Dynamic Virtual Routers Using Multiway Segment Tree

Yeim-Kuan Chang, Zi-Yang Ou
Department of Computer Science and Information Engineering,
National Cheng Kung University,
Tainan, Taiwan
{ykchang, p76001190}@mail.ncku.edu.tw

***Abstract*—Recently, research community has drawn lots of attentions in the router virtualization that allows multiple virtual router instances running on the same physical router platform. Thus, the virtualized router should be able to handle packets from different virtual networks. Once the multiple virtual routing tables are merged, memory requirement can be reduced due to the common entries among virtual routing tables. Many previous works use trie-based methods to merge the virtual routing tables. In this paper, we propose a range-based merging method. The data structure is based on the dynamic multiway segment tree (DMST) that is implemented with standard B-tree structure. As our experimental results show, faster lookup speed and incremental update can be achieved. The proposed method performs much better than the trie-based ones in lookup speed and scalability, and has similar memory consumption.**

Keywords-virtual routers; segment tree; B-tree

I. INTRODUCTION

Network virtualization [10] allows Internet Service Providers (ISPs) to dynamically define multiple heterogeneous virtual networks on top of the physical network with the isolation from each other. With the network virtualization, the networking resource can be efficiently utilized. As a result, a significant saving can be made in terms of power consumption, cost of networking devices and maintenance. Service providers can deploy and manage customized services on those virtual networks for different end users. In other words, each virtual network may run different packet forwarding protocols and the flexibility of network can be increased. To achieve network virtualization, multiple routers should be able to consolidate into a single hardware platform. This is also known as router virtualization.

Router virtualization is a technique that allows multiple virtual router instances to co-exist on the same physical router platform. In this way, the virtual router platform needs to maintain multiple forwarding information bases (FIBs). Each FIB has the same characteristics as that of the non-virtual routers. Thus, the virtualized router should have the ability to handle packets from each virtual network. We can view each virtual network as an abstraction of network away from the underlying physical network. Virtual networks are functionally independent from each other.

An example of the need for supporting different routing functions on the common platform, similar to virtualization, is the MPLS layer 3 virtual private networks (L3 VPNs) [8]. The service provider's edge routers, which perform the routing for each VPN customer, need to be capable of handling a lot of VPNs and maintaining private routing tables of different VPN customers. Due to the rapid growth in the VPN market, the memory requirement of VPN routing tables has become a bottleneck [8].

The primary limitation of router virtualization is the scalability, i.e., the number of virtual router instances that can be supported on the same physical router. Such limitation comes from the insufficient resource such as SRAM, TCAM, and other on-chip high-speed memory used for caching the data structures of packet forwarding information. For example, the latest BGP routing table contains more than 450 K prefixes [9] while a state-of-art 18-Mb TCAM can only store 500 K IPv4 prefixes. As a result, it is hard to support two different BGP routing tables using the straightforward method that partitions the memory and allocate it to each virtual router separately. Thus, we should consider the techniques of reducing the memory requirement to improve the scalability.

The kernel function of a router is the IP lookup. Thus, in this paper, we solve the IP lookup problem by treating prefixes in the virtual routing tables as ranges. A range $R = [e, f]$ matches the destination address d if and only if $e \leq d \leq f$ and R and the virtual network ID associated with the packet for d belong to the same virtual router. The proposed data structure is the modification of dynamic multiway segment tree (DMST) [1] that supports dynamic range insertions and deletions. DMST is a B-tree based data structure, and each node is augmented with range set called *canonical set*. The data structures of canonical set are proposed in [7]. Because there are many virtual routing tables, we duplicate lots of canonical sets which belong to different virtual IDs that guarantee isolation among virtual routers. However, some canonical sets are unused. To make efficient use of memory, we use the bitmap compression technique.

The keys used to build the DMST are not the traditional endpoints. For a range $R = [e, f]$, we use $e - 1$ and f as the keys to be inserted into DMST based on the minus-1 endpoint scheme [7], instead of the traditional endpoints e and f. The minus-1 endpoint scheme uses fewer keys than the traditional ones. Thus, the height of DMST can be smaller and the lookup speed of DMST can be faster.

We use the incremental method to merge multiple virtual routing tables into a DMST structure. We first insert all the prefixes in the first virtual routing table into DMST structure, and then the next virtual routing table. The same process is repeated for all the other virtual routing tables and the DMST is augmented.

The rest of the paper is organized as follows: Background and related work is discussed in Section II; our proposed scheme is given in Section III; the experimental results of the proposed method are presented in Section IV; Section V concludes the paper.

II. BACKGROUND AND RELATED WORK

A. Related work

Two methods for router virtualization have been proposed in the literature. In the *separate scheme*, a separate router instance is created for each virtual router. While in the *merge scheme*, all the virtual routing tables are merged into a single routing table. Both schemes have their pros and cons.

The *separate scheme* provides perfect traffic isolation among virtual routers hence avoids interference from one virtual router to the others. However, the much more router hardware resource usage makes it less attractive. In [11], the authors use the separate method to implement up to four virtual router instances on hardware (on NetFPGA). Their experimental results show that the throughput and scalability are not high due to the extensive hardware resource usage.

On the other hand, the *merge scheme* has higher scalability than the separate one. However, its traffic isolation is not so strong because all the virtual routers are on the same platform. Another important feature of network virtualization is the fair resource allocation. This feature is hard to be guaranteed since one router can use a larger part of available resources, causing the remaining virtual routers starving. Multiroot [2], Trie Overlapping [5], Trie Braiding [6] and Tree based virtualization [12] adopt this method.

In [5], the authors present a small, shared data structure for IP lookup in a virtualized router using the merged method. They use a simple overlapping scheme to merge a number of tries of different virtual routers into a merged trie. They have used the shared data structure to achieve significant memory saving. Their algorithm performs well when the routing tables contain similar data structure. They also proposed to use leaf pushing to further reduce the node size in the merged trie, by pushing all the next-hop in non-leaf nodes to leaf nodes. After leaf pushing, every non-leaf node stores only two children pointers, while every leaf node stores only next-hop information.

In [6], the authors present another algorithm for the merged method by mapping the tries to a merged trie using a heuristic method, in order to increase the overlap among the different tries and make the final merged trie more compact compared to [5]. The scheme enables each trie node to swap its left child node and right child node freely. Moreover, the algorithm enables the traversal behavior at each node changed if the braiding bit set or not. For example, with the braiding bit set, the child to be traversed will be different (e.g. taken the left child instead of the right child). Although it is memory-efficient, the complexity of its algorithm makes it less attractive. Contrary to [5], their algorithm performs well when the virtual routers have less similar data structures.

In [2], the authors present an improved overlapping scheme of [5]. They take account of the address space allocation of provider edge networks. Before merging, they examine the trie of each virtual router and find the common prefix. Due to the extraction of common prefix nodes, they can further reduce the memory requirement and improve the scalability compared to [5] and [6].

B. Prerequisites

We list some definitions of the core part of DMST proposed in [7] that we will need.

Definition 1 (Elementary Interval) *Let the set of S elementary intervals constructed from a set of W-bit ranges R be* $X = \{X_i \mid X_i = [e_i, f_i]$, for $i = 1$ to $S\}$. *Then X must satisfy the following properties:*

1. $e_1 = 0$ and $f_s = 2^W-1$,
2. $f_i = e_{i+1}-1$, for $i = 1$ to $S-1$,
3. *all addresses in Xi are covered by the same subset of R called range matching set of* X_i (EI_i), *and*
4. EI_i *is not equal to* EI_{i+1}, *for* $i = 1$ to $S-1$.

Definition 2 (Minus-1 Endpoint Scheme) *The two endpoints of a range* $[e, f]$ *are* e-1 *and* f.

Definition 3 (Range Allocation Rule) *Range R is stored in the canonical set of a node x (x.Cset) if and only if the interval of x (intvl(x)) is contained in R, but the interval of the parent of x (intvl(parent(x))) is not contained in R.*

According to the minus-1 endpoint scheme, the set of endpoints constructed from the nine 6-bit ranges of two virtual routing tables in Table I are {3, 7,15, 21, 23, 31, 39, 47, 51, 54, 55}. Considering $e_1 = 0$ and $f_{12} = 63$, the twelve elementary intervals X_1 to X_{12} can be constructed as shown in Fig. 1.

For each prefix in Table I, we use its virtual ID *vid* to identify which virtual routing tables it belongs to. Fig. 1 shows a possible order-3 tree for the prefixes of two virtual routing tables in Table I. Each leaf node is associated with an interval called the elementary interval, while each non-leaf node *x* is associated with the one denoted by *intvl(x)*, which is the union of elementary intervals in the sub tree rooted at node *x*. For example, the interval associated with the root node in the tree covers the entire address space $[0, 2^W-1]$. Each node is also associated with *K* canonical sets, where *K* is the number of virtual routing tables. For example, in Fig. 1, the table at top of each node is used to represent the 2 canonical sets. If all the canonical sets are unused, the table is omitted due to the page limitations. Each endpoint is stored in exactly one node as its key.

Based on the range allocation rule, the range matching set of an elementary interval is equal to the union of canonical sets traversed on the path from the root down to the leaf. For example, by taking *vid* = 0 into consideration, we can know the range matching set of elementary interval 1 is {P1} while the range matching set of elementary interval 2 is {P1, P3}. We can see that every two consecutive elementary intervals

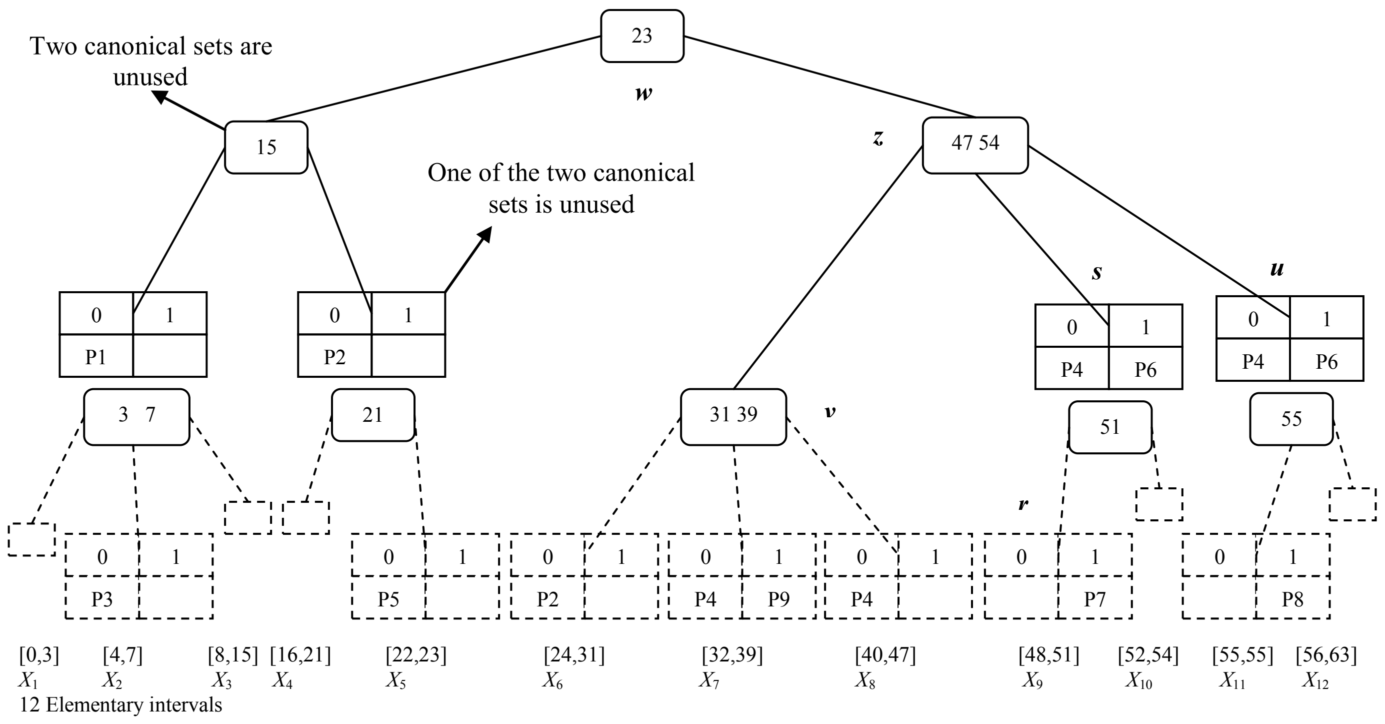

Figure 1. A possible tree built according to Table I.

TABLE I. Example of two routing tables with nine 6-bit prefixes

ID	Virtual ID	Prefix	Range	Endpoints	
				start	finish
P1	0	000000/2	[0,15]	-	15
P2	0	010000/2	[16,31]	15	31
P3	0	000100/4	[4,7]	3	7
P4	0	100000/1	[32,63]	31	-
P5	0	010111/5	[22,23]	21	23
P6	1	110000/2	[48,63]	47	-
P7	1	110000/4	[48,51]	47	51
P8	1	110111/6	[55,55]	54	55
P9	1	100000/3	[32,39]	31	39

will not have the same range matching set. When range R1 is more specific than range R2, R1 must be stored in the lower level than R2. For example, P3 is stored in the lower level than P1, because it is more specific.

The data structure of non-leaf node consisting of *t* keys is formatted in a linear list as [t, $Cset_1$, ..., $Cset_K$, $child_0$, key_1, $child_1$,...,key_t, $child_t$], where $Cset_i$ is a canonical sets belonging to *vid* = *i* for *i* = 1 to *K* and $child_j$ is a pointer to the *j*th sub tree for *j* = 0 to *t*. Also, the *t keys* stored in an internal node are sorted in increasing order. A leaf node only stores canonical sets.

III. Proposed scheme

A. Lookup Process

When a packet arrives, its virtual ID *vid* and destination IP address *d* are extracted. According to *vid* and *d*, the lookup process finds the ranges belonging to *vid* and containing *d*. If each range is assigned a priority, the lookup process finds the highest priority range among all matching ranges. In this paper, we use the traditional priority assignment method to assign the priority of a range as follows: Range R1 is assigned with a higher priority if R1 is more specific than R2. That is to say, the prefix with the longest prefix length obtains the largest priority. Hence the routing table lookups find the longest prefix among all matching prefixes of *vid* and *d*.

```
Algorithm Lookup(root,vid,d)
{
    x=root;
    k=0;
    while($x \neq null$){
        if($x.Cset_{vid} \neq \emptyset$)  $Cset[++k] = x.Cset_{vid}$;
        if(x is a leaf node)  break;
        $x.key_0 = predecessor(x.key_1)$;
        $x.key_{x.t+1} = successor(x.key_{x.t})$;
        Binary search on $x.key_0$ to $x.key_{x.t+1}$;
        if($x.key_{i-1} < d \leq x.key_i$)  $x = x.child_{i-1}$
    }
    return the highest priority range in Cset[k];
}
```

Figure 2. The lookup process

The Fig. 2 shows the proposed lookup algorithm. A tree traversal is first performed from the root to the leaf node corresponding to the elementary interval containing *d*. While traversing the tree, all of the explored nonempty canonical sets belonging to *vid* are recorded in the array *Cset*[1..*k*]. Finally, the highest priority (the most specific) range must exist in the

Cset[*k*], and the lookup process returns it as the best matching prefix. Its associated next-hop is used to forward the packet.

For instance, in Fig. 1, we assume that a packet with a destination address of $d = 48$ with a virtual ID $vid = 1$ arrives. The nodes w,z,s,r are traversed, and the nonempty canonical sets belonging to $vid = 1$ are {P6} and {P7}. So, the matching ranges are P6 and P7, and the most specific range is P7.

B. *Insertion*

For each prefix in IP routing tables, we can find its corresponding range. Suppose the two endpoints of the range are e and f, where e is the starting endpoint which can be generated by padding the prefix with 0 up to the maximum length and f is the finishing endpoint which can be generated by padding the prefix with 1 up to the maximum length. Based on the minus-1 endpoint scheme, the two endpoints that we want to put into the tree are e-1 and f. When inserting endpoints, we don't take the virtual ID into account. We just insert e-1 and f merely.

For each virtual routing table, there are three steps to insert a range $R = [e, f]$ of each prefix:

1. If e is not zero, insert e-1 as a new key in the tree.
2. If f is not 2^w-1, insert f as a new key in the tree.
   (The w is 32 for IPv4, and 128 for IPv6.)
3. Insert R into the tree according to the range allocation rule.

C. *Insert an endpoint*

Fig. 3 shows the proposed algorithm that inserts an endpoint *ep* into the tree. It is an adaptation of standard B-tree insertion algorithm and is described as follows:

Step 1: Like lookup process, a tree traversal is performed to find a key equal to *ep*. If *ep* is already in the tree, the search for *ep* terminates at a node that has *ep* as one of its keys. If *ep* is not in the tree, the search for *ep* terminates at a leaf node whose parent node will contain *ep*.

Step 2: k is decremented by one, then x and i are set to $p[k]$ and $b[k]$, respectively. Let $x.key_0$ and $x.key_{x.t+1}$ be the predecessor($x.key_1$) = $s[k]$-1 and the successor($x.key_{x.t}$) = $f[k]$, respectively. The endpoint *ep* is inserted into node x between $x.key_{i-1}$ and $x.key_i$, where $x.key_{i-1}<ep<x.key_i$. Because the insertion of *ep* splits the old elementary interval [$x.key_{i-1}$+1, $x.key_i$] into two smaller intervals, a new leaf node pointed to by y has to be created. Node y is a duplication of leaf node pointed to by $x.child_{i-1}$. Endpoint *ep* and node y are inserted as $x.key_i$ and $x.child_i$, respectively, and $x.t$ is incremented by one.

Step 3: When $x.t$ is smaller than m, the insertion of *ep* is finished. Else node x is full, evenly split it into two nodes denoted by x' and y, respectively. The middle key *keyg* of x is inserted into the x's parent, where g = m/2. Specifically, the keys less than the left of *keyg* along with the associated child pointers remain in x, those greater than the right of *keyg* are put into the new node y, and *keyg* and y are inserted into the x's parent. Let x' denote the new x. After node x is split, the canonical sets belonging to x' and y need to be adjusted to be in keeping with the range allocation rule. As mentioned in step 1, node x is pointed to by $p[k-1].child_{j-1}$ after the tree traversal, where $p[k-1]$ is the parent of x and j is $b[k-1]$, respectively.

```
Algorithm Insert_Endpoint(root,ep) {
  // Traverse the tree to find ep (Step 1)
01  Perform tree traversal to find arrays p[], s[], f[], b[], and
      k, where p[i] for i=1 to k records the traversed nodes,
      [s[i],f[i]] is the interval associated with node p[i],
      b[i] is index for node p[i] such that
      $p[i].key_{b[i]-1}<ep<p[i].key_{b[i]}$;
02  if(p[k] is not a leaf node) return;
  // Insert ep which is not in the tree (Step 2)
03  k=k-1; x=p[k]; i=b[k];
04  y = duplicate_a_leaf_node($x.child_{i-1}.Cset$)
05  insert ep and y as $x.key_i$ and $x.child_i$ in node x, and x.t++;
  // node overflow, split x into two nodes, x and y (Step 3)
06  while( x.t = m ){
  // Create a new node y (Step 3.1)
07    g = m/2; keyg = $x.key_g$;
08    y = create_new_node();
09    move $child_g$, [$key_{g+1}$,$child_{g+1}$], …, [$key_m$,$child_m$] from
        node x to node y;
10    y.Cset = x.Cset;
11    y.t = m-g; x.t = g-1;
  // Adjust x.Cset (Step 3.2)
12    xSet = {R|R ∈ $x.child_{g-1}.Cset$ and R covers [s[k],keyg]};
13    for( h = 0 ; h ≤ x.t ; h++)
14      $x.child_h.Cset = x.child_h.Cset - xSet$;
15    x.CSet = x.CSet + xSet;
  // Adjust y.Cset (Step 3.3)
16    ySet = {R|R ∈ $y.child_0.Cset$ and R covers [keyg+1,f[k]]};
17    for( h = 0 ; h ≤ y.t ; h++)
18      $y.child_h.Cset = y.child_h.Cset - ySet$;
19    y.CSet = y.CSet + ySet;
  // Step 3.4:
20    if(k = 1){
21      root = create_node(t=1, $child_0$=x, $key_1$=ep, $child_1$ = y);
22      break;}
23    k = k-1; x = p[k]; j = b[k]; x.t++;
24    insert keyg and y as $x.key_j$ and $x.child_j$ in node x; }
    }
```

Figure 3. The algorithm that inserts a new endpoint

Before proceeding to insert *keyg* and y. As we can see in Fig. 4. Ranges like R1 that contains the $intvl(x')$ = [$p.key_{j-1}$+1, $x.key_g$] was stored in all canonical sets of the children of node x' before splitting. Thus, R1 needs to be removed from all these canonical sets of the children of x' and be inserted into $x'.Cset$. Similarly, those like R2 that contains the $intvl(y)$ = [$x.key_g$+1, $p.key_j$] needs to be removed from all canonical sets of the children of y and be inserted into $y.Cset$. The above canonical set adjustments are shown in Line 12-15 and Line 16-19 of Fig 3. Finally, *keyg* and y are inserted as key_j and $child_j$ in $p[k-1]$, respectively. Since node $p[k-1]$ gets one more key, the same split process may need to repeat at $p[k-1]$ if $p[k-1]$ were full again. In the end, the split process may reach the root of tree. As in the regular B-tree, a new root node may

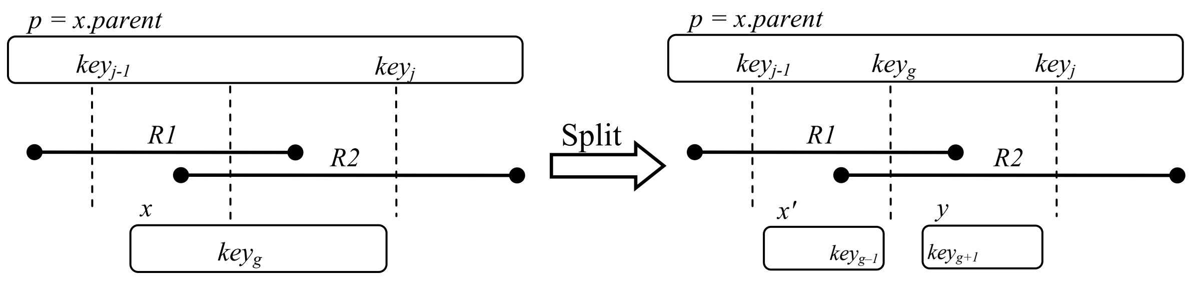

Figure 4. Node splitting around $x.key_g$ (a) Before split (b) After split

need to be created and thus the height of tree is increased by one, as shown in Line 20-24 of Fig. 3.

D. Insert a Range

Fig. 5 shows the proposed algorithm that inserts range R based on the range allocation rule. It shows the procedure that puts range R into proper canonical sets belonging to *vid*. We describe the detailed steps as follows.

```
Algorithm Insert_Range (root,vid,R) // assume R=[e, f]
{
  // Step 1:
01  Find LCA node $y$ and the interval $[lb, ub]$ covered by $y$.
  // Step 2:
02  if( $[lb, ub]$ is contained in $R$){Add $R$ in $y.Cset_{vid}$;  return;}
  // Step 3:
03  Set $y.key_0 = lb - 1$, $y.key_{y.t+1} = ub$.
04  for( $k = 1$ to $y.t+1$ )
05    if( $R$ covers $[y.key_{k-1}, y.key_k]$)
06      Add $R$ in $y.child_{k-1}.Cset_{vid}$;
  // Step 4:
07  if($y.key_{i-1} < e-1 < y.key_i$){
   // $i \in \{1, \ldots, y.t+1\}$ and $e-1 \neq$ any key in $y$
08    $x = y.key_{i-1}$;
09    while( $x$ is not a leaf node){ //$i \in \{1, \ldots, x.t\}$
10    if($x.key_i = e-1$){
11      for( $k = i$ to $x.t$ )  Add $R$ in $x.child_k.Cset_{vid}$;
12      break; }
13    if($x.key_{i-1} < e-1 < x.key_i$){
14      for( $k = i$ to $x.t$ )  Add $R$ in $x.child_k.Cset_{vid}$;
15      $x = x.child_{i-1}$; }
16    }
17  }
  // Step 5:
18  if clause which is the same as Step 4 except
    1. '$e$ - 1' is replaced with '$f$'
    2. The first for-loop is replaced with
        for($k = 0$ to $i - 1$) Add $R$ in $x.child_k.Cset_{vid}$; .
    3. The second for-loop is replaced with
        for($k = 0$ to $i - 2$) Add $R$ in $x.child_k.Cset_{vid}$; .
}
```

Figure 5. The algorithm that inserts a range

Step 1 (line 01): Find the lowest common ancestor (LCA), - node y, of the two nodes with keys $e-1$ and f first.

Step 2 (line 02): If R contains the interval covered by the node y, then R is added in $y.Cset_{vid}$.

Step 3 (lines 03-06): If R contains the interval associated with any of the children of node y, R is added in that child's canonical set belonging to *vid*.

Step 4 (lines 07-17): If e-1 is equal to any key in node y, the insertion for R terminates. If $y.key_{i-1} < e-1 < y.key_i$, the tree is traversed from node y toward leaf to finds the node has $e-1$ as one of its keys. At each node that x traversed, R is added in canonical sets belonging to some of x's children and *vid*.

Step 5 (line 18): similar step like Step 4.

For instance, in Fig. 1, we insert a range R = [32, 63] belonging to $vid = 0$, the endpoint 31 is first inserted in the tree as shown in Fig. 1. Step 1 finds that the LCA node is z. Step 2 does nothing, and step 3 inserts R into $s.Cset_0$ and $u.Cset_0$ because R contains $intvl(s)$ and $intvl(u)$ but not $intvl(z)$. In step 4, the node v which contains key 31 is reached and R is inserted in $v.child_1.Cset_0$ and $v.child_2.Cset_0$.

IV. EXPERIMENTAL RESULTS

A. Experimental Setup

Twelve IPv4 core routing tables were collected from RIS on July 16, 2012 [3]. However, router virtualization primarily happens at provider edge networks, which are relatively small compared to core networks. Thus, we take those IPv4 core routing tables as the input of FRuG [4] to generate twelve close-to-real synthetic IPv4 routing tables, each has 100k prefixes.

The proposed algorithm was implemented in C and compile with gcc-4.4.5 compiler under Debian 6.0 with an optimization level –O2 is used. The simulations are run on a 3.20-GHz Intel Core i5 650 PC that has 8GB main memory. To get an accurate count of the clock cycles of processor, we use the instruction called ReaD Time Stamp Counter (RDTSC).

B. Lookup Speed

We compare our lookup process with the state-of-art designs. These candidates are the trie overlapping [5] and the

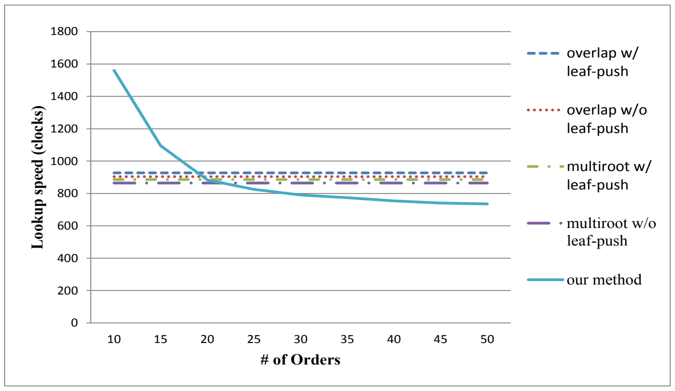

Figure 6. Lookup speed variation for different orders (12 FIBs).

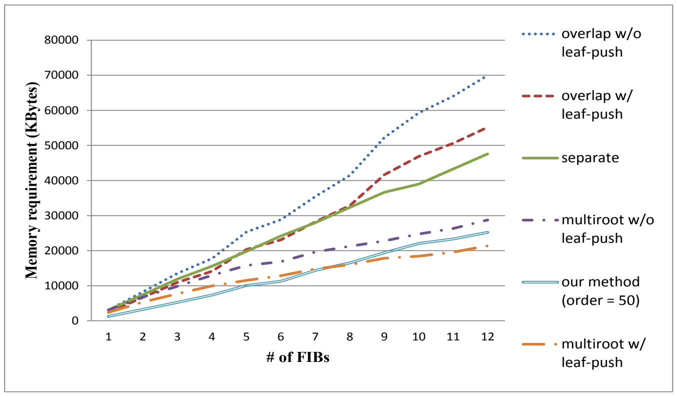

Figure 7. Memory analysis for 12 virtual routing tables

Multiroot [2], each has two variations: with and without leaf-push. All the candidates are trie based methods, so the worst-case number of node accesses is close to 32. But, our lookup process is the B-tree based methods. So the number of node accesses is $O(log_m N)$, where m is the order of B-tree and the N is the total number of prefixes. For example, Fig. 6 shows that when m increases, the number of node accesses decreases. And when m is great than 20, the average lookup speed of our method will be faster than the average lookup speed of these ones.

C. *Memory Efficiency and Scalibility*

The experiment was conducted using routing tables with random common prefixes like Multiroot does. The length of each common prefix varies from 2 to 5. We count the memory usages for four cases: 1) separated method, which stores each virtual routing table separately in a binary trie; 2) trie overlapping; 3) Multiroot; 4) our method.

Except for the bitmap compression technique mentioned before, we use the base offset technique to further reduce memory usages. In this way, each node stores only the first child pointer. Any child pointer can be computed by adding the base address to the offset of child pointer. To implement such technique, we use the dynamic memory allocation in the C programming language. Thus, all the child nodes belonging to one node are stored in ordered.

Fig. 7 shows that our method requires much less memory compared to the straightforward separate method and has similar performance compared to trie-based methods. Therefore, our method leads to savings in memory needed compared with the existing methods, hence improves the scalability considerably.

D. *Update Performance*

In router virtualization, the insertion or deletion of a virtual routing table to the virtualized router should be as quick as possible. In [2], [5] and [6], authors indicate that when any prefix is inserted or deleted, the entire data structure may need to be reconstructing in the worst case.

Therefore, our method is good for update because of the support of incremental update.

V. CONCLUSION

In this paper, we proposed a novel range-based approach to merge a number of virtual routing tables. The data structure is the modification of DMST, which is implemented with a B-tree for dynamic routing tables. Due to the B-tree structure of our method, we have improved the lookup speed and the update performance. The experiments employing synthetic IPv4 provider edge routing tables showed that our method performs much better than trie-based methods in terms of lookup speed, and has similar memory consumption hence improves the scalability.

REFERENCES

[1] Yeim-Kuan Chang, Yung-Chieh Lin, and Cheng-Chien Su, "Dynamic Multiway Segment Tree for IP Lookups and the Fast Pipelined Search Engine," IEEE Transactions on Computers, VOL. 59, NO. 4, pp. 492-506, APRIL 2010.
[2] Thilan Ganegedara, Weirong Jiang, Viktor K. Prasanna, "Multiroot: Towards Memory-Efficient Router Virtualization", IEEE International Conference on Communications (ICC 2011), Kyoto, Japan, June 2011.
[3] RIS RAW DATA [Online]. [http://data.ris.ripe.net].
[4] T. Ganegedara, W. Jiang, V.K. Prasanna, "FRuG: A Benchmark for Packet Forwarding in Future Networks," to appear in proc. IEEE IPCCC, 2010.
[5] Jing Fu and Jennifer Rexford, "Efficient IP-address lookup with a shared forwarding table for multiple virtual routers," in proc. ACM CoNEXT, 2008, pp. 1-12.
[6] Haoyu Song, M. Kodialam, Fang Hao, T.V. Lakshman, "Building Scalable Virtual Routers with Trie Braiding," in proc. IEEE INFOCOM, 2010, pp. 1-9.
[7] Y.-K. Chang and Y.-C. Lin, "Dynamic Segment Trees for Ranges and Prefixes," IEEE Trans. Computers, vol. 56, no. 6, pp. 769-784, June 2007.
[8] E. Rosen and Y. Rekhter, "RFC 2547: BGP/MPLS VPNs," http://www.ietf.org/rfc/rfc2547.txt, 1999.
[9] "Route Views project," Univ. Oregon, Eugene, OR, 2013 [Online].[http://www.routeviews.org].
[10] N.M. Chowdhury, Kabir Mosharaf and Raouf Boutaba, "A Survey of Network Virtualization," The International Journal of Computer and Telecommunications Networking, 2010, pp. 862 - 876.
[11] Deepak Unnikrishnan and Ramakrishna Vadlamani, Yong Liao and Abhishek Dwaraki Jeremie Crenne, Lixin Gao, Russell Tessier, "Scalable network virtualization using FPGAs", in proc. ACM/SIGDA FPGA, 2010, pp. 219 – 228.
[12] H. Le, T. Ganegedara, and V. Prasanna, "Memory-efficient and scalable virtual routers using fpga," Field Programmable Gate Arrays (FPGA), 2011 International Symposium on, 20.