TECache: Traffic-Aware Energy-Saving Cache With Optimal Utilization for TCAM Flow Tables in SDN Data Plane

Bing Xiong, Guanglong Hu, Songyu Liu, Jinyuan Zhao, Jin Zhang, Baokang Zhao, and Keqin Li, *Fellow, IEEE*

***Abstract*—In the paradigm of Software-Defined Networking (SDN), its data plane generally perform packet forwarding based on flow table lookup on TCAM with high energy consumption. Popular energy-saving methods employ caching techniques for most packets to bypass energy-intensive TCAM lookups. However, existing energy-saving caches cannot adapt to network traffic fluctuation with sufficient utilization of cache space due to non-negligible hash conflicts. To overcome this issue, we design a traffic-aware energy-saving cache with optimal utilization for TCAM flow tables in SDN data plane. In particular, we first devise a nearly conflict-free hashing algorithm for the cache called FelisCatus, which provides three candidate locations for each incoming flow by adjacent hopping, and searches for an empty or replaceable entry for each conflicting flow by co-directional kicking. Then, we propose an adaptive adjustment mechanism of flow activity criterion, i.e., packet inter-arrival time threshold, for enabling the cache to consistently accommodate the most active exact flows in network traffic. Furthermore, we build an energy-efficient SDN flow table storage architecture by applying the above cache and exploiting the accessing features of different memories. Finally, we verify the performance of our designed energy-saving cache and flow table storage architecture by experiments with backbone network traffic traces. Experimental results indicate that, our designed energy-saving cache obtains stable and high hit rates around 75% even under network traffic fluctuation, and our proposed flow table storage architecture achieve high energy saving rates around 71%, with the increase of 7.89% compared to state-of-the-art ones.**

***Index Terms*—SDN data plane, TCAM flow tables, energy consumption, traffic-aware energy-saving cache, nearly conflict-free hashing, network traffic fluctuation.**

Received 9 January 2025; revised 8 May 2025; accepted 29 June 2025. Date of publication 3 July 2025; date of current version 7 October 2025. 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, 23A0735). The associate editor coordinating the review of this article and approving it for publication was P. Papadimitriou. *(Corresponding author: Baokang Zhao.)*

Bing Xiong, Guanglong Hu, Songyu Liu, and Jin Zhang are with the School of Computer Science and Technology, Changsha University of Science and Technology, Changsha 410114, China.

Jinyuan Zhao is with the School of Information Science and Engineering, Changsha Normal University, Changsha 410100, China.

Baokang Zhao is with the College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China (e-mail: bkzhao@nudt.edu.cn).

Keqin Li is with the Department of Computer Science, State University of New York at New Paltz, New Paltz, NY 12561 USA.

Digital Object Identifier 10.1109/TNSM.2025.3585703

I. Introduction

THE EMERGENCE of Software-Defined Networking (SDN) marks a revolutionary transformation in network architecture, by decoupling control logic from data forwarding and centralizing network control logic, to form independent data plane and control plane. By employing Protocol-independent Packet Processing Programming language P4, SDN achieves uniform management over network devices, simplifies network configuration, and promotes network flexibility and innovation capabilities. This architecture has been applied in a variety of environments such as data centers, wide area networks and cloud computing [1], [2]. For example, Facebook extensively applied SDN technology in its Altoona data center with SDN controllers and automated tools, to quickly configure and manage its huge data center networks, significantly reducing network configuration time and error rate [3]. Furthermore, Microsoft Azure employs SDN technology to provide virtual network services, which allows users to flexibly configure and manage their network resources in the cloud environment, achieving dynamic allocation and optimization of network resources [4].

SDN data plane commonly utilizes Ternary Content Addressable Memory (TCAM) to store flow tables for rapid wildcarding [5], [6]. This facilitates fast flow table lookup and packet forwarding in SDN switches. However, TCAM achieves rapid wildcarding by performing parallel lookup on flow tables at the cost of high energy consumption. Specifically, each lookup on TCAM consumes energy approximately 448 times that on SRAM for $1K$ flow table entries [7], [8]. More seriously, large-scale SDN deployments will be confronted with a large amount of packet traffic and a substantial number of concurrent flows, especially in supercomputing centers and cloud data centers [9]. This gives rise to intensive lookup on large-scale flow table, inducing the performance bottleneck of extremely high energy consumption of TCAM lookup. Moreover, TCAM energy consumption is vulnerable to network traffic fluctuation. Therefore, it is urgent to realize efficient and stable energy saving of TCAM flow tables under network traffic fluctuation.

Until now, there have been much research progress on reducing the energy consumption of TCAM flow tables through flow entry compression, flow table aggregation, and TCAM lookup bypass, etc. Initially, several researchers have substituted match fields in SDN flow tables with flow

labels or identifiers, to decrease the match width of each flow entry, such as Compact TCAM [10], Tag-in-Tag [11], RETCAM [13], and HMPC [14]. However, each incoming packet still needs to traverse entire TCAM flow table to complete its forwarding with substantial energy consumption. Afterwards, some researchers have applied wildcards to aggregate similar entries within SDN flow tables for cutting down the number of flow entries, such as HTFA [15], Q-fag [16], and WildMinnie [19]. Nevertheless, their stringent aggregation conditions result in low aggregation levels, along with limited energy saving effects. Furthermore, recent researchers have designed filters or caches to bypass energy-intensive TCAM flow table lookup for a certain number of incoming packets, such as DevoFlow [21], TSA-BF [8], CacheCAM [23], and CuckooFlow [26]. However, filters can only filter out packets launching new flows, which just account for a small fraction of packet traffic. Meanwhile, existing caches are difficult to achieve sufficient utilization due to the lack of excellent hash collision resolution methods and favorable adaptability to network traffic fluctuations.

To address the above-mentioned issues, this paper initially takes an insight into network traffic characteristics, and the high energy consumption of TCAM flow tables. Then, we design a traffic-aware energy-saving cache called TECache, which always accommodates packet flows with top activity for optimal cache utilization. Specifically, we propose a nearly conflict-free hash algorithm FelisCatus, to deal with the hash collision of exact active flows to be cached. Meanwhile, we devise an adaptive adjustment mechanism of packet inter-arrival time threshold as flow activity criterion and its step size, to match the number of active flows with cache capacity. Furthermore, we derive cache utilization rates and cache hit rates based on the assumption of Poisson process for flow arrivals and exact-flow activity model. With the TECache, we construct an energy-efficient SDN flow table storage architecture, by taking advantage of different flow table memories. Finally, we verify the performance of our designed TECache and flow table lookup storage architecture by experiments with backbone network traffic traces. The main contributions of this paper are summarized as follows:

- Designing a traffic-aware energy-saving cache with optimal utilization, which finds out and strives to accommodate a certain number of most active packet flows in line with its capacity, for stably maintaining high cache hit rates even under network traffic fluctuation.
- Proposing a nearly conflict-free hash algorithm for our designed energy-saving cache called FelisCatus, which provides three candidate buckets in the cache for each inserted flow by adjacent hopping, and searches for a vacancy in the cache for each conflicting flow by co-directional kicking, to almost perfectly resolve hash collision.
- Devising an adaptive adjustment mechanism of flow activity criterion, which dynamically adjusts the packet inter-arrival time threshold of active flows along with its step size, for adapting the number of active flows to cache capacity.
- Constructing an energy-efficient SDN flow table storage architecture by employing the above traffic-aware energy-saving cache, and formulating its energy saving rate and packet forwarding delay based on the assumption of flow activity model and Poisson process for packet flow arrivals.

The rest of this paper is organized as follows. Section II introduces the related works. In Section III, we investigate into high energy consumption of TCAM flow tables and network traffic characteristics. Section IV designs traffic-aware energy-saving cache with nearly conflict-free hash algorithm and adaptive adjustment mechanism of flow activity criterion, and derives the formal expression of its utilization rate and hit rate based on packet flow assumption. In Section V, we construct a energy-efficient SDN flow table storage architecture based on the above cache, and provide the pseudo-code implementation of the corresponding flow table lookup algorithm, and formulate its performance metrics such as energy saving rate and packet forwarding delay. Section VI evaluates the performance of our proposed traffic-aware energy-saving cache and flow table storage architecture by experiments with simulative network traffic traces. In Section VII, we summarize the paper.

II. Related Work

During the evolution of SDN, some research work has strived to reduce the matching length of flow entry by compressing its match fields, to tackle the increasing number of match fields in flow tables along with higher and higher energy consumption of TCAM flow table lookups. Initially, Kannan and Banerjee [10] designed a compact TCAM flow entry scheme that employs flow identifier to replace 15 match fields, for reducing the match length of TCAM flow table lookup. Banerjee and Kannan [11] also presented an efficient SDN flow table management method called Tag-in-Tag, which designs short two-layer routing path tags associated with packet flows by exploiting SDN features. Wang et al. [12] further built a source-controlled OpenFlow packet switching model, which utilizes source routing address called vector address as packet forwarding label, without the requirement of SDN flow tables. Meanwhile, Zhang et al. [13] established a TCAM flow table compression model called RETCAM, which designs inter-field merge, field mapping, and intra-field compression algorithms, to shorten the length of each flow entry for reducing the storage of TCAM flow tables. Furthermore, Srinivasavarma et al. [14] proposed a rule compression mechanism based on hierarchical encoding, which classifies all rules into different levels and encodes them into shorter forms, for reducing the storage space of rules. The above schemes can effectively cut down the energy consumption of TCAM flow table lookup, by shortening the matching length of each flow entry. However, each incoming packet still has to match against all flow entries in parallel with high energy consumption.

To mitigate the substantial energy consumption issue from parallel matching against all TCAM flow entries, numerous scholars have proposed flow table aggregation algorithms to aggregate similar flow entries for decreasing the number of

flow entries. Cheng et al. [15] introduced a flow rule reduction algorithm, which organizes all rules in a binary tree and aggregates non-prefix rules by an optimal routing table construction algorithm. Subsequently, they applied secondary aggregation by employing a modified Quine-McCluskey algorithm, for a notable reduction in the number of flow rules within SDN switches. Nevertheless, this algorithm needs to undergo complex operations and slow aggregation speeds. In response, Wu et al. [16] designed a proactive flow aggregation scheme based on destination address and source port on demand conversion, which aggregates multiple flow table entries with different source addresses but same destination address into a single entry, significantly reducing the size of flow tables in SDN switches. Then, Saha et al. [17] proposed a QoS-aware flow rule aggregation scheme for software-defined IoT, which employs a path selection heuristic algorithm to find out bottlenecked switches, and determines optimal aggregation points and paths to minimize the total number of flow rules. However, this scheme tends to induce congestion on certain links along the best path. In response, Li and Hu [18] devised an efficient flow forwarding scheme based on segment routing, which aggregates flow table entries in accordance with the overlapping degree of flow paths. Meanwhile, they employed an intelligent encoding algorithm to continuously learn online and update flow path information for optimal path aggregation, but did not consider possible conflicts between rules. To solve this problem, Khanmirza [19] proposed a rule aggregation method based on pattern dominance relation, which aggregates multiple rules at an output port by leveraging non-prefix wildcard patterns of universal addresses, for reducing the number of rules required to be stored in SDN switches. However, these flow table aggregation algorithms demand aggregated flow entries with identical action sets, which limits the degree of aggregation along with unsatisfactory TCAM energy saving effect.

Owing to the limited compression and aggregation of flow tables, some researchers have strived to bypass energy-intensive TCAM flow table lookup for incoming packets, for better energy-saving effect. Li et al. [20] accommodated exact flows with hash tables for fast lookup, and adopted TCAM to hold flows with hash collision. However, it does not apply to wildcard flows. In response, Mogul et al. [21] further designed a flow management scheme called DevoFlow, which accommodates TCAM flow entries in exact-flow tables by rule cloning, and looks up exact-flow tables in priority for less TCAM lookup. Kao et al. [8] built a TSA-BF filter with partitioned caching and automatic scaling, which predicts the inexistence of newly arrived flows in flow tables, to bypass needless lookup on TCAM flow tables for their packets. However, the filter merely takes effect on the first packet in each flow, which only take up a small fraction of packet traffic. Li et al. [22] presented a low-power TCAM packet classification scheme based on decision tree, which divides original rule set into several subsets in terms of their small fields, recursively maps rules into decision trees without rule replications, and applies top-down traversal algorithm to obtain index items. Congdon et al. [23] further devised a CAM prediction cache to map packet fingerprints into flow identifier, for most packets to skip TCAM flow table lookup after cache hit. Nonetheless, each cache lookup still consumes much energy due to its parallel matching. To mitigate this drawback, Xiong et al. [26] devised an energy-saving cache based on Cuckoo filter, which caches active exact flows in SRAM and employs kicking operations to find storage locations for each conflicted flow, for high cache utilization rates and hit rates with resultant significant energy consumption reduction. Nevertheless, the above caches are difficult to achieve sufficient space utilization due to the lack of perfect hash collision resolution. More seriously, they cannot be adaptively adjusted in line with network traffic fluctuation, to stably achieve favorable energy saving effect.

III. Motivation

A. TCAM Energy Consumption in SDN Switches

In SDN data plane, a switch takes charge of fast packet forwarding in accordance with flow rules generated by its controller, illustrated in Fig. 1. As for each arrived packet, the switch first caches it in the receiving queue of its ingress port, and parses it to extract key fields in its protocol headers for flow identifier generation. Subsequently, the flow identifier is applied to match against SDN flow table, composed of flow rules complying with southbound protocol specification. If the match succeeds, the packet will be cached in the sending queue of the egress port indicated by the matched entry to wait for forwarding, after performing a series of operations including ACL application, counter update and backplane switching. Otherwise, the switch will send a flow setup request involving the key content of the packet to its controller. After receiving corresponding flow rule, the switch installs it into the flow table, and forwards subsequent packets in the flow according to it. In summary, flow table lookup is a core step of packet forwarding in SDN switch and has a critical impact on its performance.

SDN introduces wildcards into the match fields of its flow tables, to achieve flexible definition of packet flows with different granularities. Flow tables are generally stored in TCAM (Ternary Content Addressable Memory) for fast wildcarding. However, TCAM adopts ternary logic (represented by 0, 1, and X, where X stands for “any” value), which requires more transistors than traditional binary storage units. Meanwhile, TCAM needs to activate all circuit units in storage array to implement a parallel matching mechanism during flow table lookup, which leads to high energy consumption during SDN packet forwarding. More seriously, TCAM flow tables must perform parallel matching for a large number of packets in high-speed networks, further aggravating the problem of its energy consumption. In summary, it is essential to reduce TCAM energy consumption in SDN switches.

B. Network Traffic Characteristic

There is a universal phenomenon in network traffic that it exhibits distinct features of locality and fluctuation [24], [27]. Specifically, network traffic locality refers that a small proportion of flows account for a majority of packet traffic from the space point of view, and packets within each flow tend

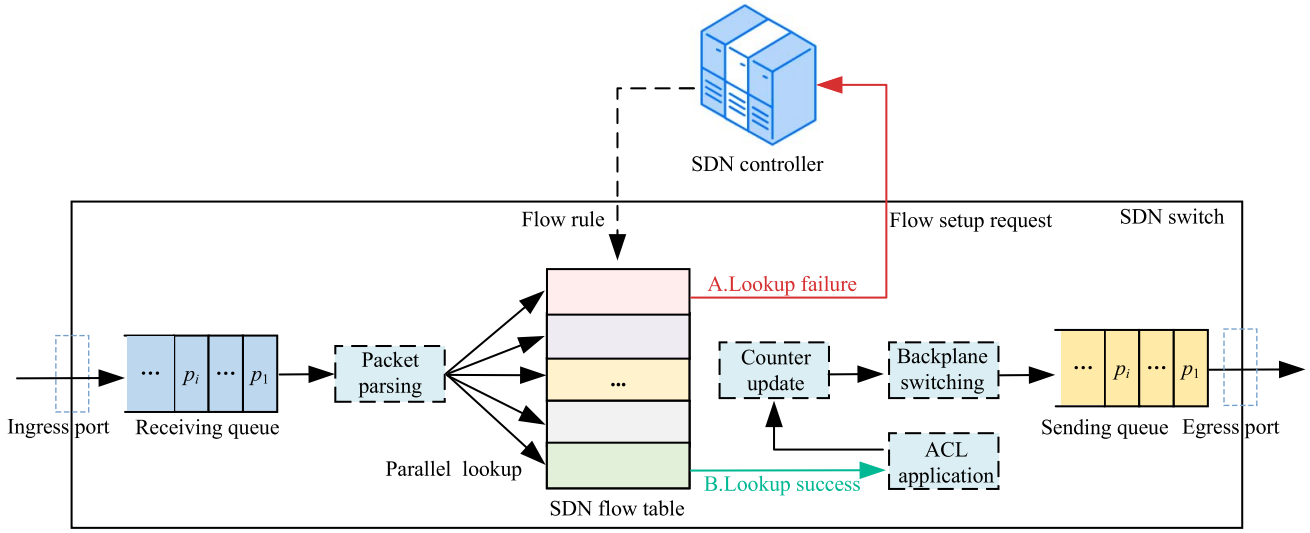

Fig. 1. Packet forwarding in a SDN switch.

to arrive in batches from the time point of view. Therefore, packet flows can be categorized into active ones and idle ones. In particular, an active flow is characterized by that many packets intensively arrive in a short time, and can be identified by Packet Inter-Arrival Time (PIAT), i.e., the interval between the arrival time of its two consecutive packets. Note that SDN incorporates wildcards into the match fields of its flow tables, where each wildcarding flow can be considered as an aggregation of multiple exact flows. Hence, it is feasible to mitigate the problem of high TCAM energy consumption, by caching active exact flows with low-power memories like SRAM. The cache is expected to greatly reduce TCAM energy consumption, by enabling most packets to directly hit the cache and bypass energy-intensive TCAM flow table lookups. However, the number of active exact flows is easily affected by network traffic fluctuation, owing to the combined effect of various factors, such as diverse user behaviors, application load variations, and unexpected cyberspace activities.

To verify the locality and fluctuation of network traffic, we choose network traffic traces released by the Jiangsu Provincial Key Laboratory of Computer Network Technology. Particularly, we select a 300-second trace around midnight on 2016/10/17 and a 100-second trace around midnight on 2013/9/3 [25]. Fig. 2(a) illustrates the real-time proportion of active exact flows and packets in batches, with the PIAT threshold set as 256*ms*. As shown in Fig. 2(a), active flows account for about 20% of all packet flows, while packets in batches comprise approximately 80% of all packets. Furthermore, Fig. 2(b) exhibits the varying number of active exact flows for different values of the PIAT threshold. Taking the PIAT threshold 256*ms* as an example, the flow number can drop to 6.1K and peak around 9.3K within 100 seconds. In short, the number of active exact flows presents significant fluctuation over different time periods. This poses a tough challenge for designing active-exact-flow caches to stably achieve favorable energy saving effect of TCAM flow tables in SDN switches.

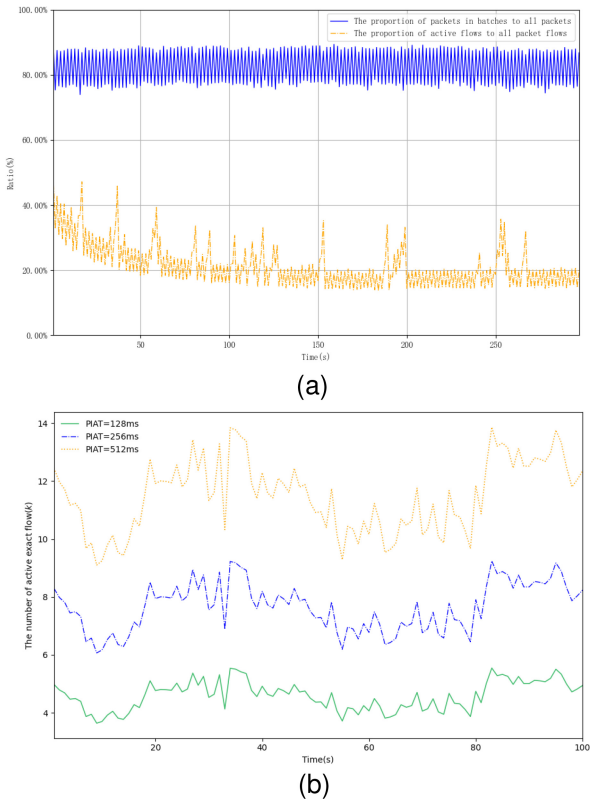

Fig. 2. Network traffic characteristics: a) Network traffic locality; b) Network Traffic Fluctuation.

IV. Traffic-Aware Energy-Saving Cache With Optimal Utilization

This section takes a deep insight into the locality and volatility of network traffic, and builds a traffic-aware energy-saving storage architecture for SDN flow tables.

A. Design Concept

1) Cache Structure: For perfect adaption to network traffic fluctuations, this paper designs a Traffic-aware Energy-saving

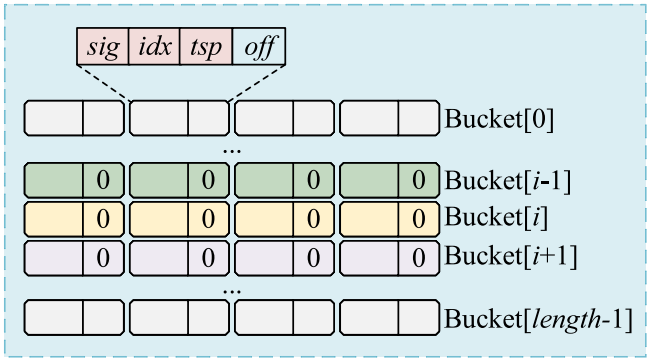

Fig. 3. The data structure of our designed TECache.

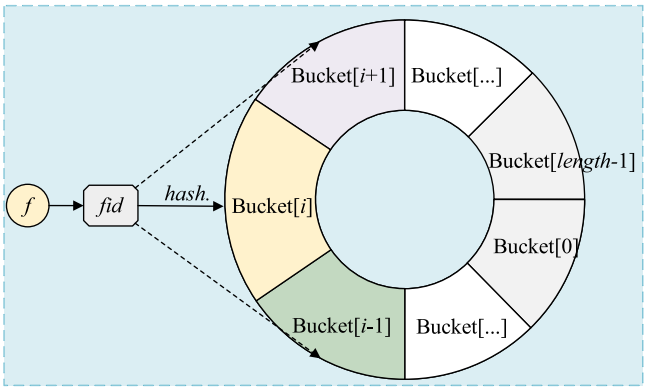

Fig. 4. The FelisCatus hash algorithm for our designed TECache.

Cache (TECache) with low-power high-speed memory in Fig. 3. The TECache accommodates most active flows all the time for optimal cache utilization. From the perspective of data structure, it consists of several hash buckets, each of which contains multiple cache entry. In particular, each cache entry records the essential information of an active exact flow, including flow signature *sig*, flow entry index *idx*, timestamp *tsp* and offset *off*. The flow signature *sig* is generated by signature function with the identifier of the active exact flow as input, typically 2 or 4 bytes. The flow entry index *idx* indicates the TCAM flow entry which the active exact flow belongs to. The timestamp *tsp* records the arrival time of most recently arrived packet in the active exact flow. The offset *off* records the storage offset of the active exact flow $(0, +1, -1)$ from its actual storage position to its hash mapping position.

2) Nearly Conflict-Free Hash Algorithm: Existing typical energy-saving caches [26] generally employed Cuckoo hashing to resolve hash collisions, while performing fast cache lookup. However, the Cuckoo hashing requires extra computational overhead for each cache lookup, and is easy to fall into potential looping status due to its kicking operation for conflicting flows. To address this problem, we design a nearly conflict-free hash algorithm called FelisCatus for our designed cache with its logical structure in Fig. 4. Specifically, the cache is logically considered as a circular array, which joints the first hash bucket and the last one. The FelisCatus maps each active exact flow to a hash bucket in the cache, and takes its two adjacent buckets as candidate storage positions by adjacent hopping. As for cache lookup, it only needs to match against its directly mapped bucket and two adjacent buckets, which can effectively ensure decent flow table lookup performance.

As for a newly emerged active exact flow to be inserted, we strive to find a storable position even by repeatedly kicking out flows in an identical direction. In particular, we accommodate it into its directly mapped bucket or two adjacent buckets, if there is any vacant entry. Otherwise, we kick out a flow from either of the adjacent buckets to make room for the inserted flow. Then, the kicked flow is inserted into its adjacent bucket in the kicking direction away from the directly mapped bucket, and we kick out any flow that may exist there. The above process proceeds, until we find a vacant entry for insertion or an inactive flow for replacement, or the number of kicking operations reaches a predefined threshold. Setting a higher value for the predefined threshold provides more opportunities to find out a vacant entry, which significantly reduces hash collision rates and enhances cache utilization, but also increases average flow insertion time. Consequently, the FelisCatus hash algorithm can always find out a storable position for any inserted flow, only if there is a vacant entry in the cache. This resolves hash collisions almost perfectly, which provides a solid foundation for sufficient utilization of cache space.

3) Adaptive PIAT Adjustment Mechanism: Active flows are generally identified by Packet Inter-Arrival Time (PIAT), i.e., the interval between the arrival time of its two consecutive packets. Existing energy-saving caches [26] adopts fixed PIAT threshold, leading to the problem that the number of active flows greatly varies with network traffic fluctuations, along with resultant unstable energy saving effect. To address this problem, we propose an adaptive adjustment mechanism of the PIAT threshold as flow activity criterion, to always cache packet flows with highest activity in network traffic. The PIAT threshold is initially set as a typical interval, such as 100*ms* or 200*ms*. After that, the mechanism monitors the fitness of the number of active flows and cache capacity, and adaptively adjusts the PIAT threshold in the case of obvious misfit.

Fig. 5 demonstrates the working principle of the adaptive PIAT adjustment mechanism. When the number of active flows in network traffic obviously goes beyond cache capacity, the cache cannot accommodate all active flows, and will randomly hold a part of active flows with suboptimal cache hit rate. In this case, we decrease the PIAT threshold to reduce the number of active flows, for matching with cache capacity. By this way, the cache will keep the most active flows in network traffic, achieving satisfactory cache hit rate. Conversely for a sharp drop in the number of active flows, there will appear many vacant entries in the cache with a great waste of cache space. At this time, we increase the PIAT threshold to augment the number of active flows, for coming close to cache capacity with full cache space utilization.

For convenient description of our adaptive PIAT adjustment mechanism, we formalize the relationship of the PIAT threshold at next time period with that at current one as follows. Suppose the PIAT threshold at time t as $PIAT(t)$ and the step size of the threshold adjustment $\delta(t)$, we can express the PIAT

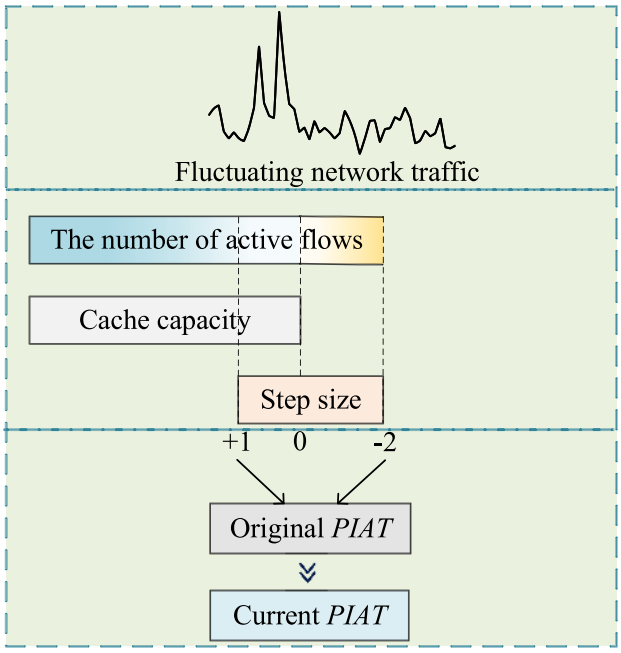

Fig. 5. Adaptive PIAT adjustment mechanism.

threshold $PIAT(t+1)$ at the next time period $t+1$ as:

$$PIAT(t+1) = PIAT(t) + \delta(t) \tag{1}$$

The step size $\delta(t)$ in (1) is not suitable to set a fixed interval, since network traffic probably undergo varying degrees of fluctuation. If the step size is set as a small interval, the number of active flows still exhibits obvious change in the case of drastic network traffic fluctuation, and no longer matches the cache capacity. Conversely, it is difficult to achieve a precise match between the number of active flows and the cache capacity, if the step size is set as a large interval. Therefore, we real-timely adjust the step size, according to the fitness of the number of active flows and cache capacity. In particular, the step size $\delta(t)$ in (1) is designed as:

$$\delta(t) = \kappa \cdot \left(1 - \frac{F(t)}{C}\right) \tag{2}$$

where *F(t)* represents the number of active flows at time *t*, *C* denotes the number of entries in the cache, and κ signifies the cardinality of step size adjustment. The value of κ depends on the fluctuation amplitude of network traffic in different network scenarios. As for network scenarios with drastic traffic fluctuations, we will set a higher value of κ for faster growth of the PIAT to quickly match the number of active flows with cache capacity. Specifically, we first calculate the ratio of the number of active flows *F(t)* to that of cache entries *C*, and its deviation degree compared to full load condition of the cache. Then, we obtain the step size $\delta(t)$ by multiplying the cardinality κ and the deviation degree. In summary, the PIAT adaptive adjustment mechanism can real-timely sense the varying number of active flows and adapt to network traffic fluctuations, for ensuring full cache space utilization and consistently maintaining high cache hit rate.

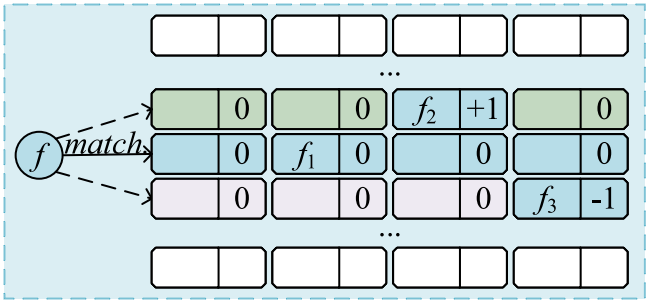

Fig. 6. Cache lookup.

B. Cache Operations

1) Cache Lookup: Fig. 6 demonstrates a typical example of lookups on the TECache. Upon the arrival of a packet *p* within a flow f arrives at the time t, we first parse its various protocol headers to extract its flow identifier *fid*, and compute its flow signature *sig*. Then, we retrieve its directly mapped bucket by hashing the flow signature *sig*, and match against the bucket and its adjacent buckets by the flow signature *sig*. If a cache entry is successfully matched, we directly locate the corresponding entry in the TCAM flow table with the flow entry index *idx* in the cache entry, and process the packet *p* by the actions in the flow entry. Finally, we update the timestamp *tsp* in the cache entry with the arrival time of the packet *p*. As for failed cache matching, we return null.

2) Cache Insertion: As for an active exact flow to be inserted, we first compute its flow signature *sig* with its flow identifier *fid*. Then, we map the flow into the cache by our designed hashing algorithm FelisCatus, to locate its directly mapped bucket and two adjacent buckets. After that, we check whether there is any vacant entry in the three buckets. Fig. 7 shows three cases of checking results and corresponding cache insertion operations:

Case 1: If there is any vacant entry in directly mapped bucket, we will directly record the information of the inserted flow in the entry, including its flow signature *sig*, the flow entry index *idx* and the timestamp of its most recently arrived packet *tsp*. Particularly for a newly arrived flow f_1 in Fig. 7, its directly mapped bucket has a vacant entry, and we insert f_1 into the vacant entry with its offset 0.

Case 2: If there is no vacant entry in directly mapped bucket, and one of its two adjacent buckets has at least a vacant entry, we will record the information of the inserted flow in the entry. As for the next inserted flow f_2 in Fig. 7, its directly mapped bucket is filled up, and its upper adjacent bucket has a vacant entry. In such case, we put f_2 into the vacant entry and set its offset to $+1$.

Case 3: If no vacant entry exists in both directly mapped bucket and its two adjacent buckets of the inserted flow, we will randomly select either of the two adjacent buckets, and kick out a flow in the selected bucket to its adjacent bucket in the direction far from the directly mapped bucket. In particular, we prefer to choose a flow whose offset direction is opposite to the kick-out direction, so it can return to its directly mapped bucket while making room for the inserted flow. The above kick-out operation proceeds until a vacant entry is found or the

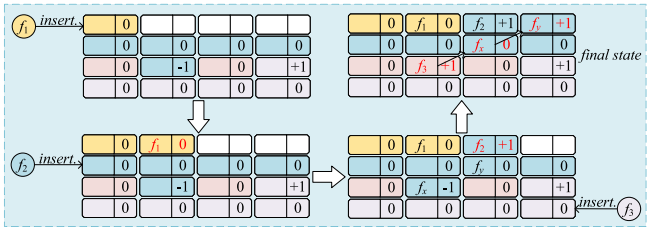

Fig. 7. Cache insertion.

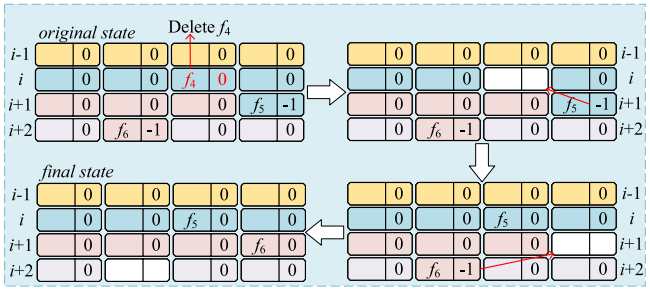

Fig. 8. Cache deletion.

number of kick-out operations reaches a predefined threshold. When the inserted flow f_3 arrives, its three candidate buckets are all full and there is a cached flow f_x with the offset -1 in its upper adjacent bucket in Fig. 7. At this time, it is proper to kick out the flow f_x upward to its directly mapped bucket for accommodating the flow f_3. Then, we randomly select a flow f_y in the directly mapped bucket of f_x to be kicked out, and insert f_y into a vacant entry in its upper adjacent bucket. Meanwhile, we change the offset of f_y to $+1$.

3) Cache Deletion: When a SDN switch receives a flow deletion message from its controller, it identifies and deletes corresponding flow entry based on flow rule information. Additionally, each flow entry also needs to be eliminated when it expires. As for the deletion of a flow entry, the SDN switch first locates and deletes all its associated exact flows in the cache. When a cached flow is deleted, we check if there is an offset cached flow in its adjacent bucket, directly mapped into the bucket of the deleted flow. If there exists, we move the offset flow back to its directly mapped bucket. Fig. 8 exhibits a typical example of deleting a cached flow. When deleting a flow f_4 in the ith bucket, we find that there is a flow f_5 in the $i+1$th bucket, which is kicked from the ith bucket. Then, we move the flow f_5 to the vacated cache entry of the deleted flow f_4 in the ith bucket and reset its offset to 0. Afterwards, we similarly transfer the flow f_6 with the offset -1 in the $i+2$th bucket into the $i+1$th bucket. The above process is performed repeatedly until there is no flow required to be moved back to its directly mapped bucket. This enables as many cached flows as possible to be stored in their directly mapped buckets, facilitating their packets to quickly locate them and perform fast forwarding.

C. Cache Performance Metrics

1) Cache Utilization Rate: Owing to hash collisions, cache space is usually difficult to be completely utilized for accommodating active flows. Nevertheless, the number of cached flows has a direct impact on the energy-saving effect of the cache. Hence, cache utilization rate, generally defined as the proportion of the number of cache flows to cache capacity, is a critical metrics of cache performance [29]. Suppose that there are K active exact flows in network traffic, and N hash buckets in the cache, each of which contains M entries. With the assumption of the probability p_{fail} that an active flow is unsuccessfully inserted into the cache, the cache utilization rate U_{cache} can be formulated as:

$$U_{cache} = \frac{K\left(1 - p_{fail}\right)}{NM} \tag{3}$$

As for our designed TECache with the hash algorithm FelisCatus, a flow can be successfully inserted in the cache for Case1, Case2, and Case3 in Section IV-B2. Its cache insertion fails only when it is full for its directly mapped buckets and two adjacent hash buckets and the number of kicks exceeds its threshold T. This implies that there is no vacant entry in the $T+3$ buckets from the mapped adjacent bucket to the bucket that was last attempted to kick into. Given that each active flow is independent and randomly mapped to a bucket in the cache, we can model the cache insertion of all active flows as a Poisson process with the parameter $\lambda = K/N$. Subsequently, we can express the probability of a bucket containing i flows as:

$$p_i = \frac{e^{-\lambda}\lambda^i}{i!} \tag{4}$$

As for a full bucket, there are at least M active flows mapped into it. Then we can calculate the probability of a bucket in full p_{full} as:

$$p_{full} = 1 - \sum_{i=0}^{M-1} p_i = 1 - \sum_{i=0}^{M-1} \frac{e^{-\lambda}\lambda^i}{i!} \tag{5}$$

Suppose each bucket in the cache is independent of each other, the probability of a failed flow insertion p_{fail} can be solved by computing the probability that all $T+3$ hash buckets are full in (6).

$$p_{fail} = p_{full}^{T+3} = \left(1 - \sum_{i=0}^{M-1} \frac{e^{-\lambda}\lambda^i}{i!}\right)^{T+3} \tag{6}$$

With the cache capacity NM and the number of inserted flows K, we can eventually calculate the cache utilization rate U_{cache} as:

$$\begin{aligned} U_{cache} &= \frac{K\left(1 - p_{fail}\right)}{NM} \\ &= \frac{K\left(1 - \left(1 - \sum_{i=0}^{M-1} \frac{e^{-\lambda}\lambda^i}{i!}\right)^{T+3}\right)}{NM} \end{aligned} \tag{7}$$

To estimate the cache utilization rates of our designed TECache, its related parameters are assumed as: there are 4096 active flows in network traffic, the cache contains 1024 hash buckets each of which has 4 entries, and its kicking number threshold is set as 10. With these parameter settings, we can approximate the flow insertion failure probability pfail as 0.062%, and the cache utilization rate U_{cache} about

99.94%. These indicate that our designed hash algorithm can almost perfectly resolve hash collisions, and our designed cache nearly accommodate all active flows when their number is equivalent to cache capacity. In summary, our designed TECache can achieve sufficient cache space utilization.

2) Cache Utilization Rate: Cache hit rate is a crucial metrics for evaluating cache performance. Since our designed TECache holds active flows, it is necessary to define the activity degree of a flow at certain time, generally as the probability that the next arrived packet belongs to the flow. Suppose that the activity degrees of all packet flows vary in equal proportion after being sorted [28]. Then, the cache hit rate can be simplified as the sum of the activity degrees of top k_s flows, where k_s is the number of flows physically stored in the cache. Therefore, we can compute the cache hit rate of our designed TECache as follows:

Assume that there are L flows in network traffic, and their activity degrees follow a geometric progression with a common ratio q ($0 < q < 1$). After sorting all packet flows in descending order of their activity degrees, we can give the activity degree of the most active flow a_1 by the properties of geometric sequences as:

$$a_1 = \frac{1-q}{1-q^L} \tag{8}$$

Subsequently, we can derive the activity degree of the *i*th flow a_i ($1 \le i \le L$) as:

$$a_i = a_1 q^{i-1} = \frac{1-q}{1-q^L} q^{i-1} \tag{9}$$

Note that our designed TECache almost perfectly resolves hash collisions by the hash algorithm FelisCatus, and adapts the number of active flows with cache capacity by the adaptive PIAT adjustment mechanism. Consequently, it can be inferred that our designed TECache will accommodate the most active flows with their number *Ks* equivalent to the cache capacity *NM*. Finally, we can deduce the cache hit rate $CHR_{TECache}$ by summing the activity degrees of these flows in (10).

$$CHR_{TECache} = \sum_{i=1}^{Ks} a_i = \frac{1-q^{K_s}}{1-q^L} = \frac{1-q^{NM}}{1-q^L} \tag{10}$$

According to (10), we can estimate the cache hit rate $CHR_{TECache}$ in Table I, for different number of all packet flows in network traffic *L*, common ratio of flow activity degree q and cache capacity *NM*.

As seen from Table I, the cache hit rate is chiefly determined by the common ratio q and the cache capacity *NM*, while almost independent on the number of all packet flows *L*. Particularly for fixed number of packet flows *L* and cache capacity *NM*, the cache hit rate rapidly increases along with the decreasing common ratio q. This is because the common ratio q reflects the skewness of packet traffic distributed over flows. As for smaller common ratio q, packet traffic distribution will be more skewed towards most active flows kept in the cache. Thus, the cache will get higher hit rate. Meanwhile, the cache hit rate will naturally rise up with the increasing cache capacity *NM*, since the cache will store more active flows. On the contrary, the number of packet flows *L* almost has no effect on the cache hit rate, since the cache already contains the most active flows, and more inactive flows will not affect the cache hit rate.

TABLE I
THE ESTIMATED CACHE HIT RATES OF OUR DESIGNED TECACHE

L	q	NM	$CHR_{TECache}$
$100k$	0.9997	$3.5k$	65.01%
$100k$	0.9997	$4k$	69.89%
$100k$	0.9996	$4k$	79.82%
$100k$	0.9996	$4.5k$	83.48%
$200k$	0.9996	$4k$	79.82%
$200k$	0.9996	$4.5k$	83.48%
$200k$	0.9995	$4.5k$	89.47%
$200k$	0.9995	$5k$	91.80%
$400k$	0.9995	$4.5k$	89.47%
$400k$	0.9995	$5k$	91.80%
$400k$	0.9994	$5k$	95.03%
$400k$	0.9994	$5.5k$	96.32%

V. ENERGY-EFFICIENT SDN FLOW TABLE STORAGE ARCHITECTURE

A. Architecture Design

By exploiting the characteristics of different storage media, we propose an energy-efficient flow table storage architecture for SDN switches in Fig. 9, based on our designed traffic-aware energy-saving cache. This architecture primarily consists of the TECache in SRAM, match sub-table in TCAM, and content sub-table in DRAM. The match sub-table and content sub-table are built by accommodating match fields and content fields in a SDN flow table respectively with TCAM and DRAM. The separation of content fields from the flow table enables TCAM with fixed capacity to accommodate more packet flows. This will effectively mitigate the problem of limited TCAM capacity in SDN switches, especially for specific network scenarios with large-scale flow tables. The TECache adopts SRAM to hold active exact flow contained in the flow table, for fast lookup and low energy consumption.

As for an arrived packet, the SDN switch first computes its flow identifier with its key fields, and matches against the TECache. If the match succeeds, we can directly locate the corresponding content sub-entry mapped from the matched cache entry. Then, the packet can be rapidly forwarded in conformity to the action set in the content sub-entry, without TCAM flow table lookup. Otherwise, the switch continues to look up the match sub-table in TCAM. If the lookup succeeds to find out a match sub-entry, we similarly perform packet forwarding in line with the content sub-entry corresponding to the match sub-entry. By this way, a majority of packets within active flows will directly hit the TECache in SRAM with low energy consumption. Only a minority of packets within inactive flows require additional lookup on the match sub-table in TCAM. In summary, the above architecture can significantly reduce the lookup energy consumption of the SDN flow table. Meanwhile, the TECache employs our designed hash algorithm FelisCatus to achieve fast lookup, which ensures favorable lookup performance of the flow table.

In this architecture, the size of SRAM plays a crucial role in both energy efficiency and performance. A larger SRAM can

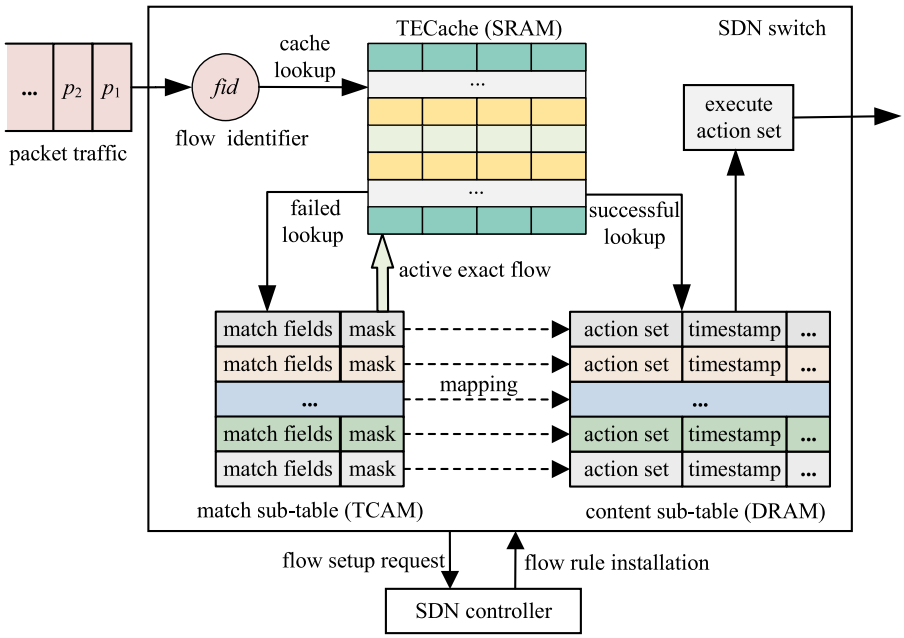

Fig. 9. Energy-efficient SDN flow table storage architecture.

store more active flows, which improves the cache hit rate and reduces TCAM accessing frequency, leading to lower energy consumption. However, both its static power consumption and leakage power rise with the increase of its capacity. This is because larger memory arrays require more circuits to drive access signals (e.g., row and column lines), and the larger SRAM contains more memory cells, which must maintain their state even when not accessed, resulting in higher static power consumption. As for our SRAM cache, it will get less the energy-saving benefits from persistently increasing the cache size. This is because the architecture has already cached most of the active flows, and newly added capacity is primarily used to store less active flows, following the principle of diminishing marginal returns. By optimally setting the size of SRAM cache, we are able to achieve satisfactory energy savings while maintaining favourable lookup performance.

B. Flow Table Operations

1) Packet Forwarding Algorithm: Algorithm 1 provides the pseudo-code implementation of packet forwarding algorithm based on energy-efficient flow table lookup. Upon receiving a packet *p*, a SDN switch first parses its headers at each protocol layer, to extract key fields (e.g., source/destination IP address, source/destination port and protocol type, etc.). Then, we calculate its flow identifier *fid* with these key fields, to generate its flow signature *sig* for looking up the TECache (lines 1-2). If the cache lookup succeeds, we locate the corresponding TCAM match entry *te* by the flow entry index *idx* in the matched cache entry *ce*, and verify whether the packet matches with *te*. Then, we retrieve the corresponding DRAM content entry *de* and forward the packet in accordance with its action set (lines 3-7). Finally, we update the DRAM content entry *de*, including counters and the arrival time of most recently arrived packet. At the same time, we also update the timestamp *tsp* in the matched cache entry *ce* (line 8-9).

If the cache lookup fails, we continue to look up the TCAM match sub-table (line 12). If the TCAM lookup successfully returns a flow entry index, we read its corresponding DRAM content entry *de*, and perform packet forwarding in accordance with its action set (lines 13-15). Then, we examine the data transmission state of the flow which the received packet belongs to. Particularly, we judge whether the received packet belongs to the identical flow as the latest packet of the matched flow entry, and the arrival interval time between these two packets exceeds the preset PIAT threshold. If the flow comes into the active state, we insert it into the cache (lines 16-18). Finally, we update the DRAM content entry *de* including the flow identifier and the arrival time of its latest packet (line 19). If the TCAM lookup fails, it means that the received packet *p* belongs to a new flow. In such case, we encapsulate the packet *p* into a Packet-in message, and send it to the SDN controller for instructions (lines 23-24).

C. Algorithm Performance

1) Energy Saving Rate: To quantify the energy saving effect of a SDN flow table storage architecture, we define its energy saving rate as the declined percentage of its average energy consumption per packet compared to pure TCAM flow table without any optimization. Let E_{TCAM} and E_{EFT} respectively be average energy consumption per packet of pure TCAM flow table and an energy-efficient flow table storage architecture, the energy saving rate can be expressed as $ESR = 1 - E_{TCAM}/E_{EFT}$. Assuming that the TCAM flow table contains l_{TCAM} entries, each of which occupies d_{TCAM} bytes. Let ω_{TCAM} be average energy consumption of per-byte TCAM lookup, we can represent the average lookup

Algorithm 1 Bool PacketForward(Packet p)

```
key ← ParsePacket(p)
sig ← Hash(key)
ce.idx ← CacheLookup(sig)
if ce.idx ≠ NULL then
    te ← ReadTcamEntry(ce.idx)
    de ← ReadDramEntry(ce.idx)
    ExecuteActions(de.actions)
    UpdateFlow(de)
    ce.tsp ← Normalize(p.time)
    return true
end if
index ← QueryTcam(key)
if index is valid then
    de ← ReadDramEntry(index)
    ExecuteActions(de.actions)
    if fid = de.fid && p.time − de.time < PIAT_THR
    then
        f ← NewFlow(fid, idx, p.tsp)
        CacheInsert(f)
        UpdateFlow(de)
        return true
    end if
end if
msg ← CreatePacketInMessage(p)
SendMessageToController(msg)
return false
```

energy consumption of pure TCAM flow table per packet as:

$$E_{TCAM} = l_{TCAM}\, d_{TCAM} \omega_{TCAM} \tag{11}$$

As for an energy-efficient flow table storage architecture based on caching, its energy saving effect mainly depends on its cache hit rate. Suppose its cache hit rate p and the average energy consumption of its cache lookup E_{cache}, we can signify its average energy consumption per packet as $E_{EFT} = E_{cache} + (1-p)E_{TCAM}$. Consequently, its energy saving rate can be derived as:

$$ESR_{EFT} = 1 - \frac{E_{EFT}}{E_{TCAM}} = p - \frac{E_{cache}}{E_{TCAM}} \tag{12}$$

As for our proposed energy-efficient flow table storage architecture EFT-TECache, the application-specific integrated circuit (ASIC) in an SDN switch performs a hash computation, for each packet to determine its position in the TECache cache. Assuming that each hash bucket of the TECache contains $l_{TECache}$ entries, each packet needs to match with $3l_{TECache}$ entries for its cache lookup. Let the flow signature in each cache entry take up d_{FS} bytes, with ω_{SRAM} denoting the energy consumption per byte for an SRAM lookup, and E_{ASIC} representing the energy consumption for a single hash computation performed by the ASIC. Then, we can obtain the average energy consumption of the TECache per packet as $E_{TECache} = 3l_{TECache} d_{FS} \omega_{SRAM} + E_{ASIC}$. Let $p_{TECache}$ be the hit rate of the TECache, we can infer the energy saving rate of our proposed architecture EFT-TECache in (13) from (12).

$$ESR_{EFT-TECache} = p_{TECache} - \frac{3l_{TECache} d_{FS} \omega_{SRAM} + E_{ASIC}}{l_{TCAM}\, d_{TCAM} \omega_{TCAM}} \tag{13}$$

As for the energy-efficient flow table storage architecture based on Cuckoo cache EFT-Cuckoo [26], each packet needs to match with $2l_{Cuckoo}$ entries in the Cuckoo cache, with the assumption of l_{Cuckoo} entry in its each hash bucket. With the average accessing energy consumption of the Cuckoo cache in SRAM per byte $\omega_{SRAM}$and the width of flow signature d_{FS}, we can formulate the energy consumption of the Cuckoo cache as $E_{Cuckoo} = 3l_{Cuckoo} d_{FS} \omega_{SRAM}$. Let p_{Cuckoo} be the hit rate of the Cuckoo cache, we can derive the energy saving rate of the EFT-Cuckoo architecture in (14) from (12), where r_{ST} signifies the ratio of the average accessing energy consumption of SRAM memory ω_{SRAM} to that of TCAM memory ω_{TCAM}.

$$ESR_{EFT-Cuckoo} = p_{Cuckoo} - \frac{2l_{Cuckoo} d_{FS} \omega_{SRAM} + 2E_{ASIC}}{l_{TCAM}\, d_{TCAM} \omega_{TCAM}} \tag{14}$$

As for the energy-efficient flow table storage architecture based on elastic energy-saving cache EFT-EEC [28], the ASIC needs to perform one hash computation in each segment of the EEC cache, for each packet to determine its position. Suppose that the EEC cache consists of k_{EEC} segments, each packet needs to perform k_{EEC} times hash computation and match against k_{EEC} entries for its cache lookup. With the average accessing energy consumption of the EEC cache in SRAM per byte ω_{SRAM} and the width of flow signature d_{FS}, the energy consumption for a single hash computation performed by the ASIC E_{ASIC}, we can gain the energy consumption of the EEC cache as $E_{\mathrm{EEC}} = k_{\mathrm{EEC}} d_{FS} \omega_{\mathrm{SRAM}} + k_{\mathrm{EEC}} E_{\mathrm{ASIC}}$. Let p_{EEC} be the hit rate of the EEC cache, we can deduce the energy saving rate of the EFT-EEC architecture in (15) from (12).

$$ESR_{EFT-EEC} = p_{EEC} - \frac{k_{EEC} d_{FS} \omega_{SRAM} + k_{EEC} E_{ASIC}}{l_{TCAM}\, d_{TCAM} \omega_{TCAM}} \tag{15}$$

Taking the P4 Language Specification v1.0.2 as an example, the matching field of each flow entry occupies d_{TCAM}=13bytes in the IP routing scenario. Based on extensive investigations and actual measurements on the energy consumption of various memory technologies [23], [31], [32], ω_{SRAM} and ω_{TCAM} can be respectively set to 0.05 $\mu J/byte$, 2.0 $\mu J/byte$, and ASIC hash computation is set to 0.12 $mJ/operation$. In addition, the matching field of the flow table entry is compressed into a shorter flow signature by a hashing algorithm, which is usually configured with length d_{FS}=4bytes. Furthermore, the Cuckoo cache adopts the same hash bucket size as TECache, whereas the EEC cache is configured with a number of segments that is twice the hash bucket size of TECache. With these parameter configurations, we can estimate the energy saving rates of the above three

TABLE II
THE ENERGY SAVING RATES OF THREE ENERGY-EFFICIENT FLOW TABLE STORAGE ARCHITECTURES

l_{TCAM}	$l_{TECache}$	l_{Cuckoo}	k_{EEC}	$p_{TECache}$	p_{EEC}	p_{Cuckoo}	$ESR_{EFT-TECache}$	$ESR_{EFT-EEC}$	$ESR_{EFT-Cuckoo}$
$6K$	2	2	4	0.80	0.75	0.70	79.921%	74.691%	69.844%
$6K$	2	2	4	0.85	0.80	0.75	84.921%	79.691%	74.844%
$6K$	2	2	4	0.90	0.85	0.80	89.921%	84.691%	79.844%
$6K$	4	4	8	0.80	0.75	0.70	79.918%	74.381%	69.843%
$6K$	4	4	8	0.85	0.80	0.75	84.921%	79.381%	74.843%
$6K$	4	4	8	0.90	0.85	0.80	89.921%	84.382%	79.843%
$8K$	2	2	4	0.80	0.75	0.70	79.940%	74.768%	69.883%
$8K$	2	2	4	0.85	0.80	0.75	84.940%	79.768%	74.883%
$8K$	2	2	4	0.90	0.85	0.80	89.940%	84.768%	79.883%
$8K$	4	4	8	0.80	0.75	0.70	79.938%	74.535%	69.882%
$8K$	4	4	8	0.85	0.80	0.75	84.938%	79.535%	74.882%
$8K$	4	4	8	0.90	0.85	0.80	89.938%	84.535%	79.882%

energy-efficient flow table storage architectures in Table II, by setting different typical values for other parameters.

As shown in Table II, our proposed EFT-TECache architecture achieves higher energy saving rates than the other two flow table storage architectures. This is because the EFT-TECache architecture can achieve higher cache hit rate compared to the other ones, since it always sufficiently exploits cache space to accommodate the most active flows in network traffic, and it only needs to perform hashing once for each lookup. Meanwhile, we can see from Table II that the energy saving rate of the flow table storage architecture primarily depends on the cache hit rate and is almost regardless of other factors. This is chiefly attributed to the fact that a majority of packets will directly hit the cache with low energy consumption, and only a minority of packets need to look up the TCAM flow table with high energy consumption.

2) Packet Forwarding Delay: Packet forwarding delay is a critical performance metric for evaluating SDN flow table storage architecture. As for our proposed EFT-TECache architecture, it first looks up our designed traffic-aware energy-saving cache for each arrived packet. The cache lookup first performs one hashing by ASIC with the time overhead t_{ASIC}. Then, it will take the search length $(l_{TECache}+1)/2$, if we successfully find out a flow in its directly mapped bucket with $l_{TECache}$ entries. Otherwise, we need to look up the directly mapped bucket with the search length $l_{TECache}$, and continue to look up its two adjacent buckets in parallel. Suppose that there is equal probability to find out a matched flow in its directly mapped bucket and two adjacent buckets, we can compute the packet forwarding delay of successful cache lookup in (16), with each accessing time of SRAM t_{SRAM}.

$$\begin{aligned} SFD_{TECache} &= \frac{1}{2}\left[t_{ASIC}+\frac{1}{2}(l_{TECache}+1)t_{SRAM}\right] \\ &\quad +\frac{1}{2}\left[(t_{ASIC}+l_{TECache}t_{SRAM}) \right. \\ &\quad \left. +\frac{1}{2}(l_{TECache}+1)t_{SRAM}\right] \\ &= t_{ASIC}+\left(l_{TECache}+\frac{1}{2}\right)t_{SRAM} \end{aligned} \tag{16}$$

As for failed cache lookup, we can obtain its forwarding delay $FFD_{TECache}=t_{ASIC}+2l_{TECache}t_{SRAM}$ as it needs to perform hashing and traverse its directly mapped bucket and two adjacent buckets. Then, it needs to further look up the TCAM flow table with the time overhead t_{TCAM}. In short, we can represent the packet forwarding delay of the EFT-TECache architecture $PFD_{EFT-TECache}$ in (17) with the cache hit rate $p_{TECache}$.

$$\begin{aligned} PFD_{EFT-TECache} &= p_{TECache}SFD_{TECache} \\ &\quad +(1-p_{TECache})(FFD_{TECache}+t_{TCAM}) \\ &= t_{ASIC}+\left(2-p_{TECache}+\frac{1}{2l_{TECache}}\right)l_{TECache}t_{SRAM} \\ &\quad +(1-p_{TECache})t_{TCAM} \end{aligned} \tag{17}$$

As for the energy-efficient flow table architecture based on Cuckoo cache EFT-Cuckoo [26], the successful lookup of a packet within its directly mapped bucket in the cache is similar to that of the TECache with the forwarding delay $t_{ASIC}+(l_{Cuckoo}+1)t_{SRAM}/2$. Otherwise, we need to look up the directly mapped bucket with the forwarding delay $l_{Cuckoo}t_{SRAM}$, and perform hashing with the time cost to look up its other bucket with the forwarding delay $t_{ASIC}+(l_{Cuckoo}+1)t_{SRAM}/2$. Then, we can compute the forwarding delay of successful Cuckoo cache lookup in (18).

$$\begin{aligned} SFD_{cuckoo} &= \frac{1}{2}\left[t_{ASIC}+\frac{1}{2}(l_{cuckoo}+1)t_{SRAM}\right] \\ &\quad +\frac{1}{2}\left[(t_{ASIC}+l_{cuckoo}t_{SRAM}) \right. \\ &\quad \left. +\frac{1}{2}(l_{cuckoo}+1)t_{SRAM}\right] \\ &= \frac{3}{2}t_{ASIC}+\left(l_{cuckoo}+\frac{1}{2}\right)t_{SRAM} \end{aligned} \tag{18}$$

As for failed cache lookup, we can obtain its forwarding delay $FFD_{Cuckoo}=2t_{ASIC}+2l_{Cuckoo}t_{SRAM}$ as it needs to perform hashing twice and traverse its two candidate buckets. Then, it needs to further look up the TCAM flow table with the time cost t_{TCAM}. In conclusion, we can calculate the packet forwarding delay of the EFT-Cuckoo architecture $PFD_{EFT-Cuckoo}$ in (19) with the cache hit rate p_{Cuckoo}.

$$\begin{aligned} &PFD_{EFT-Cuckoo} \\ &=\left(2-p_{Cuckoo}+\frac{1}{2}p_{Cuckoo}\right)l_{Cuckoo}t_{SRAM} \\ &\quad +\left(2-\frac{1}{2}p_{Cuckoo}\right)t_{ASIC}+(1-p_{Cuckoo})t_{TCAM} \end{aligned} \tag{19}$$

As for the energy-efficient flow table architecture based on elastic energy-saving cache EFT-EEC [28], it first looks up the EEC cache for each arrived packet. Assume that the EEC cache contains k_{EEC} segments, we can obtain the forwarding delay of successful and failed cache lookup respectively as $SFD_{EEC} = (1+k_{EEC})(t_{ASIC}+t_{SRAM})/2$ and $FFD_{EEC} = k_{EEC}(t_{ASIC} + t_{SRAM})$, since it needs to performed hashing and matching for each cache segment. In conclusion, we can express the packet forwarding delay of the EFT-EEC architecture $PFD_{EFT-EEC}$ in (20) with the cache hit rate p_{EEC}.

$$PFD_{EFT-EEC} = (1 - p_{EEC})t_{TCAM} \left[k_{EEC} + \frac{p_{EEC}(1 - k_{EEC})}{2}\right](t_{ASIC} + t_{SRAM}) \quad (20)$$

According to extensive memory research [32], [33], [34], the accessing time of SRAM, TCAM and ASIC is typically set to 2, 12.5 and 3 nanoseconds, respectively. By setting different typical values for other parameters, we can estimate the packet forwarding delay of the three energy-efficient flow table storage architectures in Table III.

As shown in Table III, our proposed EFT-TECache architecture achieves less packet forwarding than the other ones. In particular, our proposed EFT-TECache architecture perform flow table lookup with the packet forwarding delay below 10*us*, for the case that each hash bucket only contains 1 entry. This is due to the fact that the EFT-TECache architecture only needs to perform hashing once for each lookup. In contrast, the EFT-Cuckoo architecture needs to perform one more hashing to locate two candidate buckets. Additionally, the EFT-EEC architecture needs to perform hashing once for each segment.

VI. Experiments

A. Experimental Methodology

In our experiments, we construct a dedicated platform with SDN controller and switch to evaluate the performance of TECache in different network environments. The experimental platform is mainly composed of SDN controller ONOS (Open Network Operating System), SDN switch Pica8 P-3297, traffic trace generator Spirent TestCenter and monitoring device. It aims to simulate a variety of typical network traffic scenarios such as edge networks and Internet of things. Pica8 P-3297 is a prevalent SDN switch supporting OpenFlow 1.3 protocol with high performance forwarding capability and flexible flow table configuration function. The ONOS controller is connected to a Pica8 P-3297 SDN switch via 10 Gigabit Ethernet. Spirent TestCenter generates different types of network traffic, simulates two network scenarios of edge network and IoT, and injects them into SDN switches to test the performance of switches in these two scenarios. At the same time, the monitoring device is connected to the exit port of the switch for real-time monitoring of traffic behavior. Fig. 10 exhibits the topological map of the experimental platform.

In our experiments, we first implement and deploy various energy-efficient flow table storage architecture in the SDN switch, including our proposed EFT-TECache architecture. As for our designed FelisCatus hash algorithm, we optimize it for both performance and code size by inlining critical functions, employing efficient bit-level operations, and using compact data structures. We also update the firmware to PicOS 2.9.3 and apply a custom patch to enable hardware-accelerated hash operations, ensuring robust SDN mode operation. Then, we generate traffic traces for two types of network scenarios by Spirent TestCenter. As for edge network trace, we simulate the communication between edge computing devices characterized by moderate traffic density, where most inter-packet intervals range from 250*ms* to 500*ms* with substantial fluctuations. As for IoT network trace, we emulate data transmissions from low-power sensors with low traffic density, where the majority of packets have expanded inter-packet intervals (PIT >500*ms*) with pronounced periodic pattern. The generated traffic traces are directly injected into the Pica8 P-3297 SDN switch via Spirent TestCenter. Lastly, we record the classification performance of each packet, and periodically calculate performance metrics such as cache hit rate and energy saving rate for each energy-efficient SDN flow table storage architecture.

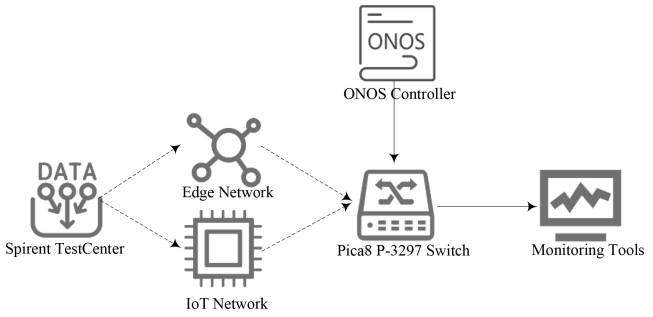

Fig. 10. The Topology of our experimental platform.

To ensure the reliability of our statistical experimental results, each configuration was rigorously evaluated through 10 independent experiments under identical initial conditions. Performance metrics were periodically calculated over a standardized 1-second interval. These results are presented as mean values with superimposed error bars denoting the 95% confidence intervals derived from Student's t-distribution, providing a transparent visualization of measurement dispersion and enhancing the reproducibility claims of our methodology.

B. Energy-Saving Cache Performance

1) Cache Utilization Rate: We compare the cache utilization rate of our designed TECache with those of the Cuckoo cache [26] and the EEC cache [28], by simulating traffic traces generated from various network scenarios. As for edge network trace, we set the capacity of the TECache, the Cuckoo cache and EEC cache as 3.5*K* entries. The Cuckoo cache is configured with 2 hash functions, the number threshold of kicking operations 5, the hash table length 512, and 7 entries in each hash bucket. The EEC cache for the edge network trace is configured with a maximum of 7 segments, each of which has 512 entries. As for IoT network trace, we set the capacity of the TECache, the Cuckoo cache and EEC cache as 4*K* entries. The Cuckoo cache is configured with 2 hash

TABLE III
THE ESTIMATION OF PACKET FORWARDING DELAY FOR ENERGY-EFFICIENT FLOW TABLE STORAGE ARCHITECTURES

$l_{TECache}$	l_{Cuckoo}	k_{EEC}	$p_{TECache}$	p_{Cuckoo}	p_{EEC}	$PFD_{EFT-TECache}$	$PFD_{EFT-Cuckoo}$	$PFD_{EFT-EEC}$
1	1	2	0.72	0.66	0.69	9.96*us*	12.94*us*	12.15*us*
1	1	2	0.74	0.68	0.71	10.14*us*	13.03*us*	12.25*us*
1	1	2	0.76	0.70	0.73	10.22*us*	13.13*us*	12.35*us*
2	2	4	0.78	0.72	0.75	10.42*us*	13.42*us*	12.55*us*
2	2	4	0.80	0.74	0.77	10.5*us*	13.52*us*	12.65*us*
2	2	4	0.82	0.76	0.79	10.58*us*	13.62*us*	12.75*us*
4	4	8	0.84	0.78	0.81	10.78*us*	13.83*us*	12.95*us*
4	4	8	0.86	0.80	0.83	10.86*us*	13.93*us*	13.05*us*
4	4	8	0.88	0.82	0.85	10.94*us*	14.03*us*	13.15*us*

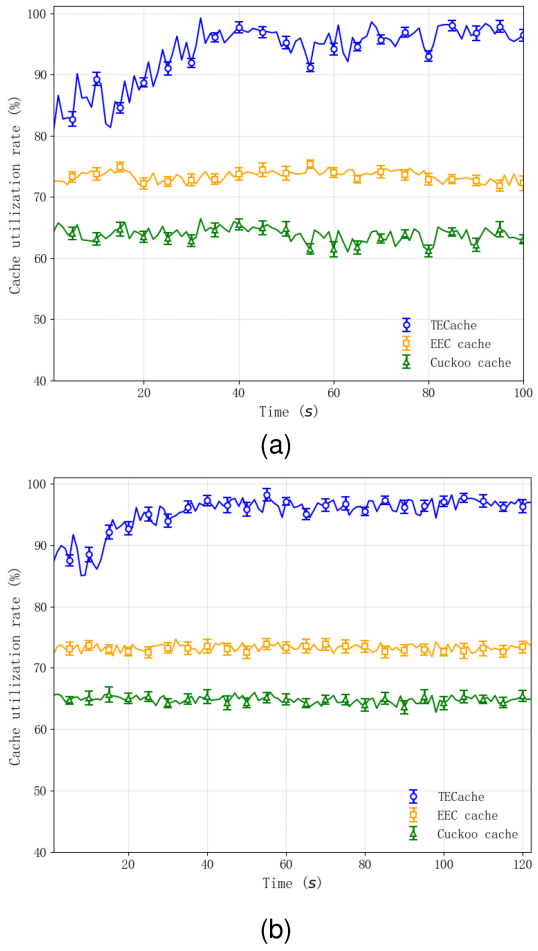

Fig. 11. The Utilization rates of different energy-saving caches.: a)Edge network trace; b)IoT network trace.

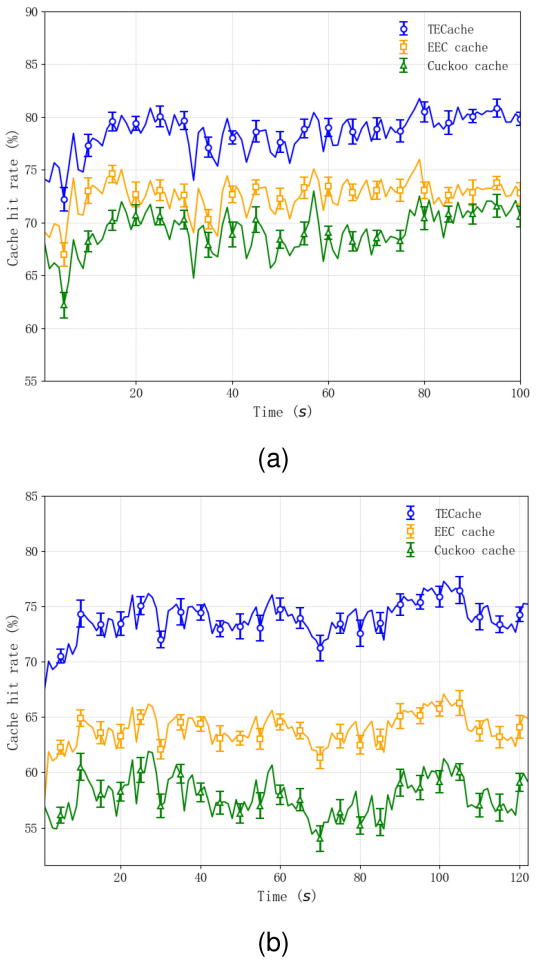

Fig. 12. The Hit rates of different energy-saving caches.: a)Edge network trace; b)IoT network trace.

functions, the number threshold of kicking operations 4, the hash table length 1024, and 4 entries in each hash bucket. The EEC cache for the IoT network trace is configured with a maximum of 8 segments, each of which has 512 entries. With these parameter configurations, we respectively operate the three energy-saving caches on the two traces in the SDN switch, and obtain their cache utilization rates in Fig. 11.

As seen from Fig. 11, the TECache achieves much higher cache utilization rates than the EEC cache and Cuckoo cache, regardless of network traffic traces. Particularly, our designed TECache obtains extremely high cache utilization rates close to 100%. This is because the TECache almost perfectly solves hash collisions by adjacent hopping and co-directional kicking. Meanwhile, the TECache matches the number of active flows with its capacity, by dynamically adjusting the packet inter-arrival time threshold. In contrast, the Cuckoo cache is difficult to achieve favorable cache utilization rate, since there are growing possibilities to fall into cyclic kicking state with the increasing number of kicking operations. Additionally, the EEC cache achieves relatively low cache utilization rates, due to its original hashing without any optimization of its hash collisions.

2) Cache Hit Rate: With the same parameter settings as above, we respectively perform the three energy-saving caches on our simulated network traffic traces. Each experiment was

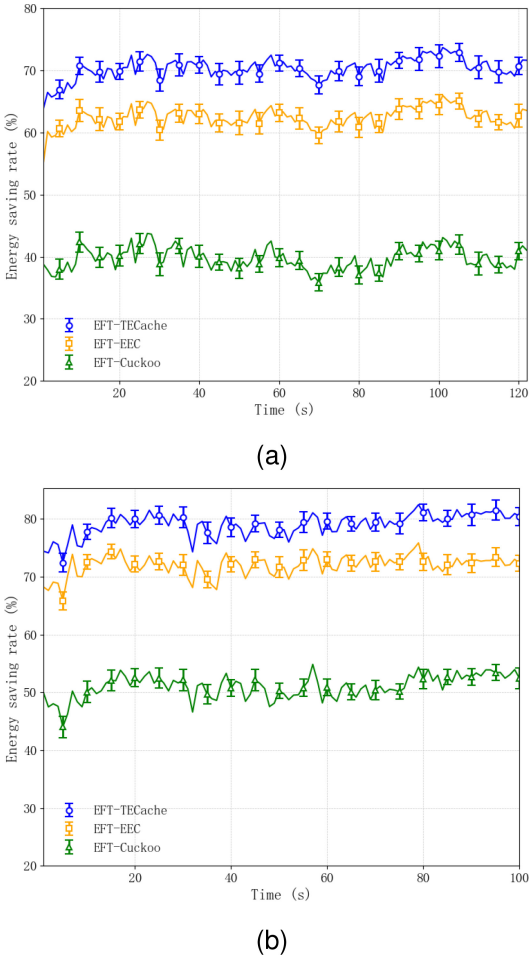

Fig. 13. The Energy saving rate of different flow table storage architectures.: a)Edge network trace; b)IoT network trace.

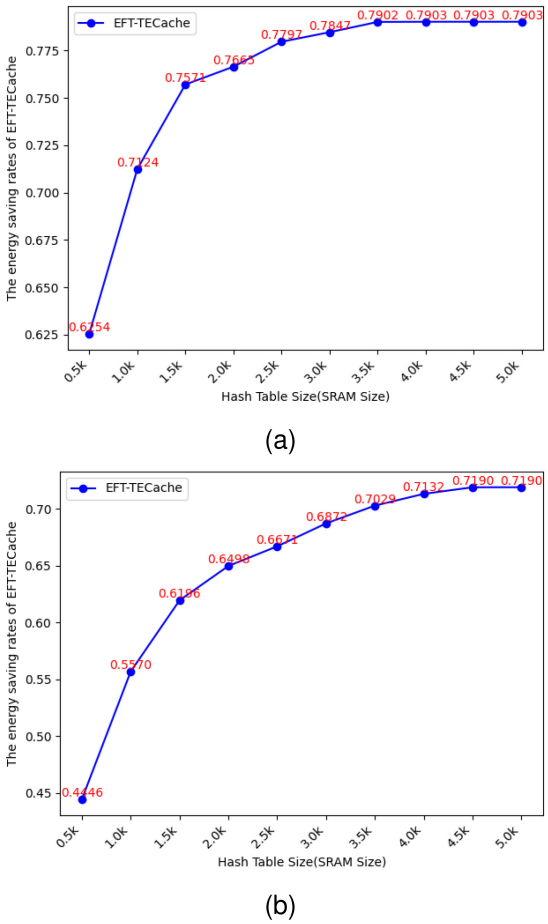

Fig. 14. The Energy saving rates of EFT-TECache with different size of the hash table(SRAM cache).: a)Edge network trace; b)IoT network trace.

repeated for ten times to compute 95% confidence intervals, and record their cache hit rates in Fig. 12. As shown in Fig. 12, our designed TECache achieves higher and more stable hit rates than the EEC cache and the Cuckoo cache, regardless of network traffic traces. As for the edge network trace, the TECache obtains average cache hit rate 78.49%±1.31%, while 72.51%±1.10% for the EEC cache and 69.15%±1.43% for the Cuckoo cache. As for the IoT network trace, the TECache gets average cache hit rate 73.89%±1.14%, while 70.98%±1.03% for the EEC cache and 57.81%±1.25% for the Cuckoo cache.This is chiefly attributed to the fact that the TECache adequately exploits cache space to accommodate most active flows in network traffic. In contrast, the EEC cache holds relatively poor cache utilization and the Cuckoo cache cannot adapt to the fluctuating number of active flows, leading to their relatively low cache hit rates.

C. Flow Table Storage Architecture Performance

1) Energy Saving Rate: With the above parameter configurations for the three energy-saving caches and the capacity of the TCAM flow table set as $8K$, we respectively perform the energy-efficient flow table lookup of the three storage architectures EFT-TECache, EFT-EEC [28] and EFT-Cuckoo [26]. We repeat each experiment for ten times and obtain their energy saving rates with 95% confidence intervals in Fig. 13. As for the edge network trace, the EFT-TECache architecture obtains average energy saving rate 79.02%±1.37%, while 71.99%±1.23% for the EFT-EEC architecture and 51.01%±1.43% for the EFT-Cuckoo architecture. As for the IoT network trace, the EFT-TECache architecture gets average energy saving rate 71.32%±1.18%, while 62.40%±1.13% for the EFT-EEC architecture and 39.68%±1.34% for the EFT-Cuckoo architecture.This is primarily due to high cache hit rates of our designed TECache and the fact that each packet hitting the cache only needs to match against three candidate buckets at most. In contrast, the EFT-EEC architecture obtains lower cache hit rates, and each cache lookup requires parallel matching in all cache segments. As for the EFT-Cuckoo architecture, its cache hit rates are significantly lower than those of the TECache, resulting in its poor energy-saving effect.

As for our proposed EFT-TECache, we further evaluate the impact of hash table size on its energy saving rates with our simulated network traffic traces in Fig. 14. As shown in Fig. 14, the energy saving rates increase with the growth of the hash table size on the whole. Specifically, the energy saving rate sharply improves for both network traffic traces when the hash table size increases from $0.5k$ to $2.5k$. This reflects the

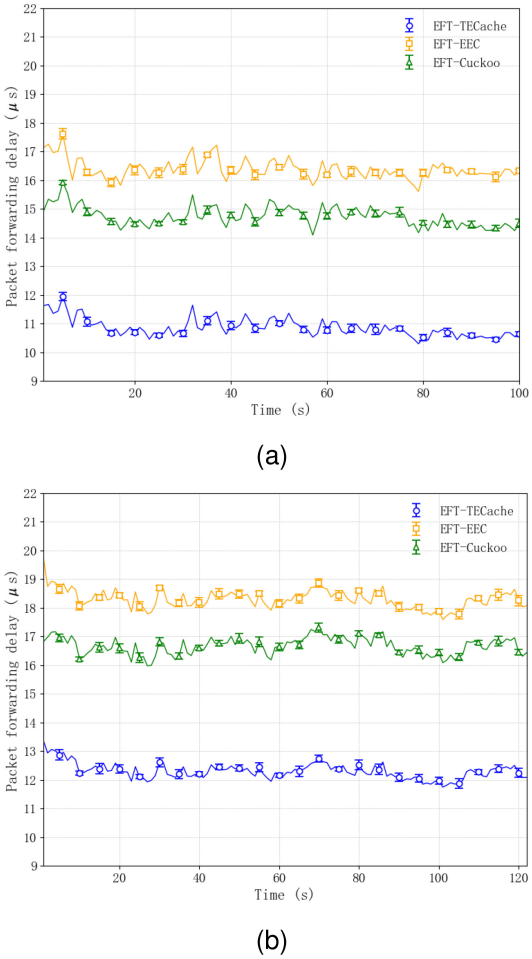

Fig. 15. The Packet forwarding delay of different energy-efficient flow table storage architectures.: a)Edge network trace; b)IoT network trace.

critical role of caching active flows in energy saving effect. However, the improvement diminishes and tends to be stable for the hash table size beyond $3.5k$, which indicates that further hash table expansion brings about limited benefit. This is due to the fact that more hash table only holds less active flows, which produces a slight increase in the energy saving rate. Therefore, it is essential to select an appropriate hash table size for specific SDN deployment scenarios.

2) Packet Forwarding Delay: With the same parameter settings as above, we respectively execute the energy-efficient flow table lookup of the three storage architectures. We repeat each experiment for ten times and obtain their packet forwarding delay with 95% confidence intervals in Fig. 15. As seen from Fig. 15, our proposed EFT-TEC architecture obtains much shorter packet forwarding delay than the other ones. As for the edge network, the EFT-TECache architecture obtains packet forwarding delay $10.87\pm0.23us$, while $16.39\pm0.25us$ for the EFT-EEC architecture and $14.74 \pm 0.25us$ for the EFT-Cuckoo architecture. As for the IoT network trace, the EFT-TECache architecture gets packet forwarding delay $12.31\pm0.20us$, while $18.30\pm0.23us$ for the EFT-EEC architecture and $16.67\pm0.21us$ for the EFT-Cuckoo architecture. This is attributed to relatively high hit rates of our designed TECache, which implies that most packets directly hit the cache without further lookup on the TCAM flow table. Meanwhile, our proposed EFT-TECache architecture only needs to perform hashing once for each cache lookup. In striking contrast, the EFT-Cuckoo architecture has to operate one more hashing to locate two candidate buckets in its energy-saving cache, and the EFT-EEC architecture is requested to execute hashing once for each cache segment.

VII. Conclusion

To mitigate unfavorable energy-saving effect of existing energy-efficient flow table storage architecture especially under network traffic fluctuation, we design traffic-aware energy-saving cache with optimal utilization, and propose an energy-efficient flow table storage architecture for SDN switches. In the architecture, we first design an energy-saving cache with low-power memory to store active flows, enabling most packets to bypass energy-intensive TCAM flow table lookup. Then, we devise a nearly conflict-free hash algorithm FelisCatus for the energy-saving cache, which provides three candidate buckets for each inserted flow by adjacent hopping, and looks for a vacancy in the cache for each conflicting flow by co-directional kicking. Meanwhile, we propose an adaptive adjustment mechanism of flow activity criterion, which dynamically adjusts the packet inter-arrival time threshold of active flows along with its step size, for adapting the number of active flows to cache capacity.

In our experiments, we evaluate the performance of our designed traffic-aware energy-saving cache and energy-efficient flow table storage architecture, by network traffic traces with significant fluctuations. Experimental results indicate that: (a) Our designed cache achieves high hit rates up to around 75%, with the increase of 5.97% and 9.34% respectively compared to those of the EEC one and the Cuckoo one; (b) Our proposed energy-efficient flow table storage architecture obtains high energy saving rates close to 74%, with the increase of 7.02% and 28% respectively in comparison to those of the EFT-EEC one and the EFT-Cuckoo one; (c) Our proposed flow table storage architecture always has less packet forwarding delay around $11us$. In conclusion, our proposed flow table storage architecture achieves favorable and stable energy-saving effects while implementing fast flow table lookup, even for fluctuating network traffic.

References

[1] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015.
[2] S. Date et al., "SDN-accelerated HPC infrastructure for scientific research," *Int. J. Inf. Technol.*, vol. 22, no. 1, p. 30, 2016.
[3] T. Koponen et al., "Network virtualization in multi-tenant datacenters," in *Proc. 11th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2014, pp. 203–216.
[4] D. Firestone, "VFP: A virtual switch platform for host SDN in the public cloud," in *Proc. 14th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2017, pp. 315–328.
[5] X.-N. Nguyen, D. Saucez, C. Barakat, and T. Turletti, "Rules placement problem in OpenFlow networks: A survey," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 2, pp. 1273–1286, 2nd Quart., 2016.
[6] S. Y. Qiao and C. C. Hu, "A mechanism of taming the flow table overflow in OpenFlow switch," *Chin. J. Comput.*, vol. 41, no. 9, pp. 2003–2015, 2018.

[7] H. Lim, K. Lim, N. Lee, and K.-H. Park, "On adding bloom filters to longest prefix matching algorithms," *IEEE Trans. Comput.*, vol. 63, no. 2, pp. 411–423, Feb. 2014.
[8] S.-C. Kao, D.-Y. Lee, T.-S. Chen, and A.-Y. Wu, "Dynamically updatable ternary segmented aging bloom filter for OpenFlow-compliant low-power packet processing," *IEEE/ACM Trans. Netw.*, vol. 26, no. 2, pp. 1004–1017, Apr. 2018.
[9] Y. Han, S.-S. Seo, J. Li, J. Hyun, J.-H. Yoo, and J. W.-K. Hong, "Software defined networking-based traffic engineering for data center networks," in *Proc. 16th Asia–Pacific Netw. Oper. Manag. Symp.*, 2014, pp. 1–6.
[10] K. Kannan and S. Banerjee, "Compact TCAM: Flow entry compaction in TCAM for power aware SDN," in *Proc. Int. Conf. Distrib. Comput. Netw. (ICDCN)*, 2013, pp. 439–444.
[11] S. Banerjee and K. Kannan, "Tag-In-Tag: Efficient flow table management in SDN switches," in *Proc. 10th IEEE Int. Conf. Netw. Service Manag. (CNSM)*, Rio de Janeiro, Brazil, 2014, pp. 109–117.
[12] Z. Wang, M. Liao, and X. Ji, "Source-controlled OpenFlow data plane," *J. Commun.*, vol. 2015, no. 3, pp. 181–187, 2015.
[13] C. Zhang, P. Sun, G. Hu, and L. Zhu, "RETCAM: An efficient TCAM compression model for flow table of OpenFlow," *J. Commun. Netw.*, vol. 22, no. 6, pp. 484–492, Dec. 2020.
[14] V. S. M. Srinivasavarma, S. R. Pydi, and S. N. Mahammad, "Hardware-based multi-match packet classification in NIDS: An overview and novel extensions for improving the energy efficiency of TCAM-based classifiers," *J. Supercomput.*, vol. 78, no. 11, pp. 13086–13121, 2022.
[15] M.-H. Cheng, W.-S. Hwang, Y.-J. Wu, C.-H. Lin, and J.-S. Syu, "An effective flow-rule-reducing algorithm for flow tables in software-defined networks," in *Proc. Int. Comput. Symp. (ICS)*, Tainan, China, 2020, pp. 25–30.
[16] R. Wu, W. K. Jia, and X. Wang, "Header-translation based flow aggregation for scattered address allocating SDNs," in *Proc. IEEE Conf. Dependable Secure Comput. (DSC)*, Fukushima, Japan, 2021, pp. 1–8.
[17] N. Saha, S. Misra, and S. Bera, "Q-Flag: QoS-aware flow-rule aggregation in software-defined IoT networks," *IEEE Internet Things J.*, vol. 9, no. 7, pp. 4899–4906, Apr. 2022.
[18] Z. Li and Y. Hu, "PASR: An efficient flow forwarding scheme based on segment routing in software-defined networking," *IEEE Access*, vol. 8, pp. 10907–10914, 2020.
[19] H. Khanmirza, "WildMinnie: compression of software-defined networking (SDN) rules with wildcard patterns," *PeerJ Comput. Sci.*, vol. 8, no. 809, pp. 1–30, 2022.
[20] C. Q. Li, Y. Q. Dong, and X. G. Wu, "OpenFlow table lookup scheme integrating multiple-cell hash table with TCAM," *J. Commun.*, vol. 37, no. 10, pp. 128–140, 2016.
[21] J. C. Mogul et al., "DevoFlow: Cost-effective flow management for high performance enterprise networks," in *Proc. 9th ACM SIGCOMM Workshop Hot Topics Netw. (HotNets)*, New York, NY, USA, 2010, pp. 1–6.
[22] W. J. Li et al., "Decision tree based pre-classifier for energy-efficient TCAM based packet classification," *Appl. Res. Comput.*, vol. 38, no. 1, pp. 237–241, 2021.
[23] P. T. Congdon, P. Mohapatra, M. Farrens, and V. Akella, "Simultaneously reducing latency and power consumption in OpenFlow switches," *IEEE/ACM Trans. Netw.*, vol. 22, no. 3, pp. 1007–1020, Jun. 2014.
[24] B. G. Assefa and Ö. Özkasap, "A survey of energy efficiency in SDN: Software-based methods and optimization models," *J. Netw. Comput. Appl.*, vol. 137, pp. 127–143, Jul. 2019.
[25] "Network traffic traces," 2024. [Online]. Available: http://iptas.edu.cn/src/system.php
[26] B. Xiong, Z. Hu, Y. Luo, and J. Wang, "CuckooFlow: Achieving fast packet classification for virtual openflow switching by exploiting network traffic locality," in *Proc. IEEE Int. Conf. Parallel Distrib. Process. Appl. Big Data Cloud Comput. Sustain. Comput. Commun. Social Comput. Netw. (ISPA/BDCloud/SocialCom/SustainCom)*, 2019, pp. 1123–1130.
[27] J.-F. Wang, X. He, S.-Z. Si, H. Zhao, C. Zheng, and H. Yu, "Using complex network theory for temporal locality in network traffic flows," *Physica A Stat. Mechan. Appl.*, vol. 524, pp. 722–736, Jun. 2019.
[28] B. Xiong et al., "Elastically accelerating lookup on virtual SDN flow tables for software-defined cloud gateways," *Comput. Netw.*, vol. 238, Jan. 2024, Art. no. 110092.
[29] J. Tao, M. Kunze, and W. Karl, "Evaluating the cache architecture of multicore processors," in *Proc. 16th Euromicro Conf. Parallel, Distrib. Netw.-Based Process. (PDP)*, 2008, pp. 12–19.
[30] B. Xiong, R. Wu, J. Zhao, and J. Wang, "Efficient differentiated storage architecture for large-scale flow tables in software-defined wide-area networks," *IEEE Access*, vol. 7, pp. 141193–141208, 2019.
[31] R. Panigrahy and S. Sharma, "Reducing TCAM power consumption and increasing throughput," in *Proc. 10th Symp. High Perform. Interconnects*, 2002, pp. 107–112.
[32] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Amsterdam, The Netherlands: Elsevier, 2011.
[33] O. Panait, L. Dumitriu, and I. Susnea, "Hardware and software architecture for accelerating hash functions based on SoC," in *Proc. 22nd Int. Conf. Control Syst. Comput. Sci. (CSCS)*, Bucharest, Romania, 2019, pp. 136–139.
[34] H. Song, J. Turner, and J. Lockwood, "Shape shifting tries for faster IP route lookup," in *Proc. 13th IEEE Int. Conf. Netw. Protocols (ICNP)*, Boston, MA, USA, 2005, p. 10.

Bing Xiong received the Ph.D. degree in computer science by master-doctorate program from the Huazhong University of Science and Technology, China, in 2009, and the B.S. degree from Hubei Normal University, China, in 2004. He has been working with the Changsha University of Science and Technology, China, since 2010, where he is currently an Associate Professor with the School of Computer Science and Technology. He worked as a Visiting Scholar with the Department of Computer and Information Science, Temple University, USA, from 2018 to 2019. His main research interests include future network architecture, network traffic measurements, and artificial intelligence applications.

Guanglong Hu received the B.S. degree in network engineering from the Hunan Institute of Technology, China, in 2022. He is currently pursuing the M.S. degree with the School of Computer Science and Technology, Changsha University of Science and Technology, China. His main research interests include software-defined networking, packet classification, and green communications.

Songyu Liu received the degree from the School of Computer Science and Technology, Changsha University of Science and Technology, China, in 2021. His research interests focus on software-defined networking, packet classification, and green communications.

Jinyuan Zhao received the Ph.D. degree in computer science from Central South University, China, in 2020, and the M.S. degree from Huazhong Normal University, China, in 2007. She worked with the School of Computer and Communication, Hunan Institute of Engineering, China, from 2007 to 2020. She is currently an Associate Professor with the School of Information Science and Engineering, Changsha Normal University, China. Her main research interests include furture network architecture, software-defined networking, and network traffic measurements.

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 the Changsha University of Science and Technology since 2021.

Baokang Zhao received the B.S., M.S., and Ph.D. degrees in computer science from the National University of Defense Technology, where he is currently an Associate Professor with the School of Computer Science. His research interests include system design, protocols, algorithms, and security issues in computer networks.

Keqin Li (Fellow, IEEE) is a SUNY Distinguished Professor of computer science with the State University of New York at New Paltz. He is also a National Distinguished Professor with Hunan University, China. 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. 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 is also a Member of Academia Europaea (Academician of the Academy of Europe). He is an AAAS Fellow, and an AAIA Fellow.