ProgrammabilityMedic: Predictable Path Programmability Recovery under Multiple Controller Failures in SD-WANs

Songshi Dou, Zehua Guo*, Yuanqing Xia
Beijing Institute of Technology

***Abstract*—Software-Defined Networking (SDN) promises good network performance in Wide Area Networks (WANs) with the logically centralized control using physically distributed controllers. In Software-Defined WANs (SD-WANs), maintaining path programmability, which enables flexible path change on flows, is crucial for maintaining network performance under traffic variation. However, when controllers fail, existing solutions are essentially coarse-grained switch-controller mapping solutions and only recover the path programmability of a limited number of offline flows, which traverse offline switches controlled by failed controllers. In this paper, we propose ProgrammabilityMedic (PM) to provide predictable path programmability recovery under controller failures in SD-WANs. The key idea of PM is to approximately realize flow-controller mappings using hybrid SDN/legacy routing supported by high-end commercial SDN switches. Using the hybrid routing, we can recover programmability by fine-grainedly selecting a routing mode for each offline flow at each offline switch to fit the given control resource from active controllers. Thus, PM can effectively map offline switches to active controllers to improve recovery efficiency. Simulation results show that PM outperforms existing switch-level solutions by maintaining balanced programmability and increasing the total programmability of recovered offline flows up to 315% under two controller failures and 340% under three controller failures.**

I. Introduction

Wide Area Network (WAN) is a crucial infrastructure in real world. It connects and transfers traffic among different types of networks (e.g., data centers, cellular networks, and metropolitan area networks). To improve performance, emerging Software-Defined Networking (SDN) has been deployed in WANs, known as the SD-WANs. Microsoft SWAN [1] and Google B4 [2] show that flexible flow control enabled by SDN can significantly improve utilization of WANs. Basically, an SD-WAN has a large scale and is usually composed of multiple domains, each of which consists of one SDN controller to control SDN switches placed in its domain. The flexible flow control of SDN comes from *path programmability*. Once a flow traverses an SDN switch, it becomes a *programmable flow* and is able to change its forwarding path.

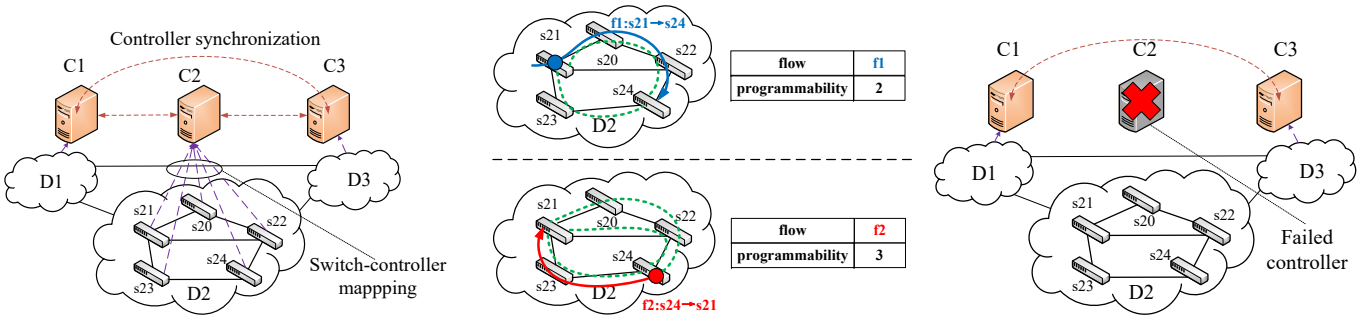
Path programmability is empowered by the SDN controller through deploying flow entries to adjust flows' forwarding paths. A controller is a network control software installed in a physical server or a virtual machine. Because of unpredictable issues (e.g., hardware/software bugs and power failure), the controller may fail, and its controlled switches will be unmanageable and become *offline*. In this case, flows, which traverse offline switches, lose their path programmability to change their paths and become *offline flows*. As a result, the network suffers from path programmability degradation and potentially experiences performance fluctuation under traffic variation. Therefore, recovering path programmability for offline flows under controller failure is the key to guarantee network performance. Maintaining path programmability under multiple controller failures is important since multiple controller failures can lead to more significant performance variation than single controller failures due to the severe decrease of network programmability. Several controllers may fail simultaneously or fail successively. Some existing works [3]–[7] also study the problem of multiple controller failures.

Programmability recovery is accomplished by remapping the control of offline switches to active controllers to let offline flows in offline switches become programmable again. The main challenge of this mapping problem is how to use the limited control resource from active controllers to handle the control cost for enabling the programmability of offline flows under different controller failures. Without appropriate remapping, active controllers could be overloaded to recover offline flows, and network performance may degrade. In the worse case, network may face significant performance degradation due to the cascading controller failure [8].

Existing solutions can be categorized into two classes: switch-level solutions and flow-level solutions. Switch-level solutions consider all flows in a switch as an unit and remap offline switches to active controllers based on the switches' and controllers' current status information [3], [5], [6]. The coarse-grained switch-controller mapping only recovers a small number of offline flows from offline switches because the fixed control cost of switches does not always match controllers' control resource properly. A recent flow-level solution proposes to fine-grainedly remap offline flows to active controllers by introducing a middle layer between controllers and switches [9]. This solution performs well in terms of programmability recovery, but the middle layer design brings new issues (e.g., increasing the processing delay [10] and bringing new unreliability).

In this paper, we propose ProgrammabilityMedic (PM), a novel switch-controller mapping solution to improve the programmability recovery of offline flows. The key of PM is to take advantage of the hybrid SDN/legacy routing mode supported by high-end commercial SDN switches to approximately realize fine-grained flow-controller mappings. The hybrid routing employs a high-priority flow table using OpenFlow and a low-priority legacy routing table based on OSPF. With the proper configuration, we can jointly use the two tables in a sequential fashion to route flows in different modes and thus recover the path programmability in a fine-grained per-flow fashion based on given control resource of

* Corresponding Author: Zehua Guo



(a) An SD-WAN. (b) Path programmability of two flows. (c) Controller failure in SD-WAN.

Fig. 1: Example of an SD-WAN and the path programmability. A circle denotes that a flow is programmable at this switch.

active controllers. Consequently, as offline switches efficiently mapped to active controllers, the programmability of offline flows is effectively recovered.

The contributions of this paper are summarized as follows:

- We propose to recover path programmability of offline flows by approximately realizing flow-controller mappings using hybrid SDN/legacy routing supported by high-end commercial SDN switches.
- We formulate the Flow Mode Section and Switch Mapping (FMSSM) problem, which uses hybrid routing and aims to recover offline flows with balanced programmability by deciding the routing mode of flows at offline switches and the mappings between offline switches and active controllers.
- The proposed FMSSM problem is a mix integer programming with high computation complexity. To efficiently solve the problem, we reformulate the problem with linear techniques and solve the problem with the proposed efficient heuristic algorithm named PM.
- We evaluate the performance of PM under different controller failure scenarios. The results show that PM outperforms existing switch-level solutions by maintaining balanced programmability and increasing the total programmability up to 315% under two controller failures and 340% under three controller failures.

The rest of the paper is organized as follows. In Section II, we introduce the background and motivation of this paper. Section III presents our design considerations. Section IV mathematically formulates our programmability recovery problem as the FMSSM problem. Section V proposes PM to efficiently solve the problem. We evaluate and analyze the performance of PM in Section VI. Section VII introduces related work, and Section VIII concludes this paper.

II. Background and Motivation

A. Network control and path programmability in SD-WANs

We use Fig. 1 to explain the SD-WAN. In Fig. 1(a), the SD-WAN consists of three domains, and the SDN control plane includes three controllers, each of which controls one domain and synchronizes with others to maintain a logically centralized network view. For simplicity, we only concentrate on domain D_2, which has five SDN switches $s_{20} - s_{24}$ and is controlled by controller C_2.

SDN introduces path programmability to improve network performance by dynamically rerouting flows under network variation. The path programmability of one flow denotes the ability of switch to change this flow's path. Fig. 1(b) illustrates the path programmability of two flows. For flow f^1:$s_{21} \rightarrow s_{24}$, it has two paths (i.e., $s_{21} \rightarrow s_{20} \rightarrow s_{22} \rightarrow s_{24}$ and $s_{21} \rightarrow s_{23} \rightarrow s_{24}$) at s_{21}, and thus its programmability is two at s_{21}. Similarly, the programmability of flow f^2: $s_{24} \rightarrow s_{21}$ is three at s_{24} since it has three paths (i.e., $s_{24} \rightarrow s_{23} \rightarrow s_{21}$, $s_{24} \rightarrow s_{22} \rightarrow s_{21}$, and $s_{24} \rightarrow s_{22} \rightarrow s_{20} \rightarrow s_{21}$) at s_{24}.

However, SDN controllers may fail unpredictably. Under controller failures, SDN switches controlled by failed controllers become *offline*, and flows, which traverse offline switches, become offline and lose their programmability to change their paths to cope with network variation. As a result, it is difficult to maintain resilient network control. We use Fig. 1(c) to illustrate this issue. In Fig. 1(c), controller C_2 fails, and its controlled five switches in D_2 become offline. Thus, f^1 and f^2 become offline flows.

In SD-WANs, the key of maintaining resilient network control is to enable path programmability. To recover the path programmability, we must hand over the control of switches $s_{20} - s_{24}$ to active controllers C_1 and C_3. A feasible solution to recover path programmability of offline flows is to remap offline switches to active controllers. In this way, all the offline flows that traverse remapped switches are re-controlled by active controllers and become programmable again.

B. Existing solutions and their limitations

Existing solutions for recovering path programmability can be categorized into the following two classes:

1) Switch-level solutions: These solutions employ the default path programmability recovery solution originated from OpenFlow [11] to remap offline switches to active controllers. By remapping one offline switch to an active controller, all flows that traverse this remapped switch are controlled by this controller and become programmable again [3], [5], [12]. Switch-level solutions do not work well under multiple controller failures due to its coarse-grained mappings. To mitigate the impact of coarse-grained mapping, a recent work proposes

TABLE I: Comparison of PM and existing solutions

Solution	Mapping granularity	Extra device	Routing mode	Routing mode granularity	Recovery performance
switch-level solution	switch-controller	no	SDN, hybrid	per-switch	mediocre
flow-level solution	flow-controller	yes	SDN	per-flow	optimal
PM	switch-controller	no	hybrid	per-flow	near optimal

a hybrid switch mode solution [6], which reduces the control cost of offline switches by configuring them with different routing modes (i.e., SDN and legacy) and only remapping switches with the SDN mode to controllers. This solution improves recovery efficiency but still suffers from coarse-grained mapping under serious controller failures.

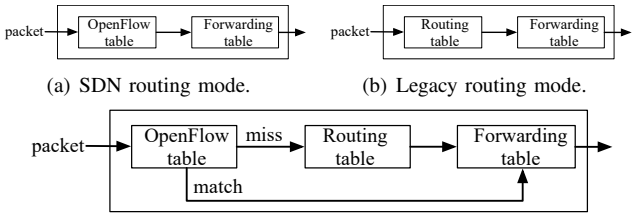
2) Flow-level solutions: ProgrammabilityGuardian (PG) [9] is a flow-level solution. Instead of remapping all flows in an offline switch to an active controller, PG introduces a middle layer between controllers and switches using existing SDN slicing techniques (i.e., FlowVisor [10]) and uses the layer to fine-grainedly remap each offline flow to one active controller. This flow-level solution performs much better than switch-level solutions but relies on an extra middle layer, which is usually realized by physical servers or virtual machines. The layer needs extra time to process incoming requests [10] and also suffers from unpredictable issues (e.g., hardware/software bugs and power failure). Thus, this layer not only increases the processing delay but also brings new unreliability to the system.

III. Design considerations of PM

In this section, we introduce the opportunity and some key design considerations of PM.

A. Opportunity

Based on the above analysis, we can see that one good recovery solution is to realize good recovery performance similar to flow-level solutions without extra devices. High-end commercial SDN switches provide new opportunity for designing this solution. Brocade MLX-8 PE [13] supports different routing modes: SDN model, legacy mode, and hybrid SDN/legacy routing mode, specifically OpenFlow/OSPF routing mode. Fig. 2 shows the data packet process of the three modes in the SDN switch. In Figs. 2(a) and 2(b), either OpenFlow or OSPF routing mode is used. In Fig. 2(c), OpenFlow and OSPF run at the same time, and the priority of the flow table used for OpenFlow is higher than that of the legacy routing table used by OSPF. The flow table can be installed with a default low-priority entry, which sends each unmatched packet to the legacy routing table. When the switch works in this hybrid mode, it first checks the flow table for each incoming packet. If a matched entry is found, the packet is processed according to the operation in the entry. Otherwise, the legacy routing table is then checked to route the packet using destination-based entries. The hybrid routing mode is tested in production networks (e.g., CHN-IX [14]),



(a) SDN routing mode. (b) Legacy routing mode.

(c) Hybrid SDN/legacy routing mode.

Fig. 2: Routing modes in high-end commercial SDN switches.

and existing works also use hybrid routing mode to realize the improvement of network performance [15].

Using this hybrid mode, we can change the control cost of offline switches by selectively deciding the routing mode for each offline flow at each offline switch and thus realize fine-grainedly path programmability recovery for each offline flow and improve the recovery efficiency. We use Table I to present the difference between existing solutions and our solution.

B. Design considerations

To efficiently recover the path programmability, our design should consider the following three objectives:

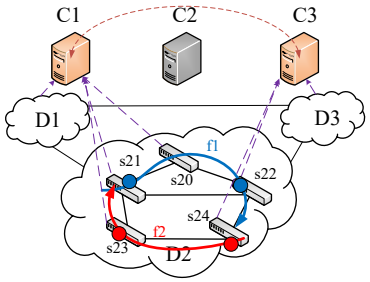
Recovering offline flows as many as possible: recovering path programmability of offline flows affects network performance and is the first priority for our problem. However, if some offline flows are not recovered, and these flows' traffic load vary significantly. These flows cannot be adaptively rerouted, and network performance could significantly fluctuate. Thus, we should recover offline flows as many as possible to be programmable.

Balancing path programmability: this consideration is to enable balanced path programmability of recovered flows. Generally speaking, a flow with a long path usually has higher programmability than a flow with a short path because the long path increases the control probability of the flow. The path length implicitly distinguishes the priority of the flow and leads to an imbalanced path programmability. However, the size of flows may change over time, and unbalanced path programmablilty becomes another uncertainty for network performance. Thus, we should treat each offline flow equally by recovering each offline flow with the similar programmability.

Fully utilizing controllers' control resource: this consideration compensates the above two considerations. Due to various conditions (e.g., topology difference), in some cases, an offline flow can only be recovered with the low programmability. Following the first and second considerations, other offline flows are also recovered with the same low programmability even if there is large room to improve the programmability of these flows by establishing more mappings. Hence, we should fully utilizing controllers' control resource to improve overall programmability of the network.

C. An example

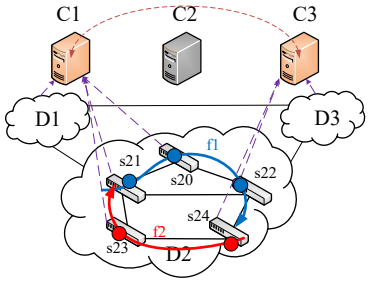
We use Fig. 3 as an example to illustrate the three considerations for designing PM under the same controller failure in



flow	programmability
f1	4
f2	4

controller	control resource
C1	1
C3	0

(a) The result of only recovering offline flows with balanced programmability.



flow	programmability
f1	6
f2	4

controller	control resource
C1	0
C3	0

(b) The result of PM, which recovers offline flows with balanced programmability and maximizes the total programmability.

Fig. 3: An example that illustrates design considerations. A colorful circle denotes that the flow with the same color works under SDN mode and is programmable at the switch.

Fig. 1(c). Fig. 3(a) shows the result of a hybrid flow routing-based solution. In this figure, switches s_{20}, s_{21}, and s_{23} are mapped to active controller C_1, while other two switches are remapped to controller C_3. Flow f^1 is configured with SDN mode at switch s_{21} and s_{22}, flow f^2 is recovered at switch s_{24} and s_{23} with SDN mode. This solution can maintain balanced programmability of recovered flows because all the flows are recovered with the programmability of 4. However, in this figure, controller C_1 is not fully utilized. Fig. 3(b) shows the result of PM, which realizes the three objectives.

IV. Problem Formulation

In this section, we formulate the FMSSM problem to decide the routing mode of offline flows at offline switches and remap offline switches to active controllers.

A. System description

Typically, an SD-WAN consists of H distributed controllers, and each controller controls a domain of switches. Controllers C_{M+1}, ... , C_H fail, and they previously 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 $\{C_{M+1}, ..., C_H\}$ are $\mathcal{S} = \{s_1, ..., s_i, ..., s_N\}$. D_{ij} denotes the propagation delay between switch s_i ($i \in [1, N]$) and controller C_j ($j \in [1, M]$). The set of flows traversing the set of offline switches $\mathcal{S}$ is $\mathcal{F} = \{f^1, f^2, ..., f^l, ..., f^L\}$. If the forwarding path of flow f^l ($l \in [1, L]$) 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 $x_{ij} = 1$ to denote that switch s_i ($i \in [1, N]$) is mapped to controller C_j; otherwise $x_{ij} = 0$. If flow f^l's forwarding path traverses switch s_i and is configured with the SDN routing mode, we have $y_i^l = 1$; otherwise $y_i^l = 0$. The relationship between y_i^l and β_i^l can be expressed as follows:

$$y_i^l \leq \beta_i^l, \forall\ i, \forall\ l. \tag{1}$$

In the above inequality, the equal sign comes when f^l is forwarded under SDN mode at s_i. If the inequality sign is applied, f^l traverses s_i and is forwarded based on the legacy routing table without the controller. For simplicity, in the rest of this section, we use a switch instead of an offline switch.

B. Constraints

1) Switch-controller mapping constraint: Each switch can be mapped to at most one controller. That is:

$$\sum_{j=1}^{M} x_{ij} \leq 1, \forall\ i. \tag{2}$$

2) Controller resource constraint: When controllers fail, active controllers should try their best to control offline switches, which are previously controlled by failed controllers without interrupting their normal operations. The control load of a controller equals to the total overhead of controlling the 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 [16]). The control load of a controller should not exceed the controller's available control resource. This can be written as follows:

$$\sum_{l=1}^{L} \sum_{i=1}^{N} (x_{ij} * \beta_i^l * y_i^l) \leq A_j^{rest}, \forall\ j. \tag{3}$$

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

3) Flow programmability constraint: If switch s_i on the path of flow f^l and uses SDN mode to route this flow, it can only change the next hop on the path of flow f^l. We use p_i^l to 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:

$$pro_i^l = \sum_{j=1}^{M} (x_{ij} * \beta_i^l * y_i^l * p_i^l), \forall\ i, \forall\ l.$$

The path programmability of flow f^l is denoted as pro^l and equals to the sum of the path programmability at all switches. We use $\bar{p}_i^l$ to represent $\beta_i^l * p_i^l$, and pro^l can be formulated as follows:

$$pro^l = \sum_{i=1}^{N} pro_i^l = \sum_{i=1}^{N} \sum_{j=1}^{M} (\bar{p}_i^l * x_{ij}^l * y_i^l), \forall\ l.$$

We use r to denote the least programmability of all offline flows. Thus, we have:

$$pro^l = \sum_{i=1}^{N} \sum_{j=1}^{M} (\bar{p}_i^l * x_{ij}^l * y_i^l) \geq r, \forall\ l. \tag{4}$$

4) Propagation delay constraint: Normally, to reduce the control propagation delay, switches are mapped to active controllers which have the enough control resource and are near to the switches. However, during controller failures, active controllers which are close to offline switches may not have enough control resource to recover offline switches. Simply mapping offline switches to their near active controllers could introduce extra processing delay [16] or even lead to cascading controller failure [8]. To maintain low control propagation delay when recovering offline switches, we restrict the total propagation delay not exceeding the propagation delay of the ideal recovering case, where each offline switch is mapped to its nearest active controller without introducing extra delay. That is:

$$\sum_{l=1}^{L}\sum_{j=1}^{M}\sum_{i=1}^{N}(x_{ij} * \beta_i^l * y_i^l * D_{ij}) \leq G, \tag{5}$$

where G denotes the total propagation delay of the ideal recovery case and is formulated as follows:

$$G = \sum_{j=1}^{M}\sum_{i=1}^{N}(\alpha_{ij} * \gamma_i * D_{ij}), \tag{6}$$

where α_{ij} denotes the relationship between s_i and controller C_j (i.e., if controller C_j is the nearest controller of switch s_i, $\alpha_{ij} = 1$; otherwise $\alpha_{ij} = 0$), and γ_i denotes the number of flows in switch s_i.

C. Objective functions

The objective functions reflect our design considerations. Our problem has two objectives. This first objective reflects the first and second considerations and aims to recover offline flows as many as possible and let each flow have the similar programmability. That is

$$obj_1 = r. \tag{7}$$

The second objective reflects the third consideration and is to make full use of the control resource of active controllers by improving the total programmability of recovered flows at offline switches. That is:

$$obj_2 = \sum_{l=1}^{L} pro^l. \tag{8}$$

D. Problem formulation

The goal of our problem is to recover offline flows with the similar path programmability and maximize the total programmability of the offline flows at offline switches by selecting the routing mode of flows at recovered switches and smartly mapping the recovered switches to active controllers.

Typically, we have two options to formulate the FMSSM problem. The first option is to formulate a two-stage problem. In practice, the first objective has the highest priority because it maintains the essential of SDN. Thus, the first-stage problem tries to recover each offline flow to have the similar path programmability. The second-stage problem maximizes the total programmability based on the path programmablilty of flows obtained by solving the first-stage problem. The second option is to formulate one problem by combining the two objectives into one objective. Compared with the two-stage problem formulation that needs to solve two problems, the second option only solves one problem. Existing works show that selecting the right weight among two objectives can guarantee the optimal solution same to the two-stage problem [17]. In this problem, we select the second option. We formulate the objective as follows:

$$obj = obj_1 + \lambda * obj_2 = r + \lambda \sum_{l=1}^{L} pro^l,$$

where λ ($\lambda \geq 0$) is a constant number gives different weights of the two objective terms and is selected following [17]. Therefore, we can formulate the problem as follows:

$$\begin{aligned} \max_{r,x,y} \quad & r + \lambda \sum_{l=1}^{L} pro^l \\ \text{s.t.} \quad & (1)(2)(3)(4)(6)(5), \\ & r \geq 0, x_{ij}, y_i^l \in \{0,1\}, \forall\ i, \forall\ j, \forall\ l. \end{aligned} \tag{P}$$

E. Problem reformulation

In the FMSSM problem, the objective function is linear, variables are binary integers, and constraints are nonlinear. Thus, this problem is a Mixed-Integer NonLinear Programming (MINLP), which is widely known for the NP-hard computation complexity. One complexity of the FMSSM problem comes from Eqs. (3), (4), and (5) since two binary variables are multiplied in the two equations. To efficiently solve the problem, we reformulate the problem to an Integer Programming (IP) by equivalently linearizing the bilinear terms of binary variables $x_{ij} * y_i^l$ in Eqs. (3), (4), and (5). Specifically, we can replace the bilinear term $x_{ij} * y_i^l$ by introducing an auxiliary binary variable ω_{ij}^l and add the following linear constraints:

$$x_{ij} \geq \omega_{ij}^l, \forall\ i, \forall\ j, \forall\ l. \tag{9}$$

$$y_i^l \geq \omega_{ij}^l, \forall\ i, \forall\ j, \forall\ l. \tag{10}$$

$$(x_{ij} + y_i^l - 1) \leq \omega_{ij}^l, \forall\ i, \forall\ j, \forall\ l. \tag{11}$$

Thus, Eqs. (3) (4) and (5) can be respectively reformulated as follows:

$$\sum_{l=1}^{L}\sum_{i=1}^{N}(\omega_{ij}^l * \beta_i^l) \leq A_j^{rest}, \forall\ j. \tag{12}$$

$$\sum_{i=1}^{N}\sum_{j=1}^{M}(\overline{p}_i^l * \omega_{ij}^l) \geq r, \forall\ l. \tag{13}$$

$$\sum_{l=1}^{L}\sum_{j=1}^{M}\sum_{i=1}^{N}(\omega_{ij}^l * \beta_i^l * D_{ij}) \leq G. \tag{14}$$

TABLE II: Notations

Notation	Meaning
$\mathcal{S}$	the set of offline switches, $\mathcal{S} = \{s_i \mid i \in [1, N]\}$
$\mathcal{C}$	the set of active controllers, $\mathcal{C} = \{C_j \mid j \in [1, M]\}$
$\mathcal{D}(i)$	the propagation delay of switch s_i to active controllers, $\mathcal{D}(i) = \{D_{i1}, ..., D_{ij}, ..., D_{iM}\}, i \in [1, N]$
$\mathcal{C}(i)$	the set of active controllers by sorting $\mathcal{C} = \{C_j \mid j \in [1, M]\}$ following the ascending order of $\mathcal{D}(i)$, $i \in [1, N]$
$\mathcal{A}$	the set of the available processing capacity of controllers, $\mathcal{A} = \{A_j^{rest} \mid j \in [1, M]\}$, where A_j^{rest} denotes the available processing capacity of controller C_j
$\mathcal{H}$	the set of temporary path programmability of flows, $\mathcal{H} = \{h^l \mid l \in [1, L]\}$, where h^l denotes the temporary path programmability of flow f^l
$\mathcal{P}$	the set of the path programmability for flow f^l at switch s_i, $\mathcal{P} = \{pro_i^l \mid i \in [1, N], l \in [1, L]\}$, where pro_i^l denotes the path programmability of flow f^l at switch s_i
$\mathcal{X}$	the set of the mapping relationship between offline switches and active controllers, $\mathcal{X} = \{(i, j) \mid i \in [1, N], j \in [1, M]\}$
$\mathcal{Y}$	the set of the mode setting relationship between offline flows and offline switches, $\mathcal{Y} = \{(i, l) \mid i \in [1, N], l \in [1, L]\}$
σ	the auxiliary programmability variable that indicates the least programmability of all offline flows
δ	the auxiliary flow variable that indicates the number of offline flows that have the least programmability in all offline switches
Γ	the set of the number of flows at switches, $\Gamma = \{\gamma_i \mid j \in [1, M]\}$, where γ_i denotes the number of flows traversing s_i
TOTAL_ITERATIONS	the maximum number of offline switches in the original path of offline flows

Therefore, the FMSSM problem can be reformulated as follows:

$$\begin{aligned} \max_{r,x,y,\omega} \quad & r + \lambda \sum_{l=1}^{L} pro^l \\ \text{s.t.} \quad & (1)(2)(6)(9)(10)(11)(12)(13)(14), \\ & r \geq 0, x_{ij}, y_i^l, \omega_{ij}^l \in \{0, 1\}, \forall\ i, \forall\ j, \forall\ l. \end{aligned} \tag{P'}$$

V. PM Design

The typical solution of the FMSSM problem is to get its optimal result using existing IP optimization solvers. However, as the network size increases, the solution space could increase significantly, and finding a feasible solution could cost a long time or perhaps is impossible. We design the heuristic algorithm named PM to solve the problem and achieve the trade-off between the performance and time complexity.

The key idea of PM includes two parts: (1) preferably recovering offline flows, which have the least programmability and (2) making full use of available control resource to improve the total programmability. Details are summarized in Algorithm 1, and the notations used in the algorithm are listed in Table II. At the beginning of the algorithm, in line 1, we initialize $\mathcal{X}$ and $\mathcal{Y}$ to be empty, set test switch-set $\mathcal{S}^*$ to $\mathcal{S}$, auxiliary programmability variable σ to 0, and tested_counter to 0. In line 2, we start iteratively to find the feasible mappings between switches and controllers and select the routing mode for offline flows at switches. During an iteration, the programmability of flows, which have the least programmability, can be increased by one at most to balance the path programmability of each flow. To guarantee that all the offline switches in the original path of each flow are tested, the program totally runs TOTAL_ITERATIONS iterations.

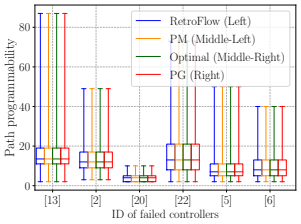
Algorithm 1 PM

Input: $\mathcal{S}$, $\mathcal{C}$, $\mathcal{A}$, $\mathcal{D}$, $\mathcal{P}$, $\mathcal{R}$, TOTAL_ITERATIONS;
Output: $\mathcal{X}$, $\mathcal{Y}$;

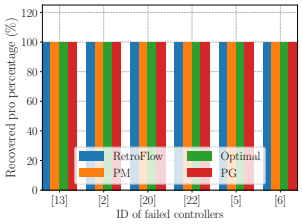
1: $\mathcal{X} = \emptyset$, $\mathcal{Y} = \emptyset$, $\mathcal{S}^* = \mathcal{S}$, $\sigma = 0$, test_count = 0;
2: **while** test_count $\leq$ TOTAL_ITERATIONS **do**
3: δ = 0, i_0 = NULL, j_0 = NULL;
4: //find switch s_{i_0} to recover
5: **for** $s_i \in \mathcal{S}^*$ **do**
6: TEST_NUM $= 0$;
7: **for** $l \in \{\beta_i^l = 1, l \in [1, L]\}$ **do**
8: **if** $h^l == \sigma$ **then**
9: TEST_NUM = TEST_NUM + 1;
10: **end if**
11: **end for**
12: **if** TEST_NUM $> \delta$ **then**
13: $\delta =$ TEST_NUM, $i_0 = i$;
14: **end if**
15: **end for**
16: //map switch s_{i_0} to controller C_{j_0}
17: **if** $(i_0, *) \in \mathcal{X}$ **then**
18: use i_0 to find j_0 from $\mathcal{X}$;
19: **else**
20: **for** $C_j \in C(i_0)$ **do**
21: **if** $A_j^{rest} \geq \gamma_{i_0}$ **then**
22: $j_0 = j$;
23: **end if**
24: **end for**
25: **if** $j_0 ==$ NULL **then**
26: A_j^{rest} = max($\mathcal{A}$), $j_0 = j$;
27: **end if**
28: **end if**
29: $\mathcal{X} \leftarrow \mathcal{X} \cup (i_0, j_0)$, $\mathcal{S}^* \leftarrow \mathcal{S}^* \setminus s_{i_0}$;
30: //select the routing mode for flows at switch s_{i_0}
31: **for** $l_0 \in \{\beta_{i_0}^l = 1, l \in [1, L]\}$ **do**
32: **if** $h^{l_0} \leq \sigma$ and $(i_0, l_0) \notin \mathcal{Y}$ and $A_{j_0}^{rest} > 0$ **then**
33: $A_{j_0}^{rest} = A_{j_0}^{rest} - 1$, $h^{l_0} = h^{l_0} + pro_{i_0}^{l_0}$;
34: $\mathcal{Y} \leftarrow \mathcal{Y} \cup (i_0, l_0)$;
35: **end if**
36: **end for**
37: **if** $|\mathcal{S}^*| == \emptyset$ **then**
38: $\mathcal{S}^* = \mathcal{S}$, test_count++, σ = min($\mathcal{H}$);
39: **end if**
40: **end while**
41: //improve the total programmability
42: **for** $(i_0, l_0) \in \{\beta_i^l = 1, i \in [1, M], l \in [1, L]\}$ **do**
43: **if** $(i_0, *) \in \mathcal{X}$ **then**
44: use i_0 to find j_0 from $\mathcal{X}$;
45: **if** $A_{j_0}^{rest} > 0$ and $(i_0, l_0) \notin \mathcal{Y}$ **then**
46: $A_{j_0}^{rest} = A_{j_0}^{rest} - 1$, $h^{l_0} = h^{l_0} + pro_{i_0}^{l_0}$;
47: $\mathcal{Y} \leftarrow \mathcal{Y} \cup (i_0, l_0)$;
48: **end if**
49: **end if**
50: **end for**
51: return $\mathcal{X}, \mathcal{Y}$;

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

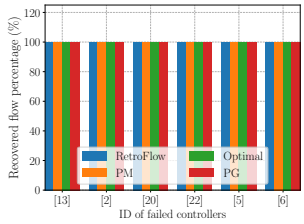
Controller ID	2				5				6				13					20		22					
Switch ID	2	3	9	16	4	5	8	14	0	1	6	7	10	11	12	13	15	19	20	17	18	21	22	23	24
Number of flows	143	71	107	55	49	143	53	61	81	49	89	97	63	59	71	213	67	49	63	125	49	81	111	49	57



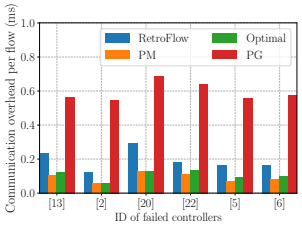
(a) Path programmability of recovered flows. The higher, the better.



(b) Percentage of total path programmability of recovered flows to RetroFlow. The higher, the better.



(c) Percentage of recovered programmable flows from offline switches. The higher, the better.



(d) Per-flow communication overhead. The lower, the better.

Fig. 4: Results of one controller failure.

If the program continues after TOTAL_ITERATIONS times iterations, the path programmability of offline flows will not get improved because all possible improvements for each offline flow are already tested. In line 3, for each iteration, we set auxiliary flow variable δ to 0, the index of recovering switch i_0 to NULL, and the index of mapping controller j_0 to NULL. In lines 5-15, we select switch s_{i_0} to recover. First, we collect the number of flows, which have the least programmability from $\mathcal{S}^*$ (lines 6-11), and select the switch, which has the maximum number of flows with the least programmability. Then, we choose switch s_{i_0} to recover by updating this switch's index to i_0 (lines 12-14).

Lines 17-29 map switch s_{i_0} to controller C_{j_0}. In line 18, if switch s_{i_0} is already remapped, we use index i_0 to find index its controller's index j_0 in $\mathcal{X}$. Otherwise, switch s_{i_0} is not remapped to any active controller, and we find an active controller in lines 20-27. In lines 20-24, we test controllers following the ascending order of the propagation delay with switch s_{i_0}. If there exists a controller that has enough control resource to control switch s_{i_0}, we establish the mapping between switch s_{i_0} and controller C_{j_0}. If we cannot find a capable controller, in line 26, we find C_{j_0}, the controller with the maximum available control resource, to control switch s_{i_0}. In line 29, the mapping between switch s_{i_0} and controller C_{j_0} is established, set $\mathcal{X}$ is updated according, and the mapped switch s_{i_0} is removed from $\mathcal{S}^*$.

Lines 31-36 select routing mode for offline flows in switch s_{i_0}. We try to select offline flows with the least programmability as many as possible and configure them working under the SDN mode without overloading controller C_{j_0}. C_{j_0}'s control resource A_j^{rest}, recovered flow f^{l_0}'s routing mode and programmability h^{l_0} are updated accordingly.

In lines 37-39, if all switches are tested, we should start a new iteration. This is because the least programmability could be further improved by configuring more flows working under SDN mode at switches. To restart the iteration, we configure $\mathcal{S}^*$ to $\mathcal{S}$, increase tested_counter by one, and upgrade the least programmability σ.

After the iteration from lines 2 to 40, the least programmability cannot be further increased. However, active controllers' control resource may not be fully utilized, and there is still room to improve the total programmability. As a result, in lines 42-50, we test all flows to find out if there still exists feasible SDN mode configurations for flows to further improve the total programmability. In line 51, the routing mode of recovered flows at recovered switches and switch-controller mappings are finally generated.

VI. Simulation

A. Simulation setup

We use a typical backbone topology ATT from Topology Zoo [18] to evaluate the performance of PM. The ATT topology is a national primary topology of US and consists of 25 nodes and 112 links. In ATT topology, each node has a unique ID, a latitude, and a longitude. We calculate the distance between two nodes using Haversine formula [19] and use the distance divided by the propagation speed (i.e., 2×10^8 m/s) [20] to represent the propagation delay between the two nodes. In our simulation, each node is an SDN switch, and any two nodes have a traffic flow forwarded on the shortest path. Following existing works [6], [9], the control plane consists of six controllers, and the processing ability of each controller is 500. Table III shows the default relationship of controllers, switches, and the number of flows in the switches under the ATT topology.

B. Comparison algorithms

1) Optimal: it is the optimal solution of problem P'. We solve it with an advanced solver GUROBI [21].
2) RetroFlow [6]: 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.
3) PG [9]: this solution realizes fine-grained flow-controller mappings using a middle layer between controllers and switches.
4) PM: our solution recovers offline flows with the similar path programmability and maximizes the total pro-

grammability of recovered flows by deciding the proper mode of flows at recovered offline switches and configuring switch-controller mappings. The details are presented in Algorithm 1.

C. Simulation results

We compare the performance of ProgrammabilityMedic with other algorithms under three scenarios: one controller failure, two controller failures and three controller failures. We also evaluate PM's computation efficiency. Following our two objectives, we use two metrics: path programmability of recovered flows, which reflects the performance of obj_1 in Eq.(7), and total path programmability of recovered flows, which reflects the performance of obj_2 in Eq.(8).

1) One controller failure: Fig. 4 shows the results of four algorithms when one of six controllers fails. One controller failure is a common scenario, and there are 6 combinations. In all six cases, active controllers usually have enough control resource to recover all offline switches. In Fig. 4(a), all algorithms realize the same programmability of recovered flows. In Fig. 4(b), all solutions exhibit the same total programmability of recovered flows since all algorithms recover 100% offline flows, as shown in Fig. 4(c). Fig. 4(d) shows the per-flow communication overhead of all algorithms. The total communication overhead comes from the propagation delay due to the geographical distance, and the overhead is the total communication overhead divided by the number of recovered flows. In this figure, PM requires the least overhead because it takes the propagation delay into consideration. While PG performs worst since it introduces a middle layer using FlowVisor, which needs 0.48 ms on average to pull for port status [10].

2) Two controller failures: Fig. 5 shows the results of four algorithms when two of six controllers fail. Two controllers failure is a moderate failure scenario, and there are 15 combinations.

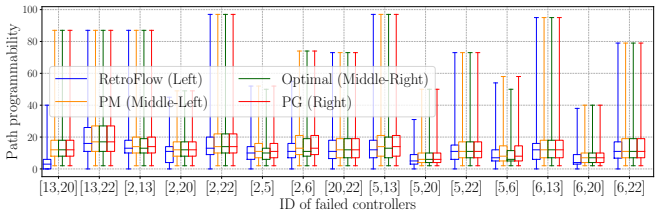
(1) Path programmability of recovered flows: Fig. 5(a) shows the path programmability of recovered flows. For all cases, RetroFlow's least path programmability is 0 since some flows are not recovered, and its median is always lower than PM, Optimal, and PG. PM exhibits performance similar to Optimal and PG.

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 with short paths. 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 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 types of flows' programmability is limited to 2, and thus the performance of PM, Optimal, and PG is limited by these flows. However, the results still indicate that PM, Optimal, and PG maintain balanced programmability because all the flows are recovered to have the programmability of 2 at least. This figure also proves that our second objective of maximizing the total programmability in Eq. (8) is critical because it can improve the utilization of control resource in active controllers for flow recovery. Without this objective, all flows will just be recovered with the least programmability.

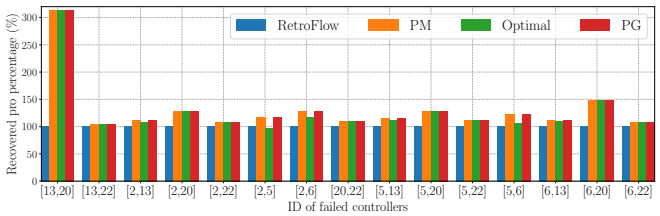
(2) Total path programmability of recovered flows: Fig. 5(b) shows the total path programmability. To clearly show the performance difference, we normalize the result of each algorithm to the result of RetroFlow. PM and PG perform best among the four algorithms, and PM improves the performance from 105% to 315% because PM recovers more offline flows. Fig. 5(c) shows the percentage of recovered flows, and Fig. 5(d) shows the percentage of recovered switches. Fig. 5(e) shows the control resource of active controllers. RetroFlow performs worse than others since it recovers a small number of offline flows with much higher control resource of active controllers. PM performs nearly the same to PG. For all cases, RetroFlow performs worst because it does not recover the same number of offline flows as other algorithms do. RetroFlow only recovers flows in the range of 71% to 99% and only recovers part of offline switches because the coarse-grained switch-controller mappings cannot fully occupy active controllers to recover flows without overloading active controllers. If an offline flow only traverses unrecovered switches, it remains offline.

In contrast, PG recovers all the offline switches thanks to the fine-grained flow-controller mappings. By employing the hybrid routing mode, Optimal and PM approximately realize fine-grainedly flow-level recovery. Thus, they can change the control cost of offline switches and enable active controllers to recover more switches than existing switch-level solutions, Thus, they increase the number of recovered switches and achieve the performance similar to PG to recover all flows. For the special failure case (13, 20), PM's performance is 315% of RetroFlow's. Specifically, with RetroFlow, the control cost of offline switch s_{13} is 213, but the available control resource of active controllers C_2, C_5, C_6, and C_{22} are 124, 194, 184, and 28, respectively. Thus, switch s_{13}'s control cost cannot match the control resource of any controllers, and thus it cannot be recovered. On the contrary, PM can recover the programmability in a fine-grained per-flow mode by dynamically altering the control cost based on the given control resource. Supported by high-end commercial SDN switches with hybrid routing mode, 188 of 213 flows are configured with SDN routing mode at switch s_{13}, and the rest of flows are with legacy mode. So, PM can remap switch s_{13} to controller C_5 by dynamically altering the control cost of switch s_{13} based on the control resource of active controllers.

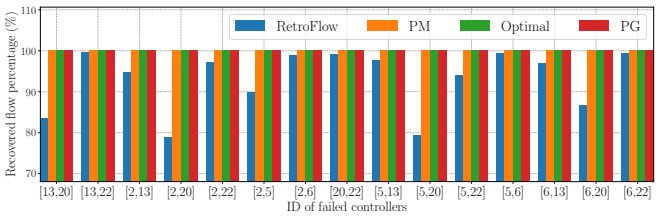
(3) Communication overhead: Our problem formulation uses Eqs. (6) and (14) to limit the overall communication overhead. Because each algorithm recovers different number of flows, both the number of recovered flows and overall communication overhead should be considered in the evaluation metric. Fig. 5(f) shows the per-flow communication overhead. For all cases, PG performs worst and is about three to four times higher than PM on average. This is because the middle



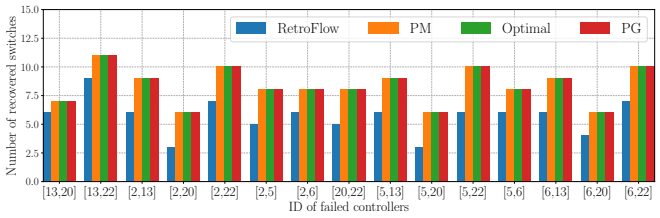
(a) Path programmability of flows. The higher, the better.



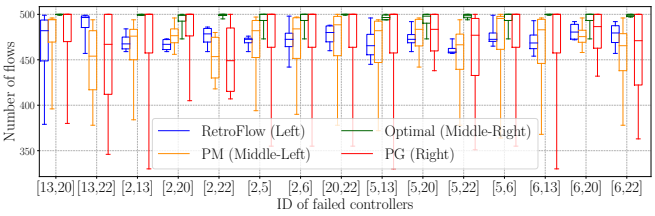
(b) Percentage of total path programmability of recovered flows to RetroFlow. The higher, the better.



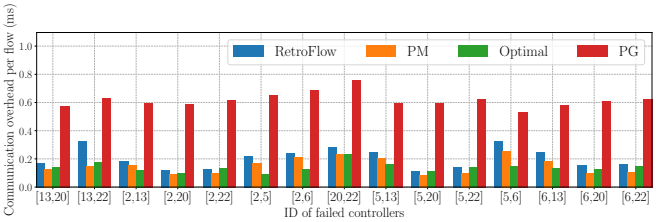
(c) Percentage of recovered programmable flows from offline switches. The higher, the better.



(d) Number of recovered offline switches. The higher, the better.



(e) Control resource of active controllers.



(f) Per-flow communication overhead. The lower, the better.

Fig. 5: Results of two controller failures.

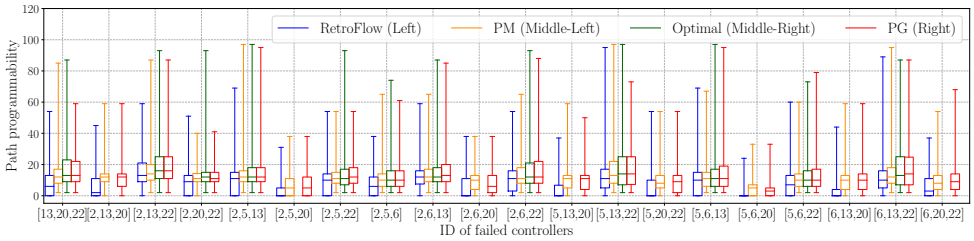
layer's processing time significantly increases the overhead. PM outperforms other three algorithms in 8 of 15 cases while Optimal performs best in rest of 7 cases. That is because the total propagation delay G under the ideal recovery case varies under different failure cases. In the 8 of 15 cases, the total propagation delay of heuristic PM is lower than G, but the total propagation delay of Optimal can be only limited to G.

3) Three controller failures: Fig. 6 shows the results of four algorithms when three of six controllers fail. Three controllers failure is a serious failure scenario, and there are 20 combinations. Note that we only present the results of Optimal in 12 of 20 cases. Recall that our problem has a constraint of not interrupting active controllers' normal operations. Under this constraint, optimization solver may not always generate a feasible solution. The similar situation can also be found in [6], [9]. PM is a heuristic algorithm and always has a result.

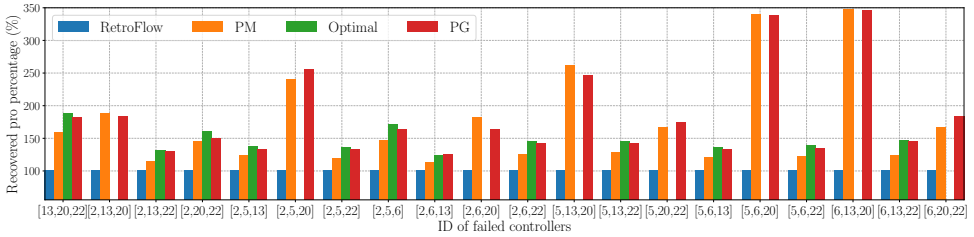
(1) Path programmability of recovered flows: Fig. 6(a) shows the path programmability of recovered flows. Similar to the two controller failures, for all cases, RetroFlow's least path programmability is 0 because some flows are not recovered, and its median is always lower than PM, Optimal, and PG. PM and PG exhibit balanced programmability because the least programmability is recovered to 2 in most of the cases. Note that PM's median is close to PG's, but its highest value is lower than PG's in the majority of cases. This result indicates PM does better in term of maintaining balanced programmability than PG does.

(2) Total path programmability of recovered flows: Fig. 6(b) shows the total path programmability. For the 20 cases, Optimal only gets the result in 12 of 20 cases, and it performs best in these 12 cases. PM performs closer to PG and even better than PG in the cases of (2, 13, 20), (2, 6, 20), and (5, 13, 20) because PG also considers the communication overhead as its optimization objective. For failure case (5, 6, 20), PM's performance is 340% of RetroFlow's. The control cost of offline switch s_{13} is 223, but the available control resource of active controllers C_2, C_5, and C_{22} are 124, 184, and 34, respectively. Neither of the active controllers can re-control switch s_{13} due to the per-switch mapping. On the contrary, PM can recover the programmability in a fine-grained per-flow mode by dynamically altering the control cost of offline switches based on given control resource of active controllers.

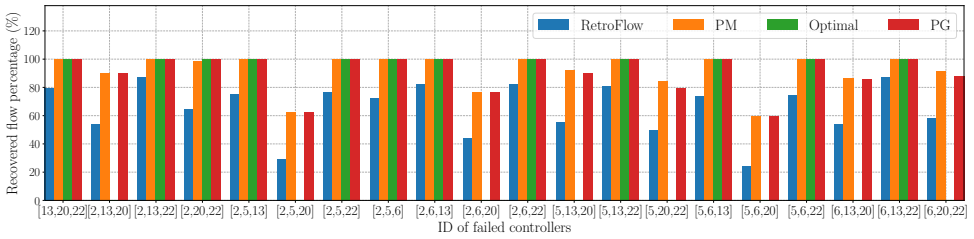
RetroFlow performs worse than others since it performs worse in path programmability and the percentage of recovered flows. Fig. 6(c) shows the percentage of recovered flows. For all cases, RetroFlow only recovers flows in the range of 25% to 85%. PM, Optimal, and PG realize 100% flow recovery in 12 of 20 cases. For the rest 8 cases, PM's performance is comparable with PG by recovering flows in the range of 60% to 92%. Fig. 6(d) shows the percentage of recovered switches. For all cases, RetroFlow only recovers a small part of offline switches because the switch-level mapping solution cannot fully utilize the available resource of active controllers, as shown in Fig. 6(f). PM's offline switch recovery performance



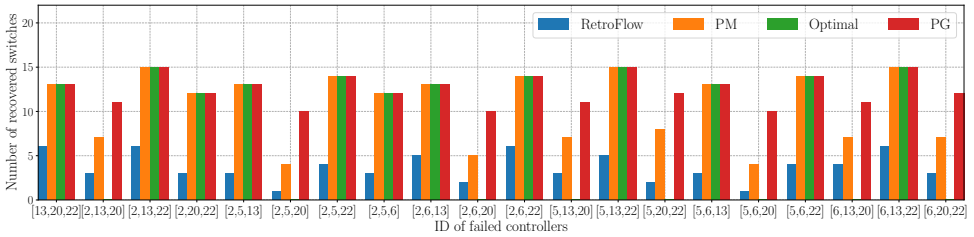
(a) Path programmability of flows. The higher, the better.



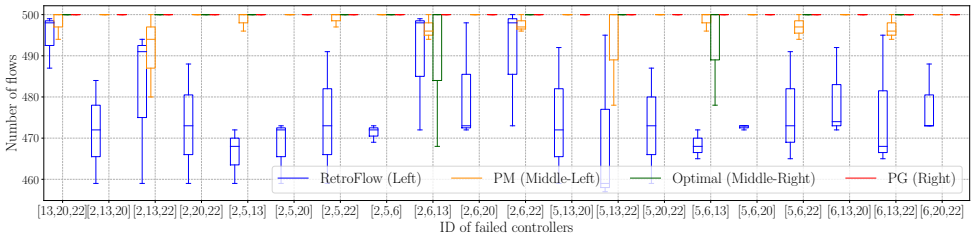
(b) Percentage of total path programmability of recovered flows to RetroFlow. The higher, the better.



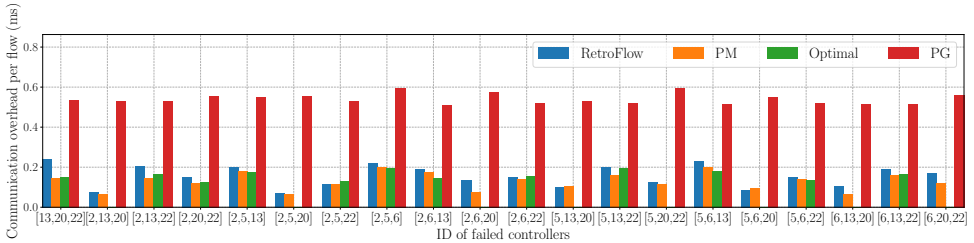
(c) Percentage of recovered programmable flows from offline switches. The higher, the better.



(d) Number of recovered offline switches. The higher, the better.



(e) Control resource of active controllers. Closer to 500 is better.



(f) Per-flow communication overhead. The lower, the better.

Fig. 6: Results of three controller failures. Optimal cannot always have results due to the FMSSM problem's high complexity.

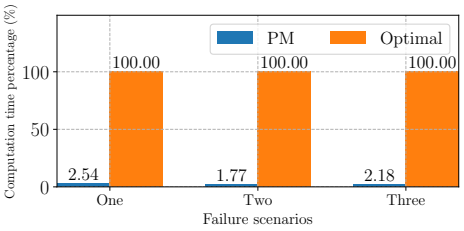


Fig. 7: Percentage of computation time of PM to Optimal. The lower, the better.

is close to PG's. In Fig. 6(e), PM, Optimal, and PG fully utilize active controllers' control resource for the recovery.

(3) Communication overhead: Fig. 6(f) shows per-flow communication overhead. PG performs worst for all cases while PM performs best in most of cases. Note that in cases (5, 13, 20) and (5, 6, 20), RetroFlow slightly outperforms PM because the path programmability of each recovered flow of RetroFlow is far less than PM, as shown in Fig. 6(a).

4) Computation time: To show the computation efficiency, we evaluate the computation time of PM and Optimal under the above three scenarios, and Fig. 7 shows the result. In the figure, the computation time of PM is only 2.54%, 1.77%, and 2.18% of Optimal on average under three failure scenarios. Thus, considering recovery performance in Figs. 4, 5, and 6, the result indicates that the proposed PM can achieve good recovery performance with low computation time.

VII. Related Work

A. Resilient network control

In order to realize resilient network control, Vizarreta et al. [22] present two strategies to address the Reliable Controller Placement (RCP) problem, which protects the control plane against link and node failures. Tivig et al. [23] attempt a critical study of different placement solutions to provide resiliency. Alenazi et al. [24] present a new metric to measure node's importance to determine the locations of controllers against network failure.

B. Controller-switch mapping

Tanha et al. [3] take both the switch-controller and inter-controller latency requirements and the capacity of the controllers into consideration to select the resilient placement of controllers. RetroFlow [6] intelligently configures a set of selected offline switches working under the legacy routing mode to relieve the active controllers from controlling the selected offline switches while maintaining the flow programmability of SDN. ProgrammabilityGuardian [9] improves the path programmability of offline flows and maintains low communication overhead by using a middle layer to establish the fine-grained flow-controller mappings. Wang et al. [25] propose DCAP to consider dynamic controller assignment so as to minimize the average response time of the control plane in data center networks.

VIII. Conclusion

In this paper, we propose PM, a switch-level programmability recovery solution to recover offline flows with balanced programmability and improve the total programmability of offline flows. Thanks to the hybrid SDN/legacy routing mode supported by high-end commercial SDN switches, PM approximately realizes the performance of the flow-level recovery solution by smartly deciding the routing mode of each offline flow at recovered switches and establishing effective mappings between offline switches and active controllers for programmability recovery.

Acknowledgements

This paper is supported by the National Natural Science Foundation of China under Grants 62002019 and 61802315, and the Beijing Institute of Technology Research Fund Program for Young Scholars.

References

[1] C. Hong, S. Kandula, R. Mahajan, M. Zhang, and et al., "Achieving high utilization with software-driven WAN," in *ACM SIGCOMM'13*.
[2] S. Jain, A. Kumar, S. Mandal, and et al., "B4: Experience with a globally-deployed software defined WAN," in *ACM SIGCOMM'13*.
[3] M. Tanha and et al., "Capacity-aware and delay-guaranteed resilient controller placement for software-defined wans," *IEEE TNSM*, vol. 15, no. 3, pp. 991–1005, 2018.
[4] N. Perrot and T. Reynaud, "Optimal placement of controllers in a resilient sdn architecture," in *IEEE DRCN'16*.
[5] F. He, T. Sato, and E. Oki, "Master and slave controller assignment model against multiple failures in software defined network," in *ICC'19*.
[6] Z. Guo and et al., "Retroflow: Maintaining control resiliency and flow programmability for software-defined wans," in *IEEE/ACM IWQoS'19*.
[7] S. Dou, G. Miao, Z. Guo, and et al., "Matchmaker: Maintaining network programmability for software-defined wans under multiple controller failures," *Elsevier Computer Networks*, vol. 192, p. 108045, 2021.
[8] G. Yao, J. Bi, and L. Guo, "On the cascading failures of multi-controllers in software defined networks," in *IEEE ICNP'13*.
[9] Z. Guo and et al., "Improving the path programmability for software-defined wans under multiple controller failures," in *IEEE/ACM IWQoS'20*.
[10] R. Sherwood and et al., "Flowvisor: A network virtualization layer," *OpenFlow Switch Consortium, Tech. Rep*, vol. 1, p. 132, 2009.
[11] "Openflow switch specification 1.3," https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf.
[12] 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.
[13] "Brocade mlx-8 pe," https://www.dataswitchworks.com/datasheets/MLX_Series_DS.pdf.
[14] "Chn-ix," http://www.chn-ix.net/.
[15] H. Xu, H. Huang, S. Chen, and G. Zhao, "Scalable software-defined networking through hybrid switching," in *IEEE INFOCOM'17*.
[16] J. Xie and et al., "Cutting long-tail latency of routing response in software defined networks," *IEEE JSAC*, vol. 36, no. 3, pp. 384–396, 2018.
[17] Z. Guo and et al., "Joint switch upgrade and controller deployment in hybrid software-defined networks," *IEEE JSAC*, vol. 37, no. 5, pp. 1012–1028, 2019.
[18] S. Knight and et al., "The internet topology zoo," *IEEE JSAC*, vol. 29, no. 9, pp. 1765 –1775, 2011.
[19] C. C. Robusto, "The cosine-haversine formula," *The American Mathematical Monthly*, vol. 64, no. 1, pp. 38–40, 1957.
[20] "Speed, rates, times, delays: Data link parameters for cse 461," https://courses.cs.washington.edu/courses/cse461/99wi/issues/definitions.html.
[21] "Gurobi optimization," http://www.gurobi.com.
[22] P. Vizarreta, C. M. Machuca, and W. Kellerer, "Controller placement strategies for a resilient sdn control plane," in *IEEE RNDM'16*.
[23] P. Tivig and E. Borcoci, "Critical analysis of multi-controller placement problem in large sdn networks," in *IEEE COMM'20*.
[24] M. J. Alenazi and E. K. Çetinkaya, "Resilient placement of sdn controllers exploiting disjoint paths," *Wiley ETT*, p. e3725, 2019.
[25] T. Wang and et al., "Dynamic sdn controller assignment in data center networks: Stable matching with transfers," in *IEEE INFOCOM'16*.