Frontiers of Information Technology & Electronic Engineering
www.zju.edu.cn/jzus; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

A splitting-after-merging approach to multi-FIB compression and fast refactoring in virtual routers*

Da-fang ZHANG†‡1, Dan CHEN†1, Yan-biao LI1, Kun XIE1,2, Tong SHEN1
(1College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China)
(2Department of Electrical and Computer Engineering, State University of New York, New York 11790, USA)
†E-mail: dfzhang@hnu.edu.cn; danchen@hnu.edu.cn

Received Dec. 31, 2015; Revision accepted Feb. 29, 2016; Crosschecked Nov. 8, 2016

Abstract: Virtual routers are gaining increasing attention in the research field of future networks. As the core network device to achieve network virtualization, virtual routers have multiple virtual instances coexisting on a physical router platform, and each instance retains its own forwarding information base (FIB). Thus, memory scalability suffers from the limited on-chip memory. In this paper, we present a splitting-after-merging approach to compress the FIBs, which not only improves the memory efficiency but also offers an ideal split position to achieve system refactoring. Moreover, we propose an improved strategy to save the time used for system rebuilding to achieve fast refactoring. Experiments with 14 real-world routing data sets show that our approach needs only a unibit trie holding 134 188 nodes, while the original number of nodes is 4 569 133. Moreover, our approach has a good performance in scalability, guaranteeing 90 000 000 prefixes and 65 600 FIBs.

Key words: Virtual routers, Merging, Splitting, Compression, Fast refactoring
http://dx.doi.org/10.1631/FITEE.1500499 **CLC number:** TP393

1 Introduction

Virtual routers (VRs) are key components of some emerging technologies, such as virtual private networking (Fu *et al.*, 2001; Wang *et al.*, 2010), network function virtualization (Bando and Chao, 2010; Bao *et al.*, 2010; Han *et al.*, 2015), and software-defined networking (McKeown *et al.*, 2008; Sezer *et al.*, 2013). Multiple logical routers coexist in a physical device on the virtual router platform, which enhances the resource utilization ratio as well as controlling flexibility. However, VRs suffer from more challenges due to the same reason, particularly in terms of memory efficiency and scalability.

1.1 Problem statement

In the environment of virtual routers, each router instance on the virtual router platform (Liu *et al.*, 2011; Xie *et al.*, 2011) has its own forwarding information base (FIB) and works separately to process packets. If all the FIBs are maintained separately, memory resource will not be shared and it is useless for improving memory efficiency.

Moreover, to leave space for potential updates, some 'head room' is reserved for each FIB. Because different FIBs have different characteristics and potential behaviors, some FIBs will run out of memory while others remain far from the memory limit after a long-term update.

Therefore, the first challenge is the desirable trade-off between memory share and work isolation.

‡ Corresponding author
* Project supported by the National Basic Research Program (973) of China (No. 2012CB315805), the National Natural Science Foundation of China (Nos. 61173167 and 61472130), the Prospective Research Project on Future Networks of Jiangsu Future Networks Innovation Institute, China (No. 2013095-1-05), the Hunan Provincial Innovation Foundation for Postgraduate, China (No. CX2014B150), and the State Scholarship Fund of China (No. 201406130048)
ORCID: Dan CHEN, http://orcid.org/0000-0001-7049-9945

Moreover, when the memory is running out or in the case of instance insertion/deletion, the whole system will have to be refactored. Because all functionalities will be suspended during the refactoring, it is very important yet challenging to perform refactoring as quickly as possible.

1.2 Prior research

Many approaches have been proposed for the storing of multi-FIB; they all aim to achieve efficient memory share. TrieOverlap (Fu and Rexford, 2008) was the first attempt to share a large number of trie nodes by overlapping; this method owns multiple tries and each of them represents a separate FIB. Furthermore, TrieBraiding (Song *et al.*, 2010; 2012) focuses on reversing some nodes in the tries to produce more ‘similarity’; thus, more nodes will be shared. To avoid keeping track of each FIB in all the nodes, both approaches apply the LeafPushing algorithm (Srinivasan and Varghese, 1999) to compress the node structure and simplify the lookup process. However, the LeafPushing algorithm produces redundancies of numbers, which results in undesirable update performance. Instead of the LeafPushing algorithm, TrieMerging (Eatherton and Dittia, 2003; Luo *et al.*, 2013) was proposed to compress the node structure by using a bitmap (Chan and Ioannidis, 1998; Wu *et al.*, 2006) in each node of the overlapped trie, and each bit represents whether the corresponding FIB has a valid match in this node. However, these overlapping-based schemas have a common disadvantage that the updates in any FIB will disrupt the lookup of other FIBs due to the highly shared structure.

At the same time, researchers proposed another type of approach that merges FIBs rather than tries. Specifically, a unique identification number is prepended in front of all the prefixes for each FIB, so prefixes from different FIBs can be distinguished from each other by the front bits. Thus, the updates of one FIB will affect only the specified subtrie. Due to considerations of memory efficiency, a 2-3 tree (Le *et al.*, 2011) method is used, but the cost of this method is the high reliance on special hardware design for desirable update performance.

In the field of FIB compression, many novel structures have also been proposed for pretty high memory efficiency (Song *et al.*, 2009; Huang *et al.*, 2011; Li *et al.*, 2014). However, all of them need an optimized configuration to achieve desirable effects. Moreover, they may need to recalculate the best configuration after some updates, which increases the overhead of refactoring.

1.3 Our approach

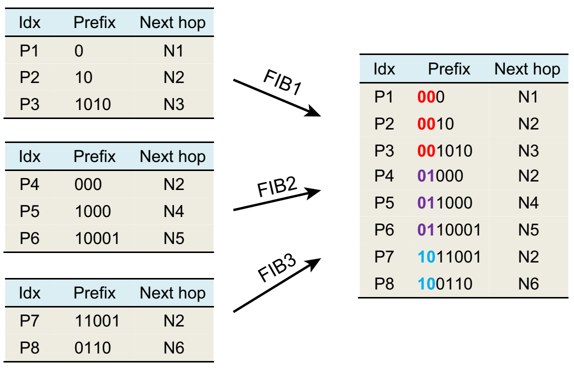
In this paper, we aim to develop a schema that not only keeps each FIB isolated, especially in view of aggregate updates, but also achieves high memory efficiency, scalability, and reasonable refactoring overhead. Instead of working from scratches, we learn from two existing ideas: FIB merging and prefix splitting. First, we adopt the FIB-merging approach to construct a merged FIB from multiple FIBs by assigning a unique identification number to all prefixes in each FIB. Then, we split all prefixes in the merged FIB at some specified position by using the splitting-after-merging method. Finally, we investigate how the selection process of split positions affects memory efficiency through a series of experiments with a large range of configurations. With further analysis of the statistical results, we demonstrate some reasonable strategies to select a proper split position, which reduces the overhead of refactoring sharply.

2 Splitting-after-merging approach

2.1 FIB merging

The trie-based method (Degermark *et al.*, 1997; Nilsson and Karlsson, 1999; Srinivasan and Varghese, 1999; Eatherton *et al.*, 2004; Song *et al.*, 2005) saves the storage time due to the partial similarity between the prefixes, but the update becomes more complex and lookup is hindered by the label updating of the shared node. On the contrary, the approach of merging FIBs by adding a unique identification to the front of each prefix can prevent similar prefixes sharing nodes and isolate the updates, but this approach suffers from large storage overhead. The 2-3 tree method optimizes the storage to a certain extent; it has the problem of unstable lookup and the high reliance on special hardware design.

In the splitting-after-merging method, our purpose is to optimize storage to provide a stable and fast lookup, which can still isolate the updates of multiple different FIBs. So, to keep the updates isolated, we construct a merged FIB containing every prefix of all the FIBs with each prefix being assigned a unique identification number. For example, suppose there are three FIBs (Fig. 1). First, we add a unique identification number '00' to the front of all the prefixes in FIB1, '01' to prefixes in FIB2, and '10' to prefixes in FIB3 (these unique identification numbers are the binary representations of the number of FIBs). Then, we merge these changed FIBs into a final one. Thus, all the updates of these three different FIBs can be isolated, because the importance of the merging process is not to compress the prefixes but isolate the FIBs. Because of the isolated FIBs, the update process will be much more simple.



Idx	Prefix	Next hop
P1	0	N1
P2	10	N2
P3	1010	N3

Idx	Prefix	Next hop
P4	000	N2
P5	1000	N4
P6	10001	N5

Idx	Prefix	Next hop
P7	11001	N2
P8	0110	N6

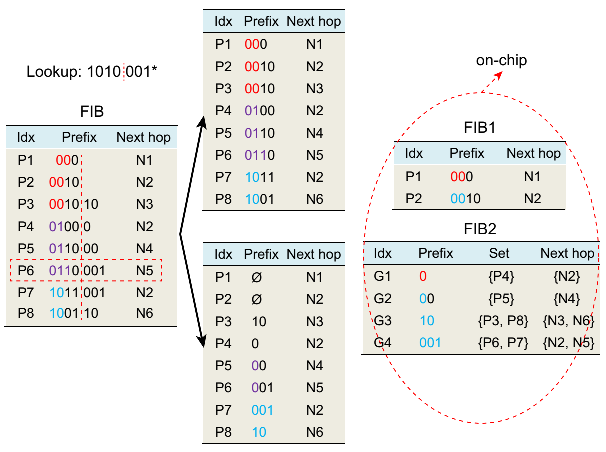
Idx	Prefix	Next hop
P1	000	N1
P2	0010	N2
P3	001010	N3
P4	01000	N2
P5	011000	N4
P6	0110001	N5
P7	1011001	N2
P8	100110	N6

Fig. 1 Three FIBs merging into one with the added identification number

2.2 Prefix splitting

Because the FIBs have been merged to ensure isolation, the key question now is to reduce storage consumption. Based on the partial similarity between prefixes, our next step is to split the merged trie. As illustrated in Fig. 2, if we choose four as the split position, the FIB is then split into two parts at the position of four. As we can see, the maximum length of the prefixes in either FIB1 or FIB2 is reduced after splitting. In particular, the maximum length of the prefixes in the first part is reduced from seven to four. Finally, we integrate the redundancies produced by the splitting. We find that the total number of prefixes in the new FIBs is reduced from eight to six. In other words, the memory consumption for storing the nodes is reduced and the lookup time is also saved due to the shorter prefixes.

Though the memory of storing has been reduced by the merging process, another key factor can also affect the memory efficiency. It is certain that the on-chip memory consumption changes while



Idx	Prefix	Next hop
P1	000	N1
P2	0010	N2
P3	0010 10	N3
P4	0100 0	N2
P5	0110 00	N4
P6	0110 001	N5
P7	1011 001	N2
P8	1001 10	N6

Idx	Prefix	Next hop
P1	000	N1
P2	0010	N2
P3	0010	N3
P4	0100	N2
P5	0110	N4
P6	0110	N5
P7	1011	N2
P8	1001	N6

Idx	Prefix	Next hop
P1	Ø	N1
P2	Ø	N2
P3	10	N3
P4	0	N2
P5	00	N4
P6	001	N5
P7	001	N2
P8	10	N6

FIB1

Idx	Prefix	Next hop
P1	000	N1
P2	0010	N2

FIB2

Idx	Prefix	Set	Next hop
G1	0	{P4}	{N2}
G2	00	{P5}	{N4}
G3	10	{P3, P8}	{N3, N6}
G4	001	{P6, P7}	{N2, N5}

Fig. 2 Splitting the merged FIB into two FIBs

the split position and the number of FIBs change. Moreover, the storage space of a fixed split position changes because the numbers of FIBs and prefixes have changed. Fig. 3 shows the curves of on-chip memory consumption varying with the split position and different numbers of FIBs. As we can see, the trends of all the curves look the same when the split position changes. The reason is that they all follow the same rule that the best split position changes as the number of FIBs increases. According to Fig. 4, we choose 18 as the split position; the on-chip memory consumption increases when the numbers of total prefixes and FIBs grow. These observations indicate that the split position should be adjusted due to the changes in the numbers of FIBs and total prefixes.

As the split position may influence the comprehensive performance of the split trie significantly, it is obvious that the core of our approach is to choose an appropriate split position. In addition, the faster we

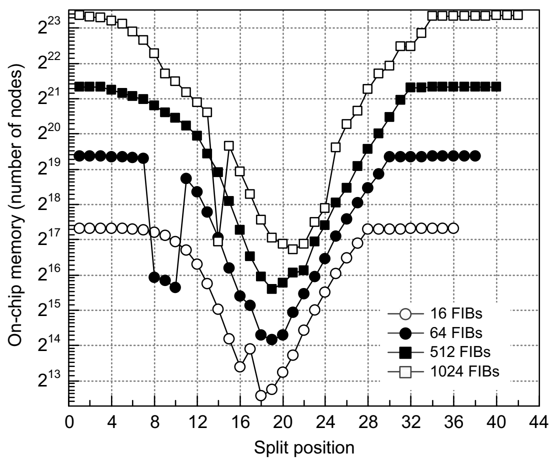


Fig. 3 The trend of on-chip memory varying with split position and the number of FIBs

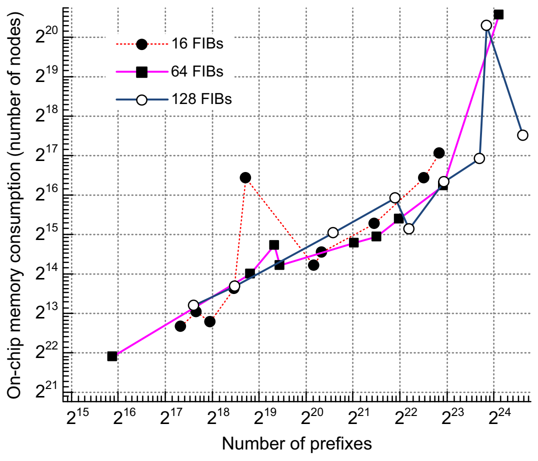


Fig. 4 The trend of on-chip memory varying with the number of FIBs (the split position is chosen as 18)

choose the split position, and the more accurate the split position chosen, the less time that is needed to build the split trie. It is also of great importance for the refactoring mechanism when the system is idle (update or adding new router instances frequently is likely to trigger the reconstruction). Under the condition of a single FIB, the maximum length of the prefix is fixed. However, in the case of a merged FIB, the length of the identity number is determined by the number of FIBs; so, the maximum prefix length changes when the number of FIB varies. Therefore, this increases the difficulty in choosing the ideal split position. In Section 4, we will conduct a large number of experiments for different parameters and then predict the ideal split position through statistical analysis, which can shorten the split position detection time.

2.3 Lookup algorithm

Because the splitting-after-merging approach merges the original FIBs and splits each FIB into two FIBs, the lookup algorithm is far more complicated than the simple and traditional longest prefix match (LPM) (Kobayashi *et al.*, 2000; Bass *et al.*, 2005). On the one hand, the address that we look up should be splitted. On the other hand, the hash mechanism (Broder and Mitzenmacher, 2001; Yu *et al.*, 2009; Li *et al.*, 2012) will be needed because the LPM in FIB2 may find only a group of original prefixes whose back parts are the same.

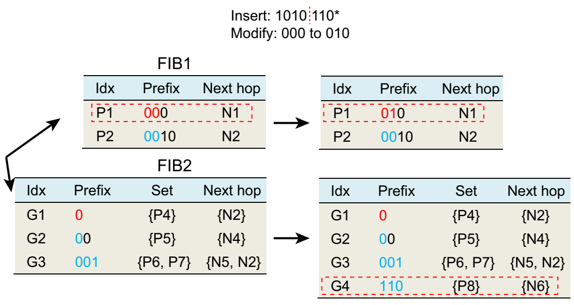
We use the example in Fig. 2 to illustrate our lookup algorithm. First, like the splitting of the prefixes in the FIB, the prefix should be split into two parts: as depicted in Fig. 2, the prefix 1010001 has been split into 1010 and 001. Second, we use 1010 to perform LPM in FIB1 and 001 in FIB2. It seems that the algorithm will be run after the LPM in each FIB, but as mentioned before, the prefixes in FIB2 have been compressed. In other words, when we perform LPM in FIB2, the results may represent several prefixes. As we can see, because P6 and P7 have the same back parts, it is unknown whether G4 represents P6 or P7; the front parts of P6 and P7 should be compared by using 1010 as the key to perform hashing until we find that 1010 is the first part of P6. The lookup algorithm is described in Algorithm 1.

Algorithm 1 Lookup algorithm /* look up the addr in FIB1 and FIB2 */

```
Input: FIB1, FIB2, addr, p
          /* split address at position p */
Output: P /* the index of the address */
 1: if (addr.length < p) then
 2:     P = FIB1.LPM(addr)
 3:     if (P == NULL) then
 4:         FIB1.insert(addr)
 5:         return NULL
 6:     else
 7:         return P
 8:     end if
 9: else
10:     [addr1, addr2] = addr.split(p)
11:     G = FIB2.LPM(addr2)
12:     if (G == NULL) then
13:         FIB2.insert(addr2)
14:         return NULL
15:     else
16:         if (P == NULL) then
17:             FIB2.insert(addr2)
18:             return NULL
19:         else
20:             return P
21:         end if
22:     end if
23: end if
```

2.4 Update approach

As the original FIBs have been merged into one and then split as two final FIBs, our update approach toward the prefixes is similar to the lookup process. Usually, there are three types of updates: inserting a new prefix, deletion of a prefix, and modifying an existing prefix. It is clear that these three types of updates are linked to each other closely. First, when we receive the prefix, it should be split at the splitting position. Then, if the new incoming prefix is to be inserted, its first part should be inserted into FIB1 and the back part into FIB2. However, this operation is based on the fact that all the parts of the new prefix do not exist in FIB1 and FIB2, or it will affect the modification process by changing the index of the group. To deal with the modification, the change of the index is only one aspect; when it comes to changing the prefix itself, we should delete the index of the prefix, insert the new prefix, and create the index. We can observe that the deletion is included in the modification process. Fig. 5 can be used to illustrate the update process; we assume that 1010110 is the new prefix to be inserted and we will modify 000 to 010. As explained in the lookup algorithm, the first step is splitting the prefix at the splitting position 4. We find that 110 does not exist in FIB2; so, 110 should be inserted into FIB2 with the new index in the set. As can be seen from Fig. 5, the modification from 000 to 010 leads to the deletion of 000 in FIB1 and 010 finally replaces the location of 000 stored previously.



Idx	Prefix	Next hop
P1	000	N1
P2	0010	N2

Idx	Prefix	Next hop
P1	010	N1
P2	0010	N2

Idx	Prefix	Set	Next hop
G1	0	{P4}	{N2}
G2	00	{P5}	{N4}
G3	001	{P6, P7}	{N5, N2}

Idx	Prefix	Set	Next hop
G1	0	{P4}	{N2}
G2	00	{P5}	{N4}
G3	001	{P6, P7}	{N5, N2}
G4	110	{P8}	{N6}

Fig. 5 Insertion and modification

3 Fast refactoring strategy

3.1 Challenges of fast refactoring

In many prior works, the best configuration needs to be recalculated after some update operations, which increases the overhead of refactoring, while in our approach, the reason causing the refactoring is not the novel structures, but the unknown nature of the FIBs to be inserted. As mentioned before, even the split position may influence the comprehensive performance of the split trie significantly, much less than that with the length of the prefixes changing all the time. Thus, once the newly inserted FIBs increase the length of the leading bits, the whole system needs to be rebuilt.

Generally, rebuilding the whole system in our approach needs first the selection of a split position, then the construction of a unibit split trie on the selected position, and finally the transform of it into a multibit trie (Richardson *et al.*, 2002; Saravanan and Senthilkumar, 2015). Theoretically, we have to test all possible split positions, and then we should construct a multibit trie and calculate its memory consumption on each position to determine which is the best choice. As a consequence, the first step is the most time-consuming operation.

3.2 Our mechanism for fast rebuilding

In today's network, rebuilding the system in a real-time manner is required to avoid losing too many packets. In other words, the lookup process will be suspended by the rebuilding process. In this context, accelerating system rebuilding is another key point to enhance the throughput. To achieve efficient rebuilding, we present a novel mechanism to select a position approximate to the best split position with as few tests as possible.

To rebuild the system, the first step is to test all the possible split positions. Moreover, a multibit trie must be constructed based on the unibit trie to calculate its memory consumption. Therefore, we conducted two contrast experiments. The first is aimed to compare the system rebuilding time between the original scheme and the improved scheme. The second is aimed to compare the memory efficiency of these two schemes.

The comparison of the system rebuilding time is shown in Fig. 6. There are 372 FIBs with 8 595 010 prefixes and the original number of FIBs is 128. In other words, 244 new FIBs are inserted into the original FIB set that contains 3 143 411 prefixes and the best split position is 18. Because the inserted FIBs have changed the length of the leading bits, we have to rebuild the whole system. As indicated in Fig. 6, if we adopt the original method, all the possible positions have to be tested and the system rebuilding time will be added up to 621.929 s. On the other hand, as the improved method shows, only three positions need to be tested. First, we can choose the position only from 18 to 20 according to the conclusion

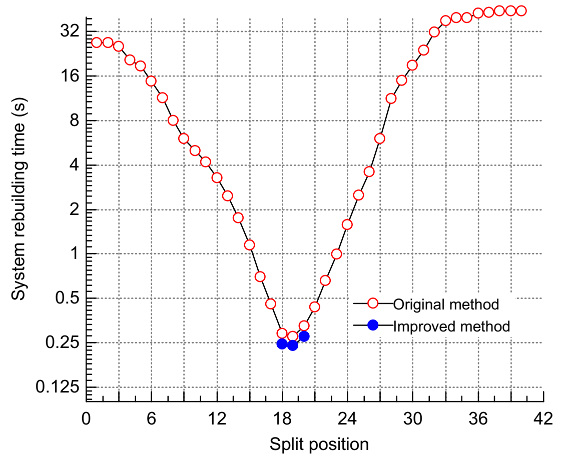


Fig. 6 Time efficiency of the two strategies of system rebuilding

based on Fig. 7. Then, according to Fig. 9, we can suppose that the best position may be 19. When the number of FIBs increases, the best split position will move forward. In addition, according to a large number of experiments on real-world data sets, we found that the trend of memory consumption with respect to split position for a multibit trie is always similar to that for a unibit trie (Section 4). Thus, we construct only the unibit trie rather than transform it into a multibit trie when we test one split position, which can help us save a lot of time. As we can see, the total reconstruction time has been reduced to 0.758 s. The experimental results demonstrate that our improved method actually saves a lot of time and achieves efficient rebuilding.

Fig. 7 shows the memory efficiency of the two strategies. For the first strategy, the memory requirements of the unibit trie and the multibit trie

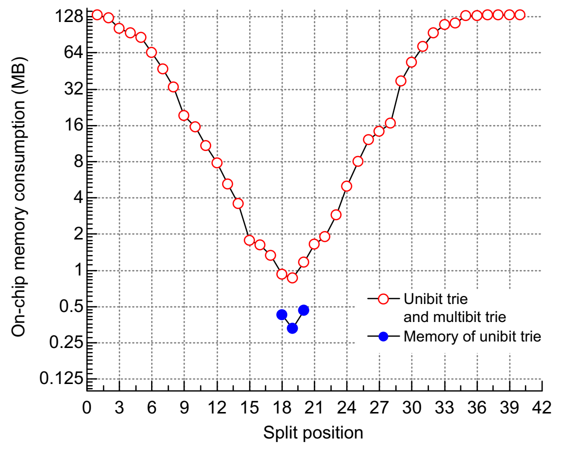


Fig. 7 Memory efficiency of the two strategies of system rebuilding

add up to the total memory consumption. As shown, both methods consume the least memory when the position is 19. However, the improved method consumes only 0.329 MB memory, while the original one consumes 0.867 MB. The valley values are obtained at the best split position. This observation completely conforms to the trend of on-chip memory varying with split position in Fig. 3.

The steps of the rebuilding procedure are summarized as follows:

1. Make the statistics of the numbers of current FIBs and total prefixes.

2. Make the statistics of the numbers of newly added FIBs and their prefixes.

3. Select the testing positions according to the cumulative distribution of the ideal split position in Fig. 9.

4. Choose the test position based on the current split position and the information of the newly added FIBs.

5. Test the left and right positions of the selected position to make sure which is the best split position.

4 Evaluation experiments

In this section, we first conducted an experiment to illustrate how the best split position changes due to the changes in the numbers of FIBs and total prefixes. Then, we summarized the cumulative distribution rule of the ideal split position. Finally, we made three contrast experiments: the first is to compare the on-chip memory consumptions of unitrie and multitrie methods, the second is to compare on-chip memory and off-chip memory, and the third is to compare our method with other conventional methods.

To evaluate the memory efficiency, a large number of routing tables are needed. We collected 272 real-world, public border gateway protocol (Rekhter and Li, 1994) routing data sets from the RIPE RIS Project (Table 1). We randomly combined the prefixes (about 90 000 000 prefixes) into multiple groups of FIBs and the largest number of FIBs reaches 65 600, which is enough to simulate the real virtual routers. Besides, we assume that the smallest FIB contains 1000 prefixes and the largest one contains 560 000 prefixes.

As mentioned in Section 2, the core of our

Table 1 Routing data set

Router	Location	Collected time	Collected number
Rrc00	RIPE NCC, Amsterdam	2001-04 to 2015-05	22
Rrc01	LINX, London	2001-10 to 2015-04	20
Rrc02	SFINX, Paris	2006-12 to 2008-10	3
Rrc03	AMS-IX, Amsterdam	2001-12 to 2015-05	22
Rrc04	CIXP, Geneva	2001-12 to 2015-05	21
Rrc05	VIX, Vienna	2002-12 to 2015-05	19
Rrc06	Otemachi, Japan	2001-12 to 2015-05	19
Rrc07	Stockholm, Sweden	2002-12 to 2015-05	21
Rrc08	San Jose (CA), USA	2002-12 to 2004-01	5
Rrc09	Zurich, Switzerland	2003-05 to 2004-01	5
Rrc10	Milan, Italy	2004-12 to 2015-04	17
Rrc11	New York (NY), USA	2004-12 to 2015-05	19
Rrc12	Frankfurt, Germany	2004-12 to 2015-05	20
Rrc13	Moscow, Russia	2005-12 to 2015-05	17
Rrc14	Palo Alto, USA	2006-10 to 2015-05	15
Rrc15	Sao Paulo, Brazil	2005-12 to 2015-05	15
Rrc16	Miami, USA	2006-12 to 2012-03	8

Total number of data sets collected: 272
Total number of prefixes collected: 89 122 790

approach is to choose an appropriate split position. Under the condition of a single FIB, the maximum length of the prefix is fixed. However, in the case of merged FIB, the identity number is determined by the number of prefixes and the number of FIBs. It seems that there is no rule for choosing the ideal split position. However, after hundreds of experiments and statistical analysis, we found something interesting. Fig. 8 shows that when the FIB sets have the same number of FIBs, if the FIB has more prefixes, the ideal split positions then move forward. Furthermore, the best positions move backward when the number of FIBs increases. Fig. 9 shows the cumulative distribution of the ideal split position from our experiments and statistical analysis. When the

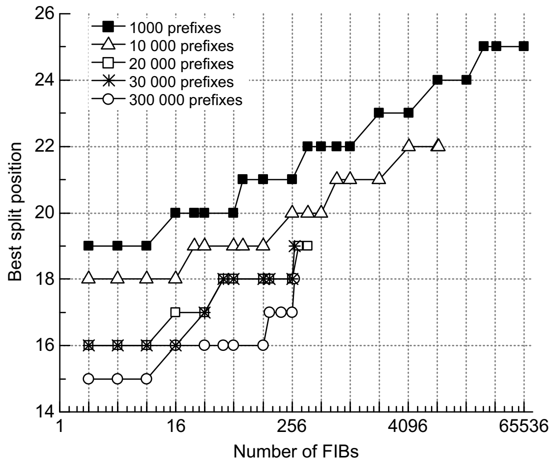


Fig. 8 The trends of the best split position

number of FIBs ranges from 2 to 512, the ideal split position mainly changes from 15 to 20. That is, when we choose the position, there is no need to try from 1 to the end; instead, we may just choose it in the range of 15 to 20, regardless of whether the number of FIBs is 2 or 512. In addition, the ideal position changes from 20 to 25 when the number of FIBs varies between 252 and 65 600. Furthermore, as long as we obtain the number of FIBs, we can select a split position according to the cumulative distribution as soon as possible, which is of great importance for the reconstruction mechanism.

Fig. 10 shows the comparison of the best split position between the unitrie and multitrie methods. As depicted, the best split position basically remains unchanged. Therefore, when we choose the split position, there is no need to build a multitrie to test, and a unitrie is enough. As presented in Fig. 11, the on-chip and off-chip memories form a sharp contrast. This is because when we choose the best position, the

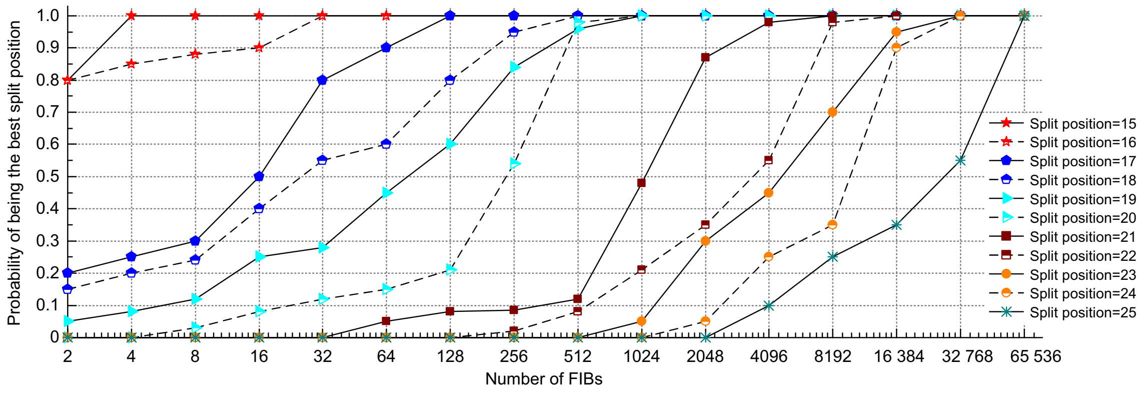


Fig. 9 The cumulative distribution of the ideal split position

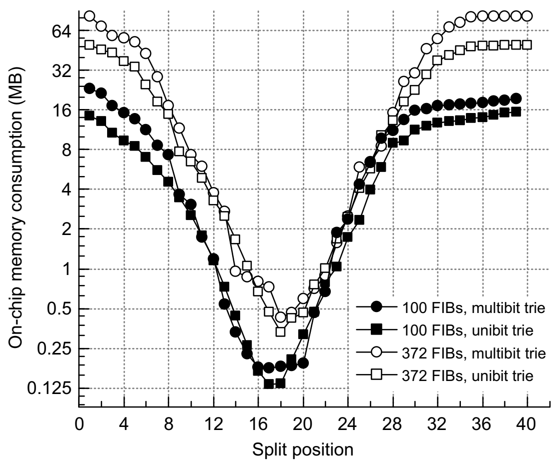


Fig. 10 The best split position of unibit and multibit tries

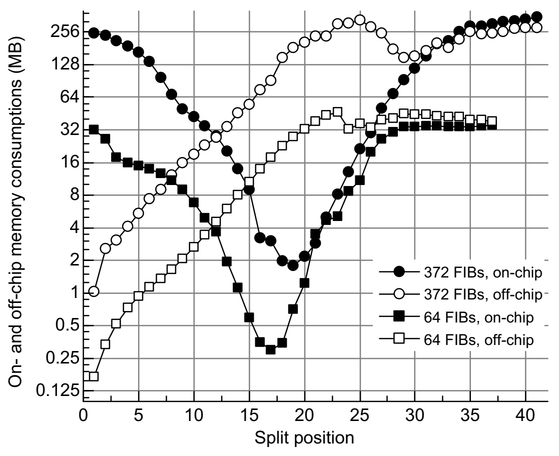


Fig. 11 On- and off-chip memory consumptions

off-chip hash mechanism that we need to deal with will be more complex and require more memory.

To compare the memory efficiency, we collected 14 experimental IPV4 (Internet Protocol Version 4) core routing tables which were used previously (Le *et al.*, 2011) and the total number of prefixes is 4 569 133. Our approach requires only 14.517 MB memory with 0.196 MB for on-chip memory (the remainder for off-chip memory); it is absolutely much better than the scheme in Le *et al.* (2011), which requires 40 MB memory. On the other hand, our approach supports 2 167 619 prefixes and just needs 7.9 MB memory, while the memory required is 18.6 MB by using the method in Le *et al.* (2011). When the number of prefixes reaches 89 122 90, the required memory increases to 766 MB far more than that of our approach, which is 109 MB.

5 Conclusions

In this paper, we proposed a simple approach to merge and compress multiple virtual routers. We introduced an identity number to each prefix based on the number of FIBs, which prevents similar prefixes sharing nodes, so that the updates are isolated. We split the merged trie by using the splitting-after-merging approach, which is memory efficient and supports fast update. Because the split position is the key of the splitting-after-merging approach, we proposed a selection strategy for the best splitting position through experiments and statistical analysis. When adding a new virtual router instance, the cumulative distribution of the ideal split position can help predict the best split position, which will shorten the detection time and accelerate the system rebuilding process.

References

Bando, M., Chao, H.J., 2010. FlashTrie: hash-based prefix-compressed trie for IP route lookup beyond 100 Gbps. Proc. IEEE INFOCOM, p.1-9. http://dx.doi.org/10.1109/INFCOM.2010.5462142

Bao, J., Chen, Y., Yu, J.S., 2010. A regeneratable dynamic differential evolution algorithm for neural networks with integer weights. *J. Zhejiang Univ. Sci. C (Comput. & Electron.)*, **11**(12):939-947. http://dx.doi.org/10.1631/jzus.C1000137

Bass, B.M., Calvignac, J.L., Heddes, M.C., *et al.*, 2005. Longest Prefix Match (LPM) Algorithm Implementation for a Network Processor. US Patent 7 383 244.

Broder, A., Mitzenmacher, M., 2001. Using multiple hash functions to improve IP lookups. Proc. IEEE INFOCOM, p.1454-1463. http://dx.doi.org/10.1109/INFCOM.2001.916641

Chan, C.Y., Ioannidis, Y.E., 1998. Bitmap index design and evaluation. Proc. ACM SIGMOD Int. Conf. on Management of Data, p.355-366. http://dx.doi.org/10.1145/276304.276336

Degermark, M., Brodnik, A., Carlsson, S., *et al.*, 1997. Small forwarding tables for fast routing lookups. *ACM SIGCOMM Comput. Commun. Rev.*, **27**(4):3-14. http://dx.doi.org/10.1145/263109.263133

Eatherton, W.N., Dittia, Z., 2003. Data Structure Using a TREE Bitmap and Method for Rapid Classification of Data in a Database. US Patent 6 728 732.

Eatherton, W., Varghese, G., Dittia, Z., 2004. Tree bitmap: hardware/software IP lookups with incremental updates. *ACM SIGCOMM Comput. Commun. Rev.*, **34**(2):97-122. http://dx.doi.org/10.1145/997150.997160

Fu, J., Rexford, J., 2008. Efficient IP-address lookup with a shared forwarding table for multiple virtual routers. ACM CoNEXT Conf., p.21. http://dx.doi.org/10.1145/1544012.1544033

Fu, Z., Wu, S.F., Huang, H., *et al.*, 2001. IPSec/VPN security policy: correctness, conflict detection, and resolution. Proc. Int. Workshop on Policies for Distributed Systems & Networks, p.39-56.

Han, B., Gopalakrishnan, V., Ji, L.S., *et al.*, 2015. Network function virtualization: challenges and opportunities for innovations. *IEEE Commun. Mag.*, **53**(2):90-97. http://dx.doi.org/10.1109/MCOM.2015.7045396

Huang, K., Xie, G.G., Li, Y.B., *et al.*, 2011. Offset addressing approach to memory-efficient IP address lookup. Proc. IEEE INFOCOM, p.306-310. http://dx.doi.org/10.1109/INFCOM.2011.5935151

Kobayashi, M., Murase, T., Kuriyama, A., 2000. A longest prefix match search engine for multi-gigabit IP processing. IEEE Int. Conf. on Communications, p.1360-1364. http://dx.doi.org/10.1109/ICC.2000.853719

Le, H., Ganegedara, T., Prasanna, V.K., 2011. Memory-efficient and scalable virtual routers using FPGA. Proc. 19th ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays, p.257-266.

Li, X.L., Wang, H.M., Guo, C.G., *et al.*, 2012. Topology awareness algorithm for virtual network mapping. *J. Zhejiang Univ.-Sci. C (Comput. & Electron.)*, **13**(3): 178-186. http://dx.doi.org/10.1631/jzus.C1100282

Li, Y.B., Zhang, D.F., Huang, K., *et al.*, 2014. A memory-efficient parallel routing lookup model with fast updates. *Comput. Commun.*, **38**(1):60-71. http://dx.doi.org/10.1016/j.comcom.2013.10.005

Liu, J., Huang, T., Chen, J.Y., *et al.*, 2011. A new algorithm based on the proximity principle for the virtual network embedding problem. *J. Zhejiang Univ. Sci. C (Comput. & Electron.)*, **12**(11):910-918. http://dx.doi.org/10.1631/jzus.C1100003

Luo, L.Y., Xie, G.G., Salamatian, K., *et al.*, 2013. A trie merging approach with incremental updates for virtual routers. Proc. IEEE INFOCOM, p.1222-1230. http://dx.doi.org/10.1109/INFCOM.2013.6566914

McKeown, N., Anderson, T., Balakrishnan, H., *et al.*, 2008. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Comput. Commun. Rev.*, **38**(2):69-74. http://dx.doi.org/10.1145/1355734.1355746

Nilsson, S., Karlsson, G., 1999. IP-address lookup using LC-tries. *IEEE J. Sel. Areas Commun.*, **17**(6):1083-1092. http://dx.doi.org/10.1109/49.772439

Rekhter, Y., Li, T., 1994. A Border Gateway Protocol 4 (BGP-4). RFC 1654, T.J. Watson Research Center & CISCO.

Richardson, N.J., Rajgopal, S., Huang, L.B., 2002. Method for Increasing Storage Capacity in a Multi-bit Trie-Based Hardware Storage Engine by Compressing the Representation of Single-Length Prefixes. US Patent 7 162 481.

Saravanan, K., Senthilkumar, A., 2015. An efficient parallel prefix matching architecture using Bloom filter for multi-bit trie IP lookup algorithm in FPGA. *Optoelectron. Adv. Mat.-Rap. Commun.*, **9**(5):803-807.

Sezer, S., Scott-Hayward, S., Chouhan, P.K., *et al.*, 2013. Are we ready for SDN? Implementation challenges for software-defined networks. *IEEE Commun. Mag.*, **51**(7):36-43. http://dx.doi.org/10.1109/MCOM.2013.6553676

Song, H.Y., Turner, J., Lockwood, J., 2005. Shape shifting tries for faster IP route lookup. IEEE Int. Conf. on Network Protocols, p.358-367. http://dx.doi.org/10.1109/ICNP.2005.36

Song, H.Y., Kodialam, M., Hao, F., *et al.*, 2009. Scalable IP lookups using shape graphs. 17th IEEE Int. Conf. on Network Protocols, p.73-82. http://dx.doi.org/10.1109/ICNP.2009.5339697

Song, H.Y., Kodialam, M., Hao, F., *et al.*, 2010. Building scalable virtual routers with trie braiding. Proc. IEEE INFOCOM, p.1-9. http://dx.doi.org/10.1109/INFCOM.2010.5461960

Song, H.Y., Kodialam, M., Hao, F., *et al.*, 2012. Efficient trie braiding in scalable virtual routers. *IEEE/ACM Trans. Netw.*, **20**(5):1489-1500. http://dx.doi.org/10.1109/TNET.2011.2181412

Srinivasan, V., Varghese, G., 1999. Fast address lookups using controlled prefix expansion. *ACM Trans. Comput. Syst.*, **17**(1):1-40. http://dx.doi.org/10.1145/296502.296503

Wang, Z., Chen, H.F., Xie, L., *et al.*, 2010. Retransmission in the network-coding-based packet network. *J. Zhejiang Univ.-Sci. C (Comput. & Electron.)*, **11**(7):544-554. http://dx.doi.org/10.1631/jzus.C0910475

Wu, K.S., Otoo, E.J., Shoshani, A., 2006. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, **31**(1):1-38. http://dx.doi.org/10.1145/1132863.1132864

Xie, G.G., He, P., Guan, H.T., *et al.*, 2011. PEARL: a programmable virtual router platform. *IEEE Commun. Mag.*, **49**(7):71-77. http://dx.doi.org/10.1109/MCOM.2011.5936157

Yu, H., Mahapatra, R., Bhuyan, L., 2009. A hash-based scalable IP lookup using Bloom and fingerprint filters. Proc. 17th IEEE Int. Conf. on Network Protocols, p.264-273. http://dx.doi.org/10.1109/ICNP.2009.5339676