Approximation Algorithms for Dependency-Aware Rule-Caching in Software-Defined Networks

Jie Wu, Yang Chen, and Huanyang Zheng
Department of Computer and Information Sciences, Temple University

***Abstract*—Software-defined networks (SDNs) can support fine-grained forwarding policies in the underlying switches. The new content addressable memory, Ternary Content Addressable Memory (TCAM), enables fast lookups for matching rules in message forwarding, represented as binary strings with wildcards. However, the cost and power limit the number of matching rules a TCAM can support. Therefore, rule caching is needed to place high-weight (high-hit) rules in the TCAM hardware, while large, but slow, software switches handle cache-miss traffic. We assume that matching these rules form a forest of trees. A rule R' is a descendant of another rule R if R' is a special case of R. Dependent rules are evaluated in a particular matching order: when a rule is included in the cache, all its descendants in the rule set have to be included as well. Our objective is to maximize the number of rule hits, while limiting the number of cached rules. Three greedy rule-caching algorithms are proposed, including two with approximation ratios of 2 and $\frac{24}{5}$, respectively. In addition, we propose a dynamic programming solution that is optimal but slow. The efficiency of the proposed approaches are evaluated through real data-driven simulations.**

***Index Terms*—Approximation ratio, caching, dependency constraint, greedy algorithms, software-defined networks.**

I. Introduction

In software-defined networks (SDNs), the central controller installs packet-processing rules in switches to manage traffic [1]. There are two types of switches: hardware-based content address memory switches, like Ternary Content Addressable Memory (TCAM) [2], which are fast but small in capacity [3], and software-based switches that are large in capacity but are relatively slow. These two types of switches form a classic cache and main memory pair that stores and processes rules. In this paper, we study an efficient caching scheme for rules subject to a unique constraint on rule placement.

Caching IP address is well known in a general setting [4] as well as in TCAM [5]. The state-of-the-art caching in TCAM uses *descendant matching* rules [6]. Rules in SDNs are binary string matching rules with wildcards, using * to match any single character. A rule processing order corresponds to a matching between a message header and the binary string of a rule [7]. A string, R', is a descendant of another string, R, if R' is a special case of R. R' and R are dependent from each other. A rule together with all its descendants form a branch of a tree. Two rules are independent if they do not have a common ancestor in a tree. In Table 1, rule R_6, as the root, forms a tree consisting of R_2, R_3, R_4, and R_5. Rule R_1 is independent. Together, the rule set forms a forest of trees [8, 9]. For any two dependent rules, there is a matching order. The descendant rule is evaluated first. This order is used to minimize the number of rules used in the matching process.

[1]This research was supported in part by NSF grants CNS 1757533, CNS 1629746, CNS 1564128, CNS 1449860, CNS 1461932, CNS 1460971, and IIP 1439672.

TABLE I: Rules with binary strings and weights.

Rule	Code	Weight
R_1	0***	5
R_2	1000	7
R_3	100*	13
R_4	10**	43
R_5	1110	20
R_6	1***	12

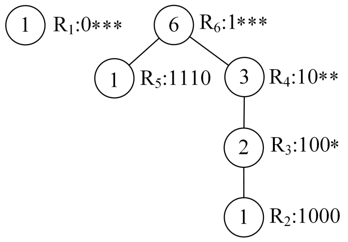
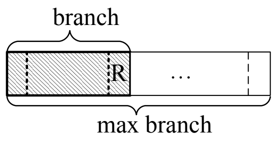


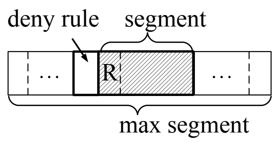
Fig. 1: A forest of rules.

The matching order complicates the caching policy because when a rule is placed in a cache, all its descendants have to be placed in the cache to ensure correctness. In Table 1, if R_4: 10** is placed in the cache, both its prefixes R_2: 1000 and R_3: 100* have to be in the cache. This is because if R_2 and R_3 were not placed in the cache, messages with the headers 1000 and 1001 would have matched with R_4, rather than R_2 and R_3, respectively. It has been shown in [6] that an optimal caching, in terms of maximizing total weight (hit ratio), is NP-hard under the general dependency constraints where rules form a DAG. Several greedy solutions have been proposed, but without no approximation bounds. In this paper, we discuss three greedy solutions for rules that form a forest of trees, including two with a constant bound in terms of total weight. The first one respects the dependency constraint; the second one breaks the constraint by inserting a dummy rule (called *deny rule*) in the cache ; and the third one combines the first and second greedy solutions. In addition, we use dynamic programming (DP) to solve problem optimally (in a setting without deny rule), but with a high complexity (i.e., heavy-weight). Our contributions can be summarized as follows, where n is the rule number and k is the cache size:

1) Our greedy algorithm with the dependency constraint has a complexity of $O(n + k \log n + k^2)$ and a approximation ration of 2.
2) The complexity of the greedy algorithm that breaks the dependency constraint by introducitng deny rule is $O(kn)$ using two-level heaps.
3) The combined greedy aglrotihm using the above two greedy approaches has an approximation ratio of 24/5.
4) The DP solution has complexity of $O(k^2 n)$.
5) Extensive simulations are conducted together with an



(a) Branch in a max branch



(b) Segment a max segment

Fig. 2: Topological sorting order for a max branch (a) and a max segment (b) in non-increasing order from left to right.

optimal solution based on dynamic programming.

The remainder of the paper is organized as follows. Sections 2 and 3 discuss two greedy solutions, one for the basic model using a branch for caching and the other one for an enhanced model that cuts a branch with a dummy rule, together with complexity and performance analysis. Section 4 reviews an optimal solution based on dynamic programming. Section 5 conducts a simulation study. Section 6 concludes the paper.

II. Greedy Solution One

Given a rule set in a forest of trees (see Fig. 1), a tree *branch* consists of a rule and all its descendants. Due to the space limit of the cache, we can only accommodate a branch of a size up to k (cache size). A *max branch* is a branch that meets the following conditions: (1) the branch size is k, or (2) if its branch size is less than k, then it is not a branch of another branch with a size of k or less. In Table 1, when $k = 3$, the max branches are $[R_1]$, $[R_2, R_3, R_4]$, and $[R_5]$; when $k = 5$, the max branches are $[R_1]$ and $[R_2, R_3, R_4, R_5, R_6]$. Note that when $k = 3$, R_6 does not belong to any branch, i.e., it cannot be cached. This represents the fact that not all rules can be cacheable due to the dependency condition.

Based on the definitions of the branch and the max branch (see their topological sorting order representations in Fig. 2(a)), we have the following properties: the set of max branches includes all eligible branches that can be placed in the cache. Each rule in a max branch together with its descendants form a branch. For example, in max branch $[R_2, R_3, R_4, R_5, R_6]$, R_2 forms branch $[R_2]$, R_3 forms $[R_2, R_3]$, R_4 forms $[R_2, R_3, R_4]$, R_5 forms $[R_5]$, and R_6 forms $[R_2, R_3, R_4, R_5, R_6]$.

Now, we calculate the *unit benefit* for each branch in a max branch. Each rule has a unit cost, C, with a weight, W. Our goal is to find a branch that maximizes the unit benefit, defined as the ratio of rule weight to rule cost ($\Delta W/\Delta C$). The challenge lies in the fact that once a candidate branch is selected and removed for future consideration, unit benefit values of other branches in the same max branch need to be updated. Therefore, we need to find an appropriate data structure to control the complexity.

The solution is to maintain a special branch that corresponds to the maximum unit benefit for each max branch. Then we build a heap that consists of all these special branches.

Max branches

Algorithm 1 Greedy Algorithm One

```
1: Construct max branches with each max branch arranged
   in a topological sorting order.
2: Calculate unit benefit values for all branches in the max
   branches.
3: Find a special branch, called the max unit benefit branch,
   that has the maximum unit benefit in each max branch.
4: Maintain a heap for all max unit benefit branches.
5: repeat
6:     Determine the branch that has the maximum unit benefit in the heap.
7:     Remove the branch and update the corresponding max
       branch, including its new max unit benefit branch.
8:     Update the heap accordingly based on the new max
       unit benefit branch selected by b).
9: until the cache exceeds its capacity
10: Suppose Steps 5 to 9 stop at round $i+1$, i.e., the first $i$
    branches have not exceeded the cache capacity. Then, we
    select the larger of the two in terms of weight: the first $i$
    branches or the $(i+1)$th branch.
```

1) Calculate the branch size for each node through a simple traversal of each tree from lowest to highest in the topological sorting order.
2) Traverse each tree again from lowest to highest and identify each branch that is not a descendant of another branch of size k or less.
3) Extract the branch as a new max branch before continuing the traversal process.

Fig. 1 shows the branch size for each rule shown in Table 1. When $k = 3$, $[R_1]$, $[R_2, R_3, R_4]$, and $[R_6]$ are three max branches identified through the tree traversal process starting from the root of each tree. It is clear that the complexity for extracting the max branches is $O(n)$. Now we can present the greedy algorithm.

Theorem 1: The complexity of Greedy Algorithm One is $O(n + k \log n + k^2)$.

Proof: The cost of Step 1 is $O(n)$ for constructing max branches and maintaining a topological sorting order for each max branch. Steps 2 and 3 together cost $O(n)$ because one scan of each max branch is sufficient. Step 4 costs $O(n)$ for heap construction. For the loop in Steps 5 to 9, the max heap in Step 6 costs $O(\log n)$; one max branch update in Step 7 costs $O(k)$, and the heap update in Step 8 costs $\log n$. As there are at most k iterations in Step 5, the overall cost for Step 5 is $O(k(\log n + k))$. Adding all the costs together, we obtain the result. $\square$

Note that k in general is relatively small compared to n. If $k = O(\sqrt{n})$ as in many practical cases, the complexity becomes linear $O(n)$.

Theorem 2: Greedy Algorithm One has an approximation ratio of 2 compared to the optimal result on the total weight.

Proof: By way of algorithm construction, it is easy to verify that all combinations of rules as cache units are considered

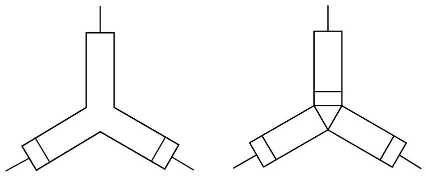

(a) A fork (b) Three segments

Fig. 3: Converting a fork into multiple segments.

in the branch selection algorithm, subject to the dependency and cache capacity constraints. The branch selection strictly follows the unit benefit value of each unit. Also, it is clear that the total weight of the optimal solution is less than the total weight of the first $i + 1$ selected branches. As the final result is at least half of that weight, which is either the first i selected branches or the $(i+1)$th branch, whichever is larger, the approximation ratio is bounded by 2. □

In Table 1, when $k = 3$, branch $[R_2, R_3, R_4]$ is selected first with a unit benefit of 21. Then branch $[R_5]$ is selected. When $k = 5$, branch $[R_2, R_3, R_4, R_5, R_6]$ is selected with a unit benefit of 19.

III. Greedy Algorithm Two and Combined Greedy

The dependency constraint causes inefficiencies in cache placement, because when a rule is placed in cache, all its descendants need to be in the cache as well. This problem is more prominent when there are many low-weight rules in the dependents of a high-weight rule. [6] introduced a *deny rule* to cut some sub-branches from a branch which results in a *segment* as shown in Fig. 2 (b). This is done by introducing a dummy rule in place for sub-branch R' with the same binary string as R', and an action for *forward_to_software_switch* (SS) is used when there is a match for R'. In Table 1, R_2 and R_3 are not profitable when they are used together with R_4 with a combined unit benefit value of 21. Suppose we remove R_2 and R_3 and replace them with a special rule R_3^* : $100*$ with action SS. R_4 and deny rule $R_3^{'}$ together have a new unit benefit value of 21.5. Messages with headers that match both R_2 and R_3 will match R_3^* in the cache, but will jump to software switches. In this way, we can use any segment of a branch without the dependency constraint. In this case, the extra slots saved in the cache will be used later for other rules, but one slot used by the deny rule is wasted.

One challenge in the extraction of a segment occurs when the segment goes across a *fork*, where a branch spawns out multiple sub-branches (see an example shown in Fig. 3 (a)) with two sub-branches). The main problem with forks is the algorithmic complexity in selecting a max unit benefit segment. For example, to select a segment of length k at a fork with l sub-branches, there are $\binom{l+k}{k}$ choices instead of one choice in a branch without a fork. In addition, l deny rules are used to extract such a segment (see Fig. 3 (a) for an example of $l = 2$). In Greedy Algorithm Two, we only consider segments that do not contain a fork. A max branch is replaced by a *max segment*, defined as the longest segment without a fork. In Table 1, max segments include $[R_1]$, $[R_2, R_3, R_4]$, $[R_5]$, and $[R_6]$. Note that the length of a max segment can exceed

Algorithm 2 Greedy Algorithm Two

```
1: Construct max segments through traversing the tree once.
2: Calculate $\max s(R)$ for each rule in the max segments.
3: Maintain a local heap ($l$-heap) of $\max s(R)$ for each max
   segment. The max heap of this local heap is the max unit
   benefit segment.
4: Maintain a global heap ($g$-heap) for the max heaps of all
   local heaps.
5: repeat
6:    Determine the max heap of the global heap.
7:    Remove the segment of the global max heap and update
      the corresponding max segment, including the new max
      heap in its local heap.
8:    Update global heap based on the new local max heap.
9: until the cache exceeds its capacity
10: Same as Step 10 of Greedy Algorithm One
```

k, unlike a max branch. However, a segment for caching is limited to $k-1$ as one slot is used for the deny rule.

We denote a special set of segments, $s(R)$, to represent all segments starting with R in a max segment (see Fig. 2 (b)). $\max s(R)$ is a segment in $s(R)$ with the maximum unit benefit: $\Delta W/(\Delta C + 1)$, where ΔC is the chain length and the extra 1 is the slot used for the deny rule. No deny rule is needed if R is a leaf rule without any descendants. If R is an independent single rule (such as R_1 in Table 1), the value for $\max s(R)$ is W/C, as no deny rule is needed.

Fig. 4 shows the construction of a global heap (g-heap) from the max heaps of all local heaps (l-heaps). When we remove a rule segment, we do not remove the rule that is currently held by a deny rule. The reason for this is that if that rule is selected, the deny rule can be removed (by "merging" two rule chains into one). In Table 1, if R_3: 100* is eventually selected, the deny rule R_3^*: 100* (SS) for R_4 will be removed.

Theorem 3: The complexity of Greedy Algorithm Two is $O(kn)$.

Proof: Step 1 costs $O(n)$ for one scan. Step 2 can use the same scan as Step 1, but for each rule, up to k updates to its descendants are needed. Therefore, the cost is $O(kn)$. The cost of building local heaps is bounded by $O(n)$, as there are at most n units for all heaps. The size of a global heap is bounded by n in Step 4. Step 6 costs k (the number of rounds). Step 7 may affect values for up to k $\max s(R)$ in all rounds in Steps 5 to 9. Therefore, the total cost is $O(k \log n)$ for all local heaps' updates. Step 8 costs $O(k \log n)$ for all global updates. Therefore, the overall cost is $O(kn)$. □

We can combine Greedy Algorithm One and Greedy Algorithm Two to create a Combined Greedy Algorithm as follows: we use the same criterion based on the maximum unit benefits of branches and segments for caching. The updates are conducted on the two separate sets of data structures used in these two algorithms. The complexity will remain $O(kn)$. The reason is that for each selection of a branch in Greedy Algorithm One, one and only one max segment will be affected. Likewise, each selection of a segment in Greedy Algorithm Two will affect one branch.

To derive the approximation ratio for the Combined Greedy Algorithm, we first prove the following bound on unit benefit loss when a fork is converted into multiple segments (see one example in Fig. 3 (b)).

***Theorem** 4:* The unit benefit loss in terms of the ratio of unit benefit of the fork and the unit benefit of the corresponding multiple segments is 6/5.

Proof: Consider a fork with l branches with weight ΔW and cost ΔC. The unit benefit of the fork is $\Delta W/(\Delta C + l)$, where l deny rules are used. The unit benefit of the segments is $\Delta W/(\Delta C + l + 1)$. Clearly, the fork to segment ratio is $1 + 1/(\Delta C + l)$. Since each branch needs at least one unit, the minimum ΔC is $l + 1$. The minimum l is 2. Therefore, the maximum ratio is 6/5.

***Theorem** 5:* The Combined Greedy Algorithm has an approximation ratio of 24/5 compared to the optimal scheme in terms of total weight.

Proof: We prove that for a segment with a deny rule, its total weight ΔW will be at least half the weight of the optimal scheme when the greedy method is used based on the unit benefit. We only need to calculate the "waste" introduced by the slot used for the deny rule. Suppose the weight of the rule used by the deny rule is more than ΔW, then, the slot for the deny rule would have been selected in an early round because this slot plus one deny rule will generate a segment with a unit benefit of over $\Delta W/2$. That is, there is no waste of "space" used for the deny rule. If the weight of the rule used by the deny rule is no more than ΔW, then, the optimal solution with the weight of the rule used by the deny rule included has a weight of no more than $2\Delta W$. Combining the results for Theorem 2 and Theorem 4, we have an approximation ratio of $2 \times 2 \times 6/5 = 24/5$. □

In the example shown in Tab. 1, $\max s(R_1)$, $\max s(R_2)$, $\max s(R_3)$, $\max s(R_4)$, $\max s(R_5)$, and $\max s(R_6)$ are $[R_1]$, $[R_2, R_3, R_4]$, $[R_3, R_4]$, $[R_4]$, $[R_5]$, and $[R_6]$, respectively. Clearly $\max s(R_4)$ has a maximum unit benefit of 21.5 with deny rule $R_3^* : 100 * (SS)$. $\max s(R_2)$ and $\max s(R_3)$ are updated to $[R_2, R_3]$ and $[R_3]$ after the removal of R_4, respectively. $\max s(R_5)$ is the next candidate with a maximum unit benefit of 20. $\max s(R_5)$ does not have the deny rule as R_5 is the head of a rule chain. The next candidate is $\max s(R_3)$ with a unit benefit of 13 because no new deny rule is used as its immediate predecessor, R_4, is already in the cache. In this case, deny rule $R_3^* : 100 * (SS)$ is replaced by deny rule $R_2^* : 1000$. If $k = 5$, $\max S(R_6) : [R_6]$ is selected with a unit benefit of 12. Again, no new deny rule is needed as its immediate successors, R_4 and R_5, are already in the cache. Therefore, for the Combined Greedy Algorithm, the deny rule is not needed when its immediate predecessor or all its immediate successors of $\max s(R)$ in the rule set are already in the cache.

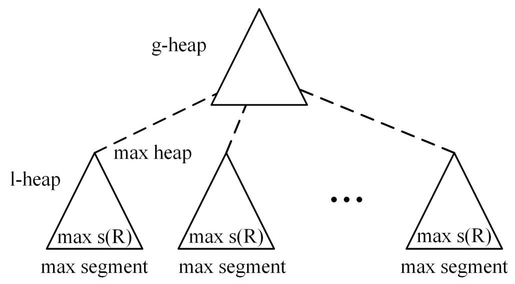


Fig. 4: Constructing the global heap (g-heap) from the max heaps of local heaps (l-heaps).

IV. Dynamic Programming Solution

This section introduces an dynamic programming (DP) solution to maximize the number of rule hits. Since rule dependencies form a forest rather than a tree, a dummy root with a weight of zero is introduced. We start with sorting all rules in a *depth-first order*. For example, in Fig. 1, such an order can be $\{R_0, R_1, R_6, R_5, R_4, R_3, R_2\}$ with dummy root R_0. Let $T[R, m]$ be the subtree induced by rule R, its first m children (in terms of the depth-first order), and all descendants of these m children. For example, in Fig. 1, $T[R_6, 0]$ only includes R_6, $T[R_6, 1]$ includes R_5 and R_6, and $T[R_6, 2]$ includes R_2, R_3, R_4, R_5, and R_6. Let $d(R)$ be the number of children of R.

Let $O[R, d, m]$ denote the optimal cache-hits in the sense that we cache m rules out of $T[R, d]$. These m rules follow only the dependencies within $T[R, d]$ and will ignore the other dependencies. By definition, $O[R_0, d(R_0), k]$ is our objective. Let W_i denote the cache-hit of R_i. The initialization is:

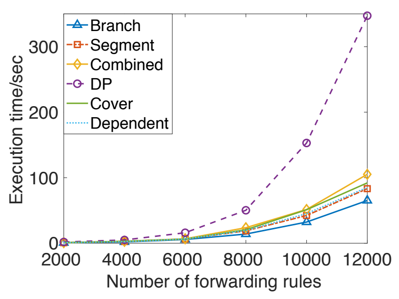
$$O[R_i, 0, m] = \begin{cases} W_i & \text{if } m \geq 1 \text{ and } i \neq 0 \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

The optimal recurrence pattern is:

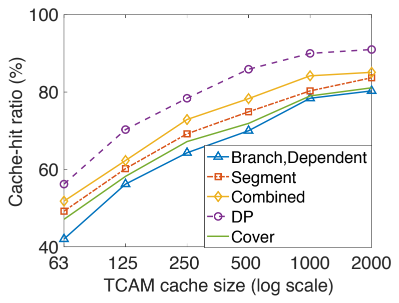
$$O[R_i, d, m] = \max \Big\{ O[R_i, d-1, m], \max_{0 \leq m' \leq m} \Big[O[R_{id}, d(R_{id}), m'] + O[R_i, d{-}1, m{-}m'] \Big] \Big\} \quad (2)$$

In Eq. 2, R_{id} is the d-th child of R_i. Note that $O[R_i, d, m]$ either ignores its descendants under the d-th child or uses these descendants. For the latter case, we additionally require that the composition of $O[R_{id}, d(R_{id}), m']$ and $O[R_i, d{-}1, m{-}m']$ follow the dependencies within $T[R_i, d]$. R_i is cached if and only if all the rules in $T[R_i, d]$ are cached.

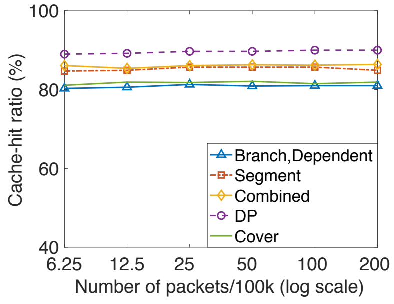
The time complexity of the dynamic programming solution is $O(k^2 n)$. This is because Eq. 2 takes $O(k)$ to calculate, and we need to compute Eq. 2 for each rule R_i and each m $(0 \leq m \leq k)$. Consequently, the DP solution has a larger time complexity than the greedy solution one, but has a better performance than the greedy solution one. However, the dynamic programming solution may not be able to incorporate deny rules. This is because deny rules lead to an exponential number of combinations, as in Eq. 2.



(a) Algorithm execution time.



(b) Cache hit traffic and TCAM size.



(c) Cache hit traffic and the number of packets.

Fig. 5: CAIDA packet trace.

V. Implementation

A. Real Traffic Traces and Settings

Simulations are conducted based on the publicly available CAIDA packet trace of 2016 from the Equinix datacenter in Chicago [10] and the real-world Cisco router configuration on a Stanford backbone router [11].

CAIDA: The packet traces have a total of 20 million packets sent over 30 minutes on high-speed Internet backbone links. Since CAIDA does not publish the policy used to process these packets, we build a policy by extracting forwarding rules based on the destination IP addresses of the packets in the trace. We obtain around 12,000 IP destination-based rules.

Stanford Backbone: The policy has around 180K OpenFlow rules. We follow the processing method in [6] and generate a packet trace that matches the routing policy by assigning a traffic volume to each rule drawn from a Zipf distribution [12]. The resulting packet trace had around 30 million packets randomly shuffled over 30 minutes.

B. Comparison Algorithms and Metrics

Two baseline algorithms, Dependent (set) and Cover (set), in [6] and our four proposed algorithms are included. The Greedy Algorithm One, Two, and the Combined Algorithm are denoted as Branch, Segment, and Combined, respectively. The difference between Dependent and Branch is that Branch includes how to split dependent rules and ensure the complexity bound. The Cover algorithm in [6] relaxes the rule dependency constraints by creating a small number of deny rules that cover many low-weight rules, but it does not guarantee the performance.

Three comparison metrics are employed: (i) Execution time. It measures the time of running the algorithms to decide which rules are preferred to be cached in the TCAM hardware. The time unit is seconds. (ii) Cache-hit traffic ratio on the variability of the TCAM size. We observe the cache-hit traffic ratio over the total traffic when the TCAM size changes from 125 to 2,000 entries. (iii) Cache-hit traffic ratio on the variability of the number of packets. We verify the efficiency of our three algorithms by applying the real trace set. We randomly select packets to test the metrics. Additionally, to further improve the performance without increasing the complexity, we use the following simple enhancement: when the cache is not full, our algorithms will continue filling the cache using the remaining rules based on the topological order until either the cache is full or the rule set is exhausted.

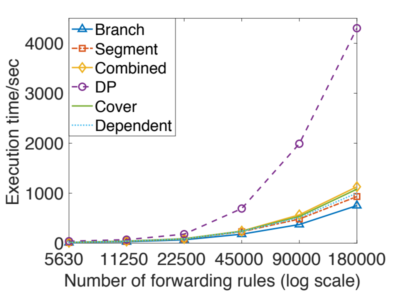
Since the rule coding has been studied in [8], it is out of our scope. As rules are inserted/deleted dynamically based on the changes in policy/traffic demands, rule caching should adapt quickly (i.e., not offline), if not immediately, in most of current systems. A long adaption time is intolerable, for example, 15 minutes update time of 180K rules to mitigate a DDos attack [6]. There is a trade-off between caching efficiency and program execution time.

C. Performance with the CAIDA traffic trace

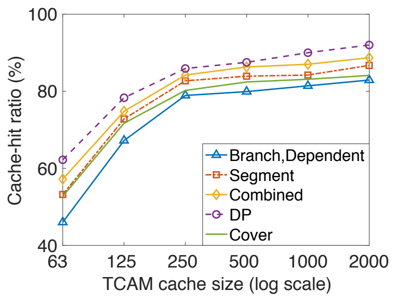
Fig. 5 shows results with the CAIDA traffic trace. We find that there are many shallow dependencies in the dependency graph. The depth of the dependency chains varies from 1 to 5. Many leaves' depths are 2 or 3. These shallow dependencies incur the advantages of our greedy algorithms not so obvious. Fig. 5(a) shows the execution time of the six algorithms with the variability of rules. The dynamic programming method DP, the red line, takes the longest time. The other five take much less time than DP, because they are greedy algorithms that make locally optimal choices without the need to search all possibilities. All of them take at most 30% of Algorithm DP's execution time. Because of the elaborate heap implementation, Algorithm Branch is faster than Algorithm Dependent by at most 21%, and Algorithm Segment takes only 84% time on average compared to Algorithm Cover. As Algorithm Combined selects the better choice between Branch and Segment, it costs a little more time but the performance improves a lot.

Fig. 5(b) lists the cache-hit traffic ratio over the total traffic on the variability of TCAM size. An advanced Pronto-Pica8 3290 switch has an ASIC that can hold 2,000 OpenFlow rules. We use the logarithm as the x-axis to show the difference more clearly. Algorithm DP does the best among all six algorithms because it sacrifices more time and memory to pursue the optimal solution. Our four algorithms achieve at least a 79.8% hit ratio with 2,000 rules, which is just 1.1% of the total rule table. Algorithm Branch achieves around 77.8% , Algorithm Segment is around 84.2% and Algorithm Combined is around 86.6% against the optimal algorithm DP's hit ratio.

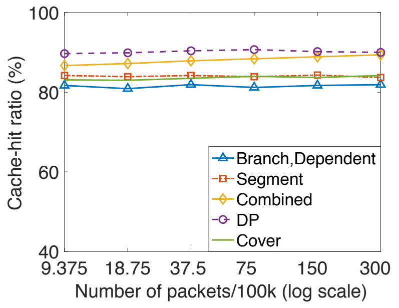
Fig. 5(c) shows the results of the cache-hit traffic ratio on the variability of the number of packets. We aim at testing our algorithms' universality. The performance of all four



(a) Algorithm execution time.



(b) Cache hit traffic and TCAM size.



(c) Cache hit traffic and the number of packets.

Fig. 6: Stanford backbone router.

algorithms keeps steadily. Algorithm DP still does the best among all six algorithms. It achieves a cache-hit rate around 90.0% when the packet number varies from 0.6 to 20 million.

D. Performance with the Stanford backbone router

The performance with the Stanford backbone router is shown in Fig. 6. The forwarding actions have fewer choices. Thus, the depth of the dependency chains varies from 1 to 8 and there are fewer shallow dependencies. The performances of our algorithms are better. Fig. 6(a) is the execution time result depending on the forwarding rules that vary from 563 to 18,000. It has more rules than CAIDA. The growth in number makes Algorithm DP try more possibilities while searching for the optimal one. Then, the complexity increases and it needs more time as shown in the red line. Reversely, compared to Fig. 5(a), our three greedy algorithms' times are only 82%, 79% and 83%, respectively. This illustrates that these algorithms have better performances in the deeper chain.

The cache-hit ratio on the variability of TCAM size is shown in Fig. 6(b). The basic increasing tendency is analogous to Fig. 5(b). With more entries, the cache-hit probability of the hardware switch becomes larger. All six lines stabilize earlier than CAIDA. This is relative to the deeper dependency chain and the less shallow dependencies in this data set. It also illustrates that deeper dependencies need less forwarding entries because applying a deeper dependency rule gains more benefits. Additionally, the optimal solution's performance is undoubtedly the best and the ratio reaches 92.3% when the TCAM entry number is 2,000. Our three greedy algorithms achieve better ratios than the CAIDA one with the same TCAM size.

Fig. 6(c) shows the performance on the variability of the number of packets. It is similar to the CAIDA one in that the more packets there are, the higher the cache-hit ratio is. All our four algorithms' performances are better than in the CAIDA data set. For 30 million packets, Algorithm DP's cache-hit ratio reaches 90.2%, Algorithm Combined reaches 89.4%, Algorithm Segment reaches 83.7% and Algorithm Branch reaches 81.9%. Thus, Algorithm DP achieves the best performance in the rule placement at the cost of a longer execution time and more memory space. Algorithm Branch takes the least execution time and achieves the same performance as Algorithm Dependent, the same situation of Algorithm Segment against Algorithm Cover. Algorithm Combined is the best deal because of the trade-off between its complexity and its performance.

VI. Conclusion

This paper studies the efficient rule cache problem in SDNs with dependencies of rules form a forest of trees. We propose three greedy effective algorithms. The first one with approximation 2 respects the dependency constraint. The second one inserts deny rules that relax the dependency constraint. The third one combines the first and second to have an approximation of $\frac{24}{5}$. We also apply dynamic programming to generate an optimal solution without deny rules, but slow (i.e., heavy-weight), solution. We evaluate our algorithms through real data-driven simulations.

References

[1] R. MacDavid, R. Birkner, O. Rottenstreich, A. Gupta, N. Feamster, and J. Rexford, "Concise encoding of flow attributes in sdn switches," ser. SOSR, 2017.

[2] B. Salisbury, "Tcams and openflow: What every sdn practitioner must know," *See http://tinyurl. com/kjy99uw*, 2012.

[3] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter, "Past: Scalable ethernet for data centers," in *CoNEXT, 2012*.

[4] L. Peng, W. Lu, and L. Duan, "Power efficient ip lookup with supernode caching," in *GLOBECOM, 2007*.

[5] Z. Huang, G. Liu, and J. Peir, "Greedy prefix cache for ip routing lookups," in *I-SPAN, 2009*.

[6] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Cacheflow: Dependency-aware rule-caching for software-defined networks," in *Symp. on SDN Research*, 2016.

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

[8] S. Hu, K. Chen, H. Wu, W. Bai, C. Lan, H. Wang, H. Zhao, and C. Guo, "Explicit path control in commodity data centers: Design and applications," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2768–2781, 2016.

[9] H. Lim, C. Yim, and E. E. Swartzlander Jr, "Priority tries for ip address lookup," *IEEE Transactions on Computers*, vol. 59, no. 6, pp. 784–794, 2010.

[10] "The caida anonymized internet traces 2016 dataset," http://www.caida.org/data/passive/passive 2016 dataset.xml.

[11] "Stanford backbone router forwarding configuration." http://tinyurl.com/oaezlha.

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