T-Cache: Efficient Policy-Based Forwarding Using Small TCAM

Ying Wan, Haoyu Song, *Senior Member, IEEE*, Yang Xu, *Senior Member, IEEE*, Yilun Wang, Tian Pan, *Senior Member, IEEE*, Chuwen Zhang, Yi Wang, *Senior Member, IEEE*, and Bin Liu, *Senior Member, IEEE*

***Abstract*— Ternary Content Addressable Memory (TCAM) is widely used by modern routers and switches to support policy-based forwarding due to its incomparable lookup speed and flexible matching patterns. However, the limited TCAM capacity does not scale with the ever-increasing rule table size due to the high hardware cost and high power consumption. At present, using TCAM just as a rule cache is an appealing solution, but one must resolve several tricky issues including the rule dependency and the associated TCAM updates. In this paper, we propose a new approach which can generate dependency-free rules to cache. By removing the rule dependency, the complex TCAM update problem also disappears. We provide the complete T-cache system design including slow path processing and cache replacement, and implement a T-cache prototype on Barefoot Tofino switches. We conduct comprehensive software simulations and hardware experiments based on real-world and synthesized rule tables and packet traces to show that T-cache is efficient and robust for network traffic in various scenarios.**

***Index Terms*— Ternary rule, cache, TCAM, forwarding.**

I. Introduction

POLICY-BASED forwarding uses a set of packet header fields as the flow ID to match against a predefined rule set and applies the associated policy of the best match to the packet. Today, as the Software-Defined Networking (SDN) [2] prevails, network operation is evolving to be more and more application-aware and service-oriented. As an indispensable component in modern routers and switches, policy-based forwarding plays the roles far beyond the conventional Access Control List (ACL) [3] and Firewall (FW). To name a few, Service Function Chaining [4] uses the rule set to classify network traffic and applies different service chains to different flows; Segment Routing [5] uses the rule set to forward packets on different paths with a source routing mechanism; Network Slicing [6] uses the rule set to assign user flows to different virtual layers which bear different QoS treatments; Network Telemetry [7] uses the rule set to pick specific packets for behavior monitoring and performance measurement.

However, policy-based forwarding is also a notoriously challenging problem. A rule is usually an aggregation of multiple flows and rules may overlap. As a result, a packet can match multiple rules and the matching rule with the highest priority needs to be found. This process must be fast enough to sustain the line-speed forwarding. Unlike address-based forwarding, traditionally policy-based forwarding lacks efficient algorithmic solutions to sustain the ever-increasing network throughput which is now in the magnitude of terabits per second per device.

The recourse to Ternary Content Addressable Memory (TCAM) is effective for now. Using TCAM for policy-based forwarding has become the *de facto* industry standard for two reasons: (1) TCAM allows a search key extracted from an incoming packet to compare with all the stored rules in parallel, thus ensuring line-speed forwarding; (2) TCAM rules support a wide range of patterns, including exact matching, Longest Prefix Match (LPM), and range matching, which are flexible enough for policy representation.

However, on the one hand, TCAM has long been criticized for high hardware cost and high power consumption. TCAM roughly consumes 100× more hardware resources and 100× more power than DRAM of the same capacity [8]. As a result, TCAM's capacity is limited. Most commercial switches can only support a relatively small number of rules in TCAM, ranging from a few hundreds to ten thousands [8].

Moreover, TCAM update, especially for rule insertion, is a slow process during which the lookup is paused and a number of existing rules in TCAM need to be relocated due to the rule dependency [9], which poses serious challenges to dynamic policy deployment in large networks [10]–[16] where the update latency requirement is stringent. Fig. 1 shows the measurements on a high-end commercial switch EdgeCore Wedge 100BF-32X [17]. For a 1K-entry TCAM-based rule table, it takes less than 40us to insert a rule that does not

Manuscript received December 16, 2020; revised April 27, 2021 and July 3, 2021; accepted July 8, 2021; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor D. Hay. Date of publication July 28, 2021; date of current version December 17, 2021. The work of Ying Wan, Yi Wang, and Bin Liu was supported in part by the National Natural Science Foundation of China under Grant 62032013, Grant 61872213, and Grant 61432009; and in part by the Guangdong Basic and Applied Basic Research Foundation under Grant 2019B1515120031. An early version of this work [1] was presented in part at the Annual IEEE International Conference on Computer Communications (INFOCOM), 2020. That paper has been modified and revised to reflect comments received at the conference. *(Corresponding authors: Yi Wang; Bin Liu.)*

Ying Wan, Yilun Wang, and Chuwen Zhang are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: wany16@mails.tsinghua.edu.cn).

Haoyu Song is with Futurewei Technologies, Santa Clara, CA 95050 USA (e-mail: shykcl@gmail.com).

Yang Xu is with the School of Computer Science, Fudan University, Shanghai 200433, China (e-mail: xuy@fudan.edu.cn).

Tian Pan is with the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China.

Yi Wang is with the University Key Laboratory of Advanced Wireless Communications of Guangdong Province, Southern University of Science and Technology, Shenzhen 518055, China, and also with the PCL Research Center of Networks and Communications, Peng Cheng Laboratory, Shenzhen 518066, China (e-mail: wy@ieee.org).

Bin Liu is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China, and also with the PCL Research Center of Networks and Communications, Peng Cheng Laboratory, Shenzhen 518066, China (e-mail: liub@mail.tsinghua.edu.cn).

Digital Object Identifier 10.1109/TNET.2021.3098320

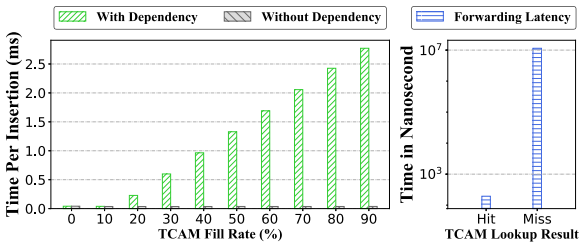


Fig. 1. Measurement results on a Tofino switch.

overlap with the other rules regardless of the TCAM fill rate. However, the time to insert a rule that overlaps with the other rules becomes longer and longer as the TCAM fill rate increases. When the TCAM fill rate reaches 90%, the insertion of an overlapping rule requires almost 3ms. It is reported that some other switches, such as NoviSwitch 1132, Pica8 P-3290, Dell 8132F, and HP 5406zl, support less than 50 rule insertions per second [18], [19].

The imminent end of Moore's Law [20] implies that we must accept the limit on the TCAM capacity in a single chip as a starting point to design a policy-based forwarding system that can meet the throughput and service requirements. An embedded TCAM with around ten thousand entries is at the upper end of affordability, while the network services need to handle tens of thousands of rules and millions of flows. The pressing issue of scalability begs for new solutions.

We have two options to this end: (1) Make better use of the available TCAM resource through architectural optimizations, or (2) Seek alternative algorithmic solutions which can replace TCAM with cheaper and relatively abundant memory such as eDRAM and SRAM.

The first option is the current research focus. The key idea is to use TCAM as a rule cache on the fast path to hold only a subset of rules. Those packets that cannot find a matching rule in TCAM are punted to the slow path (*e.g.*, the control CPU) for forwarding decision. The rationale of this approach comes from the observation that the typical traffic presents a strongly skewed distribution: a small percentage of active flows claim most of the traffic while a long tail of active flows contributes just a small portion of traffic [21], [22]. It follows that the "hot" rules matched by majority packets at any time are only a small subset of the rule set. If we keep only hot rules in TCAM dynamically, a much larger rule table can be supported without compromising the throughput.

Two key factors determine the performance of a caching system: cache hit-rate and update speed. A low hit-rate means a large number of unmatched packets are punted to the control CPU, which not only consumes the limited and precious bandwidth between the data plane and control plane but also leads to a much longer forwarding delay. As shown in Fig. 1, the packet forwarding delay is only a few hundreds of nanoseconds if the packet is matched in TCAM; however, it can increase to 10ms if the packet is punted to the control CPU. Therefore, a high hit-rate is of significant importance.

On the other hand, the rules cached in TCAM need be updated periodically to maintain the high hit-rate. Previous works on TCAM caching systems mainly consider the policies for rule replacement to achieve a high hit-rate, but overlook the cost associated with the updates. Moreover, to the best of our knowledge, no existing algorithm is well suitable for TCAM batch update when TCAM is used as a cache, due to the excessive computing time or rule moves.

In this paper, we make several critical design decisions and develop the corresponding algorithms which make our system significantly outperform the existing works in terms of cache hit-rate and update speed. The novelty and superior performance of the resulting T-cache system mainly lie in two points. First, the system constantly measures the "heat" of rules and keeps only the "hottest" rules in TCAM, which ensures the best possible cache hit-rate. Second, the rules in TCAM are purposely made dependency-free (*i.e.*, no rule overlaps) through an isolate-rule construction process.

Although the construction of dependency-free rules requires additional computation, the time for generating m dependency-free rules to fill TCAM is less than that for choosing m rules under dependency constraints. Also, dependency-free rules are cached only if they are hot, leading to high hit-rate, but we cannot provide such guarantee for dependent rules. More important, dependency-free rules can be placed anywhere in TCAM, eliminating the complex TCAM update problem.

We exhibit the advantages of T-cache in terms of cache hit-rate and update speed by comparing it with the state-of-the-art schemes through software simulation using the real-world policies and packet traces. The results show that T-cache consumes a $\sim$200$\times$ shorter computation time to generate the isolate rules to fill the TCAM, and achieves a 20%$\sim$50% higher cache hit-rate than the state-of-the-art schemes. Meanwhile, the rule insertion time is constant and much shorter than the other schemes, thanks to the elimination of rule dependency in T-cache.

We also implement a T-cache prototype on Barefoot Tofino switches EdgeCore Wedge 100BF-32X. The experimental results demonstrate that a 1K-entry TCAM for a forwarding table of 760K IP prefixes in the switch data plane can sustain a 99.8% hit-rate on an OC-192 backbone link. T-cache also consumes 20$\sim$60$\times$ less time than the state-of-the-art solutions for TCAM refresh when the time window is set to 30 seconds.

II. Background

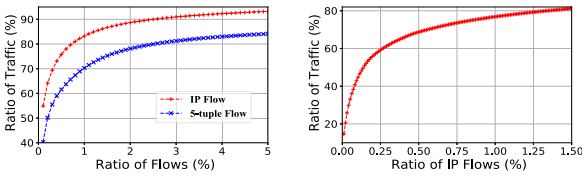
To have an effective rule caching system for policy-based forwarding, some conditions (*e.g.*, traffic temporal and spatial locality) need to hold and several challenges (*e.g.*, rule dependency and the subsequent TCAM update/refresh problem) need to be addressed. This section provides the necessary background.

A. Viability of Rule Caching

We need to ensure a small TCAM cache populated with the hottest rules can indeed handle a large enough portion of total traffic. In other words, the traffic punted to slow path needs to be small enough to not inundate the slow path processor and the overall line-speed forwarding can be guaranteed.

We study rule sets and traffic traces from various real network environments and find that the network traffic flows generally follow a Zipf distribution [23] at any given time.

An archived CAIDA Internet traffic trace was collected from an OC-192 backbone link of a Tier 1 ISP at Equinix datacenter on March 15, 2018 at 13:00 UTC [24]. The trace lasts one minute and the number of concurrent 5-tuple flows reaches 10^6. We measure the accumulated traffic ratio contributed by the top flows. Fig. 2(a) shows that 70% of traffic attributes to the top 1% of flows. Similar results hold for the CERNET IP traces collected by IPTAS on March 8, 2018 at 12:56 CST [25], as shown in Fig. 2(b).



(a) Equinix from CAIDA. (b) CERNET from IPTAS.

Fig. 2. Temporal locality of real-world network traces.

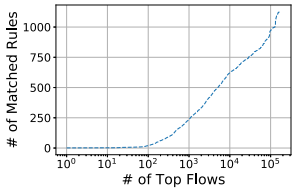


Fig. 3. Spatial locality of flows.

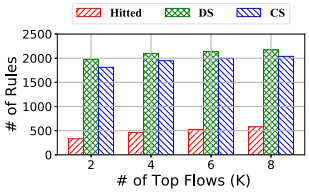
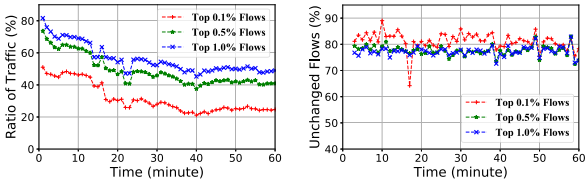


Fig. 4. Expansion of cached rules.



(a) Variability of traffic of top flows. (b) Variability of top flows.

Fig. 5. High turnover of the top flows on Equinix trace.

Numerous previous studies also confirm the existence of such temporal locality [21]–[23], [26]–[31]. In addition, we notice that the traffic also presents spatial locality. The flows tend to cluster in the matching space, making multiple flows match the same rule. As shown in Fig. 3, with more than 10^5 concurrent IP flows and 760K rules for the Equinix trace and table, the number of the matching rules is only 1,126. Similar findings are reported in some previous works [27], [30].

Such temporal and spatial locality of flow distribution is critical to support a TCAM-based rule cache.

B. Rule Dependency

Although we have established the viability of rule caching with TCAM, TCAM's unique features make the actual implementation of such a cache complex and challenging.

Since the ternary rules may overlap, if a rule r is to be inserted into TCAM, all the precedent rules of r in the dependency graph of the rule set, even if some of them are cold, must be loaded into TCAM with r simultaneously for the correctness of lookups. These rules form the *dependent-set* (DS) of r [8]. Apparently, the need of loading the whole dependent-set defies our design principle and causes inefficiency in terms of TCAM space and operation. A measurement in [27] shows that, in a rule set generated by ClassBench [32], the average number of dependent rules for each rule is 350.

CacheFlow [8] alleviates the challenge of dependent-set to some extent by introducing the concept of *cover-set* (CS). A rule r's cover-set is composed of the rules that directly overlap with r in r's dependent-set. Only the rules in r's cover-set are to be loaded into TCAM along with r. To ensure the correctness, however, the matching action for rules in a cover-set must be modified to punt the matching packets to slow path which contains the entire rule table for further lookups, essentially making a match on any rule in a cover-set a cache miss. Since TCAM avoids storing full dependent-sets, the saved TCAM space can be used to cache more hot rules and the overall cache hit-rate is expected to improve. However, the effectiveness of this optimization heavily relies on the length of the dependency chain. When the dependency graph of r has a large fan-out but short paths, the benefit is limited.

A test on the Equinix datacenter on March 15, 2018 at 13:00 UTC exemplifies this point. As shown in Fig. 4, in a minute the top 2K flows only hit 332 rules out of the 760K rules. The number of matching rules increases at a much slower pace when more top flows are considered, in favor of good cache performance. However, due to the rule dependency, caching these hot rules in TCAM actually requires 1,971 rules to be cached to cover the dependent-set. The expansion ratio is almost six times. The cover-set optimization does not help much: 1,892 rules remain to be cached. The above optimizations also complicate the cache replacement. Multiple rules need to be evicted and care needs to be taken to avoid destroying the dependency relationship among rules in the cache. If a rule is evicted, all its descendent rules in the dependency graph of the rule set must be evicted too.

C. TCAM Insertion and Update Challenges

A more challenging problem is inserting new rules into the cache. Due to the priority order requirement of overlapped rules, a rule r cannot be inserted into an arbitrary TCAM entry. All the descendent and precedent rules of r in TCAM must be identified so that r can be inserted in between. If no such entry is available, a series of entry moves which require complex calculation is necessary [8]. The calculation consumes the slow path computing resource; the updates block TCAM lookups; the delay stales the cache. Given that a high and constant cache refresh rate is expected, the update issue poses to be the major hurdle for realizing a TCAM-based rule cache.

We show the inefficiency of TCAM updates by the average number of TCAM entry moves due to a rule insertion. A measurement in [9] shows that inserting a single ACL rule for a 1K rule set generated by ClassBench [32] requires up to 466 entry moves. Although the state-of-the-art optimizations [9], [33] require on average less than 10 entry moves per rule insertion, these algorithms are time consuming, which require hundreds of milliseconds to compute a moving scheme [34].

This update performance hardly meets the requirement of cache refresh that is performed when a cold rule becomes hot or vice versa. In fact, the traffic ratio contributed by a set of top flows fluctuates greatly. Fig. 5(a) shows an example of traffic ratio contributed by top flows over time on a real network trace. When using a one-minute time window, the traffic contributed by the initial 1% of top flows keeps dropping from 81% to 48% in an hour. Fig. 5(b) provides an explanation. Only around 80% of the current top flows can keep its status in the next time window. The high turnover of the top flows

leads to the high turnover of hot rules, requesting frequent cache replacements or TCAM updates.

D. Cache Operation Mode

In a classic cache, any cache miss will result in an immediate cache replacement and the cache replacement only happens on a cache miss. This operation mode does not quite suit the rule cache. A missed rule in the cache may be cold. Moving a cold rule into TCAM is at best useless and at worst harmful. A good criterion for cache replacement is to always replace colder rules with hotter ones whenever such a condition is found based on the current traffic.

Therefore, the ideal rule cache should differ from the classic cache in at least two aspects. First, the rule cache does not have to load every missed rule into the cache when a cache miss happens. Second, the system constantly seeks opportunities to load rules that become hot into TCAM to replace rules that become cold. Of course, the accurate measurement of rule popularity is a must, which requires extra resources and work. Nevertheless, this *measurement-based* approach is more robust for a consistent and predictable rule cache performance.

III. Related Work

TCAM is widely used for rule-based table lookup (*e.g.*, routing [35]–[37] and packet classification [38], [39]), and becomes a standard component in SDN switches. However, due to its high hardware cost and power consumption, the limited TCAM capacity may fail to accommodate complete rule tables [40]. The inefficient support of ranges in TCAM makes the situation worse [41]. Some researchers tackle this issue by trying to compress the rule tables [42]–[45]. Such approaches are proved to be NP-hard by Rottenstreich *et al.* [45]. Moreover, most of these works assume static rule sets and they cannot handle the dynamic rule sets fast enough when the rules change frequently. Many efforts are made to reduce the number of TCAM entries needed to hold a rule with ranges [46]–[50]. This kind of work does not fundamentally solve the problem as the gap between the TCAM capacity and rule table size broadens. More important, since the rule dependency remains after the optimization, applying such techniques is still challenging for rule updates.

Some other researchers thus turn their attention to the direction of using TCAM as a cache. The pioneer works mostly use TCAM to cache exact flows [41], [51], [52]. Caching exact flows, although simple, is an inefficient use of TCAM. Dong *et al*. therefore try to merge some top flows into ternary rules to improve the TCAM utilization [21]. However, their scheme requires traversing all the generated rules in TCAM to decide whether a newly identified top flow can be merged in. This process, with the assumption of a stable rule table, fails to keep up with the high churn rate of rules in today's SDN networks.

Recent works focus on caching popular rules, taking care of the rule dependency issue with cover-set [8], [53]. Based on CacheFlow [8], Sheu *et al*. design a sophisticated algorithm to select the cached rules for better TCAM hit-rate [53]. Bienkowski *et al*. conduct an in-depth theoretical analysis of the caching problem when the to-be-cached items have dependencies and propose an online caching algorithm [54].

However, these works overlook the potential performance impact of TCAM updates due to rule insertion. Ding *et al*. take the update cost of a rule into consideration when choosing cached rules [55]. The TCAM operation optimization is at a cost of lower TCAM hit-rate. Yan *et al*. resolve the rule dependency issue by partitioning the field space into logical buckets, and cache buckets along with all the associated rules [27]. Yu *et al*. partition the rule set into new non-overlapping rules in the controller and cache them into underlying switches [56]. These schemes are all computing-intensive. Li *et al*. do not consider the rule dependency when selecting the rules to be cached so the cache achieves a high hit-rate and avoids the update problem [57]. The consequence is that it requires every packet to be sent to the control plane for further process regardless of the matching status in TCAM, nullifying the purpose of the cache.

The TCAM update issue persists as long as there is dependency between the cached rules. It has been extensively studied since TCAM is used for forwarding lookups [58]–[60]. Some works optimize the update process by reducing the redundant updates at the controller level [61]–[64]; some other works aim to optimize the individual rule insertion process by designing algorithms to avoid unnecessary entry moves [9], [33], [34], [59], [60], [65]. These solutions either fail to achieve the ideal TCAM entry move reduction [34], [59], [60], [65], or require excessive computation time [9], [33]. Besides, these works assume the TCAM updates are driven by sporadic rule set changes. However, the TCAM updates driven by cache replacements are more frequent and bursty, making the cache schemes based on dependent-set or cover-set difficult to sustain. Some algorithms [11], [66] are alleged to support batch updates, but in fact they still rely on the individual update techniques mentioned above and the results are barely satisfactory for a TCAM-based cache.

Put in perspective, T-cache differs from the existing works in several aspects: (1) Since the rule table compression techniques cannot fundamentally close the gap between the TCAM capacity and the rule table size, T-cache focuses on using TCAM as a cache; (2) Since the rule dependency not only causes the TCAM update issue but also complicates the selection of cached rules, T-cache uses a rule isolation method to deliberately remove any rule dependency; (3) In order to cope with frequent rule and traffic pattern changes, T-cache uses fast algorithms to track the changes and speed up the calculation of cached rules; (4) Thanks to the dependency-free cached rules, T-cache can use a specific algorithm for fast TCAM refreshing at a low cost.

IV. Isolate Rule

A critical question we ask is if we can eliminate the rule dependency altogether at a low cost. If it is possible, both challenges (*i.e.*, dependent set and insertion update) disappear.

A strawman solution exists along this line of thinking. Instead of considering the heat of rules, we can shift our focus on the heat of flows, *i.e.*, consider the flows that claim most of the packets during a period of time. Now the rule caching problem is transformed into a flow caching problem.

A flow can be imagined as a point in the multi-dimensional space occupied by its best matching rule. Clearly, flows are independent and never overlap, so they can be stored in arbitrary locations in TCAM. The only drawback is that storing exact flows in TCAM does not take advantage of the "ternary" aggregation feature boasted by TCAM. If the TCAM capacity is m entries, then at best we can cache the top m flows. The percentage of traffic composed by these flows determines the upper bound of the cache performance.

Yet we can do better than this by introducing the concept of *isolate-space*. For a flow f and its best matching rule r, we define the isolate-space $S_{r,f}$ as the sub-space of r that covers f but does not overlap with r's cover-set.

Assume we find that a flow f is hot enough to deserve a cache entry. Instead of caching f or the dependent-set or cover-set of f's best matching rule r, we cache an *isolate rule*. For a hot flow f and the corresponding isolate-space $S_{r,f}$, we define an isolate rule as a single TCAM-cacheable rule r', which is the largest multi-dimensional box that is fully contained by $S_{r,f}$ and contains f.

Clearly, r' can be inserted into arbitrary TCAM entry due to its independence and meanwhile r''s maximized volume allows it to cover more fate-sharing flows.

Depending on where f is located in the space, the resulting r' may differ. To support the case that two flows f_1 and f_2, while sharing the same best matching rule r, cannot be covered by a single isolate-rule but both claim a cache entry, two isolate-rules r'_1 and r'_2 can be generated accordingly. These two isolate-rules may overlap, but they can be considered independent because they inherit the same action from r. This important feature assures us that, at any time, when a new isolate-rule is generated, we can freely insert it into any TCAM entry without considering the existing isolate-rules in TCAM.

Of course, any update on r and r's cover-set may nullify one or more isolate-rules generated from r. In this case, to simplify the processing, we simply remove all these isolate-rules from TCAM.

A. Analysis of Isolate-Rule Calculation

Since an isolate-rule occupies a continuous space and can be stored by a single TCAM entry, it is easy to see that each dimension of the isolate-rule can be represented as a prefix or an exact value (an exact value is a special case of prefix).

We assume all the rules in the original rule set are also prefix-based. As a common practice, any range-based rule can be converted into a set of independent prefix-based rules.

Let f_i^r denote the range length on the rule r's i-th dimension of the k dimensions. If the full range of the rule set's i-th dimension covers L_i bits, let l_i^r denote the length of the prefix representing r's range on i-th dimension. The volume of r is:

$$V(r) = \prod_{i=1}^{k} f_i^r = 2^{\log_2 \prod_{i=1}^{k} f_i^r} = 2^{\sum_{i=1}^{k} \log_2 f_i^r} = 2^{\sum_{i=1}^{k} (L_i - l_i^r)} = 2^{L - \sum_{i=1}^{k} l_i^r} \quad (1)$$

If f's best matching rule is r and $C(r)$ represents the cover-set of r, we introduce Lemma 1 and Lemma 2 to explain the method to calculate r'.

Lemma 1: For any rule r_j in $C(r)$, f mismatches it on at least one dimension.

Proof: This is obvious. If f matches r_j on every dimension, r_j should be f's best matching rule because r_j has a higher priority than r. ∎

Lemma 2: For any rule r_j in $C(r)$, if f mismatches r_j on the i-th dimension and the position of the first bit that makes f mismatch r_j is x, x must be larger than l_i^r.

Proof: This can be proved by contradiction. If $x \leq l_i^r$, the x-th bits of r and r_j on the i-th dimension are different. This means r does not overlap with r_j on the i-th dimension, so r_j cannot belong to $C(r)$. ∎

Since r' is fully contained in r, we know that, for each dimension i, $l_i^{r'} \geq l_i^r$ and the first l_i^r bits of r' must be equal to that of r. Since r' covers f, the first $l_i^{r'}$ bits of r' on the i-th dimension must be equal to the corresponding bits of f. Our goal is to maximize $V(r')$ without causing any overlaps between r' and rules in $C(r)$.

Basically, Lemma 1 and Lemma 2 tell us that it is possible to eliminate the overlap of r' and a rule in $C(r)$ by increasing $\{l_i^{r'}\}$ on any mismatched dimension. Note that two rules are considered overlapping in the space if they overlap on each dimension.

Since r' can mismatch a rule in $C(r)$ in multiple dimensions, there are multiple possible ways to eliminate the rule overlaps. Since the overlaps between r' and all the rules in $C(r)$ must be avoided, a multitude of combinations of $\{l_i^{r'}\}$ need to be evaluated to maximize $V(r')$. The problem can be solved with a $(\prod_{i=1}^{k} L_i)^n$-step brute-force algorithm which is too expensive for large k and n, where k and n are the number of matching fields and the size of $C(r)$, respectively. In fact, we can prove the following Theorem 1.

Theorem 1: Maximizing $V(r')$ without causing any overlaps between r' and rules in $C(r)$ is NP-hard.

Proof: Given a flow f, its best matching rule r, and r's cover-set $C(r)=\{r_1, \ldots, r_n\}$, we calculate $\{\mathcal{C}_{i,d}\}$, a subset of $C(r)$, which includes rules that will not overlap with r' if $l_i^{r'}$ is set to $d+l_i^r$. The problem of maximizing $V(r')$ while avoiding the overlaps between r' and rules in $C(r)$ can be translated into the following problem: Find a sub-collection $\mathcal{L}$ of $\{\mathcal{C}_{i,d}\}$ that satisfies two requirements: (1) $\cup_{\mathcal{C}_{i,d} \in \mathcal{L}} \mathcal{C}_{i,d} = C(r)$; and (2) $\sum_{\mathcal{C}_{i,d} \in \mathcal{L}} d$ is minimized.

The problem of calculating isolate-rule can be proved to be NP-hard by a reduction from an equivalent NP-hard problem, the Weighted Set Cover Problem (WSCP) [67]. WSCP is formulated as follows. Given a finite universe $\mathcal{U} = \{1, 2, \ldots, n\}$ of n members, a collection of subsets of $\mathcal{U}$ that $\mathcal{S} = \{s_1, s_2, \ldots, s_m\} \wedge \forall i,\ s_i \subseteq \mathcal{U}$, and a weight function $w{:}s \rightarrow \Re^+$ that assigns a positive real weight w_i to each subset s_i, the goal is to find the minimum weight of subcollection of $\mathcal{S}$ whose union is $\mathcal{U}$.

For a given instance of WSCP, we construct an instance for calculating the isolate-rule in the following manner in polynomial time. Each element i in $\mathcal{U}$ is mapped to a unique rule r_i of $C(r)$. Thus, each subset s_i with the weight of w_i corresponds to the subset $\mathcal{C}_{i,w_i}$ of $C(r)$. If the problem of calculating isolate-rule can be solved in polynomial time, *i.e.*, we can find a sub-collection of $\{\mathcal{C}_{i,w_i}\}$ that their union equals to $C(r)$ and the sum of their weights is minimized in polynomial time, which means WSCP can be solved in polynomial time, contradicting with the NP-hardness of WSCP. Therefore, the problem of calculating isolate-rule cannot be solved in polynomial time. Its NP-hardness is proved. ∎

Although the problem of calculating the isolate-rule is NP-hard, due to the limited search space, the optimal solution can still be found quickly through exhaustive search. Some features backed by Lemma 3 and Lemma 4 in the next section can be used to accelerate the search process. Specifically, for the IP prefix table, a fundamental and important type of rule table, an efficient polynomial algorithm, shown in Algorithm 1, exists for isolate-rule generation.

B. Algorithm for Isolate IP Prefix Rule Generation

Rule r in IP prefix tables has only one matching field (*i.e.*, $k = 1$) and its volume $V(r)$ is 2^{32-l^r}, where l^r is the prefix length of r. In this case, given a hot flow f and its matching

Rule	Prefix	Rule	Prefix
R0	***	R3	011
R1	00*	R4	1**
R2	000	R5	110

Flow	Feild	Flow	Field
f0	100	f1	010

Fig. 6. Prefixes and flows.

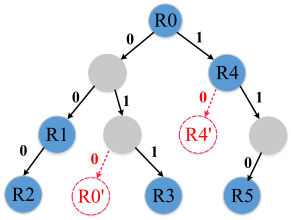


Fig. 7. The prefix tree.

rule r, our target is to generate a rule r' with the shortest possible prefix which does not overlap with rules in $C(r)$.

We use an auxiliary trie to store the prefix tables. Fig. 6 shows a small prefix table with 6 prefixes $\{R0 \sim R5\}$, and 2 hot flows $\{f0, f1\}$ which match $R4$ and $R0$, respectively. Next we show how to generate the isolate-rule $R4'$ and $R0'$.

As shown in Fig. 7, we traverse the trie T_p from the root $T_p.root$ using the bit string of $f0$. If a leaf node is reached, the prefix rule corresponding to the leaf node is returned as the isolate-rule. However, if the traversal stops at an intermediate node, we need to grow a leaf node from this node based on the current bit value of $f0$ and return that node as the isolate-rule. It is trivial to show that the generated prefix r' is exactly the isolate-rule we expect: r' is indeed the shortest prefix within r's isolate space.

Algorithm 1 takes at most 32 steps to generate an isolate-rule. The memory consumption of the trie is also small. Our test shows only a few tens of megabyte used for real IP forwarding tables containing more than 800K prefixes.

C. Algorithm for General Isolate-Rule Generation

Now we describe the isolate-rule generation algorithm for general multi-dimensional rule tables. The pseudo code of isolate-rule generation is given in Algorithm 2. We use Fig. 8(a) as an accompanying example to explain the Algorithm 2. As shown in Fig. 8(a), f's best matching rule is $R1$ and $C(R1)$=$\{R2, R3\}$. (f, $R1$, $C(R1)$) is used as input of Algorithm 2 to generate the isolate-rule r'.

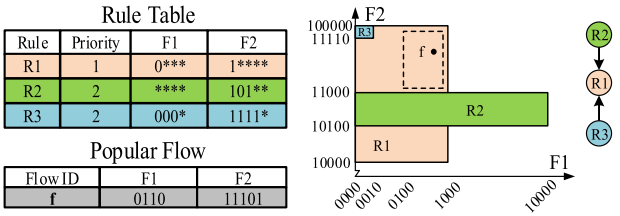
Algorithm 2 first calculates $d_i^{r_j}$ for $1 \le i \le k$ and $1 \le j \le n$ in which k is the number of rule dimensions and n is the number of rules. $d_i^{r_j}$ represents the least number of bits of r' that needs to be further specified based on r on the i-th dimension in order to avoid the overlap between r' and r_j. In our example, f mismatches $R3$ at the second bit position on the first dimension $F1$, so d_1^{R3}=$2-l_1^{R1} = 1$. This means, r' only needs to extend $R1$'s prefix length on $F1$ by one to avoid the overlap between r' and $R3$. For another example, f matches $R2$ on $F1$, so d_1^{R2}=∞, which means the overlap between r' and $R2$ cannot be eliminated no matter what prefix length r' takes on $F1$. Similarly, the results of d_2^{R2} and d_2^{R3} are 1 and 3, respectively.

Algorithm 2 then calculates a collection of subsets of $C(r)$, $\mathcal{C} = \{\mathcal{C}_{i,d}\}$ from $\{d_i^{r_j}\}$. $\mathcal{C}_{i,d}$ consists of the rules that will not overlap with r' if $l_i^{r'}$ is set to $l_i^r + d$. In our example, $\mathcal{C}_{1,1} = \{R3\}$, $\mathcal{C}_{2,1} = \{R2\}$, and $\mathcal{C}_{2,3} = \{R2, R3\}$. Note that it is unnecessary to calculate $\mathcal{C}_{i,d}$ for every i and d (*e.g.*, $\mathcal{C}_{2,2} = \{R2\}$) according to Lemma 3.

Algorithm 1: Isolate IP Prefix Rule Generation (f, T_p)

Input: f: the hot flow; T_p: the prefix tree.
Output: r': the isolate routing rule.

```
curNode = T_p.root, bitIndex = 0, r'[31:0] = '*'
while curNode.left != null or curNode.right != null do
    if f[bitIndex] == '0' then
        if curNode.left == null then
            r'[bitIndex] = '0'
            return r'
        curNode = curNode.left
    else
        if curNode.right == null then
            r'[bitIndex] = '1'
            return r'
        curNode = curNode.right
    r'[bitIndex] = f[bitIndex]
    bitIndex = bitIndex + 1
return r'
```



(a) A rule table and a flow. (b) Rule space and dependency graph.

Fig. 8. An example of calculating isolate-rule.

Next, Algorithm 2 searches for a sub-collection $\mathcal{L}_o$ of $\mathcal{C}$ by calling the function *SolutionSearch*(1, $\mathcal{L}_t$, $\mathcal{L}_o$). $\mathcal{L}_o$ should meet the following requirements: (1) The union of $\mathcal{L}_o$'s elements, $\mathcal{U}_{\mathcal{L}_o}$, equals to $C(r)$, and (2) The sum of d of $\mathcal{L}_o$'s elements, $\mathcal{W}_{\mathcal{L}_o}$, is minimized among all possible $\mathcal{L}_t$ that meets the first requirement. Starting from the first dimension, the search progresses through all the dimensions.

According to Lemma 4, $\mathcal{C}_{i,d_1}$ and $\mathcal{C}_{i,d_2}$ for the same dimension i will not be included in $\mathcal{L}_o$ together. Hence, for each dimension i, $\mathcal{L}_o$ either includes just one $\mathcal{C}_{i,d}$ or none which means $l_i^{r'}$ is identical to l_i^r. During the search, if the current $\mathcal{W}_{\mathcal{L}_t}$ is equal to or greater than $\mathcal{W}_{\mathcal{L}_o}$, there is no need to continue the evaluation on this dimension and proceed to the next dimension. Instead, we should turn back to the previous dimension and continue searching from there, because the optimal $\mathcal{L}_o$ does not exist in the skipped search space. In fact, this is also why Algorithm 2 uses $\sum_{\mathcal{C}_{i,d} \in \mathcal{L}} d$ instead of $\sum_{i=1}^{k} l_i^{r'} = \sum_{i=1}^{k} l_i^r + \sum_{\mathcal{C}_{i,d} \in \mathcal{L}} d$ to measure the quality of $\mathcal{L}$. Although there is no fundamental difference between the two parameters since $\sum_{i=1}^{k} l_i^r$ is a constant, we are in favor of the first one because the second one always needs to search every dimension. In our example, three candidate solutions $\mathcal{L}_1 = \{\mathcal{C}_{1,1}, \mathcal{C}_{2,1}\}$, $\mathcal{L}_2 = \{\mathcal{C}_{1,1}, \mathcal{C}_{2,3}\}$, and $\mathcal{L}_3 = \{\mathcal{C}_{2,3}\}$ are

Algorithm 2: General Isolate-Rule Generation (*f*,*r*,*C(r)*)

Input: f: the hot flow; r: the matching rule; *C(r)*: the collection of rules on which r depends.

Output: r': the isolate rule.

$C(r)=\{r_1, ..., r_n\}$, $r_j=(f_1^{r_j}, ..., f_k^{r_j})$, $f_i^{r_j}=(p_i^{r_j}, l_i^{r_j})$

$d_i^{r_j}$:f mismatches r_j in the $l_i^r+d_i^{r_j}$ bit of i-th dimension

$\mathcal{D}=\{\mathcal{D}_i \mid 1\le i\le k\}$, $\mathcal{D}_i=\{d_i^{r_j} \mid 1\le j\le n\}$

$\mathcal{C}=\{\mathcal{C}_{i,d} \mid 1\le i\le k,\ d\in\mathcal{D}_i\}$, $\mathcal{C}_{i,d}=\{r_j \mid d_i^{r_j}\le d\}$

$\mathcal{L}$: subset of $\mathcal{C}$; $\mathcal{U}_{\mathcal{L}}$: union of $\mathcal{L}$; $\mathcal{W}_{\mathcal{L}}$: sum of d in $\mathcal{L}$;

$\mathcal{L}_o=\{\mathcal{C}_{1,L_1}, \mathcal{C}_{2,L_2}, ..., \mathcal{C}_{k,L_k}\}$, $\mathcal{L}_t=\varnothing$

SOLUTIONSEARCH1, $\mathcal{L}_t$, $\mathcal{L}_o$

$r'\leftarrow$RULEGENERATION r, f, $\mathcal{L}_o$

return r'

Function *SolutionSearch(i, $\mathcal{L}_t$, $\mathcal{L}_o$)*

  if $i>k$ **then return**;

  if $\mathcal{U}_{\mathcal{L}_t}=C(r)$ **then** $\mathcal{L}_o\leftarrow\mathcal{L}_t$;

  else

    SOLUTIONSEARCHi+1, $\mathcal{L}_t$, $\mathcal{L}_o$

    for $\mathcal{C}_{i,d}\in\mathcal{C}$ in d's ascending order **do**

      if $\mathcal{W}_{\mathcal{L}_t}+d<\mathcal{W}_{\mathcal{L}_o}$ **then**

        SOLUTIONSEARCH *i+1*, $\mathcal{L}_t\cup\mathcal{C}_{i,d}$, $\mathcal{L}_o$

      else break;

Function *RuleGeneration(r, f, $\mathcal{L}_o$)*

  for *(i = 1, i ≤ k, i = i+1)* **do**

    if $\mathcal{C}_{i,d}\in\mathcal{L}_o$ **then** $l_i^{r'}=l_i^r+d$ **else** $l_i^{r'}=l_i^r$;

    specify $f_i^{r'}$ according to f and r

  return r'

possible. However, $\mathcal{L}_1$ should be chosen as the final solution because $\mathcal{W}_{\mathcal{L}_1}<\mathcal{W}_{\mathcal{L}_3}<\mathcal{W}_{\mathcal{L}_2}$.

Now Algorithm 2 can generate r' based on r, f, and $\mathcal{L}_o$. It first determines the prefix length of r' on every dimension. For a dimension i that $l_i^{r'}>l_i^r$, the first $l_i^{r'}$ bits of f is taken as the prefix of r' on dimension i. It specifies those bits according to the values of corresponding bits of f, which ensures f to be covered by r'. In our example, $l_1^{r'}$ is 2 so the first dimension of r' is $01**$. Similarly, the second dimension of r' is $11***$. The resulting r' is illustrated by the dotted box in Fig. 8(b).

Algorithm 2 uses the following lemmas to optimize the process.

Lemma 3: It is unnecessary to calculate $\{\mathcal{C}_{i,d}\}$ for every d that $0\le d\le L_i-l_i^r$. Instead, calculating $\{\mathcal{C}_{i,d}\}$ for every $d=d_i^{r_j}$ that satisfies $r_j\in C(r)$ is enough because $\mathcal{L}_o$ will include only $\mathcal{C}_{i,d}$ for which $d=d_i^{r_j}$.

Proof: This can be proved by contradiction. If $\mathcal{L}_o$ is the optimal solution for r' and it includes one $\mathcal{C}_{i,d'}$ that d' is not equal to any $d_i^{r_j}$ for which $r_j\in C(r)$. In such a case, we can find the $\mathcal{C}_{i,d_i^{r_j}}$ that $d_i^{r_j}$ is the largest one less than d'. According to the definition of $\mathcal{C}_{i,d}$, $\mathcal{C}_{i,d'}$ and $\mathcal{C}_{i,d_i^{r_j}}$ are identical. Therefore, we can use $\mathcal{C}_{i,d_i^{r_j}}$ to replace $\mathcal{C}_{i,d'}$ in $\mathcal{L}_o$ and the corresponding solution is reasonable and better than the original $\mathcal{L}_o$, which contradicts our assumption. ■

Lemma 4: $\mathcal{L}_o$ will never include two elements, $\mathcal{C}_{i,d_1}$ and $\mathcal{C}_{i,d_2}$, from the same dimension.

Proof: This can be proved by contradiction. Assume $\mathcal{L}_o$ is the optimal solution for r' and $\mathcal{L}_o$ includes two elements $\mathcal{C}_{i,d_1}$ and $\mathcal{C}_{i,d_2}$. If $d_1<d_2$, it is easy to see that $\mathcal{C}_{i,d_1}\subset\mathcal{C}_{i,d_2}$, so we can remove $\mathcal{C}_{i,d_1}$ from $\mathcal{L}_o$ and the resulting solution is better than the original $\mathcal{L}_o$, which contradicts our assumption. ■

V. T-CACHE ARCHITECTURE

Above we explained a core concept of T-cache that it only caches isolate-rules in TCAM. As a complete rule caching system, T-cache also has the following two essential aspects: (1) T-cache can identify the rules that are getting cold in TCAM, creating opportunities for swapping in new generated isolate-rules; (2) T-cache can identify the hot flows that are not yet covered by the TCAM, and generate and cache isolate-rules accordingly. In this section, we describe the details of these functions.

A. Find Cold Rules in TCAM

Each entry in TCAM has an associated counter which records the number of times the rule in the entry is matched. A rule is considered to be cold if during a period of time, its matching counter is smaller than a threshold.

In previous works, all the matching counters for rules in TCAM and in slow-path processor need to be periodically polled and sorted. A shorter polling period consumes more bandwidth, while a longer polling period may reduce the cache freshness. It is difficult to evade such an approach when rule dependency exists because of the fate-sharing of the dependent rules. A rule with a lower hit-rate in TCAM does not necessarily mean it has a higher chance being evicted. All the rules that r depends on share the same fate as r so their status must also be checked. Fortunately, T-cache only stores isolate-rules in TCAM, allowing decisions to be made on individual rules. To avoid polling all TCAM counters and sorting them as in existing works which consume high bandwidth and cause long delay, we design a lightweight hashing algorithm to accurately identify the cold rules at a low cost.

A hash table H is used to keep track of the cold rules in TCAM. H contains 2^p buckets with each holding a cold rule's entry address. The $m=2^n$ TCAM entries are mapped to the 2^p hash buckets in a group of 2^{n-p}. Without loss of generality, the hash function simply takes the first p bits of the TCAM entry address as the index to map an entry to H. Therefore, only the remaining $n-p$ bits of the entry address need to be stored in H to uniquely identify a TCAM entry.

In addition to the address bits a, each bucket also contains a flag field v and a value field $rate$. The 1-bit flag v is used to indicate if the bucket contains a valid TCAM entry. $rate$ holds the floating-point value for the hashed TCAM entry and it is equal to the number of times the rule in that entry is matched since the last read divided by the time duration between the two consecutive reads:

$$rate=\frac{hit\ counter}{time\ duration} \tag{2}$$

Algorithm 3 describes the hash table insertion and update process. The TCAM entry counters are read in a round-robin fashion. For each entry, its corresponding bucket in H is examined. If the bucket is empty (*i.e.*, v='0'), v is set to '1', and a and $rate$ of the current entry is filled into this bucket. Otherwise, the entry's value is compared against the value in the bucket. If the entry's value is smaller, the address and value of the bucket are updated with the entry's address and value.

Although this process cannot produce the globally optimal results, it readily picks out the coldest rule for every 2^{n-p}

Algorithm 3: Identify Cold Rules(*idx*)

Input: *idx*: the index of hash bucket to be read.
Output: the TCAM entry address of the cold rule.

C[:].val: values of the $m = 2^n$ entries last read
C[:].t: the time the counter was last read
H[:]: a hash table with 2^p buckets
base_a = *idx*≪(*n-p*)
for (*inc_a* = 0; *inc_a* < 2^{n-p}; *inc_a*++) **do**
  tmp_val: new value of {*base_a*+*inc_a*}-th counter
  $tmp\_rate = \frac{(tmp\_val - C[base\_a+inc\_a].val)}{(cur\_time - C[base\_a+inc\_a].t)}$
  if *H[idx].v*==0 **then**
    H[idx].{v, a, rate} = {1, *inc_a*, *tmp_rate*}
  else if *H[idx].rate* > *tmp_rate* **then**
    H[idx].{a, rate} = {*inc_a*, *tmp_rate*}
  C[base_a+inc_a].{t, val} = {*cur_time*, *tmp_val*}
return *base_a* + *H[base_a].a*

rules in TCAM, and reads only 2^{n-p} counters each round. Our analysis and evaluation show the trade-off renders good enough results.

Assuming the 2^n isolate-rules are randomly distributed the 2^n TCAM entries, we use $P(x)$ to denote the probability that the 2^p cold rules selected by Algorithm 3 are all among the x globally coldest rules ($x \geq p$), and use A_i ($i < 2^p$) to indicate that the i-th selected rule belongs to the x globally coldest rules. We have the following equation:

$$\begin{aligned} P(x) &= P(A_0 A_1 \dots A_{2^p-1}) \\ &= P(A_0) \times P(A_1|A_0) \dots \times P(A_{2^p-1}|A_0 \dots A_{2^p-2}) \\ &= (1 - \frac{\binom{2^n-x}{2^{n-p}}}{\binom{2^n}{2^{n-p}}}) \times (1 - \frac{\binom{2^n-x}{2^{n-p}}}{\binom{2^n-1}{2^{n-p}}}) \times \dots (1 - \frac{\binom{2^n-x}{2^{n-p}}}{\binom{2^n-2^p+1}{2^{n-p}}}) \\ &= \prod_{i=0}^{2^p-1} (1 - \frac{\binom{2^n-x}{2^{n-p}}}{\binom{2^n-i}{2^{n-p}}}) \end{aligned} \quad (3)$$

Fig. 9 shows the probability distribution given different n and p values, where the horizontal axis represents the value of x, and the vertical axis represents the probability that the 2^p rules selected by Algorithm 3 are all among the x globally coldest rules. A data point on the green dotted curve in the figure can be interpreted like this: in a TCAM with 1,024 entries, the probability that the 32 cold rules selected by Algorithm 3 are all among the coldest 173 rules is 90%.

Fig. 9 also provides hints for choosing hash table parameters. If we want to achieve high identification accuracy at the cost of high bandwidth consumption, we can reduce the size of the hash table. In the extreme case the hash table has only one bucket. In this case all counters need to be read every time and the globally coldest entry can be found. On the contrary, if we can tolerate some inaccuracy in exchange of low bandwidth consumption, we can increase the number of buckets in the hash table.

The hash table read process is also in a round-robin fashion but it is synchronous with the insertion and update process. Whenever a new isolate-rule is generated, if the TCAM is not full, it is directly inserted into an empty entry; otherwise, the software will request for a TCAM entry, which is acquired by

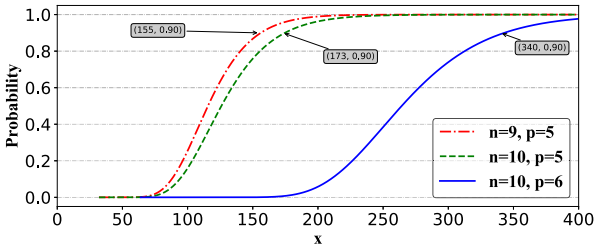


Fig. 9. The theoretical accuracy of finding cold rules.

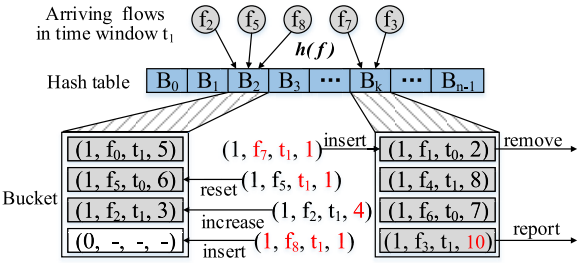


Fig. 10. Using hash-based flow table to detect the hot flows.

consulting the next bucket in H. The bucket is cleared (*i.e.*, v is reset to '0') after each read.

Note that some commercial switches do not provide user interface to directly read and write a particular TCAM entry, and the rule move scheme is hidden from users. But the switches always allow users to uniquely distinguish each rule. This information is enough for Algorithm 3 to function.

B. Find Hot Flows in Slow Path

A flow is considered to be hot if the number of packets of it exceeds a threshold in a period of time or an epoch. Among the packets punted to the slow path, the software constantly identifies the hot flows that are not yet covered by the fast path. Sketch-based algorithms [68]–[70] can be used to acquire various flow statistics. Since we only need to find the currently active hot flows, we design a simpler scheme instead without resorting to more sophisticated sketch algorithms.

We use a flow table to keep track of active flows and figure out the hot flows among them in slow path. The flow table is implemented as a q-way hash table (*i.e.*, each hash bucket can hold q flow records) as shown in Fig. 10. Each flow record is composed of four fields: the valid flag v, the flow ID f, the epoch number t, and the packet counter c.

Algorithm 4 describes the hot flow detection process based on the flow table. For each packet handled by the slow path, its flow ID is extracted and used to consult the flow table. If the flow is new (*i.e.*, no matching record in the corresponding hash bucket) and there is still an empty slot in the bucket, the flow ID with the current epoch number and the counter value of 1 is recorded in the slot (*e.g.*, f_8). When no empty slot is available for the new flow (*e.g.*, f_7), an existing flow record needs to be overwritten. The record with the oldest epoch number is chosen as the victim. If there is a tie, it is broken by choosing the record with the smallest counter value (*e.g.*, f_1).

For the packets whose flow record is found in the flow table, if its epoch number is current and its counter value reaches the threshold, a hot flow is identified (*e.g.*, f_3). Otherwise, if the epoch number is outdated, it is updated to the current and the record's counter value is reset to 1 (*e.g.*, f_5). If neither condition is met, the counter is incremented (*e.g.*, f_2).

Algorithm 4: Find Hot Flow(f)

Input: f: the input flow header
Output: whether the given flow f is hot or not.

```
H[:]:hash table with n buckets and q entries per bucket
v: the flag to indicate whether the entry is occupied
t: the epoch number of the current time
c: the counter values of the corresponding entry
th: the threshold for judging whether a flow is hot
B = H[hash_function(f)], res_idx = 0
for (cur_idx = 1; cur_idx < q; cur_idx ++) do
    if B[cur_idx].v == 0 then res_idx = cur_idx;
    else
        if B[cur_idx].f == f then
            if B[cur_idx].t == t then B[cur_idx].c++;
            else B[cur_idx].{t, c} = {t, 1};
            if B[cur_idx].c > th then
                B[cur_idx].v = 0
                return true
            return false
        else
            if B[res_idx].v == 0 then continue;
            else if B[res_idx].t > B[cur_idx].t then
                res_idx = cur_idx
            else if B[res_idx].t == B[cur_idx].t then
                if B[res_idx].c > B[cur_idx].c then
                    res_idx = cur_idx
B[res_idx].{v, f, t, c} = {1, f, t, 1}
return false
```

Our algorithm takes $O(q)$ time and $O(nq)$ memory to locate and process a flow record. Due to hash collisions, multiple flows may be mapped to the same hash bucket. Therefore, q should be carefully selected to balance the detection speed and the detection accuracy.

C. Rule Table Lookups in Slow Path

Rule table lookups are needed for the packets punted to the slow path to make forwarding decisions. A large number of algorithmic solutions are available [40]. We adopt the simple trie and *HiCuts* [71] for IP prefix table lookup and multiple-field rule table lookup, respectively. We avoid other sophisticated optimizations for the following reasons: (1) The software forwarding load is light thanks to the existence of T-cache; (2) The memory in software is abundant; (3) The rule table updates may be frequent. If the requirements for speed or memory cannot be met by these basic algorithms, more advanced algorithms, such as *SAIL* [72] and *NeuroCuts* [73], can be applied.

D. Put Everything Together

Fig. 11 shows the overall architecture of the T-cache system with the omission of the rule table update process. Any miss on the fast-path TCAM triggers a slow path lookup on the rule table. Meanwhile, the slow path processor keeps track of the cold rules in the cache and monitors the emerging hot flows. Once a new hot flow is identified, an isolate-rule is generated for it. After acquiring a TCAM entry from the cold rule table, the isolate-rule is installed to the TCAM. To cope with the cold start, we set a low threshold for the hot flow identifier at the beginning until TCAM is sufficiently populated.

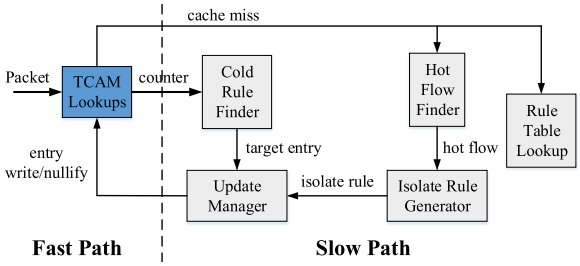


Fig. 11. The overall architecture of the T-cache system.

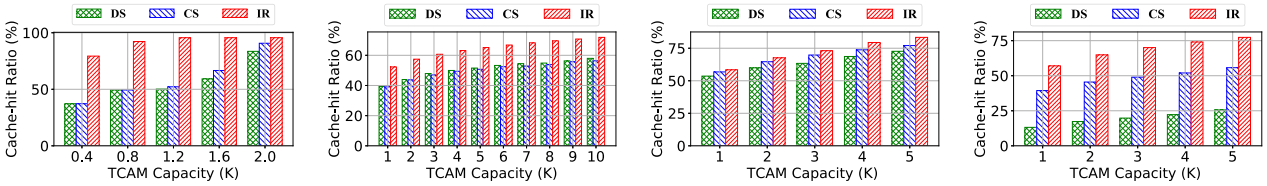
To improve the CPU utilization, the T-cache software process is split into three threads: (1) Rule table lookup, (2) Hot flow detection, and (3) Cache update with cold rule finder and isolate-rule generator. The first two threads work on every packet that enters the slow path. Only if a hot flow is detected, the third thread is awakened. Since all the control components of T-cache are in software, T-cache can be deployed not only on switches, but also on other network devices such as NICs and Middleboxes.

VI. Implementation and Evaluation

A. Experimental Setup

We first compare the isolate-rule (IR) based T-cache with the cover-set (CS) and dependent-set (DS) based TCAM cache schemes through software simulation, in which we ignore the cost of the data interaction between the control plane and the data plane (*e.g.*, counter read and write, rule insertions and deletions). The CS-based scheme is essentially an instance of CacheFlow [8]. All these schemes are implemented using C/C++ language and compiled by g++ with -O2 optimization. We run the simulators on a commodity server with the Ubuntu 16.04-LTS operating system.

We also prototype the above schemes in a programmable 32×100Gbps switch EdgeCore Wedge100BF-32X. The slow path processor is an Intel Xeon D-1517 CPU with the Open Network Linux operating system. The fast path is mainly a Barefoot Tofino switch chip on which the packet processing and forwarding are programmed using P4-14 language. To ensure the IP lookup table is handled by TCAM, we set the matching type to be ternary rather than LPM. The switch API does not allow users to control the location of rules in TCAM and read the associate matching counter of a particular entry. Fortunately, we can bind a counter to each rule and access the counter using the rule specification. Thus we can still run our algorithm without knowing the rule's exact location in TCAM. The software part of the schemes is implemented using C/C++ language and compiled by g++. The hardware part is implemented using P4-14 language and compiled by *p4c*. The APIs (*e.g.*, insert/delete rules and read/write counters) for communication between the control plane and the data plane are generated by *p4c*. By setting the egress port of the default matching action to the CPU port, the unmatched packets are punted to the slow path. Since the isolate-rules are independent with each other, they are assigned the same priority value and inserting them in TCAM does not incur any rule moves.



(a) CAIDA. (b) Stanford. (c) ACL. (d) FW.

Fig. 12. Comparison of TCAM cache hit-rate on IP prefix tables and 5-tuple rule tables.

TABLE I
DATASETS USED IN OUR EXPERIMENTS

Name	Source	Property	
		Policy	Trace
Equinix	CAIDA	real-world, DIP	real-world
Stanford	Stanford Backbone	real-world, DIP	synthetic
ACL	ClassBench-ng	synthetic, 5-tuple	synthetic
FW	ClassBench-ng	synthetic, 5-tuple	synthetic

TABLE II
POLICIES AND TRACES GENERATED BY CLASSBENCH-NG

Type	# of Policy	# of Rule	# of Packet	# of Flow
ACL	19.9×10^3	27.9×10^3	1.14×10^7	8.87×10^5
FW	18.4×10^3	80.4×10^3	1.84×10^7	1.14×10^6

B. Dataset

The rule tables and packet traces used in our experiments are shown in Table I.

CAIDA: The Equinix datacenter routing table [24] includes 760K IP prefixes. The timestamped packet trace contains 1,521 million packets during a 60-minute window from 13:00 UTC on March 15, 2018.

Stanford Backbone: The routing table is downloaded from a Cisco router on the Stanford backbone network [74]. Lacking a real packet trace, we use ClassBench-ng [75] to generate a trace with 93 million packets based on the routing table. Since the ClassBench-ng is designed for 5-tuple rules only, the single field input makes the output trace packets vary only in the destination IP address, following a Zipf distribution. The packet trace is not timestamped.

ClassBench-ng: It is difficult to access real-world multi-field rule sets due to security concerns, so we resort to ClassBench-ng to generate several synthetic ones, namely Access Control List (ACL) and Firewall (FW), with each having the characteristics matching the real-world rule sets. The rules that cannot be stored in a single TCAM entry are transformed into a set of prefix-based rules as a common practice. An accompanying packet trace is also generated for each rule set. A drawback of synthetic traces is that they do not present enough spatial locality, making the tests based on them only partially exhibit T-cache's potential. The synthetic rule sets and packet traces are summarized in Table II.

C. Simulation Results

TCAM Hit-Rate: Suppose the traffic distribution is known in advance and the hottest rules at each moment are cached in TCAM. This allows us to test the best-case TCAM hit-rate for IR, CS, and DS.

Fig. 12(a) shows the achieved TCAM hit-rates on CAIDA in one-minute time windows. When the TCAM size is 1.2K, DS and CS achieve only 50% and 52% TCAM hit-rate, respectively, while IR can achieve a hit-rate more than 95%. This is because DS and CS both need to cache some extra

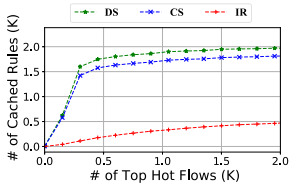


Fig. 13. Cost of caching hot flows.

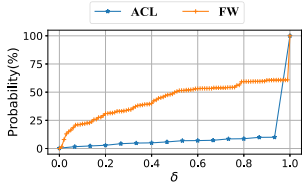


Fig. 14. The distribution of δ on popular rules.

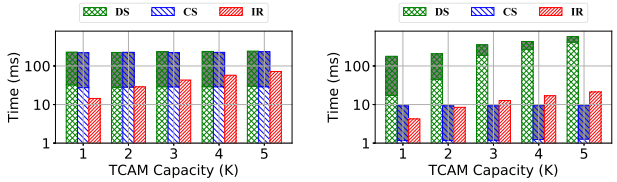
rules which are actually cold. We also notice that CS is only slightly better than DS in this case.

Fig. 13 explains the finding from another angle by showing the number of rules needed for each method to ensure different number of hot flows to hit the TCAM. The top 1K flows require DS, CS and IR to cache 1,895, 1,725, and 322 rules, respectively. An isolate-rule in IR can cover about 3 hot flows.

Fig. 12(b) shows the achieved TCAM hit-rates on Stanford Backbone, given the same TCAM capacity. Since the packet trace has no timestamp, we assume the packets are injected in constant speed. In this case the achieved TCAM hit-rate is much lower than that on CAIDA, due to the lack of spatial locality in the synthetic trace. Despite this, IR is still 20% better than CS and DS.

Fig. 12(c) shows the achieved TCAM hit-rates on the multi-field rule table ACL, given the same TCAM capacity. Although the difference is not significant, IR is still better than DS and CS.

Fig. 12(d) shows the achieved TCAM hit-rates on FW, given the same TCAM capacity. IR demonstrates much better performance than DS and CS in this case, because FW's rule table is 2.5× larger than ACL's and with more severe dependency among rules. We also notice that CS performs much better than DS in rule set FW, but not in rule set ACL. We use a parameter $\delta_r = \frac{a}{b}$ to help reveal the reason, where r denotes a specific rule, a denotes the size of r's cover-set and b denotes the size of r's dependent-set. We select the top-hitted 5K rules from both ACL and FW and plot the distribution of δ on the selected rules in Fig. 14. From the figure, we can see that about 60% of the selected rules in FW (and 10% in ACL) have their δ less than 1.0. These rules require more caching overhead in DS than in CS. For rules with $\delta = 1$, their caching overheads in DS and CS are the same. Since more rules in FW have $\delta < 1$, CS is able to demonstrate better performance than DS as CS incurs relatively less overhead for such rules.



(a) CAIDA. (b) FW.

Fig. 15. Comparison of time consumption to fill the TCAM.

Time Consumption for Cache Filling: When the cache is empty, all new flow arrivals will trigger cache misses. It takes time for the cache to be filled by selected rules (by schemes such as IR) to reach a certain hit-rate.

We conduct the experiments and Fig. 15(a) shows that IR is about 200× faster than DS and CS to fill the cache to reach a certain hit-rate under CAIDA.

Actually, reaching the highest TCAM hit-rate for CS and DS is proved to be NP-hard so they adopt the following heuristic search algorithm. For a rule r matching n packets, a value η is defined as the ratio of $\frac{n}{b}$ and $\frac{n}{a}$ by DS and CS, respectively. DS and CS calculate η for each rule and greedily select the rules with larger η first to fill the TCAM. The major time complexity of DS and CS lies in the calculation of η and sorting them for all rules, which is represented by the shaded portion of the bar. The selection of rules takes less time which is represented by the unshaded portion of the bar.

In contrast, IR only needs to generate an isolate-rule for each flow and requires much less computation time than DS and CS. In our experiments, Algorithm 1 takes only about 0.12us to generate an isolate-rule under CAIDA.

Fig. 15(b) shows a similar comparison under FW. Although the size of FW is much smaller than that of CAIDA, the time consumption of DS increases significantly under FW, because the rule set contains more complicated dependency among rules. Meantime, we can see that IR consumes more time on FW to generate an isolate-rule when the TCAM size exceeds 3,000. This is because the time complexity of Algorithm 2 is proportional to the number of rule match fields. Although the time for Algorithm 2 to generate m isolate-rules is larger than the time for CS to find m specific rules, T-cache does not generate m rules at a stroke, but generates each rule in time according to traffic changes, which takes only about 4us and is enough to keep up with the emergence speed of new hot flows.

TCAM Cache Replacement: Under normal working conditions, DS and CS update TCAM incrementally. Assume the rules to be inserted into and evicted from TCAM have been identified by comparing m newly selected rules and m existing rules in TCAM. It is not easy to conduct the cache replacement due to the constraint of rule dependency. A rule in TCAM can be evicted only after all its dependent rules are evicted, and a rule can be inserted into TCAM only after all rules depending on it are inserted. Both incur long computation time and many rule moves, which make DS and CS hardly practical. Therefore, we do not evaluate DS and CS's TCAM cache replacement performance but show IR's cache replacement performance in the system level of T-cache. Since IR is dependency-free, it can directly insert a newly generated isolate-rule into TCAM without moving any existing rules.

Finding Cold Rules in TCAM: DS and CS periodically poll TCAM counters to measure the heat of rules in TCAM. They then use the greedy algorithm mentioned above to find out the rules to be cached in TCAM. A rule in TCAM is considered cold if it will not be matched in the next period. On the one hand, periodically polling all TCAM counters consumes a lot of bandwidth. On the other hand, the excessive computation time will result in a relatively long update cycle (500 seconds in CacheFlow), which can stale the cache.

In contrast, IR uses a small hash table to constantly track cold rules, so a newly generated rule can be inserted into TCAM instantly, keeping the cache fresh. Since the hash table is consulted only once per each rule insertion, the bandwidth consumption between the slow path and the fast path is negligible.

We explore how the hash table size influences the accuracy of the cold rules identified. We conduct the experiment as follows. We search the TCAM using one-minute worth of packets from the CAIDA trace without updating the TCAM.

We then calculate the number of cold rules recorded in the hash table with a size of 2^p. We use $\gamma_c = \frac{k}{2^p}$ to reflect the accuracy, where k indicates the number of identified rules by T-cache that are actually among the top c coldest rules. We run the experiment for the 60-minute trace and the average result is shown in Fig. 16(a). 55% and 63% of the 2^p identified cold rules are among the top $c = 2^p$ coldest rules for $p = 5$ and $p = 6$ when $m = 1$K (*i.e.*, top 3.2% and top 6.4%), respectively. If we relax the requirement to test whether the identified cold rules are among the top 10% and top 20% coldest ones for $p = 5$ and $p = 6$, respectively, the accuracy becomes 100%, which is consistent with the theoretical analysis in Fig. 9. Meantime, we can see that the 2^4 cold rules identified by the T-cache are among the top 3.5% coldest rules when $m = 2$K. Such an accuracy is achieved with only 0.8% of the communication overhead comparing to the case of reading all the counter values.

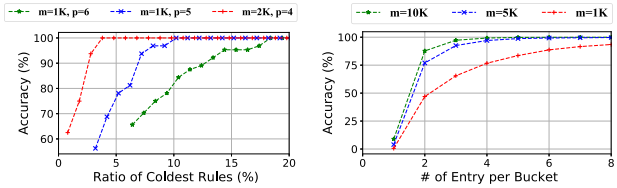
Finding Hot Flows in Slow Path: T-cache uses a software hash table in CPU to identify those newly emerged hot flows in real time. Since most of the traffic is forwarded through TCAM and only a small proportion of traffic reaches the slow path, and the time complexity of hashing operation is $O(1)$, the speed for hot flow identification is not a problem. However, the way we deal with the hash collision may lead to the miss of some real hot flows. To measure the accuracy of hot flow detection, we conduct experiments as follows.

First, We need to choose a proper hash function. We conduct experiments on several well-known hash functions including SDBM, BKDR, and RS to evaluate their performance in terms of speed and hash collisions based on the CAIDA traffic trace (including ∼19M packets). The experimental results are shown in Table III.

TABLE III
THE PERFORMANCE OF HASHING FUNCTIONS

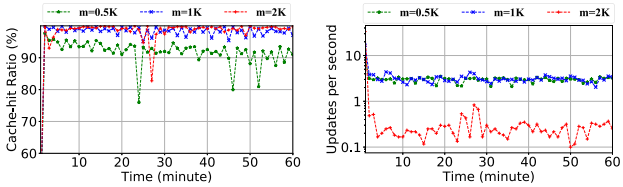
Hash Name	#Bucket (K)	#Entry	Mem (MB)	Speed (Mpps)	Collision (%)
SDBM	10	4	0.4	9.07	1.1
BKDR	10	16	1.6	5.42	0.1
		4	0.4	9.17	1.0
RS	10	4	0.4	9.15	0.7
	20	2		9.84	0.9

For a hash table with 10K buckets and 4 entries per bucket, RS achieves the lowest collision rate (a collision means more than 4 flows are mapped to the same bucket). The results of BKDR show that the number of entries per bucket has a significant influence on the speed. Moreover,



(a) Accuracy of finding cold rules. (b) Accuracy of finding hot flows.

Fig. 16. Accuracy of finding cold rules and hot flows.



(a) TCAM hit-rate. (b) Update frequency.

Fig. 17. Performance of T-cache in a dynamic environment.

the results of RS show that, when the memory footprint is fixed, the fewer number of entries per bucket leads to a higher speed but a higher collision rate. Based on the above observations, we choose RS as the hash function for hot flow detection.

We also see that the speed of finding hot flow reaches about 10Mpps. Remember that most of the traffic is forwarded through TCAM and only a small proportion of them reaches the slow path, the speed for hot flow identification is fast enough for our purpose.

In a time window t, T-cache reports a flow as a large one if more than s packets belong to it. We calculate how many actual hot flows are detected by T-cache and the results are shown in Fig. 16(b), where t is 60 seconds and s is 1K. Fig. 16(b) shows that, when the number of entries q per hash bucket is 4, 75%, 98% and 99% hot flows are correctly identified when the size of the hash table is 1K, 5K and 10K, respectively. Meantime, by comparing the accuracy in the case of $m = 5$K, $q = 4$ and $m = 10$K, $q = 2$ in Fig. 16(b), we can see that given the same storage space, more entries in one bucket will help to improve the accuracy. Besides, considering that a larger m requires more processing time and more storage but with limited improvement on performance, we set m and q to 5K and 4, respectively, which consumes only 5K $\times$ 4 $\times$ 10B = 200KB = 0.2MB memory.

Overall Performance of T-Cache: In order to compare IR with DS and CS, the above experiments are carried out under a relatively static situation. Now we run the complete T-cache system in a dynamic environment and the results are shown in Fig. 17. For a high-speed backbone router with 760K rules, T-cache needs a small TCAM with 500 entries to achieve a TCAM hit-rate of about 93%. Meantime, maintaining such a high hit-rate only requires 3.5 rule insertions per second. We also find that smaller TCAM corresponds to higher update frequency. This is because the competition among hot flows will be more intensive in a small TCAM, which will lead to frequent caching and eviction of rules in a dynamic traffic environment. Since T-cache adopts IR to achieve dependency-free rule insertion, each insertion only requires a single TCAM write operation and the extra computation required by other schemes such as CacheFlow is avoided.

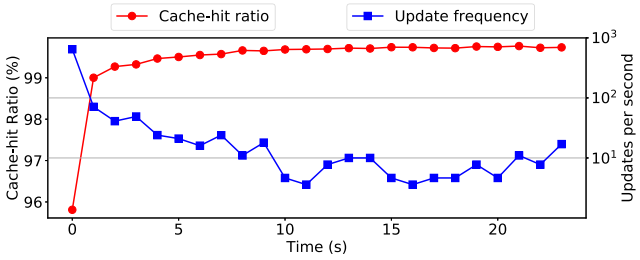


Fig. 18. TCAM hit-rate and update frequency in cold start.

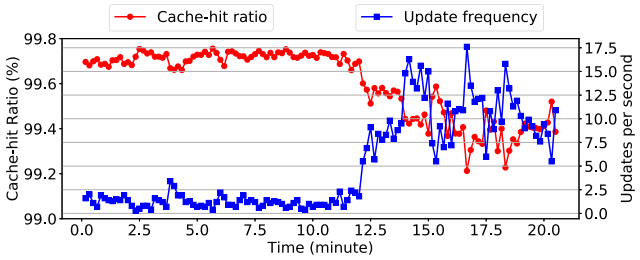


Fig. 19. TCAM hit-rate and update frequency in steady state.

D. Hardware Evaluation Results

We test the hardware prototype using the time-stamped traffic trace and rules from CAIDA. The data plane TCAM has 1K entries, the hot flow finder has 10K buckets and 4 entries per bucket, and the cold rule finder has 32 buckets.

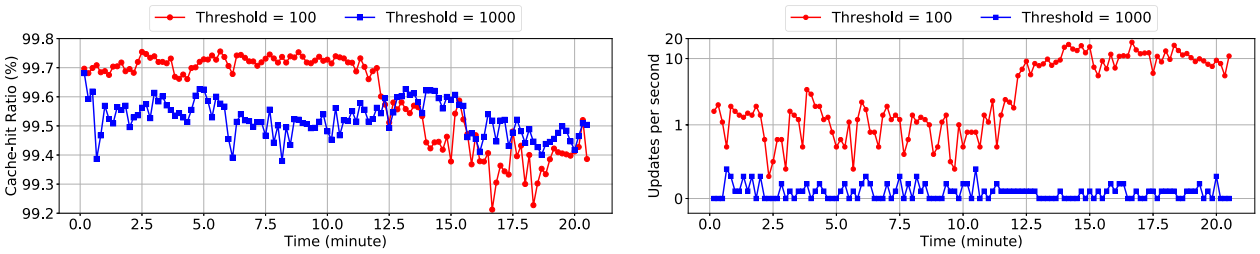
Cold Start: To cope with the cold start, we set a time window of 1 second and a threshold of 10 for the hot flow identifier at the beginning until TCAM is sufficiently populated. T-cache does not evict any exiting rules before it is full. The results are shown in Fig. 18.

The T-cache hit-rate increases rapidly from 0 to 99.27% in the first three seconds, and gradually stabilizes at 99.73% in the next 20 seconds. Meanwhile, the update frequency is 647 updates per second during the cold start and quickly decreases to about 10 per second. The high update rate for the cold start is challenging for schemes such as CacheFlow, which requires 4ms to insert a rule, but it is easy for T-cache because it takes only 40us to insert an isolate-rule. Besides, the rapid decrease of the update frequency exhibits the efficiency of the hot flow finder.

Steady State: At the beginning, in order to reduce the slow path workload and the transmission between the slow path and the fast path, we set a small time window and a low threshold for hot flow identification to fill the TCAM quickly. After that, we increase the threshold and the time window, which reduces the TCAM update frequency and increases the cache hit-rate. Of course, the high hit-rate is also attributed to the accuracy of the cold rule finder.

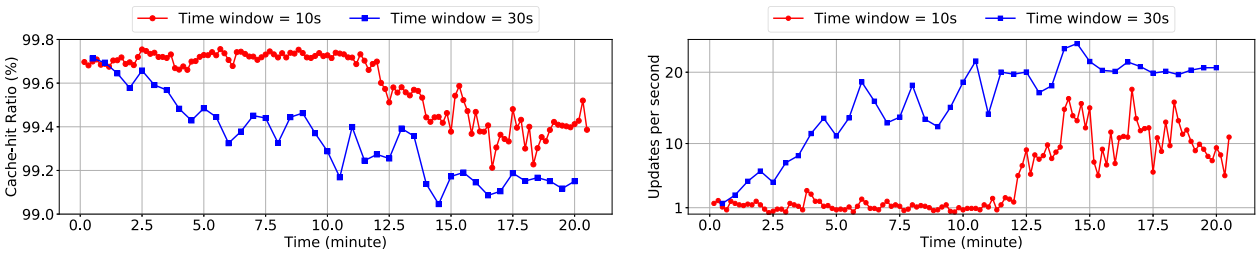
After the TCAM is fully filled, the time window and threshold of hot flow finder are switched to 10 seconds and 100, respectively. The other experimental settings remain unchanged. T-cache continuously works for 20 minutes and the test result is shown in Fig. 19.

Fig. 19 shows that in the first 10 minutes, the TCAM hit-rate remains above 99.6%; in the next 10 minutes, the TCAM hit-rate declines slightly, but is still above 99.2%. In contrast, the CS and DS schemes, as shown in the Fig. 12(a), achieve only a 50% TCAM hit-rate when TCAM size is 1.2K and the time window is 1 minute (due to the high computation complexity for selecting caching rules and the high TCAM



(a) TCAM hit-rate. (b) Update frequency.

Fig. 20. Comparison of T-cache performance under different thresholds for hot flow.



(a) TCAM hit-rate. (b) Update frequency.

Fig. 21. Comparison of T-cache performance under different time windows.

update cost, their cache refresh time window cannot be shorter than one minute).

The decrease of T-cache hit-rate in the last 10 minutes is due to the acceleration of traffic locality changes, which is consistent with the statistical results shown in Fig. 5(b). In the first 10 minutes, about 85% of top 0.1% flows in one minute remains hot in the next minute; for the next 10 minutes, only about 80% of top 0.1% flows in one minute remains hot in the next minute.

Fig. 19 also shows that, in the first 10 minutes, T-cache requires only 3 updates per second. Compared with the update frequency during the cold start shown in Fig. 18, the drop is due to the increase of threshold for hot flows. In the next 10 minutes, the update frequency is still below 15 per second.

Compared with the software simulation results in Fig. 17(a) and Fig. 17(b), the TCAM hit-rate and update frequency in hardware experiments are higher. This is because the time window is shorter and the threshold for hot flows is lower in hardware experiments. T-cache in hardware experiments delivers the hot flows into TCAM immediately, without needing to wait for one minute as in the software simulation.

Impact of Threshold: The threshold for the hot flow finder decides the generation speed of hot flows as well as the update frequency of TCAM. We investigate the influence of threshold and exhibit the results in Fig. 20.

Fig. 20(a) shows that in the first 10 minutes, a smaller threshold achieves a higher TCAM hit-rate than a larger one. This is because a larger threshold requires more packets of a hot flow to be processed by the slow path before it is identified. Since the TCAM has handled more than 99.5% of the packets and very few packets are sent to the slow path, the impact of the threshold on the overall TCAM hit-rate is obvious. However, Fig. 20(a) also shows that the TCAM hit-rate with a smaller threshold decreases when the traffic locality varies; Fig. 20(b) shows that the update frequency with a lower threshold increases when the traffic locality changes. More flows are identified as hot when the threshold is small. This also increases the chance that an evicted cold flow is actually hot, so it will be inserted back into TCAM soon.

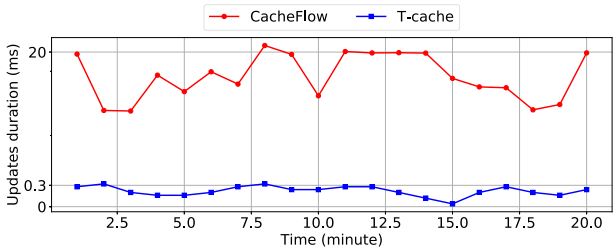


Fig. 22. Comparison of update cost between T-cache and CacheFlow.

Impact of Time Window: T-cache generates isolate-rules and inserts them into TCAM dynamically. However, the hot flow finder uses a time window to exclude the long-lived but low-rate flows and find recent hot flows. We test the influence of the time window and exhibit the results in Fig. 21.

Fig. 21(a) shows that a smaller time window helps maintain the freshness of TCAM. Fig. 21(b) shows that a smaller time window corresponds to a lower update frequency. This is because, for the same threshold, a larger time window tends to find more hot flows, but some of them do not contribute much to the improvement of the TCAM hit-rate. Although a smaller time window may also be beneficial for other caching schemes (*e.g.*, CacheFlow), only T-cache can take advantage of it. This is because other schemes need to read all the TCAM counters and run complex algorithms to select the rules to be cached and replaced in each time window, not to mention the time-consuming TCAM update process.

Update Cost: The TCAM update cost is an important performance metric for a TCAM-based cache system. We count the time required to perform all the updates within one minute for T-cache and CacheFlow. T-cache sets the time window to 30 seconds and the hot flow threshold to 1,000. CacheFlow chooses the cover-set as its caching rule selection method. As shown in Fig. 22, T-cache requires 20∼60× less time to refresh the TCAM as compared to CacheFlow.

Note that the number of moves per rule insertion is related to the number of unique priorities. In an IP LPM table, there are at most 25 different priorities. For multi-dimensional

rule tables with more priorities, the update performance of CacheFlow will be conceivably lower, while T-cache is not affected at all.

Line Speed Performance Prediction: Assume the switch works at the line speed of 32×100Gbps and the average packet size is 500 bytes. Under the 99.0% TCAM hit-rate, a conservative estimation, the slow path of T-cache needs to process 8Mpps of the traffic for rule table lookup and hot flow identification. Our evaluations have shown that the slow path processor can easily handle this load. Therefore, T-cache can support line speed processing on the state-of-the-art switches.

VII. Conclusion

T-cache takes advantage of the traffic temporal and spatial localities to meet the challenges of network traffic throughput and rule table scalability. T-cache avoids the troublesome TCAM update issue by crafting dependency-free rules to cache. The software and hardware evaluations show T-cache outperforms the existing rule caching schemes. T-cache is easy to implement on switches, NICs, and Middleboxes. In future work we seek to deploy and test T-cache in real networks for better system tuning.

References

[1] Y. Wan *et al.*, "T-cache: Dependency-free ternary rule cache for policy-based forwarding," in *Proc. IEEE INFOCOM*, Jul. 2020, pp. 536–545.

[2] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Apr. 2008.

[3] R. S. Sandhu and P. Samarati, "Access control: Principle and practice," *IEEE Commun. Mag.*, vol. 32, no. 9, pp. 40–48, Sep. 1994.

[4] B. Niven-Jenkins *et al.*, *Problem Statement for Service Function Chaining*, document IETF RFC 7498, Apr. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7498.txt

[5] C. Filsfils, N. K. Nainar, C. Pignataro, J. C. Cardona, and P. Francois, "The segment routing architecture," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2015, pp. 1–6.

[6] J. Ordonez-Lucena, P. Ameigeiras, D. Lopez, J. J. Ramos-Munoz, J. Lorca, and J. Folgueira, "Network slicing for 5G with SDN/NFV: Concepts, architectures, and challenges," *IEEE Commun. Mag.*, vol. 55, no. 5, pp. 80–87, May 2017.

[7] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *Proc. ACM SIGCOMM*, 2015.

[8] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "CacheFlow: Dependency-aware rule-caching for software-defined networks," in *Proc. ACM SOSR*, 2016, pp. 1–12.

[9] P. He, W. Zhang, H. Guan, K. Salamatian, and G. Xie, "Partial order theory for fast TCAM updates," *IEEE/ACM Trans. Netw.*, vol. 26, no. 1, pp. 217–230, Feb. 2018.

[10] G. Li, Y. Qian, C. Zhao, Y. R. Yang, and T. Yang, "DDP: Distributed network updates in SDN," in *Proc. IEEE ICDCS*, Jul. 2018, pp. 1468–1473.

[11] H. Chen and T. Benson, "Hermes: Providing tight control over high-performance SDN switches," in *Proc. ACM CoNEXT*, Nov. 2017, pp. 283–295.

[12] G. Li, Y. R. Yang, F. Le, Y.-S. Lim, and J. Wang, "Update algebra: Toward continuous, non-blocking composition of network updates in SDN," in *Proc. IEEE INFOCOM*, Apr. 2019, pp. 1081–1089.

[13] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined WAN," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 3–14, 2013.

[14] J. Zheng, H. Xu, G. Chen, and H. Dai, "Minimizing transient congestion during network update in data centers," in *Proc. IEEE ICNP*, Nov. 2015, pp. 1–10.

[15] W. Zhang, G. Liu, A. Mohammadkhan, J. Hwang, K. K. Ramakrishnan, and T. Wood, "SDNFV: Flexible and dynamic software defined control of an application- and flow-aware data plane," in *Proc. ACM Middleware*, Nov. 2016, pp. 1–12.

[16] D. Li, S. Wang, K. Zhu, and S. Xia, "A survey of network update in SDN," *Frontiers Comput. Sci.*, vol. 11, no. 1, pp. 4–12, 2017.

[17] *Edge-Core Wedge100BF Series Switches*. Accessed: Mar. 2021. [Online]. Available: https://www.edge-core.com/

[18] M. Kuźniar, P. Perešíni, D. Kostić, and M. Canini, "Methodology, measurement and analysis of flow table update characteristics in hardware OpenFlow switches," *Comput. Netw.*, vol. 136, pp. 22–36, May 2018.

[19] M. Kuźniar *et al.*, "What you need to know about SDN flow tables," in *Proc. Passive Act. Meas. Conf. (PAM)*, 2015, pp. 347–359.

[20] M. M. Waldrop, "The chips are down for Moore's law," *Nature*, vol. 530, no. 7589, p. 144, 2016.

[21] Q. Dong, S. Banerjee, J. Wang, and D. Agrawal, "Wire speed packet classification without TCAMs: A few more registers (and a bit of logic) are enough," in *Proc. ACM SIGMETRICS*, 2007, pp. 253–264.

[22] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *Proc. IEEE INFOCOM*, Mar. 2010, pp. 1–9.

[23] N. Sarrar, S. Uhlig, A. Feldmann, R. Sherwood, and X. Huang, "Leveraging Zipf's law for traffic offloading," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 1, pp. 16–22, Jan. 2012.

[24] *The CAIDA UCSD Anonymized Internet Traces*. Accessed: Mar. 2018. [Online]. Available: https://www.caida.org/data/passive/passive dataset.xml

[25] H. Wang, W. Ding, and Z. Xia, "A cloud-pattern based network traffic analysis platform for passive measurement," in *Proc. Int. Conf. Cloud Service Comput.*, Nov. 2012, pp. 1–7.

[26] W. Mao, Z. Shen, and X. Huang, "Facilitating network functions virtualization by exploring locality in network traffic: A proposal," in *Proc. 2nd Int. Conf. Comput. Sci. Artif. Intell. (CSAI)*, 2018, pp. 495–499.

[27] B. Yan, Y. Xu, H. Xing, K. Xi, and H. J. Chao, "CAB: A reactive wildcard rule caching system for software-defined networks," in *Proc. 3rd Workshop Hot Topics Softw. Defined Netw.*, Aug. 2014, pp. 163–168.

[28] J. Wallerich and A. Feldmann, "Capturing the variability of internet flows across time," in *Proc. IEEE INFOCOM*, Apr. 2006, pp. 1–6.

[29] K. Papagiannakit, N. Taft, and C. Diot, "Impact of flow dynamics on traffic engineering design principles," in *Proc. IEEE INFOCOM*, Mar. 2004, pp. 2295–2306.

[30] J. Wallerich, H. Dreger, A. Feldmann, B. Krishnamurthy, and W. Willinger, "A methodology for studying persistency aspects of internet flows," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 2, pp. 23–36, Apr. 2005.

[31] D. Lin, Y. Zhang, C. Hu, B. Liu, X. Zhang, and D. Pao, "Route table partitioning and load balancing for parallel searching with TCAMs," in *Proc. IEEE IPDPS*, Mar. 2007, pp. 1–10.

[32] D. E. Taylor and J. S. Turner, "ClassBench: A packet classification benchmark," *IEEE/ACM Trans. Netw.*, vol. 15, no. 3, pp. 499–511, Jun. 2007.

[33] X. Wen *et al.*, "RuleTris: Minimizing rule update latency for TCAM-based SDN switches," in *Proc. IEEE ICDCS*, Jun. 2016, pp. 179–188.

[34] K. Qiu, J. Yuan, J. Zhao, X. Wang, S. Secci, and X. Fu, "Fast lookup is not enough: Towards efficient and scalable flow entry updates for TCAM-based OpenFlow switches," in *Proc. IEEE ICDCS*, Jul. 2018, pp. 918–928.

[35] V. C. Ravikumar, R. N. Mahapatra, and L. N. Bhuyan, "EaseCAM: An energy and storage efficient TCAM-based router architecture for IP lookup," *IEEE Trans. Comput.*, vol. 54, no. 5, pp. 521–533, May 2005.

[36] V. C. Ravikumar and R. Mahapatra, "TCAM architecture for IP lookup using prefix properties," *IEEE Micro*, vol. 24, no. 2, pp. 60–69, Mar./Apr. 2004.

[37] C. Zhang *et al.*, "OBMA: Minimizing bitmap data structure with fast and uninterrupted update processing," in *Proc. IEEE/ACM IWQoS*, Jun. 2018, pp. 1–6.

[38] E. Spitznagel, D. Taylor, and J. Turner, "Packet classification using extended TCAMs," in *Proc. IEEE ICNP*, Nov. 2003, pp. 120–131.

[39] V. Demianiuk, S. Nikolenko, P. Chuprikov, and K. Kogan, "New alternatives to optimize policy classifiers," *IEEE/ACM Trans. Netw.*, vol. 28, no. 3, pp. 1088–1101, Jun. 2020.

[40] D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Comput. Surv.*, vol. 37, no. 3, pp. 238–275, Sep. 2005.

[41] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla, "Packet classifiers in ternary CAMs can be smaller," in *Proc. ACM SIGMETRICS*, 2006, pp. 311–322.

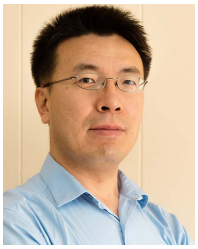
[42] K. Kannan and S. Banerjee, "Compact TCAM: Flow entry compaction in TCAM for power aware SDN," in *Proc. Int. Conf. Distrib. Comput. Netw.*, 2013, pp. 439–444.

[43] H. Liu, "Routing table compaction in ternary CAM," *IEEE Micro*, vol. 22, no. 1, pp. 58–64, Jan. 2002.

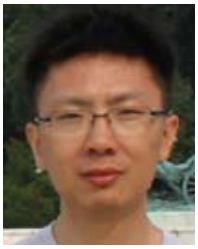
[44] M. Bienkowski, N. Sarrar, S. Schmid, and S. Uhlig, "Online aggregation of the forwarding information base: Accounting for locality and churn," *IEEE/ACM Trans. Netw.*, vol. 26, no. 1, pp. 591–604, Feb. 2018.

[45] O. Rottenstreich and J. Tapolcai, "Optimal rule caching and lossy compression for longest prefix matching," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 864–878, Apr. 2017.
[46] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for advanced packet classification with ternary CAMs," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 4, pp. 193–204, Oct. 2005.
[47] J. V. Lunteren and T. Engbersen, "Fast and scalable packet classification," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 4, pp. 560–571, May 2003.
[48] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "SAX-PAC (scalable and expressive packet classification)," in *Proc. ACM Conf. SIGCOMM*, Aug. 2014, pp. 15–26.
[49] H. Liu, "Efficient mapping of range classifier into ternary-CAM," in *Proc. IEEE Hot Interconnects*, Aug. 2002, pp. 95–100.
[50] K. Kogan, S. I. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "Exploiting order independence for scalable and expressive packet classification," *IEEE/ACM Trans. Netw.*, vol. 24, no. 2, pp. 1251–1264, Apr. 2016.
[51] J. Xu, M. Singhal, and J. Degroat, "A novel cache architecture to support layer-four packet classification at memory access speeds," in *Proc. IEEE INFOCOM*, Mar. 2000, pp. 1445–1454.
[52] F. Chang, W.-C. Feng, and K. Li, "Approximate caches for packet classification," in *Proc. IEEE INFOCOM*, Mar. 2004, pp. 2196–2207.
[53] J. P. Sheu and Y. C. Chuo, "Wildcard rules caching and cache replacement algorithms in software-defined networking," *IEEE Trans. Netw. Service Manage.*, vol. 13, no. 1, pp. 19–29, Mar. 2016.
[54] M. Bienkowski, J. Marcinkowski, M. Pacut, S. Schmid, and A. Spyra, "Online tree caching," in *Proc. 29th ACM Symp. Parallelism Algorithms Archit.*, Jul. 2017, pp. 329–338.
[55] Z. Ding, X. Fan, J. Yu, and J. Bi, "Update cost-aware cache replacement for wildcard rules in software-defined networking," in *Proc. ISCC*, Jun. 2018, pp. 457–463.
[56] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, pp. 351–362, Oct. 2010.
[57] R. Li, B. Zhao, R. Chen, and J. Zhao, "Taming the wildcards: Towards dependency-free rule caching with FreeCache," in *Proc. IEEE/ACM IWQoS*, Jun. 2020, pp. 1–10.
[58] X. Jin *et al.*, "Dynamic scheduling of network updates," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 539–550, 2014.
[59] Z. Wang, H. Che, M. Kumar, and S. K. Das, "CoPTUA: Consistent policy table update algorithm for TCAM without locking," *IEEE Trans. Comput.*, vol. 53, no. 12, pp. 1602–1614, Dec. 2004.
[60] H. Song and J. Turner, "NXG05-2: Fast filter updates for packet classification using TCAM," in *Proc. IEEE GlOBECOM*, Nov. 2006, pp. 1–6.
[61] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks," in *Proc. USENIX NSDI*, 2013, pp. 1–13.
[62] C. J. Anderson *et al.*, "NetKAT: Semantic foundations for networks," *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 113–126, 2014.
[63] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," *ACM SIGPLAN Notices*, vol. 47, no. 1, pp. 217–230, Jan. 2012.
[64] N. Foster *et al.*, "Frenetic: A network programming language," *ACM SIGPLAN Notices*, vol. 46, no. 9, pp. 279–291, 2011.
[65] D. Shah and P. Gupta, "Fast incremental updates on ternary-CAMs for routing lookups and packet classification," in *Proc. Hot Interconnects*, 2000, pp. 145–153.
[66] B. Zhao, R. Li, J. Zhao, and T. Wolf, "Efficient and consistent TCAM updates," in *Proc. IEEE INFOCOM*, Jul. 2020, pp. 1241–1250.
[67] M. Cygan, Ł. Kowalik, and M. Wykurz, "Exponential-time approximation of weighted set cover," *Inf. Process. Lett.*, vol. 109, no. 16, pp. 957–961, Jul. 2009.
[68] X. Yu, H. Xu, D. Yao, H. Wang, and L. Huang, "CountMax: A lightweight and cooperative sketch measurement for software-defined networks," *IEEE/ACM Trans. Netw.*, vol. 26, no. 6, pp. 2774–2786, Dec. 2018.
[69] Y. Zhai, H. Xu, H. Wang, Z. Meng, and H. Huang, "Joint routing and sketch configuration in software-defined networking," *IEEE/ACM Trans. Netw.*, vol. 28, no. 5, pp. 2092–2105, Oct. 2020.
[70] T. Yang *et al.*, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. ACM SIGCOMM*, 2018, pp. 561–575.
[71] P. Gupta and N. McKeown, "Classifying packets with hierarchical intelligent cuttings," *IEEE Micro*, vol. 20, no. 1, pp. 34–41, Jan./Feb. 2000.
[72] T. Yang *et al.*, "Constant IP lookup with FIB explosion," *IEEE/ACM Trans. Netw.*, vol. 26, no. 4, pp. 1821–1836, Aug. 2018.
[73] E. Liang, H. Zhu, X. Jin, and I. Stoica, "Neural packet classification," in *Proc. ACM SIGCOMM*, Aug. 2019, pp. 256–269.
[74] *Stanford Backbone Router Forwarding Configuration*. Accessed: Mar. 2021. [Online]. Available: http://tinyurl.com/o8glh5n
[75] J. Matoušek, G. Antichi, A. Lucansky, A. W. Moore, and J. Korenek, "ClassBench-ng: Recasting classbench after a decade of network evolution," in *Proc. ACM/IEEE ANCS*, May 2017, pp. 204–216.

Ying Wan received the B.S. degree in communication engineering from Northwestern Polytechnical University in 2016. He is currently pursuing the Ph.D. degree in computer science with Tsinghua University, Beijing, China, under the advice of Dr. B. Liu. His research interests include bloom filter design, high performance network algorithm, and software defined networking.



Haoyu Song (Senior Member, IEEE) received the B.E. degree in electronics engineering from Tsinghua University in 1997 and the M.S. and D.Sc. degrees in computer engineering from Washington University in St. Louis in 2003 and 2006, respectively. He is currently a Senior Principal Network Architect with Futurewei Technologies, USA. His research interests include software defined networks, network virtualization and cloud computing, high performance networked systems, algorithms for network packet processing, and intrusion detection.



Yang Xu (Senior Member, IEEE) received the B.E. degree from the Beijing University of Posts and Telecommunications in 2001 and the Ph.D. degree in computer science and technology from Tsinghua University, China, in 2007. He was a Faculty Member of the Department of Electrical and Computer Engineering, New York University Tandon School of Engineering. He is currently the Yaoshihua Chair Professor with the School of Computer Science, Fudan University. He has published more than 80 journal articles and conference papers, and holds more than ten U.S. and international granted patents on various aspects of networking and computing. His research interests include SDN, data center networks, distributed machine learning, edge computing, network function virtualization, and network security. He served as a TPC member for many international conferences, an Editor for the *Journal of Network and Computer Applications* (Elsevier), and a Guest Editor for the IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS–Special Series on Network Softwarization and Enablers and *Security and Communication Networks* Journal (Wiley)–Special Issue on Network Security and Management in SDN.

Yilun Wang received the B.E. degree from the Department of Computer Science and Technology, Tsinghua University. He has participated in student research training program and worked on several research projects in computer networks. His research interests include cloud computing, data center networks, and network measurements.

Tian Pan (Senior Member, IEEE) received the Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 2015. He was a Post-Doctoral Researcher with the Beijing University of Posts and Telecommunications from 2015 to 2017, where he has been an Associate Professor since 2020. His research interests include network architecture, software-defined networking, programmable data plane, and satellite networks.

Chuwen Zhang received the B.S. degree in communication engineering from Northwestern Polytechnical University, Xi'an, China, in 2015. He is currently pursuing the Ph.D. degree in computer science with Tsinghua University, Beijing, China, under the advice of Dr. B. Liu. His research interests include high-performance switches/routers, named data networking and vehicle networks.

Yi Wang (Senior Member, IEEE) received the Ph.D. degree in computer science and technology from Tsinghua University in July 2013. He is currently a Research Associate Professor with the SUSTech Institute of Future Networks, Southern University of Science and Technology. His research interests include future network architectures, information centric networking, software-defined networks, and the design and implementation of high-performance network devices.

Bin Liu (Senior Member, IEEE) received the M.S. and Ph.D. degrees in computer science and engineering from Northwestern Polytechnical University, Xi'an, China, in 1988 and 1993, respectively. He is currently a Full Professor with the Department of Computer Science and Technology, Tsinghua University, Beijing, China. His current research interests include high-performance switches/routers, network processors, high-speed security, and greening the Internet. He has received numerous awards from China, including the Distinguished Young Scholar of China and won the inaugural Applied Network Research Prize sponsored by ISOC and IRTF in 2011.