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

Jang-Ping Sheu, *Fellow, IEEE*, and Yen-Cheng Chuo

***Abstract*—In software-defined networking, flow tables of OpenFlow switches are implemented by ternary content addressable memory (TCAM). Although TCAM can process input packets in high speed, it is a scarce and expensive resource providing only a few thousands of rule entries on a network switch. Rules caching is a technique to solve the TCAM capacity problem. However, the rule dependency problem is a challenging issue for wildcard rules caching where packets can mismatch rules. In this paper, we use a cover-set approach to solve the rule dependency problem and cache important rules to TCAM. We also propose a rule cache replacement algorithm considering the temporal and spatial traffic localities. Simulation results show that our algorithms have better cache hit ratio than previous works.**

***Index Terms*—Software-defined-networking, TCAM, rule dependency problem, wildcard rules caching algorithm, cache replacement algorithm.**

I. Introduction

SOFTWARE-Defined Networking (SDN) is a promising network architecture that enables the network control with more efficiency and flexibility. In the recent network architecture, decision of routing path and packet forwarding both happen in switches or routers. The switches or routers must encapsulate and decapsulate packets constantly. Therefore, they cannot utilize their network bandwidth efficiently. Besides, network administrator needs to set up switches or routers one by one while adjusting customized network protocols. The feature of SDN is decoupling control components from underlying devices, dividing network into control plane and data plane. Because the controller manages and operates a network from a global view of the network, it can utilize network bandwidth efficiently or facilitate customized protocol updating [1]–[3].

There are many available controller platforms implemented by different programming languages (e.g. NOX, Floodlight, or Ryu) [4]–[6]. However, SDN requires a common method for the controller to communicate with underlying devices. One such mechanism, OpenFlow [7], lets the controller or network administrator remotely control routing table (flow table). The controller will find routing path for each flow, and install rules in the flow tables of the corresponding OpenFlow switches or routers. Each flow table entry consists of three fields (i.e. match, actions, and counters). The match field determines which packets can match the rules. Once a packet matches a rule, the OpenFlow device will apply actions in actions field to the packet. Otherwise, the packet will be dropped or sent to the controller by default rule. Counters field is used when controller desires statistics of the network such as link bandwidth or error rate.

In the *de facto* industry standard, the flow table is implemented by TCAM [8], [9]. TCAM is a high-speed memory which can match packet headers against stored entries in parallel. The controller can install exact-match rules or wildcard-match rules in the TCAM [7], [10]. Although TCAM can match packet headers with rules in the wire speed, network devices are equipped with limited TCAM size because they are expensive hardware and extremely power-hungry [11], [12].

Previous work on TCAM size capacity problem falls into three main categories, packet classification compression [13], [11], rules distribution along traffic route [14]–[16], and rules caching [17]–[21], [23] respectively. Packet classification compression techniques would merge two combinable rules into a new wildcard rule. So, we can get another semantically equivalent smaller packet classification. In rules distribution along traffic route, we distribute safety rules across the network according to routing policies to reduce the requirement of a large flow table. In the rules caching system, the most popular rules are cached in the small TCAM, while relying on software to handle the small amount of "cache miss" traffic. Thus, the application in the control plane can have the abstraction of arbitrarily large flow table in the OpenFlow switch. In this paper, we focus on the rules caching system.

In the rules caching system, since wildcard rules may overlap with each other in the field space, packet classification policy assigns different priorities to different rules to avoid conflicts. If one packet matches multiple rules in a flow table, the OpenFlow switch will apply the actions on the highest priority matched rule to the packet. Unfortunately, the rule dependency problem needs to be solved in the wildcard rule caching system [17], [19]. For example, a high-priority rule and a low-priority rule overlap with each other in the field space. The rule dependency problem is that if we only cache the low-priority rule in the TCAM, the packets falling in the overlapping regions of field space would incorrectly match the low-priority rule (because these packets should match the high-priority rule). To maintain the semantic correctness of packet matching, extra cache cost is inevitable.

Here, we propose a wildcard rules caching algorithm and a rule cache replacement (RCR) algorithm. Our wildcard rules caching algorithm will cache the frequently matched wildcard

Manuscript received August 31, 2015; revised November 29, 2015 and February 7, 2016; accepted February 7, 2016. Date of publication February 16, 2016; date of current version March 9, 2016. This work was supported by the Ministry of Science and Technology of Taiwan, R.O.C., under Grants 103-2221-E-007-067-MY3 and 104-2622-8-009-001. The associate editor coordinating the review of this paper and approving it for publication was L. Granville.

The authors are with the Department of Computer Science, National Tsing Hua University, Hsinchu 30013, Taiwan (e-mail: sheujp@cs.nthu.edu.tw; s102062560@m102.nthu.edu.tw).

Digital Object Identifier 10.1109/TNSM.2016.2530687

rules with less extra cache cost. After caching wildcard rules in the TCAM, once cache miss occurs, the RCR algorithm would find victim rules and replace them with the cache miss rule according to current temporal and spatial traffic localities. This paper has following two contributions. First, the wildcard rules caching algorithm can utilize the TCAM space more efficiently compared with the cover-set caching algorithm in [19]. Second, the RCR algorithm considering traffic locality would maintain the hit ratio high. Simulation results show that our wildcard rules caching algorithm performs better in cache hit ratio than cover-set caching algorithm. And, the RCR algorithm gets higher cache hit ratio than least recently used (LRU) algorithm, random replacement (RR) algorithm, and adaptive replacement cache (ARC) algorithm [22].

We organize the rest of this paper as follows. In Section II, we introduce the related works of wildcard rules caching. We present our algorithms in Section III. Simulation results are shown in Section IV. We conclude the paper in Section V.

II. Related Works

Previous work on TCAM size capacity problem falls into three main categories, packet classification compression, rules distribution along traffic route and rules caching, respectively.

A. Packet Classification Compression

Packet classification compression tries to use less number of TCAM entries to represent another semantically equivalent packet classification [13]. The compression scheme proposed in [13] can only be used in prefix classifier. Prefix classifier is a kind of packet classification where each field of a rule's predicate is specified as a prefix. All the *'s are at the end of the ternary string. Bit Weaving [11] is the first compression scheme for non-prefix packet classifiers. In Bit Weaving, if match fields of two entries differ by only one bit and their action fields are same, these two entries can be merged. The difference bit is replaced with a wildcard bit (*). We could either use Bit Weaving alone or make Bit Weaving as a post-processing routing to cope with other compression schemes. Nevertheless, the network application or administrator cannot get statistics of the original rule (e.g. flow count). The reason is that when we combine two rules, both of their counter fields are also merged. We can only get the combined statistics rather than separated statistics of the original rule.

B. Distributing Rules Along Traffic Route

Generally, the size of a packet classification policy is much larger than the capacity of an OpenFlow switch. The concept of rules distribution along traffic route is decomposing a large packet classification policy into smaller ones and then distributing them across the network. If packets enter the network, they must traverse along their routes in the network to perform the complete packet classification. In [15], the authors divide a large packet classification policy into two smaller ones iteratively, named Pivot Bit Decomposition (PBD). PDB chooses an appropriate bit in the field space as pivot bit. Rules with 0 in the pivot bit are collected together in a new flow table. The rules with 1 in the pivot are collected in another one. Rules with don't care (*) will be replicated in both of flow tables.

In [16], the authors divide the network policies into two categories, Endpoint policies and Routing policies. Endpoint policies specify ingress and egress points of the network for each flow. In other words, Endpoint policies do not care the detail network topology. Routing policies specify paths between ingress and egress points for each flow. The authors in [16] use linear programming scheme to distribute the Endpoint policies along the traffic route as specified by routing policies. Therefore, a large packet classification can be shared by switches in the traffic route. Nevertheless, only safety rules (e.g. access control or modifying packet header) can be distributed across the network. Safety rules must conform following two conditions. First, the match field of the safety rule does not specify a switch port. Second, there is no execution order in the safety rules.

C. Rules Caching

The idea of rule caching is to cache the important or needed rules in the TCAM of the switches. In [20], the authors proposed a forwarding information base (FIB) caching scheme that stores only non-overlapping FIB entries into the fast memory (FIB cache) while storing the complete FIB in slow memory. FIB caching is different from traditional caching schemes which may cause a cache-hiding problem. If a packet has a matching prefix in the cache, it may not be the correct entry for forwarding the packet if there is a longer matching prefix in the full FIB. The proposed scheme can prevent the cache-hiding problem. In [21], the authors aim to speed up the rule matching of high entropy packet fields. The high entropy packet fields are the values of packet fields which are frequently changed from packet to packet flowing through a switch, like layer 4 port field. The proposed algorithm can support flow caching of forwarding decisions including L4 headers, without requiring every new forwarded transport connection to be handled by the slow path. The authors in [23] studied the rule caching problem to minimize the sum of remote controller processing cost and TCAM occupation cost. One off-line algorithm and two on-line algorithms were proposed in this paper.

The rule dependency problem is an important challenge for wildcard rules caching. The rule dependency problem will decrease the efficiency of wildcard rules caching. To solve the rule dependency problem, the authors in [18] converts wildcard rules to a set of new micro wildcard rules without overlapping. In [17], the authors utilize the two-stage pipeline property of flow tables in OpenFlow switch. They partition the full field space into small hyper-rectangles or buckets which are non-overlapping with each other. The OpenFlow switch will cache the bucket matched by current packets in the first stage of flow tables. At the same time, all wildcard rules falling in the field space of the bucket are cached in the second stage of flow tables. With such a bucket scheme, the control bandwidth can be saved and the semantic correctness of packet matching can also be complied.

The authors in [19] proposed a cover-set concept to deal with the rule dependency problem. The cover-set concept is that we calculate new rules that cover these packets which would incorrectly match the low-priority rule. The authors find immediate ancestors of the low-priority rule in a dependency graph as the cover-sets of the low-priority rule, but the actions of cover-sets are replaced with *forward_to_SW_switch* actions (forward to software switch). The authors splice the long chains of dependent rules by creating cover-sets for each rule. Cover-sets can help us to avoid caching high-weight rules along with lots of low-weight dependent rules. For each un-cached rule R, the authors calculate the ratio of the expected number of packets matching R to the number of required TCAM entries for caching R as a contribution value of R and cache the rule which has the maximum contribution value. The authors repeat above steps to cache wildcard rules until there is no available TCAM entry.

The shortcoming of the cover-set caching algorithm [19] is that it only considers the contribution value of an individual rule and cache the most one in each selection round. The algorithm does not consider the accumulated contribution value for a set of rules. In Section III, we will propose a wildcard rules caching algorithm to cache a set of rules in each selection round. We also propose a rule cache replacement algorithm considering temporal and spatial traffic localities. Traffic locality can be separated into temporal and spatial localities [25]. Temporal locality is that if one rule is matched by current traffic, the rule would be matched again soon after. Spatial locality is that the traffic would concentrate at some block of the field space during a short period.

III. Algorithms

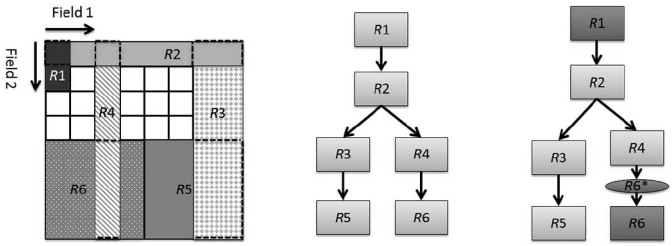
In this section, we present our wildcard rules caching algorithm and rule cache replacement algorithm considers the traffic localities.

A. System Model and Problem Formulation

We use the hardware-software hybrid switch design proposed in [19] as our switch prototype. Initially, we have a wildcard rules set for rules caching. An example is shown in Fig. 1. Here, we assume the priority order of the rules is $R1 > R2 > R3 > R4 > R5 > R6$. Each rule has two match fields, Field 1 and Field 2, which are 3 bits wide. In fact, Field 1 and Field 2 can be represented as the source IP address and destination IP address, respectively. The multi-dimensional space expressed by the match fields of rules is called the field space (FS). For example, Fig. 2(a) shows the wildcard rules in a two-dimensional FS expressed by the Field 1 and Field 2. A direct dependency exists between two rules if their FS overlap with each other. Fig. 2(b) shows the rule dependency graph of the rules in Fig. 2(a). For example, since the priority of $R1$ is higher than $R2$ and their wildcard rules are overlap, we add a directed edge from $R1$ to $R2$ to denote their dependency. In Fig. 2(b), if rules $R1$ and $R6$ are selected to be cached in the TCAM, a cover-set of the rule $R6$ (i.e. $R6^*$) will be cached in TCAM as shown in Fig. 2(c). The cover-set $R6^*$ is used to forward a packet to the software switch if the packet matches the intersection of rules $R6$ and $R4$. The content of cover set $R6^*$ in TCAM is shown in Fig. 2(d).

Rule	Match		Action
	Field 1	Field 2	
R1	000	00*	Forward to port 1
R2	***	000	Forward to port 2
R3	11*	***	Drop
R4	010	***	Forward to port 3
R5	1**	1**	Set VLAN ID = 1 and Forward to port 3
R6	0**	1**	Set Destination IP = 10.10.0.1 and Forward to port 4

Fig. 1. Example of a packet classification policy with six wildcard rules.



(a) Rules' FS in Fig. 1 (b) Rule dependency graph (c) Rules caching

Rule	Match		Action
	Field 1	Field 2	
R6*	010	***	*forward_to_SW_switch*

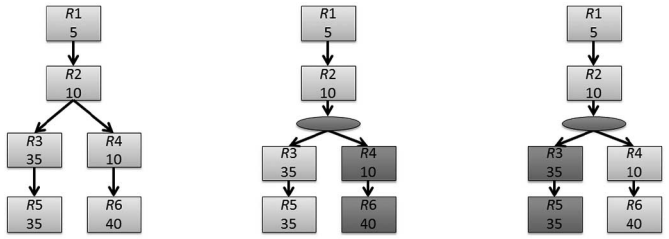
(d) The content of cover set $R6^*$ in TCAM

Fig. 2. Wildcard rules distribution in the FS and the dependency graph of wildcard rules in Fig. 1.

In wildcard rules caching system, for each rule R, there is a *weight* to represent the number of packets expected to match the rule $R(R.\Delta weight)$. Besides, there is also a *cost* to denote the number of required TCAM entries for caching the rule $R(R.\Delta cost)$. Given a set of wildcard rules and the TCAM size, wildcard rules caching algorithm would cache rules to maximize the total weight of the cached rules and do not overflow the TCAM size. However, the problem can be reduced to an all-neighbors knapsack problem which is an NP-hard problem [19], [26]. So, we propose a heuristic algorithm for the wildcard rules caching.

B. Wildcard Rules Caching Algorithm

Cover-set caching algorithm proposed in [19] calculates contribution value of each rule R. The contribution value of rule R is defined as the ratio of $R.\Delta weight$ to $R.\Delta cost$. The contribution value is a metric for caching rules. Cover-set caching algorithm would iteratively calculate contribution value of each un-cached rule and cache the rule which has the maximum contribution value until there is no available TCAM entry. For example, Fig. 3(a) denotes a dependency graph of wildcard rules in Fig. 1 and the weight of each rule. In general, the lower priority rule has larger rule weight. Assume the TCAM size is three. First, the cover-set caching algorithm would cache $R6$



(a) dependency graph (b) cover-set caching result (c) our caching result

Fig. 3. Example of wildcard rules caching (TCAM size is three).

and its cover-set since $R6$ has the maximum contribution value 40/2. Then, the remaining TCAM entry is one. There are only rules $R1$ and $R4$ can be further cached since both of $R1.\Delta cost$ and $R4.\Delta cost$ are one. For $R1.\Delta cost$, since $R1$ does not have a parent, we require only one TCAM entry for $R1$. For $R4.\Delta cost$, we require one TCAM entry for $R4$ and another one for cover-set of $R4$. However, the cover-set of $R6$ on edge $< R4,\ R6 >$ which is redundant and can be removed. Therefore, the contribution values of $R1$ and $R4$ are 5/1 and 10/1, respectively. The cover-set caching algorithm caches $R4$ and then there is no more TCAM entry for further rules caching. Fig. 3(b) shows the result of the cover-set caching algorithm. The cover-set caching algorithm only considers the contribution value of an individual rule.

However, in our wildcard rules caching algorithm, we consider the contribution value of a group of rules. For each rule R, we would layer-by-layer combine its neighboring rules and then calculate the accumulated contribution value from the rule R. For example, the individual contribution value of $R5$ is $35/2$. However, when $R5$ combines its neighbor $R3$, the accumulated contribution value is $(R5.\Delta weight + R3.\Delta weight)\ /\ (R5.\Delta cost + R3.\Delta cost) = (35 + 35)\ /\ (2 + 1) = 70/3$ which is larger than the contribution value of individual rule $R6$ (i.e. $40/2$). So, we can cache rules $R5$, $R3$ and cover-set of $R3$ at the same time. Fig. 3(c) shows the result of our wildcard rules caching algorithm which is better than the result of Fig. 3(b).

Our wildcard rules caching algorithm will calculate the accumulated contribution value for each un-cached rule. Then, we cache a set of rules which contribute the maximum value. We repeat the above steps until there is no remaining TCAM entry. We divide the explanation of our wildcard rules caching algorithm into two parts. First, we describe how to calculate accumulated contribution value for an un-cached rule in Algorithm 1. Then, our overall wildcard rules caching algorithm is described in Algorithm 2.

We use Fig. 4 to illustrate how to calculate accumulated contribution value for an un-cached rule. Assume the available TCAM size is 8. The accumulated contribution value (ACV) for an un-cached rule R is the contribution value of a set of rules ($rule\_set$) which are layer-by-layer collected upwards from the rule R to the root in a dependency graph. In other words, the accumulated contribution value of R is layer-by-layer accumulated until the contribution value does not increase. This procedure includes four steps. First, in each

Algorithm 1. Calculate_ACV(R, $Available\_TCAM$, k_1, k_2)

Input: R: an un-cached rule; $Avail\_TCAM$: available TCAM size;
k_1: a constant denotes the maximum candidates array size;
k_2: a constant denotes the maximum number of layers;
Output: the final ACV and $rule\_set$ of the rule R;

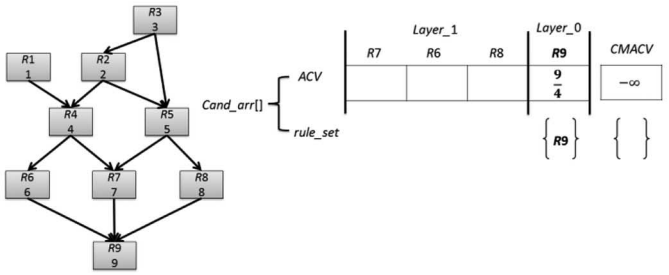
```
1.  if R.Δcost ≤ Avail_TCAM then
2.      Mark the valid bit of R(R.isValid) as true;
        Let CMACV be the contribution value of individual
        rule R;
3.      Heap sort the un-cached parents of R in decreasing
        order based on their contribution value;
        Choose the first k1 parents as the members of
        Cand_arr[R];
4.  else
5.      Mark the valid bit of R (R.isValid) as false and return;
6.  end else
7.  flag ← true; nLayer ← 1;
8.  while (flag is true and nLayer ≤ k2) do
9.      for each candidate c in Cand_arr[] do
10.         if we can combine the candidate c and do not
            overflow Avail_TCAM then
11.             Calculate the accumulated contribution value
                by combining c;
                Store the value in Cand_arr[c].ACV and
                update Cand_arr[c].rule_set
12.         end if
13.     end for
14.     Let h be a candidate whose ACV stored in
        Cand_arr[h].ACV is maximum in the candidate array;
15.     if Cand_arr[h].ACV > CMACV then
16.         Let Cand_arr[h].rule_set be the contributive
            candidates;
            Update CMACV ← Cand_arr[h].ACV;
            Update Cand_arr[R].rule_set ← Cand_arr[h].
            rule_set;
17.         Heap sort the un-cached parents of the contributive
            candidates;
            Choose the first k1 parents as new candidates for the
            next layer and nLayer++;
18.         else
19.             Set flag as false;
20.         end else
21.   end while
22.   Let CMACV be the final ACV of the rule R;
```

layer, we calculate accumulated contribution values by combing candidates iteratively if it does not overflow the available TCAM size (candidates are un-cached parent rules of lower layer). And we store the accumulated contribution values in a candidate array ($Cand\_arr[]$). For example, in Fig. 4(a), there is only one rule $R9$ in the bottom layer ($layer\_0$). Since $layer\_0$ is the bottom layer, we use the ratio of $R9.\Delta weight$ to $R9.\Delta cost$ as the initial accumulated contribution value i.e., $9/4$. Note that, the $R9.\Delta cost = 4$ because we need one TCAM entries for $R9$ and another three for cover-sets of $R9$. We store

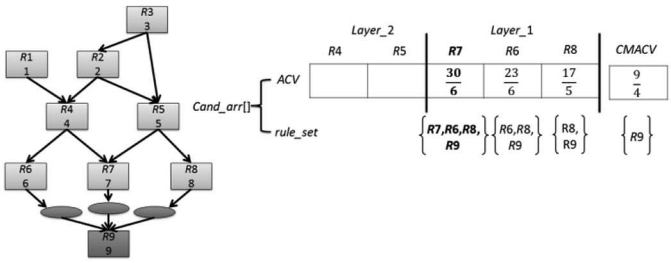
Algorithm 2. Wildcard_Rules_Caching_Algorithm(*Policy*, $Avail\_TCAM, k_3$)

Input: *Policy*: a set of wildcard rules; $Avail\_TCAM$: TCAM size;
k_3: at most k_3 un-cached rules are chosen to calculate their accumulated contribution value separately in each round.
Output: *Cached*: a set of cached rules

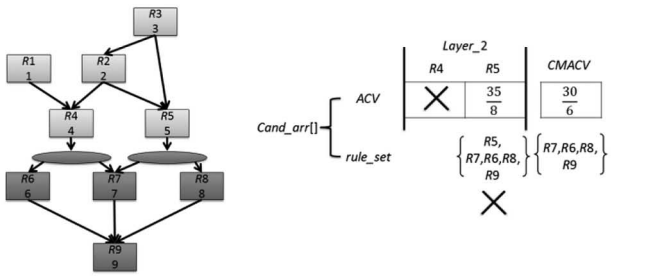
1. Initially, set affected region (*AR*) as empty set and let each rule's $rule\_set$ be empty set.
2. $flag \leftarrow true$;
3. **while** (*flag* is *true*) **do**
4. We sort all un-cached rules in decreasing order by their contribution values;
   Choose the first k_3 un-cached rules and store them in a rules array ($rules\_arr[]$)
5. **for** each rule *R* in $rules\_arr[]$ **do**
6. **if** $R.rule\_set = \phi$ **or** number of TCAM entries for caching $R.rule\_set > Avail\_TCAM$ **or** $R.rule\_set \cap AR \neq \phi$ **then**
7. Invoke $calculate\_ACV(R, Avail\_TCAM, k_1, k_2)$ and store the return value in *R*;
8. **end if**
9. **end for**
10. Among the rules which is in $rules\_arr[]$ and whose valid bit is true, we choose the rule which has the largest accumulate contribution value as *Rmax*;
11. **if** we can find the *Rmax* **then**
12. $AR \leftarrow Rmax.rule\_set \cup$ parents and children of rules in $Rmax.rule\_set$;
13. $Cached \leftarrow Cached + Rmax.rule\_set$;
14. $Avail\_TCAM \leftarrow Avail\_TCAM$ – number of TCAM entries for caching $Rmax.rule\_set$;
15. **else**
16. $flag \leftarrow false$;
17. **end else**
18. **end while**



(a) Calculation accumulated contribution value for the first layer (*Layer*_0)



(b) Calculation accumulated contribution value for the second layer (*Layer*_1)



(c) Calculation accumulated contribution value for the third layer (*Layer*_2)

Fig. 4. Layer-by-layer calculating the accumulated contribution value of the rule *R*9. Available TCAM size is 8.

this value in $Cand\_arr[R9].ACV$ and insert the rule *R*9 in the $Cand\_arr[R9].rule\_set$.

Second, we choose the maximum *ACV* in the candidate array and whose number of required TCAM entries is smaller than the available TCAM size as the current maximum accumulated contribution value (*CMACV*). The candidates which have a contribution in $CMACV$ are contributive candidates. Initially, the value of CMACV is $-\infty$. In Fig. 4(a), the *CMACV* of $layer\_0$ is 9/4 and *R*9 is the contributive candidate. Third, un-cached parents of the contributive candidates are new candidates for the next layer (upper layer). In Fig. 4(a), un-cached parents of *R*9 are rules *R*6, *R*7 and *R*8. For $R6.\Delta cost$, we need one TCAM entry for *R*6 and another one for the cover-set of *R*6, but the cover-set of *R*9 on edge $< R6,\ R9 >$ which becomes redundant and can be removed. Thus, $R6.\Delta cost = 1$. Since $R6.\Delta weight = 6$, the contribution value of *R*6 is 6/1. Similarly, the contribution values of *R*7 and *R*8 are 7/2 and 8/1, respectively. We sort these new candidates in decreasing order according to their contribution values. Therefore, the order of new candidates are *R*8, *R*6, and *R*7. Fourth, we combine the new candidates in order and accumulate their contribution values if the TCAM size is enough to store the candidate rules. We continue above steps for the next layer (upper layer) until the *CMACV* does not increase.

Fig. 4(b) demonstrates the procedure of calculating accumulated contribution value of *R*9 for its second layer ($Layer\_1$). The accumulated contribution value by combining candidate *R*8 is the ratio of $(9+8)$ to $(4+1) = 17/5$ since $R8.\Delta weight$ and $R8.\Delta cost$ are 8 and 1, respectively. Then, we calculate the accumulated contribution value by further combining candidate *R*6. The accumulated contribution value is the ratio of $(17+6)$ to $(5+1) = 23/6$ because $R6.\Delta weight$ is 6 and $R6.\Delta cost$ is 1. Then, we combine candidate *R*7 and get the accumulated contribution value is the ratio of $(23+7)$ to $(6+0) = 30/6$. Note that, since the cover-sets of *R*6 and *R*8 are shared by *R*7, we only require one TCAM entry for rule *R*7. Besides, the cover-set on edge $< R7,\ R9 >$ can be removed. Thus, there is no entry cost to combine *R*7. After processing all candidates in $Layer\_1$, the *CMACV* becomes 30/6. And, rules *R*7, *R*6, and *R*8 are the new contributive candidates. Then the un-cached parents of these contributive rules are new candidates for the next layer (*R*4 and *R*5). Since the contribution values of *R*4 and *R*5 are 4/2 and 5/2, respectively, the order of next layer candidates are *R*5 and *R*4.

For the third layer ($layer\_2$), as shown in Fig. 4(c), we get that the accumulated contribution value by combining $R5$ is $35/8$. When we further combine candidate $R4$, it overflows the available TCAM $size = 8$. We cannot combine the candidate $R4$. However, the accumulated contribution value by combining $R5$ is smaller than the *CMACV*. We cannot get a larger accumulated contribution value in this layer. Our algorithm of calculating the accumulated contribution value of $R9$ would terminate at $layer\_2$. Therefore, for the un-cached rule $R9$, when it combines with its neighboring rules $R6$, $R7$, and $R8$, we get an accumulated contribution $value = 30/6$ which is larger than the contribution value of individual rule $R9 = 9/4$.

Algorithm 1 shows the overall algorithm which calculates the accumulated contribution value of an un-cached rule R and the corresponding rule set. $Avail\_TCAM$ denotes the number of available TCAM entries for rules caching. However, the running time for calculating the accumulated contribution value of an un-cached rule could be high if we traverse an entire dependency graph. We use constant k_1 to limit the maximum number of candidates in each layer and constant k_2 to limit the maximum number of layers we traverse. If the number of available TCAM entries is enough for caching an individual rule R, the contribution value of R is the *CMACV* in the first layer. Then, the next layer's candidates are in the decreasing order of their contribution values (Lines 1-3). Otherwise, mark the valid bit of R ($R.isValid$) as false and return (Lines 4-6). For each layer, we calculate *ACV*s by combining candidates iteratively if it does not overflow the available TCAM size (Lines 9-13). After processing all candidates, let h be a candidate whose *ACV* stored in $Cand\_arr[h].ACV$ is maximum in the candidate array (Line 14). If the candidate h exists and its *ACV* is larger than the *CMACV*, algorithm 1 would update the *CMACV* and its corresponding rule set. The candidates which have contribution in $Cand\_arr[h].ACV$ become the contributive candidates. The un-cached parents of the contributive candidates are new candidates for the next layer (Lines 15-17). Otherwise, the algorithm breaks the while loop and lets *CMACV* be the *accumulated contribution value* of R (Lines 19-22).

Assume there are N rules in a packet classification policy. To calculate an individual rule's contribution value (or $\Delta cost$), we need to check the rule's one-hop neighboring rules whether are cached or not. There are $O(N)$ one-hop neighbors for a rule. The time complexity of calculating an individual rule's contribution value (or $\Delta cost$) is $O(N)$. In Algorithm 1, in a layer, every time we combine one candidate and calculate an *ACV*. In other words, we calculate *ACV*s by adding candidates' contribution value iteratively. There are at most k_1 candidates in a layer. The time complexity of combining candidates in a layer is $O(k_1 * N)$. Then, we calculate $O(N)$ un-cached parents' contribution values and do heap sort for them in decreasing order. And we choose the first k_1 parents as new candidates for the next layer. The time complexity of finding new candidates for the next layer is $O(N * N + N \log N + k_1)$. Therefore, the total time complexity of each layer is $O(k_1 * N + N * N + N \log N + k_1) = O(N^2)$. There are at most k_2 layers in Algorithm 1. The time complexity of Algorithm 1 is $O(k_2 * N^2) = O(N^2)$.

TABLE I
THE ACCUMULATED CONTRIBUTION VALUE OF EACH UNCACHED RULE IN FIG. 4 ACROSS DIFFERENT ROUNDS OF OUR WILDCARD RULES CACHING ALGORITHM

(a) First round

Rule	Accumulated $\Delta weight$	Accumulated $\Delta cost$	*ACV*	*rule_set*
$R1$	1	1	1	$\{R1\}$
$R2$	5	2	2.5	$\{R3, R2\}$
$R3$	3	1	3	$\{R3\}$
$R4$	10	4	2.5	$\{R3, R1, R2, R4\}$
$R5$	10	3	3.33	$\{R2, R3, R5\}$
$R6$	6	2	3	$\{R6\}$
$R7$	22	6	3.67	$\{R1, R2, R3, R4, R5, R7\}$
$R8$	8	2	4	$\{R8\}$
$R9$	30	6	5	$\{R6, R7, R8, R9\}$

(b) Second round

Rule	Accumulated $\Delta weight$	Accumulated $\Delta cost$	*ACV*	*rule_set*
$R1$	1	1	1	$\{R1\}$
$R2$	5	2	2.5	$\{R3, R2\}$
$R3$	3	1	3	$\{R3\}$
$R4$	5	2	2.5	$\{R1, R4\}$
$R5$	10	2	5	$\{R2, R3, R5\}$

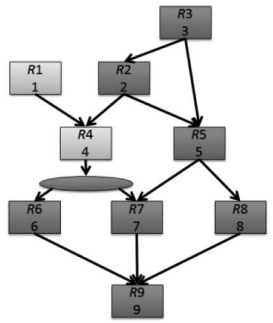


Fig. 5. The result of our wildcard rules caching algorithm in the example Fig. 4.

Our wildcard rules caching algorithm calculates accumulated contribution value of each un-cached rule by Algorithm 1. Then, we cache a set of rules which has the maximum accumulated contribution value. The above steps are called a *round* in our wildcard rules caching algorithm. We repeat rounds until there is no remaining TCAM entry. For example, in Fig. 4, we not only layer-by-layer calculate the accumulated contribution value of $R9$, but also layer-by-layer calculate the other un-cached rules' accumulated contribution values separately with Algorithm 1. Table I(a) lists the nine un-cached rules in Fig. 4 along with their *ACV* and $rule\_set$. Since $R9$ has the maximum accumulated contribution value, we cache rules $R6$, $R7$, $R8$ and $R9$ in the first round of our wildcard rules caching algorithm. In the next round, the accumulated contribution value of each un-cached rule is recalculated and shown in Table I(b). We cache rules $R2$, $R3$ and $R5$ to TCAM and then there is no remaining TCAM entry for further rules caching. The result of our wildcard rules caching algorithm is shown in Fig. 5.

The second part of our wildcard rules caching algorithm is presented in Algorithm 2. The *Policy* is a set of wildcard rules for rules caching and $Avail\_TCAM$ is the TCAM size. If we calculate the accumulated contribution values of all un-cached rules in each round, the running time of our algorithm could be high. Thus, we choose at most k_3 un-cached rules and calculate their accumulated contribution values in each round. The term *Cached* denotes the final set of cached rules. We use affected region (AR) to represent a set of rules which are dirty. A rule d which is cached in the previous

round or whose contribution value ($d.\Delta cost$) would change is the dirty rule. In other words, if an un-cached rule whose $rule\_set$ intersects with *AR*, we need to recalculate its accumulated contribution value. In each round of Algorithm 2, first, we sort all un-cached rules in decreasing order according to their contribution value and choose the first k_3 un-cached rules (Lines 3-4). Second, for the k_3 un-cached rules, we invoke Algorithm 1 to calculate their accumulated contribution values separately (Lines 5-9). Then, among the rules which are the first k_3 un-cached rules and whose valid bit is true, we choose the rule which has the maximum accumulated contribution value as *Rmax* (Line 10). If more than one rule has the same and maximum accumulated contribution value, we choose the rule which has the least $\Delta cost$ as *Rmax*. If we cannot find the *Rmax* rule, Algorithm 2 would stop caching rules (Lines 15-18). Otherwise, we cache rules in $Rmax.rule\_set$ to TCAM and update residual $Avail\_TCAM$ and *AR* (Lines 11-14). Because we cache the rules in $Rmax.rule\_set$ to TCAM, only parents and children of rules in $Rmax.rule\_set$ can change their contribution value ($\Delta cost$). We include them in *AR* (Line 12). Therefore, in the next round, not all the k_3 un-cached rules need to invoke the Algorithm 1. Only those un-cached rules whose $rule\_set$ intersects with *AR* need to invoke Algorithm 1 for calculating a new accumulated contribution value. In other words, *AR* technique can help us to speed up our algorithm.

In Algorithm 2, in the beginning of each round, we need to calculate $O(N)$ un-cached rules' contribution value and sort them in the decreasing order. We store the first k_3 un-cached rules in a rules array. The time complexity of choosing the first k_3 un-cached rules is $O(N*N + N\log N + k_3)$. Then, we invoke Algorithm 1 at most k_3 times for calculating their accumulated contribution values separately. So, the time complexity of each round is $O\left(N*N + N\log N + k_3 + k_3*N^2\right) = O\left(N^2\right)$. If the total TCAM size is T, there are at most T rounds. The time complexity of our wildcard rules caching algorithm is $O\left(N^2T\right)$. In the cover-set caching algorithm [19], in each round, they calculate $O(N)$ un-cached rules' contribution value and cache the rule with the maximum contribution value. There are at most $O(T)$ rounds. The time complexity of the cover-set caching algorithm is $O\left(N^2T\right)$ which is same as our wildcard rules caching algorithm.

C. Rule Cache Replacement (RCR) Algorithm

After caching wildcard rules according to their weights, the input packets can match rules either in the TCAM (cache) or software switch. Once cache miss occurs, RCR algorithm would replace victim cached rules with the cache miss rule. If the cache hit ratio is high, we can avoid miss penalties. So, the hit ratio is one of the most important metrics for measuring the speed of processing incoming packets of a switch. The goal of the RCR algorithm is to increase the cache hit ratio. We assign a counter for each rule to represent its temporal traffic locality. Each rule *R* contains a 2-bit saturating counter *value* (*R.value*). Once cache miss occurs, the RCR algorithm replaces the victim cached rule which has the lowest *value* with the cache miss rule. Then, set the cache miss rule's *value* according to its neighboring rules' *values* (considering spatial traffic locality).

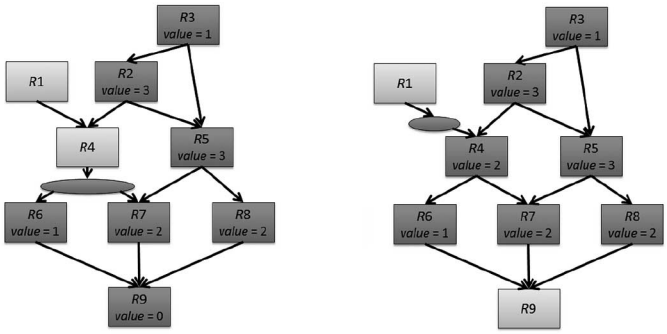
Algorithm 3. RCR Algorithm(*Packets*, *Cached*)

Input: *Packets*: a set of packets; *Cached*: a set of cached rules;
1. **for** each packet *p* in *Packets* **do**
2. **if** the rule *R* matched by *p* is in the TCAM (*Cached*) **do**
3. $R.value \leftarrow R.value + 1$;
4. **else**
5. Among all cached rules and whose *R.value* is 0, find a victim rule *V* which can release the largest TCAM entries. If we cannot find *V*, $R.value \leftarrow R.value - 1$ for all cached rules and repeat Line 5.
6. Repeat the step in Line 5 until the number of released TCAM entries is enough for the number of consumed TCAM entries for caching *R*.
7. Calculate *locality*(*R*) according to equation (1). $R.value \leftarrow$ rounding of *locality*(*R*) and then cache *R* to the TCAM;
8. **end else**
9. **end for**
10. **return**

We present the RCR algorithm in Algorithm 3. The *Packets* denote the input packet. We use Algorithm 2 to cache a set of rules (*Cached*) to TCAM and set their *values* as zero before feeding *Packets*. Once an input packet matches on the rule which has been in the TCAM already, the rule's *value* is increased by one due to the cache hit (Lines 1-4). So, frequently matched rules have a higher *value* than infrequent ones. If the rule matched by the input packet is not in the TCAM, we call the rule as the cache miss rule. We would find victim rules and replace victim rules with the cached miss rule (Lines 5-6).

However, the number of released TCAM entries of a victim rule may less than the number of consumed TCAM entries for caching the cache miss rule. So, in Line 6, we would repeat the step in Line 5 to find the other victim rules for further releasing TCAM space. Then, we set the cache miss rule's *value* and cache it to the TCAM (Line 7). We use *locality* computed by equation (1), shown at the bottom of the page, to represent the cache miss rule's spatial traffic locality. In equation (1), if the cache miss rule *R* has neighboring rules, its *locality* is computed as the ratio of its cached neighboring rules' total value to the number of its neighboring rules. Otherwise, its *locality* is one. Then, we set the cache miss rule's *value* as the rounding of equation (1). In other words, a cache miss rule with high *locality* means that the current traffic locality may be close to the cache miss rule's field space. So, we assign a high *value* to the cache miss rule to keep it in the TCAM.

$$locality(R) = \begin{cases} \dfrac{\sum_{i=0}^{\text{all } R'\text{s cached neighbors}} R_i.value}{\text{number of } R'\text{s neighbors}}, & \text{if number of } R'\text{ s neighbors } != 0 \\ 1, & \text{else} \end{cases} \quad (1)$$



(a) Before cache replacement (b) After cache replacement

Fig. 6. Example of cache replacement algorithm.

For example, Fig. 6 illustrates our RCR algorithm when an input packet matches on the cache miss rule $R4$. Assume the TCAM size is 8. Therefore, we must find victim cached rules and replace them with the cache miss rule. In Fig. 6(a), we would find rule $R9$ as the victim rule since its *value* is zero. The TCAM space released by $R9$ is enough for caching the rule $R4$ and its cover-set. So, we replace the victim rule with the cache miss rule and set the cache miss rule's *value*. The rule $R4$'s cached neighbors are rules $R2$, $R6$, and $R7$. And, the rule $R4$'s neighbors are rules $R1$, $R2$, $R6$, and $R7$. So, *locality*($R4$) is the ratio of $(3 + 1 + 2)$ to $(4) = 1.5$ according to equation (1). Then, the cache miss rule $R4$'s *value* is set as the rounding of $1.5 = 2$. We show the result of the RCR algorithm in Fig. 6(b).

IV. Simulations

In this section, we present the simulations of our proposed algorithms. First, we compare the experimental results of our wildcard rules caching algorithm with the cover-set caching algorithm in [19]. We evaluate the performance of each caching algorithm in the cache hit ratio. Second, we compare our RCR algorithm with LRU and RR algorithms which are typical cache replacement algorithms in memory management. We also compare the RCR algorithm with ARC algorithm which maintains two LRU lists that keep tracking of recently used elements and frequently used elements. Here, the cache hit ratio is also used as the performance metric for the cache replacement algorithms.

A. Simulation Environment

Since we cannot get a real data center traffic, we use ClassBench [24] to generate synthetic rule policy with a different type of packet classification and implement it in C language. ClassBench uses the real data center database to generate synthetic rule policy with the desired rule number and dependency. To simulate a near real data center environment, ClassBench is a good choice in simulations. In this simulation, we take the standard policy Access Control List, IP Chain and Firewall from the seed files provided by ClassBench.

In our simulations, we use ClassBench's filter set generator to generate six policies with different parameter files. Each policy contains 10K rules that match on five space fields (source/destination IP address, source/destination port and protocol number). Three of them are synthetic Access Control Lists (ACLs) generated by $acl2\_seed$ and $acl3\_seed$ parameter files in ClassBench. The other three policies are synthetic Fire Walls (FWs) generated by $fw1\_seed$, $fw2\_seed$ and $fw3\_seed$ parameter files. Table II lists the six policies along with their number of edges and maximum depth in the dependency graph.

TABLE II
The Policies Used in Our Simulation With Different Number of Edges and Maximum Depths

Policy	Number of edges	Maximum depth
ACL1	15512	34
ACL2	18484	27
ACL3	27887	38
FW1	59067	52
FW2	7958	8
FW3	63022	52

We generate input packets according to the wildcard rules' field space sizes to analyze the cache hit ratios of our wildcard rules caching algorithm and the cover-set caching algorithm. The number of packets matching a rule is proportional to each rule's field space size. And, we let each rule's traffic volume is in a range (e.g. [1, 1,000]). That is, a rule with larger field space size has a higher probability to be matched by more packets than a rule with smaller field space size. In our simulations of wildcard rules caching algorithms, the maximum TCAM capacity is 3K entries. And, the constants k_1, k_2, and k_3 for our wildcard rules caching algorithm are 60, 5 and 100, respectively. On the other hand, in the simulations of cache replacement algorithms, the TCAM capacity is 2K entries. We first use our wildcard rules caching algorithm to cache rules to TCAM. Then, we evaluate the RCR algorithm with LRU, RR, and ARC algorithms by feeding input packets with different traffic localities.

In the hardware-software hybrid switch system, the cover-set algorithm is used before the traffic flow into the switches. The cover-set algorithm fills the hardware TCAM (cache) with the important wildcard rules which have higher weight values. Thus, both of our algorithms 1 and 2 are invoked to cache important wildcard rules to the TCAM before the traffic pass through the switches. For 10,000 rules, algorithms 1 and 2 take less than 0.15 and 0.16 seconds to select 2,000 important rules to a TCAM, respectively. When the traffic flow into the switches, there may exist rule cache miss. Once the rule cache miss occurs, our proposed cache replacement algorithm would replace hardware rules with the software rules. Note that, the rules exchange only occurs in the switches and the replacement time is small.

B. Simulation Results

We first generate each rule's traffic volume according to its field space size to analyze the cache hit ratios of our wildcard rules caching algorithm and the cover-set caching algorithm. Let H_a and H_b denote the average cache hit ratio

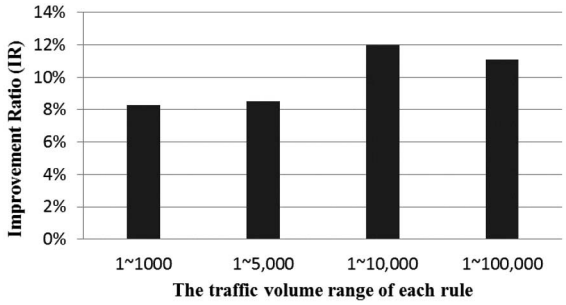


Fig. 7. Improvement ratio IR for various traffic volume ranges.

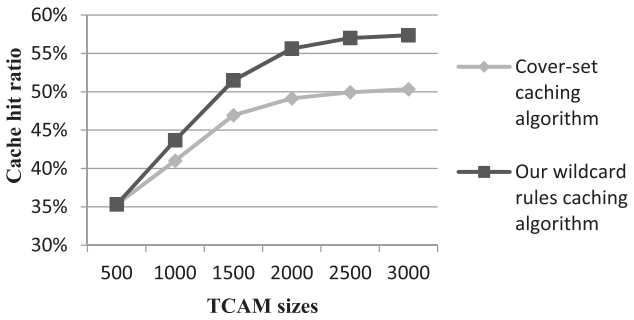


Fig. 8. Average cache hit ratio of six policies for different TCAM sizes.

of six policies performed by our caching algorithm and cover-set caching algorithm, respectively. Let IR $= (H_a - H_b) / (H_b)$ be improvement ratio of our algorithm to the cover-set algorithm. Fig. 7 shows the improvement ratio IR for various traffic volume ranges. We found that our caching algorithm always performs better than the cover-set caching algorithm for various traffic volume ranges. The average improvement ratio is 10%. The reason is that for each rule R, we calculate its accumulated contribution value by combining its neighboring rules rather than the contribution value of individual R. While comparing with the cover-set caching algorithm, we take more rules into consideration and reduce the cover-set overhead.

Fig. 8 shows the impact of different TCAM sizes to wildcard rules caching algorithms where each rule's traffic volume is in the range [1, 10,000]. We average the cache hit ratios of six policies and show in Fig. 8. We observe that when the number of TCAM entries is less than 1K, there is no difference in the performances of our proposed wildcard rules caching algorithm and the cover-set caching algorithm. This is because both algorithms can cache default rules first due to these default rules having much larger weight than general rules. When TCAM size is larger than 1K, our caching algorithm performs much better than the cover-set caching algorithm. Our wildcard rules caching algorithm is suitable for rules caching when the TCAM size is larger than 1,000.

In our wildcard rules caching algorithm, we use three constants: k_1, k_2 and k_3 to reduce the time complexity. We first show the impact of k_3 to the performance of our wildcard rules caching algorithm in Fig. 9 where each rule's traffic volume is in the range [1, 10,000]. In each caching round, at most k_3 un-cached rules are selected to calculate their ACVs. It is obvious that when k_3 is larger, our caching algorithm consumes more time complexity to get higher IR. The performance with

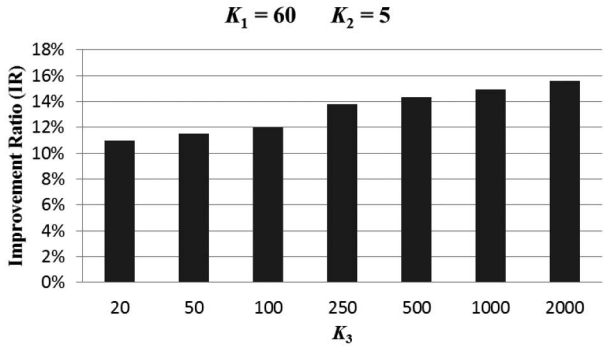


Fig. 9. The impact of k_3 to our wildcard rules caching algorithm.

$k_3 = 100$ is only 3% lower than the performance with $k_3 =$ 2,000. The constants k_1 and k_2 limit the Algorithm one traversing range in a dependency graph. We also evaluate our caching algorithm for different k_1 and k_2. The impact of k_1 and k_2 to our caching algorithm is similar to k_3's. If we set $k_1 = 60$ and $k_3 = 100$, the IR would increase until k_2 is larger than 5. If we set $k_2 = 5$ and $k_3 = 100$, the IR would increase until k_1 is larger than 60.

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. So, 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 to limit the number of extra generated spatial locality packets. First, we utilize the ClassBench trace generator to generate 10K input packets for a policy. Then, for each input packet p, we choose a random number r which is computed as random() modulo SL. If the input packet p matches on the rule R, we generate the r extra packets which randomly match R or the neighboring rules of R. And, we insert these r extra packets right behind the sequence of the input packet p. Finally, we reform the sequence of p and its extra packets by randomly arranging packets sequence for every 4,000 packets. Since the performances of cache replacement algorithms in our six policies are similar, we take ACL3 as an example for the following evaluations of cache replacement algorithms. We feed our new input packets to ACL3 and show the cache hit ratios of RCR algorithm with LRU, RR, and ARC schemes in Fig. 10. RCR algorithm has higher cache hit ration than LRU, RR, and ARC schemes. Since we randomly arrange packets sequence for every 4,000 packets which is larger than the TCAM $size = 2,000$, the time interval of repetitive packets may be larger than the TCAM size. LRU cannot keep temporal traffic locality in the TCAM well. Thus, the cache hit ratio of LRU is similar to RR. On the contrary, our algorithm and ARC can keep track of temporal traffic locality, so they have better performance than LRU.

Fig. 11 shows cache hit ratios of cache replacement algorithms for different TCAM sizes with SL $= 20$. When the TCAM size is small, RCR algorithm performs much better than LRU, RR, and ARC schemes because RCR algorithm considers not only temporal but also spatial traffic locality. The reason

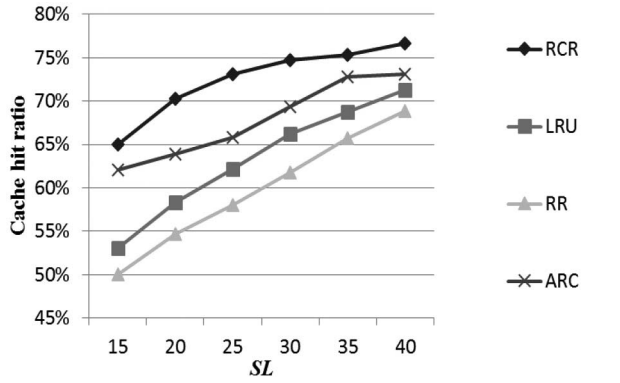


Fig. 10. Cache hit ratios of different cache replacement algorithms.

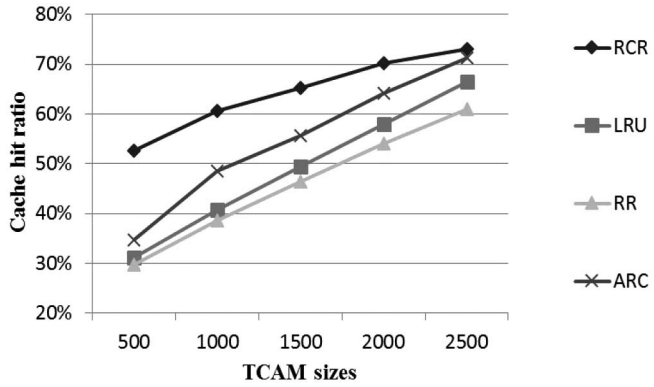


Fig. 11. Cache hit ratios of cache replacement algorithms for different TCAM sizes.

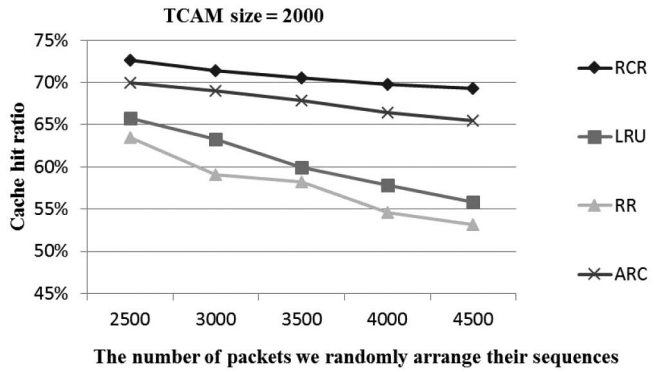


Fig. 12. Cache hit ratios of cache replacement algorithms.

is that in the previous extra spatial packets generation, we randomly arrange the sequence of input packets for every 4,000 packets. The time interval of repetitive packets may be larger than the TCAM size, so LRU cannot get high cache hit ratio according to temporal traffic locality. ARC maintains an LRU list containing frequently used elements, therefore it has higher cache hit ratio than LRU. On the other hand, when the TCAM size is large, the time interval of repetitive packets can be less than the TCAM size. RCR algorithm and ARC have similar cache hit ratios.

In Fig. 12, we randomly arrange packets sequence for various sizes of random arrangement and measure the cache hit ratios. Since ARC can keep track of recently used rules and frequently used rules, its performance is close to RCR algorithm. When the size of random arrangement is small, the input packets have high temporal traffic locality. The time interval of repetitive packets would be small. The cache hit ratio of LRU with small time interval of repetitive packets is better than the large time interval of repetitive packets.

V. Conclusion

In wildcard rules caching, the cover-set is an efficient skill to solve rule dependency problem. Compared with the cover-set caching algorithm, we calculate the accumulated contribution value of a set of rules instead of the individual contribution value of a rule. Therefore, the performance of our rules caching algorithm is better than the cover-set caching one. Besides, we propose an RCR algorithm to consider the traffic temporal and spatial localities. If the cache miss occurs and the current traffic locality is close to the missed rule, we replace a victim rule with the cache miss rule and set a high value to the missed rule to keep the rule in the TCAM. Otherwise, the value of cache miss rule is set to low and it can be replaced soon after. Simulation results show that the RCR has higher cache hit ratio than LRU, RR, and ARC schemes. Our future work includes: (1) investigate other potential wildcard rules caching algorithm to improve the cache hit ratio and (2) refine our cache replacement algorithm to calculate the weight value of the cached rules which is more conform to the traffic locality.

References

[1] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proc. ACM SIGCOMM*, Helsinki, Finland, Aug. 2012, pp. 323–334.
[2] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *Proc. 2nd ACM SIGCOMM Workshop Hot Top. Softw. Def. Netw.*, Hong Kong, Aug. 2013, pp. 49–54.
[3] X. Jin *et al.*, "Dynamic scheduling of network updates," in *Proc. ACM SIGCOMM*, Chicago, IL, USA, Aug. 2014, pp. 539–550.
[4] N. Gude *et al.*, "NOX: Towards an operating system for networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, Jul. 2008.
[5] Floodlight [Online]. Available: http://www.projectfloodlight.org/floodlight/
[6] RYU [Online]. Available: http://osrg.github.io/ryu/
[7] N. McKeown *et al.*, "Openflow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Apr. 2008.
[8] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for advanced packet classification with ternary CAMs," in *Pro. ACM SIGCOMM*, Philadelphia, PA, USA, Aug. 2005, pp. 193–204.
[9] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "SAX-PAC (scalable and expressive packet classification)," in *Proc. ACM SIGCOMM*, Chicago, IL, USA, Aug. 2014, pp. 15–26.
[10] M. Casado *et al.*, "Rethinking enterprise network control," *IEEE/ACM Trans. Netw.*, vol. 17, no. 4, pp. 1270–1283, Aug. 2009.
[11] C. R. Meiners, A. X. Liu, and E. Torng, "Bit weaving: A non-prefix approach to compressing packet classifiers in TCAMs," *IEEE/ACM Trans. Netw.*, vol. 20, no. 2, pp. 488–500, Apr. 2012.
[12] Y.-C. Cheng and P.-C. Wang, "Packet classification using dynamically generated decision trees," *IEEE Trans. Comput.*, vol. 64, no. 2, pp. 582–586, Feb. 2015.
[13] A. X. Liu, C. R. Meiners, and E. Torng, "TCAM razor: A systematic approach towards minimizing packet classifiers in TCAMs," *IEEE/ACM Trans. Netw.*, vol. 18, no. 2, pp. 490–500, Apr. 2010.
[14] X.-N. Nguyen, D. Saucez, C. Barakat, and T. Turletti, "Optimizing rules placement in openflow networks: Trading routing for better efficiency," in *Proc. 3rd Workshop Hot Top. Softw. Def. Netw.*, Chicago, IL, USA, Aug. 2014, pp. 127–132.

[15] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing tables in software-defined networks," in *Proc. INFOCOM*, Turin, Italy, Apr. 2013, pp. 545–549.
[16] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the "one big switch" abstraction in software-defined networks," in *Proc. 9th ACM Conf. Emerg. Netw. Exp. Technol.*, Santa Barbara, CA, USA, Dec. 2013, pp. 13–24.
[17] B. Yan, Y. Xu, H. Xing, K. Xi, and H. J. Chao, "CAB: A reactive wildcard rule caching system for software-defined networks," in *Proc. 3rd Workshop Hot Top. Softw. Def. Netw.*, Chicago, IL, USA, Aug. 2014, pp. 163–168.
[18] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," in *Proc. ACM SIGCOMM*, New Delhi, India, Aug. 2010, pp. 351–362.
[19] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. (2014). *Rule-Caching Algorithms for Software-Defined Networks* [Online]. Available: https://www.cs.princeton.edu/~jrex/papers/cacheflow-long14.pdf
[20] Y. Liu, V. Lehman, and L. Wang, "Efficient FIB caching using minimal non-overlapping prefixes," *Comput. Netw.*, vol. 83, no. 4, pp. 85–99, Jun. 2015.
[21] N. Shelly, E. J. Jackson, T. Koponen, N. McKeown, and J. Rajahalme, "Flow caching for high entropy packet fields," in *Proc. 3rd Workshop Hot Top. Softw. Def. Netw.*, 2014, pp. 151–156.
[22] N. Megiddo and D. S. Modha, "Outperforming LRU with an adaptive replacement cache algorithm caching," *IEEE Comput.*, vol. 37, no. 4, pp. 58–65, Apr. 2004.
[23] H. Huang, S. Guo, P. Li, W. Liang, and A. Zomaya, "Cost minimization for rule caching in software defined networking," *IEEE Trans. Parallel Distrib. Syst.*, May 11, 2015, doi: 10.1109/TPDS.2015.2431684.
[24] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark," *IEEE/ACM Trans. Netw.*, vol. 15, no. 3, pp. 499–511, Jun. 2007.
[25] M. Kharbutli and R. Sheikh, "LACS: A locality-aware cost-sensitive cache replacement algorithm," *IEEE Trans. Comput.*, vol. 63, no. 8, pp. 1975–1987, Aug. 2014.
[26] G. Borradaile, B. Heeringa, and G. Wilfong, "The knapsack problem with neighbour constraints," *J. Discr. Algorithms*, vol. 16, pp. 224–235, Oct. 2012.

Jang-Ping Sheu (S'85–M'86–SM'98–F'09) received the B.S. degree in computer science from Tamkang University, Taiwan, in 1981, and the M.S. and Ph.D. degrees in computer science from National Tsing Hua University, Hsinchu, Taiwan, in 1983 and 1987, respectively.

He is currently a Chair Professor of the Department of Computer Science and the Associate Dean of the College of Electrical and Computer Science, National Tsing Hua University. He was a Chair of Department of Computer Science and Information Engineering, National Central University, Taiwan, from 1997 to 1999. He was the Director of Computer Center, National Central University, from 2003 to 2006. He was the Director of Computer and Communication Research Center, National Tsing Hua University from 2009 to 2015. His research interests include wireless communications, mobile computing, and software-defined networks. He is a Member of Phi Tau Phi Society. He was an Associate Editor of the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS and *International Journal of Sensor Networks*. He is an Associate Editor of the *International Journal of Ad Hoc and Ubiquitous Computing*.

Dr. Sheu was the recipient of the Distinguished Research Awards of the National Science Council of the Republic of China in 1993–1994, 1995–1996, and 1997–1998. He was also the recipient of the Distinguished Engineering Professor Award of the Chinese Institute of Engineers in 2003, the K.-T. Li Research Breakthrough Award of the Institute of Information and Computing Machinery in 2007, the Y. Z. Hsu Scientific Chair Professor Award, and Pan Wen Yuan Outstanding Research Award in 2009 and 2014, respectively.

Yen-Cheng Chuo received the master's degree in computer science from National Tsing-Hua University, Hsinchu, Taiwan, in 2015. His research interests include software-defined networks and wireless networks.