LOOP: Layer-based Overlay and Optimized Polymerization for Multiple Virtual Tables

Zhian Mi*, Tong Yang*, Jianyuan Lu*, Hao Wu*, Yi Wang*, Tian Pan*, Haoyu Song† and Bin Liu*
*Tsinghua National Laboratory for Information Science and Technology
*Department of Computer Science and Technology, Tsinghua University, China †Huawei Technologies, USA
{mzaort, yangtongemail, thlujy, wuhao.thu, pig020623, platinum127, lmyujie}@gmail.com, hs1@arl.wustl.edu

***Abstract*—Network virtualization allows multiple virtual routers to coexist in the same physical router but offer independent routing services. Each virtual router needs to perform millions of lookups and thousands of updates per second to meet the requirements of high-speed Internet. The coexistence of these virtual routers intensifies scalability challenges to the routing lookup scheme: Can it scale well in storage, lookup speed and update performance as the number of virtual routers increases? In this paper, we propose Layer-based Overlay and Optimized Polymerization (LOOP) which has favorable scalability regardless of the number of virtual routers. Experiments on the general-purpose CPU show that LOOP achieves efficient storage, fast lookup, and fast incremental update. It compacts 18 FIBs with about 7M prefixes in total to only 4.6MB. One single thread can perform about 50M lookups per second on real-world traces. LOOP allows an update thread to run in parallel with lookup threads and barely interrupt them, and pure update testing indicates it can perform about 1M updates per second. One of the key advantages of LOOP is that it supports inserting and deleting virtual routers incrementally so it is ideal for fast and dynamic configuration of virtual networks.**

I. INTRODUCTION

Network virtualization [1] allows multiple logically isolated virtual networks to coexist on the same underlying physical substrate. Network virtualization abstracts physical network resources as virtual nodes and virtual links, which form virtual topologies. A physical router typically acts as multiple virtual routers in different virtual topologies. These coexistent virtual routers need to offer independent forwarding services based on their own virtual tables, *i.e.* Forwarding Information Bases (FIBs). Each virtual network is assigned a Virtual network ID (VID), so a router uses two fields (DIP, VID) in an IP packet as keys to find its forwarding port.

Virtual routing lookup schemes for multiple FIBs have great impacts on virtual network performance. They face the following challenges: *i*) *Limited Storage Resource.* As the number of FIBs and the size of each FIB increase, the storage increases accordingly. The limited storage resource in a router's line-card cannot afford the linear scaling. *ii*) *Wire-speed Lookup Requirement.* Lookups need to run at wire-speed for high-speed Internet. As a result, the storage occupied by lookup data structures needs to be small enough to fit into caches. *iii*) *Fast Incremental Update Support.* Due to the high frequency of route updates to all the FIBs aggregately, the router must support fast incremental update.

While these common challenges have been deeply researched in recent literatures, two additional issues remain poorly addressed: *iv*) *Incremental FIB Insertion and Deletion.* Network management, such as planned maintenance and new service deployment, may require dynamic creation and release of virtual routers [3]. Consequently, a routing lookup scheme needs to support incremental FIB insertion and deletion. When a FIB is inserted or deleted, the process should limit its impact on lookup operations. *v*) *Scalability on the number of FIBs.* Routing lookup schemes should scale to support hundreds of FIBs with sub-linear increase of storage. A real example comes from Juniper's NS5400 [5], which claims to support up to 503 virtual routers.

In addition, practical schemes should pay extra attention to system cost and flexibility. Hardware-based solutions can be faster than software-based solutions but often incur higher system cost and reduce flexibility. By far, most algorithm-based routing lookup schemes for virtual routers are built on prefix trie structure, *e.g.* Trie Overlap [2], Bitmap-assisted Trie [9], Trie Braiding [7], Multiroot [8], and even TCAM-based scheme [10]. These schemes merge all the FIBs into a shared prefix trie and transform the merged trie for storage compaction and small search depth. Every node in the merged trie stores an array of next-hops. However, the transformations often complicate incremental update and incremental FIB insertion and deletion.

In this paper, we propose Layer-based Overlay and Optimized Polymerization (LOOP), a software-based virtual routing lookup scheme that tackles these challenges. In the work, we take a different approach to merge multiple FIBs into the consolidated data structure. Instead of using the merged trie as the base data structure, we first compress the prefix trie of each individual FIB into bitmaps and lookup tables, and then merge the bitmaps and lookup tables of all the FIBs together. In conclusion, we made the following major contributions in this paper: 1) LOOP takes advantage of the next-hop redundancy in each FIB and the similarity among FIBs thus adopts bitmap technique for compaction. It achieves good scalability in storage, lookup speed and update performance as the number of FIBs increases. 2) LOOP inherently facilitates incremental FIB insertion and deletion to support dynamic configuration of virtual networks, and its time overhead is relatively steady regardless of the number of the current existing FIBs. 3) LOOP splits bitmap into groups and adopts group regeneration and replacement to support fast update.

This work is supported by 863 project (2013AA013502), NSFC (61073171, 61202489), the Specialized Research Fund for the Doctoral Program of Higher Education of China (20100002110051), Tsinghua University Initiative Scientific Research Program (20121080068). Corresponding Author: Bin Liu.

978-1-4799-1270-4/13/$31.00 ©2013 IEEE

Experimental results using real-world data from [16]–[18] show that LOOP tackles the above challenges. LOOP can compact 18 FIBs with about 7 million prefixes in total to 4.6MB lookup data structure, indicating each prefix occupies 0.673 bytes in average. Even though both the number of FIBs and the number of prefixes in each FIB have been increasing, the storage overhead presents a lazy growth trend because the average storage consumed by each prefix is also in decline. LOOP consumes 13$\sim$48 CPU cycles per lookup when cache hits; For real-world traces, one single thread can perform about 50 million lookups per second. Update thread can run in parallel with lookup threads and barely interrupt them, and pure update testing indicates it can perform about 1M updates per second. Moreover, LOOP supports inserting and deleting FIBs incrementally for dynamic configuration of virtual networks.

The rest of the paper is organized as follows. Section II presents a comprehensive review of the related work. Next, Section III details the data structure construction and algorithm design, and a novel bridge will be built to effectively support multiple FIBs in virtual routers. We follow this with system implementation and performance evaluation in Section IV and V, respectively. Finally, we make conclusions in Section VI.

II. Related Work

We advocate the use of software-based schemes for virtual routing lookup. In this section we comprehensively analyze various existing schemes, mainly focusing on trie-based schemes, to give an overview of the state of arts.

The existing algorithm-based schemes for virtual routing lookup are mostly built on prefix trie structure. The structure is suitable for RAM-based, FPGA-based, and TCAM-based implementation platform. Lots of efforts have been done along this line [2], [7]–[9].

Fu et al. proposed Trie Overlap in [2]. It first adopts leaf pushing [15] to ensure the correct Longest Prefix Match (LP-M) [15] and then applies level compression [11] to transform prefix trie into a trie node array. Trie Overlap achieves efficient storage but fails to support incremental update. Moreover, it is difficult to support incremental FIB insertion and deletion. When inserting and deleting FIBs, it requires rebuilding the entire data structure and re-downloading it to the search engine.

In [7], Trie Braiding allows swapping the left and right children of trie nodes to increase the structural similarity among multiple FIBs. The children rotation mechanism can efficiently improve the storage efficiency when the FIBs are dissimilar. Ganegedara et al. proposed Multiroot [8] which merges the tries at the split nodes. The sub-tries rooted at the split nodes can further be merged using Trie Overlap and Trie Braiding. Both Trie Braiding and Multiroot suffer from the same issues as Trie Overlap in terms of incremental update and incremental FIB insertion and deletion.

In order to support fast incremental update, Bitmap-assisted Trie in [9] avoids the use of leaf pushing. However, experimental results show that it requires about 11MB for the merged trie plus 20MB for the next-hop-pointer array when the number of FIBs is 18. In contrast, LOOP consumes only 4.6MB for all the data structures.

TCAM-based schemes such as [10] provide deterministic and high-speed routing lookups, but they suffer from the excessive power consumption, high cost, and low density. And its update performance is poor in the worst case.

Pipelined single-bit routing lookup schemes on FPGA platform such as [12]–[14] can achieve high lookup speed by applying pipelines. LOOP can also use pipeline for further speedup. However, these schemes require large memory bandwidth and many distributed memory blocks. Worse still, due to the lack of node sharing, their storage size scales linearly with the number of FIBs and the size of each FIB.

Inspired by Lulea algorithm [6], LOOP adopts bitmap technique for data structure compaction. However, our bitmap is different from Lulea's in three aspects: 1) Lulea adopts leaf pushing before building bitmap thus complicates the incremental update. Instead, our bitmap keeps the prefix trie unaltered but effectively ensures the correct LPM. 2) LOOP splits bitmap into groups to facilitate efficient incremental update. 3) LOOP merges the individual bitmap of each FIB to build the consolidated data structure for multiple FIBs. In summary, LOOP properly tackles the five key challenges mentioned in Section I and outperforms all the existing schemes.

III. Data Structure and Algorithm

LOOP supports a single FIB as well as multiple FIBs. First, we detail the data structure construction of the single FIB scheme. Next, we build a novel Bridge between the Single FIB scheme and the Multi-FIB scheme (BSM) across their differences in the supporting number of FIBs. Based on BSM, we elaborate how to insert and delete FIBs incrementally. Last, we present the incremental update mechanism for each virtual router. We reuse some terms in Lulea algorithm [6] that are associated with our work and redefine some of them if needed.

Lookup data structure of LOOP stems from prefix trie. When building the data structure, we conduct three transformations step by step: *i*) prefix trie to layer chunks and Forwarding Port Arrays (FPAs); *ii*) FPA to bitmap and lookup table; *iii*) bitmap to codewords. In the design of the transformation process, we focus on three performance metrics: *i*) storage efficiency, *ii*) lookup speed, and *iii*) fast incremental update.

A. Layer Chunk and Equivalent FPA

LOOP partitions the prefix trie into three layers. Figure 1 shows a normal partition mode [16-8-8]: Layer 1 covers trie levels from 0 to 16, Layer 2 from 17 to 24, and Layer 3 from 25 to 32. The partition mode is flexible and we just use [16-8-8] as an example to elaborate LOOP scheme.

A *Layer Chunk* is a small subtree confined in a layer. The original prefix trie is therefore organized as many layer chunks within the three layers. For a chunk that covers n trie levels, its equivalent FPA has 2^n elements. Each element stores either the next-hop information (forwarding port) or a pointer to a chunk in the next layer.

FPAs are built by pushing down the next-hop information of the chunks into linear arrays. We propose an efficient method based on FPA to ensure the correct LPM without resorting to the inefficient leaf pushing mechanism [15]. Figure 2 illustrates the FPA construction on a four-level chunk.

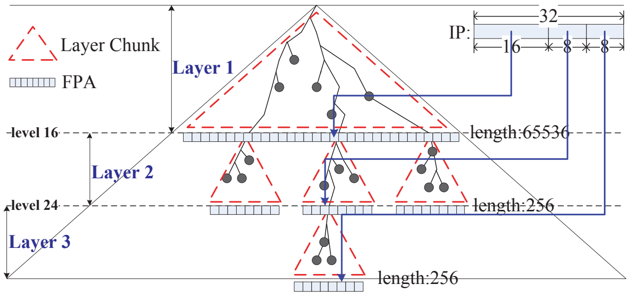


Fig. 1: Prefix trie is organized as layer chunks in partition mode [16-8-8]. Layer chucks are represented as FPAs for lookups

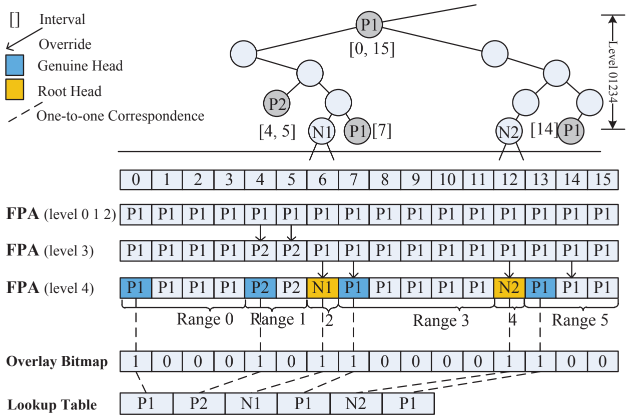
Typically, each prefix covers one or more consecutive intervals in an FPA, which means all the FPA elements in these intervals should store the same next-hop information of this prefix. In Figure 2, the prefix *P1* covers the interval [0, 15] while the longer prefix *P2* covers the interval [4, 5]. To ensure the correct LPM, the FPA elements in *P2*'s interval should store *P2*'s next-hop information. When a prefix trie is traversed in level order, the shorter prefixes are visited earlier, so that the longer prefixes can automatically override the shorter ones if their intervals overlap. Following this rule, *P2* overrides *P1* in *P2*'s interval when the 3rd level is visited. Similarly, *N1, P1, N2, P1* override the original values when the 4th level is traversed. The procedure is described in Algorithm 1 (line 1∼7).

According to the partition mode, a lookup thread uses the first 16 bits, the middle 8 bits, and the last 8 bits of a destination IP address to access the corresponding chuck FPAs from Layer 1 to Layer 3. It is possible to retrieve the forwarding port and terminate the lookup at any layer. Otherwise, the lookup thread obtains a pointer to a chunk in the next layer to continue the search. All the FPA elements of the chunks in the last layer (in this case, Layer 3) only store forwarding ports.

B. Overlay Bitmap and Lookup Table

FPAs are used to represent the layer chunks to allow efficient lookups. However, FPA contains significant horizontal redundancy: continuous FPA elements often record the same next-hop information. That is to say, FPA is divided into a series of *range*s and all the elements within the same range contain the same value. In Figure 2, the FPA has *Range 0∼5* (the pointers to the next-layer chunks are considered special ranges). Therefore, we transform FPA into *overlay bitmap* and *lookup table* to compact it. Each bit in the overlay bitmap corresponds to an FPA element. The lookup table stores FPA elements after eliminating the horizontal redundancy.

The lookup table contains one value for each *range*. FPA elements can be classified into two categories: *root head* ($e.g.$ $N1, N2$) and *genuine head* ($e.g.$ $P1, P2, P1, P1$) in Figure 2. *Root head* indicates the presence of a next-layer chunk. In this case the lookup table stores a pointer to the next-layer chunk. *Genuine head* means the common next-hop information of the range. The first bit of each range in the overlay bitmap is encoded as 1, and the other bits are encoded as 0. So there is a fixed relationship between an overlay bitmap and its corresponding lookup table: the Nth '1' (*head*) in the overlay bitmap corresponds to the Nth entry in the lookup table.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
FPA (level 0 1 2)	P1	P1	P1	P1	P1	P1	P1	P1	P1	P1	P1	P1	P1	P1	P1	P1
FPA (level 3)	P1	P1	P1	P1	P2	P2	P1	P1	P1	P1	P1	P1	P1	P1	P1	P1
FPA (level 4)	P1	P1	P1	P1	P2	P2	N1	P1	P1	P1	P1	P1	N2	P1	P1	P1
Overlay Bitmap	1	0	0	0	1	0	1	1	0	0	0	0	1	1	0	0
Lookup Table	P1	P2	N1	P1	N2	P1										

Fig. 2: FPA, overlay bitmap and lookup table for a four-level chunk

Algorithm 1 construct FPA, overlay bitmap and lookup table for an eight-level chunk.

```
Require: The chunk has a default port: $default$.
Ensure: The chunk's next-hop information is pushed down into FPA.
    And then FPA is compacted into an overlay bitmap and a lookup table.
 1: FPA[1..256] ← $default$ /*initialize FPA of $2^8 = 256$ elements*/
 2: for $level$ ← 1 to 8 do
 3:     for each prefix node $\mu$ in this level do
 4:         calculate the interval $[start, end]$ $\mu$ covers
 5:         FPA$[start..end]$ ← $nextHop[\mu]$
 6:     end for
 7: end for
 8: $previous$ ← $Null$
 9: for $i$ ← 1 to $2^8$ do
10:     if FPA$[i] = previous$ then
11:         bitmap$[i]$ ← 0
12:     else
13:         bitmap$[i]$ ← 1
14:         *lookup++ ← FPA$[i]$
15:         $previous$ ← FPA$[i]$
16:     end if
17: end for
```

Note that the overlay bitmap and the lookup table together represent the FPA of a chunk. Algorithm 1 (line 8∼17) describes the procedure. For a given IP address, it always addresses a specific bit in the bitmap. To locate the match entry in the lookup table, all one needs to do is to count the number of 1s ahead of this bit in the overlay bitmap.

We build overlay bitmap and lookup table as efficient representation of FPA. This helps to improve not only storage but also incremental updates in comparison with bitmap technique used in Lulea algorithm [6]: *i*) Generally, overlay bitmap contains less *1*s and achieves a tighter compaction. Experimental results show overlay bitmap reduces the number of *1*s by 53%∼59% in comparison with Lulea bitmap using real-world FIBs. LOOP effectively eliminates all the horizontal redundancy. *ii*) To ensure the correct LPM, some schemes require leaf pushing before building bitmaps. However, leaf pushing complicates the incremental updates. LOOP keeps the original prefix trie unaltered so that the overlay bitmaps can be rebuilt easily when new updates come.

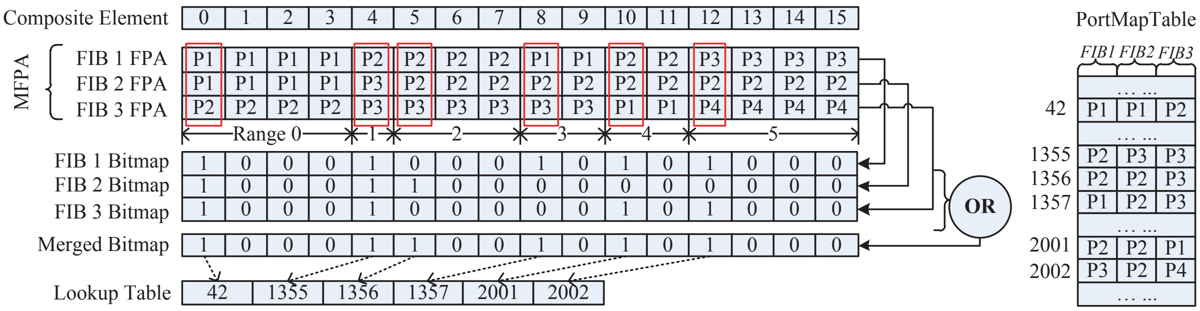


Fig. 3: BSM: MFPA and Merged Bitmap for FIB1, FIB2, FIB3

TABLE I: FIBs of 4 Virtual Routers

FIB1 VID=1		FIB2 VID=2		FIB3 VID=3		FIB4 VID=4	
****	P1	****	P2	****	P3	****	P4
01**	P2	00**	P1	00**	P2	0***	P2
101*	P2	0100	P3	101*	P1	000*	P1
11**	P3	-	-	11**	P4	11**	P3

C. A Bridge towards Multi-FIB Scheme

The Multi-FIB scheme is supposed to achieve the similar performance to the single FIB scheme regardless of the number of FIBs. The prefix similarity among different FIBs suggests the existence of vertical redundancy. To eliminate such redundancy, we build a novel Bridge between the Single FIB scheme and the Multi-FIB scheme (BSM) using bitmap technique. The bridge makes it easy to extend the single FIB scheme to support multiple FIBs. Figure 3 illustrates the BSM construction for three FIBs as shown in Table I: *FIB1*, *FIB2*, *FIB3*.

First we build three prefix tries for the three FIBs separately, and then push down their next-hop information into three independent FPAs: *FIB1 FPA*, *FIB2 FPA*, and *FIB3 FPA*. The process is the same as that in the single FIB scheme. To eliminate the vertical redundancy, the three FPAs are merged vertically into a Multiplex FPA (MFPA). MFPA has the same length as each FPA. Its ith element is the composition of the ith elements of all the FPAs ($e.g.$ $[P1P1P2]$, ..., $[P2P3P3]$, $[P2P2P3]$,...). Two composite elements are equal if and only if all pairs of corresponding FPA elements are equal. For example, the 4th composite element $[P2P3P3]$ is not equal to the 5th one $[P2P2P3]$ because the 2th pair of FPA elements $P3 \neq P2$.

Since multiple consecutive composite elements can be equal, LOOP divides MFPA into ranges based on the equality of adjacent composite elements and builds the merged bitmap for compaction. The process is similar to building the overlay bitmap in the single FIB scheme. The bit value '1' in the merged bitmap is assigned for the first composite element of each range and the bit value '0' is assigned for the other elements. It is easy to deduce that

$$MergedBitmap = Bitmap_1|Bitmap_2|...|Bitmap_n, \quad (1)$$

$$\texttt{simplified by } MB = B_1|B_2|...|B_n, \quad (2)$$

where B_i is the overlay bitmap of the ith FIB. Namely, the merged bitmap can be obtained by simply applying logic OR operations on all the overlay bitmaps.

At last, the lookup table and $PortMapTable$ are used to store the composite elements. Unlike the single FIB scheme, the entries in lookup table only record indices to $PortMapTable$, while $PortMapTable$ stores each unique existing composite element only once. This method can significantly reduce the memory consumption when the number of FIBs is large.

In summary, multiple FIBs can be efficiently represented by the merged bitmap, lookup table and $PortMapTable$. BSM extends the single FIB scheme to the multi-FIB scheme in a straightforward way and enforces their structural similarity and algorithmic consistency. The bridge brings many benefits to improve the system performance: *a*) Bitmap technique leads to a retardant growth in storage overhead, which makes the whole system more scalable. *b*) Regardless of the number of FIBs, the multi-FIB scheme requires only one more array access than the single FIB scheme to retrieve the forwarding port. *c*) Bitmap technique makes incremental updates in the multi-FIB scheme similar to the single FIB scheme. In conclusion, we build a bridge to address the two challenges troubling the multi-FIB scheme. As a result, *i*) the Multi-FIB scheme can achieve the similar performance to the single FIB scheme. *ii*) Router systems scale well as the number of FIBs increases.

D. Incremental Insertion/Deletion of FIBs

BSM merges N FIBs into consolidated data structures to support parallel lookup operations for N virtual routers. However, the dynamic creation of virtual routers brings in new FIBs to the system and the release of virtual routers removes some existing FIBs from the system to free up resources. Consequently, the multi-FIB scheme demands for incremental insertion and deletion of FIBs. BSM avoids building data structures from scratch and supports incremental insertion and deletion of FIBs. Next, we detail the changes to the main data structures caused by inserting/deleting FIBs.

`Insertion of FIBs`: Figure 4 illustrates how to insert *FIB4* (See Table I) into the existing lookup data structures generated by *FIB1*∼*3* (See Figure 3). The insertion process updates *current merged bitmap* (*MBC*), *current lookup table* (*LTC*) and *current* $PortMapTable$ (*PMTC*) to *new merged bitmap* (*MBN*), *new lookup table* (*LTN*) and *new* $PortMapTable$ (*PMTN*) incrementally.

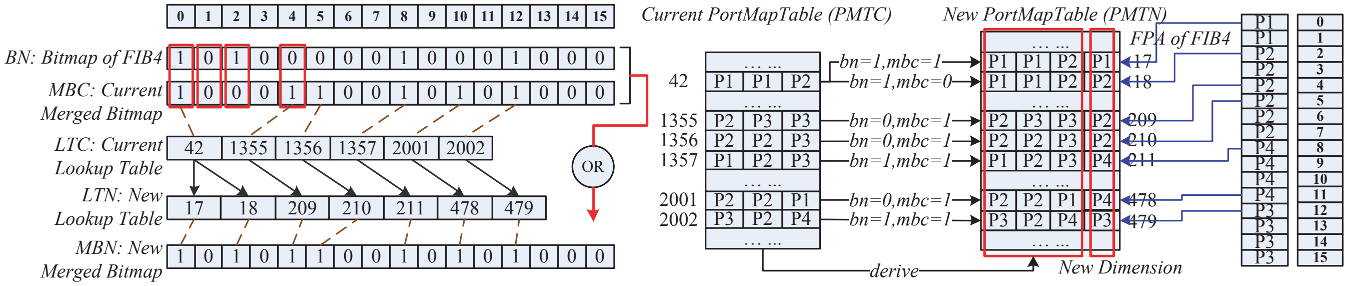


Fig. 4: Insert FIB4 into Current Structure

TABLE II: Action Table of Inserting FIB

NO.	bn	mbc	mbn	*PortMapTable*	Lookup Table
0	0	0	0	-	-
1	0	1	1	divide FPA	modify
2	1	0	1	divide MFPA	insert
3	1	1	1	extend	modify

First, the prefix trie, FPA, and bitmap (*BN*) of *FIB4* are built. Then *MBN* can be obtained directly by applying logic *OR* operation. From Formula (2), $MBC = B_1|B_2|B_3$ where B_i is the bitmap of the ith FIB. After inserting FIB4, $MBN = B_1|B_2|B_3|BN$. It can be deduced that $MBN = MBC|BN$.

PMTN increases one more dimension on the basis of *PMTC* to store the next-hop information for the new FIB. Both FPA and MFPA contains a series of ranges. When merging MFPA and FPA, the inequality in FPA may divide MFPA ranges into small ones; Reversely, the inequality in MFPA may also divide FPA ranges. Table II lists the actions that need to build *LTN* and *PMTN*. Based on these action rules, the process constructs lookup entries for all the 1s in *MBN* and generates *PMTN* entries when the port combinations are non-existent in *PMTN*. It ignores all the 0s because they do not correspond to any lookup entries or generate a new port combination. The following examples enumerate all the cases. *i*) For the 0th bit, $bn = 1$ and $mbc = 1$, so $mbn = 1|1 = 1$. Then the 42th entry in *PMTC* is `extended` to be the 17th entry of *PMTN*. The new dimension value of *PMTN* is assigned as the 0th entry in FPA ($P1$). Accordingly, the first lookup entry is `modified` to be 17. *ii*) For the 2nd bit, $bn = 1$ and $mbc = 0$ deduce $mbn = 1|0 = 1$. This indicates MFPA range is `divided`, so it produces *PMTN* entry (18) and `inserts` it into *LTN*. The new dimension value of the entry is $P2$ (the 2nd entry in FPA). *iii*) For the 4th bit, $bn = 0$ and $mbc = 1$, and the FPA range is `divided` again. In this case, it just needs to `modify` the 1355th lookup entry to be 209 with a new dimension value of $P2$ (the 4th entry in FPA) rather than insert a new entry, because the corresponding lookup entry already exists.

`Deletion of FIBs`: Insertion of FIBs divides *ranges* and increases $PortMapTable$ dimensions. On the contrary, deletion of FIBs merges *ranges* and decreases $PortMapTable$ dimensions. To update the lookup data structures, the deletion process just needs to traverse the current data structures and check whether adjacent ranges can be merged together. Assume *FIB2* is deleted from the lookup data structures generated by *FIB1*∼*3* (See Figure 3). For $Range1$ and $Range2$, the corresponding lookup entries ($[1355 : P2P3P3]$) and ($[1356 : P2P2P3]$) will become identical when the 2nd dimension is removed, so the two ranges can be merged together. Consequently, the 5th bit in *merged bitmap* changes to 0 and the lookup entry 1356 is removed.

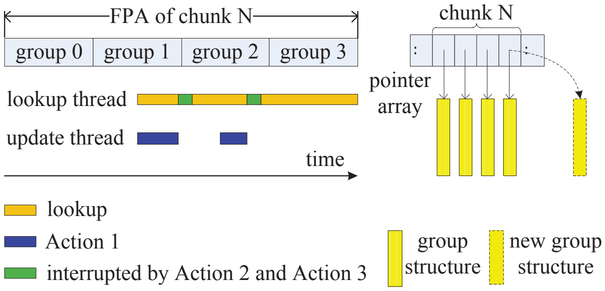


Fig. 5: Split FPA into Groups

In conclusion, both the insertion and deletion processes traverse all the chunks only once to insert and delete a FIB incrementally so that their time overhead remains constant as the number of FIBs increases. Especially, our scheme is capable of inserting or deleting several FIBs at the same time.

E. Split Bitmap for Incremental Update

The one-to-one correspondence between bitmap *1*s and lookup entries enables fast positioning, but makes incremental updates difficult. For example, assume prefix node $P2$ is deleted in Figure 2. The 4th bit in bitmap would change to be 0, and four lookup entries from the 2nd to the 5th require moving forward by one location to preserve the correct correspondence. This process requires interrupting the normal lookups and the lock time can be considerable for long bitmaps. A naive solution to avoid moving lookup entries is to rebuild the whole data structure from scratch for every update. Lookup threads continue to perform searches in the old structure until the update thread finishes building and downloading the new structure. Although this method barely interrupts lookups, it consumes too much resource and incurs large converging latency.

We observe the correspondence essentially maintains a relationship of all the lookup entries relative to the first one. Updates trigger entry movements because their positions need to be adjusted to maintain the correspondence. Both inserting and deleting one lookup entry in lookup table may lead to

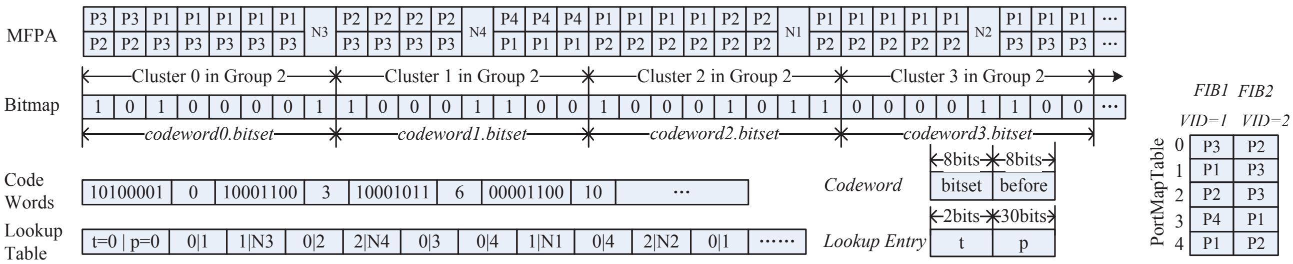


Fig. 6: Building Group Structure for the 2nd group of chunk N

the cascading entry movements. To mitigate the effect, we split the whole FPA into smaller equal-sized groups and build separated *group structure*s for them. Now the whole data structure consists of a list of independent *group structure*s. In this way, the update thread requires rebuilding and replacing just the affected *group structure*s, and leaves the other *group structure*s untouched. Figure 5 illustrates that FPA of the chunk N is split into 4 groups to build *group structure*s. When an update comes, it performs three actions: 1) rebuild the affected *group structure*, 2) download the updated group structure to line card, and 3) modify the pointer. Action 1 is the most time consuming but it works in parallel with the normal lookup thread with the old data structure. Action 2 and Action 3 would interrupt lookup thread to finalize the update but they consume negligible time. Before modifying the pointer, the lookup thread still searches in the old structure. After that, it switches to the updated structure. The update converging time and the search interruption time are both significantly reduced.

`Optimization 1 in Update`: There are three kinds of updates: announcement (add prefixes), withdrawal (remove prefixes) and path change (modify next-hop of prefixes). When an update meets three constraints: i)it belongs to announcement or path change, ii) the length of prefixes updated is 24, and iii) the prefix node is a leaf, the update thread can rebuild group structure based on a mirror copy of the current group structure. BGP update reports [4] show more than 80% updates are for announcement or path change and the prefix length is 24. Hence, the update thread can significantly improve the average update performance.

IV. System Implementation

In Section III, a prefix trie can be represented as three equivalent forms: layer chunk, MFPA, and bitmap/lookup table. We combine these forms into the final data structure for storage and speed tradeoff.

In our demonstration, we partition a prefix trie into layer chunks with mode [16-8-8] and transform them into MFPAs. Layer 1 has only one 16-level-high chunk and its MFPA has $2^{16} = 65536$ elements. The other chunks in Layer 2 and Layer 3 are all 8 level high and each of them has $2^8 = 256$ MFPA elements. To build independent *group structure*s, we split each MFPA into groups, with each containing 64 MFPA elements ($i.e.$ *group size = 64*). Therefore Layer 1 chunk has $2^{group1} = 1024$ groups and Layer 2 and 3 chunks have $2^{group2} = 2^{group3} = 4$ groups each. The IP address can be partitioned into three bit groups, $group1$, $group2$, and $group3$, which can be used to locate the target *group structure* directly.

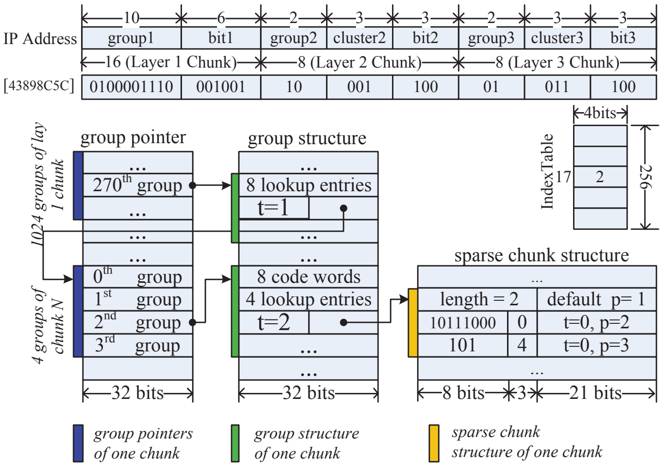


Fig. 7: Lookup Data Structure

In Figure 6, a $lookup\ entry$ consists of t (2 bits) and p (30 bits), occupying a total of 4 bytes. There are three entry types: i) $t = 0$, p stores an index to $PortMapTable$. ii) $t = 1$, p stores a pointer to a next-layer chunk. iii) $t = 2$, p stores a pointer to a sparse chunk which will be described later. We build *group structure*s as follows:

i) Group structure in Layer 1 stores 64 lookup entries corresponding to 64 MFPA elements in a group without any compression. Because Layer 1 has only one chunk and all lookups need to search from Layer 1, we sacrifice a little extra storage for fast lookups.

ii) For the group in Layer 2 and Layer 3, we use the bitmap/lookup table to build *group structure*. Figure 6 illustrates how to build the 2nd *group structure* of the chunk N. Each group (64 MFPA elements) has $2^{cluster2} = 8$ clusters and each cluster contains $2^{bit2} = 8$ elements. Eight $codeword$s record the information of the bitmap of the group and the lookup table stores a list of lookup entries. One $codeword$ occupies 2 bytes and stores 8 bits in the bitmap. The total 8 $codeword$s correspond to 8 clusters one by one. The upper 8 bits ($bitset$) of $codeword$ store the 8 bits of the bitmap in the cluster and the lower 8 bits ($before$) record the number of 1s ahead of this cluster in the group. In the example group, [10100001, 0], [10001100, 3] and [10001011, 6] are the first three $codeword$s. The number “3” in the 2nd $codeword$ is the number of 1s in the 1st $codeword.bitset$, “10100001”; The number “6” in the 3rd $codeword$ is the total number of 1s in the first two $codeword.bitsets$, “10100001” and

"10001100". The 8-bit $bitset$ has 2^8 possible combinations. To facilitate counting the number of *1*s in each cluster, a $2^8 \times 4bits = 128bytes\ IndexTable$ is used to record the number of 1s for each combination. For example, the 38th (00100110) item in $IndexTable$ is 3 and the 20th (00010100) item is 2. We use an example to illustrate how we use $IndexTable$ to calculate the number of 1s up to any bit location. To calculate the number of 1s in the first 5 bits of "10100001", we right shift the string by $8 - 5 = 3$ bits and get the new string "00010100" $= 20$, so the 20th entry in the $IndexTable$ stores the result (which is 2). Using $codeword$s and $IndexTable$, we first retrieve $codeword.before$ of the cluster to get the number of 1s ahead of the cluster, and then count the number of 1s ahead of the given location in the cluster. Therefore, a lookup thread can count the number of 1s ahead of a certain location in bitmap and retrieve the corresponding lookup entry quickly through computing rather than sequential searching.

`Optimization 2 in Storage`: Many chunks in Layer 2 and Layer 3 contain only a few prefixes. When the number of prefixes is less than a predefined value K ($e.g.$ K = 3 in our demonstration), the chunk is denoted as a *sparse chunk* and can be represented in a special way. In Figure 7, a *sparse chunk* records a *number*, a *default* and an array of *special entries*. The *number* indicates the number of *special entries*. The *default* records the default next-hop information of the chunk. Every *special entry* contains *prefix*, *shift*, *t*, and *p*. For example, the prefix (0101****, P4) can be represented as ([$prefix$: 00000101], [$shift$: 4], [t: 0], and [p: 3]). When an IP address comes, a lookup thread first right shifts the IP address by *shift* and then makes a comparison against *prefix* for match. For example, $[01101101] >> 4 \neq [00000101]$, so the address [01101101] does not match the *special entry*. The address [01011100] would match because $[01011100] >> 4 = [00000101]$. Since $t = 0$, p stores an index to $PortMapTable$ (p occupies 19 bits here). *Special entries* are arranged in a non-descending order by *shift* to ensure the correct LPM. When a lookup thread searches the *special entries* in order, the first match will be the longest match so the next-hop information or a pointer to the next-layer chunk can be retrieved. If there is no match for all the special entries, *default* will return as the index to $PortMapTable$. When a lookup thread accesses a *sparse chunk*, the time overhead varies depending on the number of *special entries*. The worst-case scenario occurs when a *sparse chunk* contains K *special entries* and the match fails after searching all of them. In practice, K needs to be set to a small value for fast lookup.

Figure 7 describes the lookup procedure. A lookup starts from Layer 1 chunk and searches down to the next-layer chunks until the next-hop information is found. Assume a lookup thread will search an IP address [0x43898C5C] and VID = 1. It first extracts $group1$ ([0100001110] = 270) and $bit1$ ([001001] = 9) from the IP address directly and then accesses the 9th lookup entry in the 270th *group structure* of Layer 1 chunk. The lookup thread gets a pointer to the chunk N in Layer 2. The chunk is not sparse, so it obtains $group2$ = "10" $= 2$, $cluster2$ = "001" $= 1$ and $bit2$ = "100" $= 4$" from the IP address. Then $group2$ and $cluster2$ together locate the 1st $codeword$ [10001100, 3] in "Group 2". The $codeword.before = 3$ indicates the number of 1s ahead of the cluster. $IndexTable[codeword.bitset >> (7 - bit2)]$ stores the number of 1s ahead of the target location within the cluster. Since $3 + IndexTable[$"10001"$] = 5$, the 5th lookup entry in the lookup table is retrieved (which is $N4$). The lookup thread continues to access the Layer 3 chunk in the same way as the layer 2 chunk. Because the type of $N4$ is 2, the lookup thread gets a pointer to a sparse chunk structure. The sparse chunk contains 2 special entries and the 2nd entry matches the last 8 address bits [01010000]. The lookup thread then access $PortMapTable$ using (index = 3, VID = 1) to get the result (which is $P4$).

V. Performance Evaluation

A. Experimental Setup

To evaluate LOOP in real-world environments, all the data (route tables, update packages, and IP packet traces) are downloaded from Router Views Archive Project [16], RIPE Network Coordination Center [17], and CAIDA equinix-sanjose [18]. Figure 8 shows the core parameters of these experimental data sets: table names (FIB names), the numbers of prefixes and next-hops in each FIB, and the length distribution of updated prefixes. Our CPU specification is Intel(R) Xeon(R) CPU E5520 quad core processor (2.26GHz) and each of the cores has an L1 and L2 cache (4×256KB), and an additional 8MB of shared L3 cache. Our software build environment is visual C++ 2010 (64 Bits).

In the section, we compare LOOP with several existing schemes: Merged Trie, Trie Overlap [2], Trie Braiding [7], and Bitmap-assisted Trie [9]. In our implementations, we adopt $PortMapTable$ structure for all these schemes to store their next-hop arrays. We choose the partition mode [16-8-8] and $group\ size = 64$ for LOOP unless stated otherwise.

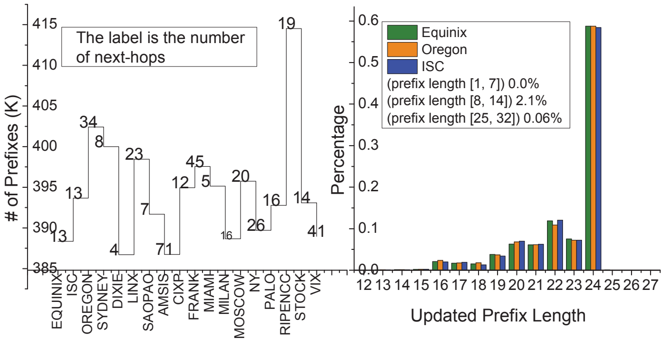


Fig. 8: Experimental Data Sets: FIBs and updated prefixes (Jan, 2012)

B. Storage

LOOP's storage size is $S_{loop} = S_{fix} + S_{pmt}$, where S_{fix} is the total size of pointers, lookup entries, codewords, sparse chunks and $IndexTable$, and S_{pmt} is the size of $PortMapTable$. Figure 9 shows the growth trends of S_{fix}, S_{pmt}, and S_{loop} as the number of FIBs increases. *Number of FIBs* N means the first N FIBs in Figure 8 are used for the virtual routers. S_{fix} is determined by the storage size of all *sparse chunks* (sp) and the numbers of *pointers* ($p : 4bytes$), *codewords* ($cw : 2bytes$), and *lookup entries* ($e : 4bytes$) ($i.e.$ $S_{fix} = p \times 4 + cw \times 2 + e \times 4 + sp + 128$). All the

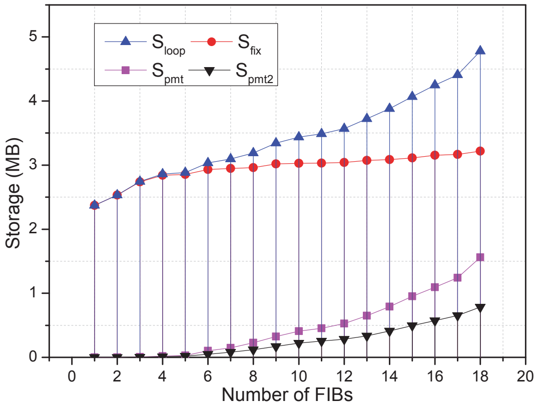


Fig. 9: Growth Trends of S_{fix}, S_{pmt} and S_{loop}

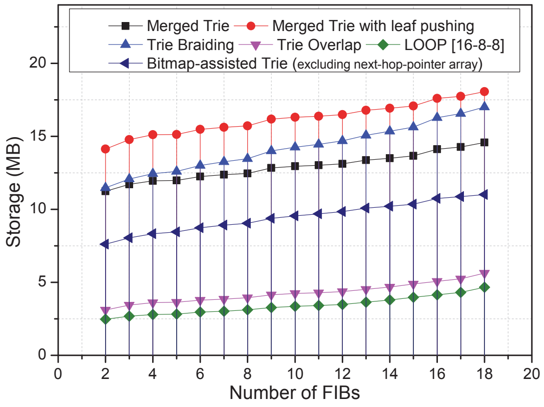


Fig. 10: Storage Comparison

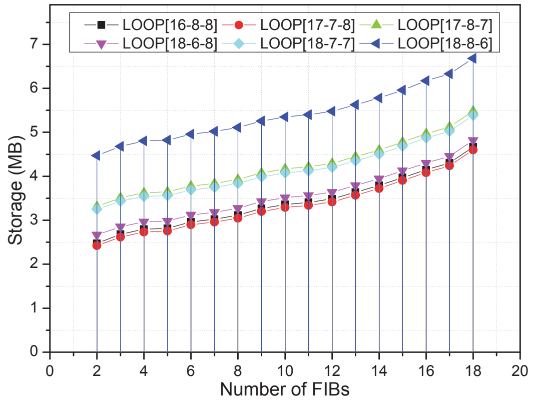


Fig. 11: Storage Overhead using different partition mode

parameters (p, cw, e, sp) are positively correlated to the number of chunks but the number increases moderately thanks to the prefix similarity among FIBs. As a result, S_{fix} grows slowly as the number of FIBs increases.

The main storage growth attributes to S_{pmt}. $S_{pmt} = t \times fq \times 1$ where t is the number of $PortMapTable$ entries, fq is the number of FIBs (*i.e.* the number of virtual routers) and *1* byte can support up to 256 next-hops. As the number of FIBs increases, t grows fast and fq enlarges linearly. S_{pmt} contributes 32.66% of S_{loop} when *Number of FIBs* $N = 18$. S_{pmt} will further contribute larger percentage of S_{loop}; meanwhile the information redundancy of $PortMapTable$ is also on the rise. Based on this observation, we can also employ bitmap technique to compress $PortMapTable$. In Figure 9, S_{pmt2} is the size of $PortMapTable$ after bitmap compression. When *Number of FIBs* $N = 18$, S_{pmt2} is about 50% of S_{pmt}. With even larger number of FIBs, $PortMapTable$ contains more entries but also contains more information redundancy, so S_{pmt2} grows much slower than S_{pmt}.

Figure 10 shows the storage comparison among several schemes. Bitmap-assisted Trie stores a bitmap in each node so its trie structure is smaller than Merged Trie. However, it needs a next-hop-pointer table to store the next-hop information, which is about 20MB When *Number of FIBs* $N = 18$. Leaf pushing generates more prefix nodes thus increases the storage size of Merged Trie. Compared with Merged Trie, LOOP's storage is about 3$\sim$4 times smaller. LOOP partitions a prefix trie into three layers and employs bitmap compression. In contrast, Trie Overlap adopts level compression but partitions the prefix trie into more layers. Hence, LOOP's storage overhead is about 20% less than Trie Overlap. Trie Braiding is used to reduce the number of nodes in Merged Trie by transforming the dissimilar prefix tries for different FIBs. It is a technique to preprocess prefix tries for other trie-based schemes. However, the tables we downloaded present strong prefix similarity, so Trie Braiding does not help to reduce the storage.

Figure 11 shows the impact of partition modes on storage. For different partition modes, the storage overheads are almost the same when their total levels of Layer 1 and Layer 2 are equal. We choose the partition mode [16-8-8] for efficient storage and memory alignment. In LOOP, all the 18 FIBs with about 7M prefixes in total are compacted to about 4.6MB. This translates to just 0.673B per prefix in average. Even if both the number of FIBs and the number of prefixes in each FIB keep growing, the storage of LOOP presents a retardant growth trend because the average storage consumed by each prefix is also in decline.

C. Lookup

LOOP partitions a prefix trie into three layers and searches from Layer 1 down to Layer 3. A lookup thread may get and return the next-hop information in any layer. A lookup thread requires two memory accesses for one $group\ pointer$ and one $lookup\ entry$ in Layer 1. When a Layer 2 chunk is not sparse, it consumes four memory accesses for one $group\ pointer$, one $codeword$, one $lookup\ entry$, and $IndexTable$. Otherwise, it accesses a sparse chunk structure of at most 16 bytes, requiring at most two memory accesses in a 64-bit system. Layer 3 is the same as Layer 2. Therefore, a lookup thread requires 2, $4 \sim 6$ $(2+2 \sim 2+4)$, and $6 \sim 10$ $(4+2 \sim 6+4)$ memory accesses to get the final lookup entry when it returns in Layer 1, Layer 2, and Layer 3, respectively. The final lookup entry stores an index to $PortMapTable$, so it requires another memory access to get the forwarding port.

When sparse limit $K = 3$, there are 41 possible cases of lookup instruction streams (Layer 1: 1, Layer 2: $1{+}2{\times}K{+}1 = 8$, Layer 3: $(1+K) \times 8 = 32$). For a given IP address, a lookup thread executes one of the 41 instruction streams.

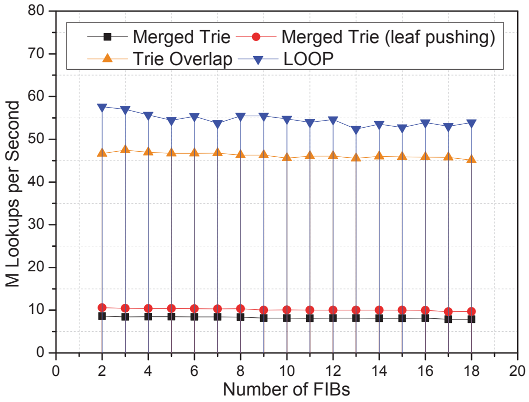


Fig. 12: Lookup Speed of Software Schemes

As the instruction streams are deterministic, cache behavior strongly impacts lookup speed. Tracing all the 41 instruction streams, our experiment shows that LOOP requires 13∼48 CPU cycles per lookup when cache hits. Most of lookups get the next-hop information in Layer 2 and consume 23∼30 CPU cycles. Figure 12 shows the lookup speed when packet traces in [18] are fed in. In Merged Trie, a lookup thread searches from level 1 down to a leaf node. Merged Trie with leaf pushing avoids backtracking and increases lookup speed slightly. Merged Trie has a large search depth and requires too many memory accesses. Bitmap-assisted Trie based on pure software and basic Trie Braiding are both slower than Merged Trie because of extra bit operations. Typically, Bitmap-assisted Trie uses pipeline technique for speedup and Trie Braiding combines with other schemes to improve both storage and lookup performance. LOOP can perform about 50M lookups per second, which is about 4∼5 times faster than Merged Trie. LOOP and Trie Overlap constrain the search depth of lookup thread by using layer chunks and by using level compression, respectively. Generally, Trie Overlap requires more search depths than LOOP. Compared with Trie Overlap, LOOP increases the lookup speed by about 15%.

D. Update

We use two metrics to evaluate an update algorithm. One is *Dead Time*, which represents the update induced interrupt period during which the normal lookups are prohibited. The other is *Execution Time*, which represents the time consumed by the update algorithm itself to update data structure. The former is crucial for router QoS (Quality of Service) because longer *Dead Time* leads to more packets buffering in queue and even causes packet loss. The latter has direct impact on the network state convergence time.

The update performance of LOOP relates to the number N of *group structures* and the number M of *sparse chunks* changed by the updated prefix. In LOOP, *Execution Time* is the time to rebuild N *group structures* and M *sparse chunks*, and *Dead Time* is the time to download the new structure to line card and modify $N+M$ pointers. Since most of updated prefixes are distributed in Layer 2 chunks and Layer 3 chunks, N and M are small. We explore *Execution Time* and *Dead Time* separately using real-world FIBs and their update packages. The experiment shows that our update mechanism barely interrupts lookups.

TABLE III: *Dead Time*: Distribution of the Number of Modified Pointers

Modified Pointers ($N+M$)	1	2	≥ 3
Oregon (group size = 64)	98.34%	0.65%	1.01%
Equinix (group size = 64)	98.94%	0.53%	0.53%
Isc (group size = 64)	99.81%	0.12%	0.07%

TABLE IV: *Group size* affects storage and update

Group Size	32	64	128
Bytes per prefix	5.449	4.713	4.356
Cycles per update	1313	1732	2533

Execution Time: Figure 13 shows *Execution Time* one by one for thousands of updated prefixes in Jan 1, 2012. On average, each updated prefix consumes 1732 cycles in Oregon, 1390 cycles in Equinix, and 1473 cycles in Isc, respectively. To rebuild *group structure*, *Execution Time* ranges from 750 to 3000 cycles, depending on prefix distribution in the group. The number can drop down to about several hundred cycles when an updated prefix satisfies the three constraints of `Optimization 1`. It takes less time to rebuild *sparse chunk* than to rebuild *group structure*. There are some updated prefixes that change several *group structures* or *sparse chunks* so they cause spikes of *Execution Time* as shown in Figure 13.

Dead Time: Table III shows the distribution of the number of modified pointers ($N+M$) by updated prefixes. More than 98% updated prefixes change only one *group structure* or *sparse chunk*, which means $N=1$ and $M=0$, or $N=0$ and $M=1$. Generally, a *sparse chunk* occupies at most 16 bytes and the *group structure* of Layer 2 and Layer 3 occupies about 60∼150 bytes. Therefore, in most cases the *Dead Time* is extremely short.

The $group\ size$ impacts both storage and update performance. A smaller group size leads to faster updates but results in larger storage. With different group size of 32, 64, and 128, Table IV lists the storage overhead and update speed with a single Oregon FIB to exemplify the effect of group size. It shows that the storage decreases by 13.5% and the update speed slows down by 31% as the group size doubles from 32 to 64. When the group size doubles again, the storage further reduces by 7.6% but the update speed also decreases sharply by 46%. For performance and storage tradeoff, we choose $group\ size = 64$.

Among these schemes, Merged Trie has the fastest update speed. After leaf pushing, incremental update becomes more difficult. Trie Braiding and Trie Overlap optimize the data structure globally using tree isomorphism and level compression techniques, respectively. A local update may break the global optimization. Moreover, Trie Overlap employs leaf pushing before level compression, so its update becomes more complex. By contrast, LOOP can perform 1.3 ∼ 1.6 million updates per second. The incremental update thread runs in parallel with the lookup threads and barely interrupts the normal lookups. In conclusion, LOOP exhibits the favorable incremental update capability and performance.

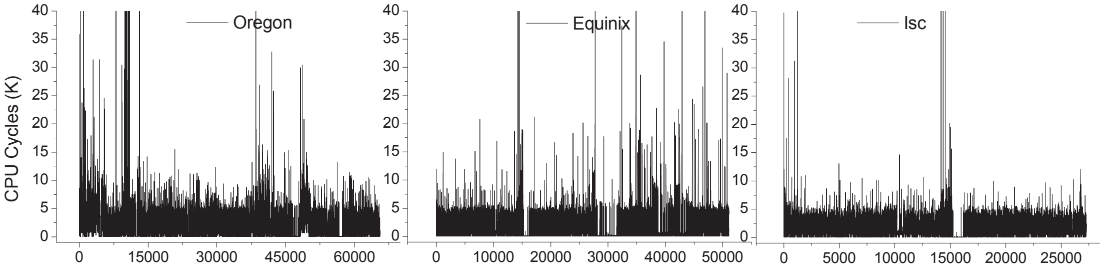


Fig. 13: *Execution Time*: Update Prefixes in Oregon, Equinix, and Isc One by One

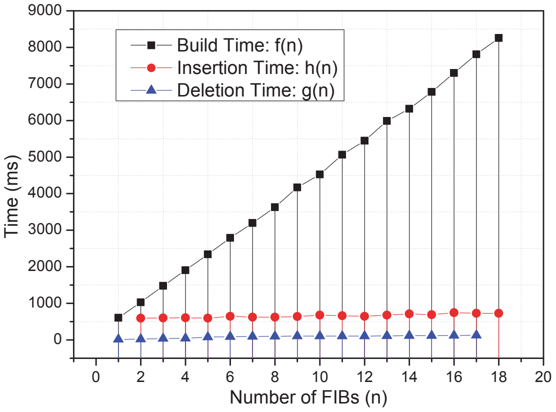


Fig. 14: Time Overhead of Inserting and Deleting FIBs

E. *Incremental Insertion/Deletion of FIBs*

LOOP inherently supports building lookup data structure incrementally without needing to merge all the FIBs from scratch. Similarly, Merged Trie can also support incremental insertion and deletion of FIBs if it reserves space in each node. By contrast, Trie Overlap requires loading all the FIBs and building all the data structure from scratch. In Figure 14, the $f(n)$ is the time overhead to build the data structure for the first n FIBs in Figure 8. Assume the current data structure supports the first $n + 1$ FIBs. The $g(n)$ is the time overhead to delete the $(n + 1)$th FIB and generate data structure for the first n FIBs. Likewise, the $h(n)$ is the time overhead to insert the nth FIB incrementally into the data structure of the first $n - 1$ FIBs. Both $g(n)$ and $h(n)$ scale well as the number of FIBs increases. This feature will make a big difference when hundreds of virtual routers coexist in the same physical router.

VI. Conclusions

In this paper, we have proposed LOOP, an innovative IP routing lookup scheme, which supports a single FIB and multiple FIBs. The multi-FIB scheme can achieve the close performance to that of the single FIB scheme. LOOP scales well in storage, lookup speed, and update performance as the number of FIBs increases. Overlay bitmap avoids transforming prefix trie and efficiently eliminates information redundancy of next-hops. In addition, LOOP inherently facilitates incremental insertion and deletion of FIBs to support dynamic configuration of virtual networks. Our update mechanism splits the whole bitmap into fine-grained groups and adopts group regeneration and replacement to reduce update converging time and the search interruption time, which not only benefits incremental updates but also effectively supports parallel lookup and update operations. Experiments on real-world FIBs demonstrate that LOOP achieves high lookup speed, fast incremental update with low cost, compact storage, and good storage scalability.

References

[1] N.M. M. Chowdhury, and R. Boutaba. A survey of network virtualization. Computer Networks, 862–876, 2010.

[2] J. Fu, and J. Rexford. Efficient IP-Address Lookup with a Shared Forwarding Table for Multiple Virtual Routers. In *Proc. ACM CoNEXT*, 2008.

[3] Yi Wang, E. Keller, B. Biskeborn, Jacobus. Merwe, and J. Rexford. Virtual Routers on the Move: Live Router Migration as a Network-Management Primitive. In *Proc. ACM SIGCOMM*, 231–242, 2008.

[4] BGP Routing Table Analysis Reports (AS65000). [Online] Available: http://bgp.potaroo.net/.

[5] NetScreen Series (NS5400). [Online] Available: www.juniper.net/us/en/products-services/security/netscreen/ns5400/.

[6] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. In *Proc. ACM SIGCOMM*, 3–14, 1997.

[7] H. Song, M. Kodialam, F. Hao, and T. V. Lakshman. Building scalable virtual routers with trie braiding. In *Proc. IEEE INFOCOM*, 2010.

[8] T. Ganegedara, W. Jiang, and V. Prasanna. Multiroot: Towards memory-effcient router virtualization. In *Proc. IEEE ICC*, 1–5, 2011.

[9] L. Luo, G. Xie, K. Salamatian, S. Uhlig, L. Mathy, and Y. Xie. A Trie Merging Approach with Incremental Updates for Virtual Routers. In *Proc. IEEE INFOCOM*, 1246–1254, 2013.

[10] L. Luo, G. Xie, S. Uhlig, L. Mathy, K. Salamatian, and Y. Xie. Towards TCAM-based Scalable Virtual Routers. In *Proc. ACM CoNEXT*, 2012.

[11] S. Nilsson, and G. Karlsson. IP-Address Lookup Using LC-Tries. *IEEE JSAC*, 17(6):1083–92, June 1999.

[12] G. Lu, Y. Shi, C. Guo, and Y. Zhang. CAFE: A Configurable pAcket Forwarding Engine for Data Center Networks. In *Proc. ACM SIGCOMM Workshop on Programmable Routers for the Extensible Services of Tomorrow*, 25–30, 2009.

[13] M. B. Anwer, and N. Feamster. Building a Fast, Virtualized Data Plane with Programmable Hardware. In *Proc. ACM SIGCOMM Workshop on Virtualized Infrastructure Systems and Architectures*, 2009.

[14] M. B. Anwer, M. Motiwala, M. Tariq, and N. Feamster. SwitchBlade: A platform for rapid deployment of network protocols on programmable hardware. In *Proc. ACM SIGCOMM*, 183–194, 2010.

[15] V. Srinivasan, and G. Varghese. Fast Address Lookups Using Controlled Prefix Expansion. *ACM Trans. Computer Systems*, October 1999.

[16] University of Oregon Route Views Archive Project. [Online] Available: http://archive.routeviews.org/.

[17] RIPE NCC: RIPE Network Coordination Centre. [Online] Available: http://www.ripe.net/.

[18] CAIDA Anonymized Internet Trace. [Online] Available: http://www.caida.org/data/monitors/passive-equinix-sanjose.xml.