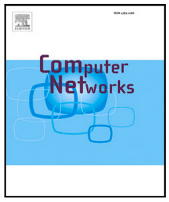
Computer Networks 238 (2024) 110092

Contents lists available at ScienceDirect

Computer Networks

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



Elastically accelerating lookup on virtual SDN flow tables for software-defined cloud gateways

Bing Xiong [a], Jing Wu [a], Qiaorong Huang [a], Jinyuan Zhao [b], Qiang Tang [a], Jin Zhang [a,*], Kun Yang [c], Keqin Li [d]

[a] School of Computer and Communication Engineering, Changsha University of Science and Technology, Changsha 410114, PR China
[b] School of Information Science and Engineering, Changsha Normal University, Changsha 410199, PR China
[c] School of Computer Science and Electronic Engineering, University of Essex, Wivenhoe Park, Colchester CO43SQ, UK
[d] Department of Computer Science, State University of New York at New Paltz, NY 12561, USA

ARTICLE INFO

MSC:
00-01
99-00

Keywords:
Software-defined cloud gateways
Virtual SDN flow tables
Tuple space search
Network traffic jitters
Elastic accelerating caches

ABSTRACT

In recent years, Software-Defined Networking has been gradually applied to cloud gateways to provide efficient, reliable and flexible data transmission services for various large-scale cloud platforms. However, massive concurrent accessing produces huge network traffic from tenants to cloud platforms, which aggregates at cloud gateways and brings serious performance bottlenecks regarding packet classification. To solve this problem, this paper proposes an elastically accelerated lookup method of virtual SDN flow tables for software-defined cloud gateways. The method caches active exact flows in virtue of network traffic locality, enabling most packets to directly hit the cache and bypass tuple space search, which significantly accelerates flow table lookup. Focusing on network traffic jitters, the cache adaptively adjusts its capacity according to the dynamic changes of the number of active exact flows to maintain high cache hit rates, aiming to achieve elastic acceleration of flow table lookup. Furthermore, we theoretically derive the performance metrics of our proposed method such as cache hit rates, cache yield rates and average search length, based on the Zipf distribution model of network traffic. Eventually, we evaluate the performance of our proposed elastically accelerated lookup method by experiments with real network traffic traces. Experimental results indicate that our proposed method outperforms existing cache-accelerated methods with stable cache hit rates around 80% and the speedup of average search length up to 2.84 even under network traffic jitters.

1. Introduction

As a popular information service pattern in today's Internet era, cloud platforms have gradually penetrated into various types of industries in recent years, and become a key information infrastructure in contemporary society [1]. In cloud platforms, a cloud gateway is responsible for information interaction between datacenters and end-users or other datacenters, and has a crucial impact on cloud service performance and quality of user experience. However, traditional cloud gateways only support the conversion between specific protocols often bound to hardware, and is difficult to adapt to new published protocols. This greatly impedes the rapid update and upgrade of cloud gateways to meet the fast development of cloud platforms. As a novel network paradigm with the separation of control from data and high programmability, Software-Defined Networking (SDN) [2] significantly simplifies the functions of packet switching devices, raises data transmission efficiency, reduces network construction and operation costs, and support rapid network updates and upgrades. Nowadays, a number of vendors design software-defined cloud gateways to achieve fast and flexible packet classification [3,4].

To achieve the flexible definition of packet flows with different granularity, SDN introduces wildcards into the match fields of its flow tables, primarily composed of key fields from protocol headers. However, this disables virtual SDN flow tables to directly apply hashing methods to achieve fast lookup. Until now, tuple space search (TSS) [5] is a prevalent approach to address this problem. It divides all rules in a flow table into a certain number of tuples in accordance with the mask identifying the position of wildcards in the match fields. Subsequently, Each tuple can be organized with a hash table and be looked up with its unique mask and the match fields in each flow entry. As for an arrived packet, it is ignorant of its mask and thus cannot locate its corresponding tuple in the flow table. Consequently, each packet has to match against all tuples one by one until a flow entry is found.

* Corresponding author.
E-mail address: jzhang@csust.edu.cn (J. Zhang).

https://doi.org/10.1016/j.comnet.2023.110092
Received 16 June 2023; Received in revised form 16 October 2023; Accepted 5 November 2023
Available online 9 November 2023

This implies that it needs to undergo the lookup failures of multiple tuples before a successful match. More seriously, it will present a sharp increase in the number of tuples and the size of each tuple under network traffic surges especially induced by cyber attacks [6], which leads to significant growth in the lookup overheads of the flow table.

To speed up the lookup of virtual SDN flow tables, some researchers proposed to offload the flow table lookup of incoming packets in cloud gateways to general-purpose PCs, programmable NICs, and other hardwares, such as AccelNet [7] and OVS-CAB [8]. However, these approaches usually require to add additional hardwares with high cost, which impedes their extensive applications in virtualized platforms. Moreover, much work utilized decision trees to divide flow rule space into several subspaces, and locate the respective subspace for each arrived packet, such as EffiCuts [9], PartitionSort [10,11] and CutTSS [12]. But each insertion or deletion requires a series of adjustment operations on decision trees with slow update speeds. In addition, some scholars also suggested to cache important flows in network traffic to bypass flow table lookup, such as CuckooFlow [13] and BLP [14]. Nevertheless, existing flow caches are hard to achieve stable and high hit rates due to their fixed capacity, especially under network traffic jitters even cyber attacks [15].

To solve the above problems, this paper proposes an elastically accelerated lookup method of virtual SDN flow tables for software-defined cloud gateways. We first investigates into network traffic locality and jitters in cloud gateways. Then we devise an elastic accelerating cache to achieve stable acceleration of flow table lookups, in the presence of network traffic jitters even cyber attacks. Further, we design an elastically accelerated lookup method of virtual SDN flow tables and quantitatively analyze its average search length. Finally, we verify the performance of our proposed elastic accelerating cache and flow table lookup method with real network traffic traces. The main contributions of this paper are summarized as follows.

- Devising an elastic accelerating cache to accommodate active exact flows at software-defined cloud gateways, according to the distribution characteristics of network traffic. In particular, the cache capacity is dynamically adjusted to keep consistence with the varying number of active exact flows, for stably achieving high cache hit rates.
- Designing cache scaling conditions based on cache replacement rates, which real-timely senses the dynamic changes in the number of active exact flows. Meanwhile, we derive the hit rates and yield rates of our proposed elastic accelerating cache based on the Zipf distribution of network traffic.
- Building an elastically accelerated lookup method of virtual SDN flow tables, which enables the majority of packets directly hit the above cache to bypass their tuple space searches, and stably achieves fast flow table lookup in the case of network traffic jitters even cyber attacks.
- Providing the algorithmic implementation of the flow table lookup method, deriving average search length of our proposed flow table lookup method based on its cache hit rates, and verifying its superiority by experiments with real network traffic traces.

The remainder of this paper is organized as follows. Section 2 presents related works. In Section 3, we introduce software-defined cloud gateways and analyze the distribution characteristics of its network traffic. Section 4 describes our designed elastic accelerating cache including its various operations, and deduces the theoretical expressions of its hit rates and yield rates. In Section 5 we provide our proposed elastically accelerated lookup method of virtual SDN flow tables, and theoretically derive its average search length. Section 6 evaluates the performance of our proposed elastic accelerating cache and flow table lookup method with real network traffic traces by experiments. In Section 7, we conclude the paper.

2. Related work

To accelerate packet classification in virtual SDN switches, some researchers proposed to offload the flow table lookup of each arrived packet to hardware, such as general-purpose PCs, special hardware and programmable NICs. Molnár et al. [16] designed a switch architecture ESwitch, which exploits on-the-fly template-based code generation to compile any SDN pipeline into efficient machine code as fast data path. However, this architecture limits network flexibility and programmability. Miano et al. [17] offloaded the portion of rules supported by hardware ASICs while keeping the remaining ones in software, supporting the presence of multiple hardware tables. However, their acceleration is limited, since they cannot offload flow rules incompatible with the ASICs. Similarly, Varvello et al. [18] designed a GPU-accelerated virtual switch GSwitch, which accelerates the linear and tuple search of virtual SDN flow tables by exploiting high parallelism and latency-hiding capabilities of GPUs. Fu et al. [19] proposed a FPGA-accelerated SDN software switch, which offloads time-consuming functions such as packet buffer management and packet parsing to FPGAs. Firestone et al. [7] presented AccelNet to offload packet forwarding function from host networking to the Azure SmartNIC, which increases packet forwarding rates. Furthermore, Gao et al. [8] offloaded entire data plane of OVS to SmartNIC, which effectively enhances the packet forwarding performance of OVS. However, these designs usually require to add additional hardwares with high cost, and is difficult to be extensively deployed in the virtualized environments.

Some research efforts accelerated the lookup of virtual SDN flow tables by elaborately designing their data structures and algorithms. Lookup methods based on decision trees, such as HiCut [20], HyperCuts [9], EffiCuts [21], CutSplit [22] and ByteCut [23], divided flow rule space into several subspaces, and located respective subspace for each arrived packet. However, decision trees are hard to update, which requires to reconstruct the trees with a series of adjustments. As an alternative, tuple space search (TSS) is applied to achieve fast updates of virtual SDN flow tables. Nevertheless, the lookup performance of TSS is sensitive to the size of flow tables, especially the number of tuples. To improve upon TSS, James et al. [24] proposed an online packet classification algorithm TupleMerge, which reduces the number of tuples by merging similar flow rules. To incorporate the advantages of decision trees and TSS, Yingchareonthawornchai et al. [10,11] presented an integrated approach PartitionSort, which leverages ruleset sortability to gain a small number of partitions and utilizes multi-dimensional interval trees to obtain fast lookup and updates on each partition. Li et al. [12] proposed a two-stage scheme CutTSS, which first constructs partial decision trees from several rule subsets grouped with respect to their small fields, and applies TSS to handle packet classification at non-leaf terminal nodes. However, the number and size of tuples will surge in the presence of network traffic jitters and even cyber attacks [25], which gives rise to a sharp decline in the lookup performance of the flow tables.

To date, many scholars leveraged caching techniques to accelerate the lookup of virtual SDN flow tables. Congdon et al. [26] proposed a prediction cache based on network traffic locality, which maps packet signature into flow key and predicts respective flow rules for incoming packets, thereby reducing the latency of packet classification. Pfaff et al. [27] developed a well-known virtual switch, Open vSwitch, which first selects non-intersecting and non-priority megaflows and holds them in its kernel space, enabling the majority of packets to go through fast kernel data-path. Then it caches microflows typically TCP connections in its megaflow table, which boosts packet classification speed in the kernel. Moreover, Wang et al. [28] devised a packet-tuple mapping cache CuckooDistributor to directly locate tuples for arrived packets, aiming to reduce the tuple space search overheads of the megaflow table. Besides, they adaptively adjusted the insertion speed of the microflow cache in terms of its historical miss rates to increase its hit rates. Xiong et al. [13] designed an active-exact-flow cache based

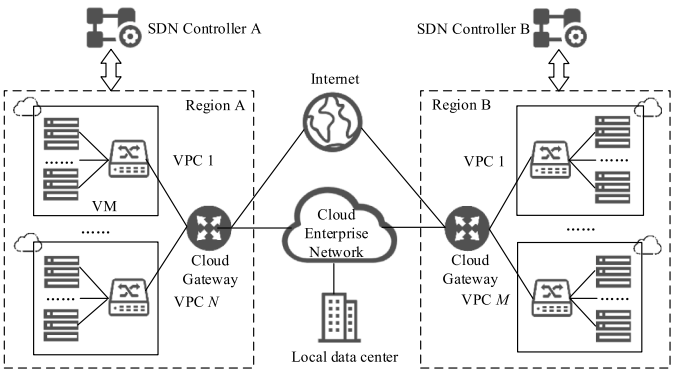


Fig. 1. A typical deployment scenario of software-defined cloud gateways.

on Cuckoo hashing, which enables most of packets directly hit the cache and locate their corresponding flow entries. Zhou et al. [14] applied bounded linear probing to reduce the hash collisions of the flow cache, and designed probabilistic bubble LRU to achieve fast cache replacements, which improves the cache utilization and hit rates. However, these caches are hard to achieve stable and high hit rates due to their fixed capacity, especially under network traffic jitters.

3. Motivation

3.1. Software-defined cloud gateways

Cloud computing has become a prevalent information service pattern in today's Internet era, with its on-demand services, dynamic scalability, strong reliability, and cost-effectiveness. It has been widely employed in various kinds of areas to build cloud platforms, which provide efficient and flexible information services. In cloud platforms, a cloud gateway takes charge of data transmission between datacenters and end-users or other datacenters, and has a great effect on communication performance and quality of user experience. However, traditional cloud gateways only implement the translation between specific protocols closely associated with hardware, without the ability to accommodate new released protocols. This greatly prevents cloud gateways from fast update and upgrade to satisfy the rapid development requirements of cloud platforms. With the growing maturity and widespread applications of SDN, some manufacturers have designed software-defined cloud gateways, such as Tripod [1] and Cloud-Gateway Automation [4], to be deployed in their cloud platforms. By leveraging the separation of data and control and high programmability of SDN, cloud gateways can reduce their reliance on hardware, achieve flexible network configuration and management, improve data transmission efficiency, and significantly reduce network deployment and maintenance costs.

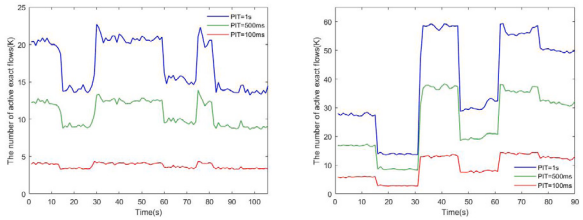
Fig. 1 depicts a typical deployment scenario of software-defined cloud gateways. The cloud platform is segmented into multiple regions, each of which contains numerous virtual private clouds (VPCs) that provide independent and secure information services to tenants. Each VPC consists of a varying number of virtual machines interconnected through virtual switches. All VPCs in a region are uniformly connected to cloud gateways, which provide access channel to the Internet and local datacenters. As for each region, its SDN controller establishes its network view, and calculates the forwarding paths of packet flows to generate flow rules, installed into cloud gateways and virtual switches. Upon receiving a packet, the cloud gateways perform lookup on its virtual SDN flow table with the flow identifier of the packet. If the lookup succeeds, we retrieve the actions in the matched flow entry, and apply them to the packet. Otherwise, a flow setup request is sent to the controller.

A large cloud platform usually hosts numerous enterprise tenants and tens of thousands of individual tenants. Each enterprise tenant probably owns thousands of VMs and provides concurrent accessing for millions of customers. The number of visits on the cloud platform demonstrates a trend of rapid growth, with the expanding business of enterprise tenants and the increasing number of individual tenants. The simultaneous accessing of numerous customers generate huge network traffic converged on cloud gateways, leading to their serious performance bottlenecks of packet forwarding. Meanwhile, customer visits demonstrate significant diversity during different time periods, such as, daytime vs. evening, weekdays vs. weekends, off-season vs. peak season. Moreover, many e-commerce platforms sometimes conduct online promotion activities on shopping festivals like Double Eleven and Black Friday, resulting in an explosion of customer visits and online transactions. These situations pose unprecedented challenges on the data transmission of cloud gateways. In summary, it is imperative to achieve stable and fast packet forwarding in software-defined cloud gateways.

3.2. Network traffic characteristics at cloud gateways

In software-defined cloud gateways, network traffic chiefly originates from information services provided by private clouds leased by enterprises or individuals, as well as management, upgrades, new business deployments and other system operations in cloud platforms. Both customer accessing behavior and system operations generate a sequence of data transmission activities in a short time period, and the amount of transmitted data greatly varies in packet flows. This means that network traffic at cloud gateways presents obvious locality [29], manifested as: (a) from the perspective of spatial dimension, a majority of packets are aggregated in a few flows, while most flows only account for a small number of packets [30]; (b) from the perspective of time dimension, packets in an elephant flow tend to arrive in batches [31]. Consequently, the state of a flow can be classified into active state with a batch of transmitting packets and idle state with only a few scattered packets or none in transmission. Specifically, each flow is supposed to stay at the idle state when it first appears. For subsequent packets, the flow will come into the active state, if the interval time between currently-arrived packet and the latest one in the flow is less than a preset threshold [32]. Since SDN introduces wildcards into the match fields of its flow tables, each wildcarding flow can be regarded as a convergence of multiple exact flows. Thus we can cache active exact flows in software-defined cloud gateways, to skip heavy flow table lookup for subsequent packets and achieve fast packet classification.

The distribution of customer visits exhibits a noticeable bias over time, and system operations also demonstrate evident burstiness. These are prone to result in the jitter of network traffic in cloud gateways, as well as the number of active exact flows. To verify network traffic jitters, we select publicly released datasets [33] and observe the quantitative change of active exact flows. We choose a network traffic trace with the duration of 106 s around midnight on January 22, 2020, and a synthetic one with the duration of 15 s on each following days (2017/7/26, 2018/9/3, 2018/11/9, 2019/9/3, 2019/10/18, 2020/9/26). We set the packet interval threshold (PIT) of active exact flows respectively as 100 ms, 500 ms, and 1 s, and measure the number of active exact flows in Fig. 2. As seen from Fig. 2, the number of active exact flows shows apparent volatility regardless of the value of the PIT threshold. For instance, the number of active exact flows in Fig. 2(a) varies in a small range between $15K$ and $20K$ when PIT is set to 1 s, while that in Fig. 2(b) sharply fluctuates between $15K$ and $60K$. Consequently, the acceleration cache of virtual SDN flow table should adapt to the dynamic changes in the number of active exact flows, so that software-defined cloud gateways can stably achieve favorable packet classification performance.



(a) Single time period (b) Multiple time periods

Fig. 2. The number of active exact flows with different thresholds.

4. Elastic accelerating cache of virtual SDN flow tables

4.1. Cache design

To effectively reduce heavy lookup overheads of virtual SDN flow tables, several research efforts have designed exact-flow caches, such as MicroFlow [27], CuckooFlow [13], to bypass the tuple space search of flow table lookup. However, the MicroFlow cache kept newly arrived exact flows without sufficiently taking advantage of network traffic locality, which leads to a obvious room for improvement in its hit rates. To remedy this defect, the CuckooFlow cache held active exact flows to effectively increase cache hit rates. Nevertheless, the Cuckoo cache is difficult to cope with network traffic jitters in software-defined cloud gateways, and cannot stably achieve satisfactory acceleration effects, due to its fixed cache capacity. When network traffic sharply grows, the cache cannot accommodate newly-arrived active exact flows, resulting in a significant decline in cache hit rates and cache acceleration effect. Conversely for rapidly decreasing network traffic, there will be an increasing number of vacant entries emerged in the cache, which gives rise to cache space waste and low cache utilization. In this paper, we design an elastic accelerating cache (EAC) in Fig. 3, which adaptively adjusts its capacity in accordance with the varying number of active exact flows in network traffic, to steadily achieve high cache hit rates and utilization.

The EAC cache consists of k logical segments with the length l and their corresponding sub-hash functions. As for a segment, each active exact flow is mapped into a candidate position in it by its respective sub-hash, where flow information is recorded. An active exact flow is usually identified by multiple protocol fields such as source IP address, destination IP address, source port, destination port, protocol type, etc., with at least 13 bytes. To save cache space and accelerate cache matching, each cache entry keeps flow fingerprint *ffp* as its key typically 2 or 4 bytes, generated from the identifier *fid* of its active exact flow. Besides, each cache entry also contains the flow entry address *addr* and the timestamp time. The *addr* indicates the flow entry corresponding to the active exact flow for further packet forwarding. The time records the arrival time of the latest packet within the flow, for the replacement and timeout scanning of the cache. The sub-hash function for each segment is designed as the modulo l of the random permutation of m ($\log_2 l \le m < n$) non-repeating bits randomly selected from the *ffp* with the length n.

It is requisite to real-timely sense the dynamic changes of the number of active exact flows for precise cache capacity adjustment. We define cache replacement rates, i.e., the number of cache replacements per unit time, as the indicator for determining whether the cache needs to be scaled. As for rapid increases in the number of active exact flows, the cache will be quickly filled up. Subsequent active exact flows intensively arrived will frequently replace into the cache, resulting in a significant growth in the number of cache replacements. In this case, we add a new segment into the cache to accommodate newly-arrived active exact flows to maintain high cache hit rates. When the number of active exact flows sharply decreases, subsequent active exact flows sparsely arrived will be directly stored in vacant cache entries generally without replacements, since more and more expired flows will be eliminated from the cache in time by timeout scanning. This significantly reduces the number of cache replacements, and we delete one cache segment after transferring active exact flows in it to other segments. By this way, we aim to improve cache utilization while keeping high cache hit rates. In summary, the number of active exact flows fluctuates with network traffic jitters, and the cache adaptively expands or retracts to steadily achieve high cache hit rates and utilization.

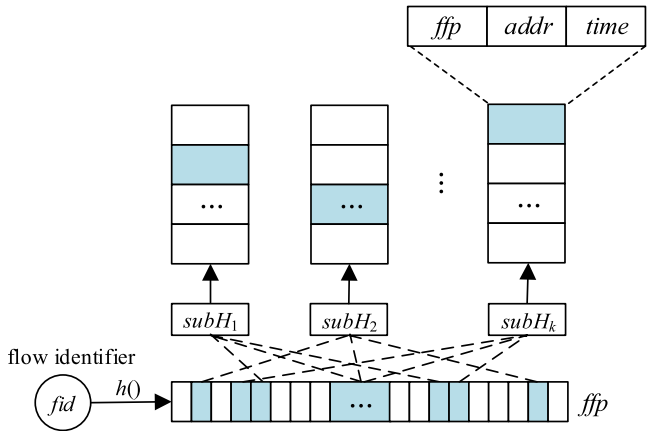


Fig. 3. Our designed elastic accelerating cache.

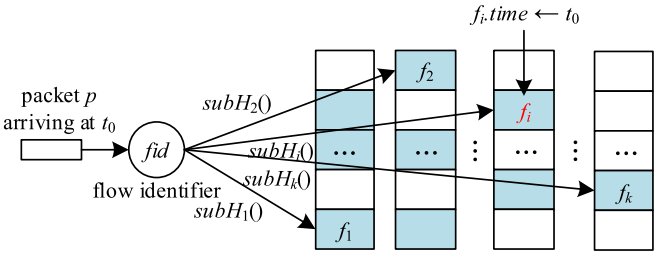


Fig. 4. An example of cache lookup.

4.2. Cache operations

4.2.1. Cache lookup

Fig. 4 describes an example of cache lookup. Suppose that the EAC cache contains k segments, each of which corresponds to a sub-hash function $subH_i (1 \le i \le k)$. When a packet p within a flow f arrives at time t_0, its flow identifier *fid* is calculated and hashed into its flow fingerprint *ffp*, for subsequent parallel lookup on all segments. For the ith segment, we calculate the mapping position pos ($1 \le pos \le l$) of the flow by its respective sub-hash function $subH_i(.)$, and locate the corresponding candidate cache entry $Cache[i][pos]$, for further matching with the flow fingerprint *ffp*. If the flow fingerprint successfully matches a mapped cache entry with a flow f_i, we locate its entry in virtual SDN flow table with the index *addr* in the cache entry. Meanwhile, the timestamp *time* in the cache entry is updated to t_0. Otherwise, we return null.

4.2.2. Cache update

In a software-defined cloud gateway, each exact flow repeatedly switches between the active state and the idle one, along with the intermittent arrival of packet batches [32]. Therefore, the active-exact-flow cache should support dynamic update to ensure cache acceleration effect. When a flow entry is deleted from a virtual SDN flow table, all exact flows mapped into it should be deleted from the cache, to ensure correct mapping between the cache and the flow table. We describe the insertion and deletion of our proposed EAC cache as follows.

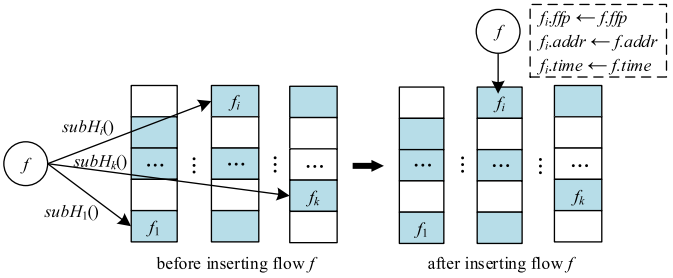


Fig. 5. An example of cache insertion.

(a) **Cache insertion**: For an active exact flow f to be inserted, we first calculate its flow fingerprint *ffp* with its flow identifier *fid*. Then, we map the flow into each segment of the cache with its sub-hash function, and check whether there is any empty mapped cache entry. Each cache insertion has two cases:

Case 1: If there is any empty mapped cache entry, we directly record the information of the flow f into the first one, such as the flow fingerprint *ffp*, the index of the corresponding flow entry *addr*, and the timestamp *time*.

Case 2: If there is no empty mapped cache entry, we will replace the flow without any arrived packet for longest time in all mapped cache entries. Fig. 5 provides a specific example of the cache insertion in this case. We first retrieve the timestamp *time* of each mapped cache entry, and find out the flow f_i with the oldest timestamp. Then, we replace f_i with the flow f.

(b) **Cache deletion**: Similarly to the above cache insertions, we first calculate the flow fingerprint *ffp* with the identifier *fid* of the flow f to be deleted. Then, we calculate the mapping position of the flow in each segment of the cache, and match all mapped cache entries one by one with the *ffp*. If a cache entry is successfully matched, we will reset it.

(c) **Timeout scanning**: As time goes on, an active exact flow in the cache probably becomes idle or even terminated. Therefore, the cache needs to apply timeout scanning to timely eliminate all expired flows to accommodate newly-arrived active exact flows. We first obtain current system time, and then scan each cache segment to check whether each flow within it has expired. In particular, we read the timestamp in each cache entry, and calculate the time interval between it and the current system time. Then, we compare the time interval with the packet interval threshold *PIT*. If the time interval goes beyond the *PIT*, it means that the flow in the cache entry has expired, and we reset the entry.

4.2.3. Cache scaling

When the number of active exact flows noticeably increases, the EAC cache needs to add segments to accommodate newly-arrived active exact flows to keep high cache hit rates. However, adding too many segments probably leads to poor cache utilization, due to the presence of quite a few empty entries in added segments. Conversely, a sharp decrease in the number of active exact flows is likely to result in a non-negligible number of empty cache entries. Hence, it is imperative to design appropriate scaling conditions for the EAC cache, so its capacity always matches with the number of active exact flows. In our design, the EAC cache timely senses the dynamic changes of the number of active exact flows by cache replacement rates, and adaptively adjusts its capacity by adding or deleting segments. Considering the evident diversity of network traffic at different time under various application scenarios, it is extremely difficult to set a uniform threshold of cache replacement rates. Hence, we employ the variation of cache replacement rates as the indicator of performing cache scaling. Fig. 6 illustrates an example of cache expansion and retraction.

In Fig. 6, the EAC cache calculates its cache replacement rate in each fixed period (e.g., 1 s) to obtain its variation from that in previous one.

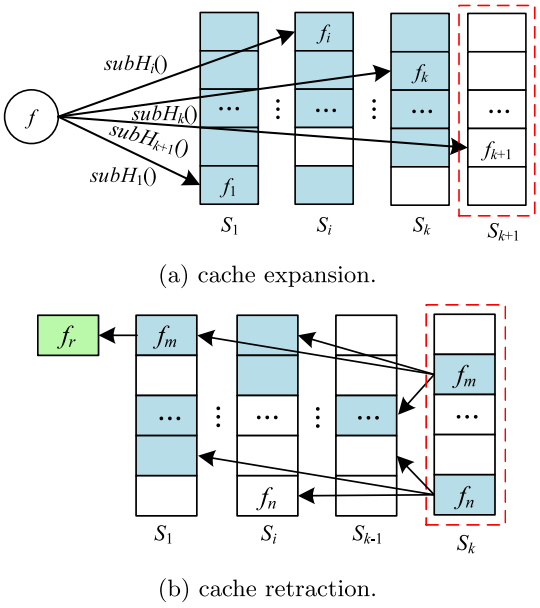


Fig. 6. An example of cache expansion and retraction.

As for an apparent increasing range in cache replacement rates, the cache immediately adds a segment S_{k+1} and generates a corresponding sub-hash function $subH_{k+1}$ (.) in Fig. 6(a). The newly-arrived active exact flows f is kept in the segment S_{k+1}, due to the complete occupation of its all mapping positions in the front k segments. As for a distinct drop in cache replacement rates, we transfer all active exact flows in the last segment S_k to the other segments as much as possible, and delete the segment S_k in Fig. 6(b). For each flow to be transferred (e.g., f_m, f_n), we calculate its mapping position in each preceding segment. If there is an empty position, we directly deposit the transferred flow f_n into it. Otherwise, we find out the flow without any arrived packet for longest time among them, and replace it with the transferred flow f_m. In addition, when there is a very small number of cache segments especially in initial phases, it is vital to add segments to ensure high cache hit rates. Meanwhile, when there is an excessive number of cache segments for the sharply-increasing number of active exact flows, the number of segments should be limited to ensure cache yield rates. Therefore, we set the lower bounds of cache hit rates and cache yield rates, to respectively impose lower and upper bounds on the number of segments, aiming to ensure overall cache performance.

4.3. Performance metrics

4.3.1. Cache hit rates

The cache hit rate is a critical metric to evaluate cache performance. In cloud gateways, network traffic exhibits evident locality due to user access behavior and system operations. Specifically, most packets belong to a few flows from the perspective of packet distribution [30], and packets within a flow tend to arrive in batches in terms of packet transmission. Extensive studies have applied the Zipf distribution to characterize the quantitative distribution of packets in flows [34] according to network traffic locality. Suppose that there are N packet flows ranked as $f_1, f_2, \ldots, f_N$ in descending order of their size, the packet arrival probability $p(r)$ of the flow f_r with the ranking r (r=1, 2, …, N) can be expressed as (1).

$$p(r) = \frac{C}{r^{\alpha}}. \tag{1}$$

The parameter α in (1) represents the skewness of all packets distributed over flows, and C is a positive constant. Considering that the EAC cache holds active exact flows, the packet arrival probability

Table 1
The estimation of the hit rates of the EAC cache.

N	α	k	CHR_{EAC}
$100K$	0.96	5	81.18%
$100K$	0.96	4	79.03%
$100K$	0.98	4	80.88%
$100K$	0.98	3	78.32%
$200K$	0.98	7	80.66%
$200K$	0.98	6	79.36%
$200K$	1.00	6	81.35%
$200K$	1.00	5	79.92%
$400K$	1.00	9	80.17%
$400K$	1.00	8	79.30%
$400K$	1.02	8	81.45%
$400K$	1.02	7	80.35%

$p(r)$ can be regarded as the metric to measure the activity degree of the exact flow fr. Suppose that the EAC cache always keeps the top kl active flows, its cache hit rates CHR_{EAC} can be calculated as:

$$CHR_{EAC} = \frac{\sum_{r=1}^{kl} p(r)}{\sum_{r=1}^{N} p(r)} = \frac{\sum_{r=1}^{kl} r^{-\alpha}}{\Gamma_{\alpha}(N)}. \quad (2)$$

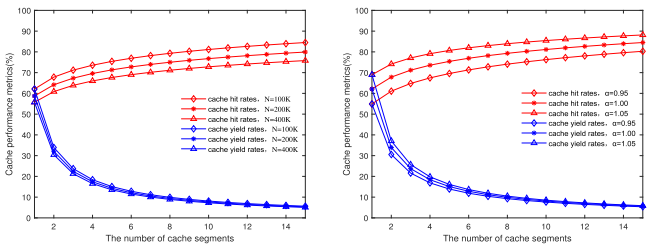
where $\Gamma_{\alpha}(N)=\sum_{i=1}^{N} r^{-\alpha}$. According to (2), we estimate the cache hit rate CHR_{EAC} in Table 1, with the cache segment length l fixed as $3K$, and suitable segment number K for different number of exact flows N and skewness of flow distribution α. As seen from Table 1, we can still keep high cache hit rates even if appropriately reducing the number of cache segments k, for the increasing skewness of flow distribution α and a certain number of exact flows N. By contrast, it needs to increase the number of cache segments k to keep cache hit rates at similar levels, for specific skewness of flow distribution α and significant increase in the number of exact flows N. In summary, cache hit rates can be maintained at approximately 80%, by adaptively adjusting cache capacity for various network traffic with different parameters. This means that our designed EAC cache can steadily achieve favorable acceleration effect, and effectively adapt to network traffic jitters.

4.3.2. Cache yield rates

When network traffic jitters, the EAC cache dynamically expands its capacity to maintain high cache hit rates, which also brings about more storage overheads. Thus it needs to design metrics to measure the performance benefit of invested cache capacity. With reference to the yield rates in economics [35,36], we define the cache yield rate (CYR) in (3) as the ratio of the cache hit rate CHR_{EAC} in (2) to the number of cache segments k, since the EAC cache is composed of a varying number of segments.

$$CYR_{EAC} = \frac{CHR_{EAC}}{k} = \frac{\sum_{r=1}^{kl} r^{-\alpha}}{k\Gamma_{\alpha}(N)}. \quad (3)$$

By fixing the length of each segment l as $1K$, we estimate the cache hit rate and cache yield rate with increasing number of cache segments in Fig. 7, for different number of exact flows N and different skewness of flow distribution α. As seen from Fig. 7, the cache hit rate steadily rises while the cache yield rate reduces with increasing number of cache segments. When the number of segments grows from 4 to 8 with $N = 100K$ and $\alpha = 1.00$, the cache hit rate increases from 73.6% to 79.3% and the cache yield rate declines from 18.4% to 9.9%. As for the increasing number of exact flows N, the number of active exact flows will accordingly rise and the EAC cache needs to add segments to ensure high cache hit rates. When N increases from $100K$ to $200K$ in Fig. 7(a), the number of segments should be added from 6 to 9 for keeping cache hit rates above 75%. As for the growing skewness of flow distribution α with stronger network traffic locality, the EAC cache with an identical number of segments will achieve higher cache hit rates and cache yield rates. When α rises from 1.00 to 1.05 with the number of segments 10 in Fig. 7(b), the cache hit rate will increase from 81.2% to 85.4%, and the cache yield rate will come up from 8.1% to 8.5%. Under such circumstances, it is suitable to reduce the number of segments to save cache space, while maintaining high cache hit rates.



(a) α=1.0 (b) N=100K

Fig. 7. The cache hit rate and cache yield rate with increasing number of segments.

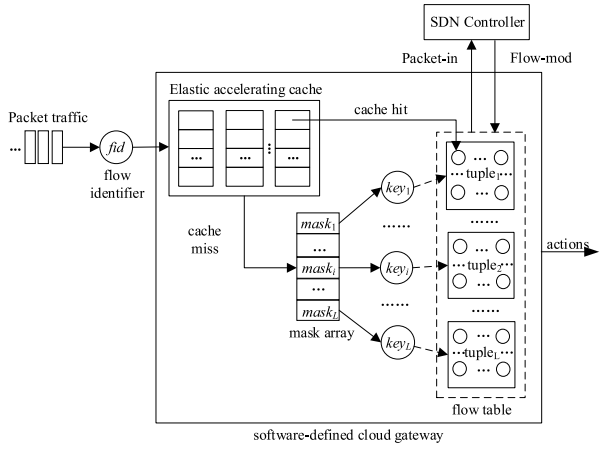


Fig. 8. The elastically accelerated lookup method of virtual SDN flow tables in software-defined cloud gateways.

5. Elastically accelerated lookup methods of virtual SDN flow tables

5.1. Algorithmic description

With the above EAC cache, we further construct an elastically accelerated lookup method of virtual SDN flow tables for software-defined cloud gateways in Fig. 8. This method identifies active exact flows in network traffic, and accommodates them in the EAC cache. As for each arrived packet, the cloud gateway extracts its flow identifier, and matches against the EAC cache. If the match succeeds, we locate a flow entry by the matched cache entry, and directly forward the packet in accordance with the action set in the flow entry. Otherwise, it still needs to further perform tuple space search on the flow table for packet forwarding. Owing to the accommodation of active exact flows, the EAC cache will be hit by a majority of packets in network traffic, by-passing flow table lookup. When network traffic jitters, the EAC cache adaptively expands or retracts its capacity to always hold all active exact flows. By this way, it is expected to keep high cache hit rates, and the lookup method can stably achieve fast packet classification in the cloud gateway.

Algorithm 1 describes the algorithmic implementation of the above elastically accelerated flow table lookup method. Upon receiving a packet, the cloud gateway first parses its protocol headers to extract its key fields, and calculates its flow identifier fid for its lookup on the EAC cache (line 2). If the lookup succeeds to match a cache entry, we locate the corresponding flow entry with the index addr within it. Subsequently, we retrieve the match fields in the flow entry and verify

whether they match with the flow identifier fid (line 3–5). As for a successful match, we skip the tuple space search of the flow table and directly forward the packet in accordance with the actions in the flow entry (line 6). Finally, we update the flow entry including counters, and the timestamp in the cache entry with the arrival time of the packet (line 7–8).

As for the case of failed cache lookup, we continue to perform tuple space search on the flow table. In particular, we match against all tuples one by one with their masks and the flow identifier, until a flow entry is found. As for each tuple, we compute a masked key by the bitwise AND of its mask and the flow identifier, to match against the tuple (line 12–14). If the match succeeds, we will apply the actions in the matched flow entry to the packet, and update the flow entry (line 15–18). Subsequently, we check whether the exact flow, which the packet resides in, has turned into the active state. In particular, we calculate the arrival time interval between the packet and the latest packet in the exact flow, and compare it with the PIT threshold (line 22). If it is less than the PIT threshold, we will insert the exact flow into the EAC cache (line 23–24). If the tuple space search fails, the cloud gateway will create a packet-in message as flow setup request, and send it to SDN controller for instructions (line 29–30).

Algorithm 1: elastically accelerated flow table lookup algorithm

```
1  Function PacketClassify(Packet pkt)
       Input: A packet p arrived at software-defined cloud
              gateway
       Output: Flow table lookup result
2      fid ← ParsePacket(pkt);
3      ce ← CacheLookup(fid);
4      if ce ≠ NULL then
5          fe ← GetFlowEntry(ce.addr);
6          if fid == fe → fid then
7              ExecuteActions(fe.actions, pkt);
8              UpdateFlow(fe, pkt);
9              ce.time ← Normalize(pkt.time);
10             return True;
11     for i ← 1 to TUPLE_NUM do
12         key ← pkt.fid&tuples[i].mask;
13         for fe in tuples[i] do
14             if fe.key == key then
15                 ExecuteActions(fe.actions, pkt);
16                 UpdateFlow(fe, pkt);
17                 break;
18             end
19         end
20         if fe ≠ NULL then
21             if fid == fe.latest_fid ∧ pkt.time − fe.time ≤ PIT
                then
22                 f ← NewFlow(fid, idx, p.time);
23                 CacheInsert(f);
24             return True;
25         end
26     end
27     msg ← CreateMessage(pkt);
28     SendMessage(msg);
29     return False;
```

5.2. Algorithmic performance

The average search length is a primary metric to characterize the performance of our proposed elastically accelerated lookup method of virtual SDN flow tables OFT-EAC. Thus we theoretically derive the average search length of our proposed method OFT-EAC and contrast it with that of state-of-the-art flow table lookup methods. For simplicity, we assume that each packet only matches a flow entry in the flow table at most. Suppose that the flow table contains T tuples, and each tuple has the load factor β. For our proposed method OFT-EAC, each arrived packet first performs parallel lookup on the EAC cache with the search length l. If the lookup succeeds, we locate the corresponding flow entry indicated by the cache entry with the total search length 2. Otherwise, we have to perform the tuple space search on the flow table, with the average search length expressed as $ASL_{cache-miss}$. Then, we can calculate the average search length of our proposed method $ASL_{OFT-EAC}$ in (4), with the hit rates of the EAC cache CHR_{EAC} in (2).

$$ASL_{OFT-EAC} = 2CHR_{EAC} + (1 - CHR_{EAC})ASL_{cache-miss}. \tag{4}$$

As for cache lookup failure, the packet has to traverse all tuples one by one until successfully matching a flow entry in any tuple. Suppose all tuples share equal probability of successful matching, we can infer that each packet will perform failed lookup of $(T-1)/2$ tuples on average. As for each failed tuple lookup, we need to travel through the tuple with the search length β. As for final successful lookup on a tuple, it takes the search length $\beta/2+1$ to locate a flow entry in the tuple. With the length of each cache lookup 1, we further deduce the average search length of the flow table for each failed cache lookup $ASL_{cache-miss}$ in (5).

$$ASL_{cache-miss} = 1 + \frac{T-1}{2}\beta + \frac{\beta}{2} + 1 = \frac{T\beta}{2} + 2. \tag{5}$$

By substituting CHR_{EAC} and $ASL_{cache-miss}$ in (4) respectively with (2) and (5), we derive the average search length of our proposed flow table lookup method ASLOFT-EAC in (6).

$$ASL_{OFT-EAC} = 2 + \frac{T\beta}{2}\left(1 - \frac{\sum_{r=1}^{kl} r^{-\alpha}}{\Gamma_{\alpha}(N)}\right). \tag{6}$$

The popular virtual SDN switch, Open vSwitch [27], designs the MicroFlow cache holding recently arrived exact flows in kernel space, to accelerate the lookup on virtual SDN flow tables. For each arrived packet, the switch divides its 32-bit flow identifier into 4 parts to locate 4 positions in the cache, and parallelly matches with cache entries in them with the search length 1. If successfully matching a cache entry, the switch will retrieve a mask in it, and perform lookup on the corresponding tuple with the search length $\beta/2+1$. Thus we can get the search length for cache hit as $\beta/2+2$. If the cache misses, the switch will perform tuple space search with the average search length identical to (5). Setting the hit rate of the MicroFlow cache as p, we can deduce the average search length of the flow table lookup method in Open vSwitch named as OFT-MicroFlow in (7).

$$ASL_{OFT-MicroFlow} = \frac{T\beta}{2} - \frac{1}{2}p\beta(T-1) + 2. \tag{7}$$

According to (6) and (7), we estimate the average search length and the speedup ratios of the above two lookup methods of virtual SDN flow tables in Table 2, with the number of tuples T as 16 and the segment length l of the EAC cache as $3K$. As seen from Table 2, our proposed method OFT-EAC has much shorter average search length than the OFT-MicroFlow, with average speedup ratio around 1.64. Furthermore, the average search length of our proposed OFT-EAC grows in relatively slow way while that of the OFT-MicroFlow rapidly increases, when the number of exact flows doubles. This is attributed to the fact that our proposed EAC cache can keep high cache hit rates, which implies that most packets directly hit the cache without looking up the flow table.

6. Experiments

6.1. Experimental methodology

According to our flow cache design, all scattered packets and the first two packets of each batch in any flow will miss the cache, and need to proceed with tuple space search on our flow table. Apparently, these packets will be sparsely distributed in the flow from the time dimension, which implies the absence of temporal locality. Nonetheless,

Table 2
The average search length of different flow table lookup methods.

N	α	k	β	p	$ASL_{\text{OF T-EAC}}$	$ASL_{\text{OFT -MicroFlow}}$	Speed-up
100K	0.96	5	2	0.60	5.01	9.00	1.80
100K	0.96	4	2	0.65	5.36	8.25	1.54
100K	0.98	4	2	0.70	5.06	7.50	1.48
100K	0.98	3	2	0.75	5.47	6.75	1.23
200K	0.98	7	4	0.60	8.19	16.00	1.95
200K	0.98	6	4	0.65	8.60	14.50	1.69
200K	1.00	6	4	0.70	7.97	13.00	1.63
200K	1.00	5	4	0.75	8.43	11.50	1.36
400K	1.00	9	8	0.60	14.69	30.00	2.04
400K	1.00	8	8	0.65	15.25	27.00	1.77
400K	1.02	8	8	0.70	13.87	24.00	1.73
400K	1.02	7	8	0.75	14.46	21.00	1.45

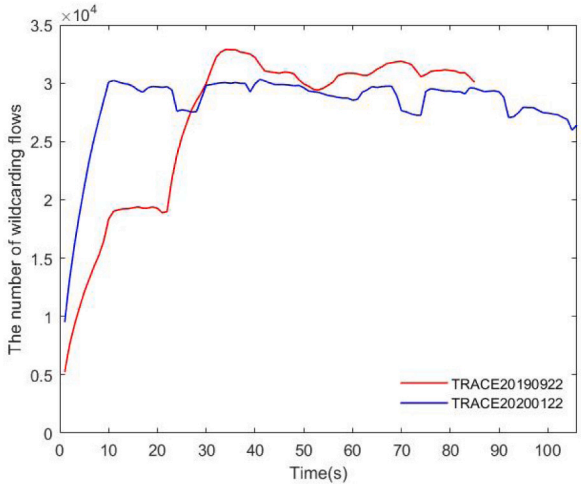
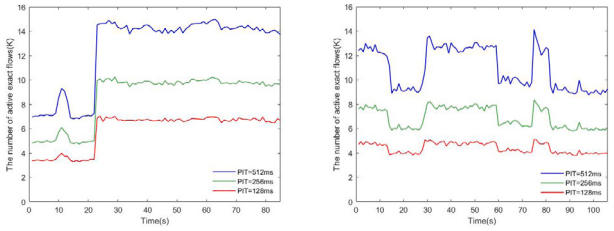


Fig. 9. The number of wildcarding flows in network traffic traces.

spatial locality probably still exists, as each flow is likely to contain a different number of packet batches and scattered packets. Specifically, a majority of packets are expected to gather in a small number of elephant flows, while a large number of mice flows only account for a minority of packets. Consequently, packet traffic for the tuple space search will exhibit the spatial locality in terms of traditional exact flows.

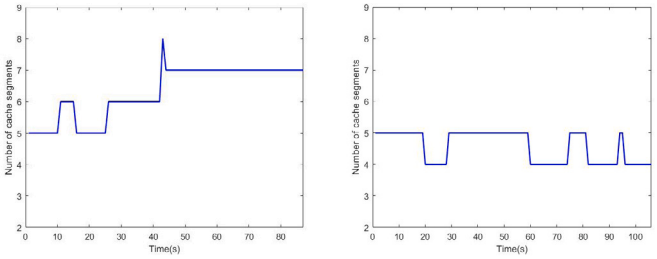
As for network traffic traces, we select TRACE20190922 and TRACE20200122, collected from a 10Gps backbone link at the border of Jiangsu Province in the CERNET (China Education and Research Network). Each traffic trace involves 15,420,235 packets with a sampling ratio of 1:4 [33]. Fig. 9 illustrates the varying number of wildcarding flows in the two traces. As shown in Fig. 9, TRACE20190922 and TRACE20200122 respectively last for approximately 85 s and 106 s, and contain a steady number of wildcarding flows around 31K and 29K after the timeout interval 10 s.

By setting the PIT threshold of active exact flows respectively as 128 ms, 256 ms and 512 ms for the above trace, we get a varying number of active exact flows in Fig. 10. As seen from Fig. 10, the number of active exact flows presents a similar pattern for different PIT thresholds. Specifically, the TRACE20190922 demonstrates a small fluctuation until 23 s, with transient fluctuations between 9 s and 14 s. However, the number of active exact flows experiences a sharp increase and stabilizes at around 10K at 23 s. In contrast, the TRACE20200122 shows frequent and significant fluctuations in the number of active exact flows, with sudden decreases at 14 s, 59 s, and 81 s followed by rebounds at 29 s and 75 s. In short, both traffic traces exhibit momentary fluctuations in the number of active exact flows, which can



(a) TRACE20190922 (b) TRACE20200122

Fig. 10. The number of active exact flows in network traffic traces.



(a) TRACE20190922 (b) TRACE20200122

Fig. 11. The real-time number of segments in the EAC cache.

effectively verify the performance of our proposed EAC cache and the corresponding lookup method of virtual SDN flow tables.

6.2. Cache performance

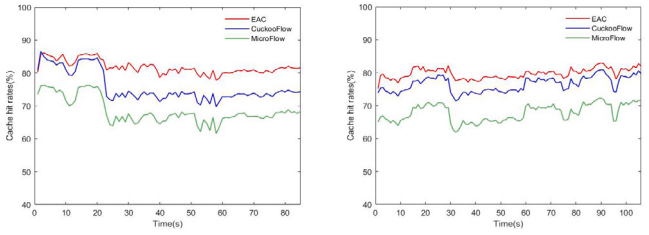
6.2.1. Real-time cache capacity

As for the above network traffic traces, the EAC cache adapts its capacity to the varying number of active exact flows, in accordance with its scaling conditions. In particular, we set the PIT threshold as 256 ms, the length of each segment as 1K, the initial number of segments as 5, the expansion period as 1 s, the lower limits of cache hit rates and cache yield rates respectively as 75% and 10%, and the decrease and increase threshold of its replacement rates for its extensions and retractions respectively as 10.5% and 7.9%. With the above cache parameter settings, we perform flow table lookup on the network traffic trace, and obtain the real-time number of segments in our proposed EAC cache in Fig. 11.

By comparing Fig. 10 with Fig. 11, we can see that the number of segments accordingly increases or decreases for significant rising or falling on the number of active exact flows. This is attributed to the significantly decreased or increased proportion of active exact flows in the EAC cache, which leads to distinct reduction or growth in cache replacement rates beyond its threshold, and triggers the extensions or retractions of the cache. When the number of active exact flows surges, the cache replacement rates rise rapidly and the EAC cache quickly adds its segments to maintain high cache hit rates, such as during 27 s–30 s in Fig. 10(b). When the cache yield rates reach the lower bound, the cache immediately deletes its segments to enhance overall cache performance, such as 41 s in Fig. 11(a). On the whole, the average number of cache segments is respectively 6.30 and 4.57, for the trace TRACE20190922 and TRACE20200122.

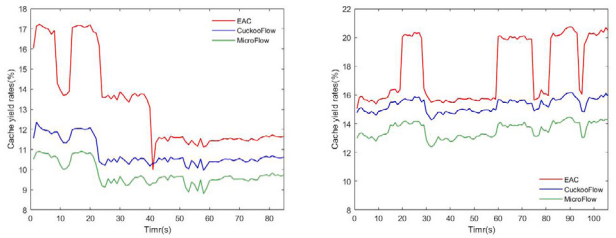
6.2.2. Cache hit rates

We compare the cache hit rates of our proposed EAC cache, CuckooFlow cache [13] and MicroFlow cache [27], with the above network traffic traces. The capacity of the CuckooFlow cache and the MicroFlow cache is suited to be respectively set as 7K and 5K in the light of the average number of segments in the EAC cache in Fig. 11. With the above parameter settings, we respectively operate the EAC cache, the



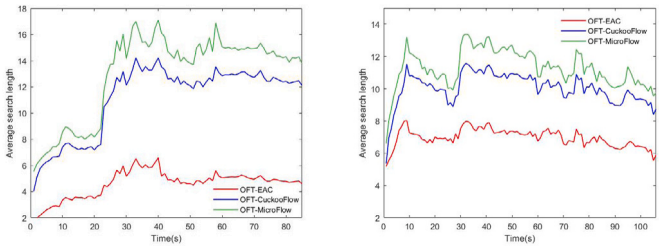
(a) TRACE20190922 (b) TRACE20200122

Fig. 12. The hit rates of different caches.



(a) TRACE20190922 (b) TRACE20200122

Fig. 13. The hit rates of different caches.



(a) TRACE20190922 (b) TRACE20200122

Fig. 14. The average search length of different accelerated lookup methods of virtual SDN flow table.

CuckooFlow cache and the MicroFlow cache on the network traffic in the trace, and get respective cache hit rates in Fig. 12.

As shown in Fig. 12, our proposed EAC cache achieves much higher and more stable hit rates than the CuckooFlow and the MicroFlow, regardless of network traffic traces. For the TRACE20190922 and TRACE20200122, the average cache hit rate of the EAC cache is respectively 81.77% and 79.70%, exhibiting an improvement of 6.09% and 3.27% over the CuckooFlow cache, while 13.29% and 11.89% over the MicroFlow cache. Additionally, the fluctuation of the hit rates of the EAC cache is respectively 8.31% and 7.78%, which is 50.58% and 17.57% lower than the CuckooFlow cache, while 43.64% and 24.72% lower than the MicroFlow cache. These are primarily attributed to the fact that the EAC cache holds active exact flows and its capacity is adaptively adjusted to the dynamic changes in the number of active exact flows. In contrast, both the CuckooFlow and the MicroFlow keep fixed capacity, without effective adaptability to the dynamic changes of network traffic.

6.2.3. Cache yield rates

With the above parameter settings, we perform accelerated lookup methods of virtual SDN flow tables respectively based on the EAC cache, the CuckooFlow and the MicroFlow on each traffic trace, and calculate corresponding cache yield rates in Fig. 13. As seen from Fig. 13, the EAC cache achieves higher cache yield rates than the CuckooFlow and the MicroFlow, regardless of network traffic traces. The EAC cache with a smaller capacity still achieves higher cache hit rates than those of the CuckooFlow and the MicroFlow, which results in much higher yield rates than those of the two caches. As for the EAC cache with a larger capacity, its hit rates are significantly higher than those of the CuckooFlow and the MicroFlow, which gives rise to its higher yield rates.

The above experimental phenomenon can be explained by combinative observations of Figs. 11, 12 and 13. As for the TRACE20190922, our proposed EAC cache only contains 5˜6 segments during its first half (1–40 s), with smaller capacity and higher cache hit rates. This naturally brings about higher cache yield rates than those of the CuckooFlow and the MicroFlow. As for its latter half (after 42 s), the EAC cache stabilizes at about 7 segments, with its capacity equivalent to that of the CuckooFlow and the MicroFlow, but slightly higher cache hit rates. These still result in higher yield rates of the EAC cache. As for the TRACE20200122, the EAC cache keeps steady at 4 or 5 segments, with the capacity no more than that of the CuckooFlow and the MicroFlow, while significantly higher cache hit rates. Hence, the EAC cache achieves much higher yield rates, especially for the case of only 4 segments during 20–28 s, 60–74 s, 82–93 s and 96–106 s.

6.3. Average search length

Average search length is a key performance metric for the lookup methods of virtual SDN flow tables. With the above configurations and the hash length of each tuple set as 2^9, we respectively perform flow table lookup methods OFT-EAC, OFT-CuckooFlow and OFT-MicroFlow on network traffic traces, and calculate their average search length in Fig. 14.

As shown in Fig. 14, our proposed OFT-EAC has much shorter and stable average search length than that of the OFT-CuckooFlow and the OFT-MicroFlow. In Fig. 14, the average search length of the OFT-EAC remains around 4.87 and 6.98 respectively for the TRACE20190922 and the TRACE20200122, after the initial phase (10 s). Its average speedup ratios are respectively 2.43 and 1.45 compared to the OFT-CuckooFlow, while 2.84 and 1.63 in contrast to the OFT-MicroFlow. Moreover, the average search length of the OFT-EAC respectively fluctuates by 3.22 and 2.43, which are 54.15% and 23.39% less than the OFT-CuckooFlow, while 64.57% and 36.67% less than the OFT-MicroFlow. This is attributed to the fact that the EAC cache in the OFT-EAC can adapt to network traffic jitters and always keep high cache hit rates, which means that most of packets will hit the cache without looking up virtual SDN flow tables.

7. Conclusion

Aiming at the problem of unstable packet classification performance of software-defined cloud gateways caused by network traffic jitters, this paper proposes an elastically accelerated lookup method of virtual SDN flow tables. Particularly, we first cache active exact flows by exploiting network traffic locality, which enables most packets to bypass tuple space search, significantly accelerating the lookup of virtual SDN flow tables. Furthermore, the cache adaptively expands or retracts its capacity in accordance with the dynamic changes in the number of active exact flows to consistently maintain high cache hit rates, which achieves elastic acceleration of flow table lookup. Lastly, we evaluate the performance of our proposed flow table lookup method and its elastic accelerating cache with jittered network traffic traces.

The experimental results indicate that our proposed flow table lookup method outperforms the OFT-CuckooFlow and the OFT-MicroFlow in terms of cache hit rates, cache yield rates and average search length. As for network traffic with significant fluctuation, our proposed elastic accelerating cache stably achieves hit rates around 80%, approximately 6.09% and 13.29% higher than those of the CuckooFlow and the MicroFlow respectively. Moreover, our proposed

elastic accelerating cache achieves much higher yield rates, especially for the case of small number of cache segments. Finally, our proposed flow table lookup method achieves the speedup ratios of average search length about 2.43 and 2.84, and its fluctuation reduced by 54.15% and 64.57%, respectively compared to the OFT-CuckooFlow and the OFT-MicroFlow. In conclusion, our proposed lookup method of virtual SDN flow tables significantly promotes the efficiency and robustness of packet classification in software-defined cloud gateways.

In our future work, more network traffic traces will be collected from different network scenarios to verify the effectiveness and applicability of our proposed elastically accelerated lookup method of virtual SDN flow tables. In the future, we will implement and integrate the method into popular virtual SDN switches, such as Open vSwitch. Furthermore, we also plan to apply our proposed elastic accelerating cache to other flow-based devices and systems, to address other issues including the energy consumption of TCAM flow table lookup and the measurement precision of elephant flows.

CRediT authorship contribution statement

Bing Xiong: Methodology, Writing – original draft, Writing – review & editing, Project administration. **Jing Wu:** Writing – review & editing, Visualization. **Qiaorong Huang:** Writing – original draft, Software. **Jinyuan Zhao:** Investigation, Formal analysis. **Qiang Tang:** Validation. **Jin Zhang:** Resources, Supervision. **Kun Yang:** Conceptualization, Methodology. **Keqin Li:** Conceptualization, Theoretical guidance.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Bing Xiong reports financial support was provided by National Natural Science Foundation of China. Bing Xiong reports financial support was provided by Hunan Provincial Natural Science Foundation of China. Bing Xiong reports financial support was provided by Scientific Research Fund of Hunan Provincial Education Department. Jing Wu reports financial support was provided by Postgraduate Scientific Research Innovation Project of Hunan Province. Bing Xiong, Qiaorong Huang has patent #ZL202111110409.5 licensed to Changsha University of Science and Technology.

Data availability

The authors do not have permission to share data.

Acknowledgments

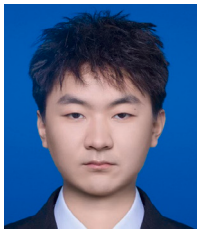
This work was supported in part by National Natural Science Foundation of China (62272062), Hunan Provincial Natural Science Foundation of China (2023JJ30053), Scientific Research Fund of Hunan Provincial Education Department, China (22A0232, 22B0300), and Postgraduate Scientific Research Innovation Project of Hunan Province, China (CX20230913).

References

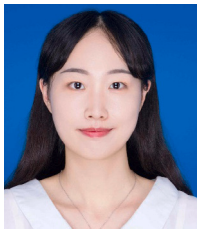
[1] M.H. Zhang, J. Bi, K. Gao, et al., Tripod: Towards a scalable, efficient and resilient cloud gateway, IEEE J. Sel. Areas Commun. 37 (3) (2019) 570–585.
[2] Sood M. Nishtha, A survey on issues of concern in software defined networks, in: 3rd International Conference on Image Information Processing, ICIIP, Waknaghat, India, 2015, pp. 295–300.
[3] T. Pan, N.B. Yu, C.H. Jia, et al., Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches, in: ACM Conference o Special Interest Group on Data Communication, SIGCOMM, New York, USA, 2021, pp. 194–206.
[4] S. Natarajan, A. Ramaiah, M. Mathen, A software defined cloud-gateway automation system using OpenFlow, in: 2nd IEEE International Conference on Cloud Networking, CloudNet, San Francisco, USA, 2014, pp. 219–226.
[5] S. Venkatachary, S. sub hash, V. George, Packet classification using tuple space search, in: ACM Conference on Special Interest Group on Data Communication, SIGCOMM, Cambridge, USA, 1999, pp. 135–146.
[6] D. Tang, Y.D. Yan, S.Q. Zhang, et al., Performance and features: Mitigating the low-rate TCP-targeted doS attack via SDN, IEEE J. Sel. Areas Commun. 40 (1) (2022) 428–444.
[7] D. Firestone, A. Putnam, S. Mundkur, et al., Azure accelerated networking: SmartNICs in the public cloud, in: 15th USENIX Symposium on Networked Systems Design and Implementation, NSDI, Renton, USA., 2018, pp. 50–64.
[8] P. Gao, Y. Xu, H.J. Chao, OVS-CAB: Efficient rule-caching for open vSwitch hardware offloading, Comput. Netw. 188 (2) (2021) 1–14.
[9] S. Singh, F. Baboescu, G. Varghese, et al., Packet classification using multidimensional cutting, in: ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM, New York, USA, 2003, pp. 213–224.
[10] S. Yingchareonthawornchai, J. Daly, A.X. Liu, et al., A sorted partitioning approach to high-speed and fast-update OpenFlow classification, in: 24th IEEE International Conference on Network Protocols, ICNP, Singapore, 2016, pp. 1–10.
[11] S. Yingchareonthawornchai, J. Daly, A.X. Liu, et al., A sorted-partitioning approach to fast and scalable dynamic packet classification, IEEE/ACM Trans. Netw. 26 (4) (2018) 1907–1920.
[12] W. Li, T. Yang, O. Rottenstreich, et al., Tuple space assisted packet classification with high performance on both search and update, IEEE J. Sel. Areas Commun. 38 (7) (2020) 1555–1569.
[13] B. Xiong, Z. Hu, Y. Luo, et al., CuckooFlow: Achieving fast packet classification for virtual OpenFlow switching by exploiting network traffic locality, in: 17th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA, Xiamen, China, 2019, pp. 1071–1078.
[14] D. Zhou, H. Yu, M. Kaminsky, et al., Fast software cache design for network appliances, in: 2020 USENIX Annual Technical Conference, USENIX ATC, Boston, USA, 2020, pp. 657–671.
[15] D. Tang, S. Wang, B. Liu, et al., GASF-IPP: Detection and mitigation of LDoS attack in SDN, IEEE Trans. Serv. Comput. (2023) 1–12, (early access).
[16] L. Molnár, G. Pongrácz, G. Enyedi, et al., Dataplane specialization for high-performance OpenFlow software switching, in: ACM Conference on Special Interest Group on Data Communication, SIGCOMM, Florianopolis, Brazil, 2016, pp. 539–552.
[17] S. Miano, F. Risso, H. Woesner, Partial offloading of OpenFlow rules on a traditional hardware switch ASIC, in: IEEE Conference on Network Softwarization, NetSoft, Bologna, Italy, 2017, pp. 1–9.
[18] M. Varvello, R. Laufer, F. Zhang, et al., Multilayer packet classification with graphics processing units, IEEE/ACM Trans. Netw. 24 (5) (2016) 2728–2741.
[19] W.W. Fu, T. Liu, Z.G. Sun, FAS: Using FPGA to accelerate and secure SDN software switches, Secur. Commun. Netw. 2018 (2018) 1–13.
[20] P. Gupta, N. Mckeown, Classifying packets with hierarchical intelligent cuttings, IEEE Micro 20 (1) (2000) 1–9.
[21] B. Vamanan, G. Voskuilen, T.N. Vijaykumar, EffiCuts: Optimizing packet classification for memory and throughput, ACM SIGCOMM Comput. Commun. Rev. 40 (4) (2010) 207–218.
[22] W. Li, X. Li, H. Li, et al., CutSplit: A decision-tree combining cutting and splitting for scalable packet classification, in: IEEE Conference on Computer Communications, INFOCOM, Honolulu, USA, 2018, pp. 2645–2653.
[23] J. Daly, E. Torng, ByteCuts: Fast packet classification by interior bit extraction, in: IEEE Conference on Computer Communications, INFOCOM, Honolulu, USA, 2018, pp. 2654–2662.
[24] D. James, B. Valerio, L. Leonardo, et al., TupleMerge: Fast software packet processing for online packet classification, IEEE/ACM Trans. Netw. 27 (4) (2019) 1–15.
[25] D. Tang, S. Zhang, Y. Yan, et al., Real-time detection and mitigation of LDoS attacks in the SDN using the HGB-FP algorithm, IEEE Trans. Serv. Comput. 15 (6) (2022) 3471–3484.
[26] P.T. Congdon, P. Mohapatra, M. Farrens, et al., Simultaneously reducing latency and power consumption in OpenFlow switches, IEEE/ACM Trans. Netw. 22 (3) (2014) 1007–1020.
[27] B. Pfaff, J. Pettit, T. Koponen, et al., The design and implementation of open vSwitch, in: 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI, Oakland, USA, 2015, pp. 2–16.
[28] Y. Wang, T.Y.C. Tai, R. Wang, et al., Optimizing open vSwitch to support millions of flows, in: IEEE Global Communications Conference, GlobeCom, Singapore, 2017, pp. 1–7.
[29] W. Mao, Z. Shen, X. Huang, Facilitating network functions virtu-alization by exploring locality in network traffic, in: 2nd International Conference on Computer Science and Artificial Intelligence, CSAI, New York, USA, 2018, pp. 495–499.
[30] J. Wallerich, H. Dreger, A. Feldmann, et al., A methodology for studying persistency aspects of internet flows, ACM SIGCOMM Comput. Commun. Rev. 35 (2) (2005) 23–36.
[31] Y. Wan, H. Song, Y. Xu, et al., T-cache: Dependency-free ternary rule cache for policy-based forwarding, in: IEEE Conference on Computer Communications, INFOCOM, Toronto, Canada, 2020, pp. 536–545.

[32] B. Xiong, R. Wu, J. Zhao, et al., Efficient differentiated storage architecture for large-scale flow tables in software-defined wide-area networks, IEEE Access 7 (2019) 141193–141208.
[33] Network traffic traces, http://iptas.edu.cn/src/system.php.
[34] R.B. Basat, X.Q. Chen, G. Einziger, et al., Randomized admission policy for efficient top-k, frequency, and volume estimation, IEEE/ACM Trans. Netw. 27 (4) (2019) 1432–1445.
[35] J.Y. Campbell, R.J. Shiller, Yield spreads and interest rate movements: A bird's eye view, Rev. Econ. Stud. 58 (3) (1991) 495–513.
[36] P.R. Yang, Using the yield curve to forecast economic growth, J. Forecast. 39 (7) (2020) 1057–1080.

Bing Xiong received the Ph.D. degree in Computer Science by master-doctorate program from Huazhong University of Science and Technology (HUST), China, in 2009, and the B.S. degree from Hubei Normal University, China, in 2004. He worked as a visiting scholar in the Department of Computer and Information Science, Temple University, USA, from 2018 to 2019. He is currently an associate professor in the School of Computer and Communication Engineering, Changsha University of Science and Technology, China. His main research interests include future network architecture, network measurements, and digital twins.



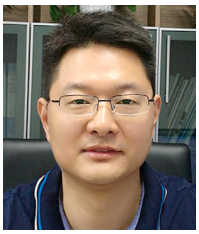
Jing Wu received the B.S. degree in Computer Science and Technology from Changsha University of Science and Technology, China, in 2022. He is currently pursuing the M.S. degree in the School of Computer and Communication Engineering, Changsha University of Science and Technology, China. His main research interests include software-defined networking, network virtualization and packet classification.



Qiaorong Huang received the B.S. degree in Software Engineering from Changsha University of Science and Technology, China, in 2020. She is currently pursuing the M.S. degree in the School of Computer and Communication Engineering, Changsha University of Science and Technology, China. Her main research interests include software-defined networking, network virtualization and packet classification.

Jinyuan Zhao, received the Ph.D. degree in Computer Science from Central South University, China, in 2020, and the M.S. degree from Central China Normal University, China, in 2007. She worked in the School of Computer and Communication, Hunan Institute of Engineering, China, from 2007 to 2020. She is currently an assistant professor in the School of Information Science and Engineering, Changsha Normal University, China. Her main research interests include future network architecture, network measurements.

Qiang Tang received the BE, ME, and Ph.D. degrees from the Department of Control Science and Engineering from Huazhong University of Science and Technology, Wuhan, China, in 2005, 2007, and 2010, respectively. He was an academic visitor sponsored by CSC in University of Essex during 2016–2017. He is currently a lecturer with the School of Computer and Communication Engineering, Changsha University of Science and Technology, Changsha, China. His research interests include vehicle network resource optimization, mobile edge computing, smart grid, and wireless sensor network.



Jin Zhang received the B.S. degree in communication engineering and the M.S. degree in computer application from Hunan University, Changsha, China, in 2002 and 2004, respectively, and the Ph.D. degree in biomedical engineering from Zhejiang University, Hangzhou, China, in 2007. He has been a Professor with Changsha University of Science and Technology since 2021. From 2008 to 2009, he worked as an Associate Professor with the Hunan University, Changsha, China. From 2009 to 2011, he worked as a Postdoctoral Fellow with the Beijing Normal University, Beijing, China. From 2012 to 2013, he worked as a Postdoctoral Fellow with the University of Chicago, Chicago, IL, USA. From 2014 to 2021, he has been a Professor with Hunan Normal University, Changsha, China. His research interests include computer network, software engineering, and artificial intelligence.

Kun Yang received his Ph.D. from the Department of Electronic & Electrical Engineering of University College London (UCL), UK. He is currently a Chair Professor in the School of Computer Science & Electronic Engineering, University of Essex, leading the Network Convergence Laboratory (NCL), UK. He is also an affiliated professor at UESTC, China. Before joining in the University of Essex at 2003, he worked at UCL on several European Union (EU) research projects for several years. His main research interests include wireless networks and communications, IoT networking, data and energy integrated networks and mobile computing. He manages research projects funded by various sources such as UK EPSRC, EU FP7/H2020 and industries. He has published 400+ papers and filed 30 patents. He serves on the editorial boards of both IEEE (e.g., IEEE TNSE, IEEE ComMag, IEEE WCL) and non-IEEE journals (e.g., Deputy EiC of IET Smart Cities). He was an IEEE ComSoc Distinguished Lecturer (2020–2021). He is a Member of Academia Europaea (MAE), a Fellow of IEEE, a Fellow of IET and a Distinguished Member of ACM.

Keqin Li is a SUNY Distinguished Professor of Computer Science with the State University of New York. He is also a National Distinguished Professor with Hunan University, China. His current research interests include cloud computing, fog computing and mobile edge computing, energyefficient computing and communication, embedded systems and cyber–physical systems, heterogeneous computing systems, big data computing, high-performance computing, CPU–GPU hybrid and cooperative computing, computer architectures and systems, computer networking, machine learning, intelligent and soft computing. He has authored or coauthored over 900 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He holds nearly 70 patents announced or authorized by the Chinese National Intellectual Property Administration. He is among the world's top 5 most influential scientists in parallel and distributed computing in terms of both single-year impact and career-long impact based on a composite indicator of Scopus citation database. He has chaired many international conferences. He is currently an associate editor of the ACM Computing Surveys and the CCF Transactions on High Performance Computing. He has served on the editorial boards of the IEEE Transactions on Parallel and Distributed Systems, the IEEE Transactions on Computers, the IEEE Transactions on Cloud Computing, the IEEE Transactions on Services Computing, and the IEEE Transactions on Sustainable Computing. He is an AAAS Fellow, an IEEE Fellow, and an AAIA Fellow. He is also a Member of Academia Europaea (Academician of the Academy of Europe).