storage requirement would be reduced as well. Since there is no guarantee for the height of the SP-BST, the concept of Scalar Prefix Search has been applied to some balanced trees such as B-tree, RB-tree and AVL-tree. The complexity of the search and update procedures for these trees are $O(logn)$. These tree structures are explained in the next section and the comparison results with the current solutions are presented in section V.

IV. Other Scalar Prefix Balanced Trees

One solution to decrease the worst case memory access time in Scalar Prefix Search, is to implement it on the balanced trees. This concept is applied on three types of trees and results are explained in the following.

A. SP-BT

This version of scalar prefix Search uses B-tree to store prefixes and is called the Scalar Prefix Search B-Tree (SP-BT). Its worst case memory access for lookup and update procedures is $O(logn)$.

Almost all of the properties of SP-BST are true for SP-BT. The main difference results from the point that SP-BT can store more than one key in each node. Therefore the lookup and update procedures will be different.

Let's remind that each node in the B-tree of degree t, except the root node, may hold from $t-1$ to $2t-1$ keys. Entries are in the form of $2t-1$ pairs of $(key.\underline{mv},key)$. During the insertion, as long as the number of the keys in a node is smaller than $2t-1$, there is no necessity for node splitting. Although, if a key is going to be inserted in a node which already has $2t-1$ keys, then the node should split into two nodes. Similarly, when deleting a key in a node with the number of the keys smaller than t keys, it is necessary to do merge or borrowing operation in B-tree [11].

Assuming the properties of SP-BST are still held, to search for an address d, it would traverse through nodes of the B-tree. At each node, it examines all the keys in the node and searches their match vectors for the matching prefixes. It keeps a match vector for the address d and updates it through search. At any point if either it finds a key that its max length prefix is a prefix of d or reaches a leaf of the tree, it would stop the search and returns the result.

The insert procedure of a prefix p is also derived from the insert procedure of B-tree with the following differences. The procedure starts traversing the tree in the search path of the inserting prefix. When the procedure reaches a node, all the keys of a node will be checked to find the location of the prefix among them.

Without node splitting, it behaves similar to SP-BST, modifying the existing match vectors, but instead of adding node to the tree, it adds key to the existing node.

To split a node, some additional updates should be done. Let's look at Fig.6. Fig.6.a shows the tree nodes before split. Fig.6.b shows the tree nodes after split. Fig.6.c and Fig.6.d show ordering of the prefixes in Fig.6.a and Fig.6.b if they were implemented on SP-BST. Since in Fig.6.a (Fig.6.c), *X, A* and *B* are Master keys compared to *C,* and *C* is the Slave key, the relationships of Master/Slave keys should be transferred to Fig.6.b (Fig.6.d) after splitting and match vectors should be updated accordingly.

Similarly, the delete procedure of a prefix *p*, is derived from the delete procedure of B-tree again with some differences.

Again, the procedure is started from the root node by searching the deleting prefix. The procedure is continued until it reaches the node in which the prefix is stored in key *k*. Assume that *p≠MP(k)*. In this case, *k.mv(len(p)-1)* will be set to *zero* and the algorithm returns.

However, if *p=MP(k),* we might need to perform Merge or Borrow operations at B-tree, when the number of keys in node fall below *t*. Here, some additional updates are needed. Two examples for these cases are given below.

In an example of the borrow operation, depicted in Fig.7, after deleting the prefix p, it is necessary to move *B* and *D* keys. *D* should be updated by *B* and *C* because their levels are interchanged. Also, *X, Y* and *Z* should be updated by *B*.

In the merge operation which is depicted in Fig.8.d, since *height(B)* is become larger than *height(C)*, it should update *C*. Also, *B* should update *Y* and *Z*. Then it should reset its corresponding bits in *B.mv* to zero.

While there are some more details and necessary corner cases for these operations to keep the properties of Scalar Prefix Search intact, they are omitted to make the paper short and readable.

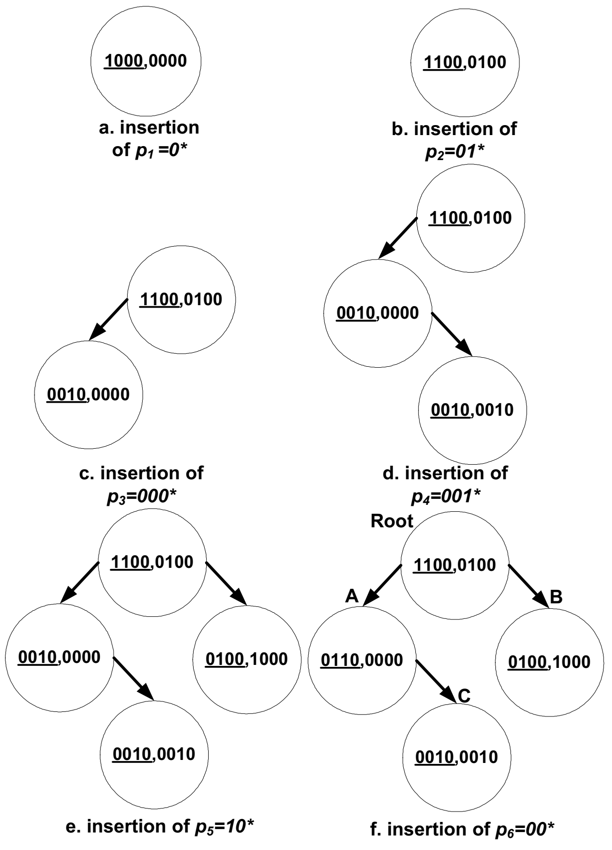
1000,0000
a. insertion of p_1 =0*
1100,0100
b. insertion of p_2=01*
1100,0100
0010,0000
c. insertion of p_3=000*
1100,0100
0010,0000
0010,0010
d. insertion of p_4=001*
1100,0100
0010,0000
0100,1000
0010,0010
e. insertion of p_5=10*
Root
1100,0100
A
B
0110,0000
0100,1000
C
0010,0010
f. insertion of p_6=00*

Figure 2. The steps of insert procedure of SP-BST

```
SPBST_Insert(newPrefix){
if(root is null){
  insert newPrefix in new root node
  return
 }
n=root
while(n){
  $key_n$ = the key of node n
  p=MP($key_n$ )
if((newPrefix is a prefix of p) | (p is a prefix of newPrefix)){
  store newPrefix in ($key_n$.mv,$key_n$)
  return
 }
else if(newPrefix<p){
  if(n.left is null){
    insert newPrefix in new left child of n
    return
  }
 else
   n=left child of n
 }
else if(newPrefix>p) {
  if(right child of n is null){
    insert newPrefix in new right child of n
    return
  }
  else
    n=right child of n
  }
 }
}
```

Figure 3. Insert pseudo code for SP-BST

B. SP-BTe

To enhance the performance of the SP-BT, the following observation was made. As stated in the previous sub-section, during the search procedure of SP-BT, at each node, all the keys and their match vectors of a node should be searched and examined to find all possible matching prefixes until reaching the key that its MP is greater than *d*. During this search, *lmp(d)* will be updated by the match vector of each key. This was due to the fact that each prefix was stored exactly in one key.

To enhance the performance and reduce the search time within a node, during the insertion of a prefix the following action is performed. *Only within each node*, the existence of each prefix is flagged into all of the match vectors of the keys that this prefix is a prefix of them. Therefore, during the search time in this node, it would be sufficient to just check the keys to find the location of this address *d* among the keys and then finally check the match vectors of only the two keys before and after *d*, to find *lmp(d)* in this node.

```
SP-BST_SearchForLMP(d){
matchVec =0
n =root
while(n){
  $Key_n$= the key stored in n
  p=MP($Key_n$)
  len = length(longest common prefix of $Key_n$ and d)
  if($Key_n$.mv[len..W-1]==0)
    return len
  matchVec = (matchVec | n.mv[0..len-1])
  if(d<$Key_n$)
    n=left child of n
  else
    n=right child of n
}
return index of least significant one of matchVec +1
}
```

Figure 4. Search pseudo code for SP-BST

This will reduce the search time by a large factor, so it is called the SP-BTe (SP-BT enhanced). The drawback is for delete and update time. When there is a delete, merge, or split, extra cautious should be used to make sure that all of the match vectors within a node have been updated and prevent the existence of a prefix to be claimed by more than one node.

Again, details have been removed for the sake of the readability of the paper. Examples of the SP-BT and SP-BTe nodes are shown in Fig.9 and Fig.10.

Consider the following prefixes: p_0=*0**, p_1=*000**, p_2=*001**, p_3=*11**. Assuming *w*=3, the corresponding keys of these prefixes, (k_0=*000* related to p_0 and p_1, k_1=*001* related to p_2 and k_2=*110* related to p_3) can be stored in a SP-BT node as it is depicted in Fig.9.

If the destination address is *d=010,* it will be between two keys k_1= *001* and k_2=*110*. If we start the search from left to right, p_0 = *0** will be found in k_0*.mv* which is *101*. However, k_1 and k_2 don't contain any prefix of *d*. Therefore, *lmp(d)* will be *0**. As this example shows, *0** is also a prefix of k_1, however, it is not stored in its match vector (*001*). This is the reason of the need to search the match vectors of the other keys.

If each prefix of p_2, is stored in the match vector of k_1, it is not necessary to search the match vectors of the keys before k_1. This is the main idea of SP-BTe. Using this rule, the result of storing these prefixes in SP-BTe is shown in Fig.10.

C. *SP-RB* and *SP-AVL*

SP-RB and SP-AVL are Scalar Prefix search trees which store prefixes in RB-tree and AVL-tree. The same idea has been applied to these trees. The main difference of these two types of trees with SP-BST is that they try to keep the tree balanced. Therefore, the worst case height of the tree and the worst case search time of the algorithms are different.

Basically, the search and update procedure for these trees are similar to the previous ones. The only difference is the possible rotation which needs to be done to keep the tree balanced. Therefore, the required procedure should be used to keep the SP-BST properties intact. Again, details of these procedures have been removed because of the space limitation. Two types of rotations with their required updates are shown in Fig.11.

```
SP-BST_Remove(delPrefix){
n=root
while(n){
  $Key_n$= the key of node n
  p= MP($Key_n$)
  if(delPrefix exists in n){
    if(delPrefix is not equal to p) {
      $Key_n$.mv(len(delPrefix)-1)=0
      return
    }
    else if(delPrefix is the the only prefix of n){
      delete n same as BST
      return
    }
    else {//n has other prefixes
      reomve delPrefix from n
      npred=n's predecessor key
      nsucc=n's successor key
      if(others($Key_n$) can be stored in npred){
        store them in npred
        pullUp(n,npred)
        return
      }
      else if(others($Key_n$) can be stored in nsucc){
        store them in nsucc
        pullUp(n,nsuc)
        return
      }
      else{
        return
      }
    }
  }
  else if(delPrefix<p)
    n=left child of n
  else
    n=right child of n
}
}
pullUp(n, nsucc (or npred)){
  for each node m with the key, $key_m$ in nsucc (or
  npred) path
    if there is a prefix p in $key_m$.mv that p is a prefix
    of MP(nsucc (or npred))
      move p from $key_m$ to nsucc (or npred)
}
```

Figure 5. Delete pseudo code for SP-BST

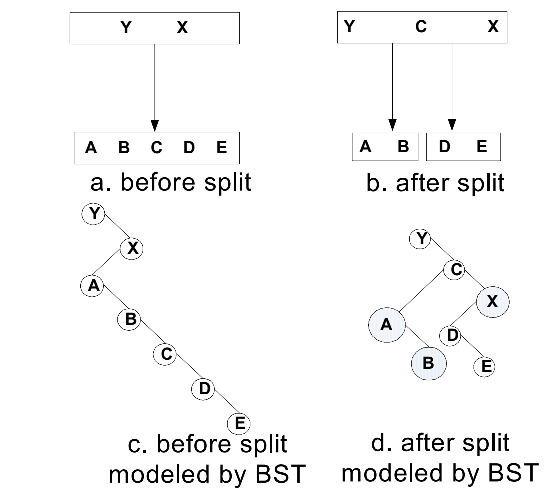

Figure 6. Split in SP-BT

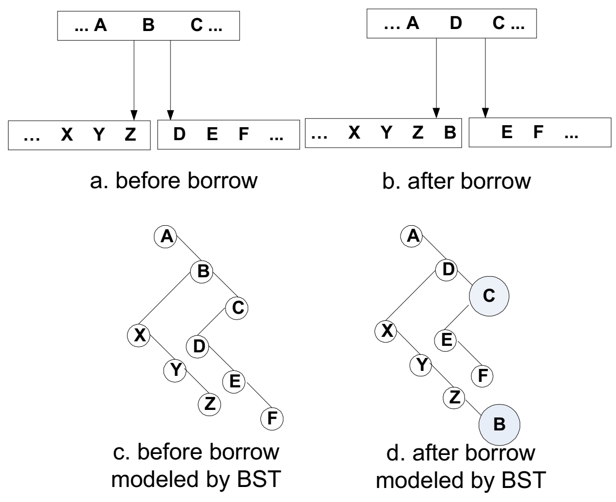

Figure 7. Borrow operation in SP-BT

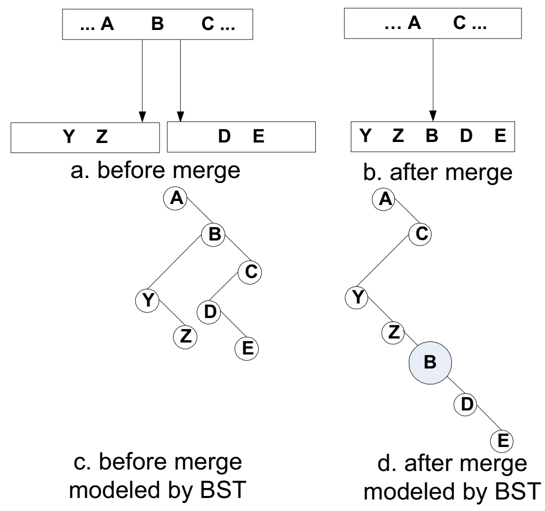

Figure 8. Merge operation in SP-BT

As it is shown in Fig.11.a, in this rotation, X should update Y and then reset its corresponding match vector bits to zero. In Fig.11.b, Y and X will update Z and their corresponding match vector bits will be reset to zero. Other search and update parts are similar to SP-BST.

The result of applying the Scalar Prefix Search concept to these different trees and their comparison are presented in the next section.

V. Complexity Analysis and Comparison Results

Scalar Prefix Search which was introduced in this paper was implemented with three types of balanced trees: B-tree, RB-tree and AVL-tree. For comparison propose PIBT and LPFST were also implemented. The B-Tree degree t, was selected to be 14 and 8. The codes are written in C++ and executed on an Intel Pentium4 PC with CPU of 3GHz and 1GB RAM. Two random databases of 100000 and 500000 prefixes called respectively R100 and R500 with the current prefix lengths distribution of IPV4 prefixes [12], and two BGP tables, as4637 [12] (139519 prefixes) and as1221 [12] (191566 prefixes) are used for this comparison. The results are shown in Fig.12, Fig.13 and Fig.14.

As it is shown in Fig.12, the results are ordered on the growing size of the database. This figure shows the average search time and it is obvious that the search performance of SP-BTe is much better for all of the 4 databases. LPFST shows better results for the two smaller databases (the R100 random prefixes database and AS4637). However, after the table sizes grow (AS1221 and the R500 database), SP-BT shows better results. The reason is the good compression performance and also the worst case tree height of SP-BT.

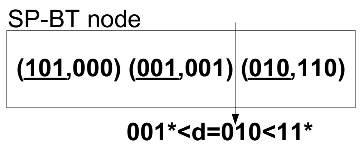

Figure 9. An example of a SP-BT node

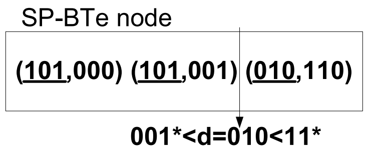

Figure 10. An example of SP-BTe node

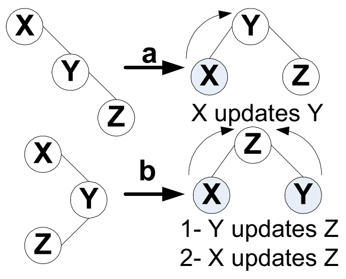

Figure 11. Example of rotations and required updates

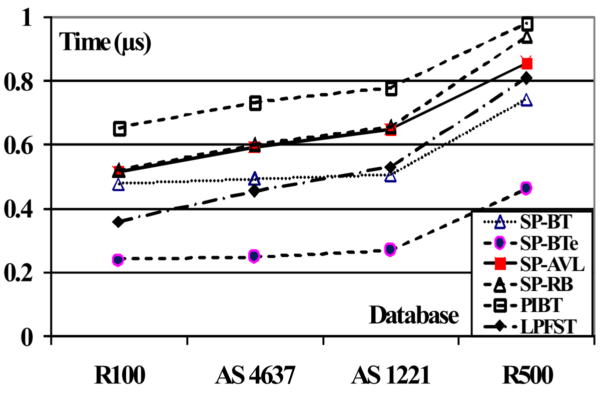

Figure 12. Search results

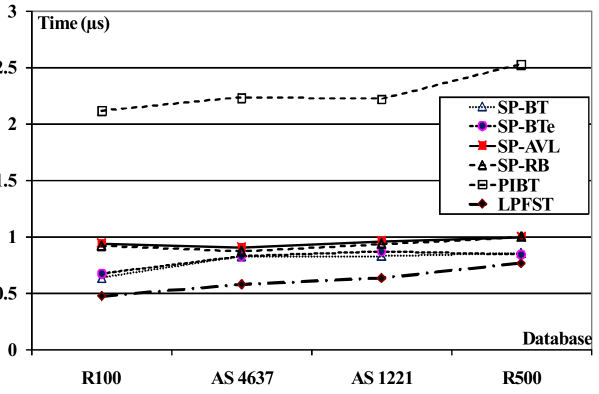

Figure 13. Update results

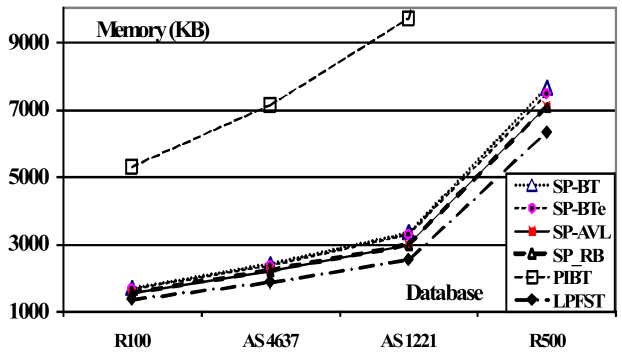

Figure 14. Storage results

Another advantage is the update times (Fig.13). It is clear that all of Scalar Prefix Search trees perform better than PIBT. Also, although LPFST performance is better than SP-BTe in this Figure, but looking at the LPFST increasing curve, it can be concluded that the result of the update time of SP-BTe will be comparable or even better if database sizes grow larger. In other words, the gradient of the curve of SP-BT and SP-BTe is less than that of LPFST. This is the result of slow growing rate of the height of the B-trees for SP-BT and SP-BTe when the table size grows. The height of these trees grows logarithmically with the increasing size of the prefixes database. Another reason is that each key of the Scalar Prefix Search may represent up to *w* prefixes and this represents its compression capability. Also when the number of prefixes increases, the probability of having new disjoint prefixes will decrease, therefore in many occasions, if a prefix is added to the table, the update procedure doesn't need to traverse the tree height completely and this makes the update time better.

Many linear operations like shifting are done in the update procedure. These operations have the complexity $2t$ in software where t is the order of the B-tree. However, the complexity of these operations will be *O(1)* in hardware. Therefore Scalar Prefix Search algorithm would have better search and update performance in hardware implementation. Additionally SP-BT, SP-BTe, SP-RB and SP-AVL are all using balanced trees, whose node access complexities are *O(log n)* where n is the number of prefixes. Therefore, they would have better worst case performance compared to LPFST which its worst case performance depends on *w*.

Fig.14 shows that the memory requirements of Scalar Prefix Search cases are about one third of PIBT. The reason is that PIBT needs to store 2 end points, about 4 vectors and 2 pointers for each prefix, in contrast, each key of a Scalar Prefix Search, may contain more than one prefix and it needs only one match vector. In comparison, the memory requirements of Scalar Prefix search cases are a little more than that of LPFST, using these databases in which a low percent of the prefixes are non-disjoint (about 8% for AS4637, AS 1221, R100 and 17% for R500). However, in the databases with less disjoint prefixes, Scalar Prefix Search Trees become more compressed. As it was explained in section III, up to *w* prefixes might be stored in one key. Therefore, according to the increasing size of routing tables and increasing percent of non-disjoint prefixes, we believe that Scalar Prefix Trees can reduce the size of future routing tables by a big ratio, compared to other existing methods. To prove our idea by an example, let's check the results of applying a random database of 100000 prefixes with 50% disjoint prefixes. Table 1 shows the search time, update time and storage requirements. As it is shown, the memory requirements of scalar prefix trees have become comparably improved compared to PIBT and LPFST. Also the search and update times results have become better than the previous experiment with mentioned four databases.

Table 1- The results of applying a random database with 50% disjoint prefixes

	LPFST	PIBT	Scalar Prefix Search Trees			
			RB	AVL	BTe	BT
Search Time (us)	0.42	0.55	0.45	0.45	0.23	0.46
Update Time (us)	0.44	1.78	0.50	0.50	0.60	0.64
Memory (KB)	1128	3180	852	852	908	908

VI. CONCLUSION

In this paper, a new prefix search algorithm was introduced called Scalar Prefix Search. In contrast to other current solutions, this algorithm treats each prefix as a number. Using keys and match vectors, it can store several prefixes in one key. Properties of this method, enables the search and insertion algorithm to terminate without the need to traverse the whole height of the tree when they found the required result. This, results to the simple prefix search and update procedures. Its properties allow its implementation using different balanced trees to control the worst case search and update time. They also make it possible to use both binary and non-binary trees to further reduce the height of the tree. Other advantages of this algorithm are its update time, compression performance and storage requirements especially for large routing tables and possibility of an efficient hardware implementation. The comparison results show much better search times compared to current solutions like PIBT and LPFST.

REFERENCES

[1] H. Lu and S. Sahni, "A B-Tree Dynamic Router-Table Design", *IEEE Trans. Computers, Vol.54, pp.813-823, 2005*

[2] L. C. Wnn, K. M. Chen and T. J. Liu, "A Longest Prefix First Search Tree for IP Lookup", *ICC'05, pp.989 – 993, May 16-20, 2005.*

[3] "Internet Protocol", September, 1981, *RFC 791*

[4] Fuller, V., Li, T., Yu, J., and Varadhan, K., 1993, "Classless Inter-Domain Routing (CIDR): An Address Assignment and Aggregation Strategy," *RFC 1519*.

[5] S. Nilson and G.Karlsson, "IP address lookup using LC-tries", *IEEE J. on Sel. Area in Comm, Vol.17, pp.1083-1092, June.1999*.

[6] V. Sirinivasan and G. Varghes, "Faster IP lookup using controlled prefix expansion". *Proceedings of the ACM SIGMETRICS joint international conference on measurement and modeling of computer systems, pp.1-10, 1998.*

[7] P. Gupta, S. lin, N. McKeown, "Routing lookups in Hardw are at Memory Access Speeds", *INFOCOM '98, 1998.*

[8] B. Lampson, V. Srinivasan, and G. Varghese, "IP Lookup Using Multi-Way and Multicolumn Search", *INFOCOM '98, 1998.*

[9] Q. Sun, X. Zhao, X. Huang, W. Jiang and Y. Ma, "A Scalable Exact matching in Balance Tree Scheme for IPv6 Lookup", *ACM SIGCOMM 2007 data communication festival, IPv6'07, August.27-31, 2007.*

[10] M. Behdadfar, H. Saidi, "The CPBT: A Method for Searching the Prefixes Using Coded Prefixes in B-Tree", *IFIP Networking 2008, pp. 562-573, May.5-9, 2008.*

[11] T.Cormen, C.Leiserson, R.Rivest, Introduction to Algorithms. *Hill Book Company, McGraw-Hill, New York (1999)*

[12] http://bgp.potaroo.net

APPENDIX

Lemma 1- If X is a node of SP-BST containing a key k that $MP(k)= p$, and r is a key in X's right sub-tree and "l" is a key in X's left sub-tree, then

a. if ($len(MP(r)) \geq i$ && $len(MP(l)) \geq i$ && $pref_i(MP(r))=pref_i(MP(l))$) then $len(p)\geq i$ and $pref_i(p)=pref_i(MP(r))$
b. if $pref_i(p)= pref_i(MP(r))$ then $r.mv(i)=0$
c. if $pref_i(p)= pref_i(MP(l))$ then $l.mv(i)=0$
d. p, $MP(r)$ and $MP(l)$ are disjoint.

Proof idea:
a. Looking at Fig.15, let's assume that $q=pref_i(MP(l))=pref_i(MP(r))$, if $lp=len(p)<i$, since $MP(l)<p$, it results that $pref_{lp}(MP(l))<p$. Therefore $pref_{lp}(MP(r))<p$ and it results that $MP(r)<p$ which is a contradiction. Therefore $len(p)\geq i$.
Now let's assume that $q'= pref_i(p)$ and $q'\neq q$. Two states would be possible; $q'> q$ or $q'< q$. If $q'> q$, it means $k>r$ which is a contradiction. If $q'<q$, it means $k<l$, which is a contradiction. Therefore, $q'= q$.

b. Assume that $r.mv(i-1)$=1. This means that a prefix e.g. z has been added to the tree which is equal to $pref_i(MP(r))$. Since there is only one path from the root to r and this path contains k, the insertion path of z, should have traversed k before reaching r. Since $z= pref_i(p)$, when the insertion procedure has reached k, it should have modified $k.mv(i-1)$ to *one* and terminated the insertion. Therefore $r.mv(i-1)$ would remain *zero* which is a contradiction. Similarly it can be proved that the delete operation doesn't affect this property.

c. The proof is similar to b.

d. It should be proved that each 2 prefixes are disjoint. The remaining cases are:
$MP(l) \rightarrow p$, $MP(r) \rightarrow p$, $p \rightarrow MP(l)$, $MP(r) \rightarrow MP(l)$, $p \rightarrow MP(r)$, $MP(l) \rightarrow MP(r)$

Assume that $MP(l) \rightarrow p$. According to the insertion algorithm, as the insertion procedure of l reaches p, l will be stored in $p.mv$, not in the match vector of a disjoint key which is a contradiction. The proof is similar for the remaining cases. Similarly, it can be proved that the delete operation doesn't affect this property.

Lemma 2- The existence of a prefix p in a SP-BT is illustrated in only one node of the tree.
Proof- omitted.

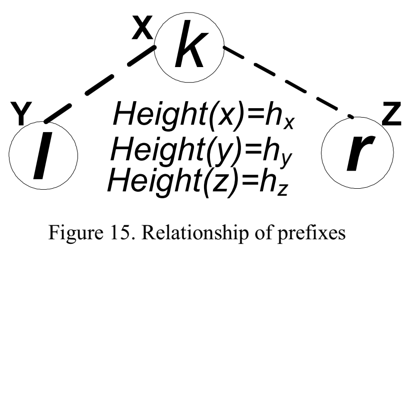

Figure 15. Relationship of prefixes