CoLUE: Collaborative TCAM Update in SDN Switches

Ruyi Yao*, Cong Luo*, Hao Mei*, Chuhao Chen*, Wenjun Li†¶, Ying Wan‡, Sen Liu*, Bin Liu§, Yang Xu*¶
*School of Computer Science, Fudan University, China †Harvard University, USA §Tsinghua University, China
‡China Mobile (Suzhou) Software Technology Co., Ltd, China ¶Peng Cheng Laboratory, China

IEEE INFOCOM 2023 - IEEE Conference on Computer Communications | 979-8-3503-3414-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/INFOCOM53939.2023.10229074

***Abstract*—With the rapidly changing network, rule update in TCAM has become the bottleneck for application performance. In traditional software-defined networks, some application policies are deployed at the edge switches, while the scarce TCAM spaces exacerbate the frequency and difficulty of rule updates. This paper proposes CoLUE, a framework which groups rules into switches in a balance and dependency minimum way. CoLUE is the first work that combines TCAM update and rule placement, making full use of TCAM in distributed switches. Not only does it accelerate update speed, it also keeps the TCAM space load-balance across switches. Composed of ruleset decomposition and subset distribution, CoLUE has an NP-completeness challenge. We propose heuristic algorithms to calculate a near-optimal rule placement scheme. Our evaluations show that CoLUE effectively balances TCAM space load and reduces the average update cost by more than 1.45 times and the worst-case update cost by up to 5.46 times, respectively.**

I. Introduction

Software-Defined Networking (SDN) [1] provides an abstraction of the network and enables centralized control of distributed switches. A variety of applications (such as flow engineering, access control and traffic monitoring) are supported by customizing the ruleset in switches. As the most widely used memory in SDN switches to store the flow table, Ternary Content-Addressable Memory (TCAM) [2] is designed for unparalleled high lookup throughput rather than fast update. However, with the rapidly changing network, TCAM update has become the main bottleneck for performance.

Network applications are placing increasing demands on the frequency and speed of SDN rule updates. The emerging Intent-Driven Networking (IDN) [3] and autonomous networks apply faster rule updates in real time intelligently. Besides, machine learning applications have become a trend to meet various objectives, which frequently adapt policies to the dynamically changing network states [4] [5]. Sudden events such as outages, Denial of Service attacks, or flash crowds also bring burst updates. Meanwhile, for networks with high real-time requirements, rules have to be installed within 25ms to meet stringent Quality of Service (QoS) requirements [6]. Traffic engineering SDN control programs such as Google's B4 [7] and Microsoft's SWAN [8] leave only a 20ms time budget for flow table update. In security systems and critical infrastructures, instant response is even more vital [9] [10].

It is indispensable to make TCAM updates fast to meet application requirements. However, TCAM update is a great challenge. To ensure semantic correctness, rules with overlapped match fields must be placed in descending order of priority, which makes rule insertion the most time-consuming operation. A rule insertion can cost as long as 100ms [11]. While the update is in progress, packet lookup has to be suspended for consistency. The longer the update operation takes, the greater the packet delay, which potentially causes packet loss and deteriorates application quality.

The gap between TCAM update performance and application requirements is further broadened due to the limited TCAM capacity. The total number of rules in a traditional enterprise network can reach 8 million [12], and even 1 billion in the public cloud [13]. In contrast, TCAMs in commercial switches can only store 750 to 20k rules [14], [15]. The shortage of TCAM capacity causes the rules being frequently swapped in and out, triggering a large number of updates [16]. Meanwhile, full of rules in a TCAM chip makes updates even slower.

To accelerate TCAM update and mitigate TCAM space burden, we propose a novel framework called CoLUE, which combines TCAM update with the rule placement. CoLUE roots in two observations: (1) the less overlapping of rules, the faster rule update in a single TCAM [17], [18]; (2) the more balanced TCAM space load, the smaller probability of overflow when new rules come. By splitting ruleset and distributing sub-rulesets in an overlapping minimum and balanced way among multiple switches, not only can CoLUE reduce the update cost, but also activates less-used TCAMs to spread the load.

To the best of our knowledge, CoLUE is the first framework to coordinate multiple switches to accelerate TCAM updates. It utilizes a delicate ruleset placement scheme to improve TCAM update efficiency while balancing TCAM space load. Although the rule placement problem has been extensively studied [12]–[14], [19]–[23], existing solutions neglect the final step: installing rules into TCAMs. They pay no attention to rule overlapping when making a rule placement plan, which is the root cause of slow updates in switches. CoLUE faces several

This work is supported by Key-Area Research and Development Program of Guangdong Province (2021B0101400001), NSFC (62150610497, 62172108, 61872213, 62032013, 62272258, 62002066, 62102203), NSFC-RGC (62061160489), Open Research Projects of Zhejiang Lab (2022QA0AB07), Shanghai Pujiang Program (2020PJD005), the Major Key Project of PCL (PCL2021A15, PCL2021A02), National Key Research and Development Program of China (2022ZD0115303), Basic Research Enhancement Program of China (2021-JCJQ-JJ-0483), and China Postdoctoral Science Foundation (2020TQ0158, 2020M682825, PC2021037). Corresponding author: Yang Xu (xuy@fudan.edu.cn)

new challenges for accelerating TCAM updates and proposes effective heuristic algorithms. The framework is orthogonal to rule insertion algorithms in a TCAM [17], [18], [24], [25], and many existing insertion algorithms can be used in it.

In summary, we make the following contributions:

- We design a novel framework CoLUE, which coordinates multiple switches to bring faster updates and better TCAM space load-balance. It can cooperate with many existing rule insertion algorithms for a single TCAM.
- To accelerate TCAM update speed, CoLUE decomposes rulesets into multiple subsets with minimum dependencies, which is a relaxed graph coloring problem with NP-completeness. Consideration for balance distribution to switches also adds difficulty, especially when the topology is complex. Bron-Kerbosch-Cost and Weighted Space Allocation algorithms are heuristically proposed. They work collectively to keep sub-ruleset balance with minimum dependency.
- The advantages of CoLUE are demonstrated through simulations on three network topologies. Compared with rule placement algorithms neglecting TCAM updates, CoLUE reduces the average update cost by more than 1.45 times and the worst cost by up to 5.46 times.

II. Background and Motivation

A. TCAM Update Challenge

Application policies are typically translated into prioritized ruleset R and stored in a flow table. These rules can be abstracted as $r = (match, action, pri)$ [26]. $r.pri$ denotes the priority of a rule, and a smaller number indicates a lower priority. Rules in a flow table may overlap (i.e., $r_i.match \cap r_j.match \neq \varnothing$). When a packet arrives at the switch, the matched rule with the highest priority is found by querying the flow table and the corresponding action is performed on the packet. For example, a policy is listed in Fig. 1 (a). Actions are omitted for generality. Specified by match fields F_1 and F_2, each rule is embodied as a rectangle on a 2D plane as shown in Fig. 1 (b). A new packet $f(F_1 = 000, F_2 = 100)$ falls in the overlapping regions of r_2 and r_5. $r_2.pri$ is higher than $r_5.pri$ so r_2 should be taken.

TCAM is the most widely used memory in a switch to store the flow table. All rules are compared with the search key simultaneously. When multiple rules are matched, TCAM will return the first of them. Therefore, overlapping rules must be placed in TCAM in descending order of priority, so that the highest priority matching rule can be returned. The relative rule position constraint is named Priority Order Constraint (POC) [27]. Slow insertion is attributed to POC [17], which requires many rules to be moved to make room for a newly inserted rule. The insertion time is proportional to the number of TCAM accesses, which equals movement steps plus 1. The more rules need moving, the longer the rule insertion time and the slower the update speed. Hereafter we use movement steps to indicate update speed. Deletion and modification are not considered as they are simple operations [28].

POC is modeled as the rule dependency. If $r_i.match \cap r_j.match \neq \varnothing$, and $r_i.pri > r_j.pri$, we say r_j is dependent on r_i and use $r_i \rightarrow r_j$ to denote the dependency. The TCAM entries storing the rules should conform to the condition:

$$\forall r_i, r_j \in R,\ if\ r_i \rightarrow r_j, A[r_i] < A[r_j] \tag{1}$$

$A[r]$ is the TCAM address of rule r. Relationship $\rightarrow$ is transitive. If $r_i \rightarrow r_j \wedge r_j \rightarrow r_k$, even though $r_i.match \cap r_k.match = \varnothing$, r_i must be put above r_k. We call r_k is indirectly dependent on r_i. A movement sequence is valid as long as the condition is satisfied during rules movement.

A directed acyclic graph (DAG) is commonly used to describe all the direct dependencies in the flow table [29]–[31]. Fig. 1 (c) shows the DAG for Fig. 1 (b). Each node represents a rule. If r_j is dependent on r_i, a directed edge is formed, pointing from r_i to r_j. We call their relationship ancestor and child. Paths from the oldest ancestors to the youngest children are dependency chains. Many chains are intertwined to form the DAG. The number of movements for a rule insertion is bounded to the longest chain [18]. We use the longest chain length to indicate dependency degree, which is represented by LC. If rules in a TCAM are all independent, there is no chain and $LC = 0$. In this case, when a new rule comes, at most one movement is needed to make space for its insertion. So $LC = 0$ is the optimal case. Generally speaking, the shorter the chain lengths, the smaller the cost of rules insertion [18]. We are committed to reducing LC as much as possible. For one thing, a small LC means that the worst-case cost of a rule insertion is small. For another thing, minimizing the longest chain squeezes the length differences between chains. We can expect that all chains are small, and the average cost of rule insertions is low.

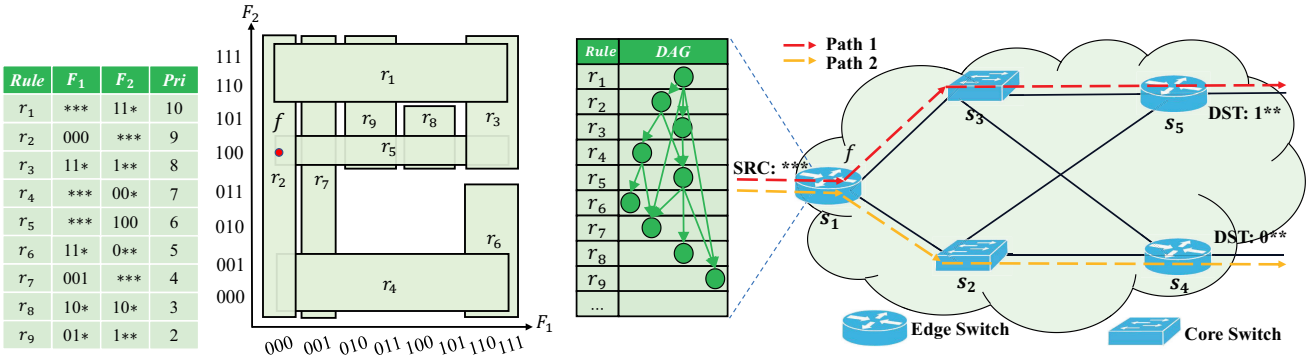
B. Decomposable Rulesets

Various applications have different policies according to the objectives. Policies can be divided into routing policies and endpoint policies based on objectives, which are undecomposable and decomposable respectively.

An endpoint policy specifies the action to process each packet entering the network. Firewall and ACL policies are typical representatives. They are often installed in ingress switches rather than core switches or egress switches, such as Google Compute Engine's firewall and Amazon's EC2 security group. Policies stored in ingress switches can be differentiated by VLAN tag [21]. The ruleset is allowed to be distributed along the flow path as long as the entire policy is applied to packets before leaving the network. So endpoint policies are decomposable. We are devoted to decomposing the endpoint policies and placing the sub-rulesets in all switches to reduce update time and improve TCAM space load balance while maintaining flow table lookup correctness. We do not deal with routing policies.

C. Motivation

We propose CoLUE, a framework that combines TCAM update with the rule placement. Here is an example demon-



(a) (b) (c)

Fig. 1: An Example of Policy

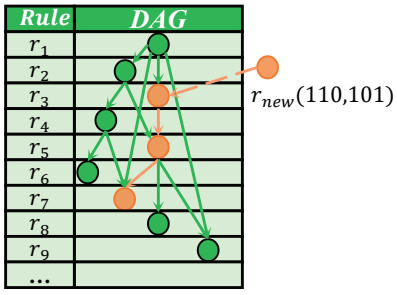


Fig. 2: New rule insertion

strating our basic idea and the effect of collaborative TCAM updates among switches.

Suppose the ruleset in Fig. 1 (a) is a generic endpoint policy, and F_1 and F_2 are source and destination address respectively. A traditional policy placement is shown in Fig. 1 (c), in which s_1 is the ingress switch, s_4 and s_5 are egress switches, with others core switches. Flows traveling along path 1 and path 2 must conform to the endpoint policy.

If a new rule $r_{new} = (F_1 = 110, F_2 = 101, pri = 9)$ is to be added in the policy, one of the most efficient movement sequence to make room for r_{new} is $r_3 \rightarrow r_5 \rightarrow r_7$, which needs three movements (four TCAM accesses), as colored in orange in Fig. 2. As the number of rules in s_1 continues to grow, the dependencies of rules will get far more complex, consequently the update delay is prolonged.

However, if we decompose the endpoint policy into multiple sub-rulesets and distribute them along paths, the dependency degree of rules in each switch can be minimized, and switches are not easily overloaded.

Since endpoint policy in the ingress switch must be applied to all packets to realize application requirements, sub-rulesets should be placed in all paths originating from s_1, introducing unnecessary rules installation and rule replication in the network. Luckily, only part of the rules is useful in each path, and we can store only useful rules along a path [20], [21]. For example, $U_1 = \{r_1, r_2, r_3, r_5, r_7, r_8, r_9\}$ should be applied to flows whose destination addresses overlap with 1**, so U_1 is supposed to be placed along path 1. Instead, flows traveling along path 2 will not match U_1, so it is wasteful to place U_1 along path 2. Similarly, $U_2 = \{r_2, r_4, r_6, r_7\}$ is supposed to

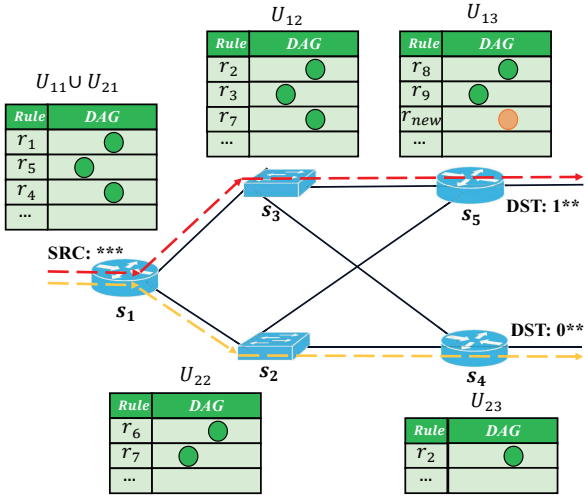


Fig. 3: Ideal Rule Placement Scheme

be placed along path 2 rather than path 1.

Both path 1 and path 2 have 3 switches, and we divide U_1 into relatively balanced and dependency-minimum subsets $U_{11} = \{r_1, r_5\}, U_{12} = \{r_2, r_3, r_7\}, U_{13} = \{r_8, r_9\}$ and U_2 into $U_{21} = \{r_4\}, U_{22} = \{r_6, r_7\}, U_{23} = \{r_2\}$. Each sub-ruleset is placed in one and only one switch along the corresponding path without order. The ideal rule placement scheme is shown as Fig. 3, with $LC = 0$ for all switches. Because packets matching r_{new} should pass along path 1, r_{new} is to be inserted in any switch in path 1. The best choice is to insert r_{new} into s_3, causing 0 movements.

By placing rules with a low dependency degree in a switch, the number of rules in each switch is reduced and the update speed is increased. When a large amount of new rules come, multiple switches can install rules in parallel, further improving the update. CoLUE is able to achieve extremely fast updates with the balanced and dependency-minimum decomposition of rulesets. Packets must be processed by the highest priority rule instead of others and the CoLUE lookup process is shown as below.

D. Proposed Lookup Procedure

When a packet traverses path 1, we first look up the flow table in the first switch on the path, and load the priority and

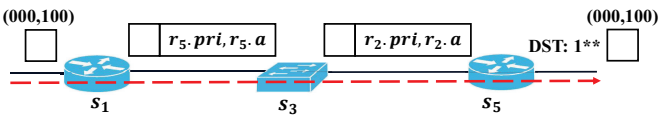


Fig. 4: Lookup process

action of the returned rule into the packet header. Arriving at the next switch, if the returned rule has a higher priority than the load, it replaces the old one, otherwise no change happens. This operation continues until leaving the egress switch. Before the packet leaves the egress switch, the action carried is performed on it and the load is removed. Fig. 4 shows the example. New packet $f(F_1 = 000, F_2 = 100)$ should travel along Path 1 in Fig. 3, which consists of s_1, s_3, s_5. Packet f enters the network from s_1 and r_5 is matched, it carries $r_5.pri$ and $r_5.action$ ($r_5.a$ in short) along the path. At s_3, it matches r_2 and $r_2.pri > r_5.pri$, so it changes the load into $r_2.pri$ and $r_2.a$. None of rules in s_5 are matched, and the load stays unchanged. Before leaving the network, $r_2.a$ is performed on the packet, which is the same as described in Section II-A and consistent with the policy. Finally, $r_2.pri$ and $r_2.a$ are removed from the packet header.

III. Overview of CoLUE

CoLUE aims to find a good rule placement scheme for endpoint policies to improve update speed and balance TCAM space load. Rulesets in switches should have minimum overlapping and balanced sizes. In this section, problem formulation for the rule placement problem is shown, and we emphasize the challenges of optimizing update performance.

A. Problem Formulation

Given a topology $G(S, L)$, S is the set of switches and L is the set of links. The set of ingress switches is represented by S^{in}. We deem policies attached with different ingress switch s_i^{in} as one large policy Q. Suppose $\mathbb{P}$ is the set of paths in G, U_i is the set of rules from Q which are useful along P_i. $U_i = \{r_j^i, j = 1, ..., N_i\}$, $P_i = \{s_k^i, k = 1, ..., M_i\}$. N_i is the number of rules in U_i and M_i is the number of switches in P_i. Only U_i has to be distributed along P_i. We intend to find a mapping from all rules in U_i to switches in P_i, namely the rule placement scheme. Define all rules assigned to s_k is R_k, which are from all paths across s_k. The part of rules belonging to U_i is R_k^i. We build DAG_k and DAG_k^i respectively for R_k and R_k^i, with dependency degree LC_k and LC_k^i. CoLUE has 2 objectives and 2 constraints when decomposing U_i and distributing rules. Both objectives are applied in each path P_i since we choose to place rules based on paths.

Objective 1: minimize the maximal dependency degree in switches along a path. Thus when new rules come, the worst update time of a rule is bounded and the average update time is shortened. In CoLUE, the rule placement scheme is **not** an Integer Linear Programming (ILP) problem as LC cannot be represented linearly.

$$\min \max_{\forall k \ \ s_k^i \in P_i} LC_k^i \tag{2}$$

Objective 2: keep TCAM space load balance across switches along a path. The more balance, the less overflow and unnecessary rules in and out. A balance TCAM space load is also good for fast batch updates.

$$\min \max_{\forall k \ \ s_k^i \in P_i} |R_k| \tag{3}$$

Optimization should be achieved while satisfying the following constraints.

Constraint 1: rules installed in a switch should not exceed the capacity.

$$\forall s_k \in S, \ |R_k| < s_k.capacity \tag{4}$$

Constraint 2: semantic correctness requires that all useful rules are installed along paths.

$$\forall i \ U_i \subset \cap_{\forall k \ \ s_k^i \in P_i} R_k^i \tag{5}$$

B. Challenges of CoLUE

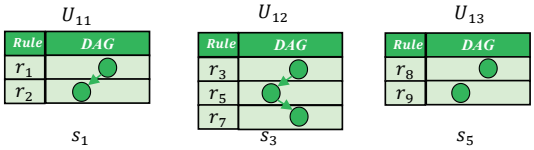
In pursuit of high-speed updates and TCAM space load balance, we should provide rule decomposition and subset distribution algorithms for a good rule placement. CoLUE faces three challenges.

1) Challenge 1: Ruleset Decomposition For Minimum Dependency Degree: Our main objective is to minimize the dependency degree in a TCAM by putting rules with the least overlapping together. One sub-ruleset will only be installed in a switch. Generally, the more subsets, the more positions each rule can choose, thus the easier it is to minimize the dependency degree and distribute evenly. The maximal number of subsets that U_i can be divided into is the path lengths M_i. In ruleset decomposition, we always split U_i into M_i subsets.

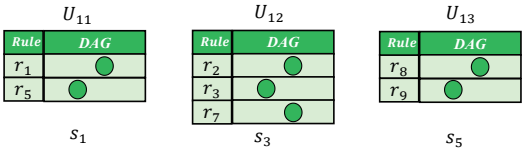
To minimize the dependency degree in each TCAM, we decompose U_i into M_i sub-rulesets $\{U_{ij}, j = 1, ..., M_i\}$, and put the j^{th} sub-ruleset in s_j^i[1]. Rules in U_{ij} should have as least LC_j^i as possible. Fig. 5 gives instances of decomposing U_1 and U_2 in Fig. 3. The former two sub-figures are the decomposition of U_1 and the last one of U_2. Obviously, decomposition scheme shown in Fig. 5 (b) preserves less dependency than (a), with $\max LC$ are 0 and 2 respectively. If a new rule is inserted in U_{12} in Fig. 5 (a), in the worst case it will bring 3 movements, while in Fig. 5 (b), it at most causes one rule movement. The dependency degrees shown in Fig. 5 (b) and (c) are all zeros and they are the best decomposition of U_1 and U_2 respectively.

Essentially, ruleset decomposition is to take partitions of nodes from DAG, and places each of them in a switch. Nodes in a partition generate the induced sub-DAG. The pursuit of edgeless ($LC = 0$) sub-DAGs makes ruleset decomposition a graph coloring problem. Graph coloring problem colors adjacent nodes with different colors. In ruleset decomposition, rules painted in the same color are placed in a switch. When the number of switches is smaller than the chromatic number of the graph, it is impossible to make all induced sub-DAGs edgeless. We want the chain lengths in induced sub-DAGs to be as small as possible. Under such a circumstance, ruleset decomposition is a relaxed graph coloring problem, allowing

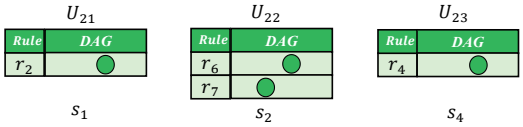
[1]Sub-rulesets are disordered in deed, and we number them for ease of description.



(a) A Bad Decomposition of U_1



(b) The Best Decomposition of U_1

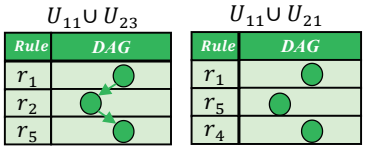


(c) The Best Decomposition of U_2

Fig. 5: Rule Decomposition

nodes with edges to be painted in the same color while minimizing the consecutive length of the same colored nodes (the longest chain). Graph coloring is a well known NP-Complete problem and relaxed graph coloring is no easier than it. As it is more common that there are not enough switches, CoLUE is expected to solve the relaxed graph coloring problem. So how to split ruleset into sub-rulesets with minimum dependency degree is a challenge.

2) Challenge 2: Tradeoff Between Objectives: The optimal decomposition method (shown in Challenge 1) may lead to unbalance when pursuing the minimum dependency in each switch, while a balanced decomposition may lead to a larger dependency degree. If P_1 has only 2 switches and U_1 stays unchanged, the decomposition for minimum dependency of U_1 is $U_{11} = \{r_2, r_3, r_6, r_7, r_8, r_9\}$ and $U_{12} = \{r_1, r_4, r_5\}$. The size of U_{11} is twice that of U_{12}, which is unbalanced. Moving a rule from U_{11} to U_{12} can strike a balance, but all rules in U_{11} will bring dependency to U_{12}, failing to minimize the dependency degree. Sometimes it is impossible to achieve the best of the two objectives at the same time, and how to make a tradeoff is a challenge.



(a) Bad Share (b) Good Share

Fig. 6: Rules in Shared Switches

3) Challenge 3: Stress in shared switch: If a switch lies at multiple paths, we call it a shared switch and others non-shared switches. The existence of shared switches adds difficulty for CoLUE. For one thing, it is hard to keep load balance among shared switches and non-shared switches. For another thing, subsets from different paths which are placed in the shared switches may generate dependencies. Fig. 6 (a) shows such a case. The best decomposition of U_1 and U_2 is shown in Fig. 5 (b) and (c), with all the sub-rulesets dependency-free. However, as the sub-rulesets are disordered, chances are that U_{11} and U_{23} are installed in s_1, prolonging LC_1 to 2. If U_{11} and U_{21} are installed in s_1, they will not form dependency and LC_1 is 0. DAG for the good case is shown in Fig. 6 (b). How to avoid the unbalance and rule dependency in shared switches is another challenge.

IV. Algorithm

The rule placement of CoLUE consists of two components: ruleset distribution and ruleset decomposition. The former computes the rule-space allocation in switches for global balance. The latter splits the ruleset into multiple sub-rulesets with minimum dependencies.

Since the three challenges are closely related to both sub-ruleset distribution and ruleset decomposition and NPC problem exists, CoLUE tackles three challenges heuristically to get an ideal solution. The rule placement of CoLUE can be broken down into 3 sub-problems from simple to difficult: (1) how to minimize dependency degree along a single path; (2) how to minimize dependency degree under balance constraint along a single path; and (3) how to minimize dependency degree under balance constraint over multiple paths. We propose 3 schemes for the sub-problems respectively, and each of them mainly solves a challenge in Section III-B. We use CoLUEi to represent the three schemes respectively and CoLUE3 is the final state of CoLUE. ***'CoLUE' alone without a number mentioned in the paper refers to CoLUE3***.

A. CoLUE1 for Sub-problem 1

Scheme CoLUE1 focuses only on Challenge 1 along a path. Since a relaxed graph coloring problem that minimizes the dependency of the subset of U_i is NP-complete, we heuristically propose the Bron-Kerbosch-Cost (BKC) algorithm to decompose U_i.

First, the Bron-Kerbosch (BK) algorithm takes out M_i maximal independent subsets. Then, for the residual rules, put them in one of the M_i subsets based on the cost function.

$$Cost_k = lc_k^i \tag{6}$$

lc_k^i denotes the chain length in the k^{th} subset caused by the insertion of the rule. A rule will be placed in the subset in which it brings the least cost based on the idea of greed. Since only a path is considered, subsets can be placed in switches randomly. CoLUE1 achieves the least dependency degree compared to the other two schemes.

B. CoLUE2 for Sub-problem 2

Scheme CoLUE2 focuses on both dependency minimum and balance along a path, which corresponds to Challenge 2. Ruleset decomposition also applies the BKC algorithm in CoLUE2, but balance is taken into account. The maximum dependent set is taken out by the Bron-Kerbosch algorithm and divided evenly into M_i subsets. For the residual rules, the cost function is extended into

$$Cost_k = \alpha \cdot lc_k^i + \beta(\frac{n_k^i}{N_k^i} - 1)^3 \tag{7}$$

In the cost function, the first term is the same as Section IV-A. The second term corresponds to objective 2: TCAM

space load balance. n_k^i is the number of rules already placed in the k^{th} subset after the insertion of r and N_k^i is the expected sizes of subsets for the best balance. $N_k^i = \frac{N_i}{M_i}$ and it indicates that U_i should be evenly divided. The closer the ratio is to 1, the better the placement. So the second term encourages rules to be installed in switches whose ruleset sizes are smaller than expected. We use the cubic formula to imply that, when $n_k^i \leq N_k^i$, the larger the difference, the cost value decreases more sharply, and the more likely the k^{th} subset will be chosen.

BKC can balance the 2 objectives well by adjusting the ratio of coefficients according to application goals. Subsets can be placed in switches randomly as CoLUE2 only considers a single path.

C. CoLUE3 for Sub-problem 3

Scheme CoLUE3 is devoted to making rules in each switch as balanced as possible with as few dependencies as possible and it solves challenge 3. A complete balance may be impossible when there is a large difference in the sizes of the path useful rules, and CoLUE3 just does its best. It is the collaboration of ruleset decomposition and subset distribution that supports the goal of CoLUE3. In CoLUE3, shares of switches for paths are calculated first in Section IV-C1, then rulesets are decomposed into subsets with specified sizes and minimum dependency degree to guarantee the global balance and fast update in Section IV-C2.

1) Rule Distribution: To distribute rules in switches with global balance across multiple paths, we first calculate how much space a switch should spare for a path and then decompose the ruleset according to the share. Weighted Space Allocation (WSA) algorithm is proposed. It first assigns each switch a weight, the value of which is the total number of rules on the paths that pass through the switch. Namely $s_k.w = \sum_{\forall i} I(s_k \in P_i) \cdot N_i$. I is the indicator function which equals 1 when the input is true. Based on the weights, WSA then calculates how many rules each passing path P_i should place on s_k. A higher weight means that each path should assign fewer rules to avoid TCAM space overloading, so the rule number is determined by the percentage of the inverse of the weight. $W_k^i = \frac{1/s_k^i.w}{\sum_{j=1}^{j=M_i} 1/s_j^i.w}$, W_k^i is the share of rules from U_i that should be placed in s_k.

Here is an example for WSA. Suppose both Path 1 and Path 2 have 6k rules in Fig. 1. For switches in Path 1, s_1, s_2, s_3 are assigned the weight of 12k,6k,6k respectively and the $W_1^1 = \frac{1}{5}, W_2^1 = \frac{2}{5}, W_3^1 = \frac{2}{5}$. U_1 for Path 1 should be divided into 3 subsets with sizes ratios $1:2:2$.

2) Ruleset Decomposition: In rule decomposition, W_k^i outputed by WSA is used as the input, and the ratio of subset sizes should approach W_k^i.

As U_i has N_i rules and the subsets size ratio should be W_k^i, the expected rules in s_k is $N_k^i = N_i \cdot W_k^i$. In order to generate subsets with N_k^i rules, we further modify the BKC algorithm. Firstly, the maximum independent set is taken out by the Bron-Kerbosch algorithm. Then we partition it into M_i subsets according to W_k^i, and place subsets on the switches with corresponding space share. For the residual rules, the cost function is the same as before, with different expected rule numbers N_k^i. Although no longer do we decompose ruleset evenly, we can achieve balance across all switches.

Apart from balance, we should also pay attention to the dependency degree in shared switches. We explore the reasons for the phenomenon in Section III-B3. A U_i is determined by an (ingress switch, egress switch) pair. Define switches on multiple paths as shared switches, the common rules in U_i and U_j are shared rules and other rules are non-shared. Shared rules appear only when the ingress switch of P_i and P_j are the same. Suppose P_i and P_j intersects at the shared switch s_k, and U_{ik} and U_{jk} are put in s_k. Chances are that rules from U_{ik} and U_{jk} form a longer chain, namely $LC_k > \max\{LC_k^i, LC_k^j\}$.

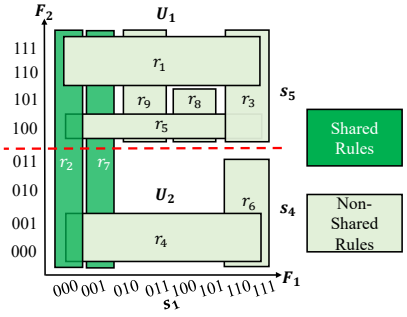


Fig. 7: Useful Rules Space

The coexistence of shared rules and non-shared rules in a switch is the culprit of this phenomenon. Fig. 7 shows an example. Rules r_2, r_7 are shared rules in U_1 and U_2. r_2, r_7 placed along P_1 can form dependency with non-shared rules r_4 placed along P_2 in shared switches. Similarly, r_2, r_7 placed along P_2 can form dependency with non-shared rules r_1, r_5 placed along P_1 (seen in Fig. 6 (a)). A simple way to avoid such cases is to place shared rules at the latest. We take shared rules from U_i first, then perform the BKC algorithm on the remaining rules. After the remaining rules are placed in switches, shared rules employ the cost function to choose a switch. Finally, in terms of balance and dependency degree, CoLUE3 gives a good solution.

Although the complexity of the BKC algorithm is large due to $O(3^{n/3})$ of BK, the ruleset in each path is much smaller than the original policies. Meanwhile, the rule-placement computation of multiple paths is independent and can be accelerated with multi-threaded parallelism, which further mitigates the time consumption. Moreover, BK is only performed once for a network to install policies.

D. Dynamic Update

After presenting the rule placement scheme of CoLUE, we show the update design. The network changes rapidly over time. When new rules come, we should update incrementally rather than re-computation to reduce update delay. Update cases can be (1) a policy update such as updating a firewall, (2) policy installation when new switches join the network and bring new policies or (3) a path change such as mobile destination moves [21].

For the three cases, re-performing CoLUE for an optimal rule placement is too time-consuming. We regard rules that need to be (re)installed as batch update requests, and install them based on the existing rule placement. The cost function is extended to support batch update scenarios. We first determine the target paths for rules and then employ the extended cost function. Rule insertion on multiple paths can be processed in parallel and we consider a single path.

Suppose the set of new rules to be inserted on the target path is R', and the size is $|R'|$. If $|R'|$ is larger than the remaining capacities of switches on the target path, CoLUE must be re-implemented to reallocate switches' share for rules. Otherwise, the choices of switches for rules are based on the extended cost function:

$$Cost_k = \alpha \cdot lc_k^i + \beta(\frac{n_k^i}{N_k^i} - 1)^3 + \gamma(\frac{c_k^i}{C_k^i} - 1)^3 \qquad (8)$$

The definition of parameters is the same as that of the BKC algorithm, and new parameters c_k^i, C_k^i are introduced to improve the concurrency of rule installation among switches, denoting the number of new rules inserted into s_k and the expected load respectively. $C_k^i = \frac{|R'|}{M_i}$. The closer the third term is to 1, the better the placement. This term avoids that some switches are busy installing rules while others are idle when rules need to be updated. The fewer new rules are assigned to a switch, the higher probability that a new rule will be placed in it. If there is only one new rule, $\gamma = 0$.

As the calculation of lc_k^i utilizes Depth-First-Search, the cost function can be calculated fast. With parallel installation in switches, batch updates can be performed rapidly.

When the number of rules in the network exceeds the total capacity of all switches, CoLUE fails to handle. We leave the rule replacement scheme for high hit rates as future work.

V. Evaluation

Since CoLUE is a framework which only focuses on ruleset decomposition and sub-ruleset distribution, the existing rule insertion algorithm Chain [17] is employed to place rules in TCAM. The network is initialized with the rule placement scheme and then new rules continue coming for insertion during all the experiments. The main metrics used in experiments are update cost (# of TCAM accesses) and relative standard deviation (RSD) of TCAM space load. Updates focus on insert cases in evaluation, as insert operations are the most expensive. We use CoLUEi_max to denote the worst update cost and use CoLUEi to indicate the average cost of the corresponding scheme. The naming of legend is similar to this in the comparison schemes. The update cost is mainly determined by the degree of dependency in a TCAM and the parallel processing of multiple switches, which can be reflected by serial and parallel updates. In serial insertion, we record the exact update cost of each rule which reflects the single TCAM update speed, and in parallel insertion, we focus on the finish time of a batch rules insertion and calculate how many update costs per rule on average.

We construct three topologies: (1) a small random topology with 10 switches, (2) a large random topology with 50 switches, and (3) a fat tree with pod size 4 and 20 switches in total. ACL rulesets are generated by ClassBench [32] with sizes ranging from 6k to 25k. Algorithms are implemented using C/C++ language and compiled by g++. Simulators are run on a commodity server with the Ubuntu 18.04-LTS operating system.

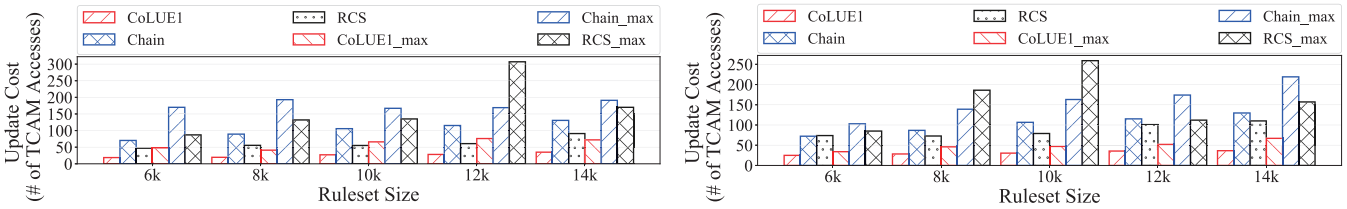
We first show the reduction of update cost to demonstrate the advantage of minimum dependency with serial updates. Then we demonstrate the advance of CoLUE compared with the rule placement algorithm OBS [20] and ETRD [23] with batch updates.

A. Comparison with Ingress-Switch-Close Schemes

Some rule placement methods tend to put rules close to ingress switches, while CoLUE fully utilizes all switches. Although CoLUE can insert rules in switches in parallel, we calculate the average and worst update cost of a rule serially to reflect the effect of minimum dependency using CoLUE1. Fig. 8 (a) and (b) are the update cost in Topology 1 and 3 respectively. As the performance trend in Topology 1 and 2 are similar, we do not show the metrics in Topology 2 due to space limits. Chain represents the scheme where rules are stored only in TCAM of ingress switches, assuming a single TCAM is capable. RCS [33] breaks the ILP rule placement problem into multiple independent ILP subproblems and prefers placing rules close to ingress switches along a path. We assign the same practical capacity of a single TCAM to RCS and CoLUE1. In Fig. 8 (a), the average cost of CoLUE1 is reduced by 2.9 times and 1.2 times compared with Chain and RCS respectively. The worst cost of CoLUE1 is reduced by 1.8 times and 1.3 times compared with Chain_max and RCS_max respectively. We can see from Fig. 8 (b) that in Topology 3, the worst update cost of CoLUE1 is even better than the average update cost of Chain and RCS. This is attributed to dependency minimization.

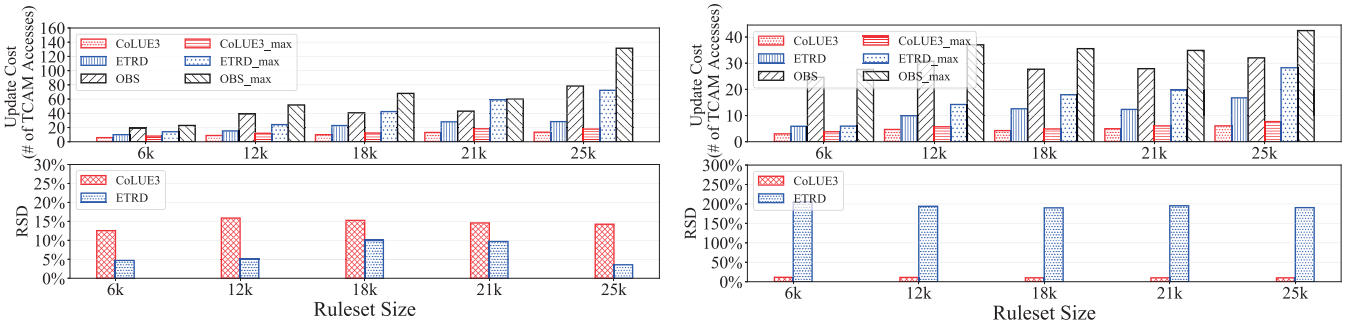
B. Comparison with Other All-switch Schemes

OBS covers the rules and packs groups of rules into switches to guarantee the correctness of flow table lookups. ETRD is designed for balanced placement across switches and lacks an update plan. Policies along all paths are the same in ETRD. To ensure fairness, we set the rules to be the same on each path in the experiment and use our cost function to guide the updates in ETRD. Fig. 9 (a) and (b) show the comparison of CoLUE3 with OBS and ETRD in Topology 1 and 3 respectively. Parameters are set to $\alpha = 50, \beta = 25, \gamma = 25$ for the cost function and batches of rules are installed in parallel. In Topology 1, CoLUE3 reduces the average cost by 3.2 times and the worst cost by 3.6 times compared with OBS. The average cost is twice smaller than that of ETRD and the worst cost is 1.9 times smaller. In Topology 3, CoLUE3 reduces the average update cost by 5.44 times and the worst update cost by 5.46 times compared with OBS. The reduction is 1.45 times for the average update cost and 1.93 times for the worst update cost compared with ETRD. The increased update speed of our scheme is attributed to that we reduce the insertion cost of a



(a) Topology 1 (b) Topology 3

Fig. 8: Comparison with Ingress-Switch-Close Schemes



(a) Topology 1 (b) Topology 3

Fig. 9: Comparison with All-switch Schemes

rule in each switch, and the insertion concurrency is better. OBS performs worst because it covers spatially close rules, resulting in large dependency degrees.

RSD in Fig. 9 indicates the percentage distribution of TCAM load ratios. A greater value means that the loads are farther from the average. We can see that CoLUE3 is slightly less balanced than ETRD in Topology 1, with update cost reduction by more than half. In Topology 3, CoLUE3 outperforms ETRD. Since there are more paths in Topology 3, CoLUE3 can flexibly adjust the ratio of rules for switches, so as to be more balanced. OBS generates replicated rules along each path during covering and packing. To guarantee policies can be installed, we pack all the remaining rules into the last switch to reduce the complexity of reperforming OBS multiple times. This operation has little effect on the update performance of OBS, but incurs a large RSD. Since OBS does not take balance into consideration, we do not compare the balance with OBS.

C. Comparison among Three CoLUE Schemes

The update cost under three schemes is shown in Fig. 10, with (a) measured in parallel and (b) serially. Eq. 8 is applied to CoLUE2 and CoLUE3 for updates, with parameters as $\alpha = 50, \beta = 25, \gamma = 25$, while CoLUE1 employs Eq. 6. Fig. 10 (b) suggests CoLUE1 performs the best in serial updates. This is because CoLUE1 cuts dependency thoroughly, and per rule update cost in each TCAM is the smallest. While in parallel updates, we take the concurrency of installing rules into account. As shown in Fig. 10 (a), CoLUE3 performs the best in parallel updates. CoLUE3 achieves a good compromise between dependency and balance, which brings a high concurrency and fast update in each TCAM. We also notice that CoLUE1 performs best in a 12k ruleset, this is because low dependency degrees overcome disadvantages in balance and concurrency. Generally speaking, CoLUE3 performs best in update speed.

D. Calculation Time of Update

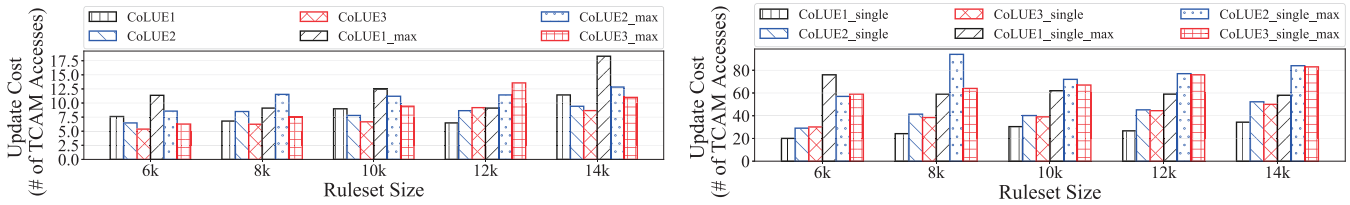
It is important to make update decisions quickly in dynamic updates. Fig. 11 shows that the average computation time for a rule insertion is within 1ms in three topologies. It attributes to our simple and effective cost function, as well as the balanced rule placement. CoLUE calculates fastest in Topology 3 as the symmetric topology brings more balance. We also test the calculation time of our rule placement scheme and find it can be done in tens of seconds. Since policies are not frequently redeployed, we think the time consumption is acceptable.

E. Effects of Parameters

As mentioned in Section III-B, there is a tradeoff between minimum dependency and balance. As we only care about the relative ratio of the coefficients and pay no attention to the absolute values, we test the effects of different weights for β by fixing the other two parameters. It can be seen from Fig. 12 that the existence of β matters in both update cost and balance. This is probably because the balanced rulesets curb the growth of dependency degrees and improve the parallelism of installation in dynamic updates. $\alpha = 50, \beta = 25, \gamma = 25$ is a good setting. An even larger β does not improve the performance, perhaps due to that the larger β causes intolerable chain lengths.

VI. Discusson on The CoLUE Overhead

In this section, we analyze the overhead on switch resources consumption, bandwidth and latency introduced by CoLUE. We implement a CoLUE prototype for a firewall policy on Barefoot Tofino switch EdgeCore Wedge 100BF-32X.



(a) Parallel Updates

(b) Serial Updates

Fig. 10: Comparison among Three CoLUE Schemes (Topology 3)

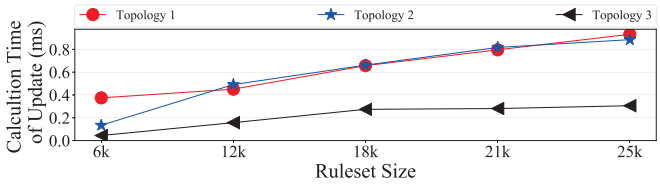


Fig. 11: Calculation Time of Update in Three Topologies

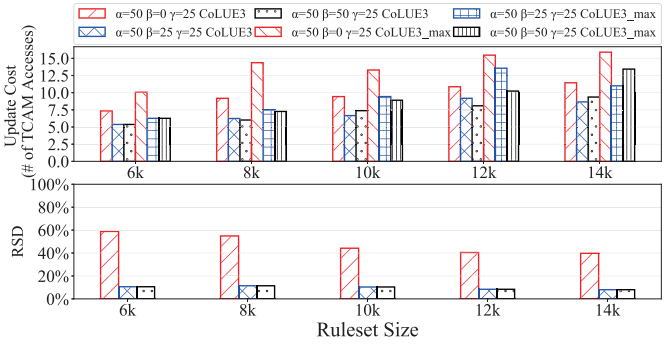


Fig. 12: The Effect of β

A. Impact on Switch Performance

Four extra stages are used by CoLUE's unoptimized P4 code as it has to compare the table lookup result and the carried data. CoLUE only occupies some more gateway resources, however, the total SRAM and TCAM consumption does not increase. As CoLUE does not recirculate or mirror packets, the switch throughput is unaffected. The packet processing latency adds 17ns, from 305ns to 322ns.

B. Impact on Traffic

In CoLUE, packets are dropped at the egress switches instead of the ingress switches, and they carry extra data. However, both the delayed dropped packets and extra carried data bring little bandwidth overhead. The reason is as follows. CoLUE's lookup procedure is transparent to users. It is similar to INT (Inband Network Telemetry), but does not increase the load overhead as the number of switch hops traversed increases. We use 16 bits to carry the priority and 1 bit for the action. For TCP flows, only the SYN packets need to carry extra data. If a flow should be denied according to the rules, it will not establish the connection since its SYN packet is dropped. In this case, the bandwidth overhead is small enough to neglect (i.e., only the overhead for the small SYN packet is introduced). For UDP flows, we can cooperate with T-cache [34] to set the ingress switches as a cache for rules whose actions are drop to reduce the bandwidth overhead. We leave it for future work.

VII. Related Work

A. TCAM Update Algorithms in A Single Switch

There have been lots of works studying how to insert rules quickly and correctly in a TCAM such as RuleTris [24], PoT [17], FastRule [18], FastUp [25], GreedyJump [35] and so on. These studies are orthogonal to our work and can be used in CoLUE to insert rules. Some works utilize multiple flow tables in a switch for update acceleration, including TreeCAM [36], Hermes [37], MagicTCAM [38] and BW-split [17]. However, these schemes are still in a switch, which may fail to bear the large number of rules.

B. Distributed Rule Management

Rulesets are usually split and distributed over the network to achieve flexible objectives. Many previous works are devoted to minimizing the total number of rules needed. For example, Palette [19] and ETRD [23] partition a single ruleset based on bits and distribute the sub-ruleset to each path using a coloring algorithm. OBS [20], AARP [21] and RCS [33] build an optimization model, and obtain a rule placement scheme with an ILP solver and some heuristics. Raptor [39] maximizes the rules sharing by multiple switches. Other objectives include maximizing traffic satisfaction [22], [33], reducing the controller burden [12], reducing table miss [40], [41], reducing policy conflicts under multiple controllers [42], etc. CoLUE aims for fast updates and TCAM load balance. Some work also considers quick updates [20], [33]. Our solution differs from them in that we take the rule dependencies into account, which is the root cause of slow updates. Although both CoLUE and CORA [42] reduce overlapping, CORA only focuses on conflict rules under multiple controllers, leaving alone conflict-free rules, which are a vast majority of rules. Update speed is still encumbered by their overlapping.

VIII. Conclusion

To narrow the gap between update delay and application requirements, we propose a novel framework CoLUE, which first combines TCAM update with ruleset placement over the network. CoLUE decomposes rulesets and distributes sub-rulesets in a balance and dependency minimum way. Our evaluation shows that compared with rule placement algorithms neglecting TCAM updates, CoLUE reduces the average update cost by more than 1.45 times and the worst cost by up to 5.46 times, with good balance as well.

REFERENCES

[1] J. Tourrilhes, P. Sharma, S. Banerjee, and J. Pettit, "The evolution of SDN and OpenFlow: a standards perspective," *IEEE Computer Society*, vol. 47, no. 11, pp. 22–29, 2014.

[2] B. Salisbury, "TCAMs and OpenFlow-what every SDN practitioner must know," http://tinyurl.com/kjy99uw.

[3] V. Heorhiadi, S. Chandrasekaran, M. K. Reiter, and V. Sekar, "Intent-driven composition of resource-management SDN applications," in *ACM CoNEXT*, 2018.

[4] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. H. Liu, and D. Yang, "Experience-driven networking: A deep reinforcement learning based approach," in *IEEE INFOCOM*, 2018.

[5] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat *et al.*, "Hedera: dynamic flow scheduling for data center networks." in *USENIX NSDI*, 2010.

[6] B. Niven-Jenkins, D. Brungard, M. Betts, N. Sprecher, and S. Ueno, "Requirements of an MPLS transport profile," 2009.

[7] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined WAN," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 3–14, 2013.

[8] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven WAN," in *ACM SIGCOMM*, 2013.

[9] "Sdn security considerations in the data center," https://opennetworking.org/wp-content/uploads/2013/05/sb-security-data-center.pdf.

[10] H. Pan, Z. Li, P. Zhang, K. Salamatian, and G. Xie, "Misconfiguration checking for SDN: Data structure, theory and algorithms," in *IEEE ICNP*, 2020.

[11] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Fast failure recovery for in-band OpenFlow networks," in *IEEE international conference on the Design of reliable communication networks*, 2013.

[12] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 351–362, 2010.

[13] M. Moshref, M. Yu, A. Sharma, and R. Govindan, "vCRIB: Virtualized rule management in the cloud," in *USENIX HotCloud*, 2012.

[14] X.-N. Nguyen, D. Saucez, C. Barakat, and T. Turletti, "Rules placement problem in OpenFlow networks: A survey," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 2, pp. 1273–1286, 2015.

[15] B. Isyaku, M. S. Mohd Zahid, M. Bte Kamat, K. Abu Bakar, and F. A. Ghaleb, "Software defined networking flow table management of OpenFlow switches performance and security challenges: A survey," *Future Internet*, 2020.

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

[17] P. He, W. Zhang, H. Guan, K. Salamatian, and G. Xie, "Partial order theory for fast TCAM updates," *IEEE/ACM Transactions on Networking*, vol. 26, no. 1, pp. 217–230, 2017.

[18] K. Qiu, J. Yuan, J. Zhao, X. Wang, S. Secci, and X. Fu, "FastRule: Efficient flow entry updates for TCAM-based OpenFlow switches," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 484–498, 2019.

[19] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing tables in software-defined networks," in *IEEE INFOCOM*, 2013.

[20] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the "one big switch" abstraction in software-defined networks," in *ACM CoNEXT*, 2013.

[21] S. Zhang, F. Ivancic, C. Lumezanu, Y. Yuan, A. Gupta, and S. Malik, "An adaptable rule placement for software-defined networks," in *IEEE/IFIP DSN*, 2014.

[22] X.-N. Nguyen, D. Saucez, C. Barakat, and T. Turletti, "OFFICER: A general optimization framework for OpenFlow rule allocation and endpoint policy enforcement," in *IEEE INFOCOM*, 2015.

[23] J.-P. Sheu, W.-T. Lin, and G.-Y. Chang, "Efficient TCAM rules distribution algorithms in software-defined networking," *IEEE Transactions on Network and Service Management*, vol. 15, no. 2, pp. 854–865, 2018.

[24] X. Wen, B. Yang, Y. Chen, L. E. Li, K. Bu, P. Zheng, Y. Yang, and C. Hu, "RuleTris: Minimizing rule update latency for TCAM-based SDN switches," in *IEEE ICDCS*, 2016.

[25] Y. Wan, H. Song, H. Che, Y. Xu, Y. Wang, C. Zhang, Z. Wang, T. Pan, H. Li, H. Jiang *et al.*, "FastUp: Compute a better TCAM update scheme in less time for SDN switches," in *IEEE ICDCS*, 2020.

[26] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM computer communication review*, vol. 38, no. 2, pp. 69–74, 2008.

[27] C. Luo, C. Chen, H. Mei, R. Yao, Y. Wan, W. Li, S. Liu, B. Liu, and Y. Xu, "BubbleTCAM: Bubble reservation in SDN switches for fast TCAM update," in *IEEE/ACM IWQoS*, 2022.

[28] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 539–550, 2014.

[29] D. Shah and P. Gupta, "Fast updating algorithms for TCAM," *IEEE Micro*, 2001.

[30] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple: Simplifying SDN programming using algorithmic policies," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 87–98, 2013.

[31] H. Song and J. Turner, "NXG05-2: Fast filter updates for packet classification using TCAM," in *IEEE GLOBECOM*, 2006.

[32] D. E. Taylor and J. S. Turner, "ClassBench: A packet classification benchmark," *IEEE/ACM transactions on networking*, vol. 15, no. 3, pp. 499–511, 2007.

[33] Y.-W. Chang and T.-N. Lin, "An efficient dynamic rule placement for distributed firewall in SDN," in *IEEE GLOBECOM*, 2020.

[34] Y. Wan, H. Song, Y. Xu, Y. Wang, T. Pan, C. Zhang, and B. Liu, "T-cache: Dependency-free ternary rule cache for policy-based forwarding," in *IEEE INFOCOM*, 2020.

[35] Y. Wan, H. Song, and B. Liu, "Greedyjump: A fast tcam update algorithm," *IEEE Networking Letters*, 2021.

[36] B. Vamanan and T. Vijaykumar, "TreeCAM: Decoupling updates and lookups in packet classification," in *ACM CoNEXT*, 2011.

[37] H. Chen and T. Benson, "Hermes: Providing tight control over high-performance SDN switches," in *ACM CoNEXT*, 2017.

[38] R. Yao, C. Luo, X. Liu, Y. Wan, B. Liu, W. Li, and Y. Xu, "MagicTCAM: A multiple-TCAM scheme for fast TCAM update," in *IEEE ICNP*, 2021.

[39] P. G. Kannan, M. C. Chan, R. T. Ma, and E.-C. Chang, "Raptor: Scalable rule placement over multiple path in software defined networks," in *IFIP Networking*, 2017.

[40] O. Rottenstreich, A. Kulik, A. Joshi, J. Rexford, G. Rétvári, and D. S. Menasché, "Cooperative rule caching for SDN switches," in *IEEE CloudNet*, 2020.

[41] M. Jiménez-Lázaro, J. Berrocal, and J. Galán-Jiménez, "Deep reinforcement learning based method for the rule placement problem in software-defined networks," in *IEEE/IFIP NOMS*, 2022.

[42] H. Li, K. Chen, T. Pan, Y. Zhou, K. Qian, K. Zheng, B. Liu, P. Zhang, Y. Tang, and C. Hu, "CORA: Conflict razor for policies in SDN," in *IEEE INFOCOM*, 2018.