©2008 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

2005 Asia-Pacific Conference on Communications, Perth, Western Australia, 3 - 5 October 2005.

Fast IP Table Lookup Construction Using Lexicographic Prefix Ordering

Lely Hiryanto[†], Sieteng Soh[†], Suresh Rai[‡], Raj Gopalan[†]
[†]Department of Computing, Curtin University of Technology, Australia ({hiryanto,soh,raj}@computing.edu.au)
[‡]Department of Electrical and Computer Engineering, LSU, Baton Rouge, USA (suresh@ece.lsu.edu)

Abstract – **Crescenzi et.al. have proposed a fast Full-Expansion-Compression (FEC) technique that requires exactly three memory accesses for each IP lookup. However, prefix updates on their FEC scheme require the forwarding table data structure to be rebuilt from scratch. Therefore, fast table reconstruction is important to make the scheme effective for use in a dynamic environment. In this paper, we propose an efficient technique to reduce the FEC table construction time. Our approach generates the RLE sequences for the forwarding table directly from a list of decreasing lexicographic ordered prefixes to avoid the expansion step of the previous technique. We also propose an improved unification technique to construct the FEC tables more efficiently. Our approach has been implemented in C, and several experiments using six databases from some real IPv4 router show that our proposed technique runs 4 to 29 times faster than the previous technique.**

I. INTRODUCTION

The IP address lookup in a router decides the next hop to forward each incoming packet towards its destination, and it is still the remaining bottleneck among the major tasks of a router [1]-[16]. A router maintains a set of prefixes and as a part of its forwarding task the router has to quickly find the longest prefix that matches the m-bit destination address (m=32 bits in IPv4) of an incoming packet. Table I(a) shows an example of a set of prefixes with their corresponding next hops. In this example, a 4-bit destination address 0011 matches prefixes *, 00*, and 001*, and therefore the router should forward the packet to a next hop *C* since 001* is the longest matching prefix.

The ideal scheme for an IP lookup solution includes fast lookup time, fast prefix update time, small memory requirement, and good scalability [11]. Since the optimal solution to the IP lookup problem has been shown to be NP-hard [6], every existing IP lookup algorithm makes tradeoffs among those parameters. The most critical measure is obviously the worst-case lookup time since failure to meet the required time may result in a misclassified packet. Nevertheless, the update time, memory requirements, and scalability must also be considered [11]. Every time there is a route change, the contents of a forwarding table should be updated to reflect the route change. Small update time is essential, since a study conducted by Labovitz et. al [4] has shown that the Internet instability at backbone routers can cause several hundred prefix updates per second. With this rate, it is estimated that the update operations must be performed in 10 milliseconds or less.

TABLE I
SAMPLE OF ROUTING TABLE, *T*

Prefix	Nexthop
10*	*B*
00*	*A*
01*	*C*
001*	*B*
0010	*C*
*	*C*

(a) Unsorted

Prefix	Nexthop
10*	*B*
01*	*C*
0010*	*C*
001*	*B*
00*	*A*
*	*C*

(b) Sorted

References [1], [2], [5], [9], [15] present efficient IP lookup algorithms. However, prefix updates in these static schemes are very difficult to implement due the optimized structures used in them, and therefore rebuilding the structures may be necessary to deal with updates [11]. There are two different approaches to deal with prefix updates. In the first approach, update-efficient dynamic data structures [3], [7]-[8], [13]-[14] are used. In the second approach, update operations are batched and the tables are periodically reconstructed [1], [2], [11], [16]. This solution is acceptable if the table reconstruction can be done fairly fast. Degermark, et al. [2] argued that even though routing updates can be frequent, 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. Thus, each table reconstruction (off-line updates) can be done after some sequence of prefix updates. It is obvious, however, that the faster the time to reconstruct the table the better so that the table can represent more up-to-date network routing information.

Improved techniques to reduce construction time for forwarding tables are found in the literature [10], [12], [16]. Ravikumar *et al.* [10] proposed a new technique to reduce the construction time of the LC-trie [9]. Sahni and Kim [12] have reduced the construction time of the multibit trie [15]. Wang *et al.* [16] provided an improved routing table construction algorithm for the multiway search tree [5]. Among the existing IP lookup algorithms, Full Expansion Compression (FEC) scheme [1] results in the fastest lookup time [11]. The technique requires exactly three memory accesses for each IP lookup. However, a prefix update (insertion, deletion, or next hop alteration) on the scheme is expensive. To facilitate the technique for prefix updates, Crescenzie *et al.* [1] proposed to divide the computer cache into two banks: the first bank is

used for the lookup and the other is used for table update. For some typical backbone forwarding table, the algorithm [1] requires hundreds of milliseconds to build the FEC tables (see Table I, column $t_{[1]}$). Thus, a more efficient table construction technique is required so that the scheme can be used effectively in a dynamic environment. In this paper, we propose an improved technique to reduce the construction time of FEC router table. In Section II, we briefly describe the FEC scheme [1]. Section III presents our proposed technique. Section IV gives a set of experimental results to show the merits of our algorithm. Section V concludes this paper.

II. THE FEC TECHNIQUE – BACKGROUND

For a binary alphabet $\Sigma=\{0,1\}$, we denote a set of all binary strings of length k (at most m) by Σ_k ($\Sigma_{\leq m} = \bigcup_{k=0}^{m} \Sigma_k$). For two binary strings ,$\nu \in \Sigma_{\leq m}$ of length l = | | and $l_\nu = |\nu|$, respectively, we say that is a *prefix* of ν if the first $l \leq l_\nu$ bits of ν are equal to . Also, is the longest matching prefix (*LMP*) of ν if no other prefix of ν has larger cardinality than . Let $^{l}_{s}$ represent a substring of from bit s with length l, for $0 \leq s \leq |\ |$-1 and $l \leq |\ |$.

A routing table T contains a list of pairs $T_x =(x, h_x)$, where prefix $x \in \Sigma_{\leq m}$ and its *next hop* interface h_x. It is assumed that T always contains a pair $(\varepsilon, h_\varepsilon)$, where ε denotes an empty string. For a given routing table T, FEC technique [1] constructs the FEC tables in two phases: *expansion* and *compression*. From a sequence of decreasing lexicographic ordered prefixes, the expansion phase implicitly constructs a table T' which contains all the 2^{32} possible IPv4 addresses from T. As an example, Fig. 1(a) presents the expanded table T' of Table I(b) for m = 4 bits. In the compression phase, an optimal break bit value $1 \leq k \leq m$ is used to cluster the expanded prefixes. The clustering produces a set of RLE sequences based on the run-length-encoding (RLE) scheme [1]. As part of this step, any RLE sequence that contains exactly the same information as that in another sequence is deleted. Fig. 1(b) shows the RLE sequences $\langle s_1, s_2, s_3, s_4 \rangle$ of Fig. 1(a), in which s_2, s_4 are duplicate sequences. To build the FEC tables that comprise *row_index*, *col_index*, and *interface* tables, a *unification* step using a recursive *function* φ [1] is performed on each of the RLE sequences. Fig. 1(c) shows the results of this step, which in turn are used to construct the FEC data structures as shown in Fig. 1(d).

III. IMPROVED TECHNIQUE FOR FEC TABLE CONSTRUCTION

A. Disjoint Prefix Binary Trie

We propose to use a disjoint-prefix binary trie (DPBT) [11] to represent all pairs (X, h_X) of a table T'. In a DPBT each next hop is stored only in its leaves, while each intermediate node contains only pointers to its two descendants. Disjoint-prefixes do not overlap, and no address prefix is itself a prefix of another [11]. Fig. 2(a.iv) presents a DPBT for table T' of Fig. 1(a). A binary-path (a string of bits) to a leaf in a DPBT represents a pair (disjoint-prefix X, h_X) that can be expanded to generate a set of m-bit addresses, $X \cdot \Sigma_{m-|X|}$, each of which has a next hop h_X. We obtain T' by expanding each disjoint-prefix of a DPBT. As an example, expanding the binary-path to leaf A (=000) of the DPBT in Fig. 2(a.iv) we obtain addresses 0000 and 0001 in table T', each with next hop A.

B. Efficient Disjoint Prefix Binary Trie Construction

A DPBT is constructed by pushing each prefix at the internal node of a binary-trie down to the leaves of the trie [11]. Fig. 2(a) illustrates the construction of the DPBT of Table I(a). Fig. 2(a.i) shows the result of inserting prefixes 10*, 00*, and 01*. Fig. 2(a.ii) shows that inserting a prefix 001* causes the trie to be no longer a DPBT; there is an internal node A (with next hop information) that must be pushed down. We call a node that needs to be pushed down a *pseudo-leaf*. When a pseudo-leaf exists, the insertion process needs to backtrack to the node, creates new leaves down the trie, and stores the next hop of the pseudo leaf in each of them. Similarly, inserting a prefix 0010 into the trie as shown in Fig. 2(a.iii) creates a pseudo-leaf. It is obvious that backtracking and extra leaf-pushing steps make a DPBT construction process inefficient. The following property of a DPBT states the necessary condition for the existence of a pseudo-leaf.

Property 1. Inserting a prefix x into a DPBT creates a pseudo-leaf if there is a binary-path Y in DPBT as a prefix of x.

Using the following lemma we are able to build a DPBT without creating any pseudo-leaf. We also will use this lemma to construct the RLE-sequences more efficiently.

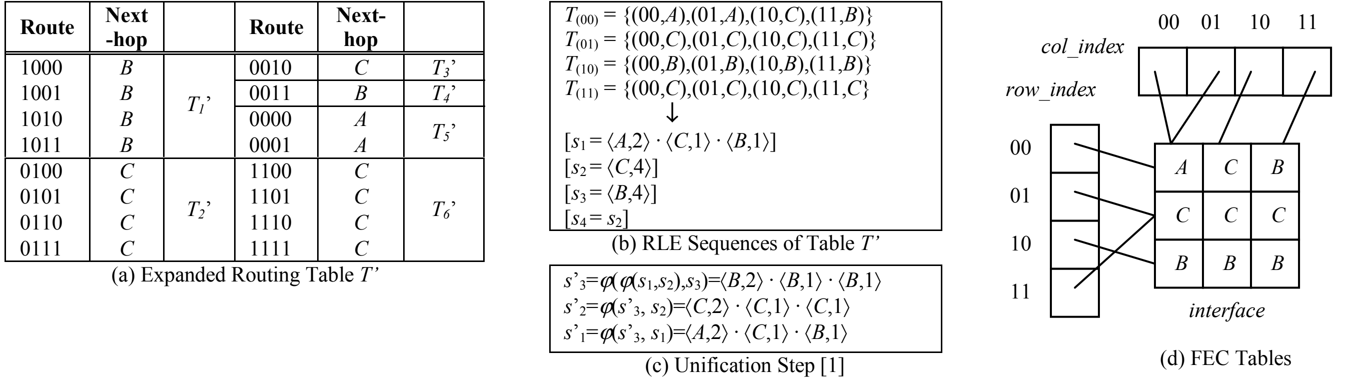
Lemma 1. Constructing a DPBT from a set of decreasing lexicographic ordered prefixes does not create pseudo-leaves.

Proof. Let $P_1, P_2, \ldots, P_{i-1}, P_i$ denote a sequence of prefixes sorted in decreasing lexicographic order. Assume $P_1, P_2, \ldots, P_{i-1}$ have been inserted into a DPBT, and let P_i be the next inserted prefix. By definition of a DPBT, a leaf in the trie does not have a binary-path which is a subset of $P_1, P_2, \ldots, P_{i-1}$. Since P_i cannot be a superset of any of $P_1, P_2, \ldots, P_{i-1}$, it cannot be a superset of any binary-path to the leaves in the trie. Thus, the insertion of prefix P_i will create no pseudo-leaf.

Fig. 2(b) illustrates the construction of a DPBT from the prefixes of Table I(b). Notice that no pseudo-leaf is generated.

C. RLE Sequence Generation from a DPBT

A set of RLE sequences can be represented by an array $R[0 .. 2^k-1]$ in which each element is a pointer to a sequence of RLE, $\langle h_x, l \rangle$, where h_x is the next-hop of a prefix x and l is the run length of h_x. In other words, each element of the array $R[i] = \langle h_1, l_1 \rangle \cdot \langle h_2, l_2 \rangle \cdot \ldots \cdot \langle h_n, l_n \rangle$, and $l_1 + l_2 + \ldots + l_n = 2^{m-k}$. Our representation of the RLE sequences is equivalent to that shown in Fig. 1(b). Following reference [1] and utilizing a constructed DPBT, the RLE sequences can be obtained using the following two steps. First, we generate a set of increasing lexicographic ordered disjoint-prefixes, which is straightforward by traversing the DPBT in preorder. Then, we generate the RLE sequences by using the Steps 2 to 3 of the following Algorithm-1.



Route	Next-hop		Route	Next-hop	
1000	*B*	T_1'	0010	*C*	T_3'
1001	*B*		0011	*B*	T_4'
1010	*B*		0000	*A*	T_5'
1011	*B*		0001	*A*	
0100	*C*	T_2'	1100	*C*	T_6'
0101	*C*		1101	*C*	
0110	*C*		1110	*C*	
0111	*C*		1111	*C*	

Figure 1. FEC table construction using FEC technique [1]

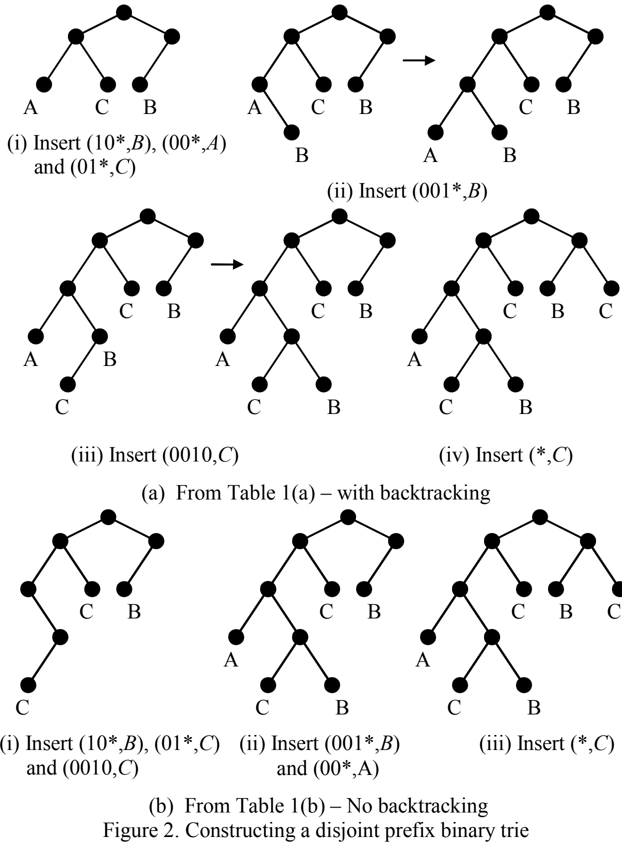


Figure 2. Constructing a disjoint prefix binary trie

Algorithm-1:

1. From the decreasing lexicographic ordered prefixes of a table T, construct a DPBT, and obtain increasing lexicographic ordered disjoint-prefixes – pairs (X, h_X).
2. For each pair (X, h_X):

   a. Generate one or more row addresses, $index_i$, as follows:

   $$index_i = \begin{cases} X \cdot \Sigma_{k-|X|} & ; if\ |X| < k \\ X_0^k & ; if\ |X| \geq k \end{cases}$$

   b. Generate an RLE $\langle h_X, l\rangle$, where l is obtained as follows:

   $$l = \begin{cases} 2^{m-|X|} & ; if\ |X| > k \\ 2^{m-k} & ; if\ |X| \leq k \end{cases}$$

   c. For each $r_i \in index_i$, insert $\langle h_X, l\rangle$ to an RLE sequence pointed by $R[r_i]$ using the following rules:

      i. if $R[r_i]$ points to an empty RLE sequence, $R[r_i] = \langle h_X, l\rangle$

      ii. if $R[r_i]$ points to a non-empty RLE sequence, insert $\langle h_X, l\rangle$ at the end of the sequence
3. For any RLE sequence $\langle h_j, l_j\rangle \cdot \langle h_{j+1}, l_{j+1}\rangle \cdot \ldots \cdot \langle h_n, l_n\rangle$ with $h_j = h_{j+1} = \ldots = h_n$, replace the sequence with a pair $\langle h_j, L\rangle$, where $L = l_j + l_{j+1} + \ldots + l_n$.

D. Efficient RLE Sequence Generation

In this section we present an efficient algorithm to construct RLE sequences directly from a set of decreasing lexicographic ordered prefixes. Notice that following Lemma 1, a binary-path of a prefix x will not be a prefix of any binary-path of a prefix y that is lexicographically less than x. This observation makes us able to directly generate the RLE sequences from the sorted prefixes. Here, we only need to implicitly generate disjoint-prefixes, and hence the construction of a DPBT is avoided. However, these implicitly enumerated disjoint prefixes are not necessarily in increasing lexicographic order, and therefore we need to modify Step 2c of Algorithm-1, and represent an RLE using a triple $\langle h_x,a,b\rangle$, where $0 \leq a,b \leq 2^{m-k}-1$ are respectively the binary value of the starting and ending column addresses of a prefix x with next hop h_x. As an example, we show the equivalent RLE sequences of Fig. 1(b) using this new notation in Fig. 3(b). The following Algorithm-2 generates the RLE sequences without explicitly constructing a DPBT.

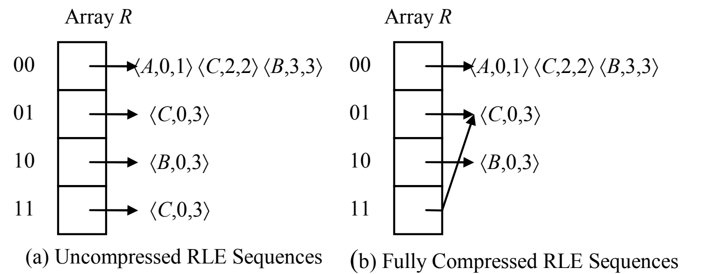


Figure 3. Generating RLE Sequences using Algorithm 2.

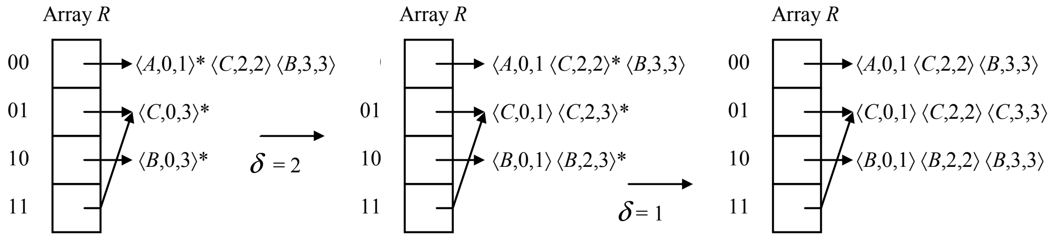


Figure 4. Sequence-Expansion of RLE Sequences for Figure 3(b)

Algorithm-2:

1. Sort pairs $T_x = (x, h_x)$ in decreasing lexicographic order.
2. For each pair (x, h_x):
   a. Generate one or more row addresses, $index_i$, as follows:
   $$index_i = \begin{cases} x \cdot \Sigma_{k-|x|} & ; if\ |x| < k \\ x_0^k & ; if\ |x| \geq k \end{cases}$$
   b. Generate an RLE $\langle h_x,a,b\rangle$, where a and b are the binary values of a' and b', respectively. a' and b' are obtained as follows (Σ^0_k represents a binary string with k 0s):
   $$a' = \begin{cases} \Sigma^0_{m-k} & ; if\ |x| \leq k \\ x_k^{|x|-k} \cdot \Sigma^0_{m-|x|} & ; if\ |x| > k \end{cases} \qquad b' = \begin{cases} \Sigma^1_{m-k} & ; if\ |x| \leq k \\ x_k^{|x|-k} \cdot \Sigma^1_{m-|x|} & ; if\ |x| > k \end{cases}$$
   c. For each $r_i \in index_i$, and for each $R[r_i]$ that is not full, i.e. $\sum_{i=1}^{n}(b_i - a_i + 1) < 2^{m-k}$, do each of the following (assume $R[r_i] = p_1 \cdot p_2 \cdot \ldots \cdot p_n$ and an RLE $p_j = \langle h_j,a_j,b_j\rangle$)
   (i) If $R[r_i]$ is empty, insert an RLE $\langle h_x,a,b\rangle$ to the row.
   (ii) Else, do each of the following steps:
      1. If $a_1 > a$, insert RLE $\langle h_x,a,\min(b,a_1-1)\rangle$ before p_1
      2. If $b_n < b$ insert RLE $\langle h_x, \max(b_n+1,a),b\rangle$ after p_n
      3. For each pair $p_j \cdot p_{j+1}$, where j = 1, 2, …, n-1 do
         If $b_j \neq a_{j+1}-1$, insert RLE $\langle h_x, \max(a,b_j+1), \min(b,a_{j+1}-1)\rangle$ between the pair.
3. Replace any RLE sequence $\langle h_j,a,b\rangle \cdot \langle h_{j+1},b+1,c\rangle \cdot \ldots \cdot \langle h_n,y+1,z\rangle$ where $h_j = h_{j+1} = \ldots = h_n$, with a triple $\langle h_j,a,z\rangle$.

Figure 3(a) illustrates Algorithm-2. Given the sorted prefix list in Table I(b), the algorithm generates an RLE from a pair (10*,B), This prefix has row address 10 and because R[10] is empty, Step 2c(i) creates $R[10] = \langle B,a,b\rangle$, where a=0 is the binary value of a string $10_2^{4-2}\cdot\Sigma^0_{4-|10|}$ = 00, and b=3 is the binary value of a string $10_2^{4-2}\cdot\Sigma^1_{4-|10|}$ = 11. Similarly, Step 2c(i) creates an RLE $\langle C,0,3\rangle$ ($\langle C,2,2\rangle$) from a pair (01*,C) ((0010*, C)) and insert it to R[01] (R[00]). Because R[00] is not empty, Step 2c(ii.2) is used on pair (001*,B) to obtain an RLE $\langle B,3,3\rangle$ and insert it at the end of the sequence. Similarly, Step 2c(ii.1) is used on a pair (00*,A) to create an RLE $\langle A,0,1\rangle$, and insert it as the first RLE in R[00]. Deleting the redundant RLE sequence, we obtain the compressed RLE sequences in Fig. 3(b) which is equivalent to that shown in Fig. 1(b).

E. An Improved Unification Technique

Crescenzi, *et al.* [1] use a function φ so that each of the RLE sequences contains the same number of RLE. In this unification step [1], a pair $\langle h_X, L\rangle$ may be split into $\langle h_X,l_1\rangle \cdot \langle h_X,l_2\rangle \cdot \ldots \cdot \langle h_X,l_n\rangle$ where $l_1 + l_2 + \ldots + l_n = L$ or equivalently, a triple $\langle h_x,a,b\rangle$ may be expanded into $\langle h_x,a,b_1\rangle \cdot \langle h_x,b_1+1,b_2\rangle \cdot \ldots \cdot \langle h_x,b_n+1,b\rangle$. The function φ in [1] performs a row-based splitting. Notice that our experimental results (Table II) show that typically an *interface* table has smaller column (β_k) than row (α_k), and therefore intuitively an equivalent unification method (proposed in the following) that does a column-based splitting is expected to be more efficient. We propose the RLE-sequence-expansion (RSE) algorithm as follows.

Algorithm RSE:

1. Flag the first RLE in each RLE sequence R[i], for $i \geq 0$.
2. Set δ = the minimum (b_i-a_i+1) value among those of the flagged RLE.
3. Split each RLE $\langle h_x,a,b\rangle$ that has $(b-a+1)>\delta$ into $\langle h_x,a,\delta+a-1\rangle \cdot \langle h_x,\delta+a,b\rangle$ that replace the original RLE $\langle h_x,a,b\rangle$, and flag the second RLE of the resulting split.
4. Unflag each RLE $\langle h_x,a,b\rangle$ that has $(b-a+1) = \delta$, and flag its next RLE in the sequence.
5. Repeat Step 2, until all flagged RLE have the same δ of (b_i-a_i+1), for all i.

Figure 4 illustrates the RSE algorithm to expand the RLE sequences shown in Fig. 3(b).

IV. Experimental Results

A. Environment and Test Data

We have implemented our algorithm in ANSI C and run it on a 3.2 GHz Pentium 4 computer with 1 MB cache and 1 GB RAM. To evaluate its performance, we used six real IPv4 databases: AADS (22/11/2001), Mae-West (22/11/2001), Mae-East (5/11/1997), OIX (30/10/2003), Paix (22/11/2001), and PB (22/11/2001). Table II (column #pref) shows the number of prefixes in each database. We fixed k = 16 bits to minimize the size of the *row_index* and *column_index* tables so that each table has exactly 65536 entries.

Table II shows the size of the constructed interface table (α_k and β_k) and the table construction time ($t_{[1]}$) using the technique in [1] for each database. The memory usage of each FEC table was calculated by taking the sum of the memory requirements for a *row_index* (2^{16} * 2 bytes; fixed-size), a *col_index* (2^{16} * 2 bytes; fixed-size), and an *interface* table which is calculated as α_k * β_k * 1 byte. The table also shows the average IP lookup time for each forwarding table. We observed that an IP lookup on a larger database requires longer time, and we noticed that a lookup on FEC table of size

522.9KB (fit into the available system cache memory) is significantly faster than on the other larger sized tables, although theoretically the FEC scheme requires exactly 3 memory accesses for each lookup. We suspect that this is caused by more cache misses on larger sized tables.

B. Performance of Our Algorithm

We use Algorithm-2 to construct an FEC table for each of the databases. Our algorithm constructed exactly the same tables as those obtained by the algorithm in [1] but in 4 to 29 times faster (column ρ of Table III). Note that each ρ value is obtained by dividing the running time of technique [1] (column $t_{[1]}$ in table II) with that of our algorithm (column $t_{Alg2+RSE}$ in table III). Our algorithm reduces the FEC table construction because (i) it generates RLE-sequences more efficiently, and (ii) it converts the RLE-sequences into FEC table faster. Column t_{Alg1} (t_{Alg2}) in Table III shows the running time of Algorithm-1 (Algorithm-2) that constructs a compressed RLE sequences for each database. Note that t_{Alg1} is comparable to that required by the technique in [1]. Algorithm-2 outperforms Algorithm-1 because it avoids DPBT construction (compare columns t_{Alg2} and t_{Alg1}). Column t_{RSE} shows the required time for our RSE algorithm to construct the FEC tables. The algorithm is faster than the unification step of [1]. We can show empirically the efficiency of RSE by considering Algorithm-1 to be as efficient as the corresponding step (to generate RLE-sequences) in [1]. Notice that the total of each row in t_{Alg1} and t_{RSE} (Table III) is less than the corresponding row of column $t_{[1]}$ in Table II, and thus it is obvious that the time saving is due to our improved column-based RSE technique.

TABLE II
FEC TABLE: SIZE, MEMORY REQUIREMENT, CONSTRUCTION TIME, AND LOOKUP TIME

Database	#pref	Size of Interface Table		Memory usage (KB)	$t_{[1]}$ (s)	IP lookup time (s)
		α_k	β_k			
AADS	31827	3173	628	2,201.9	304952	0.0410
Mae West	28889	3017	271	1,054.4	185971	0.0340
Mae East	53345	3198	326	1,274.1	252963	0.0410
OIX	1798	624	438	522.9	233967	0.0190
Paix	16172	2180	629	1,595.1	293959	0.0340
PB	22225	2058	259	776.5	187968	0.0310

TABLE III
PERFORMANCE OF OUR PROPOSED ALGORITHM

Database	Table Construction Time (s)				ρ
	t_{Alg1}	t_{Alg2}	t_{RSE}	$t_{Alg2+RSE}$	
AADS	57992	23996	49993	73989	4.12
Mae West	55991	21997	23996	45993	4.04
Mae East	72989	32995	28996	61991	4.08
OIX	6998	1000	6998	7998	29.25
Paix	37994	8999	29996	37994	7.74
PB	38994	25996	17998	43994	4.27

V. CONCLUSION

We have developed an approach to improve the construction time of FEC forwarding table. Using a decreasing lexicographic ordered list of prefixes as input, our approach reduces the time to construct RLE sequences which, in turn, are used to construct the FEC table. Our column-based RSE technique, in contrast to the row-based unification step in [1] further reduces the complexity of constructing the tables. Simulations on several routing tables show that our approach constructs the FEC tables 4 to 29 times faster than the technique in [1]. Our efficient DPBT construction (Section III.B) can be generalized to reduce the construction time of a disjoint multibit trie [15].

ACKNOWLEDGEMENTS

The authors are grateful to L. Dardini who made available the code of technique in [1], and S. Sahni and H. Lu for providing us the various routing table databases. Dr. Rai acknowledges his partial support from the NSF grant CCR 0310919.

REFERENCES

[1]. P. Crescenzi, L. Dardini, and R. Grossi, “IP address lookup made fast and simple,” Proceedings *of the 7th Annual European Symposium on Algorithms*, July, 1999, pp. 1-10.

[2]. M. Degermark, A. Brodnik, A. Carlsson, and S. Pink, “Small forwarding tables for fast routing lookups,” ACM SIGCOMM, 1997, pp. 3-14.

[3]. W. Eatherton, G. Varghese, Z. Dittia, “Tree Bitmap: Hardware/Software IP lookups with Incremental updates,” *ACM SIGCOMM Computer Communications Review*, vol. 34, no. 2, April, 2004, pp. 97-122.

[4]. C. Labovitz, G.R. Malan, and F. Jahanian, “Internet routing instability,” *IEEE/ACM Transaction on Networking,* Vol.6, No.5, 1998, pp. 515-528.

[5]. B. Lampson, V. Srinivasan, and G. Varghese, “IP lookups using multiway and multicolumn search,“ *IEEE/ACM Transactions on Networking,* Vol. 7, No. 3, 1999, pp. 324-334.

[6]. D.H. Lorenz, A. Orda, D. Raz, and Y. Shavitt, “How good can IP routing be?,” DIMACS Technical Report, May 2001.

[7]. H. Lu and S. Sahni, “O(log n) dynamic router-tables for prefixes and ranges,” *IEEE Transaction on Computers,* Vol.53, No.10, 2004, pp. 1217 – 1230.

[8]. H. Lu, and S. Sahni, “Enhanced interval trees for dynamic IP router-tables,” *IEEE Transaction on Computers,* Vol. 53, No. 12, 2004, pp. 1615 - 1628.

[9]. S. Nilsson, and G. Karlsson, “Fast address lookup for Internet routers,” *IEEE Journal on Selected Areas in Communications*,Vol. 17, No. 6, June, 1999, pp. 1083 - 1092.

[10]. V.C. Ravikumar, R. Mahapatra, and J.C. Liu, “Modified LC-trie based efficient routing lookup,” *Proceeding of the 10th IEEE international Symposium on Modelling, Analysis, & Simulation of Computer & Telecommunications Systems*, 2002, pp. 1-6.

[11]. M.A. Ruiz-Sanchez, E.W. Biersack, and W. Dabbous, “Survey and taxonomy of IP address lookup algorithms”, *IEEE Network,* Vol. 15, No. 2, 2001, pp. 8-23.

[12]. S. Sahni, and K.S. Kim, “Efficient construction of multibit tries for IP lookup,” *IEEE/ACM Transaction on Networking,* Vol.11, No.4, 2003, pp. 650-662.

[13]. S. Sahni, and K.S. Kim, “An O(log n) dynamic router-table design,” *IEEE Transaction on Computers,* Vol. 53, No. 3, 2004, pp. 351 – 363.

[14]. S. Soh, L. Hiryanto, S. Rai, R. Gopalan, “Dynamic router tables for full expansion/compression IP lookup,” On Submission.

[15]. V. Srinivasan, and G. Varghese, “Fast Address Lookups Using Controlled Prefix Expansion,” *ACM Transaction on Computer Systems,* Vol. 17, No. 1, 1999, pp. 1-40.

[16]. P.C. Wang, C.T. Chan, and Y.C. Chen, “A fast table update scheme for high performance IP forwarding,” A fast table update scheme for high performance IP forwarding,” *International Conference on Parallel and Distributed Systems*, 2001, pp. 592-597.