Multi-Stage Flow Table Caching: From Theory to Algorithm

Ying Wan[1], Haoyu Song[2], Tian Pan[3], Bin Liu[4], Yu Jia[1], Ling Qian[1]
[1] China Mobile (Suzhou) Software Technology Co., Ltd, China [2] Futurewei Technologies, USA
[3] Beijing University of Posts and Telecommunications, China [4] Tsinghua University, China
{wanying-cx, qianling, jiayu}@cmss.chinamobile.com

2023 IEEE Symposium on Computers and Communications (ISCC) | 979-8-3503-0048-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/ISCC58397.2023.10217871

***Abstract*—Flow table capacity in programmable switches is constrained due to the limited on-chip hardware resource. The current mainstream approach is to cache only the popular rules in hardware. By taking advantage of traffic locality, the majority of packets can be forwarded directly after matching the rules cached in hardware and the remaining missed packets are handled by software that accommodates the full flow table. Existing works focus on selecting the cache entries for a single-stage flow table to achieve a high cache hit-rate, which cannot adapt to multi-stage flow tables. Due to hardware constraints as well as service requirements, it is often necessary to decompose a single-stage flow table to a multi-stage flow table or directly create multiple stages of tables in hardware. For the first time, we abstract and model the multi-stage flow table caching problem and prove the NP-hardness of the Optimal Multi-stage Flow table Caching (OMFC). Further, we propose a Greedy Caching Algorithm (GCA) for OMFC, which considers both the rule popularity across multiple stages of flow tables and entry popularity within the same stage of flow table when determining the content and size of the multi-stage flow tables. The simulation results show that GCA achieves a 10∼30% higher cache hit-rate than the existing algorithms.**

I. Introduction

Software-Defined Networking (SDN) [1] endows networks with a high level of flexibility and programmability by the separation of data plane and control plane. The data plane is responsible for processing and forwarding packets based on the rules in flow tables installed by the control plane. As an indispensable component of SDN routers and switches, flow tables play the roles beyond the conventional Access Control List (ACL) and Firewall. For example, cloud providers use flow tables for communication between different Virtual Private Clouds (VPCs) or between VPC and the enterprise's local data center [2], [3], Network Telemetry uses a flow table to filter specific packets for traffic measurement [4], and Load Balancing uses a flow table to classify the packets [3].

With the fast growth of network throughput, now in the magnitude of terabits per second per device, the flow table matching process needs to be fast enough to sustain the line-speed packet processing, entailing the use of hardware (*e.g.*, programmable ASIC, FPGA) to implement flow tables. At present, the mainstream programmable ASICs include Intel Tofino [5], Broadcom Trident [6], and Cisco SiliconOne [7], all of which use on-chip TCAM and SRAM for flow tables.

The work of Ying Wan, Yu Jia, and Ling Qian was supported in part by the NSFC under Grant U21B2022, and in part by the research project (NO. R23100TM) of China Mobile Communications Group Co., Ltd. The work of Bin Liu was supported in part by the NSFC under Grant 62032013, and Grant 62272258, and in part by the NSFC-RGC under Grant 62061160489. Corresponding Author: Ling Qian (qianling@cmss.chinamobile.com).

VPC	VM IP	VPC	NH
VNI=2	10.0.3.1	VNI=5	192.168.1.2
VNI=2	10.0.3.2	VNI=5	192.168.1.7
VNI=4	10.0.3.1	VNI=5	192.168.1.2
VNI=4	10.0.3.2	VNI=5	192.168.1.7
VNI=6	10.0.3.1	VNI=5	192.168.1.2
VNI=6	10.0.3.2	VNI=5	192.168.1.7

(a) 1-stage flow table

VPC	VM IP	VPC
VNI=2	10.0.3.0/24	VNI=5
VNI=4	10.0.3.0/24	VNI=5
VNI=6	10.0.3.0/24	VNI=5

VPC	VM IP	NH
VNI=5	10.0.3.1	192.168.1.2
VNI=5	10.0.3.2	192.168.1.7

(b) 2-stage flow table

Fig. 1. Flow table for VPC-to-VPC communication in PGW.

However, the limited on-chip hardware resource (*e.g.*, SRAM and TCAM) usually cannot hold all the rules of the flow tables. At the upper end, the hardware-based flow table can afford around ten thousand rules [8], while the application may require a flow table to support millions or even tens of millions of rules. For example, in China Mobile eCloud (CMCloud), VPCs can hold up to 100K Virtual Machines (VMs) and a VPC needs to communicate with up to 128 other VPCs. Therefore, to support the VPC-to-VPC communication, as shown in Fig. 1(a), the cloud Peering GateWay (PGW) needs to hold up to 128×100K rules in the form of "<(source)VNI,VM_IP>→<(destination)VNI,NH>", where VNI uniquely identifies the VPC, VM_IP indicates the IP address of the destination VM (*i.e.*, Overlay), and NH represents the next hop to the destination VM (*i.e.*, Underlay). In other cases, network devices may need to accommodate multiple services in a hyper-converged manner to save hardware costs and deployment footprints, where multiple stages of tables representing different services are installed into the same network device, occupying different pipeline stages [2].

With the imminent end of Moore's Law [9], the limited flow table capacity on a single chip can only be compensated by the better use of available hardware resource through flow table compression and caching. In the past two decades, many algorithms (*e.g.*, TCAM Razor [10]) have been proposed to reduce the size of flow tables while ensuring semantic equivalence. These algorithms require complex pre-processing on the original flow table, making the flow table update

process complicated and slow after compression. Moreover, such methods can only alleviate but cannot eliminate the gap between the hardware resource and flow table size.

In recent years, flow table caching becomes more popular. The key idea is to use the limited hardware as a high-speed cache on the fast path to hold only a small subset of flow table rules. Those packets that cannot find a matching rule in hardware are punted to the slow path (*e.g.*, Open vSwitch (OVS) [11] in the control CPU) for further processing. The rationale of this approach comes from the observation that real-world traffic presents a strongly skewed distribution [12], *i.e.*, the popular rules matched by the majority packets at any time only account for a small subset of the flow table. If we keep only the popular rules in hardware, a much larger rule table can be supported without compromising the throughput.

Existing flow table caching schemes mainly focus on the methods to pick the caching rules from one flow table to achieve the high cache hit-rate under the restriction of rule dependencies [8]. Such methods require that a popular rule r to be cached along with the rules overlapping with r to avoid mismatch, regardless of the popularity of those overlapping rules. In fact, we analyzed four types of CMCloud gateways implemented with programmable ASICs and found that the rule dependency exists in less than 10% of the flow tables. On the one hand, most of the flow tables are Exact Matching (EM) which ensures that rule dependency cannot occur. On the other hand, for the flow table that has the Longest Prefix Matching (LPM) or range matching (RM) fields, there is usually an EM field that prevents rules from overlapping (*e.g.*, VNI).

A single flow table may be decomposed into a multi-stage flow table. These flow tables form a logical table chain. The match in the first i stages leads to the search in the $(i{+}1)$-stage flow table. Only when all the stages are matched, a rule is matched. If a flow is processed directly in hardware, it means the k entries of the rule the flow matches are all cached. There are three reasons for using the multi-stage flow tables (explained in Section II): (1) reducing flow table size, (2) simplifying flow table update, and (3) adapting to hardware limitations. For multi-stage flow table caching, we need to determine the cache content for each stage of the flow table. If each stage is considered independently (*i.e.*, cache the top matched partial rules in each stage of the flow table), the cache hit-rate is not optimal. Interestingly, caching only the decomposing results of the top-matched rules leads to sub-optimal results too. In fact, the cache content for the multi-stage flow table should be considered jointly, since a caching entry can be shared by multiple rules.

To the best of our knowledge, we are the first to analyze the multi-stage flow table caching problem and provide an efficient solution. We not only prove the NP-hardness of the problem of the Optimal Multi-stage Flow table Caching (OMFC), but also propose a greedy algorithm to optimize the hit-rate for multi-stage flow table caching. Our contributions are as follows.

- We introduce the concept of Entry Sharing Graph (ESG) to abstract the multi-stage flow table, which reflects the popularity of entries and the combinations of entries.
- We prove the NP-hardness of OMFC by reduction from a well-known problem proved to be NP-hard, indicating the impossibility of finding the optimal solution for multi-stage flow table caching in polynomial time.
- We propose a Greedy Caching Algorithm (GCA) for the sub-optimal solution for the problem of OMFC. GCA can judge whether an entry is worth caching by considering both rule popularity and entry popularity. GCA processes all stages of the flow table jointly and determines the cache size and content for each stage simultaneously. The simulation results show that GCA achieves a 10$\sim$30% higher cache hit-rate than the state-of-the-art solutions.

The remainder of the paper is organized as follows. Section II provides the background. Section III discusses the related works. Section IV analyzes the problem of OMFC in theory. Section V describes the approximation algorithm GCA for OMFC. Section VI presents the implementation and evaluation. Finally, Section VII concludes the work.

II. Background

In this section, we first introduce the basis of flow table caching. Then we introduce the concept and motivation of decomposing a single flow table into a multi-stage flow table.

A. Flow Table Caching

As the gap between hardware resource and flow table size becomes larger, flow table caching becomes increasingly important to solve the problem of high-speed matching on large flow tables, thanks to the strong temporal locality and spatial locality of the network traffic.

The temporal locality is reflected in that the traffic of a few elephant flows dominates the total traffic in a relatively short period of time. The spatial locality is reflected in that the traffic does not evenly match every flow table rule, and most traffic matches only a few flow table rules. As reported in [13], the accumulated traffic of the top 1% of flows accounts for more than 70% of the total traffic for a one-minute trace; meanwhile, more than 100K flows match only 1K route prefixes in a table with 760K prefixes.

A flow table cache has a different refresh pattern from a classic cache in which an immediate cache replacement is done when a cache miss happens. Due to the slow and complex update process of the hardware-based (especially TCAM) flow table [14], flow table caching generally determines caching rules based on the rule matching results over a period of time (*e.g.*, rule's popularity [15]) as well as the update cost [16].

B. Flow Table Decomposing

There are three reasons for producing multi-stage flow tables: (1) reducing flow table size, (2) simplifying flow table update, and (3) adapting to hardware limitations.

A single-stage flow table uses the key covering all the matching fields (*e.g.*, microflow in the OVS and flow table in OpenFlow 1.0 [17]). However, when more and more fields are involved in a flow table, the key size becomes too wide to fit in existing hardware, and the table size explodes due

to the cross-product effect of the rules. Decomposing the single-stage flow table into a multi-stage flow table can greatly reduce the flow table size, because not only each stage has a narrower key but also the overall number of entries in all the stages is much smaller than the original table size [18]. Take Fig. 1(b) as an example, when implementing PGW, CMCloud first uses a small LPM flow table in the form of "<VNI, VM_IP>→ <VNI>" to find the VPC where the destination VM is located, and then uses the second stage flow table in the form of "<VNI, VM_IP> → <NH>" to find the next hop information. In this way, the size of the second stage flow table is reduced from 128×100K to 100K, since each entry of the second stage flow table is shared by up to 128 entries of the first stage flow table.

The multi-stage flow table also simplifies the update process when the rules change. As shown in Fig. 1, when the VM with the IP address 10.0.3.1 migrates and the corresponding NH changes from 192.168.1.2 to 192.168.1.4, the single-stage flow table in Fig. 1(a) needs to modify 3 rules for routing to this VM, while the two-stage flow table in Fig. 1(b) only needs to modify a single entry in the second stage flow table.

In some cases, flow table decomposition is required to match hardware limitations. For example, the number of operations (*e.g.*, modify packet headers and write/read registers) that can be executed after a flow table match (*i.e.*, stage) is limited [5], so it is sometimes necessary to convert a single-stage flow table which covers many matching fields and executes many actions into a multi-stage flow table, so that each stage only covers a subset of fields and performs limited operations. As shown in Fig. 1(b), modifying the packet header VNI and outer destination IP address recorded in NH are separately executed in the first and the second flow table. Of course, flow table decomposing may not achieve positive returns in every circumstance and its benefit is subject to the decomposing strategy and the flow table features such as key size and width.

III. Related Work

In recent years, with the large-scale deployment of SDN in cloud networks, the size of flow tables has far exceeded what the hardware resource can support. Some works [10], [19] tried to narrow the gap between flow table size and hardware capacity by compressing the flow tables. These schemes reduce the average hardware resource needed for storing a flow table rule or hash long matching keys into shorter ones, but fail to solve the problem fundamentally.

More and more researchers make use of capacity-limited hardware as a high-speed cache of a large flow table. Some works are about caching rules of EM flow tables, and most of the works are devoted to guaranteeing rule dependency and achieving a high cache hit-rate when caching LPM or RM flow tables [20]. CacheFlow [8] applies two methods, Cover-Set and Dependent-Set, to select the caching rules according to the popularity of the rules. Based on the idea of CacheFlow, some works [15], [16] design more complex algorithms to further improve the cache hit-rate or exploit the trade-off between cache hit-rate and cache refresh cost.

The above works all aim at caching for a single-stage flow table and do not adapt to a multi-stage flow table well. For a multi-stage flow table, high cache hit-rate at each stage does not necessarily imply high cache hit-rate of the overall scheme. PipeCache [21] is the only work for multi-stage flow table caching. It simply disperses the caching rules into multiple stages of flow tables in order to reduce hardware resource consumption. PipeCache inherits the method of CacheFlow to maintain rule dependency while evicting and caching rules. However, PipeCache evicts old rules and issues new rules every time cache misses occur and does not make use of rule popularity or entry popularity to select the caching content, resulting in high update frequency. Moreover, PipeCache does not consider the optimal assignment of the limited hardware resource to a multi-stage flow table.

IV. Optimal Multi-Stage Flow Table Caching

In this section, we first abstract and model the multi-stage flow table caching problem, then consider the factors when designing caching algorithms, and finally prove the optimal caching of a multi-stage flow table is an NP-hard problem.

A. Problem Statement

A flow table F comprises a set of entries, and each entry e is represented by three attributes: priority, field specification, and action. If a packet p matches the field specification of one or more entries of F, p should execute the action of the matched entry with the highest priority.

F can be decomposed into k stages of sub flow tables, $F_0, ..., F_{k-1}$, which consist of $n_0, ..., n_{k-1}$ entries, respectively. A rule r is the combination of one entry from each stage (*i.e.*, the i-th part of r, $r[i]$, is from F_i for $0 \leq i < k$). The total number of rules n is at most $\prod_{i=0}^{k-1} n_i$, and an entry in each stage can be shared by multiple rules.

When caching a rule r in hardware (*e.g.*, TCAM and SRAM), the k entries of r must be separately cached into the corresponding k sub flow tables in hardware. When looking up the k tables sequentially, if p matches no entry in the i-th flow table, a cache miss happens. The subsequent matching process is interrupted and p is sent to the slow path for the complete table lookup. The problem of the optimal multi-stage flow table caching can be described as follows.

OMFC: for n rules which are the combinations of entries of the k-stage flow tables $F_0, ..., F_{k-1}$, given the number of matched packets of each rule during a period of time and the hardware resource m, identify the entries to cache for each flow table so as to maximize the number of packets that match a rule in hardware, *i.e.*, achieve the highest cache hit-rate.

B. Theoretical Analysis

A straightforward solution is to cache the entries of the rules that match the most packets. In such a case, the content of every flow table can be specified together. The resource that should be allocated to each stage of the flow table can also be inferred from the statistics of the popular rules. However, this approach does not take into account the fact that an entry

may be shared by multiple rules and the entries of the most popular rules are not necessarily the most popular entries.

Another solution exists along this line of thinking. Instead of selecting entries according to the rule popularity, we can process the sub flow tables separately and select the most popular entries of each sub flow table to cache. However, the popular entries of different sub flow tables do not necessarily correspond to the same set of rules. Since a miss in any sub flow table means a cache miss, such a method cannot guarantee the best result. Moreover, this method needs to statically specify the resource allocated to each stage of the flow table, which may further deviate from the optimal solution.

C. NP-hardness Proof

Through the above discussion we can see that for the multi-stage flow table caching, the popularity of both rules and entries should be considered. In fact, the problem of OMFC is NP-hard and cannot be solved in polynomial time.

Theorem 1. The problem of OMFC is NP-hard.

Proof. The NP-hardness of the problem of OMFC can be proved by a reduction from an equivalent NP-hard problem, the densest k-subgraph problem on the bipartite graph (DkS) [22]. The problem of DkS is formulated as follows.

DkS: Given an undirected graph composed of two disjoint sets U and V and no two vertices within the same set are adjacent, to find the subgraph with exactly k vertices such that the number of edges in the subgraph is maximal.

For a given instance of DkS, we construct an instance for OMFC in the following manner in polynomial time. Each vertex $u \in U$ or $v \in V$ is mapped to a unique entry that consumes one unit of hardware resource. Thus, U and V correspond to the first and the second stage flow table, respectively; each edge between U and V corresponds to a rule consisting of the two entries corresponding to its two endpoints, the aim of finding the subgraph with exactly k vertices such that the number of edges in the unweighted subgraph is maximal is then transformed to find the k entries such that the hit-rate after caching them is maximal, in which all rules forward the same number of packets.

If the problem of OMFC can be solved in polynomial time, *i.e.*, we can find the k entries leading to the maximized cache hit rate in polynomial time, since the rules' popularity is the same and each rule corresponds to an edge, we can find the densest k-subgraph and the problem of DkS is solved in polynomial time, contradicting with the NP-hardness of DkS. Therefore, the problem of OMFC cannot be solved in polynomial time and its NP-hardness is proved. ■

V. Approximation Algorithm For OMFC

In this section, we first introduce the concept of the Entry Sharing Graph (ESG). Then, we describe how to use the ESG to estimate the hardware resource allocated for each flow table. After that, we show how to calculate the rule cache profit while taking the entry popularity into account. Finally, we explain how to specify the entries to cache.

TABLE I
A TYPICAL 3-STAGE FLOW TABLE

Rule	Specification			Counter	Profit
	F1	F2	F3		
r_1	e_1^1	e_1^2	e_2^3	7	$7+\frac{2}{8}$
r_2	e_2^1	e_3^2	e_1^3	7	$7+\frac{13}{12}+\frac{8}{6}+\frac{5}{8}$
r_3	e_2^1	e_3^2	e_3^3	6	$6+\frac{14}{12}+\frac{9}{6}+\frac{4}{8}$
r_4	e_2^1	e_2^2	e_3^3	4	$4+\frac{16}{12}+\frac{6}{8}$
r_5	e_2^1	e_4^2	e_1^3	3	$3+\frac{17}{12}+\frac{9}{8}$
r_6	e_1^1	e_3^2	e_1^3	2	$2+\frac{7}{12}+\frac{13}{6}+\frac{10}{8}$

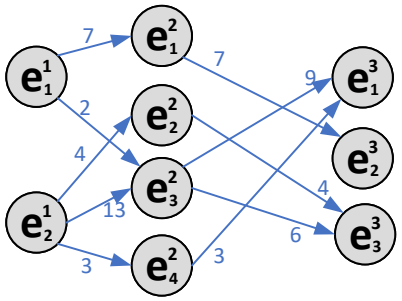

Fig. 2. The ESG of the flow table in Table I.

A. Construction of ESG

For a k-stage flow table, we map it into the ESG in the following manner. We abstract each flow table entry into a unique graph vertex and add k-1 directed edges between the k vertices corresponding to each rule, where the edge always points from the vertex corresponding to the i-stage flow table pointing to the (i+1)-stage flow table. Meanwhile, the entry/vertex counter value is equal to the sum of the counter values of the rules passing through it.

Fig. 2 shows the ESG of the flow table in Table I. As shown in Fig. 2, the entry e_1^3 is shared by the rules r_2, r_5, and r_6 whose counter values are 7, 3, and 2, respectively, the counter value of e_1^3 is therefore 12.

B. Cache Profit Calculation

When caching a rule r, we should not only focus on the counter of the r itself, which indicates the least contribution to the cache hit-rate; but also pay attention to the counter values of the k entries of r.

Assume $r.cnt$ and $r[i].cnt$ represent the counter values of r and its i-th entry $r[i]$, the higher the value of $r[i].cnt$ - $r.cnt$, the higher the concomitant profit of caching the entry $r[i]$. Therefore, we define r's cache profit $r.pf$ as follows.

$$r.pf = r.cnt + \sum_{i=0}^{k-1} \frac{r[i].cnt - r.cnt}{\prod_{j=0}^{i-1} size(F_j) \prod_{j=i+1}^{k-1} size(F_j)} \quad (1)$$

Where $size(F_j)$ represents the number of entries in the j-th stage flow table. $c = \prod_{j=0}^{i-1} size(F_j) \prod_{j=i+1}^{k-1} size(F_j)$ refers to the maximum number of rules when the i-th entry $r[i]$ is determined. $\frac{r[i].cnt-r.cnt}{c}$ is the average profit of caching $r[i]$. In fact, $\frac{1}{c}$ can be regarded as the effective coefficient of the

Algorithm 1: Construct Entry Sharing Graph

```
Function ConstructESG(ft, k)
    G = (V, E), n_r = ft.rules.size()
    for (i = 0; i < n_r; i = i + 1) do
        r = ft.rules[i]
        for (j = 0; j < k; j = j + 1) do
            e = r[j]
            if e in V then
                V(e).cnt = V(e).cnt + r.cnt
            else
                V.add(e), V(e).cnt = r.cnt
        for (j = 0; j < k; j = j + 1) do
            e_j = r[j], e_{j+1} = r[j + 1]
            if V(e_j) → V(e_{j+1}) ∉ E then
                E.add(V[e_j] → V[e_{j+1}])
    return G
```

Algorithm 2: Calculate Cache Profit

```
Function CalculateCP(ft, k, G)
    n_r = ft.rules.size()
    for (i = 0; i < n_r; i = i + 1) do
        r = ft.rules[i], r.pf = r.cnt
        for (j = 0; j < k; j = j + 1) do
            e = r[j]
            r.pf = r.pf + (V(e).cnt − r.cnt) / (∏_{q=0}^{j−1} size(F_q) ∏_{q=j+1}^{k−1} size(F_q))
    return ft
```

profit of the entry, because caching $r[i]$ cannot guarantee that all rules containing it will be cached together.

Table I shows the cache profit of the rules $r_1 \sim r_6$. We take r_5 as an example to illustrate the calculation process of $r_5.pf$. First, $r_5.pf$ should be added with its counter value $r_5.cnt = 3$, which indicates the minimum number of packets that match the hardware flow table after r_5 is cached. Then we pay attention to the entry sharing profit. For r_5's first entry e_2^1, since $e_2^1.cnt = 20 > r_5.cnt$, meaning r_5 shares e_2^1 with other rules, therefore, we add the concomitant profit of caching the entry e_2^1, which should be $\frac{20-3}{3\times4}$ according to Equation 1. The concomitant profit of e_1^3 when caching r_5 is $\frac{12-3}{2\times4}$ by using the same method. As for r_5's second entry e_4^2, since $e_4^2.cnt = r_5.cnt$, which means r_5 does not share e_4^2 with any other rules, we do not need to calculate its cache profit.

C. Greedy Entry Selection

After the cache profit of each rule is calculated, the next step is to specify the cache content of each flow table. We apply the following greedy idea to select the rules to be cached under the constraints of total resource: we always select the most profitable rule that the hardware flow table can accommodate among the uncached rules. When identifying a rule r to cache, we see if its every entry has been cached. If and only if the i-th entry $r[i]$ has not been cached, we assign $F_i.width$ hardware resource to the i-th flow table, where $F_i.width$ represents the hardware cost of a single entry of the i-th flow table.

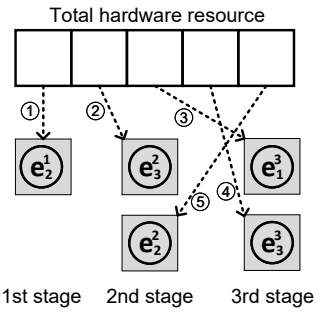

Fig. 3. The process of greedy entry selection.

We take the example in Table I to illustrate the entry selection process. Assume that the total hardware resource is 5 and each entry of the three flow tables consumes 1 hardware resource, Fig. 3 shows the content of every flow table after the entry selection process.

Algorithm 3: Greedy Entry Selection

```
Function GreedyES(rules, G, m)
    rules.sort(>, r.pf)
    n_r = rules.size()
    for (i = 0; i < n_r; i = i + 1) do
        curCost = 0
        for (j = 1; j < k; j = j + 1) do
            e = rules[i][j]
            if V(e).cached == False then
                curCost = curCost + F_j.width
        if curCost <= m then
            for (j = 1; j < k; j = j + 1) do
                e = rules[i][j]
                if V(e).cached == False then
                    F_j.addEntry(e)
                    V(e).cached = True
```

First, the rules in Table I are sorted in the decreasing order of their cache profit and the results are $< r_2, r_3, r_1, r_4, r_6, r_5 >$. Second, we process the first rule r_2. Since none of its three entries are cached, caching r_2 requires 3 hardware resources for $\{e_2^1, e_3^2, e_1^3\}$ and the available hardware resource is 2 after that. Third, we process the next rule r_3. Since the entries $\{e_2^1, e_3^2\}$ of r_3 have been cached when caching r_2, caching r_3 requires only 1 hardware resource for e_3^3 and the left hardware resource is 1 after that. Forth, we process the rule r_1. Since all entries $\{e_1^1, e_1^2, e_2^3\}$ of r_1 are not cached, caching r_2 requires 3 hardware resources but the available hardware resource is only 1, therefore, we do not cache r_1. Fifth, we process the rule r_4. Since only the entry e_2^2 of r_4 is not cached, caching r_4 requires 1 hardware resource and all the hardware resource is consumed.

Through the above selection process, not only the size of each flow table is determined, but also the contents of the three flow tables $<F_1, F_2, F_3>$ are specified, which stores $\{e_2^1\}$, $\{e_3^2, e_2^2\}$, and $\{e_1^3, e_3^3\}$, respectively. According to Table I, we can see that the cache hit-rate is $\frac{17}{29}$. If selecting the rules to cache only by their counter values, as shown in

Table I, $<F_1, F_2, F_3>$ will store $\{e_1^1\}$, $\{e_1^2, e_3^2\}$, and $\{e_2^3, e_1^3\}$, respectively, and the corresponding cache hit-rate is only $\frac{9}{29}$. If selecting the entry to cache only by their counter values, as shown in Table I, $<F_1, F_2, F_3>$ will store $\{e_2^1, e_1^1\}$, $\{e_3^2\}$, and $\{e_1^3, e_3^3\}$, respectively, and the corresponding cache hit-rate is only $\frac{15}{29}$.

Note that although the above method is applicable when the total resource is given and the resource assigned to each flow table is undetermined, it can still be used when the size of each flow table is given. When judging whether a rule r can be cached, it is necessary to determine whether each flow table has enough space to store r's corresponding entry if it is not cached until now.

VI. Implementation and Evaluation

A. Experiment Setup

We choose CacheFlow as the representative for the single stage flow table caching algorithms. Specifically, CacheFlow caches rules in a single-stage flow table and does not make full use of the entry sharing circumstances between different rules. Since there is no rule dependency in our experimental setting, CacheFlow chooses the rules to cache only according to their counter values. We also choose PipeCache, the only existing work for multi-stage flow table caching, to make the comparison. Note that PipeCache evicts old rules and caches new rules every time cache misses occur, resulting in unacceptable update frequency. PipeCache is modified to adopt the update strategy of CacheFlow in each stage of the flow table. All the algorithms are implemented in C++ and compiled by g++ with -O2 optimization. We run them on a commodity server with the Ubuntu 16.04-LTS operating system.

B. Dataset

It is difficult to access real-world multi-stage flow tables and the corresponding traffics due to security and privacy concerns, therefore, as a convention, we resort to ClassBench-ng [23] to generate synthetic ones bearing the characteristics matching the real-world datasets. However, ClassBench-ng only generates rules and traffics for a single-stage flow table, which cannot be directly used to test the multi-stage flow table caching algorithm. On the other hand, the exact method the flow table is decomposed (*e.g.*, how many stages and which fields are included in each stage) is related to the flow table properties (*e.g.*, rule popularity and entry sharing rate) and the type of hardware. Therefore, given the expected number of flow table rules n_r, the number of the stages of flow tables k, and the number of entries of each stage of flow table $n_0 \sim n_{k-1}$, we first use ClassBench-ng to generate the flow tables, then we use it again to generate the corresponding traffics, next we calculate the counter values of the n_r rules, finally, we randomly choose n_r combinations from the entries of the k-stage flow tables as n_r rules. In this way, the contained entries and the counter values of each rule can be specified.

The datasets used for algorithm comparison are summarized in Table II. Unless otherwise mentioned, the number of the stages of flow tables is set to 3, the number of rules is set to 40K and the hardware resource can hold 2% of the total rules, the number of entries per flow table is set to 40K×0.01, 40K×0.02, and 40K×0.03, respectively. To test the algorithms' scalability and their sensitivity to different parameters, we vary the rule table type, flow table size, and cache size, respectively.

TABLE II
Flow Tables and Traffics used for performance evaluation

Type	Source	Rule #	Stage #	Packet #
ACL	ClassBench-ng	∼40K	$3 \sim 6$	7.6×10^5
Firewall	ClassBench-ng	∼40K	$3 \sim 6$	5.0×10^5
IP Chain	ClassBench-ng	∼40K	$3 \sim 6$	2.1×10^6
Openflow	ClassBench-ng	∼40K	$3 \sim 6$	1.2×10^6

C. Experimental Results

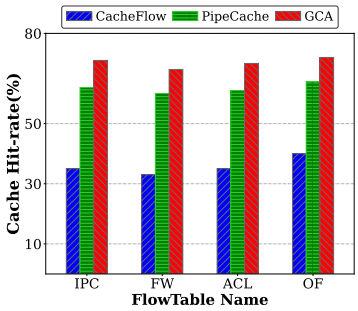

Fig. 4. The cache hit-rate on different flow tables.

Cache hit-rate. Fig. 4 shows the cache hit-rate of the algorithms on different types of flow tables. As shown in Fig. 4, on the four types of flow tables, CacheFlow achieves a much lower cache hit-rate than PipeCache and GCA, around 27% and 35% lower than PipeCache and GCA, respectively. This is because CacheFlow caches a rule in a single flow table while PipeCache and GCA cache the three entries of a rule to the corresponding stage of flow tables, in which an entry can be shared by multiple rules. Meanwhile, Fig. 4 shows that GCA achieves about a 10% higher cache hit-rate than PipeCache. On the one hand, PipeCache does not propose any strategy to select the entries to cache and directly caches the entries of popular rules which are selected by CacheFlow. GCA considers not only the popularity of rules but also the popularity of entries when specifying the cache contents. On the other hand, GCA can dynamically assign the hardware resource to the different stages of flow tables, which are the same and proportional to the size of software flow tables for CacheFlow and PipeCache, respectively.

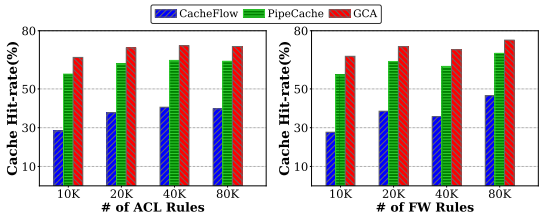

Fig. 5. The cache hit-rate under different flow table sizes.

Scalability on the number of rules. Fig. 5 shows the cache hit-rate of the algorithms under different number of rules. Note that although the number of rules n_r ranges from 10K to 80K, the hardware is always set to hold $n_r \times 1.5\%$ rules. As shown in Fig. 5, on the ACL and FW flow tables, GCA consistently beats CacheFlow and PipeCache for any number of rules, which are about 30% and 10% improvement, respectively.

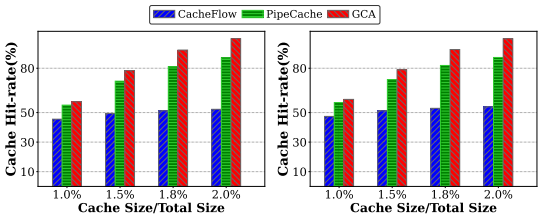

Fig. 6. The cache hit-rate under different cache sizes.

Scalability on cache size. Fig. 6 shows the algorithm performance when the cache size changes. The horizontal axis represents the number of hardware resource, in which $x\%$ indicates that the hardware resource can cache $x\%$ of the total rules when caching them in a single flow table. We can see from Fig. 6 that as the cache size increases, all the caching algorithms show a higher cache hit-rate on both ACL and FW flow tables. On the other hand, GCA always exhibits a better performance than CacheFlow and PipeCache. When the cache size is set to 1.8%, GCA achieves the 93% cache hit-rate on FW flow tables, while CacheFlow and PipeCache are only 53% and 82%, respectively.

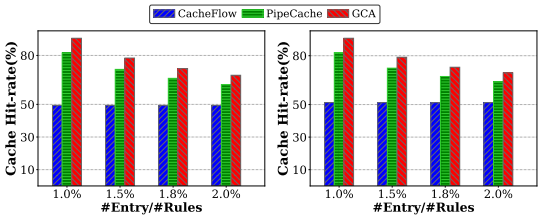

Fig. 7. The cache hit-rate under different entry sharing ratios.

Scalability on entry sharing ratio. Fig. 7 shows the algorithm performance when the entry sharing ratio varies. We set the entry sharing ratio of the first two stages of the three-stage flow table at 1% and 2%, respectively, the entry sharing ratio of the last stage ranges from 1% to 2%. As shown in Fig. 7, with the increasing of the entry sharing ratio, *i.e.*, the probability of sharing common entries between rules reduces, the cache hit-rate of PipeCache and GCA decreases. Meanwhile, CacheFlow is stable because it does not make use of entry sharing to improve the cache hit rate.

Scalability on flow table stages. Fig. 8 shows the performance of the algorithms under different number of flow table stages. Since CacheFlow is not affected by the number of stages of the flow table, so its performance is stable when the latter one changes. The cache hit-rate of GCA and PipeCache decreases when the number of the stages of the flow table increases because cache hit requires matching an entry in every stage of the flow table.

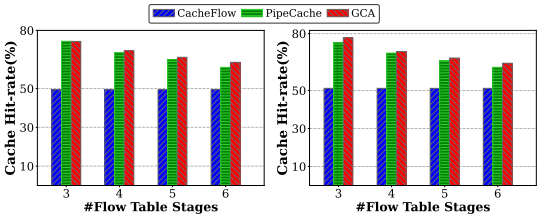

Fig. 8. Cache hit-rate under different number of stages.

VII. Conclusion

This paper abstracts and models the problem of the multi-stage flow table caching and proves the NP-hardness of optimal multi-stage flow table caching for the first time. The proposed approximation algorithm GCA adopts the greedy idea and achieves a 10∼30% higher cache hit-rate than the state-of-the-art solutions with the help of considering both the rule popularity and entry popularity and adaptive hardware resource assignment for different flow tables.

References

[1] Nick McKeown et al. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM CCR.*, 2008.
[2] Tian Pan et al. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *ACM SIGCOMM*, 2021.
[3] Rui Miao et al. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *ACM SIGCOMM*, 2017.
[4] Changhoon Kim et al. In-band network telemetry via programmable dataplanes. In *Proc. ACM SIGCOMM*, Ind. Demo, 2015.
[5] Intel. Barefoot Tofino: programmable switch series up to 6.5Tbps. https://barefootnetworks.com/media/white_papers/BarefootWorlds-Fastest-Most-Programmable-Networks.pdf.
[6] Broadcom. High Capacity StrataXGS®Trident II Ethernet Switch Series. https://www.broadcom.com/products/ethernet-connectivity/switch-fabric/bcm56850/.
[7] Cisco. Cisco Silicon One ASIC. https://blogs.cisco.com/tag/cisco-silicon-one-asic.
[8] Naga Katta et al. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *SOSR*, 2016.
[9] M Mitchell Waldrop. The chips are down for Moore's law. *Nature News*, 530(7589):144, 2016.
[10] Alex X Liu et al. TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs. *IEEE/ACM TON*, 2009.
[11] Ben Pfaff et al. The Design and Implementation of Open vSwitch. In *USENIX NSDI*, 2015.
[12] Xiaoqiao Meng et al. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *IEEE INFOCOM*, 2010.
[13] Ying Wan et al. T-cache: dependency-free ternary rule cache for policy-based forwarding. In *Proc. IEEE INFOCOM*, pages 536–545, 2020.
[14] Peng He et al. Partial order theory for fast TCAM updates. *IEEE Trans. Netw.*, 26(1):217–230, 2017.
[15] Jang-Ping Sheu et al. Wildcard rules caching and cache replacement algorithms in software-defined networking. *IEEE TNSM*, 2016.
[16] Zixuan Ding et al. Update cost-aware cache replacement for wildcard rules in software-defined networking. In *ISCC*, 2018.
[17] Raj Jain. Introduction to OpenFlow, 2013.
[18] Venkatachary Srinivasan et al. Packet classification using tuple space search. In *ACM SIGCOMM*, 1999.
[19] Kalapriya Kannan et al. Compact TCAM: flow entry compaction in TCAM for power aware SDN. In *ICDCN.*, 2013.
[20] Bo Yan, Yang Xu, Hongya Xing, et al. CAB: A reactive wildcard rule caching system for software-defined networks. In *HotSDN*, 2014.
[21] Jialun Yang et al. Pipecache: High hit rate rule-caching scheme based on multi-stage cache tables. *Electronics*, 2020.
[22] Renata Sotirov. On solving the densest k-subgraph problem on large graphs. *Optimization Methods and Software*, 35(6):1160–1178, 2020.
[23] Jiří Matoušek et al. Classbench-ng: Recasting classbench after a decade of network evolution. In *ANCS*, 2017.