Data Plane Cooperative Caching with Dependencies

Article *in* IEEE Transactions on Network and Service Management · December 2021

DOI: 10.1109/TNSM.2021.3132275

Ori Rottenstreich
Technion - Israel Institute of Technology
116 PUBLICATIONS **1,500** CITATIONS
SEE PROFILE

Ariel Kulik
CISPA
25 PUBLICATIONS **312** CITATIONS
SEE PROFILE

Jennifer Rexford
Princeton University
440 PUBLICATIONS **37,178** CITATIONS
SEE PROFILE

Daniel S. Menasché
Federal University of Rio de Janeiro
149 PUBLICATIONS **1,307** CITATIONS
SEE PROFILE

Data Plane Cooperative Caching with Dependencies

Ori Rottenstreich, Ariel Kulik, Ananya Joshi, Jennifer Rexford, Gábor Rétvári and Daniel S. Menasché

***Abstract*—Caching is at the core of most modern communication systems, where caches are used to store content and traffic classification rules. While network components can leverage caching in a cooperative manner, one important aspect of such systems concerns possible dependencies among stored items. A major use case of such dependencies appears in rule placement across software-defined networks (SDNs).**

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. 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 accounting for dependencies among rules implied by existing switch memory types, and *lay the foundations of cooperative caching with dependencies*.

***Index Terms*—Cooperative caching, Software defined networking**

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 [2]–[7]. However, the adoption of SDNs outside of data-center networks still faces a fundamental problem: rule tables require fast and power-hungry memories [8]–[13], but TCAMs must be of limited size in cheap and energy constrained devices [14]–[16]. To guarantee classification correctness a switch cannot simply "cache" the most popular rules due to rule *dependencies* [17]–[19]. 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 [17], [18], [20].

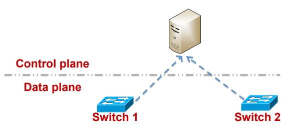
When a packet arrives at a switch and does not match any of its cached rules, it is common practice to assume that the *default rule* is to forward the packet to the control plane (or to a slower data path, like in Open vSwitch [21], [22]). A major challenge consists in coping with the control-plane 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 challenge can be met if multiple switches can cooperatively work together to achieve a high overall hit rate, benefiting from the data plane performance.

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. *We aim to lay the foundations of cooperative caching with dependencies.*

Prior art. *Cooperative caching* [23] 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 [24]–[26] and IoT networks [27], CDNs [28]–[30], social networks [31], [32], and distributed operating systems [23]. 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 [8], [12], [13]. Bauer and Zitterbart [33] suggested moving rules from a loaded switch to a switch with spare capacity. Such rules can refer to traffic processing as well as for flow monitoring but has not considered potential dependencies among rules implied by rule aggregation.

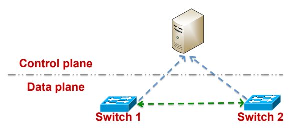
Recently, additional works tried to extend data plane functionality and deal with the scarcity of flow tables in various forms: TableVisor [34] describes a proxy layer between the SDN controller and hardware. It supports data plane device aggregation to reduce overhead for heterogeneous SDN switches with various types of flow tables and capabilities.

This manuscript is an extended version of a paper presented at IEEE CloudNet 2020 [1]. Ori Rottenstreich is with the Technion - Israel Institute of Technology, Israel. Ariel Kulik is with CISPA Helmholtz Center for Information Security, Germany. Ananya Joshi is with Carnegie Mellon University, PA. Jennifer Rexford is with Princeton University, NJ. Gábor Rétvári is with Budapest University of Technology and Economics, Hungary. Daniel Menasché is with Federal University of Rio de Janeiro, Brazil.



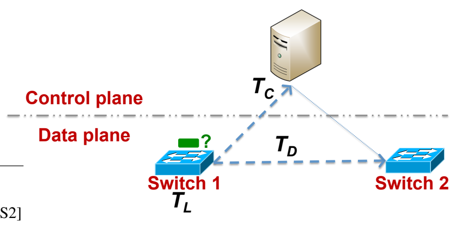
S1 Rule Table:	S2 Rule Table:
$R_1 : (000^*) \rightarrow a_1$ (0.18)	$R_7 : (00^{**}) \rightarrow a_7$ (0.20)
$R_3 : (010^*) \rightarrow a_3$ (0.16)	$R_5 : (10^{**}) \rightarrow a_5$ (0.32)
$R_6 : (11^{**}) \rightarrow a_6$ (0.45)	$R_6 : (11^{**}) \rightarrow a_6$ (0.23)
$R_2 : (001^*) \rightarrow a_2$ (0.04)	$R_8 : (010^*) \rightarrow a_8$ (0.07)
$R_4 : (011^*) \rightarrow a_4$ (0.02)	$R_9 : (011^*) \rightarrow a_9$ (0.18)
$R_5 : (10^{**}) \rightarrow a_5$ (0.15)	

(a) Traditional rule caching



S1 Rule Table:	S2 Rule Table:
$R_1 : (000^*) \rightarrow a_1$ (0.18) [S1]	$R_7 : (00^{**}) \rightarrow a_7$ (0.20) [S2]
$R_3 : (010^*) \rightarrow a_3$ (0.16) [S1]	$R_9 : (011^*) \rightarrow a_9$ (0.18) [S2]
$R_6 : (11^{**}) \rightarrow a_6$ (0.45+0.23) [S1,S2]	$R_5 : (10^{**}) \rightarrow a_5$ (0.32+0.15) [S1,S2]

(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.

Similarly, Maple++ is a framework for encoding policies over heterogeneous switches in a pipeline-based tables [35]. This provides a programming API to network designers generating small rules tables with the support policy updates.

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-VII).

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

II. System description and results

Rule caching [17], [18], [36] allows maintaining a large set of rules within an hierarchy of two memory levels: A subset of the rules, typically those accessed frequently, are maintained in a fast but small memory while remaining rules are kept in a slower larger memory level. For a particular traffic, the classification is completed within a short time if a matching rule can be found in the small memory. Otherwise, the classification takes longer time and requires accessing the slower memory.

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.

Under cooperative rule caching, we assume that each switch locally caches rules, e.g., in its TCAM, indicating to which switch they apply to. In addition, each switch has a set of neighbors, to whom it will forward packets in case they need help to complete their classification. We also assume that all rules are stored by the controller that can ultimately classify any packet if needed.

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. The pairwise matching of switches may leverage the overlap coefficient or the Jaccard index between rule sets, aiming to pair switches based on number of common rules. 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

Table I
NEW CACHING POLICY RESULTS (IN BOXED BOLD)

		# Switches		
		single	two	multiple (≥ 3)
Rule Dependencies	Exact match (no rule dependencies)	Optimal	$\boxed{\textbf{Optimal}}$	$\boxed{\textbf{Optimal}}$
	Prefix match	Optimal [36], [37]	$\boxed{\textbf{Optimal}}$	$\boxed{\textbf{Heuristics}}$
	Wildcard match (NP-hard)	Heuristics [17], [18]	$\boxed{\textbf{Heuristics}}$	

Table II
TABLE OF NOTATION

Variable	Description
k	number of switches
S	set of rules
S_j	ordered set of rules of switch j
R_r	r-th rule, $r = 1, \dots, \vert S\vert$
$R_{j,\ell}$	ℓ-th rule at switch j
n_j	size of cache of switch j
C_j	set of rules cached at switch j
$\lambda_{j,r}$	popularity of rule R_r in switch j
M_R^i	set of origin switches to which rule R stored in i applies to, $M_R^i \subseteq \{j \vert R \in (S_j \cap C_i)\}$
T_L	delay of local classification
T_D	delay of data plane non-local classification
T_C	delay of control plane classification

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).

Implementation details. Note that the headers of packets considered in the above solution must contain three fields: TTL, origin switch and matched action. Under pairwise cooperation, *TTL* is a binary field. Its value is set to one to indicate that the packet can still be forwarded in the data plane, and zero otherwise. The *origin switch* field stores the switch which first handled the packet (Definition 1) and is used by a cooperative neighbor to return the *matched action* of the matched rule, in case the neighbor is able to find a rule that matches the packet. Otherwise, if TTL equals zero and there are no matching rules, the packet is forwarded to the control plane.

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 multiple 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 [17], [18], motivating a greedy heuristic for cooperative switches introduced in Section VI.

III. MODELING COOPERATIVE CACHING

We consider a network of k switches $1, \dots, k$. Each switch i has an ordered set S_i of $|S_i|$ rules $(R_{i,1}, \cdots, R_{i,|S_i|})$. 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 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, namely, rules are ordered in decreasing priority. 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. Notation is summarized in Table II.

Rules and popularities. Let $S = \{R_1, \dots, R_{|S|}\}$ be the set of distinct rules that appear in one or more switches 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 [38], [39]. Without loss of generality, we do not consider rules that are never matched in any of the switches. For $i \in [1, k]$, let $\lambda_{i,r}$ be the popularity of rule R_r in switch i, describing the amount of traffic that matches the rule. In particular, if a rule R_r appears only in switch 1, we have $\lambda_{i,r} = 0$ for $i \neq 1$.

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. Assume 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 case of a miss in a switch cache, the classification time in all other switches is identical. As illustrated in Fig. 1(c), we refer to these three classifications as *L*ocal, in the *C*ontrol plane and by cooperation among switches in the *D*ata plane. We denote their time by T_L, T_C and T_D, respectively, such that $T_L \leq T_D < T_C$. Typically accessing the data plane occurs after a miss within the local cache such that the data plane total classification time takes $T_L + T_D$. Throughout this work, we assume that T_L, T_C and T_D are constant, and that T_D includes the time required for sending the packet to other switch, processing it and sending it back to the origin switch. *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 [40] 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_r in both switch 1 and switch 2, $R_r \in C_1 \cap C_2$, is $\lambda_{1,r} + \lambda_{2,r}$, 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_r only in switch 1, $R_r \in C_1 \setminus C_2$, is $\lambda_{1,r} + \alpha \cdot \lambda_{2,r}$, where $\alpha \in (0, 1]$, as above, is determined by the delays of the various classification options. Symmetrically, a rule R_r cached only in switch 2, $R_r \in C_2 \setminus C_1$, contributes $\alpha \cdot \lambda_{1,r} + \lambda_{2,r}$. There is no contribution for rules not cached in any switch.

Rule dependencies and correctness. To guarantee the correctness of a rule caching, we must ensure that if a rule is the first to match a packet among a subset of cached rules, the same rule is the first to match that packet in the complete set of rules. Consider a switch $i \in [1, k]$. We say that rule $R \in S_i$ depends on a rule $R' \in S_i$ if R' has higher priority than R and their sets of packets intersect. For correctness, caching R on switch $j \in [1, k]$ and marking it applicable for origin switch i, requires R' to be cached in either switch i or j, and marked as applicable for origin switch i. In particular, the condition must also hold when $i = j$, entailing that caching R at a switch i for which rule R applies implies also caching R' at i.

Formally, consider a caching $C_1, \ldots, C_k$ with markings M_R^i for rule $R \in C_i$ cached in switch $i \in [1, k]$.

Definition 5 (Correctness requirement)**.** *A rule placement is* correct *if for any switch $i \in [1, k]$, and any two rules $R, R' \in S_j$ originated in switch i such that R depends on R', the following condition is satisfied: if $R \in C_j$ for some $j \in [1, k]$ with $i \in M_R^j$ then at least one of the following holds*

- *$R' \in C_i$ and $i \in M_{R'}^i$*
- *$R' \in C_j$ and $i \in M_{R'}^j$.*

Note that in the above definition we allow $j = i$, i.e., if $R \in C_i$ with $i \in M_R^i$ then the correctness requirement implies $R' \in C_i$ with $i \in M_{R'}^i$ – if a rule is cached in a switch for which it applies, all the rules that have higher priority must also be stored at that switch.

IV. Exact Rule Matching

We first consider the simplest setup of *exact match* wherein rules have no dependencies (Definition 2). Every packet matches at most one rule. We present two solutions for computing the optimal rule caching for exact match. The first uses a linear programming relaxation and applies for an arbitrary number of switches.[1] The second refers to a pair of switches and is based on maximum graph matching, allowing us to derive structural properties of the optimal solution.

A. A Linear Programming Approach.

For an arbitrary number of switches, let $G(\boldsymbol{x})$ be the average gain in cost reduction when caching $\boldsymbol{x} = [x_{i,r}]$ is deployed, where $x_{i,r} = 1$ if R_r is stored in cache C_i ($R_r \in C_i$ and $M_{R_r}^i = \{j | R_r \in (S_j \cap C_i)\}$), and $x_{i,r} = 0$ otherwise. For each rule R_r, if R_r is stored in at least one cache, the gain is at least $\alpha \sum_i \lambda_{i,r}$. Each cache storing rule R_r experiences an additional gain of $(1-\alpha)\lambda_{i,r}$. Denote by x_r an indicator variable, equal to 1 if rule R_r is stored in at least one cache, and 0 otherwise. We pose the following mixed integer linear program (MILP),

$$\max\ G(\boldsymbol{x}) = \sum_{r \in [1, |S|]} \left(\alpha \cdot x_r \sum_{i=1}^{k} \lambda_{i,r} + (1-\alpha) \sum_{i=1}^{k} x_{i,r} \cdot \lambda_{i,r} \right) \tag{1}$$

[1]Note that linear programs are considered to have algorithms which are only weakly polynomial-time but lack solutions which are strongly polynomial [41]. In practice, there is an ongoing research on bounding the degree of the running time complexity polynomial [42]. Our second approach, based on maximum graph matching, guarantees a strongly polynomial running time.

where

$$\sum_{r\in[1,|S|]} x_{i,r} \le n_i \quad \forall i \in [1,k], \tag{2}$$

$$x_r \le \sum_{i\in[1,k]} x_{i,r} \quad \forall r \in [1,|S|], \tag{3}$$

$$x_r \in \{0,1\} \quad \forall r \in [1,|S|], \tag{4}$$

$$x_{i,r} \in \{0,1\} \quad \forall i \in [1,k], \forall r \in [1,|S|]. \tag{5}$$

When considering pairwise matchings of switches, we let $k=2$ in the above MILP and treat each pair of switches independently. Constraints (2)-(3) correspond to cache capacities and to the definition of x_r. Next, we present the main result of this section, whose proof relies on [28], [43], [44].

Theorem IV.1. *An optimal cooperative caching with exact rule matching can be found in (weakly) polynomial time, as the problem* (1)-(3) *without constraints* (4)-(5) *admits an integral solution.*

Proof. The proof of the above theorem relies on the relaxed version of the MILP wherein constraints (4)-(5) are replaced by the corresponding real constraints $0 \le x_r \le 1$ and $0 \le x_{i,r} \le 1$, respectively. As the objective function is convex, the solution to the relaxed problem can be found in polynomial time. By showing that the relaxed problem admits an integral solution, we conclude that the solution to the relaxed problem is also a solution to the original MILP problem.

The constraints of the relaxed problem can be written in the form $\boldsymbol{Az} \le \boldsymbol{b}$ where matrix $\boldsymbol{A}$ and vector $\boldsymbol{b}$ are all integers and $\boldsymbol{z}$ is a vector of $x_{i,r}$ and x_r. That is since the standard form LP: max $\boldsymbol{cz}$ s.t. $\boldsymbol{Az} \le \boldsymbol{b}$, $\boldsymbol{z} \ge 0$, with integral matrix $\boldsymbol{A}$, integral vector $\boldsymbol{b}$ and an arbitrary vector $\boldsymbol{c}$, has an integral optimal solution $\boldsymbol{z}$ if its constraint matrix $\boldsymbol{A}$ is totally unimodular [28], [43], [44]. By the Hoffman sufficient condition (HSC) [45], matrix $\boldsymbol{A}$ is totally unimodular if it contains no more than one +1 and no more than one -1 in each column.

Proposition 1 (Hoffman Sufficient Condition)**.** *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. □

Example. We illustrate below a matrix $\boldsymbol{A}$ accounting for constraints (2) (first two rows in the matrix) and (3) (last three rows), for a system with $k=2$ caches and $|S|=3$ rules. Clearly, each column contains exactly one element +1 and no more than one -1, satisfying the conditions of Proposition 1.

$$\boldsymbol{A} = \begin{array}{c} \begin{array}{ccccccccc} x_{1,1} & x_{1,2} & x_{1,3} & x_{2,1} & x_{2,2} & x_{2,3} & x_1 & x_2 & x_3 \end{array} \\ \left(\begin{array}{cccccc|ccc} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ \hline -1 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 1 \end{array}\right) \end{array}$$

B. A Graph Maximum Matching Approach.

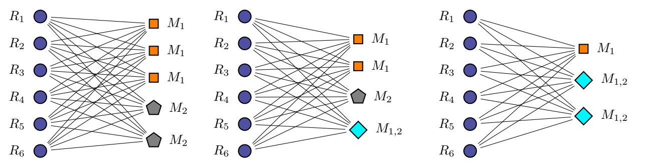
We present another approach that applies for a pair of switches. Recall that the gain of a caching policy is its reductions in the classification time, in comparison with a scheme without caching. The gain achieved by a caching policy is given by the sum of the gains for each of the rules. However, note that with any level of cooperation (the model assumes $\alpha > 0$), the contribution of caching the same rule in two switches is smaller than the sum of contributions of caching it in each of the switches. Intuitively, having two copies prevents taking advantage of the ability for mutual help among switches. That is the total value of the cached rules is sublinear in the contributions for the various switches.

To represent the cooperative caching constraints, we rely on the construction of multiple bipartite graphs. We divide the unique cached rules (in at least one of the two switches) of a possible solution into *three* memory categories. The categories are M_1 - rules cached only in switch 1, M_2 - rules cached only in switch 2, and $M_{1,2}$ - rules cached in both switches. Let n_c denote the number of shared rules in a solution. Such rules occupy space in both switches. For a known value of $n_c \in [0, \min(n_1, n_2)]$, we build a bipartite graph $G_{n_c} = (U, V, E)$ where the left nodes of U represent the $|S|$ distinct rules of S. The right nodes in V are composed of three types, describing the three memory categories of rules M_1, M_2 and $M_{1,2}$. Accordingly, the rules cached in switch 1 are $C_1 = M_1 \cup M_{1,2}$ and those cached in switch 2 are $C_2 = M_2 \cup M_{1,2}$. The number of nodes in the categories are $n_1 - n_c, n_2 - n_c, n_c$, respectively. The total number of nodes in V is $n_1 + n_2 - n_c$ that stands for the number of distinct cached rules. An edge between R_r and M_j (resp., $M_{1,2}$) indicates that R_r should be placed in switch j, for $j = 1, 2$ (resp., in both switches 1 and 2). A matching, i.e., a set of edges without common vertices, fully characterizes how rules should be cached across switches. We consider a fully connected bipartite graph, and assign the weight of an edge between a rule R_r and a memory category node $v \in V$ as the value of caching R_r in switches according to the category of v. If v is of category M_1 (resp., M_2) the weight of edge (R_r, v) is $\lambda_{1,r} + \alpha \cdot \lambda_{2,r}$ (resp., $\alpha \cdot \lambda_{1,r} + \lambda_{2,r}$). If v is of category $M_{1,2}$, the weight of edge (R_r, v) is $\lambda_{1,r} + \lambda_{2,r}$.

The following lemma simply follows by the linearity of the value of rule caching solutions with respect to the contributions of different rules. Intuitively, the total gain is given by the sum of contributions of the different rules.

Lemma 1. *Let Λ_{n_c} be the weight of a maximal weight matching in the bipartite graph G_{n_c}. A corresponding caching policy achieves a cost reduction of Λ_{n_c} in classification time.*

The number n_c of rules cached in both switches can be flexible. In the most general case the range for that number is $n_c \in [0, \min(n_1, n_2)]$. Of course, n_c cannot exceed the number of common rules between the switches. These options are represented by the different bipartite graph instances. We deduce that the optimal caching policy is described by a maximal weight matching for one of the instances, namely the the instance whose maximal matching has the highest weight.



(a) no shared rules (b) one shared rule (c) two shared rules

Figure 2. *Exact rule matching (no rule dependencies) for two switches - Graph matching approach*: Illustration of the complete bipartite graph with $|S| = 6$ (distinct) rules and switch capacities $n_1 = 3, n_2 = 2$.

Theorem IV.2. *An optimal caching policy is described by the matching in one of the bipartite graphs G_{n_c} that achieves the highest weight.*

To find an optimal caching policy we construct the bipartite G_{n_c} for the different values of n_c. We find the maximal weight matching in each instance and select the caching according to the one with the highest weight over the various n_c values. Finding a maximal weight matching in a bipartite graph can be done in polynomial time of $O(n^2 \cdot \log(n))$ for a graph with n nodes (see e.g., [46]).

The construction is illustrated in Fig. 2. Here, for the case of $|S| = 6$ distinct rules and switch capacities of $n_1 = 3, n_2 = 2$. Three bipartite instances are considered (in (a), (b) and (c)) with number of shared rules $n_c \in [0, 2]$. The number of right nodes in a bipartite instance is $n_1 + n_2 - n_c \in [3, 5]$. The three memory categories $M_1, M_2, M_{1,2}$ are illustrated as squares, pentagons and diamonds, respectively.

Time complexity. The algorithm builds the bipartite graph instance and runs the bipartite weighted matching algorithm $\min(n_1, n_2) + 1$ times. The graph has $|S| \leq |S_1| + |S_2|$ left nodes and at most $n_1 + n_2 \leq 2 \cdot |S|$ right nodes. Thus the total time complexity is $O(\min(n_1, n_2) \cdot |S|^2 \cdot \log(|S|))$.

We derive properties characterizing an optimal caching policy. Intuitively, in such an optimal caching, there are no changes in the rules cached in one or more of the switches that would further increase the gain (i.e., reduce the classification cost).

Theorem IV.3. *An optimal caching policy satisfies* (i) *For a rule R_r cached according to one of the three categories M_1, M_2, $M_{1,2}$, the value in that category is not smaller than the value in the same category of any rule not cached in any of the three categories.* (ii) *For a rule $R_r \in M_1$, the value $(1 - \alpha) \cdot \lambda_{2,r}$ is not larger than the value $\alpha \cdot \lambda_{1,r'} + \lambda_{2,r'}$ for any rule $R_{r'} \in M_2$ (and vice versa).* (iii) *For $\alpha < 1$, there might exist a rule $R_r \in M_1$ with $\lambda_{1,r} = 0$ or a rule $R_{r'} \in M_2$ with $\lambda_{2,r'} = 0$, but there cannot be such rules simultaneously in both switches.*

Proof. We show by contradiction that if one property does not hold, we can make changes in an optimal caching to further reduce the cost. In (i), we can replace a rule in one of the three categories by a rule that is not cached at all and has a larger value for the specific category. In (ii), for a rule cached in switch 1, we can also cache it in addition in switch 2 instead of a rule cached in this switch. In (iii), we can switch between two rules, each one cached in one of the switches, to achieve gain of $(1 - \alpha)$ of their corresponding rule popularities. Note that an optimal rule caching solution can include for instance in switch 1 a rule R_r with $\lambda_{1,r} = 0$ when its contribution $\alpha \cdot \lambda_{2,r}$ is large enough in terms of rules cached in switch 1 but is relatively low in terms of rules cached in switch 2. □

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. 3 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-A. Indeed, prefix rules correspond to constraints of the form $x_{i,r} \leq x_{i,r'}$ for every rule r which is a parent of r' in the prefix tree in switch i. 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.

A. An overview on the dynamic programming approach

We propose a dynamic programming algorithm to overcome the aforementioned challenges. The algorithm facilitates ideas from the algorithm of [37] for dependent caching on a single switch. In particular, our algorithm overcomes the difficulties introduced by cooperative caching over multiple switches. We define a sub-problem for every combination of prefix P, two cache sizes m_1, m_2 such that $m_1 \leq n_1$ 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 caching gain. The optimal cooperative caching is an optimal caching for $(P_{root}, n_1, n_2, \emptyset)$, where P_{root} 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

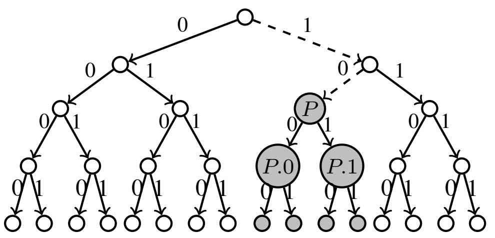


Figure 3. Prefix rule dependency. Caching a prefix $P = 10**$ requires caching all prefixes in the colored subtree such as $100*$, $101*$ (if exists).

descendants of P in the prefix tree (see Fig. 3). 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.

A major insight consists in noting that given solutions to the two sub-problems, it suffices to determine whether prefix P should be added or not to the switches. In particular, by properly selecting values for m'_i and m''_i we ensure that P can be cached on top of the solutions for the sub-problems. Hence, evictions are unnecessary.

B. Algorithm details

For simplicity, in this section we assume that if $(P, a_1) \in S$ and $(P, a_2) \in S$ then $a_1 = a_2$. That is, the same prefix is only associated with a single action for all the switches. Then, we can refer to rules as prefixes (though, not all prefixes are rules). In particular, we write $P \in S$ (resp., $P \in C$) when there is an action a such that $(P, a) \in S$ (resp., $(P, a) \in C$). Our algorithm can be easily extended to the general case without this requirement.

We establish a partial ordering among prefixes, such that $P \leq P'$ when a prefix P shares all its bits with a longer or equal prefix P'. We also write $P < P'$ when $P \leq P'$ and $P \neq P'$. We say that P covers (resp., strictly covers) P' when $P \leq P'$ (resp., $P < P'$). We denote by S_P (resp., $\hat{S}_P$) the set of prefix rules covered by P, i.e., the set of prefixes of rules belonging to the subtree rooted by P, $S_P = \{P' \in S | P \leq P'\}$ (resp., strictly covered by P, $\hat{S}_P = \{P' \in S | P < P'\}$).

Recall that S_j is the set of input rules applicable to switch $j \in \{1, 2\}$. We denote by S_P^j the set of rules applicable to switch j, covered by P, $S_P^j = \{P' \in S_j | P \leq P'\}$.

Next, we formally define the mentioned compliance requirements $\mathcal{Q}$. We use the common cartesian set product notation $\{1, 2\} \times \{1, 2\}$ representing the set $\{(1, 1), (1, 2), (2, 1), (2, 2)\}$. Given a solution to the caching problem, a set T of prefixes and $(i, j) \in \{1, 2\} \times \{1, 2\}$, we say the caching is *(i, j)-compliant* with respect to T if for every prefix $P \in T \cap S_i$, either

- $P \in C_i$ and $i \in M_P^i$ or
- $P \in C_j$ and $i \in M_P^j$.

For a given rule $P \in S_i$, if a solution is (i, j)-compliant with respect to $\hat{S}_P$ it is possible to insert P to C_j and add i to M_P^j (i.e., mark P as effectively applicable to origin switch i) while preserving the correctness requirement. Indeed, the above two bullets correspond to the two bullets in the definition of the correctness requirement (Definition 5).

A *compliance requirement* is a set $\mathcal{Q} \subseteq \{1, 2\} \times \{1, 2\}$. It holds $|\mathcal{Q}| \in [0, 4]$. Given a set of prefixes T, we say a solution to the caching problem is *$\mathcal{Q}$-compliant* with respect to T if for every $(i, j) \in \mathcal{Q}$ the solution is (i, j)-compliant with respect to T. For example, for $\mathcal{Q} = \{(1, 1), (1, 2)\}$, a solution is $\mathcal{Q}$-compliant with respect to T if it is both $(1, 1)$ and $(1, 2)$-compliant with respect to T.

We use the notation 2^S for the power set of a set S, namely the set of all subsets of S. For instance $2^{\{1,2\}\times\{1,2\}}$ includes 16 elements. Likewise, let $\mathcal{P}$ be the set of all prefixes (shown as nodes in the binary prefix tree, not necessarily in S).

We now characterize the caching gains. We define $\Lambda(P, m_1, m_2, \mathcal{Q})$ as a function

$$\Lambda : \mathcal{P} \times [0, n_1] \times [0, n_2] \times 2^{\{1,2\}\times\{1,2\}} \to \mathbb{R} \cup \{-\infty\}.$$

The value of $\Lambda(P, m_1, m_2, \mathcal{Q})$ is the highest achievable gain from a correct caching of prefixes from S_P which is $\mathcal{Q}$-compliant with respect to S_P and in which $|C_i| \leq m_i$ for $i = 1, 2$. If no such caching exists let $\Lambda(P, m_1, m_2, \mathcal{Q}) = -\infty$.

Recall P_{root} is the root of the prefix tree. By definition, $\Lambda(P_{root}, n_1, n_2, \emptyset)$ is the highest gain achievable by a correct caching of rules from the prefix tree, with up to n_1 rules in switch 1 and n_2 rules in switch 2 (without particular compliance requirements). We also refer to $\Lambda(P_{root}, n_1, n_2, \emptyset)$ as the *value* of the optimal caching. Therefore, our objective is to compute $\Lambda(P_{root}, n_1, n_2, \emptyset)$ and a caching that attains this gain.

Our algorithm utilizes an additional auxiliary function

$$\Gamma : \mathcal{P} \times [0, n_1] \times [0, n_2] \times 2^{\{1,2\}\times\{1,2\}} \to \mathbb{R} \cup \{-\infty\},$$

defined similarly to $\Lambda(P, m_1, m_2, \mathcal{Q})$, with a distinction that Γ refers to $\hat{S}_P$ rather than S_P, namely it cannot cache the prefix P itself. It describes the highest achievable gain from a correct caching of prefixes from $\hat{S}_P$ which is $\mathcal{Q}$-compliant with respect to $\hat{S}_P$ and in which $|C_i| \leq m_i$ for $i = 1, 2$. Again, $\Gamma(P, n_1, n_2, \emptyset) = -\infty$ if no such caching exists.

Next, we focus on the computation of the values of Λ and Γ. The optimal caching itself can be recovered using standard backtracking techniques.

We start with Γ. For a prefix P such that $\hat{S}_P = \emptyset$ by definition $\Gamma(P, m_1, m_2, \mathcal{Q}) = 0$, for all $\mathcal{Q}$, m_1, m_2 in the domain of Γ. For a prefix P such that $\hat{S}_P \neq \emptyset$, let $P.0$ and $P.1$ be its children. A solution for $\Gamma(P, m_1, m_2, \mathcal{Q})$ is comprised of sets C_1, C_2 and $M_{P'}^i$ for $i = 1, 2$ and $P' \in C_i$. This solution can be split into two parts, a solution with the prefixes in $S_{P.0}$ and another with the prefixes in $S_{P.1}$. That is, for $d \in \{0, 1\}$, the solution for $P.d$ is given by sets $C_1^d = C_1 \cap S_{P.d}$ and $C_2^d = C_2 \cap S_{P.d}$ and sets $M_{P'}^i$ for $P' \in C_i^d$. It holds that $C_i = C_i^0 \cup C_i^1$. As the solution for P is correct and $\mathcal{Q}$-compliant with respect to $\hat{S}_P$ then necessarily the solutions for $P.0$ and $P.1$ are also correct and $\mathcal{Q}$-compliant with respect to $S_{P.0}$ and $S_{P.1}$, respectively. The gain of the solution for P equals the sum of gains for the two solutions for $P.0$ and $P.1$. That is, a solution with m_i rules in C_i for P can be seen as the union of two solutions with m'_i and $m_i - m'_i$ for $P.0$ and $P.1$.

Similarly, a solution for $\Lambda(P, m_1, m_2, \mathcal{Q})$ can be constructed from solutions for $\Gamma(P, m_1', m_2', \mathcal{Q})$ and $\Gamma(P, m_1 - m_1', m_2 - m_2', \mathcal{Q})$ for any $0 \leq m_1' \leq m_1$, $0 \leq m_2' \leq m_2$.

By the above arguments we obtain the following formula.

$$\Gamma(P, m_1, m_2, \mathcal{Q}) = \max_{m_1' \in [0,m_1], m_2' \in [0,m_2]} \Big(\Lambda(P.0, m_1', m_2', \mathcal{Q}) + \Lambda(P.1, m_1 - m_1', m_2 - m_2', \mathcal{Q}) \Big). \tag{6}$$

Next, we describe how to evaluate $\Lambda(P, m_1, m_2, \mathcal{Q})$ given $\Gamma(P, m_1', m_2', \mathcal{Q}')$. The difference between the caching implied by Λ and by Γ refers to the potential caching of P. Clearly, if $P \notin S$, that is, P is not a rule, then

$$\Lambda(P, m_1, m_2, \mathcal{Q}) = \Gamma(P, m_1, m_2, \mathcal{Q}). \tag{7}$$

Consider the case where $P \in S$. We characterize the caching decision for P using a set $D_P \in 2^{\{i|P \in S_i\} \times \{1,2\}}$, where $(i,j) \in D_P$ indicates that P is cached at switch j and is marked as applicable to i, i.e., $(i,j) \in D_P$ if $P \in C_j$ and $i \in M_P^j$. In what follows, prefix P should be clear from context, and we refer to D_P simply as D.

Let $V_P(D)$ be the gain from caching P according to D. For each switch i, there is a gain of $\alpha \lambda_{i,P}$ if P is cached at any of the switches and marked as effectively applicable to i. If the rule is also cached at switch i and marked as effective for switch i, there is an additional gain of $(1-\alpha)\lambda_{i,P}$. Then,

$$V_P(D) = \sum_{i \in \{1,2\} \ | \ \exists j \in \{1,2\}: \ (i,j) \in D} \alpha \cdot \lambda_{i,P} + \sum_{i \in \{1,2\} \ | \ (i,i) \in D} (1-\alpha)\lambda_{i,P}.$$

For P, let $\mathcal{Q} \subseteq \{1,2\} \times \{1,2\}$ be a given compliance requirement. If $P \in S_i$ and $(i,j) \in \mathcal{Q}$ we must cache P in either i or j and mark it as applicable for i in order to maintain a caching $\mathcal{Q}$-compliant with respect to S_P. Therefore, we say that $D \in 2^{\{i|P \in S_i\} \times \{1,2\}}$ is *P-consistent* with $\mathcal{Q}$ if for every $(i,j) \in \mathcal{Q} \cap 2^{\{i|P \in S_i\} \times \{1,2\}}$ we have

- $(i,i) \in D$ or
- $(i,j) \in D$.

The above two conditions correspond to the two bullets in the definition of the correctness requirement (Definition 5).

Denote

$$\Phi_P(\mathcal{Q}) = \left\{ D \in 2^{\{i|P \in S_i\} \times \{1,2\}} \ \middle| \ D \text{ is } P\text{-consistent with } \mathcal{Q} \right\}.$$

Informally, $\Phi_P(\mathcal{Q})$ describes all possibilities to cache P (in either none, one or two of the switches) that would allow $\mathcal{Q}$-compliance with respect to S_P.

Let $\mu_j(D)$ be an indicator variable denoting whether rule P is cached in switch j under allocation D.

$$\mu_j(D) = \begin{cases} 1, & (1,j) \in D \text{ or } (2,j) \in D \\ 0, & \text{otherwise.} \end{cases}$$

The gain $\Lambda(P, m_1, m_2, \mathcal{Q})$ is attained by a caching solution, C_1, C_2 and subsets $M_{P'}^i$ for $P' \in C_i$ and $i = 1,2$, which is $\mathcal{Q}$-compliant with respect to S_P. Define $D' = \{(i,j) \mid P \in C_j, i \in M_P^j\}$. As the solution is $\mathcal{Q}$-compliant then $D' \in \Phi_P(\mathcal{Q})$. If we remove the rule P from this caching solution, we get a caching solution for $\hat{S}_P$ which is both $\mathcal{Q}$-compliant and D'-compliant with respect to $\hat{S}_P$.

Likewise, an opposite argument can be used to construct a caching for $\Lambda(P, m_1, m_2, \mathcal{Q})$ for any $D \in \Phi_P(\mathcal{Q})$ from a caching for $\Gamma(P, m_1 - \mu_1(D), m_2 - \mu_2(D), \mathcal{Q} \cup D)$, by adding a caching for P as prescribed by D.

This allows us to derive the recursive formula

$$\Lambda(P, m_1, m_2, , \mathcal{Q}) = \max_{D \in \Phi_P(\mathcal{Q})} \Gamma(P, m_1 - \mu_1(D), m_2 - \mu_2(D), \mathcal{Q} \cup D) + V_P(D). \tag{8}$$

In the formula, the maximum operator applied over an empty set equals $-\infty$.

Theorem V.1. *An optimal cooperative caching with prefix matching for up to two switches can be found in polynomial time.*

Proof. Equations (6), (7) and (8) derived in this section can be used to evaluate $\Gamma(P, m_1, m_2, \mathcal{Q})$ and $\Lambda(P, m_1, m_2, \mathcal{Q})$ over all the prefixes $P \in \mathcal{P}$, with $S_P \neq \emptyset$, in a bottom-up fashion. Therefore, we get $\Lambda(P_{root}, n_1, n_2, \emptyset)$ as required. For each action corresponding to a prefix P of length up to W, we evaluate a polynomial number of nodes in the prefix tree, as only nodes along the path from prefixes to the root must be traversed, and their number is linear in W and $|S|$. $\square$

In a possible implementation, the algorithm iterates over the prefixes in $\mathcal{P}$ according to their lengths from the longest to the shortest. For each prefix P it first evaluates $\Gamma(P, m_1, m_2, Q)$ for every $(m_1, m_2, Q) \in [0, n_1] \times [0, n_2] \times 2^{\{1,2\} \times \{1,2\}}$ according to (6) (or sets all these values to 0 if $\hat{S}_P = \emptyset$), and afterwards it evaluates $\Lambda(P, m_1, m_2, Q)$ for every $(m_1, m_2, Q) \in [0, n_1] \times [0, n_2] \times 2^{\{1,2\} \times \{1,2\}}$ according to either (7) or (8). The optimal achievable gain for the caching problem is $\Lambda(P_{root}, n_1, n_2, \emptyset)$. The algorithm finds a caching which obtains this gain via standard backtracking and returns it.

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) [17], [18]. 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 [17], [18]. 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

Next, we discuss extensions beyond pairwise matching.

A. Fundamental results

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 $\lambda_r = \lambda_{2,r} \cdot (1-\alpha \cdot I(r \in C_1)) + \alpha \cdot \lambda_{1,r} \cdot I(r \notin C_1)$ for a rule R_r, where $I(\cdot)$ is the indicator function.*

Proof. Assume a given caching for switch 1. Clearly, a rule R_r that is not cached by switch 1, can reduce the cost by $\lambda_{2,r} + \alpha \cdot \lambda_{1,r}$ if it is cached by the second switch, reducing the cost for traffic matching the rule in both switches. If the rule is cached by switch 1, if it is then also cached by switch 2, the potential cost reduction is $\lambda_{2,r} \cdot (1-\alpha)$, reducing the classification cost only for traffic of the second switch. □

Theorem VII.2. *Consider switches selected in a round-robin manner. For a switch select its cached rules as a best response to the allocations of the other switches. Then,*

(i) *The total classification cost is monotonically non-increasing.*

(ii) *The process converges after a finite number of iterations.*

(iii) *The process does not necessarily converge to the optimal caching with the minimal total classification cost.*

Proof. In each step (either the time a switch is first considered or later), the caching of a switch is by definition the best possible from the system cost perspective and in particular, does not increase the cost of the previous selection of the switch. Moreover, since the number of solutions is finite and the optimal cost is bounded the process converges. Indeed, it can be shown that the process does not necessarily end up with the optimal cache selection even for the case of $k = 2$ switches. Consider two switches: the first with many rules, all with small popularity smaller than $\alpha/3$ and a second with two rules $R_{2,1}, R_{2,2}$ of popularities $0.5+\epsilon$, $0.5-\epsilon$. Assume a capacity of one rule in each switch. Starting from the first switch, the algorithm selects to cache in the first switch the rule $R_{2,1}$ from the second switch. Next, the rule $R_{2,2}$ is cached in the second switch. Later iterations do not change the cached rules. Unlike the solution of the algorithm, since the popularity of $R_{2,1}$ is greater than that of $R_{2,2}$, a lower classification cost is obtained when the rule $R_{2,1}$ is cached in its original second switch and the rule $R_{2,2}$ is cached in the first switch. □

Clearly, to avoid the phenomena of reaching a local minimum point of the classification cost, one can consider standard randomization techniques where the caching of a switch is selected given the previous selections not as the best achievable but as an efficient one with almost similar performance. More generally, approaches like simulated annealing can be useful in this context [47].

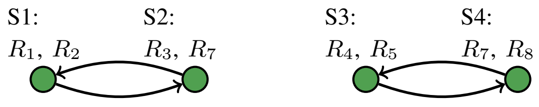
B. Simple and practical cooperative caching for multiple switches

Accordingly, while still relying on the power of cooperative caching, we restrict its degrees of freedom for simplicity and efficiency. We do so by designing (non optimal) solutions, where each switch has a single additional switch, selected in advance among all k switches, where he can solve a given local cache miss. The various schemes are illustrated in Fig. 4. They describe a tradeoff between simplicity and efficiency.

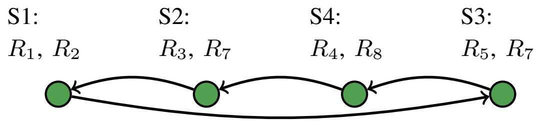
(i) Cooperation in pairs (illustrated in Fig. 4(b)) - This scheme has the advantage of being simple but with the cost of loss of potential efficiency. We partition the switches into pairs based on the similarity of their rule sets and traffic distributions. Then, cooperation is performed only within each pair of switches. We estimate the similarity between each pair of switches. Then, a partition can be found by a graph matching algorithm in a weighted general (not necessarily bipartite) graph. This can be done in polynomial-time by various implementations of Edmond's algorithm [48]. While ideally, we would like this estimation to describe the classification cost reduction achieved by each pair, this might require running the caching algorithm for each pair. Alternatively, we simply estimate this similarity by finding the top accessed rules in each of the switches and calculating the (weighted)

S1:	S2:	S3:	S4:
R_1 : 0.4	R_3 : 0.40	R_5 : 0.24	R_4 : 0.26
R_2 : 0.2	R_7 : 0.21	R_7 : 0.25	R_8 : 0.30
- - - - - - - -	- - - - - - - -	- - - - - - - -	- - - - - - - -
R_3 : 0.15	R_2 : 0.20	R_1 : 0.10	R_3 : 0.10
R_4 : 0.05	R_4 : 0.10	R_4 : 0.20	R_5 : 0.09
R_5 : 0.2	R_6 : 0.09	R_6 : 0.21	R_7 : 0.25

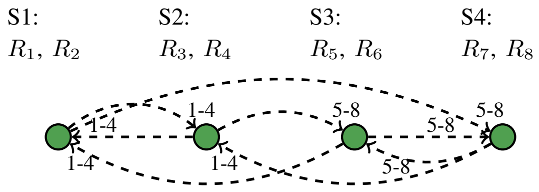
(a) Traditional rule caching: delay of $0.486 \cdot T_C$



(b) Cooperation in pairs



(c) Cooperation in chains



(d) Address-dependent cooperation

Figure 4. Example of multi-switch schemes: The subfigures illustrate the schemes described in Section VII-B An arrow is directed from a switch to another switch where its cache misses can be solved. A value on an arrow helps to determine the accessed switch based on the rule number observing a local miss.

intersection between those rules for each pair of switches. If the classification cost alternatives among the different switches vary (e.g., based on the switches locations and their differences), they can also be taken into account in the partition.

(ii) Cooperation among chains (illustrated in Fig. 4(c)) - We generalize the above approach. While still a switch can solve its local misses by accessing another switch, this relation is not required to be symmetric and cooperation has the form of chains. We start with a single switch that calculates its local caching (e.g., by a simple scheme for prefix rules or by techniques from [17], [18] for more general rule dependencies). Then, we examine the similarity of its non-cached rules to each of the rules in all other switches. We select as the next switch in the chain the switch that would benefit the most from the ability to access the cached rules in the last switch. We then continue to add more switches, enabling the selection of switches that have been selected earlier. In case the switch selected is not new, it might be the first in a chain or a later one. This would lead to a structure of switch relations that is not necessarily a chain. If this occurs, in the next step another switch is selected to start a new chain. Upon a miss, classification is completed by the next switch in the chain.

(iii) Address-dependent cooperation (illustrated in Fig. 4(d)) - We provide a solution that enables cooperation between a large number of switches while still enabling simple identification of the switch that has to be accessed in the case of a local cache miss. We divide the matching address space into disjoint parts among switches while trying to maximize inclusion of the local traffic in each switch. In each switch, in addition to caching of some locally popular rules, the rest of the capacity is dedicated to cache popular rules, possibly from all switches, that fall within the address space of the switch. Local cache misses are forwarded to a single adjacent switch covering the relevant space portion potentially having the corresponding rule. This can be done by adding to each cache, a rule directing local misses in the relevant portion to the corresponding switch. Some rule duplication might be required to deal with rules intersecting multiple portions.

Example 1. *Consider $k = 4$ switches, each with five rules and a cache size (in each switch) of two rules . Let the delay values satisfy $T_C/T_L = 11, T_D/T_L = 2$ implying $\alpha = 0.9$. The rule popularities are demonstrated in Fig. 4(a), for the seven distinct rules $R_1, \dots, R_7$. We assume no rule dependencies. In Fig. 4(a) the local rule caching are shown, where each switch caches its two popular rules, namely the two with the highist popularity among the five rules it has. This implies hit rates of 0.6, 0.61, 0.49 and 0.56 for the various switches, achieving a average delay reduction of $(0.6 + 0.61 + 0.49 + 0.56)/4 \cdot (T_C - T_L)$ = $2.26/4 \cdot (T_C - T_L) = 0.565 \cdot (T_C - T_L)$ which implies a delay of $0.486 \cdot T_C$. In Fig. 4(b) ,as in scheme (i) above, cooperation is done in pairs, where the division to pairs achieving the minimal delay is with the pairs (S1, S2) and (S3, S4). For instance, as S1 and S2 have in common rules R_2, R_3, R_4 and similarly S3 and S4 have R_4, R_5, R_6, namely pairs share rules with relatively large total popularity, each such pair can benefit from caching rules of the other switch in the pair and thus these two pairs are grouped. The average delay reduction increases with the use of the scheme to $0.76375 \cdot (T_C - T_L)$ which results in a delay of $0.306 \cdot T_C$. In Fig. 4(c), cooperation is done among chains as in scheme (ii) above. A switch can solve rule cache misses in the switch towards it has an outgoing arrow. We show the most efficient chain structure where the next switch in a chain is selected as the one that benefit the most from the last selected cache. For instance, as S1 caches only R_1, R_2 it can benefits from the caching of R_5 in S3 while S3 itself aches $R5, R7$ and enjoys the fact that S4 caches R_4. Similarly, S4 benefits from the caching of R_3, R_7 in S2. This achieves a further improvement, increasing the delay reduction to $0.78875 \cdot (T_C - T_L)$ implying a delay of $0.283 \cdot T_C$. Last, the address-dependent cooperation, detailed as scheme (iv) above, is shown in Fig. 4(d). Assume that following the rule structure, upon a local miss, a switch can look for one of the first 1-4 rules in one switch and for the last 5-8 in another switch. Intuitively, rules $R_1 - R_4$ have a relatively high popularity in S1, S2 while $R_5 - R_8$ in S3, S4, (while of course a particular rule does not necessarily even appear in the original set of rules in each switch). This further reduces the delay to $0.81525 \cdot (T_C - T_L)$ leading to a delay of $0.259 \cdot T_C$.*

VIII. 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, illustrated in Fig. 5, 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. 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 [49] 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.

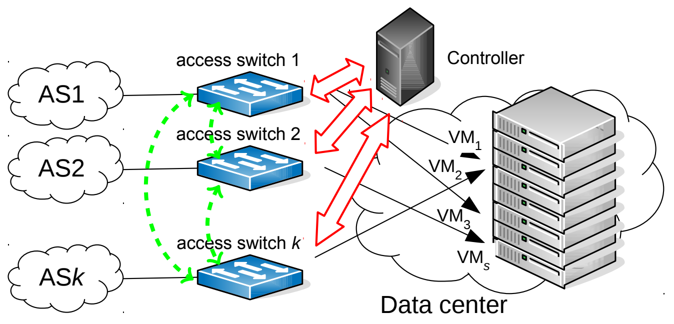


Figure 5. Cloud access gateway: k switches provide access for distinct ASes to s data center services. Dashed arrows in green illustrate potential communication between the switches to solve local cache misses within the data plane.

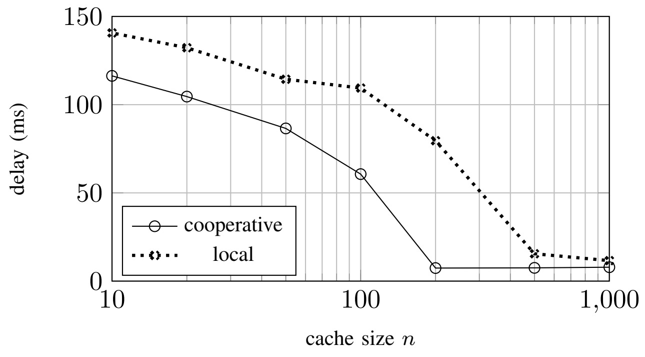


Figure 6. Delay vs. cache size. The knee of the cooperative (resp., local) curve occurs roughly at $n = 200$ (resp., $n = 500$), close to optimal offloading.

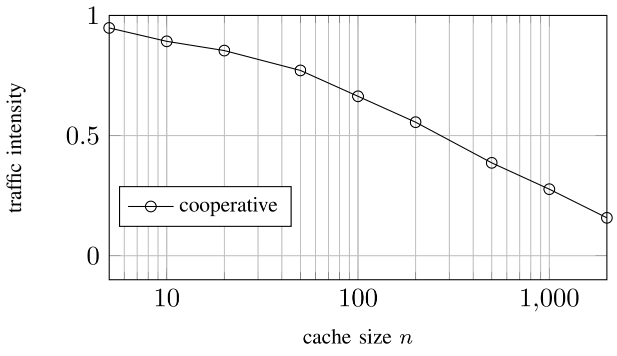


Figure 7. Traffic intensity (partial overhead) vs. cache size. Cooperative caching uses excess bandwidth among switch links. Local caching does not offload traffic.

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, respectively). 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.

Latency vs. cache size. Fig. 6 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 storing 2000 rules, out of which 1096 are shared. A partial traffic trace, taken again from the CAIDA dataset [49], was fed into one of the access switches, containing 10,000 packets in 334 flows, with the individual flow sizes varying between 10 to 216 packets each, following Zipf's law with a best fit exponent of $s = 0.52$. Shared rules account for 41% of the total ingress traffic. While cooperative caching maintains a comfortable two-times edge over local caching at basically all reasonable

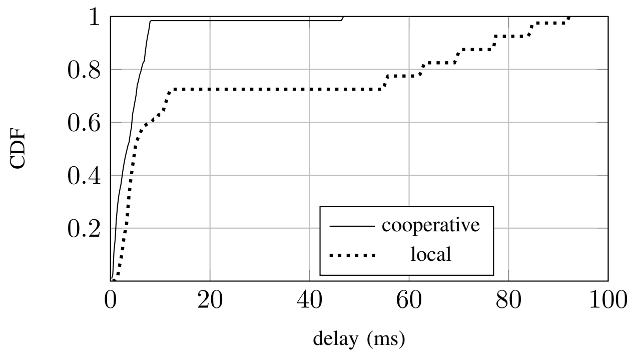


Figure 8. Cumulative one-way delay distribution. The delay 99-percentile under the cooperative (resp., local) solution is roughly 10 ms (resp., 95 ms).

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).

Offloaded traffic intensity. The price for cooperative caching is that certain rules are enforced at remote switches and sending offloaded traffic to the fallback(s) imposes additional load on the data plane. Fig. 7 shows the normalized offloaded traffic rate as a function of the cache size. For very small caches we observed that the majority of traffic is classified remotely whereas caching only 10% of the rules ($n = 200$) renders roughly half of the traffic being handled locally. Note that local caching does not direct *any* traffic to the link between the switches, essentially wasting the bandwidth of the inter-switch link, whereas cooperative rule caching leverages spare resources at the fast data plane.

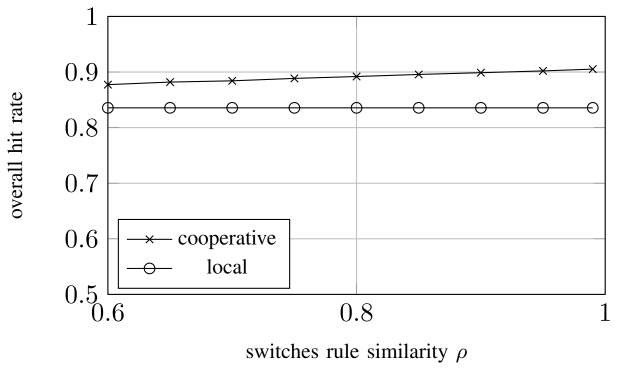
In summary, the proposed cooperative caching scheme has clear benefits for the given use case. Nonetheless, at arbitrary networks inter-switch links may become bottlenecks. In those cases, the proposed solution must be adjusted to cope with the trade-off between data plane overhead against control plane delays. We leave this study as subject for future work.

Overall delay. Fig. 8 shows the one-way delay CDF (2000 rules and cache size 200). The delay 99-percentile under cooperative (resp., local) caching is roughly 10 ms (resp., 95 ms), which is in agreement with the fact that under local caching delays are caused by large queue backlogs building up at the controller. As cooperative caching can successfully keep most traffic in the data plane, it avoids the controller round-trip almost completely, implying significantly smaller delays.

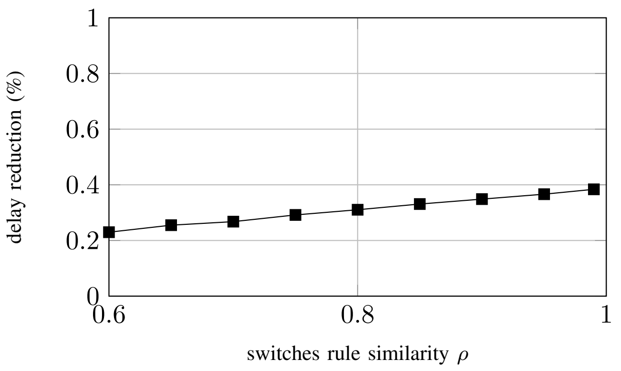
IX. Impact of Workload and Rule Similarity

Next, we evaluate the impact of workload and rule similarity on hit rate and delay. To this aim, we conduct synthetic simulations where we assume that a Zipf distribution with parameter μ determines the popularity of a rule. Each switch has a complete set of 10K rules, and a rule i has popularity

$$p_i = \frac{1}{\eta} \cdot i^{-\mu}, \quad i = 1, \dots, 10^4,$$



(a) hit rate vs. rule similarity



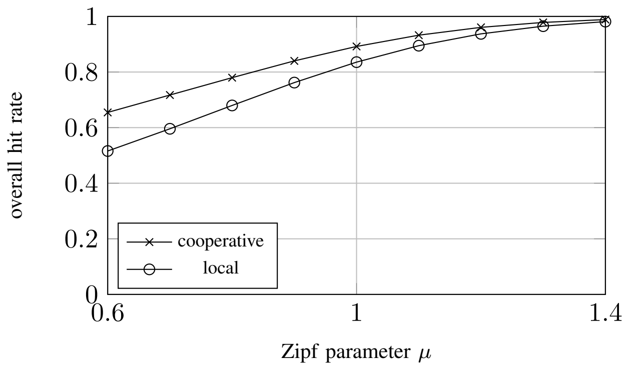
(b) delay reduction vs. rule similarity

Figure 9. Comparison of cooperative vs. local caching: hit rate and delay as a function of rule similarity.

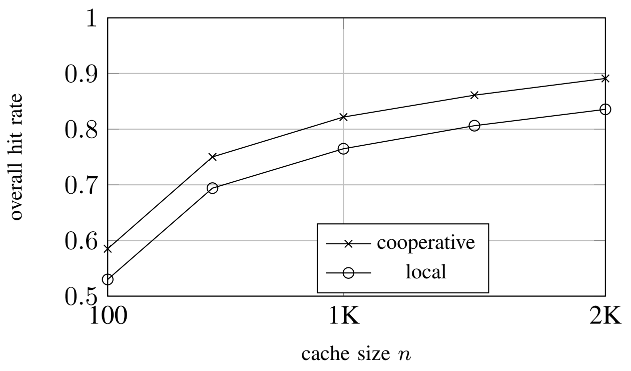
where $\eta = \sum_{j=1}^{10K} j^{-\mu}$ is a normalization constant. We compare the average overall hit rate (ratio of traffic that can be classified within the data plane) and classification time that can be achieved with cooperative caching vs. that of the traditional local caching. In the implementation of the cooperative caching, we simply selected the cache of the switches iteratively as the best response following on Theorem VII.1.

We focus on the case of two switches. We used a parameter ρ to determine the probability of two switches to share a given rule. If this is not the case, we assume they have two distinct rules, associated for simplicity with the same popularity. The results are illustrated in Fig. 9. In Fig. 9(a), we assume a Zipf parameter $\mu = 1$, a cache size of n=2K rules in each switch, and rule similarity of $\rho \in [0.6, 1]$ between the two switches. For the local independent caching, the rule similarity has no impact on the performance and a fixed hit rate of 0.836 is achieved in each switch. For the cooperative caching, the improvement is an increasing function of the rule similarity. The achievable hit rates are between 0.877 and 0.905, describing a reduction of 25%-42% in the traffic sent to the controller. As demonstrated in Fig. 9(b), this increase in the amount of traffic served within the data plane is translated to a reduction of 23%-38% in the average classification time, where we used values of $T_C/T_L = 101, T_D/T_L = 2$ implying $\alpha = 0.99$.

In Fig. 10(a)-10(b) we examine the impact of the Zipf parameter μ and the cache size n. We assume rule similarity of $\rho = 0.8$. Cooperative caching increases the total hit rate



(a) hit rate vs. Zipf parameter



(b) hit rate vs. cache size

Figure 10. Comparison of cooperative vs. local caching: hit rate as a function of workload and cache size.

from 0.516 to 0.655 for $\mu = 0.6$ and from 0.981 to 0.989 for $\mu = 1.4$ considering $n = 2\text{K}$. A maximal relative decrease of 31% in the delay is obtained for $\mu = 1.1$. Similarly, in examining the impact of the cache size $n \in [100, 2\text{K}]$, we can see a reduction of 11%-31% in the delay.

X. 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 an approach to circumvent the limitations imposed by memory constrained devices, e.g., of IoT networks [2], [16]. This work paves the way towards that vision. As future work, we plan to evaluate the proposal in additional settings, including IoT networks and NFV scenarios where SmartNICs or P4 switches are leveraged as caches to offload x86 cycles for bandwidth-intense or latency-sensitive flows.

XI. Acknowledgements

This research was partially supported by the Technion Hiroshi Fujiwara Cyber Security Research Center, the Israel Cyber Bureau and the Israel National Cyber Directorate, by the European Research Council (ERC) consolidator grant No. 725978 SYSTEMATICGRAPH, by the NKFIH/OTKA Project ANN-135606 and the MTA–BME Momentum Network Softwarization Research Group, by CAPES, CNPq and FAPERJ through grants E- 26/203.215/2017 and E-26/211.144/2019.

References

[1] O. Rottenstreich, A. Kulik, A. Joshi, J. Rexford, G. Rétvári, and D. S. Menasché, "Cooperative rule caching for SDN switches," in *IEEE International Conference on Cloud Networking (CloudNet)*, 2020.

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

[3] 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.

[4] 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.

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

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

[7] 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.

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

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

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

[18] N. P. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Infinite cacheflow in software-defined networks," in *ACM workshop on Hot topics in software defined networking (HotSDN)*, 2014.

[19] R. Li, B. Zhao, R. Chen, and J. Zhao, "Taming the wildcards: Towards dependency-free rule caching with freecache," in *IEEE/ACM International Symposium on Quality of Service (IWQoS)*, 2020.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] R. Bauer and M. Zitterbart, "Port based capacity extensions (PBCEs): Improving SDNs flow table scalability," in *IEEE International Teletraffic Congress (ITC)*, 2016.

[34] S. Geissler, S. Herrnleben, R. Bauer, A. Grigorjew, T. Zinner, and M. Jarschel, "The power of composition: Abstracting a multi-device SDN data path through a single API," *IEEE Trans. Netw. Serv. Manag. (TNSM)*, vol. 17, no. 2, pp. 722–735, 2020.

[35] J. Wang, S. Cheng, and X. Fu, "SDN programming for heterogeneous switches with flow table pipelining," *Scientific Programming*, vol. 2018, no. 2848232, pp. 1–13, 2018.

[36] O. Rottenstreich and J. Tapolcai, "Lossy compression of packet classifiers," in *ACM/IEEE Symposium on Architectures for networking and communications systems (ANCS)*, 2015.

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

[38] 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.

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

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

[41] S. Smale, "Mathematical problems for the next century," *The mathematical intelligencer*, vol. 20, no. 2, pp. 7–15, 1998.

[42] M. B. Cohen, Y. T. Lee, and Z. Song, "Solving linear programs in the current matrix multiplication time," in *ACM SIGACT Symposium on Theory of Computing (STOC)*, 2019.

[43] 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.

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

[45] 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.

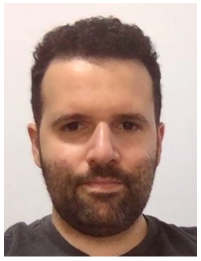
[46] J. Schwartz, A. Steger, and A. Weißl, "Fast algorithms for weighted bipartite matching," *Lect. notes comp. sci.*, vol. 3503, pp. 476–487, 2005.

[47] A. Khachaturyan, S. Semenovskaya, and B. Vainshtein, "Statistical-thermodynamic approach to determination of structure amplitude phases," *Sov. Phys. Crystallography*, vol. 24, no. 5, pp. 519–524, 1979.

[48] J. Edmonds, "Paths, trees, and flowers," *Canadian Journal of mathematics*, vol. 17, no. 3, pp. 449–467, 1965.

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

Ori Rottenstreich is an assistant professor at the department of Computer Science and the department of Electrical Engineering of the Technion, Haifa, Israel. Previously, he was a Postdoctoral Research Fellow at Princeton university. Ori received his B.Sc. degree in Computer Engineering and Ph.D. degree in Electrical Engineering from Technion.



Ariel Kulik is currently a postdoctoral researcher at The CISPA Helmholtz Center for Information Security in Germany. Ariel received the Ph.D degrees in Computer Science from the Technion in 2021. His research is focused on the design and analysis of algorithms for resource allocation problems and graph problems.

Ananya Joshi is currently a graduate student in the Computer Science Department at Carnegie Mellon University and works on forecasting epidemics.

Jennifer Rexford (Fellow, IEEE) received the B.S.E. degree in electrical engineering from Princeton University in 1991, and the Ph.D. degree in electrical engineering and computer science from the University of Michigan in 1996. She is currently the Gordon Y. S. Wu Professor of Engineering and the Chair of Computer Science with Princeton University. Before joining Princeton in 2005, she worked for eight years at AT&T Labs–Research. She is a coauthor of the book Web Protocols and Practice (Addison-Wesley, May 2001). She is an ACM Fellow in 2008 and a member of the American Academy of Arts and Sciences in 2013 and the National Academy of Engineering in 2014. She was the 2004 winner of ACM's Grace Murray Hopper Award for outstanding young computer professional, the ACM Athena Lecturer Award in 2016, the NCWIT Harrold and Notkin Research and Graduate Mentoring Award in 2017, the ACM SIGCOMM Award for lifetime contributions in 2018, and the IEEE Internet Award in 2019. She served as the Chair for ACM SIGCOMM from 2003 to 2007.

Gábor Rétvári received the M.Sc. and Ph.D. degrees in electrical engineering from the Budapest University of Technology and Economics in 1999 and 2007. He is now a Senior Research Fellow at the Department of Telecommunications and Media Informatics at BME and a visiting scholar at Ericsson Research. He is the co-author of 70+ scientific papers, among them cornerstone contributions to the design of programmable dataplane devices and algorithms, secure virtual switching, and complex network analysis. His research interests include all aspects of network routing and switching, but lately he has been specializing in programmable dataplane design and analysis and application-layer network virtualization.

Daniel Sadoc Menasché received his Ph.D. degree in Computer Science from the University of Massachusetts, Amherst, in 2011. Currently, he is an Associate Professor with the Computer Science Department, Federal University of Rio de Janeiro, Brazil. His interests are in modeling, analysis, security and performance evaluation of computer systems. He is an alumni affiliated member of the Brazilian Academy of Sciences.