T-cache: Dependency-free Ternary Rule Cache for Policy-based Forwarding

Ying Wan*, Haoyu Song†, Yang Xu‡¶, Yilun Wang*, Tian Pan§, Chuwen Zhang*, Bin Liu*‖
* Tsinghua University, China † Futurewei Technologies, USA ‡ Fudan University, China
‖ Peng Cheng Laboratory, China § Beijing University of Posts and Telecommunications, China
¶ Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education of China
{wany16, yl-wang15}@mails.tsinghua.edu.cn, {chuwen1992, shykcl, lmyujie}@gmail.com
xuy@fudan.edu.cn, pan@bupt.edu.cn

***Abstract*—Ternary Content Addressable Memory (TCAM) is widely used by modern routers and switches to support policy-based forwarding. However, the limited TCAM capacity does not scale with the ever-increasing rule table size. Using TCAM just as a rule cache is a plausible 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 TCAM update problem also disappears. We provide the complete T-cache system design including slow path processing and cache replacement. Evaluations based on real-world and synthesized rule tables and traces show that T-cache is efficient and robust for network traffic in various scenarios.**

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) [1], [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 apply 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 best-match resolving is necessary. This process needs to be fast enough to sustain the line-speed forwarding. Unlike the address-based forwarding, traditionally it 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 TCAM is effective for now but it has long been criticized for high power consumption, high cost, and poor scalability.

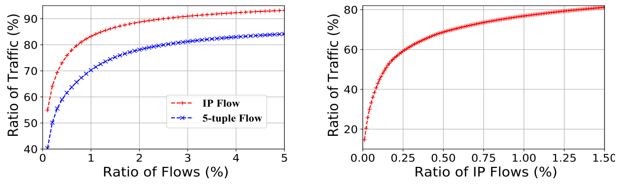
The imminent end of Moores Law [8] implies that we must accept the limit on the TCAM capacity in a single chip as a starting point to design a policy-based forwarding system that can meet the throughput and service requirements. An embedded TCAM with around ten thousand entries is at the upper end of 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 [9], [10]. 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.

Our work follows this suit, yet 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 so the TCAM update is trivial and the cache hit-rate is boosted.

This work is supported by NSFC (61432009, 68172213, 61702049). Corresponding Author: Bin Liu (lmyujie@gmail.com).



(a) Equinix from CAIDA

(b) CERNET from IPTAS

Fig. 1. Temporal locality of real-world network traces.

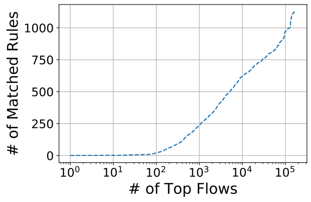


Fig. 2. Spatial locality of flows.

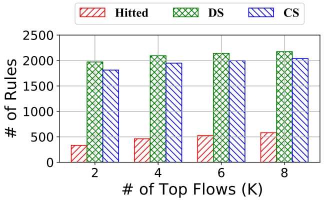


Fig. 3. Expansion of cached rules.

II. Background

To have an effective rule caching system for policy-based forwarding, some conditions need to hold and several challenges need to be addressed.

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 [11] 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 [12]. 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. 1(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 [13], as shown in Fig. 1(b).

Numerous previous studies also confirm the existence of such temporal locality [9]–[11], [14]–[19]. 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. 2, 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 [15], [18].

Such temporal and spatial locality of flow distribution is critical to supporting 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 [20]. 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 [15] shows that, in a rule set generated by ClassBench [21], the average number of dependent rules for each rule is 350.

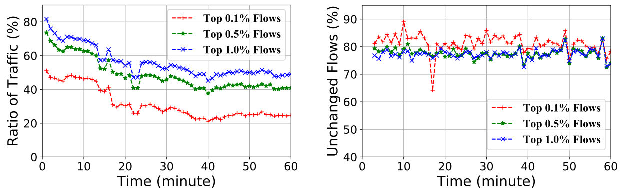
CacheFlow [20] alleviates the challenge of dependent-set to some extent by introducing the concept of *cover-set* (CS). A rule r's cover-set is composed of the rules that directly overlap with r in r's dependent-set. Only the rules in r's cover-set are to be loaded into TCAM along with r. To ensure the 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. In case 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. 3, 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 requires complex calculation is necessary [20]. 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



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

Fig. 4. High turnover of the top flows on Equinix trace

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 [22] shows that inserting a single ACL rule for a 1K rule set generated by ClassBench [21] requires up to 466 entry moves. Although the state-of-the-art optimizations [22], [23] 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 [24].

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. 4(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. 4(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 [25]–[27] and packet classification [28]–[30]), and becomes a standard component in SDN switches. However, due to its high hardware cost and power consumption [20], TCAM's capacity is limited which may fail to accommodate rule tables [31]. The inefficient support of ranges in TCAM makes the situation worse [32]. Some researchers tackle this issue by trying to compact the rule tables [33], [34]. Many efforts are made to reduce the number of TCAM entries needed to hold a rule with ranges [35]–[37]. This kind of work does not fundamentally solve the problem as the gap between the TCAM capacity and rule table size broadens.

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 [32], [38], [39]. 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 [9]. 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 [20], [40]. Based on CacheFlow [20], Sheu *et al.* design a sophisticated algorithm to select the cached rules for better TCAM hit-rate [40].

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 [41]. 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 [15]. Yu *et al.* partition the rule set into new non-overlapping rules in the controller and cache them into underlying switches [42]. These schemes are all computing-intensive.

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 [43]–[45]. Some works optimize the update process by reducing the redundant updates at the controller level [46]–[49]; some other works aim to optimize the actual rule insertion process by designing algorithms to avoid unnecessary entry moves [22]–[24], [44], [45], [50]. These solutions either fail to achieve the ideal TCAM entry move reduction [24], [44], [45], [50], or require excessive computation time [22], [23].

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 just find a single TCAM-cacheable rule r', which is the largest multi-dimensional box fully contained by $S_{r,f}$. We name r' an isolate-rule. 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. In case 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 updates 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 can 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:

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

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_i in $C(r)$, f mismatches it on at least one dimension.

Proof. This is obvious. If f matches r_i on every dimension, r_i should be f's best matching rule because r_i 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 not overlapping in the space if they are not overlapping on any dimension.

Since r' can mismatch a rule in $C(r)$ in multiple dimensions, there are multiple possible ways to eliminate the rule overlap. 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 $V(r')$.

Now we prove the NP-hardness of the problem. Given a flow f, its best matching rule r, and r's cover-set $C(r)$=$\{r_1, ..., r_n\}$, 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 can be deduced from a Weighted Set Cover Problem (WSCP) [51], which has been proved to be NP-hard. WSCP is formulated as follows. Given a finite universe $\mathcal{U} = \{1, 2, ..., n\}$ of n members, a collection of subsets of $\mathcal{U}$ that $\mathcal{S} = \{s_1, s_2, ..., s_m\} \wedge \forall i,\ s_i \subseteq \mathcal{U}$, 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. 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 $C(r)$. Since WSCP is NP-hard, we cannot 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.

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.

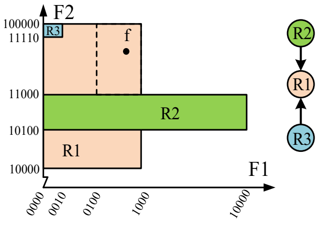
Rule Table

Rule	Priority	F1	F2
R1	1	0***	1****
R2	2	****	101**
R3	2	000*	1111*

Popular Flow

Flow ID	F1	F2
f	0110	11101

(a) A rule table and a flow.



(b) Rule space and dependency graph.

Fig. 5. An example of calculating isolate-rule.

B. Algorithm for Isolate-rule Calculation

The pseudo code of isolate-rule calculation is given in Algorithm 1. We use Fig. 5(a) as an accompanying example to explain the Algorithm 1. As shown in Fig. 5(a), f's best matching rule is $R1$ and $C(R1)$=$\{R2, R3\}$. (f, $R1$, $C(R1)$) is used as input to Algorithm 1 to generate the isolate-rule r'.

Algorithm 1 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 (Line 2). $d_i^{r_j}$ 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 1 then calculates a collection of subsets of $C(r)$, $\mathcal{C}$=$\{\mathcal{C}_{i,d}\}$ from $\{d_i^{r_j}\}$ (Line 3). $\mathcal{C}_{i,d}$ 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.

Next, Algorithm 1 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 (Line 13) 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 1 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 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 1 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. 5(b).

Algorithm 1 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

A. Find Cold Rules in TCAM

T-cache only caches isolate-rules. To maintain the freshness of the cache, we need to constantly identify the rules that are getting cold in TCAM, creating opportunities for new hot rules to be swapped in. Each entry in TCAM has an associated counter which records the number of times the rule in the entry is matched since the last read to the counter. A rule is considered to be cold if during a period of time, its matching counter is smaller than a threshold.

The logic of cold rule identification is implemented in hardware. A lightweight hash table H is used to keep track of the cold rules in TCAM. H contains 2^k buckets with each holding a cold rule's entry address. The hash function simply takes the lower k bits of the TCAM entry address as the index to map an entry to H. Therefore, given a TCAM with $m = 2^n$ entries, 2^{n-k} entries are mapped into a single bucket in H, and only the upper $n - k$ 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 counter field c. The 1-bit flag v is used

```
Algorithm 1: Isolate rule ($f$, $r$, $C(r)$)
$C(r)$={$r_1, ..., r_n$}, $r_i$=($f_1^{r_i}, ..., f_k^{r_i}$), $f_i^r$=($f_1^{r_i}, ..., f_k^{r_i}$)
$d_i^{r_j}$:$f$ mismatches $r_j$ in the $l_i^r+d_i^{r_j}$ bit of $i$-th dimension
$\mathcal{C}$={$\mathcal{C}_{i,d}$}, $\mathcal{C}_{i,d}$={$r_j | r_j \in C(r), d_i^{r_j} \leq d$}, $1 \leq i \leq k$, $1 \leq d \leq L_i$
$\mathcal{L}$: subset of $\mathcal{C}$; $\mathcal{U}_{\mathcal{L}}$: union of $\mathcal{L}$; $\mathcal{W}_{\mathcal{L}}$: sum of $d$ in $\mathcal{L}$;
$\mathcal{L}_o$={$\mathcal{C}_{1,L_1}, \mathcal{C}_{2,L_2}, ..., \mathcal{C}_{k,L_k}$}, $\mathcal{L}_t = \varnothing$ //initialization
SolutionSearch(1, $\mathcal{L}_t$, $\mathcal{L}_o$)
$r'$←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 $\mathcal{L}_o$←$\mathcal{L}_t$;
    else
        SolutionSearch($i$+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, $i \leq k$, $i$=$i$+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 $r'$
```

```
Algorithm 2: Identify Cold Rules
Input: $C[:]$ : counters of the $2^n$ TCAM entries
//$H[:]$: a hash table with $2^k$ buckets
for ($addr$=0; $addr<2^n$; $addr$++) do
    $ha$=$addr[n{:}k]$, $la$=$addr[k{:}0]$
    if $H[la].v$=0 then
        $H[la].v$=1, $H[la].a$=$ha$, $H[la].c$=$C[addr]$
    else if $H[la].a == ha$ then
        $H[la].c$=$C[addr]$
    else if $H[la].c > C[addr]$ then
        $H[la].a$=$ha$, $H[la].c$=$C[addr]$
```

to indicate if the bucket contains a valid TCAM entry. The counter c holds the counter value of the hashed TCAM entry.

Algorithm 2 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 c of the current entry is filled into this bucket. Otherwise, if the bucket is holding the information of the current entry (*i.e.*, a matches the upper $n-k$ bits of the entry address), c is updated with the new counter value. In case neither is true, the entry's counter value is compared with c. If the entry's counter value is smaller, a and c are updated with the entry's address and counter.

This process cannot produce the globally optimal results but it readily picks out the coldest rule for every 2^{n-k} rules in TCAM, which is good enough for our purpose.

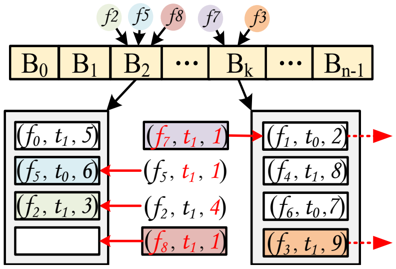


Fig. 6. Using hash-based flow table to detect the hot flows in slow path.

The hash table read process is also in a round robin fashion but it is asynchronous with the insertion and update process. Whenever a new isolate-rule is generated, the software will request for a new TCAM entry, which is acquired by consulting the next bucket in H. The bucket is cleared (*i.e.*, v is reset to '0') after the read.

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, in order to generate and cache isolate-rules for these flows and alleviate the slow path's forwarding load.

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. 6. Each flow record is composed of three fields: the flow ID f, the epoch number t, and the packet counter c.

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). In case 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).

Our algorithm takes $O(q)$ time to locate and process a flow record. The time complexity can be reduced to $O(q/4)$ by applying the single instruction, multiple data (SIMD) technique provided by the advanced CPU architecture [52].

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

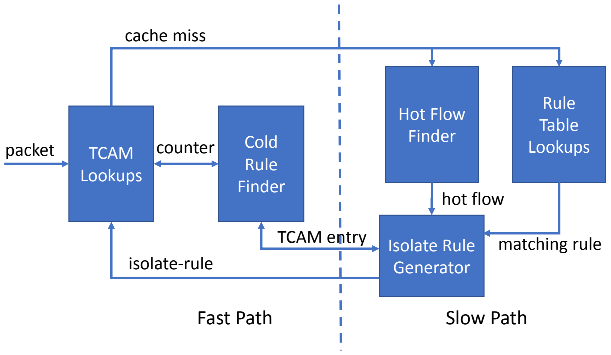


Fig. 7. The overall architecture of the T-cache system.

of algorithmic solutions are available [31]. We adopt the simple decision-tree based algorithm *HiCuts* [53] and avoid any sophisticated optimization 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.

D. *Put Everything Together*

Fig. 7 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 fast path keeps track of the cold rules in the cache and the slow path 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.

VI. Implementation and Evaluation

A. *Experimental Setup*

We compare the isolate-rule (IR) based T-cache with the cover-set (CS) and dependent-set (DS) based TCAM cache schemes through software simulation. The CS-based scheme is essentially CacheFlow [20]. 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.

B. *Dataset*

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

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

CAIDA. The Equinix datacenter routing table [12] 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.

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	8.87×10ˆ5
FW	18.4×10^3	80.4×10^3	1.84×10^7	1.14×10ˆ6

Stanford Backbone. The routing table is downloaded from a Cisco router on the Stanford backbone network [54]. Lacking a real packet trace, we use ClassBench-ng [55] 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. *Experimental Results*

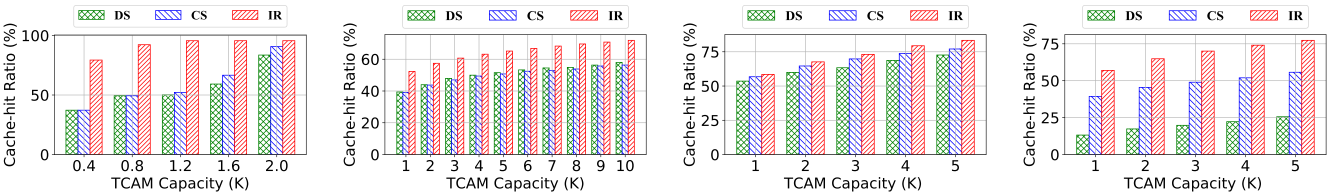
TCAM Hit-rate. Suppose the traffic distribution is known in advance and the hottest rules at each moment are cached in TCAM. This allows us to test the best-case TCAM hit-rate for IR, CS, and DS.

Fig. 8(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 rules which are actually cold. We also notice that CS is only slightly better than DS in this case.

Fig. 9 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. 8(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. 8(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.



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

Fig. 8. Comparison of TCAM cache hit-rate on IP prefix tables and 5-tuple rule tables.

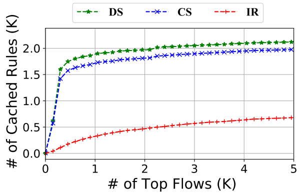


Fig. 9. Cost of caching hot flows.

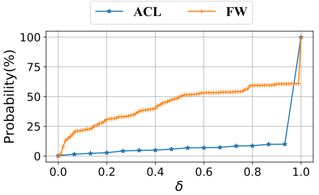
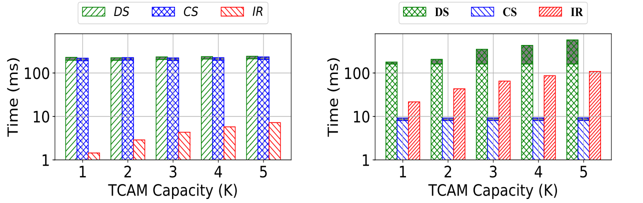


Fig. 10. The distribution of δ on popular rules.



(a) CAIDA
(b) FW

Fig. 11. Comparison of time consumption to achieve a high TCAM hit-rate.

Fig. 8(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. 10. 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 δ=1, their caching overhead in DS and CS are the same. Since more rules in FW have δ <1, CS is able to demonstrate better performance than DS as CS incurs relatively less overhead for such rules.

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. 11(a) shows that IR takes much less time than DS and CS to fill the cache to reach a certain hit rate under CAIDA.

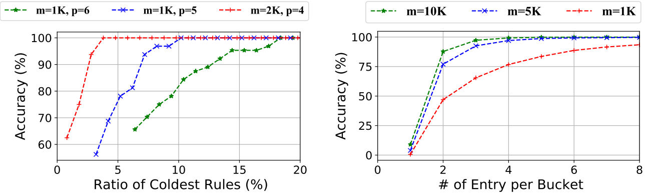
Actually, reaching the highest TCAM hit-rate for CS and DS is proved to be NP-hard so they adopt the following heuristic search algorithm. For a rule r matching n packets, a value v is defined as the ratio of $\frac{n}{b}$ and $\frac{n}{a}$ by DS and CS, respectively. DS and CS calculate v for each rule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

Fig. 11(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. This is because the time complexity of Algorithm 1 is proportional to the number of rule match fields. Although the time for Algorithm 1 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.

TCAM Cache Replacement. In normal working condition, 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 rule.

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 the rules to be cached in TCAM. A rule in TCAM is consid-



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

Fig. 12. Performance of T-cache in finding cold rules and hot flows.

ered 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 hardware-based 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. 12(a).

As shown in the figure, 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=1K (*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%. 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=2K. It is worth noting that such good accuracy is achieved with only $\frac{2^4}{2000}$ of the original TCAM counters.

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 amount 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.

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. 12(b), where t is 60 seconds and s is 1K. Fig. 12(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=5K, q=4 and m=10K, q=2 in Fig. 12(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.

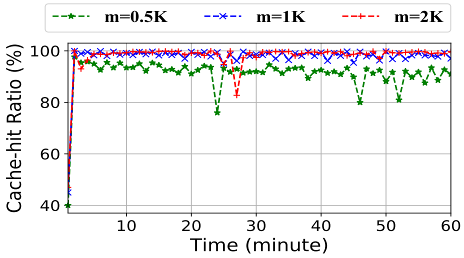


Fig. 13. TCAM hit-rate of T-cache in a dynamic environment.

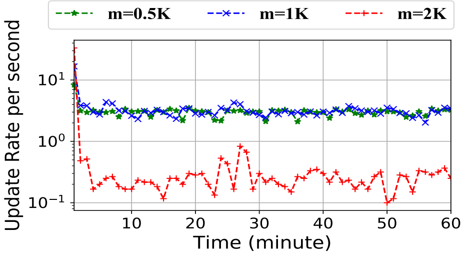


Fig. 14. TCAM update frequency of T-cache in a dynamic environment.

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. 13 and 14. 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 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.

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 evaluations show T-cache outperforms the existing rule caching schemes. In future work we seek to deploy and test T-cache in real networks for better system tuning.

References

[1] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. *ACM SIGCOMM CCR*, 37(4):1–12, 2007.

[2] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM CCR*, 38(2):69–74, 2008.

[3] Ravi S Sandhu and Pierangela Samarati. Access control: Principle and practice. *IEEE communications magazine*, 32(9):40–48, 1994.

[4] Paul Quinn and Tom Nadeau. Problem statement for service function chaining. Technical report, 2015.

[5] Clarence Filsfils, Stefano Previdi, Les Ginsberg, Bruno Decraene, Stephane Litkowski, and Rob Shakir. Segment routing architecture. Technical report, 2018.

[6] Jose Ordonez-Lucena, Pablo Ameigeiras, Diego Lopez, Juan J Ramos-Munoz, Javier Lorca, and Jesus Folgueira. Network slicing for 5G with SDN/NFV: Concepts, architectures, and challenges. *IEEE Communications Magazine*, 55(5):80–87, 2017.

[7] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, 2015.

[8] M Mitchell Waldrop. The chips are down for Moores law. *Nature News*, 530(7589):144, 2016.

[9] Qunfeng Dong, Suman Banerjee, Jia Wang, and Dheeraj Agrawal. Wire speed packet classification without tcams: a few more registers (and a bit of logic) are enough. In *ACM SIGMETRICS Performance Evaluation Review*, volume 35, pages 253–264. ACM, 2007.

[10] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *INFOCOM*, 2010.

[11] Nadi Sarrar, Steve Uhlig, Anja Feldmann, Rob Sherwood, and Xin Huang. Leveraging Zipf's law for traffic offloading. *ACM SIGCOMM Computer Communication Review*, 42(1):16–22, 2012.

[12] The CAIDA UCSD Anonymized Internet Traces[20180315]. www.caida.org/data/passive/passive_dataset.xml. Accessed in Mar, 2018.

[13] Hengbo Wang, Wei Ding, and Zhen Xia. A cloud-pattern based network traffic analysis platform for passive measurement. In *2012 International Conference on Cloud and Service Computing*, pages 1–7. IEEE, 2012.

[14] Wei mao *et al*. Facilitating Network Functions Virtualization by Exploring Locality in Network Traffic: A Proposal. In *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*, pages 495–499. ACM, 2018.

[15] Bo Yan, Yang Xu, Hongya Xing, Kang Xi, and H Jonathan Chao. Cab: A reactive wildcard rule caching system for software-defined networks. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 163–168. ACM, 2014.

[16] Jorg Wallerich and Anja Feldmann. Capturing the variability of internet flows across time. In *INFOCOM*, 2006.

[17] K Papagiannakit, Nina Taft, and Christophe Diot. Impact of flow dynamics on traffic engineering design principles. In *INFOCOM*, 2004.

[18] Jörg Wallerich, Holger Dreger, Anja Feldmann, Balachander Krishnamurthy, and Walter Willinger. A methodology for studying persistency aspects of internet flows. *ACM SIGCOMM CCR*, 35(2):23–36, 2005.

[19] Dong Lin *et al*. Route Table Partitioning and Load Balancing for Parallel Searching with TCAMs. In *IPDPS*, 2007.

[20] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proceedings of the Symposium on SDN Research*, page 6. ACM, 2016.

[21] David E Taylor and Jonathan S Turner. Classbench: A packet classification benchmark. *IEEE/ACM TON*, 15(3):499–511, 2007.

[22] Peng He, Wenyuan Zhang, Hongtao Guan, Kavé Salamatian, and Gaogang Xie. Partial order theory for fast tcam updates. *IEEE/ACM Transactions on Networking (TON)*, 26(1):217–230, 2018.

[23] Xitao Wen *et al*. RuleTris: Minimizing rule update latency for TCAM-based SDN switches. In *ICDCS*, 2016.

[24] Kun Qiu *et al*. Fast lookup is not enough: Towards efficient and scalable flow entry updates for TCAM-based OpenFlow switches. In *ICDCS*, 2018.

[25] VC Ravikumar, Rabi N Mahapatra, and Laxmi Narayan Bhuyan. Ease-CAM: An energy and storage efficient TCAM-based router architecture for IP lookup. *IEEE Transactions on Computers*, 2005.

[26] VC Ravikumar and Rabi N Mahapatra. TCAM architecture for IP lookup using prefix properties. *IEEE Micro*, 2004.

[27] Chuwen Zhang *et al*. OBMA: Minimizing Bitmap Data Structure with Fast and Uninterrupted Update Processing. In *IWQoS*. IEEE, 2018.

[28] Ed Spitznagel, David E Taylor, and Jonathan S Turner. Packet classification using extended TCAMs. In *ICNP*, 2003.

[29] Kirill Kogan *et al*. Exploiting order independence for scalable and expressive packet classification. *In TON*, 24(2):1251–1264, 2015.

[30] Vitalii Demianiuk *et al*. New alternatives to optimize policy classifiers. In *ICNP*, pages 121–131. IEEE, 2018.

[31] David E. Taylor. Survey Taxonomy of Packet Classification Techniques. *Acm Computing Surveys*, 37(3):238–275, 2005.

[32] Qunfeng Dong *et al*. Packet classifiers in ternary CAMs can be smaller. In *ACM SIGMETRICS Performance Evaluation Review*. ACM, 2006.

[33] Kalapriya Kannan and Subhasis Banerjee. Compact TCAM: Flow entry compaction in TCAM for power aware SDN. In *International conference on distributed computing and networking*. Springer, 2013.

[34] Huan Liu. Routing table compaction in ternary CAM. *IEEE Micro*, 22(1):58–64, 2002.

[35] Karthik Lakshminarayanan *et al*. Algorithms for advanced packet classification with ternary CAMs. In *ACM SIGCOMM CCR*. ACM, 2005.

[36] Huan Liu. Efficient Mapping of Range Classifier into Ternary-CAM. In *Hot Interconnects*, volume 10, pages 95–100, 2002.

[37] Jan Van Lunteren and Ton Engbersen. Fast and scalable packet classification. *IEEE Journal on Selected Areas in Communications*, 21(4):560–571, 2003.

[38] Jun Xu, Mukesh Singhal, and Joanne Degroat. A novel cache architecture to support layer-four packet classification at memory access speeds. In *INFOCOM*, 2000.

[39] Francis Chang, Wu-chang Feng, and Kang Li. Approximate caches for packet classification. In *INFOCOM*, 2004.

[40] Jang-Ping Sheu and Yen-Cheng Chuo. Wildcard rules caching and cache replacement algorithms in software-defined networking. *IEEE Transactions on Network and Service Management*, 13(1):19–29, 2016.

[41] Zixuan Ding, Xinxin Fan, Jinping Yu, and Jingping Bi. Update Cost-Aware Cache Replacement for Wildcard Rules in Software-Defined Networking. In *ISCC*, 2018.

[42] Minlan Yu, Jennifer Rexford, Michael J Freedman, and Jia Wang. Scalable flow-based networking with DIFANE. *ACM SIGCOMM*, 2010.

[43] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. In *ACM SIGCOMM*, 2014.

[44] Zhijun Wang, Hao Che, Mohan Kumar, and Sajal K Das. CoPTUA: Consistent policy table update algorithm for TCAM without locking. *IEEE Transactions on Computers*, 53(12):1602–1614, 2004.

[45] Haoyu Song and Jonathan Turner. Nxg05-2: Fast filter updates for packet classification using tcam. In *GLOBECOM*, 2006.

[46] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software defined networks. In *NSDI*, pages 1–13, 2013.

[47] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. *Acm sigplan notices*, 49(1):113–126, 2014.

[48] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. *ACM SIGPLAN Notices*, 47(1):217–230, 2012.

[49] Nate Foster *et al*. Frenetic: A network programming language. *ACM SIGPLAN Notices*, 46(9):279–291, 2011.

[50] Devavrat Shah and Pankaj Gupta. Fast incremental updates on Ternary-CAMs for routing lookups and packet classification. In *Proceedings of Hot Interconnects*, 2000.

[51] Marek Cygan, Łukasz Kowalik, and Mateusz Wykurz. Exponential-time approximation of weighted set cover. *Information Processing Letters*, 109(16):957–961, 2009.

[52] SIMD. http://en.wikipedia.org/wiki/SIMD.

[53] Pankaj Gupta and Nick McKeown. Classifying packets with hierarchical intelligent cuttings. *Ieee Micro*, 20(1):34–41, 2000.

[54] Stanford backbone router forwarding configuration. http://tinyurl.com/o8glh5n.

[55] Jiří Matoušek *et al*. Classbench-ng: Recasting classbench after a decade of network evolution. In *2017 ACM/IEEE ANCS*, pages 204–216. IEEE, 2017.