IJICTR International Journal of Information & Communication Technology Research

Volume 4- Number 4- September 2012 (1-12)

Route Lookup Algorithms Using the Novel Idea of Coded Prefix Trees

Mohammad Behdadfar
Engineering Department of IRIB University
Tehran, Iran
Behdadfar@iribu.ac.ir

Hossein Saidi
Department of Electrical & Computer Engineering
Isfahan University of Technology
Isfahan, Iran
hsaidi@cc.iut.ac.ir

Masoud-Reza Hashemi
Department of Electrical & Computer Engineering
Isfahan University of Technology
Isfahan, Iran
hashemim@cc.iut.ac.ir

Received: February 23, 2012- Accepted: July 4, 2012

Abstract— **This paper introduces a new prefix matching algorithm called "Coded Prefix Search" and its improved version called "Scalar Prefix Search" using a coding concept for prefixes which can be implemented on a variety of trees especially limited height balanced trees for both IPv4 and IPv6 prefixes. Using this concept, each prefix is treated as a number. The main advantage of the proposed algorithms compared to Trie-based solutions is that the number of node accesses does not depend on IP address length in both search and update procedures. Therefore, applying this concept to balanced trees, causes the search and update node access complexities to be *O(log n)* where *n* is the number of prefixes. Also, compared to the existing range-based solutions, it does not need to store both end points of a prefix or to store ranges. Finally, compared to similar tree based solutions; it exhibits good storage requirements while it supports faster incremental updates. These properties make the algorithm capable of potential hardware implementation.**

Keywords- Coded Prefix; Scalar Prefix; Route Lookup; Longest Matching Prefix

I. INTRODUCTION

Longest Prefix Matching or *LPM* is the problem of finding the longest prefix of a *w*-bit address *d*, among a set of binary prefixes with the maximum length of *w* stored in a router table. A straightforward algorithm to find the Longest Matching Prefix or *LMP* of a given address is linear search [1]. Considering *n* as the number of prefixes, the complexity of this algorithm will be *O(n)* which is not acceptable for large databases.

Using a Radix tree or Trie [1] in which each tree edge corresponds to one bit, the search and update complexities become *O(w)* which is better than linear search where *w* is 32 for IPv4 and 128 for IPv6. However, the search and update complexities of Trie are not acceptable for high speed switches. Some other versions of Trie such as Patricia [2], LC-Trie [3], Prefix Expansion [4], LPFST [5] or [6], [7] have also been introduced to improve search performance. But most of them suffer from the similar drawback and also do not support incremental updates or would not be extendable to

IPv6. Some hardware schemes have also been introduced to improve the performance of Trie based structures using pipelining [8], searching different lengths on parallel RAMs [9], using FPGA[11] or parallel RAMs and FPGA [10], using graphics processor unit (GPU) [12] and compact clustered tries [13]. However, the main drawback of the Trie-based searches still exists in these newly introduced methods.

To reduce the dependency of prefix search algorithms to the address length, some "range based" algorithms were introduced. In these algorithms, two *end points* are defined for each prefix and stored in a binary search tree. By defining a range for each prefix, the search in the tree may be done by finding the most specific range corresponding to an address [14]. The node access complexity of this structure is independent from *w* for the search procedure. The range based prefix search algorithms often need long time for the prefix update procedures even for some of those which used balanced trees to store the prefix end points [15]. To support the fast search and also incremental updates, some range-based algorithms were introduced by the authors of [16] among which PIBT has the best search performance [16]. Since PIBT stores prefix end points in a B-tree, its search and update complexity is *O(log(n))*. It should be mentioned that *two* additional *w*-bit vectors for each prefix endpoint are applied in PIBT structure. Therefore, since each prefix has two end points, about 6 vectors might be stored in the tree instead of the prefix itself. This leads to a large storage requirment. One of the well known range-based lookup algorithms with better average results than PIBT is BTLPT [17]. This algorithm uses two structures: a B-tree for disjoint prefixes and an LPFST for the remaining prefixes. However, the complexity of BTLPT still depends on *w,* based on the Trie-based behavior of LPFST.

Some other range based methods like using segment trees [18] or [19] have also been introduced, however they still suffer from the main drawback of long prefix update procedures.

"Prefix Trees" and "M-way Prefix Trees" are other algorithms proposed in [20], 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)* like Trie-based solutions and the long update procedures.

It should be mentioned that other hardware solutions like using Hash[21], TCAM [22] or Bloom filters [23] have been used as well as the above methods since introduction of CIDR. However, most of them suffer from the problem of not supporting incremental updates.

In [24], 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 >. Therefore using this concept, unlike range based algorithms, it is not required to store both prefix end points in the tree. The main idea of CP-BT is extended in this paper by introducing Coded Prefix Trees in which the tree height does not depend on IP address length. The idea of Coded Prefix Trees can be applied to many types of trees especially balanced trees like B-tree, RB-tree [1] and AVL-tree[1]. It is also extended to another version of the algorithm called "Scalar Prefix Search" which was partially presented in [25], [26] and [27]. In this paper, the results of both versions are compared with some competitive well-known algorithms like PIBT, BTLPT and LPFST for both IPV4 and IPV6 prefix databases.

Based on the above discussions, the main weakness of Trie-based algorithms is their dependency of number of Trie node accesses to IP address length for search and update procedures. Also, other important weakness of range based algorithms is their inability of performing incremental updates and high storage requirements. The novelty of the idea which is introduced in this paper is covering the followings:

- The number of node accesses for search and update procedures does not depend on the IP address length.
- In comparison with most of the range based solutions, this scheme fully supports incremental updates in a single tree downward pass.
- The proposed schemes are implementable on most of the tree data structures among which balanced trees are selected in this paper.
- Coded Prefix Search is the first scheme that introduces a coding concept for comparing the prefixes like numbers without considering two end points per prefix. Actually, this coding concept plays the main role in most parts of the algorithms.

The rest of the paper is organized as follows: The main idea of Coded Prefixes is reviewed in section 2. Section 3 describes the Coded Prefix Search algorithm. Scalar Prefix Search, the improved version of Coded Prefix Search is introduced in section 4. Application of both versions to balanced trees is discussed in section 5. Section 6 presents the complexity analysis, the implementation and simulation results of both versions of the algorithm compared to some well-known competitive solutions. Section 7 concludes the paper. The Appendix corresponds to lemma proofs.

II. Coded Prefixes: The Main Idea

We propose a coding 'concept' in which unlike the range-based algorithms, prefixes are treated as numbers. Consider a *k*-bit prefix *p*. Since the IP address is considered to be *w* bits, the number of blank places of *p* is '*w-k*'. For comparison, each bit of *p* would be encoded using a 2-bit value as follows: for each bit '1', the 2-bit '10'; for each bit '0', the two-bit '01' and for each *blank place*, the 2-bit '00' is used. For example, considering *w*=5, the prefix 01* will be encoded to 0110000000.

Based on this definition, the prefixes can be compared like numbers using "=", "<" and ">". As an example, we will have:

00001* < 0001* < 001* < 0010* < 010*< 100*

This definition leads us to the following lemma. .

Lemma1- consider two prefixes *p, q*. If *p* is a prefix of *q*, then $p \leq q$.

Proof- It is given in the appendix

Please note that we will use the 'concept' of the mentioned coding to *only* 'describe' the algorithm, but 'not to store the prefixes'. The prefixes are stored in the memory as a usual 'w' bit vector with one additional vector which will be introduced in the next section.

III. CODED PREFIX SEARCH: THE IDEA

The method of section 2 can be applied to many types of trees. Although it is not efficient to apply this method to Binary Search Tree, just to simply describe the 'Coded Prefix Search' procedures, we will apply it to Binary Search Tree and then will extend it to other trees. After applying the method to this tree, it is named "Coded Prefix Binary Search Tree" or CP-BST. First let's define some notations:

- *len(p)* shows the length of a prefix *p*.
- *p(i)* shows i^{th} bit of prefix *p*.
- For each prefix *p* with *len(p)=k,* and *k<w*, we add *w-k* zero pads 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"

  e.g. if *w=4* and *p*=101*, then: *key(p)*="1010".
- The notation $p \rightarrow q$ shows that *p* is a prefix of *q*.
- The notation $p! \rightarrow q$ indicates that *p* is not a prefix of *q*.
- If $p! \rightarrow q$ and $q! \rightarrow p$, then *p* and *q* are called "disjoint" prefixes.
- A prefix of *p* with the length of *k* is shown by $pref_k(p)$.
- The Longest Matching Prefix of a string "*S*" is denoted as *LMP(S)*.
- For a key "*r*", a *w* bit "Match Vector" is defined and abbreviated with "*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* of *r* with the length of *i+1* or *len(q)=i+1* or $q=pref_{i+1}(r)$ in the database. Please note that the Match Vector bit numbers indexing starts from "0".
- The length of the path from the root of the tree to 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.
- The longest prefix of each key derived from its match vector 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*.

Using these definitions, prefixes can be inserted into any search tree such as Binary Search Tree (BST). Although each prefix will be stored in form of two vectors called "Match Vector" and "key", the procedure may be simply mentioned as inserting the prefix instead of storing the key or Match Vector.

The insert and search procedures for CP-BST are explained in the following sections. The delete procedure is removed due to space limitation. However, detailed delete procedure of CP-BT is included in [24].

A. Insertion

Before describing the insert procedure, the following lemma should be stated:

Lemma 2- Consider a set *P* of prefixes which are inserted into a binary search tree with an arbitrary order and based on the comparison definition of section II. Consider *d* as an input IP address and assume that the objective is "to search *d*" in the tree. If p_i is a member of *P* and also $p_i \rightarrow d$, then at least one key *q* will be found in the search path of *d* such that $p_i \rightarrow q$.

Proof- It is given in the appendix.

This lemma states the main idea of the insertion procedure. Since *q* is found in the search path and $p_i \rightarrow q$, if the existence of prefixes of *q* can be distinguished by some additional information in the node which stores *q*, the search procedure of *d* will find all of the prefixes of *d* on the search path. This is the reason for defining the Match Vector for a key which was stated in the last section. Based on the above lemma, the insertion procedure will be as follows:

To insert a "*newPrefix*" in the tree, 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, the "*newPrefix*" is compared with *r*.

1- If the "*newPrefix*" is a prefix of the Max-length Prefix of r, then the corresponding bit in the "match vector" of r will be set to one, and the algorithm will be continued. In other words:

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

2- The insertion procedure selects the next node to go through. Based on the result of comparison if *"newPrefix"<MP(r), then* the procedure goes to the left child of *r*, *otherwise* it goes through the right child.

3- This procedure will continue till reaching a leaf node. Then, a right or left child will be created based on the above procedure and the prefix will be inserted into this new node. It is necessary to emphasize that part 1 is done only one time during the insertion procedure of each *"newPrefix"*.

This is the result of a property of CP-BST which is stated by lemma 3:

Lemma 3- In the insertion process of a prefix p, consider the following set of inserted prefixes:

$$P=\{p_i \mid 1\leq i\leq n,\ p \rightarrow p_i\}$$

If P is not empty, for the insertion of p in CP-BST:

a. The existence of at least one member of P is indicated in the match vectors of insertion path.

b. It is only necessary to update the match vector of the first visited member of P in the insertion path of p.

Proof- part 'a' can be proved using lemma 2. However, a complete proof for parts 'a' and 'b' is given in the Appendix.

Lemma 3 and its proof point to an important property of Coded Prefix Search trees which we call "Master/Slave property" and it is also true for CP-BST. According to Master/Slave property, starting from the root of the tree:

For each sub-tree S of a coded prefix tree, a match vector bit stored in *root(S)* (which indicates the existence or nonexistence of a prefix in its sub-tree) overrules all of the match vector bits of sub-tree S for the same prefix. In other words, if p is prefix of both the key stored in the root of S (named "*skey*") and another key k stored somewhere in S and *len(p)=i,* then *skey.mv(i-1)* indicates the existence or nonexistence of p in the database. This means that *k.mv(i-1)* will be overruled by *skey.mv(i-1)*.

For an example of insertion procedure with $w=7$, consider the following prefixes:

p_1=*010000*,* p_2=*0100011,* p_3=*01000*,* p_4=*0100*,* p_5=*010* and* p_6=*00**

Also consider that they will be inserted into the tree with the following order: $p_1, p_3, p_4, p_5, p_6, p_2$

Based on the result of the insertion process in Fig.1, although p_4 is a prefix of p_3 (*01000** in node B), since it already has set one match vector bit once during its insertion (in node A of Fig.1), it does not update the match vector of p_3 (Node B) and the same procedure is done for inserting p_5 and p_6.

As an example of the Master/Slave property, look at the pairs of match vector and key in nodes *A (0011110,0100000)* and *B (0000100,0100000)* of Fig.1. The match vector and key pair of node *A*, tells us that *010*,0100*, 01000* and 010000** exist in the database. However, the same pair *(mv=0000100, key=0100000)* in node B only indicates that *01000**, the Max-length prefix of *key=0100000,* exists in the database. Since A is the Master and B is the Slave, the information of *A* overrules the information of *B* in a search procedure which reads the information of both nodes. It means that *010*, 0100*, 01000* and 010000** exist in the database.

It is worth mentioning that one of the major differences between this algorithm and Trie based algorithms is the worst case height of the tree. Most of the Tries have the worst case height of *O(w)* where w is the IP address length. But the worst case tree height of this algorithm completely depends on the number of prefixes n and the type of tree used. As shown in the next sections, using some types of balanced trees causes the worst case tree height to become *O(log(n))* which is a good result since it makes the tree height independent from the IP address length while making the algorithm capable of doing incremental updates.

B. Search Procedure

Based on the insertion procedure and lemma 2 which were stated in the previous section, the search procedure for CP-BST is as follows:

A simple search algorithm is done to search the LMP of a given address d. First of all, a match vector *d.mv* will be considered for d (without loss of generality, this vector may be considered as all "x" bits). Then, starting from the root, d will be compared with *m=MP(root node key)* and its prefixes. If any prefix of m (including m itself) is also a prefix of d, then its corresponding bit in the match vector of *m (0 or 1)* will update the same bit of *d.mv,* but only if this is the first time this bit is being updated. This is due to the fact that based on the Master/Slave property mentioned earlier, if any bit of *d.mv* has been updated once by '0' or '1' during the search procedure, it will not be updated again.

Using the result of the comparison of d and the root node key, the search will continue through one of its child nodes in a similar manner.

For an example of the search procedure in the tree of Fig.1, let's assume *d=0100010, then d.mv='xxxxxxx'* . In the root node, the search procedure, finds 010*, 0100*, and 01000* as prefixes of *d,* then *d.mv* will be updated to *xx111xx*. Since *d>0100000*, the search continues to node C. Although the corresponding match vector bits of 010*, 0100*, and 01000* are equal to zero in node C, since the corresponding bits of *d.mv* are updated to '1', according to the Master/Slave property, these bits should not update *d.mv* again. Noting that 0100011 (the Max-length prefix stored in node C) is not a prefix of d, the final *d.mv* will be *xx111xx* and therefore *LMP(d)=01000**. Note that the whole height of the tree should be traversed by the search procedure to find *LMP(d)*. For example, if *d=0100011,* checking the Max-length prefix stored in node C, causes *d.mv[6]* to become '1' and the final *d.mv* to become '*xx111x1*'which means that *LMP(d)=0100011*.

IV. Scalar Prefix Search: Another Version of Coded Prefix Search

Look at Fig.A1 and the proof of Lemma 2. Assume that the prefixes are inserted into a Binary Search Tree with an arbitrary order. Consider d as an input IP address and assume that the objective is to search *LMP(d)* in the tree. If p_i is a prefix stored in the tree and also $p_i \rightarrow d$, based on the proof of Lemma 2 and Fig.A1, if the search path of p_i is separated from the search path of d in a node containing a vector e.g. q, the relation $p_i \rightarrow q$ will always be true.

Based on the above property, we introduced Coded Prefix Trees in the last section. Since the existence of p_i is indicated by both match vectors of p_i and q, *key*(p_i) whose Max-length prefix is p_i and is located in

the left sub-tree of *q* in Fig.A1, can be removed from the tree, because its information is redundant. *Scalar Prefix Trees* are introduced based on the idea of removing all such redundancies and compressing the Coded Prefix Trees as much as possible. Removing these redundancies causes the prefixes of each node to become completely different from the other nodes i.e. each node key and its match vector are representatives of a set of prefixes that do not exist in any other node.

The idea of scalar prefix trees is also applicable to many types of trees including balanced trees such as B-tree, RB-tree and AVL-tree by some modifications in their search and update procedures. However, to simply describe the main idea, we explain its application to Binary Search Tree and call it Scalar Prefix Binary Search Tree or SP-BST. For the details of its application to the B-tree (called SP-BT and SP-BTe), the Red-Black tree (called SP-RB) and the AVL-tree (called SP-AVL), and also the major modifications in the search and update procedures of these trees, refer to [25].

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 the algorithm returns. In other words:

If "*newPrefix*" →*MP(r)*, then:

$$r.mv(len(newPrefix)-1)= 1.$$

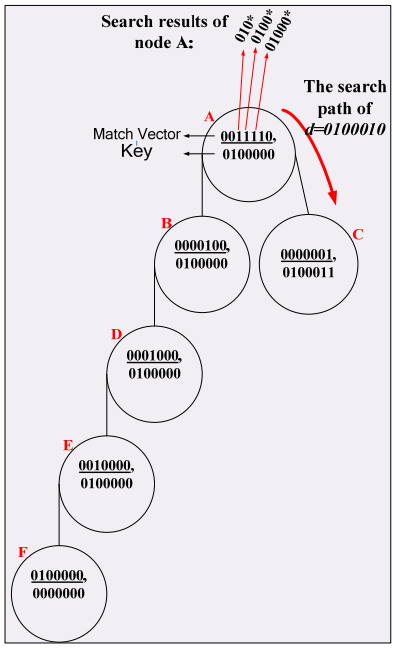

Figure 1 Example for insertion steps into CP-BST

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 or:

If *MP(r)* → "*newPrefix*", then:

$$r.mv(len(newPrefix)-1)=1 \text{ and } r=key(newPrefix).$$

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*, or else it goes through the right child.

This procedure will continue till it is terminated in a node or it reaches a leaf node but is 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 the insertion process, consider the prefixes of the example of section III.A with the same arriving order. Fig.2 shows the tree after the insertion of the above prefixes. Comparing Fig.2 with Fig.1, the SP-BST of the above prefixes, shows a good compression ratio and also a shorter tree height compared to Coded Prefix Trees. Details of the prefix deletion procedure for SP-BST are included in [25].

B. Search Procedure for SP-BST

The search procedure for the Longest Matching Prefix of 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 other words, let's consider key_n as the key stored in *n*. If $MP(key_n) \rightarrow d$, then:

$MP(key_n)= LMP(d)$ 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.

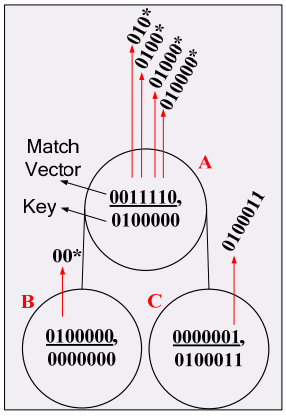

Figure 2 The SP-BST for the prefixes of Figure 1

For example, assume that the objective is to find *LMP(d)* in Figure 2 considering *d=0100010*. The search starts from the root node. Key_r*=0100000* is stored in this node.

Since $MP(Key_r)! \rightarrow d$, but some other prefixes of Key_r match *d*, their corresponding bits in *d.mv* will be set to *one*. Therefore, *d.mv=0011100*. Then, since $d > Key_r$, the procedure should check the right child of the root node. Since in the right child node, no new matching prefix of *d* is found, the procedure will be terminated after checking the match vector and key of node C. Therefore, *LMP(d)=01000** which corresponds to the least significant '*one*' in *d.mv*.

As another example, consider *d=0100001*. Checking the root node, $MP(Key_r) \rightarrow d$ (010000*→0100001). Therefore, the first condition is met. This, guarantees that $LMP(d)= MP(Key_r)=p_1=010000*$. In this case, it is not necessary to continue traversing the tree.

C. Properties of Scalar Prefix Trees

Based on the search and insert procedures and the above examples, SP-BST has some properties listed below:

a. The Max-length prefixes of all node keys in the tree are disjoint. For example, in Figure 2, the disjoint Max-length prefixes of the nodes are *010000**, *0100011* and *00**.

b. In Scalar Prefix Search, any time the search for address *d* reaches a key *k* whose 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 one advantage of Scalar Prefix Trees compared to Coded Prefix Trees because the search may be terminated in a non-leaf node. For the proof, please look at the *Lemma 4* which is explained in the appendix.

c. A prefix is stored in the match vector of only one key in the tree.

d. If *p* is a prefix of $k_1, k_2, k_3, \ldots, k_n$ 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: $k_j.mv(len(p)-1)=1$.

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. This is also one advantage of Scalar Prefix Trees compared to Coded Prefix Trees because a key of SP-BST may contain up to *w* prefixes. Therefore, the average height of the tree is reduced. On the other hand, compared to range based algorithms, since all of *w* prefixes of a key can be stored in "one" pair of (match vector, key) and also our scheme does not need to store both of the end points of a prefix, the average storage would be reduced as well. Compared to Trie based solutions, it has the advantage of not being dependent on the IP address length in the number of node accesses for both search and update procedures.

V. Balanced Tree Versions of the Main Idea

Since there is no guarantee for the height of the SP-BST and the CP-BST, the concept of Coded and Scalar Prefix Search has been applied to some balanced trees such as B-tree (named CP-BT, SP-BT), RB-tree (named CP-RB, SP-RB) and AVL-tree (named CP-AVL, SP-AVL). 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 is *O(log n)* as well.

VI. Complexity Analysis and Comparison Results

Since there is no guarantee for the height of the CP-BST or SP-BST, we focus on the balanced tree versions of both algorithms in finding the search, update and memory complexities for hardware implementation.

Since the height of a CP-BT or SP-BT with the degree *t* is always less than $\log_t(n+1)/2$ [1] in which *n* is the number of prefixes, the number of node accesses for search, insert and delete will be $O(\log_t n)$. Similarly, it will be *O(log n)* for Red-Black and AVL versions.

Many linear operations like shifting operations are done in the update procedures in each node. 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. Similar discussions can be done for the search operations. Therefore the algorithms have better search and update performances in hardware implementation.

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 and Coded Prefix search, CP-BT

- The Red-Black and AVL tree versions of "Coded Prefix Trees" and "Scalar Prefix Trees" named CP-RB, CP-AVL and SP-RB, SP-AVL.

Additionally, two famous B-tree solutions PIBT [16] and BTLPT [17] and one Trie based solution LPFST [5] were implemented in software using real databases for both IP versions IPV4 and IPV6, to compare our algorithms with other solutions.

A. Used Databases

To compare different solutions for IPV4 databases, three IPV4 prefix real databases AS4637, AS1221 and AS131072 have been used. The first one which contains 139519 prefixes was downloaded in August 2008 from [28] which is the main reference for IPv4 and IPv6 real databases. The second one contains 191566 prefixes and it was downloaded from [28] in August 2008. The third one which contains 313453 prefixes was downloaded from [28] in January 2010.

Also two IPV6 databases AS1221 and AS131072 have been used to compare different solutions for

IPV6. The first one contains 933 prefixes which was downloaded from [28] in August 2008 and the second one that contains 2523 prefixes was downloaded from [28] in January 2010.

B. 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 were averaged. The test method is as follows:

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

C. The Results of B-tree Schemes

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 3 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 shown in Fig 3.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 3.b. Figure 3.c shows the results of storage requirements of these solutions. It is clear from the results that although the average search improvement of SP-BT(SP-BTe) might be small compared to PIBT, the update performance has improved substantially. Also, while the memory storage of BTLPT is slightly less than our algorithms, both search and update performances of our algorithms have been improved a lot.

Please also note that the presented performances are for 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. A similar situation exists for the worst case update procedure of both BTLPT and PIBT.

Figure 4 shows the search, update and storage results for the above B-tree schemes for IPV6 databases. Checking these figures, a similar conclusion can be made for IPV6.

D. The Results of other Balanced Tree Schemes

After extending the idea of CP-BT and SP-BT to Red Black 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 5 and 6 show their search, update and memory results for IPV4 and IPV6 prefix databases. As depicted in these figures for average case, although LPFST has slightly better storage results, the search results of SP-RB, SP-AVL, CP-RB and CP-AVL are better than LPFST for all IPV4 and IPV6 databases, and the update results of SP-RB are the best among them. This result is due to the balanced structure of these trees compared to LPFST.

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.

VII. Conclusion

In this paper, novel schemes called 'Coded Prefix Search' and 'Scalar Prefix Search' which introduce a coding concept for prefixes to make them numerically comparable were proposed. Using this concept, in Coded Prefix Trees, a pair of (match vector, key) is used to show a prefix. However, in Scalar Prefix Trees, this pair is the representative of at most w prefixes which makes the tree more compressed. Various tree data structures among which balanced trees are preferred due to their limited $O(\log n)$ height; where n is the number of prefixes; may be used by these schemes. The required operations to correctly perform the search and update procedures were given and proved. The schemes were implemented and simulated in software using B-tree, Red Black tree and AVL-tree, and the results were compared with those of current well-known competitive solutions which also use balanced trees like PIBT and BTLPT or Trie-based solutions like LPFST for both IPV4 and IPV6 databases. To be independent from the software or hardware platforms, all simulations are compared based on the number of required node accesses for search and update procedures and the storage requirements. Both two proposed schemes, show superior results for search, update and also storage requirements both in average and worst case.

Finally, the main contribution of this work is treating prefixes like numbers. This makes the ability of searching and updating prefixes without being depended on IP address length and also the possibility of fast incremental updates compared to some well-known competitive solutions.

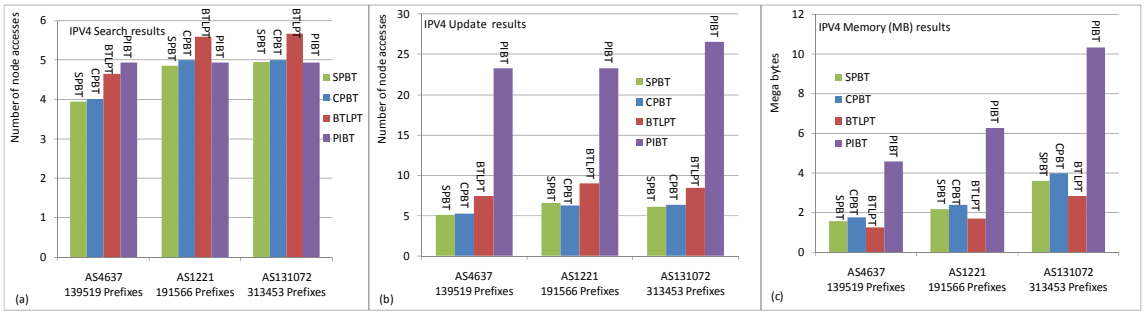

Figure 3 The results of the B-tree schemes for IPV4 databases

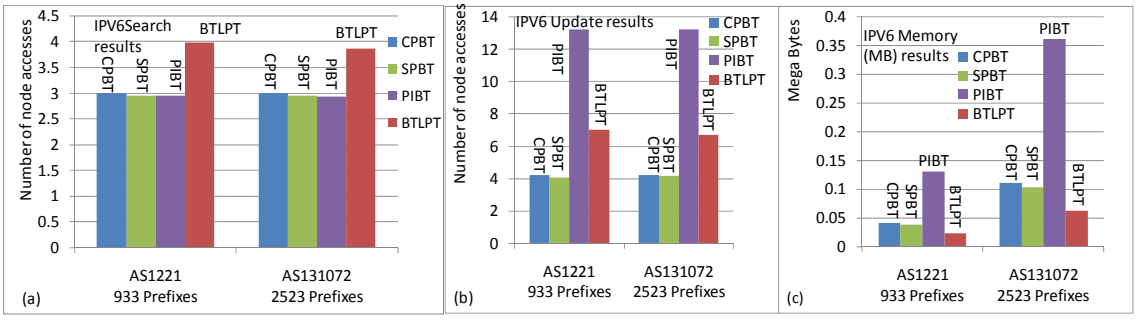

Figure 4 The results of the B-tree schemes for IPV6 databases

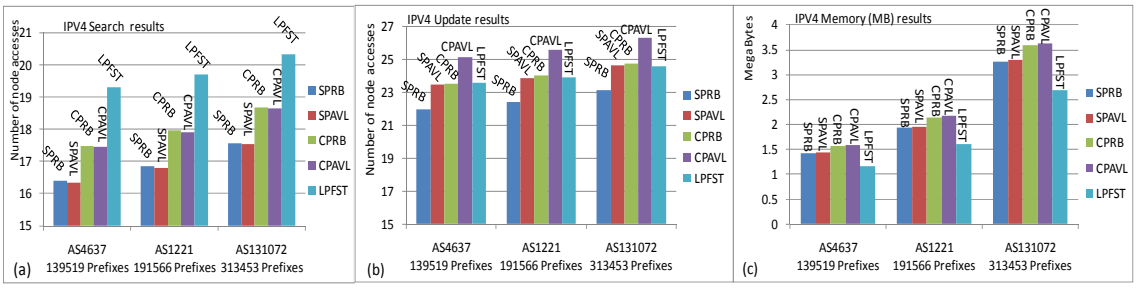

Figure 5 The results of other balanced tree schemes for IPV4 databases

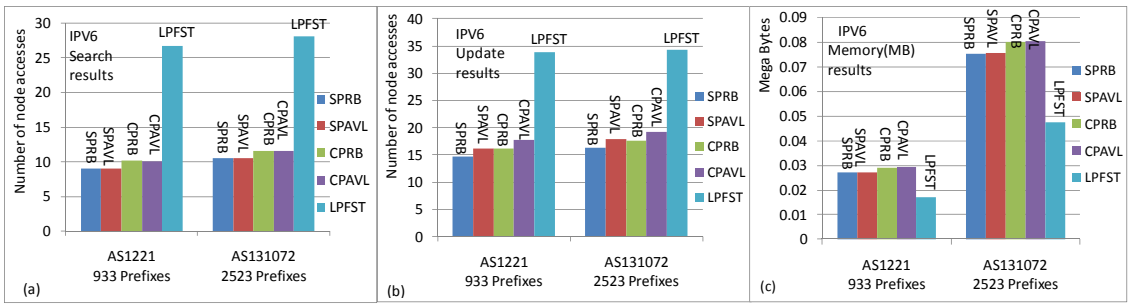

Figure 6 results of other balanced tree schemes for IPV6 databases

APPENDIX

Lemma1- consider two prefixes *p, q*. If *p* is a prefix of *q*, then $p \leq q$.

Proof- Since *p* is a prefix of *q*, if $len(p)=l_p$, the first *lp* bits of *p* and *q* will be the same. Also, the "$lp+1^{th}$ place of p will be blank space which is less than or equal to the "$lp+1^{th}$ place of *q* which is 0,1 or blank space. Therefore, *p* will be less than or equal to *q*.

Lemma 2- Consider a set *P* of prefixes which are inserted into a binary search tree with an arbitrary order. Consider *d* as an input IP address and assume that the objective is to "search *d*" in the tree. If p_i is a member of *P* stored in the tree and also $p_i \rightarrow d$, then at least one vector *q* will be found in the search path of *d* with the property of $p_i \rightarrow q$.

Proof idea- The proof uses contradiction. Consider that the conclusion of the lemma is not true. If p_i is in the search path of *d*, then $q=p_i$ which contradicts the assumption. If p_i is not in the search path of *d*, assume that the search path of p_i is separated from the search path of *d* in a node containing a vector e.g. *q* (Figure A1). Also, the following relationships can be verified from Figure A1 and the assumptions of the lemma:

$p_i \leq q,\ q \leq d,\ p_i \rightarrow d$ (1)

$p_i \rightarrow d$ results in:

$p_i(0:len(p_i)-1) = d(0:len(p_i)-1)$ (2)

Now, two states may exist:

$len(q) \geq len(p_i)$ and $len(p_i) > len(q)$.

If $len(q) \geq len(p_i)$, we can say:

$p_i(0:len(p_i)-1) < q(0:len(p_i)-1)$ (3)

(2), (3) result in:

$d(0:len(p_i)-1) < q(0:len(p_i)-1)$ (4)

which means $d<q$ which contradicts (1).

If $len(p_i) > len(q)$, the proof will be similar.

Therefore, at least one vector *q* will be found in the search path of *d* with the property of $p_i \rightarrow q$.

Lemma 3- In the insertion process of a prefix *p*, let's consider the following set of inserted prefixes:

$P=\{p_i \mid 1 \leq i \leq n,\ p \rightarrow p_i\}$

If *P* is not empty, then for insertion of *p* in CP-BST:

a. The existence of at least one member of *P* is indicated in the match vectors of insertion path.

b. It is only necessary to update the match vector of the first visited member of *P* in the insertion path of *p*.

Proof of a:

Select a prefix p_i *P*. $p \rightarrow p_i$ results in $p<p_i$. Therefore *p* should be inserted in the left side of p_i in the tree. This is shown in Figure A2.

Assume that *k* is the Max length prefix of the node key, *at which the search path of p is separated from the search path of* p_i (Figure A2). Looking at Figure A2, it results in:

$p<k<p_i$.

We will show that in this case $p \rightarrow k$ .To prove this, we use contradiction. Let's assume $p \not\rightarrow k$, then two cases may exist:

Case a.1- $len(k)<len(p)$

Case a.2- $len(k) \geq len(p)$,

we check each case separately.:

Case a.1. $len(k)<len(p)$:

Since $p<k$, and $len(k)<len(p)$, we conclude: $p(0:len(k)-1)<k(0:len(k)-1)$.

Note that $p(0:len(k)-1)$ and $k(0:len(k)-1)$ represent the first $len(k)$ bits of each of *p* and *k* .

On the other hand, since $p \rightarrow p_i$, and $len(k)<len(p)$, we conclude:

$p(0:len(k)-1)=p_i(0:len(k)-1)<k(0:len(k)-1)$.

This results in $p_i<k$ which is a contradiction.

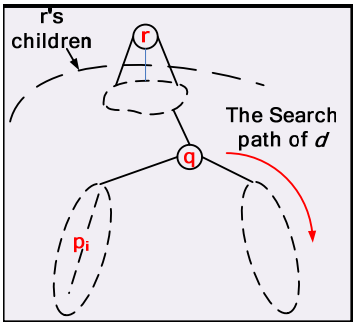

Figure A1 A BST for prefixes using the concept of coded prefixes

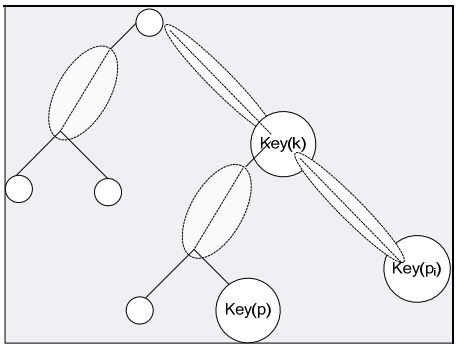

Figure A2 The places of *p* and p_i in the tree

Case a.2. $len(k) \geq len(p)$:

This case results in:

$p(0:len(p)-1)<k(0:len(p)-1)$.

On the other hand, since $p \rightarrow p_i$, it is clear that:

$p(0:len(p)-1)=p_i(0:len(p)-1)<k(0:len(p)-1)$.

This will lead to $p_i<k$ which is also a contradiction with Figure A2.

As these two cases contradict the assumptions, it can be concluded that $p \rightarrow k$ which means $k \in P$. Therefore, at least one of the members of P is traversed along the insertion search path of p.

Proof of b:

Again we will prove it using contradiction. Consider the following definition for *key(k)*:

"key(k)" is the first stored key in the search path for insertion of p with the property of $p \rightarrow k$ where k is *MP(key(k))*.

Since $p \rightarrow k$ then previous definitions and lemmas result in:

(1) $k(0:len(p)-1)=p(0:len(p)-1)$

(2) $key(p) \leq key(k)$

Assume that part *b* of the lemma is not true. It means that it is not sufficient to only update the match vector of *key(k)* in the search path for insertion of p.

As a result of this assumption, if only the match vector of *key(k)* is updated in the update procedure of p, there should be at least one LPM search procedure that *will not find* p for an arbitrary address d for which $p \rightarrow d$.

(3) Since $p \rightarrow d$, it results in $key_p \leq d$,

Now, consider the following definition for key_j:

key_j is the first key in the search path of d with the property of $p \rightarrow j$ (or $p \rightarrow (MP(key_j)=j)$),

Now, two cases may occur:

Case b.1: key_j exists

Case b.2: key_j does not exist.

Let's consider case b.1. Based on the existence of key_j:

(4) Since $p \rightarrow j$ then, $j(0:len(p)-1)=p(0:len(p)-1)$

(5) $Key_p \leq key_j$

It is also shown in Figure A3

Since based on the assumption the search procedure should not find p, then:

(6) $key_j.mv(len(p)-1)=0$

This means that the insertion path of p is separated from the search path of d in a node e.g. *"n"* storing a key e.g. key_s whose Max length prefix is *"s"*. Now two cases may occur,

Case b.1.1: $d<key_s$

Case b.1.2: $key_s \leq d$ (Figure A3).

Let's consider Case b.1.1 where we have:

i. $d<key_s$

Also since key_s is the separation point, then:

ii. $key_s<key(p)$

(i), (ii) result in $d<key(p)$ which contradicts (3).

Now, consider Case b.1.2: $keys \leq d$,

Note that using the assumption that we could not find p during the search of d and the definition of key_k and key_j, neither of them should be seen in the search path until reaching key_s. Because if either of them is seen before reaching key_s, this means that these two keys are the same and the search of d will find p in its path which is a contradiction. This also means $key_s \neq key_j$.

Again, since key_s is the separation point and also $d \geq key_s$, we conclude $key_s \leq key_j$.

Using $key_s \neq key_j$ results in:

iii. $key_s < key_j$.

Since key_k should select a separate path:

iv. $key_k < key_s$.

(iii) and (iv) are shown in Figure A3.

(iii), (iv) and (2) result in:

v. $key_p < key_s < key_j$

Now, two cases may occur:

either $len(s)<len(p)$ or $len(s) \geq len(p)$.

Consider the first case: $len(s)<len(p)$. In this case:

I. $p(0:len(s)-1)<s(0:len(s)-1)$

Based on this and since $p \rightarrow j$, we can conclude:

II. $j(0:len(s)-1)<s(0:len(s)-1)$

This means that $key_j<key_s$ and this contradicts (v).

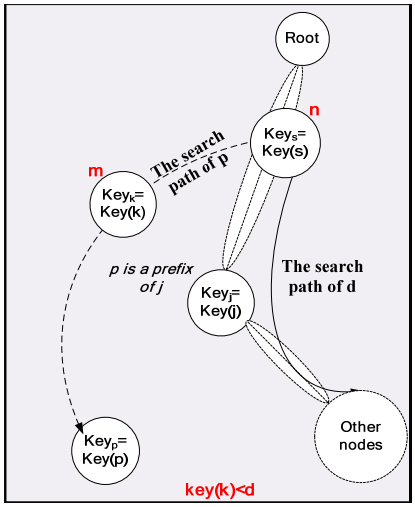

Figure A3 Proof of lemma 2

The second case: $len(s) \geq len(p)$, results in:

III. $p(0:len(p)-1)<s(0:len(p)-1)$

Then, $p \rightarrow j$ and III result in:

IV. $j(0:len(p)-1)<s(0:len(p)-1)$

This means: $key_j<key_s$ which contradicts (v).

For the case b.2, when in the search path of d in Figure A3 no key_j such that $p \rightarrow MP(key_j)$ is found, again let's consider node n and its key named key_s as the separation point. Similar to case b.1:

(7) $key_p<key_s<d$

Again two cases may exist:

Case b.2.1: $len(p) \leq len(s)$

Case b.2.2: $len(p)>len(s)$

Case b.2.1 results in:

vi. $p(0:len(p)-1)<s(0:len(p)-1)$.

Therefore:

vii. $d(0:len(p)-1)<s(0:len(p)-1)$ which contradicts (7).

Case b.2.2 results in:

viii. $p(0:len(s)-1)<s(0:len(s)-1)$.

Therefore:

ix. $d(0:len(s)-1)<s(0:len(s)-1)$

Therefore:

x. $d<key_s$ which again contradicts (7).

Therefore, at least one key_j is found such that $p \rightarrow MP(key_j)$ in the search path of d.

This proof guarantees that it is only needed to update $key_k.mv(len(p)-1)$ to '1' at the time of insertion.

Lemma 4- In Scalar Prefix Search, any time the search for address d reaches a key k whose 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.

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 $p' \rightarrow d$, $p \rightarrow p'$, then these 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 $p \rightarrow k'$.

Based on the properties of Scalar Prefix Trees mentioned in section 4.3, the Max-length prefixes of all of the keys must be disjoint. Therefore, the above relations contradict this property, and the search procedure is terminated in the node containing k.

REFERENCES

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, 'Introduction to algorithms', MIT Press, 1990.

[2] D.R. Morrison, 'PATRICIA Practical algorithm to retrieve information coded in alphanumeric', Journal of the ACM, Vol.15,no.14, (October 1968), pp.514-34.

[3] S. Nilson, G. Karlsson, 'IP address lookup using LC-tries', IEEE JSAC, Vol.17, (June.1999), pp.1083-1092.

[4] V. Srinivasan, G. Varghese, 'Faster IP lookups using controlled prefix expansion', ACM Transactions on Computer Systems, vol. 17, no. 1, (February 1999), pp.1-40.

[5] L.C. Wnn, K.M. Chen, T.J. Liu, 'A longest prefix first search tree for IP lookup', Proc. ICC'05, May 16-20, 2005, pp.989 – 993.

[6] P. Gupta, S. Lin, N. McKeown, 'Routing lookups in hardware at memory access speeds', Proc. IEEE INFOCOM, 1998.

[7] W. Eatherton, G. Varghese, Z. Dittia, 'Tree bitmap: hardware/software IP lookups with incremental updates', Proc. ACM SIGCOMM,, 2004, pp. 97-122.

[8] Le, H., Jiang, W. and Prasanna, V.K., 'A SRAM-based architecture for Trie-based IP lookup using FPGA', Proc. 16th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '08), 2008.

[9] Fadishei, H., Saeedi, M., and Zamani, M. S., 'A fast IP routing lookup architecture for multi-gigabit switching routers based on reconfigurable systems', *Microprocessors and Microsystems*, vol. 32, no. 4. pp.223-233, 2008.

[10] Y-H. E. Yang, O. Erdem, V.K. Prasanna, 'Scalable Architecture for 135 GBPS IPv6 Lookup on FPGA', FPGA '12 Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays, 2012

[11] Jiang, W. and Prasanna, V.K., 'Sequence-preserving parallel IP lookup using multiple SRAM-based pipelines', *Journal of Parallel and Distributed Computing*, vol. 69, no. 9. pp.778-789, 2009.

[12] J.Zhao, X. Zhang, X. Wang, Y. Deng, X. Fu, 'Exploiting Graphics Processors for High-performance IP Lookup in Software Routers', Proc.IEEE INFOCOM 2011.

[13] O. Erdem, C.F.Bazlamacci, 'High Performance IP Lookup Engine with Compact Clustered Trie Search', The computer Journal, Section B, February 2012.

[14] B. Lampson, V. Srinivasan, G. Varghese, 'IP lookups using multiway and multicolumn search', IEEE/ACM Transactions on Networking, Volume 7, Issue 3, (June 1999).

[15] S. Suri, G. Varghese, P.R. Warkhede, 'Multiway range trees: scalable IP lookup with fast updates', Computer Networks, vol. 44, no. 3, (2004), 289-303.

[16] H. Lu, S. Sahni, 'A B-Tree Dynamic Router-Table Design', IEEE Transactions on Computers, Vol.54, (2005), 813-823.

[17] Q. Sun, X. Zhao, X. Huang, W. Jiang, 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.

[18] Y.K. Chang, Y.C. Lin, 'Dynamic segment trees for ranges and prefixes', *IEEE Transactions on Computers*, Vol 56, No. 6, pp. 769-784, 2007.

[19] H. Lim, H. Kim, and C. Yim, 'IP Address Lookup for Internet Routers Using Balanced Binary Search with Prefix Vector', *IEEE Trans. Commun.,* vol. 57, no. 3, Mar. 2009, pp. 618-621.

[20] N. Yazdani, P. Min, 'Prefix Trees: New Efficient Data Structures for Matching Strings of Different Lengths', Proc. International Database Engineering and Applications Symposium, Grenoble, France , Jul. 2001.

[21] F. Pong, 'Concise Lookup Tables for IPv4 and IPv6 Longest Prefix Matching in Scalable Routers', IEEE/ACM Transactions on Networking, Vol. 20, Issue 3, pp. 729-741, June 2012.

[22] R. Govindaraj, I. Sengupta, S. Chattopadhyay, 'An Efficient Technique for Longest Prefix Matching in Network Routers', Springer LNCS, Vol. 7373, pp. 317-326, July 2012.

[23] H. Lim, K. Lim, N. Lee, Kyeong-hye Park, 'On Adding Bloom Filters to Longest Prefix Matching Algorithms', IEEE Transactions on Computers, 08 Aug. 2012

[24] M. Behdadfar, H. Saidi, 'The CPBT: A Method for Searching the Prefixes Using Coded Prefixes in B-Tree', Proc. IFIP Networking 2008, pp. 562-573.

[25] M. Behdadfar, H. Saidi, H. Alaei, B. Samari, 'Scalar Prefix Search: A New Route Lookup Algorithm for Next Generation Internet', Proc. IEEE INFOCOM 2009.

[26] M. Behdadfar, H. Saidi, M.R. Hashemi, A. Ghiasian and H. Alaei, 'IP Lookup Using the Novel Idea of Scalar Prefix Search with Fast Table Updates', IEICE Trans. INF. And SYST., VOL. E93-D, NO11, Nov. 2010, pp. 2932-2943.

[27] M. Behdadfar, H. Saidi, M.R. Hashemi, Y-D. Lin, 'Coded and Scalar Prefix Trees: Prefix Matching Using the Novel Idea of Double Relation Chains', ETRI Journal, Vol. 33, No. 3, June 2011, pp. 344-354.

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

Mohammad Behdadfar was born in Iran in September, 1977. He received his B.Sc., M.Sc. and Ph.D. degrees in 1999, 2002 and 2010 respectively from Isfahan University of Technology all in Electrical & Computer Engineering. He is currently an Assistant Prof. in the Engineering Department of IRIB University. His current research interests are in the area of high-speed networking, video networking, switch/router design and algorithms.

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. degree 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 has been with Isfahan University of Technology. His current research interests include switch architecture, data centric networking, smart grid, and sensor networks.

Hossein Saidi received B.Sc. and M.Sc. degrees in Electrical Eng. in 1986 and 1989 respectively, both from Isfahan University of Technology (IUT), Isfahan Iran. He also received his D.Sc. in Electrical Eng. from Washington University in St. Louis, USA in 1994.

Since 1995 he has been with the Dept. of Electrical & Computer Engineering at IUT, where he is currently an Associate Prof. of Electrical & Computer Engineering.

His research interest includes high speed switches and routers, communication networks, QoS in networks, queuing system, security and information theory.