OBMA: Minimizing Bitmap Data Structure with Fast and Uninterrupted Update Processing

Chuwen Zhang*, Yong Feng*, Haoyu Song†, Ying Wan*, Wenquan Xu*,
Yilun Wang*, Huichen Dai*, Yang Li*, Bin Liu*
*Department of Computer Science and Technology, Tsinghua University
‡Futurewei Technologies, Santa Clara

***Abstract*—Software-based IP route lookup is one of the key components in Software Defined Networks. To address challenges on density, power and cost, Commodity CPU is preferred over other platforms to run lookup algorithms. As network functions become richer and more dynamic, route updates are more frequent. Unfortunately, previous works put less effort on fast incremental updates. On the other hand, The cache in CPU could be a performance limiter due to its small size, which requires algorithm designers to give high priority on storage efficiency in addition to time complexity. In this paper, we propose a new route lookup algorithm, OBMA, which improves update performance and storage efficiency while maintaining high lookup speed. The extensive experiments over real-word traces show that OBMA reduces the memory footprint to just 4.52 bytes/prefix, supports update speed up to 7.2 M/s which is 12.5 times faster than the state-of-the-art algorithm Poptrie. Besides, OBMA achieves up to 195.87 Mpps lookup speed with a single thread. Tests on comprehensive performance of lookup and update show that OBMA can sustain high lookup speed with update speed increasing.**

I. INTRODUCTION

In Software Defined Networks (SDN), more and more network functions adopt software-based approaches in order to provide an open programming environment and gain update flexibility. Route lookup is a key function in routers. CPU-based lookup solutions can speed up the time-to-market, extend product life cycle, and reduce system cost. Today, backbone route tables maintain an annual growth rate of around 15%. Meanwhile, the update rate grows steadily and shows a strong burst characteristic. While software-based algorithms are attractive, they face some new challenges.

First, commodity CPUs, especially those embedded low power CPUs, are equipped with a small cache which can be a performance limiter. Therefore, we should give high priority to the algorithm's storage efficiency because of the small cache with CPUs. Second, measurements show that the route updates exhibit a strong burst characteristic. Paper [1] suggests a rising trend of update burst in Internet, reaching up to hundreds of kilo-prefixes extremely. If a router fails to process several concurrently arriving update bursts quickly, more time spent on update processing can deteriorate the slow BGP convergence, and the update prefixes waiting in the queue can increase the packet loss rate. Particularly, paper [1] finds that routers often see bursts of withdrawals and 84% of the bursts include prefixes announced by "popular" ASes, which means poor performance on update will influence customers' Internet experience significantly and cause huge potential economic losses.

Most of the existing CPU-based software lookup algorithms are derived from the binary trie data structure [2]–[4]. Theoretically, binary trie [2] allows the fastest update speed, but its memory efficiency and lookup performance are poor. While the common challenges such as boosting lookup speed and reducing memory cost have been well studied in recent literatures [3]–[6], optimizing bursty updates is largely unattended. In addition, CPU-based software algorithms can adopt multi-threading technology to accelerate lookup speed, but memory access operations which can't be parallelized will become the bottleneck and limit the improvement on lookup. Therefore, while keeping small storage and high lookup speed, we should pay more attention to the update processing.

In this paper, we propose Overlay BitMap Algorithm (OBMA) to address this issue. Different from Lulea [5], OBMA builds an overlay bitmap structure which keeps the prefix trie unaltered and ensures the correct Longest Prefix Match (LPM) processing. This change brings us the following benefits: 1) better support for incremental updates; 2) minimizing the number of "1" in the bitmaps which allows a better compression ratio and improves the storage efficiency. With further optimizations on the update structure, OBMA achieves a fast and uninterrupted update processing, which means it can adapt to the bursty updates in Internet. Specifically, we make the following contributions in this paper:

1) We propose OBMA which minimizes the bitmap structure by canceling the redundant bit "1" and presents high storage efficiency;
2) We design an adaptive bitmap segmentation algorithm which partitions the whole bitmap data structure into groups based on the update frequency. This enables updates on a fine-grained group and thus reduces the network state convergence time;
3) We conduct extensive experiments to evaluate OBMA's performance. Experimental results on newest real-world data sets show that OBMA reduces the memory footprint to just 4.52 bytes/prefix, achieves up to 195.87 Mpps lookup speed on single-threading, and supports up to 7.20 M/s update speed. Tests on comprehensive performance of lookup and update show OBMA can keep high lookup speed over a wide range of update speeds and achieve the highest lookup speed after update speed over 0.1 M/s.

978-1-5386-2542-2/18/$31.00

The rest of the paper is organized as follows. Section II details the overlay bitmap-based algorithm and its data structure. Section III describes the optimizations on update processing. Section IV conducts experiments to evaluate the performance of OBMA. Section V surveys the related work. Finally, Section VI concludes our work.

II. MINIMIZING BITMAP DATA STRUCTURE

In this section, we first describe OBMA. Then we elaborate its software implementation by a [18-6-8] three-layer example. We reuse some terms in Lulea [5] and redefine some as needed.

A. Minimizing the number of "1"s in the overlay bitmap

Similar to Lulea, the construction of the lookup structure of OBMA consists of three steps: 1) Construct a prefix trie from the original routing table; 2) Build bitmaps and the corresponding lookup tables based on the prefix trie; 3) Generate bitmap code-words. OBMA's major difference from Lulea lies in step 2, in which Lulea uses leaf pushing on the prefix trie to build bitmaps, while OBMA uses level traversal to build Forwarding Port Arrays (FPA) and overlay bitmaps. We use the example routing table in Figure 1 to show the two bitmap generation approaches. The corresponding prefix trie in Figure 1 is cut on level 3 and levels 0 to 3 are grouped into layer 1. The intermediate nodes on the cut level are named pointer nodes which will store pointers to the chunks in the next layer.

The process of level traversal is shown in the right of Figure 2. Each prefix covers a range of elements in an FPA. That is, every element in the range inherits the same output port information of the prefix. In the right of Figure 2, node P2 on level 1 covers range [0, 3], while its child node P1 covers range [2, 3]. To conduct LPM, the nodes within the range of P1 (i.e., the 2nd and 3rd elements of the FPA on level 2) should record the output port information of P1. While building the FPAs using level traversal, shorter prefixes are accessed earlier and long prefixes automatically cover the range of short ones. Hence, level traversal naturally guarantees LPM.

Although FPA can replace prefix trie for fast route lookups, its storage is still large due to horizontal bit redundancy. (i.e., consecutive elements may store the same output port information). To eliminate such redundancy, we compress the bitmaps and convert each FPA into an overlay bitmap plus a corresponding lookup table. We define a segment as the maximum range in which all elements are identical. The bit corresponding to the head element of each segment is set to 1 and the others 0. The lookup table only stores the head elements of each segment in order of appearance. Therefore, The n-th "1" in the bitmap corresponds to the n-th entry of the lookup table. Given a destination address, the lookup process first locates a bit in a bitmap, then counts the number of "1"s up to this bit position, and finally gets the next port in a lookup table using the number as index.

In contrast, Lulea generates bitmaps in three steps as shown in the left of Figure 2: 1) Modify the prefix trie to a complete tree via leaf pushing and only leaf nodes and pointer nodes

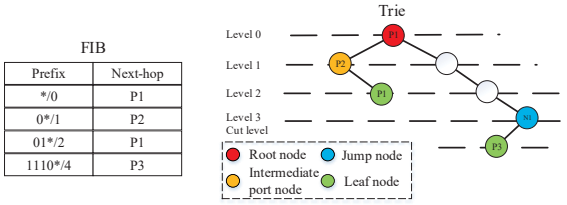


Fig. 1: Route table and prefix trie

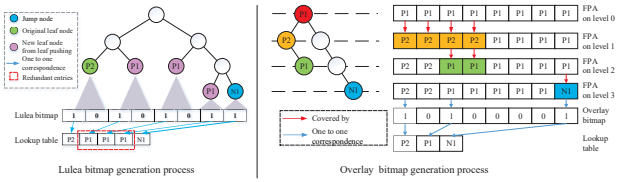


Fig. 2: Overlay bitmap generation process

contain port information; 2) Traverse all leaf nodes and pointer nodes from left to right. When visiting a node, project the node to the cut level. This step generates an equivalent FPA; 3) Set the bits corresponding to the left nodes of each projection range to "1" and others to "0". Meanwhile, store the port information of leaf nodes or the pointers of pointer nodes into the corresponding lookup table. Clearly, horizontal redundancy exists in Lulea bitmaps as shown in Figure 2. Overlay bitmap can eliminate such redundancy, so the number of "1"s is fewer, and we can indeed prove that overlay bitmap contains the minimum number of "1"s among all trie-based algorithms under a given bitmap.The proof is omitted due to space.

B. Detailed implementation of overlay bitmap

Compared with a Lulea bitmap, an overlay bitmap is not only storage-efficient but also easy to update. The reasons are as follows: 1) Level traversal merges the adjacent FPA elements with the same port into one element, leading to fewer "1"s in overlay bitmaps and smaller table storage cost; 2) Lulea modifies the prefix trie via leaf pushing to ensure LPM, but OBMA keeps the prefix trie structure unchanged. It is convenient for updates. 2) Lulea uses leaf pushing to ensure LPM, but leaf pushing changes the original structure of the prefix trie which complicates trie updates. An important feature of OBMA is to keep the prefix trie structure unchanged. So, when an update comes, the prefix trie modification and the bitmap reconstruction are easier.

OBMA uses a [18-6-8] partition to split the 32-bit IP address space into three layers. To improve the lookup speed, we adopt the direct pointing technology as in [3]. OBMA maintains one FPA on level 18, named Direct Pointing Array (DPA), which takes 512KB (i.e., $2^{18} \times 2$B) storage. Lookup process uses the first 18 bits, the middle 6 bits, and the last 8 bits of an IP address to access Layer 1, Layer 2, and Layer 3, respectively.

For convenience, we define some terms in the lookup structure of OBMA as follows:

Chunk: An 6-level (8-level) subtree in layer 2 (layer 3). A chuck can be classified as a sparse chunk or a bitmap chunk based on the number of prefixes in it.

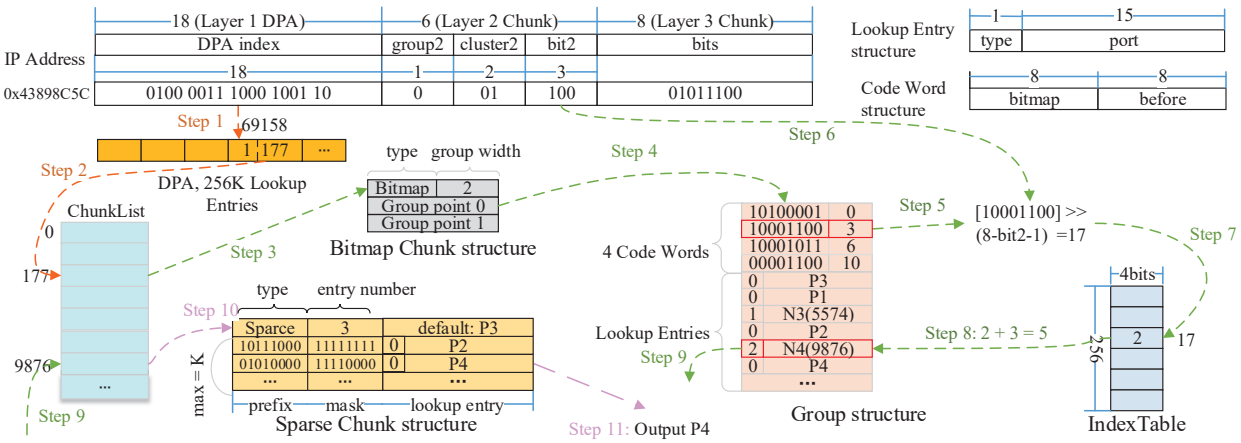


Fig. 3: OBMA lookup structure and process

Group: A w-level subtree, where w is in the range of 0-3 (0-5) in layer 2 (layer 3). Group is the basic unit for updates and lookups. A chunk is composed of 2^{3-w} (2^{5-w}) groups in layer 2 (layer 3).

Cluster: A 3-level subtree, the minimum unit for bitmaps.

Next, we illustrate the lookup structure of chunk, group, and cluster. We start from bitmap chunk and sparse chunk. As shown in Figure 3, a bitmap chunk consists of the following fields: 1) *type*, indicating the chunk type: either a bitmap chunk or a sparse chunk; 2) *group width*, indicating how many bits in the middle of the IP address are group index, and giving the number of groups contained in the chunk i.e. 2^{group_width}; 3) *group list*, containing a pointer to each group. When looking up in a bitmap chunk, OBMA needs to split the 6- or 8-bit IP address into ($group\_index$, $cluster\_index$, bit_index). Since the height of the cluster is 3, the lowest three bits are used as bit index. The values of the other two parts are determined by the group width field. For example, in Figure 3, the group width equals 1, which means a [1-2-3] partition for layer two. Therefore, $group\_index = (0)_2 = 0$, $cluster\_index = (01)_2 = 1$, and $bit\_index = (100)_2 = 4$.

The group structure consists of several *code-words* and *lookup entries* as shown in Figure 3. The width of each lookup entry is 2 bytes, including a *type* field (1 bits) and a *port* field (15 bits). When *type* is 1, the *port* field stores an index of chunk list; when *type* is 0, the *port* field stores the port information. The size of each *code-word* is also 2 bytes: one byte stores the cluster *bitmap* and the other byte (i.e., the *before* field) records the cumulative number of "1"s ahead of this cluster. As an example in Figure 3, the *before* field of the third *code-word* is 6, which means the number of "1"s in the bitmaps of the first and the second *code-word* is 6. Therefore, we can get the number of "1"s in previous clusters efficiently. To calculate the number of "1"s ahead of a given location in a cluster, we use a pre-calculated *Index Table* to store the number of "1"s for $2^8 = 256$ different bit patterns. For example, the 20th record of the *Index Table* stores the number of "1"s of $(00010100)_2$, which is 2. To calculate the number of "1"s in the first five bits of a bitmap (10100001), we first shift the bitmap to right by $8 - 5 = 3$ bits to obtain a new value $(00010100)_2 = 20$, and then retrieve the 20th record of the *Index Table* to get the result of 2.

When the number of prefixes in a chunk is less than a predefined value K (e.g., $K = 4$), we use the sparse chunk structure to save storage. Each *sparse entry* consists of three parts: a *prefix*, a *mask*, and a *lookup entry*. For prefix matching, the destination address executes logic AND operation with the *mask* and compares the results with the *prefix*.

Algorithm 1 describes the OBMA lookup step by step. We combine the example in Figure 3 to illustrate the process. Given an IP address $0x43898C5C$, OBMA uses its first 18 bits to visit the DPA (step 1) and gets a chunk list index. Next, OBMA locates the bitmap chunk in layer 2 in step 2 and 3. According to the *group width* field, OBMA splits the middle section of the IP address, visits the corresponding group structure, and gets the lookup entry in step 4-8. Since the entry is a pointer, OBMA keeps searching and locates the sparse chunk in layer 3 in step 9 and 10. Finally, after searching in a sparse chunk, the next port P4 is returned.

Algorithm 1 Lookup Algorithm

Input: $Chunklist$, DPA, IP.
Output: $port$.

$entry \leftarrow DPA[IP[31:14]]$
if $entry$ stores next port information **then**
  return $entry.port$
if $ChunkList[cur].type$ is $BITMAP$ **then**
  $width \leftarrow ChunkList[cur].groupwidth$
  Get $group2, cluster2, bit2$ from $IP[13:8]$
  $groupbase \leftarrow ChunkList[cur].grouplist[group2]$
  $codeword \leftarrow groupbase[cluster2]$
  $lookup \leftarrow groupbase + 1 << (3 - width)$
  $ix \leftarrow codeword.bits >> (7 - bit2)$
  $pix \leftarrow codeword.before + IndexTable[ix] - 1$
  $entry \leftarrow lookup[pix]$
  if $entry.type$ stores next port information **then**
    return $entry.port$
  else
    $cur \leftarrow entry.pointer$
else
  search the special entries $(0 \rightarrow K)$
/*search deep to layer 3 chunk just as layer 2 chunk*/

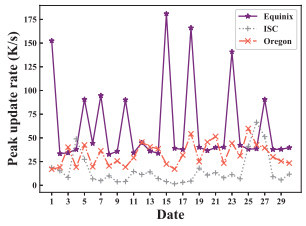


Fig. 4: Peak update statistics for Equinix, Oregon and ISC in April, 2017

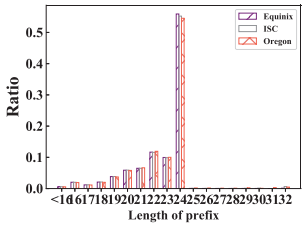


Fig. 5: Length distribution of updated prefixes on April 1st, 2017

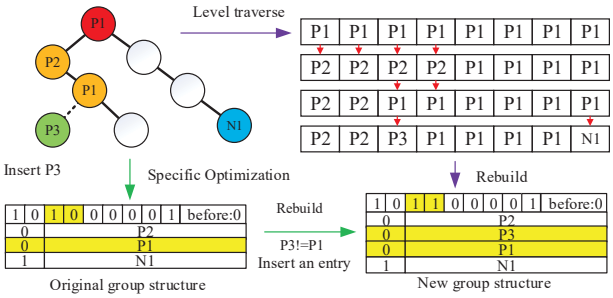


Fig. 6: Specific Optimization v.s. level traverse

III. FAST UPDATE PROCESSING

First, we examine the bursty update phenomenon through mining real-word update traces. Then, based on the update features, we propose an adaptive group partition approach to narrow the scope of level traversal as well as reduce the whole memory storage, and specific optimization for insertions or modifications to non-pointer nodes at the bottom of each layer.

A. A closer look at bursty updates

This part reports the data mining results on the update traces of three route tables: Oregon, Equinix, and ISC (downloaded from [7]–[9], from April 1 to April 30, 2017). The updates exhibit the characteristics of strong burst and regular locality.

Figure 4 shows the peak update statistics for the three tables: Equinix, Oregon, and ISC. We can see the updates are bursty. The rates of the top two bursts for Equinix are 181.0K/s and 166.1 K/s, respectively. The peak update rates are 66.5 K/s and 59.8 K/s for ISC and Oregon. We observe that the bursty updates exist throughout the whole period of time.

We count two kinds of update locality: locality on prefixes (in Figure 5) and on chunks. We find: 1) The update on prefix length 24 accounts for about 50% of the total updates. The top three prefix lengths with the highest update frequency are 24, 22, and 23, accounting for more than 70% of the total updates; 2) The update ratios on prefix length ≤ 18 for Equinix, ISC, and Oregon are all around 6%; 3) The update ratios on chunks show strong locality: 20% of chunks receive over 80% of updates.

B. Adaptive Bitmap Segmentation

The aforementioned results show the update locality: 20% of chunks receive over 80% of updates. This inspires us to develop an adaptive bitmap segmentation algorithm. Based on the update frequency, each chunk can be partitioned into a different number of equal-sized groups. Group partitioning is flexible: the smaller the group is, the fewer bytes the reconstruction needs and the faster the update is, at the cost of more storage of group pointers. As a result, the frequently updated chunks end up with having more small-size groups while the infrequently updated chunks can contain as few as just one group.

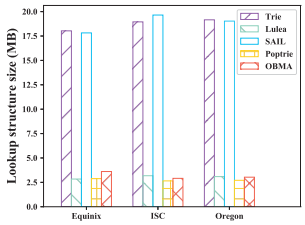
As the value of *group width* is between 0 to N, where N is 3 in layer 2 and 5 in layer 3, we can divide the group size into up to $N+1$ levels according to the update frequency. Let level i cover update frequency range $[0, f_i^{\max})$ and $f_{i+1}^{\max} = 2f_i^{\max}$. Here we apply the idea of dynamic table adjustment based on load factor [10] in which the table size is doubled whenever the load factor reaches 1 and the table size is reduced to half whenever the load factor drops below 1/2. This algorithm is composed of two processes: online addition and half-decay.

Online addition process: We maintain a counter for each chunk and let it increase by one when receiving an updated prefix. If the counter value of a chunk reaches the current level's upper limit of frequency $f_i^{\max}$, this update will double the group number and rebuild the entire chunk.

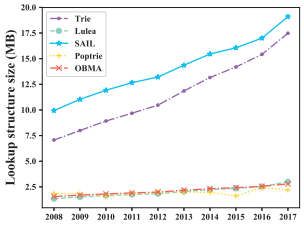
Half-decay process: We define one day as a half-life cycle. When the network traffic is relatively low, we start the half-decay process: traverse all chunks and examine the chunk counters. If a counter value is smaller than $1/2f_i^{\max}$ (i.e., half of its current upper limit), we reduce the group number and the counter value to a half and rebuild the entire chunk.

C. Specific Optimization

Level traversal can be expensive in terms of CPU cycles. If we can skip it, the update performance will be improved. Specific optimization is based on the fact that for the inserted or modified prefixes, if the length is 24 and the corresponding node in the prefix tree is not a pointer node, the reconstruction of the group structure can be done on the old group structure directly without needing to traverse the prefix trie. Our statistic results show that 99.9% of updates with the prefix length of 24 meet the conditions of the specific optimization (i.e., nearly half of the updates are suitable for the specific optimization based on the distribution of prefix length in Figure 5). We use the previous trie structure in Figure 6 as an example to illustrate the specific optimization. When dealing with an updated prefix [010/3, P3], OBMA directly visits the group structure and finds the two-bit bitmap "10" and the corresponding port P1 in Figure 6. The bitmap "10" means that the node to be modified and its right neighbor both point to P1. We can get the left neighbor's port P2 by visiting the previous entry of P1. As port P3 does not equal left port P2 and right port P1, this update will change the second bit 0 to 1 and add a new entry of P3 ahead of P1. The new group structure is the same as that from the level traversal but it is acquired in a more efficient way. The actual situations can be more complex. We need to consider both the local bitmaps and the cluster's position. Due to space limitations, we do not explain these in detail.



(a) Lookup Table Sizes



(b) Memory size increase over the last 10 years on ISC

Fig. 7: The performance on memory cost of the five schemes

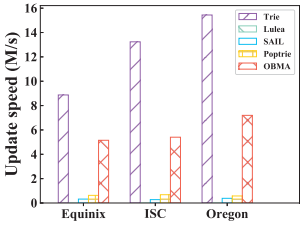


Fig. 8: Comparison on update execution speed for the five algorithms

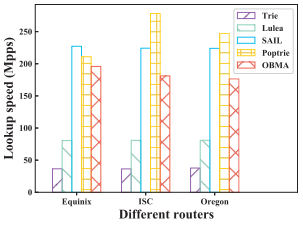


Fig. 9: Lookup speed comparison for the five algorithms

IV. PEFFORMNACE EVALUATION

We evaluate OBMA's performance on memory, lookup and update using real-world Internet traces.

Platform: We conduct experiments on a Dell M4800 mobile workstation with Intel CPU Core i7-4900MQ (4x 2.8 GHz cores with each supporting 2 threads). The workstation is equipped with 8 GB DDR3 (1.6 GHz) memory and runs 64-bit Ubuntu-14.04-LTS OS.

Datasets: We download the route table history from three routers (i.e., Oregon, Equinix, and ISC) since 2008 to 2017 [7], [9] to evaluate the memory efficiency and the memory size increase in section V-B. We use CAIDA traces [8] to test the lookup speed in section V-C. We collect update packets from RIPE Network Coordination Center [7] to test update speed in section V-D. In this section, we compare OBMA with two state-of-the-art works: SAIL [4] and Poptrie [3]. We also compare with Lulea [5] and the basic binary trie [2]. Our implementation chooses the partition mode [18-6-8] and uses the adaptive group segmentation algorithm.

A. Memory Cost and Memory Size Increase

1) Memory Cost: We run the algorithms on three route table snapshots: Oregon, Equinix, and ISC on April 1st, 2017 to get the lookup structure sizes for OBMA, Poptrie, SAIL, Lulea, and binary trie. 697K, 657K, and 675K prefixes, respectively. Figure 7(a) shows the lookup structure size comparison for the five algorithms. We can see that OBMA needs approximately equal storage with Lulea, which is a bit higher than Poptrie, while SAIL's size is comparable to the binary trie. Considering the DPA level in Lulea is 16 (about 400 KB less than level 18), OBMA is more compact than Lulea. OBMA can compress tables even smaller than Poptrie by raising the adaptive grouping threshold but it will affect the update performance significantly. The lookup structure sizes for SAIL, Poptrie and OBMA on ISC are 18.95 MB, 2.65 MB and 2.91 MB, respectively. We use the metric of bytes per prefix to calculate the storage efficiency(e.g., it requires only 4.52 Bytes/prefix for the ISC table on April 1st, 2017), and the result is that OBMA achieves a high storage efficiency.

2) Memory Size Increase: We run the algorithms on the ISC tables for the past 10 years. The memory size increase trend of the five algorithms is shown in Figure 7(b). We can see that OBMA keeps a small storage space. We would like to point out in particular that Poptrie needs extra memory including the whole lookup data structure to handle updates. Hence OBMA is still much better than Poptrie's solution considering the overall storage overhead.

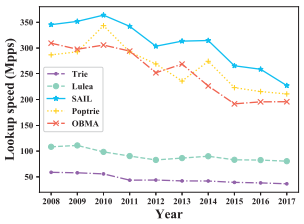


Fig. 10: Lookup speed change over the last 10 years on Equinix tables

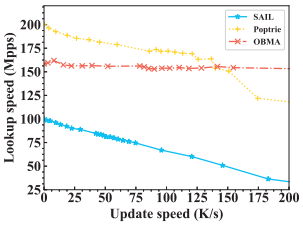


Fig. 11: The comprehensive performance of lookup and update on single-threading

B. Lookup performance

We run the algorithms on three route table snapshots, Oregon, Equinix, and ISC on April 1st, 2017, to get the lookup speeds, as shown in Figure 9. Although SAIL exhibits the fastest speed on Equinix, it needs a large storage. Poptrie is the highest on ISC and Oregon, but much slower on Equinix, refecting its strong correlation with the specific structure of the routing table. OBMA can achieve up to 195.87 Mpps lookup speed for the Equinix table on April 1st, 2017, which means it is enough to support small-packet line-speed forwarding for a 100 Gb/s link.

We continue to run the algorithms on the route tables over the last 10 years and collect the lookup speeds in Figure 10. SAIL has the highest lookup speed, but its speed drop is also the largest (e.g., about 140 Mpps from 2010 to 2017). SAIL, Poptrie and OBMA all exhibit a decline trend, while Poptrie fluctuates more intensely over the years. This implies all of the three algorithm, especially Poptrie, are sensitive to the table size and table structures.

C. Update Performance

We run experiments to measure the average update speed and show the results in Figure 8. Just as expected, binary trie has the fastest update speed which can be considered as an upper bound. Lulea cannot handle incremental updates efficiently. OBMA is significantly better than Poptrie and

TABLE I: Update performance of OBMA using different optimization means

	Storage (MB)	Lookup (Mpps)	Update (M/s)
Firm Group (0-3-3)	2.39	198.35	3.16
Firm Group (1-2-3)	2.95	195.83	4.56
Firm Group (2-1-3)	4.07	189.55	5.41
Firm Group (3-0-3)	6.16	185.04	5.77
Adaptive group	3.61	195.87	3.72
Adaptive group$^-$			5.15

SAIL for all the three tables. For the Oregon table on April 1st, 2017, OBMA can sustain 7.19 M/s update speed while Poptrie can only sustain 0.58 M/s update speed and SAIL sustains 0.38 M/s.

We also do experiments on the update performance using different optimization means. Table I shows the update performance of OBMA when using different optimization means. As for firm grouping, more groups means faster update performance but lower lookup performance and more storage cost. E.g., adopting 8-group chunk (3-0-3) can achieve 5.77 M/s update speed in contrast to 3.16 M/s of 1-group chunk (0-3-3), but the latter has advantages in lookup and storage. The Adaptive bitmap segmentation and the specific optimization can optimize the comprehensive performance on lookup, update and storage. If we do not adopt specific optimization, the update speed will drop by 28% to 3.72 M/s.

D. Comprehensive performance of lookup and update

The real routers should be able to deal with lookup and update traces simultaneously, so we first run experiments to figure out the comprehensive performance of lookup and update on single-threading. Figure 11 shows the results of Equinix. The lookup speed of SAIL begins at about 100 Mpps, which is far below the pure lookup results, due to the large memory footprint needed by update operations. The trends of OBMA are more steady and horizontal, which cross with Poptrie in the update speed interval of 100-150 K/s. As burst update speed in Internet can exceed 150 K/s mentioned before, we can claim that OBMA is more powerful than SAIL and Poptrie when dealing with bursty updates.

V. RELATED WORK

For saving space, we skip the survey on hardware-based solutions and focus on software-based lookup schemes. Basically, software-based schemes are divided into two categories: bloom filter-based and trie-based [11]. Bloom filter cannot deal with incremental element withdrawal, thus counting bloom filter is introduced [12], [13]. However, counting bloom filter suffers from the false negative as well as false positive issues. Comparatively, trie-based algorithms are more practical, including [3]–[6], [14]. Lulea [5] is a typical bitmap compression algorithm which effectively reduces the memory footprint, but the leaf-pushing changes the original trie structure which complicates the update operation. Further, leaf-pushing brings more redundant "1" bits than level traversal of OBMA. [6] achieves fast lookups and rapid routing updates, but takes a large storage space. LOOP [14] focuses on merging multiple virtual route tables and partitions the bitmap into fixed equal-sized groups to enable fast updates, but it does not adapt to the burst updates. SAIL [4] can achieve fast lookup speed, but it heavily relies on the traffic locality. Besides, SAIL occupies a large storage space too. Poptrie [3] develops a multiway method to accelerate lookup and adopts a pointer replacement approach to implement fast updates. However, it costs a large amount of memory space to reserve double lookup structures and cannot still support burst updates.

VI. CONCLUSION

We propose OBMA to adapt to burst updates in Internet and realizes fast and uninterrupted update processing, which is important to accelerate the network state convergence and improve network QoS. Meanwhile, OBMA is a CPU-based route lookup solution that costs low memory footprint and sustains high lookup speed. Because OBMA is derived from prefix trie, it can be applied to IPv6 networks and multiple virtual route tables in an NFV environment.

ACKNOWLEDGEMENT

This work is sponsored by Huawei Innovation Research Program (HIRP), NSFC (61373143, 61432009).

REFERENCES

[1] T. Holterbach, S. Vissicchio, A. Dainotti, and L. Vanbever, "Swift: Predictive fast reroute," in *Proc. of ACM SIGCOMM*, 2017, pp. 460–473.

[2] K. Sklower, "A tree-based packet routing table for berkeley unix." in *USENIX Winter*, 1991, pp. 93–99.

[3] H. Asai and Y. Ohara, "Poptrie: A compressed trie with population count for fast and scalable software ip routing table lookup," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, 2015, pp. 57–70.

[4] T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy, "Guarantee ip lookup performance with fib explosion," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, 2014, pp. 39–50.

[5] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," in *Proc. of ACM SIGCOMM*, vol. 27, no. 4, 1997.

[6] T. Yang, Z. Mi, R. Duan, X. Guo, J. Lu, S. Zhang, X. Sun, and B. Liu, "An ultra-fast universal incremental update algorithm for trie-based routing lookup," in *Proc. of IEEE ICNP*, 2012, pp. 1–10.

[7] Ripe ncc:ripe network coordination centre. [Online]. Available: http://www.ripe.net/

[8] Caida anonymized internet trace. [Online]. Available: http://www.caida.org/data/monitors/passive-equinix-sanjose.xml

[9] University of oregon route views archive project. [Online]. Available: http://archive.routeviews.org/

[10] T. H. Cormen, *Introduction to algorithms*. MIT press, 2009.

[11] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, "Survey and taxonomy of ip address lookup algorithms," *IEEE network*, vol. 15, no. 2, pp. 8–23, 2001.

[12] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using bloom filters," in *Proc. of ACM SIGCOMM*, 2003, pp. 201–212.

[13] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended bloom filter: an aid to network processing," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 4, pp. 181–192, 2005.

[14] Z. Mi, T. Yang, J. Lu, H. Wu, Y. Wang, T. Pan, H. Song, and B. Liu, "Loop: Layer-based overlay and optimized polymerization for multiple virtual tables," in *Proc. of IEEE ICNP*, 2013, pp. 1–10.