IEICE TRANS. INF. & SYST., VOL.E93–D, NO.11 NOVEMBER 2010

PAPER Special Section on Architectures, Protocols, and Applications for the Future Internet

IP Lookup Using the Novel Idea of Scalar Prefix Search with Fast Table Updates**

Mohammad BEHDADFAR[†a)], ***Student Member***, **Hossein SAIDI**[†], **Masoud-Reza HASHEMI**[†], **Ali GHIASIAN**[†], ***and*** **Hamid ALAEI**[†*], ***Nonmembers***

SUMMARY Recently, we have proposed a new prefix lookup algorithm which would use the prefixes as scalar numbers. This algorithm could be applied to different tree structures such as Binary Search Tree and some other balanced trees like RB-tree, AVL-tree and B-tree with minor modifications in the search, insert and/or delete procedures to make them capable of finding the prefixes of an incoming string e.g. an IP address. As a result, the search procedure complexity would be *O(log n)* where *n* is the number of prefixes stored in the tree. More important, the search complexity would not depend on the address length *w* i.e. 32 for IPv4 and 128 for IPv6. Here, it is assumed that interface to memory is wide enough to access the prefix and some simple operations like comparison can be done in *O(1)* even for the word length *w*. Moreover, insertion and deletion procedures of this algorithm are much simpler and faster than its competitors. In what follows, we report the software implementation results of this algorithm and compare it with other solutions for both IPv4 and IPv6. It also reports on a simple hardware implementation of the algorithm for IPv4. Comparison results show better lookup and update performances or superior storage requirements for Scalar Prefix Search in both average and worst cases.

key words: *scalar prefix, LPM, LMP, SP-BT, search, update*

1. Introduction

One of the main functions of an IP router is forwarding decision which finds the longest prefix of the destination address of the incoming IP packet among a set of prefixes stored in the router tables. To do it, an algorithm is used called Longest Prefix Matching or LPM. One of the old structures for LPM is Trie or Radix tree [1]. It is a simple tree with each edge representative of one bit '0' or one bit '1'. The left child of a node adds a '0' to the string and the right child, adds a '1'. For *w* bits IP addresses, the main drawback of this structure is its worst case *O(w)* node access complexity for search and update procedures. Some improvements of Trie [2]–[4], were also introduced to solve this problem. However, most of them suffer from the above disadvantage or time consuming update procedures.

To become independent from *w*, 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 the binary interval [1000,1011] with 1000 and 1011 as its end points.

Some of these range based algorithms store the prefix end points in a binary search tree and propose a search algorithm based on the resulting structure [5]. However, for updating the tree in the worst case, the whole structure should be constructed again. To improve these search and update procedures, *MRT* was proposed using a B-tree structure [6]. It reduces the lookup complexity to *O(log n)*. However, it needs still large number of memory accesses for the update procedure. For example, a tree with the branch factor of 14 and height of 5 needs 455 memory accesses in the worst case of a prefix update procedure [6]. To solve this problem, other range-based algorithms [7]–[10] were also introduced among which, PIBT [10] has the best search performance based on [10]. PIBT stores prefix end points in a B-tree. Therefore, its search and update complexity is *O(log n)*. Based on [10], PIBT has better update time and similar search time comparing to MRT. It should be mentioned that PIBT, uses about two additional *w* bits vectors for each endpoint. Therefore, since each prefix has two end points, up to about 6 vectors might be stored for each prefix. One of the most recent range-based lookup algorithms is BTLPT [11]. This algorithm uses two structures: a B-tree for disjoint prefixes and a Trie based structure called LPFST [12] for the remaining prefixes. Simulation results of [11] show that the performance of BTLPT is better than PIBT in average search and update times and also for the storage requirement. However, the node access complexity of BTLPT depends on *w* unlike PIBT. Prefix Trees and M-way Prefix Trees are other range-based algorithms which were proposed in [13] and their main idea is to introduce a comparison rule for storing and searching the prefixes. The main drawbacks of these algorithms are the worst case tree height which is *O(w)* and the long update procedures.

In [14], we introduced a new algorithm called "Coded Prefix B-tree" or CP-BT. This scheme considers a coding concept for prefixes to compare them like numbers with "=", "<" and ">". We call this concept "Coded Prefix Search". Importantly this makes the search and update times of the algorithm, independent of IP address length. Also, unlike the range-based algorithms, CPBT does not consider two end points or a range of IP addresses for prefixes. A prefix is stored as a number in a B-tree with only one additional vector. Therefore, the complexity of search and update will be *O(log n)*. Also, based on the simulations of current paper, its required memory will be less than that of PIBT and addition-

Manuscript received March 1, 2010.
Manuscript revised June 6, 2010.

[†]The authors are with the Department of Electrical and Computer Engineering, Isfahan University of Technology, Esfahan, Iran.
[*]Presently, with the Faculty of Computer Engineering, Amir Kabir University of Technology, Tehran, Iran.

[**]This paper was partially presented at IEEE INFOCOM 2009.

a) E-mail: behdadfar@ec.iut.ac.ir

DOI: 10.1587/transinf.E93.D.2932

Copyright © 2010 The Institute of Electronics, Information and Communication Engineers

ally, it has comparable search and better update results. The method which is proposed in this paper called Scalar Prefix Search, is an improved version of [14] and was partially presented in [15]. The property of comparing prefixes like numbers which is proposed here, makes the algorithm superior for 32 bits IPv4 addresses and especially for 128 bits IPv6 addresses. Here, the prefix encoding and comparing concept of CPBT, is extended and made easier by proposing new store and search methods. The way to implement this concept on many different types of trees is also introduced. In this paper, the Scalar Prefix Search and Coded Prefix Search algorithms are applied to BST (Binary Search Tree), B-tree, RB-tree (Red Black Tree) and AVL-tree. Finally, it is shown that their performance is comparable to recent solutions like PIBT, BTLPT and LPFST. It is worth mentioning that the correctness of Scalar Prefix search is proved completely by authors. However, parts of the proofs are removed to make the paper short and readable.

Section 2 introduces the Scalar Prefix Search procedure and its properties. Section 3 introduces balance tree versions of the algorithm. The software and hardware results are shown in Sect. 4. Section 5 concludes the paper.

2. Scalar Prefix Search Procedure

Scalar Prefix Search can be applied to many types of trees. Although, application of this algorithm to Binary Search Tree is not efficient in performance, but we will apply it to just simply describe the procedures. Then, we will extend this algorithm to other trees such as: B-tree, RB-tree and AVL-tree. Application of this method to this tree is named "Scalar Prefix Binary Search Tree" or SP-BST. To describe the algorithm, special Insertion, Search and Deletion procedures are defined. These procedures are different from similar procedures of Binary Search Tree. Following notations and definitions are used throughout this paper:

- *len(p)* shows the length of a prefix p.
- *p(i)* shows i^{th} bit of prefix p.
- *key(p)* or key_p: For each prefix p with *len(p)=k*, and *k<w*, we add *w-k* zero padding and we call it *key* and show it as *key(p)* or key_p which will be inserted into the tree instead of the original prefix. *key(p)* will be defined as:
  - *key(p)*="*p(0)p(1)p(2)...p(k-1)000...0*"
    For example, if *w=4* and *p=101**, then: *key(p)= 1010*.
- *Prefix symbol*: The notation $p \rightarrow q$ shows that p is a prefix of q.
- $p! \rightarrow q$ This notation 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 $pref_k(p)$.
- *LMP*: The Longest Matching Prefix of a string S is denoted as *LMP(S)*.
- *Match Vector*: For a key r, a w bit *"Match Vector"* is defined and abbreviated to *"r.mv"*.
  - The i^{th} bit of *r.mv* is called *r.mv(i)*. If *r.mv(i)=1*, it means that there exists a prefix q with the length of *i+1* which is also a prefix of r (i.e. *len(q)=i+1* or in other words $q=pref_{i+1}(r)$). Please note that the indexing of Match Vector bits starts from "0".
- *height*: The length of the path from the root of the tree to a node x is called *height(x)*, e.g. *height(root)* is zero, and the height of each child of the root is "one" and so on.
- *MP(key)*: The longest prefix of each key is called the "Max-length Prefix" of that key and is shown by *MP(key)*. The largest i such that *key.mv(i)=1*, shows that the length of *MP(key)* is *i+1*.
- *Predecessor 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.
- *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)*.

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

Using the above definitions, prefixes can be stored in any search tree including BST (Binary Search Tree).

2.1 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 the algorithm returns. In other words:
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 *one* and the *key(newPrefix)* is stored as r and algorithms returns. In other words:
If *MP(r)*$\rightarrow$*"newPrefix"*, then:
r.mv(len(newPrefix)-1)=1 and *r= key(newPrefix)*.
Else if *MP(r)* and *"newPrefix"* 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: $p_1=001^*$, $p_2=1^*$, $p_3=0^*$, $p_4=000^*$, $p_5=00^*$, $p_6=11^*$, $p_7=01^*$. Based on the scalar comparison method mentioned above it is clear that:

$p_3<p_5<p_4<p_1<p_7<p_2<p_6$

First of all, to insert p_1, $key(p_1)=0010$, $p_1.mv=\underline{0010}$ will be inserted in the root node as it is depicted in Fig. 1 (a). Therefore, if the root node key is named k_r, $(k_r.mv,k_r)=(\underline{0010},0010)$. After that, to insert p_2, the procedure checks the root node. Since $MP(0010)$ is disjoint with p_2 and also $p_2>MP(0010)$, a right hand child will be created to insert p_2 (Fig. 1 (b)). Since $p_3\rightarrow MP(0010)$, p_3 will be inserted in the root node without changing the root node key=0010. However, its match vector will become $\underline{1010}$ (Fig. 1 (c)). Insertion of p_4 will be similar to p_1 and a left child node will be created (Fig. 1 (d)). Insertion of p_5 only affects the match vector of the root node key and modifies it to 1110 (Fig. 1 (e)). Insertion of p_6 updates the key and match vector of node A in Fig. 1 (e) from $(\underline{1000},1000)$ to $(\underline{1100},1100)$ in Fig. 1 (f). Similar procedure is also done for insertion of p_7 in Fig. 1 (g).

2.2 Search Procedure for SP-BST

The search procedure for the Longest Matching Prefix of the address d is started from the root and may be finished in a leaf or non-leaf node.

Consider a match vector $d.mv$ for d. In each node n that is being searched, if its Max-length prefix is a prefix of d, then it is the Longest Matching Prefix we look for and the procedure will be terminated. In another word, 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. It then repeats the procedure at the child node.

As an example, consider $d=0011$. Starting from the root node R, Checking the Root node key and its match vector of Fig. 1 (g) $(k.mv=\underline{1110}, k=0010)$, results in $MP(k)\rightarrow d$. Therefore, $LMP(d)=001^*$ and the search will be terminated in the root node. It means that it does not need to go to the remaining tree nodes. As another example, consider $d=0101$. To start the procedure, consider $d.mv=0000$. Checking the Root node key and its match vector of Fig. 1 (g) $(k.mv=\underline{1110}, k=0010)$, results in " $MP(k)!\rightarrow d$.

However, checking $k.mv$, the procedure finds 0^* which is a prefix of d. Therefore, $d.mv=\underline{1000}$. Then, since $0101>0010$ i.e. $d>k$, it goes through the right child A. In node A, $j=1100$ is stored as the key and its main prefix is 11^* i.e. $MP(j)=11^*$. Since $MP(j)!\rightarrow d$ and also no prefix of j matches with d and $d<j$, the procedure goes to node C, the left child of node A which contains $(l.mv,l)=(\underline{0100}, 0100)$.

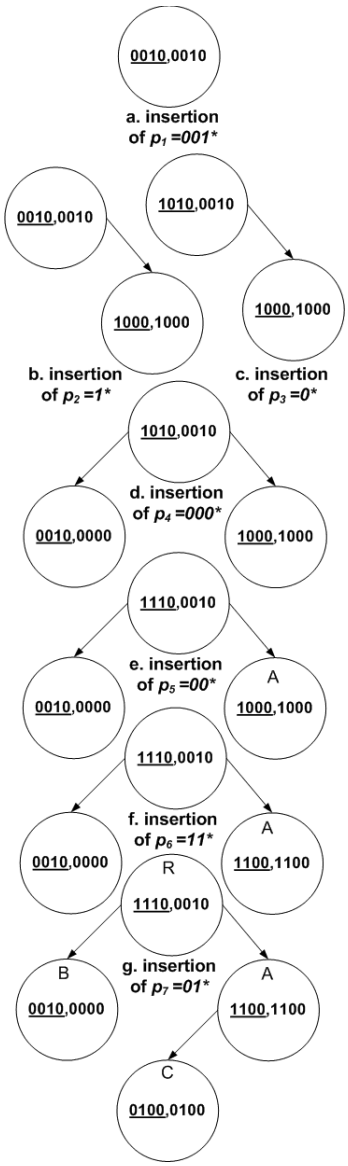

Fig. 1 Insertion example of SP-BST.

In this node, 01^* exists which is a prefix of d. Therefore, $d.mv=\underline{1100}$ and $LMP(d)$ will be the longest prefix among 0^* and 01^*, i.e. $LMP(d)=01^*$. Please also note that $MP(l)\rightarrow d$ and this is the sufficient condition to conclude that 01^* is $LMP(d)$.

2.3 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 start from the root and 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 Max-length prefix, $p=MP(key_n)$. If *delPrefix* is a prefix of p but it is not the

p itself, it only resets (sets to *zero*) the corresponding bit in the match vector and returns. In other words:
if $delPrefix \rightarrow p$ and $delPrefix \neq p$, then:

$$p.mv(len(delPrefix)-1)=0 \text{ and return.}$$

If the algorithm reaches node 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).

2.4 Properties

As it was explained in search and insert procedure and also from the example of Fig. 1, SP-BST has some properties which are listed below:

1. The Max-length prefixes of all of the node keys in the tree are disjoint. For example, in Fig. 1 (g), the Max-length prefixes of the nodes are: *001**, *000**, *11** and *01**. These prefixes are disjoint. This property is a result of *Lemma 1* which will be explained in the Appendix.
2. In Scalar Prefix Search, any time the search for address d reaches a key k that its Max-length prefix p is a prefix of d then p will be the $LMP(d)$ (i.e. if $p=MP(k)$ and $p \rightarrow d$ then $p = LMP(d)$), and therefore the search will be terminated. This is the *Lemma 2* which is explained in the appendix.
3. A prefix is stored in the match vector of only one key in the tree. This is the direct result of the insertion and deletion algorithms which was mentioned above. For Example, in Fig. 1.g, the prefix 0^* is prefix of three keys: *0010*: the key of node *R*, *0000*: the key of node *B* and *0100*: the key of node *C*. However, 0^* is stored only in the match vector of *0010* which is the key of the root node *R*.
4. To store a new prefix p in SP-BST, let's assume that $K=\{k_1, k_2, k_3, \ldots, k_n\}$ is the set of all keys stored in the tree which p is a prefix of them. If among members of K, $k_j \in K$ is the key that its node has the least height, then the prefix p will be stored only in the match vector of k_j and $k_j.mv(len(p)-1)$ will be set to *one*. This is the *Lemma 3* which is explained in the appendix.
5. Again assume that $K=\{k_1, k_2, k_3, \ldots, k_n\}$ is the set of all keys stored in the tree which p is a prefix of them and among its members, k_j is the key whose node has the least height. Then, for any arbitrary address d such that $p \rightarrow d$, the search path of d will reach the node containing k_j. This is the *Lemma 4* which is explained in the appendix.
6. Consider the same definitions for prefix p, the set K and the key k_j in properties 4 and 5. Also, consider an address d with the property of $p \rightarrow d$. Based on *Property* 4, k_j is the first member of K which is seen in the insertion path of p. Also, based on the *Property* 5, it is the first member of K which is seen in the search path of d. Therefore, if the objective is to store p in the tree, its existence will be indicated in the match vector of k_j. On the other hand, if the objective is to search d in the tree, the search procedure will reach k_j in the search path of d before any other member of K and $k_j.mv(len(p)-1)$ will indicate if p is stored in the tree or not.
   Therefore, k_j and its match vector have all the information about p and make these procedures independent of the other members of K and their match vectors. Therefore, with respect to p, we call k_j the Master key for all of the other members of K located in its subtrees. Also, the other members of K in the subtrees of k_j are called the Slave keys. The reason of this naming is that with respect to p, k_j and its match vector, overrule all of the information stored in its subtrees.

Based on 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. Therefore, if the percentage of the disjoint prefixes decreases, the tree will become more compact, because each bit of a match vector would be the representative of one prefix. Even for a high percentage of disjoint prefixes (about 92%), still we can see the effect of this compression in comparing the performance of Coded Prefix Trees and Scalar Prefix Trees in Figs. 9, 10, 11 and 12.

The pseudo code for insert procedure in SP-BST is as depicted in Fig. 2. 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. 3. The delete procedure is also depicted in Fig. 4. The SP-BST has many advantages compared to Trie based and range based algorithms. A node key 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 does not 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. These trees have the property that can guarantee and control the worst case height of the tree to be $O(\log n)$. Therefore the complexity of the search and update procedures for these trees are $O(\log n)$ as well. These tree structures are explained in the next section.

```
SPBST_Insert(newPrefix){
01  if(root is null){
02    insert newPrefix in new root node
03    return
      }
04  n=root
05  while(n){
06    key_n = the key of node n
07    p=MP(key_n )
08  if((newPrefix→ p) | (p → newPrefix)){
09    store newPrefix in (key_n.mv,key_n )
10    return
      }
11  else if(newPrefix<p){
12    if(n.left is null){
13      insert newPrefix in new left child of n
14      return
        }
15    else
16      n=left child of n
      }
17  else if(newPrefix>p) {
18    if(right child of n is null){
19      insert newPrefix in new right child of
n
20      return
        }
21    else
22      n=right child of n
        }
      }
}
```

Fig. 2 Insertion pseudo code of SP-BST.

```
SP-BST_SearchForLMP(d){
01  matchVec =0
02  n =root
03  while(n){
04    Key_n= the key stored in n
05    p=MP(Key_n)
06    len = length(longest common prefix of Key_n and d)
07    if(Key_n.mv[len..W-1]==0)
08      return len
09    matchVec = (matchVec | Key_n.mv[0..len-1])
10    if(d<Key_n)
11      n=left child of n
12    else
13      n=right child of n
      }
14  return index of least significant one of matchVec +1
}
```

Fig. 3 Search pseudo code of SP-BST.

3. Other Scalar Prefix Balanced Trees

One solution to decrease the worst case node access in Scalar Prefix Search, is to implement it on balanced trees. This concept has been applied on three types of trees and results are explained in the following sub-sections.

3.1 SP-BT

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

Almost all of the properties of SP-BST are true for SP-BT. The main difference is 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 $(key.mv,key)$. 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, to delete a key in a node with the number of the keys smaller than t keys, it is necessary to do merging or borrowing operation in B-tree [1].

Assuming the properties of SP-BST are still held, to search for an address d, it would traverse through nodes of

```
SP-BST_Remove(delPrefix){
01  n=root
02  while(n){
03    Key_n= the key of node n
04    p= MP(Key_n)
05    if(delPrefix exists in n){
06      if(delPrefix is not equal to p) {
07        Key_n.mv(len(delPrefix)-1)=0
08        return
          }
09      else if(delPrefix is the the only prefix of n){
10        delete n same as BST
11        return
          }
12      else {//n has other prefixes
13        reomve delPrefix from n
14        npred=n's predecessor key
15        nsucc=n's successor key
16        if(others(Key_n) can be stored in npred){
17          store them in npred
18          pullUp(n,npred)
19          return
            }
20        else if(others(Key_n) can be stored in nsucc){
21          store them in nsucc
22          pullUp(n,nsuc)
23          return
            }
24        else{
25          return
            }
          }
        }
26    else if(delPrefix<p)
27      n=left child of n
28    else
29      n=right child of n
      }
}
30  pullUp(n, nsucc (or npred)){
31    for each node m with the key, key_m  in nsucc (or npred) path
32      if there is a prefix p in key_m.mv that p →MP(nsucc (or npred))
33        move p from key_m to nsucc (or npred)
}
```

Fig. 4 Deletion pseudo code of SP-BST.

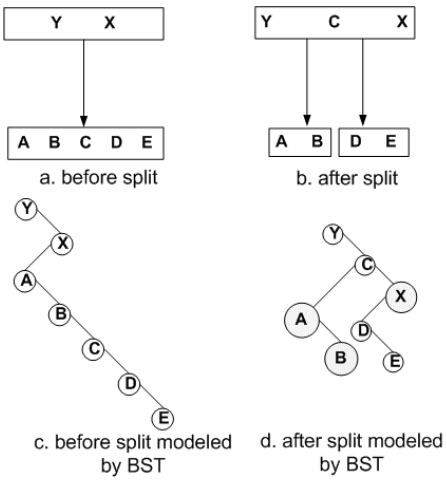

Fig. 5 Split in SP-BT.

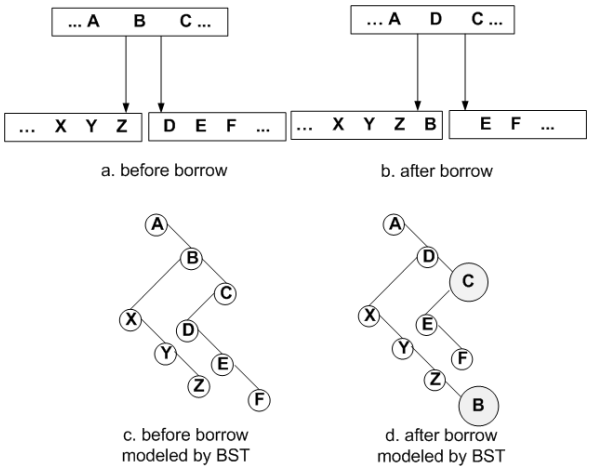

Fig. 6 Borrow in SP-BT.

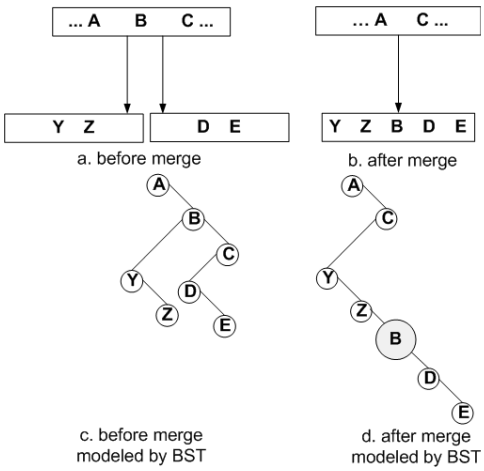

Fig. 7 Merge in SP-BT.

the B-tree. At each node, it examines all the keys in the node to find the search path. Additionally to keep track of all prefixes of *d*, it 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. Except for the node splitting, it behaves similar to SP-BST in modifying the existing match vectors. However, instead of adding a new 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. 5. Figure 5 (a) shows the tree nodes before split. Figure 5 (b) shows the tree nodes after split. Figure 5 (c) and Fig. 5 (d) show ordering of the prefixes in Fig. 5 (a) and Fig. 5 (b) if they were implemented on SP-BST. Since in Fig. 5 (a) (Fig. 5 (c)), *X, A* and *B* are the Master keys compared to *C*, and *C* is the Slave key, the relationships of Master/Slave keys should be transferred to Fig. 5 (b) (Fig. 5 (d)) after splitting and the 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 starts from the root node by searching for the deleting prefix. The procedure is continued until it reaches the node in which the prefix is stored in its 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 the 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. 6, after deleting the prefix p, it is necessary to move keys *B* and *D*. *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. 7 (d), since *height(B)* would be larger than *height(C)*, *B* should update *C* and also *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.

3.2 SP-BTe

To enhance the software performance of the SP-BT, some modifications were made to the original SP-BT [15]. These modifications noticeably speedup the search procedures within each node, while they do not affect the node accesses or tree structure [15]. Since, in this paper we have focused on the node access performance and hardware implementation, their results would not differ from each other. Hence we omit more discussion on SP-BTe.

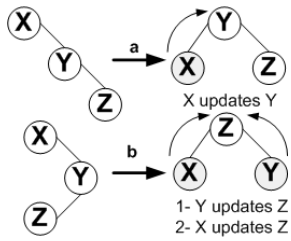

Fig. 8 Rotations and required updates.

3.3 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 these trees and their worst case search times are much less than SP-BST and with *O(log n)* order. Basically, the search and update procedures 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. Two types of rotations with their required updates are shown in Fig. 8. As it is shown in Fig. 8 (a), in this rotation, *X* should update *Y* and then reset its corresponding match vector bits to zero. In Fig. 8 (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 the comparisons are presented in the next section.

4. Comparison Results

4.1 Software and Hardware Implementations

We implemented different versions of our proposed algorithms for both IP versions IPv4 and IPv6 in software:

- The B-tree version of Scalar Prefix Search, SP-BT(SP-BTe) and Coded Prefix search, CP-BT
- SP-RB and SP-AVL
- The Red-Black and AVL tree versions of "Coded Prefix Search" named CP-RB and CP-AVL.

Additionally, to compare our algorithms with other solutions, two recent B-tree solutions PIBT [10] and BTLPT [11] and one Trie based solution LPFST [12] were implemented in software for both IP versions IPv4 and IPv6. A simple hardware implementation is also done using FPGA. Of course, there was not any emphasize on optimizing its performance through pipeline or parallel implementation.

Finding the next hop is not the main goal of the lookup problem and sometimes it is needed to combine the results of the lookup and some other engines like multi-field packet classifier to extract the next hop. Therefore, it can be done in many simple ways e.g. using a good hash function working in parallel with the main data structures. Using this scheme, next hop can be extracted in about one or two memory accesses using the LMP values, while the main data structure is being searched for the next input address.

It should also be mentioned that it is possible to store each prefix with its next hop in Coded Prefix Trees such as CP-BT, CP-RB and CP-AVL. However, this would need an additional search to locate the LMP and its next hop in the tree (after finding the LMP value).

Similar scheme might be applied in scalar prefix trees like SP-BT(SP-BTe), SP-RB and SP-AVL. But, obviously it required a more complicated memory management to efficiently store the next hop pointers in accordance with each bit of the match vector. Since PIBT does not support storing the next hop pointers in its structure, we did not consider the next hop pointers in the simulation results to be fair in the comparisons of storage requirements.

4.2 Used Databases

To compare different solutions for IPv4 databases, three IPv4 prefix databases AS4637, AS1221 and AS131072 have been used. The first one which contains 139519 prefixes was downloaded from [16] in August 2008. The second one contains 191566 prefixes and it was downloaded from [16] in August 2008. The third one which contains 313453 prefixes was downloaded from [16] in January 2010.

Also to compare different solutions for IPv6, two IPv6 databases AS1221 and AS131072 have been used. The first one contains 933 prefixes which was downloaded from [16] in August 2008 and the second one that contains 2523 prefixes was downloaded in January 2010 from [16].

4.3 Software Test Setup

To make sure that the results are independent from the CPU model, cache size or other restricting issues, all software simulations are compared based on the number of required node accesses for search and update procedures and the storage requirements. These parameters would also give a good indication of the hardware implementation efficiency and performance.

To compute the performance parameters, test scenarios were repeated several times using members of those databases with random ordering and averaged.
The test method is as follows:

First, all of the prefixes of a database were inserted into the structure to find the storage requirement. After that, each prefix was deleted and reinserted again. This may change the tree structure and create another level of randomness. Each time the insertion or deletion is done, the number of node accesses was computed. This procedure is done sev-

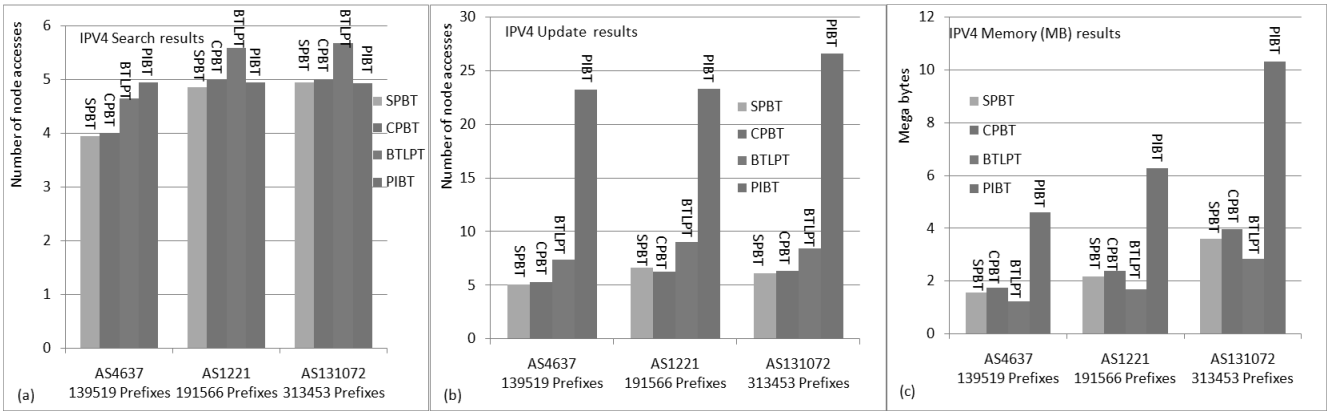

Fig. 9 The results of B-tree schemes for IPv4 databases.

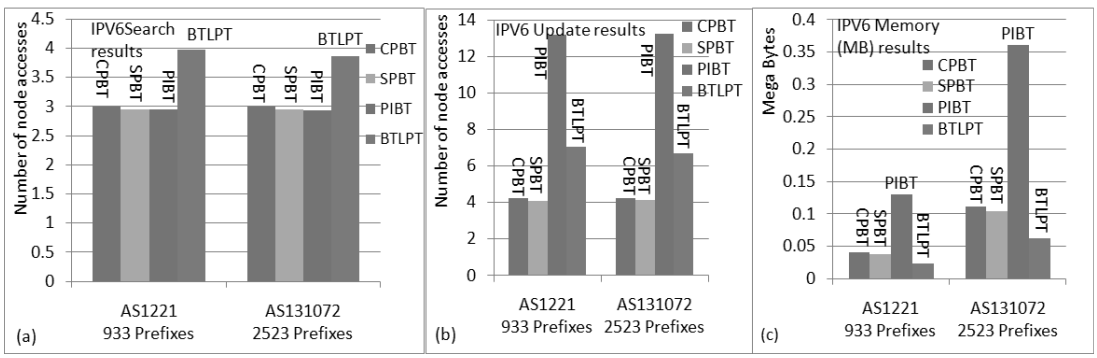

Fig. 10 The results of B-tree schemes for IPv6 databases.

eral times for all prefixes using a random ordering of the prefixes.

Each time a tree is constructed, searches are done using IP addresses which are constructed using prefixes of the databases.

4.4 The Results of B-Tree Schemes for IPv4 and IPv6 Databases

In the results presented in this paper, the minimum degree of the B-tree is t=14. However, similar results have been obtained for other degrees. Figure 9 shows the search (part a), update (part b) and memory (part c) results of CP-BT and SP-BT compared to PIBT and BTLPT for IPv4 databases.

As it is shown in Fig. 9 (a), the required number of node accesses of the search procedure of SP-BT (or SP-BTe) is the best for all three databases. The CP-BT has also comparable results. Similar update results are also shown in Fig. 9 (b). Also, Fig. 9 (c) shows the results of storage requirements of these solutions. Figure 10 shows the similar search, update and storage results of the above B-tree schemes for IPv6 databases.

In summary comparison to PIBT, the average search improvement of SP-BT(SP-BTe) might be small, but the update performance has improved substantially. This is due to the fact that for each prefix update, PIBT needs to traverse the tree three times; i.e. twice to update the prefix end points and once to update its vectors, however; the proposed scheme needs to traverse the tree only once. Also, the storage requirements of our algorithms are better than PIBT as it is shown in Fig. 9.c. The reason is that, in addition to the child pointers of tree nodes, PIBT needs about six w-bit vectors to store a prefix, while our algorithm needs two w-bit vectors in worst case. Obviously, considering its compression capability, the storage requirement reduces further. This fact was described in Sect. 2.4. Similar results exist for the worst case performances.

In comparison to BTLPT, the storage requirement of BTLPT is slightly less than our algorithms, but both search and update performances of our algorithms have been improved a lot because they do not depend on the prefix length contrary to BTLPT. Please also note that the presented performances are for the average case. In the worst case, the search procedure of BTLPT would degrade by a big factor due to its dependency on Trie based search of its LPFST part. Similar situation exists for its worst case update procedure. But for the proposed algorithm, the worst case performances do not differ much since it does not have any additional Trie part.

4.5 Results of Other Balanced Tree Schemes for IPv4 and IPv6 Databases

After extending the idea of CP-BT and SP-BT to Red Black

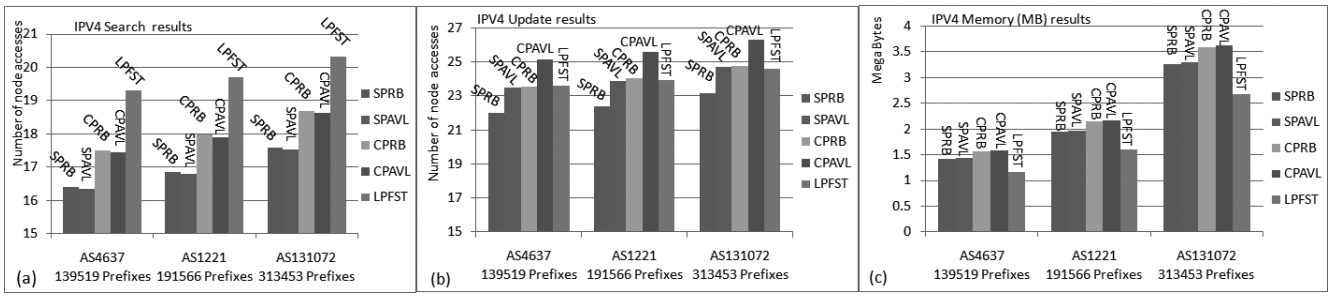

Fig. 11 The results of other balanced tree schemes for IPv4 databases.

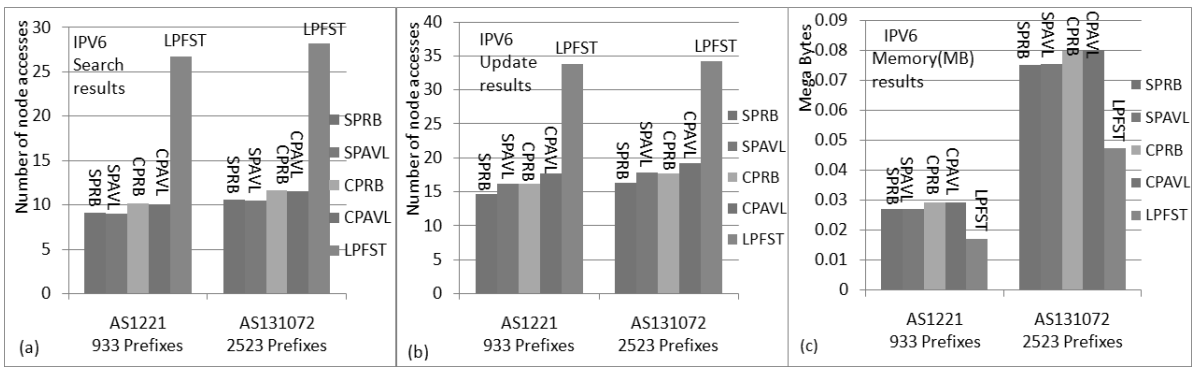

Fig. 12 The results of other balanced tree schemes for IPv6 databases.

and AVL Binary Balanced trees which were called CP-RB and CP-AVL, SP-RB and SP-AVL respectively, their results were compared with LPFST which is a binary Trie. Figures 11 and 12 show their search, update and memory results for IPv4 and IPv6 prefix databases. As it is depicted in these figures for average case, although LPFST has slightly better storage results, but the search results of SP-RB, SP-AVL, CP-RB and CP-AVL are better than LPFST for all IPv4 and IPv6 databases and also the update results of SP-RB are the best among them.

Again, in the worst case scenario, the performance of LPFST would degrade much more, due to its Trie based architecture and possible growing of the tree height as a function of w.

This dependency of the performance of LPFST to the Trie height and w does show itself for the IPv6 even for the average case and small number of prefixes in database.

4.6 Hardware Implementation of SP-BT

To show the efficiency and performance of hardware implementation of our algorithm, its B-tree version SP-BT has been implemented in hardware. For this purpose a small Xilinx Virtex 6 FPGA ($xc6vlx75t-3-ff484$) was used. The minimum degree of the B-tree, t, was set to 4 in the design. This structure can have 16K Nodes, at least 32K and up to 112K keys. Please remember that each key may contain at least one and at most 32 IPv4 prefixes. Therefore, the structure can hold from minimum of 32K prefixes to about 2 million prefixes due to its compression capability. Here, maximum of 8 tree levels i.e. a tree with height of 7 was considered based on the minimum degree of the tree and the worst case distribution of the minimum number of prefixes in the tree.

Of course, using larger FPGA devices, the number of keys in the tree can be increased based on the selected device. Again, we note that each key may contain at least one and at most 32 IPv4 prefixes.

Without using efforts for parallelism or pipelining, the algorithm uses 10 clock cycles for lookup procedure which resulted in about 28 million lookups per second with a clock period of 3.559 ns. Also, a rate of 4.12 million updates per second was achieved.

Of course efforts might be done to implement the algorithm more efficiently using parallelism and pipelining schemes to increase the lookup rates to reach about one lookup per clock cycle.

5. Conclusion

A novel idea for Longest Prefix Search with efficient incremental updates was introduced and its performance was evaluated on several tree structures both for IPv4 and IPv6. Its performance was thoroughly evaluated and showed superior results compared to other schemes specially for incremental updates. This scheme compares prefixes just as if they are scalar numbers using an implicit coding scheme which makes them suitable to be stored in different tree structures. It uses a vector of w bits to represent the existence of up to w prefixes of a w-bit key which compresses

the tree and makes it more efficient. Additionally, since the search procedure needs to find the longest prefix (all prefixes) of an address instead of finding just one key, the store, search and update procedures for this scheme has been modified compared to ordinary tree structures. A hardware implementation was also presented which shows promising results for its use in hardware.

Acknowledgments

The authors wish to thank the anonymous reviewers for their valuable and helpful comments.

References

[1] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, Introduction to Algorithms, The MIT Press, 2001.
[2] S. Nilsson and G. Karlsson, "IP-address lookup using LC-tries," IEEE J. Sel. Areas Commun., vol.17, no.6, pp.1083–1092, 1999.
[3] V. Srinivasan and G. Varghese, "Fast address lookups using controlled prefix expansion," ACM Trans. Comput. Syst., vol.17, no.1, p.40, 1999.
[4] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," IEEE INFOCOM, pp.1241–1247, 1998.
[5] B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search," IEEE/ACM Trans. Netw., vol.7, no.3, pp.324–334, 1999.
[6] P. Warkhede, S. Suri, and G. Varghese, "Multiway range trees: scalable IP lookup with fast updates," Comput. Netw., vol.44, no.3, pp.289–303, 2004.
[7] S. Sahni and K. Kim, "O (log n) dynamic packet routing," Computers and Communications, 2002. Proc. ISCC 2002. Seventh International Symposium on, pp.443–448, 2002.
[8] H. Lu and S. Sahni, "O (log n) dynamic router-tables for prefixes and ranges," IEEE Trans. Comput., vol.53, no.10, pp.1217–1230, 2004.
[9] H. Lu and S. Sahni, "Dynamic IP router-tables using highest-priority matching," Computers and Communications, 2004. Proc. ISCC 2004. Ninth International Symposium, 2004.
[10] H. Lu and S. Sahni, "A B-tree dynamic router-table design," IEEE Trans. Comput., vol.54, no.7, pp.813–824, 2005.
[11] 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, Aug. 2007.
[12] L. Wuu, T. Liu, and K. Chen, "A longest prefix first search tree for IP lookup," Comput. Netw., vol.51, no.12, pp.3354–3367, 2007.
[13] N. Yazdani and P. Min, "Prefix trees: New efficient data structures for matching strings of different lengths," Database Engineering & Applications, pp.76–85, 2001.
[14] M. Behdadfar and H. Saidi, "The CPBT: A method for searching the prefixes using coded prefixes in B-tree," Lect. Notes Comput. Sci., vol.4982, pp.562–569, 2008.
[15] M. Behdadfar, H. Saidi, H. Alaei, and B. Samari, "Scalar prefix search: A new route lookup algorithm for next generation internet," IEEE INFOCOM, 2009.
[16] http://bgp.potaroo.net

Appendix: Lemma Proofs

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 $len(MP(r))\geq i$ and
$len(MP(l))\geq i$ and
$pref_i(MP(l))=pref_i(MP(r))$, then:
$len(p)\geq i$ and $pref_i(p)=pref_i(MP(r))$.
b. If $pref_i(p)=pref_i(MP(r))$ then $r.mv(i\text{-}1)=0$.
c. If $pref_i(p)=pref_i(MP(l))$ then $l.mv(i\text{-}1)=0$.
d. p, $MP(r)$, $MP(l)$ are disjoint.

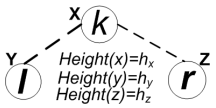

Fig. A· 1 Proof of *Lemma* 1.

Proof idea: The proof is done using contradiction.

a. Looking at Fig. A· 1, let's assume that:
$q = pref_i(MP(l)) = pref_i(MP(r))$
If $lp = len(p) < i$, since $MP(l) < p$, it results in:
$pref_{lp}(MP(l)) < p$.
Therefore $pref_{lp}(MP(r)) < p$ and it results in:
$MP(r) < p$ which is a contradiction. Therefore $len(p) \geq i$.
Now, let's consider $q' = pref_i(p)$ 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 does not affect this property.
c. The proof is similar to b.
d. It should be proved that each 2 prefixes are disjoint. The cases are:
$MP(l) \rightarrow p, MP(r) \rightarrow p, p \rightarrow MP(l)$,
$MP(r) \rightarrow MP(l), p \rightarrow MP(r), MP(l) \rightarrow MP(r)$
Consider $MP(l) \rightarrow p$. According to the insertion algorithm, as the insertion procedure of l reaches p, l will be stored in $p.mv$, which contradicts storing l in the match vector of a disjoint key. The proof is similar for the remaining cases. Similarly, it can be proved that the delete operation does not affect this property. □

Lemma 2: In Scalar Prefix Search, any time the search for address d reaches a key k that its Max-length prefix p is a prefix of d (i.e. if p=$MP(k)$ and $p\rightarrow d$), then its Max-length prefix p will be the $LMP(d)$ and therefore the search will be terminated.

Proof idea: The proof is done using contradiction. Assume

that $MP(k) \rightarrow d$ and the search is not terminated in the node containing k. If the search procedure finds another prefix p' and:
$p' \rightarrow d$, $p \rightarrow p'$
The above relations show that p' is a prefix whose existence is indicated in the match vector of a key k' and we have:
$MP(k) \rightarrow MP(k')$ or $p \rightarrow k'$
The above relations contradict *Lemma* 1.d. Therefore, the search procedure is terminated in the node containing k. □

Lemma 3: To store a new prefix p in SP-BST, let's assume that K={k_1, k_2, k_3,..., k_n} is the set of all keys stored in the tree which p is a prefix of them. If among members of K, $k_j \in K$ is the key that its node has the least height, then the prefix p will be stored only in the match vector of k_j and $k_j.mv(len(p)\text{-}1)$ will be set to *one*.

Proof idea: For the sake of simplicity, if a node x of a SP-BST contains a key e.g. key_x, we may use the notation $height(key_x)$ instead of $height(x)$ in the proof.
The insertion process will be completed by setting a bit in a match vector of a key. Using the definitions of K and k_j, it should be proved that this key is k_j. We prove it using contradiction.
Let's assume that the existence of p is indicated in the match vector of a key named $k_u \neq k_j$ after insertion. Two cases may exist:
Case A: k_u and k_j are on the same path.
Case B: k_u and k_j are not on the same path.
Let's consider *Case A* which is also depicted in Fig. A· 2. Based on the assumption of this case, there are three possibilities:
Case A.1: $height(k_u) > height(k_j)$: It is shown in Fig. A· 2 (a) and is against the assumption, since by reaching k_j the insertion procedure will be finished after setting its match vector.
Case A.2: $height(k_u) < height(k_j)$: It is shown in Fig. A· 2 (b). Again two sub-cases may exist:
Case A.2.1: k_u has been existed before the insertion of p: This is against the lemma assumption, since it means that among members of K, $k_u \in K$ is the key that its node has the least height.
Case A.2.2: k_u has been stored after the insertion of p: This means that the node containing k_u had been created before inserting p and should have contained a key e.g. k'_u. Then in the insertion process of p it is modified to k_u. Based on the insertion procedure, this case occurs only when $MP(k'_u) \rightarrow p$. This results in:
$MP(k'_u) \rightarrow$(*all members of K*)
Based on *Lemma* 1.d, the Max-length prefixes of any two keys in SP-BST are disjoint. Therefore, the above result contradicts *Lemma* 1.d.
Case A.3: $height(k_u) = height(k_j)$: This means that k_u has been stored instead of k_j in its node as it is also depicted in Fig. A· 2.c. Again, based on the insertion procedure, this case occurs if $MP(k_j) \rightarrow p$. This means that either $MP(k_j) = p$ or $MP(k_j)$ is a prefix of p except p itself. If $MP(k_j) = p$, it means that p has been existed in the tree

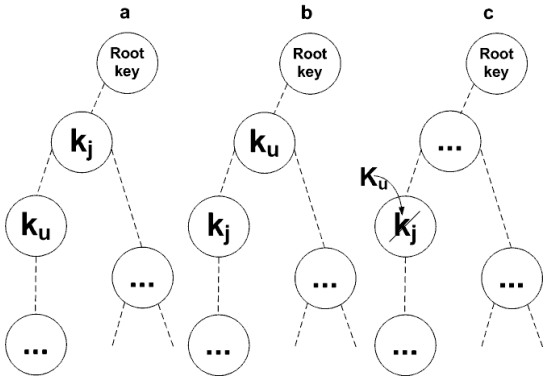

Fig. A· 2 Proof of *Lemma* 3, *Case A*.

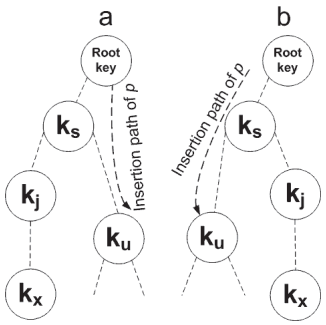

Fig. A· 3 Proof of *Lemma* 3, *Case B*.

before the insertion process. This, contradicts the lemma assumption that p is a *new* prefix. If $MP(k_j)$ is a prefix of p, since p is also a prefix of k_j, only $k_j.mv(len(p) - 1)$ should be set to '1'. This means that $k_u = k_j$, which contradicts the assumption $k_u \neq k_j$.

Now, let's consider *Case B*. Based on *Case B*, k_u and k_j are not on the same path. This means that:
Case B: The paths to k_u and k_j are separated at a node containing a key e.g. k_s as it is shown in Fig. A· 3.
By the assumption, the search path of k_u is the same as the insertion path of p. The possible positions of the keys are given in Fig. A· 3 (a) and Fig. A· 3 (b). We prove the case for Fig. A· 3 (a). For Fig. A· 3 (b), the proof will be similar.
Based on the lemma assumptions and the definition of k_u, we have:
(a) $p \rightarrow k_j$, $p \rightarrow k_u$
Using Fig. A· 3.a and based on the tree properties:
(b) $k_j < k_s < k_u$
Now, two situations may occur:
case B.1: $len(MP(k_s)) \geq len(p)$
case B.2: $len(MP(k_s)) < len(p)$
We will first consider *case B.1*. Based on the assumption in this case:
(c) $len(MP(k_s)) \geq len(p)$
Therefore, based on (a), (b) and (c) it can be concluded that:
(d) $k_j[0 : len(p) - 1] \leq k_s[0 : len(p) - 1] \leq k_u[0 : len(p) - 1]$.
But, using (a):
(e) $k_j[0 : len(p) - 1] = k_u[0 : len(p) - 1] = p$

Thus, (d) and (e) result in:
(f) $k_s[0 : len(p) - 1] = p$
which means that:
(g) $p \rightarrow k_s$
But, (g) results in:
(h) $k_s \in \{k_1, k_2, k_3, \ldots, k_n\}$
Also, since, k_j is located in a subtree of k_s, we conclude that:
(i) $height(k_s) < height(k_j)$
But finally, (h) and (i) contradict the assumption that k_j has the least height among the members of K.
Now, let's consider *case B.2*. Based on the assumption in this case:
(j) $len(MP(k_s)) < len(p)$
Also, based on (a), p is a prefix of both k_j and k_u and we have $k_j < k_s < k_u$ based on (b). Therefore, (a), (b) and (j) result in:
(k) $k_j[0{:}len(MP(k_s)) - 1] < MP(k_s) < k_u[0{:}len(MP(k_s)) - 1]$
But, based on (a) and (j):
(l) $k_j[0{:}len(MP(k_s)) - 1] = k_u[0{:}len(MP(k_s)) - 1] = p[0{:}len(MP(k_s)) - 1]$
Again, (k) and (l) result in:
(m) $MP(k_s) = k_j[0{:}len(MP(k_s)) - 1] = k_u[0{:}len(MP(k_s)) - 1]$
and (m) results in (n) and (o):
(n) $MP(k_s) \rightarrow k_j$
(o) $MP(k_s) \rightarrow k_u$
Based on *Lemma* 1.d, it can be easily verified that the Max-length prefixes of any two keys in a SP-BST are disjoint. On the other hand, based on (n), $MP(k_s)$ is a prefix of k_j. It means that either $MP(k_s) \rightarrow MP(k_j)$ or $MP(k_j) \rightarrow MP(k_s)$ which both contradict *Lemma* 1.d. Using a similar discussion, (o) also contradicts *Lemma* 1.d.
Therefore, the prefix p would be stored only in the match vector of k_j. It means that $k_j.mv(len(p)\text{-}1)$ will be set to *one*. □

Lemma 4: Assume that $K=\{k_1, k_2, k_3, \ldots, k_n\}$ is the set of all keys stored in the tree which p is a prefix of them and among its members, k_j is the key whose node has the least height. Then, for any arbitrary address d such that $p \rightarrow d$, the search path of d will reach the node containing k_j.

Proof idea: The proof is done using contradiction. Assume that the search patch of d does not meet k_j. In this case, consider k_s as the separation point of the search path of d and k_j. The remainder of the proof would be similar to the *Case B* in the proof of *Lemma* 3. □

Mohammad Behdadfar was born in September, 1977 in Iran. He received his B.Sc. degree in Electrical Engineering in 1999 and his M.Sc. degree in 2002 both from Isfahan University of Technology (IUT). Mr. Behdadfar is currently a Ph.D. candidate in the department of Electrical and Computer Engineering at IUT. His current research interests are in the area of High speed networking, Switch/Router design and Algorithms.

Hossein Saidi received B.S and M.S. degrees in Electrical Eng. in 1986 and 1989 respectively, both from Isfahan University of Technology (IUT), Isfahan Iran. He also received D.Sc. in Electrical Eng. from Washington University in St. Louis, USA in 1994. Since 1995 he has been with the Dept. of Electrical and Computer Engineering at IUT, where he is currently an Associate Prof. of Electrical and Computer Engineering. His research interest includes high speed switches and routers, communication networks, QoS in networks, queueing system, security and information theory.

Masoud-Reza Hashemi received his B.Sc. and M.Sc. degrees from Isfahan University of Technology in 1986 and 1988 respectively, and his Ph.D. from University of Toronto in 1998 all in Electrical and Computer Engineering. From 1988 to 1993 he was with Isfahan University of Technology as a faculty member. From 1998 to 2000 he was a Postdoctoral Fellow at University of Toronto. Massoud Hashemi joined Accelight Networks as a founding member in 2000. Since 2003 he is with Isfahan University of Technology. His current research interests include Communication Networks, Next Generation Services, TE in IP/MPLS, and Sensor Networks.

Ali Ghiasian was born in May, 1977 in Iran. He received his B.Sc. degree in Electrical Engineering in 1999 and his M.Sc. degree in 2002 both from Isfahan University of Technology (IUT). Mr. Ghiasian is currently a Ph.D. candidate in the department of Electrical and Computer Engineering at IUT. His current research interests are in the area of Wireless Ad Hoc networks and Algorithms.

Hamid Alaei was born in June, 1986 in Iran. He received his B.Sc. degree in Computer Engineering in 2009 and Currently a M.Sc student of Software Engineering in Amirkabir University, Tehran, Iran.