Computer Networks 251 (2024) 110590

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

MaP: Increasing node capacity of programmable cloud gateways

Ke Xu [a,b], Donghong Jiang [a], Yanbiao Li [a,c,*], Xin Wang [d], Dafang Zhang [b], Gaogang Xie [a,c]

[a] Computer Network Information Center, Chinese Academy of Science, Beijing, China
[b] College of Computer Science and Electronic Engineering, Hunan University, Changsha, China
[c] School of Computer Science and Technology, Chinese Academy of Science, Beijing, China
[d] Department of Electrical and Computer Engineering, Stony Brook University, NY, USA

ARTICLE INFO

Keywords:
Virtual routing lookup
Programmable switch
Memory compression

ABSTRACT

Virtual routing table lookup is a crucial operation in multi-tenant cloud gateways, the essential devices to enable the elastic management of Virtual Private Clouds (VPCs). However, software-based cloud gateways face performance bottlenecks when handling ever-increasing traffic. To address this challenge, a recent trend involves accelerating virtual routing table lookups using programmable switches, with Sailfish being the state-of-the-art solution. Nonetheless, Sailfish's table scale on a single programmable switch is limited by the ASIC memory sizes. We introduce a novel scheme, named MaP (Merge and Partition), to significantly increase the memory efficiency on a single Tofino 1.0 programmable switch. MaP exploits the similarities in routing table entries across various VPCs for table compression and leverages the hardware architecture of programmable chips to minimize the memory occupancy. Performance evaluations using real-world virtual routing tables demonstrate that our approach reduces the memory cost from 25% to 47.7% compared to Sailfish, without negatively impacting the throughput and latency. With MaP, a single programmable switch can manage a massive table containing over 2 million entries for nearly 1 million VPCs, while only occupying 59.0% of memory and 63.0% of stages.

1. Introduction

The exponential growth in the number of cloud service customers [1] has led to a substantial increase in the scale of cloud services. To address the need for elastic resource isolation, security, and personalized services, the concept of virtual private clouds (VPC) [2] has been introduced. By assigning a dedicated virtual routing table to each VPC, overlay network protocols like VXLAN [3] enable customers to seamlessly access cloud resources, as if they were an integral part of their own dynamically provisioned data centers.

In a multi-tenant public cloud environment, the cloud gateway assumes a pivotal role as the central hub for efficiently managing inbound and outbound traffic. The gateway faces several critical challenges to function efficiently and reliably. Firstly, the cloud gateway must exhibit robust processing performance to effectively handle traffic at the scale of gigabits or terabits. Secondly, it must demonstrate the exceptional flexibility to swiftly deploy integrated network functions in response to the demands driven by Software-Defined Networking (SDN) [4] and Network Function Virtualization (NFV) [5]. Thirdly, it must possess ample capacity to handle the rules needed for the extensive volume of virtual routing entries.

To address these challenges, major public cloud vendors have invested considerable efforts. Existing methods are commonly software-based [6–8] to meet the need of flexibility, and often rely on a cluster-based architecture to handle the large number of rules. Although they are helpful in reducing the rule space, the compression ratio is still not very high and the time taken to search for a rule is still large. A number of devices may work together as a cluster to complete the tasks, at a higher cost in building the gateway and complexity in coordinating the functions of multiple devices. The emergence of programmable switches offers an alternative for constructing high-performance cloud gateways. Programmable switch ASICs outperform x86 processors typically used in software-based approaches in terms of lookup and forwarding performance, while still offering sufficient programmability for agile deployment of new network functions. Compared to conventional software-based methods involving multiple dedicated devices, programmable switches can substantially reduce hardware requirements, maintenance overhead, and troubleshooting efforts [9]. Despite the potential, the limited capacity of programmable switches poses a great challenge in handling a huge number of rules.

* Corresponding author.
E-mail addresses: permission@hnu.edu.cn (K. Xu), jiangdonghong@cnic.cn (D. Jiang), lybmath@cnic.cn (Y. Li), x.wang@stonybrook.edu (X. Wang), dfzhang@hnu.edu.cn (D. Zhang), xie@cnic.cn (G. Xie).

https://doi.org/10.1016/j.comnet.2024.110590
Received 28 December 2023; Received in revised form 29 May 2024; Accepted 6 June 2024
Available online 28 June 2024
1389-1286/© 2024 Elsevier B.V. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

Intuitively, expanding the number of devices (i.e., programmable switches) to construct a cluster can broaden the rule capacity by distributing rules among them. However, this approach overlooks financial costs, maintenance expenses, and energy consumption. Therefore, rather than initially adopting this solution, we aim to enhance the capacity of individual devices. We observe that routing entries of different Virtual Private Clouds (VPCs) have strong similarities, and we propose to take advantage of these similarities to largely compress the virtual routing table. Different from a common route lookup process that finds a target VPC first and then looks for the destination within the VPC through IP prefix, we propose a novel **MaP** framework that transforms the virtual lookup process into two phases, IP prefix lookup followed by VPC exact match. This design significantly reduces the memory consumption while enhancing the overall lookup efficiency. In addition, in order to offload rules onto programmable switches, we introduce specific designs to take into consideration the diverse memory footprints associated with different memory types including SRAM and TCAM, and key lengths. To optimize the capacity to hold more rules on a single device, we divide the rule set into multiple independent groups. Among them, one group is responsible for holding rules of variable key lengths, with an LPM (Longest Prefix Match) table to perform the longest prefix match. Additionally, there are multiple EM groups, each holding rules of a specific key length and utilizing two EM (Exact Match) tables to perform an exact match. We propose a division algorithm to minimize the memory footprint, so we can maximize the rule capacity while efficiently utilizing the resources of the programmable switch.

Our contributions can be summarized as follows:

1. Based on our observation of the similarity among the real-world VXLAN routing rules, we propose a Merge-and-Partition scheme to reduce the number of IP prefixes. To minimize the memory cost, we propose a dynamic programming algorithm to optimally partition the SRAM-based table.
2. To further improve the memory efficiency of the partition deployed in the Tofino 1.0 programmable switch, we propose TCAM-SRAM, a hybrid partition scheme that adjusts the rule division by moving some rules from the EM table to the LPM table.
3. We prototype our scheme with a programmable switch, and conduct extensive experiments by comparing with the state-of-the-art in terms of memory cost to demonstrate that our scheme can significantly reduce the memory cost of the deployment of a virtual routing table, and meanwhile do not impact the throughput and latency.

The remainder of the paper is organized as follows. Section 3 introduces our important observation of prefix similarity among the routing tables of different VPCs, and outlines the MaP methodology which involves routing entry merge, prefix table partition, and an optimization algorithm aimed at minimizing the memory cost of deploying on the programmable switch. Section 4 introduces the prototype implemented with a programmable switch. A performance evaluation of the proposed scheme is conducted in Section 5. Finally, Section 7 concludes the paper.

2. Motivation

As the first programmable switch based gateway design, Sailfish [9] uses Tofino switches and leverages some fundamental primitives directly supported by the ASIC to deploy the rules with high priority. However, such a straightforward method is hard to get a high memory efficiency of the ASIC. Fortunately, the following important observation drive us carefully organize the rule entries' placement to minimize the overall memory cost.

There are three blocks of IP addresses reserved by the Internet Assigned Numbers Authority (IANA) for private Internets [10]. In general, a VPC will be assigned with a sub-block of IP addresses from one of them. Intuitively, there should be certain degree of similarity among the routing tables of different VPCs. Our observations on 5 real-world VXLAN routing tables indeed support this hypothesis. As shown in Fig. 1, the ratio of distinct IP prefixes of sub-nets across all VPCs is as low as 10.3%, indicating a wide space for compression. Besides, although a high portion of IP prefixes are shared by two or more VPCs, more than 90.5% of them are shared by less than 8 VPCs. This indicates that the number of IP prefixes shared among multiple VPCs is commonly distributed between 2 and 8. These observations will serve as the base to guide our design of efficient algorithms to largely compress the memory consumption of routing entries while ensuring a fast lookup in Section 3.4.

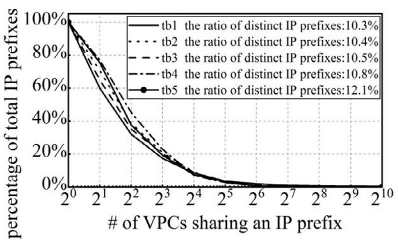


Fig. 1. CCDF with 5 real tables, referred to as tb1 through tb5.

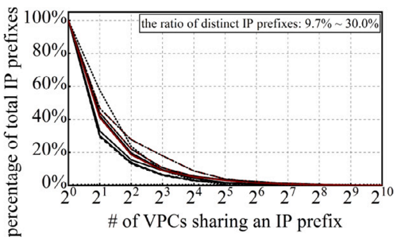
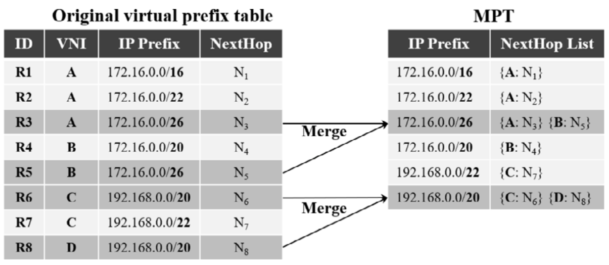


Fig. 2. CCDF with synthetic tables.

What is interesting, the similar trend can also be observed (Fig. 2) with synthetic data sets. For generating each synthetic data set, we follow the same IP address allocation scheme mentioned at beginning to allocate a sub-block of the reserved addresses to each VPC. This indicates that such an IP address allocation scheme results in the significant IP prefix similarity among the virtual routing tables of VPCs. Accordingly, to explore the similarity if IP prefix for compression, we should consider the whole space of virtual routing tables, regardless of the VPCs.

These important observations drive us to revisit a classic strategy proposed for virtual routers to compress multi-FIBs (Forwarding Information Base) [11–13] by merging multiple forwarding rules sharing the same IP prefix into one entry and recording the corresponding FIB identity in a reserved and fixed-size data structure. However, the above solutions are not appropriate for the routing table compression on the current multi-tenant cloud gateway implemented by the programmable switch. Firstly, a reserved fixed-size memory design is unsuitable for a large-scale rule set. Secondly, the fundamental primitive of these designs, which involves performing a second phase lookup on a certain block of fixed-size SRAM indexed by the output of the first phase lookup, is impractical on current programmable switch architectures.

In order to overcome the obstacle of memory compression with the merge of routing table entries across different VPCs, the key challenge is the management of millions of IP prefixes with multi-ownership from millions of VPCs at low cost for storing the routing table while high query speed for the next hop of transmission. In the rest of this section, we will present the Merge-and-Partition (**MaP** in short) approach to address this challenge step by step, by utilizing the hardware architecture of programmable ASICs with novel table designs.


Original virtual prefix table
MPT
Merge
Merge

ID	VNI	IP Prefix	NextHop
R1	A	172.16.0.0/16	N_1
R2	A	172.16.0.0/22	N_2
R3	A	172.16.0.0/26	N_3
R4	B	172.16.0.0/20	N_4
R5	B	172.16.0.0/26	N_5
R6	C	192.168.0.0/20	N_6
R7	C	192.168.0.0/22	N_7
R8	D	192.168.0.0/20	N_8

IP Prefix	NextHop List
172.16.0.0/16	{A: N_1}
172.16.0.0/22	{A: N_2}
172.16.0.0/26	{A: N_3} {B: N_5}
172.16.0.0/20	{B: N_4}
192.168.0.0/22	{C: N_7}
192.168.0.0/20	{C: N_6} {D: N_8}

Fig. 3. Entry merge of an example of 8 routing entries from 4 VPCs.

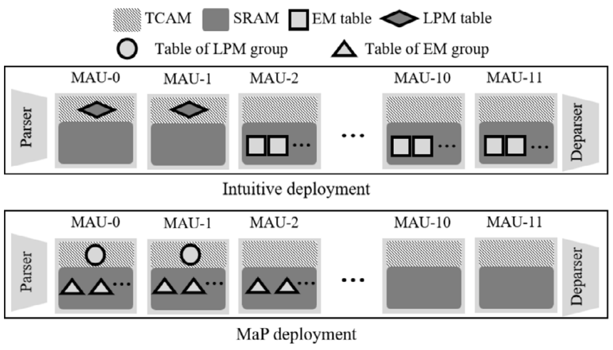


Fig. 4. Assuming that only the ingress pipeline of a switch with Tofino 1.0 architecture works while the egress pipeline is empty, the intuitive deployment of MPT in the upper and MaP deployment in the lower one.

3. Design

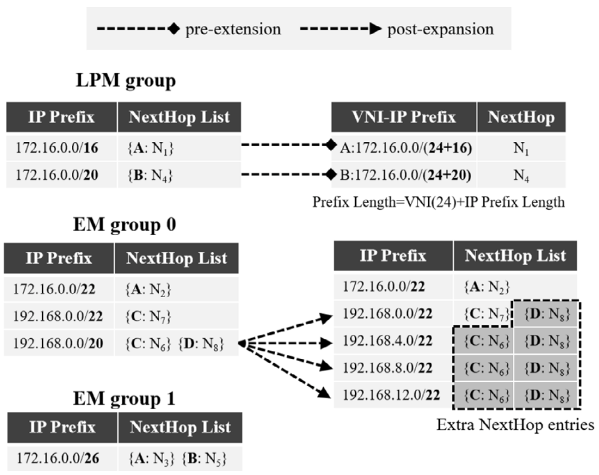
3.1. Routing entry merge and prefix table partition

As shown in Fig. 3, to compress the space occupied by routing entries, our proposed scheme merges all entries sharing the same IP prefix into one and associate it with a list of {*vni*[1], *nexthop*} pairs, with the VNI to identify the original VPC the corresponding entry belongs to, and we name the table produced as the Merged Prefix Table (MPT). To look up MPT, the IP prefix is first searched to find the associated {*vni*, *nexthop*} list, and then the *nexthop* with the *vni* is found from the list. It should be noted that a virtual routing table lacking such prefix similarity could hinder MaP's compression capabilities. For example, in an extreme scenario where each virtual private cloud (VPC) has a unique IP prefix, converting the original virtual prefix table into a MPT would not result in any reduction of IP prefixes.

Here comes the most important design issue with our approach: **how to map MPT to the hardware pipeline(s) on the switching ASIC?** Fig. 4 illustrates the pipeline architecture of the ASIC. Initially, the parser processes the raw packet and extracts some important data, such as the headers of certain protocols or the structured payload, while storing the remaining content aside. The extracted data then go through a sequence of the pipeline's Match-Action Units (MAUs), with each MAU utilizing its inner tables for data manipulation. A MAU consists of a TCAM for building the Longest Prefix Match (LPM) table to efficiently handle the lookup based on prefixes of varying lengths, and an SRAM to construct the Exact Match (EM) table for the precise match of a specific data entry. Upon traversing through all MAUs, some or all of the extracted data, along with the stored content, are reassembled to reconstruct the complete packet, ready to be transmitted outwards.

Under such a pipeline architecture, a straightforward solution is to construct one LPM table to store all prefixes and several EM tables where each holds (rules of) a nexthop list corresponding to a certain

[1] VNI stands for VXLAN Network Identifier and works as the ID of a VPC.


pre-extension
post-expansion
LPM group
Prefix Length=VNI(24)+IP Prefix Length
EM group 0
EM group 1
Extra NextHop entries

IP Prefix	NextHop List
172.16.0.0/16	{A: N_1}
172.16.0.0/20	{B: N_4}

VNI-IP Prefix	NextHop
A:172.16.0.0/(24+16)	N_1
B:172.16.0.0/(24+20)	N_4

IP Prefix	NextHop List
172.16.0.0/22	{A: N_2}
192.168.0.0/22	{C: N_7}
192.168.0.0/20	{C: N_6} {D: N_8}

IP Prefix	NextHop List	
172.16.0.0/22	{A: N_2}	
192.168.0.0/22	{C: N_7}	{D: N_8}
192.168.4.0/22	{C: N_6}	{D: N_8}
192.168.8.0/22	{C: N_6}	{D: N_8}
192.168.12.0/22	{C: N_6}	{D: N_8}

IP Prefix	NextHop List
172.16.0.0/26	{A: N_3} {B: N_5}

Fig. 5. Table partition.

prefix of the LPM table, as shown in the upper pipeline in Fig. 4. There are three critical issues with this solution. Firstly, although the IP prefixes are merged, maintaining a dedicated EM table for each unique nexthop list generates a huge number of EM tables and leads to the exhaustion of pipeline stages, as an MAU can only store a limited number of tables even when there is still memory to store more. Secondly, a single IP address may match multiple prefixes in the LPM table, and the correct nexthop may be located in any of the EM tables associated with these prefixes. However, an LPM table can only return the result matched with the longest prefix during the lookup process, providing no guarantee that the correct nexthop is located in the associated EM table. Thirdly, the dependence between the LPM table and the EM tables make them unable to be deployed in the same MAUs [14], which results in unnecessary waste of MAU resources.

To address these issues, we propose to **partition** the MPT into multiple groups. There is two types of groups, an *LPM group* consisting of the table of LPM type and an *EM group* consisting of the table of EM type. The proposed MaP consists of at most one LPM group but a number of EM groups. When the size of a table in the LPM group or EM group exceeds the capacity of one MAU, the rules not covered by that MAU's capacity will be transferred to subsequent MAUs. The lack of dependence across groups allows for a more compact placement of MaP, as shown in the lower part of the pipeline in Fig. 4. For a given packet, the destination IP address and the VNI of the destination VPC are matched with every group separately. The matched results are gathered together and the one with the longest prefix is selected to be the final result. For example, the MPT shown in Fig. 3 is partitioned into one LPM group and two EM groups shown in Fig. 5. To look for the destination IP address 172.16.0.0 for VPC B, the LPM group yields N_4 with a prefix length of 20, the EM group 1 outputs N_5 with a prefix length of 26, while the EM group 0 does not output any result. Finally, N_5 is selected as the final result as it is associated with the longest prefix.

Both the LPM group and the EM group can independently construct and hold MPT, and are subject to the principle of **zero-loss.**[2] MaP employs an LPM-EM hybrid scheme to minimize the overall memory cost. In the remainder of this section, we will provide the detailed designs of the LPM group (Section 3.2) and the EM group (Section 3.3), analyze the distinctive features of these two groups including their respective advantages and disadvantages, and illustrate how the hybrid

[2] Zero-loss refers to the principle that the memory efficiency of the design of a group must be better than or at least keep the same as that of the original prefix table without entry merge.

scheme benefits the memory cost (Section 3.4). Moving forward, in Section 3.5 and Section 3.6, we will present a strategy to minimize our memory cost.

3.2. Design of the LPM group

Intuitively, an LPM table is naturally suitable to store all IP prefixes, and the nexthop lists can be held in an exact match table by adding a tag[3] to each nexthop entry [15]. However, such a design cannot guarantee the correctness since the output of the IP prefix match should include all matched prefixes instead of just the longest one. For example, the two prefixes in the LPM group shown in Fig. 5 are owned by VPC A and VPC B respectively. To match the IP address 172.16.0.0 with the VPC A, the accurate result (N_1) is associated with the prefix 172.16.0.0/16 instead of the prefix 172.16.0.0/20 which has a longer prefix length.

To address this issue, we use a *pre-extension* scheme to prepend the VNI of the VPC to each IP prefix as shown in Fig. 5. In this way, any two prefixes belonging to different VPCs are distinguished, ensuring the correctness of the longest prefix matching. As such, an LPM group can be mapped to an LPM table directly. In Fig. 5, The extra key length of 24 after pre-extension is due to the appending of VNI, which is 24-bits in most virtual routing protocols, like VXLAN. The lookup is processed with the longest prefix matching against the LPM table using the key that combines the VNI and the IP address together.

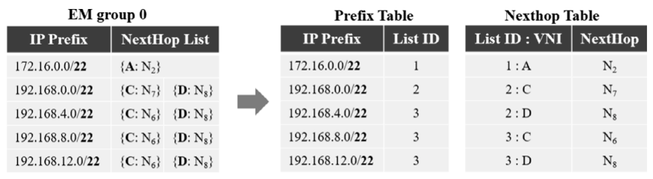
Although *pre-extension* scheme ensures the correct lookup in one step and does not violate the **zero-loss** principle, the same IP prefix can no longer be merged, and we lose the benefit of memory compression. Besides, the TCAM space (used to build the LPM table) is scarce in the ASIC pipeline, so the capacity to hold the LPM group is limited. On the other hand, there are much more SRAM resource in the ASIC that can be used to hold the EM groups, so we introduce the details of our design for EM table next.

3.3. Design of the EM group

In addition to memory size, another crucial distinction between TCAM and SRAM lies in their support for the lookup against variable lengths of IP prefixes. Unlike the LPM table, which can accommodate prefixes of different lengths, the EM table held by SRAM requires all rules to have the same fixed key length. However, due to the limitation of the number of tables, maintaining a dedicated EM group for each key length is impractical. To address this issue, we propose the *post-expansion* approach. For an EM group whose longest IP prefix has the length L, we expand all shorter IP prefixes to align their lengths with L. For example, $L = 22$ for the EM group 0 in Fig. 5. The IP prefix 192.168.0.0/20 is then expanded to generate 4 sub-prefixes 192.168.12.0/22, 192.168.8.0/22, 192.168.4.0/22 and 192.168.0.0/22, which inherit its nexthop list (N_6 for VPC C and N_8 for VPC D). If a generated sub-prefix has an overlap with an existing one, they will be merged by combining their respective nexthop lists. In case the two have the same VNI corresponding to different next hops, the nexthop entry of the longer IP prefix will be used. In Fig. 5, to resolve the conflict during the merge of the existing IP prefix 192.168.0.0/22 and the sub-prefix of 192.168.0.0/20, the final nexthop of VPC C is set to N_7.

After the post-expansion, all IP prefixes in an EM group are of the same length and thus can be processed with an exact match. To enable an efficient lookup, we further divide information in each EM group into a *prefix table* and a *nexthop table* (Fig. 6). We assign a unique ID to each nexthop list, and every prefix in the *prefix table* is mapped to the ID of its associated nexthop list. Obviously, prefixes associated with the same nexthop list are mapped to the same list ID. In the *nexthop table*,

[3] The tag is used to record the original nexthop list the entry belongs to.



EM group 0

IP Prefix	NextHop List	
172.16.0.0/**22**	{**A**: N_2}	
192.168.0.0/**22**	{**C**: N_7}	{**D**: N_8}
192.168.4.0/**22**	{**C**: N_6}	{**D**: N_8}
192.168.8.0/**22**	{**C**: N_6}	{**D**: N_8}
192.168.12.0/**22**	{**C**: N_6}	{**D**: N_8}

Prefix Table

IP Prefix	List ID
172.16.0.0/**22**	1
192.168.0.0/**22**	2
192.168.4.0/**22**	3
192.168.8.0/**22**	3
192.168.12.0/**22**	3

Nexthop Table

List ID : VNI	NextHop
1 : A	N_2
2 : C	N_7
2 : D	N_8
3 : C	N_6
3 : D	N_8

Fig. 6. Two-phase exact matching with the EM group.

Table 1
Notation.

Notaion	Definition (# of)
N_o	Original IP prefixes
N_m	Merged IP prefixes
N_l	Distinct nexthop lists after the post-expansion
N_p	Entries in the prefix table of an EM group
N_n	Entries in the nexthop table of the EM group

every entry is tagged with a key formed with the ID of the nexthop list [15] and the *vni* of a *(vni, nexthop)* pair and mapped to the *nexthop* of this pair.

The lookup of an EM group is realized in two steps. First, the first L bits of the input IP address are extracted as a key to match against the prefix table to obtain a list ID. Then the returned list ID is combined with the input VNI to form a new key used to match against the nexthop table and obtain the final result. For example, to look up the IP address 192.168.0.0 for the VPC C in the EM group shown in Fig. 6, we get a list ID 2 by matching the first 22 bits of this IP address against the prefix table. Then we construct the key by combining this list 2 with the VNI of VPC C, i.e., $2 : C$, to match against the nexthop table and get the result N_7.

3.4. Memory efficiency analysis

Both the LPM group and EM group have their own advantages and disadvantages. To make the comparison, we use the metric *memory efficiency*.

For the convenience of discussion, we first introduce 5 notations in Table 1. Suppose the length of the longest IP prefix of the original table is L, within an LPM group, they are recorded in the LPM table with an $L + 24$ bits key. The prefixes in an EM group are merged into N_m ones, which are then expanded to N_p IP prefixes of L-bit along with N_l nexthop lists. These nexthop lists are further expanded to N_n nexthop entries. So the prefix table of this EM group carries N_p entries whose key length is L, while the nexthop table carries N_n entries whose key length is $24 + \log N_l$.

The key's length of *nexthop table* of a EM group is $\log N_l$, which is generally less than L. Therefore, the key lengths of the two tables for an EM group are smaller than that of the LPM table. The tables for EM group and LPM group are SRAM-based and TCAM-based, respectively. In an MAU, not only that the total memory for the SRAM-based tables is larger than the memory for the TCAM-based tables, but also that a single memory block could hold more rule entries for SRAM type. Consequently, the memory cost of implementing an LPM table is usually higher than that of implementing an EM table with the same number of entries. Suppose the memory efficiency of implementing an EM table is X times that of implementing an LPM table, we can roughly deduce that managing a set of merged prefixes as an EM group is more memory efficient than managing it as an LPM group, as long as the sum of N_p and N_n is smaller than $X \times N_o$. This holds true in all cases in our experiments. For example, with a real world VXLAN routing table, we implement it with 10 EM groups and calculate all parameters in total. N_o is more than 30 times larger than N_p and is almost the same as N_n, while X is approximately 2.0 according to a number of tests with the **Tofino** chip we used.

As such, if there is only one choice to implement MAP, EM group is more memory efficient. However, we found that with the construction of LPM group for part of the routing entries, the overall memory efficiency can be further improved. As mentioned in Section 3.3, the reason of using the *post-expansion* scheme is because an EM group does not support variable prefix lengths, while the variable length is well supported by an LPM group and the corresponding LPM table lookup. For certain prefixes which generate many extra nexthop entries as a result of prefix alignment, an LPM group is a better choice to hold them. These prefixes are shared by a large number of VPCs and linked to long nexthop lists in MPT. Fortunately, from our observations in Section 2, there are very few heavily shared prefixes. Therefore, we can further reduce the memory cost by moving such prefixes from the EM group to the LPM group.

With the analysis above, we select an LPM-EM hybrid design to implement **MaP**. As the EM table has a high memory efficiency to hold the rule set, the EM group works as the primary to store prefixes. At the same time, for certain rules that generate too many extra ones by following the *post-expansion* scheme, we move them to a dedicated LPM group.

There remain two problems to resolve. The first is the determination of the key length of each EM group to minimize the memory cost. The second is the selection of critical prefixes to be moved to the LPM group. In the next two sections, we will introduce our resolutions for these two problems.

3.5. Partition optimization

Algorithm 1: Dynamic programming of table partition

```
    Input: $N$ length groups $\{L_i | 1 \le i \le N\}$
           and the partition number $P$.
1   calculate_cost_matrix();
2   $opt[0][0] \leftarrow 0$ ;
3   for $j \leftarrow 1$ to $P$ do
4       for $i \leftarrow 1$ to $N$ do
5           $opt[i][j] \leftarrow INF$;
6           for $k \leftarrow 0$ to $i-1$ do
7               $res \leftarrow opt[k][j-1] + cost[k][i]$;
8               if $res < opt[i][j]$ then
9                   $opt[i][j] \leftarrow res$;
10              end
11          end
12      end
13  end
```

The EM group is the key component of our approach to explore the prefix similarity for memory compression. Thus, we make the EM group the default and the preferred option to implement a partition of the MPT, while leaving the LPM group as the worst-case guarantee.

During the initialization phase, the original rules are organized into an IP prefix trie. Given our objective of achieving the global minimum memory cost for partitioning EM groups, we employ a dynamic programming method rather than a heuristic approach, which is unsuitable for resolving such a prefixes partition problem [16]. As shown in Algorithm 1, during the resolution process, we maintain two matrices, denoted as opt and $cost$, and two temporary constraints, represented by i and j, corresponding to the level of trie and maximum EM groups respectively. The matrix entry $opt[i][j]$ represents the minimum memory cost of dividing the prefixes on the first i levels into j EM groups, while $cost[i][j]$ signifies the memory cost associated with expanding all prefixes from level$_i$ to level$_{j-1}$ to level$_j$ with the *post-expansion* scheme. The matrix $cost$ can be precomputed before the partition optimization through a brute-force search. For a specified level range from level$_i$ to level$_j$ in the trie, all prefixes in this range will be "pushed" to level$_j$, fixing the lengths of both prefix and data. Considering the lengths of rules' key and data along with the total number of rules, we can roughly estimate the resource usage [15] based on the size of the memory block[4]. To precompute the matrix $cost$, we just need to traverse all possible level ranges of the IP prefix trie with a time complexity of $O(N^2)$. Srinivasan [16] employs a similar dynamic programming approach to minimize memory overhead in IP prefix lookup. However, our approach differs in that we combine the sizes of expanded prefixes' next-hop lists and the cost of entry placement to measure the cost of prefix expansion, rather than the number of expanded prefixes.

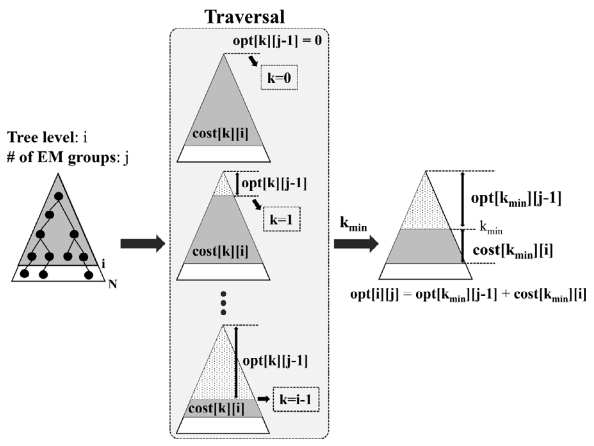


Fig. 7. Core process of partition optimization.

The core optimization process (referring to lines 5 to 11 in Algorithm 1) is illustrated in Fig. 7, the goal is to resolve $opt[i][j]$. During the traversal section in the current iteration, we explore all possibilities of a selected trie level k. For each k, the first i levels are divided into two parts, and the temporary $opt[i][j]$ is the sum of $opt[k][j-1]$ and $cost[k][i]$. In this calculation, $opt[k][j-1]$ represents the minimum cost of dividing the first k levels into $j-1$ groups (refer to the dot-filled triangle in the traversal part of Fig. 7), which has been resolved earlier. The number of EM groups is fixed at j, of which $j-1$ groups are already used to accommodate the first k levels' rules. Consequently, there is only one group for the rules from $level_k$ to $level_i$, where all of the rules from $level_k$ to $level_{i-1}$ must be expanded to $level_i$ via the *post-expansion* scheme introduced in Section 3.3. The memory cost of this expansion is stored in $cost[k][i]$ (referring to the shaded part of each triangle in the section marked by **Traversal** in Fig. 7). Following the traversal section, the minimum memory cost for these dividing attempts, along with its corresponding dividing level k_{min}, can be determined. The value of $opt[i][j]$ is the sum of $opt[k_{min}][j-1]$ and $cost[k_{min}][i]$. The maximum trie level and the maximum number of EM groups are denoted as N and P respectively. Finally, upon reaching these constraints for i and j (as outlined on lines 3 and 4 in Algorithm 1), the minimum $opt[N][P]$ is obtained. To deduce the optimized partition scheme, a backtracking process is executed, retracing each k_{min} stored in the previous iterations.

Assuming IPv4 with $N = 32$ and $P = 14$, the size of $cost$ and opt are only $32^2 = 4096$ and $32 \times 14 = 1792$. However, for a routing table consisting of several million entries, the above computation is time-consuming. We only run it once for a given routing table and mainly focus on the memory cost. However, massive rule updates may require reconfiguration, and we adopt a simple approach discussed in Section 4.4.

3.6. Group fine tuning

While the EM group exhibits superior memory efficiency compared to the LPM group table, we can further reduce the overall memory cost

[4] The size of SRAM and TCAM in the ASIC is typically under NDA.

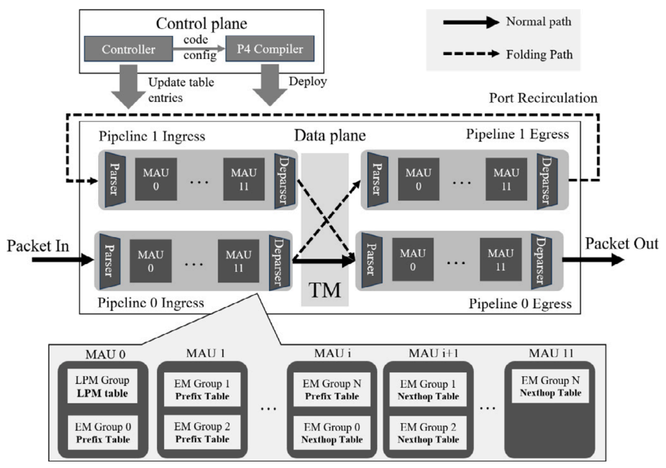


Fig. 8. Simplified system architecture of **MaP** with two pipelines, in which a packet is scheduled between ingress and egress via the TM (Traffic Manager), and the ingress and egress can reside on different pipeline.

with the help of the LPM group. When a prefix within the EM group is associated with a long nexthop list or comprises numerous sub-prefixes, applying the *post-expansion* operation results in significant memory overhead. Therefore, moving such prefixes from the EM group to the LPM group can effectively reduce the overall memory cost, because the LPM group supports holding rules of different prefix lengths.

For example, as shown in Fig. 3, R8 = {D, 192.168.0.0/20, N_8}[5] and R6 = {C, 192.168.0.0/20, N_6} are merged and managed by EM group 0 in Fig. 5. In Fig. 6, after *post-extension*, the prefix table entries generated by R8 are shared by other original rules (i.e., R6 and R7 in Fig. 3). Consequently, when relocating R8 to LPM group, we cannot eliminate these prefix table entries, but can remove its corresponding nexthop entries, including $\{2 : D, N_8\}$[6] and $\{3 : D, N_8\}$.

For a specific rule, we denote the overhead of relocating it as the **credit**. Assuming the move operation reduces x prefix entries and y nexthop entries while adding one entry to the LPM table, and the memory efficiencies of the EM group's prefix table and nexthop table are α and β times that of the LPM table, respectively, then the credit of moving this prefix is $\alpha x + \beta y - 1$. The maximum number of entries to relocate is constrained by the size of the LPM group, denoted as $Size_{LPM}$. During computing the partition of EM groups, we can manage all rules' credits and maintain their order with a time complexity of $O(C)$, where C represents the total number of rules. Subsequently, we can efficiently select $Size_{LPM}$ rules with the highest credits with a time complexity of $O(1)$.

4. System implementation

4.1. System architecture overview

As shown in Fig. 8, the *Tofino* ASIC of a commodity programmable switch has two pipelines. Via the traffic manager (TM), A packet can be scheduled between Ingress and Egress within the same pipeline or across two pipelines. In addition to the ASIC, The rule controller and p4 compiler are deployed in the Intel CPU integrated into the switch.

In the control plane, once being fed with the raw rule set, the controller generates the configuration of LPM groups and the partitions of EM groups in the form of P4 source files. After that, the P4 compiler checks the syntax and tries to allocate the necessary hardware resources.

[5] The format is {VNI, IP Prefix, Nexthop}.

[6] The format is {List ID:VNI, Nexthop}

The tables of the LPM group and EM group are distributed across the Multiple Action Units (MAUs). The decision of how to deploy them depends on the scale of the rule sets. All packets will first enter the data plane through the Ingress of *pipeline 0*. If the memory resource can accommodate the rule set, all tables will be deployed in *pipeline 0*. The packets will then pass through the ingress of *pipeline 0*, get scheduled to the egress of *pipeline 0*, and proceed along the normal path as shown in Fig. 8. However, if the partition decision indicates that the memory requirements exceed the capacity of a single pipeline, the tables will be distributed across two pipelines. In this case, the packets will pass through all tables in the ingress of *pipeline 0* and then be scheduled to the egress of *pipeline 1*. To continue processing, the packets will be directed back to the ingress of *pipeline 1* using a recirculation port and finally scheduled to the egress of *pipeline 0*. This process is referred to as the folding path, as depicted in Fig. 8. By such a **pipeline folding** scheme [9], we can ensure that all tables are accessed.

4.2. Table implementation

In **MaP**, there are three kinds of tables to implement with hardware lookup tables. The LPM group is implemented as a TCAM-based *LPM Table* that works independently for the longest prefix matching of VNI-prepended IP prefixes. With the partition optimization, the EM groups are divided into multiple partitions, and each of them possesses a unique length of key. Every EM group requires a *Prefix Table (PT)* and a *Nexthop Table (NT)*, with the PT looked up first. EMs can be implemented using the tables in SRAM to perform exact matches. However, as Tables in an MAU are accessed in parallel, PT and NT of an EM group cannot be placed in the same MAU because of their *read and write* dependence [14]. To enable a more compact table placement, we could instead place the prefix table of an EM group with the nexthop table of another group whose prefix table already looked up in the same MAU, as depicted by MAU*i* in Fig. 8. It may seem that each MAU can only hold one pair of *PT* and *NP* in Fig. 8. However, this placement is solely to demonstrate the impact of table dependence. In practice, for tables that are independent of each other, a MAU can hold as many tables as it can accommodate. Additionally, during the EM group partition optimization, we do not aim for table size balance among different EM groups. Since there is no dependence among EM groups, they are always placed into the MAUs in a 'left-aligned' manner. Specifically, the P4 compiler starts by placing all *PT*s of different EM groups from the left-most available MAU, and then proceeds to place all *NT*s. After placing all EM groups, multiple consecutive MAUs on the left side of a pipeline are fully occupied, leaving the available MAUs on the right side to hold other virtual routing rules or rules of other functions.

Assuming that in the ith partition, the length of IP prefix are aligned to L_i and the number of the unique nexthop lists are N_{np}, the key lengths of the *PT* and *NT* are set to L_i and $24 + \log N_{np}$ respectively. By default, we implement the LPM table using the built-in *lpm* table, which is implemented with TCAMs. Since the VNI is usually of 24 bits, the length of the *key* in the *LPM table* is set to $32+24 = 56$ bits for IPv4 and $128+24 = 152$ bits for IPv6 respectively. We implement the *PT* and the *NT* table with the built-in *hash* tables, which are implemented with SRAMs and only support exact match. For a *PT* where all merged IP prefixes are expanded to be of L bits in length, the length of the *key* is set to L. Besides, in the *PT*, the nexthop-list ID associated with a prefix is stored as the *action data* with the key representing this prefix. The *key* of a nexthop table is a combination of the VNI and the nexthop-list ID. The width of the nexthop-list ID is determined by the number of distinct nexthop lists, the upper bound of which is the number of distinct prefixes in the original prefix table. The reason for this outcome is that, in the worst case scenario, each prefix is associated with a unique nexthop list.

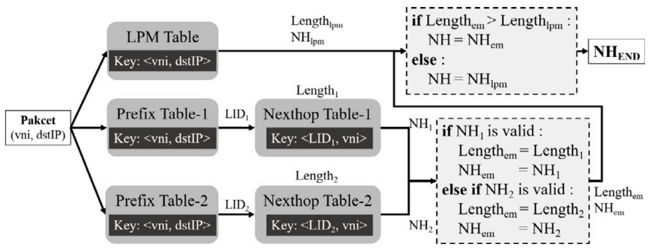


Fig. 9. An example of the lookup workflow of MaP.

4.3. Packet lookup

An Example of lookup is shown as the workflow in Fig. 9. For a given packet, its destination IP address and the VNI of the destination VPC are used to perform the virtual routing lookup. With **MaP**, the lookup of an IP address along with a VNI is processed in three steps. First, a key formed by prepending the VNI to the IP address is matched against the LPM table with a longest prefix matching. The result NH_{lpm} along with the length of the matched entry $Length_{lpm}$ are recorded as a candidate.

Then, the IP address along with the VNI is processed with all EM groups, following the decreasing order of the length of the prefix table in an EM group. In the example of Fig. 9, two EM groups are deployed. Within each group, the dstIP is used to match against the prefix table to get the nexthop list ID (i.e., *LID*), then the *LID* and the *vni* are used to match against the Nexthop table to get the nexthop ID (i.e., NH_{em}). The length of the prefix to match in an EM group is known at the compilation time and available for all MAUs instead of being stored in the entries, like the LPM group does. The NH_{em} with the longest *Length* will be set as the candidate generated by EM groups. Finally, We check the final NH_{END} between the candidates of the LPM group and EM group, which are NH_{lpm} and NH_{em} respectively, and the one with the longer *Length* will be set as the final lookup result.

One may notice that our design employs a two-phase selection process, where we first select the NH_{em} from the EM groups and then finalize the selection of the final NH_{END}. Ideally, the search through EM groups can stop once a destination is matched. This design is a compromise resulting from the limitations of $P4_{16}$ [17], which serves as the data plane language used by **MaP** but does not directly support the selection logic for multiple inputs. However, as the prefix length for each EM group is determined during compilation, the controller can organize the search of EM groups using a series of nested “if-else” statements and arrange them in descending order based on their prefix lengths within the generated P4 source file. This hard-coded approach ensures that once an EM group produces a valid result, subsequent EM groups with shorter lengths can be bypassed, potentially leading to the production of invalid results. After that, we can check the outputs of these EM groups one by one until we find the first valid one as shown in Fig. 9. This allows us to employ a standard two-element comparison logic to implement the selection process.

4.4. Rule update

All updates, including insertion, deletion, and modification, are translated into a batch of entry modifications for specific lookup tables in the data plane. The key challenge is determining the appropriate group. To improve memory efficiency in rule entry insertion, we prioritize selecting the EM group that manages the entry’s prefix length. If no entry space is available, we then consider the LPM group. To mitigate entry overflow, redundant entry space is reserved in each table of both LPM and EM groups. This approach minimizes the need for table size modifications, which require recompiling the P4 source file and restarting the pipelines. To delete or modify an existing rule entry, we first check the LPM group. If the rule entry is not found there, we proceed to the EM group managing the rule entry’s prefix length.

Extensive rule updates leading to table capacity overflow or suboptimal partition parameters necessitate reconfiguration, involving recalculating the cost matrix and rerunning the dynamic programming process. To alleviate the workload of the controller, a dedicated server enhances its performance by managing reconfiguration at the back-end while the controller handles updates at the front-end. Upon deciding to reconfigure, the programmable switch continues forwarding packets with old rules until it receives the latest group configuration sent from the dedicated server.

Rule updates cannot be performed by the programmable ASIC itself and must be managed by the controller on the equipped x86 CPU. Currently, the latest programmable switches can update approximately 4000 rules per second. However, when the redundant rule space is exhausted, that can only be delayed by reserve enough redundant space, EM group partitions must be recalculated offline before installing new rules. This partition calculation for a ruleset of about 20 million entries can take several minutes on a dedicated server. In the future, we plan to optimize update performance by improving the partition algorithm to reduce calculation time and enhancing the vendor-supplied rule installation API to speed up the process. However, our primary focus in this work is on minimizing memory costs for a given ruleset.

4.5. IPv4/IPv6 hybrid design

As the ratio of IPv4/IPv6 traffic in clouds is changing constantly, a hybrid design is recommended to pool the resources for IPv4 and IPv6 on the programmable chips [9]. Our LPM group management approach follows the same principle by setting the key length of the LPM table to $128 + 24 = 152$ bits. This reduces the memory efficiency, but the LPM table is only a small part of our approach. The design of EM groups naturally support the sharing of memory resources between IPv4 and IPv6 without any loss of memory efficiency. Because every EM group manages a range of prefix lengths and the table key is set to the longest length it manages, so IPv4 and IPv6 routing entries can be simply managed together.

4.6. System optimizations

We further leverage two simple optimizations to improve the system performance. The first is **Route Aggregation**. For a given VPC’s routing table, we construct a prefix trie containing all its entries. If a subtree exists where the root and all child nodes share the same nexthop, we only need to retain the rule for the root. This approach reduces the scale of rules without introducing any risk of lookup errors. The second is **Algorithmic LPM (ALPM)**. In our basic approach, the LPM table is implemented with TCAMs using the built-in **lpm** table. To improve the memory efficiency, we adopt the approach used in **Sailfish** [9] to implement the LPM group with the ALPM table provided by the programmable switch. In this way, the usage of TCAMs can be reduced at the expense of more usage of SRAMs. In addition, via the flexible table partition with our approach (see Section 3.5 and Section 3.6), the size of the LPM table can be controlled to achieve better memory efficiency.

5. Evaluation

5.1. Methodology

Data sets: The data set consists of 5 real-world VXLAN routing tables, which are referred to as *tb*1, *tb*2, *tb*3, *tb*4, and *tb*5 respectively. *tb*1 contains over 2M (i.e., Million) entries, whereas the other 4 tables each have over 200 K (i.e., Kilo) entries. However, due to the NDA of the data provider, we cannot disclose the exact distribution of prefix lengths. Besides, we synthesized hundreds of virtual routing tables consisting of 1M entries with 2 different rule characteristics. Firstly, we set 4 numbers of VPCs including 1 K, 10 K, 100 K, and 1M VPCs. In each case, $\frac{1M}{VPCs}$ random IP prefixes were generated to be uniformly distributed among the VPCs. Secondly, we set 4 rule distributions including "20%–80%", "30%–70%", "40%–60%", and "50%–50%". Here, "20%–80%" indicates that more than 80% of rule entries are evenly distributed among less than 20% of the VPCs, and . For each rule characteristic, 100 virtual routing tables consisting of random rule entries are generated.

Evaluated schemes: We used the implementation of **MaP** with two optimizations from Section 4.6 by default. Then, both **Sailfish** and the LPM group of **MaP** leverage the ALPM table to support the longest prefix match. For **Sailfish**, we select the most memory-efficient configuration, referred to as **Sailfish-m**, from numerous parameter combinations of the ALPM table. We found that the impact of the LPM group on memory usage for **MaP** is significantly smaller than that of the EM group. Therefore, we used a fixed configuration of the LPM group for all of our experiments. **MaP** comes with its own parameters specifying the number of EM groups and the size limitation of the LPM group respectively, and we evaluated it with several configurations to select the one with the lowest memory cost, referred to as **MaP-m**.

Performance metrics: Our focus is on hardware resource utilization, specifically memory and stage (i.e., MAU) usage. We will demonstrate that the increase of memory usage efficiency does not affect the throughput or latency. To assess resource utilization, memory usage is presented as percentages, in alignment with the device vendor's NDA. We generated all experimental results of resource usage by parsing the compiler's log files. It is crucial to highlight that some results may exceed 100%, indicating the successful task accomplishment of the compiler in placing the tables of the EM groups and the LPM group into specific MAUs of the pipeline. However, it also suggests that this configuration cannot be deployed on a single device.

Experimental results: If there were no compilation errors, the memory occupation, stage usage, and pipeline latency could be obtained by parsing the compiler's output. Thus, from Section 5.2 to Section 5.4, we can get the experimental results right after the compilation. In Section 5.5, both **MaP** and **Sailfish** are deployed within a Tofino 1.0 programmable switch connected to the packet generator server via a Mellanox ConnectX-5 100Gbps NIC. We utilized the Trex software packet generator[7] for measuring throughput in Gbps and packet latency in us.

5.2. Performance overview

In this experiment, we evaluated the performance of **Sailfish** and **MaP** using real-world VXLAN routing tables in IPv4.

In Section 3.5, we discussed how the EM group partition with minimized memory cost is derived by calculating $opt(N, P)$, where N represents the number of different prefixes' lengths, and P denotes the number of EM groups. In our device, the upper bound for P is 14. Afterward, as discussed in Section 3.6, we can further reduce the memory cost by moving some prefixes to LPM group. Nonetheless, it is crucial to consider that such a prefix moving operation could potentially increase TCAM usage. Consequently, the pipeline's stage resources might be depleted quickly since TCAM resources for each stage are limited. We address this concern by setting the upper bound for the number of stages that can accommodate LPM group rules to the number of stages available, which is 12 in the selected device. To identify the optimal configuration, we perform an exhaustive search of all combinations of partition numbers (i.e., the number of EM groups) and the number of stages capable of storing LPM group rules. We conduct our experiments on the virtual routing table $tb1$, and the corresponding results are depicted in Fig. 10.

[7] https://trex-tgn.cisco.com/

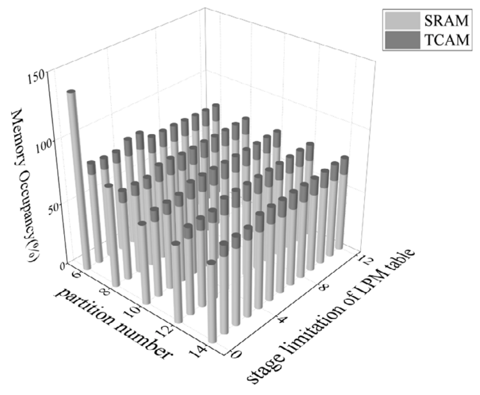


Fig. 10. Memory usage with different configurations.

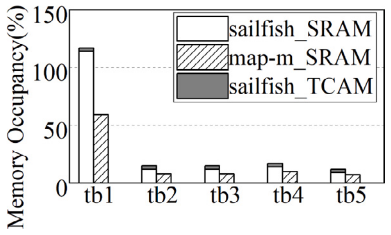


Fig. 11. Memory cost.

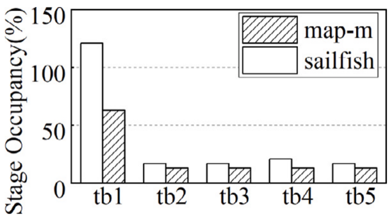


Fig. 12. Stage usage.

As the number of partitions increases from 6 to 14, the SRAM usage of **MaP** decreases first and then slightly increases. Accordingly, we can choose appropriate parameters, including the number of stages of the LPM table (denoted as N_{LPM}) and the number of EM group partitions (i.e., N_{EM}) to configure the **map-m**. For the largest table, $tb1$, we set $N_{LPM} = 1$ and $N_{EM} = 10$. For tables $tb2$ to $tb5$, due to their similar rule scale and shared IP similarity, we set $N_{LPM} = 1$ and $N_{EM} = 8$ for all of them.

Fig. 11 demonstrates clearly that, compared with **Sailfish-m**, **MaP-m** reduces the usage of SRAMs by 25.3% $\sim$ 47.7%. One may notice that **MaP-m** does not utilize TCAM. This is attributed to the Alpm primitive of the programmable switch, as mentioned in Section 4.6. In addition, **MaP** is able to hold the biggest table (i.e., tb1), with only 59.4% SRAMs occupied. However, **Sailfish** exceeds 100% memory consumption, making it unsuitable for deployment on the hardware.

As shown in Fig. 12, **MaP**'s superiority on memory efficiency also affects the usage of pipeline stages, reducing the average number of stages 25.0% $\sim$ 50.0% compared to **Sailfish**. However, due to several hardware limits, less memory cost does not always means less stage usage. For instance, **MaP** can hold the biggest routing table with one device, but it requires more stages than a single pipeline offers. As such, we adopt the "pipeline folding" technique [9] to use two pipelines together.

Fig. 13 presents the ratios of the pipeline latency in our approaches to that in **Sailfish-m**. The pipeline latency is calculated by summing all stages' latencies, which are obtained from the compiler's output.

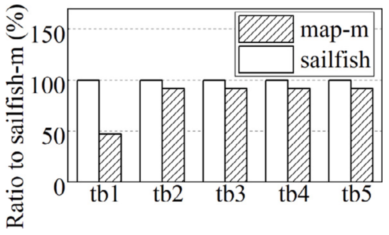


Fig. 13. Pipeline latency.

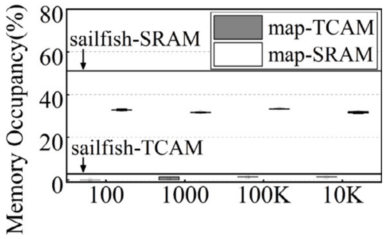


Fig. 14. Different scales of VPCs.

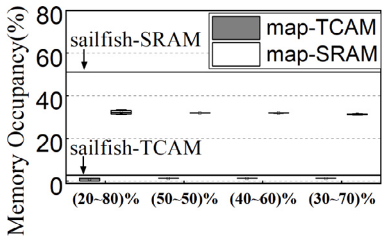


Fig. 15. Diverse distribution.

Thus, despite **Sailfish** being unable to deploy **tb1** onto a single device, a valid latency value can still be calculated. As depicted, **MaP-m** can reduce the pipeline latency by more than 50% compared with **Sailfish-m** in processing the biggest table. However, their superiority sharply decreases when processing smaller tables, because the additional latency brought in by virtual routing lookup against a small table is too low in comparison with the base pipeline latency.

5.3. Performance with diverse rule distribution

In this experiment, we evaluated the performance of the two schemes using all synthetic data sets with only IPv4 tables. Fig. 14 and Fig. 15 present the memory occupancies, including TCAMs and SRAMs, with different scales of VPC and different rule distributions respectively. **MaP** uses the memory-oriented configuration, via different data sets of different characteristics, each of which has 1 million routing entries. **Sailfish** shows the same performance because all rule sets have the number of routing entries.

As shown in Fig. 14, when the number of VPCs grows from 100 to 100 K while ensuring that less than 20% of the VPCs own more than 80% routing entries, the usage of TCAMs and SRAMs with **MaP** outperforms **Sailfish** by 50.0% ∼ 100.0% and 34.4% ∼ 39.3% respectively. Besides, in 18.6% cases, **MaP** does not use any TCAMs, while its usage of SRAMs in all cases varies by at most 7.5%. A similar trend can be observed via Fig. 15 when the number of VPCs is fixed at 100 K and the distribution of the entries' ownership varies. As depicted, for any given virtual routing table of a certain rule scale, as if the prefix similarity is significant, **MaP** has stable high memory efficiency and outperforms **Sailfish**.

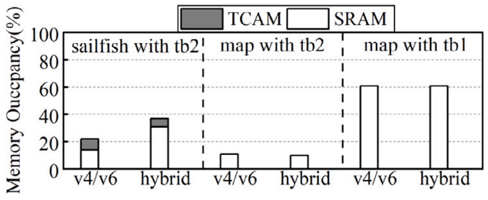


Fig. 16. Performance with a hybrid design of IPv4/v6.

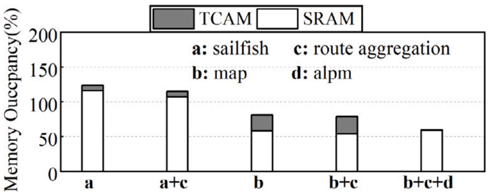


Fig. 17. Performance with different optimizations.

5.4. Performance with different optimizations

Fig. 16 presents how the performance is affected by pooling IPv4 and IPv6 tables. As depicted, pooling IPv4 and IPv6 tables decreases the memory efficiency of the two schemes in most cases. This occurs because the key length of IPv4 tables is aligned with that of IPv6 tables, increasing the memory usage. However, there are two exceptions. First, with **Sailfish**, the table pooling reduces the TCAM usage by 25.0% though the SRAM usage increases by 54.8%. Second, with **MaP**, the hybrid design for $tb2$ can reduce the SRAM usage by 9.1%. In this experiment, **MaP**'s superiority is reflected as the fact that the performance influence of pooling tables with **MaP** is obviously lighter than with **Sailfish**. Actually, **Sailfish** even fails in the compilation with a large table $tb1$.

Next, we evaluate the effects of two system optimizations. The routing table used in this experiment is $tb1$, and other tables show a similar optimization trend. However, in the trend of increasing the size of rule entries in the future, we believe the experimental results of $tb1$ are more instructive. Since **Sailfish** fails in the compilation with pooling tables, we select the version of separate IPv4 and IPv6 tables. As shown in Fig. 17, route aggregation is always useful in reducing the memory cost for both schemes because it compresses the original tables. Replacing the built-in LPM table with an ALPM table to implement the LPM group of **MaP** can sharply reduce the usage of TCAM, but it comes at the cost of a slight increase in the SRAM usage. From a global perspective, the most effective optimization strategy is to implement MaP with route aggregation and Alpm, as indicated by the right-most bar in Fig. 17, which reduces memory usage by 41.3% compared to Sailfish.

5.5. System performance

In this experiment, we use $tb1$ to assess large table deployment performance. Additionally, because $tb2$ to $tb5$ share similar rule scales and prefix similarities, only $tb2$ is used for small table deployment evaluation. Both **Sailfish** and **MaP** deploy $tb2$ individually. However, **Sailfish** cannot deploy $tb1$ onto a single device, so only **MaP** handles $tb1$ deployment. In $tb1$ deployment, **MaP** adopts a pipeline folding architecture (refer to Fig. 8).

As illustrated by the leftmost 2 bars of each group in Fig. 18, the performance of **MaP** processing $tb2$ aligns seamlessly with that of **Sailfish**. However, for the larger table $tb1$, where two pipelines are employed, the system throughput of **MaP** experiences a marginal decrease ranging from 1.0% to 6.9%, as illustrated by the rightmost bars of each group. Nevertheless, even under these conditions, **MaP** consistently forwards packets of 1024 bytes at line speed.

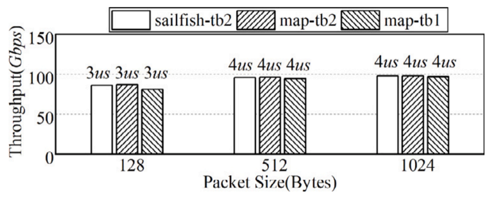


Fig. 18. System performance in forwarding packets.

6. Related work

The cloud gateway is a important component in a multi-tenant cloud environment. Previous designs have predominantly focused on software-based solutions [6–8], managing large-scale rules and traffic with the aid of device clusters and high-performance IO techniques. Jasper [18] focuses on implementing a near-simultaneous multicast service tailored for multiple cloud tenants. Amazon Web Services (AWS) Transit Gateway functions as a central hub, routing all traffic to and from each VPC in a single location, thereby eliminating the need to maintain complex peering relationships [19]. Sailfish [9] proposes the first design based on programmable switches, yet its limited rule space and memory manipulation capabilities pose challenges for deploying extensive virtual routing rule sets at scale.

Before the emergence of the programmable switch, several works attempt to improve the virtual routing lookup's memory efficiency with prefix aggregation. Luo et al. [12] propose a trie merging approach, using a merged trie and a table of next-hop-pointer arrays to represent multiple FIBs. Fu et al. [13] optimize the memory cost by maintaining a prefix trie and a shared nexthop table. Huang et al. [11] leverage an on-chip DAG to find the longest matching prefix by converting the trie forest to DAG. Prefix aggregating is also used to reduce the entry scale and the power consumption of a TCAM-based forwarding architecture [20–23]. However, some unsupported fundamental primitives make them impractical to the programmable switch. In addition to prefix aggregation, there are other solutions aiming to mitigate the memory overhead. Both Song [24] and Mi [25] try to design a concise prefix trie structure. However, in such solutions, data structures are designed to minimize storage overhead on other platforms, making it hard to deploy them on the programmable switch.

In addition to enhancing memory efficiency, some studies aim to broaden the programmable ASIC rule space using external memory. Kim et al. [26] advocate employing RDMA to extend the rule space of the programmable switch, followed by a detailed design proposal [27]. Nonetheless, this approach introduces additional time overhead for communication and maintenance overhead for external devices.

7. Conclusion

In this paper, we present the design of **MaP**, which adopts a novel **M**erge-and-**P**artition approach to compress the virtual routing tables and is verified to be **M**emory-efficient according to our performance evaluation with extensive experiments. In processing real world VXLAN routing tables with a programmable switch, **MaP** reduces the memory cost by 25% $\sim$ 47.7% in comparison to the state-of-the-art approach **Sailfish**, without any adverse impact on throughput and latency. The largest real-world table available to us contains over 2 million entries for nearly 1 million VPCs. **MaP** could deploy it into a single programmable switch, only consuming 59% of memory and 63% of stages. However, **Sailfish** requires two switches to handle the table. **MaP** takes advantage of the strong similarity of routing entries across different VPCs to largely compress the tables in the cloud gateway. This is a key point for table compression in the cloud gateway, which may lead to other optimizations with different system architectures.

CRediT authorship contribution statement

Ke Xu: Writing – original draft, Software, Methodology, Data curation, Conceptualization. **Donghong Jiang:** Writing – review & editing, Writing – original draft, Software, Methodology, Data curation, Conceptualization. **Yanbiao Li:** Writing – review & editing, Writing – original draft, Methodology, Data curation, Conceptualization. **Xin Wang:** Writing – review & editing, Validation, Methodology, Conceptualization. **Dafang Zhang:** Writing – review & editing, Validation, Methodology, Conceptualization. **Gaogang Xie:** Writing – review & editing, Validation, Supervision, Methodology, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The data that has been used is confidential.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful and constructive suggestions. This work was partially supported by the National Natural Science Foundation of China (Grant Nos. 62072430) and National Defense Science and technology Innovation key deployment project of Chinese Academy of Sciences (Grant Nos. JCPYJJ22019).

References

[1] K. Costello, M. Rimol, Gartner forecasts worldwide public cloud end-user spending to grow 23% in 2021, 2021, https://www.gartner.com/en/newsroom/press-releases/2021-04-21-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-grow-23-percent-i-2021. (Accessed 21 April 2022).

[2] T. Wood, P.J. Shenoy, A. Gerber, J.E. van der Merwe, K.K. Ramakrishnan, The case for enterprise-ready virtual private clouds, in: HotCloud, 2009.

[3] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, C. Wright, Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks, 2014, http://dx.doi.org/10.17487/RFC7348, RFC 7348, https://www.rfc-editor.org/info/rfc7348. (Accessed 21 April 2022).

[4] D. Kreutz, F.M. Ramos, P.E. Verissimo, C.E. Rothenberg, S. Azodolmolky, S. Uhlig, Software-defined networking: A comprehensive survey, Proc. IEEE 103 (1) (2014) 14–76.

[5] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, R. Boutaba, Network function virtualization: State-of-the-art and research challenges, IEEE Commun. Surv. Tutor. 18 (1) (2015) 236–262.

[6] K. Yap, M. Motiwala, J. Rahe, S. Padgett, M.J. Holliman, G. Baldus, M. Hines, T. Kim, A. Narayanan, A. Jain, V. Lin, C. Rice, B. Rogan, A. Singh, B. Tanaka, M. Verma, P. Sood, M.M.B. Tariq, M. Tierney, D. Trumic, V. Valancius, C. Ying, M. Kallahalla, B. Koley, A. Vahdat, Taking the edge off with espresso: Scale, reliability and programmability for global internet peering, in: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017, ACM, 2017, pp. 432–445, http://dx.doi.org/10.1145/3098822.3098854.

[7] H. Shao, X. Wang, Y. Lu, Y. Yu, S. Zheng, Y. Zhao, Accessing cloud with disaggregated Software-Defined router, in: 18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 21, USENIX Association, 2021, pp. 1–14, URL https://www.usenix.org/conference/nsdi21/presentation/shao.

[8] M.T. Arashloo, P. Shirshov, R. Gandhi, G. Lu, L. Yuan, J. Rexford, A scalable vpn gateway for multi-tenant cloud services, SIGCOMM Comput. Commun. Rev. 48 (1) (2018) 49–55, http://dx.doi.org/10.1145/3211852.3211860.

[9] T. Pan, N. Yu, C. Jia, J. Pi, L. Xu, Y. Qiao, Z. Li, K. Liu, J. Lu, J. Lu, E. Song, J. Zhang, T. Huang, S. Zhu, Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches, in: SIGCOMM '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 194–206, http://dx.doi.org/10.1145/3452296.3472889.

[10] R. Moskowitz, D. Karrenberg, Y. Rekhter, E. Lear, G.J. de Groot, Address allocation for private internets, 1996, http://dx.doi.org/10.17487/RFC1918, RFC 1918. URL https://www.rfc-editor.org/info/rfc1918.

[11] K. Huang, G. Xie, Y. Li, D. Zhang, Memory-efficient IP lookup using trie merging for scalable virtual routers, J. Netw. Comput. Appl. 51 (2015) 47–58, http://dx.doi.org/10.1016/j.jnca.2014.02.007.

[12] L. Luo, G. Xie, K. Salamatian, S. Uhlig, L. Mathy, Y. Xie, A trie merging approach with incremental updates for virtual routers, in: Proceedings of the IEEE INFOCOM 2013, Turin, Italy, April 14-19, 2013, IEEE, 2013, pp. 1222–1230, http://dx.doi.org/10.1109/INFCOM.2013.6566914.

[13] J. Fu, J. Rexford, Efficient ip-address lookup with a shared forwarding table for multiple virtual routers, in: Acm Conference on Emerging Network Experiment & Technology, 2008.

[14] Y. Li, J. Gao, E. Zhai, M. Liu, K. Liu, H.H. Liu, Cetus: Releasing P4 programmers from the chore of trial and error compiling, in: A. Phanishayee, V. Sekar (Eds.), 19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022, USENIX Association, 2022, pp. 371–385, URL https://www.usenix.org/conference/nsdi22/presentation/li-yifan.

[15] V. Rios, G. Varghese, Mashup: Scaling tcam-based IP lookup to larger databases by tiling trees, 2022, http://dx.doi.org/10.48550/arXiv.2204.09813, CoRR abs/2204.09813. arXiv:2204.09813.

[16] V. Srinivasan, G. Varghese, Fast address lookups using controlled prefix expansion, ACM Trans. Comput. Syst. (TOCS) 17 (1) (1999) 1–40.

[17] P4$_{16}$ language specification, 2023, https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html. (Accessed 27 July 2023).

[18] M. Haseeb, J. Geng, U. Butler, X. Hao, D. Duclos-Cavalcanti, A. Sivaraman, Jasper: Scalable and fair multicast for financial exchanges in the cloud, 2024, arXiv preprint arXiv:2402.09527.

[19] Aws transit gateway, 2024, https://aws.amazon.com/transit-gateway/. (Accessed 28 May 2024).

[20] V. Ravikumar, R.N. Mahapatra, L.N. Bhuyan, Easecam: An energy and storage efficient tcam-based router architecture for ip lookup, IEEE Trans. Comput. 54 (5) (2005) 521–533.

[21] W. Lu, S. Sahni, Low-power tcams for very large forwarding tables, IEEE/ACM Trans. Netw. 18 (3) (2009) 948–959.

[22] T. Mishra, S. Sahni, Petcam—A power efficient tcam architecture for forwarding tables, IEEE Trans. Comput. 61 (1) (2011) 3–17.

[23] J.-Y. Huang, P.-C. Wang, Tcam-based ip address lookup using longest suffix split, IEEE/ACM Trans. Netw. 26 (2) (2018) 976–989.

[24] H. Song, M. Kodialam, F. Hao, T. Lakshman, Building scalable virtual routers with trie braiding, in: 2010 Proceedings IEEE INFOCOM, IEEE, 2010, pp. 1–9.

[25] Z. Mi, T. Yang, J. Lu, H. Wu, Y. Wang, T. Pan, H. Song, B. Liu, Loop: Layer-based overlay and optimized polymerization for multiple virtual tables, in: 2013 21st IEEE International Conference on Network Protocols, ICNP, 2013, pp. 1–10, http://dx.doi.org/10.1109/ICNP.2013.6733611.

[26] D. Kim, Y. Zhu, C. Kim, J. Lee, S. Seshan, Generic external memory for switch data planes, in: Proceedings of the 17th ACM Workshop on Hot Topics in Networks, HotNets '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 1–7, http://dx.doi.org/10.1145/3286062.3286063.

[27] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, S. Seshan, Tea: Enabling state-intensive network functions on programmable switches, in: Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, 2020, pp. 90–106.

Ke Xu is a PreDoc student at the College of Computer Science and Electronic Engineering at Hunan University. He is currently pursuing a Ph.D in the field of computer networking. His main research interest is information-centric networking and programmable network. Ke Xu received his M.Sc. in computer science from Hunan University.

Donghong Jiang is a PreDoc student at the Computer Network Information Center at Chinese Academy of Science. He is currently pursuing a Ph.D in the field of computer networking. His main research interest is high performance packet process and IP lookup.

Yanbiao Li received the B.S. degree in mathematics and the Ph. D. degree in computer science from Hunan University in 2009 and 2016, respectively. He is currently an Associate Professor with the Computer Network Information Center (CNIC), Chinese Academy of Sciences (CAS), and the University of Chinese Academy of Sciences (UCAS). His research interests include networked systems, packet processing algorithms and routing security.

Xin Wang (Senior Member, IEEE) received the Ph.D. degree in electrical and computer engineering from Columbia University, New York, NY, USA. She is currently an Associate Professor with the Department of Electrical and Computer Engineering, The State University of New York, Stony Brook University, Stony Brook, NY, USA. Her research interests include algorithm and protocol design in wireless networks and communications, mobile and distributed computing, and networked sensing and detection. She is a member of ACM.

Dafang Zhang received the Ph.D. degree in application mathematics from Hunan University, Changsha, China, in 1997. He is currently a Professor with the College of Computer Science and Electronic Engineering, Hunan University. His current research interests include dependable systems/networks, network security, big data, and privacy.

Gaogang Xie received the Ph.D. degree in computer science from Hunan University in 2002. He is currently a Professor at Computer Network Information Center, Chinese Academy of Sciences, and the University of Chinese Academy of Sciences. His research interests include Internet architecture, packet processing and forwarding, and Internet measurement.