Journal of Network and Computer Applications 199 (2022) 103307

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Raze policy conflicts in SDN

Yadong Zhou [a], Hao Li [a,*], Kaiyue Chen [a], Tian Pan [b], Kun Qian [c], Kai Zheng [d], Bin Liu [c], Peng Zhang [a], Yazhe Tang [a], Chengchen Hu [e]

[a] MOE KLINNS Lab, Faculty of Electronic and Information Engineering, Xi'an Jiaotong University, China
[b] State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, China
[c] Tsinghua University, China
[d] Huawei 2012 Labs, China
[e] Xilinx Labs Asia Pacific, Singapore

ARTICLE INFO

Keywords:
software-defined networking
Flow table compression
Policy conflict

ABSTRACT

Software Defined Networking (SDN) enables flexible network management with a well-defined abstraction between control and data plane. In this way, operators could issue the policies, *e.g.*, forwarding path, flow counting and rate limiting, from the control plane, which will be enforced by the flow table rules in the data plane. However, multiple active policies with the same priority will potentially trigger conflicts among policies with overlapped flow space, causing the flow table explosion. In contrast to the local switch conflict resolution schemes proposed by previous works, this paper tackles the same problem from a different angle and resolves the policy conflict problem by coordinating all switches under a global centralized view. Specifically, we propose COnflict RAzor (CORA), which tremendously reduces the storage cost of conflicting policies leveraging the global network information obtained in the controller. The basic idea of CORA is migrating policies causing large explosions across the network if necessary, while keeping the semantics equivalence. We prove CORA's NP hardness and propose a heuristic to efficiently search a near-optimal policy migration strategy. Our experiments demonstrate that, CORA can effectively reduce the flow table storage occupation by averagely 79.8% within less than 40 s, which is 47.9% more efficient than the state-of-the-art.

1. Introduction

Software Defined Network (SDN) decouples switch's control and data plane, offering enhanced programmability via a higher-level abstraction, *i.e.*, intent-based north-bound interface (NBI). The operator intents would be represented with native code like Python program in Ryu controller, or domain specific languages (DSL) like SNAP scripts (Arashloo et al., 2016). In either way, such intents would be firstly compiled into distributed *policies*, associating the packet class (*i.e.*, flow) with the corresponding actions (*e.g.*, forward, drop, count) for each underlying switch. The policies would be further translated into hardware-specific *rules* when loaded into a particular device, *e.g.*, prefix entries for Ternary Content Addressable Memory (TCAM). However, policies issued by different controllers (NBIs) may specify different actions on an overlapped flow space at the same network device. The so-called "policy conflicts" should be resolved to perform the correct combination of the actions. But, such conflict resolution will potentially incur either performance penalty or resource inefficiency for the underlying network.

We use a simple example to illustrate how the conflict happen and how to resolve it. Consider that a QoS function and a monitoring function coexist in a network, where the former has a policy at switch s to limit the bandwidth to 10 Mbps for packets with DstPort range 1–6, and the latter sets a policy to count the number of packets with DstPort range 2–7 at the same switch. Such two policies are conflicting due to the different actions on the overlapped port range 2–6. A straightforward method to cooperate these two policies in a single switch is to split them into three sub-policies with non-overlapping flow spaces: (P_1) 1–1→limit 10 Mbps, (P_2) 2–6→limit 10 Mbps and count, and (P_3) 7–7→limit 10Mbps[1]. Intuitively, fragmented policies will inevitably be generated when resolving conflicts. In a bad case, each policy at the switch would be conflicting with all others, producing many more sub-policies on fragmented flow spaces. In addition, when it comes to multi-dimension conflicts in many flow entry fields, things will get even worse than the single-field scenario. As a result, the rule conflicting problem aggravates heavily in the context of SDN, since the switch

* Corresponding author.
E-mail address: hao.li@xjtu.edu.cn (H. Li).

[1] There are definitely other solutions to divide the flow space more efficiently. We will discuss them in the latter sections, and only demonstrate a base case here.

https://doi.org/10.1016/j.jnca.2021.103307
Received 15 October 2020; Received in revised form 30 October 2021; Accepted 3 December 2021
Available online 22 December 2021

checks more fields other than the traditional 5-tuples. Please notice that the number of extra generated policies does not reflect the complete overhead, because it may be much more costly when translating the policies into hardware-specific rules. For example, to represent a value range of packet fields like DstPort, a TCAM-based switch has to convert the range into one or more prefix entries during the so-called "rule expansion" process, *e.g.*, at least three entries (110, 01*, 10*) are needed for the above DstPort range 2–6.

In the literature, many works have investigated the mechanisms of efficient conflict resolution. One possible way is to find a more efficient cut of the flow space by creating a new policy on the overlapped flow space with a higher priority while retaining the original ones with a lower priority (Jin et al., 2015). This may avoid producing too many flow space fragments, since the policies that are fully covered by the higher-priority ones are redundant and can be eliminated to save memory cost. However, even with this technique, extra policies would be inevitably generated as long as one flow space does not fully cover the other. Another attempt is to reduce/minimize the rule expansion specifically for TCAM-based devices (Kogan et al., 2015; Liu and Gouda, 2010; Meiners et al., 2012; Bremler-Barr and Hendler, 2012; Katta et al., 2014; Liu et al., 2010). However such low-level optimization does not address the root cause of the policy conflicts either, and in the worst case, a W-bit range value (indexed by some policy) will consume W TCAM entries (Rottenstreich et al., 2013). Although the OpenFlow specification (ONF, 2015) proposes the flow table pipeline to mitigate the aftermath of policy conflicts, simply employing such pipeline will dramatically increase either the number of flow tables or the bit width for each flow table. Network slicing solutions (Al-Shabibi et al., 2014; Koponen et al., 2014; Doriguzzi Corin et al., 2012) provides individual flow spaces for each network function to isolate the conflicts, which actually disables multiple network functions to operate on the same traffic, *i.e.*, functions can only manage the disjoint flow space. Nowadays, many high-level SDN languages offer the resource constraints in their syntaxes and compilation process, which can be adopted to generate a more efficient placement of policies to mitigate the conflicts beforehand (Arashloo et al., 2016; Prakash et al., 2015). However, this requires the global view of all operators' intents, which violates our assumption: the intents may be issued by individual operators from different controllers that cannot cooperate at the language level (Jin et al., 2015).

We observe that the policy/rule explosion is ascribed to the flow space overlaps, and as a result, our basic idea is to move/migrate the conflicting policies from the local switch to reduce the overlaps and further decrease the #policies and entries. The prerequisite is to ensure that all network function will always hold their semantics after moving/modifying the policies, *i.e.*, the flows should be forwarded to the original destination along the same path with the same actions applied, *e.g.*, rewrite, count, mirror to controller, *etc*. SDN offers the global information of data plane policies, which can be utilized to guarantee the above requirements. Recall the aforementioned example, the conflicts can be eliminated if we move the counting policy from the switch s to an adjacent switch s' along the routing path of the flow, since the flow forwarding behavior remains the same and the counting action can be applied at any switch along the path.

Based on the above insights, we propose COnflict RAzor (CORA) to efficiently resolve the policy conflicts in SDN, which collects policies from all network functions, and migrates the conflict-makers from the current switch to other feasible switches to relieve the conflicts while keeping the equivalent semantics. In a nutshell, two designs make CORA efficient. First, to the end of conflict elimination, CORA leverages the cross-switch information from the data plane, which would dramatically decrease the possible conflicts. Second, to the end of finding an (near-)optimal solution, CORA applies a simple yet effective heuristic to conquer such an NP-hard problem. It is worth noting that CORA focuses on efficiently eliminating the conflicts in the global network, thus can well cooperate with any existing solution that reduces/minimizes the #policies at a single switch. To be specific, the rule transferring enables much more possibilities if cooperating with the local conflict resolver, because we can find a "combination" to improve the optimization performance, *e.g.*, transferring specific rules to a certain switch, such that the local conflict resolver can maximize its effect.

This is paper is an extension to our preliminary paper (Li et al., 2018), in which we have made the following three contributions:

- We explore the potential benefits and technical challenges of migrating policies across the network, and propose semantics-preserving migration mechanisms to address the challenges, *e.g.*, retaining the routing paths, slicing the endpoint actions, *etc*.
- We formally define the problem of finding an optimal policy placement with many policy conflicts. After proving its NP hardness, we give some heuristics to fast generate a near-optimal placement.
- We implement a prototype of CORA, and use synthetic policy configurations and topologies to evaluate its performance. The experimental results show that CORA can reduce at least 49% of the total conflict overhead within acceptable time, while retaining the original intents.

We further make the following contributions in this paper.

- We check and port two typical local optimizations on CORA to demonstrate its compatibility to existing local-switch rule compression algorithms.
- We compare CORA with existing approaches including rule packing and local resolving to demonstrate its optimality.

The reminder of this paper is organized as follows. Section 2 demonstrates the policy conflict problem and our basic idea. Section 3 proposes the semantic-preserving transferring to retain the high-level intents. Section 4 defines the problem of finding an optimal placement from the global view, and designs a heuristic algorithm to obtain a near-optimal solution within acceptable time cost. Section 5 presents two aggregation-based optimizations on local switch, and demonstrates their compatibilities with CORA. Section 6 evaluates CORA's performance. Section 7 discusses the possible extension of CORA. And after discussing the related work in Section 8, Section 9 concludes this paper.

2. Transfer policy to raze the conflicts

2.1. Policy conflict problem

SDN high-level languages specify two categories of operators' intents: the routing intent and the endpoint intent (Kang et al., 2013). The routing intent is to specify the paths between the ingress and egress of s packet class, driven by traffic engineering goals. The endpoint intent focuses on the end-to-end packet behaviors other than forwarding, *e.g.*, counting, mirroring to controller, modifying header, *etc*. In the single-controller scenario, such intents are issued by a same NBI, and the controller would compose and compile them into distributed policies, while the policies in each switch have been properly prioritized (Arashloo et al., 2016; Prakash et al., 2015). However, it has been advocated that multiple controllers should coexist in a network with a hypervisor, which provides the ability of running any combination of controller applications (Jin et al., 2015). Therefore, the policies from different controllers can have overlaps with a same priority, triggering the policy conflicts. Notice that due to the language-barrier of different NBIs, the existing hypervisors cannot reconcile the policies at the language level, but only resolve them in the local switch.

Formally, a policy can be denoted as $p = (sw, pri, fs, a)$, where sw is the switch storing the policy, pri is the priority of the policy, fs is a hyperspace with different fields to describe the flow space, and a is the action set that applies on fs. Two policies p_1 and p_2 conflicts,

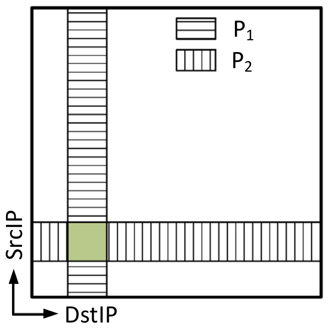


Fig. 1. Two policies conflict on two fields.

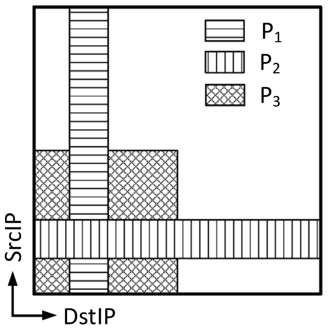


Fig. 2. More conflicts caused by a third policy.

iff $p_1.sw = p_2.sw$, $p_1.pri = p_2.pri$, $p_1.fs \cap p_2.fs \neq \emptyset$ and $p_1.a \neq p_2.a$[2]. To resolve such conflict, a naive method is to fully decouple the overlapped space into continuous fragments, each of which performs the combination of actions from corresponding policies. For example, the above two conflicting policies would be decoupled into three ones: $p'_1 = (p_1.sw, p_1.pri, p_1.fs \setminus p_2.fs, p_1.a)$, $p'_2 = (p_1.sw, p_1.pri, p_2.fs \setminus p_1.fs, p_2.a)$, and $p'_3 = (p_1.sw, p_1.pri, p_2.fs \cap p_1.fs, p_1.a \cup p_2.a)$. Notice that these three policies may expand to more because the flow space like $p_1.fs \setminus p_2.fs$ may not be continuous, which will be further transformed into two policies. In the worst case, one policy would conflict with all the other policies, and the naive decoupling process will lead to a policy increase at a complexity of $O(N^2)$, where N is the number of original policies. Fig. 1 illustrates a simple example of policy conflict in two dimensions DstIP and SrcIP, where P_1 is a QoS policy to limit the bandwidth and P_2 is to count the number of packets. The flow space would be cut into five sub-spaces to fully resolve the conflict.

The naive method is not efficient enough when the conflicts result from the inclusion of flow space. Taking the above two-dimension conflict as an example, P_2 includes P_1 from the perspective of X axis (SrcIP). Therefore, it is not necessary to divide the X axis into three segments, instead, the original P_2 can be retained and a new policy would be added with higher priority that represents the overlapped flow space in X axis. We can apply similar analysis in Y axis, and as a result, only one extra policy is needed to resolve the conflict (the solid shadowed part in Fig. 1), *i.e.*, $P' = (P_1.sw, P_1.pri+1, P_1.fs \cap P_2.fs, P_1.a \cup P_2.a)$.

However, the above priority-based method is still not efficient enough to tame the explosive growth of policies. The reason lies in that this method only works for inclusion cases, which would commonly happen in IP fields due to the prefix representation. In contrast, some fields like DstPort and SrcPort are represented by ranges, which leads to

[2] In the following sections, we assume the involved policies have the same priority if not specified.

Table 1
The policies to decouple the conflicts in Fig. 2.

P	$FlowSpace$	$Action$	Pri
1	$P_1.fs \cap P_2.fs \cap P_3.fs$	$P_1.a \cup P_2.a \cup P_3.a$	3
2	$P_1.fs \cap P_3.fs$	$P_1.a \cup P_3.a$	2
3	$P_2.fs \cap P_3.fs$	$P_2.a \cup P_3.a$	2
4	$P_1.fs$	$P_1.a$	1
5	$P_2.fs$	$P_2.a$	1
6	$P_3.fs$	$P_3.a$	1

overlaps rather than inclusions in most cases, and cannot be resolved by the priority-based method. For example, Fig. 2 adds a third rule P_3 with the same priority to the example in Fig. 1, and they will be transformed into six policies even we sophisticatedly prioritize the overlaps, as shown in Table 1. Even worse, many switches use TCAM to implement the flow tables, which would expands the entries to represent the range values (Liu et al., 2010). Formally, each range defined over a W-bit field can be encoded in W entries with the internal expansion in the worst case, and if the flow space specifies W ranges on d fields, it will consume up to W^d entries in TCAM (Rottenstreich et al., 2013). Since the policy conflicts are producing more fragments on the flow space, the overhead in the real scenarios would go far beyond the $O(N^2)$ complexity, and squeeze the limited TCAM resources.

2.2. Basic idea of CORA

All the existing techniques focus on how to minimize the overhead when decoupling the conflicts. In contrast, CORA treat the same problem from a different angle, aiming to eliminate the conflicts by migrating the policies, *i.e.*, to reduce the number of conflicts in global network instead of the number of expanded policies in a local switch.

Considering a simple linear topology with two switches S_1 and S_2, the policies at each switch are shown in Fig. 3. It needs 16 and 7 sub-policies to fully decouple all the conflicts in S_1 with naive and priority method, respectively. However, if we transfer P_2 to S_2 as shown in Fig. 4, only 4 sub-policies are produced in S_1 with priority method, and S_2 generates no more policies other than P_2 from S_1, which means the overall #policies decreases from 8 to 6. The performance gain can be more significant for TCAM-based switches.

The above simple example shows the potential benefits if properly migrating policies. However, arbitrarily migrating policies may break the high-level intents from the operators. Recall the above simple network configurations, there are many prerequisites to transfer P_2 to s_2. First, P_2 cannot be a routing policy that forwards the packet, since such transfer would produce a black hole at s_1 for the packets that match the flow space of P_2. Second, P_2 can be transferred to s_2 only when there is other routing policy that forwards all the packets in P_2's flow space to s_2 (*e.g.*, P_3), or P_2's action cannot be properly triggered. These special cases need to be carefully addressed to retain the original high-level intents. The other challenge is to find an optimal policy placement with acceptable time cost. The #policy in network can easily reach 1000+ with highly dependent conflicts, *e.g.*, P_2, P_3 and P_4 have common overlaps, while they also pairwisely conflict with each other. Simply exhausting all possible policy placements is obviously infeasible in time cost, because the #policy combinations can be up to 2^n for n policies.

In summary, to well design CORA, we need to address the following major challenges.

Semantics equivalence. The transferring operations should be semantics-preserving, which guarantees the correct fulfillment of multiple network functions.

Optimal placement. After policy transferring, rules must satisfy the constraint of the physical switch capacity, and is expected to be minimal in the network.

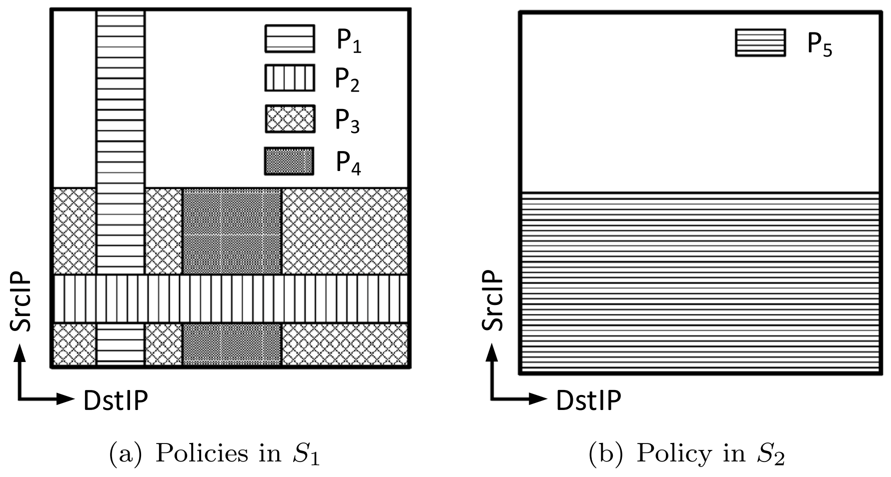


Fig. 3. The policy placement in two adjacent switches.

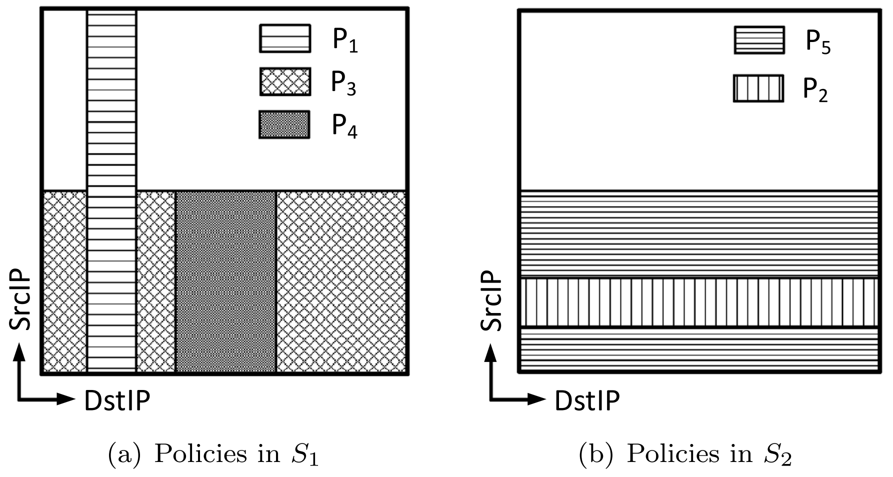


Fig. 4. The new policy placement solution if transferring P_2 from S_1 to S_2.

3. Semantics-preserving transfer

It is not trivial to correctly transfer the policies because arbitrary change of the policy placement may break the high-level intents, *i.e.*, routing intent and endpoint intent.

3.1. Routing intent

The routing intent would be compiled into routing policies for individual switches, each of which forwards the packet to the next hop. That is, there lies strong dependencies between the those policies; if we transfer one routing policy to another switch, we have to modify the related routing policy at the previous hop, and we may need to create new routing policies to fulfill a complete routing path. Besides, we cannot guarantee the traffic engineering requirements are satisfied by the new path, because such intents are hidden in the compilation process, and cannot be reverse engineered from the policies. Therefore, the routing policies are considered as fixed in CORA to ensure the semantics equivalence of routing intent.

Please notice the conflicts between two routing policies are not resolvable in CORA, because we cannot forward a single packet to two different next hops. Such conflicts may happen if multiple controllers decide the routing paths individually, and resolving them need composing the routing intents at a higher semantics level (Li et al., 2016). In this paper, we assume the routing intents are handled by a single controller application, or the flow space is isolated for different routing applications, *i.e.*, the routing policies are not conflicting with each other.

3.2. Endpoint intent

The endpoint intent is to perform specific actions on the packets, and such intent will hold, as long as the actions is triggered for all the packets whose headers fall in the flow space. Initially, the endpoint intent would be compiled into several endpoint policies, each of which covers partial flow space of the intent. The split of the flow space depends on the placement of the routing intent, since it is possible that not all packets in the flow space traverse a single switch. The intuitive idea is to transfer the endpoint policy along the routing path, as long as the target switch has the ability to perform the action. To be specific, we have the following principles of transferring endpoint policies.

First, the flow space of the routing policy that covers the endpoint policy should be consistent through the transferring, or the endpoint policy needs to be further divided. Considering a routing policy P_{fwd} that forwards the packets with DstPort 1–6 to port 1, and an endpoint policy P_{cnt} that counts all packets with DstPort 2–7, P_{cnt} cannot be directly transferred to the switch that connects to port 1, since it will fail to count the packets with DstPort 7. As a result, we need to divide P_{cnt} into two policies, $P_{cnt,1} = (P_{cnt}.pri, P_{cnt}.fs \cap P_{fwd}.fs, count)$ and $P_{cnt,2} = (P_{cnt}.pri, P_{cnt}.fs \setminus P_{fwd}.fs, count)$, and we can transfer $P_{cnt,1}$ through port 1. To this constraint, an endpoint policy (or a slice of an endpoint policy) p_e in switch s can be transferred through port i, if there exists a routing policy p_f in switch s, where $p_e.fs \in p_f.fs$ and $p_f.a = \text{fwd}(i)$ (forward transferring), or there exists a routing policy p_b in switch s', where $p_e.fs \in p_b.fs$, $p_b.a = \text{fwd}(j)$ and port j in s' connects to port i in s (backward transferring). Here we only discuss the one-step transferring to the pre or next hop, and the multi-hop transferring can be seen as a combination of multiple one-step moves.

The above principle only ensures the semantics if there is only one endpoint intent, because the dependency between endpoint intents may further constrain the placement of endpoint policy. Considering an endpoint policy p_m modifies the VLAN id to 10 for the packets with VLAN id 1, and another policy p_c counts the packets with VLAN id 1, the order of triggering the two policies reflects the high level intent; if p_c is triggered before p_m, the two policies just stick to their scripts; otherwise, p_c is only to verify whether p_m correctly works. We assume the initial placement has already satisfied the high level intent, and therefore, the order of triggering the two policies cannot be violated. More generally, we say p_1 depends on p_2, if $p_2.a$ will cut or produce packets to be processed by p_1. To maintain the original intents, the order of two dependent policies cannot be changed. For example, if an endpoint policy is conflicting with a header modifying policy, then it can only be transferred between the ingress/egress and the header modifying policy.

Please notice that this constraint also forbids the transferring of header modifying policies, because the routing policies definitely depend on header modifying policies; if we transfer it to the next hop, the modification would break the routing path, because there is no routing policy that handles the unmodified headers in next hop; likewise, the pre hop is also infeasible, because there is no routing policy to forward the modified headers in the current switch.

In summary, we say a policy (or a slice of policy) *can be transferred* to a certain switch, if it satisfies the above two constraints, *i.e.*, the routing policy restriction and the order of critical actions. Following this definition, we further define a one-step semantics-preserving function ST, which takes a policy p and an adjacent switch s as the input, and outputs a set of new policies. Specifically, $ST(p,s) = \emptyset$, if neither p nor a slice of p can be transferred to s; $ST(p,s) = \{p[sw \mapsto s]\}$, if p can be completed transfer to switch s; $ST(p,s) = \{p[fs \mapsto p.fs \backslash p'.fs], p'[sw \mapsto s]\}$, if a slice of p, denoted as p', can be transferred to switch s. The notation $p[f \mapsto v]$ is to replace $p.f$ with value v.

4. Finding optimal placement

4.1. Problem formulation

The optimal policy placement is a placement that the cost (*i.e.*, the number of rules) of all switches is minimized. Previously, the optimal policy placement has been discussed in several papers (Kang et al., 2013; Arashloo et al., 2016), most of which models the problem as follows: n policies should be assigned to m switches, while each assignment (policy j to switch i) has its profit p_{ij} and cost w_{ij}, and each switch has its capacity W_i. The goal is to maximize the profits while assuring each switch does not run out of its capacity. Such model captures the well-known general assignment problem (GAP), thus is also NP-Hard. However, the original GAP assumes the cost w_{ij} is fixed, and not dependent of the placement of other policies, while in our scenario, w_{ij} depends on the policies previously assigned to switch i, because the number of rules varies to the conflicts between the policies in the same switch.

In this paper, we define the above extended placement problem as *policy optimal placement problem with dependent cost* (POPDC). To address such problem, we divide POPDC into two sub-problems: (1) decide the combinations of policies (DCP), and (2) assign the combinations to the switches (ACS). These two sub-problems are independent, because DCP only considers the penalty of putting certain policies together, which determines the total the number of policies/rules of the network, while ACS focuses on finding an optimal placement for the combinations to satisfy the capacity constraint. In the following, we will first define the variables involved in POPDC, and address the two sub-problems respectively.

Algorithm 1 Divide the original policies into fragments.

```
1: $P \leftarrow P_e$
2: for all $p$ in $P$ do
3:     for all $s$ that connect to $p.sw$ do
4:         if $|ST(p,s)| > 1$ then
5:             $P \leftarrow P \setminus \{p\} \cup ST(p,s)$
```

Variables and notations. The first variable in POPDC is the policy set that to be assigned, denoted as P, which however cannot directly map to the original endpoint policy set P_e. The reason is that it is possible that the policy cannot be assigned to a certain switch, or only a slice of the policy can be transferred to that switch, due to the semantics-preserving transfer restriction. To address this problem, we utilize the one-step semantics-preserving function ST to divide the policies into fragments, each of which can be independently assigned to a switch. The set of these fragments forms the policy set P. The divide process is illustrated in Algorithm 1.

The other variables in POPDC is quite straightforward: there are m switches in the network, each of which has a capacity W_i. We use S to denote a set of policies, and $w(S)$ represents the cost of decoupling S, which can be measured with the number of decoupled policies or the number of expanded rules.

DCP: decide the policy set to be assigned. To model DCP, we first expand all the candidate policy set. Assume we have l policies, there are 2^l candidate combinations of policies, the set of which is denoted as $S = \{S_i\}, i = 1, \ldots, n$, where $n = 2^l$. Our goal is to find a subset of S, denoted as C, to satisfy the following requirements: (1) the number of selected sets must not be larger than the number of switches, (2) the policy combinations in C are pairwise disjoint, (3) the union of C equals to P, and (4) the total cost of C is minimal. The first goal constrains the number of sets to the number of switches, or the sets cannot be assigned to switches independently. The second and third goal is to seek a disjoint set cover of P and the last goal is to minimize the total costs, *i.e.*, the number of policies/rules.

Based on the above analysis, we formulate DCP with the following integer linear program.

$$\text{maximize} \quad \sum_{s \in S} (x_s / \sum_{ps \in s} w(ps)) \tag{1}$$

$$\text{subject to} \quad \sum_{s:e \in s} x_s = 1, \qquad \text{for all } e \in P \tag{2}$$

$$\sum_{s \in S} x_s \leq m, \qquad \text{for all } s \in S \tag{3}$$

$$x_s \in \{0,1\}, \qquad \text{for all } s \in S \tag{4}$$

Eq. (1) is to maximize the profit of the selected policy sets, where the profit is defined as the reciprocal of the policy set cost. Eq. (2) restricts that every policy must be selected exactly once to produce a set cover. Eq. (3) constrains the number of sets to the number of switches. Eq. (4) defines a 0–1 variable to represent every set is either selected or not. It is clear that DCP has the same representation with *weighted disjoint set cover* problem, which has been proved to be NP-Hard (Pananjady et al., 2015).

ACS: assign the policy sets to switches. Given an optimal policy sets C by DCP, the next step is to assign them to different switches. The goal of ACS is to ensure the assignment will not exceed the capacity of each switch. Assume we have n policy sets in C, m switches in the network, $n \leq m$, we can formulate ACS with the following integer linear program.

$$\text{maximize} \quad \sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij} \tag{5}$$

$$\text{subject to} \quad \sum_{j=1}^{n} r_{ij}x_{ij} \leq W_i, \; i = 1,\dots,m \tag{6}$$

$$\sum_{i=1}^{m} x_{ij} = 1, \quad j = 1,\dots,n \tag{7}$$

$$\sum_{j=1}^{n} x_{ij} = 1, \quad i = 1,\dots,m \tag{8}$$

$$x_{ij} \in \{0,1\}, \quad i = 1,\dots,m, j = 1,\dots,n \tag{9}$$

Notice we introduce a new cost parameter r_{ij} to represent the cost of assigning set j to switch i. Specifically, $r_{ij} = w(C_j)$, if all policies in C_j can be transferred to switch i; $r_{ij} = \infty$, if at least one policy in C_j cannot be transferred to switch i. With Eq. (5)–(9), ACS can be reduced to GAP, if we assume the profit of assigning a policy set equals to 1. Therefore, ACS is a NP-Complete problem.

In summary, due to the high complexity of both DCP and ACS, POPDC cannot be solved in polynomial time.

4.2. Heuristics of near-optimal placement searching

As discussed above, POPDC is computationally hard, an intuitive idea is to utilize the existing approximate algorithms to find near-optimal solutions. However, the first step of modeling DCP, *i.e.*, expanding all the possible policy combinations as the candidates, would largely impact the total complexity of solving DCP, because it exponentially increases the problem scale. Therefore, we do not use existing approximate algorithms, but propose some simple heuristics to approach the optimal policy placement, under the acceptable time consumption. Specifically, the optimal placement is expected to satisfy the following requirements: (1) the number of rules in each switch should not go beyond the capacity of the switch, (2) the total the number of rules are minimized for the entire network, and (3) the standard deviation of the number of rules in each switch should be minimized, so it is not likely to overflow when a new policy comes.

Our basic idea is to greedily find a "conflict-maker", *i.e.*, the highest-cost policy, among all endpoint policies P_e, and iteratively make a one-step semantics transfer to the target switch that leads to the best profit. The cost of policy p is measured by the total cost decrement of switch s if we remove p from s. To find a conflict maker, a straightforward method is to traverse all policies, while simple heuristics and optimizations can be applied for this searching; we can use the number of conflicts produced by the policy as the cost instead of measuring the precise the number of rules, which may reduce the searching time, especially for TCAM-based switches; we can pre-compute the cost of all policies beforehand, because the transferring only impacts two switches, and it does not need to re-compute the cost for policies in the rest switches, which could accelerate the conflict-maker searching in the next round. With the conflict maker, we have to choose where to transfer it for larger profit. Specifically, we use K to denote the number of switches that exceeds the capacity in current placement, and O to denote the total the number of exceeded rules in the network. We further define D as the standard deviation of the cost for each switch, and use C to denote the total cost of the placement, which can be measured with the number of policies or the number of rules. Based on these notations, we define the profit of a placement as $B = T/(C \times D) - O$, where $T = 0$, if $K > 0$, and otherwise $T = 1$.

The searching process is to seek a better profit through the semantics-preserving transfer to the conflict-maker. If transferring any policy under current placement would not lead to a larger B, the process ends, and the current placement is an optimal solution if $B > 0$.

Algorithm 2 Greedily searching for the optimal placement.

```
Procedure: Pre-Computing the Policy Cost
 1: $P_{sw} \leftarrow \{\{p \in P_e | p.sw = i\}, i = 1,...,n\}$
 2: for $P_i$ in $P_{sw}$ do
 3:     for all $p$ in $P_i$ do
 4:         $p.cost \leftarrow c(P_i) - c(P_i \setminus \{p\})$
 5: sort $P_e$ by the policy cost in descending order
Procedure: Greedy Searching for Optimal Placement
 1: Pre-Computing the Policy Cost
 2: Compute $C$, $D$, $K$, $T$, $O$ according to the cost
 3: $B \leftarrow T/(C \times D) - O$
 4: $i \leftarrow 0$
 5: while True do
 6:     while True do
 7:         $cm \leftarrow P_e[i]$
 8:         $ts \leftarrow cm.s$
 9:         for all $s$ that connects to $cm.s$ do
10:             $ST(cm, s)$ and update $C$, $D$, $K$, $T$, $O$ accordingly
11:             $b \leftarrow T/(C \times D) - O$
12:             if $b > B$ then
13:                 $B \leftarrow b; ts \leftarrow s$
14:             rollback the transfer of $cm$ and restore $C$, $D$, $K$, $T$, $O$
15:         if $ts = cm.s$ then
16:             break
17:         $ST(cm, ts)$ and update $C$, $D$, $K$, $T$, $O$ accordingly
18:         update the cost of policies in original $cm.s$ and $ts$
19:         sort $P_e$ by the policy cost in descending order
20:         $B \leftarrow T/(C \times D) - O$
21:         $i \leftarrow 0$
22:     $i \leftarrow i + 1$
23:     if $i > |P_e|$ then
24:         break
```

The complete process is illustrated in Algorithm 2. Notice if $B \leq 0$, the solution is not acceptable due to the exceeded capacity, which needs to be reported to operators for further process.

Incremental placement update. The optimized placement needs to be incrementally adjusted when adding or deleting policies. If a policy p is deleted from switch s, we just remove all the sub-policies that produced by p (including p itself). Since the cost of s must be reduced due to the removal, we only need to recompute the profit of the adjacent switches of s, to see whether some additional transferring can make larger profit. If a policy p is added to switch s, we compute the conflicts it produces with the existing ones, and since the cost of s must be increased, we only need to try limited policy transferring from s to obtain a new optimal placement. The policy modification can be seen as a combination of deleting a policy and adding a policy. In practice, it is common that a group of policies are updated for an entire routing path, and we can re-perform CORA after that batch update.

5. Cooperation of local optimizations

In a nutshell, when CORA decides whether a rule is "worthy" of being transferred to an adjacent switch, it will compute a profit for the potential transferring, and performs the transferring only if the profit is the big enough. In computing the profit, CORA could employ the local conflict resolver to reveal the benefit of the transferring, i.e., the factual overhead if transferring this rule to another switch. The computation of profit is extensible in CORA's architecture. Thus CORA can cooperate with all existing local conflict resolvers, by adopting their computation.

In this section, we check two typical local optimizations based on the policy aggregation, and demonstrate the compatibilities of CORA with these optimizations.

Local TCAM compression algorithms fall into two categories in the current study: tree-based algorithms (*e.g.*, TCAM Razor Liu et al., 2010 and Ternary Razor Meiners et al., 2012) and list-based algorithms (*e.g.*, Bit Weaving Meiners et al., 2012, Redundancy Removal Liu and Gouda, 2010, Equivalent packet classifiers Dong et al., 2006). The former algorithms can aggregate rules with prefix format, as they convert rules into a decision tree and traverse it for optimal aggregations. Apart from the prefix rules, there are also ternary rules, which can be aggregated by list-based algorithms: they compare rules in a rule set and aggregate those with specific bit relations.

5.1. Tree-based compression

We use the FIB aggregation algorithm (Khare et al., 2010; Zhao et al., 2010) as the typical tree-based compression method. Specifically, FIB aggregation first transforms entries' prefix into binary string, and compose them to a binary tree. Next, it can aggregate entries that have the same next hop by inserting, deleting and combining binary tree's nodes. As an example for the sibling nodes, the algorithm traverses the cardinality tree: if one node has the same next hop with its sibling node, then these two nodes can be deleted, and a new parent node with the same next hop and a longest common subsequence prefix of those two nodes can be inserted.

The traditional method is designed for one-dimension rules, while in our scenario, there would be many more in the flow table, *e.g.*, 41 fields are supported in OpenFlow 1.5 (ONF, 2015). We assume a four-dimension case (SrcIP, DstIP, SrcPort and DstPort), and extend such algorithm to cooperate with the SDN-specific scenarios. Specifically, we first focus on the entries that are same in three dimensions, and aggregate the fourth by building a cardinality tree. Next, we repeat this process for other dimensions, until none of them can be aggregated. Consider the following two four-dimension policies, (192.168.2.2/32, 192.168.2.4/32, [1,5], [2,4]) and (192.168.2.3/32, 192.168.2.4/32, [1,5], [2,4]). Note that they have the DstIP, SrcPort and DstPort, we can form the SrcIP field into a binary tree: since the two SrcIP fields only differ in the last bit, they are sibling node in the binary tree. As a result, we can aggregate these two entries into one: (192.168.2.2/31, 192.168.2.4/32, [1,5], [2,4]). This entry can be further aggregated with others, following the above process.

5.2. List-based compression

The tree-based algorithms only work for prefix rules, *i.e.*, each field in a rule is specified as a prefix bit string (*e.g.*, 01**) where the "stars" only appear at the end of the string. In contrast, TCAM can work in the way that combine entry's every field and process it as a whole. Each field of a TCAM rule is a ternary bit string (*e.g.*, 0**1), where the stars can appear at any position. List-based algorithms, *e.g.*, Bit Weaving (Meiners et al., 2012), is designed for aggregating ternary rules. To be specific, we mix four dimensions of entries into a long bit string, and adjacent TCAM entries that have the same decision with a hamming distance of one (*e.g.*, differ in only one bit) can be merged into a single entry by replacing the bit with *. To improve the aggregation efficiency, some optimizations like bit swapping and bit merging can be applied.

6. Performance evaluation

6.1. Evaluation settings

In this section, we evaluate the semantics equivalence and optimization performance of the migrating operation in CORA. We implement CORA with ~2000 lines of python code, which takes the topology, the capacity of switches, and the current policy placement as the input, and produces a new placement as the output.

Table 2
The policy configurations used in the evaluations.

	Topology	#expanded policies	#rules	Standard deviation	#overflow switches
C_1	Stanford	4193	4902	586.07	1
	Fattree(4)	4094	5418	231.35	3
	Fattree(8)	3507	4888	107.60	3
C_2	Stanford	11 375	12 876	2041.76	2
	Fattree(4)	10 278	18 017	2811.52	3
	Fattree(8)	8054	12 906	834.19	2
C_3	Stanford	45 175	47 719	5383.16	5
	Fattree(4)	41 040	58 175	7913.27	4
	Fattree(8)	32 592	34 479	2904.13	5

We test CORA in three topologies, Stanford Backbone (Kazemian et al., 2012) and FatTree ($k = 4, k = 8$), which have 26, 20, and 80 switches respectively. For each topology, we slice the global network address (0.0.0.0–255.255.255.255) into n sections, where n is the number of edge switches in the topology. We assign those network sections to the edge switches as the host IP they connect to. We then simulate abundant high-level intents for the topology. Specifically, we use ClassBench (Taylor and Turner, 2007) to generate packet classification rules (SrcIP/Mask, DstIP/Mask, SrcPortRange, DstPortRange, Action), which can be regarded as the endpoint intent. The Action in the rules is just an integer number indicating different endpoint actions, and we choose a specific number to denote the header modifying action, which modifies the SrcIP and SrcPort randomly, to simulate an NAT function. The default rules with long mask length are removed.

By mapping the SrcIP and DstIP ranges to the edge switch, we obtain the corresponding routing intent; the simple shortest path is generated by SrcIP and DstIP, and we split the intent if the IP ranges cross network sections. Then we can generate routing policies for each switch according to the routing intent, and randomly place an endpoint policy along the routing path by the endpoint intent. Notice if the endpoint policy is a header modifying policy, we need to adjust the routing policies in the post switches to maintain the forwarding path. In our evaluations, the capacity of each switch is set to be 500.

Based on the above settings, we generate three configurations for the evaluations, as shown in Table 2, where the number of expanded policies is measured by the priority method, and the number of rules represents the entries used in TCAM-based switches. Note that CORA follows the heuristics to optimize the rule set, so given the pre-generated configurations, we could obtain a fixed optimized form from CORA. As a result, we do not need to re-perform CORA to report the average performance.

6.2. Semantics equivalence

We use header space analysis (HSA) to verify the semantics equivalence of CORA (Kazemian et al., 2012). Specifically, we test all pairwise connectivity on the generated topologies and policy sets, and record the internal forwarding path as well as the ID of actions when traversing the network. The results show that both the forwarding path and the triggered actions are the same before and after performing CORA. That is, the semantics-preserving transferring provided by CORA retains the high-level intents.

6.3. Placement optimization

We apply the heuristic algorithm proposed in Section 4 to reconcile an optimal placement that leads to lower cost of the global network. Two key metrics are measured to evaluate the performance of CORA, the global policy cost, and the standard deviation of policy placement. We use the priority method to optimize the policies in the single switch, and assume the data plane uses TCAM-based switches to demonstrate

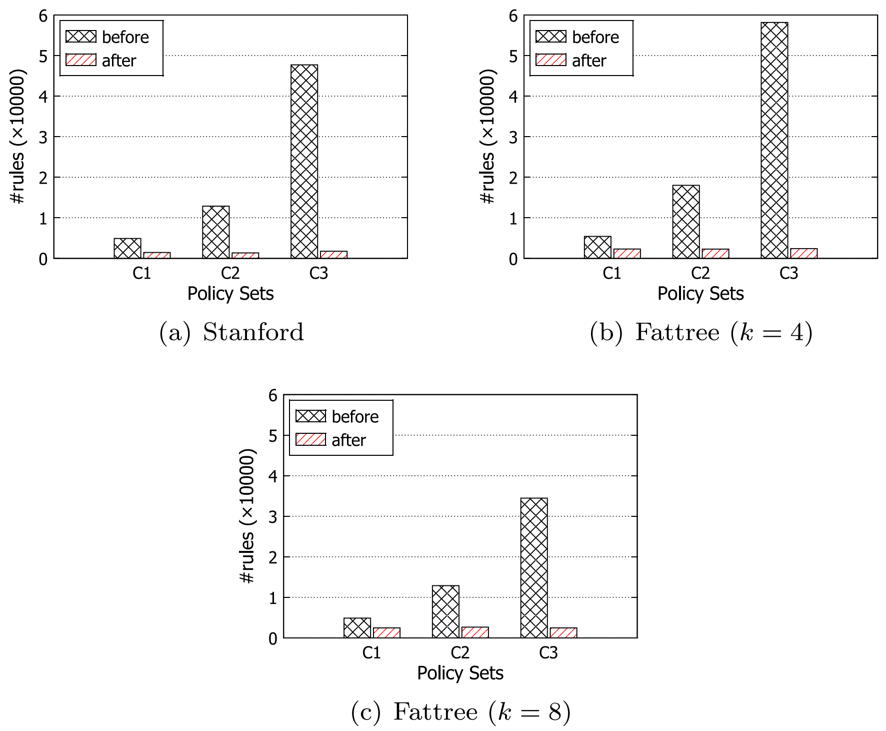


Fig. 5. The number of total rules decrement after performing CORA.

the significance, so that the policy cost is measured by the number of expanded rules.

Fig. 5 shows the policy cost decrement after performing CORA for three policy sets on different topologies. On average, 79.78% policy cost can be eliminated by properly transferring policies, and the decrement can go up to 96.31% for severe conflicting policy placement. Fig. 7 depicts similar improvements of standard deviation obtained by CORA, 96.22% and 99.73% decrements are achieved in average and at most, respectively.

In fact, the performance of CORA is determined by two factors. The first is the maldistribution of policies in the first place; if the standard deviation is high for the original placement, large profit can be expected by finding an even distribution solution. For example, C_3 on Fattree ($k = 4$) has a higher deviation than it on Fattree ($k = 8$), thus leads to a more significant decrement on the policy cost, as shown in Figs. 7(b), (c) and 5(b), (c). Second, the optimization efficiency of CORA is dependent on how many target switches can be transferred to for a single policy; more targets means larger searching space and better chance of achieving a more optimal solution. For example, due to the worse connectivity, policy sets on Stanford topology obtains lower decrements than them on Fattree ($k = 4$) topology (80.15% vs. 85.42% in average), though it has more switches (26 vs. 20), as shown in Fig. 5(a), (b).

We further compare CORA with two existing approaches, *i.e.*, packing rules with rectangle (Kang et al., 2013) (referred as *Pack*), and the local conflict resolver (referred as *Local*). Fig. 6 shows that CORA can further reduce 74.3% and 47.9% rules compared with Pack and Local, respectively. The reason to the boost of CORA mainly comes from the ability of arbitrary and flexible rule transferring. In contrast, Local only optimizes the local rule set, while Pack can only transfer the rules that can be covered by a rectangle.

Finally we want to measure the gap between CORA and the optimal solution. However, due to the extremely high complexity, we cannot obtain the optimal placement in feasible time for any of the three configurations. Instead, we use a tiny configuration that consists of only 4 switches and 100 endpoint policies, and the results show that CORA can achieve 100% optimality. We will quantify the optimality of CORA in real configurations in our future work.

6.4. Flow table aggregation

To check the compatibility with the existing local conflict resolvers, we apply the tree-based aggregation algorithm to enhance CORA.

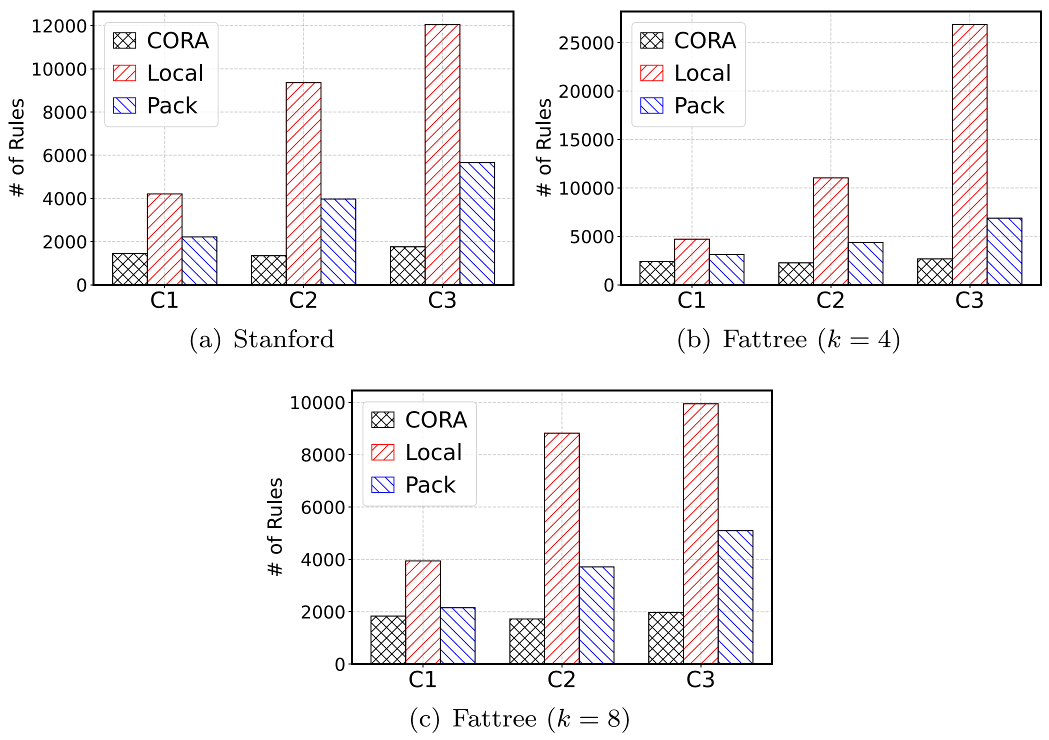
By profiling the basic case in Fig. 5, we find that the number of policies explodes because the naive priority-based optimization cuts the big policies into a set of much smaller ones, which may aggravate the number of rules in total. The aggregation-based compression method can combine many of those policies into a new bigger policy, making the policy migration more efficient. As shown in Figs. 8 and 9, the total the number of rules and the standard deviation are further decreased if applying this local optimization. This experiment proves that CORA performs better with the help of the local conflict resolver.

6.5. Overhead

We measure the time cost of performing CORA on different policy configurations to ensure the optimization can be done within acceptable overhead. Fig. 10 shows that most optimizations can be done within 40 s.

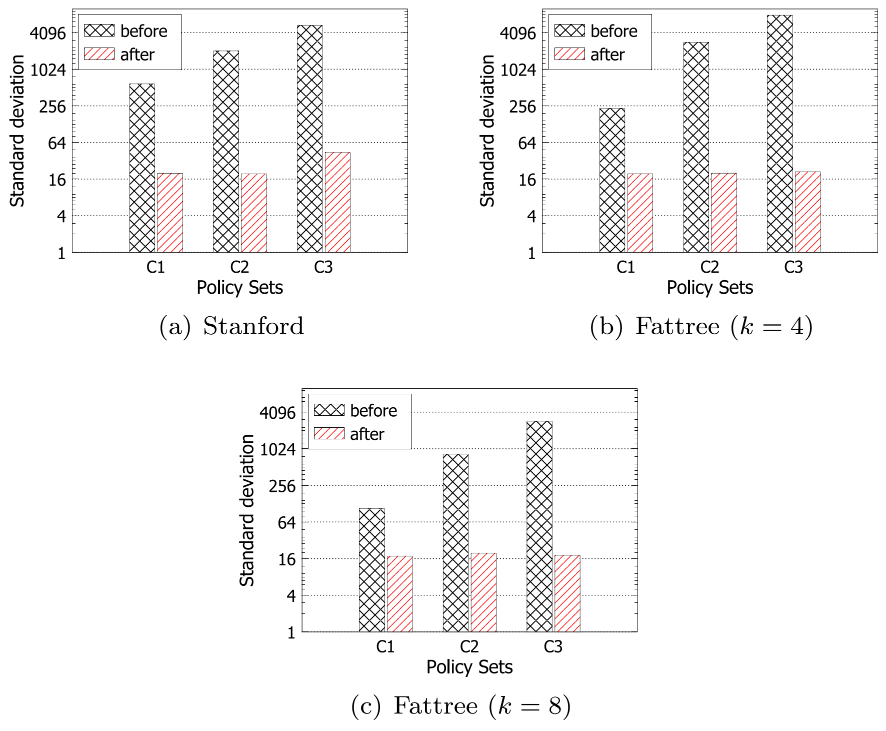
The impact factors of the time cost of CORA are similar with them of optimization efficiency, *i.e.*, the distribution of policy and the connectivity of the topology. Maldistribution and better connectivity need more time to achieve a good placement. Taking the two Fattree topologies as an example, Fattree ($k = 4$) has a large deviation with a relative worse connectivity, while in contrast, the policies are distributed more evenly in Fattree ($k = 8$) that has better connectivity. Therefore, the time cost of these two topologies are similar, as shown in Fig. 10(b), (c). Such feature also improves the scalability of CORA when handling large topologies.

We next simulate the scenario that many policies are updated by the high-level intents. Specifically, we randomly add policies and delete policies on an optimized placement, and measure how long it takes to reach a new optimal placement for CORA. Fig. 11 shows the time cost when modifying 10%–25% of the policies on Fattree ($k = 4$) topology. Due to the incremental update algorithm used in CORA, limited policies



(a) Stanford (b) Fattree $(k = 4)$ (c) Fattree $(k = 8)$

Fig. 6. The number of total rules by performing CORA, Pack and the local resolver.



(a) Stanford (b) Fattree $(k = 4)$ (c) Fattree $(k = 8)$

Fig. 7. The standard deviation decrement after performing CORA.

and switches are involved for recomputation, so the time cost is small (< 15 s) and stable.

The above experiments also depict the minor additional time consumed by the aggregation optimizations, which also benefits from the incremental update strategy.

7. Discussion

In this paper, we only transfer endpoint policies, while the routing policies are considered as fixed to ensure the semantics equivalence of routing intent. Though there are many limits for us to transfer routing policies, the effect will be better if we can take routing policies into

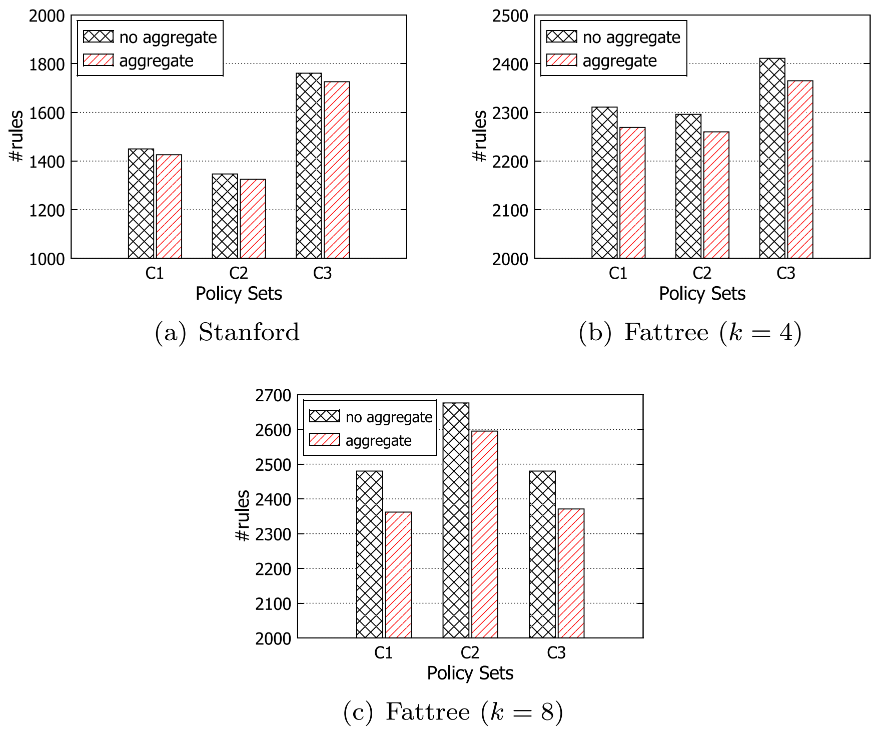


Fig. 8. The number of total rules decrement after performing flow table aggregation.

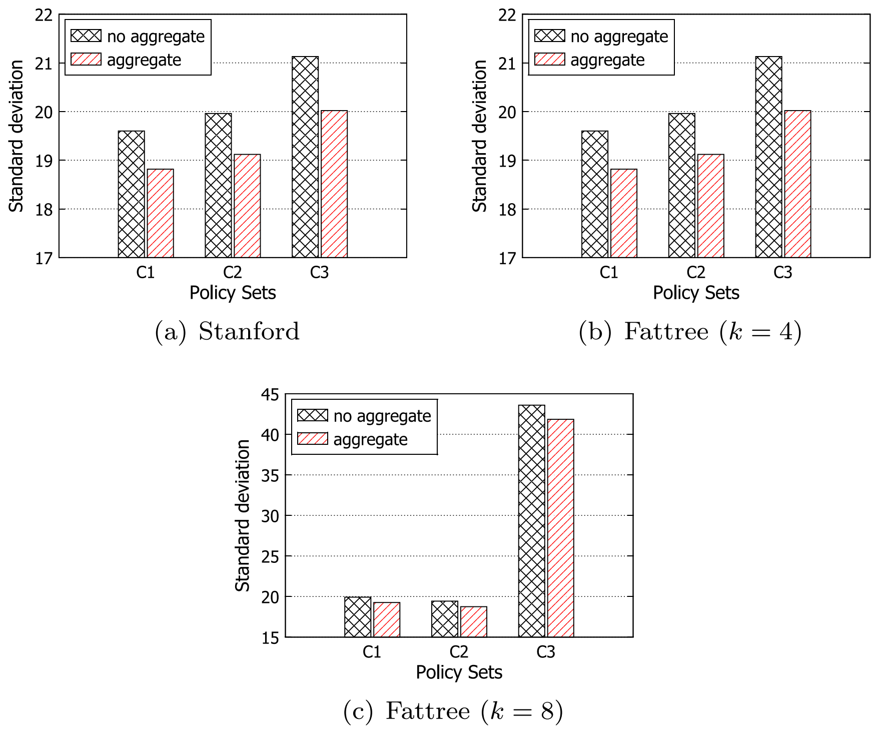
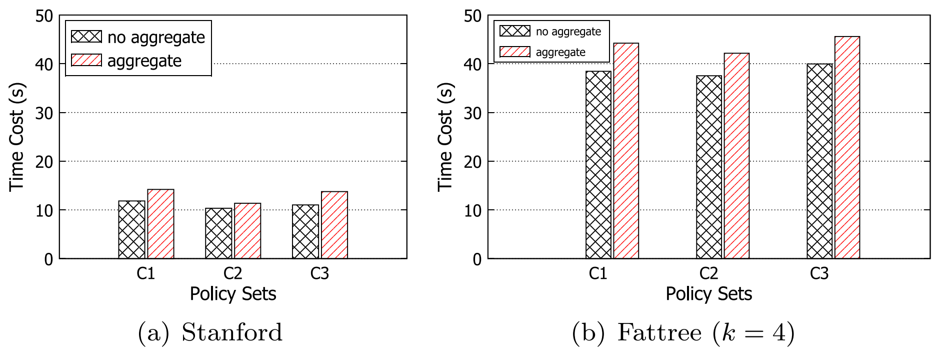


Fig. 9. The standard deviation decrement after performing flow table aggregation.

account, there will be more space for us to migrate and the process will be more flexible.

The global optimization can be further optimized if policies are separable. In the global optimization, when we transform policy to other switch, the policy is assumed as an entire one. This assumption simplifies the optimization process but it suffers a bit of performance. In some cases, the transformation of entire policy hardly decrease the storage cost. However, if splitting corresponding policies into several sub-policies, and translate part of them to other switch, flow table explosion can be extremely decreased. The process of global optimization need to be further modified to work out the optimal solution.

Besides, in SDN scenario, not just transfer, much more work can be done to further minimize the storage cost of conflicting policies since we master the policy distribution, flow table situation and the forwarding paths of the entire network in the control plane.



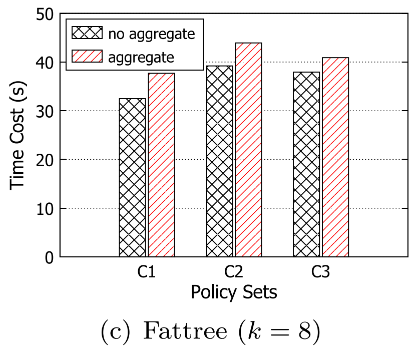
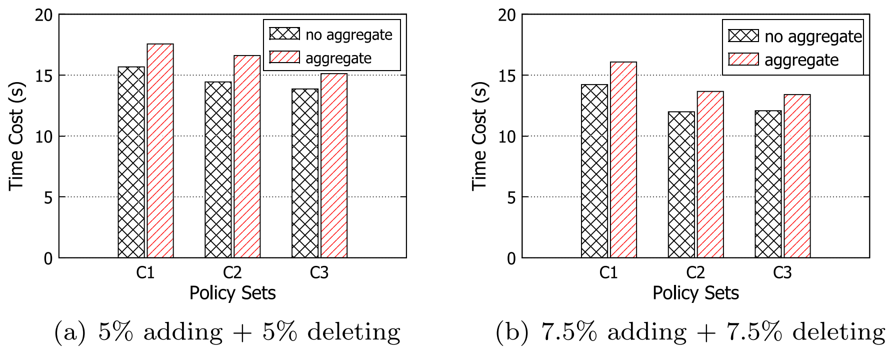


Fig. 10. The time used for performing CORA and flow table aggregation on different policy sets and topologies.



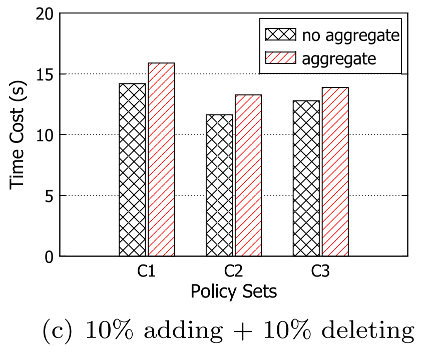


Fig. 11. The time used when updating policies in Fattree ($k = 4$) topology.

8. Related work

Many work have been done for efficient policy placement in the network. However, those works also have limitations in one or some of the following aspects. Table 3 briefly classifies those works, and we detail them in the following.

Although compressing the flow tables for IP lookup and firewall in traditional network has been widely studied, only a limited number of fields (mostly five tuples) are involved (Kogan et al., 2015; Liu and Gouda, 2010; Meiners et al., 2012; Bremler-Barr and Hendler, 2012; Katta et al., 2014; Liu et al., 2010). In SDN scenario, switches in the data plane may check an unbounded number of match fields to realize fine-grained flow control, which drastically increases the complexity of applying the traditional compressing methods. This phenomenon can be inferred from other approaches studying a broader range of SDN-based architecture (Bhatia et al., 2019a,b, 2020; Trivedi et al., 2018).

Table 3
Previous literatures related to policy conflicts resolving.

Approaches	Basic idea	Weakness
Local compressor (Kogan et al., 2015; Liu and Gouda, 2010; Meiners et al., 2012; Bremler-Barr and Hendler, 2012; Katta et al., 2014; Liu et al., 2010)	Compress flow tables in a single switch	Less effective when handling high-dimension conflicts in SDN
Network slicing (Al-Shabibi et al., 2014; Koponen et al., 2014; Drutskoy et al., 2013)	Isolate flow space for each policy	Disable the operation on the same traffic
High-level DSL (Monsanto et al., 2013; Arashloo et al., 2016; Prakash et al., 2015; Amin et al., 2019; He et al., 2017; Tian et al., 2019)	Resolve conflicts at the language level	Cannot assume all policies are described with the same DSL.

We believe with the global view offered by SDN, more benefit can be gained by properly transferring the policies. Besides, as we have mentioned in Section 1, all the existing optimizations for a single switch can be expediently employed in CORA.

Another attempt to avoid the policy conflicts is to slice the network into pieces, providing isolated flow spaces for different network functions or tenants (Al-Shabibi et al., 2014; Koponen et al., 2014; Drutskoy et al., 2013). However, such technique does not address the root cause of policy conflict: it is common to observe that more than one applications may be engaged in the same flow. Those applications will issue policies for sharing resources, *e.g.*, overlapped flow space.

Many SDN languages are proposed to facilitate composing network with individual program pieces (Foster et al., 2013; Monsanto et al., 2013; Voellmy et al., 2013; Arashloo et al., 2016; Prakash et al., 2015; Amin et al., 2019; He et al., 2017; Tian et al., 2019). The corresponding controllers of these languages will carefully place and prioritize the generated policies, so that the conflicts are resolved in the first place. We believe in near future, the rule conflicts cannot be completely avoided or detected at policy level, considering the SDN programs would not be unified by a single "perfect" language. As such, if multiple controllers coexist in a single network, none of them can handle the conflicts raise by the same-priority policies. Previous work that efficiently place the endpoint policies in different priorities are not suitable for this scenario due to the similar reason (Kang et al., 2013; Kanizo et al., 2013).

Ref. Kang et al. (2013) is the closest work to ours, which also addresses the problem of endpoint policy placement. The basic idea is to recursively find a cost-effective "rectangle" to cover the overlapped rules, process the covered flows in current switch, and leave the rest flows to the next switch (next cover), such that the number of rules can be reduced. However, since Ref. Kang et al. (2013) only selects "rectangles" in flow space, the space of "packing rules" is much smaller than CORA that can move arbitrary (fractions of) rules.

The traditional placement problem is reduced to GAP, and many existing approximate algorithms can be leveraged to obtain an optimal solution (Shmoys and Tardos, 1993). However, if we involve the policy conflicts into the problem, the cost of assigning a policy to a switch is not independent, which cannot be transformed to the original version of GAP. Several papers have investigated GAP with dependent cost, while they either assume there is only pairwise dependency between assignments (Mougouei et al., 2017; Burg et al., 1999), or just employ a global dependent variable (Tariri, 2013). In contrast, POPDC introduces more complex dependency, where the cost of each assignment is dependent on all previous assignments.

9. Conclusion

In this work, we propose CORA, a conflict razor for policies in SDN. In contrast to the policy conflict resolution in a local switch by most of the previous works, CORA solve the same problem in a distributed way. Specifically, it first detects the significant conflict-maker and then migrates it from the local switch to other switches while retaining the semantic equivalence. Since the centralized controller can grasp the global view of the entire network, such global state coordination is feasible. In this work, we identify CORA's NP hardness and propose a simple heuristic to approach the optimal solution within an acceptable time bound. Our experiments demonstrate that, CORA can effectively reduce the flow table storage occupation by at least 49% within less than 40 s. CORA can well collaborate with the existing local conflict resolver.

CRediT authorship contribution statement

Yadong Zhou: Conceptualization, Supervision. **Hao Li:** Methodology, Writing, Software. **Kaiyue Chen:** Software, Experiments. **Tian Pan:** Writing, Polishing. **Kun Qian:** Methodology. **Kai Zheng:** Methodology. **Bin Liu:** Conceptualization. **Peng Zhang:** Conceptualization. **Yazhe Tang:** Conceptualization. **Chengchen Hu:** Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 62172323 and No. 61833015) and the National Top Talent Training Program in Basic Sciences (No. 20180209).

References

Al-Shabibi, A., De Leenheer, M., Gerola, M., Koshibe, A., Parulkar, G., Salvadori, E., Snow, B., 2014. Openvirtex: Make your virtual sdns programmable. In: Proceedings of the Third Workshop on Hot Topics in Software Defined Networking. In: HotSDN '14, pp. 25–30. http://dx.doi.org/10.1145/2620728.2620741, URL http://doi.acm.org/10.1145/2620728.2620741.

Amin, R., Shah, N., Mehmood, W., 2019. Enforcing optimal acl policies using k-partite graph in hybrid sdn. Electronics 8 (6), 604.

Arashloo, M.T., Koral, Y., Greenberg, M., Rexford, J., Walker, D., 2016. Snap: Stateful network-wide abstractions for packet processing. In: Proceedings of the 2016 ACM SIGCOMM Conference. In: SIGCOMM '16, pp. 29–43. http://dx.doi.org/10.1145/2934872.2934892, URL http://doi.acm.org/10.1145/2934872.2934892.

Bhatia, J., Dave, R., Bhayani, H., Tanwar, S., Nayyar, A., 2020. Sdn-based real-time urban traffic analysis in vanet environment. Comput. Commun. 149, 162–175.

Bhatia, J., Kakadia, P., Bhavsar, M., Tanwar, S., 2019a. Sdn-enabled network coding-based secure data dissemination in vanet environment. IEEE Internet Things J. 7 (7), 6078–6087.

Bhatia, J., Modi, Y., Tanwar, S., Bhavsar, M., 2019b. Software defined vehicular networks: A comprehensive review. Int. J. Commun. Syst. 32 (12), e4005.

Bremler-Barr, A., Hendler, D., 2012. Space-efficient tcam-based classification using gray coding. IEEE Trans. Comput. 61 (1), 18–30.

Burg, J.J., Ainsworth, J., Casto, B., Lang, S.-D., 1999. Experiments with the "oregon trail knapsack problem". Electron. Notes Discrete Math. 1, 26–35.

Dong, Q., Banerjee, S., Wang, J., Agrawal, D., Shukla, A., 2006. Packet classifiers in ternary cams can be smaller. In: ACM SIGMETRICS.

Doriguzzi Corin, R., Gerola, M., Riggio, R., De Pellegrini, F., Salvadori, E., 2012. Vertigo: Network virtualization and beyond. In: 2012 European Workshop on Software Defined Networking (EWSDN). IEEE, pp. 24–29.

Drutskoy, D., Keller, E., Rexford, J., 2013. Scalable network virtualization in software-defined networks. IEEE Internet Comput. 17 (2), 20–27. http://dx.doi.org/10.1109/MIC.2012.144.

Foster, N., Guha, A., Reitblatt, M., Story, A., Freedman, M.J., Katta, N.P., Monsanto, C., Reich, J., Rexford, J., Schlesinger, C., et al., 2013. Languages for software-defined networks. IEEE Commun. Mag. 51 (2), 128–134.

He, B., Dong, L., Xu, T., Fei, S., Zhang, H., Wang, W., 2017. Research on network programming language and policy conflicts for sdn. Concurr. Comput.: Pract. Exper. 29 (19), e4218.

Jin, X., Gossels, J., Rexford, J., Walker, D., 2015. Covisor: A compositional hypervisor for software-defined networks. In: Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation. In: NSDI'15, pp. 87–101, URL http://dl.acm.org/citation.cfm?id=2789770.2789777.

Kang, N., Liu, Z., Rexford, J., Walker, D., 2013. Optimizing the one big switch abstraction in software-defined networks. In: Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies. ACM, pp. 13–24.

Kanizo, Y., Hay, D., Keslassy, I., 2013. Palette: Distributing tables in software-defined networks. In: INFOCOM, 2013 Proceedings IEEE. pp. 545–549. http://dx.doi.org/10.1109/INFCOM.2013.6566832.

Katta, N., Alipourfard, O., Rexford, J., Walker, D., 2014. Infinite cacheflow in software-defined networks. In: Proceedings of the Third Workshop on Hot Topics in Software Defined Networking. ACM, pp. 175–180.

Kazemian, P., Varghese, G., McKeown, N., 2012. Header space analysis: Static checking for networks. In: Presented as Part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12). pp. 113–126, URL https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian.

Khare, V., Jen, D., Zhao, X., Liu, Y., Massey, D., Wang, L., Zhang, B., Zhang, L., 2010. Evolution towards global routing scalability. IEEE J. Sel. Areas Commun. 28 (8), 1363–1375.

Kogan, K., Nikolenko, S., Rottenstreich, O., Culhane, W., Eugster, P., et al., 2015. Exploiting Order Independence for Scalable and Expressive Packet Classification. IEEE.

Koponen, T., Amidon, K., Balland, P., Casado, M., Chanda, A., Fulton, B., Ganichev, I., Gross, J., Gude, N., Ingram, P., et al., 2014. Network virtualization in multi-tenant datacenters. In: USENIX NSDI.

Li, H., Chen, K., Pan, T., Zhou, Y., Qian, K., Zheng, K., Liu, B., Zhang, P., Tang, Y., Hu, C., 2018. Cora: Conflict razor for policies in sdn. In: IEEE INFOCOM 2018 - IEEE Conference on Computer Communications. pp. 423–431.

Li, H., Hu, C., Zhang, P., Xie, L., 2016. Modular sdn compiler design with intermediate representation. In: Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference. ACM, pp. 587–588.

Liu, A.X., Gouda, M.G., 2010. Complete redundancy removal for packet classifiers in tcams. IEEE Trans. Parallel Distrib. Syst. 21 (4), 424–437.

Liu, A.X., Meiners, C.R., Torng, E., 2010. Tcam razor: A systematic approach towards minimizing packet classifiers in TCAMs. IEEE/ACM Trans. Netw. 18 (2), 490–500.

Meiners, C.R., Liu, A.X., Torng, E., 2012. Bit weaving: A non-prefix approach to compressing packet classifiers in tcams. IEEE/ACM Trans. Netw. (ToN) 20 (2), 488–500.

Meiners, C.R., Liu, A.X., Torng, E., 2012. Bit weaving: A non-prefix approach to compressing packet classifiers in TCAMs. IEEE/ACM Trans. Netw. 20 (2), 488–500.

Monsanto, C., Reich, J., Foster, N., Rexford, J., Walker, D., et al., 2013. Composing software defined networks. In: NSDI. pp. 1–13.

Mougouei, D., Powers, D.M., Moeini, A., 2017. An integer programming model for binary knapsack problem with value-related dependencies among elements. arXiv preprint arXiv:1702.06662.

ONF, 2015. OpenFlow Switch Specification Version 1.5.1, https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf.

Pananjady, A., Bagaria, V.K., Vaze, R., 2015. The online disjoint set cover problem and its applications. In: 2015 IEEE Conference on Computer Communications (INFOCOM). pp. 1221–1229. http://dx.doi.org/10.1109/INFOCOM.2015.7218497.

Prakash, C., Lee, J., Turner, Y., Kang, J.-M., Akella, A., Banerjee, S., Clark, C., Ma, Y., Sharma, P., Zhang, Y., 2015. Pga: Using graphs to express and automatically reconcile network policies. SIGCOMM Comput. Commun. Rev. 45 (4), 29–42. http://dx.doi.org/10.1145/2829988.2787506, URL http://doi.acm.org/10.1145/2829988.2787506.

Rottenstreich, O., Cohen, R., Raz, D., Keslassy, I., 2013. Exact worst case tcam rule expansion. IEEE Trans. Comput. 62 (6), 1127–1140. http://dx.doi.org/10.1109/TC.2012.59.

Shmoys, D.B., Tardos, E., 1993. An approximation algorithm for the generalized assignment problem. Math. Program. 62 (1–3), 461–474.

Tariri, G., 2013. The Assignment Problem with Dependent Costs (Ph.D. thesis). University of Louisville.

Taylor, D.E., Turner, J.S., 2007. Classbench: A packet classification benchmark. IEEE/ACM Trans. Netw. 15 (3), 499–511. http://dx.doi.org/10.1109/TNET.2007.893156.

Tian, B., Zhang, X., Zhai, E., Liu, H.H., Ye, Q., Wang, C., Wu, X., Ji, Z., Sang, Y., Zhang, M., et al., 2019. Safely and automatically updating in-network acl configurations with intent language. In: Proceedings of the ACM Special Interest Group on Data Communication, pp. 214–226.

Trivedi, H., Tanwar, S., Thakkar, P., 2018. Software defined network-based vehicular adhoc networks for intelligent transportation system: Recent advances and future challenges. In: International Conference on Futuristic Trends in Network and Communication Technologies. Springer, pp. 325–337.

Voellmy, A., Wang, J., Yang, Y.R., Ford, B., Hudak, P., 2013. Maple: Simplifying sdn programming using algorithmic policies. In: ACM SIGCOMM Computer Communication Review, Vol. 43. ACM, pp. 87–98.

Zhao, X., Liu, Y., Wang, L., Zhang, B., 2010. On the aggregatability of router forwarding tables. In: IEEE INFOCOM.

Yadong Zhou is an Associate Professor of School of Automation Science and Engineering at Xi'an Jiaotong University. He received his B.S. and Ph.D. degrees in Control Science and Engineering from Xi'an Jiaotong University, China, in 2004 and 2011, respectively. He was a postdoctoral researcher at The Chinese University of Hong Kong in 2014. His research focuses on Data Driven Network Security, Network Science and its Applications.

Hao Li received his Ph.D. degree in computer science from Xi'an Jiaotong University in 2016 and is now an associate professor in the School of Computer Science and Technology at the same university. His research interests are network functions and programmable network.

Kaiyue Chen received his M.S. degree in computer science from Xi'an Jiaotong University in 2019 and is now working at Tencent. This work was mainly completed before joining Tencent. His research interests is networked system.

Tian Pan received his Ph.D. degree in computer science from Tsinghua University in 2015 and is now an assistant professor in Beijing University of Posts and Telecommunications. His research interests are software defined networking and high-speed networks.

Kun Qian received his Ph.D. degree in computer science from Tsinghua University in 2020. His research interests are lossless networks and programmable networks.

Kai Zheng received the M.S. and Ph.D. degrees from Tsinghua University, China, in 2003 and 2006, respectively. His Ph.D. thesis received the first outstanding Ph.D. thesis award from the Chinese Computer Federation in 2006. He joined Huawei Technologies Co., Ltd., in 2015, as the Chief Architecture and the Director, Protocol R&D. Before that, he was a Senior Research Staff Member at IBM Research. His current research interests include data center networking, protocol intelligence, software defined (transport) protocols, WAN accelerations, and IoT protocols.

Bin Liu received the M.S. and Ph.D. degrees in computer science and engineering from Northwestern Polytechnical University, Xi'an, China, in 1988 and 1993, respectively. He is now a Full Professor with the Department of Computer Science and Technology, Tsinghua University, Beijing, China. He had led his teams to prototype numerous equipment such as large capacity of ISDN/ATM switches and high speed/core routers and transferred these prototypes to industries. His current research areas include high performance switches/routers, network processors, traffic measurement and management, in-network computing and high speed network security.

Peng Zhang received the Ph.D. degree in computer science from Tsinghua University in 2013. He is currently an Associate Professor with the School of Computer Science and Technology, Xi'an Jiaotong University. His research interests include verification, measurement, privacy, and security in computer networks.

Chengchen Hu received his Ph.D. degree from Tsinghua University, China, in 2008. He worked in Tsinghua University as an assistant professor (July. 2008–Dec. 2010) and then later severed in Xi'an Jiaotong university as associated professor (Dec. 2010–Jan. 2016) and professor since 2016, all are with the Department of Computer Science and Technology. Since the summer of 2017, he has been directing the Xilinx Research Labs Asia Pacific (XLAP). This work was mainly completed before joining XLAP. His research interests include network measurement, data center networking, and software defined networking.