OBMA: Scalable Route Lookups with Fast and Zero-interrupt Updates

Chuwen Zhang, Yong Feng, Haoyu Song, *Senior Member, IEEE,* Ying Wan, Wenquan Xu, and Bin Liu, *Senior Member, IEEE*

***Abstract*—Software-based IP route lookup is a key component for packet forwarding in Software Defined Networks. Running lookup algorithms on commodity CPU is flexible and scalable, which shows advantages on cost and power consumption over the hardware-based forwarding engines. However, dynamic network functions and services make route updates more frequent than ever. Existing algorithms often fall short of the incremental update requirements. In this paper, we propose the Overlay BitMap Algorithm (OBMA) family, which supports extraordinary update performance while maintaining their lookup speed and storage efficiency the highest in class. Starting from the basic OBMA_B, we develop two variations with different tradeoffs for different application scenarios. OBMA_L supports faster lookups than OBMA_B at a small cost of update speed. OBMA_S achieves better storage efficiency than OBMA_B at a small cost of lookup throughput. We run our algorithms on commodity CPU and evaluate them with real-world route tables and traces. The experiments show that OBMA family realizes the lowest memory footprint, over 200 Mpps lookup throughput, and highest update speed. Specifically, OBMA_S can reduce the memory footprint to 3.98 bytes/prefix, saving 25.33% over the state-of-the-art Poptrie; OBMA_L can support 252.02 Mpps lookup throughput with a single thread, significantly outperforming the state-of-the-art Poptrie and SAIL, and its lookup throughput exceeds 600 Mpps with multiple parallel threads in a single CPU; OBMA_B supports updates at a rate of 14.58M updates/s, which is 15 times faster than Poptrie. Our tests also show that the update process hardly interfere with the lookup process for the OBMA family, and zero-interrupt to lookups with multiple threads can be achieved.**

***Index Terms*—IP lookup, Bitmap, Parallel Techniques.**

I. Introduction

In Software Defined Networks (SDN), more and more network functions adopt software-based approaches for better flexibility and scalability in an open programming environment. As a key function in routers, route lookup is no exception. A software-based route lookup solution is expected to be able to speed up time-to-market, extend product life cycle, and reduce system cost. As of today, the size of the backbone route tables grows at a rate of around 15% annually. Meanwhile, the table update rate also grows and the updates tend to be more bursty. Paper [1] uncovers a rising trend of update bursts which mainly consist of withdrawals from remote outages on Internet. In the extreme, a burst involves hundred-thousands of prefixes.

The software-based algorithms must address some new challenges.

First, network device vendors want to have the similar compact-sized line-card (the thickness) as the hardware-based solutions (e.g., using Network Processors), which enforces strict constraints on the line-card form factor and power budget. Therefore, power-hungry components, such as Graphics Processing Unit (GPU) or integrated PC server board with colossal heat sink, are not favored. Instead, commodity CPUs, especially those embedded low-power CPUs, are preferred. However, such CPUs are usually equipped with a small cache which can potentially be a performance limiter. Hence, while optimizing the algorithm's time complexity, we need to give high priority to improve the algorithm's storage efficiency as well, which is measured in bytes per prefix.

Second, we need to counter the impact of update bursts. A router that fails to handle update bursts fast enough faces two negative consequences: 1) The long time spent on updating slows down the route convergence and prolongs the network downtime; 2) The queued updates delay the valid alternative paths to take effect in the Forwarding Information Base (FIB) and create transient forwarding black holes. Paper [1] reports that routers often experience bursts of route withdrawals and 84% of the bursts include prefixes announced by "popular" ASes. Poor update performance influences customers' Internet experience and causes significant economic loss.

Existing software-based route lookup algorithms are predominately derived from the binary trie data structure [2, 3, 4]. A native trie [3] allows fast updates, but its memory efficiency and lookup performance are mediocre. The past efforts focused on boosting lookup speed and reducing memory footprint of a trie at the cost of complicating the update process [2, 4, 5, 6]. While we can adopt the multi-threading technology to further accelerate lookups, the update process needs to block the lookup threads when modifying the lookup data structure, which can severely impede lookups. Therefore, we need to pay equal attention on the update process in order to retain the theoretical performance gain through lookup optimizations.

In this paper, we propose the Overlay BitMap Algorithm (OBMA) family to address the above mentioned issues. The basic OBMA (OBMA_B) is small-cache friendly and bursty-update tolerable. In contrast to traditional bitmap-based algorithms such as Lulea [5] (which is usually considered the most compact bit-map based lookup algorithm), OBMA_B adopts an overlay bitmap to keep the prefix trie unaltered and ensure

C. Zhang is a Ph.D. Candidate in the Department of Computer Science, Tsinghua University. (email: chuwen1992@gmails.com, zhang-cw15@mails.tsinghua.edu.cn)

Y. Feng, Y. Wan, W. Xu,, and B. Liu are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. (Corresponding Author: Bin Liu. email: liub@mail.tsinghua.edu.cn)

H. Song is with Futurewei Technologies, USA

the correctness of Longest Prefix Match (LPM). The proposed overlay bitmap is more compact and enables fast incremental updates. By trading off some update throughput, OBMA_L boosts the lookup throughput. OBMA_S further compresses the data structure and improves the memory efficiency, which make it suitable for application scenarios with memory constraints.

Specifically, we make the following major contributions:

1) We propose OBMA_B which realizes high performance on storage, lookup and update. It uses an overlay bitmap to eliminate the redundant bit "1" in the bitmap horizontally. The resulting compressed lookup structure improves the lookup speed by exploiting the CPU cache. It uses the adaptive grouping and 24-level optimization to reduce the update execution time. We show how it can support updates with zero-interrupt to lookups when using multiple threads.
2) We have further developed two extended variant algorithms, OBMA_B and OBMA_S, for applications adapted to ultra-high speed lookup and ultra-low cache memory requirements, respectively.
3) We conduct extensive experiments to evaluate the performance of the OBMA family. Experimental results on the real-world data sets [7, 8, 9] show that OBMA_B and OBMA_S can reduce the memory footprint to 4.85 and 3.98 bytes/prefix, respectively. OBMA_B supports updates at a rate of 14.58M updates/s. OBMA_B and OBMA_S can support 219.56 Mpps and 252.02 Mpps lookup throughput with a single thread. Their lookup throughput exceeds 600 Mpps with multiple parallel threads in a single CPU. Our tests also show that the update process does not interfere with the lookup process for the OBMA family of algorithms, and zero-interrupt to lookups with multiple threads can be achieved.

The rest of the paper is organized as follows. Section II surveys the related work. Section III details the data structure and lookup process of the basic OBMA algorithm(OBMA_B). Section IV describes OBMA_L, the variation of OBMA_B optimized for lookup throughput. Section V describes OBMA_S, the variation of OBMA_B optimized for storage. Section VI examines the bursty update phenomenon through real-world update traces and proposes optimizations to reduce the update execution time and realize zero-interrupt for updates. Section VII evaluates the performance of the OBMA family by comparing with the state-of-the-art algorithms. Finally, Section VIII concludes the paper.

II. Related Work

The problem of route lookup is well researched with a large body of literature. Due to the space limit, we only discuss the software-based schemes which are more closely related to our work.

Basically, the software-based schemes can be divided into two categories: Bloom Filter-based and trie-based [10]. Since Bloom Filter cannot handle element withdrawal, Counting Bloom Filter is introduced [11][12]. However, Counting Bloom Filter suffers from false negative in addition to false positive.

Trie-based algorithms are immune to these issues and are more widely used in practice. While such algorithms can be implemented in both CPU and GPU, CPU-based implementations are more appreciated by the community, because apart from generality, GPU is inferior to CPU in terms of power consumption and lookup latency due to packet batch [13][14], which offsets GPU's lookup performance gain from massively parallel processing. Incremental updates in GPU is also more difficult.

. Lulea [5] is considered to be the most representative bitmap compression algorithm to date, which effectively reduces the memory footprint. However, by carefully exploring its data structure, we find that Lulea's leaf-pushing technique changes the original trie structure and complicates the update operation. Moreover, leaf-pushing brings more redundant "1" bits in the data structure. Differently, OBMA avoids leaf-pushing and keeps the original trie structure to support fast incremental updates.

The state-of-art software-based algorithms are all trie-based [2][4][6][15]. BS [6] achieves fast lookups and updates, but it takes a large storage space. LOOP [15] applies the overlay bitmap to merge multiple virtual route tables. However, it partitions the bitmap into equal-sized groups, which does not adapt to the bursty updates. The fast lookup performance of SAIL [4] relies on the traffic locality. and requires a storage larger than the basic binary trie. Poptrie [2] develops a multiway method to accelerate lookups and adopts a pointer replacement approach to support fast updates. However, Poptrie has a long update delay which can cause packet drops while dealing with bursty updates. Poptrie also needs to reserve a large amount of memory to hold the updated structure. The OBMA family achieves better memory efficiency, and higher lookup and update speeds than these algorithms.

III. Basic Overlay Bitmap Data Structure

In this section, we first describe the basic overlay bitmap. Then we elaborate the software implementations of OBMA_B and OBMA_S through a [18-6-8] three-layer trie. For the convenience of discussion, we reuse some terms in Lulea [5].

A. Reduce the Number of 1s in Overlay Bitmap

Similar to Lulea, OBMA takes three steps to construct the lookup data structure: 1) Build 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 differs from Lulea mainly in step 2, in which Lulea uses leaf-pushing to build bitmaps, while OBMA uses level-traversal to build Forwarding Port Arrays (FPA) and overlay bitmaps. We use the routing table example in Figure 1 to illustrate the difference of these two approaches. As shown in the figure, the corresponding prefix trie is cut at level 3. The nodes from level 0 to 3 are grouped into layer 1. The non-leaf nodes on the cut level (e.g., the blue node N1) are named pointer nodes which store pointers to the chunks in the next layer.

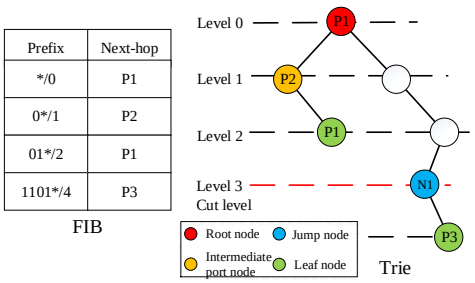


Fig. 1. Route table and prefix trie

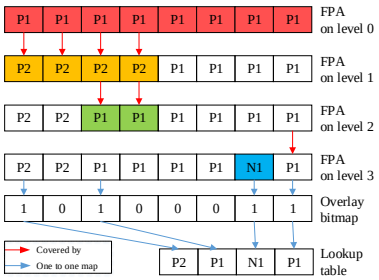


Fig. 2. Overlay bitmap generation process

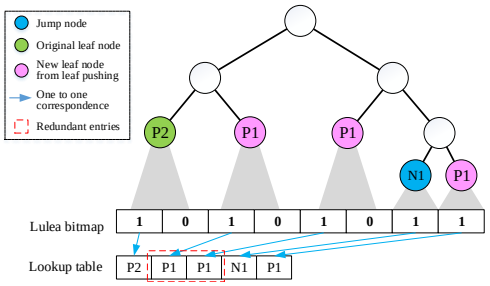


Fig. 3. Lulea bitmap generation process

The process of level traversal is shown in Figure 2. Each prefix covers a range of elements in an FPA so every element in the range inherits the same output port information of the prefix. For example, in Figure 2, node P2 on level 1 covers range [0, 3], and its child node P1 covers range [2, 3]. To support LPM, the nodes within the range of P1 (i.e., the range [2, 3] of the FPA at 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 the correctness of LPM.

Although an FPA can be used directly for route lookups, its storage is still large. One thing we can optimize is the horizontal redundancy (i.e., consecutive elements store the same output port information). To remove such redundancy, we convert each FPA into an overlay bitmap plus a corresponding lookup table. Each bit in the overlay bitmap corresponds to an entry in the FPA. We name the maximum consecutive range containing identical elements a segment. The bit corresponding to the first element of each segment is set to "1" and the others "0". The lookup table orderly stores the first elements of each segment. 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 proper bitmap, then counts the number of 1s up to a proper bit position, and finally gets the next port information in a proper lookup table using the number as index.

In contrast, Lulea generates bitmaps in three steps as shown in Figure 3: 1) Modify the prefix trie to a complete tree via leaf pushing so that only leaf nodes contain port information or pointers; 2) Traverse all leaf nodes from left to right and project the nodes to the cut level; 3) Set the bits corresponding to the beginning nodes (named as genuine heads in [5]) of each projection range to "1" and others to "0", and orderly store the port or pointer information in a lookup table.

Clearly, the horizontal redundancy exists in Lulea bitmaps as shown in Figure 3. In comparison, overlay bitmap can eliminate such redundancy so that the number of entries in the corresponding lookup table is reduced. The overlay bitmap actually minimizes the number of 1s for a fixed prefix trie, which can be proved by reduction to absurdity.

Theorem 1. *Given a same prefix trie, the overlay bitmaps contain the minimum possible number of 1s among all the trie compression algorithms.*

Proof. Let B denote a bitmap array (specifically, B_o denote an overlay bitmap and B_n denote a bitmap from other algorithms), T denote the corresponding lookup table, and L denote the length of B. The number of 1s in B up to index i is $S(B,i) = \sum_{k=0}^{i} B[k]$.

Suppose $\exists B_n \ s.t. \ S(B_n, L) < S(B_o, L)$.

So, $\exists i, \ s.t.$

$$B_o[i] = 1, \ B_n[i] = 0 \tag{1}$$

Based on Equation (1), if the IP address is i, we have

$$T_o\left[S\left(B_o, i\right)\right] = T_n\left[S\left(B_n, \ i\right)\right] = T_n\left[S\left(B_n, i-1\right)\right] \tag{2}$$

For the IP address $i-1$, we have

$$T_o\left[S\left(B_o, i-1\right)\right] = T_n\left[S\left(B_n, i-1\right)\right] \tag{3}$$

Comparing Equation (2) and (3), we have

$$T_o\left[S\left(B_o, i\right)\right] = T_o\left[S\left(B_o, i-1\right)\right]$$

This contradicts the fact that there is no identical adjacent entry in an overlay bitmap. Therefore, the assumption is wrong and the theorem is proved. □

B. Implementation of OBMA_B

Compared with the Lulea bitmap, the 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 information into one element, leading to fewer 1s in overlay bitmaps and smaller-sized lookup tables; 2) Lulea modifies the prefix trie via leaf pushing to support LPM, but leaf pushing changes the original structure of the prefix trie, which complicates trie updates. In contrast, OBMA family keeps the original prefix trie structure unchanged, which is convenient for updates.

A specific implementation of OBMA_B uses a [18-6-8] trie partition to split the 32-bit IP address space into three layers, as shown in Figure 4. To improve the lookup speed, we adopt the same direct pointing technology used in [2]. OBMA_B maintains one FPA on level 18 which is named Direct Pointing Array (DPA). The size of the DPA is 512KB (i.e., $2^{18} \times 2$B). The lookup process uses the first 18 bits, the middle 6 bits, and the last 8 bits of an IP address to access the data structures at layer 1, layer 2, and layer 3, respectively.

For the lookup in layer 1, we directly use the highest 18 bits of the IP address as index to locate an entry in the DPA. A lookup entry has two bytes, containing a *type* field (1 bit) and a *port* field (15 bits). When *type* is 0, the *port* field stores the port information, and when *type* is 1, the *port* field stores the index of a Layer 2 chunk list. The Layer 2 chunk list stores

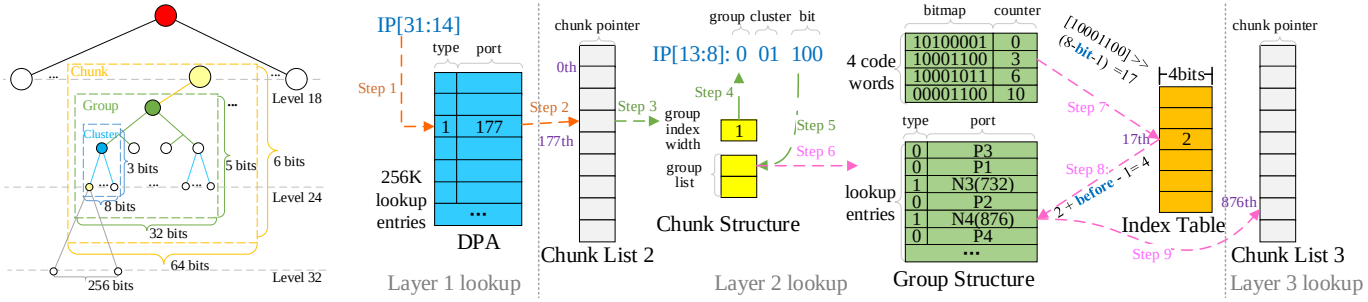


Fig. 4. [18-6-8] IP address partition. Fig. 5. OBMA_B lookup structure and process.

pointers to locate a specific chunk structure for lookups in layer 2. For example, in Figure 5, accessing the DPA with the highest 18 bits of the IP address leads to the 177th element of the Layer 2 chunk list according to the port information, and finally gets the chunk pointer.

To improve the update performance, OBMA_B applies the adaptive grouping technique (to be explained in Sec VI) which treats frequently updated area with smaller granularity. Some terms used in the layer 2 and 3 lookups are defined as follows:

Chunk: A subtree in layer 2 (layer 3), which is 6 (8) levels deep.

Group: A d-level subtree in a chunk, where d is in the range of 3-6 (3-8) in layer 2 (layer 3). A group is the basic unit for lookups and updates. A chunk is composed of 2^{6-d} (2^{8-d}) groups in layer 2 (layer 3). As shown in Figure 4, d is five in the chunk, so it has two five-level groups.

Cluster: A 3-level subtree in a group. A d-level group has $2^{(d-3)}$ clusters. A cluster corresponds to an 8-bit bitmap and serves for the fast determination of the number of 1s.

We illustrate the lookup structure of chunk, group, and cluster in Figure 5. A chunk structure contains two parts: 1) *group index width*, indicating the number of bits in an IP address segment used for layer 2 and 3 are group index; 2) *group list*, containing the pointers to each group structure. For example, the IP address segment is $(001100)_2$ and the *group index width* is 1 in Figure 5, so the index for the group structure is 0 and the first pointer in the *group list* is used to obtain the group structure.

Since the height of the cluster is three, the lowest three bits of the address segment are used as the bit index in a cluster and the other bits excluding the group index are used as the cluster index. For example, in Figure 5, the *group index width* is 1, so the cluster index is $(01)_2 = 1$ and the bit index is $(100)_2 = 4$.

A group structure consists of code words indexed by the cluster index and lookup entries. An entry stores a pointer to a Layer 3 chunk list if its *type* field is 1. The sizes of a code word and a lookup entry are both 2 bytes. In a code word, one byte stores the cluster *bitmap* and the other byte (i.e., the *counter* field) records the cumulative number of 1s up to this cluster in this group. As shown in Figure 5, the *counter* field of the third code word is 6, which means the total number of 1s in the two previous *code word* bitmaps is 6. An auxiliary *Index Table*, using the 8-bit bitmap as index, stores the number of 1s in each bitmap. A part of cluster bitmap up to the given bit position is used as the index to access the *Index Table*. To get the number of 1s ahead of a given bit location in a cluster, we simply add the value in the *counter* field and the value got from the *Index Table*.

For example, the bitmap $(00010001)_2$ has two 1s in it and its decimal value is 17. So the 17th entry of the Index Table stores the value 2. In our example, the cluster index is 1 so the second code word is chosen. The bit index is 4 so the cluster bitmap $(10001100)_2$ in the code word is shifted right by $8-4-1=3$ bits. The resulting value $(0001001)_2 = 17$ is used as index to access the Index Table, which returns the value 2. Adding this value to 3, the value in the *counter* field of the code word, we get the number 5, which is used as the index to the lookup entries.

Algorithm 1 describes the OBMA_B lookup process. Given an IP address, OBMA_B uses its first 18 bits to visit the DPA to get a chunk list index in step 1. Next, OBMA_B locates the chunk in layer 2 in step 2 and 3. Then, OBMA_B splits the middle segment of the IP address, visits the corresponding group structure, and gets the lookup entry in step 4-8. If the entry stores a pointer, OBMA_B keeps searching and locates the chunk in layer 3 in step 9. The search in layer 3 is similar to that in layer 2.

Algorithm 1 Lookup Algorithm of OBMA_B

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

1: $entry \leftarrow DPA[IP[31:14]]$
2: **if** $entry.type == 0$ **then**
3: **return** $entry.port$
4: $width \leftarrow ChunkList2[entry.port].groupWidth$
5: Get $groupIdx, clusterIdx, bitIdx$ from $IP[13:8]$
6: $group \leftarrow ChunkList2[entry.port].groupList[groupIdx]$
7: $codeword \leftarrow group[clusterIdx]$
8: $lookup \leftarrow group + 1 << (3 - width)$
9: $ix \leftarrow codeWord.bits >> (7 - bitIdx)$
10: $pix \leftarrow codeWord.counter + IndexTable[ix] - 1$
11: $entry \leftarrow lookup[pix]$
12: **if** $entry.type == 0$ **then**
13: **return** $entry.port$
14: **else**
15: /*search deep to layer 3 chunk like layer 2 chunk*/

IV. Lookup Optimization

Since the update performance of OBMA_B is extremely good, we try to trade off some of the update performance for

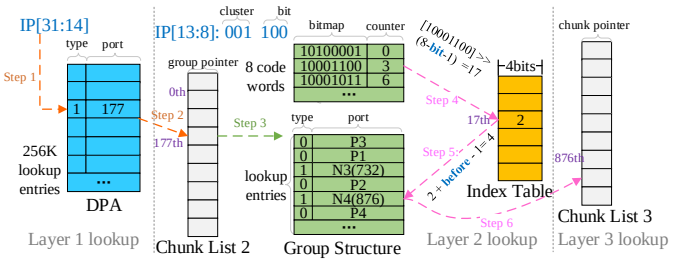


Fig. 6. OBMA_L lookup structrue and process.

better lookup performance. OBMA_L is such a variation.

Generally, a memory access incurs long latency. OBMA_L focuses on simplifying the lookup process by reducing the number of memory accesses. It does not apply adaptive grouping. Instead, it applies a fixed [0-3-3] partition in layer 2 and [0-5-3] in layer 3. As a result, the chunk structure can be omitted and the chunk lists actually store group pointers as shown in Figure 6.

For consistency, we still use the term of chunk list for OBMA_L. OBMA_L gets the group pointer without needing to visit the chunk structure, making the lookup faster. However, the big group makes the update slower because rebuilding the entire bitmap and group structure needs more time now.

The lookup process of OBMA_L can be deduced from that of OBMA_B easily, so we skip it here.

V. Storage Optimization

It is possible to generate a sparer bitmap with fewer 1s for some trie structure using a Bit Inversion Sequence (BIS). To get the BIS, we propose an optimal algorithm with $O(n^2)$ time complexity and an approximate one with $O(n)$ time complexity, where n denotes the bitmap length. We prove the lower bound of the compression ratio based on the Redfield–Pólya theorem [16, 17] and compare it with the achieved compression ratio under the uniform distribution. By embedding BIS to the OBMA_B data structure, the resulting OBMA_S achieves better storage efficiency than OBMA_B.

A. Bit Inversion Sequence

The number of 1s in an overlay bitmap directly determines the number of lookup table entries. The overlay bitmap achieves the minimum number of 1s for a prefix trie. However, we can further reduce the number of 1s in the bitmap by taking advantage of graph isomorphism.

In graph theory, two graphs G and H are isomorphic if there exists a bijective function, $f\colon V(G) \to V(H)$, which satisfies that any two vertices u and v of G are adjacent in G iff $f(u)$ and $f(v)$ are adjacent in H. Figure 7 shows two isomorphic graphs of the original trie in Figure 1. We say an isomorphic graph is optimal if its corresponding overlay bitmap contains the minimum number of 1s. In this sense, the two graphs in Figure 7 are both optimal.

An isomorphic graph is the result of permuting the FPA with a bijective function. The problem of finding an optimal isomorphic graph of the original trie is therefore to find such an optimal permutation. Although $n!$ permutations exist for an FPA of length n, we can use a general sorting algorithm with

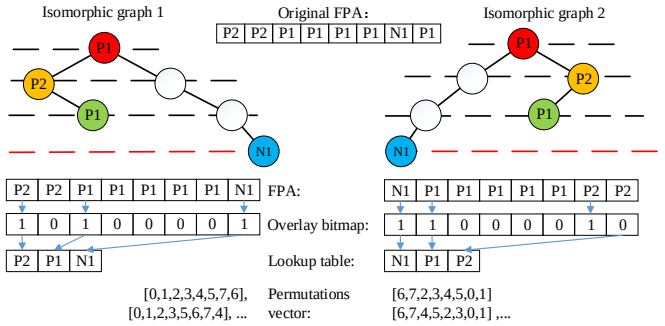


Fig. 7. Isomorphic graphs example and the corresponding permutation vector v. $v[i] = j$ means the i-th element of the original FPA should be moved to the j-th position.

the time complexity of $O(n \lg n)$ to find the optimal permutations. However, an optimal permutation may not lead to an ideal solution for the following reasons: 1) The permutation, which is encoded as a permutation vector as shown in Figure 7, needs to be stored along with the bitmap for lookup and update. Unfortunately, the permutation vector may cancel out the storage space saving acquired from fewer 1s in the overlay bitmap. For example, the isomorphic graph 1 in Figure 7 represents a permutation from the original FPA, which can be encoded as a permutation vector [0,1,2,3,4,5,7,6]. The resulting overlay bitmap contains one less "1", which saves one entry (i.e., two bytes) from the lookup table. However, storing the permutation vector requires eight bytes. 2) Even if the overall storage is reduced, the lookup or update process needs to take up to n table lookups on the permutation vector to recover the original IP address, which influences the performance, especially when n is large.

We use BIS to optimize the encoding for permutation and limit its influence on lookup and update performance.

If the length of the original FPA is $n = 2^N$, a BIS is a N-bit sequence which encodes a specific permutation. After a permutation, a bit's new position index in a bitmap is the result of the bit-wise XOR operation on the bit's old position index and the BIS. For example, if the BIS is (001), the original first and second bits will be swapped, because their indexes, $(000)_2$ and $(001)_2$, after XORed with (001), are swapped. Applying this to every bit position, we get a permutation equivalent to the permutation vector [1,0,3,2,5,4,7,6].

The N-bit BIS can encode 2^N permutations. To recover the original trie structure, only bitwise XOR operations are needed, which is much faster than decoding the permutation vector. BIS cannot encode all the $n!$ possible permutations, so it cannot guarantee to capture the optimal overlay bitmap with the minimum number of 1s. However, as long as a BIS can generate a better bitmap with fewer 1s, a better storage efficiency can be achieved. The BIS that can lead to the minimum number of 1s in the resulting bitmap is optimal.

B. Algorithm for the Optimal BIS

Testing all the 2^N permutations allowed by BIS to figure out the optimal one is time consuming with a complexity of $\Theta(2^N \cdot (n-1)) = \Theta(n^2)$. OBMA allows n to be as large as 256. The exhaustive search for the optimal BIS can significantly drag down the update speed.A more efficient algorithm is needed.

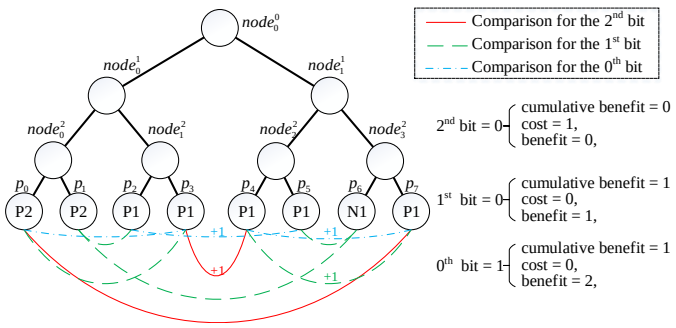


Fig. 8. A BIS example using a greedy top-down algorithm

Instead of finding the globally optimal BIS, we use a one-pass greedy algorithm to obtain a locally optimal BIS with the time complexity of $\Theta(n)$. We use a full binary trie shown in Figure 8 to explain the algorithm, in which $node_t^d$ denotes a non-leaf node with the index of t at the depth of d, and p_t denotes a leaf node with the index of t. Clearly, the FPA is the array of elements contained in the nodes from p_0 to p_{n-1}.

Each bit in a BIS corresponds to a depth in the trie. The bit value 1 means the two subtries of each node at this depth are swapped. We search a solution by determining the value of BIS bits in order and in one pass which is essentially to decide whether or not to switch the subtries at each depth from top to bottom.

As described in Algorithm 2, each step determines the value of one BIS bit based on two criteria: 1) the benefit of setting the bit to 1, which is evaluated by the number of newly emerging identical adjacent pairs; 2) the cost of setting the bit to 1, which is evaluated by the net loss of identical adjacent pairs over the cumulative benefits from previous steps. Only when the benefit is higher than the cost, the current bit of the BIS is set to be 1 and the accumulated benefit is updated.

In the example shown in Figure 8, the value of the first BIS bit is determined first. Since swapping the two subtries of $node_0^0$ gives no benefit but has a cost of 1 (i.e., the new adjacent pair of nodes, p_0 and p_7, have different port information; meanwhile, p_3 and p_4, who have the same port information, are set apart), the swap should not happen and the first BIS bit should remain to be 0. We work through the BIS bit by bit and eventually obtain the final BIS "001", which reduces one "1" in the resulting overlay bitmap.

From the pseudo code of Algorithm 2, we can derive the algorithm's time complexity.

$$T = 2\sum_{i=0}^{N-1}\sum_{j=0}^{2^i-1}\Theta(1) + \sum_{i=1}^{N-1}\sum_{s=0}^{i-1}\sum_{j=0}^{2^s-1}\Theta(1) = \Theta(2^N) = \Theta(n)$$

Although this greedy algorithm further reduces the permutation search space and may be trapped in a local optimal solution, it runs much faster than the exhaustive search and the result is not too far from the global optimal based on our evaluation.

C. Compression Ratio Analysis

After applying a BIS on an FPA, we obtain a new overlay bitmap, which is named bit-inversion bitmap. The compression ratio is defined as the ratio of the number of 1s in the bit-inversion bitmap to that in the original overlay bitmap.

Algorithm 2 Greedy Top-to-down Algorithm

Input: N, PA.
Output: inv.

$inv[N-1:0] \leftarrow 0, cumulative\_benefit \leftarrow 0$
$ml \leftarrow 2^{(N-1)}-1, mr \leftarrow 2^{(N-1)}, el \leftarrow 0, er \leftarrow 2^N-1$
if $PA[ml] == PA[mr]$ **then**
  $sum\_cost \leftarrow 1$
else if $PA[el] == PA[er]$ **then**
  $inv[n-1] \leftarrow 1$
  $sum\_cost \leftarrow 1$
for $i = N-2$ to 0 **do**
  $benefit \leftarrow 0, cost \leftarrow 0$
  $range = 1 << (i+1)$
  for $j = 0$ to $2^{(N-i-1)}-1$ **do**
    $base \leftarrow j << (i+1)$
    $ml \leftarrow base + range/2 - 1, \quad mr \leftarrow ml + 1$
    $el \leftarrow base, \quad er \leftarrow base + range - 1$
    if $PA[ml] == PA[mr]$ **then**
      $cost$ += 1
    if $PA[el] == PA[er]$ **then**
      $benefit$ += 1
  for $s = 0$ to $N-i-2$ **do**
    $range =\leftarrow 1 << (n-s-1)$
    for $j = 0$ to 2^s - 1 **do**
      $base1 \leftarrow j << (n-s) + \overline{inv[i+1]} << (n-s-1)$
      $base2 \leftarrow j << (n-s) + \overline{inv[i+1]} << (n-s-1)$
      $ml \leftarrow base1 + range/2 - 1$
      $mr \leftarrow base2 + range/2$
      if $PA[ml] == PA[mr]$ **then**
        $benefit$ += 1
  if $benefit > cost + cumulative\_benefit$ **then**
    $inv[i] \leftarrow 1$
    $cumulative\_benefit$ += $benefit$
  else
    $cumulative\_benefit$ += $cost$
return inv

Theorem 2. *Assume that a port array has $n = 2^N$ elements and each element can take an arbitrary port out of M different ports. The total number of element-port combinations is M^n, and the lower bound of the compression ratio is $\frac{1}{n}$.*

Proof. In group theory, each permutation can be represented as a cyclic notation. The permutation vector [0,1,2,3,4,5,6,7] can be written as (0)(1)(2)(3)(4)(5)(6)(7), and [1,0,3,2,5,4,6,7] can be written as (0,1)(2,3)(4,5)(6,7). The number of "()" pairs is the cycle number, so the cycle number is 8 for the first case and 4 for the second. The arrangements are in an orbit if they can be transformed to each other by the permutation group.

A BIS represents a specific permutation. All the permutations that can be represented by BIS make up a permutation group. In this permutation group, one permutation keeps the original order, whose cycle number is n. The rest 2^N-1 permutations have the same characteristic, i.e., one-to-one swaps between element, whose cycle number is $n/2$.

According to Redfield–Pólya theorem [16], the number of orbits is as below,

$$l = \frac{1}{|G|}\sum_{g\in G} M^{c(g)} = \frac{M^n + (n-1)M^{\frac{n}{2}}}{n} \tag{4}$$

TABLE I
REAL COMPRESSION RATIO UNDER $n = 8, 16$

	Theoretical lower bound	M=2	M=3	M=4	M=5	M=6
n=8	12.5%	50.0%	48.97%	51.66%	54.79%	57.82%
n=16	6.25%	29.37%	27.76%	29.08%	30.80%	32.56%

where l denotes the number of orbits, G denotes the permutation group, $|G|$ denotes the element number in G, and $c(g)$ denotes the number of cycles of the element g.

Because the permutation group divides the M^n arrangements to l orbits, and at least one arrangement in an orbit has the bitmap with the fewest 1s, the lower bound of the compress ratio is

$$\frac{l}{|G|} = \frac{1}{n} + \frac{n-1}{n} M^{-\frac{n}{2}} > \frac{1}{n} \tag{5}$$

□

Theorem 2 shows that longer bitmap has a potential to achieve a higher compression ratio. It also provides an ideal compression limit, which can be approached if there is only one combination whose bitmap has the fewest 1s for all orbits.

We set n to 8 and 16, and M from 2 to 6 to show some real compression ratios in Table I. We have some observations on the results: 1) the real compression ratios shows not so close to the lower bound, due to more than one combination in an orbit with the minimum number of 1s; 2) The optimal compression ratios are achieved for both n when $M = 3$; 3) the real compression ratios grow slowly after $M = 3$. Most importantly, although there is a big gap between the real compression ratios and the theoretical limit, the compression effect is still significant and a larger n can make the result better.

The above analysis is based on the assumption that the port distribution is uniform which may not be true for real IP tables. We do experiments to get the compression ratio on real IP tables in Section VII, where it actually indicates a very encouraging result..

D. Structure of OBMA_S

OBMA_S's architecture is similar to OBMA_B. In addition to the application of BIS, OBMA_S adopts two new approaches to further optimize the storage.

First, when the number of prefixes in a chunk is less than a predefined value K (e.g., $K = 4$), we use a *sparse chunk* structure to replace the original bitmap chunk. A sparse chunk is identified by a new 1-bit *type* field in the chunk structure. A sparse chunk is composed of k ($k \leq K$) *sparse entries*, which are arranged in descending order of prefix length. Each sparse entry consists of three fields: *prefix*, *mask*, and *lookup entry*. In a sparse chunk, the IP address segment does a linear search to find the best match.

Second, we orchestrate the BIS into the OBMA_S structure by taking advantage of the memory word alignment without introducing extra storage overhead. Each group is assigned a BIS. The BISes for all the groups in a chunk are packed together into seven bytes in the chunk structure, following 1-bit type and 7-bit group width fields. This assignment constrains the *group width* to be smaller than 4, because otherwise the size required by all the BISes will exceed seven bytes.

The lookup process of OBMA_S is similar to what is described in Algorithm 1, except that the cluster index and bit index are obtained from the bitwise XOR operation over overlay bitmap and BIS. In the example in Fig 5, if the BIS stored in the chunk structure is (01010), the new cluster index is $(01)_2 \oplus (01)_2 = 0$ and bit index is $(100)_2 \oplus (010)_2 = 6$.

VI. FAST AND ZERO-INTERRUPT UPDATE

OBMA and the other trie-based algorithms need two steps for an incremental update: 1) Modify the trie according to the updated prefix(es) and generate a new partial or entire lookup structure; 2) Use the new structures to modify or replace the running one.The first step can run in parallel with lookups using the multi-threading technique. We define the time consumed in this step as *execution time*. The second step needs to lock the data structure temporarily. We define the time consumed in this step as *interrupt time*.

In this section, we first examine the characteristics of updates via real update traces. Second, we present a series of update optimizations for the OBMA family algorithms which reduce the execution time and achieve zero interrupt time.

A. A Closer Look at Bursty Updates

We download the RIB update traces for three route tables: Oregon, Equinix, and ISC from April 1 to April 30, 2017. The trace sizes are 45 GB, 6 GB, and 55 GB data, respectively [7, 9]. We are interested in the peak update rate in each days and the update locality.

Figure 9 shows the update statistics for the three tables. The updates are bursty and the bursts are recurring. At peak time the update rate is up to 181.0 K/s, 66.5 K/s and 59.8 K/s for Equinix, ISC and Oregon, respectively.

We examine the update locality on prefixes and on chunks. Figure 10 shows the length distribution of updated prefixes. The updates on prefix length 24 account 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. Figure 11 shows the CDF of updates over chunks for the three tables on April 1st, 2017 (the results for the other days are similar). The updates on chunks show strong locality: 20% of chunks receive over 80% of updates.

B. Reducing Execution Time

When an update arrives, OBMA updates the prefix trie first. OBMA does not perform leaf-pushing, so the time spent on modifying the original prefix trie is negligible. OBMA then conducts level traversal and generates a new bitmap. The complexity of level traversal is $O(2^D)$, where D is the height of subtrie. However, the updated prefix only affects its coverage nodes and it is unnecessary to start the traversal from the root node in most instances. We also segment the bitmap into groups to narrow the scope of level traversal in Chunks. In order to exploit the update locality, we apply an *adaptive grouping* approach on OBMA except for OBMA_L.

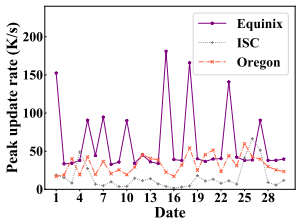


Fig. 9. Peak update statistics for the tables

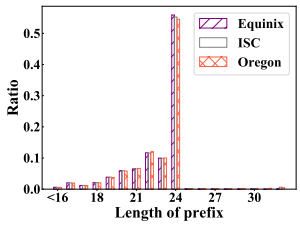


Fig. 10. Length distribution of updated prefixes

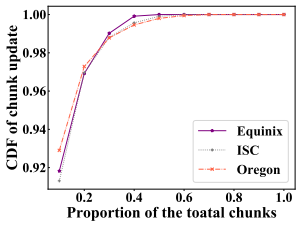


Fig. 11. CDF of chunk update

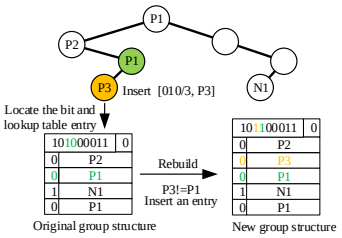


Fig. 12. Example of 24th-level optimization

Because in many cases, the update is just a non-pointer node insertion or modification at the 24th level, we can use a *24th-level optimization* to skip the level traversal and rebuild the group structure based on the existing one directly. The details of the adaptive grouping and the 24-level optimization are as follows.

1) Adaptive grouping: The update locality inspires us to develop an adaptive grouping approach. Based on the update frequency, the FPA of each chunk can be partitioned into a different number of equal-sized groups. We rebuild each group as a separate lookup structure and the entire lookup table is organized as separable and independent group structures. Frequent updates end up in fine-grained groups, only requiring the reconstruction and replacement of the affected group structures. Group partitioning is flexible. A smaller group, at the cost of more storage for group pointers, needs fewer bytes to access for reconstruction so the updates on it are faster. As the result of adaptive grouping, the frequently updated chunks contains more small-sized groups and the infrequently updated chunks can contain only one group.

Let the chunks with the *group width* of i bear an update frequency in the range of $[f_{i-1}, f_i)$, where $f_i = \beta f_{i-1}$. Especially, in layer 2, the first frequency range is $[0, f_0)$ and the last is $[f_2, \infty)$. Therefore, given an initial frequency threshold f_0 and a factor β, we can get the mapping between update frequency and the *group width*. Our adaptive grouping approach is composed of two processes: online addition and periodic decay.

Online addition: we maintain a counter for each chunk and increase it by one whenever the chunk receives an update. If the counter reaches the current range's upper bound, the *group width* is incremented by one and the chunk is rebuilt.

Periodic decay: We check the counters of all chunks periodically (e.g. each hour or each day). If a counter value is smaller than its current lower bound , we rebuild the chunk according to the update frequency and the *group width*.

2) 24th-level Optimization: Level traversal can be expensive in terms of CPU cycles. If we can avoid the level traversal, the update performance will be improved. The 24th-level optimization is based on the fact that for an updated prefix with the length of 24, if its corresponding node in the prefix tree is neither existent nor 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. The statistic results show that 99.9% of updates with the prefix length of 24 meet the condition and nearly 50% of the updates are for prefixes with the length of 24, so the optimization is effective.

TABLE II
BITMAP TRANSFORMS ON MIDDLE NODES

Original bitmap	Updated bitmap	Example
(00)	(11)	insert B, $\underline{A}A \leftarrow \underline{B}A$
(01)	(10)	insert B, $\underline{A}B \leftarrow \underline{B}B$
	(11)	insert C, $\underline{A}B \leftarrow \underline{C}B$
(10)	(01)	insert A, $A\underline{B}B \leftarrow A\underline{A}B$
	(11)	insert C, $A\underline{B}B \leftarrow A\underline{C}B$
(11)	(00)	insert B, $B\underline{A}B \leftarrow B\underline{B}B$
	(01)	insert C, $C\underline{A}B \leftarrow C\underline{C}B$
	(10)	insert B, $C\underline{A}B \leftarrow C\underline{B}B$
	(11)	insert D, $C\underline{A}B \leftarrow C\underline{D}B$

We use the trie structure in Figure 12 as an example to illustrate the optimization at level 3. 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 ports P1. The bitmap "10" means that the node to be modified and its right neighbor both store P1. We can also get P1's left neighbor's port P2 by visiting P1's previous entry. As P3 does not equal to either P2 or P1, this update sets the second bit of the bitmap to 1 and adds a new entry of P3 in front of P1. The resulting group structure is the same as that derived from the level traversal, but it is acquired faster.

The actual situation is more complex than the example. We need to consider both the local bitmaps and the cluster's position. For example, if the node to be modified is at the head or tail of a group, we only need to consider one neighbor. If the node is at the tail of a cluster but not at the tail of a group, we also need to consider the *counter* field of the *code word*. All the possible bitmap transformations on a middle nodes are listed in Table II for the example. Note that we do not always need to access the lookup table three times to get the node and its two neighbors. If one of the two-bit is 0, the lookup entry corresponding to it is the same as its left neighbor. One lookup table access is saved in this case.

C. Zero-interrupt Update

We use multi-threading to achieve zero-interrupt update. The update process has its dedicated thread. OBMA needs to generate and maintain a prefix trie for building its lookup structure. Since the lookup process only accesses the lookup structure and the first step of the udapte process only works on the prefix trie, they can work in parallel. The second step of the update process needs to modify the DPA elements or replace the OBMA structure with the new pointers of chunks or groups, incurring the thread synchronization problem.

To address this problem, we do not apply any thread lock due to its low efficiency. Instead, we use atomic operation to

ensure the safety of OBMA structure reading and writing.A key problem is to ensure the safety of the situation that the lookup thread is reading the stale memory while the update thread is releasing it. To solve this problem, we use a pointer buffer with enough size to cache the replaced pointers. The memory will not be released until its pointer is popped from the buffer. Figure 10 shows that few updated prefixes are shorter than 18 and the chunk/group number is very large, so the situation of reading a chunk/group and replacing its pointer rarely happens at the same time. Even if it happens, it only produces a wrong lookup results but cannot crash the system due to illegal memory access. The transient forwarding errors are tolerable for routers. The benefit of zero-interrupt update is substantial.

The updates to the lookup structures are related to the length of the updated prefix. If the length is shorter than 19, only the DPA needs to be updated. If the length is greater than 18, we need to generate several new group pointer(s) and even new chunk pointer(s). Our experiments show that on average an update does not modify more than one group or chunk pointer, requiring only tens of bytes of extra storage space. In contrast, Poptrie also claims to support zero-interrupt updates, but it needs to rebuild a new lookup structure entirely and replace the root pointer to switch from the old one by an atomic operation, which is more complex and results in a larger memory footprint in practice. Moreover, rebuilding a new lookup structure can take tens of milliseconds when the update prefix is shorter than 19. As a negative result,the long delay of the zero-interrupt update in Poptrie could cause more packets are forwarded incorrectly.

VII. Evaluation

In this section, we evaluate the performance of OBMA_B, OBMA_S, and OBMA_L on storage, lookup, and update. As OBMA_S and OBMA_L have different optimization objects, we evaluate their performance on storage and lookup, respectively. In order to eliminate random deviation, each reported lookup or update speed value is the result of the average from 20 repeated experiments.

A. Experiment Setup

Platform: We conduct experiments on a Dell M4800 mobile workstation with an Intel CPU Core i7-4900MQ. The CPU contains four 2.8GHz cores with each supporting two threads. Each CPU core has an independent L1 cache (128KB D-Cache and 128KB I-Cache) and an L2 cache (1MB). The four cores share an L3 Cache (8 MB). The workstation is equipped with 8GB DDR3 (1.6GHz) memory and runs 64-bit Ubuntu-14.04-LTS OS.

Datasets: The route tables used for evaluation are downloaded from three routers (i.e., Oregon, Equinix, and ISC) on April 1st, 2019 [7, 9]. We use CAIDA traces [8] to test the lookup speed. We collect update packets from RIPE Network Coordination Center [7] to test the update speed.

Algorithms: We compare the OBMA family with two state-of-art works, SAIL [4] and Poptrie [2], and two classic works, Lulea [5] and binary trie [3]. OBMA, Poptrie, and Lulea all use the partition mode [18-6-8]. OBMA_B adopts the update optimizations discussed earlier. Specifically, we set the initial threshold f_0, the factor β, and the cycle of the adaptive grouping to 32, 4, and one day, respectively.

B. Memory Efficiency

1) The Number of 1s: We first examine the benefit gained from the overlay bitmap and bit-inversion bitmap on real route table snapshots in Figure 13. For fairness, all the bitmaps are fixed to be 16 bits long. The three tables contain 769K, 780K, and 804K prefixes, respectively. The corresponding next-hop port counts are 18, 6, and 43. The overlay bitmaps reduce about 2/3 of 1s from the basic Lulea bitmaps by eliminating the horizontal redundancy.

The BIS compression ratios are 89.2%, 88.7%, and 77.7%, respectively, which are decent but not as good as the results in Table I. An overlap bitmap is more compressible if the number of 1s in it far exceeds the number of unique ports in the corresponding FPA. Such condition is less likely in real route tables than in synthesized route tables. For example, in the Equinix table, the bitmaps of 66% of groups are incompressible and 54.1% of groups contain just one port. As a result, only 23.4% of groups are actually compressed. However, the gain is still substantial enough to justify the optimization effort.

2) Storage Comparison: We use the three route tables to compare the storage efficiency in the unit of bytes per prefix for OBMA_B, OBMA_S Poptrie, SAIL, Lulea, and Binary Trie. Figure 14 shows the results for the five algorithms. Lulea, Poptrie, OBMA_B, and OBMA_S require relatively small storage, which are 5.87, 5.33, 4.85, and 3.98 bytes per prefix on Equinix, respectively. OBMA_S achieves the best storage efficiency with BIS and the sparse chunk optimization, saving 25.33% storage of Poptrie. OBMA_B is the second best except on the Oregon table.Although Poptrie has a good storage efficiency for the lookup structure, it needs to pre-allocate more than ten megabytes of storage for updates, which we do not take into account. Lulea also has a good storage efficiency but its update performance is poor. While fast in lookups, Sail presents a poor storage efficiency, which is only better than the basic Trie but multiple times worse than the other algorithms.

It is possible to further compress the lookup structure size of OBMA_B and OBMA_S by reducing the DPA size and raising the adaptive grouping threshold, but these will affect the update performance negatively. We give the lookup and update performance a higher priority than the storage efficiency so no further memory reduction is pursued.

To test the memory scalability, we randomly select some prefixes in the Equinix table to form tables with different sizes, and run the algorithms on them. Figure 15 shows that OBMA_S and OBMA_B continue to be the most efficient for all the route table sizes. Lulea and Poptrie also scales. Poptrie consumes more space than Lulea when the table is small.

C. Lookup Speed

1) Single-threading: Figure 16 shows the lookup speeds of the algorithms on the three route tables. OBMA_S$^-$ means

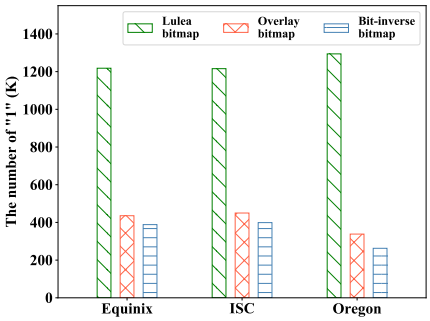


Fig. 13. The number of 1s in Lulea, overlay and bit-inversion bitmaps on Equinix, ISC and Oregon

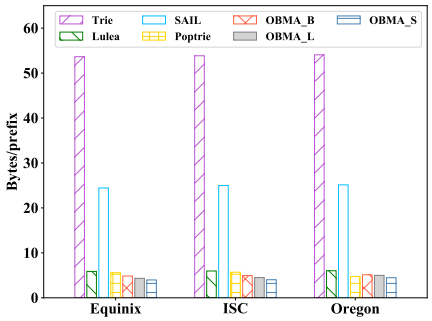


Fig. 14. Storage efficiency of Trie, Lulea, SAIL, Poptrie, OBMA_B and OBMA_S on the tables

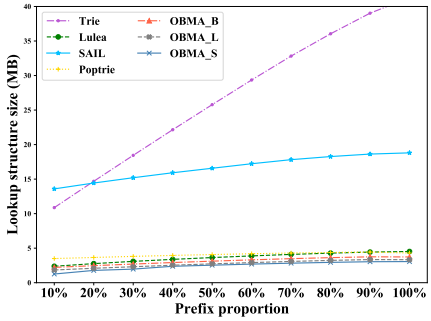


Fig. 15. Storage overhead of the algorithms over different prefix proportions of Equinix

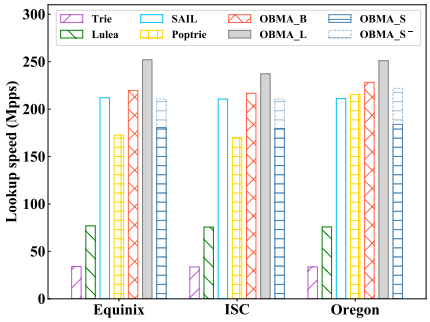


Fig. 16. Lookup speed of the algorithms on Equinix, ISC and Oregon

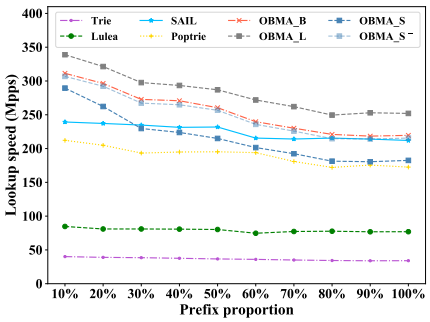


Fig. 17. Lookup speed of the algorithms over different prefix proportions of Equinix

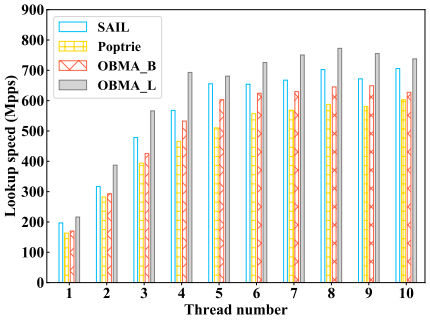


Fig. 18. lookup speed of SAIL, Poptrie,OBMA_B and OBMA_L with Multi-threading on Equinix

OBMA_S without the sparse chunk optimization. OBMA_L is the fastest, achieving about 252.02, 237.21 and 251.01 Mpps on the three tables, respectively. The second best is OBMA_B which achieves 219.56, 216.74, and 228.19 Mpps lookup speeds on the three tables, respectively. The speed is fast enough to support small-packet line-speed forwarding for a 100Gbps link.

OBMA_S is slower than OBMA_B, but it is faster than Poptrie on Equinix and ISC table. To reveal the reason, we use Intel VTune Amplifier [18] to analyze the codes and find that the bottleneck is in the false branch prediction when decoding the chunk type. Therefore, we test OBMA_S$^-$ without the sparse chunk optimization, and its lookup speed is significantly improved and only slightly slower than OBMA_B. The storage of OBMA_S$^-$, however, increases by 0.75 byte per prefix. Poptrie is better than SAIL on the Oregon table, but performs poorly on the other two tables, which reveals that the performance of Poptrie strongly correlates with the structure of routing tables.

For scalability test, we run the algorithms on different sized tables derived from the Equinix table and show the lookup speeds in Figure 17. Lulea and Trie's lookup speeds are slow but stable. Other faster algorithms show a speed reduction as the table size increases. The lookup speed of OBMA_B and OBMA_S$^-$ are visibly faster than SAIL up to 80% of prefix proportion. OBMA_S is faster than SAIL when the table size is small, because the sparse chunk ratio is high and the probability of false branch prediction is reduced. OBMA_L is the fastest for all the table sizes. To counter the lookup speed loss due to large tables, we can use the multithreading technique.

2) Multi-threading: Figure 18 shows the lookup speeds of SAIL, Poptrie, OBMA_B and OBMA_L under different number of lookup threads. While OBMA_L is still the fastest one, OBMA_B is a little slower than SAIL. This is because we cannot deploy the DPA and chunk lists in the stack space as local variables in multiple threads, and we have to allocate heap space for them. All the algorithms see a rising trend until the thread number reaches 8, and the lookup speeds peak at 706, 602, 772, and 650 Mpps, respectively. This is because the CPU has 4 cores and 8 hardware threads in total. When the number of software threads exceeds 8, the lookup performance cannot be further improved due to hardware limitations.

D. Update Performance

We measure the update speeds of Trie, Lulea, SAIL, Poptrie, and OBMA, and show the results in Figure 19. Unsurprisingly, Trie exhibits the fastest update speed, which sets an upper bound for the update performance. Lulea handles incremental updates poorly with an update speed so low that the value is hardly visible. Poptrie and SAIL are slightly better than Lulea. OBMA_B is the only one whose update performance is close to the upper bound. It is 15 and 37 times faster than Poptrie and SAIL in terms of update speed.

The update speed of OBMA_L is a half of that of OBMA_B due to OBMA_L's fixed group size which makes rebuilding the group structure for updates more time consuming. OBMA_S is even slower than OBMA_L for updates due to the extra BIS processing. However, in general the OBMA family is much

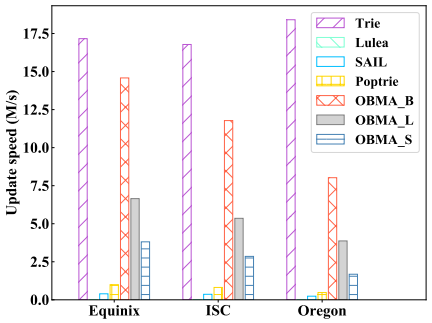


Fig. 19. Update speed of Trie, Lulea, SAIL, Poptrie and OBMA_B on Equinix, ISC and Oregon

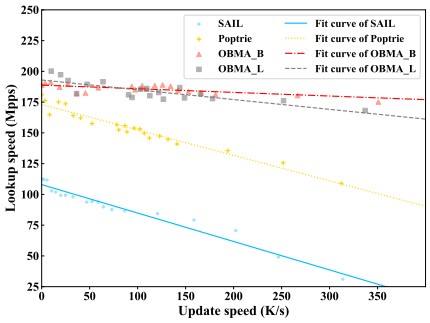


Fig. 20. Joint performance of lookup and update with single-threading on Equinix

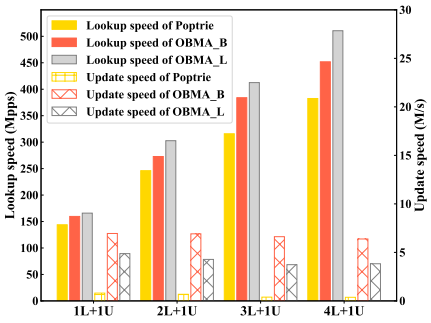


Fig. 21. Joint performance of lookup and update with multi-threading on Equinix

TABLE III
AVERAGE NUMBER OF POINTERS AND PDA ELEMENTS MODIFIED, AND EXTRA STORAGE NEEDED BY ZERO-INTERRUPTION TIME

	Equinix	ISC	Oregon
Average changed pointers	0.40	0.44	0.82
Average modifed DPA elements	0.64	0.82	1.53
Average additional memory	4.05 B	5.22 B	10.79 B

TABLE IV
UPDATE PERFORMANCE OF OBMA_B WITH DIFFERENT OPTIMIZATIONS

	Storage (MB)	Lookup (Mpps)	Update (M/s)
OBMA_L (0-3-3)	3.33	252.02	6.65
Fixed Group (1-2-3)	3.90	216.21	10.52
Fixed Group (2-1-3)	5.08	212.04	13.17
Fixed Group (3-0-3)	7.45	209.84	14.83
Adaptive grouping$^-$	3.73	219.56	10.81
Adaptive grouping			14.58

better than the other state-of-art algorithms in terms of the update performance.

OBMA_B's update performance varies on different tables. We measure some micro indicators to explore the underlying reason. 1) *Average changed pointers*, all the chunk or group pointers that are changed; 2) *Average modified DPA elements*, all the DPA elements that are modified; 3) *Average extra memory*, all the required memory that the new data structures occupy. The results are reported in Table III. In general, an update by OBMA_B needs to change less than 1 pointers and modify at most 1.53 DPA elements. The average additional memory is just a few bytes. All of these contribute to the high update speed. The largest values occur on the Oregon table, which explain the relatively low update speed on it.

Table IV shows the update performance of OBMA_B on the Equinix table when using different optimization parameters and configurations. The 24th-level optimization is configured by default. *Adaptive grouping*$^-$ means the adaptive grouping optimization is turned off. For the fixed grouping configurations, having more groups means a higher update performance, a lower lookup performance, and a higher storage cost. For example, when using 8 groups per chunk (i.e., the grouping configuration of (3-0-3)), the update speed can achieve 14.83 M/s, which is much faster than the case of using one group per chunk (i.e., the grouping configuration of (0-3-3)). However, the latter case has better lookup and storage performance. The adaptive grouping and the 24th-level optimization can help to achieve a well-balanced performance on lookup, update, and storage. Without the 24th-level optimization, the update speed will drop by about 26%.

E. Joint Performance of Lookups and Updates

In real applications, the lookup and update processes always work at the same time. Therefore, it is important to understand how these two processes interact with each other and what is their joint performance.

1) Single-threading: We first run experiments to check the joint performance of lookups and updates when both are running in a single thread. The input trace is a mix of IP addresses and update prefixes.

Figure 20 shows the scattered lookup-update performance for SAIL, Poptrie, OBMA_B, and OBMA_L on the Equinix table and their linear regression lines. A higher line means faster lookup speed and a flatter line means the update process has less impact to the lookup process. OBMA_B and OBMA_L are clear winners. When the update speed is high, OBMA_B performs better than OBMA_L for lookups. This result implies that OBMA_L should be applied in scenarios with relatively lower update requirements.

2) Multi-threading: Figure 21 shows the joint lookup and update performance for Poptrie, OBMA_B and OBMA_L on the Equinix table with multiple threads, where "$nL + 1U$" means n threads for lookups and one thread for updates. Since SAIL does not provide a multi-threading version supporting parallel lookups and updates, we only compare Poptrie, OBMA_B and OBMA_L. To realize the zero-interrupt updates, we use the scheme described in Section VI-C on OBMA_B. The lookup speed keeps increasing as the number of lookup threads increases, and OBMA_L keeps the highest position. However, compared with the case that only lookup threads are tested as shown in Figure 18, they all see some lookup speed drop. This is because the update thread modifies the lookup structure, which nullifies many cache lines for lookups. Since OBMA_B and OBMA_L only conducts partial structure updates but Poptrie always replaces the entire lookup structure, they has a higher cache hit rate and less influenced by the update process. The update speeds keep almost stable, and OBMA_B outperforms others due to its update optimizations. Overall, with four lookup threads, OBMA_L and OBMA_B supports more than 500 Mpps and 450 Mpps lookup throughput with zero-interrupt updates, respectively.

VIII. Conclusion

The OBMA family improves the memory efficiency and lookup/update performance of software-based route lookup and update algorithms. With a series of architectural and engineering optimizations, OBMA_B supports fast and zero-interrupt updates, which is effective in reducing the packet buffer size in routers, accelerating the network state convergence, and improving the network QoS. OBMA_L further optimizes OBMA_B's lookup throughput and OBMA_S optimizes OBMA_B's storage, which provide us more flexibility when choosing algorithms in different application scenarios. Since OBMA is trie-based, it can be naturally applied to lookups on IPv6 networks or multiple virtual route tables in an NFV environment.

Acknowledgements

This work is sponsored by NSFC (61373143, 61432009, 61872213).

References

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

[2] 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. ACM, 2015, pp. 57–70.

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

[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 *IEEE International Conference on Network Protocols (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] 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.

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

[12] 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.

[13] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a gpu-accelerated software router," in *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, 2010, pp. 195–206.

[14] Y. Li, D. Zhang, A. X. Liu, and J. Zheng, "Gamt: a fast and scalable ip lookup engine for gpu-based software routers," in *Proc. of ACM/IEEE ANCS*. IEEE Press, 2013, pp. 1–12.

[15] 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 *IEEE International Conference on Network Protocols (ICNP)*, 2013, pp. 1–10.

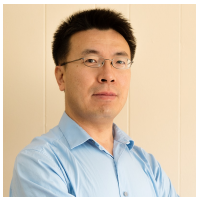
[16] P. G, "Kombinatorische anzahlbestimmungen für gruppen, graphen und chemische verbindungen," *Acta mathematica*, vol. 68, no. 1, pp. 145–254, 1937.

[17] J. H. Redfield, "The theory of group-reduced distributions," *American Journal of Mathematics*, vol. 49, no. 3, p. 433, 1927.

[18] Intel vtune amplifier. [Online]. Available: https://software.intel.com/en-us/vtune

Chuwen Zhang Received the B.S. degree in communication engineering from Northwestern Polytechnical University, Xi'an, China, in 2015. He is Currently working toward the Ph.D. degree in Computer Science at Tsinghua University, Beijing, China and his advisor is Dr. Bin Liu. His research interests include high-performance switches/routers, named data networking and vehicle networks.

Yong Feng received the B.S. degree in software engineering from Northwestern Polytechnical University, Xi'an, China, in 2018. He is currently working toward the Ph.D. degree in Computer Science at Tsinghua University, Beijing, China and his advisor is Dr. Bin Liu. His research interests include route compression, network measurement and congestion control.



Haoyu Song received the BE degree in electronics engineering from Tsinghua University, in 1997, and the MS and DSc degrees in computer engineering from Washington University in St. Louis in 2003 and 2006, respectively. He is a senior principal network architect with Futurewei Technologies, USA. His research interests include software defined network, network virtualization and cloud computing, high performance networked systems, algorithms for network packet processing and intrusion detection. He is a senior member of the IEEE.

Ying Wan received the B.S. degree in Communication Engineering from Northwestern Polytechnical University in 2016. He is currently working toward the Ph.D. degree in Computer Science at Tsinghua University, Beijing, China and his advisor is Dr. Bin Liu. His research interests include high performance network algorithm, and software defined network.

Wenquan Xu Received the B.S. degree in microelectronics science and engineering from Northwestern Polytechnical University, Xi'an, China, in 2017. He is Currently working toward the Ph.D. degree in Computer Science at Tsinghua University, Beijing, China and his advisor is Dr. Bin Liu. His research interests include deep learning, named data networking and vehicle networks.

Bin Liu received the M.S. and Ph.D. degrees in computer science and engineering from Northwestern Polytechnical University, Xi'an, China, in 1988 and 1993, respectively. He is now a Full Professor with the Department of Computer Science and Technology, Tsinghua University, Beijing, China. His current research areas include high-performance switches/routers, network processors, high-speed security, and greening the Internet. He has received numerous awards from China, including the Distinguished Young Scholar of China and won the inaugural Applied Network Research Prize sponsored by ISOC and IRTF in 2011.