CORA: Conflict Razor for Policies in SDN

Hao Li[*†], Kaiyue Chen[*†], Tian Pan[‡], Yadong Zhou[†], Kun Qian[*§]
Kai Zheng[¶], Bin Liu[§], Peng Zhang[*†], Yazhe Tang[*], Chengchen Hu[*†]
[*]Department of Computer Science and Technology, [†]MOE KLINNS Lab, Xi'an Jiaotong University
[‡]Beijing University of Posts and Telecommunications [§]Tsinghua University [¶]2012 Labs, Huawei Technologies

***Abstract*—Software Defined Network (SDN) enables flexible update of network functions with a well-defined abstraction between the control and the data plane. However, multiple active network functions 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 at least 49% within less than 40 seconds.**

I. Introduction

Software Defined Network (SDN) decouples switch's control and data plane, offering enhanced programmability via a higher-level abstraction (*i.e.*, intent-based NBI) to flexibly implement a variety of network functions. During the network operation, the operator's 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.

Considering 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 10Mbps 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$\rightarrow$limit 10Mbps, (P_2) 2-6$\rightarrow$limit 10Mbps and count, and (P_3) 7-7$\rightarrow$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 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 [9]. 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 [4, 12, 14, 17–19]. 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 [24]. Although the OpenFlow specification [1] 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 [2, 6, 15] provides individual flow spaces for each network function to isolate the conflicts, which actually disables multiple network functions to operate on the same traffic. Nowadays, many high-level SDN languages offer the resource constraints in their syntaxes and compilation process, which can be adopted to generate a more optimal placement of policies to mitigate

[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.

the conflicts beforehand [3, 10, 23]. 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 [9].

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. It is worth of 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.

In summary, we make the following 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.

The reminder of this paper is organized as follows. Section II demonstrates the policy conflict problem and our basic idea. Section III proposes the semantic-preserving transferring to retain the high-level intents. Section IV defines the problem of finding an optimal placement from the global view, and design a heuristic algorithm to obtain a near-optimal solution within acceptable time cost. Section V evaluates CORA's performance. And after discussing the related work in Section VI, Section VII concludes this paper.

II. Transfer Policy to Raze the Conflicts

A. Policy Conflict Problem

SDN high-level languages specify two categories of operators' intents: the routing intent and the endpoint intent [10]. 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 [3, 23]. 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 [9]. 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, iff $p_1.sw = p_2.sw$, $p_1.pri = p_2.pri$, $p_1.fs \cap p_2.fs \neq \varnothing$ 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 #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 #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

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

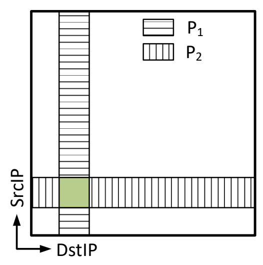


Fig. 1. Two policies conflict on two fields.

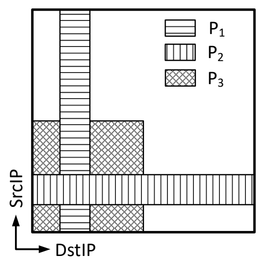


Fig. 2. More conflicts caused by a third policy.

TABLE I
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

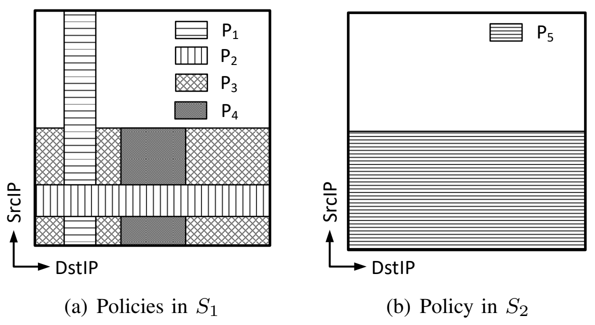
solid shadowed part in Fig. 1), *i.e.*, $P' = (P_1.sw, P_1.pri + 1, P_1.fs \cap P_2.fs, P_1.a \cup P_2.a)$.

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 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 I. Even worse, many switches use TCAM to implement the flow tables, which would expands the entries to represent the range values [18]. 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 [24]. 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.

B. Basic Idea of CORA

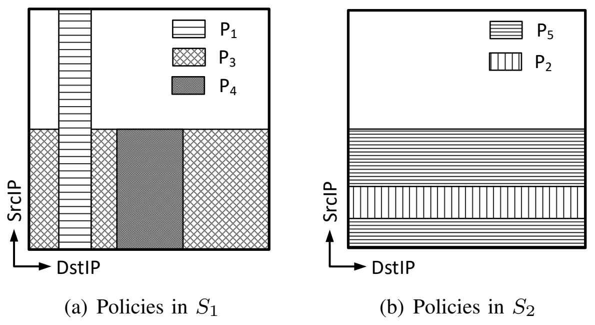
All the existing techniques focus on how to minimize the overhead when decoupling the conflicts. In contrast, CORA treat the same problem from a different angle, aiming to eliminate the conflicts by migrating the policies, *i.e.*, to reduce the #conflicts in global network instead of #expanded policies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



(a) Policies in S_1 (b) Policy in S_2

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



(a) Policies in S_1 (b) Policies in S_2

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

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.

III. 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.

A. 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 [16]. 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.

B. 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, $ST(p, s) = \varnothing$, 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; $ST(p, s) = \{p[fs \mapsto p.fs \setminus p'.fs], p'[sw \mapsto s]\}$, 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.

IV. Finding Optimal Placement

A. Problem Formulation

The goal of policy migration is to recursively find a better placement that leads to a reduced #rules in the data plane. Previously, the optimal policy placement has been discussed in several papers [3, 10], 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.

```
Procedure: Policy Divide
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:             $P \leftarrow P \setminus \{p\} \cup ST(p, s)$
```

Fig. 5. Divide the original policies into fragments, so they can be independently transferred.

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 #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 #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.

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 Fig 5.

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 #decoupled policies or #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 $S = \{S_i\}, i = 1, .., n$, 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 #sets to the #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.*, #policies/rules.

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

$$\begin{aligned}
\text{maximize} \quad & \sum_{s \in S} (x_s / \sum_{ps \in s} w(ps)) && (1) \\
\text{subject to} \quad & \sum_{s: e \in s} x_s = 1, \quad \text{for all } e \in P && (2) \\
& \sum_{s \in S} x_s \leq m, \quad \text{for all } s \in S && (3) \\
& x_s \in \{0, 1\}, \quad \text{for all } s \in S && (4)
\end{aligned}$$

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 #sets to #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 [22].

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.

$$\begin{aligned}
\text{maximize} \quad & \sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} && (5) \\
\text{subject to} \quad & \sum_{j=1}^{n} r_{ij} x_{ij} \leq W_i, \quad i = 1, ..., m && (6) \\
& \sum_{i=1}^{m} x_{ij} = 1, \quad j = 1, ..., n && (7) \\
& \sum_{j=1}^{n} x_{ij} = 1, \quad i = 1, ..., m && (8) \\
& x_{ij} \in \{0, 1\}, \quad i = 1, ..., m, j = 1, ..., n && (9)
\end{aligned}$$

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.

B. Heuristics of Near-Optimal Placement Searching

Since we have reduced POPDC to two well-known NP problems, 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 increase the problem scale. Therefore, in this section we propose

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:                draw back the transfer of $cm$ and restore $C$, $D$, $K$,
                   $T$, $O$ accordingly
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
```

Fig. 6. Greedily searching for the optimal placement, each time transferring the highest-cost policy to a switch that leads to the largest profit.

some simple heuristics to approximately approach the optimal policy placement, under the acceptable time consumption. Specifically, the optimal placement is expected to satisfy the following requirements: (1) #rules in each switch should not go beyond the capacity of the switch, (2) the total #rules are minimized for the entire network, and (3) the standard deviation 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 #conflicts produced by the policy as the cost instead of measuring the precise #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 #switches that exceeds the capacity in current placement, and O to denote the total #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 #policies or #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 $B > 0$. The complete process is illustrated in Fig. 6. 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.

V. Performance Evaluation

A. 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.

We test CORA in three topologies, Stanford Backbone [13] 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 #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 [27] 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.

TABLE II
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	11375	12876	2041.76	2
	Fattree(4)	10278	18017	2811.52	3
	Fattree(8)	8054	12906	834.19	2
C_3	Stanford	45175	47719	5383.16	5
	Fattree(4)	41040	58175	7913.27	4
	Fattree(8)	32592	34479	2904.13	5

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 II, where the #expanded policies is measured by the priority method, and #rules represents the entries used in TCAM-based switches.

B. Semantics Equivalence

We use header space analysis (HSA) to verify the semantics equivalence of CORA [13]. 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.

C. Placement Optimization

We apply the heuristic algorithm proposed in Section IV 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 the significance, so that the policy cost is measured by the #expanded rules.

Fig. 7 shows the policy cost decrement after performing CORA for three policy sets on different topologies. In 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. 8 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 Fig. 8(b), (c) and Fig. 7(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. 7(a), (b).

D. Overhead

We measure the time cost of performing CORA on different policy configurations to ensure the optimization can be done within acceptable overhead. Fig. 9 shows that most optimizations can be done within 40 seconds.

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. 9(b), (c). Such feature also improves the scalability of CORA when handling large topologies.

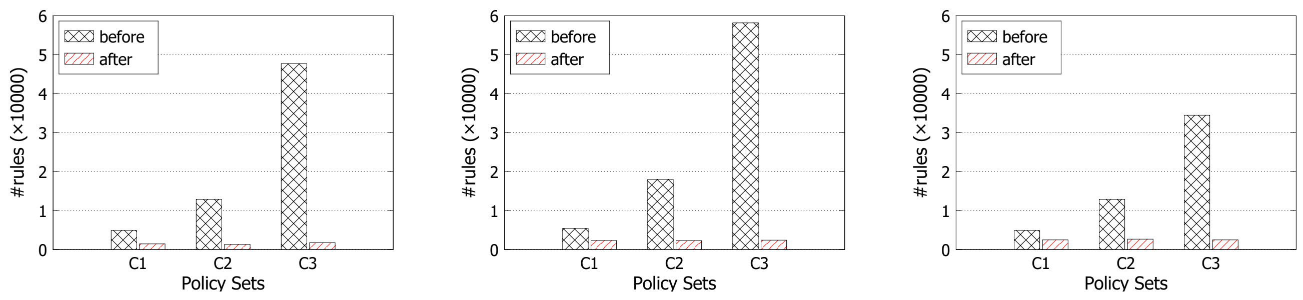
We next simulate the scenario that many policies are updated by the high-level intents. Specifically, we randomly add policies and delete policies on an optimized placement, and measure how long it takes to reach a new optimal placement for CORA. Fig. 10 shows the time cost when modifying 10%–25% of the policies on Fattree ($k = 4$) topology. Due to the incremental update algorithm used in CORA, limited policies and switches are involved for recomputation, so the time cost is small (< 15 seconds) and stable.

VI. 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.

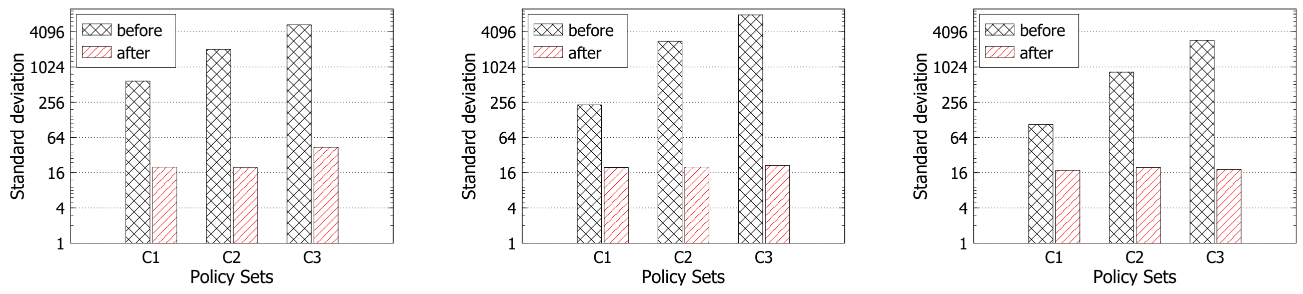
Although compressing the flow tables for IP lookup and firewall in traditional network has been widely studied, only a limited number of fields (mostly five tuples) are involved [4, 12, 14, 17–19]. In SDN scenario, switches in the data plane may check an unbounded number of match fields to realize fine-grained flow control, which drastically increases the complexity of applying the traditional compressing methods.

Another attempt to avoid the policy conflicts is to slice the network into pieces, providing isolated flow spaces for different network functions or tenants [2, 7, 15]. However, such



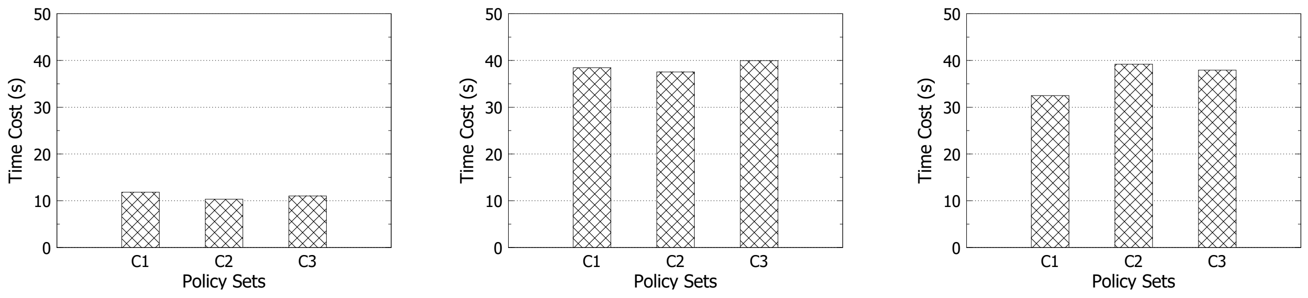
(a) Stanford (b) Fattree ($k = 4$) (c) Fattree ($k = 8$)

Fig. 7. The #total rules decrement after performing CORA.



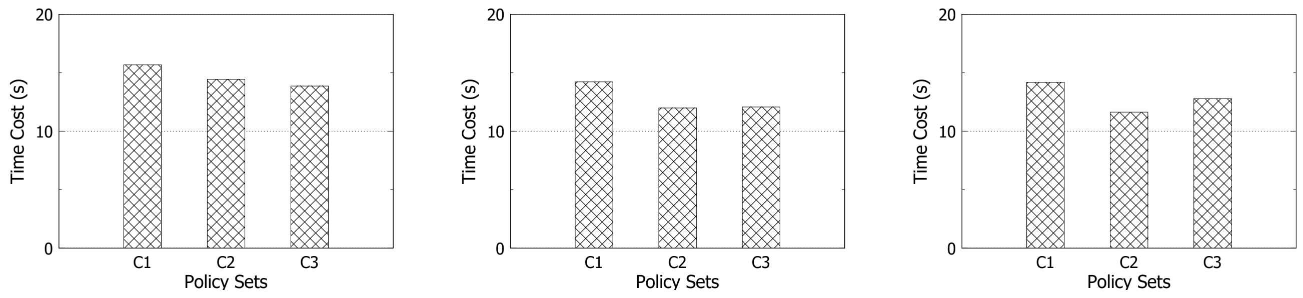
(a) Stanford (b) Fattree ($k = 4$) (c) Fattree ($k = 8$)

Fig. 8. The standard deviation decrement after performing CORA.



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

Fig. 9. The time used for performing CORA on different policy sets and topologies.



(a) 5% adding + 5% deleting (b) 7.5% adding + 7.5% deleting (c) 10% adding + 10% deleting

Fig. 10. The time used when updating policies in Fatatree ($k = 4$) topology.

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 [3, 8, 20, 23, 28]. The corresponding controllers of these languages will carefully place and prioritize the generated policies, so that the conflicts are resolved in the first place. However, 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 [10, 11].

Many existing approaches optimizes the storage of policies/rules in a local manner [4, 9, 12, 14, 17, 19]. 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 I, all the existing optimizations for a single switch can be expediently employed in CORA.

The traditional placement problem is reduced to GAP, and many existing approximate algorithms can be leveraged to obtain an optimal solution [25]. 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's only pairwise dependency between assignments [5, 21], or just employ a global dependent variable [26]. In contrast, POPDC introduces more complex dependency, where the cost of each assignment is dependent on all previous assignments.

VII. 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 seconds. CORA can well collaborate with the existing local conflict resolver.

Acknowledgment

The authors would like to thank anonymous reviewers for their valuable comments. This work is partially supported by the National Key Research and Development Program of China (2017YFB0801703), NSFC (61702407, 61672425, 61702049, 61373143, 61432009, 61772412, U1736205, 61572397, U1766215, U1736205), Huawei Innovation Research Program (HIRP), Fundamental Research Project of Natural Science in Shaanxi Province (2016JM6066). Peng Zhang is supported by the K. C. Wong Education Foundation.

References

[1] Openflow switch specification version 1.5.1. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf.

[2] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, G. Parulkar, E. Salvadori, and B. Snow. Openvirtex: Make your virtual sdns programmable. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 25–30, 2014.

[3] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker. Snap: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 29–43, 2016.

[4] A. Bremler-Barr and D. Hendler. Space-efficient tcam-based classification using gray coding. *IEEE Transactions on Computers*, 61(1):18–30, 2012.

[5] J. J. Burg, J. Ainsworth, B. Casto, and S.-D. Lang. Experiments with the oregon trail knapsack problem. *Electronic Notes in Discrete Mathematics*, 1:26–35, 1999.

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

[7] D. Drutskoy, E. Keller, and J. Rexford. Scalable network virtualization in software-defined networks. *IEEE Internet Computing*, 17(2):20–27, March 2013.

[8] N. Foster, A. Guha, M. Reitblatt, A. Story, M. J. Freedman, N. P. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, et al. Languages for software-defined networks. *Communications Magazine, IEEE*, 51(2):128–134, 2013.

[9] X. Jin, J. Gossels, J. Rexford, and D. Walker. Covisor: A compositional hypervisor for software-defined networks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 87–101, 2015.

[10] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the one big switch abstraction in software-defined networks. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 13–24. ACM, 2013.

[11] Y. Kanizo, D. Hay, and I. Keslassy. Palette: Distributing tables in software-defined networks. In *INFOCOM, 2013 Proceedings IEEE*, pages 545–549, April 2013.

[12] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. Infinite cacheflow in software-defined networks. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 175–180. ACM, 2014.

[13] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, 2012.

[14] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, P. Eugster, et al. Exploiting order independence for scalable and expressive packet classification. 2015.

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

[16] H. Li, C. Hu, P. Zhang, and L. Xie. Modular sdn compiler design with intermediate representation. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 587–588. ACM, 2016.

[17] A. X. Liu and M. G. Gouda. Complete redundancy removal for packet classifiers in tcams. *Parallel and Distributed Systems, IEEE Transactions on*, 21(4):424–437, 2010.

[18] A. X. Liu, C. R. Meiners, and E. Torng. Tcam razor: A systematic approach towards minimizing packet classifiers in tcams. *IEEE/ACM Transactions on Networking*, 18(2):490–500, April 2010.

[19] C. R. Meiners, A. X. Liu, and E. Torng. Bit weaving: A non-prefix approach to compressing packet classifiers in tcams. *IEEE/ACM Transactions on Networking (ToN)*, 20(2):488–500, 2012.

[20] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker, et al. Composing software defined networks. In *NSDI*, pages 1–13, 2013.

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

[22] A. Pananjady, V. K. Bagaria, and R. Vaze. The online disjoint set cover problem and its applications. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 1221–1229, April 2015.

[23] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang. Pga: Using graphs to express and automatically reconcile network policies. In *Proceedings of ACM SIGCOMM*, SIGCOMM '15, pages 29–42, 2015.

[24] O. Rottenstreich, R. Cohen, D. Raz, and I. Keslassy. Exact worst case tcam rule expansion. *IEEE Transactions on Computers*, 62(6):1127–1140, June 2013.

[25] D. B. Shmoys and É. Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical programming*, 62(1-3):461–474, 1993.

[26] G. Tariri. *THE ASSIGNMENT PROBLEM WITH DEPENDENT COSTS*. PhD thesis, University of Louisville, 2013.

[27] D. E. Taylor and J. S. Turner. Classbench: A packet classification benchmark. *IEEE/ACM Transactions on Networking*, 15(3):499–511, June 2007.

[28] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying sdn programming using algorithmic policies. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 87–98. ACM, 2013.