Coded and Scalar Prefix Trees: Prefix Matching Using the Novel Idea of Double Relation Chains

Mohammad Behdadfar, Hossein Saidi, Massoud Reza Hashemi, and Ying-Dar Lin

In this paper, a model is introduced named double relation chains (DRC) based on ordered sets. It is proved that using DRC and special relationships among the members of an alphabet, vectors of this alphabet can be stored and searched in a tree. This idea is general; however, one special application of DRC is the longest prefix matching (LPM) problem in an IP network. Applying the idea of DRC to the LPM problem makes the prefixes comparable like numbers using a pair of *w*-bit vectors to store at least one and at most *w* prefixes, where *w* is the IP address length. This leads to good compression performance. Based on this, two recently introduced structures called coded prefix trees and scalar prefix trees are shown to be specific applications of DRC. They are implementable on balanced trees which cause the node access complexity for prefix search and update procedures to be *O*(log *n*) where *n* is the number of prefixes. As another advantage, the number of node accesses for these procedures does not depend on *w*. Additionally, they need fewer number of node accesses compared to recent range-based solutions. These structures are applicable on both IPv4 and IPv6, and can be implemented in software or hardware.

Keywords: Totally ordered set, prefix, LPM, LMP, DRC, coded prefix, scalar prefix.

Manuscript received June 30, 2010; revised Jan. 4, 2011; accepted Jan. 19, 2011.

This work was supported by Iran Telecom Research Center (ITRC).

Mohammad Behdadfar (phone: +98 913 317 8691, email: behdadfar@ec.iut.ac.ir) is with the Electrical and Computer Engineering, Isfahan University of Technology, Isfahan, Iran, and also with the Broadcast Engineering Department, IRIB University, Iran.

Hossein Saidi (email: hsaidi@cc.iut.ac.ir) and Massoud Reza Hashemi (email: hashemim@cc.iut.ac.ir) are with the Electrical and Computer Engineering, Isfahan University of Technology, Isfahan, Iran.

Ying-Dar Lin (email: ydlin@cs.nctu.edu.tw) is with the Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan.

doi:10.4218/etrij.11.0110.0381

I. Introduction

Longest prefix matching (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* bits stored in a router's routing table. A straightforward algorithm to find the longest matching prefix (LMP) of a given address is a linear search [1]. If *n* is the number of the 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*), where *w* is 32 for IPv4 and 128 for IPv6. This is better than a linear search, but the search and update complexities of Trie are not acceptable for high-speed switches. Some other versions of Trie, such as PATRICIA [2], level compressed Trie (LC-Trie) [3], prefix expansion [4], and longest prefix first search table (LPFST) [5]-[7] have also been introduced. However, some of them suffer from similar drawbacks in that they do not support incremental updates, and others are not extendable to IPv6.

In later research, some range-based algorithms were introduced which define two end points for each prefix and store these end points in a tree [8]. By defining a range for each prefix, the search in the tree is carried out to find the most specific range corresponding to an address [8]. The node access complexity of the search procedure in this structure is independent from *w*. Range-based prefix search algorithms need a long time for prefix updates, even for those algorithms which use balanced trees to store the prefix end-points [9]. To support fast search and incremental updates, some range-based algorithms were introduced by the authors of [10] among which prefixes in B-tree (PIBT) has the best search performance [10]. Since, PIBT stores the prefix end-points in a

B-tree, its search and update complexities are $O(\log n)$. It should be mentioned that PIBT uses about 6 w-bit vectors excluding the tree child pointers for each prefix. One of the recent range-based lookup algorithms with comparable average results to PIBT is balance tree with LPFST (BTLPT) [11]. This algorithm uses two structures: a B-tree and another LPFST. However, the complexity of BTLPT still depends on w because of its Trie-based part, LPFST.

Prefix trees and M-way prefix trees [12] are other algorithms which use a new comparison rule for storing and searching the prefixes resulting in a tree with the worst case tree height of $O(w)$ and long update procedures.

We introduced new algorithms called coded prefix B-tree [13] and scalar prefix trees [14] and later, a complete version of [14] with more details and simulation results [15]. These schemes consider a coding concept for prefixes to make them comparable to numbers with =, <, and > and to store them in tree structures and particularly in balanced trees like B-tree. Also, we proved that the proposed algorithms do not modify the number of node accesses of the original search and update procedures of the trees. This makes the number of node accesses of the search and update procedures independent from the IP address length and leads to $O(\log n)$ tree operations which will be efficient for both software and hardware implementations. Lim and others demonstrated a similar search procedure to [13], [14] which also uses a balanced tree [16]. However, the main advantage of our algorithm is the ability of simple incremental updates with limited worst case time compared to the hard and unpredictable time of updates in [16]. Also, in the proposed data structure of scalar prefix trees, unlike the method of [16], each prefix exists only in one place of the tree. This makes the insertions and deletions easier. Another advantage of the proposed data structures compared to [16] is that they can be applied on many types of trees, especially balanced trees like B-tree, RB-tree, and AVL-tree, without affecting the $O(\log n)$ order of number of tree node accesses in prefix insertion and deletion procedures.

This paper introduces a mathematical model for the main idea of [13]-[15] using the concept of totally ordered sets [17] and extends it to a wide range of search trees. In this regard, the novel concept of double relation chains (DRC) is proposed to be used for modeling the scheme. The result can be extended to all versions of the algorithm which use B-tree, RB-tree, and AVL-tree. Finally, the results are compared with some competitive algorithms like PIBT, BTLPT, and LPFST for both IP versions 4 and 6.

The structure of the paper is as follows. The idea of DRC and its properties are introduced in section II. Also, it is shown that the LPM problem is a special application of DRC. Section III proposes coded prefix and scalar prefix tree structures derived from the concept of DRC. Comparison results and the conclusion are presented in sections IV and V, respectively.

II. Introduction of DRC Using the Concept of Totally Ordered Sets

Before introducing DRC, we define a prefix comparison rule. First, consider the following inequality in which β is the representative of blank space: $\beta \le 0 \le 1$.

Using this inequality, variable length prefixes can be compared. For example, consider the following prefixes: p_1=010000*, p_2=0100011, p_3=01000*, p_4=0100*, p_5=010*, and p_6=00*.

Using the mentioned inequality, these prefixes can be sorted in the following order: $p_6 \le p_5 \le p_4 \le p_3 \le p_1 \le p_2$.

Therefore, they can be stored in any search tree. To explain our idea, we consider the binary search tree (BST). Assume that the above prefixes should be inserted into a BST with this order: $p_1, p_2, p_3, p_4, p_5, p_6$.

Figure 1(a) shows the result of the insertion process. Since the prefixes can be stored in a search tree, it is enough to run the tree search algorithm to find a specific prefix. However, there is an issue as to whether the structure is capable of searching the LMP of an address.

To resolve this concern, consider the incoming address d=0100010. Searching d in the tree of Fig. 1(a) traverses nodes A and C. Obviously, none of the prefixes stored in these two nodes is the prefix of d. Therefore, the ordinary search algorithm cannot find the LMP of an address. All of the prefixes of d, 010*, 0100*, and 01000*, are located in the left subtree of node A, but the search algorithm traverses the right subtree. However, an interesting hidden property exists in this tree which makes it capable of finding the LMP by storing additional information in each node. This property is as follows.

Node A which contains p_1 is the separation point of the

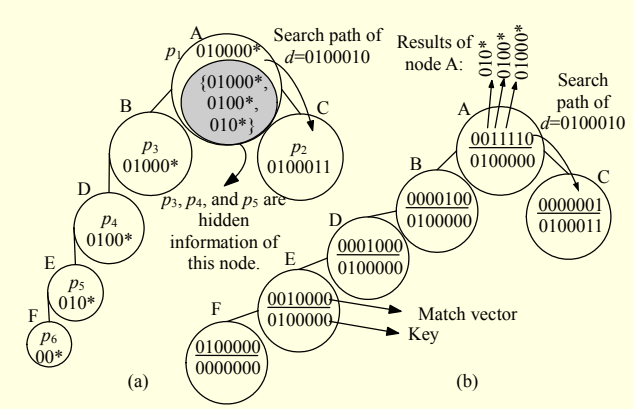

Fig. 1. (a) Example of inserting prefixes in a BST using proposed comparison rule and (b) CP-BST.

search path of d and the search path of some of the prefixes of d. However, each of these prefixes (p_3, p_4, and p_5) is a prefix of p_1, which is stored in node A.

In this paper, we will prove that each prefix of d which is missing in the search path will also be a prefix of at least one key in the search path of d. One of these keys is the one which is stored in the separation point. Therefore, this key contains some additional hidden information about all of the missed prefixes of d. This information makes the search procedure capable of finding all of the prefixes of d in the tree. Using this proof, the coded prefix trees and scalar prefix trees will be introduced. The proof is done using a novel concept named DRC. As it will be discussed, the prefixes will be modeled as vectors of the alphabet of 0, 1, and β.

Before introducing coded prefixes and scalar prefixes, some properties will be introduced using the concept of totally ordered sets [17]. Consider a set P such that

$$P=\{p_1, p_2, p_3, \ldots, p_n\}. \quad (1)$$

Assume that P is totally ordered under the relation R_t. Also, assume that a partial order [17] relation R_p over P exists which has the following relation with R_t (a and b are members of P):

$$aR_pb \Rightarrow aR_tb. \quad (2)$$

Also, assume that the set P is the union of k subsets in equality (3) with the condition (4):

$$P = \bigcup_{i=1}^{i=k} P_i, \quad (3)$$

$$P_i \bigcap P_j = \varnothing, \quad \forall i, j;\ i \neq j. \quad (4)$$

Also, consider the number of members of P_i, namely, $n(P_i)$ with the following property:

$$n(P_i) \leq w;\ 1 \leq i \leq k. \quad (5)$$

Finally, assume that each P_i is a totally ordered set (chain) [17] under the relation R_p. This also implies that each member of P has the relation R_p with itself. Note that the subsets P_i which meet (3), (4), and (5) are not unique. Using (1) and (2), a simple property can be retrieved, and since we use it in our next sections, we state it as a lemma. Also, note that (3), (4), and (5) will be used in section IV.

Lemma 1. Consider the definitions of P, R_t, and R_p which were stated above and a, b, and c as members of P. Assume that aR_tb and bR_tc. Also assume that aR_pc. Using these assumptions, the conclusion aR_pb will not always be true.

Proof. This is proved using a contradictory example. Assume that R_t is the relation $\leq$ and R_p is the relation $|$, that is, the relation of counting, for example, 2|8 means that 2 counts 8. Consider $\mathbb{N}$ as the reference set. These relations ($\leq$ and $|$) meet (2). Therefore, the required condition for R_t and R_p is met. As a contradictory example, assume that a=2, b=3, and c=4. In this case, 2|4, that is, 4 mod 2 = 0. However, 2|3 is false. □

In this stage, we need to define DRC.

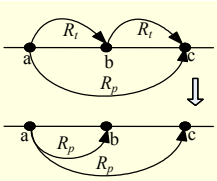

Fig. 2. Condition of DRC.

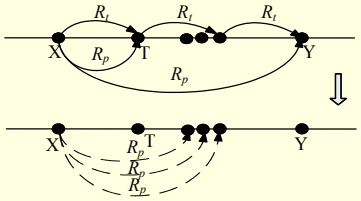

Fig. 3. Illustration of property 1.

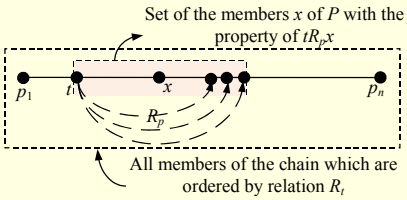

Fig. 4. Illustration of property 2.

Definition 1. Consider the set P in (1). Assume that P is a chain under the relation R_t and (2) holds for relations R_t and R_p. Assume that (3), (4), and (5) hold for R_p. We call (P, R_t, R_p) (the set P under relations R_t and R_p) a DRC if the following condition is met for $\forall a, b, c \in P$:

$$(aR_tb,\ bR_tc,\ aR_pc) \Rightarrow aR_pb\ . \quad (6)$$

Figure 2 illustrates (6). The relations are shown on a line and a, b, and c are some members of the chain constructed by (P, R_t). Also, a, b, and c are not required to be consecutive members. Therefore, any numbers of members are allowed to be between them.

1. Properties of DRCs

Based on the definition of DRC, the first property is trivial.

Property 1. Consider the set P and the relations R_t and R_p in a DRC as defined in definition 1. If the following conditions are met:

- $x, y \in P$, xR_py
- There are other m members of P between x and y in the chain constructed by P and the relation R_t. Then, x will have the relation R_p with all of the other m members.

Figure 3 is an illustration of property 1. The following property is also a result of property 1.

Property 2. Consider set P and the relations R_t and R_p in a DRC. Also, consider that all members of P which are totally ordered under the relation R_t are shown on a line, similar to a Hasse [17] diagram. Then, consider an arbitrary member t from the members of P. Using the result of property 1, it can be proved that all members, for example, x with the property of tR_px, are consecutive to each other on a segment of the line (Fig. 4). The proof of this property is trivial.

Now, consider set A with N+1 members:

$$A=\{\alpha_0, \alpha_1, \alpha_2, ..., \alpha_N\}. \quad (7)$$

Assume that $\alpha_0=\beta$, and A is a totally ordered set under a relation shown by $\prec$ and with the order of (7). The ordered vector V is defined as

$$V=(v_1, v_2, ..., v_w),\ \forall i,\ v_i \in A. \quad (8)$$

Consider two vectors V and V' using the definition of (8). The relation R_t is defined recursively as

$$V\ R_t\ V' \Leftrightarrow ((v_1 \prec v_1') \text{ or } ((v_1 = v_1') \text{ and } (v_2, ..., v_w) R_t (v_2', ..., v_w'))). \quad (9)$$

Also, the relation R_p is defined as

$$V\ R_p\ V' \Leftrightarrow \forall i,\ ((v_i = v_i') \text{ or } (v_i = \beta)). \quad (10)$$

In this case, it can be verified easily that R_t and R_p have the following relation which is exactly the same as (2).

$$V\ R_p V' \Rightarrow V\ R_t\ V. \quad (11)$$

Now, consider a set P which is defined as

$$P=\{V \mid V=(v_1, v_2, ..., v_w), \forall i, v_i \in A\}. \quad (12)$$

It can be easily verified that both (P, R_t, R_p) and (P', R_t, R_p) are DRC where P' is an arbitrary subset of P (this can be proved by considering the definition of DRC). Since P' is totally ordered under the relation R_t, any input vector can be searched inside it using ordinary search schemes. Now, consider vector d such that $d=(d_1, d_2, d_3, ..., d_w)$, $\forall i; d_i \in A$.

Since the set P and every subset P' of P are totally ordered sets under the relation R_t, adding d to the set P' will not modify this property. Therefore, d has a fixed place on the line of members of P' (Fig. 5). Now, if we assume that d is inserted in the line as it is shown in Fig. 5, and if we define the set M as

$$M=\{p_i \in P'; p_i R_p d\}, \quad (13)$$

then, it can be proved easily that M is a partially ordered set [17] (POSET) under the relation R_p. The minimal and maximal members of M might be indicated as p_m and p_M. Bearing in mind that d is not really inserted in this line, Fig. 5 shows its location virtually. Based on Fig. 5, assuming p_j is the immediate member before d, then

$$p_M R_p d,\ p_M R_t p_j,\ p_j R_t d. \quad (14)$$

Based on property 2 and (14), it can be verified that for every

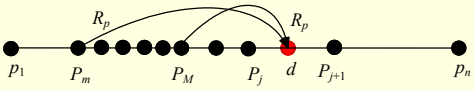

Fig. 5. Virtual insertion of vector on line.

p_i (including p_j) between p_M and d on the line of Fig. 5, the following relation exists:

$$p_M R_p p_i. \quad (15)$$

2. Main Objective

In the following sections, it will be proved that to propose an LPM algorithm based on the above considerations, it is necessary to find P_M, the maximal member of the set M defined in (13), for a given d. First, we assume that we need to find the location of d on the line.

Lemma 2. To find the location of d on the line, using any search algorithm, if the search algorithm starts from any point on the line of Fig. 5 between p_1 and p_n, at least one p_i will be encountered that meets the conditions $p_i R_t d$ and $p_M R_p p_i$.

Proof. To find the location of d on the line, the search should finally meet both p_j and p_{j+1} which would be the immediate neighbors of d if it was inserted on the line. Since based on (15) $p_M R_p p_j$, at least one member, p_j, with the mentioned property will be seen in the search procedure. □

Based on the result of lemma 2, at least one p_i meeting the condition $p_M R_p p_i$ will be encountered in the search procedure. Since $p_M R_p p_i$, if some additional information could be stored with p_i, indicating the existence of members such as p_k with the property $p_k R_p p_i$, then it would not be necessary to find p_M itself. It would be sufficient to extract the additional information of the first p_i, seen in the search path with the property of $p_M R_p p_i$.

Since the vectors are related to each other by the relation R_t, it is possible to store the vectors in a tree-based structure to decrease the search time. To show how to store the vectors, consider subset P' of set P as it was described in section II.1. Since P' is totally ordered under relation R_t, R_t can be modeled by the relation $\leq$ for a set of numbers, for example, the set of natural numbers $\mathbb{N}$ or one of its subsets. Therefore, the members of P' can be stored in a tree structure. However, it should be ensured that searching on the tree does not miss the vectors p_i with the property of $p_i R_p d$. This directly depends on the insertion algorithm of the vectors with additional information and the following lemma.

Lemma 3. Consider a DRC (P', R_t, R_p). Assume that the members of P' are inserted into a binary search tree with an arbitrary order. Consider d as an input vector 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 $p_i R_p d$, then at least one vector q will

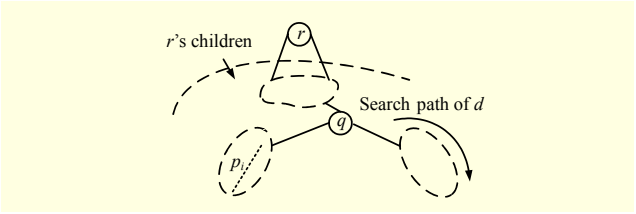

Fig. 6. Proof of lemma 3.

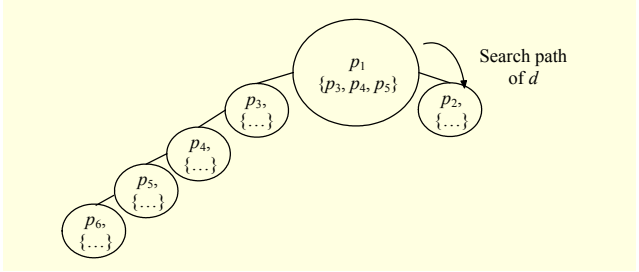

Fig. 7. Vectors inserted in tree structure.

be found in the search path of d with the property of p_iR_pq.

Proof. If p_i is in the search path of d, the lemma is proved. Assume that the search path of p_i is separated from the search path of d in a node containing a vector, for example, q. Also, the following relationships can be verified from Fig. 6 and the assumptions of the lemma: p_iR_tq, qR_td, p_iR_pd. Based on property 1 and the DRC (P', R_t, R_p), we can conclude p_iR_pq. Therefore, in the search path of d, at least one vector q will be found for which we have p_iR_pq. □

Therefore, to find the maximal member of M in (13), it is required to store additional information with each stored vector. The following example clarifies the problem.

Example 1. Consider an input vector d and the set of stored vectors P={p_1, p_2, p_3, p_4, p_5, p_6}. Also, consider the following relations between the members of P: (a_1) $p_6R_tp_5R_tp_4R_tp_3R_tp_1R_tp_2$ and (a_2) $p_5R_pp_4R_pp_3R_pp_1$.

Assume that the vectors will be inserted into the tree with the following order: (a_3) $p_1, p_2, p_3, p_4, p_5, p_6$.

Finally, assume that the objective is to find the maximal member of the set M defined in (13) under the relation R_p using the following additional considerations for d: (a_4) p_1R_td, p_3R_pd based on (a_2) and (a_4), in this example M={p_3, p_4, p_5}.

To solve the problem, the binary search tree of Fig. 7 is constructed based on the order of (a_3). The search path of d is shown in the same figure based on (a_1), (a_2), (a_3), and (a_4). Clearly, the maximal member of M is p_3, which is not in the search path. However, p_1 exists on the search path of d and based on (a_2), $p_3R_pp_1$. Also, based on (a_2) and according to the transitivity property of POSETs [17] under the relation R_p, we can say $p_3R_pp_1$, $p_4R_pp_1$ and $p_5R_pp_1$.

Therefore, the set of the vectors p_i with the property of $p_iR_pp_1$ which also meet p_iR_pd will be N={p_3, p_4, p_5}. Based on (a_4) and considering N, we can conclude that N is a subset of M (in this example, N=M). If the vectors p_i with the property of $p_iR_pp_1$ could be stored with p_1 as some additional information (Fig. 7), then none of the members of M would be missed in the search path of d. Finally, the answer to the problem, p_3, which is the maximal member of M, will be found in the root node of the tree. This result is the same as the result we saw in the similar example of prefixes of Fig. 1(a). As it will be clarified in the following section, LPM is a special application of DRC.

A. LPM: Special Application of DRC

In this stage, we will focus on a special application of DRC to the LPM problem. Consider set A as a special case of (7):

$$A = \{\beta, 0, 1\}, \tag{16}$$

where β is the blank space.

Also, consider P, a special case of (12) and its subset P':

$$P = \{V \,|\, V = (v_1, v_2, ..., v_w), \forall i, v_i \in A\}, \tag{17}$$

$$P' = \{V \,|\, V = P, ((v_j = \beta), \Rightarrow (\forall k, j < k \le w, v_k = \beta))\}. \tag{18}$$

From the definition of P', we will show that it is a set of binary prefixes of different lengths. Also, according to the relation $\prec$ which is defined for (7), the same relation would be true for (16). Therefore, it can be concluded that

$$\beta \prec 0 \prec 1. \tag{19}$$

Now, based on (9) and (19), it can be simply verified that the relation R_t is the comparison of w-bit strings with the alphabet of (16). Also, based on (10), (16), and (18), VR_pV' means that V is a prefix of V'. Based on these discussions, a novel LPM algorithm will be introduced and its performance will be evaluated in the following sections.

III. Coded and Scalar Prefix Search: Applications of DRCs

In the previous section, we showed that using the alphabet of (16), the set P' in (18), and also the relations R_t in (9) and R_p in (10), (P', R_t, R_p) will be a DRC describing the prefixes with different lengths and their relations. For this application, we use some new notations for R_t and R_p as follows: The symbol $\le$ will be used for relation R_t. The symbol $\rightarrow$ will be used for relation R_p. As an example, consider w=5. To represent a member of P', such as V= (v_1, v_2, v_3, v_4, v_5) = (0, 0, 1, β, β), we can show it with 001$\beta\beta$ and simplify it by replacing the $\beta\beta$ at the end with * to show this vector with 001*. Now, let us clarify the concept with the following examples.

Example 2. Consider P' as the set of the following prefix

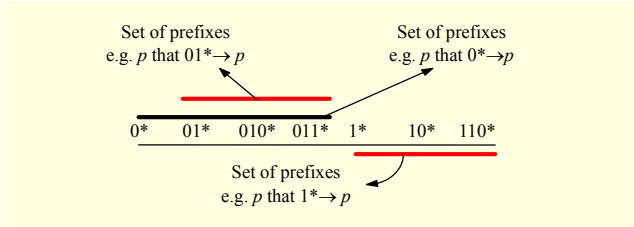

Fig. 8. Example of property 2 for prefix vectors.

vectors: P'={0*, 01*, 011*, 010*, 10*, 110*, 1*}.

Since (P', R_t) (that is, (P', ≤)) is a totally ordered set, we can write the members of P' ordered: 0* ≤ 01* ≤ 010* ≤ 011* ≤ 1* ≤ 10* ≤ 110*.

As examples of members related by R_p, we can say 0*→01*, 01*→011*,….

Now, to check the correctness of property 2 for this DRC, assume that the ordered prefixes are shown on an ordered line (Fig. 8). Based on this figure, all the prefixes p such that 0^*R_pp (0*→p) are adjacent to each other on this line. Looking at the continuous bolded lines in Fig. 8, this fact also holds for 01* (01*→01*, 01*→010*, 01*→011*), 1* (1*→1*, 1*→10*, 1*→110*). This example completely matches with property 2.

Example 3. Mapping example 1 to this DRC, consider that (i) w=7 and d=0100010 and (ii) p_1=010000*, p_2=0100011, p_3=01000*, p_4=0100*, p_5=010*, and p_6=00*.

Also, let us assume that the objective is to find *LMP*(d) in the above set of prefixes. Checking this set, it is clear that *LMP*(d) is p_3. However, we try to find this answer using example 1. Finding the *LMP*(d) from the set of prefixes is equivalent to finding the maximal member of the set M in example 1. Therefore, the search process will be the same as example 1. Also, it is required to store the information of all prefixes of p_1 in the root node. Therefore, the information about the existence of the following set of prefixes should also be stored in the root node: {p_3, p_4, p_5}={01000*, 0100*, 010*}.

Replacing all p_i stored in the tree of Fig. 7 by the prefixes of this example and comparing d with the root node prefix 010000* results in searching its right hand child. This is the same as Fig. 1(a). While *LMP*(d)=p_3 is missed from the search path, this problem is solved by the additional information stored in the root of the tree which indicates the existence of p_3 as well. It means that storing the additional information in each node causes the search to find the correct answer in a single downward pass. For example, the additional information which is stored in node A of Fig. 1(a) is p_3, p_4, and p_5. The existence of these prefixes can also be indicated using a w-bit match vector to reduce the memory size. The resulting tree is called a coded prefix tree. In order to manage the available space, the details of insert, search, and delete functions of coded prefix trees are omitted. Instead, we focus on its improved version, scalar prefix tree, which will be explained in next subsections. However, it is worth mentioning that coded prefix trees are implementable on many types of tree structures, especially on balanced trees. We have implemented the scheme on B-tree (coded prefix B-tree: CP-BT), RB-tree (coded prefix RB-tree: CP-RB), and AVL-tree (coded prefix AVL-tree: CP-AVL) without any changes in the number of node accesses for ordinary search and update functions of the trees. The details of tree operations in CP-BT are discussed in [13].

1. Scalar Prefix Trees

Figure 1(a) shows the coded prefix tree. Although p_3, p_4, and p_5 are prefixes of d, the search function of d does not traverse nodes B, D, and E, which are the representatives of p_3, p_4, and p_5, respectively. However, the existence of these prefixes is also indicated in node A, which is located in the search path. Therefore, it suffices to store the existence of such prefixes in node A only, and nodes B, D, and E can be removed from the tree. This causes the tree to become more compressed and faster in search and update procedures. Although the algorithm and its structure were given by an example, we have proven its correctness in [14]. This new kind of storing of prefixes is the basis of scalar prefix trees. To introduce scalar prefix trees, first, let us define some notations:

- *len*(p) shows the length of a prefix p.
- $p(i)$ shows the i-th bit of prefix 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)=$p(0)p(1)p(2)…p(k–1)000…0$. For example, 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 $pref_k(p)$.
- For a key r, a w-bit match vector is defined and abbreviated with *r.mv* for both coded prefix and scalar prefix trees. This vector stores the additional information of each prefix which was mentioned above. 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, *len*(q)=i+1, or q=$pref_{i+1}(r)$ in the database. Please note that indexing the match vector bit numbers starts from 0. For an example of the match vector, consider the set of prefixes in the root node of Fig. 1(a). The pair (match vector, key) of this node can be stored like its corresponding node in Fig. 1(b): (0011110, 0100000). In mv=0011110, mv(2)=1 is a representative for p_5=010* and mv(3)=1 is a representative for p_3=01000*.

- The longest prefix of each key indicated by 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. Again consider (match vector, key) the pair of the root node of Fig. 1(b) as mentioned above. *mv*(5)=1 is a representative for p_1=010000* as the max-length prefix of 0100000.
- The length of the path from the root of the tree to a node *x* is called *height*(*x*). For example, *height*(*root*) is zero, and the height of each child of the root is one and so on.

Consider Fig. 6 and the proof of lemma 3. Assume that the prefixes of the set *P′* 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 member of *P′* stored in the tree and $p_i \rightarrow d$, based on the proof of lemma 3 and Fig. 6, if the search path of p_i is separated from the search path of *d* in a node containing a vector, for example, *q*, the relation $p_i \rightarrow q$ will always be true. Based on this property, we introduced coded prefix trees at the start of section III. 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 subtree of *q* as it is also depicted in Fig. 6, can be removed from the tree, because its information is redundant compared to *q*.

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, that is, each node's key and its match vector are representatives of a set of prefixes that do not exist in any other node.

Considering *P* as the set of all prefixes and P_i as the set of prefixes of key_i stored in the tree based on its match vector, the following union will be true. This means that each key and its match vector correspond to one set P_i. Note that

$$P = \bigcup_{i=1}^{i=k} P_i. \tag{20}$$

Since the prefixes which are stored in the match vector of each key do not exist in the match vector of any other key, it will be concluded that

$$P_i \bigcap P_j = \varnothing, \ \forall i, j; \ i \neq j. \tag{21}$$

Also, each P_i contains at most *w* prefixes. It results in

$$n(P_i) \leq w. \tag{22}$$

Considering (20), (21), and (22), it can be easily verified that these equations are the same as (3), (4), and (5) in section II with the same meanings.

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 a binary search tree and call it scalar prefix binary search tree (SP-BST). For the details of its application to the two versions of B-tree (SP-BT and SP-BTe), RB-tree (SP-RB), and AVL-tree (SP-AVL), and also the major modifications in the search and update procedures of these trees, refer to [14].

A. Insert Procedure for SP-BST

The insertion procedure for SP-BST is explained in [14] and [15] in detail. What follows is a brief review.

Insertion of each prefix in the tree is similar to the insertion procedure in a binary search tree. However, during the insertion of each prefix *p*, one of the following cases may occur in each node *r* of the insertion path which contains a key key_r:

If $p \rightarrow MP(key_r)$, then $key_r.mv(len(p)-1) = 1$.
Else if $MP(key_r) \rightarrow p$, then:
$key_r.mv(len(p)-1)=1$ and $key_r = key(p)$.
Else
The algorithm is repeated till it is terminated or reaches a leaf node.

Further details of the insertion algorithm are in [14] and [15].

For an example of the insertion process, consider the prefixes of example 3 with the same order of arrivals. Figure 9 shows the tree after the insertion of the above prefixes. Comparing Fig. 9 with Fig. 7 (or Fig. 1(b)), 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 can be found in [14] and [15].

B. Search Procedure for SP-BST

The search procedure for the *LMP*(*d*) is started from the root and may be finished in a leaf or non-leaf node. First, consider a *w*-bit match vector for *d*, called *d.mv* and assume that the search is being done in a node *r* containing a key named key_r.

Although the search procedure is explained in [15], it is summarized here by a simplified pseudocode.

If $MP(key_r) \rightarrow d$, then: $LMP(d)=MP(key_r)$.
Else if some other prefixes of key_r match with *d*, the corresponding bits in *d.mv* will be set to one.
Then, the procedure goes to right or left child of the current node, based on the result of comparing *d* and key_r.

For example, assume that the objective is to find *LMP*(*d*) in Fig. 9 considering *d*=0100010. The search starts from the root node in which Key_r=0100000 is stored and $MP(Key_r)! \rightarrow d$. However, since some other prefixes of Key_r match with *d*, their corresponding bits in *d.mv* will be set to one. Therefore,

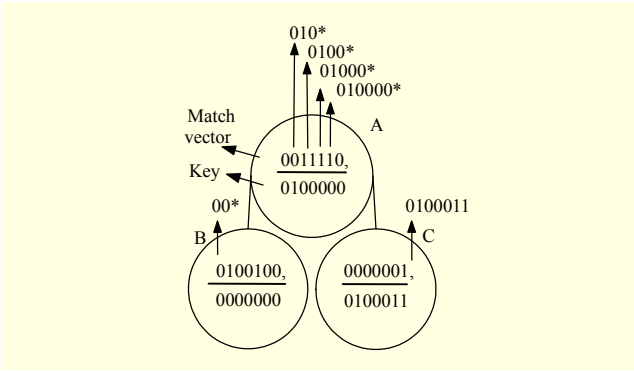

Fig. 9. SP-BST for prefixes of example 3.

$d.mv$=0011100. Then, since $d > Key_r$, the right child should be checked. It does not contain any new matching prefix of d. The procedure terminates after checking the match vector and key in node C and $LMP(d)$=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). This guarantees that $LMP(d)=MP(Key_r)=p_1$=010000*. In this case, it is not needed to continue traversing the tree.

C. Properties of Scalar Prefix Trees

Based on the search and insert procedures, SP-BST has some properties listed below. The proofs of parts (i), (iv), and (v) are omitted because of space limitations.

(i) **Lemma 4.** The max-length prefixes of all node keys in the tree are disjoint.

For example, in Fig. 9, the disjoint max-length prefixes of the nodes are 010000*, 0100011, and 00*.

(ii) **Lemma 5**. In a 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. The proof is explained in the appendix.

(iii) A prefix is stored in the match vector of only one key in the tree. This is the direct result of the insertion algorithms and the relations (20), (21), and (22) (or (3), (4), and (5) of section II).

(iv) **Lemma 6.** To store a new prefix p in SP-BST, let us assume that $K=\{k_1, k_2, k_3, \ldots, k_n\}$ is the set of all keys in the tree which p is a prefix of them. If among the members of K, $k_j \in K$ is the key with the least height node, 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*.

(v) **Lemma 7**. Again assume that $K=\{k_1, k_2, k_3, \ldots, k_n\}$ is the set of all keys stored in the tree in 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 cover the node containing k_j.

(vi) Consider the same definitions for prefix p, the set K and the key k_j in properties (iv) and (v). Also, consider an address d such that $p \rightarrow d$. Based on lemma 6, k_j is the first member of K which is seen in the insertion path of p. Also, based on the lemma 7, it is the first member of K which is seen in the search path of d. Therefore, in order to store p in the tree, its existence should be indicated in the match vector of k_j. On the other hand, the search procedure of d will reach k_j in the search path before any other member of K and $k_j.mv(len(p)-1)$ will indicate if p is stored in the tree or not. This means that 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 for this naming is that with respect to p, k_j and its submatch 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 it is always true that $n_k \le n_p$. The equality holds only when all of the prefixes are disjoint. This causes the tree to become more compressed if the percentage of non-disjoint prefixes increases. The reason is that disjoint prefixes will be stored in the tree as the main prefix of a single key. If the percentage of non-disjoint prefixes increases, most of the prefixes will be stored in the match vectors of the disjoint prefixes. Therefore, they do not need additional storage. This can make the tree compressed. Current IPv4 prefix databases contain less than 10% of non-disjoint prefixes.

The SP-BST has many advantages compared to Trie-based and range-based algorithms. A 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 the match vector of one key and also this tree does not need to store both of the end points, the average storage would be reduced as well.

Since there is no guarantee for SP-BST height, the concept of scalar prefix search has been applied to some balanced trees, such as B-tree (SP-BT), RB-tree (SP-RB), and AVL-tree (SP-AVL), without any changes in their original number of node accesses in the search and update procedures. They have the ability to guarantee and control the worst case height of the tree to be $O(\log n)$. Therefore, the search and update complexities

for these trees are $O(\log n)$ as well. We have explained the details of the search and update procedures of SP-BT in [14].

IV. Comparison Results

Different versions of the proposed algorithms were implemented for IPv4 and IPv6 databases in software, and their results were presented in [14] and [15]. In this paper, some complementary results are included based on comparing the proposed B-tree schemes with competing solutions like PIBT and BTLPT.

Three IPv4 and two IPv6 databases of different sizes are used for the simulations. The IPv4 prefix databases are AS4637 (139519 prefixes, August 2008), AS1221 (191566 prefixes, August 2008), and AS131072 (313453 prefixes, January 2010) which were downloaded from [18]. The IPv6 prefix databases [18] are AS1221 (933 prefixes, August 2008) and AS131072 (2523 prefixes, January 2010).

In [15], the search and update results were presented in terms of the average number of node accesses to ensure that the results are independent from the CPU model, cache size, or other restricting issues and also to give a proper indication of the hardware implementation efficiency.

In this paper, the results are presented in terms of the worst case prefix search and update results of the proposed B-tree schemes in comparison with PIBT and BTLPT. The results are obtained by repeating the test scenarios several times using random ordering for the members of the databases. Also, since the storage requirements are similar to the results of [15], they are not included in this paper.

To find the storage requirements, initially, all of the prefixes of each database were inserted. Then, to find the number of node accesses for each of the insertion and deletion procedures, each prefix was deleted and inserted again. For each time of doing the insertion or deletion procedure, its number of node accesses was recorded and the worst case numbers of node accesses for the insertion and deletion were finally extracted.

Although the results would not be different in the general case for different branching factors of the B-tree, the minimum degree of the B-tree is considered to be 14 in this paper for the proposed algorithms, PIBT and BTLPT.

The worst case number of node accesses for the search procedures of IPv4 and IPv6 databases are depicted in Fig. 10. As shown in Figs. 10(a) and (b), the required number of node accesses of the search procedure of SP-BT is the best for all three databases. The CP-BT also has comparable results.

Note that unlike the presented average case performances of these algorithms in [15], in the worst case, the search procedure of BTLPT is degraded by a big factor due to its dependency on

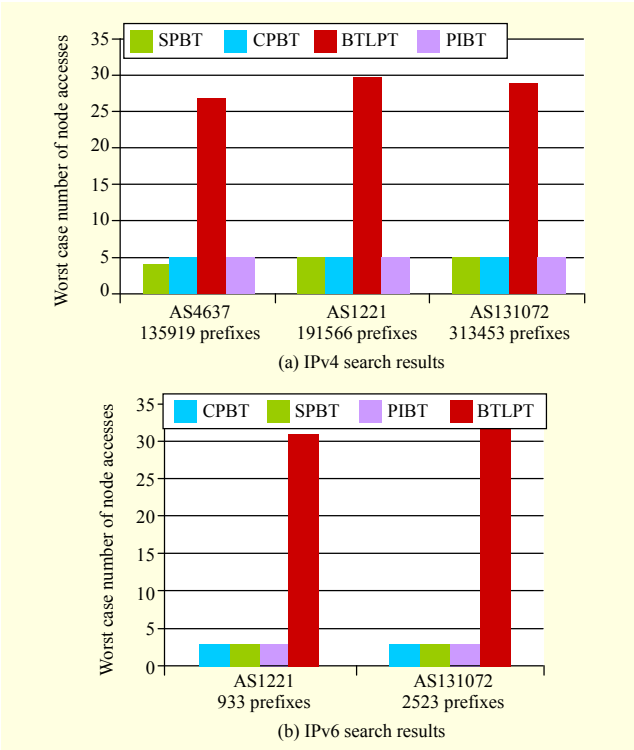

Fig. 10. Worst case node accesses results of search procedures for B-tree schemes (IPv4 and IPv6 databases).

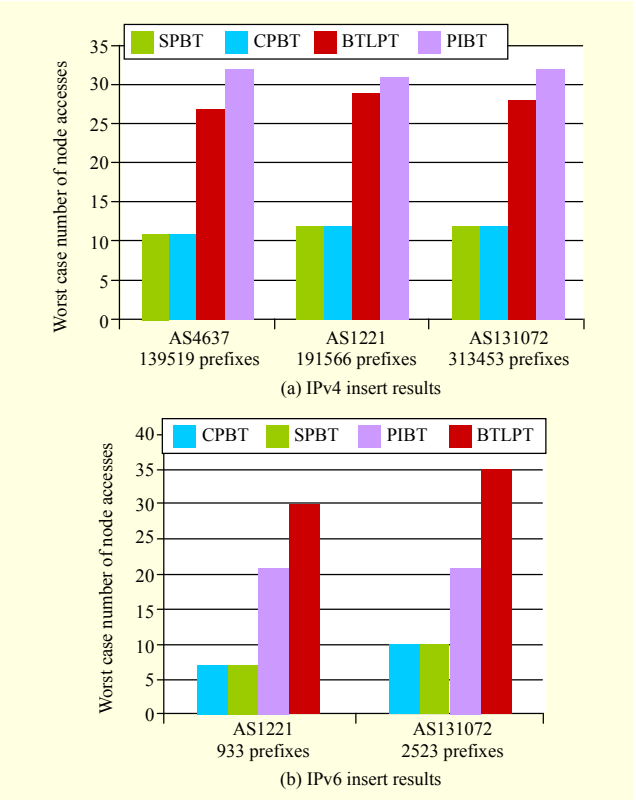

Fig. 11. Worst case node accesses results of insertion procedures for B-tree schemes (IPv4 and IPv6 databases).

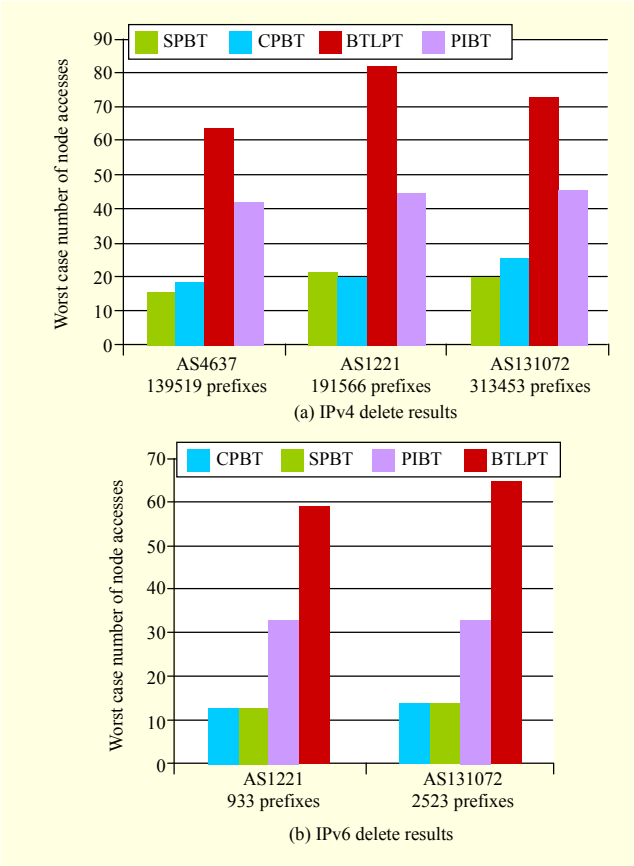

Fig. 12. Worst case node accesses results of deletion procedures for B-tree schemes (IPv4 and IPv6 databases).

Trie-based search of its LPFST part. A similar situation exists for the worst case insertion and deletion procedures of the PIBT which is depicted in Fig. 11(a) and Fig. 12(a) for the IPv4 database and Fig. 11(b) and Fig. 12(b) for the IPv6 database.

Although the search performance of the proposed schemes is very much similar to PIBT, their update performance is superior compared to PIBT and BTLPT. However, BTLPT requires the least storage as it was mentioned in [15]. Also note that, although BTLPT has the worst performance in most cases of Figs. 10, 11, and 12, it has a better average case update performance than PIBT [15].

V. Conclusion

In this paper, the novel idea of DRC was presented. Using this scheme, our recent work on the longest prefix matching algorithms was completed and modeled. Based on this idea, two sets of search trees named coded prefix and scalar prefix trees were shown to be examples of DRC. This will allow the treatment of prefixes as numbers and their storage in ordinary trees, which is a new approach compared to range-based and Trie-based solutions. This kind of treating prefixes makes the trees capable of fast search and incremental updates while the required storage has comparable results to other competitive solutions. Coded and scalar searches were implemented on B-tree, RB-tree, and AVL-tree as examples of balanced trees without any modification in the number of node accesses of their original versions. The implementation results of B-tree version of the proposed algorithms showed superior results, especially in update performance, for example, the number of node accesses for the proposed B-tree versions is about one fourth of the results of PIBT. Also, the versions which do not use the B-tree structure showed good results compared to LPFST, especially in the search of IPv6 databases in which the number of node accesses became about one third. Also, since a scalar prefix tree is able to store up to w prefixes in two w-bit words, it has the potential to compress the tree with a high ratio.

Appendix

Lemma 5. In scalar prefix search, any time the search for an address d reaches a key k that its max-length prefix is a prefix of d or if $p=MP(k)$ and $p \rightarrow d$, then p will be the $LMP(d)$; therefore, the search will be terminated.

Proof. 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 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 section III, 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, and R.L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.

[2] D.R. Morrison, "PATRICIA Practical Algorithm to Retrieve Information Coded in Alphanumeric," *J. ACM*, vol. 15, no. 14, Oct. 1968, pp. 514-534.

[3] S. Nilson and G. Karlsson, "IP Address Lookup Using LC-Tries," *IEEE JSAC*, vol. 17, June 1999, pp. 1083-1092.

[4] V. Srinivasan and G. Varghese, "Fast Address Lookups Using Controlled Prefix Expansion," *ACM Trans. Computer Syst.*, vol. 17, no. 1, Feb. 1999, pp. 1-40.

[5] L.C. Wuu, K.M. Chen, and T.J. Liu, "A Longest Prefix First Search Tree for IP Lookup," *Proc. ICC*, May 2005, pp. 989-993.

[6] P. Gupta, S. Lin, and N. McKeown, "Routing Lookups in Hardware at Memory Access Speeds," *Proc. IEEE INFOCOM*, 1998.

[7] W. Eatherton, G. Varghese, and Z. Dittia, "Tree

Bitmap: Hardware/Software IP Lookups with Incremental Updates," *Proc. ACM SIGCOMM*, 2004, pp. 97-122.

[8] B. Lampson, V. Srinivasan, and G. Varghese, "IP Lookups Using Multiway and Multicolumn Search," *IEEE/ACM Trans. Networking*, vol. 7, no. 3, June. 1999, pp. 324-334.

[9] P.R. Warkhede, S. Suri, and G. Varghese, "Multiway Range Trees: Scalable IP Lookup with Fast Updates," *Computer Netw.*, vol. 44, no. 3, 2004, pp. 289-303.

[10] H. Lu and S. Sahni, "A B-Tree Dynamic Router-Table Design," *IEEE Trans. Computers*, vol. 54, no. 7, 2005, pp. 813-824.

[11] Q. Sun et al., "A Scalable Exact Matching in Balance Tree Scheme for IPv6 Lookup," *ACM SIGCOMM Data Communication Festival, IPv6*, Aug. 2007.

[12] N. Yazdani and P. Min, "Prefix Trees: New Efficient Data Structures for Matching Strings of Different Lengths," *Proc. Int. Database Eng. Appl. Symp.*, July 2001, pp. 76-85.

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

[14] M. Behdadfar et al., "Scalar Prefix Search: A New Route Lookup Algorithm for Next Generation Internet." *Proc. IEEE INFOCOM*, Apr. 2009.

[15] M. Behdadfar et al., "IP Lookup Using the Novel Idea of Scalar Prefix Search with Fast Table Updates," *IEICE Trans. Inf. & Syst.*, vol. E93-D, no. 11, Nov. 2010, pp. 2932-2943.

[16] 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.

[17] K.H. Rosen and J.G. Michaels, *Handbook of Discrete and Combinatorial Mathematics*, CRC Press, 2000.

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

Mohammad Behdadfar received his BSc, MSc, and PhD in 1999, 2002, and 2010, respectively, from Isfahan University of Technology (IUT), all in electrical and computer engineering. He is currently with IUT and is an assistant professor in the Broadcast Engineering Department of IRIB University. His current research interests are in the area of high-speed networking, switch/router design, and algorithms.

Hossein Saidi received the BS and MS in electrical engineering in 1986 and 1989, respectively, both from IUT, Isfahan, Iran. He also received the DSc in electrical engineering from Washington University in St. Louis, USA, in 1994. Since 1995, he has been with the Department of Electrical and Computer Engineering at IUT, where he is currently an associate professor of electrical and computer engineering. His research interests include high speed switches and routers, communication networks, QoS in networks, queueing system, security, and information theory.

Masoud Reza Hashemix received his BSc and MS from IUT in 1986 and 1988 respectively, and his PhD from the University of Toronto in 1998, all in electrical and computer engineering. From 1988 to 1993, he was with IUT as a faculty member. From 1998, to 2000, he was a postdoctoral fellow at the University of Toronto. He was a founding member of AcceLight Networks in 2000 and has held various positions, including those in operational teams. Since 2003, he has been with IUT. His current research interests include communication networks, next generation services, TE in IP/MPLS, and sensor networks.

Ying-Dar Lin is a professor of computer science at National Chiao Tung University (NCTU), Taiwan. He received his PhD in computer science from UCLA in 1993. He spent his sabbatical year as a visiting scholar at Cisco Systems in San Jose during 2007 to 2008. Since 2002, he has been the founder and director of Network Benchmarking Lab (NBL), (www.nbl.org.tw), which reviews network products with real traffic. He also cofounded L7 Networks Inc. in 2002, which was later acquired by D-Link Corp. His research interests include design, analysis, implementation, and benchmarking of network protocols and algorithms, quality of services, network security, deep packet inspection, P2P networking, and embedded hardware/software co-design. His work on "multi-hop cellular" has been cited over 470 times. He is currently on the editorial boards of IEEE Communications Magazine, IEEE Communications Surveys and Tutorials, IEEE Communications Letters, Computer Communications, and Computer Networks. He published the textbook *Computer Networks: An Open Source Approach* with Ren-Hung Hwang and Fred Baker through McGraw-Hill in February 2011.