Journal of Network and Computer Applications xxx (xxxx) xxx

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

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

Research paper

FastTSS: Accelerating tuple space search for fast packet classification in virtual SDN switches

Bing Xiong [a], Jing Wu [a], Guanglong Hu [a], Jin Zhang [a,*], Baokang Zhao [b,*], Keqin Li [c,*]

[a] School of Computer and Communication Engineering, Changsha University of Science and Technology, Changsha 410114, PR China
[b] School of Computer Science, National University of Defense Technology, Changsha 410073, PR China
[c] Department of Computer Science, State University of New York at New Paltz, New York 12561, USA

ARTICLE INFO

Keywords:
Dynamic tuple sorting
Network traffic locality
Packet classification
Virtual SDN switches
Well-exploited flow cache

ABSTRACT

The increasing tendency of network virtualization gives rise to extensive deployments of virtual switches in various virtualized platforms. However, virtual switches are encountered with severe performance bottlenecks with regards to their packet classification especially in the paradigm of Software-Defined Networking (SDN). This paper is thus motivated to design a fast packet classification scheme based on accelerated tuple space search, named as FastTSS, for virtual SDN switches. In particular, we devise a well-exploited cache with active exact flows to directly retrieve respective flow entries for most incoming packets, in virtue of the temporal locality of network traffic. Furthermore, we propose a novel hash algorithm to resolve the hash collisions of the cache, by providing three candidate locations for each inserted flow and making room for conflicting flow through kicking operation. As for the case of cache miss, we utilize the spatial locality of packet traffic to accelerate tuple space search, by dynamically sorting all tuples in terms of their reference frequencies and load factors. Eventually, we evaluate our designed packet classification scheme with physical network traffic traces by experiments. Extensive experimental results indicate that our designed FastTSS scheme outperforms the state-of-the-art ones with stable cache hit rates around 85% and the speedup of average search length up to 2.3, significantly promoting the packet classification performance of virtual SDN switches.

1. Introduction

Nowadays, existing Internet architecture is restricted to simple limited updates and difficult to deploy novel network technologies. To break this restriction, network virtualization is proposed to separate network services from hardware infrastructure and enable multiple different network architectures to coexist on a shared physical substrate. Due to its high flexibility, manageability and security, it is generally considered as a promising solution to eradicate the Internet ossification in the long term (Chowdhury and Boutaba, 2010). With the rapid development of network virtualization, virtual switches have been widely deployed in a variety of virtualized environments in recent years. Particularly, virtual switches, functionally equivalent to physical switches, can provide logical network functions and services for virtual machines in various cloud platforms. Meanwhile, Software-Defined Networking (SDN) has become a prevalent paradigm for modern large-scale networks in recent years. Until now, virtual SDN switches have become a prevalent provider of network services across various virtualized environments, particularly within cloud data centers (Yi et al., 2018).

In order to flexibly define packet flows of different granularity, SDN incorporates wildcards into the match fields of its flow tables, chiefly comprised of key fields in the headers of protocol layers (Mckeown et al., 2008). Nevertheless, this prevents virtual SDN flow tables from directly employing hashing techniques to support fast lookup. To address this problem, virtual SDN switches generally adopt tuple space search (TSS) to implement wildcarding. It partitions all flow rules into several tuples according to the mask that identifies the position of wildcards in the match fields. Then, each tuple is managed as a hash table, and searched in terms of its unique mask and the match fields in each flow entry. As for an arrived packet, it is unaware of its mask, and unable to directly retrieve its respective tuple in the flow table. Therefore, each packet must look up all tuples in turn until a successful lookup. This brings high overheads regarding flow table lookups and imposes a great penalty on packet classification performance. 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 (Tang et al., 2022a). Thus, it is crucial to expedite packet classification in virtual SDN switches (Emmerich et al., 2018).

* Corresponding authors.
E-mail addresses: jzhang@csust.edu.cn (J. Zhang), bkzhao@nudt.edu.cn (B. Zhao), lik@newpaltz.edu (K. Li).

https://doi.org/10.1016/j.jnca.2025.104112
Received 24 September 2024; Received in revised form 21 December 2024; Accepted 14 January 2025
Available online 22 January 2025
1084-8045/© 2025 Elsevier Ltd. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

To date, considerable effort has been devoted to enhancing packet classification performance for virtual SDN switches. These work is essentially classified into three categories: optimizing packet switching architectures, enhancing packet classification algorithms, and employing caching techniques. Initially, significant advancements in packet classification performance were made by offloading packet switching from CPU to programmable NICs as fast data path (Luo et al., 2010; Firestone, 2017). Subsequently, some researchers optimized packet classification algorithms by speeding up the update of decision trees supporting fast lookups (He et al., 2014; Li et al., 2018), and decreasing the lookup overheads of flow tables with fast updates (Daly and Torng, 2017; Daly et al., 2019). In proximity to our research, cache techniques were employed to directly locate corresponding flow entries for incoming packets by leveraging temporal locality in network traffic (Pfaff et al., 2015; Wang et al., 2017). But there is still room for improvement primarily due to their insufficient utilization of network traffic locality, which gives rise to inadequate cache utilization and unsatisfactory cache hit rates.

This paper concentrates on how to further accelerate packet classification by adequately exploiting network traffic locality. To this end, we have an insight into the temporal and spatial locality of network traffic in terms of packet flows in virtual SDN switches. Afterwards, the temporal locality is leveraged to optimize the flow cache design of SDN flow tables, by caching presently active exact flows rather than newly emerged ones. Furthermore, we devise a Well-Exploited Flow cache called WEFcache, by proposing a novel hash algorithm to resolve hash collisions. Meanwhile, the spatial locality is utilized to expedite the tuple space search of the flow table by dynamically sorting its all tuples for the case of cache miss. By this way, we achieve a fast packet classification scheme named FastTSS for virtual SDN switches. The key contributions of this paper are summarized as follows.

- Taking an insight into network traffic locality from the perspective of packet flows in the paradigm of SDN both in time and space. The temporal and spatial locality of network traffic is respectively exploited to increase the hit rates of flow caches and decrease the overhead of tuple space search in respective flow tables.
- Designing a well-exploited flow cache to achieve high cache hit rates, by caching currently active exact flows for more cache-hit packets, and devising a novel hash algorithm which provides three candidate locations for each inserted flow and makes room for conflicting flow through kicking operation, to resolve hash collisions for higher cache utilization.
- Accelerating the tuple space search of the respective flow table for cache-miss packets, by dynamically sorting all tuples in terms of their reference frequencies and load factors, and formulating the average number of failed tuple lookups with the geometric progression assumption of tuple reference probabilities.
- Calculating the average search length of our proposed packet classification scheme FastTSS with the cache hit rate and average number of failed tuple lookups, and verifying the performance superiority of the FastTSS scheme by experiments with physical network traffic traces.

The rest of this paper is organized as follows. Section 2 introduces related work. In Section 3, we introduce the background of virtual SDN switches especially its flow caches/tables. Section 4 depicts our proposed packet classification scheme FastTSS, which optimizes flow cache design and tuple space search, respectively in virtue of temporal and spatial locality of packet traffic. In Section 5, we provide the algorithmic implementations of our proposed FastTSS scheme, and analyze its average search length. Section 6 evaluates the packet classification performance of the FastTSS scheme, in terms of cache hit rate and average search length. Eventually, we conclude the paper in Section 7.

2. Related work

In recent years, optimization techniques have been extensively studied to improve packet classification performance for virtual SDN switches. Initially, some researchers strived to promote software packet classification performance, by developing novel frameworks for packet I/O on general-purpose operating systems (Rizzo et al., 2012), leveraging most advanced multi-core CPUs and OS technology (Nakajima et al., 2014), and employing template-based code generation to compile any OpenFlow pipeline into efficient machine code as a fast data path (Molnár et al., 2016). With the advent of programmable NICs (PNIC), many developers set out to offload virtual SDN switches from host CPU to PNICs (Luo et al., 2010; Firestone, 2017), and implemented packet classification based on Intel's Data Plane Development Kit (DPDK) (Rahimi et al., 2016; Pongrácz et al., 2013). This achieves high programmability of SDN data plane and speeds up packet classification in virtual SDN switches. Moreover, some work further promoted packet classification performance by employing specialized hardware such as network processors (Blaiech et al., 2014), GPU (Varvello et al., 2016), and FPGA (Firestone et al., 2018).

Substantial progress have been achieved through the algorithmic optimization on packet classification in SDN switches. Many decision trees can achieve high-speed classification, such as HiCuts (Gupta and McKeown, 2000), HyperCuts (Singh et al., 2003), SmartSplit (He et al., 2014) and CutSplit (Li et al., 2018). Unfortunately, it will generate heavy overheads to update decision trees, which needs the reconstruction of trees through a battery of adjustments. On the contrary, tuple space search (TSS) (Srinivasan et al., 1999) is utilized to facilitate fast updates, by partitioning a large number of flow rules into a smaller number of tuples based on the positions of wildcards in their match fields. However, TSS performs lookups with great overheads especially in scenarios with a great many tuples. To enhance TSS, Daly et al. Daly and Torng (2017)Daly et al. (2019) further designed a novel packet classification algorithm TupleMerge, which merges flow rules by ignoring some bits to cut down the number of tuples. This greatly promotes classification performance while keeping similar update speeds. To incorporate the advantages of decision trees and TSS, Yingchareonthawornchai et al. (2016)Yingchareonthawornchai et al. (2018) designed an integrated approach PartitionSort, which utilizes ruleset sortability to achieve a reduced number of partitions and employs multi-dimensional interval trees for fast lookup and updates within each partition. Li et al. (2020) presented a two-stage scheme CutTSS, which initially builds partial decision trees from various rule subsets classified based on their small fields, and utilizes TSS to perform packet classification at non-leaf terminal nodes. But the classification performance will be degraded with unbearable sorting overheads for the case of too many rulesets.

The caching techniques have been extensively employed to improve packet classification performance in virtual SDN switches. Congdon et al. (2014) established a prediction circuitry for arrived packets in virtue of the temporal locality of network traffic to reduce latency and power consumption simultaneously. Pfaff et al. developed a popular virtual SDN switch, Open vSwitch, widely deployed in virtualized platforms. It first cached mega-flows without priority and overlapping as a kernel-mode flow table, which enables most packets to pass fast kernel data-path (Pfaff et al., 2009). Subsequently, it cached micro-flows in the form of connections or sessions in its mega-flow table, which further enhances packet classification performance (Pfaff et al., 2015). Additionally, Wang et al. (2017) Wang et al. (2018) designed a tuple mapping cache CuckooDistributor to directly retrieve tuples for incoming packets, for decreasing the overhead of tuple space search within the mega-flow table. Furthermore, they dynamically regulated the speed of inserting new flows into the micro-flow cache, in accordance with its recent miss rate, for improving its hit rate. To further optimize the flow cache, Zhou et al. (2020) employed bounded linear probing to address its hash collisions, and devised probabilistic

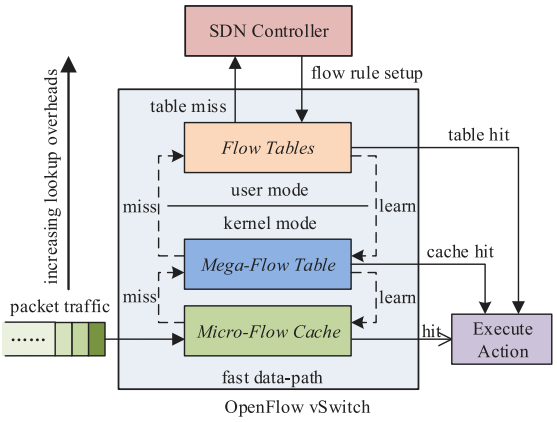


Fig. 1. Packet classification in Open vSwitch.

bubble LRU to facilitate its replacements, for sufficient cache utilization. Rashelbach et al. (2022)Rashelbach et al. (2023) applied the neural network algorithm NuevoMatch for packet classification, which was used as the front-cache of mega-flow cache or completely replaced the OVS datapath, to improve the OVS packet forwarding performance. But the above flow caches accommodate arbitrary sets of exact flows, and their hit rates can be further elevated.

To break the limitation of above flow caches, we previously devised an active-exact-flow cache by applying Cuckoo hashing, allowing a majority of packets to directly retrieve their corresponding flow entries (Xiong et al., 2019). However, cuckoo hashing needs additional hash computation to locate another candidate position from directly mapped position, for both its lookup and insertion operations. More seriously, its insertion operations will produce cyclic kicking due to mutual indexing of its two candidate positions, especially when cache utilization rates reach a certain value typically 60% (Duffield et al., 2005). This gives rise to inadequate cache utilization and unsatisfactory cache hit rates, especially under traffic jitters even cyber attacks (Tang et al., 2023). To address these problems, we propose a novel hash algorithm to resolve the hash collisions of the cache, by providing three candidate locations for each inserted flow and making room for conflicting flow through kicking operation. Furthermore, we accelerate tuple space search in the case of cache miss, by keeping all tuples in order of their reference frequencies and load factors. Through these improvements, it is expected to significantly promote packet classification performance in virtual SDN switches.

3. Background

Virtual SDN switches are commonly installed on general-purpose hardware platforms without ternary content addressable memories (TCAM) supporting wildcarding, and their flow tables must run in random access memories (RAM). Thus, virtual SDN flow tables can no longer apply classical fast lookup algorithms such as hash tables, owing to the introduction of wildcards into their match fields. To address this problem, a mask is designed for each flow entry to indicate the positions of wildcards in the match fields, similarly to the subset mask of an IP address. Subsequently, all entries in a flow table are classified into a small number of tuples in terms of their masks. Each tuple can be looked up in a hash table by the bitwise AND of its unique mask and the match fields in each flow entry. These tuples compose a tuple space, which will be searched for a matched flow entry on each packet arrival. This solution is called tuple space search (TSS) (Srinivasan et al., 1999).

According to the above principle, the TSS solution will produce heavy lookup overheads of a flow table, especially with a great number of tuples. Upon receiving a packet, the switch is unaware of its mask, and unable to determine its corresponding tuple in the flow table. Hence each packet requires to match against all tuples individually, until a flow entry is successfully identified. This means that it has to go through failed lookup over many tuples until a successful match. When it comes to a flow table with abundant tuples, the tuple space search will implement flow table lookups with great overheads, and brings a performance bottleneck to packet classification. To mitigate this bottleneck, caching techniques are extensively applied to speed up the tuple space search for fast packet classification. Particularly, some most advanced virtual switches such as Open vSwitch (Pfaff et al., 2015) designed multiple levels of flow caches/tables in Fig. 1, to provide fast classification paths for incoming packets.

As shown in Fig. 1, the switch first picks out disjoint flows without a priority from flow tables in user space and caches them as a mega-flow table in its kernel module. The mega-flow table is implemented in the form of tuple space search, but supports much faster lookups than flow tables in user space for two reasons: (a) its tuple space search will terminate in the case of a successful match; (b) there is only one classifier, in contrast to a pipeline of them requiring a long series of flow table lookups in user space. The micro-flow cache maps traditional flows (typically TCP connections) into the masks of their belonging mega-flows. This allows radical simplification of its implementation as a plain hash table, and requires only one time of hash table lookup for an incoming packet. Unfortunately, the cache hit rate will be sharply decreased for a considerable number of short lived connections in the case of network traffic jitters and cyber attacks (Tang et al., 2022b). Hence it is requisite to optimize the micro-flow cache and the mega-flow table for fast packet classification in virtual SDN switches. For simplicity, flow table and flow cache are respectively short for mega-flow table and micro-flow cache in the rest of this paper.

4. Fast packet classification based on network traffic locality

This section builds an efficient packet classification scheme based on network traffic locality, which designs a well-exploited flow cache with low hash collisions and expedites tuple space search by dynamically sorting all tuples.

4.1. Packet classification scheme

Massive network investigations indicate that packet traffic shows evident locality properties both in time (temporal locality) and space (spatial locality) (Duffield et al., 2005; Williamson, 2001). Network traffic locality brings about the reference locality of tuples and flow entries in flow tables, which inspires us to further accelerate packet classification for virtual SDN switches. In particular, we devise a well-exploited flow cache in virtue of the temporal locality of network traffic. The cache accommodates currently active exact flows, other than recently emerged ones, for more incoming packets to hit the rate. Furthermore, we leverage the spatial locality of cache-miss packet traffic, by dynamically sorting all tuples in the flow table to reduce average tuple space search overheads. Fig. 2 illustrates the fundamental principle of our proposed packet classification scheme FastTSS for virtual SDN switches.

As for an arrived packet, the switch first extracts its flow identifier, and queries the well-exploited flow cache before its flow table lookup. If the cache hits, we will obtain a cache entry to directly locate the respective flow entry in the flow table, bypassing expensive tuple space search. Once the cache misses, it needs to perform tuple space search on the flow table, by looking up all tuples one by one with their associated masks. The tuple space search will continue before a successful match in any tuple. As for a matched tuple, we adaptively adjust its position in the tuple space if necessary to keep all tuples in order, with the aim to cut down the tuple space search overheads for subsequent cache-miss packets. By this way, it is expected to greatly boost the classification speed of incoming packets in virtual SDN switches.

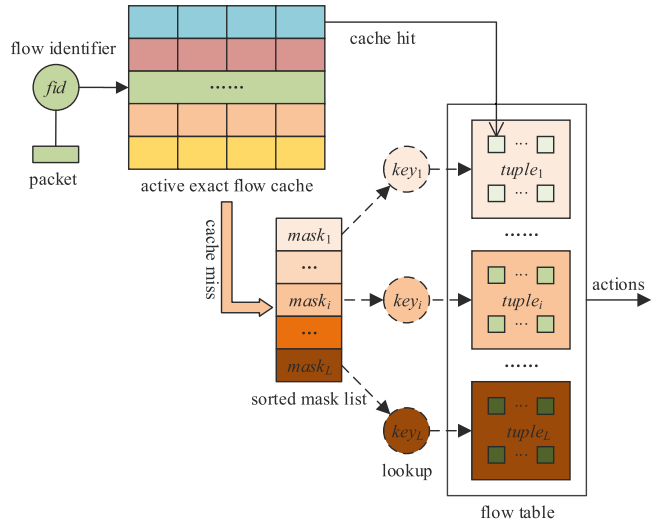


Fig. 2. Our proposed packet classification scheme FastTSS.

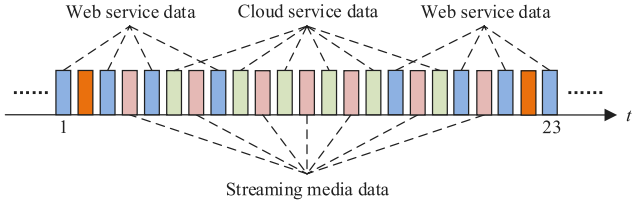


Fig. 3. A typical example of packet traffic with temporal locality.

4.2. Flow cache design

This part throws light on the temporal locality of network traffic in terms of packet flows, and designs a well-exploited flow cache to directly retrieve respective flow entries for most packets.

4.2.1. Temporal locality of network traffic

Extensive network measurements have indicated that locality phenomenon widely occurs in packet traffic under various network scenarios (Duffield et al., 2005; Williamson, 2001). This phenomenon is the combined effect of network protocols and applications, respectively specifying and launching data transmission behaviors between communication entities. Firstly, the most prevalent Internet service WWW generally exhibits a series of file downloading/uploading behaviors between Web servers and browsers. Secondly, popular streaming media applications, such as Internet television and live video streaming, generate persistent content distributions from data centers or proxy servers to end-users. Thirdly, various cloud services produce a great deal of bulk data transfer activities in the case of user operations and data migrations.

Fig. 3 illustrates a typical example of packet traffic with temporal locality in terms of flows. As shown in Fig. 3, there are 23 successive packets mainly distributed in 4 traditional flows, including 3 TCP connections transmitting Web and cloud service data, and 1 UDP sessions delivering streaming media data. We can observe from Fig. 3 that successive packets are not independent of each other, but highly related. Particularly, packets in a short time intensively reside in very few flows. Furthermore, packets within a flow is apt to arrive in groups. These observations are summarized as temporal locality, which demonstrates the short-term characteristic of network traffic.

SDN enables flexible definitions of packet forwarding behaviors, but hardly changes network traffic characteristics. On the contrary, it strengthens the temporal locality with the incorporation of wildcards into the match fields identifying a packet flow. From the viewpoint of packet flows, each wildcard aggregates two exact flows, typically TCP connections or UDP sessions, into one wildcarding flow. As for network applications, an online interaction usually induces multiple exact flows, which can be aggregated into a single wildcarding flow. Consequently, packet traffic will be more intensively distributed, and the temporal locality will be more evident in terms of wildcarding flows in SDN.

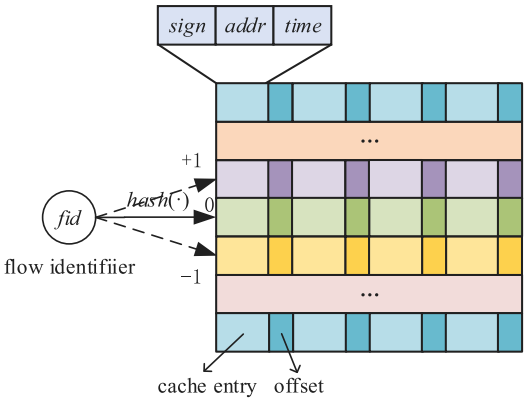


Fig. 4. Well-exploited flow cache.

4.2.2. Well-exploited flow cache

The above temporal locality implies that packets within a flow are liable to arrive in batches. Therefore, a flow will exhibit two alternate states: (a) the active one with a batch of packets being transmitted; (b) the idle one with only a few scattered or even no packet in transmission. The state of a flow can be distinguished by the inter-arrival time of a packet and its previous one within the flow. Once a flow comes into the active state, there probably will be multiple packets within it arriving soon. This inspires us to design a cache accommodating all active flows for more subsequent packets to hit it. As for an incoming packet, its flow will be put into the cache in case of its packet inter-arrival time (PIT) below a predefined threshold. The threshold should be dynamically tuned to keep the number of active flows matched with the cache size.

Our designed WEFcache holds active exact flows instead of wildcarding ones in SDN flow tables, for applying hashing techniques to achieve fast lookup. To reduce hash collisions in our designed cache, we propose a novel hashing algorithm to make the utmost of cache space for sufficient cache utilization. Our hashing algorithm provides three candidate locations among the cache for each active exact flow: its directly mapped location and two adjacent locations. As for an inserted flow, if its three candidate locations have already filled up, we kick out a flow from either of the adjacent locations to provide storage space for the inserted flow. The kicked flow is then inserted into its adjacent location in the kicking direction, and we kick out any flow that may exist there. The above process proceeds, until a vacant location is found, or the number of kicking operations comes up to a predefined upper limit. This hash algorithm is expected to reduce hash collisions, maximize cache utilization, and significantly improve the performance of packet classification in virtual SDN switches. Fig. 4 illustrates our designed WEFcache with our proposed hashing algorithm.

As shown in Fig. 4, the cache consists of k buckets, each of which contains one or more entries separately keeping an active exact flow. To save cache space, each cache entry is identified by flow signature (generally 2 or 4 bytes), rather than flow identifier at least with 13 bytes for classical 5-tuple (source ip/port, destination ip/port and protocol type). As for an active exact flow, we directly map it into a bucket in the cache by hashing its identifier, whose two adjacent buckets are also its candidate locations. We set an offset field for each cached flow to record the offset value (0, +1, −1) between its actually stored bucket

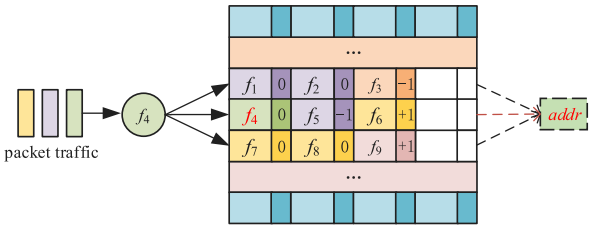


Fig. 5. Cache lookup.

and its directly mapped bucket. The value 0 represents that the cached flow is kept in its directly mapped bucket. The value +1/-1 denote that the cached flow is hold in the adjacent bucket in the upper/lower direction.

In addition, each cache entry also contains other necessary information regarding its cached flow, including the address of its respective entry in the flow table and the timestamp of the recently arrived packet within it. With the address, we can directly locate a flow entry for an arrived packet, provided that its flow signature matches with any cache entry. In such cases, we no longer require to perform tuple space search with heavy lookup overheads. The timestamp is used for calculating the idle time of the cached flow in a cache entry until now. When a full bucket is obliged to accommodate a new active exact flow, we will replace it into the cache entry with the longest idle time in the bucket. Furthermore, periodic timeout scanning will eliminate all expired flows in terms of their idle time.

4.2.3. Cache operations

(1) **Cache lookup**. Fig. 5 depicts a typical example of cache lookup. Upon receiving a packet p belongs to a flow f_4 at the time t, we first extract its flow identifier fid and calculate its flow signature $sign$. Then, we retrieve its directly mapped bucket by hashing the flow identifier fid, and match against the bucket and its adjacent buckets by the flow signature $sign$. If a cache entry is successfully matched, we directly retrieve the corresponding entry in the flow table in terms of the flow entry address $addr$ in the cache entry. Furthermore, we update the timestamp $time$ in the cache entry to t. As for failed matching, we return null.

(2) **Cache insertion**. To insert an active exact flow newly emerged, we first compute its flow signature $sign$ by its flow identifier fid. Next, we hash fid to map the flow into the cache, and locate its directly mapped bucket and two adjacent ones. Subsequently, we verify if there is any vacant entry in the three buckets. Each cache insertion has three cases:

Case 1: If there is any vacant entry in directly mapped bucket, we will directly write the key information of the inserted flow f_{10} into the entry, including its flow signature $sign$, the address of its corresponding flow entry $addr$, and the timestamp of its recently arrived packet $time$. As shown in Fig. 6(a), there is a vacant entry in the directly mapped bucket of the newly arrived flow f_{10}. In this case, we place f_{10} into the vacant entry and set its offset as 0.

Case 2: If the inserted flow f_{11} is directly mapped into a full bucket and either of its two adjacent buckets has a vacant entry at least, we will write the information of the flow f into the entry. As shown in Fig. 6(b), there is a vacant entry in the upper adjacent bucket, we place f_{11} into the vacant entry and set its offset as +1.

Case 3: If there is no vacant entry in the directly mapped bucket and its two adjacent buckets of the inserted flow f_{13}, we will randomly choose one of the two adjacent buckets, and kick out a flow in it to its adjacent bucket in the direction far from the directly mapped bucket. In particular, we prefer to select a flow whose offset direction is opposite to the kick-out direction, for returning the flow to its directly mapped bucket while making room for the inserted flow. The above kick-out

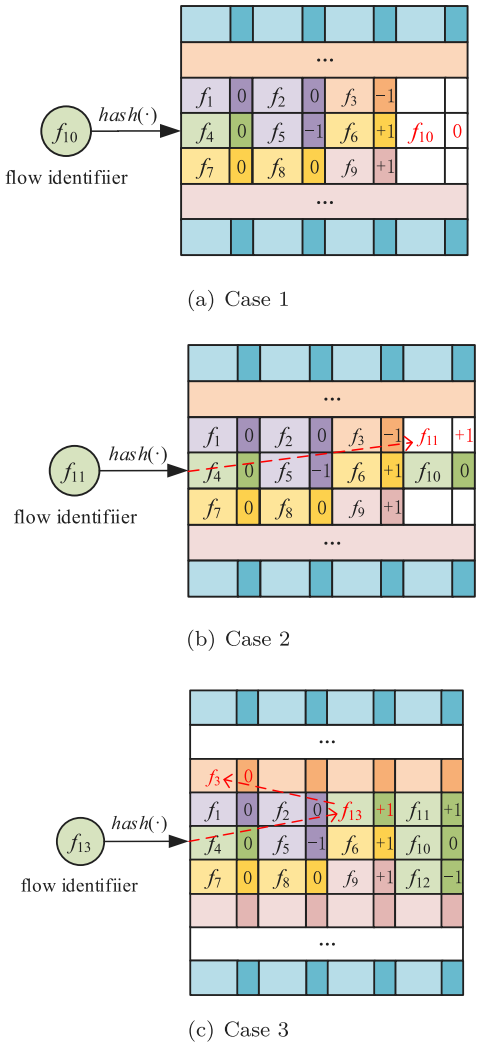


Fig. 6. Cache insertion.

operation proceeds until a vacant entry is found or the number of kick-out operations reaches a predefined limit. As shown in Fig. 6(c), the three candidate buckets of the inserted flow f_{13} are all full, and it is suitable to kick out f_3 with the offset −1 in the upper adjacent bucket for accommodating f_{13}. Then, f_3 is pushed back into a vacant entry in its directly mapped bucket.

(3) **Cache deletion**. To delete an expired flow in the cache, we first compute its flow signature $sign$ by its identifier fid. Next, we retrieve its three candidate buckets by hashing the flow identifier fid, and match all cache entries in the three candidate buckets one by one with the flow signature sign. Once succeeding to match a cache entry, we will reset it.

4.2.4. Cache hit rates

Cache hit rate is a key metric for measuring cache performance. Since our designed cache accommodates active exact flows, it is necessary to define the activity degree of an exact flow at a certain time, generally as the probability that the next arrived packet belongs to the flow. Assume the activity degree model of exact flows as follows: if the activity degree of all exact flows decreases in equal proportion after being sorted, the cache hit rate can be simplified as the sum of the activity degrees of top exact flows. Afterwards, the cache hit rate can be derived in accordance with the above activity degree model of exact flows. Let the common ratio of the activity degree be q ($0 < q < 1$),

Table 1
The estimation of cache hit rate.

N	q	n	$P_{WEFcache}$	$P_{CuckooFlow}$
100K	0.9991	2.0K	83.48%	76.32%
100K	0.9991	2.2K	86.21%	79.49%
100K	0.9992	2.0K	79.82%	72.21%
100K	0.9992	2.2K	82.81%	75.55%
200K	0.9992	2.2K	82.81%	75.55%
200K	0.9992	2.4K	85.35%	78.48%
200K	0.9993	2.2K	78.57%	70.84%
200K	0.9993	2.4K	81.37%	73.93%
400K	0.9993	2.4K	81.37%	73.93%
400K	0.9993	2.6K	83.81%	76.69%
400K	0.9994	2.4K	76.32%	68.41%
400K	0.9994	2.6K	79.00%	71.30%

which reflects the locality degree of network traffic over exact flows. Suppose there is N exact flows in network traffic, we can get the activity degree of the ith flow a_i $(1 \le i \le N)$ in all sorted flows as:

For simplicity, assuming that our designed cache contains the top n flows in terms of activity degree, the cache hit rate can be computed as the sum of their activity degrees in (2).

$$a_i = a_1 q^{i-1} = \frac{1-q}{1-q^N} q^{i-1}, \tag{1}$$

$$P = \sum_{i=1}^{n} a_i = \frac{1-q^n}{1-q^N}, \tag{2}$$

Subsequently, we can estimate the cache hit rate $P_{WEFcache}$ by setting the activity common ratio q for different number of exact flows N and cached flows n in Table 1.

As for the scheme CuckooFlow, the cache utilization will be lower than our designed cache because of the different hash algorithm. We assume that the utilization of our cache is 100% and that of CuckooFlow is 80%. Then, we obtain the cache hit rates of CuckooFlow $P_{CuckooFlow}$ in Table 1.

4.3. Tuple space search optimization

This part takes an insight into the spatial locality of packet traffic missing the above flow cache, and reduces the tuple space search overheads of the above flow table by dynamically sorting all tuples in terms of their reference frequencies and load factors.

4.3.1. Spatial locality of packet traffic

As inferred from the working principle of our designed flow cache, it will be missed for every scattered packet and the initial two packets of each batch within a flow. These packets must undergo tuple space search on the flow table. Obviously, these packets are temporally scattered over the flow, with the loss of temporal locality. Fortunately, there is spatial locality among these packets, since all flows are excepted to vary in the number of packet batches and scattered packets. In particular, most packets will concentrate on a handful of elephant flows, while numerous mice flows will only contribute to a minority of packets. In summary, the spatial locality will appear in packet traffic for tuple space search from the respective of exact flows.

As for software-defined networking, wildcards in the match fields of a flow entry aggregates multiple exact flows into a single wildcarding flow. Due to the non-uniform distribution of packet traffic on exact flows, the traffic will similarly exhibit spatial locality in terms of wildcarding flows. With regards to tuple space search, tuple references will also demonstrate spatial locality, as each tuple consists of a certain number of wildcarding flow entries. The above spatial locality is manifested as the long-term characteristic of packet traffic. Fig. 7 illustrates the spatial locality of packet traffic in terms of exact flows, wildcarding flows, and tuples.

As shown in Fig. 7, we exhibit 24 successive packets non-uniformly attached to 8 exact flows $\{e_1, e_2, \ldots, e_8\}$. Specifically, there are 5, 4, 3, and 1 packets respectively in each flow of the sets $\{e_6\}$, $\{e_1, e_3\}$, $\{e_2, e_7, e_8\}$ and $\{e_4, e_5\}$. Subsequently, the exact flow sets $\{e_1, e_3\}$, $\{e_2, e_4, e_5, e_7\}$ and $\{e_6, e_8\}$ are respectively aggregated into 3 wildcarding flows w_1, w_2, and w_3, with 8 packets in each of them. The two tuples t_1 and t_2 respectively contain the wildcarding flow sets $\{w_1, w_3\}$ and $\{w_2\}$ with 16 and 8 packets. This implies strong spatial locality of packet traffic with its uneven quantitative distribution on tuples. Consequently, there will be a chance to speed up tuple space search by exploiting the spatial locality.

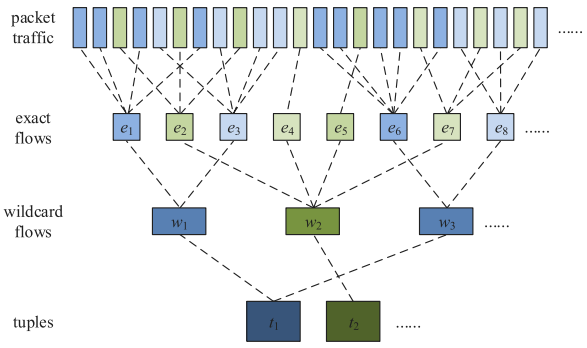


Fig. 7. The spatial locality of packet traffic.

4.3.2. Dynamic tuple sorting

According to the above spatial locality of packet traffic, all tuples are expected to be referenced with different frequencies. To reduce the average number of tuple references for cache-miss packets, we tentatively put all tuples in descending order by their reference frequencies. Nevertheless, fewer tuple references do not necessarily mean lower search overheads for a tuple space, as each tuple probably involves a different number of flow entries and requires different lookup overheads. Note that the lookup overheads of a tuple are generally characterized by average search length, chiefly relying on the load factor of its hash table. Consequently, we design a sorting metric for the tuple space of our flow table, by combining the reference frequency and the load factor of each tuple.

As for each tuple, its sorting metric is accumulated on the arrival of each packet belonging to any one of its flow entries, to reflect its reference frequency. Particularly, the increment of the metric for a packet is initially devised as the reciprocal of the load factor of its associated tuple, because a tuple with a larger load factor has higher lookup overheads and should be prudently pushed forward. Considering the special cases of too small or large load factors, we eventually design the metric increment $m_i(p_{ij})$ for the jth packet of the ith tuple p_{ij} as the truncation function,

$$m_i(p_{ij}) = \begin{cases} \lambda & \alpha_i(p_{ij}) \le \frac{1}{\lambda}, \\ \frac{1}{\alpha_i(p_{ij})} & \frac{1}{\lambda} < \alpha_i(p_{ij}) < \lambda, \\ \frac{1}{\lambda} & \alpha_i(p_{ij}) \ge \lambda, \end{cases} \tag{3}$$

where $\alpha_i(p_{ij})$ represents the load factor of the ith tuple on the arrival of the packet p_{ij}, and λ denotes the truncation value larger than 1. As for implementations, the metric m_i for the ith tuple $(1 \le i \le L)$ is the accumulation of the metric increment $m_i(p_{ij})$ for all packets p_{ij} of the ith tuple. Subsequently, all tuples are kept in descending order by their metrics all the time. Fig. 8 illustrates the schematic diagram of dynamical tuple sorting. Upon receiving a packet p_{ij} belonging to a flow entry in the ith tuple, we will calculate the metric increment $m_i(p_{ij})$ with (2) and update the metric of the tuple m_i. If the updated metric goes beyond its front ones, the tuple should be moved forward to keep all tuples in descending order. Considering continuous tuple

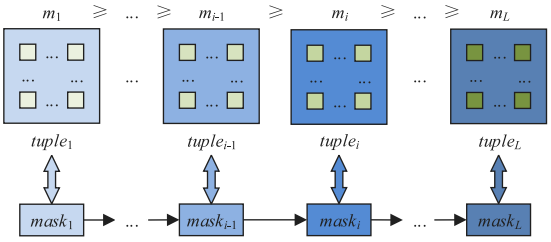


Fig. 8. The schematic diagram of dynamical tuple sorting.

references by packet traffic, all metrics of the tuple space should be regularly normalized by their maximal one.

Note that the above metric increment truncates the reciprocal of the load factor to fall in the range $[1/\lambda, \lambda]$, where λ typically values 4. In contrast, the accumulation frequency of the metric, i.e., the reference frequency of the tuple, reaches up to tens of thousands per second (Anon, 0000). This implies that the reference frequency has a much greater impact on the sorting metric than the load factor for a tuple. The tuple space will become stable with all tuples primarily sorted by their reference frequencies, after the arrival of a series of packets. Consequently, a majority of packets will match front tuples with much shorter search length, while only a minority of packets need to go through lookup failures of most tuples before matching their tuples in the rear. In summary, it will bring an obvious decrease to the average search length for cache-miss packets by dynamically sorting all tuples.

4.3.3. Search length

For simplicity, we take the average number of failed tuple lookups per packet as the performance metric of tuple space search. For feasible deduction of the performance metric, suppose tuple reference probabilities decrease in a geometric progression with the common ratio r ($r \leq 1$). Suppose each packet will and only will match a tuple, we can express the reference probability of the ith ($1 \leq i \leq L$) tuple:

$$a_i = a_1 r^{i-1} = \frac{1-r}{1-r^L} \cdot r^{i-1}. \tag{4}$$

Consequently, we can derive the average number of failed tuple lookups per packet:

$$N_{\text{fail}} = \sum_{i=1}^{L} (i-1) a_i = \frac{r}{1-r} - \frac{L r^L}{1-r^L}. \tag{5}$$

The factor r^L in (5) comes close to 0, due to the common ratio r less than 1 and the number of tuples L generally dozens and even up to hundreds. Therefore, we can approximate the average number of failed tuple lookups:

$$N_{\text{fail}} \approx \frac{r}{1-r}. \tag{6}$$

As seen from (6), the average number of failed tuple lookups is merely dominated by the common ratio of tuple reference probability r. Specifically, when the common ratio r values 0.7, 0.8 and 0.9, we can estimate the average number of failed tuple lookups N_{fail} respectively as 2.33, 4, and 9. As for direct tuple space search, suppose all tuples are referred with identical probability, we can easily achieve its average number of failed tuple lookups as $(L–1)/2$. This is to say, the average number of failed tuple lookups grows almost linearly with the increasing number of tuples, and goes far beyond that of sorted tuple space. In conclusion, it will significantly reduce the average number of failed tuple lookups.

5. Algorithmic implementation and performance analysis

This section presents the algorithmic implementations of our proposed packet classification scheme FastTSS, and analyzes its algorithmic complexity in regards to average search length.

5.1. Packet classification algorithm

Algorithm 1 provides the pseudo-code implementation of our proposed packet classification scheme FastTSS for virtual SDN switches. As for an arrived packet, a switch initially decodes its protocol headers to obtain its key fields, and computes its flow identifier fid for the lookup on our designed cache (lines 1–2). If the cache lookup is successful to locate a cache entry, we proceed to retrieve the respective flow entry with its address $addr$ recorded in the entry. Afterwards, we obtain the match fields from the flow entry and check if they are matched with the flow identifier fid (lines 3–5). Once the match succeeds, it can directly forward the packet with the action set in the flow entry, bypassing the tuple space search on the flow table (line 6). Ultimately, the cache entry should be updated, i.e., its timestamp is reset by packet arrival time (line 7). Meanwhile, the flow entry also needs to be updated such as counters (line 8).

If the cache lookup fails, it has to proceed with tuple space search on the virtual SDN flow table. Specifically, we sequentially iterate through all tuples in the flow table, with their respective masks and the flow identifier, for a matched flow entry. With regard to each tuple, we first operate the bitwise AND on its mask and the flow identifier to obtain a masked key, and perform lookup on the tuple (lines 12–14). As for successful tuple lookup, we will retrieve a flow entry and directly process the packet by its action set (lines 15–16). Moreover, the flow entry should be updated, such as counters and its recently arrived exact flows (line 17). Afterwards, we verify whether the corresponding exact flow has transitioned into the active state. Specifically, we compare the packet arrival interval between the current one and the recently arrived one in the exact flow with the PIT threshold. If the interval goes below the PIT threshold, the exact flow should be inserted into the cache (lines 21–24). Otherwise, we update the sorting metric of the matched tuple, and move the tuple ahead if required to keep tuple space in order (lines 26–29). As for failed tuple space search, the virtual SDN switch will transmit a flow setup request to its SDN controller for a flow rule (lines 35–36).

5.2. Algorithmic complexity analysis

Average search length is a key performance metric of our proposed packet classification scheme FastTSS. Given that packet traffic can be broken down into exact flows, we formulate average search length for each packet within an exact flow. As for packets in each exact flow, each of them either hit the above cache, or perform the tuple space search for the case of cache miss. The average search length of the tuple space can be deduced with the average number of failed tuple lookups. Then we can further derive the average search length of our proposed FastTSS scheme based on the cache hit rate.

Algorithm 1 The pseudo-code implementation of our proposed packet classification scheme FastTSS.

```
PacketClassify (Packet pkt)
1: fid ← ParsePacket(pkt);
2: ce ← CacheLookup(fid);
3: if ce is not NULL, then
4:     fe ← GetFlowEntry(ce.addr);
5:     if fe==fe− > fid then
6:         ExecuteActions(fe.actions, pkt);
7:         UpdateFlow(fe, pkt);
8:         ce.time ← Normalize(pkt.time);
9:         return true;
```

```
    end if
end if
for i ← 1, TUPLE_NUM do
    key ← pkt.fid&tuples[i].mask;
    for fe ∈ tuples[i], do
        if fe.key == key, then
            ExecuteActions(fe.actions, pkt);
            UpdateFlow(fe, pkt);
            break
        end if
    end for
    if fe is not NULL, then
        if fe.latest_fid = fid && pkt.time − fe.time ≤ PIT, then
            f ← NewFlow(fid, addr, pkt.time);
            CacheInsert(f);
        else
            UpdateMetric(tuples[i].metric);
            while i ≥ 1 && tuples[i].metric > tuples[i − 1].metric, do
                Swap(tuples[i], tuples[i − 1]);
                i − −;
            end while
        end if
        return true;
    end if
end for
msg ← CreateMessage(pkt);
SendMessage(msg);
return false;
```

Upon receiving a packet, the switch first looks up the cache. As for successful cache lookup, we will directly retrieve a matched cache entry and locate the respective flow entry with the search length $SL_{\text{cache-hit}}$=1. Otherwise, we must perform tuple space search, by sequentially looking up all tuples until a matched flow entry is found. As for the lookup failure of each tuple, we need to traverse the tuple with the search length $SL_{\text{tuple-fail}} = \alpha$, where α denotes the load factor of the tuple. If a tuple lookup finally succeeds, it needs the search length $SL_{\text{tuple-hit}} = \alpha/2 + 1$ to find out a flow entry in the tuple. With the approximate average number of failed tuple lookups N_{fail} in (6), we further derive the search length for the case of cache miss:

$$\begin{aligned} SL_{\text{cache-miss}} &= 1 + N_{\text{fail}} \cdot SL_{\text{tuple-fail}} + SL_{\text{tuple-hit}} \\ &\approx \frac{\alpha(1+r)}{2(1-r)} + 2. \end{aligned} \tag{7}$$

With the cache hit rate in (2), we finally derive the average search length of our proposed FastTSS scheme:

$$\text{ASL}_{\text{FastTSS}} = P \cdot SL_{\text{cache-hit}} + (1-P) SL_{\text{cache-miss}}. \tag{8}$$

As shown in (8), the average search length primarily depends on the cache hit rate P, the common ratio of tuple reference probability r, and the load factor of each tuple α. By substituting typical values of each parameter into (8), we can compute the estimated average search length of our proposed FastTSS scheme in Table 2.

As for the packet classification scheme CuckooFlow, its average search length primarily relies on the cache hit rate $P_{CuckooFlow}$, the common ratio of tuple reference probability r, and the load factor of each tuple α. By substituting typical values of each parameter into 2, we can calculate the estimated average search length of the classical packet classification scheme in Table 2.

As shown in Table 2, our proposed FastTSS scheme achieves shorter average search length than the packet classification scheme CuckooFlow by estimation, with the speedup ratio above 1.2 all the time. Specifically, the average search length of our proposed FastTSS scheme is estimated to fall below 6 in most of the time, while that of the CuckooFLow keeps above 5 most of the time. Furthermore, we can also see that the average search length of our proposed FastTSS scheme chiefly depends on the cache hit rate P and the load factor of each tuple α.

Table 2
The estimation of average search length for different packet classification scheme.

α	r	$P_{WEFcache}$	$P_{CuckooFlow}$	$ASL_{FastTSS}$	$ASL_{CuckooFlow}$	$Speedup$
2	0.8	0.7632	0.6841	3.36	4.15	1.23
2	0.9	0.7632	0.6841	5.73	7.31	1.27
4	0.8	0.7632	0.6841	5.49	7.00	1.27
4	0.9	0.7632	0.6841	10.23	13.32	1.30
2	0.8	0.7900	0.7130	3.10	3.87	1.24
2	0.9	0.7900	0.7130	5.20	6.74	1.29
4	0.8	0.7900	0.7130	4.99	6.45	1.29
4	0.9	0.7900	0.7130	9.19	12.19	1.32
2	0.8	0.8381	0.7669	2.61	3.33	1.27
2	0.9	0.8381	0.7669	4.23	5.66	1.33
4	0.8	0.8381	0.7669	4.07	5.42	1.33
4	0.9	0.8381	0.7669	7.31	10.09	1.37
2	0.8	0.8621	0.7949	2.37	3.05	1.28
2	0.9	0.8621	0.7949	3.75	5.10	1.35
4	0.8	0.8621	0.7949	3.62	4.89	1.35
4	0.9	0.8621	0.7949	6.37	8.99	1.41

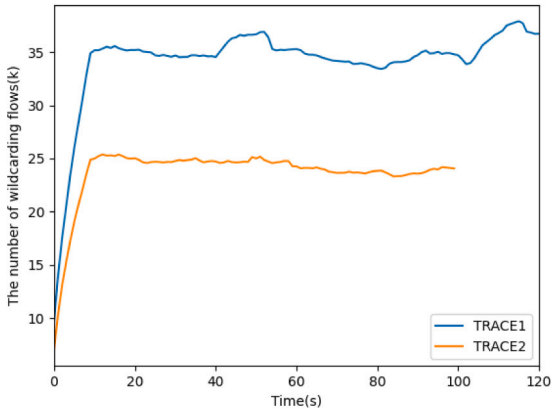


Fig. 9. The number of wildcarding flows in the traces.

6. Experiments

This section evaluates the packet classification performance of our proposed FastTSS scheme in terms of cache hit rate and average search length with physical network traffic traces.

6.1. Experimental methodology

To facilitate convenient experimental comparison, we implement various packet classification schemes for virtual SDN switches with C/C++ programming language. For multiple experimental evaluations, the program is repeatedly operated off-line on physical network traffic traces. As for our experimental traces, we select a popular public network traffic TRACE1 and TRACE2 (Anon, 0000). Each trace contains 15,420,235 packets captured from a 10Gps backbone network connecting Jiangsu Province to the CERNET, with the sampling ratio 1:4. Fig. 9 demonstrates the changing number of wildcarding flows in the above traces. It can be seen that TRACE1 and TRACE2 respectively have approximate duration 122 s and 100 s. After initial 10 s set as timeout, both TRACE1 and TRACE2 keep a relatively steady number of wildcarding flows, respectively with about 35 k and 24.5 k.

Particularly, we perform different classification schemes for packets in traffic traces, and record statistical information to compute performance metrics such as cache hit rate and average search length. For simplicity, we choose the classical five fields, i.e., source ip, source port, destination ip, destination port and protocol type, as the match fields of flow tables. The mask of each match field is configured as follows: $0xff$ for protocol type, 0×8000 for both types of ports, and default subnet mask for both types of IP ($0xffffffff$ for classes D and E addresses). This will generate 16 different masks. Furthermore, we set 10 s for the timeout interval of each flow entry.

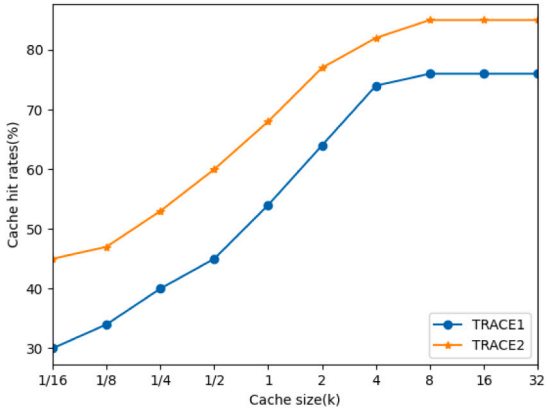


Fig. 10. The cache hit rate for different cache sizes.

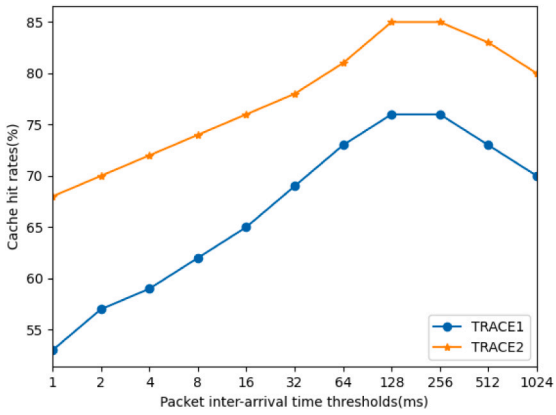
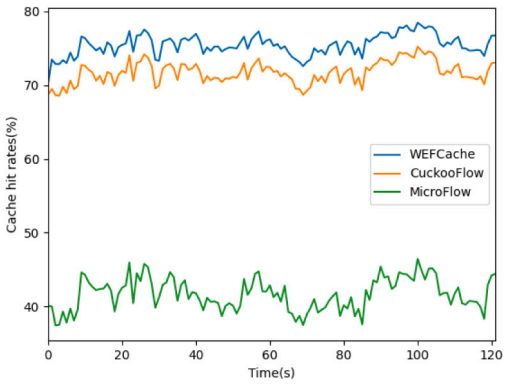
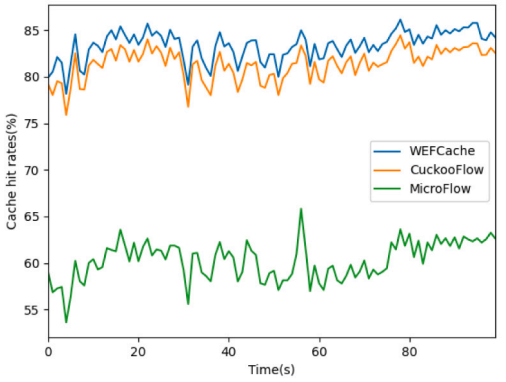


Fig. 11. The cache hit rate for different PIT thresholds.



(a) TRACE 1



(b) TRACE 2

Fig. 12. The cache hit rate for different packet classification schemes..

6.2. Cache hit rate

Cache hit rate is a primary performance metric of flow caches. As for our designed flow cache, the cache hit rate primarily relies on the cache size and the PIT threshold of active exact flows. By fixing the PIT threshold as 256 ms and setting different cache sizes, we perform our proposed FastTSS scheme on the above traffic traces and get the respective cache hit rate in Fig. 10.

As shown in Fig. 10, the cache hit rate stably goes up with the increasing cache size no more than $8k$, and turns to remain constant for larger cache size, no matter which traffic trace. Then we can infer that there are around $8k$ active exact flows in traffic traces with the PIT threshold 256 ms. As for the cache size smaller than $8k$, we can obtain a higher cache hit rate by increasing the cache size to keep more active exact flows. When the cache size goes beyond $8k$, the cache has already kept all active exact flows, and it no longer brings any benefit by enlarging the cache size.

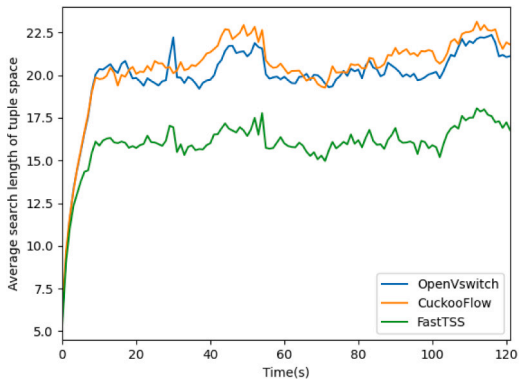
Similarly to the above experiment, we obtain the cache hit rate of our proposed FastTSS scheme for the above traffic traces in Fig. 11, by fixing the cache size as $8k$ and setting different PIT thresholds. As seen from Fig. 11, no matter which traffic trace, the cache hit rate shows a similar rule of first rising and then falling, and reaches its peak for the PIT threshold between 128 ms and 256 ms. When the PIT threshold is too small, there will be only a small number of active exact flows far from filling up the cache. As for too large PIT thresholds, almost all exact flows are deemed to be active, and it is nearly equal to cache recently emerged exact flows. These two extreme cases will lead to low cache hit rates. In particular, the cache hit rate will get its maximum when the cache size just matches with the number of active exact flows.

According to the above experimental results, it is suitable to configure the cache size and the PIT threshold respectively as $8k$ and 256 ms for our proposed FastTSS scheme with the WEFcache. Subsequently, we operate typical packet classification schemes on each traffic trace, count the number of packets with cache hit, and calculate the cache hit rate in Fig. 12. We can see from Fig. 12 that our proposed WEFcache achieves the highest cache hit rate up to 77% and 85% respectively for TRACE1 and TRACE2. Specifically, the two packet classification schemes based on active-exact-flow caches obtain much higher cache hit rates compared to Open vSwitch with the micro-flow cache. This is because our designed WEFcache achieve better utilization of cache space than cuckoo hashing in the CuckooFlow scheme, by applying our proposed hash algorithm with low hash collision.

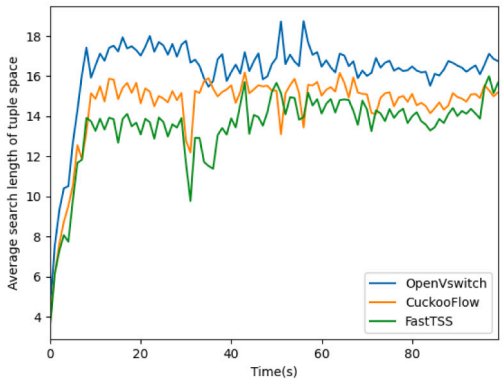
6.3. Average search length

Average search length is a classical performance metric of packet classification for virtual SDN switches. As for the tuple space search in the case of cache miss, its performance is characterized by average search length of tuple space. With the above configurations and the hash length of each tuple set as 2^9, we operate typical packet classification schemes on each traffic trace, and calculate average search length of tuple space for each cache-miss packet and that of flow table for each packet respectively in Figs. 13 and 14.

As shown in Fig. 13, our proposed FastTSS scheme has shorter average search length of tuple space compared to the other two packet classification schemes. This is attributed to the fact that the FastTSS scheme adjusts the tuple space if necessary after each packet arrival, which effectively reduces the search length of the tuple space for subsequent packets in accordance with the spatial locality of packet

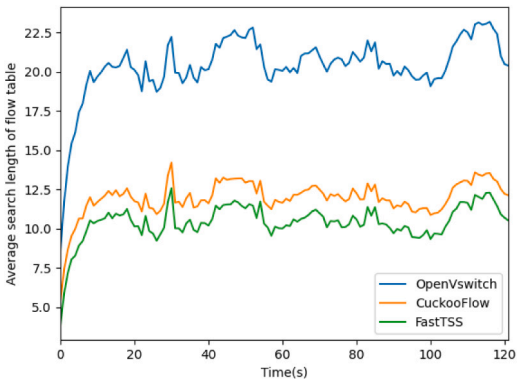


(a) TRACE 1

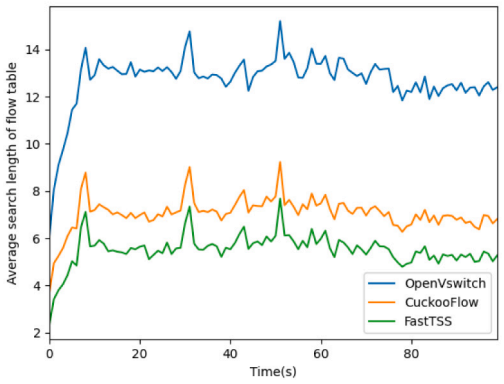


(b) TRACE 2

Fig. 13. Average search length of tuple space for different packet classification schemes.



(a) TRACE 1



(b) TRACE 2

Fig. 14. Average search length of tuple space for different packet classification schemes.

traffic. In contrast, the other two packet classification schemes keep the tuple space constant all the time without consideration of network traffic characteristic.

As seen from Fig. 14, our proposed FastTSS scheme achieves shorter average search length of flow table compared to the other packet classification schemes. Specifically, the FastTSS scheme performs packet classification with the average search length about 10.42 for TRACE1, while the CuckooFlow and Open vSwitch schemes need to take the average search length respectively around 11.96 and 20.37. As for TRACE2, we also achieve a great improvement with the average search length of the FastTSS scheme about 5.51, while that of the CuckooFlow and Open vSwitch schemes respectively around 7.06 and 12.72. In summary, our FastTSS scheme achieve the speedup of average search length 1.15 2.30 compared to the other two schemes. This is because our FastTSS scheme achieve higher cache hit rates than the other two schemes, and accelerate tuple space search by dynamically sorting the tuple space.

7. Conclusion

Virtual switches performance is seriously deteriorated in SDN, owing to the incorporation of wildcards into its flow tables. This paper is motivated to propose a fast packet classification scheme called FastTSS. In particular, we design a well-exploited flow cache to directly retrieve respective flow entries for incoming packets. Furthermore, we propose a novel hash algorithm to resolve the hash collisions of the cache, by providing three candidate locations for each inserted flow and making room for conflicting flow through kicking operations. However, non-empty cache items will converge to integrate several blocks, and it is difficult to carry out kicking operation. As for cache-miss packets, we expedite tuple space search by dynamically sorting all tuples in accordance with their reference frequencies and load factors.

We have experimentally evaluated the performance of our proposed FastTSS scheme with physical network traffic traces. Based on the experimental result, it is evident that our proposed FastTSS scheme surpasses both Open vSwitch and CuckooFlow in terms of cache hit rate and average search length. Specifically, the FastTSS scheme achieves high cache hit rates of approximately 77% and 85%, along with short average search length about 10.4 and 5.5, respectively for TRACE1 and TRACE2. In conclusion, the FastTSS scheme achieves significant acceleration of packet classification for virtual SDN switches.

In our future work, we will collect more traffic traces from various network scenarios, and utilize them to validate the superiority of our proposed packet classification scheme. Meanwhile, we are planing to implement the scheme, and integrate it into prevalent virtual SDN switches including Open vSwitch. Furthermore, other applications of the scheme are also within our future work plan. Particularly, we will attempt to employ our proposed flow cache to other flow-based network appliances such as hardware switches, to mitigate various kinds of performance bottlenecks like high energy consumption.

CRediT authorship contribution statement

Bing Xiong: Writing – review & editing, Project administration. **Jing Wu:** Writing – original draft, Visualization. **Guanglong Hu:** Software, Formal analysis. **Jin Zhang:** Methodology, Investigation. **Baokang Zhao:** Supervision, Resources. **Keqin Li:** Methodology, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Baokang Zhao 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 Foundation of Hunan Provincial Education Department. Jing Wu reports financial support was provided by Postgraduate Scientific Research Innovation Project of Hunan Province. Bing Xiong has patent pending to Changsha University of Science and Technology. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported in part by National Natural Science Foundation of China (U22B2005, 61972412), Hunan Provincial Natural Science Foundation of China (2023JJ30053), Scientific Research Fund of Hunan Provincial Education Department (22A0232), and Postgraduate Scientific Research Innovation Project of Hunan Province (CX20230913).

Data availability

The data that has been used is confidential.

References

Anon, Network traffic traces, http://iptas.edu.cn/src/system.php.

Blaiech, K., Hamadi, S., Mseddi, A., et al., 2014. Data plane acceleration for virtual switching in data centers: NP-based approach. In: Proceedings of the 3rd IEEE International Conference on Cloud Networking. CloudNet, Luxembourg, pp. 108–113.

Chowdhury, K., Boutaba, R., 2010. A survey of network virtualization. Comput. Netw. 54 (5), 862–876.

Congdon, P.T., Mohapatra, P., Farrens, M., et al., 2014. Simultaneously reducing latency and power consumption in OpenFlow switches. IEEE/ACM Trans. Netw. 22 (3), 1007–1020.

Daly, J., Bruschi, V., Linguaglossa, L., et al., 2019. TupleMerge: Fast software packet processing for online packet classification. IEEE/ACM Trans. Netw. 27 (4), 1417–1431.

Daly, J., Torng, E., 2017. Tuplemerge: building online packet classifiers by omitting bits. In: Proceedings of the 26th IEEE International Conference on Computer Communication and Networks. ICCCN, Vancouver, Canada, pp. 1–11.

Duffield, N., Lund, C., Thorup, M., 2005. Estimating flow distributions from sampled flow statistics. IEEE/ACM Trans. Netw. 13 (5), 933–946.

Emmerich, P., Raumer, D., Gallenmüller, S., et al., 2018. Throughput and latency of virtual switching with Open vSwitch: A quantitative analysis. J. Netw. Syst. Manage. 26 (2), 314–338.

Firestone, D., 2017. VFP: A virtual switch platform for host SDN in the public cloud. In: Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation. NSDI, Boston, USA, pp. 315–328.

Firestone, D., Putnam, A., Mundkur, S., et al., 2018. Azure accelerated networking: Smartnics in the public cloud. In: Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation. NSDI, Renton, USA, pp. 51–64.

Gupta, P., McKeown, N., 2000. Packet classification using hierarchical intelligent cuttings. IEEE Micro 4 (1), 34–41.

He, P., Xie, G., Salamatian, K., et al., 2014. Meta-algorithms for software-based packet classification. In: Proceedings of the 22nd IEEE International Conference on Network Protocols. ICNP, Research Triangle Park, North Carolina, USA, pp. 308–319.

Li, W., Li, X., Li, H., Xie, G., 2018. CutSplit: A decision-tree combining cutting and splitting for scalable packet classification. In: Proceedings of the IEEE Conference on Computer Communications. INFOCOM, Honolulu, USA, pp. 2645–2653.

Li, W., Yang, T., Rottenstreich, O., et al., 2020. Tuple space assisted packet classification with high performance on both search and update. IEEE J. Sel. Areas Commun. 38 (7), 1555–1569.

Luo, Y., Murray, E., Ficarra, T., 2010. Accelerated virtual switching with programmable NICs for scalable data center networking. In: Proceedings of the 2nd ACM SIGCOMM Workshop on Virtualized Infrastructure Systems and Architectures. New Delhi, India, pp. 65–72.

Mckeown, N., Anderson, T., Balakrishnan, H., et al., 2008. OpenFlow: Enabling innovation in campus networks. ACM SIGCOMM Comput. Commun. Rev. 38 (2), 69–74.

Molnár, L., Pongrácz, G., Enyedi, G., et al., 2016. Dataplane specialization for high-performance openflow software switching. In: Proceedings of the ACM Conference on Special Interest Group on Data Communication. SIGCOMM, Florianopolis, Brazil, pp. 539–552.

Nakajima, Y., Hibi, T., Takahash, H., et al., 2014. Scalable, high-performance, elastic software OpenFlow switch in userspace for wide-area network. In: Proceedings of the Research Track in Open Networking Summit. ONS, Santa Clara, USA, pp. 1–2.

Pfaff, B., Pettit, J., Koponen, T., et al., 2009. Extending networking into the virtualization layer. In: Proceedings of the 8th ACM SIGCOMM Workshop on Hot Topics in Networks. HotNets, New York, NY, pp. 1–16.

Pfaff, B., Pettit, J., Koponen, T., et al., 2015. The design and implementation of Open vSwitch. In: Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation. NSDI, Oakland, CA, pp. 2–16.

Pongrácz, G., Molnar, L., Kis, Z., 2013. Removing roadblocks from SDN: OpenFlow software switch performance on intel DPDK. In: Proceedings of the 2nd European Workshop on Software Defined Networks. EWSDN, Berlin, Germany, pp. 62–67.

Rahimi, R., Veeraraghavan, M., Nakajima, Y., et al., 2016. A high-performance OpenFlow software switch. In: Proceedings of the 17th IEEE International Conference on High Performance Switching and Routing. HPSR, Yokohama, Japan, pp. 93–99.

Rashelbach, A., Rottenstreich, O., Silberstein, M., 2022. Scaling Open vSwitch with a computational cache. In: 19th USENIX Symposium on Networked Systems Design and Implementation. NSDI 22.

Rashelbach, A., Rottenstreich, O., Silberstein, M., 2023. Scaling by learning: Accelerating Open vSwitch data path with neural networks. IEEE/ACM Trans. Netw. 31 (3), 1230–1243.

Rizzo, L., Carbone, M., Catalli, G., 2012. Transparent acceleration of software packet forwarding using netmap. In: Proceedings of the 31st Annual IEEE International Conference on Computer Communications. INFOCOM, Orlando, USA, pp. 2471–2479.

Singh, S., Baboescu, F., Varghese, G., et al., 2003. Packet classification using multidimensional cutting. In: Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications. SIGCOMM, New York, pp. 213–224.

Srinivasan, V., Suri, S., Varghese, G., 1999. Packet classification using tuple space search. ACM SIGCOMM Comput. Commun. Rev. 29 (4), 135–146.

Tang, D., Wang, S., Liu, B., et al., 2023. GASF-IPP: Detection and mitigation of LDoS attack in SDN. IEEE Trans. Serv. Comput. 1–12 (early access).

Tang, D., Yan, Y.D., Zhang, S.Q., et al., 2022a. Performance and features: Mitigating the low-rate TCP-targeted dos attack via SDN. IEEE J. Sel. Areas Commun. 40 (1), 428–444.

Tang, D., Zhang, S., Yan, Y., et al., 2022b. Real-time detection and mitigation of LDoS attacks in the SDN using the HGB-FP algorithm. IEEE Trans. Serv. Comput. 15 (6), 3471–3484.

Varvello, M., Laufer, R., Zhang, F., et al., 2016. Multilayer packet classification with graphics processing units. IEEE/ACM Trans. Netw. 24 (5), 2728–2741.

Wang, Y., Gobriel, S., Wang, R., et al., 2018. Hash table design and optimization for software virtual switches. In: Proceedings of the ACM SIGCOMM Afternoon Workshop on Kernel Bypassing Networks. KBNets, Budapest, Hungary, pp. 22–28.

Wang, Y., Tai, T.Y.C., Wang, R., et al., 2017. Optimizing Open vSwitch to support millions of flows. In: Proceedings of the IEEE Global Communications Conference. GlobeCom, Singapore, pp. 1–7.

Williamson, C., 2001. Internet traffic measurement. IEEE Internet Comput. 5 (6), 70–74.

Xiong, B., Hu, Z., Luo, Y., et al., 2019. CuckooFlow: Achieving fast packet classification for virtual OpenFlow switching by exploiting network traffic locality. In: Proceedings of the 17th IEEE International Symposium on Parallel and Distributed Processing with Applications. ISPA, Xiamen, China, pp. 1071–1078.

Yi, B., Wang, X., Li, K., et al., 2018. A comprehensive survey of network function virtualization. Comput. Netw. 133, 212–262.

Yingchareonthawornchai, S., Daly, J., Liu, A.X., et al., 2016. A sorted partitioning approach to high-speed and fast-update OpenFlow classification. In: Proceedings of the 24th IEEE International Conference on Network Protocols. ICNP, Singapore, pp. 1–10.

Yingchareonthawornchai, S., Daly, J., Liu, A.X., et al., 2018. A sorted-partitioning approach to fast and scalable dynamic packet classification. IEEE/ACM Trans. Netw. 26 (4), 1907–1920.

Zhou, D., Yu, H., Kaminsky, M., et al., 2020. Fast software cache design for network appliances. In: Proceedings of the 2020 USENIX Annual Technical Conference. USENIX ATC, Boston, MA, pp. 657–671.

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 artificial intelligence applications.

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.

Guanglong Hu received the B.S. degree in Network Engineering from Hunan Institute of 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, packet classification and green communications.

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.

Baokang Zhao received the B.S., M.S., and Ph.D. degrees from National University of Defense Tec hnology, all in computer science. He is currently an Associate Professor in the School of Computer Science, NUDT. His research interests include system design, protocols, algorithms, and security i ssues in computer networks.

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, energy efficient 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).