Efficient Collaborative Rule Caching Through Pairing of P4 Switches in SDNs

Mohammad Saberi*, Mahdi Dolati*, Ali Movaghar*, Tooska Dargahi§, and Ahmad Khonsari†‡
*Department of Computer Engineering, Sharif University of Technology, Tehran, Iran.
‡School of Computer Science, Institute for Research in Fundamental Sciences, Tehran, Iran.
§Manchester Metropolitan University, Manchester, UK.
†Department of Electrical and Computer Engineering, University of Tehran, Tehran, Iran.
Emails: {mhmd.saberi, mdolati, movaghar}@sharif.edu, t.dargahi@mmu.ac.uk, a_khonsari@ut.ac.ir

***Abstract*—Software-defined networks (SDNs) provide customizable traffic control by storing numerous rules in on-chip memories with minimal access latency. However, the current on-chip memory capacity falls short of meeting the growing demands of SDN control applications. While rule eviction and aggregation strategies address this challenge at the switch level, programmable data planes enable a more flexible approach through cooperative rule caching. However, current solutions rely on computationally intensive off-the-shelf solvers to perform rule placement across the network. In this paper, we present an efficient solution for the cooperative rule caching problem. We first present the design of a resource-efficient switch capable of caching rules for its neighbors alongside a lightweight protocol for retrieving cached rules. Then, we introduce RaSe, an approximation algorithm for minimizing rule lookup latency across the network through optimized cooperation-aware rule placement. We conduct a theoretical analysis of RaSe, followed by a P4-based proof-of-concept assessment in Mininet and a large-scale numerical evaluation using real-world network topology. In comparison with existing solver-based solutions, the proposed method obtains the solution 160 times faster and improves the average rule lookup latency by about 21% compared to several algorithmic baselines.**

***Index Terms*—Software-defined networks, rule caching, switch cooperation, data plane, optimization, approximation algorithm.**

I. Introduction

Software-Defined Networks (SDNs) provide network administrators with unprecedented flexibility and control over traffic management. This level of customization allows switches to direct packets to specific destinations, where crucial network functions are executed to achieve performance and functionality objectives. At the core of this capability is the ability of SDN-enabled switches to analyze data packets and make routing decisions based on a high-level network policy [1]. To enable this customized routing, each switch requires a set of rules that identify incoming traffic and trigger appropriate actions. These rules are stored in specialized on-chip memories, such as SRAMs and TCAMs, which are selected for their ability to provide extremely low access latency. This low latency is crucial for effectively handling traffic from various applications, such as virtual and augmented reality, which are highly sensitive to delays.

Despite recent progresses, the capacity of on-chip memories does not meet the demands of SDN control applications. This limitation forces certain traffic to retrieve rules from slower off-chip memory (such as DRAM) or the remote central controller. Unfortunately, these alternatives are deemed unacceptable for delay-sensitive flows [2]. Expanding the on-chip memory capacity faces various physical limitations while equipping switches with larger memories leads to increased costs, power consumption, and heat generation [3]. Consequently, SDNs have seen the emergence of alternative strategies aimed at optimizing the available capacity, either at the switch level or across the entire network [4]. Some efforts focus on solving the problem of evicting infrequent rules to create space for crucial rules. Another switch-level technique involves aggregating [5] and compressing [6] existing rules, enabling the same semantic representation with a reduced number of rules by using wildcards.

Orthogonal to switch-level solutions, the programmable data plane offers flexible network-wide remedies for mitigating memory constraints in SDNs. Seminal studies have illustrated the potential of assigning pairs to each switch and utilizing neighboring pairs' memory as an in-network cache [2]. However, these studies often overlook memory allocation optimization or rely on computationally intensive integer linear program (ILP) solvers, which struggle to meet the time constraints of modern networks. We address this research gap by presenting an efficient algorithm that finds optimal solution for the joint problem of switch-pairs selection and rule placement.

We formulate the **Pa**ir **Se**lection (PaSe) problem, which incorporates the load and traffic pattern within an ILP. Then, we design a switch (based on the Protocol Independent Switch Architecture [7]) that is able to share its memory seamlessly among its neighbors. Then, we propose a low-overhead protocol to retrieve the cached rules. Finally, we design a **R**ounding-based P**a**ir **Se**lection (RaSe) algorithm for PaSe that runs in polynomial time. RaSe is designed based on deterministic rounding of ILPs, a robust framework for crafting approximate algorithms. The key contributions of this paper are as follows:

- We design a switch capable of collaborative rule caching as a proof-of-concept and implement it in P4 language.

- We introduce RaSe, an algorithm addressing the joint challenge of pair selection and rule placement in SDNs.
- We analyze the approximation ratio of RaSe.
- We emulate and test the P4 implementation of the designed switch in Mininet using iperf and ping tools.
- We show the scalability and performance of RaSe by comparing it with the optimal solution obtained from an ILP solver [8] and other baselines through numerical simulations across a wide range of parameters and conditions.

In the following, Section II reviews the relevant works. The problem is described in Section III, followed by the proposed solution in Section IV. Evaluation results are discussed in Section V. Finally, conclusions are drawn in Section VI.

II. Related Work

In [2], the authors enhance rule caching in SDNs by allowing switches to cache matching rules for their neighboring switches. However, this work neglects the load imbalance in the network and adopts a simplistic random pairing strategy, which limits the overall efficiency. The authors in [9] use software switches to overcome the constraints posed by on-chip memories in SDNs. They argue that consideration of network load imbalance is vital and propose a traffic-aware rule cache assignment algorithm using a matching framework. The work in [10] considers both temporal and spatial traffic patterns and proposes a selective caching algorithm for TCAMs. Authors in [11] present polynomial-time algorithms for caching of longest-prefix-match policies in SDNs.

The authors in [12] present a deep Q-learning-based approach to overcome the limitations of flow table capacities in SDNs. Their method considers idle timeouts and rule dependencies to dynamically adapt to network conditions, ensuring rules are removed in the correct order. SmartTime [13] employs an adaptive timeout selection for proactive eviction of forwarding rules in its flow management strategy. Similarly, FlowMaster [14] predicts the likely expiration of an entry and proactively removes it to create space for new entries. This prediction is facilitated by a Markov-based learning predictor that considers the ongoing value of individual flows. FlowMaster categorizes flows into Live, Unique, Revisited, and Discarded, and decides on rule retention or eviction based on these categories. Authors in [15] present a randomized competitive algorithm for the general problem of caching with dependencies.

III. Problem Statement and Formulation

In this section, we describe the problem of rule caching in SDNs and subsequently present its formulation as an ILP.

A. System Model

We consider an SDN where the controller aims to place a set of rules, $\mathcal{R}$, in the memory of a subset of switches, $\mathcal{S}$. Each rule $r \in \mathcal{R}$ must reside in switch $w_r \in \mathcal{S}$ to route a subset of traffic with rate ρ_r packets per second. Each switch $w \in \mathcal{S}$ is equipped with a total of M_w memory slots and each rule r occupies q_r memory slots. Switches search for the matching rules in a predetermined order, dictated by priority (as per the

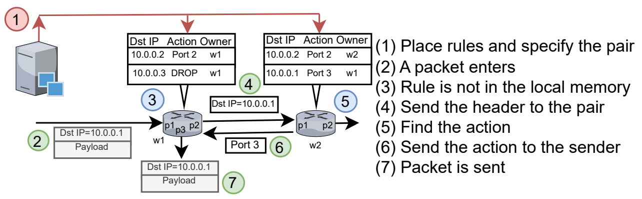


Fig. 1. Example of collaborative packet handling with cached rules, highlighting notable interactions among entities in the considered system.

OpenFlow protocol). As a result, if a rule does not exist in the memory of a switch, it may incorrectly use another rule with a lower priority. To address this rule dependency, we assume that each rule $r \in \mathcal{R}$ possesses a set of dependent rules, denoted as $\mathcal{R}_r$, such that if r is in a switch all rules in $\mathcal{R}_r$ should also be placed in the same location.

To alleviate memory limitations, we assume that each switch is paired with another switch capable of storing a subset of rules on its behalf [2]. See Fig. 1 for an example. When queried by the original switch, the paired switch can respond with the appropriate action associated with the cached rules. If the queried rule is not cached, the pair will relay the query to the controller, which will respond to the originator of the query directly. This collaborative rule retrieval is faster than querying the remote controller or the local off-chip memory due to the ultra-low transmission and on-chip processing latency in switches [2]. We assume that the rule retrieval time from the local on-chip memory is d_1, from the paired switch is d_2, and from the remote controller or the off-chip memory is at most d_3, where $d_1 \leq d_2 \leq d_3$.

B. PaSe: Pair Selection problem

We use two binary decision variables to formulate the problem as an ILP. The first variable, $z_{w,v}$, is set to one if switch v caches rules for switch w, and zero otherwise. We refer to switch v as the *pair of* switch w. Two switches are not necessarily each other's pair. The second variable, $y_{w,r}$, is set to one when rule r is placed in switch w and zero otherwise. The formulation is as follows:

$$\min \sum_{r\in\mathcal{R}} \rho_r \cdot \Big\{ d_1 \cdot y_{w_r,r} + \sum_{v\in\mathcal{S}-\{w_r\}} d_2 \cdot y_{v,r} + d_3 \cdot \big(1 - \sum_{v\in\mathcal{S}} y_{v,r}\big) \Big\} \tag{1}$$

$$\text{s.t.} \sum_{v\in\mathcal{S}_w} z_{w,v} = 1, \quad w \in \mathcal{S}, \tag{2}$$

$$\sum_{w\in\mathcal{S}} y_{w,r} \leq 1, \quad r \in \mathcal{R}, \tag{3}$$

$$y_{v,r} \leq 1_{v=w_r} + z_{w_r,v}, \quad r \in \mathcal{R}, v \in \mathcal{S}_{w_r}, \tag{4}$$

$$\sum_{r\in\mathcal{R}} q_r \cdot y_{v,r} \leq M_v, \quad v \in \mathcal{S}, \tag{5}$$

$$y_{v,r} \leq y_{v,r'}, \quad r \in \mathcal{R}, r' \in \mathcal{R}_r, v \in \mathcal{S}. \tag{6}$$

The objective function in (1) shows the average rule retrieval latency weighted by rule usage rate. The weights prioritize the storage of frequently used rules in the local memory of switches. The objective comprises three terms: (i) if rule r is placed in w_r, considering the minimum latency d_1; (ii) if rule r is in any other switch (i.e., $v \in \mathcal{S} - w_r$), then the moderate latency d_2 is considered; and (iii) if rule r is not present in any switch, implying it resides either in the controller or the off-chip memory of the switch, then the maximum latency of d_3 is considered.

Algorithm 1 Table containing cached rules.

```
table cache_t {
  key = {
    cache_h.action_or_owner:
          exact;
    ipv4.dstAddr: lpm;
  }
  actions = {
    get_self_action;
    get_cached_action;
    NoAction;
  }
}
```

Algorithm 2 Table containing the ports to pair and controller.

```
table pairing_t {
  key = {
    cache_h.type: exact;
  }
  actions = {
    query_pair;
    send_2ctrl;
    NoAction;
  }
}
```

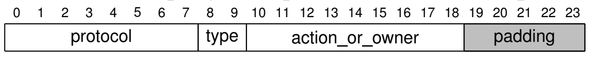
Constraint (2) guarantees that each switch w has exactly one pair, where set $\mathcal{S}_w$ represents the feasible pairs for switch w. Note that a switch may be the pair of multiple switches. Constraint (3) ensures that each rule r may be placed within at most one switch. If $y_{w,r}$ equals zero for all w, then that rule r is stored in the remote controller or the off-chip memory. Constraint (4) enforces that each rule r can only be stored in w_r (i.e., $1_{v=w_r} = 1$) or its pair (i.e., $z_{w_r,v} = 1$). Note that $1_{v=w_r}$ is an indicator function that is equal to one if $v = w_r$ and zero otherwise. Constraint (5) ensures that the memory capacity of switches is respected. Constraint (6) ensures that if rule r is placed in a switch, all of its dependent rules, i.e., $\mathcal{R}_r$, are also placed there.

IV. Proposed Solution

In this section, we first design a switch capable of rule caching based on the Portable Switch Architecture and present its implementation in P4 language in Subsection IV-A. Then, we propose an algorithm to solve the RaSe problem in Subsection IV-B.

A. Design and P4 Implementation

We introduce the following packet header (named **cache_h**) to enable switches to query their pairs and receive a response:



The **protocol** field is the backup for the **protocol** field in the IPv4 header. We discuss the reason for this backup in the following paragraph. The **type** field takes four values **TYPE_L=0, TYPE_Q1=1, TYPE_Q2=2, TYPE_R=3**. A switch uses **TYPE_L** to search for its own rules in its local cache. The value **TYPE_Q1** allows to search the local cache for rules belonging to other switches. The value **TYPE_Q2** specifies a query that is sent to the controller and **TYPE_R** specifies a response. The **action_or_owner** field contains a number that, depending on the value of the **type** field, either identifies the originator of the query or a response value. A response value specifies the port to which the packet should be forwarded. Since many switch targets only support headers that are a multiply of eight bits long, the last five bits in **cache_h** are padding bits.

Upon adding **cache_h**, the switch must store a unique number in a IPv4 header field called **protocol** to allow the recipient parse the packet correctly. We use the unassigned value 146 as the unique number for **cache_h**. The switch removes **cache_h** after receiving a response from the pair or the controller. Upon removal, the switch must restore the **protocol** value in the IPv4 header. Therefore, before writing 146 into the IPv4 header, the switch stores its current value in the **protocol** field of **cache_h** for later restoration.

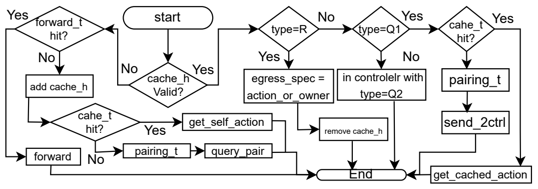


Fig. 2. Ingress processing logic of the designed cache enabled switch.

Each switch contains a table called **forward_t**, which stores the switch's local rules. The switch forwards a packet as usual when it has a matching entry in **forward_t**. To cache rules belonging to other switches, we add two additional tables called **cache_t** and **pairing_t** to each switch. See Algorithms 1 and 2 for a description of these switches in P4 language. Table **cache_t** can store forwarding rules for the neighbors or the switch itself. Table **pairing_t** provides information to send a query to the pair, to the controller, or a response packet to the original sender of the query.

When a switch does not find a matching rule in **forward_t**, it adds the **cache_h** header with **type=TYPE_L** and **action_or_owner=0** and searches **cache_t**. If a match is found, the header is removed and the packet is forwarded regularly. We refer to this action by **get_self_action** in table **cache_t**. This design enables each switch to utilize the cache capacity for its own purposes. If a match is not found in **cache_t**, the switch uses table **pairing_t** to send a query to the pair. Table **pairing_t** matches on any packet with header field **type=TYPE_L** and contains a value that shows the port number from which the pair is accessible. Before sending out the packet, the switch sets the value of the field **type** to **TYPE_Q1**. We call this action **query_pair** is table **pairing_t**.

When a switch receives a packet containing **type=TYPE_Q1**, it sets the value of **action_or_owner** to the ingress port of the packet and searches table **cache_t** for a match. If a rule is found, the switch modifies the packet according to the rule, changes the value of **type** to **TYPE_R**, stores the port number that the original sender of the query should forward the packet in **action_or_owner** and sends the packet back to the sender of the query through the same port that the query packet has been received. These operations are performed under an action called **get_cached_action** in table **cache_t**. However, if a match is not found, the switch uses an action called **send_2ctrl** in table **pairing_t** to set **type=TYPE_Q2** and forward the packet to the controller. The controller uses the ingress port and the number in field **action_or_owner** to find the originator of the query and sends a response directly to it.

When a switch gets a packet containing **type=TYPE_R**, it restores the **protocol** value in IPv4 header, removes the header, and forwards the packet through the port specified in the **action_or_owner** field. Figure 2 depicts the proposed ingress processing.

B. Algorithm Design

In this section, we propose the RaSe algorithm within the deterministic rounding framework for linear programs. As shown in Algorithm 3, RaSe starts by building an instance of PaSe using the input parameters. Then, in line 2, RaSe relaxes the problem instance by discarding the integrality constraints on decision variables $z_{w,v}$ and $y_{w,r}$ to form a linear program, which is

Algorithm 3 RaSe: **R**ounding-Based P**a**ir **Se**lection Algorithm

```
Input: $\mathcal{R}$, $\mathcal{S}$, $\{\rho_r\}$, $\{d_1, d_2, d_3\}$, $\{M_v\}$, $\{q_r\}$
Output: $\{z_{w,v}, y_{w,r}\}$  // pairing and placement decisions
 1: $m \leftarrow$ PaSe($\mathcal{R}, \mathcal{S}, \{\rho_r\}, \{d_1, d_2, d_3\}, \{M_v\}, \{q_r\}$)
 2: $\widetilde{m} \leftarrow$ relax($m$)
 3: $\{\widetilde{z}_{w,v}, \widetilde{y}_{w,r}\} \leftarrow$ solve($\widetilde{m}$)
 4: for $w \in \mathcal{S}$ do
 5:     $v \leftarrow \arg\max_{v' \in \mathcal{S}} \widetilde{z}_{w,v'}$
 6:     $\widetilde{m}$.add_constraint($z_{w,v} = 1$)
 7: $\widetilde{m} \leftarrow$ update($\widetilde{m}$, $\{\frac{M_v}{3}\}$)
 8: $\{\widetilde{z}_{w,v}, \widetilde{y}_{w,r}\} \leftarrow$ solve($\widetilde{m}$)
 9: sort($\mathcal{R}$, key=$\{|R_r|\}$)
10: $\{L_r\} \leftarrow$ NULL
11: for $r \in \mathcal{R}$ do
12:     $v \leftarrow \arg\max_{v' \in \mathcal{S}} \widetilde{z}_{w,v'}$
13:     $\alpha \leftarrow 1 - \widetilde{y}_{w_r,r} - \widetilde{y}_{v,r}$
14:     $s \leftarrow$ NULL
15:     if $\widetilde{y}_{w_r,r} \geq \widetilde{y}_{v,r}$ and $\widetilde{y}_{w_r,r} \geq \alpha$ and ($L_r$ is NULL or $L_r = w_r$) then
16:         $s \leftarrow w_r$
17:     else if $\widetilde{y}_{v,r} \geq \widetilde{y}_{w_r,r}$ and $\widetilde{y}_{v,r} \geq \alpha$ and ($L_r$ is NULL or $L_r = v$) then
18:         $s \leftarrow v$
19:     else
20:         $\widetilde{m}$.add_constraints($y_{w_r,r} = 0, y_{v,r} = 0$)
21:     if $s$ is not NULL then
22:         $\widetilde{m}$.add_constraint($y_{s,r} = 1$)
23:         $\mathcal{R} \leftarrow \mathcal{R} - \mathcal{R}_r$
24:         for $r' \in \mathcal{R}_r \cup \{r\}$ do
25:             for $r'' \in \mathcal{R}'_{r'}$ do
26:                 $L_{r''} \leftarrow s$
27: $\widetilde{m} \leftarrow$ update($\widetilde{m}$, $\{M_v\}$)
28: $\{\widetilde{z}_{w,v}, \widetilde{y}_{w,r}\} \leftarrow$ solve($\widetilde{m}$)
29: return $\{\widetilde{z}_{w,v}, \widetilde{y}_{w,r}\}$
```

solvable in polynomial time [16]. Then, RaSe solves the relaxed problem instance, denoted as $\widetilde{m}$, in line 2 by employing an off-the-shelf solver (e.g., SciPy [8]). In the subsequent discussion, we refer to the values of decision variables $z_{w,v}$ and $y_{w,r}$ obtained from solving the linear program as $\widetilde{z}_{w,v}$ and $\widetilde{y}_{w,r}$, respectively. RaSe uses the fractional solution to construct an integer solution in two phases: (1) pair selection and (2) rule placement.

C. Pair Selection

RaSe interprets $\widetilde{z}_{w,v}$ as a measure of fitness, choosing the switch v that maximizes $\widetilde{z}_{w,v}$ to serve as the pair for switch w. This step is outlined in lines 4 to 6 in Algorithm 3. RaSe adds the new constraint $z_{w,v} = 1$ to enforce that v is w's selected pair.

Lemma 1. *Let $\widehat{S} = \max_{w \in \mathcal{S}} |\mathcal{S}_w|$. Deterministic rounding of $z_{w,v}$ variables increases the delay by at most a factor of $1 + d_3/d_2 \left(1 - 1/\widehat{S}\right)$.*

Proof. RaSe rounds the maximum $\widetilde{z}_{w,v}$ to one, which is at least $1/|\mathcal{S}_w|$ for each switch w due to constraint (2). Hence, $1 - 1/|\mathcal{S}_w|$ of rules may be in other fractionally selected pairs. After rounding the maximum $\widetilde{z}_{w,v}$ to one, its capacity may be insufficient to store the rules in the remaining fractionally selected pairs. All such rules are forced to be placed in the controller, which increases the objective by a factor of $1 + d_3/d_2(1 - 1/|\mathcal{S}_w|)$. Therefore, the rounding procedure, in the worst-case scenario, increases the objective by a factor of $1 + d_3/d_2(1 - 1/\widehat{S})$. ■

D. Rule Placement

RaSe assigns a fitness score to rule placement options based on fractional values of variables $\widetilde{y}_{w,r}$. The score of the original switch, its pair, and the remote controller for each rule r are, respectively, $\widetilde{y}_{wr,r}$, $\widetilde{y}_{v,r}$, and $1 - \widetilde{y}_{w_r,r} - \widetilde{y}_{v,r}$, where $z_{w_r,v} = 1$. Since there are three options, the maximum of these values is at least $1/3$. Consequently, RaSe should ensure that rounding it to 1 does not cause a memory capacity violation. To this end, RaSe divides the memory capacity of all switches by three and resolves the problem in lines 7 and 7. This modification ensures that rounding any variable from at least $1/3$ to 1 would not violate a memory capacity constraint.

RaSe places rules in the descending order of the number of their dependants by first sorting them in line 9. When a rule is placed in a switch, RaSe places all of its dependant rules (i.e., $\mathcal{R}_r$) in that switch and removes them from the set of rules $\mathcal{R}$ in line 23. However, this strategy is not sufficient to guarantee the rule dependency constraints. To this end, RaSe uses set $\mathcal{R}'_r$ that shows the set of rules that depend on r. When rule r is placed in a switch, RaSe assigns the identifier of that switch to an auxiliary variable $L_{r'}$, initialized NULL, for all $r' \in \mathcal{R}'_r$. In subsequent iterations, if variable $L_{r'}$ is not NULL, r' may only be placed in the specified switch or the remote controller. RaSe introduces two constraints in line 20 when rule r is placed in neither w_r nor its pair. This situation occurs when the values of $\widetilde{y}_{wr,r}$ and $\widetilde{y}_{v,r}$ are less than $1 - \widetilde{y}_{w_r,r} - \widetilde{y}_{v,r}$.

Lemma 2. *The objective of the modified problem with a third of capacity increases by a factor of $(\frac{1}{3} + \frac{2d_3}{3d_1})$.*

Proof. A third of traffic is forwarded similar to the original problem and $2/3$ of traffic is pushed to the controller. The difference between the data plane delay and the controller delay is at most $\frac{d_3}{d_1}$, which proves the lemma. ■

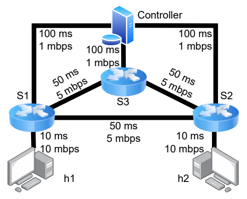
Lemma 3. *Let κ be the ratio between the volume that can be handled entirely in the data plane to the total volume of traffic. Assuming $n = \max\{2, \frac{d_3}{d_1} - 1\}$, the multiplication of variables by 3 increases the objective by a factor of $(1 + n\kappa)$.*

Proof. Let ρ_1, ρ_2, and ρ_3 be the amounts of traffic that are handled locally, in the pair, and in the controller, respectively. Thus, the objective of the modified linear program is $\psi_1 = d_1\rho_1 + d_2\rho_2 + d_3\rho_3$. Let ρ_i^1 be part of ρ_i that is multiplied by three and let $\rho_i^2 = \rho_i - \rho_i^1$. The objective after rounding is $\psi_2 = 3d_1\rho_1^1 + 3d_2\rho_2^1 + d_3(\rho_3 + \rho_1^2 + \rho_2^2)$. Thus, $\psi_2 = \psi_1 + \rho_1^1(2d_1) + \rho_2^1(2d_2) + \rho_1^2(d_3 - d_1) + \rho_2^2(d_3 - d_2) \leq \psi_1 + nd_1(\rho_1 + \rho_2) \leq \psi_1 + n\kappa\psi_1$. ■

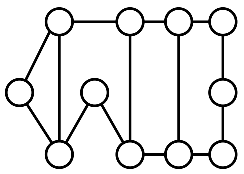
Lemma 4. *Let ρ and q be the ratio of largest to smallest rates and memory slots between any pair of rules, respectively. The objective increases at most by a factor of $\frac{d_1 + d_3\rho q}{d_1\rho q + d_3}$, due to placing rules in the descending order of their number of dependants.*

Proof. RaSe ignores the number of slots required by a rule in its strategy to place rules. Consequently, it may select a rule with large q_r and small ρ_r over a rule with small q_r and large ρ_r. The worst-case of this situation proves the lemma. ■

It is possible to characterize the approximation ratio of RaSe from Lemmas 1, 2, 3, and 4.



(a) Emulation Topology



(b) Simulation Topology

Fig. 3. Topology used for evaluations.

V. Performance Evaluation and Discussions

In this section, we first evaluate the proof-of-concept P4 implementation in Subsection V-A using emulation in Mininet. Then, we perform small-scale and large-scale simulations to compare RaSe with the optimal algorithm and existing baselines. All the instructions and codes required for reproducing the presented results are available in [17].

A. Emulation Setup

We implemented the topology depicted in Fig. 3a (showing the link delays and bandwidths) using P4 software switch (bmv2) and Mininet. We use `ping` and `iperf` to measure the delay between hosts and the bandwidth through uploading data from h1 to h2, respectively. We consider the following three placement schemes to show the effect of rule caching in the data plane:

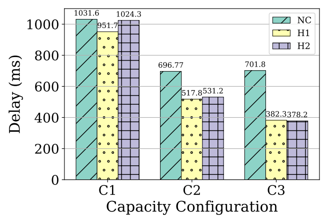
- **No Caching (NC)**: Switches are unable to utilize free capacity of neighboring switches.
- **Prioritize h1 (H1)**: Traffic destined to h1 has a higher priority to receive the free capacity of neighboring switches.
- **Prioritize h2 (H2)**: Similar to H1, however, traffic destined to H2 is prioritized.

In all schemes, switch S3 is the pair of switches S1 and S2, and all inbound traffic of the controller arrives through switch S3.

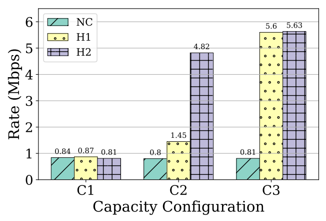
B. Emulation Results

We investigate the impact of rule caching on network traffic by considering three capacity configurations: C1, C2, and C3. In C1, the switches' memory is completely full, leading to all traffic being routed through the remote controller via the pair switch S3. As depicted in Figures 4a and 4b, all placement schemes achieve a maximum end-to-end delay of approximately one second and a minimum upload rate of around 0.8 Mbps. In C2, each switch has a single empty memory slot. Figure 4a reveals that it is possible to avoid controller involvement in the communication between hosts h1 and h2 in one direction. As a result, the end-to-end delay is reduced to approximately 500 ms. Furthermore, traffic destined for h2 bypasses the bottleneck links between switches and the controller, resulting in an upload rate of 4.82 Mbps for h2. This outcome demonstrates the significance of the rule placement strategy in improving delay and throughput.

In C3, switches S1 and S2 have a single available memory slot and switch S3 has two available memory slots, which is sufficient to completely avoid sending the traffic to the controller. Therefore, under both rule placement schemes H1 and H2, the end-to-end delay reduces to about 380 ms and the upload rate increases to a maximum of 5.6 Mbps. We conclude that it is possible to significantly improve the quality of service by utilizing the available capacity dispersed across the network through a collaborative mechanism.

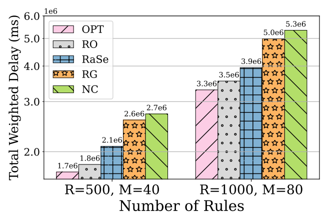


(a) Delay.

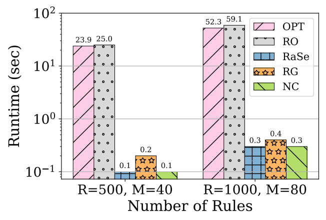


(b) Rate.

Fig. 4. The attainable rate and delay between two hosts under different capacity and rule placement configurations.



(a) Delay.



(b) Rate.

Fig. 5. Comparison of attainable objective and runtime with the optimal solution.

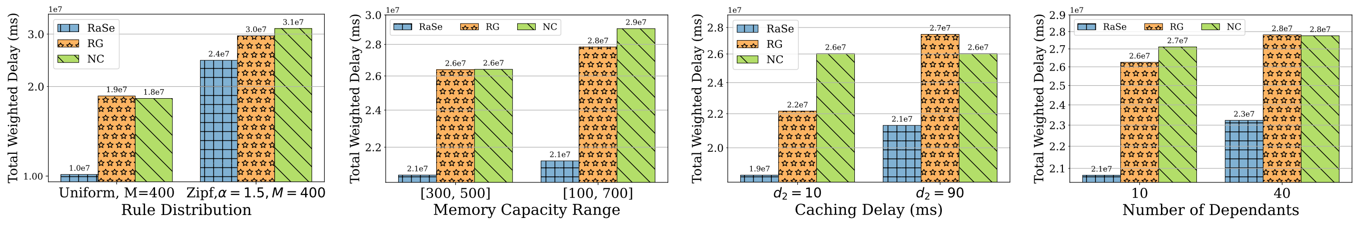
C. Simulation Setup

We implemented the algorithms in Python 3.10 and used SCIPY [8] as the solver for ILP and LP tasks. We employ the Abilene network topology [18,19], depicted in Fig. 3b. We compare RaSe with the following baselines:

- **Optimal (OPT)**: The solution obtained from the ILP solver.
- **Random Pairing, Optimal Placement (RO)**: This method represents the strategy of the paper by Rottenstreich et al. [2]. The switches are paired randomly and then rules are placed optimally using the ILP solver.
- **Random Pairing, Greedy Placement (RG)**: Similar to RO, this algorithm pairs the switches randomly. Then, it places rules in a greedy manner by first selecting the original switch, then the pair, and finally the controller.
- **No Caching (NC)**: This baseline represents the standard SDN scenario where collaborative rule caching is not available.

D. Simulation Results

Optimality and Scalability. Figure 5 shows the result of algorithms as the number of rules is increased from 500 to 1000. OPT and RO achieve the lowest weighted delay as they use the solver to place rules optimally. However, these methods are not applicable in delay-sensitive environments as their runtimes reach 60 seconds for placing one thousand rules. RaSe is able to perform the placement of 500 and 1000 rules in about 0.1 and 0.3 seconds, respectively. RaSe is 21% and 11% worse than OPT and RO, while being at least 170 times faster. Figure 5 shows that RaSe is about 20% and 26% better than RO and NC, respectively.



(a) Load Distribution. (b) Capacity Distribution. (c) Caching Delay. (d) Rule Dependency.

Fig. 6. The attainable rate and delay between two hosts under different capacity and rule placement configurations.

In the following, we perform multiple benchmarks to test the effect of different factors on the performance of RaSe. Benchmarks consider 5000 rules, where OPT and RO do not terminate after hours of wait. Thus, we omit them from the plots.

Effect of Load Imbalance. We select the original switch of the rules by using two distributions Uniform and Zips with parameter $\alpha = 3$. Uniform creates a balanced load while Zipf creates a highly imbalanced load. Figure 6a shows that the weighted delay of RaSe increases by a factor of 1.5 as the load becomes imbalanced. RG and NC also experience a similar effect as they increase by a factor of 2.4 and 1.7, respectively. Also, we observe that RaSe achieves a better result than RG and NC, improving the objective by at least ten percent compared to them.

Effect of Capacity Imbalance. In this benchmark, we investigate the effect of capacity imbalance on the performance of the algorithms. We consider two cases by specifying the capacity of switches by drawing random samples from two intervals $[300, 500]$ and $[100, 700]$. The former shows a uniform memory availability, while the latter shows a more imbalanced network. In Fig. 6b, we see that as the available resource becomes more asymmetric, the pairing and placement strategy becomes more important and influential. Specifically, in the uniform case, RaSe improves RG by about 5 percent, however, the improvement reaches about 15 percent in the imbalanced environment. Also, the effect of rule caching is prominent as both caching algorithms improve the NC by at least 13 percent.

Effect of Caching Delay. In this benchmark, we consider two extreme cases for the caching delay: $d_2 = 10\,\text{ms}$ and $d_2 = 90\,\text{ms}$. As expected, Fig. 6c shows that NC does not change in these two cases. Interestingly, RG becomes even worse than NC, as it greedily uses the space of neighbors, which can be used to store local rules of that switch which incurs lower delays. We can observe that even at $d_2 = 90$ ms RaSe performs better than both RG and NC for $d_2 = 10\,\text{ms}$, demonstrating its effectiveness at utilizing the network-wide resources.

Effect of Dependency. Figure 6d shows the weighted delay as the number of dependants changes from 10 to 40. Although the delay under RaSe increases by 10% it is still about half of delay under RG, and NC. Evidently, RaSe handles high levels of dependency among rules well and finds significantly better solutions compared to both RG and NC.

VI. Conclusion

This paper introduces RaSe, an approximation algorithm that leverages rounding of ILPs for pairing switches to cache rules in P4-enabled SDNs. We detail the design and P4 implementation of a caching-capable switch, evaluating its performance in a Mininet-based emulation. Extensive numerical simulations assess RaSe's scalability and effectiveness, especially under uneven network loads. Future research could explore scenarios involving multiple switch pairings or non-immediate neighbors.

References

[1] J.-P. Sheu, W.-T. Lin *et al.*, "Efficient TCAM rules distribution algorithms in software-defined networking," *IEEE Trans. Netw. Serv. Manag.*, vol. 15, no. 2, pp. 854–865, Jun. 2018.

[2] O. Rottenstreich, A. Kulik *et al.*, "Data plane cooperative caching with dependencies," *IEEE Trans. Netw. Serv. Manag.*, vol. 19, no. 3, pp. 2092–2106, Sep. 2021.

[3] Y. Wan, H. Song *et al.*, "T-Cache: Efficient policy-based forwarding using small TCAM," *IEEE/ACM Trans. Netw.*, vol. 29, no. 6, pp. 2693–2708, Dec. 2021.

[4] X.-N. Nguyen, D. Saucez *et al.*, "Rules placement problem in OpenFlow networks: A survey," *IEEE Commun. Surv. Tutor.*, vol. 18, no. 2, pp. 1273–1286, 2016.

[5] T. V. Phan, M. Hajizadeh *et al.*, "Destination-aware adaptive traffic flow rule aggregation in software-defined networks," in *Proc. IEEE NetSys*, 2019, pp. 1–6.

[6] M. Rifai, N. Huin *et al.*, "Too many SDN rules? Compress them with MINNIE," in *Proc. IEEE GLOBECOM*, 2015, pp. 1–7.

[7] P. Bosshart, D. Daly *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.

[8] SciPy, "Fundamental algorithms for scientific computing in Python." [Online]. Available: https://scipy.org/

[9] S. Misra, N. Saha *et al.*, "Traffic-aware rule-cache assignment in SDN: Security implications," in *Proc. IEEE ICC Workshops*, 2020, pp. 1–6.

[10] J.-P. Sheu and Y.-C. Chuo, "Wildcard rules caching and cache replacement algorithms in software-defined networking," *IEEE Trans. Netw. Serv. Manag.*, vol. 13, no. 1, pp. 19–29, Mar. 2016.

[11] I. Gozlan, C. Avin *et al.*, "Go-to-controller is better: Efficient and optimal lpm caching with splicing," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 7, no. 1, 2023.

[12] Q. Li, N. Huang *et al.*, "HQTimer: a hybrid {Q}-learning-based timeout mechanism in software-defined networks," *IEEE Trans. Netw. Serv. Manag.*, vol. 16, no. 1, pp. 153–166, Mar. 2019.

[13] A. Vishnoi, R. Poddar *et al.*, "Effective switch memory management in OpenFlow networks," in *Proc. ACM Int. Conf. on Distr. Event-Based Syst. (DEBS)*, 2014, pp. 177–188.

[14] K. Kannan and S. Banerjee, "Flowmaster: Early eviction of dead flow on SDN switches," in *Proc. ICDCN*, 2014, pp. 484–498.

[15] J. Dallot, A. J. Fesharaki *et al.*, "Dependency-aware online caching," in *IEEE INFOCOM*, 2024, pp. 871–880.

[16] N. Karmarkar, "A new polynomial-time algorithm for linear programming," *Combinatorica*, vol. 4, no. 4, p. 373–395, Dec 1984.

[17] https://github.com/mmdsbri98/Efficient-Rule-Caching/. Accessed Aug. 23, 2024.

[18] S. Knight, H. Nguyen *et al.*, "The internet topology zoo," *IEEE J. Sel. Areas Commun.*, vol. 29, no. 9, pp. 1765 –1775, october 2011.

[19] L. Li, D. Alderson *et al.*, "A first-principles approach to understanding the Internet's router-level topology," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 4, p. 3–14, Aug. 2004.