Wildcard-Rule Caching and Cache Replacement Algorithms in Software-Defined Networking

Jang-Ping Sheu, Po-Yao Wang, and Jagadeesha RB
Department of Computer Science, National Tsing Hua University
Hsinchu, 30013, Taiwan
sheujp@cs.nthu.edu.tw, s103062501@m103.nthu.edu.tw, jagadeesha.rb@gmail.com

Abstract- **Ternary Content Addressable Memory (TCAM) capacity problem is an important issue in Software-Defined Networking. Rule caching is an efficient technique to solve the TCAM capacity problem. However, there exists rule dependency problem in wildcard-rule caching technique. In this paper, we utilize cover-set method to solve the rule dependency problem and propose a wildcard-rule caching algorithm to cache rules into TCAM. In addition, we also propose a cache replacement algorithm considering both temporal and spatial localities. According to the simulation results, our wildcard-rule caching algorithm and cache replacement algorithm have better performance than previous works in terms of caching ratio and hit ratio, respectively.**

Keywords- Software-Defined Networking, TCAM, rule dependency problem, cache replacement algorithms.

I. INTRODUCTION

Software-Defined Networking (SDN) [1, 2] is a brand new network architecture that provides a global view of network state for network administrators to manage network services. SDN controller maintains the flow tables in the switches to comply with network policies. The flow tables are implemented by the TCAM in the modern switches. TCAM can look up a packet's header and compare the matching patterns of the packet to the match field of all rules in the flow table in parallel. In other words, it can forward packets at line rate. Although TCAM can forward packets fast, there are only 2-20K flow table sizes in commodity SDN switches which are much less than RAM-based storage [3]. Therefore, the TCAM capacity problem is an important issue in SDN.

Previous works on using TCAM efficiently could be classified into three categories, packet classification compression, rules distribution, and rules caching. Packet classification compression [4, 5, 6] is a technique that combines two similar rules into a new wildcard rule, which is semantically equivalent to the original rules. As a result, we can reduce the number of required rules. Rules distribution technique [7, 8, 9] splits the set of rules which are safe according to the network policies and distributes them along the network path. In rule-caching technique [10, 11, 12, 13, 14, 15, 16], it treats TCAM as a cache which stores the most popular or most-likely matched rules in the future. Once cache miss happens, we should choose the victim rules and evict the victim rules out in order to cache the cache miss rule into TCAM.

One of the rule caching techniques is wildcard-rule caching [13, 14, 15, 16]. Wildcard-rule caching could keep more TCAM space than exact-match rule matching. However, there are different priorities between different rules according to the network policies. If two wildcard rules overlap with each other and we only cache the lower-priority one in the TCAM, the packets matching the overlapping field space of these two rules will improperly match lower-priority rule. In other words, only caching a wildcard rule overlapped with others would cause ambiguous matching when the packets come. Therefore, the most difficult challenge for wildcard-rule caching is to deal with the rule dependency problem.

In [15], the authors use the cover-set method to solve the wildcard-rule dependency problem. The cover-set method creates a small number of new rules that cover many low-priority rules overlapped with the high-priority rule. High-priority rules usually have higher weight due to their larger field space. Therefore, cover-sets help us to avoid caching high-weight rules together with many low-weight rules overlapped with them. The cover-set caching (CSC) algorithm in [15] calculated the contribution value of each un-cached rule and cached the rule has the maximum contribution value into the TCAM until the TCAM is full. Since the CSC algorithm only considers the contribution value of each un-cached rule, the authors in [16] proposed a wildcard-rule caching (WRC) algorithm, which considers the accumulated contribution value for a set of rules. They calculated the accumulated contribution value of a set of related rules and cached the set of rules have the maximum accumulated contribution value into the TCAM until the TCAM is full.

In this paper, we propose a novel wildcard-rule caching algorithm and a cache replacement algorithm to make use of TCAM space efficiently. Our wildcard-rule caching algorithm repeats caching a set of important rules into TCAM until there is no TCAM space. Our cache replacement algorithm takes temporal and spatial traffic localities into consideration, which could make hit ratio high. There are two main contributions in this paper. One is that our wildcard-rule caching algorithm could have better caching ability than the algorithms in [15, 16]. Another one is that our cache replacement algorithm could have higher hit ratio than the RCR algorithm in [16] and the traditional cache replacement algorithm like Least Recently Used (LRU) and Adaptive Replacement Cache (ARC) algorithms [17].

The rest of this paper is organized as follows. In Section II, we review the related work of TCAM space problem. Section III presents our wildcard-rule caching algorithm and cache replacement algorithm. We show our simulation results in Section IV and conclude our paper in Section V.

II. RELATED WORK

Packet classification compression is a technique that uses wildcard rules to reduce the number of required rules according to the original semantics. Therefore, we could place smaller rule list compared with the original one into TCAM. TCAM Razor [4] is a compression algorithm based on prefix classifiers. TCAM Razor represents prefix classifiers to wildcard formation, which puts the wildcards at the end of classifier. Then, it decomposes the multidimensional classifiers into one-dimensional classifiers and solve one-dimensional classifiers

978-1-5386-3783-6/17/$31.00©2017.

optimization problem. The authors in [5, 6] proposed algorithms with non-prefix ternary classifiers. The basic idea is that rules whose matching field is differ by one bit and which have the same action field can be merged into single rule by replacing this bit with wildcard "*". However, since the rule compression technique combines several rules into one rule, the controller could not get the statistic information of the original rules.

Rule distribution techniques separate the network policies into several smaller rule-sets and distribute them through the network. DomainFlow [7] separates a data center topology into three parts, Domain 1, the turning point, and Domain 2. Domain 1 is the flow control with wildcard matching table and Domain 2 is the flow control with exact matching table. As a result, all rules can be partitioned into the wildcard matching rules and the exact matching rules, which will be assigned to Domain 1 and Domain 2, respectively. Palette [8] is a distributing framework using Pivot Bit Decomposition (PBD) iteratively to decouple the table in order to preserve the correct semantics. PBD selects one pivot bit in the table and splits the table into two smaller sub-tables at each iteration. Then, they tried to let each packet traverses all the sub-tables in order to fulfil semantically equivalent according to the original table.

The main idea of rule caching technique is to cache the significant or most-likely used rules in the future into TCAM of the switches. The authors in [10] proposed an efficient Forwarding Information Base (FIB) caching scheme that stores only non-overlapping FIB entries which may not cause the cache-hiding problem into the fast memory (i.e., TCAM), whereas storing the complete FIB in slow memory. However, their scheme only focus on handling single address field. Thus, their scheme could not make use of multiple header fields or wildcards and the counters associated with rules. The authors in [12] proposed a flow-driven rule-caching algorithm, which is a heuristic algorithm to solve the caching optimization problem, and they designed two strategies to assign timers for predictable and unpredictable packet flows, respectively. If the rule replacement happens at a switch, the rule that has the maximum timer value will be evicted and the new rule will be placed into the TCAM.

Although wildcard-rule caching can save more TCAM space than exact-match rule caching, rule dependency problem [13] is a significant challenge for wildcard-rule caching. Rule dependency problem is that if we only cache the lower-priority rule, the packets matching the overlapping region between the lower-priority rule and the higher-priority rule will mismatch the lower-priority rule. DIFANE [13] divides the wildcard rules, which have rule dependency with each other into a set of new micro rules without overlapping. However, DIFANE will create more rules. Caching in Buckets (CAB) [14] implements a two-stage flow table pipeline for switches. One is bucket filter, another one is flow table. CAB partitions the field space into small hyper-rectangles (i.e., buckets), which can be expressed as wildcard rules. Each bucket has the associated rules overlapping with the bucket in the field space. Although CAB can solve rule dependency problem, it still has to cache a bucket of rules.

III. Our Proposed Algorithms

In this section, we first present about our system model. Second, we describe the wildcard-rule caching problem. Third, we demonstrate our wildcard-rule caching algorithm and wildcard-rule replacement algorithm in detail. Finally, we analyze the time complexity of the two algorithms.

A. System Model and Problem Formulation

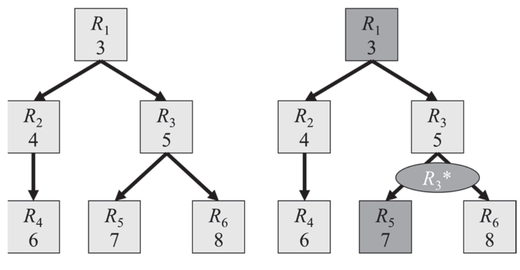
CacheFlow system proposed in [15] combines the hardware switches with the software switches act like a single switch with an infinite rule capacity. The hardware switch provides high port density, high throughput, and a modestly-sized TCAM. On the other hand, the software switches provide high rule capacity at reasonable throughput to handle "cache misses" rules in the hardware switches. We use this hardware-software hybrid switch as our switch prototype in order to cache the significant rules into hardware switches. In SDN, each rule has a match filed, a priority field, a counter field, and an action field. We show an example in Fig. 1. In this example, we assume that the match field has two dimensions, field 1 and field 2, and higher priority number has higher priority order.

The wildcard-rule caching problem is easy to solve if the rules have disjoint field space. However, wildcard rules would have rule dependency with each other. The two rules have direct dependency if their field space overlapped with each other. In Fig. 1, assume the weights of rules R_1, R_2, R_3, R_4, R_5, and R_6 are 3, 4, 5, 6, 7, and 8, respectively. If the TCAM can store four rules, we cannot select the four rules with the highest weight (i.e., R_3, R_4, R_5, and R_6). This is because a packet with pattern 000000 that should match R_1 would match R_3. Similarly, a packet with pattern 010011 that should match R_2 would match R_4. That is, rules R_3 and R_4 depend on (overlap) rules R_1 and R_2, respectively.

The cover-set method [15] can be used to solve the wildcard-rule dependency problem. We can transform the rule dependency into a directed acyclic graph (DAG). We add a directed edge from the higher priority rule to the lower priority rule if they have rule dependency. We show the DAG of the Fig. 1 and the weight of each rule in Fig. 2(a). The match field of R_1 overlaps with the match field of R_2, so we add a directed edge between them. Now, if we cache R_5 into TCAM, a new cover-set rule R_3* will be created and cached into TCAM as shown in Fig. 2(b). Cover-set rule R_3* is used to forward the packets matching the overlapping region of R_3 and R_5 to the software switches. Thus, the match field of cover-set rule R_3* is the same as R_3 but the action field of cover-set rule R_3* will be modified to *forward to software switches*. For example, a packet with pattern 101000 would match R_3* and will be forwarded to the software switches.

Rule	Match		Priority	Action
	Field 1	Field 2		
R_1	000	0**	6	Forward
R_2	0**	011	5	Drop
R_3	***	000	4	Forward
R_4	01*	01*	3	Modify Forward
R_5	10*	***	2	Drop
R_6	11*	***	1	Modify Forward

Figure 1. Network policy with six wildcard rules.



(a) Direct acyclic graph (b) DAG with weight

Figure 2. Direct acyclic graph of Fig. 1 and example of cover-set method.

B. k-Hop Neighboring Set (k-HNS) Algorithm

In wildcard-rule caching, there are a weight and a cost for each rule. Each wildcard rule is associated with a weight value.

The weight value represents the access frequency of a wildcard rule in SDN switches. In general, a rule with larger field space has a higher probability to be matched by more packets than a rule with smaller field space. Therefore, the number of packets expected to match a rule R_i as the weight of R_i, and the number of required TCAM space for caching R_i and its cover sets as the cost of R_i. Let the individual contribution value (CV) of R_i be the weight of R_i divided by the cost of R_i. For example, in Fig. 2(b), the individual CV of R_5 is 7/2 since the weight of R_5 is 7 and we need to cache the cover set R_3* when we cache R_5. Therefore, given a network policy (i.e., a set of wildcard rules) and a TCAM, we propose a wildcard-rule caching algorithm to maximize the total weight of the cached rules but the total cost should not exceed the TCAM size. The wildcard-rule caching problem can be reduced from an all-neighbors knapsack problem [18], which is an NP-hard problem. Consequently, we propose a heuristic algorithm to solve this problem.

Let k-hop neighbors of a rule R_i be the rules within k-hop distance of R_i. The k-hop distance of a rule r is the rules that can reach r in k-hop or the rules that can be reached by r in k-hop. If a rule x is 1-hop distance of a rule y and y is 1-hop distance of rule z, x is 2-hop distance of z. Fig. 3 shows the 1-hop and 2-hop neighbors of a rule R_i. Let k-hop CV of R_i be the CV of R_i and some of its k-hop neighbors, which has the maximum CV within the k-hop neighbors of the rule R_i. Our k-HNS algorithm first calculates the k-hop CV of each un-cached rule in a given DAG. Second, we will cache the rule and its k-hop neighbors which have the maximum k-hop CV into the TCAM. Then, we will update the cost of all the un-cached rules in the DAG since some of the rules have been cached into the TCAM and the cost of each un-cashed rule may be changed. We repeat the above steps until there is no TCAM space.

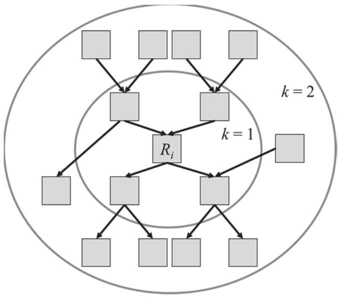


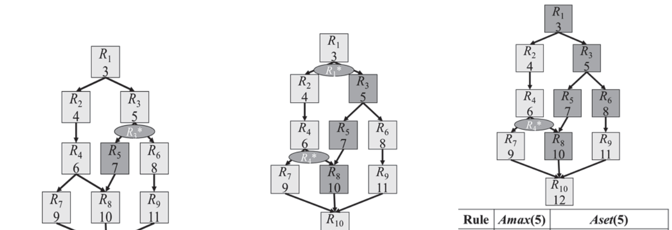
Figure 3. Illustration of the k-hop neighboring set of rule R_i.

In the following, we detail our k-HNS algorithm. First, we sort the individual CV of the un-cached rules in decreasing order. Then, we calculate the k-hop CV of each un-cached rule following the sorting order. The k-hop CV of a rule R_i is calculated in each hop from hop 0 to hop k. In each hop, we try to find a set of rules, which have the maximum accumulated CV in this hop. The accumulated CV is the total weight of all the combined rules divided by their cost. Let $Amax(i)$ and $Aset(i)$ denote the k-hop CV of rule R_i and its corresponding set of rules, respectively. Let $Rmax$ be the maximum k-hop CV among all the un-cached rules. Thus, $Rmax = \max_{1 \le i \le N} Amax(i)$, where N is the number of un-cached rules in each round. Note that, the number of un-cached rules N will decrease after some rules are cached into the TCAM. Let $Rset$ be the set of rules corresponding to $Rmax$. When we find the $Rmax$, we will cache the rules in $Rset$ to TCAM. The above steps are repeated until TCAM space is full.

To calculate the k-hop CV of a rule R_i, we set $Amax(i)$ = the individual CV of rule R_i and $Aset(i) = \{R_i\}$ initially. Then, we update the 1-hop neighboring cost of R_i and sort them in decreasing order according to their individual CV. If more than one rule has the same CV, the lower cost one has higher priority. To calculate the 1-hop CV of R_i, we will combine its 1-hop neighbors one by one in decreasing order if they satisfy the following condition. For each rule r, if the accumulated CV of the rules $Aset(i) \cup r$ is larger than $Amax(i)$ and the total cost of $Aset(i) \cup r$ is no larger than the available TCAM space, let $Aset(i) = Aset(i) \cup r$ and update the $Amax(i)$ to the new accumulated CV. After each of the 1-hop neighbors is checked, we have the maximum accumulated CV $Amax(i)$ and its corresponding set $Aset(i)$ in this hop. After finding the $Amax(i)$ and $Aset(i)$ in the 1-hop neighbors of R_i, we update the 2-hop neighbors cost of R_i which are connected to the rules in the $Aset(i)$ and repeat the above steps to find $Amax(i)$ and $Aset(i)$ for 2-hop of R_i and so on.

For example, in Fig. 4, we assume the TCAM size is 6 and there are 10 rules. We can calculate the maximum 2-hop CV of R_5 as follows. As shown in Fig. 4(a), the cost of R_5 is two because we need to cache the cover-set rule R_3* when caching R_5. Thus, $Amax(5) = 7/2$ and $Aset(5) = \{R_5\}$ initially. Then, we will update the cost of R_3 and R_8 which are the 1-hop neighbors of R_5. The initial cost of R_3 and R_8 are 2 and 3, respectively. We can decrease the cost of R_3 by one since the cover-set rule R_3* can be removed if we combine R_3 into the rule set $Aset(5)$. Similarly, we can decrease the cost of R_8 by one since we do not need to cache the cover-set rule R_5*. Therefore, the new individual CV of R_3 and R_8 are 5/1 and 10/2, respectively. Although R_3 and R_8 have the same individual CV, we select R_3 first due to its cost is lower than R_8. When we combine R_3 into the rule set $Aset(5)$, the total accumulated weight is 7+5 and the total cost is 2+1. Thus, the 1-hop accumulated CV of R_3 and R_5 is 12/3 which is larger than $Amax(5) = 7/2$. Thus, the new CV of $Amax(5)$ is updated to 12/3 and the corresponding set $Aset(5) = \{R_5, R_3\}$. Next, we check the potential rule R_8. The total weight includes R_8 is 12+10 and the total cost is 3+2, so the 1-hop accumulated CV of R_3, R_5, and R_8 is 22/5 which is larger than 12/3. In Fig. 4(b), we update the $Amax(5)$ and $Aset(5)$ to 22/5 and $\{R_5, R_3, R_8\}$, respectively.

After that, the 2-hop cost of R_1, R_6, R_4 and R_{10} are updated to 0, 1, 1 and 3, respectively. Since the cost of R_1 is zero, we will combine it into $Aset(5) = \{R_5, R_3, R_8, R_1\}$ first and update $Amax(5)$ to 25/5. Next, we combine R_6 and the accumulated CV is 33/6 which is larger than 25/5. So, we update the $Amax(5)$ and $Aset(5)$ to 33/6 and $\{R_5, R_3, R_8, R_1, R_6\}$, respectively. When we try to combine R_4 or R_{10}, the total cost will larger than the available TCAM space, so we do not combine any one of them. Fig. 4(c) shows the maximum 2-hop CV of R_5 and its corresponding rule set.



Rule	Amax(5)	Aset(5)
R_5	7/2	$\{R_5\}$

Rule	Amax(5)	Aset(5)
R_5	7/2	$\{R_5\}$
R_3	12/3	$\{R_5, R_3\}$
R_8	22/5	$\{R_5, R_3, R_8\}$

Rule	Amax(5)	Aset(5)
R_5	7/2	$\{R_5\}$
R_3	12/3	$\{R_5, R_3\}$
R_8	22/5	$\{R_5, R_3, R_8\}$
R_1	25/5	$\{R_5, R_3, R_8, R_1\}$
R_6	33/6	$\{R_5, R_3, R_8, R_1, R_6\}$

(a) Initial condition (b) After combine 1-hop neighbors of R_5 (c) After combine 2-hop neighbors of R_5

Figure 4. Example of calculating the 2-hop CV of R_5 with TCAM size = 6.

After finding the k-hop CV of each un-cached rule, we can find the $Rmax$ and cache the corresponding rules in $Rset$ to TCAM. For example, Table 3.1 lists the 2-hop CV of the ten rules in Fig. 4. At the first round, R_3 has the maximum 2-hop CV (= 6.8) among the 10 rules as shown in Table 3.1(a). We cache the rules in $Rset = Aset(3) = \{R_3, R_1, R_6, R_5, R_9\}$ into the TCAM. The remaining TCAM space is one. Next, we update the individual CV of each un-cached rule in the DAG and go on next round. At the second round, we recalculate the 2-hop CV of each un-cached rule as shown in Table 3.1(b). Since the cost of R_4, R_7, R_8 and R_{10} are larger than one, we cache R_2 (cost = 1) into the TCAM and the total CV of all cached rules is (34+4)/(5+1) as shown in Fig. 5. The *k*-HNS algorithm is summarized in Algorithm 1. In order to reduce the time complexity of our algorithm, the maximum number of un-cached rules used to calculate their *k*-hop CVs is set to a constant k_1. We also limit the maximum number of the neighboring rules checked in each hop to a constant k_2.

TABLE 3.1.

The 2-HNS CV of ten rules in Fig. 4 with TCAM size = 6.

(a) First round

Rule	*Amax*()	*Aset*()
R_1	23/4 = 5.75	$\{R_1, R_3, R_5\}$
R_2	27/5 = 5.4	$\{R_2, R_1, R_4, R_9, R_3\}$
R_3	34/5 = 6.8	$\{R_3, R_1, R_6, R_5, R_9\}$
R_4	15/3 = 5	$\{R_4, R_7\}$
R_5	33/6 = 5.5	$\{R_5, R_3, R_8, R_1, R_6\}$
R_6	19/3 = 6.33	$\{R_6, R_9\}$
R_7	15/3 = 5	$\{R_7, R_4\}$
R_8	28/6 = 4.67	$\{R_8, R_5, R_4, R_3\}$
R_9	19/3 = 6.33	$\{R_9, R_6\}$
R_{10}	32/6 = 5.33	$\{R_{10}, R_9, R_7\}$

(b) Second round

Rule	*Amax*()	*Aset*()
R_2	4/1 = 4	$\{R_2\}$
R_4	-1	$\{\emptyset\}$
R_7	-1	$\{\emptyset\}$
R_8	-1	$\{\emptyset\}$
R_{10}	-1	$\{\emptyset\}$

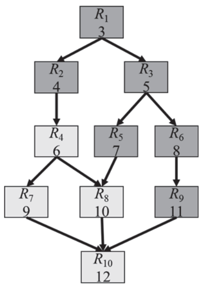


Figure 5. Final caching result of Fig. 4 by *k*-NHS algorithm.

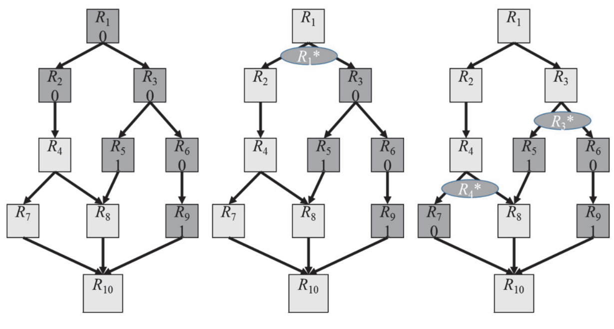
Algorithm 1 : *k*-Hop Neighboring Set (k-HNS) Algorithm

1. Calculate the individual CV for each rule in *DAG*;
2. **while** *Avail_TCAM* > 0 **do**
3. Heap sort the un-cached rules in *DAG* by their individual CV in decreasing order; (Without loss of generality, we assume that the sorting order is R_1, R_2, R_3 and so on.)
4. **for** i = 1 to k_1 **do**
5. $Amax(i)$ = the individual CV of R_i and $Aset(i) = \{R_i\}$;
6. **for** h = 1 to k **do**
7. **if** the total cost of $Aset(i) \leq Avail\_TCAM$ **do**
8. Sort the h-hop neighbors of R_i by their individual CV in decreasing order;
9. **for** each rule r in the first k_2 h-hop neighbors of R_i **do**
10. **if** the accumulated CV of the rules in $Aset(i)$ and r > $Amax(i)$ **and** their total cost < *Avail_TCAM* **then**
11. $Aset(i) \leftarrow Aset(i) \cup r$;
12. Update the $Amax(i)$ to the new accumulated CV;
13. **end if**
14. **end for**
15. Update the (h+1)-hop neighbors' cost of R_i;
16. **end for**
17. **end for**
18. $Rmax = \max_{1 \leq i \leq k_1} Amax(i)$ and *Rset* is the set of rules corresponding to *Rmax*;
19. Cache the rules in *Rset*;
20. $Avail\_TCAM \leftarrow Avail\_TCAM$ – the total cost of the rules in *Rset*;
21. Let *Dirty* = the 1-hop neighbors of the rules in *Rset*;
22. Update the cost of each rule in *Dirty* and calculate their individual CV;
23. **end while**

C. Neighboring Set Replacement (NSR) Algorithm

After using our *k*-HNS algorithm to cache the rules into the TCAM, the packets go through the switches could match these rules in the TCAM or the cache miss rules which are not cached in the TCAM. As soon as cache miss happens, our cache replacement algorithm, Neighboring Set Replacement (NSR), will evict some rules and cache the cache miss rule. In our NSR algorithm, we consider both the temporal and spatial localities. Temporal locality is that the repetitive packets will appear again in a short period. Spatial locality is that the traffic would concentrate at some block of the field space during a short period. In our NSR algorithm, we assign a 1-bit counter for each cached rule due to the consideration of temporal locality. The bit of a counter is called referenced bit. When a packet matches the rule in the TCAM, we set the counter of the rule to one. A victim rule is the rule, which can release the maximum TCAM space among the cached rules whose counter value is zero. Let *Ovictim* denote as the original victim rule that we first choose to evict out of TCAM at each round and *Victim* denote as the victim rules we select from the 1-hop neighbors of *Ovictim*. When the cache miss occurs, we repeat the following two steps until we can store the cache miss rule. First, we evict out the rule *Ovictim*. Second, we find *Victim* from the *Ovictim* and remove them out of the TCAM. For the cache miss rule, if the number of its 1-hop neighbors cached in the TCAM is more than one half, it represents that there are many traffics near the field space of the miss rule. Therefore, we set the default counter of the miss rule and its 1-hop neighbors cached in the TCAM to one according to the spatial locality. Otherwise, we set the default counter of the miss rule to zero.

Fig. 6 shows the example that the cache miss rule is R_7. First, the cost of R_7 and its cover set R_4* is equal to two. In Fig. 6(a), R_1, R_2, R_3 and R_6 have the same counter value 0. We choose the rule R_2 as *Ovictim* and it can release the TCAM space is one. However, the available space of TCAM is not enough to cache R_7. We find victim rules in the 1-hop neighbors of R_2. R_1 is the only cached rule whose counter value is 0, so we choose R_1 as *Victim*. Although R_1 can release one TCAM space, we need to add R_1* to the TCAM entry. The available space of TCAM is still not enough to cache R_7 as shown in Fig. 6(b). Since there are no other 1-hop neighbors of R_2 cached in the TCAM, we choose R_3 as *Ovictim*. After we evict R_3, the available TCAM space is enough to cache R_7. Since the number of 1-hop neighbors of R_7 cached in the TCAM is less than one half, we set the default counter value of R_7 to zero as shown in Fig. 6(c). We summary our NSR algorithm in Algorithm 2.



(a) DAG before replacement (b) DAG after first round (c) DAG after replacement

Figure 6. Example of our NSR algorithm with TCAM size = 6.

Algorithm 2 : Neighboring Set Replacement (NSR) Algorithm

1. **for** each packet p in *Packets* **do**
2. **if** the rule R_i matched by the packet p is cached in the TCAM **then**
3. $R_i.counter \leftarrow 1$;
4. **else**

```
    while the cost of R_i > Avail_TCAM do
      if we can find Ovictim then
        Release the space of Ovictim from TCAM;
        Avail_TCAM ← Avail_TCAM + the cost of Ovictim;
        while the cost of R_i > Avail_TCAM do
            if we can find Victim then
              Release the space of Victim from TCAM;
              Avail_TCAM ← Avail_TCAM + the cost of Victim;
            else
              break;
            end else
          end while
        else
          We reset counter of each rule cached in the TCAM;
          continue;
        end else
    end while
    if the number of R_i's 1-hop neighbors cached in the TCAM is more than one half, we set the counter of R_i and its 1-hop neighbors in the TCAM to 1. Otherwise, we set the counter of R_i to 0;
    Cache R_i into the TCAM;
  end else
end for
```

D. Time Complexity

Assume there are N rules in a network policy. The number of each rule's 1-hop neighbors is at most N. In Algorithm 1, we first calculate the individual CV for each rules in the DAG and the time complexity is $O(N^2)$. In each round, we heap sort the un-cached rules in decreasing order and choose the first k_1 rules to calculate their k-hop CVs. To calculate the k-hop CV of a rule R_i, we combine its neighboring rules from 1-hop to k-hop neighbors. At each hop, we sort the neighboring rules by their individual CV in deceasing order. Then we combine the first k_2 sorted rules in decreasing order. Finally, we update the cost of the next-hop neighbors of the rule R_i. Thus, the time complexity of combining the neighboring rules at each hop is $O(N \log N + k_2 N) = O(N \log N)$. Since there are at most k hops, the total time complexity of calculating the k-hop CV for a rule is $O(k * N \log N) = O(N \log N)$. Then, we choose *Rmax* and cache the corresponding rule set *Rset*. Finally, we recalculate the individual CV of dirty rules for the next round. Hence, the time complexity of each round is $O(N \log N + k_1 N \log N + N^2) = O(N^2)$. Assume that we have T entries in the TCAM, there are at most $O(T)$ rounds. Therefore, the total time complexity of our k-HNS algorithm is $O(N^2 + TN^2) = O(TN^2)$ which is the same as the previous work.

IV. Simulations

In this section, we first describe our simulation environment. Second, we compare the caching ratio of our k-hop neighboring set algorithm with the CSC algorithm in [15] and the WRC algorithm in [16]. The caching ratio is the total weight of cached rules and the total weight of all the rules in the DAG. Third, we compare the cache hit ratio of our cache replacement algorithm with the traditional cache replacement algorithms and RCR algorithm in [16]. Each simulation result is the average of 10 times.

A. Simulation Environment

We use ClassBench [19] to generate synthetic network policies with sets of wildcard rules. We use C language to write simulator for wildcard-rule caching and cache replacement algorithms. In our simulations, we use filter set generator of ClassBench to generate six policies with different parameter files. Each policy contains 10,000 rules matching on five fields namely source IP address, destination IP address, source port, destination port, and protocol number. We set the weight value of each rule according to the size of their field space. In other words, we set higher weight value to the rule with larger field space since it has higher probability to be matched by packets. We use equation (1) to normalize the weight value of each rule. In the rear world, we can obtain the weight of rules by the statistics of the packet flows in the SDN controller. After caching the rules into TCAM by using our k-hop neighboring set algorithm, we feed the packets generated by the ClassBench to the switches. We evaluate the cache hit ratio of cache replacement algorithms when these packets go through the switches. Moreover, we use different temporal and spatial parameters to change the traffic condition for our evaluation.

$$Weight(R_i) = \frac{no.\ of\ "*"}{96} \log_2 10{,}000 \qquad (1)$$

B. Simulation Results

In our k-HNS algorithm, we use k_1 and k_2 to reduce the time complexity of our algorithm. The k_1 is used to constraint the maximum number of un-cached rules chosen to calculate their k-hop CVs and k_2 is used to constraint the maximum number of the neighboring rules used in each hop. In our following simulations, we set k_1 and k_2 to 100 and 60, respectively. Figure 7 shows the caching ratio of our k-HNS algorithm with $k = 1$, $k = 2$, and $k = 3$ under different TCAM sizes. Since our k-HNS algorithm with $k = 2$ has similar performance to our algorithm with $k = 3$, we set $k = 2$ in the following simulations.

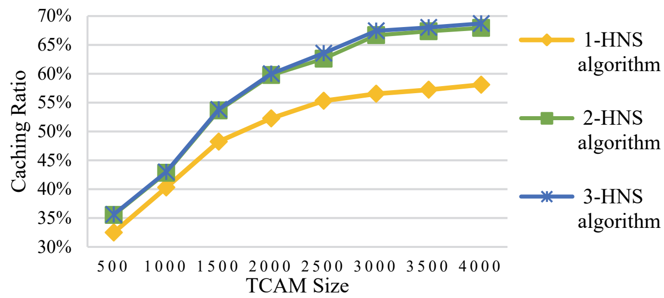


Figure 7. Average caching ratio vs. TCAM size with $k = 1$, 2, and 3.

Figure 8 shows the result of three caching algorithms with different TCAM sizes. When the TCAM space is 500, our k-HNS algorithm, the CSC algorithm, and the WRC algorithm have similar caching ratio since they tend to cache the lowest priority rules, which have higher weight in the beginning. As soon as the TCAM space grows more than 500 entries, the CSC algorithm has the lowest caching ratio since it only considers the CV of individual rule. The WRC algorithm and our k-HNS algorithm consider the accumulated CV of a group of rules, which makes caching ratio is higher than that of the CSC algorithm. However, our k-HNS algorithm could group the set of rules with higher total weight than the WRC algorithm.

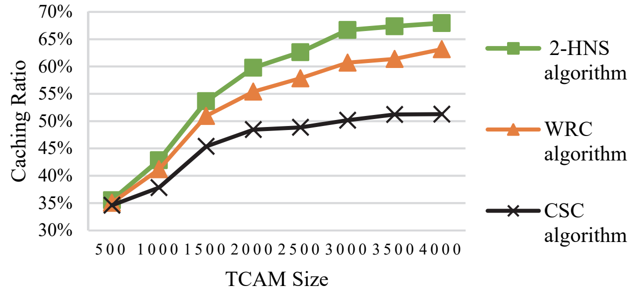


Figure 8. Average caching ratio with different TCAM sizes.

In the real network communications, there are temporal and spatial traffic localities. However, the input packets generated by the ClassBench trace generator only represents temporal traffic locality and do not represent spatial traffic locality. Therefore, we generate extra spatial locality packets in the input packets to evaluate the performances of different cache replacement algorithms. We use a spatial-locality variable *SL* [16] to limit the number of extra-generated spatial locality packets. The larger SL represents the higher spatial locality.

Figure 9 shows the result for cache replacement algorithms influenced by different spatial-locality variable *SL*. In the simulation, the TCAM space is set to 2,000 entries. Our NSR algorithm and RCR algorithm both consider spatial locality, so the performance of these two algorithms is higher than the other cache replacement algorithms. Our NSR algorithm evicts out a set of rules at each round and assigns the default counter value for a cache miss rule by considering the number of its 1-hop neighbors cached in the TCAM. As a result, the cache hit ratio of our NSR algorithm is higher than RCR algorithm, and much better than NRU, LRU and ARC algorithms, respectively.

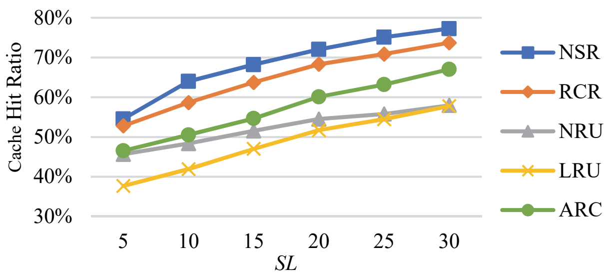


Figure 9. Cache hit ratio with different SL sizes.

In Fig. 10, we show the cache hit ratio of cache replacement algorithms by utilizing the different TCAM space. We set *SL* to 20, which is the medium value of the spatial locality evaluation. Our NSR algorithm still has higher cache hit ratio than RCR algorithm due to our strategy for the consideration of spatial locality and the 1-bit counter for the consideration of temporal locality. For the smaller TCAM space, the interval of repetitive packets is larger than the TCAM space. Therefore, LRU gets lower hit ratio than the other cache replacement algorithms. When the TCAM space become larger, LRU can maintain temporal locality to get better performance.

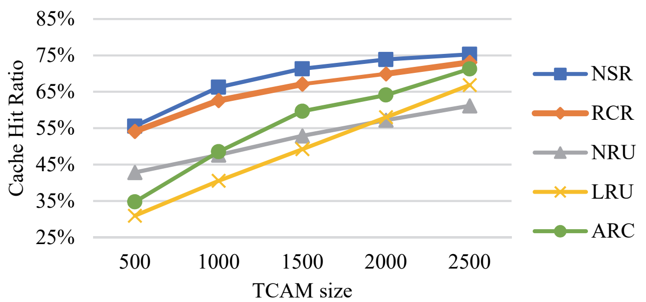


Figure 10. Cache hit ratio with different TCAM sizes.

V. Conclusion

Wildcard-rule caching is an efficient technique to deal with TCAM capacity problem. Although there is rule dependency problem in wildcard-rule caching, we can utilize cover-set method to solve this problem. Our *k*-HNS algorithm can cache a set of rules into TCAM and has better capability to build the set with higher total weight for each rule. On the other hand, our NSR algorithm both considers temporal and spatial localities, which make the cache hit ratio high. We use six different policies generated by filter set generator of ClassBench simulator to simulate our two proposed algorithms. By the simulation results, our *k*-HNS algorithm has higher caching ratio than CSC and WRC algorithms. Our cache replacement algorithm NSR has higher cache hit ratio than RCR, NRU, LRU and ARC algorithms.

References

[1] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking Control of the Enterprise," *Proceedings of the ACM SIGCOMM*, pp. 1-12, Kyoto, Japan, Aug. 2007.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69-74, April 2008.

[3] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter, "PAST: Scalable Ethernet for Data Centers," *Proceedings of the 8th ACM CoNEXT*, pp. 49-60, New York, USA, Dec. 2012.

[4] A. X. Liu, C. R. Meiners, and E. Torng, "TCAM Razor: A Systematic Approach Towards Minimizing Packet Classifiers in TCAMs," *IEEE/ACM Trans. on Networking*, vol. 18, no. 2, pp. 490-500, April 2010.

[5] C. R. Meiners, A. X. Liu, and E. Torng, "Bit weaving: A Non-Prefix Approach to Compressing Packet Classifiers in TCAMs," *IEEE/ACM Trans. on Networking*, vol. 20, no. 2, pp. 488-500, April. 2012.

[6] W. Braun and M. Menth, "Wildcard Compression of Inter-domain Routing Tables for OpenFlow-Based Software-Defined Networking," *IEEE 3rd European Workshop on Software Defined Networks*, pp. 25-30, Budapest, Hungary, Sep. 2014.

[7] N. Yukihiro, H. Kazuki, L. Chunghan, K. Shinji, S. Osamu, and S. Takeshi, "DomainFlow: Practical Flow Management Method using Multiple Flow Tables in Commodity Switches," *Proceedings of the 9th ACM CoNEXT*, pp. 399-404, Santa Barbara, CA, USA, Dec. 2013.

[8] K. Yossi, H. David, and K. Isaac, "Palette: Distributing Tables in Software-Defined Networks," *Proceedings of the IEEE INFOCOM*, pp. 545-549, Turin, Italy, April 2013.

[9] N. Kang, Z. Liu, J. Rexford, and D. Walker. "Optimizing the "One Big Switch" Abstraction in Software-Defined Networks," *Proceedings of the 9th ACM CoNEXT*, pp. 13-24, Santa Barbara, CA, USA, Dec. 2013.

[10] Y. Liu, S. O. Amin, and L. Wang, "Efficient FIB Caching Using Minimal Non-Overlapping Prefixes," *ACM SIGCOMM Computer Communication Review*, vol. 43, issue 1, pp. 14-21, Jan. 2013.

[11] H. Huang, S. Guo, P. Li,W. Liang, and A. Y. Zomaya, "Cost Minimization for Rule Caching in Software Defined Networking," *IEEE Trans. on Parallel and Distributed Systems*, vol. 27, issue 4, pp. 1007-1016, May 2015.

[12] H. Li, S. Guo, C. Wu, J. Li, "FDRC: Flow-Driven Rule Caching Optimization in Software Defined Networking," *Proceedings* of *ICC*, pp. 5777-5782, London UK, June 2015.

[13] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable Flow-Based Networking with DIFANE," *Proceedings of ACM SIGCOMM*, pp. 351-362, New Delhi, India, Aug. 2010.

[14] B. Yan, Y. Xu, H. Xing, K. Xi, and H. J. Chao, "CAB: A Reactive Wildcard Rule Caching System for Software-Defined Networks," *Proceedings of the 3rd Workshop on Hot Topics in Software Defined Networking*, pp. 163-168, Chicago, IL, USA, Aug. 2014.

[15] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Infinite CacheFlow in Software-Defined Networks," *Proceedings of the 3rd Workshop on Hot Topics in Software Defined Networking*, pp. 175-180, Chicago, IL, USA, Aug. 2014.

[16] J. P. Sheu, and Y. C. Chuo, "Wildcard Rules Caching and Cache Replacement Algorithms in Software-Defined Networking," *IEEE Trans. on Network and Service Management*, vol. 43, issue 1, pp. 19-29, March 2016.

[17] N. Megiddo and D. S. Modha, "Outperforming LRU with an Adaptive Replacement Cache Algorithm," *IEEE Computer*, vol. 37, no. 4, pp. 58–65, April 2004.

[18] G. Borradaile, B. Heeringa, and G. Wilfong, "The Knapsack Problem with Neighbour Constraints," *Journal of Discrete Algorithms*, vol. 16, pp. 224-235, Oct. 2012.

[19] D. E. Taylor and J. S. Turner, "ClassBench: A Packet Classification Benchmark," *IEEE/ACM Trans. on Networking*, pp. 499-511, vol. 15, no. 3, June 2007.