Efficient Prefix Updates for IP Router Using Lexicographic Ordering and Updateable Address Set

Sieteng Soh, *Member*, *IEEE*, Lely Hiryanto, and Suresh Rai, *Senior Member*, *IEEE*

Abstract—Dynamic IP router table schemes, which have recently been proposed in the literature, perform an IP lookup or an online prefix update in $O(log_2|T|)$ memory accesses (MAs). In terms of lookup time, they are still slower than the full expansion/compression (FEC) scheme (compressed next-hop array/code word array (CNHA/CWA)), which requires exactly (at most) three MAs, irrespective of the number of prefixes $|T|$ in a routing table T. The prefix updates in both FEC and CNHA/CWA have a drawback: Inefficient offline structure reconstruction is arguably the only viable solution. This paper solves the problem. We propose the use of lexicographic ordered prefixes to reduce the offline construction time of both schemes. Simulations on several real routing databases, run on the same platform, show that our approach constructs FEC (CNHA/CWA) tables in 2.68 to 7.54 (4.57 to 6) times faster than that from previous techniques. We also propose an online update scheme that, using an *updatable address set* and selectively decompressing the FEC and CNHA/CWA structures, modifies only the next hops of the addresses in the set. Recompressing the updated structures, the resulting forwarding tables are identical to those obtained by structure reconstructions, but are obtained at much lower computational cost. Our simulations show that the improved FEC and CNHA/CWA outperform the most recent $O(log_2|T|)$ schemes in terms of lookup time, update time, and memory requirement.

Index Terms—Dynamic router tables, IP address lookup, lexicographic ordering, online prefix updates.

1 Introduction

The IP address lookup in a router decides the next hop to forward each incoming packet toward its destination and it is still the bottleneck among the major tasks of a router [15]. A router maintains a list of pairs (prefix, next hop) and, as part of its forwarding task, the router has to quickly find the longest prefix that matches the W-bit destination address ($W = 32$ bits in IPv4) of an incoming packet. Fig. 1 shows an example of a set of prefixes with their corresponding next hops. In this example, a destination address 200.27.112.170 matches all prefixes except 200.27.240/20 and 200.27.128/20 and, therefore, the router should forward the packet to a next hop C since 11001000 00011011_0111* is the *longest matching prefix* (LMP).

An ideal scheme for an IP lookup solution includes fast lookup time, fast prefix update time, small memory requirement, and good scalability with respect to both the number and length of IP addresses [15]. The most important measure is obviously the lookup time since failure to meet the required time may result in loss of packets. Nevertheless, one should also consider the other metrics [15]. Every time there is a route change, for example, route replenishment, route failure, or route repair, one should update the contents of the router table to reflect the change. Table updates include an alteration to the next hop of an existing prefix or that of its default next hop and an insertion (deletion) of a new (existing) pair (prefix, next hop). Small update time is essential: Considering routing instability [6], update operations within 10 ms have been suggested [21].

The existing IP address lookup schemes fall into *software* and *hardware* approaches [15]. For each case, there are two different solutions to deal with prefix updates: *offline* and *online*. In an offline scheme, update operations are batched and the tables are periodically reconstructed [2], [3], [4], [5], [15], [16], [17]. References [2] and [5] have suggested offline software-based and hardware-based approaches, respectively. Because the routing protocols need time to converge, forwarding tables can be a little stale and therefore need not change more than at most once per second [3]. Note that an offline update scheme is acceptable if each table reconstruction can be done fast so that the table represents up-to-date network routing information. A technique in [14] reduces the construction time of the level-compressed trie (LC-trie) [12], while Sahni and Kim [17] and Wang et al. [24] improved the construction time of the multibit trie [21] and the multiway search tree [7], respectively.

For online updates, several dynamic router tables for IP lookup have been proposed [8], [9], [10], [11], [13], [18], [19], [20]. A modification to the LC-Trie [12] for online updates is described in [13]. The range encoding concept is proposed in [18] to improve the Multi-Way-Multi-Column scheme [7] so that each IP lookup or update can be performed in

- *S. Soh is with the Department of Computing, Curtin University of Technology, GPO Box U1987 Perth, Western Australia 6845. E-mail: S.Soh@curtin.edu.au.*
- *L. Hiryanto is with the Fakultas Teknologi Informasi, Blok R Lantai 11, Kampus I Universitas Tarumanagara, Jl. S. Parman No. 1, Grogol, Jakarta, Indonesia 11440. E-mail: lely.fti.untar@gmail.com.*
- *S. Rai is with the Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803. E-mail: suresh@ece.lsu.edu.*

Manuscript received 25 Oct. 2005; revised 30 Jan. 2007; accepted 13 June 2007; published online 18 July 2007.
Recommended for acceptance by F. Lombardi.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0370-1005.
Digital Object Identifier no. 10.1109/TC.2007.70776.

Prefix	Next Hop
200.27.240/20 = 11001000 00011011 1111*	*B*
200.27.128/20 =11001000 00011011 1000*	*A*
200.27.112/20 = 11001000 00011011 0111*	*C*
200.27.64/18 = 11001000 00011011 01*	*A*
200.27/16 = 11001000 00011011*	*C*
200.26/15 = 11001000 0001101*	*D*
200.24/14 = 11001000 000110*	*C*
ε	*D*

Fig. 1. An example of routing table T for IPv4.

$O(log_2|T|)$ memory accesses (MAs) for a routing table T with $|T|$ prefixes. Lu and Sahni proposed a priority search tree (PST) in [9], an enhanced interval tree in [10], and a B-tree data structure in [8] for use in dynamic IP router tables so that each IP lookup and prefix update can be performed in $O(log_2|T|)$. Reference [11] suggests the use of prefix and interval partitioning to improve their dynamic table structures.

In their survey paper, Ruiz-Sanchez et al. [15] have found the *full expansion/compression* (FEC) [2] to be the fastest software-based approach. With exactly three MAs per IP lookup (irrespective of $|T|$), the FEC obviously outperforms the recently proposed approaches [8], [9], [10], [11], [18], [19], which require $O(log_2|T|)$ MA (*plus*, presumably unreported, $\xi > 3$ clock cycles due to more complex lookup steps). On the other hand, with one MA or three MAs, the hardware-based scheme *compressed next-hop array/code word array* (CNHA/CWA) [5] is considerably faster than the recently proposed technique in [22] and *balanced routing table* (*BART*) *search* [23] that require five to nine and five to eight MAs per IP lookup, respectively. Note that the *fixed-stride trie* (FST) and *variable-stride trie* (VST) [21] can be tuned to provide a worst-case lookup time of three MAs; however, FST (VST) requires an additional 31 (35) clock cycles [21], in contrast to three for FEC. Furthermore, as will be discussed in Section 5.4, the average lookup time of FEC and CNHA/CWA is significantly faster than that of FST and VST [17]. However, a prefix update on either the FEC or CNHA/CWA scheme is difficult. An offline structure reconstruction is arguably [1], [15] the only viable update solution for both schemes and, thus, more efficient prefix updates algorithms for both FEC and CNHA/CWA are needed.

This paper proposes the use of decreasing lexicographic-ordered prefixes to speed up the FEC and CNHA/CWA construction time. The ordered prefixes help construct a set of *run length encoding* (RLE) sequences that, in turn, find use in the FEC and CNHA/CWA structures. Next, we describe an online prefix update scheme, each for FEC and CNHA/CWA. We employ an *updatable address set* to selectively decompress the FEC and CNHA/CWA structures, modifying only the next hops of the addresses in the set. Recompressing the updated structures, the resulting tables are identical to those obtained by the offline structure reconstruction, but at much lower computational cost.

The layout of this paper is organized as follows: Section 2 presents the notations and background that describe the FEC and CNHA/CWA schemes. Section 3 describes properties of lexicographic-ordered prefixes and their use in constructing RLE sequences. This section also discusses the application of RLE sequences to construct the FEC and CNHA/CWA structures. Section 4 explains the updatable address set and shows how the concept can be applied to enable the FEC and CNHA/CWA schemes for online updates. In Section 5, we present our experimental results by using the proposed techniques on several real routing tables and compare them with some existing IP lookup solutions. Finally, Section 6 concludes this paper.

2 Notations and Background

2.1 Notations

For a binary alphabet $\Sigma = \{0, 1\}$, we denote the set of all binary strings of length k (at most m) by Σ_k^* ($\Sigma_{\leq m}^* = \cup_{k=0}^{m} \Sigma_k^*$). Let Σ_k^0 (Σ_k^1) denote a string of 0s (1s) with length k. For two binary strings $\mu, v \in \Sigma_{\leq m}^*$ of length $l_\mu = |\mu|$ and $l_\nu = |\nu|$, respectively, we say that μ is a *prefix* of v, denoted by $\mu = prefix(\nu)$, if the first $l_\mu \leq l_\nu$ bits of ν are equal to μ. Furthermore, μ is the LMP of ν in some set T of prefixes if no other prefix of ν is longer than μ. We call ν an *exception* of μ if the first $l_\mu \leq l_\nu$ bits of ν are equal to μ. Let $LMP(\nu)$ be a function that obtains the LMP of ν from the set T and $exception(\mu)$ denote a function that returns all exceptions of μ in T. We denote by $\mu \cdot \nu$ the concatenation of μ and ν, that is, a string whose first $|\mu|$ bits equal μ and whose last $|\nu|$ bits equal ν. A prefix μ is the *aggregation* of a set of W-bit IP addresses μ^* that have μ as a prefix, that is, $\mu^* = \mu \cdot \Sigma_{W-|\mu|}^*$ and $W = 32$ for IPv4. Given a string μ and a set S of Σ_k^*, we define $\mu \cdot S = \{x | x = \mu \cdot \nu \text{ with } \nu \in S\}$. Let μ_s^l represent a substring of μ from bit s with length l for $0 \leq s \leq |\mu| - 1$ and $l \leq |\mu| - s$. As an example, for $\mu = 101001$, μ_0^3 returns 101, whereas μ_3^3 returns 001.

A routing table T contains a list of pairs $T_i = (p_i, h_i)$, where prefix $p_i \in \Sigma_{\leq W}^*$ and its next-hop interface h_i is an integer $[1 \ldots H]$, where H represents the total number of next-hop interfaces. Assume that T contains a pair $(\varepsilon, h_\varepsilon)$, where $\varepsilon(h_\varepsilon)$ is an empty string (the default next-hop interface). Let $|T|$ denote the total number of prefixes in T. A table T is sorted in decreasing lexicographic order if it contains a sequence $T_1, T_2, \ldots, T_{|T|}$ such that T_i precedes T_j if and only if $i < j$ and p_j is lexicographically in lower order than p_i (denoted as $p_i > p_j$). Note that $p_i > p_j$ if 1) $p_j = prefix(p_i)$ or 2) for some value of $0 < k \leq min(|p_i|, |p_j|)$, the first $k - 1$ bits of the two prefixes agree, but the kth bit of $p_i(= 1)$ is larger than the kth bit of $p_j(= 0)$. Fig. 1 illustrates the order.

2.2 Background

Given a routing table T, we can construct a *next-hop array (NHA)* of size 2^W*1, $NHA_1^{2^W} = \bigcup_{i=1}^{|T|} T_i'$, in which T_i' is constructed as described in [2]. Fig. 2 shows the *NHA* (called expanded table T' in [2]) of Fig. 1. Thus, the *NHA* contains all possible 2^W pairs (p_i, h_i) and we could solve the IP lookup problem in one MA. Unfortunately, the size of the *NHA* is prohibitively large (4 Gbytes for IPv4). To reduce the size of the forwarding table, an indirect lookup is employed [2], [3], [4], [5], [15]. In the following, we briefly describe the FEC [2] and CNHA/CWA [5] schemes.

Address Set	Next Hop	T'_i
200.27.240.0 to 200.27.255.255	B	T'_1
200.27.128.0 to 200.27.143.255	A	T'_2
200.27.112.0 to 200.27.127.255	C	T'_3
200.27.64.0 to 200.27.111.255	A	T'_4
200.27.0.0 to 200.27.63.255 200.27.144.0 to 200.27.239.255	C	T'_5
200.26.0.0 to 200.26.255.255	D	T'_6
200.24.0.0 to 200.25.255.255	C	T'_7
0.0.0.0 to 200.23.255.255 200.28.255.255 to 255.255.255.255	D	T'_8

Fig. 2. The expanded routing table T' for Fig. 1.

2.2.1 The FEC Techniques

Crescenzi et al. [2] propose an FEC structure that is comprised of a 2D $NHA_{\beta_c}^{\alpha_r}$ (called table F) with $\alpha_r * \beta_c$ entries ($r = c = 16$, $\alpha_r \leq 2^r$, and $\beta_c \leq 2^c$) and two segment tables: *row index* R and *column index* C, each with 2^r and 2^c entries, respectively, such that each entry in $R(C)$ is a pointer to a corresponding row (column) of F. In the FEC scheme, a 32-bit address $X = a.b.c.d$ is split into $X_0^r = a.b$ and $X_r^c = c.d$ and a lookup for X obtains $h_x = F[R[X_0^r], C[X_r^c]]$ in three MAs.

To build the FEC structure, [2] implicitly uses an $NHA_1^{2^{32}} = \bigcup_{i=1}^{|T|} T'_i$. A break bit, $r = 16$, is used for grouping the 32-bit routes into rows 0.0 through 255.255 such that a route μ with row address $\mu_0^r = a.b$ is in row $a.b$. To reduce the size of table F, the elements in each row are compressed into a sequence of RLEs such that a sequence of elements in the row that have the same next hop are represented by one RLE. Next, make any set of rows that contain the same RLE sequence information point to only one copy of information. Finally, [2] applies a *unification* step by using the recursive *function* φ. It performs on each of the RLE so that all RLE sequences have the same length β_c, which, in turn, are used for constructing the FEC structure (refer to Fig. 3).

Even though the approach has a worst-case memory requirement of $O(2^{W/2} + |T|^2)$, several simulations using real routing tables show that the technique requires less than 2.3 Mbytes of memory. However, a prefix update on the scheme is difficult. An offline structure reconstruction has been suggested in [2], [15]. However, the software implementation of the algorithm in [2] requires hundreds of milliseconds to build the FEC tables and, therefore, a more efficient table construction technique is required. In this paper, we propose a faster algorithm for constructing RLE sequences and an efficient unification technique for reducing the FEC construction time. Furthermore, utilizing the *updatable address set* concept (described in Section 4.1), we propose a *dynamic* FEC (DFEC) structure that supports online prefix updates.

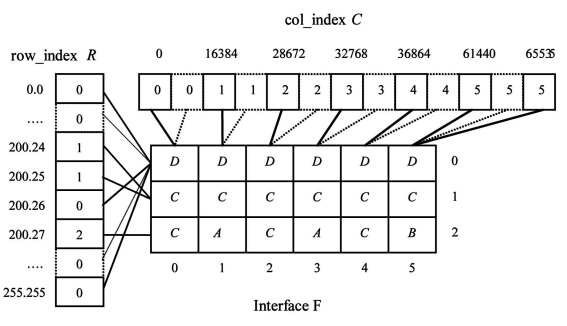

Fig. 3. FEC tables for Fig. 1.

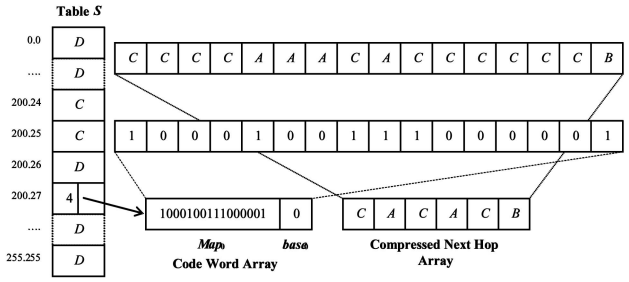

Fig. 4. Tables S, *NHA*, *CBM*, *CNHA*, and *CWA*.

2.2.2 The CNHA/CWA Scheme

The CNHA/CWA technique splits each IP address $X = a.b.c.d$ into a *segment* $a.b$ and an *offset* $c.d$. Huang and Zhao [5] proposed using an ST S with 2^{16} entries, each of which stores either a next hop (value < 256 if the length of the longest prefix in this segment $l \leq 16$) or a pointer (value ≥ 255) to an associated $NHA_1^{2^{16}}$ with 2^{16} entries that contains the next hop. They [5] took advantage of the distribution of the prefixes within a segment to reduce the size of its *NHA* so that the size depends on the length of the longest prefix in the segment $16 < l \leq 32$. The *NHA* of a segment with an *offset* length $k = l - 16$ has 2^k entries. In this approach, each entry in table S contains a 28-bit pointer or a 28-bit next hop and a 4-bit *offset* length k.

To further reduce the memory requirement of the scheme, they [5] converted each *NHA* into a *CWA* and a *CNHA*. A *compression bitmap* (*CBM*) is used for forming a *CWA*. The ith sequence of entries in an *NHA* (for example, from positions c to d for $0 \leq c \leq d \leq 2^k - 1$) with the same next hop h_x are represented by a "1" ("0") in bit position c (in each bit position ν for $c < \nu \leq d$) in its *CBM* and an h_x in the ith entry of its *CNHA*. An entry in *CWA* is comprised of a 16-bit *map* and a 16-bit *base*. A *CBM* obtains a *CWA* as follows: First, partition the *CBM* into a sequence of 16 bitstreams. Then, convert each ith 16 bitstream in the *CBM* into a map_i and $base_i$ in the ith entry of its *CWA* by copying the stream $(\sum_{j<i} \tau_j)$ to the $map_i(base_i)$, where τ_j represents the total number of bits "1" in the jth stream. Fig. 4 shows the CNHA/CWA structure of table T in Fig. 1.

Consider an IP lookup for an address $X = a.b.c.d$ that maps to $S[a.b]$, which contains an offset length k. We use $q = (c.d)_0^k$ to compute $s = (q \text{ DIV } 16)$ and $w = (q \text{ MOD } 16)$ and we calculate position $t = base_s + |w| - 1$ in *CNHA* that stores the next hop. Note that $|w|$ refers to the total number of bits "1" in bit positions 0 to w of map_s. For the CNHA/CWA structure in Fig. 4 and an address $X = 200.27.112.170$, we obtain $k = 4$ from $S[200.27]$ and, thus, $q = 0111 = 7$, $s = 0$, $w = 7$, $base_s = 0$, $map_s = 1000100111000001$, $|w| = 3$, and $t = 0 + 3 - 1 = 2$, and a next hop $CNHA[2] = C$ is obtained. Assuming a hardware implementation, each s, w, and $|w|$ are computable in one step [5]. Thus, the worst-case

IP lookup time is three MAs: one each to access S, *CWA*, and *CNHA*.

Huang and Zhao [5] proposed offline prefix updates by reconstructing the *CNHA* and *CWA* of each of the affected segments. They [5] used an $O(mlog_2m)$ function to construct the CNHA/CWA directly from the m prefixes in a segment. Unfortunately, their algorithm does not work in general. For the prefixes in Fig. 1, Step 4 of their algorithm produces an incorrect $A = \{S_0^0 = (0, C), S_1^0 = (4, A), S_3^0 = (8, A), S_0^3 = (9, C), S_4^0 = (15, B)\}$ for segment $S[200.27]$. To correct the problem, we propose including a step before Step 4 of the algorithm which sorts the elements in A in increasing order based on their $S_i^k.ma$. Elements with identical $S_i^k.ma$ are kept in the same order. Including our proposed step, we obtain the correct

$$A = \{S_0^0 = (0, C), S_1^0 = (4, A), S_2^0 = (7, C), S_3^0 = (8, A), S_0^3 = (9, C), S_4^0 = (15, B)\}.$$

In this paper, we propose the use of lexicographically decreasing ordered prefixes to construct the *CNHA* and *CWA* structures for a given segment in $O(m)$ time. In addition, using the *updatable address set* concept, we propose a technique to enable the CNHA/CWA scheme for online prefix updates.

3 Efficient Prefix Updates Using Lexicographic Ordered Prefixes

3.1 Some Properties of Lexicographic Ordered Prefixes

Let $A_q = p_q \cdot \Sigma^*_{W-|p_q|}$ denote the aggregated IP addresses of a prefix p_q, let $s_q = p_a \cdot \Sigma^0_{W-|p_q|}$ ($e_q = p_q \cdot \Sigma^1_{W-|p_q|}$) be the lowest (highest) address in A_q and consider a sequence of prefixes $\langle p_0, p_1, \ldots, p_{m-1}\rangle$ sorted in decreasing lexicographic order. In the following, we describe three properties of the relationships among the sorted prefixes and their address ranges.

Property 1. $s_0 \geq s_1 \geq \ldots \geq s_{m-1}$.

Proof. For $i < j$, consider two prefixes, p_i and p_j, in the sequence and their starting 32-bit aggregated IP addresses $s_i = p_i \cdot \Sigma^0_{W-|p_i|}$ and $s_j = p_j \cdot \Sigma^0_{W-|p_j|}$, respectively. By definition, 1) $p_j = prefix(p_i)$ or 2) for some value of $0 < k \leq min(|p_i|, |p_j|)$, the first $k-1$ bits of the two prefixes agree, but the kth bit of $p_i(=1)$ is larger than the kth bit of $p_j(=0)$. For case 1, $|p_j| < |p_i|$ and s_j contains at most as many bits "1" as s_i in their respective most significant bits and, thus, $s_i \geq s_j$. For case 2, s_i contains more bits "1" than s_j in their respective most significant bits and, thus, $s_i > s_j$. □

Property 2. *For $i < j$, if $p_j = prefix(p_i)$, then $A_i \subseteq A_j$.*

Proof. The relationship $A_i \subseteq A_j$ implies that $s_i \geq s_j$ and $e_i \leq e_j$. The proof for the case that $s_i \geq s_j$ follows case 1 of the proof for Property 1. When p_j is a prefix of p_i, $|p_j| < |p_i|$ and, thus, e_j contains at least as many bits "1" as e_i in their respective most significant bits and, thus, $e_j \geq e_i$. □

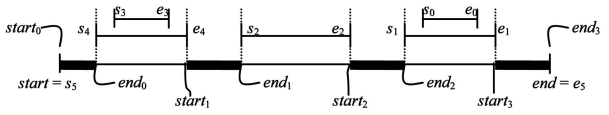

Fig. 5. Properties of lexicographic ordered prefixes.

Property 3. *For $i < j$, if $p_j \neq prefix(p_i)$, then $(s_j < e_j < s_i < e_i)$.*

Proof. The proof for the case that $s_i > s_j$ follows case 2 of the proof for Property 1. When p_j is not a prefix of p_i, by definition, e_j contains a smaller number of bits "1" than e_i in their respective most significant bits and, thus, $e_j < e_i$. □

Property 2 implicitly states that the next hop of each address in A_i is that of prefix p_i, whereas Property 3 implies that, for any two IP addresses $\eta \in A_i$ and $\mu \in A_j$, $\eta \neq \mu$ and $\eta > \mu$. Fig. 5 illustrates the properties for a sequence of sorted prefixes $\langle p_0, p_1, \ldots, p_5\rangle$, where $p_1(p_4)$ is a prefix of $p_0(p_3)$ and p_5 is a prefix of every other prefix in the sequence.

In this paper, we propose representing all pairs $T_i = (p_i, h_i)$ of a routing table T by using an ST which contains 2^{16} entries. A segment q, denoted by $ST[q]$ or ST_q, contains the length of the longest prefixes in ST_q, $l = max(l_j)$, and a list of triples $ST_q^j = (subprefix\ sp_j,\ prefix\ length\ l_j \leq 32,\ \text{next hop}\ h_j)$ for $j = 0, 1, \ldots, |ST_q| - 1$, sorted in decreasing lexicographic order following their *subprefixes*. Let $m = |ST_q|$ represent the number of prefixes in segment ST_q and *prefix list* denote a list of triples ST_q^j. Each T_i, with $|p_i| \geq 16$, is represented in a segment $ST[q = (p_i)_0^{16}]$ by a triple $ST_q^j = ((p_i)_{16}^{|p_i|-16}, |p_i|, h_i)$. On the contrary, each T_i, with $|p_i| < 16$, needs to be expanded into a set $(p_i \cdot \Sigma^*_{16-|p_i|}, h_i)$. Then, for each $T_i^{'} = (p_i^{'}, h_i) \in (p_i \cdot \Sigma^*_{16-|p_i|}, h_i)$, we create a triple $(0.0, |p_i|, h_i)$ and put it in each segment $ST[p_i^{'}]$. Thus, a T_i, with $|p_i| < 16$, is represented as a triple $(0.0, |p_i|, h_i)$ in $2^{16-|p_i|}$ segments. As an example, a

$$T_i = 200.27.240/20/B (200.27/16/C)$$

is represented in segment $ST[200.27]$ as a triple (240.0, 20, B) ((0.0, 16, C)) and $T_i = 200.24/14/C$ is (0.0, 14, C) in four segments: $ST[200.24]$, $ST[200.25]$, $ST[200.26]$, and $ST[200.27]$. Each triple (0.0, 14, C) in segments $ST[200.24]$ and $ST[200.25]$ is stored directly in the segments as a pair (14, C). Note that a triple in the segments requires at most 4 bytes of memory: 2 bytes for the *subprefix*, 5 bits for the *length*, and 1 byte for the *next hop*. Fig. 6a shows an ST for T in Fig. 1. Since the default pair $(\varepsilon, h_\varepsilon)$ may belong to all entries of segment ST, we do not explicitly store this pair in the table but store only its next hop h_ε as a global variable *default*. Note that any segment $ST[a.b] = \phi$ in Fig. 6a indicates that all addresses in range $(a.b.\Sigma^0_{16}, a.b.\Sigma^1_{16})$ are represented by $h\varepsilon$.

From a triple (sp_j, l_j, h_j) in $ST[a.b]$, we can obtain its equivalent pair $T_j = (p_j, h_j)$ as $p_j = (a.b)_0^{l_j}(p_j = (a.b \cdot sp_j)_0^{l_j})$ when $l_j \leq 16 (l_j > 16)$. Let $SL_q = \{(ss_0, se_0, h_0), (ss_1, se_1, h_1), \ldots, (ss_{m-1}, se_{m-1}, h_{m-1})\}$ be a sequence of triples (ss_i, se_i, h_i) generated from ST_q, where $se_i(ss_i)$ denotes the ending (starting) address range of sp_i for $0 \leq i \leq |ST_q| - 1$. The address ranges $0 \leq ss_i \leq se_i \leq 2^{16} - 1$

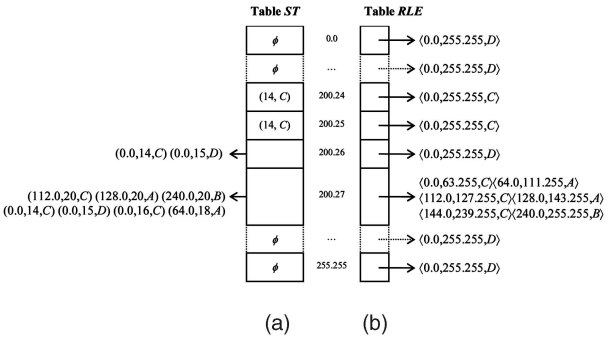

Fig. 6. Tables (a) ST and (b) RLE for T in Fig. 1.

are obtained as $se_i = 2^{16} - 1$ ($se_i = sp_i \cdot \Sigma^1_{32-|p_i|}$) and $ss_i = 0$ ($ss_i = sp_i \cdot \Sigma^0_{32-|p_i|}$) for $|p_i| \leq 16 (|p_i| > 16)$.

3.2 Efficient RLE Sequence Generation

For each segment $ST[q]$, we generate a set of sequences $RLE[q]$ (also tagged RLE_q); thus, for an ST, we obtain a table RLE. Each $RLE[q]$ contains a sequence of $RLE^i_q = \langle start_i, end_i, h_i \rangle$, where $i = 0, 1, \ldots, \rho - 1$ shows the RLE sequence number within the segment and $0 \leq start_i \leq end_i \leq 2^{16} - 1$ denotes the *starting* and *ending* addresses, respectively, such that each address within the range has a next hop h_i. Note that we use this table *RLE* to construct the FEC and CNHA/CWA structures.

Fig. 7 describes our function that constructs an $RLE[q]$ from the

$$SL[q] = \{(ss_0, se_0, h_0), (ss_1, se_1, h_1), \ldots, (ss_{m-1}, se_{m-1}, h_{m-1})\}$$

of a segment $ST[q]$. Let $RLE^i_q.field$ denote a $field \in \{start_i, end_i, h_i\}$ of the RLE^i_q, and a pair $(srange_t, erange_t)$ refers to the top element in *Stack* (TOS) that stores the available address ranges. Step 1 first sets the initial value for *Stack* and $RLE[q]$. Based on the content of $ST[q]$, there are three conditions that can cause an $SL[q]$ to be empty: 1) $ST[q]$ contains a pair of prefix information (l_j, h_j), 2) $ST[q]$ contains a pointer to a prefix list where all triples (sp_j, l_j, h_j) in the list each have $l_j \leq 16$, or 3) $ST[q] = \phi$. For conditions 1 and 2, the *hop* is set to h_j and h_g, respectively, while, for condition 3, the *hop* is set to *default*. Note that h_g is the next hop of a triple with the longest l_j. For these three conditions, Step 2 of function **RLEGen** is skipped and only Steps 3 and 4 are executed to create $RLE[q] = \langle 0, 2^{16} - 1, hop \rangle (= \langle 0.0, 255.255, hop \rangle)$. In addition, when $ST[q]$ contains a pointer to a prefix list, where all $l_j > 16$, Step 1b sets $hop = default$. On the other hand, if there are one or more triples (sp_j, l_j, h_j) in $ST[q]$, with $l_j \leq 16$, Step 1b removes those triples and sets $hop = h_g$, where h_g is the next hop of the longest prefix among the triples. For these last conditions, the function in Fig. 7 then executes Steps 2 to 6 to generate more than one RLE.

Step 1 of **RLEGen** is computable in $O(m)$, whereas Step 2 is repeated m times. Note that the *while* loop in Steps 2b.iii and 5 are repeated at most $(2m - 1)$ times in total. With Step 6, which can be done at most $(2m - 1)$ times, the time complexity of **RLEGen** is $O(m)$.

Function RLEGen (*ST*[*q*]):
Input: the prefix information in *ST*[*q*]
Output: a set of *RLE* triples ($start_i$, end_i, h_i) for *RLE*[*q*]

1. Initialisation:
   a) *Stack*=(0,2^{16}-1) and *RLE*[*q*]=ϕ
   b) **if** (*ST*[*q*] contains a pair of (l_j,h_j)) **then** set *hop* = h_j
      else if (*ST*[*q*] contains a pointer to one or more triples (sp_j,l_j,h_j) with $l_j \leq 16$) **then**
      i. remove each triple's corresponding (ss_j, se_j, h_j) from *SL*[*q*]
      ii. set *hop* to h_g, the next hop of the longest prefix among the triples
      else set *hop* to *default*
2. **for** each triple (ss_j,se_j,h_j) in *SL*[*q*] **do** //when *SL*[*q*] $\neq \phi$
   //*t* = |*Stack*|-1, an index to the top element of *Stack* (TOS)
   a) **if** ($se_j \leq erange_t$) **then** // if $p_j \neq prefix(p_{j-1})$, where *j*>0
      i. insert $\langle ss_j, se_j, h_j \rangle$ into *RLE*[*q*] at the front
      ii. **if** ($se_j < erange_t$) **then** //the available address range exists
         replace TOS with (se_j+1,$erange_t$)
      else pop TOS
   b) **else** // if $p_j = prefix(p_{j-1})$, where *j*>0
      i. **if** ($erange_t \geq ss_j$) **then** insert $\langle ss_j, erange_t, h_j \rangle$ into *RLE*[*q*] at the front
      ii. pop TOS
      iii. **while** (*Stack*$\neq\phi$ and $se_j > srange_t$) **do**
         if ($se_j < erange_t$) **then**
            replace TOS with (se_j+1,$erange_t$)
            insert $\langle srange_t, se_j, h_j \rangle$ into *RLE*[*q*] after RLE^i_q where ($RLE^i_q.end$ +1==$srange_t$)
         else
            insert $\langle srange_t, erange_t, h_j \rangle$ into *RLE*[*q*] after RLE^i_q where ($RLE^i_q.end$ +1==$srange_t$)
            pop TOS
   c) push (0,ss_j–1) into *Stack*
3. **if** ($erange_t \neq$-1) **then** insert $\langle ss_j, erange_t, h_j \rangle$ into *RLE*[*q*] at the front
4. pop TOS
5. **while** (*Stack*$\neq\phi$) **do**
   a) insert $\langle srange_t, erange_t, hop \rangle$ into *RLE*[*q*] after RLE^i_q where ($RLE^i_q.end$ +1==$srange_t$)
   b) pop TOS
6. Replace any *RLE* sequence $\langle ss_j, se_j, h_j \rangle \cdot \langle ss_{j+1}, se_{j+1}, h_{j+1} \rangle \cdot \ldots \cdot \langle ss_{j+x}, se_{j+x}, h_{j+x} \rangle$ in *RLE*[*q*] where $h_j = h_{j+1} = \ldots = h_{j+x}$, with $\langle ss_j, se_{j+x}, h_j \rangle$

Fig. 7. Function **RLEGen**.

As an example, consider $ST[200.27]$ in Fig. 6a and

$$SL_{200.27} = \{(240.0, 255.255, B), (128.0, 143.255, A), (112.0, 127.255, C), (64.0, 127.255, A), (0.0, 255.255, C), (0.0, 255.255, D), (0.0, 255.255, C)\}.$$

We use **RLEGen** to generate an *RLE* sequence for $RLE[200.27]$. Initially, $Stack = (0, 255.255)$, $RLE[q] = \phi$, $hop = C$ (because 0.0/16 is the longest prefix among the prefixes with $l_j \leq 16$) and $SL_{200.27}$ is updated to

$$\{(240.0, 255.255, B), (128.0, 143.255, A), (112.0, 127.255, C), (64.0, 127.255, A)\}.$$

For the first $SL[q]$, (240.0, 255.255, B), Step 2a.i generates $RLE^0_{200.27} = \langle 240.0, 255.255, B \rangle$ and updates

Function RSE (table *RLE*):
Input: a set of non-duplicated RLE sequences, i.e., table *RLE*
Output: aligned table *RLE*
1. Flag the first RLE in each non-duplicated RLE sequence.
2. Set δ = the minimum (end_i-$start_i$+1) value among those of the flagged RLE.
3. Replace each flagged $RLE\langle start_i, end_i, h_i\rangle$ that has (end_i-$start_i$+1)>δ with $RLE\langle start_i, \delta$+$start_i$-1,$h_i\rangle$ and $RLE\langle \delta$+$start_i, end_i, h_i\rangle$, and flag the second *RLE* of the result.
4. Unflag each flagged *RLE* $\langle start_i, end_i, h_i\rangle$ that has (end_i-$start_i$+1)=δ, and flag its next RLE in the sequence.
5. Repeat Step 2, until all flagged *RLE* have the same $\delta = end_i$-$start_i$+1, for all non-duplicated RLE sequences.

Fig. 8. Function RSE.

$$Stack = \{(0, 239.255)\}.$$

For the next $SL[q]$, (128.0,143.255, A), Step 2a.i generates $RLE^1_{200.27} = \langle 128.0, 143.255, A\rangle$ and Step 2a.ii removes the top element and pushes two new address ranges into $Stack(= \{(0, 127.255)(144.0, 239.255)\})$, where the last pair in *Stack* (that is, (144.0, 239.255)) is the available address range between $RLE^0_{200.27}$ and $RLE^1_{200.27}$. $RLE^2_{200.27} = \langle 112.0, 127.255, A\rangle$ is obtained from (112.0, 127.255, C) by popping the range (0, 127.255) from *Stack* and pushing a new range (0, 111.255) into the *Stack*. For the last $SL[q]$, (64.0, 127.255, A), Steps 2b.i and 2b.ii are executed. However, since $se_3(= 127.255)$ is less than $srange_t(= 144.0)$, the RLE generation is continued to Step 3. Executing Steps 3 to 6, we obtain

$$\begin{aligned} RLE[200.27] = \{&\langle 0.0, 63.255, C\rangle\langle 64.0, 111.255, A\rangle \\ &\langle 112.0, 127.255, C\rangle\langle 128.0, 143.255, A\rangle \\ &\langle 144.0, 239.255, C\rangle\langle 240.0, 255.255, B\rangle\}. \end{aligned}$$

Fig. 6b shows the resulting table *RLE* of *ST* in Fig. 6a.

3.3 Improved Technique for FEC Table Construction

This section shows how we can convert table *RLE* into the FEC structure. Note that table *RLE* is equivalent to row R of the FEC structure and, hence, its conversion is straightforward. In addition, the row compression steps for FEC can be directly processed by sequentially deleting any duplicate RLE_q and adjusting its corresponding pointer. Let α_r be the number of nonduplicated RLE sequences of table *RLE*. As an example, each pointer in $RLE_{0.1}$ through $RLE_{200.23}$, $RLE_{200.26}$ and $RLE_{200.28}$ through $RLE_{255.255}$ ($RLE_{200.25}$) in Fig. 6b is adjusted to point to the element pointed by $RLE_{0.0}(RLE_{200.24})$ to obtain three *RLE*s: $RLE_{0.0}$, $RLE_{200.24}$, and $RLE_{200.27}$.

Crescenzi et al. [2] used a function φ so that each of the nonduplicate RLE sequences contains the same number of RLEs. In this unification step, an $RLE^i_q = \langle start_i, end_i, h_i\rangle$ may be expanded into

$$\langle start_i, \mu_1, h_i\rangle\langle \mu_1 + 1, \mu_2, h_i\rangle \ldots \langle \mu_{v-1} + 1, \mu_v, h_i\rangle,$$

where $end_i = \mu_v$ and $\mu_j + 1 = \mu_{j+1}$ for $1 \le j \le v - 1$. The function φ in [2] performs a row-based splitting. Our experiments show that, typically, table F has smaller columns than rows and, therefore, our unification method

R[0.0] : ⟨0,65535,D⟩*
R[200.24] : ⟨0,65535,C⟩*
R[200.27] : ⟨0,16383,C⟩*⟨16384,28671,A⟩⟨28672,32767,C⟩…
R[0.0] : ⟨0,16383,D⟩⟨16384,65535,D⟩*
R[200.24] : ⟨0,16383,C⟩⟨16384,65535,C⟩*
R[200.27] : ⟨0,16383,C⟩⟨16384,28671,A⟩*⟨28672,32767,C⟩…
R[0.0] : ⟨..⟩⟨16384,28671,D⟩⟨28672,65535,D⟩*
R[200.24] : ⟨..⟩⟨16384,28671,C⟩⟨28672,65535,C⟩*
R[200.27] : ⟨..⟩⟨16384,28671,A⟩⟨28672,32767,C⟩*⟨32768,36863,A⟩…
…
R[0.0] : ⟨..⟩⟨..⟩⟨..⟩⟨32768,36863,D⟩⟨36864,61439,D⟩⟨61440,65535,D⟩
R[200.24] : ⟨..⟩⟨..⟩⟨..⟩⟨32768,36863,C⟩⟨36864,61439,C⟩⟨61440,65535,C⟩
R[200.27] : ⟨..⟩⟨..⟩⟨..⟩⟨32768,36863,A⟩⟨36864,61439,C⟩⟨61440,65535,B⟩

Fig. 9. Unification of RLE sequences in Fig. 6b.

RLE-sequence-expansion (RSE), which is shown in Fig. 8 and does a columnwise adjustment, is expected to be more efficient.

Fig. 9 illustrates function RSE to uncompress the *RLE*s in Fig. 6b. Each of Steps 1 through 4 is computable in $O(\alpha_r)$ and Step 5 is repeated β_c times and, therefore, function **RSE** has a time complexity of $O(\alpha_r\beta_c)$. Note that Step 2 can be done while doing Step 1 and Steps 3 and 4.

Using functions **RLEGen** and **RSE**, we propose the FEC construction algorithm in Fig. 10, which constructs tables F, R, and C from table ST. For a routing table T, the worst-case complexity of Step 1 is $2^{16} * O(m)$, which can be estimated as $O(|T|)$, where $m \le |T|$ denotes the total number of prefixes in a segment. Step 2 can be completed in $O(2^{16} * m) = O(|T|)$, Step 3 can be done in $O(\alpha_r\beta_c)$, and Step 4 can be implemented as part of Step 3. Therefore, the complexity of the **FEC_construction** is $O(|T| + \alpha_r\beta_c)$. Note that the FEC construction approach in [2] requires $O(|T|log_2|T| + \alpha_r\beta_c)$ asymptotic time, considering an $O(|T|log_2|T|)$ sorting algorithm. Our approach is more efficient than that in [2] because 1) it does not require the prefixes to be sorted for generating RLE sequence and 2) it uses column-based unification, in contrast to the row-based unification in [2].

3.4 Improved Technique for the CNHA/CWA Construction

The CNHA/CWA structure [5] can be constructed from table *RLE*. We first construct table S from ST. Let $0 \le l \le 32$ be the length of the longest prefix in ST_q. For $l \le 16(l > 16)$, $S[q] = h_0(S[q].offset\_length = l - 16)$. For $l > 16$, we then use the function in Fig. 11 to construct

Algorithm FEC_construction:
Input: table *ST*
Output: tables *R*,*C* and *F*
1. **for** each segment *ST*[*q*] **do** //construct table *RLE*
   call **RLEGen** (*ST*[*q*]) //to construct table *RLE*[*q*]
2. Row-compress table *RLE* //table *R* is constructed
3. call **RSE** (table *RLE*) //to align all columns of the compressed table *RLE*
4. Construct tables *F* and *C* from the aligned table *RLE*

Fig. 10. Algorithm FEC_construction.

Function CNHA/CWA_from_RLE(RLE_q) :
Input: a set of *RLE* pairs ($start_i$, end_i, h_i) in $RLE[q]$
Output: $CNHA_q$ and CWA_q
1. $nbase = 0$
2. **for** $i = 0$ to $|RLE_q|$-1 **do**
3. $CNHA_q[i] = h_i$;
4. $a_i = start_i/2^{32-l}$; // l=$max(l_j)$
5. $s = a_i$ DIV 16; $w = a_i$ MOD 16;
6. $CWA_q[s].map_w = 1$; $nbase = nbase + 1$
7. $CWA_q[s+1].base = nbase$

Fig. 11. Function **CNHA/CWA_from_RLE**.

a $CNHA_q$ and CWA_q from $RLE[q] = \langle start_0, end_0, h_0\rangle \langle start_1, end_1, h_1\rangle \ldots \langle start_{\rho-1}, end_{\rho-1}, h_{\rho-1}\rangle$.

Note that $0 \le start_i \le end_i \le 2^{16} - 1$ and, thus, Step 4 of the function adjusts the range to $0 \le a_i \le 2^{l-16} - 1$. Then, Steps 5, 6, and 7 convert the adjusted *RLE* into *CNHA* and *CWA*. To illustrate the function, consider $RLE[200.27]$ in Fig. 6b, with $|RLE_q| = 6$ and $l = 20$. For $i = 0$, $CNHA[0] = C$, $start_0 = 0$, $a_0 = 0$, $s = 0$, $w = 0$, $CWA[0].map = 1000000000000000$, and $CWA[1].base = 1$. For $i = 1$, $CNHA[1] = A$, $start_1 = 16,384$, $a_1 = 4$, $s = 0$, $w = 4$, $CWA[0].map = 1000100000000000$, and

$$CWA[1].base = 2.$$

Repeating the process, we obtain the *CNHA* and *CWA*, as shown in Fig. 4. Note that each $|RLE_q| \le 2m - 1$ and, therefore, the time complexity of the function is $O(m)$, which is more efficient than the $O(mlog_2 m)$ approach in [5].

Using functions **RLEGen** and **CNHA/CWA_from_RLE**, in Fig. 12, we propose a CNHA/CWA construction algorithm that builds tables *S*, *CNHA*, and *CWA* from table *ST*. Since the *for loop* is repeated at most 2^{16} times, the complexity of our technique is $O(|T|)$.

4 Using Updatable Address Set for Online Prefix Updates

4.1 Updatable Address Set for an Inserted/Deleted Prefix

We consider two prefix update operations: insertion and deletion. A next-hop alteration can be done by a prefix insertion. Consider the insertion/deletion of a pair $T_i = (p_i, h_i)$ to/from a table T. Since prefix p_i represents aggregated addresses $p_i^* = p_i \cdot \Sigma^*_{W-|p_i|}$, it is obvious that its insertion/deletion *may* affect only the next hop of the addresses in p_i^*. However, as illustrated in Fig. 13, the next hop of some of the addresses in p_i^* should be kept unchanged

Algorithm CNHA/CWA_construction:
Input: all of the prefix lists in table *ST*
Output: tables *S*,*CNHA* and *CWA*
for each segment $ST[q]$ **do**
call **RLEGen**($ST[q]$) //to construct table $RLE[q]$
if $|RLE[q]| = 1$ **then** store h_i directly in segment q
else call **CNHA/CWA_from_RLE**($RLE[q]$) //to construct //the segment's *CNHA* and *CWA*.

Fig. 12. The **CNHA/CWA_construction** algorithm.

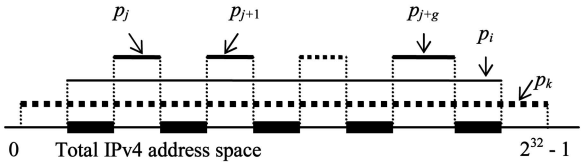

Fig. 13. The updateable address set.

because it *may* represent that of $exception(p_i)$, that is, $p_j, p_{j+1}, \ldots, p_{j+g}$, where $p_i = prefix(p_{j+\theta})$ for $\theta = 0, 1, \ldots, g$. For $p_i = prefix(p_j)$, let $excepted(p_i, p_j)$ be the addresses in p_j^* that are part of the addresses in p_i^*. In this paper, we call a set of addresses whose next hop *should* be updated when a T_i is inserted or deleted from the routing table T an *updateable address set*, denoted as $updateable(p_i)$. The following property shows how we can generate the *updateable address set*:

Property 4. *$updateable(p_i) = p_i^* - \cup_j excepted(p_i, p_j)$ for all $p_j \in exception(p_i)$, where "$-$" is a set difference operator.*

As an example, prefix 200.27.112/20 in Fig. 1 is the only prefix exception of 200.27.64/18 and, therefore, $updateable(200.27.64/18)$ are the addresses from 200.27.64.0 to 200.27.111.255. Following the property and to support online prefix update, we use table *ST* (described in Section 3.1) so that the prefix exceptions of the inserted/deleted prefix can be obtained. Once the set $updateable(p_i)$ is generated, the next hop of each address in the set should be replaced with a *new* next hop. For an *inserted* pair (p_i, h_i), h_i should replace that of each address in the *updateable address set*. The following property shows the new next hop for case prefix deletion.

Property 5. *Consider two pairs, (p_i, h_i) and (p_k, h_k), where prefix $p_k = LMP(p_i)$ and $p_i^* \subseteq p_k^*$. The deletion of (p_i, h_i) from the routing table T makes h_k the next hop of each address in set p_i^*.*

As illustrated in Fig. 13, a deletion of (p_i, h_i) results in replacing the next hop of each address in $updateable(p_i)$ (the boldest lines) with that of a prefix of $p_k = LMP(p_i)$, that is, h_k. As an example, if $T_i = 200.27.112/20/C$ is deleted from table T in Fig. 1, the next hop of each of the addresses from 200.27.112.0 to 200.27.127.255 should be updated with A (that is, the next hop of prefix $200.27.64/18 = LMP(200.27.112/20)$).

4.2 Generating Exception(p_i) and LMP(p_i)

Consider an inserted/deleted pair $T_i = (p_i, h_i)$ and a table *ST*. Let $PE_q(p_i) = exception(p_i)$ in $ST[q]$ and, without loss of generality, assume that each of the prefixes p_j in $PE_q(p_i)$ is denoted by its subprefix sp_j sorted in decreasing lexicographic order. The function in Fig. 14 generates the set $PE(p_i) = \{PE_q(p_i)\}$.

For case deletion, the search process continues to find the $LMP(p_i)$, as required by Property 5. Because elements in each segment are sorted in decreasing lexicographic order, an $LMP(p_i)$ is obtained from the first $prefix(p_i)$ down the list. Note that this step can be a part of function **gen_exception**. During the search process of finding

Function `gen_exception` (p_i):
Input: inserted/deleted prefix p_i
Output: a set of prefix exceptions $\{PE_q\}$ for all of the affected segments
1. **if** $|p_i|<16$ **then**
   for each $q \in p_i \cdot \Sigma^*_{16-|p_i|}$ **do**
   $PE_q = \phi$
   if $ST[q]$ contains a pair (l_j, h_j) where $l_j > |p_i|$ **then**
   $PE_q = \Sigma^*_{16}$ //indicates the prefix exceptions cover the range starting from 0.0 to 255.255
   else if $ST[q]$ contains one or more triples (sp_j, l_j, h_j) with $l_j > |p_i|$ **then** //for all j
   insert sp_j into set PE_q.
2. **if** $|p_i| \geq 16$ **then**
   $q = (p_i)^{16}_0$; $PE_q = \phi$
   if $ST[q]$ contains a pair (l_j, h_j) where $l_j > |p_i|$ **then**
   $PE_q = \Sigma^*_{16}$ //indicates the prefix exceptions cover the range starting from 0.0 to 255.255
   else if $ST[q]$ contains one or more triples (sp_j, l_j, h_j) with $l_j > |p_i|$ and p_i=*prefix*(p_j) **then** //for all j
   insert sp_j into set PE_q.

Fig. 14. Function `gen_exception`.

the exceptions, p_i can also be inserted (deleted) into (from) its proper location in $ST[q]$. It is obvious that, since $|ST_q| = m$, the time complexity of this function is bounded above by $O(m \cdot 2^{16-|p_i|})$, that is, when $|p_i| < 16$. Note that $2^{16-|p_i|} \leq 256$ and, therefore, the time complexity of the function is $O(m)$.

4.3 The Updateable Address Set Generation

Consider a set of prefix exceptions $PE_q = (sp_0, sp_1, \ldots, sp_{\beta-1})$ from segment $ST[q]$ of a prefix p_i and let $EL_q = (se_0, ss_0, se_1, ss_1, \ldots, se_{\beta-1}, ss_{\beta-1})$ be a sequence of address range for the subprefixes in PE_q. Utilizing Properties 2 and 4 and considering EL_q, we propose function **`gen_updatable`**, as shown in Fig. 15, to generate an *updatable address set* $U_q = \{(start_0, end_0), (start_1, end_1), \ldots\}$ of a subprefix sp_i. In the following, let *start* (*end*) be the lowest (highest) address range of sp_i of p_i in segment q, which can be computed as $start = 0$ and $end = 2^{16} - 1$ if $|p_i| \leq 16$. In addition, $start = (p_i)^{|p_i|-16}_{16} \cdot \Sigma^0_{W-|p_i|}$ and $end = (p_i)^{|p_i|-16}_{16} \cdot \Sigma^1_{W-|p_i|}$ if $|p_i| > 16$.

Fig. 5 illustrates the *updateable address set* (bold lines) for a sequence of sorted prefixes $< p_0, p_1, \ldots, p_5 >$, where $p_1(p_4)$ is a prefix of $p_0(p_3)$ and $p_i = p_5$ is a prefix of every other prefix in the sequence. As another illustration, consider the sequence of prefixes in Fig. 6a and an inserted pair $(p_i, h_i) = 200.27.224/19/B$ that maps to segment $ST[200.27]$ with address range 224.0 to 255.255. Function **`gen_exception`** finds only one subprefix $240.0/20/B$ in segment $ST[200.27]$, with address range 240.0 to 255.255 and function **`gen_updatable`** and then obtains $U_{200.27} = \{(57344, 61439)\}$. Similarly, for a deleted $(p_i, h_i) = 200.27.64/18/A$, we obtain a subprefix $112.0/20/B$, which is used for generating $U_{200.27} = \{(16384, 28671)\}$ and $LMP(p_i) = 0.0/16$ with next hop C. Because β in the function is, at most, the total number of prefixes in segment $ST[q](= m)$, the time complexity of function `gen_updateable` is $O(m)$.

Function `gen_updatable` (EL_q, p_i):
Input: a sequence of address ranges, EL_q, for prefixes in PE_q and the inserted and deleted prefix p_i
Output: the updatable address set, U_q
1. $U_q = \phi$; compute *start* and *end* from sp_i of p_i
2. **if** ($start < ss_{\beta-1}$) **then** insert a pair ($start$, ss_β-1) to U_q
3. $k = \beta$-1
4. **for** ($i = \beta$-2 to 0) **do**
   if ($se_k < se_i$) **then**
   if (se_k+1<ss_i-1) **then**
   insert a pair (se_k+1, ss_i-1) to U_q
   $k = i$
5. **if** ($se_k < end$) **then** insert a pair (se_k,*end*) to U_q

Fig. 15. Function `gen_updatable`.

4.4 DFEC Scheme

In an FEC structure with $\alpha_r \leq 2^r (\beta_c \leq 2^c)$, a row (column) in F may be used by more than one row (column) pointer and, hence, $F[\alpha, \beta]$ may represent the next hop of more than one IP address. Let $\Gamma_r(\alpha)(\Gamma_c(\beta))$ represent a set of row (column) addresses $\{X_r | R[X_r] = \alpha, X_r \leq 2^r\}(\{X_c | R[X_c] = \beta, X_c \leq 2^c\})$. In other words, for $X_r \in \Gamma_r(\alpha)$, and $X_c \in \Gamma_c(\beta)$, an IP address $X = X_r \cdot Xc$ has its next hop represented by $F[\alpha, \beta]$. Note that $|\Gamma_r(\alpha)|(|\Gamma_c(\beta)|)$ gives the total number of pointers to row α (column β) and $F[\alpha, \beta]$ represents the next hop of $|\Gamma_r(\alpha)| * |\Gamma_c(\beta)|$ number of IP addresses. Let us call $|\Gamma_r(\alpha)|(|\Gamma_c(\beta)|)$ the degree of row α (column β). The following property gives a set of IP addresses whose next hop is represented by $F[\alpha, \beta]$:

Property 6. *$F[\alpha, \beta]$ represents the next-hop information of a set of W-bit addresses $\Gamma_r(\alpha).\Gamma_c(\beta)$.*

As an example, $F[1, 0](= C)$ in Fig. 3 represents the next hop of 16384^*2 addresses: 200.24.0.0 to 200.24.63.255 and 200.25.0.0 to 200.25.63.255. Note that the column addresses in each $\Gamma_c(\beta)$ are consecutive 16-bit integers. This observation is stated in the following property:

Property 7. *For any column β in table F, the 16-bit addresses in $\Gamma_c(\beta)$ are one or more consecutive integers in the range $f_\beta, f_\beta + 1, \ldots, f_\beta + |\Gamma_c(\beta)| - 1$ for $f_\beta \geq 0$.*

As an example, the column pointers to $F[*, 0]$ in Fig. 3 are 16,384 consecutive addresses (0 to 16,383). In DFEC, an array *row_degree* of size $\alpha^*_r 2$ bytes is used for storing $|\Gamma_r(\alpha)|$ of each row α. In addition, an array *column_degree* is used for storing pairs of $(|\Gamma_c(\beta)|, f_\beta)$ of each column β, where f_β denotes the starting column address in table C, whose content is a pointer to column β.

Given an inserted/deleted pair (p_i, h_i), our approach runs in three phases. In the *partial expansion* phase, our method selectively expands the rows and/or columns of table F so that the next hop of each address in the *updateable address set* can be modified. In the *update* phase, we replace the next hop of each address in $updateable(p_i)$ with a new next-hop information. Finally, in the *compression* phase, the updated tables F, R, and C are recompressed. In Fig. 16, we show our proposed online prefix update technique for

Algorithm prefix_ update_DFEC:
Input: the inserted/deleted prefix p_i
Output: updated tables F, R and C
1. PE = **gen_exception** (p_i)
2. **for** each $PE_q \in PE$ **do**
3. U_q = **gen_updatable** (EL_q, p_i)
4. **if** $U_q \neq \phi$ **then**
$\alpha = R[q]$
if $|\Gamma_r(\alpha)| > 1$ **then**
$F[\alpha_r, *] = F[\alpha, *]$; $R[q] = \alpha_r$; $\alpha_r = \alpha_r + 1$
call **update_FEC_table** (U_q, hop)
5. Recompress the updated tables F, R, and C

Fig. 16. Algorithm **prefix_update_DFEC**.

DFEC. For an inserted (deleted) pair $T_i = (p_i, h_i)$, let $hop = h_i (hop = h_j; p_j = LMP(p_i))$.

In Step 4, if the row degree of each of the updateable rows is more than one, it creates a copy of the row at a new row $\alpha_r + 1$, increments α_r by one, and adjusts the corresponding addresses in table R to point to the new row. Step 4 then updates the contents of tables F, R, and C by using function **update_FEC_table** in Fig. 17, where $U_q = \{(start_0, end_0), (start_1, end_1), \ldots, (start_l, end_l)\}$, hop is the new next-hop information, and $\Gamma_c(\beta) = \{f_\beta, f_\beta + 1, \ldots, f_\beta + |\Gamma_c(\beta)| - 1\}$.

Function **update_FEC_table** is used for each nonempty U_q. Note that each column address range from $start_i$ to end_i may span 1) all column addresses within column β, 2) a part of the column addresses in β, or 3) column addresses of more than one column starting from β. Case 3 occurs when the consecutive columns in a row contain the same next hop. Fig. 18 illustrates the three cases. For case 1, we simply update the next-hop information in $F[\alpha, \beta]$. Let us denote f_β as the starting address of the address range $\Gamma_c(\beta)$. In case 2, if $start_i$ (end_i) is greater (less) than f_β $(f_\beta + |\Gamma_c(\beta)| - 1)$, we need to create a column for the address range f_β to $start_i - 1((end_i + 1)$ to $(f_\beta + |\Gamma_c(\beta)| - 1))$ before updating the

Function update_FEC_table (U_q, hop):
Input: the updatable address set U_q; for insertion, let $hop=h_i$, while for deletion, let $hop=h_j$, where $p_j=LMP(p_i)$
Output: updated $F[\alpha,*]$
for each pair $(start_j, end_j) \in U_q$ **do**
1. $\alpha = R[q]$
2. $\beta = C[start_j]$
3. $\gamma = C[end_j]$
4. **if**($f_\beta < start_j$) **then** //for cases (ii) and (iii)
$F[*,\beta_c] = F[*,\beta]$ //copy column
for $g = f_\beta$ to $start_j - 1$ **do** $C[g] = \beta_c$ //update C
$\beta_c = \beta_c + 1$
5. **if**($f_\gamma + |\Gamma_c(\gamma)| - 1 > end_j$) **then** //for cases (ii) and (iii)
$F[*,\beta_c] = F[*,\gamma]$ //copy column
for $g = end_j + 1$ to $f_\gamma + |\Gamma_c(\gamma)| - 1$ **do** $C[g] = \beta_c$ //update C
$\beta_c = \beta_c + 1$
6. **if**($\beta = \gamma$) **then** $F[\alpha,\beta] = hop$ //new hop for cases (i) and (ii)
7. **else** //case (iii)
$g = start_j$
while $g \leq end_j$ **do** //new hop for e columns in row α
$\beta = C[g]$; $F[\alpha,\beta] = hop$; $g = f_\beta + |\Gamma_c(\beta)|$

Fig. 17. Function **update_FEC_table**.

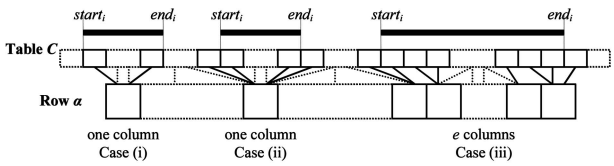

Fig. 18. Three cases for address range $start_i, \ldots, end_i$.

content of $F[\alpha, \beta]$ with a new next hop. For case 3, assume that the address range spans e columns (as illustrated in Fig. 18). Similarly to the step in case 2, if $start_i (end_i)$ is greater (less) than $f_\beta (f_{\beta+e} + |\Gamma_c(\beta + e)| - 1)$, we need to create a column for the address range f_β to $start_i - 1$ $(end_i + 1$ to $(f_{\beta+e} + |\Gamma_c(\beta + e)| - 1))$. Also update the contents of e columns (of table F in row α that are pointed to by the update address range) with a new next hop. Thus, at most two columns are created in cases 2 and 3.

Our simulation shows that a typical prefix insertion or deletion requires more row expansions than column expansions, with some of the expanded rows and/or columns being duplicates. In our implementation, the F table is constructed from an array of rows and, therefore, copying a row is faster than duplicating a column. After a row is created and updated, we check if the row is a duplicate of any of the other existing rows. To speed this step up, we keep a hash value for each row in an array *row_hash* so that a possible duplication can be detected in $O(1)$ if the hash value of the new row is the same as that for any of the existing rows. The contents of the two rows are compared (in $O(\beta_c)$) when the rows have the same hash value. To merge two identical rows, all pointers to the deleted row are updated to point to the surviving row. This step can be done in $O(2^{16-|p_i|} \leq 256) = O(1)$. The column compression is done similarly, with the additional task of recomputing the hash values for the rows, which requires $O(\alpha_r)$ steps. Step 4 of the function in Fig. 17 creates at most two column copies, each requiring $O(\alpha_r)$. Note that the *for loops* in Steps 4 and 5 are each executed in a total of $2^{32-|p_i|} < 2^{16}$ times (bounded above by the size of *col_index* C) and the *while loop* in Step 7 repeats a total of β_c times. Because all of the steps are repeated $|U_q| = m$ times, the complexity of **update_FEC_table** is $O(m^*\alpha_r + \beta_c + 2^{32-|p_i|}) = O(m^*\alpha_r)$. We compute the complexity of the **prefix_update_DFEC** algorithm in Fig. 16 as follows: Steps 1 and 3 each need $O(m)$, while Steps 4 and 5 require $O(m^*\alpha_r)$ and $O(\alpha_r^*\beta_c)$, respectively. Because Steps 3 and 4 are repeated $2^{16-|p_i|} \leq 256$ times, the computational complexity of our online prefix update for the FEC scheme is $O(m^*\alpha_r^* 2^{16-|p_i|} + \alpha_r^*\beta_c) = O(m^*\alpha_r)$, assuming that $\beta_c < m^* 2^{16-|p_i|}$.

To illustrate our DFEC, consider table T in Fig. 1 with its FEC in Fig. 3 and an insertion of $(p_i = 110010000001101110^*, A)$. We generate

$$p_i^* = 110010000001101110 \cdot \Sigma_{14}^*,$$
$$exception(p_i) = PE_i = \{1100100000110111000^*\},$$

and $excepted(p_i, PE_i) = \{200.27.128.0, \ldots, 200.27.143.255\}$ and, hence,

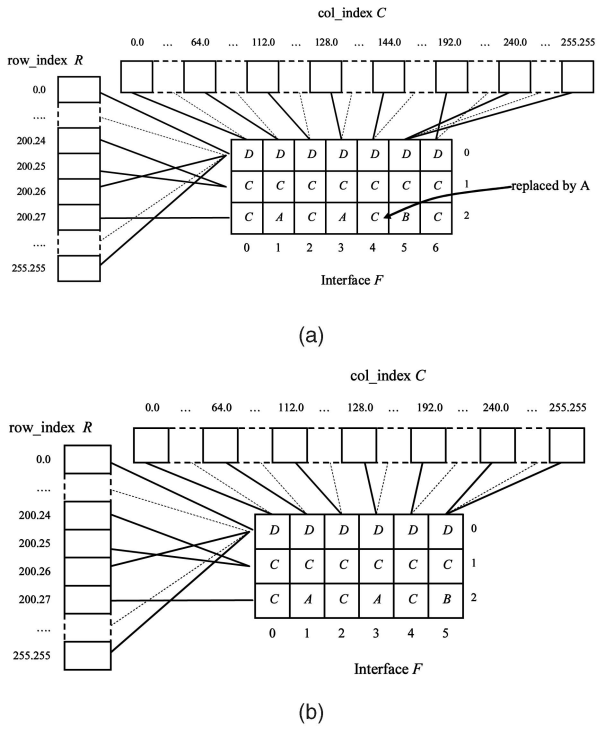

Fig. 19. Inserting 200.27.128/18/A into the FEC in Fig. 3. (a) Expand-and-update step. (b) Compressed step.

$$updateable(p_i) = \{200.27.144.0, \ldots, 200.27.191.255\}.$$

Note that the *updateable* addresses refer to row $\alpha = 2$ (pointed to by index $q = 200.27$) and column $\beta = 4$ (pointed to by index 49,152 ... 61,439). Following case 2, the column is expanded, and the content of $F[2,4]$ is replaced by A, as shown in Fig. 19a. The updated FEC table, as shown in Fig. 19b, is obtained after compressing columns 3 and 4 in Fig. 19a. Similarly, deleting a pair $(p_i = 110010000001101110^*, A)$ from the FEC in Fig. 19b, we find C as the next hop of $LMP(p_i) = 200.27/16$, and $updateable(p_i) = \{200.27.144.0, \ldots, 200.27.191.255\}$. Partially expanding the FEC in Fig 19b, the FEC in Fig. 19a is obtained. Replacing $F[2,4]$ with C and compressing the result, we obtain the FEC in Fig. 3.

4.5 Online Update Technique for the CNHA/CWA Scheme

For a pair (p_i, h_i) that is deleted (inserted) from (into) table T, when $|p_i| \geq 16$, there is only one affected segment, $S[q = (p_i)_0^{16}]$. For $|p_i| < 16$, on the other hand, there is a set of $2^{16-|p_i|}$ affected segments, $A = p_i \cdot \Sigma^*_{16-|p_i|}$. An affected segment may contain the next-hop information (case 1) or a pointer to its CNHA/CWA structure (case 2). In Fig. 20, we propose an algorithm for performing online update on CNHA/CWA of an affected segment $S[q]$. Note that $PE_q = exception(p_i)$ in $ST[q]$ and $hop = h_k (hop = h_i)$ for a deleted (inserted) pair (p_i, h_i), where $p_k = LMP(p_i)$.

For case 1, our function **update_CNHA_CWA_1** (shown in Fig. 21) utilizes the *updateable address set* concept to modify the CNHA/CWA structure when a pair (p_i, h_i) is inserted or deleted. When $|p_i| \leq 16$ and p_i is not a prefix of any prefix in the segment, we only need to update the next-hop information for the segment with the new next hop.

Algorithm prefix_update_CNHA/CWA:
Input: the inserted/deleted prefix p_i
Output: modified table *S*, *CNHA* and *CWA*
1. *PE* = **gen_exception** (p_i)
2. **for** each $PE_q \in PE$ **do**
   Case 1: **if** *S*[*q*] contains a next hop information **then**
   call **update_cnha_cwa_1** (p_i, *hop*, PE_q, S_q)
   Case 2: **if** *S*[*q*] contains a pointer to CNHA/CWA **then**
   call **update_cnha_cwa_2** (p_i, *hop*, PE_q, S_q)

Fig. 20. Algorithm **prefix_update_CNHA/CWA**.

The function first obtains U_q (Step 2), which, in turn, is used for modifying the affected RLE sequence. Then, in Step 6, if the updated RLE sequence contains only one field $\langle start_i, end_i, h_i \rangle$, h_i is directly stored in the segment. Otherwise, Step 7 generates the updated CNHA/CWA from the modified RLE sequence.

Steps 1 and 7 of the function in Fig. 21 each need $O(m)$ and the *for loop* in Step 4 is visited $|U_q| = O(m)$ times. Therefore, the time complexity of the function is $O(m)$.

For case 2 in Fig. 20, our function **update_CNHA_CWA_2** (shown in Fig. 22) performs the necessary partial *CWA* and *CNHA* expansions and updates each nonempty U_q. For an affected $ST[q]$, Step 1 of the function generates U_q. Let l be the length of the longest prefix in $ST[q]$. Step 2 considers two different cases. For an inserted prefix p_i, when $|p_i| > l$, we need to expand the size of the existing *CWA*. Note that $|CWA| = \lceil 2^{l-16}/16 \rceil$ and, thus, when $|p_i| > l > 16$, the updated *CWA* contains $\lceil 2^{|p_i|-16}/16 \rceil$ *maps* and *bases*. Let $b = |p_i| - l$ be the resized factor. A "1" in bit position d of CWA_i maps to a "1" in bit position $((i^*16 + d)^*2^b)$ MOD 16 in its expanded $newCWA_s$, where $s = ((i^*16 + d)^*2^b)$ DIV 16. This resizing step includes recalculating the *bases* of the new *CWA*. As an example, consider the *CWA* of a segment $S[200.27]$ in Fig. 4, with $l = 20$, and an inserted $p_i = 220.27.192/21/B$. Because $|p_i| > l$, we obtain $b = 21 - 20 = 1$ and, hence, the *CWA* is expanded from size 1 into $\lceil 2^{21-16}/16 \rceil = 2$. A "1" in bit position 0 of CWA_0 maps to a "1" in position $((0^*16 + 0)^*2^1)$ MOD $16 = 0$ of $newCWA_0$ because $s = ((0^*16 + 0)^*2^1)$ DIV $16 = 0$. On the

Function update_cnha_cwa_1 (p_i, *hop*, PE_q, S_q) :
Input: The updatable address set U_q and the affected segment S_q containing the next hop information; For the inserted (deleted) p_i, let *hop*=h_i (*hop*=h_j, where p_j=*LMP*(p_i))
Output: modified *S*[*q*]
1. *range*=0, *port* = next hop in *S*[*q*], and *RLE*[*q*]=ϕ
2. U_q = **gen_updatable** (EL_q, p_i)
3. **if** *hop* ≠ *port* and $U_q \neq \phi$ **then**
4. **for** each pair ($start_j$, end_j) ∈ U_q **do**
   if ($start_j$ > *range*) **then** insert <*range*, $start_j$-1, *port*> into *RLE*[*q*] at the back
   insert <$start_j$, end_j, *hop*> into *RLE*[*q*] at the back
   range = end_j + 1
5. **if** (*range* < 2^{16}) **then** insert <*range*, 2^{16}-1, *port*> into *RLE*[*q*] //at the back
6. **if** (| *RLE*[*q*] | = 1) **then** *S*[*q*] = h_0
7. **else** *S*[*q*] = **CNHA/CWA_from_RLE** (*RLE*[*q*])

Fig. 21. Function **update_CNHA_CWA_1**.

Function `update_cnha_cwa_2` (p_i, *hop*, PE_q, S_q) :
Input: The updatable address set U_q and the affected segment S_q containing a pointer to *CNHA* and *CWA*; For the inserted (deleted) p_i, let *hop*=h_i (*hop*=h_j, where p_j=*LMP*(p_i))
Output: modified *S*[*q*]

1. U_q = **`gen_updatable`**(EL_q, p_i)
2. Case *insertion*:
   if $|p_i| > l$ **then**
   CWA=*resize_CWA*(S_q, $|p_i|$-*l*) // a positive resized factor
3. b_3=(end_0+1) DIV 2^{32-l}; t_e=b_3 DIV 16; w_e= b_3 MOD 16
   for (*i*=0; *i*≤ $|U_q|$-1; *i*++) **do**
   a. b_1=($start_i$ DIV 2^{32-l}); t_s=b_1 DIV 16; w_s= b_1 MOD 16; *f*=*CWA*[t_s].*base*+$|w_s|$-1
   b. **if** *CNHA*[*f*]=*hop* **then** do nothing //no hop update
   c. **elseif**(*CWA*[t_s].map_{ws}=1) && (*CWA*[t_e].map_{we}=1) **then** //case1
   if (*CNHA*[*f*-1]≠*hop*) && (*CNHA*[*f*+1]≠*hop*) **then**
   CNHA[*f*]=*hop* //no compression
   else //further compress CNHA
   delete *CNHA*[*f*]
   if (*CNHA*[*f*-1]=*hop*) **then**
   CWA[t_s].map_{ws}=0
   if(*CNHA*[*f*+1]=*hop*) **then** delete *CNHA*[*f*+1];
   CWA[t_e].map_{we}=0
   else if (*CNHA*[*f*+1]=*hop*) **then**
   CWA[t_e].map_{we}=0
   d. **elseif**(*CWA*[t_s].map_{ws}=1) && (*CWA*[t_e].map_{we}≠1) **then** //case 2
   CWA[t_e].map_{we}=1 // for non-updated Y
   if (*CNHA*[*f*-1]=*hop*) **then** *CWA*[t_s].map_{ws}=0
   else split *CNHA*[*f*] into: *CNHA*[f_1], *CNHA*[f_2]
   CNHA[f_1]=*hop* //update next hop
   e. **elseif** (*CWA*[t_s].map_{ws}≠1) && (*CWA*[t_e].map_{we}=1) **then** //case 3
   CWA[t_s].map_{ws} = 1
   if (*CNHA*[*f*+1]=*hop*) **then** *CWA*[t_e].map_{we} = 0
   else split *CNHA*[*f*] into: *CNHA*[f_1], *CNHA*[f_2]
   CNHA[f_2]=*hop*
   f. **elseif** (*CWA*[t_s].map_{ws}≠1) && (*CWA*[t_e].map_{we}≠1) **then** //case 4
   CWA[t_s].map_{ws} = 1
   CWA[t_e].map_{we} = 1
   split *CNHA*[*f*] into:*CNHA*[f_1],*CNHA*[f_2],*CNHA*[f_3]
   CNHA[f_2] = *hop*
   g. **if** (*i*+1≤ $|U_q|$-1) **then** //get the next t_e and w_e
   b_3=(end_i+1) DIV 2^{32-l};nt_e=b_3 DIV 16;nw_e=b_3 MOD 16
   for (*j*=t_s to nt_e) **do** //update the affected *CWA bases*
   CWA[*j*+1].*base*= |*CWA*[*j*].map_{nwe}|+*CWA*[*j*].*base*
   t_e = nt_e; w_e = nw_e
   h. **else** //the last range in U_q
   for (*j*=t_s to $|CWA|$-1) **do**
   CWA[*j*+1].*base*= $|CWA[j].map|$ + *CWA*[*j*].*base*
4. **if** ($|CNHA|$=1) **then** //*CNHA* contains only 1 hop
   S_q =*CNHA*[0]; Delete *CNHA* and *CWA*
5. **else** //Case *deletion* only
   if $|p_i|$>*l* **then** // *l* is a new value after deletion
   CWA=*resize_CWA*(S_q, *l*-$|p_i|$) //negative resized factor

Fig. 22. Function **`update_CNHA_CWA_2`**.

other hand, a "1" in bit position 9 of CWA_0 maps to a "1" in position $((0^*16+9)^*2^1)$ MOD $16=2$ of $newCWA_1$ because $s=((0^*16+9)^*2^1)$ DIV $16=1$. Using this mapping formula, the *CWA* in Fig. 4 is converted into equivalent $CWA_0: map=1000000010000010$, $base=0$, and $CWA_1: map=1010000000000010$, $base=3$.

Function `resize_CWA`(S_q, *b*) :
Input: the affected segment S_q and the resized factor *b*
Output: *newCWA*

1. Set the followings: $newCWA_j$=0 for *j*=0,1,2,..., $\lceil 2^{l+b}/16 \rceil$-1
   nbase=0
2. **for** *i*=0 to $|CWA|$-1 **do**
   j=*i**16
   for each ($CWA_i.map_d$=1) **do** // 0≤*d*≤15
   loc = (*j*+*d*) * 2^b; *s*=*loc* DIV 16
   w=*loc* MOD 16; $newCWA_s.map_w$=1
   $newCWA_s$.*base*=$newCWA_s$.*base*+1
3. **for** *j*=0 to $|newCWA|$-1 **do** //update the CWA bases.
   x= $newCWA_j$.*base*; $newCWA_j$.*base* = *nbase*
   nbase= *nbase*+*x*

Fig. 23. Function **`resize_CWA`**.

For case deletion, we remove p_i from ST_a and recalculate the segment's *l*. If the new $l < |p_i|$, then the *CWA* needs to be converted into a *newCWA* of size $\lceil 2^{l-16}/16 \rceil$ and a negative $b = l - |p_i|$ is used for shrinking the *CWA*. Note that the *CWA* is resized in Step 5 after we perform a modification to the *CNHA* and *CWA* in Steps 3 and 4. The function in Fig. 23 resizes a *CWA*. Let $b = |p_i| - l (b = l - |p_i|)$ for case insertion (deletion). The complexity of function **`resize_CWA`** depends on the number of *maps*/*bases* in *CWA* ($= \lceil 2^{l-16}/16 \rceil \leq 4096$) and the number of bits in Step 2 ($= O(m)$) and, so, its time complexity is $O(m)$.

Step 3 of **`update_CNHA_CWA_2`** performs the partial *CNHA* expansions, content updates, and recompression and modifies the affected bits in its *CWA*. Note that the IP addresses with next hop Y in a *CNHA* are denoted by a sequence of bits in its *CBM* starting from a "1" in bit position c_1 and ending at a "0" at bit position c_2. As an example, the $CNHA[1] = A$ in Fig. 4 is the next hop of addresses represented by bits 100 in positions $c_1 = 4$ to $c_2 = 6$. For each pair $(start_i, end_i)$ of U_q, we can obtain the starting (ending) bit $b_1 = start_i$ DIV 2^{32-l}($b_2 = end_i$ DIV 2^{32-l}) in the *CBM* that is affected by the update. As illustrated in Fig. 24, we consider four possible cases for expanding the *CNHA*. In case 1, because $b_1 = c_1$, and $b_2 = c_2$, we need not expand the *CNHA*: The next hop Y can be directly updated. For cases 2 and 3, we need the two next hops for the address ranges represented by the bit span between c_1 and c_2 because the nonupdateable addresses require Y as their next hop. Therefore, for case 2 (case 3), a new slot in *CNHA* (empty slot in Fig. 24) is created before (after) Y to store the next hop of the updateable addresses. Finally, in case 4, the first (second) Y is used for representing the next hop of the nonupdateable addresses.

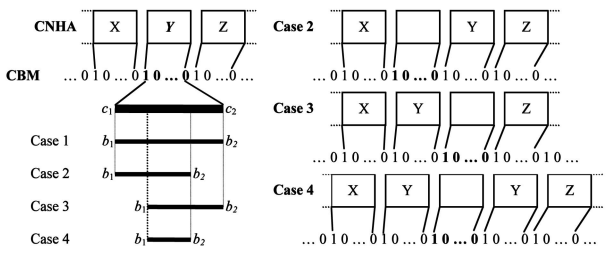

Fig. 24. Four cases of partial *CNHA* expansion.

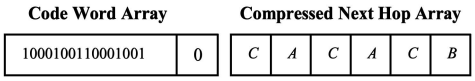

Fig. 25. Updated CNHA/CWA of Fig. 4.

For cases 2 and 3, if the new next hop is the same as that of its neighbor's, the two elements need to be merged. In other words, for this scenario, cases 2 and 3 do not require *CNHA* expansion. Therefore, for these two cases, we check for such possible recompression to avoid unnecessary complexity of the *CNHA* partial expansion. However, in all cases, the contents of *CWA* (*maps* and *bases*) need to be modified to reflect the updated next-hop representation in the *CNHA*. Step 3g in Fig. 22 updates the contents of the *CWA bases*. Note that the bit position $b_1(b_2)$ in a *CBM* is equivalent to bit position $w_1 = b_1$ MOD 16 in map $t_1 = b_1$ DIV 16 ($w_2 = b_2$ MOD 16 in map $t_2 = b_2$ DIV 16) in its corresponding *CWA*. Further, bit position $w_3 = b_2 + 1$ MOD 16 in map $t_3 = b_2 + 1$ DIV 16 represents a bit "1" for the *next* next hop in the *CNHA* (for example, Z in Fig. 24).

Steps 1, 2, and 5 of **`update_CNHA_CWA_2`** each are computable in $O(m)$ and the *for loop* in Step 3 is repeated $|U_q| \leq m$ times. The *for loops* in Steps 3g and 3h, in total, are repeated $|CWA| = \lceil 2^{l-16}/16 \rceil \leq 4,096$ times and, thus, the complexity of the function is $O(m + m + m + |CWA| + m) = O(m)$. In Fig. 20, Step 2 is executed $|PE| = \lceil 2^{16-|p_i|} \rceil \leq 256$ times and, therefore, the computational complexity of our online prefix update for CNHA/CWA is $O(m^*|PE|) = O(m)$.

To illustrate our online update, consider the CNHA/CWA structure in Fig. 4 and a pair $(p_i = 110010000001101110^*, A)$. The previous example for DFEC obtained $U_{200.27} = (\{36864, 49151\})$. With $l = 20$, $w_s = (36864 \text{ DIV } 2^{32-20})$ MOD $16 = 9$, $t_s = 0$, $f = 0 + 5 - 1 = 4$, $w_e = ((49151 + 1) \text{ DIV } 2^{32-20})$ MOD $16 = 12$, and $t_e = 0$. Because $CNHA[4] = C \neq hop(= A)$ and, in Step 3c, bit position $w_s = 9(w_e = 12)$ in map_0 is "1" ("0"), the prefix insertion falls into case 2. Since $CNHA[4-1]$ contains A, the update affects only *CWA* and, thus, bit position $w_s = 9(w_e = 12)$ in map_0 is set to "0" ("1"). Fig. 25 shows the result of updating the CNHA/CWA structure of Fig. 4.

5 Experimental Results

5.1 Environment and Databases for Test Data

We have implemented our algorithms in ANSI C and run them on a 3.2 GHz Pentium IV computer with 1 Mbyte cache and 1 Gbyte RAM. To evaluate their performances, we used seven databases: AADS (2001), Mae-West (2001), Mae-East (1997), Paix (2001), Paix (2000), PB (2001), and PB (2000). Table 1 shows the total number of prefixes (#prefix), the prefix length distribution, the total number of ports in each database, and the size of ST for each routing table T. Although each prefix p_i, with $|p_i| < 16$, is represented in $2^{16-|p_i|}$ segments of ST, most of these prefixes are stored directly in the segment and, therefore, $\#sp_j < \#\text{prefix}$, where $\#sp_j$ indicates the number of subprefixes in table ST. A 1-byte variable is used for h_i in each triple (sp_i, l_i, h_i) of ST and, therefore, our system supports databases with up to 255 ports.

5.2 Experiments for the FEC Scheme

5.2.1 FEC Table Construction Time

Table 2 shows the size of table F (α_r and β_c) for each database. The memory usage of the FEC table was calculated by taking the sum of the memory requirements for arrays $R(= 2^{16} * 2)$ and $C(= 2^{16} * 2)$ and table F, which is calculated as $\alpha_r * \beta_c * 1$ byte. Because each database contains less than 11 percent of prefixes with a length of at most 16 bits (Table 1), fixing $r = c = 16$ minimizes the size of the FEC table. The IP lookup time for each forwarding table is obtained by taking the average of lookup time of 1,000,000 randomly generated IP addresses. We noticed that the average lookup time on each database is slightly different, although the FEC scheme requires exactly three MAs per lookup, We suspect that these differences are caused by the different numbers of cache misses that occurred among the tables. Table 2 also shows the FEC table construction time by using the technique in [2] ($t_{[2]}$) and our technique (t_{total}). For $t_{[2]}$, we ran the source code in [2]. Our algorithm constructed exactly the same tables as those obtained by the technique in [2], but 2.56 to 7.74 times faster (column ρ). Column t_{RLE} (t_{RSE}) shows the runtime of our **`RLEGen (RSE)`**, where $t_{total} = t_{RLE} + t_{RSE}$.

5.2.2 Online Prefix Update Time on DFEC

To measure the performance of our DFEC scheme, we generated a random permutation of the prefixes for each

TABLE 1
Databases for Test Data

Test Data (Table *T*)							Table *ST*			
No	Database	#prefix	Prefix Length (*l*) Distribution			#port	Size (KB)	#subprefix	Max|*ST*[*q*]|	Average l_i
			$l \leq 16$	$16 < l \leq 24$	$l > 24$					
1	AADS	31827	2768	28500	559	29	372.94	29936	112	22.39
2	Mae West	28889	2521	26352	16	28	362.96	27381	105	22.22
3	Mae East	53345	5567	47687	91	63	451.57	50067	172	21.65
4	Paix 2001	16172	1264	14338	570	27	316.15	15399	110	22.18
5	Paix 2000	85987	7203	78725	60	39	576.61	82076	230	21.90
6	PB 2001	22225	1880	20324	21	1	337.98	20987	84	22.40
7	PB 2000	35302	2876	32372	55	120	387.89	33763	124	21.61

TABLE 2
The FEC Table Size, Memory Requirement, Construction Time, and Lookup Time

No	Table Construction Time (ms)					Lookup Time(ns)	Table F		FEC Table (KB)	DFEC Table (KB)
	$t_{[2]}$	Our Algorithm			ρ		α_r	β_c		
		t_{RLE}	t_{RSE}	t_{total}						
1	305	12	50	62	4.92	41	3173	628	2,201.9	2,595.93
2	186	12	24	36	5.17	34	3017	271	1,054.4	1,436.14
3	252	15	29	44	5.75	41	3198	326	1,274.1	1,745.70
4	294	9	30	39	7.54	34	2180	629	1,595.1	1,926.46
5	281	20	85	105	2.68	47	5065	338	1,927.9	2,535.45
6	188	16	18	34	5.53	31	2058	259	776.5	1,127.58
7	197	13	42	55	3.58	43	3947	334	1,543.4	1,955.72

database. For prefix insertion (deletion), we built the tables F, R, and C from the first 70 percent (all) entries of each database, inserted (deleted) the remaining (the last) 30 percent by using our online update technique, and measured the average insertion (deletion) time.

Table 3 shows the average insertion (deletion) time t_i (t_d) for inserting (deleting) 30 percent of the prefixes in each database. Note that #prefix shows the number of inserted or deleted prefixes. For each insertion or deletion, tables F, R, and C are recompressed and, hence, the resulting table F is expected to be the same as that obtained by the offline structure reconstruction. To verify the correctness of our online insertion (deletion) technique, we compared the resulting table from inserting (deleting) the 30 percent of prefixes of each database with that constructed by the method in [2] by using all (the first 70 percent) prefixes. As shown in Table 3, the average prefix update time is at most 10.1 μs, which shows the efficiency of our online prefix update technique while maintaining the scheme's fast lookup time of three MAs.

The table also presents the total number of row and column expansions for both insertion and deletion cases. As shown, each insertion or deletion requires, on average, less than one row and one column expansion. Notice that the number of column expansion is far less than that of row expansion. We observed that the execution time for each column expansion/recompression is significantly more than that for row. This fact is due to the use of row-based array to implement table F and, thus, each column expansion/recompression requires $O(\alpha_r)$ MA.

As described in Section 4.4, DFEC needs additional memory space for its arrays *row_hash*, *row_degree*, and *column_degree*. Comparing the memory requirements of FEC and DFEC (Table 2), we observed that the latter requirement is at most 600 Kbytes larger, with the benefit of enabling FEC for online prefix update.

5.3 Experiment for the CNHA/CWA Scheme

5.3.1 The CNHA/CWA Construction Time

Table 4 compares the performances of our proposed CNHA/CWA structure construction technique and the method in [5]. For this simulation, we have corrected the inconsistencies of the technique in [5] by including an additional sorting step, as described in Section 2.2.2. Our technique runs 4.57 to 6 times faster than that in [5] (column $\rho = t_{[5]}/t_{\text{ours}}$). The last column of the table shows our software simulations for IP lookup time on the CNHA/CWA structure for each database. The time is obtained by taking the average lookup time of 1,000,000 randomly generated IP addresses for each forwarding table. We needed two table references to compute $|w|$ for each IP lookup and, hence, each lookup in our experiment required at most five MAs. Comparing Tables 2 and 4 in terms of IP lookup, the software implementation of the FEC scheme outperforms that of CHNA/CWA because the former (latter) requires exactly three (at most five) MAs per IP lookup. However, the FEC scheme requires significantly larger memory than that needed by the CHNA/CWA scheme. Note that the memory requirements for the CHNA/CWA scheme depend on the length of the longest prefix in a segment (the last column in Table 1) and the total number of CNHA/CWA in the structure (#CNHA in Table 4).

TABLE 3
Average Insertion and Deletion Time on DFEC

No	#prefix	Insertion			Deletion		
		t_i (μs)	Expansion		t_d (μs)	Expansion	
			#*row*	#*col*		#*row*	#*col*
1	9548	9.84	2324	119	9.01	2176	87
2	8666	2.65	2310	4	2.65	2202	1
3	16003	3.19	2774	23	3.19	2644	7
4	4851	10.10	1174	138	9.69	1060	125
5	25796	3.68	2208	27	3.56	1923	2
6	6667	3.00	2517	2	2.85	2567	10
7	10590	4.06	1109	19	3.68	789	0

TABLE 4
The CNHA/CWA Construction Time

No	CNHA/CWA size		Construction Time (ms)			Lookup Time (ns)
	Memory (KB)	#CNHA	$t_{[5]}$	t_{ours}	ρ	
1	566.11	3668	71	12	5.92	49
2	467.94	3574	66	12	5.50	47
3	532.02	3432	79	16	4.94	56
4	493.66	2602	32	7	4.57	41
5	679.98	5105	103	22	4.68	66
6	423.58	2960	54	9	6.00	43
7	516.52	3861	61	12	5.08	52

TABLE 5
Route Prefix Insertion/Deletion Time on the CNHA/CWA Scheme

No	#Prefix Updates	Case Insertion					Case Deletion				
		On-Line			Off-Line (μs)		On-Line			Off-Line (μs)	
		#segment	#CNHA	$t_{\text{on-line}}$ (μs)	$t_{[5]}$	t_{ours}	#segment	#CNHA	$t_{\text{on-line}}$ (μs)	$t_{[5]}$	t_{ours}
1	9548	2763	7966	1.36	16.55	4.08	2694	8035	1.47	15.4	4.40
2	8666	2571	7031	1.15	15.35	3.58	2454	7148	1.27	14.88	3.46
3	16003	4152	13837	1.12	22.06	5.12	3966	14023	1.37	21.56	6.81
4	4851	1262	3985	1.65	15.66	4.12	1081	4166	1.65	14.02	4.33
5	25796	3517	24068	1.28	26.27	8.57	3255	24330	1.55	25.31	10.50
6	6667	1882	5139	1.20	14.85	3.15	1916	5105	1.20	14.25	3.30
7	10590	1892	9662	1.42	19.26	5.48	1570	9984	1.51	18.13	6.51

5.3.2 Online Prefix Update Time for the CNHA/CWA Scheme

Table 5 shows the comparison between the offline and online updates on CNHA/CWA. For insertion (deletion), we constructed CNHA/CWA from the first 70 percent (all) prefixes of each database, inserted (deleted) the remaining (last) 30 percent, and measured the average insertion (deletion) time. To verify the correctness of our insertion (deletion) technique, we compared the resulting table from inserting (deleting) the 30 percent of prefixes of each database with that constructed by using 100 (70) percent of prefixes. For offline update, we used both our implementation of the algorithm in [5] and our approach to reconstructing the affected segments (see columns $t_{[5]}$ and t_{ours}).

As shown in the table, our offline technique runs two to five times faster than the method in [5]. The table also shows that our online approach is faster than either offline method. Comparing t_{online} and $t_{[5]}(t_{\text{ours}})$, the online approach is 9.50 to 20.54 (2.62 to 6.77) times faster than the offline technique in [5] (our offline method). Note that #segments (#CNHA) in the table shows the total number of updates that are done on the affected segments (CNHA/CWA).

5.4 Comparison with the Existing Techniques

This section compares the lookup time, memory requirement, and update speed of DFEC and CNHA/CWA with those of BART [23], three recently proposed $O(log_2|T|)$ structures (PST [9], CRBT [18], ACRBT [19]), and multibit tries [17], [21] schemes. We use both worst-case and average-case lookup and update times to show the efficiency of the IP lookup schemes. We consider both lookup time measures, whereas, for update time, we consider only the average case because such data for the methods in [9], [17], [18], [19] are readily available.

The compression scheme in [23] optimizes the memory requirement of the BART data structure and, at the same time, allows the structure to be incrementally updated. It has been reported in [23] that a Paix with 72,825 prefixes fits in 555 Kbytes on the six-segment BART, requiring only 7.9 bytes/prefix. A c-segment BART requires $c + 2$ MAs per lookup. However, as presented in [23, Fig. 14], for the same database, a three-segment BART (partition 16 8 8) requires approximately 31.64 bytes/prefix. Both DFEC and CNHA/CWA are faster than the three-segment BART, which needs $3 + 2 = 5$ MAs, and require less memory (8.1 and 30.19 bytes/prefix, respectively, for Paix with 85,987 prefixes). Thus, we may conclude that DFEC and CNHA/CWA are better than BART [23] in lookup time and memory requirements. However, the worst-case incremental update in BART [23] is faster than that in either DFEC or CNHA/CWA.

From [9], the average lookup, insertion, and deletion times of PST for database 5 (7) in Table 1 (henceforth, we refer to the database as Paix (PB)) are 1.97, 3.07, and 2.91 (1.70, 2.80, and 2.55) μs, respectively, with 4,702 (1,930) Kbytes of memory. Note that, in [9], PST is shown to be superior to ACRBT [19] and, in [19], ACRBT is shown to be better than CRBT [18] in terms of the above performance measures. In comparison to these, our DFEC (CNHA/CWA) requires 0.047, 7.02, and 9.81 (0.066, 3.95, and 4.61), respectively, for Paix and needs 0.043, 7.18, and 10.29 (0.052, 3.12, 3.4) μs, respectively, for PB. Note that either DFEC or CNHA/CWA requires less memory than PST (see Tables 2 and 4). (The platform uses Pentium III with slightly faster speed (935.5 MHz versus 700 MHz in [9]), but has the same cache size.) Therefore, we may conclude that DFEC and CNHA/CWA are better (competitive) than the three $O(log_2|T|)$ structures (PST, CRBT, and ACRBT) in terms of the average lookup time and memory requirement (update times).

The FST and VST [21] can be implemented with varying levels k such that each lookup time can be performed in k MA: Smaller k requires larger memory. A technique in [21], which is improved in [17], is used for optimizing memory requirement for a selected k. The worst-case lookup time for FST (VST) for $k = 3$, as reported in [21], is 3 MAs + 31 clock cycles (3 MAs + 35 clock cycles), whereas FEC requires 3 MAs + 3 clock cycles [2]. Both [2] and [21] have used VTune in their measurements. On the other hand, CNHA/CWA requires one or three MAs (required clock cycles were not reported). Thus, we may conclude that, for the worst-case lookup time, FEC is the fastest and we conjecture that the speed of CNHA/CWA is comparable to that of FST and VST.

Ruiz-Sanchez et al. [15] have shown that the lookup time on FEC is faster than that on multibit trie. Further, [17, Table 4] shows that the average lookup time of VST on Paix (PB) is 0.71 (0.64) μs, which is significantly slower than those required in DFEC, 0.047 (0.043) μs, or CNHA/CWA, 0.066 (0.052) μs (see Tables 2 and 4). Note that we have obtained these results by using a slightly faster Pentium IV machine than that used in [17] (that is, 3.2 versus 2.26 GHz). Nevertheless, we may

conclude that both DFEC and CNHA/CWA outperform VST and FST in terms of the average lookup time. FST is only slightly faster than VST [21].

The best three-level memory requirements of FST (VST), as reported in [17, Table] ([16, Table 2]), for Paix and PB are 3,030 and 2,328 Kbytes (1,080 and 677 Kbytes), respectively. In comparison to these, Table 4 shows that CNHA/CWA requires significantly less memory than either FST or VST and Table 2 of our paper shows that DFEC requires less (more) memory than that for FST (VST). Note that, for the purpose of insertion and deletion in either FST or VST, for each node X in the multibit trie, the algorithms in [21] maintain a corresponding 1-bit trie with the prefixes that are stored in X. We are not sure if the reported memory requirements in [16], [17] include this auxiliary structure. If not, their memory requirement will go up further.

Reference [17] has proposed three strategies for prefix updates in VST: OptVST, Batch1, and Batch2. OptVST keeps the best k-VST for the current set of prefixes, whereas the others compute the optimal VST periodically. The batch updates are reported faster than OptVST; however, a batch insertion may increase the value of k [17]. Since our techniques do not increase the MA times, we consider only the OptVST. In terms of the insertion (deletion) time, OptVST needs 325.95 and 71.25 (61.29 and 60.77) μs for Paix and PB, respectively, in contrast to 3.68 and 4.06 (3.56 and 3.68) μs for DFEC, and 1.28 and 1.42 (1.55 and 1.51) μs for CNHA/CWA. Thus, we may conclude that both DFEC and CNHA/CWA are superior to VST in the average update times: reference [17] does not provide the average update times for FST.

6 Conclusion and Future Work

We have proposed the use of decreasing lexicographic ordered prefixes to reduce the construction time of the FEC [2] and CNHA/CWA [5] structures. We have used the prefixes to construct RLE sequences, which are used for building FEC and CNHA/CWA. Our column-based RSE technique, in contrast to the row-based one in [2], further reduces the constructing time of FEC. Simulations on real routing tables show that our approach constructs FEC tables 2.68 to 7.54 times faster than that in [2] and it constructs CNHA/CWA tables 4.57 to 6 times faster than using the algorithm in [5]. The properties of the decreasing lexicographic prefixes can also be used for reducing the construction time of other existing schemes, such as the disjoint multibit trie [21].

Compressed-based IP lookup schemes provide fast lookup times, with a trade-off for slow prefix update time [15]. Therefore, the schemes were typically not for use as dynamic routers [1], [15] and offline data structure reconstruction was assumed after some prefix updates on the routers. In contrast to those results, we have used the *updatable address set* concept to enable the compressed-based schemes, FEC [2] and CNHA/CWA [5], for online prefix updates. Our simulations show that the average prefix update time, by using our techniques, is at most 10.1 (1.65) μs for FEC (CNHA/CWA) while maintaining its three (one or three) MA lookup time. A similar approach can also be employed to enable other compressed-based schemes for online prefix updates.

Acknowledgments

The authors thank the anonymous referees for their valuable comments to improve their paper. The authors are grateful to L. Dardini, who made the source code for the FEC scheme available, and S. Sahni and H. Lu for providing them with the various routing table databases. Dr. Rai is supported in part by the US National Science Foundation Grant CCR0310916.

References

[1] R.C. Chang and B.-H. Lim, "Efficient IP Routing Table VLSI Design for Multigigabit Routers," *IEEE Trans. Circuits and Systems,* vol. 51, no. 4, pp. 700-708, Apr. 2004.

[2] P. Crescenzi, L. Dardini, and R. Grossi, "IP Address Lookup Made Fast and Simple," *Proc. Seventh Ann. European Symp. Algorithms,* Technical Report TR-99-01, Univ. of Pisa, 1999.

[3] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small Forwarding Tables for Fast Routing Lookups," *Proc. ACM SIGCOMM '97,* pp. 3-14, 1997.

[4] L. Hiryanto, S. Soh, S. Rai, and R.P. Gopalan, "Fast IP Table Lookup Construction Using Lexicographic Prefix Ordering," *Proc. 11th IEEE Asia-Pacific Conf. Comm. (APCC '05),* 2005.

[5] N.-F. Huang and S.-M. Zhao, "A Novel IP-Routing Lookup Scheme and Hardware Architecture for Multigigabit Switching Routers," *IEEE J. Selected Areas in Comm.,* vol. 17, no. 6, pp. 1093-1104, June 1999.

[6] C. Labovitz, G.R. Malan, and F. Jahanian, "Internet Routing Instability," *IEEE/ACM Trans. Networking,* vol. 6, no. 5, pp. 515-528, 1998.

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

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

[9] H. Lu and S. Sahni, "$O(\log n)$ Dynamic Router-Tables for Prefixes and Ranges," *IEEE Trans. Computers,* vol. 53, no. 10, pp. 1217-1230, Oct. 2004.

[10] H. Lu and S. Sahni, "Enhanced Interval Trees for Dynamic IP Router-Tables," *IEEE Trans. Computers,* vol. 53, no. 12, pp. 1615-1628, Dec. 2004.

[11] H. Lu, K.S. Kim, and S. Sahni, "Prefix and Interval-Partitioned Dynamic IP Router-Tables," *IEEE Trans. Computers,* vol. 54, no. 5, pp. 545-557, May 2005.

[12] S. Nilsson and G. Karlsson, "Fast Address Lookup for Internet Routers," *IEEE J. Selected Areas in Comm.,* vol. 17, no. 6, pp. 1083-1092, June 1999.

[13] D. Pao and Y.-K. Lie, "Enabling Incremental Updates to LC-Trie for Efficient Management of IP Forwarding Tables," *IEEE Comm. Letters,* vol. 7, pp. 245-247, May 2003.

[14] V.C. Ravikumar, R. Mahapatra, and J.C. Liu, "Modified LC-Trie Based Efficient Routing Lookup," *Proc. 10th IEEE Int'l Symp. Modeling, Analysis, and Simulation of Computer and Telecomm. Systems,* pp. 1-6, 2002.

[15] M.A. Ruiz-Sanchez, E.W. Biersack, and W. Dabbous, "Survey and Taxonomy of IP Address Lookup Algorithms," *IEEE Network,* vol. 15, pp. 8-23, Mar.-Apr. 2001.

[16] S. Sahni and K.S. Kim, "Efficient Construction of Variable-Stride Multibit Tries for IP Lookup," *Proc. IEEE Symp. Applications and the Internet,* pp. 220-229, 2002.

[17] S. Sahni and K.S. Kim, "Efficient Construction of Multibit Tries for IP Lookup," *IEEE/ACM Trans. Networking,* vol. 11, no. 4, pp. 650-662, Aug. 2003.

[18] S. Sahni and K.S. Kim, "An O(log n) Dynamic Router-Table Design," *IEEE Trans. Computers,* vol. 53, no. 3, pp. 351-363, Mar. 2004.

[19] S. Sahni and K.S. Kim, "Efficient Dynamic Lookup for Bursty Access Patterns," *Int'l J. Foundations of Computer Science,* vol. 15, no. 4, pp. 567-591, 2004.

[20] S. Soh, L. Hiryanto, S. Rai, and R.P. Gopalan, "Dynamic Router Tables for Full Expansion/Compression IP Lookup," *Proc. IEEE Tencon '05,* 2005.

[21] V. Srinivasan and G. Varghese, "Fast Address Lookups Using Controlled Prefix Expansion," *ACM Trans. Computer Systems,* vol. 17, no. 1, pp. 1-40, 1999.
[22] X. Sun and Y.Q. Zhao, "An On-Chip IP Address Lookup Algorithm," *IEEE Trans. Computers,* vol. 54, no. 7, pp. 873-885, July 2005.
[23] J. van Lunteren, "Searching Very Large Routing Tables in Fast SRAM," *Proc. Int'l Conf. Computational Nanoscience (ICCN '01),* pp. 4-11, 2001.
[24] P.C. Wang, C.T. Chan, and Y.C. Chen, "A Fast Table Update Scheme for High Performance IP Forwarding," *Proc. Eighth Int'l Conf. Parallel and Distributed Systems,* pp. 592-597, 2001.

Sieteng Soh received the BS degree in electrical engineering from the University of Wisconsin-Madison and the MS and PhD degrees in electrical engineering from Louisiana State University, Baton Rouge. From 1993 to 2000, he was a faculty member at Tarumanagara University, Indonesia, where he was the director of the Research Institute from 1998 to 2000. He is currently a lecturer with the Department of Computing at Curtin University of Technology, Perth, Western Australia. His research interests include network reliability, and parallel and distributed processing. He is a member of the IEEE and the IEEE Computer Society.

Lely Hiryanto received the BE degree in computer science from Tarumanagara University, Indonesia, and the MSc degree in computer science from the Curtin University of Technology, Perth, Western Australia. She is currently a lecturer with the Faculty of Information Technology at Tarumanagara University. Her research interests include routing, network programming, and distributed data processing.

Suresh Rai received the PhD degree in electronics and communication engineering from Kurukshetra University, Kurukshetra, Haryana, India, in 1980. He is currently a professor with the Department of Electrical and Computer Engineering at Louisiana State University, Baton Rouge. His research interests include network traffic, wavelet-based compression, and security. He is a senior member of the IEEE and a member of the IEEE Computer Society.