Improving the Path Programmability for Software-Defined WANs under Multiple Controller Failures

Zehua Guo*, Songshi Dou*, Wenchao Jiang†
*Beijing Institute of Technology †Singapore University of Technology and Design

***Abstract*—Enabling path programmability is an essential feature of Software-Defined Networking (SDN). During controller failures in Software-Defined Wide Area Networks (SD-WANs), a resilient design should maintain path programmability for offline flows, which were controlled by the failed controllers. Existing solutions can only partially recover the path programmability rooted in two problems: (1) the implicit preferable recovering flows with long paths and (2) the sub-optimal remapping strategy in the coarse-grained switch level. In this paper, we propose ProgrammabilityGuardian to improve the path programmability of offline flows while maintaining low communication overhead. These goals are achieved through the fine-grained flow-level mappings enabled by existing SDN techniques. ProgrammabilityGuardian configures the flow-controller mappings to recover offline flows with a similar path programmability, maximize the total programmability of the offline flows, and minimize the total communication overhead for controlling these recovered flows. Simulation results of different controller failure scenarios show that ProgrammabilityGuardian recovers all offline flows with a balanced path programmability, improves the total programmability of the recovered flows up to 68%, and reduces the communication overhead up to 83%, compared with the baseline algorithm.**

***Index Terms*—software-defined networking, wide area networks, control failure, resiliency, programmability**

I. Introduction

Maintaining the control resiliency is a critical concern to employ Software-Defined Networking (SDN) into Wide Area Networks (WANs), known as the SD-WANs. In the SD-WAN, the data plane is composed of multiple network domains, each of which has many SDN switches distributed at different physical locations. The control plane, however, consists of SDN controllers, each of which is a network control software installed in a physical server or a virtual machine to control these physical SDN switches within its domain [1]. Due to some unexpected issues (e.g., hardware/software bugs, power failure), the SDN controllers may fail. Failed controllers make all connected switches *offline*, losing the ability to change the paths of flows that traverse them, known as the *path programmability*. As a consequence, these flows become *offline flows*. Recovering the path programmability of offline flows is at the core of maintaining the control resiliency under unexpected controller failures.

The state-of-the-art solutions recover the path programmability by mapping switches to controllers either in a static manner or a dynamic manner. The static solutions optimize the network deployment by carefully selecting the placement of controllers and the connection between controllers and switches to alleviate the impact of the potential controller failure [2][3][4]. However, these solutions usually ignore the switches' different control load and the dynamic change of controller's control resource. Thus, they are neither efficient nor effective in practical settings.

The dynamic solutions outperform the static solutions since they make the remapping decisions of offline switches to active controllers in real time by considering the switches and controllers' status at the moment [5]. Despite they can partially recover the path programmability, the dynamic solutions suffer from two problems. First, the path programmability of the recovered flows are unbalanced. Typically, only a limited number of offline flows with *long* paths are recovered to become programmable. Second, existing solutions are sub-optimal in recovering path programmability because they are optimized in the coarse-grained switch level instead of the fine-grained flow level.

To fill the gaps in the literature, in this paper, we propose ProgrammabilityGuardian to recover offline flows with improved programmability while maintaining low communication overhead for controlling offline flows. It is inspired by the flow-level control enabled in existing SDN techniques (e.g., FlowVisor [6]). Instead of remapping an offline switch and all the flows traversing it to an active controller, ProgrammabilityGuardian decides the optimal active controller that each flow in each offline switch should be managed by. Our remapping decisions carefully consider multiple critical factors, including the number of recovered flows, the balance of the their path programmability, the total path programmability of all offline flows, the diversity of the physical locations and control resources of active controllers, as well as the communication overhead in the process. The problem is formulated as an Integer Programming optimization problem called *Optimal Flow-Controller Mapping (OFCM)* problem. ProgrammabilityGuardian solves the problem to dispatch the control of offline flows to active controllers.

The contributions of this paper are summarized as follows:

- We propose ProgrammabilityGuardian to improve the path programmability of recovered flows with low communication overhead under multiple controller failures through the fine-grained flow-level remapping enabled in existing SDN techniques.
- We formulate the flow recovery problem as an optimization problem called OFCM problem and propose an efficient heuristic algorithm to solve the problem.
- We evaluate the performance of ProgrammabilityGuardian under different controller failure scenarios. Simulation results show that ProgrammabilityGuardian recovers all offline flows with a balanced path pro-

Corresponding Author: Zehua Guo
978-1-7281-6887-6/20/$31.00 2020 IEEE

grammability, improves the total programmability of the recovered flows up to 68%, and reduces the communication overhead up to 83%, compared with the baseline algorithm.

The rest of the paper is organized as follows. In Section II, we introduce the control plane resiliency and the motivation of this paper. Section III overviews our ProgrammabilityGuardian design, and Section IV mathematically formulates our flow recovery problem as the OFCM problem. Section V proposes ProgrammabilityGuardian to efficiently solve the problem. We evaluate and analyze the performance of ProgrammabilityGuardian in Section VI. Section VII introduces related works, and Section VIII concludes this paper.

II. Background and Motivation

A. Control plane resiliency of SD-WANs

In the SD-WAN, the core of maintaining control plane resiliency is to recover the programmability of offline flows under unexpected controller failures. It is complex under practical SDN constraints. First, the goal is to maximize the recovered path programmability for the maximum SDN control function, as well as balancing the recovered path programmability to deal with the dynamic flow variation on the paths. Second, the ability of active controllers to recover offline flows is restricted by their available control resources. Third, performance metrics (e.g., the communication overhead between switches and controllers to control flows) are also considered to provide quick responses for handling flows from offline switches during the recovery of offline flows.

Existing control resiliency solutions recover the programmability of offline flows in the switch level. For a failed controller, existing solutions adopt the default path programmability recovery solution in OpenFlow [7] to establish new mappings from the offline switches to active controllers. By mapping one offline switch to an active controller, all flows that traverse this switch are controlled by this controller and become programmable.

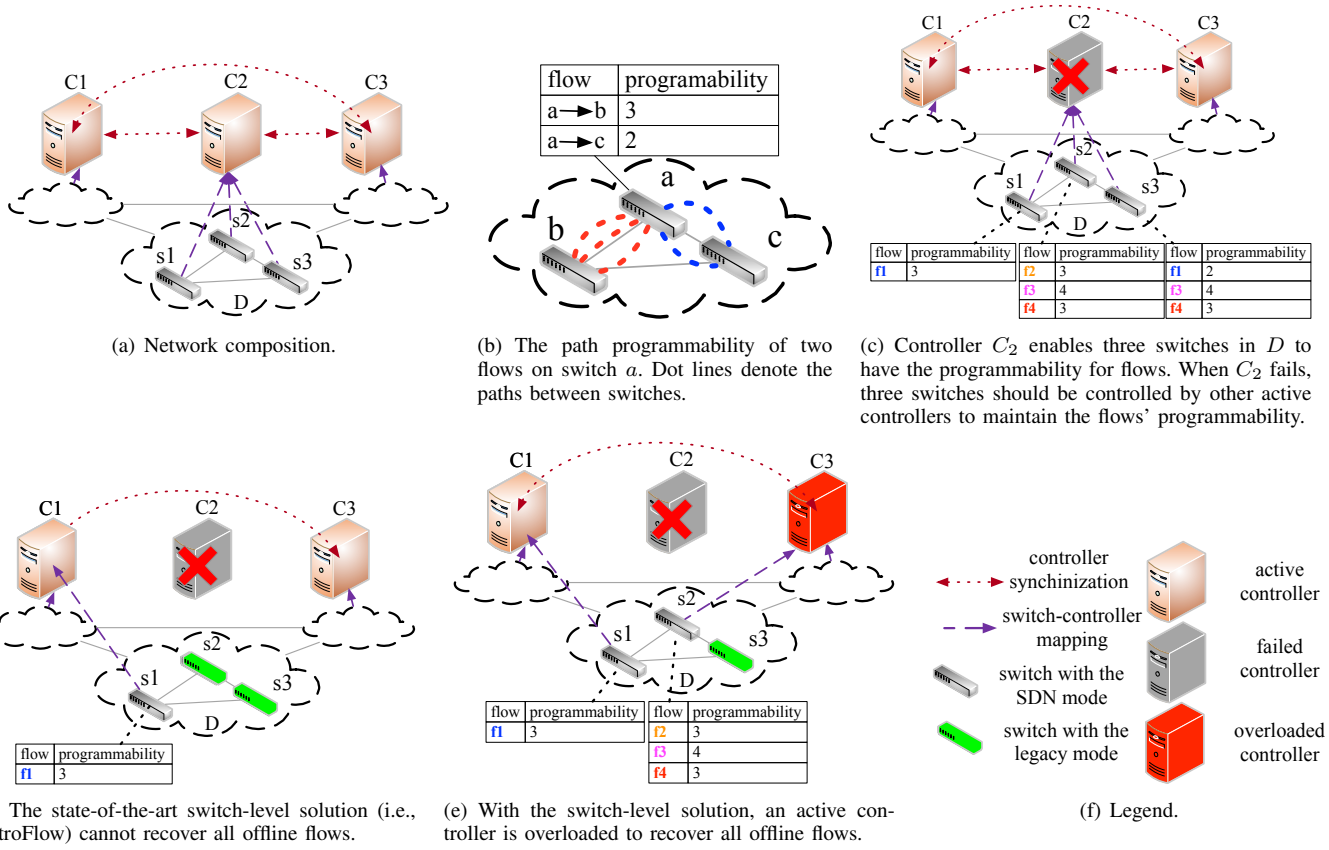
There are two typical solutions that employ the switch-level mapping to maintain the control plane's resiliency during controller failures: static solution and dynamic solution. The static solution is to optimally place, select, and map backup controllers to switches before controller failures. This solution is neither efficient due to over-reserving the control resource from controllers nor effective because of exaggerating the control ability of controllers.

The dynamic solution considers the network variation in real time to remap offline switches to active controllers. The state-of-the-art solution called RetroFlow [5] takes advantage of hybrid routing from commercial SDN switches (e.g., Brocade MLX-8 PE [8]) to establish mappings between offline switches and controllers. During controller failures, RetroFlow dynamically sets up some offline switches work under the legacy routing mode without the controllers and maps the rest offline switches with the SDN routing mode to active controllers. By changing a pure SDN to a hybrid SDN, RetroFlow reduces the active controllers' load to control offline switches and recovers many offline flows to become programmable.

B. Limitation of switch-level mapping solutions

However, employing the switch-level mapping is sub-optimal in recovering the programmability of offline flow during controller failures under the constraints of controllers' control load and resource. Here we discuss four scenarios of controller failures based on the failed controllers' control load and active controllers' control resource: (1) the failed controllers' control load is low, and the active controllers' control resource is high; (2) both the failed controllers' control load and active controllers' control resource are low; (3) both the failed controllers' control load and active controllers' control resource are high; (4) the failed controllers' control load is high, and the active controllers' control resource is low. A failed controller's control load refers to the load of controlling offline flows in offline switches associated with this controller, while an active controller's control resource refers to its available ability for recovering offline flows. Under the four scenarios, existing solutions have the following limitations and could affect the control of SD-WAN:

- **Unbalanced programmability of offline flows**: RetroFlow is the state-of-the-art solution using the switch-level mapping. It reduces the control load for flow recovery and works well for scenario (1) by configuring some offline switches to work under the legacy routing mode without the controllers. However, for scenarios (2) and (3), RetroFlow does not always work well. With the switch-level mapping, an active controller could be overloaded if a mapping is configured between this controller and an offline switch, which needs more control resource than the controller can provide. As a result, this controller's normal operation could be significantly affected [9]. This situation becomes more serious for the complicated scenario (4) under multiple controller failures. In the experiment [5], when 90% of offline flows are set to be recovered, RetroFlow cannot reach this objective for all testing cases. Specifically, some recovered flows can have high programmability of multiple rerouting paths while others are not recovered and cannot be rerouted at all. If an offline flow is not recovered, and this flow's traffic load become really heavy, existing solutions cannot reroute this flow to improve the balancing performance of the SD-WAN. Even if all offline flows are recovered to become programmable, their programmablilty could be unbalanced.
- **Under utilization of active controllers**: The switch-level mapping solutions may cause the controller under-utilization, which fails to map some offline flows to the active controllers even when the active controllers are not fully occupied. For instance, one offline switch requires the control load of three flows, but two active controllers can only provide the control resource of two flows and one flow, individually. The total available control resource perfectly matches the control load. However, the fixed requirement of one switch controlled by only one controller from the switch-level mapping prevents this switch being separately mapped to the two controllers, and the two



(a) Network composition.

(b) The path programmability of two flows on switch a. Dot lines denote the paths between switches.

(c) Controller C_2 enables three switches in D to have the programmability for flows. When C_2 fails, three switches should be controlled by other active controllers to maintain the flows' programmability.

(d) The state-of-the-art switch-level solution (i.e., RetroFlow) cannot recover all offline flows.

(e) With the switch-level solution, an active controller is overloaded to recover all offline flows.

(f) Legend.

Fig. 1: A motivation example of existing switch-level solutions that establish switch-controller remapping under controller failure.

controllers' resource are wasted and not be fully utilized to recover offline flows. Consequently, offline flows are not fully recovered, and the control function of SDN degrades.

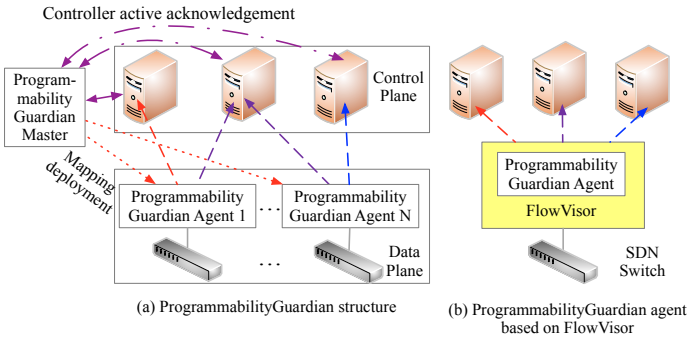
C. Example

To better illustrate our points, we use a motivation example in Fig. 1 to show flow-level solutions' limitations under controller failures. In Fig. 1(a), an SD-WAN is composed of three domains, and each domain is managed by one master controller and connected to two backup controllers. Since we mainly focus on domain D, this example does not show details of the other two domains. Controller C_2 is the master controller of domain D and controls several switches in D. For simplicity, we only show SDN switches $s1$-$s3$ here. The three switches could be connected with each other directly or indirectly though other switches, which are not shown in the figure. Controllers C_1 and C_3 are backup controllers of domain D. The three controllers maintain the consistent network information via controller synchronization [10]. Thus, all the three controllers have the consistent information for D. The control resource of a controller and the control load of a switch are measured by the number of flows. In this example, without interrupting a controller's normal operations, both C_1 and C_3 are only able to control two flows, respectively, and the total control resource of two controllers is to control four flows. Fig. 1(b) illustrates the path programmability of two flows on switch a. For switch a, the path programmability of one flow denotes the ability of switch a to change this flow's path. For flow from a to b, there are three paths traversing switch a, and its programmability is three on switch a. Similarly, flow from a to c has two paths on switch a, and its programmability is two on switch a.

The controllers of an SD-WAN could fail. In Fig. 1(c), controller C_2 fails, and the control of the three switches in D must be handed over to two active controllers C_1 and C_3 to control flows in D. As explained in Fig. 1(a), the three switches could be connected with each other directly or indirectly though other switches, and there could be multiple paths among $s1$, $s2$, and $s3$. The path programmability of flows provided by C_2 on three switches is shown in Fig. 1(c).

Three issues are raised using the switch-level solutions to recover offline flows:

1) Unbalanced programmability of offline flows: Fig. 1(d) shows the result of state-of-the-art switch-level solution (i.e., RetroFlow). In this figure, under the constraint of controllers' control resources, switch $s1$ is configured with the SDN mode and mapped to active controller C_3 while other two switches in D are configured with the legacy mode without the controllers. With this solution, only flow $f1$ is recovered with the programmability of 3. This solution prevents controllers from being overloaded but cannot recover each flow to become programmable.



(a) ProgrammabilityGuardian structure (b) ProgrammabilityGuardian agent based on FlowVisor

Fig. 2: Structure and processing logic of ProgrammabilityGuardian.

2) Under utilization of active controllers: In Fig. 1(d), flows $f2$-$f4$ do not traverse switch $s1$, and thus they are not recovered and cannot be rerouted. Controller C_3 still has some available control resource, but the resource is not enough to handle either $s2$ or $s3$ under the constraint of preventing active controllers from being overloaded. Thus, C_3 is not fully occupied to recover flows $f2$-$f4$.
3) Controller overload: Fig. 1(e) shows the result of recovering all offline flows with the switch-level solution. In this figure, switch $s2$ is configured with the SDN mode and mapped to C_3 to recover offline flows $f2$ -$f4$. This solution enables all flows to become programmable at the cost of overloading controller C_3, which only has the resource for two flows but is assigned with three flows.

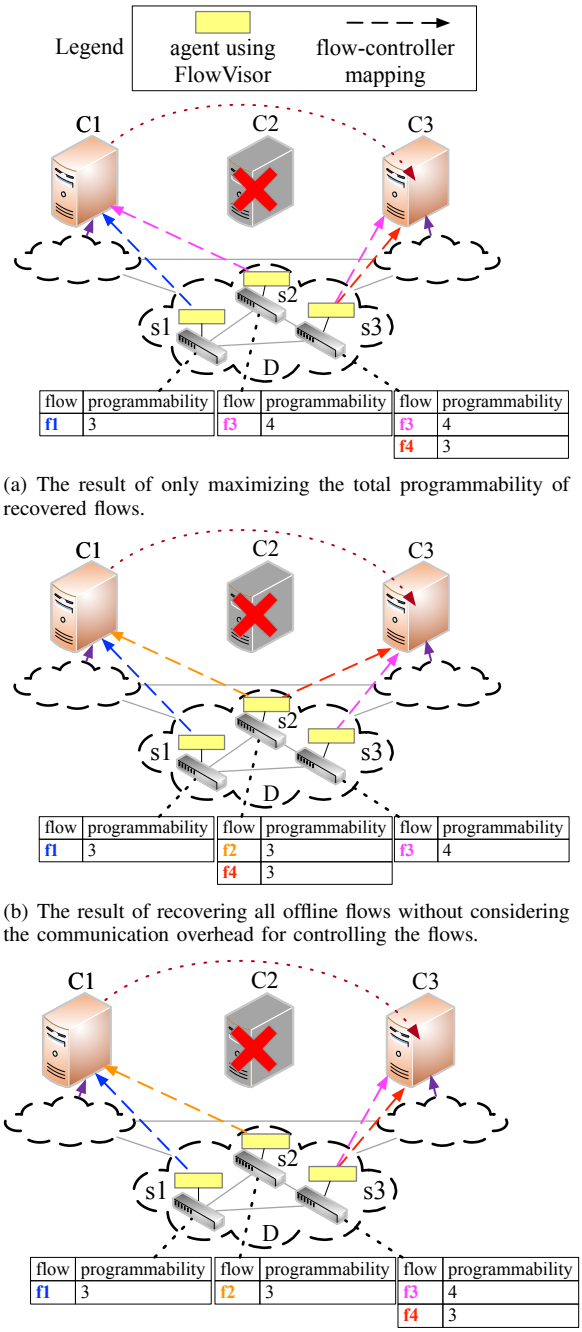
The above two examples show that existing switch-level solutions cannot well recover the programmability of offline flows under controller failures.

III. Overview of ProgrammabilityGuardian

In this section, we introduce the structure and some key design considerations of ProgrammabilityGuardian.

A. Design overview

ProgrammabilityGuardian aims at improving the path programmability in offline flow recovery under controller failures by realizing the fine-grained flows to controllers mappings. It is distinct from existing works that adopt the switch-level switch-controller mappings. Fig. 2(a) shows the structure and processing logic of ProgrammabilityGuardian, which consists of one ProgrammabilityGuardian master and one ProgrammabilityGuardian agent above each switch. ProgrammabilityGuardian master connects to controllers to detect their activities. If one or multiple controllers are identified as failure, the master calculates mappings between offline flows from offline switches and active controllers and deploy the mappings into ProgrammabilityGuardian agents of offline switches. The mappings between offline flows and active controllers are decided by solving an optimization problem named the Optimal Flow-Controller Mapping (OFCM) problem, which is detailed in Section IV. Each agent dispatches the control of offline flows to active controllers based on its mappings.



(a) The result of only maximizing the total programmability of recovered flows.

(b) The result of recovering all offline flows without considering the communication overhead for controlling the flows.

(c) The result of ProgrammabilityGuardian, which recovers all offline flows and reduces the communication overhead for controlling the flows.

Fig. 3: An example to illustrate design considerations for the flow-level remapping solution. C_3 is closer to $s3$ than to $s2$.

We note that existing SDN techniques can help us to implement ProgrammabilityGuardian. We use Fig. 2(b) to explain our logic design based on FlowVisor. FlowVisor [6] is a slicing layer that resides between the data plane of commercial SDN switches and the control plane with multiple controllers. With the virtualization, it can slice the network hardware to different virtual networks and allow multiple controllers to manage the same SDN switch. FlowVisor perfectly matches our design goal. We can deploy ProgrammabilityGuardian agent as a module in the FlowVisor and dynamically dispatch

flow control to controllers using ProgrammabilityGuardian.

B. Design considerations

We use Fig. 3 as an example to show important factors of designing ProgrammabilityGuardian under the controller failure in Fig. 1. In Fig. 3(a), a flow-level mapping solution maps the control of flow $f1$ from switch $s1$ to controller C_1, the control of flow $f2$ from switch $s2$ to controller C_1, and the control of flows $f3$ and $f4$ from switch $s3$ to controller C_3. This solution can maximize the total programmability of recovered flows, but flow $f2$ is not recovered. Fig. 3(b) shows the result of a flow-level mapping solution that aims to recover all flows. In this figure, all four flows are recovered with programmability. However, this solution does not consider the communication overhead for controlling these recovered flows. Particularly, we have two options to recover flow $f3$ either from switch $s2$ or $s3$. Since the distance from controller C_3 to $s2$ is longer than to $s3$, this solution does not work well in terms of the communication overhead. Fig. 3(c) shows the result of ProgrammabilityGuardian, which considers the programmability of offline flows and communication overhead. ProgrammabilityGuardian enables each offline flow to become programmable with a similar programmability and maintains the low communication overhead by individually establishing flow-level mappings for offline flows from their traversing offline switches to their near controllers.

IV. Problem Formulation

In this section, we formulate the OFCM problem to remap offline flows from offline switches to active controllers. For simplicity, in this section, we use a switch instead of an offline switch.

A. System description

Typically, an SD-WAN consists of H controllers at H locations, and each controller controls a domain of switches. Controllers C_{M+1}, ... , C_H fail, and they totally control N switches. The set of active controllers is $\mathcal{C} = \{C_1, ..., C_j, ...C_M\}$, and the set of offline switches controlled by the failed controllers are $\mathcal{S} = \{s_1, ..., s_i, ..., s_N\}$. We need to select flows that traverse switches from $\mathcal{S}$ and map these flows to controllers in $\mathcal{C}$. The set of flows from the set of offline switches $\mathcal{S}$ is $\mathcal{F} = \{f^1, f^2, ..., f^l, ..., f^L\}$. If flow f^l's forwarding path traverses switch s_i, and s_i has at least two paths to f^l's destination, we have $\beta_i^l = 1$, otherwise $\beta_i^l = 0$. We use $x_{ij}^l = 1$ to denote that flow f^l in switch s_i is mapped to controller C_j; otherwise $x_{ij}^l = 0$. A feasible flow-controller mapping requires that a flow that traverses a switch is mapped from this switch to an active controller.

B. Constraints

1) Flow-controller mapping constraint: If flow f^l traverses switch s_i, it can be mapped to at most one active controller. That is:

$$\sum_{j=1}^{M} x_{ij}^l \leq \beta_i^l, \forall i \in [1, N], \forall l \in [1, L], \tag{1}$$

In the above inequality, the equal sign comes when a mapping is established for $\beta_i^l = 1$ or no mapping is needed for $\beta_i^l = 0$. The inequality sign is used when a mapping is not established for $\beta_i^l = 1$. This means flow f^l traverses switch s_i, but no mapping is configured for f^l from s_i to any active controllers due to the limited control resource of the active controllers.

2) Controller resource constraint: If some controllers fail, active controllers should only do their best to control the flows from offline switches without interrupting their normal operations. The control load of a controller equal to the total overhead of controlling its associated flows in its domain. We measure a controller's control resource by the number of flows that the controller can normally control without introducing extra delays (e.g., queueing delay [11]). The control load of a controller should not exceed the controller's available control resource and can be written as follows:

$$\sum_{l=1}^{L} \sum_{i=1}^{N} x_{ij}^l \leq A_j^{rest}, \forall j \in [1, M], \tag{2}$$

where A_j^{rest} denotes the available resource of controller C_j.

3) Flow programmability constraint: Typically, a flow with a long path usually has a higher programmability than a flow with a short path since the long path increases this flow's control probability. The path length implicitly differentiates the priority of the flows and leads to unbalanced path programmability without considering dynamic traffic variations. Because the flow size could change over the time, we should treat each offline equally to recover its programmability and let each offline flow have the similar programmability.

We use p_i^l to denote the number of admissible paths of flow f^l at switch s_i. Since switch s_i can only change the next hop on the path of flow f^l, the number of admissible paths denote the number of paths from switch s_i's next hops to f^l's destination. Flow f^l's path programmability at switch s_i is denoted as pro_i^l and calculated as follows:

$$pro_i^l = \sum_{j=1}^{M} (x_{ij}^l * \beta_i^l * p_i^l)$$

Flow f^l's programmability pro^l can be formulated as follows:

$$pro^l = \sum_{i=1}^{N} pro_i^l = \sum_{i=1}^{N} \sum_{j=1}^{M} (x_{ij}^l * \beta_i^l * p_i^l), \forall l \in [1, L].$$

We use r to denote the least path programability of the offline flows and use $\bar{p}_i^l$ to represent $\beta_i^l * p_i^l$. Thus, we have:

$$\sum_{i=1}^{N} \sum_{j=1}^{M} (\bar{p}_i^l * x_{ij}^l) \geq r, \forall l \in [1, L]. \tag{3}$$

C. Objective functions

Our problem has three objectives. The first objective has the first priority, and the third one has the least importance. The first objective aims to recover all flows and let each flow have the similar programmability. That is

$$obj_1 = r. \tag{4}$$

Guaranteeing the similar programmability of offline flows is not enough since the available control resource of active controllers cannot be fully utilized. For instance, if one offline flow only has a very low programmability, the first objective only enables other flows to have the same low programmability even if their programmablility be higher by increasing the number of mappings. Hence, to fully utilize the active controllers' control resource, we introduce the total programmability of recovered flows as the second objective:

$$obj_2 = \sum_{l=1}^{L}\sum_{j=1}^{M}\sum_{i=1}^{N}(\bar{p}_i^l * x_{ij}^l). \tag{5}$$

The third objective is to minimize the communication overhead between switches and controllers to control recovered flows. We use D_{ij} $(D_{ij} \geq 0)$ to denote the propagation delay between switch s_i and controller C_j. The communication overhead of all recovered flows can be formulated as follows:

$$obj_3 = \sum_{l=1}^{L}\sum_{j=1}^{M}\sum_{i=1}^{N}(D_{ij} * x_{ij}^l). \tag{6}$$

D. Problem formulation

The goal of our problem is to recover all offline flows with the similar path programmability, maximize the total programmability of the offline flows, and minimize the communication overhead for controlling offline flows by smartly mapping the offline flows from offline switches to active controllers. In practice, the first objective has the highest priority because it maintains the essential of SDN. Hence, we formulate the OFCM problem as a three-stage problem. In the first stage, we try to recover each offline flow to have the similar path programmability without considering the second and third objectives. Thus, the first stage problem can be formulated as follows:

$$\begin{aligned} R^* = \max_{r,x} \quad & r \\ \text{s.t.} \quad & (1)(2)(3), \\ & r \geq 2, x_{ij}^l \in \{0,1\}, \\ & \forall i \in [1,N], \forall j \in [1,M], \forall l \in [1,L], \end{aligned} \tag{P1}$$

where $\{\beta_i^l\}$ and $\{A_j^{rest}\}$ are constants, and $\{x_{ij}^l\}$ and r are design variables. With the path programmablilty of flows obtained from problem (P1), we further solve the following problem to maximize the total programmability:

$$\begin{aligned} P^* = \max_{x} \quad & \sum_{l=1}^{L}\sum_{j=1}^{M}\sum_{i=1}^{N}(\bar{p}_i^l * x_{ij}^l) \\ \text{s.t.} \quad & R^* = r, \ (1)(2)(3), \\ & x_{ij}^l \in \{0,1\}, \\ & \forall i \in [1,N], \forall j \in [1,M], \forall l \in [1,L]. \end{aligned} \tag{P2}$$

Based on the results of problems (P1) and (P2), we finally minimize the communication overhead by solving the following problem:

$$\begin{aligned} \min_{x} \quad & \sum_{l=1}^{L}\sum_{j=1}^{M}\sum_{i=1}^{N}(D_{ij} * x_{ij}^l) \\ \text{s.t.} \quad & R^* = r, \ P^* = \sum_{l=1}^{L}\sum_{j=1}^{M}\sum_{i=1}^{N}(\bar{p}_i^l * x_{ij}^l), \ (1)(2)(3), \\ & x_{ij}^l \in \{0,1\}, \forall i \in [1,N], \forall j \in [1,M], \forall l \in [1,L]. \end{aligned} \tag{P3}$$

The above three-stage problem formulation needs to solve three problems. An alternative formulation is to combine the three objectives into one objective. That is

$$\begin{aligned} obj &= r + \sum_{l=1}^{L}\sum_{j=1}^{M}\sum_{i=1}^{N}(\bar{p}_i^l * x_{ij}^l) - \lambda * \sum_{l=1}^{L}\sum_{j=1}^{M}\sum_{i=1}^{N}(D_{ij} * x_{ij}^l) \\ &= r + \sum_{l=1}^{L}\sum_{i=1}^{N}\sum_{j=1}^{M}(\bar{p}_i^l - \lambda * D_{ij}) x_{ij}^l \end{aligned}$$

where $\lambda \geq 0$ is a constant number that gives different weights of the two objective terms. Existing work shows selecting the right λ can guarantee the optimal solution [12].

Therefore, we reformulate our problem as the final problem:

$$\begin{aligned} \max_{r,x} \quad & r + \sum_{l=1}^{L}\sum_{i=1}^{N}\sum_{j=1}^{M}(\omega_{ij} * x_{ij}^l) \\ \text{s.t.} \quad & (1)(2)(3), \\ & r \geq 2, x_{ij}^l \in \{0,1\}, \\ & \forall i \in [1,N], \forall j \in [1,M], \forall l \in [1,L], \end{aligned} \tag{P}$$

where $\omega_{ij} = \bar{p}_i^l - \lambda * D_{ij}$. In the above problem, the objective function is linear, $\{x_{ij}^l\}$ are binary variables, and r is an integer variable. Thus, this problem is an Integer Programming (IP). We have analyze the complexity of this problem and found it is NP-hard. Due to the limited space, we do not show the complexity analysis here.

V. SOLUTION

The typical solution of the above OFCM problem is to get its optimal result with IP optimization solvers. However, as the network size increases, the solution space could increase significantly, and finding a feasible solution may cost a long time or perhaps is impossible. We achieve the trade-off between the performance and time complexity by solving the problem with the proposed heuristic algorithm.

The key idea of the heuristic algorithm is to test and increase each flow's programability equally by following the mapping establishing probability. Details are summarized in Algorithm 1, and the notations used in the algorithm are listed in Table I. In line 1, we initialize $\mathcal{X}$ to be empty, each flow's programmability to 0, and tested counter to 0 at the beginning of the algorithm. In line 2, we generate TEST_MAP_NUM and vectors $\bar{X}^l = \{x_k^l, k \in [1, N*M]\}$ for flow f^l $(l \in [1,L])$. We first relax binary variables in problem (P) to continuous variables and obtain the Linear Programming relaxation solution $\bar{X}_l^*$ of flow f^l. We then sort the values in $\bar{X}_l^*$ in the descending order to get vectors $\bar{X}^l$.

Algorithm 1 Heuristic algorithm

Input: N, M, A, $\mathcal{G}$;
Output: $\mathcal{X}$, $\mathcal{R}$;

1: $\mathcal{X} = \emptyset$, $\mathcal{R} = \{r_l = 0, l \in [1, L]\}$, test_count = 0;
2: Generate TEST_MAP_NUM and testing mapping sets $\bar{X}^l = \{x_k^l, k \in [1, N * M]\}$, $l \in [1, L]$ by solving the Linear Programming relaxation of problem (P) and sorting the results of flow f^l ($l \in [1, L]$) in the descending order;
3: **while** test_count $\leq$ TEST_MAP_NUM **do**
4: Generate vectors L^* by getting the flow index from R, which is sorted in the ascending order;
5: // test flows based on the ascending order of their programmability
6: **for** $l_0 \in L^*$ **do**
7: // all mappings are tested for flow f^{l_0}.
8: **if** $\bar{X}^{l_0} == \emptyset$ **then**
9: $L^* \leftarrow L^*/l_0$, continue;
10: **end if**
11: IsMapped = FALSE;
12: // the rest of mappings for flow f^{l_0} will be tested.
13: **for** $\bar{x}^{l_0} \in \bar{X}^{l_0}$ **do**
14: $\bar{X}^{l_0} \leftarrow \bar{X}^{l_0}/\bar{x}^{l_0}$, test_count++;
15: find switch and controller IDs i_0 and j_0 of $\bar{x}^{l_0}$;
16: // a related mapping is already established.
17: **if** $(i_0, l_0) \in Mapped$ **then**
18: continue;
19: **end if**
20: **if** $\mathcal{X} \cup (i_0, j_0, l_0)$ satisfy the constraints in Eqs. (1) and (2) **then**
21: $A_{all}^{rest} = A_{all}^{rest} - 1$, $A_{j_0}^{rest} = A_{j_0}^{rest} - 1$;
22: $r_{l_0} = r_{l_0} + p_{i_0}^{l_0}$, $\mathcal{X} \leftarrow \mathcal{X} \cup (i_0, j_0, l_0)$;
23: $Mapped \leftarrow Mapped \cup (i_0, l_0)$;
24: Mapped_Flag = TRUE;
25: **if** $A_{all}^{rest} == 0$ **then**
26: break;
27: **end if**
28: **end if**
29: // one mapping is established for flow f^{l_0}.
30: **if** IsMapped == TRUE **then**
31: go to line 6;
32: **end if**
33: **end for**
34: **end for**
35: **end while**
36: return $\mathcal{X}, \mathcal{R}$;

The sorting operation helps us to test the mappings based on their probabilities.

In lines 3-35, we test all TEST_MAP_NUM possible mappings by rounding the decimal values in $\bar{X}^l$, $l \in [1, L]$ for flows to configure the right mappings. Recall our first objective is to realize the similar programmability for flows. Line 4 runs to make sure the flows with low programmabililty are tested first. From line 6 to line 33, we test flow f^{l_0}. Lines 8 to 10 accelerate the testing by removing the further testing for f^{l_0} if its mappings are fully tested. Following the first objective, we should equally test each flow to establish mappings. Hence, during the mapping test for f^{l_0}, we must find one feasible mapping for f^{l_0}. Thus, in line 11, we use a Boolean value IsMapped to denote whether a feasible mapping for f^{l_0} is found. For testing mapping $\bar{x}^{l_0}$ from lines 13 to 28, we first remove this mapping from f^{l_0}'s mapping testing set and increases the tested counter by one; we then find switch and controller IDs i_0 and j_0 of $\bar{x}^{l_0}$. Lines 17 to 19 ensure the mapping of f^{l_0} from switch s_{i_0} to controllers will not be redundantly tested once one feasible mapping is configured. If this mapping satisfies the constraints in Eqs. (1) and (2), it is a feasible mapping, and we will change the total processing ability of controllers, change the processing ability of controller C_{j_0}, which will control this flow, increase f^{l_0}'s path programmability, confirm this mapping for f^{l_0} and $Mapped$, and update IsMapped to verify this testing's positive result. If the total processing ability of controllers is used up, we cannot set up new mappings, and the testing ends, as shown in lines 25 to 27. After this mapping test from lines 13 to 28, if this mapping is feasible, the algorithm will test next flow; otherwise, the next mapping for f^{l_0} will be evaluated until a feasible mapping for f^{l_0} is found or all mappings of f^{l_0} are tested. In line 26, the algorithm returns the result and stops.

TABLE I: Notations

Notation	Meaning
$\mathcal{A}$	the set of the available processing capacity of controllers, $\mathcal{A} = \{A_j^{rest} \mid j \in [1, M]\}$
A_{all}^{rest}	the total available control resource of all controllers, $A_{all}^{rest} = \sum_{j=1}^{M} A^{rest}$
$\mathcal{R}$	the programability of offline flows, $\mathcal{R} = \{r_l \mid l \in [1, L]\}$
$\mathcal{X}$	the set of the mapping relationship between offline flows from offline switches with the SDN mode and active controllers, $\mathcal{X} = \{(i, j, l) \in [1, N] \times [1, M] \times [1, L] \mid x_{ij}^l = 1\}$
$\bar{X}^l$	the set of testing mappings by solving the LP relaxation of problem (P) and sorting the results of flow f^l ($l \in [1, L]$) in the descending order, $\bar{X}^l = \{x_k^l, k \in [1, N * M]\}$
$Mapped$	the set of established mappings of flow f^l through switch s_i, $Mapped = \{(i, l) \in [1, N] \times [1, L] \mid \sum_{j=1}^{M} x_{ij}^l = 1\}$
TEST_MAP_NUM	the total number of possible mappings of L flows for testing

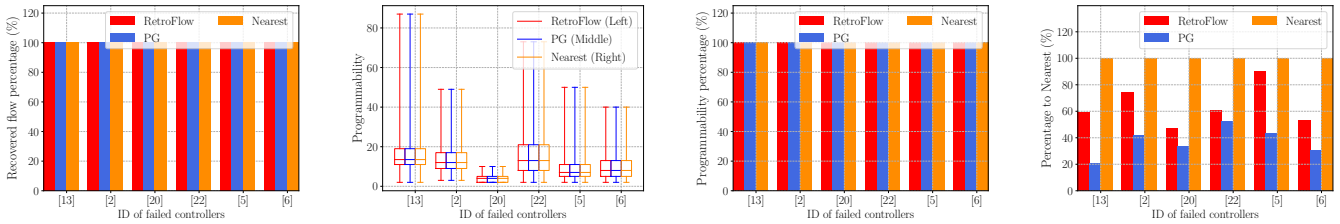
VI. Simulation

A. Simulation setup

We use a typical backbone topology named ATT from Topology Zoo [13] to evaluate the performance of ProgrammabilityGuardian. The ATT topology is a national topology of US and consists of 25 nodes and 112 links. In ATT topology, each node has a unique ID and its latitude and longitude. We employ Haversine formula [14] to calculate the distance between two nodes and use the distance divided by the propagation speed (i.e., 2×10^8 m/s) [15] to describe the propagation delay between the two nodes. In our simulation, each node is an SDN switch connected to a ProgrammabilityGuardian agent, and any two nodes have a traffic flow forwarded on the shortest path. The control plane consists of six controllers, and the processing ability of each controller is 500. Table II shows the default relationship of controllers, switches, and the number of flows in the switches.

TABLE II: Default relationship between controllers, switches, and the number of flows in the switches under ATT topology.

Controller ID	2				5				6				13					20		22					
Switch ID	2	3	9	16	4	5	8	14	0	1	6	7	10	11	12	13	15	19	20	17	18	21	22	23	24
Number of flows	143	71	107	55	49	143	53	61	81	49	89	97	63	59	71	213	67	49	63	125	49	81	111	49	57



(a) Percentage of recovered flows. The higher, the better.

(b) Path programmability of offline flows. The higher, the better.

(c) Total path programmability of offline flows. The higher, the better.

(d) Communication overhead. The lower, the better.

Fig. 4: Results of one controller failures. PG is short for ProgrammabilityGuardian.

B. Comparison algorithms

1) Nearest: during controller failures, offline flows are recovered by mapping offline switches to their nearest controllers. It exhibits the optimal results for all performance metrics except the communication overhead.
2) RetroFlow [5]: this solution recovers offline flows by configuring a set of offline switches working under the legacy routing mode and transferring the control of offline switches with the SDN routing mode to active controllers to minimize the communication overhead from these offline switches to the active controllers [5].
3) ProgrammabilityGuardian (PG): this solution recovers offline flows using Algorithm 1 to configure flow-level flow-controller mappings.

C. Simulation results

We compare the performance of PG with other algorithms under two scenarios: one controller failures and two controller failures. Following our three objectives, we use the following metrics: (1) the percentage of recovered offline flows, (2) the path programmability of recovered flows, and (3) the communication overhead. We use Nearest as the baseline algorithm and normalize the result of each solution to that of Nearest.

1) One controller failure: One controller failure is a common scenario, and active controllers usually have enough control resource to recover all offline switches. Fig. 4 shows the results of three comparison algorithms under one controller failures. In Figs. 4 (a) and (b), all three algorithms recover 100% offline flows and realize the same programmabilty of recovered flows. Thus, all the three solutions have the same total programmability shown in Fig. 4(c). However, in Fig. 4(d), PG requires the least communication overhead. Nearest overloads controllers and introduces queueing delay to increase the overhead. RetroFlow recovers flows by selectively only mapping some switches with the SDN routing model to active controllers and thus reduces the overhead. In contrast, PG provides opportunities for recovering offline flows by exploring the controllers with the small amount of control resource but near offline switches to establish the flow-controller mappings. Thus, the controllers close to offline switches are prone to be fully loaded. PG outperforms RetroFlow and reduces the communication overhead up to 75%, compared with Nearest. This result also shows minimizing the objective of the communication overhead in Eq. (6) is very important when the first two objectives are satisfied by all algorithms.

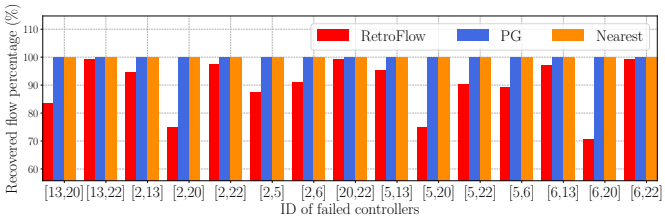
2) Two controller failure: Two controller failure is a serious controller failure, and PG shows higher programmability resiliency than others. We use two scenarios for evaluation.

Scenario 1. In this scenario, all offline flows are forwarded on their shortest paths. Given six controllers, this scenario has 15 cases, and the ideal result is to recover 100% offline flows to become programmable for all the cases. Fig. 5 shows the results of three algorithms when two controllers fail.

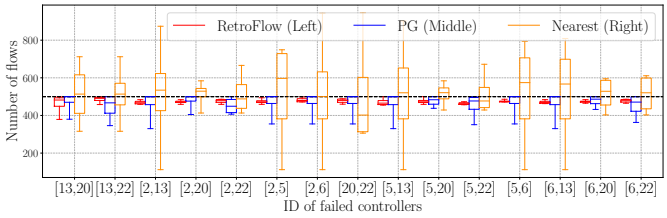
(1) Recovered flow percentage: Fig. 5(a) shows the percentage of recovered flows. For all cases, RetroFlow only recovers flows in the range of 71 % to 99 % because the switch-level mappings cannot fully occupy active controllers to recover flows under the constraint of not overloading active controllers. Nearest recovers all flows at the cost of overloading active controllers shown in Fig. 5(b). PG follows its first objective to recover all flows using flow-controller mappings and does not interrupt active controllers' normal operations.

(2) Path programmability: Fig. 5(c) shows the path programmability of recovered flows. For all cases, RetroFlow's least path programmability is 0, which denotes some flows are not recovered to be programmable, and its median is always lower than Nearest and PG. Nearest denotes the optimal performance for this metric since it maps all offline switches to active controllers to fully recover programmability. PG exhibits the same performance to Nearest.

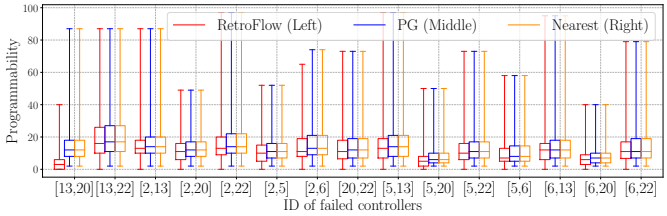
In this figure, some flows have high programmability, but the least path programmability is limited to 2. The high programmability comes from the flows with long paths, and the least programmability is constrained by specific flows. In our simulation, we generate a flow for any two nodes. In each of the 15 cases, we can always find two types of offline flows: (1) a flow's source and destination nodes are



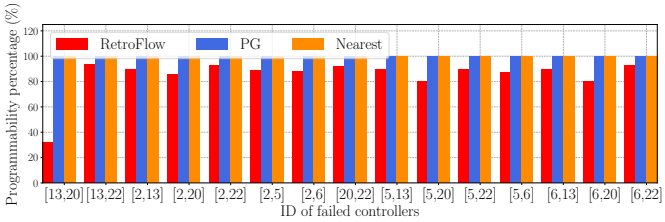
(a) Percentage of recovered flows. The higher, the better.



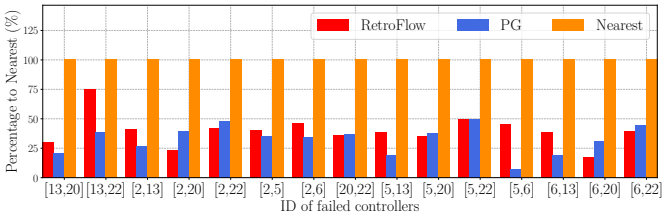
(b) Processing load of active controllers. The black dash line indicates the controller's processing ability.



(c) Path programmability of recovered flows. The higher, the better.



(d) Total path programmability of recovered flows. The higher, the better.



(e) Communication overhead. The lower, the better.

Fig. 5: Results of two controller failure's scenario 1. PG is short for ProgrammabilityGuardian.

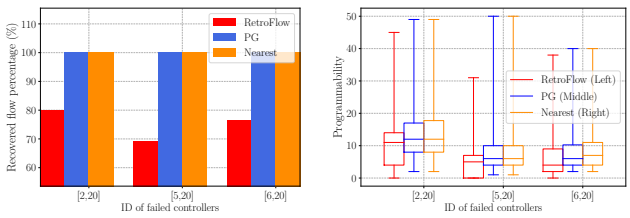
directly connected; (2) a flow's shortest path only includes one node except their source and destination nodes, and this only one node just has two paths to the flow's destination node. Either of the two flows' programmability is 2, and PG's performance is limited by the flow. This figure also supports that our second objective of maximizing the total programmability in Eq. (5) is critical because it can improve the utilization of control resource in active controllers for flow recovery. Without this objective, all flows will be recovered with the least programmability.

Fig. 5(d) shows the total path programmability. Both PG and Nearest recover all flows with the same programmability, and thus they have the same performance for this metric. For the 15 cases, RetroFlow's performance in the range of 32% to 95% of the optimal result since it performs worse in both of the recovered flow percentage and the path programmability. For failure case (13, 20), PG's performance is 68% higher than RetroFlow's.

(3) Communication overhead: Fig. 5(e) shows the communication overhead of the three algorithms. PG exhibits outstanding performance for all cases. Compared with Nearest, it realizes the same programmability in Figs. 5(a), 5(c), and 5(d) with the much lower overhead without overloading active controllers. For failure case (5, 6), PG reduces the overhead 83%. PG also performs better than RetroFlow. PG's fine-grained mappings can fully utilize the small amount of available control resource from offline switches' near controllers to recover offline flows and at the same time reduce the communication overhead. Note that PG's overhead is a little higher than RetroFlow for failure cases (2, 20), (2, 22), (5, 20), (6, 20), and (6, 22). This is because PG recovers more flows, as shown in Fig. 5(a).

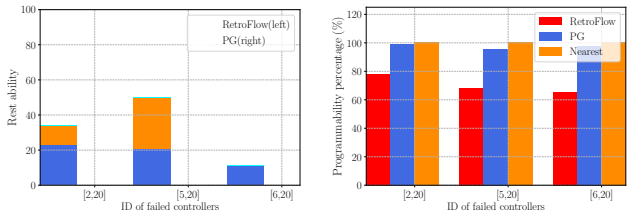
Scenario 2. In the SD-WAN, some flows could be forwarded on the longer paths to optimize network performance (e.g., load balancing). If controllers fail under this condition, active controllers face a much serious situation for flow recovery since recovering offline flows' programmability with long paths requires more control resources from controllers. Scenario 2 is a case of the above condition. In this scenario, 112 of 625 offline flows' forwarding paths are one hop longer than their shortest paths, and the rest offline flows are forwarded on their shortest paths. Under this pressure test, PG also shows high programmability resiliency.

Fig. 6 shows the results of three representative cases of 15 cases. In the three cases, the control resource of active controllers is not enough to configure all mappings for each flow from the offline switches on its paths to the controllers. For failure cases (2,20), (5,20), and (6,20), recovering all mappings of flows requires 370, 287, and 296 control resource while the available control resource is only 349, 266, and 275, respectively. Under these serious failures, PG selectively configures some mappings but still recover all flows with the similar programmability, which are shown in Figs. 6(a) and (b). Compared with Fig. 5(a), RetroFlow's performance increases a little since some flows' forwarding paths increases one hop, which provides higher chances for these flows to be recovered. However, the improvement is limited by the coarse-grained switch-controller mapping. Fig. 6(c) shows the left control resource of active controllers. In this figure, RetroFlow has two controllers (denoted by two different colors) with left resource while PG uses up all available resource to recover flows. Although PG cannot configure all mappings, its effectively selects the mappings that not only guarantee the least programmability for all flows but also maximize the total programmability. Fig. 6(d) shows the total programmability. PG's performance is better than RetroFlow's and very close to the result of Nearest, which establishes all mappings. Fig. 6(e) shows the similar results in Fig. 5(e) since PG recovers more flows, as shown in Fig. 6(a).



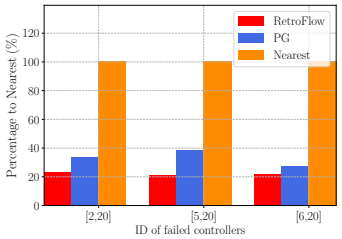
(a) Percentage of recovered flows. The higher, the better.

(b) Path programmability of recovered flows. The higher, the better.



(c) The left control resource of active controllers.

(d) Total path programmability of recovered flows. The higher, the better.



(e) Communication overhead. The lower, the better.

Fig. 6: Results of two controller failure's scenario 2. PG is short for ProgrammabilityGuardian. In (c), the results of PG are 0.

VII. Related Works

Existing works [2][3][4][16][17] try to find the trade-off between the performance and cost during the controller failure under several constraints (e.g., load balancing and QoS). Tanha et al. [18] consider both the switch-controller/inter-controller latency requirements and the capacity of the controllers to select the resilient placement of controllers. Yang et al. [19] propose to protect all single link failures in a given network by updating a smallest subset of IP routers to SDN switches. He et al. [20] formulate a master and slave controller assignment model to prevent multiple controller failures under the constraints of the propagation latency between switches and controllers. RetroFlow [5] leverages the features of hybrid routing in SDN switches to maintain the flow programmability and low communication overhead without overloading active controllers.

VIII. Conclusion

In this paper, we propose ProgrammabilityGuardian to improve path programmablility for offline flows during multiple controller failures. Thanks to the fine-grained flow-level mappings provided by FlowVisor, ProgrammabilityGuardian smartly configures the mappings of offline flows from their traversed offline switches to active controllers. ProgrammabilityGuardian can recover all offline flows with the similar path programmability, improve the total programmability of the offline flows, and reduce the communication overhead for controlling these offline flows. Our work sheds light for other studies on creatively leveraging existing SDN techniques to better solve existing problems.

Acknowledgment

This paper is supported by National Key Research and Development Program of China under Grant 2018YFB1003700 and Beijing Institute of Technology Research Fund Program for Young Scholars.

References

[1] Z. Guo, M. Su, Y. Xu, Z. Duan, L. Wang, S. Hui, and H. J. Chao, "Improving the performance of load balancing in software-defined networks through load variance-based synchronization," *Computer Networks*, vol. 68, pp. 95–109, 2014.

[2] M. Tanha, D. Sajjadi, and J. Pan, "Enduring node failures through resilient controller placement for software defined networks," in *IEEE GLOBECOM'16*.

[3] B. P. R. Killi and S. V. Rao, "Capacitated next controller placement in software defined networks," *IEEE TNSM*, vol. 14, no. 3, pp. 514–527, 2017.

[4] T. Hu, P. Yi, Z. Guo, J. Lan, and Y. Hu, "Dynamic slave controller assignment for enhancing control plane robustness in software-defined networks," *Future Generation Computer Systems*, vol. 95, pp. 681–693, 2019.

[5] Z. Guo, W. Feng, S. Liu, W. Jiang, Y. Xu, and Z.-L. Zhang, "Retroflow: Maintaining control resiliency and flow programmability for software-defined wans," in *IEEE/ACM IWQoS'19*.

[6] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," *OpenFlow Switch Consortium, Tech. Rep*, vol. 1, p. 132, 2009.

[7] "Openflow switch specification 1.3," https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf.

[8] "Brocade mlx-8 pe," https://www.dataswitchworks.com/datasheets/MLX_Series_DS.pdf.

[9] G. Yao, J. Bi, and L. Guo, "On the cascading failures of multi-controllers in software defined networks," in *IEEE ICNP'13*.

[10] K. Poularakis, Q. Qin, L. Ma, S. Kompella, K. K. Leung, and L. Tassiulas, "Learning the optimal synchronization rates in distributed sdn control architectures," in *IEEE INFOCOM19*.

[11] J. Xie, D. Guo, X. Li, Y. Shen, and X. Jiang, "Cutting long-tail latency of routing response in software defined networks," *IEEE JSAC*, vol. 36, no. 3, pp. 384–396, 2018.

[12] Z. Guo, W. Chen, Y. Liu, Y. Xu, and Z. Zhang, "Joint switch upgrade and controller deployment in hybrid software-defined networks," *IEEE JSAC*, vol. 37, no. 5, pp. 1012–1028, May 2019.

[13] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE JSAC*, vol. 29, no. 9, pp. 1765 – 1775, 2011.

[14] C. C. Robusto, "The cosine-haversine formula," *The American Mathematical Monthly*, vol. 64, no. 1, pp. 38–40, 1957.

[15] "Speed, rates, times, delays: Data link parameters for cse 461," https://courses.cs.washington.edu/courses/cse461/99wi/issues/definitions.html.

[16] A. Alshamrani, S. Guha, S. Pisharody, A. Chowdhary, and D. Huang, "Fault tolerant controller placement in distributed sdn environments," in *IEEE ICC'18*.

[17] T. Hu, Z. Guo, J. Zhang, and J. Lan, "Adaptive slave controller assignment for fault-tolerant control plane in software-defined networking," in *IEEE ICC'18*.

[18] M. Tanha, D. Sajjadi, R. Ruby, and J. Pan, "Capacity-aware and delay-guaranteed resilient controller placement for software-defined wans," *IEEE TNSM*, vol. 15, no. 3, pp. 991–1005, Sep. 2018.

[19] Z. Yang and K. L. Yeung, "Sdn candidate selection in hybrid ip/sdn networks for single link failure protection," *IEEE/ACM ToN*, 2020.

[20] F. He, T. Sato, and E. Oki, "Master and slave controller assignment model against multiple failures in software defined network," in *ICC'19*.