HybridFlow: Achieving Load Balancing in Software-Defined WANs with Scalable Routing

Zehua Guo, *Senior Member, IEEE,* Songshi Dou, *Student Member, IEEE,* Yi Wang, *Member, IEEE,* Sen Liu, *Member, IEEE,* Wendi Feng, Yang Xu, *Member, IEEE*

***Abstract*—The scalability issue hinders the deployment of Software-Defined Networking (SDN) in the Wide Area Networks (WANs). Existing solutions have two issues: (1) network performance relies on complicated controller synchronization, which increases the complexity of network control; (2) fine-grained flow processing enables flexible flow control at the cost of high processing load on the controllers and high flow table occupancy on switches. In this paper, we propose a scalable routing solution named HybridFlow, which achieves a good load balancing performance using a single controller with low control overhead (i.e., flow routing and rerouting overhead). HybridFlow mainly employs two techniques: *hybrid routing* and *crucial flow rerouting*. Hybrid routing enabled by commercial SDN switches gives us opportunities to reduce the processing load of the controller by routing flows with the hybrid OpenFlow/OSPF mode. Thus, the majority of flows can be routed by OSPF without involving the controller. Crucial flow rerouting realizes load balancing by dynamically identifying *crucial flows* based on a new metric called *Variation Slope* and rerouting these flows with the hybrid OpenFlow/OSPF mode. The simulation based on the real traffic traces and network typologies shows that compared with the optimal solution, HybridFlow can achieve 87% of the optimal load balancing performance by rerouting 36% less flows on average.**

***Index Terms*—software-defined networking, scalability, crucial flow, routing**

I. Introduction

Internet service providers (e.g., AT&T) and content providers (e.g., Google, Microsoft, Facebook, Akamai, and Amazon) use their own private backbone networks to provide diverse network services. Due to the flexible control, Software-Defined Networking (SDN) has been deployed in their networks [1]–[4]. Thus, the network is softwarized and equipped with programmable SDN switches to provide high-quality, low-latency, resilient, and customized services.

In an SDN, the data plane consists of SDN switches, which are typically controlled by the control plane, and each flow is processed based on flow entries deployed to the switches from the control plane. There are two designs on the SDN control plane: single controller and multiple controllers. The single-controller control plane is the original design of SDN. All switches connect to one controller[1] and follow the controller's control messages to process flows. For example, the controller monitors the network status and adaptively updates the paths for flows to maintain the load balancing performance. However, the single controller could be overwhelmed when it processes many operations (e.g., path calculation, establishing, and update for multiple flows). If the controller is overloaded, the operations handled by it may suffer from long-tail latency [8], which might degrade network performance [9].

The multi-controller control plane has been proposed to avoid the limited processing ability of a single controller. Distributed controllers are physically deployed at different locations to achieve the function of a logically centralized control plane, and each controller only controls a domain of switches and processes flows in its domain. To maintain the consistent network view, the controllers must synchronize with each other [10], [11]. However, the synchronization among controllers increases the complexity of network control and could lead to network performance fluctuation under several undesired exceptional situations, which are detailed in Section II-A. Additionally, many new functions (e.g., malicious traffic detection [12] and traffic prediction [13]) are proposed to deploy in the controller and will significantly increase the controller's processing load. Thus, simplifying the controller's processing load on the legacy network functions (e.g., flow routing and rerouting) and maintaining good network performance at the same time becomes very urgent.

Open Shortest Path First (OSPF) is one of the most widely used intra-domain routing protocol. However, OSPF is not flexible, and flows are fixedly forwarded on their shortest paths in spite of network variation. The performance of OSPF-based routing depends on many factors, such as link weights and topologies [14]. While these factors are usually fixed, and the

Manuscript received May 13, 2020; revised November 22, 2020 and February 18, 2021; accepted April 8, 2021. This work was supported by the National Natural Science Foundation of China under Grants 62002019, 62002066, and 61802315, the Beijing Institute of Technology Research Fund Program for Young Scholars, the Shanghai Pujiang Program under Grant 2020PJD005, and the project "PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Applications (LZC0019)". *(Corresponding authors: Yi Wang and Yang Xu)*

Z. Guo and S. Dou are with Beijing Institute of Technology, Beijing 100081, China.

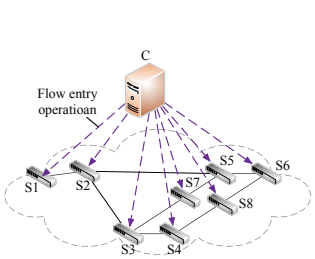
Y. Wang is with University Key Laboratory of Advanced Wireless Communications of Guangdong Province, Southern University of Science and Technology, Shenzhen 518055, China.

S. Liu is with the School of Computer Science, Fudan University, Shanghai 200433, China.

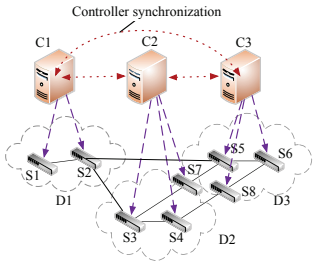
W. Feng is with Beijing University of Posts and Telecommunications, Beijing 100876, China.

Y. Xu is with School of Computer Science, Fudan University, Shanghai 200433, China, and Peng Cheng Laboratory, Shenzhen 518000, China.

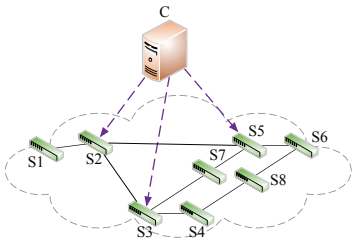
[1]Typically, an SDN controller (e.g., ONOS [5] and OpenDayLight [6]) at one location is physically implemented by a cluster of controller instances with consistent network state (e.g., using Raft [7]) to prevent single-node-of-failure. If one controller instance fails, other active instances still work without service interruption. In the rest of the paper, we use a controller short for a controller instance cluster in a location.



(a) Single controller with per-flow routing. The single-controller control plane suffers from limited the processing ability of the controller.



(b) Multiple controllers with per-flow routing. The network is divided into three domains, and each controller controls one domain. The network performance is affected by the controller synchronization.



(c) The single controller uses HybridFlow to control some crucial flows on three switches, and the rest flows are forwarded using OSPF without involving the controller.

Fig. 1. Design comparison. Gray switches in (a) and (b) are under OpenFlow mode, and green switches in (c) work under hybrid OpenFlow/OSPF mode.

shortest paths remain unchanged. As a result, when incoming traffic matrix changes under various networking scenarios, the shortest paths cannot be dynamically changed to achieve optimal or near-optimal performance [15]. Thus, the network could experience congestion during traffic variation. The solution that only uses OSPF does not guarantee to achieve optimal load balancing. SDN can provide more flexible routing than OSPF does by adjusting routing policy dynamically according to traffic variation, but it suffers from the cost of high control overhead.

In this paper, we propose a scalable routing solution named *HybridFlow*, which provides a good load balancing performance using a single controller with low control overhead (i.e., flow routing and rerouting overhead). HybridFlow is inspired by two observations: (1) the degradation of load balancing performance usually comes from several *crucial links*, which have high traffic loads. (2) current commercial SDN switches (e.g., Brocade MLX-8 PE [16]) support the hybrid OpenFlow/OSPF mode[2]. By configuration, flows at the SDN switch can be routed using OpenFlow, OSPF, or joint OpenFlow and OSPF, which are detailed in Section III-A. Based on these two observations, we design HybridFlow with two techniques: *hybrid routing* and *crucial flow rerouting*. The hybrid routing enables us to route a majority of flows with OSPF to release the controller's load and use OpenFlow entries to stitch partial existing shortest paths from OSPF for rerouting crucial flows. Crucial flow rerouting realizes load balancing by dynamically identifying several *crucial flows* on the crucial links with high traffic load based on a new metric called *Variation Slope* and rerouting these flows with the hybrid OpenFlow/OSPF mode. Using the two techniques together, HybridFlow maintains good network performance with low control overhead by reducing the number of rerouted flows and the number of the controller's operations for flow rerouting. Fig. 1 uses an example to compare HybridFlow with the two existing control plane designs.

The main contribution of this paper can be summarized as follows:

1) We propose HybridFlow to realize good load balancing performance with low processing overhead by dynamically identifying crucial flows and rerouting them on forwarding paths configured with the hybrid routing.

2) We formulate the crucial flow rerouting as an optimization problem with the objective of achieving the optimal load balancing performance under given resource constraints and propose a heuristic algorithm to efficiently solve the problem.

3) The simulation based on the real traffic traces and network topologies shows that compared with the optimal solution, HybridFlow can achieve 87% of the optimal load balancing performance by rerouting 36% less flows on average.

HybridFlow focuses on WANs that use the source IP and destination IP to define a flow. In the backbone networks, the traffic usually consists of two-tuple source-destination flows, and the aim of traffic engineering is to realize load balancing by minimizing maximum link utilization in the network. Many existing works on backbone networks [17]–[20] follow the same consideration.

Our work is different from existing works on the hybrid SDN. The existing works [21]–[28] consider a hybrid SDN, which consists of SDN switches and legacy network devices. In a hybrid SDN, each device only routes flows using a fixed routing mode: SDN switches use SDN routing, and legacy devices use OSPF routing. Fibbing [28] enables the SDN controller to control legacy switches by injecting crafted routing messages via OSPF. However, in our work, each switch is a deployable commercial SDN switch working at the hybrid OpenFlow/OSPF mode[3], and each flow can be routed with OpenFlow, OSPF, or joint OpenFlow and OSPF by the configuration dynamically decided by the controller.

The rest of the paper is organized as follows: Section II presents the background and the motivation of the paper by explaining the limitation of existing solutions and proposing research challenges. Section III overviews the main techniques and system design of HybridFlow and illustrates how HybridFlow works with one example. Section IV details three models of HybridFlow. In Section V, we compare the performance of HybridFlow with existing works and analyze the simulation results. In Section VI, we make further discussion about HybridFlow. We introduce the related work in Section VII. In Section VIII, we conclude the paper.

[2]In this paper, we consider OSPF as a representative traditional routing protocol. Our work can extend to other traditional routing protocols.

[3]In the rest paper, we use the hybrid mode short for the hybrid OpenFlow/OSPF mode.

II. Background and Motivation

In this section, we analyze the limitation of existing solutions and present research challenges.

A. Limitation of existing routing schemes

1) With the single controller: For routing each flow, the SDN control plane provides two basic functions: path calculation (using a routing algorithm) and path deployment (by deploying flow entries to install, update, or delete paths for flows). The default approach of SDN is to fine-grained calculate and establish a path for each flow. The controller installs flow entries into the corresponding switches to establish the path for a new flow; the controller observes the network change and updates the paths of existing flows to maintain good network performance. The single-controller control plane is not scalable since its limited processing ability is not enough to calculate and establish paths for many flows, especially when employing the per-flow path establishment for a large number of flows.

2) With the multiple controllers: Multi-controller control plane can enable the two functions, but its routing performance relies on the controller synchronization. Here we present some cases to show the impact of controller synchronization on network performance:

When synchronized network information arrive controllers at different time: To maintain good network performance (e.g., link load balancing), a controller usually makes a routing/rerouting decision for a flow based on the network status information, which consists of its local domain information from its controlled switches and the remote domain information synchronized from other controllers. Due to different conditions (e.g., distances between controllers and congestion of paths between controllers), the synchronized network information could arrive at controllers at different time. Thus the controllers could make decision based on out-of-date network information, leading to sub-optimal performances. In the worst case, some undesirable situations (e.g., forwarding loops and black holes) could occur when controllers operate based on the inconsistent network information among them [10], [29].

When a flow's path traverses multiple domains: If a flow's path traverses switches at multiple domains controlled by different controllers, establishing or changing the path needs the coordinated operation of these controllers. The coordinated operation significantly increases not only the complexity of network control but also the probability of incorrect network behaviors. For example, when updating a path, inappropriate order of flow entry update could result in undesired situations (e.g., forwarding loops [30]).

Extra resource consumption: The controller synchronization usually uses in-band networks, which are usually used to transmit traffic flows [31]. Hence, the controller synchronization not only consumes the extra CPU and memory resource from controllers but also the bandwidth from in-band networks. The resource consumption could affect other network operations.

To summarize, the major challenges when using multiple controllers in the SDN is to synchronize the state among controllers [10]. The state synchronization among controllers significantly degrades the network performance. Periodic synchronization is a widely used solution. However, frequent synchronizations may result in high synchronization overhead of controllers. If the controllers do not maintain consistent state, forwarding loops and black holes could occur. Most distributed SDN controller designs rely on consensus mechanisms such as Paxos (used by ONIX) [32] and Raft (used by ONOS) [7]. Panda et al. [33] propose the Simple Coordination Layer (SCL), which can turn a set of single SDN controllers into a distributed SDN system that achieves faster network response.

B. Design challenges

From the above analysis, we can see that the two types of existing routing solutions have their limitations. To avoid the negative impact of controller synchronization, an ideal solution is a simple but scalable routing scheme that employs the single controller to maintain good performance with low processing load. Thus, the main challenge is to reduce the controller processing load without degrading the network performance under a dynamic network.

In traditional networks, in order to achieve scalability, we can only depend on aggregate routing or OSPF routing and decentralized control. Flexible routing cannot be realized by these types of solutions. With the help of SDN, we can improve network performance by fine-grained per-flow routing and centralized control. However, it requires high communication and computation overhead at the controller. So, with hybrid routing method, we can achieve good network performance with the low communication overhead [34], [35].

Essentially, the network performance is maintained by dynamically rerouting flows during the network variation. The controller's load of rerouting flows depends on two factors: the number of processing flows and the operational overhead to update the path for each processing flow (e.g., how many flow entries should be generated by the controller for a path operation). Given the limited processing ability, the controller could be overwhelmed if it routes/reroutes too many flows. However, if the controller randomly reroutes some flows, good network performance cannot be maintained. Additionally, the fine-grained per-flow routing puts a heavy burden on the controller.

To maintain good network performance with few control overhead, we should consider four aspects to reroute flows: (1) when the controller reroutes flows, (2) which flows should be rerouted by the controller, (3) which new paths should be used for the rerouted flows, and (4) how the controller updates the flows' paths.

III. HybridFlow Overview

In this section, we first briefly explain HybridFlow's techniques to address the above research challenges, then overview HybridFlow's system structure, and finally use an example to illustrate how HybridFlow works.

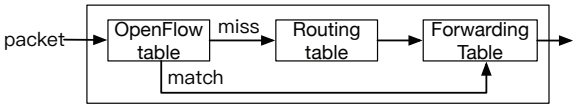


Fig. 2. Hybrid OpenFlow/OSPF routing mode in a commercial SDN switch (e.g., Brocade MLX-8 PE [16]).

A. HybridFlow techniques

HybridFlow employs two techniques to realize an efficient and scalable routing:

1) Hybrid Routing: Some SDN switches (e.g., Brocade MLX-8 PE [16]) support the hybrid OpenFlow/OSPF routing mode. Fig. 2 shows the packet process of the hybrid mode in an SDN switch. In the figure, both OpenFlow and OSPF can run at the same time, and the flow table has a higher priority than the traditional routing table. The flow table has a default entry with the lowest priority that sends each miss-matched packet to the traditional routing table. When a switch works under this mode, it first checks the flow table for its received packet. The OpenFlow mode uses the flow table to route flows. If a matched entry is found, the packet is processed based on the actions in the entry. Otherwise, the packet is further forward to the traditional routing table to route it using the destination-based entries. The OpenFlow table and OSPF routing table share the same physical memory in the switch. Based on the configuration, the physical memory can be dynamically logically divided into OpenFlow table and legacy routing table. The hybrid routing mode has been tested in real production networks (e.g., CHN-IX [36]). Some high-end commercial SDN switches may also support hybrid OpenFlow/OSPF mode. Moreover, existing works [34], [35] also consider the hybrid routing mode for designing switch for realizing scalable and optimal performance. Hybrid routing mode provides us an opportunity to route a flow in the network using the traditional routing and OpenFlow together. Using such a hybrid mode, we can address concern (4) by reducing the number of the controller's operation for path establishment/update/deletion.

2) Crucial Flow Rerouting: In a network, link congestions have significant impact on network performance but are rare events. In many cases, traditional routing protocols (e.g., OSPF) can provide a good network performance. Thus, we do not need to control each flow in the network all the time. We only need to selectively change the paths of some flows during the congestion, and other flows are always forwarded on their existing paths. Based on our analysis of real world traffic matrices, we present a metric called Variation Slope to select crucial links among all links in a network. The crucial links have high traffic load and are likely to experience congestion. Flows on the crucial links are recognized as *crucial flows*. We select new paths for crucial flows from pre-generated path-sets and reroute crucial flows to their new paths to achieve load balancing. This technique addresses concerns (1)-(3) by deciding the right time to reroute flows, reducing the number of rerouted flows, and reducing the overhead to calculate paths for the controller.

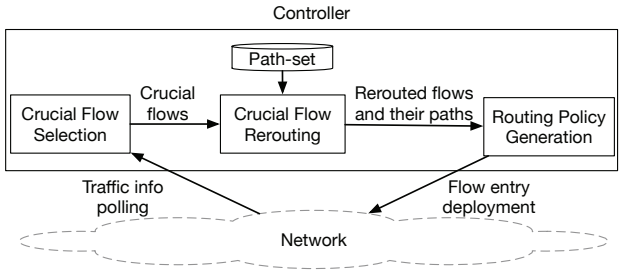


Fig. 3. HybridFlow structure.

B. Overview

Fig. 3 shows the HybridFlow's structure, which consists of three modules: Crucial Flow Selection, Crucial Flow Rerouting, and Routing Policy Generation. The first two modules realize the Crucial Flow Rerouting, and the Hybrid Routing is realized by the Routing Policy Generation module.

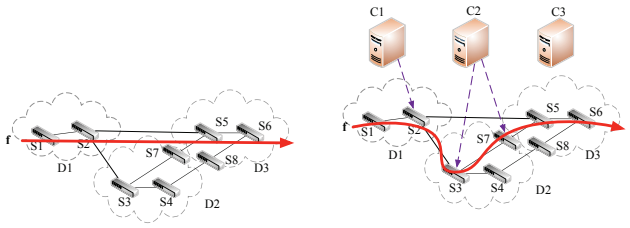
During the network initialization, we generate a set of paths for each source-destination switch pair, and each switch uses OSPF to generate destination-based routing entries in its routing table to route flows. The crucial link selection module periodically gets the information of flows using sFlow [37] or NetFlow [38] and calculates Variation Slope to decide the crucial links. The set of crucial links is sent to the rerouting module, and this module selects rerouting paths from pre-generated path-sets for crucial flows on the crucial links to reduce the crucial links' utilizations. We formulate the flow rerouting problem as an optimization problem and detail it in Section IV.

After receiving the new paths of crucial flows, the routing policy generation module compares each crucial flow's new path with the set of shortest paths from switches on the new path to this flow's destination. If the new path overlaps with some shortest paths, the flow can be forwarded on the shortest paths using destination-based entries in the routing table. The module only generates entries for switches, which are on the new path but are not on the shortest paths, to direct the flows towards the shortest paths.

C. An example

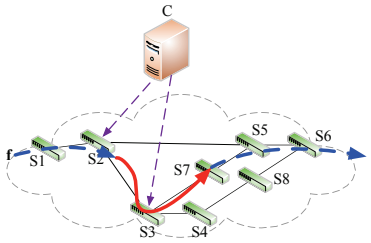
Fig. 4 shows an example that illustrates how to reroute a flow using an existing solution with multiple controllers and HybridFlow. Fig. 4(a), flow f from $S1$ to $S6$ is originally forwarded on it shorest path $S1 \rightarrow S2 \rightarrow S5 \rightarrow S6$. Fig. 4(b) shows rerouting flow f to path $S1 \rightarrow S2 \rightarrow S3 \rightarrow S7 \rightarrow S5 \rightarrow S6$ using per-flow routing and multiple controllers. In this figure, three controllers control three domains. Controller $C1$ first updates one flow entry on switch $S2$ and then notifies controller $C2$, which installs one flow entry on switches $S3$ and $S7$, respectively. Three flow entry operations and one controller notification are consumed.

Fig. 4(c) shows rerouting flow $f2$ using HybridFlow. In this figure, flow f is identified as a crucial flow to be rerouted. The controller identifies f's new path is overlapped with two existing shortest paths $p_{S1,S6}$ on $S1 \rightarrow S2$ and $p_{S7,S6}$ on $S7 \rightarrow S5 \rightarrow S6$. Thus, the controller respectively installs one flow



(a) Flow f is originally forwarded on its shortest path using OpenFlow.

(b) Rerouting flow f with multiple controllers and per-flow routing.



(c) Rerouting flow f with HybridFlow.

OpenFlow table

Priority	Source	Destination	Actions
high	S1	S6	towards S3
low	*	*	routing table lookup

Routing table

Destination	Next Hop
S6	S5

S2

OpenFlow table

Priority	Source	Destination	Actions
high	S1	S6	towards S7
low	*	*	routing table lookup

Routing table

Destination	Next Hop
S6	S2

S3

(d) Tables of $S2$ and $S3$ in (c).

Fig. 4. An example that changes the path of flow f using different routing schemes. Gray switches in (a) and (b) are under OpenFlow mode, and green switches in (c) work under the hybrid OpenFlow/OSPF mode. A red dash line denotes a path established by OpenFlow, and a blue dash line denotes a (partial) shortest path established by OSPF.

entry on switches $S2$ and $S3$ to stitches the two existing paths. The entry on switch $S2$ directs f towards switch $S3$, and the entry on switch $S3$ directs f towards switch $S7$. The flow is forwarded on its shortest paths before reaching $S2$ and after leaving $S7$.

Fig. 4(d) shows tables of $S2$ and $S3$ in Fig. 4(c). Note that if we do not install the entry on switch $S3$, $S3$ will direct f back to switch $S2$ since its routing table shows the next hop of the shortest path from itself to $S6$ is $S2$. Since the path-set of each flow is proactively generated, the entries for changing one path to another one can also proactively defined and stored in the controller. The related entries will be installed into related switches when rerouting flows.

IV. HybridFlow Design

In this section, we detail the three modules of HybridFlow.

A. System description

A network is $G = (V, E)$, where V is the set of switches, and E is the set of links between switches. The utilization of link e ($e \in E$) is C^e and cannot exceed its upper bound capacity C. In our problem, we only consider *feasible flows*. A feasible flow should satisfy two requirements: (1) there are at least two paths between the flow's source and destination, and (2) at least one path's length is two hops. If a flow does not satisfy the two requirements at the same time, we cannot reroute the flow to improve the network performance. For example, in Fig. 4, flows between $S1$ and $S2$ are not feasible flows in our problem.

TABLE I
NOTATIONS

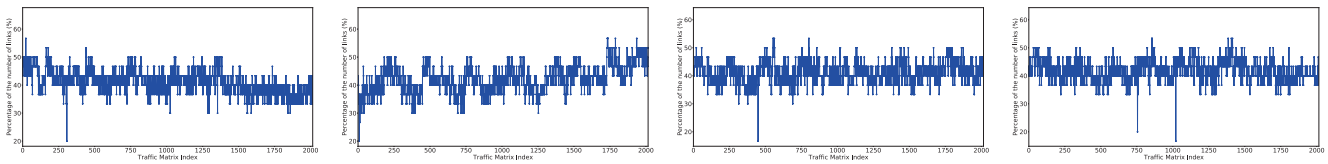
Notation	Meaning
V	the set of switches
E	the set of links e
C^e	the current link load of link e ($e \in E$)
C	the capacity of a link
T_v	the current flow table utilization of switch v ($v \in V$)
T	the capacity of flow table
$E^{crucial}$	the set of crucial links
$F^{crucial}$	the set of crucial flows that traverse link e ($e \in E^{crucial}$)
P_f	the set of paths for flow f
$\delta^{p,e}$	a binary constant that denotes if link e is on path p
$\xi^{p,v}$	a binary constant that denotes if switch v is on path p
ϕ_f^p	a binary constant that denotes if flow f is currently forwarded on path p
$\alpha^{p,v}$	a binary constant that denotes if a flow entry is installed in switch v for path p.
r_f	the flow rate of flow f
λ	a constant that denotes the different weights of two objectives

B. Crucial flow selection module

This module selects several flows as crucial flows in term of the impact on load balancing performance. This idea is inspired by analyzing link utilizations of real life topologies with traffic traces. Fig. 5 shows our analysis result of Abilene network dataset for four weeks [39]. Abilene network consists of 12 nodes and 30 directed links. Dataset includes network topology information (i.e., link connectivity, weights, and capacities) and measured traffic matrices in the time-slot of five minutes with a duration of six months. From the figure with 2016 time slots, we can see that in most time slots, only 50% links' loads exceed the average link load. Thus, to achieve load balancing, we only need to reduce the load of these links by rerouting some flows from these links to other lowly utilized links, and other flows can be kept forwarding without any change.

In this module, we first select several links as the crucial links using our proposed metric named Variation Slope and then select crucial flows from these crucial links. Specifically, the set of crucial links is denoted as $E^{crucial}$. If a flow's path traverses a crucial link, it is a crucial flow. The set of all crucial flows that traverse link e ($e \in E^{crucial}$) is $F^{crucial}$. Each feasible flow f ($f \in F^{crucial}$) is described as $f = \{r_f, src_f, dest_f, P_f\}$, where r_f, src_f, $dest_f$, and P_f are the rate, source switch, destination switch, and the candidate path-set of flow f. The rate of each flow can be dynamically obtained from switches using sFlow [37] or NetFlow [38].

1) Calculating Variation Slopes: We first explain how to calculate the Variation Slope (VS) and then explain why



(a) Week one (b) Week two (c) Week three (d) Week four

Fig. 5. Ratio of the number of links that exceed the average link load to the total number of links when using OSPF on Abilene network dataset [39].

this metric can help us to find crucial links. Given the network $G = (V, E)$, we can have the combination of different amount of links. We use $LC(i)$ to denote the combination of i $(1 \leq i \leq |E|)$ links. In $LC(i)$, an element is denoted as $lc(i) = \{lc(i)^{flow\ amount}, lc(i)^{load}, lc(i)^{flows}\}$, where $lc(i)^{flow\ amount}$ denotes the amount of unique flows of this link combination, $lc(i)^{load}$ denotes the total load of the unique flows of this link combination, and $lc(i)^{flows}$ includes all unique flows of this link combination. The amount of unique flows of i different links is the union of flows in the i links. For example, if link e_1 has flows f_1 and f_2, and link e_2 has flows f_2 and f_3, the unique flows of the combination of links e_1 and e_2 are flows f_1, f_2, and f_3, and the amount of unique flows is 3. The elements in $LC(i)$ are sorted in the decreasing order of the amount of unique flows. If elements in $LC(i)$ have the same amount of unique flows, they are further sorted in the decreasing order of the total load of the unique flows. From $LC(i)$ and $LC(i+1)$, we find the first element $lc(i) = \{lc(i)^{flow\ amount}, lc(i)^{load}, lc(i)^{flows}\}$ and $lc(i+1) = \{lc(i+1)^{flow\ amount}, lc(i+1)^{load}, lc(i+1)^{flows}\}$. VS(i+1) is calculated as follows:

$$\text{VS(i+1)} = \frac{lc(i+1)^{flow\ amount} - lc(i)^{flow\ amount}}{lc(i+1)^{load} - lc(i)^{load}}$$

A VS denotes the amount of flows in one unit of traffic load. This metric describes the variation trend of flow amount versus the variation trend of flow amount load. If the metric increases, it means one unit of traffic load includes more flows. Recall our goal is to achieve load balancing of the network. A high VS implies that for a given traffic load, we can have more flows to reroute. That is, we have a finer granularity to reroute the traffic and to achieve better load balancing in this case. If both VS(i)<VS(i+1) and VS(i+2)<VS(i+1) are satisfied, VS(i+1) is a local extreme point among the neighbor VSs. Thus, the $lc(i+1)$ is the combination of crucial links, and all flows in $lc(i+1)^{flows}$ are crucial flows.

2) Selecting crucial flows: Algorithm 1 describes how this module works. We generate the Variation Slope and do the test until we find the extreme point i. The algorithm stops and returns flows in $lc(i)^{flows}$.

C. Crucial flow rerouting module

The module calculates paths for crucial flows to achieve load balancing by solving an optimization problem. In this subsection, we introduce design variables, constraints, and the objective function, formulate our optimization problem, and present an efficient solution to solve the problem.

Algorithm 1 CrucialFlowSelection()

Input: Link set E, traffic matrix $\mathbf{M}$;
Output: Crucial flows set $F^{crucial}$;

```
for i = 2 : |E| do
    calculate VS(i-1), calculate VS(i), calculate VS(i+1);
    if VS(i-1)<VS(i) and VS(i+1)< VS(i) then
        F^crucial = lc(i)^flows;
        break;
    end if
end for
return F^crucial;
```

1) Design variables: For path $p \in P_f$, $\delta^{p,e} = 1$ denotes link e on p, otherwise $\delta^{p,e} = 0$; $\xi^{p,v} = 1$ denotes switch $v \in V$ on p, otherwise $\xi^{p,v} = 0$. For flow f $(f \in F^{crucial})$, if it is currently forwarded on p, we have $\phi_f^p = 1$, otherwise $\phi_f^p = 0$; if it is selected to reroute on a path p on the current moment, we have $y_f^p = 1$, otherwise $y_f^p = 0$.

Using the hybrid routing mode, we only need to install flow entries on a selected number of switches on a path to establish the path. For example, in Fig. 4, we only install one flow entry on switches $S2$ and $S3$, respectively, to change flow $f2$'s path. For path p, if we install a flow entry in switch v, we have $\alpha^{p,v} = 1$, otherwise $\alpha^{p,v} = 0$. $\alpha^{p,v}$ depends on the overlapping of path p with the shortest path of the corresponding flow. The default relationship among $\xi^{p,v}$ and $\alpha^{p,v}$ is $\xi^{p,v} * \alpha^{p,v} = \alpha^{p,v}$.

The path-set for each flow is generated at the beginning of network initialization, and $\alpha^{p,v}$ for paths in the path-set of each flow is pre-calculated and does not change during the whole optimization process. The controller installs flow entries to establish the hybrid routing based on the results of the Routing Policy Generation module.

2) Constraints: This problem has three constraints.

Path constraint: One flow can be only forwarded on one path. That is

$$\sum_{p \in P_f} y_f^p = 1, \forall f \in F^{crucial}. \tag{1}$$

Each path is selected from a given path-set, and thus we do not need to bound the length of paths.

Link constraint: If we change flow f $(f \in F^{crucial})$ from its old path to its new path, the link load of links on the old path could decrease, and the link load of links on the new path could increase. Link e's current load is C^e, By changing the paths of flows in $F^{crucial}$, the increasing utilization of link e is $\sum_{f \in F^{crucial}} \sum_{p \in P_f} (\delta^{p,e} * r_f * y_f^p)$. Before each traffic matrix comes, each flow is default assigned on its shortest

path, and thus we do not need to consider the decrease of the link utilization and flow table utilization here. The link load is formulated as follows:

$$load^e = \sum_{f \in F^{crucial}} \sum_{p \in P_f} (\delta^{p,e} * r_f * y_f^p) + C^e.$$

Each link's load should not exceed link capacity C:

$$load^e \leq C, \forall e \in E. \tag{2}$$

Note that if the path of f ($f \in F^{crucial}$) does not change, $y_f^p = \phi_f^p$, and the utilizations of related links do not change.

Flow table constraint: When we change f ($f \in F^{crucial}$) from its shortest path to its OpenFlow path, the controller needs to install entries in flow tables of switches on the new path. Thus, the utilization of flow tables of switches on the OpenFlow path could increase. By changing the paths of flows in $F^{crucial}$, the increasing utilization of switch v is $\sum_{f \in F^{crucial}} \sum_{p \in P_f} (\xi^{p,v} * \alpha^{p,v} * y_f^p)$. Switch v's current flow table utilization is T_v, and its flow table's utilization cannot exceed its flow table capacity T. Thus, we have:

$$\sum_{f \in F^{crucial}} \sum_{p \in P_f} (\xi^{p,v} * \alpha^{p,v} * y_f^p) + T_v \leq T, v \in V. \tag{3}$$

Note that T_v changes over time since some entries in the flow table are removed when they timeout.

3) Objective function: Conventionally, the network performance of a WAN is measured by the link load balancing performance. However, only considering this metric is not enough since it could lead to some undesirable situations. For example, a crucial flow on a crucial link is rerouted on a very long path to achieve a good load balancing performance. The rerouting operation increases traffic load on the entire network and impacts the routing of other flows. Thus, our objective considers two metrics: (1) the link load balancing performance and (2) the total link load in the network. Let u_e denote the utilization of link e, where $u_e = load^e/C$. The link load balancing performance is measured by the maximum utilization of links in the network and is formulated as follows:

$$obj_1 = \max_{e \in E}(u^e).$$

The total link load of links in the network is formulated as follows:

$$obj_2 = \sum_{e \in E} load^e.$$

Thus, we have the objective function as follows:

$$obj = obj_1 + \lambda * obj_2 = \max_{e \in E}(u^e) + \lambda * \sum_{e \in E} load^e,$$

where λ ($1 < \lambda < 0$) is a constant that denotes the different weight of two objectives. In our problem formulation, the first objective has the first priority, while the second one has the least importance. We evaluated the results of two objectives, and then carefully designed the λ to let the two objectives have different priority. The specific value of λ is 10^{-9}.

4) Problem formulation: Given the controller's processing ability, our problem's goal is to achieve the optimal load balancing performance by rationally selecting some crucial flows and rerouting them on their new paths. The problem is formulated as follows:

$$\min_{y}\{\max_{e \in E}(u^e) + \lambda * \sum_{e \in E} load^e\} \tag{P}$$

subject to

$$\text{Eqs. (1), (2), (3)}$$

$$y_f^p \in \{0,1\},\ p \in P_f,\ f \in F^{crucial},\ e \in E,\ v \in V,$$

where $\{y_f^p\}$ are binary design variables, $\delta^{p,e}$, r_f, $\xi^{p,v}$, $\alpha^{p,v}$, and ϕ_f^p are given integer constants.

5) Problem reformulation: The above problem is a binary linear programming with high computation complexity. To efficiently solve the problem, we can transform the problem into a simple one by using the linear programming relaxation technique. More specifically, we can relax binary variables y_f^p to

$$y_f^p \in [0,1],\ p \in P_f, f \in F^{crucial} \tag{4}$$

and change the objective as follows:

$$\min_{y,\,\omega}\{\omega + \lambda * \sum_{e \in E} load^e\}$$

where ω must satisfy

$$u^e \leq \omega,\ \forall e \in E. \tag{5}$$

The problem is reformulated as follows:

$$\min_{y,\,\omega}\{\omega + \lambda * \sum_{e \in E} load^e\} \tag{P'}$$

subject to

$$\text{Eqs. (1), (2), (3), (4), (5)},\ e \in E,\ v \in V,$$

where $\{y_f^p\}$ are binary design variables, ω is a linear variable, $\delta^{p,e}$, r_f, $\xi^{p,v}$, $\alpha^{p,v}$, and ϕ_f^p are given integer constants. It is worth remarking that the new formulated problem is a linear program.

6) Heuristic Solution: We propose a heuristic algorithm to solve the problem. The idea of our algorithm is to first select a new path for a flow on a crucial link following the descending order of selection probability and then test whether the path satisfies four constraints. If yes, the path is used to reroute the flow, and we update the utilization of links on the old path and new path, and the flow table utilization of switches on the old path and new path; otherwise, a new path is tested. The procedure is terminated until all flows in $F^{crucial}$ are tested.

Details of the heuristic algorithm are summarized in Algorithm 2. In line 1, we generate vector $\mathcal{Y}^*$ by solving the linear programming relaxation of problem (P') and sorting the results in the descending order. The sorting operation enables us to test the path variables based on their probabilities. From line 2 to line 22, we use our customized rounding technique to find a new path for each crucial flow by sequentially testing each path $y_k \in \mathcal{Y}^*$. In line 3, we find y_k's crucial flow f, path p, and flow f's old path p_0. Lines 4-6 guarantee that we do not test a crucial flow if it is already rerouted. In lines 7-11, we test if placing flow f on path p exceeds the link capacity. In lines 12-16, we ensure that placing flow f on path p will not

Algorithm 2 CrucialFlowRerouting()

Input: $E^{crucial}$: the set of crucial links;
$F^{crucial}$: the set of flows on crucial links;
P: the path-sets of flows on crucial links;
Output: $\mathcal{Y}$: the selected paths of rerouted flows $\{f \in F^{crucial} | \mathcal{Y}_f\}$, where $\mathcal{Y}_f = \{p \in P_f | y_f^p = 1\}$;

1: generate vector $\mathcal{Y}^* = \{y_k, k \in [1, \sum_{e \in E^{crucial}} \sum_{f \in F^{crucial}} |P_f|]\}$;
2: **for** $y_k \in \mathcal{Y}^*$ **do**
3: get y_k's flow f, path p, and flow f's old path p_0;
4: **if** $f \in \mathcal{X}$ **then** // test Equation (1)
5: continue;
6: **end if**
7: **for** $e \in p$ **do** // test Equation (2)
8: **if** $C^e + \delta^{p,e} * r_f * y_f^p > C$ **then**
9: go to line 2;
10: **end if**
11: **end for**
12: **for** $v \in p$ **do** // test Equation (3)
13: **if** $T_v + \alpha^{p,v} * y_f^p > T$ **then**
14: go to line 2;
15: **end if**
16: **end for**
17: $\mathcal{Y}_f = \{y_f^p = 1\}, \mathcal{Y} = \mathcal{Y} \cup \mathcal{Y}_f$, update link utilization for $e \in \{p, p_0\}$;
18: update flow table utilization for $v \in \{p, p_0\}$, remove f from $F^{crucial}$;
19: **if** $F^{crucial} == \emptyset$ **then**
20: break;
21: **end if**
22: **end for**
23: return $\mathcal{X}$, $\mathcal{Y}$

Algorithm 3 RoutingPolicyGeneration()

Input: $\mathcal{Y}$: the selected paths of rerouted flows $\{f \in F^{crucial} | \mathcal{Y}_f\}$;
P^{dest_f}: the set of shortest paths with destination $dest_f$;
Output: $\mathcal{Z}$: the set of generated flow entries;

1: $\mathcal{Z} = \emptyset$;
2: **for** $y \in \mathcal{Y}$ **do**
3: get y's flow f, $p_f = y$, flow f's source src_f, and destination $dest_f$;
4: **for** $p_i \in P^{dest_f}$ **do**
5: $P_i^{overlap} = p_f \cap p_i = \{p_i^{overlap_1} : v_{s_1} \rightarrow ... \rightarrow v_{d_1}, ..., p_i^{overlap_j} : v_{s_j} \rightarrow ... \rightarrow v_{d_j}\}$;
6: **if** $P_i^{overlap} == p_f$ **then**
7: go to line 1;
8: **else if** $P_i^{overlap} == \emptyset$ **then**
9: continue;
10: **end if**
11: remove paths in $P_i^{overlap}$ from p_f except v_{d_1}, v_{s_2}, ..., $v_{d_{j-1}}$, v_{s_j};
12: **end for**
13: **for** $\{v_{i1} \rightarrow v_{i2}\} \in p_f$ **do**
14: generate flow entry $z(v_{i1}) = \{$priority: $high$, match field: source=src_f, destination=$dest_f$, action: towards $v_{i2}\}$ on v_{i1};
15: $\mathcal{Z} = \mathcal{Z} \cup z(v_{i1})$;
16: **end for**
17: **end for**
18: return $\mathcal{Z}$

exceed the flow table capacity. If all the three tests are passed, this path is a new path for flow f. Thus, in lines 17-18, we place flow f on path p, update the utilization of related links and flow tables, and remove flow f from $F^{crucial}$. In lines 19-21, if all crucial flows are rerouted, the algorithm jumps out of the iterations. In line 23, the algorithm returns the result and stops.

D. Routing policy generation module

Upon receiving the rerouted flow and their new paths from the crucial flow rerouting module, this module translates the path into deployable routing policies that consist of OpenFlow flow entries and OSPF routing rules. The basic idea is to generate a few OpenFlow flow entries for each rerouted path to stitch several existing (partial) shortest paths based on OSPF together. Algorithm 3 shows the details of this module. In line 1, we initialize the set of generated flow entries, $\mathcal{Z}$, as an empty set. In line 2, this algorithm runs for each rerouted path $y \in \mathcal{Y}$. In line 3, we find path y's crucial flow f, f's source node src_f and destination node $dest_f$. From lines 4 to 12, we decide which part of path p_f should be realized by OpenFlow flow entries. Given P^{dest_f}, the set of shortest paths with destination $dest_f$, we compare each of the paths in P^{dest_f} with p_f. In line 5, we get the set of overlapped paths $P_i^{overlap}$ between $p_i \in P^{dest_f}$ and p_f. If $P_i^{overlap}$ is same to p_f, p_i and p_f are same, and we do not need to install flow entries for p_f (Lines 6-7); if $P_i^{overlap}$ does not exist, we should test the next path in P^{dest_f} (Lines 8-9). If p_i and p_f overlap with some partial paths, we do not need to install flow entries for the overlapped paths to forward flow f since the overlapped paths are already established with OSPF. Thus, in line 11, we remove the overlapped paths from p_f. Note that some specific nodes v_{d_1}, v_{s_2}, ..., $v_{d_{j-1}}$, v_{s_j} are not removed from path p_f. The specific nodes are either source or destination nodes of the overlapped paths between p_i and p_f, and we keep them in p_f because we need to install OpenFlow flow entries in these switches to stitch the overlapped paths together. After comparing paths in P^{dest_f} with p_f and removing the overlapped paths from p_f, we have a new p_f that should be realized with OpenFlow flow entries. In lines 13-16, we generate flow entries for p_f and put the entries into $\mathcal{Z}$. The algorithm terminates when all paths in $\mathcal{Y}$ are tested, and $\mathcal{Z}$ are returned to install flow entries into switches.

Fig. 6 shows an example to illustrate how the routing policy generation module works. In this figure, given an old path $p1$ and a new path p, we first generate the overlapped path and then remove the overlapped path from p to have the OpenFlow path. To stitch the overlapped path together, we then generate flow entries and installed them on related switches.

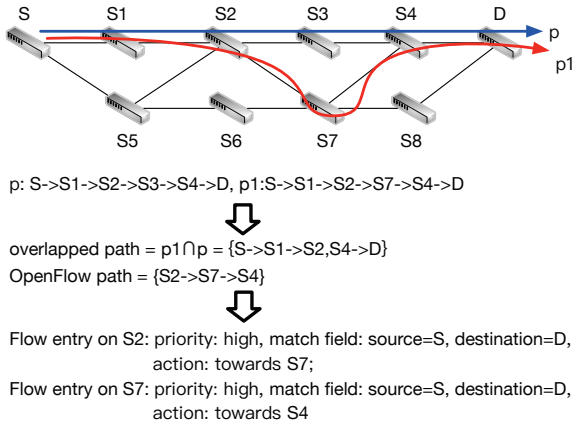


Fig. 6. An example illustrating how the routing policy generation module works.

V. Evaluation

A. Comparison algorithms

We compare HybridFlow with the following schemes:

Optimal per-flow routing (Optimal): This scheme considers all links in the network as crucial links, and thus all flows can be rerouted to achieve the optimal load balancing performance.

Open Shortest Path First (OSPF): all flows are rerouted on their weighted shortest paths calculated by OSPF.

Equal-Cost Multi-Path (ECMP): if there are multiple shortest paths between a pair of source and destination switches, each switch on a path equally splits traffic among all available next hops corresponding to the shortest paths.

Crucial Flow Hybrid Routing (CFHR): it is the optimal solution of problem (P'). We solve it with an advanced solver GUROBI [40].

HybridFlow: flows are first routed by OSPF. Every time slot, a set of crucial flows are rerouted to achieve load balancing. The rerouting algorithm are detailed in Section IV.

HybridFlow(+1): this scheme is similar to HybridFlow but selects one more crucial link than HybridFlow. This solution is used to show that HybridFlow selects enough crucial flows to achieve a good performance.

Maximizing the Utilization of links (MaxU): this scheme is similar to HybridFlow except that it selects crucial links with the purpose of maximizing these links' total utilizations. By selecting the same number of links as HybridFlow does, this solution evaluates the effectiveness of Variation Slope in term of selecting crucial flows.

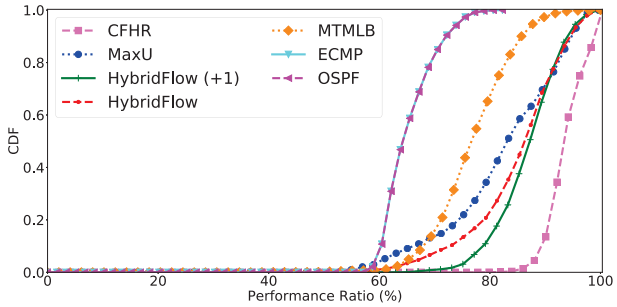
Multiple Traffic Matrix Load Balancing (MTMLB) [41]: this scheme mainly focuses on the hybrid routing that uses explicit routing and destination-based routing together. The majority of node pairs are established by destination-based routing; several key node pairs are carefully selected and their paths are established using explicit routing. Its objective is to minimize the maximum link utilization of the worst case among multiple traffic matrices.

B. Simulation setup

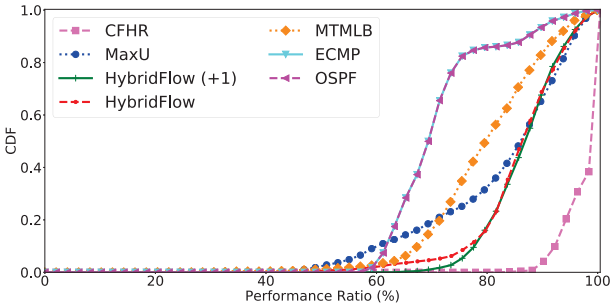
We evaluate HybridFlow's performance with a real network topology. We use a publicly available Abilene network dataset [39] in our simulation. Abilene network is an educational backbone network in North America and consists of 12 switches and 30 links. The dataset records the traffic matrix of 12 switches in a time slot of every five minutes for a duration of six months. The topology we used is bidirectional topology, and the input of our solution is a traffic matrix, which consists of $N * N$ elements, where N denotes the number of nodes in the network. Thus, the proposed solution supports symmetric routing through the networking infrastructure. The dataset also provides the OSPF weights of all the links, and we use the weights to find a set of 5 paths between every pair of switches. We choose 8064 traffic matrices in the first two weeks from March 1st, 2004 to March 14th, 2004 and the last two weeks from July 17th, 2004 to July 30th, 2004 as our dataset. The traffic of the first week holds relatively steady, but traffic of the second week exhibits a high fluctuation. The traffic of the rest two weeks We use the performance ratio to evaluate the HybridFlow's performance on load balancing, total link load, and the controller's processing overhead: $PR = \frac{P_{\text{Optimal}}}{P_{\text{scheme}}}$, where P_{optimal} denotes the result of Optimal that reroutes any flows, and P_{scheme} denotes the result of a special scheme. $PR = 1$ means that HybridFlow performs as good as Optimal. A lower ratio indicates that the HybridFlow's performance is far away from that of Optimal.

C. Simulation results

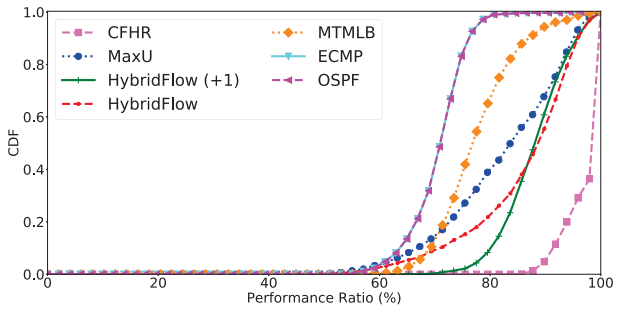
1) Load balancing performance: Fig. 7 shows load balancing performance of different schemes of four-week traffic matrix. In Fig. 7(a), ECMP and OSPF perform the worst because they always evenly forwards flows on the shortest paths without considering traffic variation, and its average performance ratio is only 66%. MaxU's performance varies significantly, and in the worst case, its performance ratio lows to 55%. In contrast, HybridFlow's average performance ratio is 86%, and in some cases, HybridFlow's performance can reach the Optimal's performance because of reasonably selecting and rerouting crucial flows. In other words, Variation Slope is a better metric than link utilization in terms of selecting crucial flows. HybridFlow(+1)'s results are almost covered by HybridFlow's except several time slots, and its average performance ratio is only 1% higher than HybridFlow's. Thus, HybridFlow selects enough crucial flows for rerouting to maintain good network performance. The performance of MTMLB is 10% worse than HybridFlow because MTMLB focus on minimizing the maximum utilization for the worst case of multiple traffic matrices. CFHR performs well, and its median is 93%. Fig. 7(b) shows the results similar to Fig. 7(a). In Fig. 7(b), ECMP and OSPF perform the worst, and the average performance ratio of ECMP and OSPF are only 71%. MaxU's performance varies significantly. In contrast, HybridFlow's average performance ratio is 87%, and its performance appears to be mostly constant. HybridFlow(+1)'s results are almost covered by HybridFlow's, and HybridFlow(+1)'s average performance ratio is only 0.5% higher than HybridFlow's.



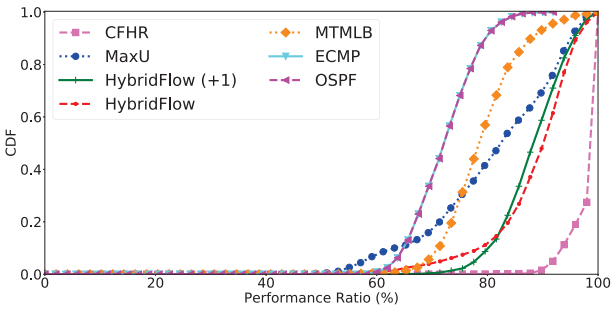
(a) CDF of performance ratio of traffic matrix week one.



(b) CDF of performance ratio of traffic matrix week two.

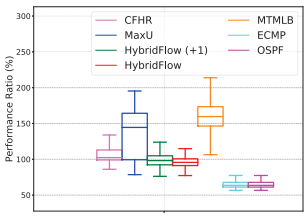


(c) CDF of performance ratio of traffic matrix week three.

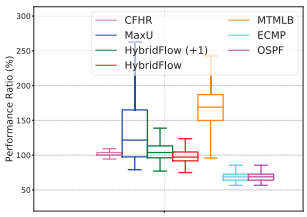


(d) CDF of performance ratio of traffic matrix week four.

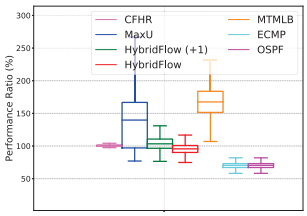
Fig. 7. Load balancing performance.



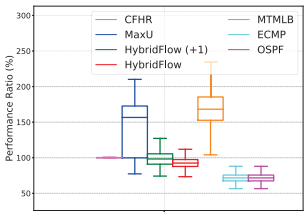
(a) Total link load of traffic matrix week one.



(b) Total link load of traffic matrix week two.



(c) Total link load of traffic matrix week three.



(d) Total link load of traffic matrix week four.

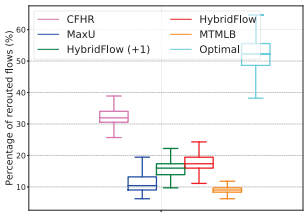
Fig. 8. Total link load performance. The lower, the better.

MTMLB performs 7% lower to HybridFlow. The performance of CFHR is similar to Optimal. As for the results in Fig. 7(c) and Fig. 7(d), HybridFlow's performance holds steady. HybridFlow's medians are 89% and 90% separately, about 10% higher than MTMLB on average. The performance of MaxU still varies greatly, and its worst is lower than ECMP and OSPF. CFHR still performs the best in both of the two weeks.

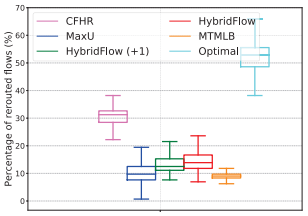
Note that the traffic matrix comes from Abilene, a real network, and is measured every five minutes. The heuristic algorithm solves the problem less than one second. Thus, the time required to solve the heuristic is far less than the length of the time slot. We add the results of the integer model in all simulations. HybridFlow's performance is about 8% lower to near Optimal's but only uses 15% of time compared with the integer model.

2) Total link load performance: Fig. 8 shows the total link load performance of different schemes of four-week traffic matrix. In Fig. 8(a), ECMP and OSPF perform the best since they only forwards traffic on their shortest paths, and thus overall traffic in the network is minimized at the cost of unbalanced traffic distribution of links as shown in Fig. 7. HybridFlow maintains a comparable performance to the optimal. Its median performance ratio is only 96%, and 6% lower than CFHR. MaxU performs worse than HybridFlow does since it reroutes less flows, which will be shown in Fig. 9. HybridFlow(+1) performs similar to HybridFlow's. MTMLB performs the worst because it reroutes the least flows. HybridFlow works for steady and varying traffic scenarios. The traffic of the second week exhibits higher fluctuation than the first week's, but Fig. 8(b) shows the results similar to Fig. 8(a). In Fig. 8(b), ECMP and OSPF's performance degrades, but the other schemes perform better. HybridFlow's median performance ratio is only 98%. The performance in Fig. 8(c) and Fig. 8(d) is close to Fig. 8(b), and the medians are 98% and 97% separately.

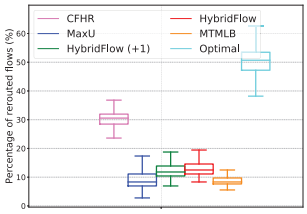
3) Number of rerouted flows: We present Fig. 9 to analyze the performance of different schemes under four-week traffic matrix. The network has 144 flows, and 12 flows are transmitted among a switch self and cannot be rerouted. Thus, Optimal can reroute at most 132 of 144 flows, which is 91.6% of all flows. Fig. 9(a) shows the result of week one. In this figure, HybridFlow reroutes 11-23% of all flows,



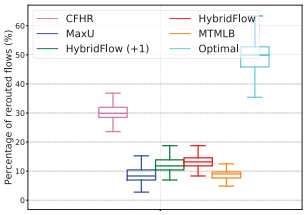
(a) The ratio of the number of rerouted flows to the total number of flows of traffic matrix week one.



(b) The ratio of the number of rerouted flows to the total number of flows of traffic matrix week two.

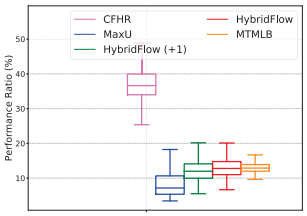


(c) The ratio of the number of rerouted flows to the total number of flows of traffic matrix week three.

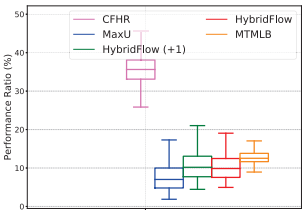


(d) The ratio of the number of rerouted flows to the total number of flows of traffic matrix week four.

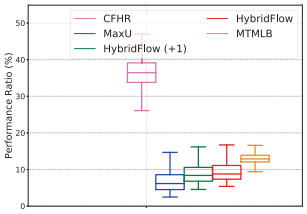
Fig. 9. Performance of rerouted flows. The lower, the better.



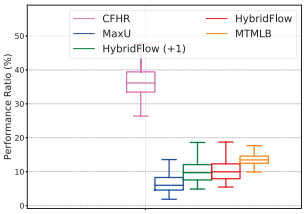
(a) Number of control messages for flow rerouting of traffic matrix week one.



(b) Number of control messages for flow rerouting of traffic matrix week two.



(c) Number of control messages for flow rerouting of traffic matrix week three.



(d) Number of control messages for flow rerouting of traffic matrix week four.

Fig. 10. Controller's processing overhead. The lower, the better.

HybridFlow(+1) reroutes 10-22% of all flows, MaxU reroutes 7-19% of all flows, Optimal reroutes 38-65% of all flows, CFHR reroutes 26-39% of all flows, and MTMLB 7-12% of all flows. MaxU and MTMLB reroute less flows, but their load balancing performance fluctuates significantly, as shown in Fig. 7. The median of HybridFlow and Optimal is 18% and 52%. Compared with Optimal, HybridFlow reroutes 34% less flows but achieves a compatible load balancing performance to Optimal. Fig. 9(b), Fig. 9(c), and Fig. 9(d) show the results similar to Fig. 9(a).

4) Controller's rerouting overhead: Fig. 10 shows the controller's processing overhead of four-week traffic matrix. In Fig. 10(a), compared with Optimal, HybridFlow reduces the number of control message from 7% to 20% with a median at 13%, HybridFlow(+1) reduces the number of control message from 6% to 20% with a median at 12%, and MaxU reduces the number of control message from 5% to 18% with a median at 8.5%. MaxU performs better than HybridFlow does since it reroutes less flows, while HybridFlow(+1)'s performance is slightly better than HybridFlow's since it reroutes less flows. CFHR reduces the number of control message from 25% to 49% with a median at 37%, while MTMLB's median is similar to HybridFlow, but it performs worse in load balancing performance. Fig. 10(b), Fig. 10(c), and Fig. 10(d) show the results similar to Fig. 10(a). HybridFlow's good performance comes from two reasons. First, HybridFlow reroutes less flows than Optimal does, as shown in Fig. 9. Second, HybridFlow uses the hybrid routing mode, which consumes less flow entries than the default per-flow routing to deploy the same path.

5) Sensitivity of λ parameter: Fig. 11 shows the load balancing performance of HybridFlow under different λ parameter. The specific value of λ is 10^{-9}. Thus, we compare it with the value of 10^{-8} and 10^{-10}. Simulation result shows that our HybridFlow is not sensitive to the λ parameter.

VI. Discussion

A. The extension of HybridFlow

We can adapt our solution in two scenarios: for hybrid OpenFlow/OSPF devices and for OpenFlow-only devices. For the hybrid OpenFlow/OSPF devices, we use the solution mentioned in our paper; for the OpenFlow-only devices, we use a dedicated table to enable prefix-based forwarding and then pre-configure some entries in the table to realize the shortest paths, which are globally computed at the controller just during the network initialization using OSPF. These shortest path-based entries are assigned with low priority, and other entries have high priority. Flows are forwarded based on the low-priority shortest path-based entries unless high-priority entries are deployed. Thus, we can realize hybrid routing in OpenFlow-only devices.

B. How to cope with link weight change

In SDN, the link weight change is usually triggered by the controller to change flows' paths, and the controller can monitor the status of switches to detect unexpected failures. Thus, the routing table change can be identified by the controller. Typically, there are two methods to cope with this problem. The first one is to recalculate the paths in the network globally. It works but costs a long time. The second one is the fast-recovery method. We can pre-calculate several backup paths to deal with the situation when unexpected links or nodes failure happen.

VII. Related Work

A. Scalable routing for SDN

Because of the increasing demand and the shortage of the resources, it is of vital importance to focus on load balancing issues to manage the incoming traffic and resources in order to improve the network performance [42]. Many effort

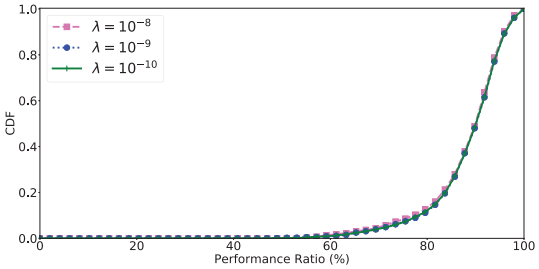


Fig. 11. The load balancing performance of HybridFlow under different λ parameter.

have been conducted to design scalable routing schemes to reduce the controller's operation load for routing/rerouting flows. JumpFlow [43] solves the problems with a source-routing based solution. Hedera [44] focuses on data center networks and achieves load balancing by identifying and rerouting large flows. Zhang et al. [45] use a similar hybrid routing but with proactively generated routing policies for given traffic matrix. OpenState [46] and FOCUS [47] propose to reduce the controller's load by offloading some control functions to SDN switches. [23], [24], [48] investigate traffic engineering (TE) in hybrid SDN. Ren et al. [49] consider the minimization of maximum link utilization and TCAM resource limitation in TE. Machine learning has been used to improve the performance of backbone networks [18], [20]. SHELL [50] and HULA [51] propose to realize stateless load balancing using P4.

B. Hybrid SDN

Hybrid SDN has been an important research for several years [52]. The research of hybrid SDN can be divided into two main categories [53]. The first category is to upgrade legacy networking devices to SDN switches in a legacy network. In this way, hybrid SDN provides a transitional period for SDN adoption as it reduces the deployment cost, mitigates the impact of other technical constraints, and enables SDN-like control over legacy network infrastructure [54], [55]. [21], [22], [56] are proposed to realize hybrid SDN. Fibbing [28] designs a centralized control over distributed IP routing to control legacy switches. CoVisor [57] enables to deploy multiple control applications for different controllers into a single network. Guo et al. [27] consider the impact of the control plane deployment on the hybrid SDN and propose to jointly deploy SDN switches and their controllers for hybrid SDNs. Cheng et al. [58] propose to monitor traffic in a hybrid SDN by collecting the load of several important links on the SDN switches and estimating the link load of the rest legacy switches. The second category is to employ hybrid SDN/legacy routing mode enabled by high-end SDN devices (e.g., Brocade MLX-8 PE) in a network. This hybrid routing can route flows using legacy routing protocols or SDN routing protocols [53]. Our proposed HybridFlow falls in the second category. Xu et al. [34] propose dedicated hybrid switches to meet both the scalability and the performance in SDN.

VIII. Conclusion

In this paper, we propose HybridFlow, a scalable routing solution to achieve a good load balancing performance with low control overhead. HybridFlow routes a majority of flows with their shortest paths based on OSPF, dynamically selects a few crucial flows, and reroutes these flows to their new paths, which are composed of OpenFlow path and partial shortest paths based on OSPF. The simulation results show that HybridFlow maintains good network performance with few control overhead by reducing the number of rerouted flows and the number of the controller's operations for flow rerouting.

References

[1] S. Jain, A. Kumar, and et al., "B4: Experience with a globally-deployed software defined WAN," in *ACM SIGCOMM'13*.
[2] C.-Y. Hong, S. Kandula, R. Mahajan, and et al., "Achieving high utilization with software-driven wan," in *ACM SIGCOMM'13*.
[3] "As providers push NFV/SDN, 3 key issues remain," https://www.rcrwireless.com/20170423/opinion/readerforum/reader-forum-nfv-sdn-issues-tag10, Nov. 2020.
[4] "The computer networks behind the social network," https://conferences.sigcomm.org/events/apnet2019/program.html, Nov. 2020.
[5] "ONOS controller," https://onosproject.org, Nov. 2020.
[6] "OpenDayLight controller," https://www.opendaylight.org, Nov. 2020.
[7] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX ATC'14*.
[8] J. Xie, D. Guo, and et al., "Cutting long-tail latency of routing response in software defined networks," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 3, pp. 384–396, 2018.
[9] A. Ksentini, M. Bagaa, and et al., "On using bargaining game for optimal placement of SDN controllers," in *IEEE ICC'16*.
[10] Z. Guo, M. Su, and et al., "Improving the performance of load balancing in software-defined networks through load variance-based synchronization," *Elsevier Comput. Networks*, vol. 68, pp. 95–109, 2014.
[11] N. Gude and et al., "NOX: Towards an operating system for networks," *Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, 2008.
[12] T. Wang and et al., "Bwmanager: Mitigating denial of service attacks in software-defined networks through bandwidth prediction," *IEEE Trans. Netw. Serv. Manag.*, vol. 15, no. 4, pp. 1235–1248, 2018.
[13] A. Azzouni and G. Pujolle, "Neutm: A neural network-based framework for traffic matrix prediction in sdn," in *IEEE/IFIP NOMS'18*, 2018.
[14] J. Zhang, K. Xi, and H. J. Chao, "Load balancing in IP networks using generalized destination-based multipath routing," *IEEE/ACM Trans. Netw.*, vol. 23, no. 6, pp. 1959–1969, 2015.
[15] Y. Guo and et al., "SOTE: traffic engineering in hybrid software defined networks," *Elsevier Comput. Networks*, vol. 154, pp. 60–72, 2019.
[16] "Brocade mlx-8 pe," https://www.dataswitchworks.com/datasheets/MLX_Series_DS.pdf, Nov. 2020.
[17] B. Mao and et al., "Routing or computing? the paradigm shift towards intelligent computer network packet transmission based on deep learning," *IEEE Trans. Computers*, vol. 66, no. 11, pp. 1946–1960, 2017.
[18] Z. Xu, J. Tang, J. Meng, and et al., "Experience-driven networking: A deep reinforcement learning based approach," in *IEEE INFOCOM'18*.
[19] P. Kumar, Y. Yuan, C. Yu, and et al., "Semi-oblivious traffic engineering: The road not taken," in *USENIX NSDI'18*.
[20] J. Zhang and et al., "CFR-RL: traffic engineering with reinforcement learning in SDN," *IEEE J. Sel. Areas Commun.*, vol. 38, no. 10, pp. 2249–2259, 2020.
[21] D. Levin, M. Canini, S. Schmid, and et al., "Panopticon: Reaping the benefits of incremental SDN deployment in enterprise networks," in *USENIX ATC'14*.
[22] K. Poularakis and et al., "One step at a time: Optimizing SDN upgrades in ISP networks," in *IEEE INFOCOM'17*.
[23] D. K. Hong and et al., "Incremental deployment of SDN in hybrid enterprise and ISP networks," in *ACM SOSR'16*.
[24] S. Agarwal, M. Kodialam, and T. Lakshman, "Traffic engineering in software defined networks," in *IEEE INFOCOM'13*.
[25] C. Chu and et al., "Congestion-aware single link failure recovery in hybrid SDN networks," in *IEEE INFOCOM'15*.
[26] X. Jia, Y. Jiang, and Z. Guo, "Incremental switch deployment for hybrid software-defined networks," in *IEEE LCN'16*.

[27] Z. Guo and et al., "Joint switch upgrade and controller deployment in hybrid software-defined networks," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 5, pp. 1012–1028, 2019.
[28] S. Vissicchio, O. Tilmans, L. Vanbever, and et al., "Central control over distributed routing," in *ACM SIGCOMM'15*.
[29] D. Levin and et al., "Logically centralized? state distribution trade-offs in software defined networks," in *ACM HotSDN'12*.
[30] M. Reitblatt and et al., "Consistent updates for software-defined networks: Change you can believe in!" in *ACM HotNets'11*.
[31] Y. Zhang, E. Ramadan, H. Mekky, and et al., "When raft meets SDN: how to elect a leader and reach consensus in an unruly network," in *ACM APNet'17*.
[32] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, 1998.
[33] A. Panda, W. Zheng, X. Hu, and et al., "SCL: Simplifying distributed SDN control planes." in *USENIX NSDI'17*.
[34] H. Xu and et al., "Scalable software-defined networking through hybrid switching," in *IEEE INFOCOM'17*.
[35] H. Xu and et al., "Achieving high scalability through hybrid switching in software-defined networking," *IEEE/ACM Trans. Netw.*, vol. 26, no. 1, pp. 618–632, 2018.
[36] "Chn-ix," http://www.chn-ix.net/, Nov. 2020.
[37] "sflow," https://sflow.org/, Nov. 2020.
[38] B. Claise, "Cisco systems netflow services export version 9," Tech. Rep., 2004.
[39] "Abilene dataset," http://www.cs.utexas.edu/yzhang/research/AbileneTM, Nov. 2020.
[40] "Gurobi optimization," http://www.gurobi.com, Nov. 2020.
[41] J. Zhang and et al., "Load balancing for multiple traffic matrices using SDN hybrid routing," in *IEEE HPSR'14*.
[42] M. R. Belgaum and et al., "A systematic review of load balancing techniques in software-defined networking," *IEEE Access*, vol. 8, pp. 98 612–98 636, 2020.
[43] Z. Guo and et al., "Jumpflow: Reducing flow table usage in software-defined networks," *Elsevier Comput. Networks*, vol. 92, pp. 300–315, 2015.
[44] M. Al-Fares and et al., "Hedera: dynamic flow scheduling for data center networks," in *USENIX NSDI'10*.
[45] J. Zhang, K. Xi, M. Luo, and et al., "Dynamic hybrid routing: Achieve load balancing for changing traffic demands," in *IEEE/ACM IWQoS'14*.
[46] G. Bianchi and et al., "Openstate: programming platform-independent stateful openflow applications inside the switch," *Comput. Commun. Rev.*, vol. 44, no. 2, pp. 44–51, 2014.
[47] J. Yang and et al., "Focus: Function offloading from a controller to utilize switch power," in *IEEE NFV-SDN'16*.
[48] J. He and W. Song, "Achieving near-optimal traffic engineering in hybrid software defined networks," in *IFIP Networking'15*.
[49] C. Ren and et al., "Achieving near-optimal traffic engineering using a distributed algorithm in hybrid SDN," *IEEE Access*, vol. 8, pp. 29 111–29 124, 2020.
[50] B. Pit-Claudel, Y. Desmouceaux, P. Pfister, and et al., "Stateless load-aware load balancing in P4," in *IEEE ICNP'18*.
[51] N. P. Katta, M. Hira, C. Kim, and et al., "HULA: scalable load balancing using programmable data planes," in *ACM SOSR'16*.
[52] Sandhya and et al., "A survey: Hybrid SDN," *J. Netw. Comput. Appl.*, vol. 100, pp. 35–55, 2017.
[53] R. Amin and et al., "Hybrid SDN networks: A survey of existing approaches," *IEEE Commun. Surv. Tutorials*, vol. 20, no. 4, pp. 3259–3306, 2018.
[54] S. Vissicchio and et al., "Opportunities and research challenges of hybrid software defined networks," *Comput. Commun. Rev.*, vol. 44, no. 2, pp. 70–75, 2014.
[55] S. Ahmad and A. H. Mir, "Scalability, consistency, reliability and security in SDN controllers: A survey of diverse SDN controllers," *J. Netw. Syst. Manag.*, vol. 29, no. 1, p. 9, 2021.
[56] C. Jin and et al., "Magneto: Unified fine-grained path control in legacy and openflow hybrid networks," in *ACM SOSR'17*.
[57] X. Jin and et al., "Covisor: A compositional hypervisor for software-defined networks," in *USENIX NSDI'15*.
[58] T. Y. Cheng and X. Jia, "Compressive traffic monitoring in hybrid sdn," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 12, pp. 2731–2743, 2018.