Cooperative Rule Caching for SDN Switches

Ori Rottenstreich*, Ariel Kulik*, Ananya Joshi†, Jennifer Rexford‡, Gábor Rétvári§ and Daniel S. Menasché¶
* Technion, Israel
† Carnegie Mellon University, PA
‡ Princeton University, NJ
§ Budapest University of Technology and Economics, Hungary
¶Federal University of Rio de Janeiro, Brazil

***Abstract*—Despite the tremendous success of SDNs in datacenters, their wide adoption still poses a key challenge: the packet-forwarding rules in switches require fast and power-hungry memories. Rule tables, which serve as caches, are of limited size in cheap and energy-constrained devices, motivating novel solutions to achieve high hit rates. In this paper, we leverage device connectivity in the fast data plane, where delays are in the order of few milliseconds, and propose multiple switches to work together to avoid accessing the control plane, where delays are orders of magnitude greater. As a low priority rule in a cache entails caching higher priority rules, we pose the problem of cooperative caching with dependencies. We provide models and algorithms for cooperative rule caching with dependencies, accounting for dependencies among rules implied by existing switch memory types. We develop caching algorithms for several typical use cases and study the difficulty to find an optimal cooperative rule placement as a function of the matching pattern, which lay the foundations of cooperative caching with dependencies.**

I. Introduction

Software defined networks (SDNs) have been tremendously successful in simplifying the management of data centers, enabling dynamic and efficient network configuration and fast failover. In an SDN, a controller enforces fine-grained policies by installing *rules* in switches that dictate how each switch handles incoming packets. Each rule consists of a *match* and an *action*. The match portion typically matches on multiple packet header fields, including wildcards, for classification. The actions can modify header fields, forward to a particular output port, or drop the packet. In addition, each rule has a *priority* that disambiguates between rules with overlapping match patterns (overlapping rules). Commodity switches implement rule tables using special hardware like Ternary Content Addressable Memory (TCAM) that checks a packet against all installed rules in parallel.

Flexible match-action processing is desirable across a range of settings, including IoT networks [1]–[6]. However, the adoption of SDNs outside of data-centers networks still faces a fundamental problem: rule tables require fast and power-hungry memories [7]–[12], but TCAMs must be of limited size in cheap and energy constrained devices [13]–[15]. To guarantee classification correctness a switch cannot simply "cache" the most popular rules due to rule *dependencies* [16]. For two rules with overlapping match patterns, caching the lower-priority (dependent) one entails caching the other with higher priority, even if caching the higher-priority rule does not contribute much to the total hit rate [16], [17].

978-1-7281-9486-8/20/$31.00 ©2020 IEEE

When a packet arrives at a switch and does not match any of its cached rules, it is common practice to assume that the *default rule* is to forward the packet to the control plane (or to a slower data path, like in Open vSwitch [18], [19]). A major challenge consists in coping with the control-plane delay, which is typically an order of magnitude larger than forwarding in the fast data plane. Hence, SDN faces a major performance challenge of needing to achieve a high hit rate despite the small rule tables. We believe this is possible, but only if multiple switches can work together to achieve a high overall hit rate.

We pose the following question: how to leverage *cooperation* among switches to improve caching performance? Conceptually, the answer is simple: switches can forward unmatched packets to other switches in the fast data plane, e.g., by configuring *default rules* and a time-to-live (TTL) counter which is decremented at every hop before relying on the control plane when its value reaches zero. Under pairwise switch cooperation, for instance, the TTL of unmatched packets is set to one. Even though such a simple idea, which poses no additional complexity regarding system design, may lead to significant gains in terms of delay reduction, it also poses novel challenges in the realm of cooperative placement of objects with dependencies among them. *One of our aims is to lay the foundations of cooperative caching with dependencies.*

Prior art. *Cooperative caching* [20] is a well-studied approach comprising the coordination of a distributed caching system to achieve a common goal, such as increasing the total system hit rate. Cooperative caching networks have been considered for a wide variety of applications, including cellular [21]–[24] and IoT networks [25], CDNs [26]–[28], social networks [29], [30], and distributed operating systems [20]. In the realm of SDNs, it has already been noted that a slight increase of load among switches may correspond to a significant reduction in communication costs across the control plane [7], [11], [12]. In this paper, we build on such previous works, considering cooperative caching with dependencies.

Contributions. Our main contributions are twofold.

(1) **Cooperative rule-caching model:** We propose an analytical model to analyze cooperative caching solutions, accounting for rule-dependencies (Section III).

(2) **Caching solutions:** Leveraging the proposed model, we design algorithms for the cooperative caching problem under

different types of rule dependencies (Sections IV-VI).

We see the proposed caching algorithms as being run periodically by the SDN controller.

II. System description and results

Traditionally, rule caching is performed independently for each switch, considering the implemented policy and the local traffic distribution. Fig. 1(a) illustrates an example of two switches implementing two different policies with six and five rules in Switches 1 and 2, respectively. For simplicity, the rules match disjoint traffic patterns and, as such, have no dependencies. The limited capacity of each switch enables caching three rules, as illustrated by the dashed lines. Each rule is associated with a matching probability based on the switch's workloads. Fig. 1(a) shows traditional rule caching where in each switch the three most popular rules are cached.

In this work, we focus on *the advantages of cooperative caching for rule caching in SDNs*. The centralized control of SDNs naturally motivates cooperative caching. By allowing packets to be forwarded to other switches to complete the classification process within the data plane, the load on the control plane and the time to resolve requests are reduced.

Fig. 1(b) illustrates a cooperative-caching solution leveraging rule similarity across two switches. In case of a miss in a switch, there is an attempt to complete the classification by finding a matching rule in an adjacent switch.

Definition 1. *The origin switch of a given packet is the first switch to handle that packet.*

A packet is first handled by its origin switch. If it does not match any of its rules, it is possibly forwarded to a neighbor switch before reaching the control plane. Throughout this paper, except otherwise noted, we consider switches that are matched in pairs. Each switch relies exclusively on its designated partner for cooperative caching purposes. We assume that the pairwise matching of switches is provided as input such that paired switches should be directly connected. In case a rule is not matched for a packet in its origin switch, the packet is forwarded to the paired switch. If a match is found in the paired switch, the packet is forwarded back to its origin switch with the information on the selected action, noting that some particular rule actions, such as a packet drop, can be executed directly by the paired switch. Only if the rule to handle the packet cannot be resolved in the data plane by a switch or its partner, the packet is directed to the control plane.

The simple setting considered in this paper, accounting for *pairwise switch cooperation*, already allows us to appreciate the benefits and challenges involved in the deployment of cooperative caching with dependencies. In particular, the functionalities required for its deployment are already supported by current SDN architectures.

Each packet stores its corresponding origin switch, and each switch maintains for each rule in its cache a set of corresponding origin switches (one or more) to which it is applicable. In Fig. 1(b), the applicable origin switches are indicated between brackets: R_1 and R_3 refer to switch 1, R_7 and R_9 refer to switch 2, and the other rules refer to both switches. The local caching in Fig. 1(a) enables local classification of 0.79 and 0.75 of the traffic in the two switches, respectively. 0.21 and 0.25 of the traffic is classified in the slow path. With the same cache sizes, the cooperative caching (Fig. 1(b)) enables classifying within the data plane (namely, by one of the switches) 0.94 and 0.93 of the traffic, reducing the traffic sent to the slow path to 0.06 and 0.07. Fig. 1(c) shows the various classification times (see Section III).

Table I
New caching policy results (in boxed bold).

Rule Dependencies	No cooperation	Pairwise cooperation
Exact match (no rule dependencies)	Optimal	**Optimal**
Prefix match	Optimal [31]	**Optimal**
Wildcard match (NP-hard)	Heuristics [16]	**Heuristics**

Our approach determines the cached rules in each of the switches as allowed by its memory capacity while also taking into account the classifiers with rule dependencies and rule popularities over other switches.

Summary of results. Our results are summarized in Table I, where new results are described in boxed bold. We categorize the problem based on two main properties: *(i)* rule matching pattern (exact match, prefixes, wildcards) that affects possible rule dependencies; *(ii)* number of switches involved in a caching decision. We focus on the case of cooperation among pairs of switches and in Section VII discuss other forms of cooperation among switches. Our goal is to determine the caches content to minimize the average classification time while preserving caching correctness.

We first refer to *exact* match for which rules have no dependencies.

Definition 2. *An* exact *matching rule is a rule with specific field values (no wildcards).*

The optimal caching under exact matching rules for a single switch encompasses caching the rules with the highest popularity, noting that all rules are of the same memory size. For cooperative switches, the optimal rule allocation is the solution to a linear program (Section IV).

Definition 3. *In a* prefix *matching rule a wildcard can appear only as a suffix.*

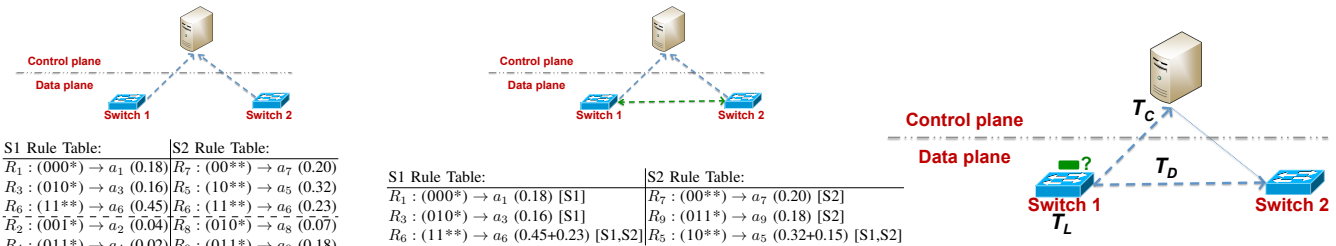
For *prefix* matching, we describe an *optimal* dynamic-programming algorithm in Section V.

Definition 4. *In a* wildcard *matching rule a wildcard can appear at any arbitrary position.*

For *wildcard* matching, the optimal rule caching problem is NP-hard even for a single switch [16], motivating a greedy heuristic for cooperative switches introduced in Section VI.

III. Modeling Cooperative Caching

We consider a network of k switches $1, \ldots, k$. Each switch i has an ordered set of r_i rules $(R_{i,1}, \cdots, R_{i,r_i})$. A rule has two parts: a matching pattern and an action. A matching pattern is composed of 0s and 1s or *s (don't cares). For instance, a



(a) Traditional rule caching (b) Cooperative rule caching (c) Classification delays

Figure 1. (a) *Traditional Rule Caching*: Each switch maximizes its (local) cache hit rate while considering rule dependencies. Each rule is characterized by its matching field DST IP and its popularity. Cached rules appear above the line. Unclassified traffic is served in the control plane. (b) *Cooperative Rule Caching*: A switch can store rules of other switches and serve their traffic. (c) *Classification delays:* $T_L \leq T_D < T_C$ for local classification, by another switch in the data plain or in the controller.

rule of the form (DST IP = 100*) $\rightarrow a$ matches both DST IP values 1000 and 1001 and applies an action $a \in \mathcal{A}$ where $\mathcal{A}$ is the set of possible actions. A packet is handled by the first rule it matches. Switch i serves traffic that follows a local traffic distribution. The distribution determines each rule popularity in a switch, i.e., its probability to be the first match of a packet.

Rules and popularities. Let $S = \{R_1, \ldots, R_r\}$ be the set of distinct rules that appear in one or more switches. We denote by S_j the set of input rules applicable to origin switch $j \in [1, k]$ such that $S = \bigcup_j S_j$. Let n_j be the cache size of switch j. The set of rules cached at switch j is denoted by C_j, $C_j \subseteq S$, $|C_j| \leq n_j$. A cached rule is marked with an indication to which origin switches it should be applied to. Let $M_R^i \subseteq \{j | R \in (S_j \cap C_i)\}$ be the set of origin switches to which rule $R \in S$ stored at $i \in [1, k]$ should be applied to.

Rule cooperation can take advantage of a partial (or complete) similarity between the rules in the different switches [32], [33]. Without loss of generality, we do not consider rules that are never matched in any of the switches. For $j \in [1, k]$, let $\lambda_{j,i}$ be the popularity of rule R_i in switch j, describing the amount of traffic that matches the rule. In particular, if a rule R_i appears only in switch 1, we have $\lambda_{j,i} = 0$ for $j \neq 1$.

Latencies and rule dependencies. We refer to the latency of the classification as the cost of the operation and would like to minimize the expected classification time. The latency is highly influenced by the location of classification.

If a packet matches one of the switch cached rules, it is classified accordingly within a very short time. If a packet does not match any of the switch cached rules, then it experiences a cache miss. In such a case, there are two alternatives where to find a corresponding rule and determine the required action for the packet. First, this can always be accomplished by sending the packet to the controller. We assume that the controller keeps an up-to-date version of the entire set of rules for all switches. In such a case the packet observes a relatively large delay. Alternatively, if the required classification information can be found in one of the other switches, the classification can be performed in such a switch within the data plane. For simplicity, we assume that in a case of a miss in a cache i, the classification time in any of the other $k-1$ switches is identical. As illustrated in Fig. 1(c), we refer to these three classifications as a *L*ocal, in the *C*ontrol plane and by cooperation among switches in the *D*ata plane. We denote their costs by T_L, T_C and T_D, respectively, such that $T_L \leq T_D < T_C$. *Our goal is to determine the content of the n_j cached rules in each switch for minimizing the average classification time.*

Let $\alpha = (T_C - T_L - T_D)/(T_C - T_L) = 1 - T_D/(T_C - T_L)$. The value of α represents the ratio of two time reductions. The first reduction is that of a data plane classification and the second reduction is that of a local classification, both in comparison with the expensive classification in the control plane. Note that $\alpha \in (0, 1)$. Intuitively, for larger values of α the relative overhead due to a classification in another switch is small when compared against a classification at the control plane, and the potential gains due to cooperative rule caching are more significant.

Latency parametrization. Consider for instance the following values as an estimation for the different delays, which we obtained in a `mininet` experiment with the Ryu [34] controller as the slow path and Open vSwitch (OVS) as the data plane: $T_L = 3$ ms, $T_D = 4$ ms, $T_C = 200$ ms (see Section VIII). Although classification in an adjacent switch is slower than a local classification, it still avoids most of the latency that occurs while using the controller. This results in a value of $\alpha = (200 - 3 - 4)/(200 - 3) \approx 0.98$, very close to 1.

Caching gain. Recall that C_i is the rule set cached at switch i. Consider a scenario with two switches. In comparison with a scheme without caching, cost reduction following caching a rule R_i in both switch 1 and switch 2, $R_i \in C_1 \cap C_2$, is $\lambda_{1,i} + \lambda_{2,i}$, in units of $T_C - T_L$. We refer to such cost reduction on top of a scheme with no caching as the *caching gain*.

A switch benefits from caching of one of its rules in another switch. The value of caching a rule R_i only in switch 1, $R_i \in C_1 \setminus C_2$, is $\lambda_{1,i} + \alpha \cdot \lambda_{2,i}$, where $\alpha \in (0, 1]$, as above, is determined by the delays of the various classification options. Symmetrically, a rule R_i cached only in switch 2, $R_i \in C_2 \setminus C_1$, contributes $\alpha \cdot \lambda_{1,i} + \lambda_{2,i}$. There is no contribution for rules not cached in any switch.

Rule dependencies and correctness. To guarantee the correctness of a rule caching strategy, we must ensure that if a rule is the first to match a packet among the cached rules, the same rule is the first to match that packet in the complete set of

rules. Consider a rule R_{low} that intersects with a higher-priority rule R_{high}. Caching R_{low} entails caching R_{high}.

Consider rules that refer to traffic having switch 1 as its origin switch (Definition 1). Rules are cached either at switch 1 or 2, or in both, with an indication they apply only to switch 1. Let $R_{low} \in S_1$ be one of such rules that intersects with a higher-priority rule $R_{high} \in S_1$. The requirement of cache correctness within a single switch implies that if $R_{low} \in C_1$ then $R_{high} \in C_1$. The requirement for two-switches is more detailed. A packet origined at switch 1 accesses switch 2 only after a miss in C_1. First, if $R_{low} \in C_1$ then $R_{high} \in C_1$. Otherwise, $R_{low} \notin C_1$. If $R_{low} \in C_2$ then necessarily $R_{high} \in C_1$ or $R_{high} \in C_2$. Thus rule R_{low} will be matched at C_2 only after R_{high} was checked, either at C_1 or C_2.

IV. Exact Rule Matching

We consider the simplest setup wherein rules have no dependencies (Definition 2). We show that for the case of *exact match* rules an optimal caching of rules on multiple switches can be found in polynomial time. The algorithm uses a linear programming relaxation.

Let $G(\boldsymbol{x})$ denote the average gain in cost reduction experienced by users when rule placement strategy $\boldsymbol{x} = [x_{j,i}]$ is deployed. For each rule i, if that rule is stored in at least one cache, the gain is at least $\alpha \sum_j \lambda_{j,i}$. In addition, each cache storing rule i experiences an additional gain of $(1-\alpha)\lambda_{j,i}$. Denote by x_i an indicator variable, equal to 1 if rule i is stored in at least one cache, and 0 otherwise. We pose the following mixed integer linear program (MILP),

$$\max\ G(\boldsymbol{x}) = \sum_{i \in S} \left(x_i \cdot \alpha \sum_{j=1}^{k} \lambda_{j,i} + (1-\alpha) \sum_{j=1}^{k} x_{j,i} \cdot \lambda_{j,i} \right) \quad (1)$$

where

$$\sum_{i \in S} x_{j,i} \leq n_j \quad \forall j \in [1,k], \qquad x_i \leq \sum_{j \in [1,k]} x_{j,i} \quad \forall i \in S, \quad (2)$$

$$x_i \in \{0,1\} \quad \forall i \in S, \quad x_{j,i} \in \{0,1\} \quad \forall j \in [1,k], \forall i \in S. \quad (3)$$

When considering pairwise matchings of switches, we let $k = 2$ in the above MILP and treat each pair of switches independently. Constraints (2) correspond to cache capacities and to the definition of x_i. Next, we present the main result of this section, whose proof relies on [26], [35], [36]. Proofs are omitted for space constraints.

Theorem IV.1. *An optimal cooperative caching with exact rule matching can be found in polynomial time, as the problem* (1)-(2) *without constraints* (3) *admits an integral solution.*

The constraints of the relaxed problem can be written in the form $\boldsymbol{A}\boldsymbol{z} \leq \boldsymbol{b}$ where matrices $\boldsymbol{A}$ and $\boldsymbol{b}$ are all integers and $\boldsymbol{z}$ is a vector of $x_{i,j}$ and x_i. The proof of Theorem IV.1 follows from the fact that the standard form LP $\boldsymbol{A}\boldsymbol{z} \leq \boldsymbol{b}$, $\boldsymbol{z} \geq 0$, with integral matrix $\boldsymbol{A}$ and integral vector $\boldsymbol{b}$, has an integral optimal solution $\boldsymbol{z}$ if its constraint matrix $\boldsymbol{A}$ is totally unimodular [26], [35], [36]. According to the Hoffman sufficient condition (HSC) [37], matrix $\boldsymbol{A}$ is totally unimodular if it contains no more than one +1 and no more than one -1 in each column. It can be readily verified that the HSC holds for the considered MILP, which completes the proof.

V. Prefix Rule Matching

We study a common case of rule dependencies that appears for the scenario of prefix matching, known as longest prefix matching. Its simple dependencies allow us to derive an optimal caching strategy. Let W be the length in bits of the matching pattern of a rule. A prefix corresponds to a node in a binary tree with 2^W leaves. A rule is a pair of a prefix and an action from $\mathcal{A}$. The dependency among rules is illustrated in Fig. 2 where caching a prefix-action pair (P, a) entails caching all existing rules in the colored subtree.

Challenges. To characterize dependencies due to prefix rules, one alternative is to add constraints to the MILP of Section IV. Indeed, prefix rules correspond to constraints of the type $x_i \leq x_j$ for every rule i which is a parent of j in the prefix tree. Such additional constraints can result in non-integral solutions to the relaxed LP. For a concrete example showing that the relaxed LP accounting for prefix rules may admit non-integral optimal solutions, it suffices to consider a single switch. The switch resolves two rules, R_1 and R_2, matching prefixes $0*$ and 00, respectively. Matches to R_1 and R_2 occur at rates $\lambda_1 = 2$ and $\lambda_2 = 1$. If the switch capacity equals one, the only feasible solution is storing rule R_2. Consider now the MILP with objective of maximizing the gain as defined in Section IV, $\max(2x_1 + x_2)$, subject to $x_1 + x_2 = 1$ and $x_1 \leq x_2$, where $x_1, x_2 \in \{0,1\}$. The relaxed version of this MILP has a single fractional solution, $x_1 = x_2 = 0.5$, which does not correspond to a rule placement.

Correctness of solution. A solution to the caching problem consists of two components. First, it comprises an assignment of rules to switches. Recall that C_1 and C_2 refer to the set of rules stored at switches 1 and 2, respectively. Second, for each prefix $P \in C_i$ cached at switch $i \in \{1,2\}$ a solution determines the set of origin switches (Definition 1) marked as effectively applicable. Let M_P^i be the set of *marked* switches corresponding to prefix P at switch i. When a packet with prefix P from origin switch j arrives at switch i, it is resolved if its origin switch is marked, i.e., if $j \in M_P^i$. Note that $M_P^i \subseteq \{j | P \in S_j \cap C_i\}$, for $i \in \{1,2\}$. A solution is *correct* if it fulfills the caching correctness requirement (Section III).

A dynamic programming approach. We propose a dynamic programming algorithm, bearing some similarity to the algorithm of [31], to overcome the aforementioned challenges. We define a sub-problem for every combination of prefix P, two cache sizes $m_1 \leq n_1$ and $m_2 \leq n_2$, and a set of *compliance requirements* $\mathcal{Q}$, which we later define and discuss. We refer to each such sub-problem as $(P, m_1, m_2, \mathcal{Q})$. An optimal solution to $(P, m_1, m_2, \mathcal{Q})$ is a correct caching of the rules in the subtree of P (i.e., all the rules P' such that P is a prefix of P') into switches with memory sizes m_1 and m_2, which adheres to the compliance requirements in $\mathcal{Q}$, and provides a maximal

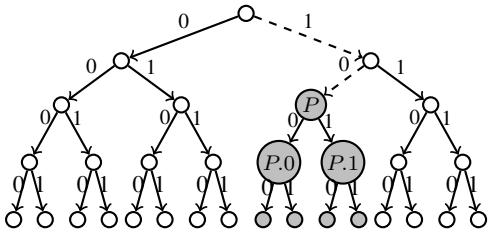


Figure 2. Prefix rule dependency. Caching a prefix $P = 10**$ requires caching all prefixes in the colored subtree such as $100*, 101*$ (if exists).

caching gain. The optimal cooperative caching is given by $(P_r, n_1, n_2, \emptyset)$, where P_r is the empty prefix that stands for the root of the prefix tree. By definition, an empty requirement set $\emptyset$ does not imply particular requirements.

For a prefix P, the sub-problems of P are all the sub-problems of the form $(P, m_1, m_2, \mathcal{Q})$ for some $m_1 \leq n_1$, $m_2 \leq n_2$ and a set of compliance requirements $\mathcal{Q}$. Then, the key insight consists in observing that the optimal caching gain $(P, m_1, m_2, \mathcal{Q})$ can be expressed as a function of the optimal caching gains of the sub-problems of $P.0$ and $P.1$, the two descendants of P in the prefix tree. Leveraging this insight, the solution to the problem is found through a bottom-up dynamic programming approach over the prefix tree.

A solution for the sub-problem $(P, m_1, m_2, \mathcal{Q})$ can be derived from solutions to $(P.0, m_1', m_2', \mathcal{Q}_0)$ and $(P.1, m_1'', m_2'', \mathcal{Q}_1)$ where, for $i = 1, 2$, either $m_i' + m_i'' = m_i - 1$ or $m_i' + m_i'' = m_i$, depending on whether prefix P is cached in switch i or not. The compliance requirements $\mathcal{Q}_0$ and $\mathcal{Q}_1$ are used to determine whether P can be added to any of the caches without violating the correctness requirement.

Consider a solution with caches C_1, C_2 for the sub-problem $(P, m_1, m_2, \mathcal{Q})$ satisfying $|C_1| \leq m_1$, $|C_2| \leq m_2$, wherein $C_1 \cup C_2$ is contained in the subtree rooted at P. An ordered pair (i, j) *can belong to the compliance requirements set* $\mathcal{Q}$ if in the solution all rules from S_j in the subtree rooted at P are either in C_j or in C_i with the indication of applicability to switch j. By the above arguments, the optimal gain under $(P, m_1, m_2, \mathcal{Q})$ can be expressed as a function of optimal gains under sub-problems subsumed by $P.0$ and $P.1$, potentially with the addition of a rule for P to C_1 or C_2 (or both). A central observation is that P can be added to C_i and marked for j if $(i, j) \in \mathcal{Q}_0$ and $(i, j) \in \mathcal{Q}_1$.

Theorem V.1. *An optimal cooperative caching with prefix matching rules under pairwise switch cooperation can be found in polynomial time.*

Note that the compliance requirements set $\mathcal{Q}$ satisfies $|\mathcal{Q}| \leq 4$ and the number of sub-problems for which solutions have to be considered is polynomial in the number of rules in $S_1 \cup S_2$.

VI. Wildcard Rule Matching

We study the case of wildcard rules. Such rules can have general rule dependencies. In the case of a single switch with general rule dependencies, the dependencies can be described in the form of a directed acyclic graph (DAG) [16]. Given such a dependency DAG, for correctness, when a rule is cached in the network switch, all its dependents (reachable via the directed edges in the DAG) have to be cached along with it. Consider two nodes u, v that refer to rules R_u, R_v. The graph is acyclic since an edge from u to v exists only if besides their intersection R_v precedes R_u in the classifier (i.e., it has a higher priority).

In this setting, the problem of rule caching is known to be NP-hard even for a single switch [16]. Naturally, the problem of rule caching across two (or more) switches is NP-hard as well. To see that note that for low enough values of the parameter α, an optimal joint caching is given by local optimal caching in each of the switches.

Corollary VI.1. *Finding an optimal caching policy for two or more switches with general rule dependencies is NP-hard.*

Correctness follows maintaining the requirements from Section III. Note that in the case of two switches, two intersecting rules can appear in the two switches in two different orders. Accordingly, for two switches the dependency graph accounting for all dependencies is not necessarily a DAG.

Since finding the optimal caching solution is NP-hard, a greedy heuristic can be used to pick the rules to be cached in a switch. For instance, one such heuristic consists in storing rules with the highest ratio of cost reduction achieved by caching the rule in the switch divided by the space needed to store its dependencies. Note that this heuristic is similar in spirit to an approximate solution to a knapsack problem.

VII. Beyond Pairwise Cooperation

We discuss potential extensions beyond pairwise matching.

Extensions. The solution from Section IV for the case of exact rule matching with no dependencies applies for an arbitrary number of k switches and optimal cooperative caching can be found in polynomial time in the same manner.

The design of optimal cooperative rule placement with dependencies, beyond pairwise matching of switches, is challenging. In particular, adapting the dynamic-programming approach that accounts for pairwise switch cooperation with prefix rules from Section V would imply running-time complexity which grows exponentially with the number of switches. For wildcard rules, the problem is NP-hard even for a single switch (Section VI).

Observation. As a potential building block in future extensions of our results beyond pairwise cooperation, we describe a simple approach of *conditionally optimal caching*. It selects the caching for a second switch after the caching in a first (or more) switches is determined. This can be done by solving the caching problem in a single switch using modified rule popularities. The approach, however, might not imply the optimal joint caching in multiple switches.

Theorem VII.1. *Consider two switches with sets of rules from S. Assume a caching, represented by the set of cached rule indices $C_1 \subseteq S$, is given for switch 1. An optimal conditional caching C_2 for switch 2, i.e., switch 2 best response, can be obtained as an optimal caching for a single switch with popularities of $\lambda_i = \lambda_{2,i} \cdot (1 - \alpha \cdot I(i \in C_1)) + \alpha \cdot \lambda_{1,i} \cdot I(i \notin C_1)$ for a rule R_i, where $I(\cdot)$ is the indicator function.*

VIII. Toy prototype testbed

We illustrate the benefits of cooperative rule caching in a simple real-life use case. To this aim, we consider a "Load Balancer and Access Gateway" prototype of an official industrial data-plane benchmark suit. This use case models a real pipeline that is actively being deployed as part of a commercial 5G mobile packet core product marketed by one of our industry partners.

Our prototype comprises a cluster of k switches, each facing a different Autonomous System (AS), providing access for the users in the AS to the web services hosted inside a data center. In particular, the switches translate the public Internet address of each service running inside the cloud to the internal private address of the VMs that run the corresponding workload (see Fig. 3). The access switches perceive different traffic intensities to the individual services and therefore need to cache different collections of translation rules; however, requests to certain popular services show up in large numbers at each of the access switches, allowing the cloud operator to take advantage of cooperative caching for these popular services.

Our prototype is built as a Ryu application, using Open vSwitch (OVS) as the switches and `mininet` as a network emulation tool. The request distribution of each switch was sampled according to the *equinix-chicago* packet trace from the CAIDA Anonymized Internet Traces 2014 Dataset [38] at different intervals of time (unfortunately, we cannot sample across different routers due to anonymization); the first $s = 2000$ most popular *(IP destination address, TCP destination port)* pairs were taken as the public service access points for the cloud-hosted services; for each such pair, a rule was set up to translate this public address to an internal address; and finally rule popularity at each switch was chosen according to the local popularity of the address-port pair as seen in the packet trace for the switch. The resultant flow tables and rule popularities were then implemented with local caching and with the best-response-time cooperative caching algorithm (Theorem VII.1); requests missing the cache are handled by the Ryu controller. We measured the one-way delay between two hosts directly attached to the switches using a home-grown delay measurement kit, which attains nanosecond precision leveraging the fact that clocks across `mininet` nodes are synchronized to the same CPU clock.

Measuring classification delays. To illustrate the distinct orders of magnitude of classification delays in SDNs, we carried out controlled experiments to assess the delay incurred by rule classification in a local switch, in the data plane (with cooperative switches) and in the control plane. Our experiments produced median delays in these three categories, respectively, of 3 ms, 4 ms and 200 ms (99-th percentiles 5.5 ms, 6.5 ms, and 370 ms, resp.). While the data plane measurements produce robust results over a wide choice of parameters (test sequence length, number of rules, etc.), the control plane measurements produced substantial variance, stemming most probably from the backlog that gradually builds up at the controller's ingress packet queue. Note that control plane delays can easily end up in the seconds range when the backlog grows large enough.

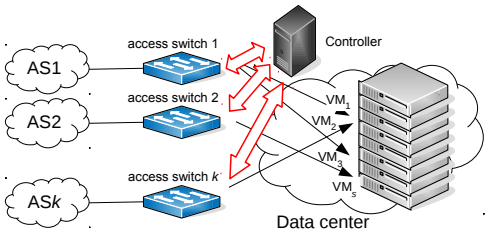


Figure 3. Cloud access gateway: k switches provide access for distinct ASes to s data center services.

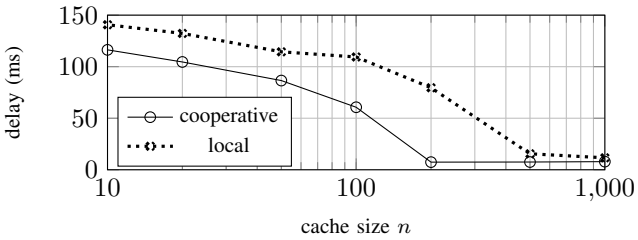


Figure 4. Delay vs. cache size. The knee of the cooperative (resp., local) curve occurs roughly at $n = 200$ (resp., $n = 500$), close to optimal offloading.

Latency vs. cache size. Fig. 4 shows the 99-th percentile one-way delay in the access gateway use case with different local cache sizes. We set $k = 2$ switches, both having 2000 rules, out of which 1096 are shared, accounting for 41% of the total ingress traffic. While cooperative caching maintains a comfortable two-times edge over local caching at basically all reasonable cache sizes, thanks to its more efficient utilization of data plane classification resources, we observe that the delay reduction can be an order of magnitude in certain cases. Strikingly, cooperative caching even when caching only 10% of the rules ($n = 200$) already reaches close to full data-plane classification (7 ms, 99-th percentile one way-delay).

IX. Conclusion

We presented models and algorithms for cooperative rule caching. Existing caching schemes either assume that caching is performed independently among caches, e.g., in SDNs, or do not account for object dependencies, e.g., in CDNs. To fill that gap, we propose novel rule caching solutions that take into account several kinds of dependencies as implied by various rule matching types. By leveraging spare resources at the fast data plane, we envision cooperative rule caching as a simple and effective approach to circumvent the limitations imposed by memory constrained devices, e.g., of IoT networks [1], [15]. This work paves the way towards that vision. As future work, we plan to evaluate the proposal beyond pairwise cooperation and in additional settings, including IoT networks.

Acknowledgment

The work was partially supported by the Technion Hiroshi Fujiwara Cyber Security Research Center and the Israel National Cyber Directorate, by the Alon fellowship, by German-Israeli Science Foundation (GIF) Young Scientists Program and by the Taub Family Foundation. Likewise, this work was partially sponsored by CAPES, CNPq and FAPERJ, through grants E-26/203.215/2017 and E-26/211.144/2019.

REFERENCES

[1] M. Kodialam, F. Hao, S. Mukherjee *et al.*, "CLAP: Compact labeling scheme for attribute-based iot policy control," in *IEEE International Conf. on Distributed Computing in Sensor Systems (DCOSS)*, 2019.

[2] P. Kortoçi, L. Zheng, C. Joe-Wong, M. Di Francesco, and M. Chiang, "Fog-based data offloading in urban iot scenarios," in *IEEE INFOCOM*, 2019.

[3] M. Uddin, S. Mukherjee, H. Chang, and T. Lakshman, "Sdn-based multi-protocol edge switching for iot service automation," *IEEE J. on Selected Areas in Comm.*, vol. 36, no. 12, pp. 2775–2786, 2018.

[4] H. Babbar and S. Rani, "Software-defined networking framework securing internet of things," in *Integration of WSN and IoT for Smart Cities*. Springer, 2020.

[5] A. G. A. Abd-Allah and A. Zaki, "Software-defned networks and security of iot," *IoT: Security and Privacy Paradigm*, p. 213, 2020.

[6] I. Alam, K. Sharif, F. Li, Z. Latif, M. Karim, S. Biswas, B. Nour, and Y. Wang, "A survey of network virtualization techniques for internet of things using sdn and nfv," *ACM Computing Surveys (CSUR)*, vol. 53, no. 2, pp. 1–40, 2020.

[7] H. Ballani, P. Francis, T. Cao, and J. Wang, "Making routers last longer with ViAggre," in *USENIX NSDI*, 2009.

[8] M. F. Bari, A. R. Roy, S. R. Chowdhury, Q. Zhang, M. F. Zhani, R. Ahmed, and R. Boutaba, "Dynamic controller provisioning in software defined networks," in *IEEE CNSM*, 2013.

[9] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine grained traffic engineering for data centers," in *ACM CoNEXT*, 2011.

[10] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker, "Rethinking enterprise network control," *IEEE/ACM Trans. Netw.*, vol. 17, no. 4, pp. 1270–1283, 2009.

[11] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," in *ACM SIGCOMM*, 2011.

[12] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," in *ACM SIGCOMM*, 2010.

[13] M.-F. Chang, C.-H. Chuang, Y.-N. Chiang, S.-S. Sheu, C.-C. Kuo, H.-Y. Cheng, J. Sampson, and M. J. Irwin, "Designs of emerging memory based non-volatile tcam for internet-of-things (iot) and big-data processing: A 5t2r universal cell," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2016.

[14] C. Wang, D. Zhang, L. Zeng, and W. Zhao, "Design of magnetic non-volatile tcam with priority-decision in memory technology for high speed, low power, and high reliability," *IEEE Transactions on Circuits and Systems I*, vol. 67, no. 2, pp. 464–474, 2019.

[15] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating systems for low-end devices in the internet of things: a survey," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 720–734, 2015.

[16] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Cacheflow: Dependency-aware rule caching for SDNs," in *ACM SOSR*, 2016.

[17] Y. Liu, S. O. Amin, and L. Wang, "Efficient FIB caching using minimal non-overlapping prefixes," *ACM SIGCOMM CCR*, vol. 43, no. 1, pp. 14–21, 2013.

[18] L. Molnár, G. Pongrácz, G. Enyedi, Z. L. Kis, L. Csikor, F. Juhász, A. Korösi, and G. Rétvári, "Dataplane specialization for high performance OpenFlow software switching," in *ACM SIGCOMM*, 2016.

[19] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of Open vSwitch," in *USENIX NSDI*, 2015.

[20] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson, "Cooperative caching: Using remote client memory to improve file system performance," in *USENIX OSDI*, 1994.

[21] K. Avrachenkov, J. Goseling, and B. Serbetci, "A low-complexity approach to distributed cooperative caching with geographic constraints," in *ACM SIGMETRICS*, 2017.

[22] A. Chattopadhyay and B. Błaszczyszyn, "Gibbsian on-line distributed content caching strategy for cellular networks," *arXiv:1610.02318*, 2016.

[23] N. Golrezaei, K. Shanmugam, A. G. Dimakis, A. F. Molisch, and G. Caire, "Femtocaching: Wireless video content delivery through distributed caching helpers," in *IEEE INFOCOM*, 2012.

[24] G. Paschos, E. Bastug, I. Land, G. Caire, and M. Debbah, "Wireless caching: Technical misconceptions and business barriers," *IEEE Comm. Magazine*, vol. 54, no. 8, pp. 16–22, 2016.

[25] L. Wang, H. Wu, Z. Han, P. Zhang, and H. V. Poor, "Multi-hop cooperative caching in social IoT using matching theory," *IEEE Trans. on Wireless Communications*, vol. 17, no. 4, pp. 2127–2145, 2017.

[26] M. Dehghan, B. Jiang, A. Seetharam, T. He, T. Salonidis, J. Kurose, D. Towsley, and R. Sitaraman, "On the complexity of optimal request routing and content caching in heterogeneous cache networks," *IEEE/ACM Transactions on Networking*, vol. 25, no. 3, pp. 1635–1648, 2017.

[27] E. Nygren, R. K. Sitaraman, and J. Sun, "The Akamai network: A platform for high-performance Internet applications," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 2–19, 2010.

[28] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras, "Inside Dropbox: Understanding personal cloud storage services," in *ACM IMC*, 2012.

[29] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, and P. Saab, "Scaling Memcache at Facebook," in *USENIX NSDI*, 2013.

[30] M. Taghizadeh, K. Micinski, S. Biswas, C. Ofria, and E. Torng, "Distributed cooperative caching in social wireless networks," *IEEE Transactions on Mobile Computing*, vol. 12, no. 6, pp. 1037–1053, 2013.

[31] J. Wu, Y. Chen, and H. Zheng, "Approximation algorithms for dependency-aware rule-caching in software-defined networks," in *IEEE GLOBECOM*, 2018.

[32] S. H. Hashemi, S. A. Noghabi, J. Bellessa, and R. H. Campbell, "Toward fabric: A middleware implementing high-level description languages on a fabric-like network," in *ACM/IEEE ANCS*, 2016.

[33] P. Nicholson, "The application of the in-tree knapsack problem to routing prefix caches," Master's thesis, University of Waterloo, 2009.

[34] R. the Network Operating System, "Ryu documentation, release 4.34," 2020, http://ryu.readthedocs.io/en/latest/index.html.

[35] A. Ghouila-Houri, "Caractérisation des matrices totalement unimodulaires," *Comptes Redus Hebdomadaires des Séances de l'Académie des Sciences (Paris)*, vol. 254, pp. 1192–1194, 1962.

[36] A. Tamir, "On totally unimodular matrices," *Networks*, vol. 6, no. 4, pp. 373–382, 1976.

[37] A. J. Hoffman and J. B. Kruskal, "Integral boundary points of convex polyhedra," in *50 Years of integer programming 1958-2008*. Springer, 2010, pp. 49–76.

[38] CAIDA, "The CAIDA UCSD anonymized Internet traces," 2014, http://www.caida.org/data/passive/passive_2014_dataset.xml.