Achieving High Scalability Through Hybrid Switching in Software-Defined Networking

Hongli Xu, *Member, IEEE*, He Huang, *Member, IEEE*, Shigang Chen, *Fellow, IEEE*, Gongming Zhao, and Liusheng Huang, *Member, IEEE*

***Abstract*— Traditional networks rely on aggregate routing and decentralized control to achieve scalability. On the contrary, software-defined networks achieve near optimal network performance and policy-based management through per-flow routing and centralized control, which, however, face scalability challenge due to: 1) limited ternary content addressable memory and on-die memory for storing the forwarding table and 2) per-flow communication/computation overhead at the controller. This paper presents a novel hybrid switching (HS) design, which integrates traditional switching and software-defined networking (SDN) switching for the purpose of achieving both scalability and optimal performance. We show that the integration also leads to unexpected benefits of making both types of switching more efficient under the hybrid design. We also design the general optimization framework via HS and propose an approximation algorithm for load-balancing optimization as a case study. Testing and numerical evaluation demonstrate the superior performance of HS when comparing with the state-of-the-art SDN design.**

***Index Terms*— Software defined networks, scalable routing, flow table constraint, load balancing, approximation.**

I. Introduction

SCALABILITY has been a core issue in the history of large network development. The conventional wisdom holds two design principles: aggregate routing paths and distributed control. Modern switches/routers forward packets from incoming ports to outgoing ports via switching fabric. The data plane uses ASIC hardware and on-die memory (such as SRAM) to process packets in real time at high speed. The on-die memory is typically a few megabytes. Increasing on-die memory is technically feasible, but it comes with a much higher price tag and access time is longer. There is a huge incentive to keep on-die memory small because smaller memory can be made faster and cheaper. To make the matter more challenging, limited on-die memory may have to be shared among routing/performance/measurement/security functions that are implemented on the same chip. The amount of on-die memory allocated for storing the forwarding (or routing) table will be limited, which makes per-flow routing (*i.e.*, one table entry for each flow) unscalable to a large network with millions of concurrent flows. To address this scalability problem, the classical design principle is to perform destination-based aggregate routing (instead of per-flow routing), where all flows with the same destination address or address prefix will share the same path. The second design principle is to decentralize the routing control function in order to avoid a single point of failure or performance bottleneck.

The emergence of software-defined networking (SDN) [2]–[8] has shattered both principles. It uses a centralized controller to determine per-flow paths and deploy these paths to the switches' forwarding tables. With the network-wide information at one place, the centralized control makes it far easier to enforce global policies and achieve optimal traffic management. These benefits outweigh the scalability concern in the compromise made by the SDN design. But the scalability problem will not go away, under per-flow routing and centralized control. Today's SDN switches typically have a few thousands of entries in their on-die flow tables. Even the high-end Broadcom Trident2 chipset supports only 16K forwarding rules [6]. When there are too many flows to fit in the flow table, we will have to reject some flows [6], replace existing flows in the table with new flows (which causes churns and increases the load of the controller to repetitively deploy paths for the same flows), or bring aggregate routing back.

Forwarding rules with wildcards were proposed for aggregate routing [5]. But wildcard rules can only be implemented through TCAM (Ternary Content Addressable Memory), which is small, costly and energy-hungry. The small number of wildcard rules may result in aggressive aggregation when facing a large number of excess flows, whereas the benefits of SDN rest upon its ability of differentiating arbitrary individual flows. Moreover, there is a lack of systematic studies on how to construct and manage optimal wildcard rules in a dynamic, heavily loaded network, which is a challenging problem. Therefore, alternative or complementary solutions to the scalability problem are under call.

If traditional aggregate routing and decentralized control help scalability while SDN helps performance with centralized control, our idea is to integrate them for hybrid switching, which achieves the benefits of both worlds and is not subject to the restriction of TCAM [1]. This paper presents a novel hybrid switching design. On the one hand, it leverages the

Manuscript received January 20, 2017; revised August 3, 2017 and October 27, 2017; accepted December 28, 2017; approved by IEEE/ACM Transactions on Networking Editor S. Rao. Date of publication January 22, 2018; date of current version February 14, 2018. This work was supported in part by the NSFC under Grant 61472383, Grant U1709217, Grant 61728207, and Grant 61472385 and in part by the Natural Science Foundation of Jiangsu Province in China under Grant BK20161257. The work of H. Huang was supported in part by the NSFC under Grant 61572342 and Grant 61672369 and in part by the NSF of Jiangsu Province under Grant BK20151240 and Grant BK20161258. The work of S. Chen was supported by the NSF under Grant STC-1562485 and Grant CNS-1719222. This paper was presented at the IEEE INFOCOM 2017 [1]. *(Corresponding author: He Huang.)*

H. Xu, G. Zhao, and L. Huang are with the School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China, and also with the Suzhou Institute for Advanced Study, University of Science and Technology of China, Suzhou 215123, China (e-mail: xuhongli@ustc.edu.cn; zgm1993@mail.ustc.edu.cn; lshuang@ustc.edu.cn).

H. Huang is with the School of Computer Science and Technology, Soochow University, Suzhou 215006, China (e-mail: huangh@suda.edu.cn).

S. Chen is with the Department of Computer and Information of Science and Engineering, University of Florida, Gainesville, FL 32611 USA (e-mail: sgchen@cise.ufl.edu).

Digital Object Identifier 10.1109/TNET.2018.2789339

mature methods of traditional switching to achieve scalability by avoiding per-flow communication/computation overhead to the controller and reducing the number of forwarding rules needed to support a large number of flows. On the other hand, it exploits the flexibility of SDN switching to achieve near optimal network performance without overflowing the forwarding table. More interestingly, we show that the integration of traditional switching and SDN switching brings unexpected benefits to each other. The SDN's centralized control will help implement traditional switching much more efficiently. It can also centrally integrate the management/security policy requirements into the computation of traditional switching/routing tables so that the paths comply with the policies. In the meanwhile, with a hybrid deployment design, we show that traditional aggregate routing will help to greatly reduce the overhead of deploying SDN paths by significantly reducing the number of forwarding rules needed. We also discuss how to perform per-flow traffic measurement without using the OpenFlow counters in the forwarding table, which is important in our hybrid switching design where many (or even most) flows are not in the table. Finally, we give a case study on how to perform global optimization at the controller for flow re-routing under hybrid switching. Our numerical evaluation shows that the proposed hybrid switching design outperforms wildcard-based DevoFlow [5] by significantly lowering the number of forwarding rules under the same traffic conditions or achieves much better network performance under the same forwarding-table size.

The rest of this paper is organized as follows. Section II introduces the switching, routing, and hybrid schemes. We present the design of the hybrid switching in Section III. Section IV describes the general optimization framework vis HS and designs an efficient algorithm for a case study of load-balancing optimization. The testing and simulation results are presented in Section V. Section VI discusses the related work. Section VII draws the conclusion.

II. Switching, Routing, and Hybrid

A. Traditional Switching

A traditional Ethernet switch uses a *switch table* to learn reachability information from the packets (or data frames in layer-2 terminology) that it receives. When a switch receives a frame from a port, it learns that the source MAC address in the frame header can be reached from that port. This information is stored in the switch table where each entry contains an MAC address and a port number.

If a switch receives a frame whose destination MAC address is in the switch table, the switch will forward the packet to the corresponding port. Otherwise, it will forward the frame to all ports except for the port from which the frame is received, generating a broadcast. For two-way communication between two hosts, broadcast will happen only once because the first exchange between the hosts will let all switches along the communication path learn how to reach them. To achieve high throughput, the switch table is often implemented as a hash table in SRAM [9].

B. Traditional Routing

Routers or layer-3 switches are able to perform distributed path selection through routing protocols such as OSPF [10] for intra-domain routing or BGP [11] for inter-domain routing. The modern router architecture consists of a control plane, where the routing protocols and management functions are implemented, and a data plane, which handles packet forwarding at high speed. To achieve high throughput, the *routing table* can be cached in on-die SRAM at the arrival network interface. Each routing-table entry consists of a destination address prefix, an output port, and other fields.

The path selection is coarse-grained. It is destination-based, not flow-based, which helps reduce the table size. There may be many flows from a source network to a destination network. They will all follow the same path because they share the same destination address prefix. This limits the flexibility in offering quality of service, balancing load, utilizing the underused alternative paths, or performing flow-level management policies.

C. Software-Defined Switching

Comparing with traditional switching/routing, a fundamental difference of the SDN architecture is its centralized control. An SDN network consists of three types of devices: a central controller, SDN switches that are inter-connected to form a network, and end hosts that are connected to the switches. An SDN switch has a *forwarding table* specifying per-flow paths. The forwarding table is typically implemented in TCAM to support wildcard fields and parallel lookup of all table entries. For exact rules without wildcards, they may be implemented in TCAM or SRAM. Although we view the forwarding table logically as a single table, it may be implemented by multi-tiered tables [12].

When a switch receives a packet, it matches the fields in the packet's headers against the table. If there is a matching entry, the packet is handled according to the instruction field, which may drop, log, or forward the packet to the output port. When there is no matching entry, the switch sends a request carrying the packet header to the controller which selects a path for the flow and installs proper rules on the switches along the path. The controller knows the network topology and collects traffic statistics from the switches. With this information and user-defined policies, the controller makes path selection.

The centralized control architecture is simple. It takes the control plane out of the devices and pushes most of the complexity to the controller, leaving the switches only with its data plane. Not only does this simplify the data-forwarding devices, but the centralized control makes it easier to enforce complex traffic management policies.

However, the shortcomings of the centralized control are also obvious. The controller can become a performance bottleneck. It has to set up the paths of all flows, incurring per-flow computation overhead for path selection and per-flow communication overhead of transiting a packet header to the controller and the forwarding rules back to the switches along the selected path, together with acknowledgement packets, as well as flow statistics collection. Such per-flow overhead becomes significant when most flows are short with only a small number of packets, which is unfortunately the common case [9].

D. Idea of Hybrid Switching

Both traditional switching/routing and SDN switching have pros and cons. The former adopts coarse-grained, destination-based path selection for space saving. With careful use of available table space, today's switches and routers are able to scale to very large networks. The latter provides fine-grained path selection and management functions at the flow level. On the one hand, fine-grained per-flow paths require more forwarding rules. But on the other hand, each forwarding rule takes much more space than an entry in the traditional switch table or routing table. That means the number of available forwarding rules will be smaller, given the same memory space. This is even more true if TCAM is used for implementing the forwarding table. Fewer forwarding rules in availability and more rules in demand contradict each other in system design.

To solve this problem and relieve the computation/communication bottleneck at the controller, we propose *hybrid switching* that combines traditional switching/routing and SDN switching for the benefits of both worlds. We refer to the forwarding paths used in traditional switching/routing as *traditional paths*, and the path in SDN switching as *SDN paths*. A device that performs hybrid switching is called a *hybrid switch*, and its benefits are summarized below.

First, studies on real network traffic showed that most flows were short-lived with light traffic [13]. Routing them via SDN's forwarding tables has little lasting impact on network performance and does not justify the associated overhead. Moreover, it causes additional delay due to communication with the controller, path selection and deployment, which hurts flow performance. With hybrid switching, we will direct these flows through the traditional paths, avoiding per-flow overhead to the controller and reducing the number of forwarding rules needed.

Second, studies also showed that the elephant flows dominate in traffic volume although their number may be relatively small [13]. From the traffic engineering's point of view, it is economic to focus on these flows and route them via optimal SDN paths for desired network performance. SDN switching allows us to directly control a selected number of flows that have the most impact.

Third, with the help of centralized control from SDN, we will be able to implement traditional switching/routing much more efficiently and make sure that the traditional paths comply with the management/security policies if there is any. With the help of traditional paths, we will be able to implement SDN paths much more efficiently by reducing the number of forwarding rules needed for the deployment. Hybrid switching is not a simple combination of SDN switching and traditional switching/routing, but instead a full integration, in which the two are inter-twined and help each other to make both perform better.

In the rest of the paper, whenever we refer to *forwarding table*, we imply SDN switching. Similarly, *switch table* implies traditional switching, and *routing table* implies traditional routing.

III. Design of Hybrid Switching (HS)

This section goes step by step in explaining our design of hybrid switching. First, we discuss how to implement traditional switching/routing more efficiently with the help of a centralized controller. Second, we integrate SDN switching with traditional switching/routing. Third, we discuss how to identify large flows by per-flow statistics measurement in a compact space without using the counters in the forwarding tables. Fourth, we design hybrid path deployment that exploits traditional paths to reduce the number of forwarding rules needed for SDN paths. Fifth, we combine all the pieces into an overall design of hybrid switching.

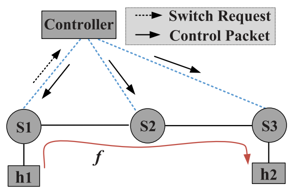


Fig. 1. Implementing switch tables with the help of a centralized controller.

A. Traditional Switching/Routing With a Controller

1) Traditional Switching With a Controller: We consider an SDN network where each switch also implements a traditional switch table, in addition to the forwarding table for SDN switching. We explain how to integrate switch tables into the SDN architecture and implement them more efficiently. In this work, we assume a *full SDN* network where all switches are SDN switches. More precisely, they are hybrid switches, implementing the traditional switch tables *with the help of a centralized controller*. We will discuss how to extend HS to a partial SDN network in Section III-F.2.

When a switch receives a data frame and does not find a matching entry in its switch table, it sends a request, carrying the destination MAC address, to the controller. The controller has the full knowledge of the switches, the hosts, and the network topology. It finds a path to the destination. When there are multiple paths, it selects one based on certain criterion such as shortest distance. It will then send a control packet, with the proper switch-table entry, to every switch on the path, including the requesting switch, as shown in Figure 1. The data frame, as well as all subsequent frames in the flow, will be forwarded along this path unimpeded towards its destination. With the help of the controller, we avoid the broadcast — which may reach all end hosts of the whole network — when a matching entry cannot be found.

2) Traditional Routing With a Controller: Next we consider an SDN network where each switch is a layer-3 switch, which implements a traditional routing protocol, in addition to its forwarding table. We use OSPF [10] as example. In the classical implementation of OSPF, every router periodically sends the state of its adjacent links to all other routers, and receives such information from other routers as well. Therefore, all routers have the same view of the whole network, based on which they compute their routing tables. Now, with a centralized controller, all routers only need to periodically send their link states to the controller, which collects a global view of the network, computes the routing tables of all routers, and updates the routers with the changes in their routing tables.

In the original OSPF, each router receives $O(|E|)$ link states, where E is the set of links in the network. The overall communication complexity for all routers is $O(|N|\ |E|)$, where N is the number of routers. With the controller's help, routers do not receive link states. Only the controller does, with a communication complexity $O(|E|)$ for the whole network. For the routing-table update, only the difference will be transmitted and the controller is able to contain such difference to a small amount by adopting a route computation algorithm that keeps the stability of the routes. In fact, even in traditional OSPF networks, the protocol is often configured to compute routes based on hop counts to avoid route churns [10]. In an SDN network, the controller has more reasons to adopt such a strategy because it has another tool, SDN switching, for dealing with traffic engineering and load balancing on different paths.

B. Integration of SDN Switching With Traditional Switching/Routing

We first consider the integration of SDN switching with traditional routing. When a switch receives a packet, it matches the packet against both the SDN forwarding table and the traditional routing table. As long as the forwarding table has a matching entry, it takes the precedence and the packet will be forwarded accordingly. If the packet belongs to a new flow and the forwarding table does not have a matching entry, there are two path selection strategies.

1) Traditional Path First (TPF): New flows will take the traditional paths by default without causing any immediate overhead to the controller. For a packet from a new flow, without a match in the forwarding table, the switch will handle the packet according to the routing table, which will always give a matching entry, meaning that it can scale to an arbitrary number of flows. New flows will not automatically generate requests to the controller for path selection, in contrast to what today's SDN switches do. This property helps reduce the controller's communication/computation burden and avoid a potential performance bottleneck in the system. While all new flows follow the traditional paths by default, the switches will monitor their flows, identify the large ones, and estimate their sizes. Periodically they will send the information of the identified large flows to the controller, which performs global optimization to improve network performance by re-routing some or all of the large flows via optimal SDN paths, subject to the size constraint of the forwarding tables at the switches. The formulation of the optimization is dependent on the user-specified performance and management requirements, which vary in practice; we will provide a case study in the next section. The controller will then update the switches' forwarding tables by installing the new paths; see [7], [14] for update schemes that ensure packet-level routing consistency.

2) SDN Path First (SPF): New flows will take the SDN paths by default. For a packet from a new flow, without a match in the forwarding table, as long as the switch's forwarding table is not overflown, it will forward the packet header to the controller for installing an SDN path. If the forwarding table is full, the switch forwards the packet based on the routing table.

Although SPF solves the overflow problem of forwarding tables, it still faces other problems of SDN switching as explained in Section II-C: per-flow communication/computation overhead to the controller (even for small flows that contain a few packets themselves) and extra delay to a flow's first packet due to the setup of SDN path. We advocate TPF not only because it avoids these problems but also because batch setup of forwarding paths for a set of flows together tend to produce better global optimization than setup of the paths one at a time sequentially.

Next, we consider the integration of SDN switching with traditional switching under TPF. When a switch receives a data frame, it matches the frame against both the SDN forwarding table and the traditional switch table. As long as the forwarding table has a matching entry, the frame will be processed based on that entry; otherwise, if the switch table has a matching entry, the frame will be forwarded to the specified output port. If there is no matching entry in either table, the switch will send a request, carrying the frame's destination MAC address, to the controller, which will establish a tradition path towards the destination and install proper entries in the switch tables along the path.

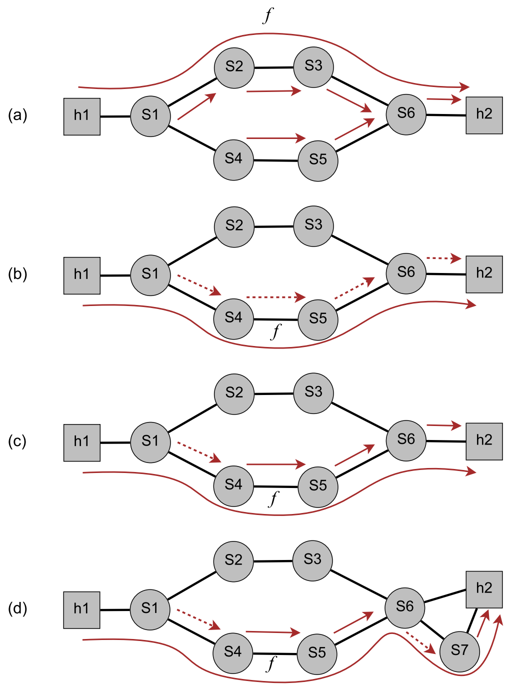


Fig. 2. Illustration of hybrid path deployment. (a) Flow f follows the traditional path, which is specified by solid arrows. (b) Flow f is re-routed to an SDN path, $s_1 \rightarrow s_4 \rightarrow s_5 \rightarrow s_6$, where the forwarding rules are shown as dashed arrows. (c) Due to hybrid deployment, only one forwarding rule at s_1 is actually deployed. (d) In a slightly different example, we show that more than one forwarding rule may be used to deviate the SDN path from the traditional path for more than once.

Again, all switches will monitor their flows and send the information of large flows to the controller, which will perform global optimization periodically by re-routing large flows to optimal SDN paths.

C. Hybrid Path Deployment

When the controller decides to re-route a flow from its traditional path to an SDN path p, under the traditional wisdom, the controller must deploy one forwarding rule at every switch on p, occupying an entry in the switch's forwarding table [6], [7]. However, the proposed hybrid switching offers a new opportunity to save forwarding-table space. Consider an arbitrary switch s on p. Let t be the output port specified in the forwarding rule that the controller intends to deploy at s. If t is also what the traditional path from s to the destination specifies, there is no need to actually deploy the forwarding rule because without this rule, switch s will use the traditional path automatically, which will forward the packet to port t.

Figure 2 shows an example of hybrid path deployment, where the top plot shows a flow f passing through a traditional path from host h_1 to host h_2. Assume all switches already have proper matching entries for h_2 in their switch tables (or routing tables), as shown by solid arrows in the top plot. Now the controller wants to re-route f to a different path, $s_1 \rightarrow s_4 \rightarrow s_5 \rightarrow s_6 \rightarrow h_2$, requiring four forwarding rules to be deployed at four switches, as shown by dashed arrows in the second plot. The forwarding rules at s_4, s_5 and s_6 specify the same outports as the traditional paths do. With hybrid deployment, the controller only needs to deploy one forwarding rule at s_1, where the rest of the SDN path follows the traditional path, as the third plot shows. In a more complicated case of the fourth plot, two forwarding rules are deployed at s_1 and s_6, respectively, allowing the SDN path to deviate from the traditional path for a second time at s_6.

As a side benefit, hybrid path deployment can help reduce the dependency in the order of forward-rule deployment, which in turn helps reduce the deployment time. According to [7] and [14], in order to ensure packet-level routing consistency, we should perform two-phase deployment when re-routing f, with the first phase installing the forwarding rules at s_4, s_5 and s_6 (the third plot), and then the second phase installing the forwarding rule at s_1; see the original paper [7] for reasons. Under hybrid path deployment, we need only one phase of installing the rule at s_1 in this example, which cuts down deployment time.

D. Per-Flow Statistics and Large-Flow Identification

According to the OpenFlow specification [15], each entry in the forwarding table has statistic counters for per-flow traffic measurement. However, this is insufficient for our design because many or even most flows will follow the traditional paths specified in the switch table (or routing table) where there is no per-flow entry. More importantly, under TPF in Section II-D, all new flows follow the traditional paths by default. When we want to find the large ones among them for re-routing, the counters in the forwarding table will not help. We need to adopt a new mechanism to collect per-flow statistics without incurring too much space overhead as the size of SRAM is limited. The idea is for each switch to produce a traffic synopsis, a compact data structure that summarizes the traffic of all flows passing the switch and supports queries on individual flows.

There is a rich literature on per-flow size measurement under tight memory. For a few examples, the method of *randomized counter sharing* (RCS) [16] was proposed to further reduce memory requirement and processing time by using virtual storage vectors. We adopt RCS because its per-packet overhead is very small (updating a single counter) and it supports an arbitrary number of flows with a pre-allocated memory space.

To cover all flows, it is sufficient for only the edge switches to perform traffic measurement, which relieves the core switches from this overhead. Each edge switch samples the arrival packets to record a number of flow identifiers, and forms a *storage vector* of the flow, denoted as C_f, where f is the identifier of the flow. Large flows have proportionally higher probabilities to be sampled. At the end of every measurement period, it estimates the size of each sampled flow f by summing up the counters in C_f after subtracting away a noise term, which is simply the average counter value across the whole synopsis. According to the analysis in [16], RCS gives very accurate estimates for large flows whose sizes are much larger than the average flow size. This fact is confirmed in our experiments. Knowing the sizes of the sampled flows, each edge switch reports its largest k flows and their sizes to the controller, where k is a system parameter set by the controller. As we will show in the case study, the computation overhead of the controller is dependent on the number of flows to be re-routed. By adjusting this value, the controller has a way to prevent the switches from overloading itself.

E. Overall Design

The overall design is illustrated through a packet-processing flow chart in Figure 3. When a packet arrives at an input port of a switch, it is processed with forwarding-table lookup and switch/routing table lookup, which may be carried out in parallel by ASIC hardware on chip. We adopt the TPF strategy: The packet will be handled by the forwarding table if a matching entry exists. If not, the packet will be forwarded based on the switch/routing table if a matching entry exists. In case of switch table, there may not exist a matching entry. When this happens, the switch will report the destination MAC address to the controller for path selection. In the meantime, for an edge switch, the packet will also be processed for flow sampling and synopsis-based traffic measurement.

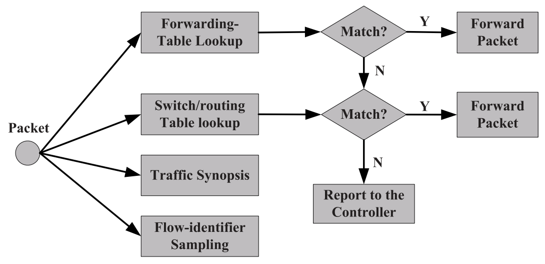


Fig. 3. Illustration of real-time packet processing.

Besides the above real-time packet processing, at the end of each measurement period, every edge switch will estimate the sizes of the sampled flows based on the information recorded in the synopsis. It reports the largest k flows and their estimated rates to the controller for possible re-routing, where a flow's estimated rate is the estimated flow size divided by the measurement period. We should note that, the optimization approach can be combined with other flow size detection mechanisms, like end-point based detection [17] instead of switch based shared-counter mechanisms.

F. Discussion

This section discusses some issues to enhance our HS mechanism.

1) Dealing With Topology Changes: We believe that topology changes have similar impact on DevoFlow and HS. When there is a link (or node) failure, some existing paths specified by wildcard rules in DevoFlow or routing tables in HS may become invalid. After a neighboring switch detects such a failure and reports it to the controller, the controller will recompute the affected rules or routing entries and updates the relevant switches. When there is a new link (or a new node), some existing paths may become non-optimal. Again, the controller will update wildcard rules or routing entries when necessary.

2) Extending HS to a Partial SDN Network: Our HS mechanism can be applied in a partial SDN network. Since the controller can only control those SDN switches in a partial SDN network, we just discuss how to process packets on SDN switches. There are two cases as a packet arrives at an SDN switch. On one hand, there is a matching entry in a routing table or a forwarding table, this switch will forward the packet directly according to the matched rule. On the other hand, there is no matching entry for this packet. The switch reports the destination MAC address to the controller for path selection. In this situation, the controller will choose an admissible path [4], [18] and install rules on the routing tables. We note that an admissible path should be compatible with the distributed routing protocol running on the legacy devices. To re-route

TABLE I
KEY NOTATIONS

Symbol	Semantics
S	a set of SDN switches
H	a set of terminals
E	a set of network links
$c(e)$	the capacity of link e
$l(e)$	the traffic load of link e
Π	a set of elephant flows
$r(f)$	the estimated rate of flow f
$\mathcal{P}(f)$	a candidate path set for flow f
$p^*(f)$	the current path of flow f
$T(s)$	the number of residual entries at switch s

some elephant flows, the controller should also select the admissible paths for them.

3) Integrating the Policy Requirements: We believe that policy enforcement in an SDN can also benefit from such default paths. Consider a policy that requires certain packets to be routed through a firewall. If this policy is applied to all packets from a certain source subnet to a certain destination subnet, it will be easy for the controller to modify Dijkstra's algorithm such that the routing path from the specified source to the specified destination includes a firewall. After this path is inserted into the routing tables that are distributed to the switches on the path, the policy is automatically enforced. More generally, when wildcard rules are used for policy enforcement (as DevoFlow does), our default paths are still helpful, depending on the specific mechanism of enforcement.

IV. Application of Hybrid Switching

The component missing so far is the controller's global optimization in re-routing large flows via SDN paths. However, the implementation of this component is directly related to the performance goal and the management policies, which are set by the system admin and vary greatly in practice.

A. A General Optimization Framework Via HS

We first give the notations: Denote the set of n switches as $S = \{s_1, \ldots, s_n\}$, and the set of m hosts (terminals) as $H = \{h_1, \ldots, h_m\}$. The network topology is modeled as a graph $G = (S \cup H, E)$, where E is the set of links. Let $c(e)$ be the capacity of a link e and $l(e)$ be its load. The switches measure traffic loads on all their ports (*i.e.*, adjacent links) and make the information available to the controller through Openflow [2]. For ease of reference, some key notations are listed in Table I.

From the large flows reported by the switches (or by other flow detection methods, *e.g.*, [17]), the controller selects a subset Π of the largest ones for re-routing so as to provide various performance optimization. The size of Π may be constrained by the budget of execution time for solving the optimization problem; the relationship between the size of Π and the execution time can be roughly estimated based on the past executions. Let $r(f)$ be the estimated rate of flow $f \in \Pi$, which is reported by the edge switch of the flow. Let $\mathcal{P}(f)$ be the set of candidate paths for flow f. $\mathcal{P}(f)$ is determined based on the management policies and performance objectives. For example, if there is a policy that matches f and states that the flow's path must pass certain types of middleboxes (*e.g.*, firewalls or IDSes), then $\mathcal{P}(f)$ must only contain such paths, and if there is too many of them, we may include only a certain number of best ones under a certain performance criterion, such as having the shortest number of hops or having the large capacities. $\mathcal{P}(f)$ also contains the path $p^*(f)$ that the flow is currently routed through.

Let $y_f^p \in \{0, 1\}$ be an indicator variable for whether flow f will be routed on a path $p \in \mathcal{P}(f)$. Let $T(s)$ be the number of residual entries in the forwarding table at switch s. Let $I(f, p, s)$ be a binary value for hybrid path deployment (Section III-C): if path p assigned to flow f overlaps with the flow's traditional path at switch s, then there is no need to deploy an entry on the forwarding table of switch s, *i.e.*, $I(f, p, s) = 0$; otherwise $I(f, p, s) = 1$. We formalize the general optimization framework as follows:

$$\max\ \mathcal{H}$$
$$s.t. \begin{cases} b(e) = l(e) - \sum_{f\in\Pi: e\in p^*(f)} r(f), \quad \forall e \in E \\ \sum_{p\in\mathcal{P}(f)} y_f^p = 1, \quad \forall f \in \Pi \\ \sum_{f\in\Pi} \sum_{p\in\mathcal{P}(f): s\in p} y_f^p \cdot I(f,p,s) \le T(s), \\ \quad \forall s \in S \\ b(e) + \sum_{f\in\Pi} \sum_{p\in\mathcal{P}(f): e\in p} y_f^p r(f) \le \lambda \cdot c(e), \\ \quad \forall e \in E \\ y_f^p \in \{0,1\}, \quad \forall p, f \\ \lambda \in \Lambda. \end{cases} \tag{1}$$

The first set of equations computes the background traffic load $b(e)$, $\forall e \in E$, when the flows in Π are taken out. The second set of equations requires that flow $f \in \Pi$ is not splittable; it will be forwarded through a single path from $\mathcal{P}(f)$. The third set of inequalities describes the size constraints of the forwarding tables on switches. The fourth set of inequalities states the traffic load on each link e, which is the sum of background traffic load and traffic from the flows scheduled on the path. Note that Λ is a feasible range for parameter λ. The optimization objective is determined by the user's requirement. We denote this as $\mathcal{H}$, which may depend on different parameters, such as λ and/or the rates of all flows in Π. It should be noted that Λ and $\mathcal{H}$ are closely related. For example, if we try to maximize the amount of traffic that passes through middleboxes, the objective function $\mathcal{H}$ is described as $\sum_{f\in\Pi} \sum_{p\in\mathcal{P}(f)} r(f) y_f^p z_p$, where z_p denotes whether there is a middlebox on path p or not. Under this situation, it is required that there is no congestion on links, thus we set $\Lambda = \{1\}$, *i.e.*, $\lambda = 1$. As another example, Section IV-B gives the load-balancing optimization for an SDN. We point out that the above integer program will always have at least one feasible solution — all flows $f \in \Pi$ take their current paths $p^*(f)$, which means no re-routing and all flows keep the status quo.

B. A Case Study of Load-Balancing Optimization

In this section, we consider a special case (aiming to load-balancing optimization) of the general frameworks. According to Eq. (1), we formalize the problem of load-balanced routing with flow tables of limited size (LBR-FT).

$$\min \ \lambda$$
$$s.t. \begin{cases} b(e) = l(e) - \sum f \in \Pi : e \in p^*(f) r(f), \quad \forall e \in E \\ \sum p \in \mathcal{P}(f) y_f^p = 1, \quad \forall f \in \Pi \\ \sum_{f\in\Pi} \sum_{p\in\mathcal{P}(f):s\in p} y_f^p \cdot I(f,p,s) \le T(s), \\ \quad \forall s \in S \\ b(e) + \sum_{f\in\Pi} \sum_{p\in\mathcal{P}(f):e\in p} y_f^p r(f) \le \lambda \cdot c(e), \\ \quad \forall e \in E \\ y_f^p \in \{0,1\}, \quad \forall p, f \\ \lambda \le 1. \end{cases} \tag{2}$$

The former three sets of constraints are same as those in Eq. (1). The fourth set of inequalities states that the traffic load on each link e should not exceed $\lambda \cdot c(e)$, where λ is the load-balance factor (less than 1). That is, $\lambda \in (0, 1]$. The objective is to minimize λ. Globally reducing the link loads and achieving load balance have many benefits. It helps prevent large queuing delays that happen when λ approaches towards 1. It leaves room for new flows or allows the existing flows to increase their rates for better network throughput. It indirectly helps balance the occupation of the forwarding tables as flows are spread among alternative paths.

Theorem 1: LBR-FT defined in Eq. (2) is an NP-hard problem.

Proof: We consider a special example of the LBR-FT problem, in which there is no constraint on the forwarding table size, and the sampled flow set will cover all the flows in a network. Then, we can deploy one flow entry for each flow in an SDN. In other words, this becomes an unsplittable multi-commodity flow with minimum congestion problem [19], which is NP-Hard. Since the multi-commodity flow problem is a special case of our problem, the LBR-FT problem is NP-Hard too. □

C. Algorithm Description for the LBR-FT Problem

Since the LBR-FT problem is NP-Hard, it is difficult to solve this problem optimally. In this subsection, we first present an approximation algorithm, called RDSR, for the LBR-FT problem. We then modify the RDSR algorithm so as to satisfy the link capacity and flow table size constraints in an SDN.

1) Rounding-Based Scalable Routing (RDSR): We describe a rounding-based scalable routing algorithm (RDSR) for load-balancing in an SDN. As there could be an exponential number of possible paths between two edge switches, following [2], [6], and [20], we assume that the controller precomputes a set of candidate paths between each pair of edge switches. Given the source and the destination of a flow f, we will set $\mathcal{P}(f)$ to be the set of candidate paths between the two corresponding edge switches which connect to the source terminal and the destination terminal, respectively. These candidate paths may simply be the shortest paths between the edge switches which can be found by depth-first search. We should note that the path set $\mathcal{P}(f)$ also includes the traditional path of this flow.

To solve the problem formalized in Eq. (2), the algorithm constructs a linear program as a relaxation of the LBR-FT problem. More specifically, LBR-FT assumes that the traffic of each flow should be forwarded only through one path. By relaxing this assumption, traffic of each flow $f \in \Pi$ is permitted to be splittable and forwarded through a path set $\mathcal{P}(f)$. We relaxed linear program is denoted by LP_1. The main difference from Eq. (2) is that variable y_f^p is not integral, but fractional in LP_1. Since LP_1 is a linear program, we can solve it in polynomial time with a linear program solver. Assume that the optimal solution for LP_1 is denoted by $\widetilde{y}$, and the optimal result is denoted by $\widetilde{\lambda}$. As LP_1 is a relaxation of the LBR-FT problem, $\widetilde{\lambda}$ is a lower-bound result for LBR-FT. Using the randomized rounding method [21], we obtain an integer solution $\widehat{y_f^p}$. More specifically, variable $\widehat{y_f^p}$, with $p \in \mathcal{P}(f)$, is set as 1 with the probability of $\widetilde{y_f^p}$ while satisfying $\sum_{p\in\mathcal{P}(f)} \widehat{y_f^p} = 1, \forall f \in \Pi$. If $\widehat{y_f^p} = 1, \exists p \in \mathcal{P}(f)$, this means that flow f selects $p \in \mathcal{P}'_f$ as its route. By this way, we have determined the route for each flow in Π. The RDSR algorithm is formally described in Algorithm 1.

Algorithm 1 RDSR: Rounding-Based Scalable Routing

```
1: **Step 1: Solving the Relaxed LBR-FT Problem**
2: Build up a feasible path set $\mathcal{P}(f)$ for flow $f \in \Pi$
3: $T(s)$ is the number of residual flow entries on switch $s$
4: Construct a linear program in $LP_1$
5: Obtain the optimal solution $\widetilde{y}$
6: **Step 2: Route Selection for Load Balancing**
7: Derive an integer solution $\widehat{y_f^p}$ by randomized rounding
8: for each sampled flow $f \in \Pi$ do
9:   for each route $p \in \mathcal{P}(f)$ do
10:    if $\widehat{y_f^p} = 1$ then
11:      Appoint a path $p$ for flow $f$
12:    end if
13:  end for
14: end for
```

Now, we illustrate our randomized rounding method. Suppose that there are four candidate paths $(p_1, p_2, p_3, p^*(f))$ in set $\mathcal{P}(f)$. The final solutions for the situation are denoted by $\{0.2, 0.2, 0.3, 0.3\}$. This means that the interval [0, 1] has been divided into four parts, (0, 0.2], (0.2, 0.4], (0.4, 0.7], and (0.7, 1.0]. We then randomly choose a value from 0 to 1. Assume that the random value is 0.3. Since this value lies in (0.2, 0.4], the second feasible path p_2 will be chosen for flow. If the random value is 0.8, the fourth one (*i.e.*, the current route $p^*(f)$) will be selected.

We analyze the approximate performance of the proposed RDSR algorithm. Assume that the minimum capacity of all the links is denoted by $c_{\min}$. We define a variable α as follows:

$$\alpha = \min\{\min\{\frac{\widetilde{\lambda} c_{\min}}{r(f)}, f \in \Pi\}, \min\{T(s), s \in S\}\} \tag{3}$$

By observing the practical network traces, we find that $\alpha \ge 1$ in most situations. Since RDSR is a randomized algorithm, we compute the expected traffic load on links and the expectation of required flow entries on switches. We give the approximation performance of the proposed algorithm.

Lemma 2: The proposed RDSR algorithm can achieve the approximation factor of $\frac{4 \log n}{\alpha} + 4$ for link capacity constraints.

The proof of Lemma 2 has been relegated to the appendix.

Similarly, we can obtain the approximation factor for the flow-table size constraint.

Lemma 3: After the rounding process, the amount of required flow entries on any switch s will not exceed the number of residual flow entries $T(s)$ by a factor of $\frac{3\log n}{\alpha}+3$.

Approximation factor: Following from our analysis, by routing a full percentage of flows on each chosen path, the capacity of links will hardly be violated by a factor of more than $\frac{4\log n}{\alpha}+4$, and the flow table size constraints will not be violated by a factor of $\frac{3\log n}{\alpha}+3$. It means that the algorithm can achieve the optimal solution, violating the link capacity constraint by at most a factor $\frac{4\log n}{\alpha}+4$ and the flow table size constraint by at most a factor $\frac{3\log n}{\alpha}+3$, which is also called as bi-criteria approximation. By using the traffic controlling method, the intensity of all the flows can be limited to specific values. Thus, we scale all flows by a factor of $\frac{4\log n}{\alpha}+4$ to satisfy the link capacity constraints. As to the cases where the flow table size constraints are violated, the default flow entries will take effect to transfer these extra flows, avoiding packets dropping.

Moreover, we want to address that, in most situations, the RDSR algorithm can reach almost the constant bi-criteria approximation. For example, let $\widetilde{\lambda}$ be 0.4 (with a moderate value). Assume there is a large-scale network with $n = 1000$ switches, so that $\log n \approx 10$. The link capacity of today's networks will be a bandwidth of 1Gbps at least. Observing the practical flow traces, the maximum intensity of a flow may reach 1Mbps or 10Mbps. Under two cases, $\frac{c_{\min}}{r(f)}$ will be 10^3 and 10^2. The approximation factor for the link capacity constraint is 4.1 and 5, respectively. Since $T(s)$ is usually at least 10^3, the approximation factor for the flow table constraint is 3.03. In other words, our RDSR algorithm can achieve almost the constant bi-criteria approximation for the LBR-FT problem in many network situations.

2) Complete Description for the RDSR Algorithm: Though the RDSR algorithm will obtain the bi-criteria approximate performance for the LBR-FT problem, its randomized rounding method does not guarantee that the flow-table size constraint is always met. Below we give the complete description for RDSR, which can always satisfy the forwarding table size constraint.

The complete RDSR algorithm consists of three steps. The former two steps are same as those in the basic RDSR algorithm, described in Section IV-C.1. By the second step, we assume that the number of required flow entries on each switch s is denoted by $T'(s)$. Since some switches may violate the flow-table size constraints, the third step will choose some flows to be forwarded through traditional routes to satisfy the forwarding table size constraint on each switch. Intuitively speaking, let $\widetilde{x}_f$ denote the probability that flow f will choose a route path different from the current one. Thus, $\widetilde{x}_f = \sum_{p\in\mathcal{P}(f)-\{p^*(f)\}} \widetilde{y}_f^p$. We rank all the sampled flows by the increasing order of $\widetilde{x}$. Without loss of generality, we assume that $\widetilde{x}_{f_1} \le \ldots \le \widetilde{x}_{f_r}$. Then, the algorithm checks all the flows one by one. For each flow f, assume that its re-routed path is p. We just consider the case in which its re-routed path is not the traditional path. If there is one switch v on path p has violated the forwarding table size constraint, we will not re-route this flow. That is, this flow will be forwarded through the traditional route. Then, we also update the number of required flow entries on these switches. After we finish checking all the sampled flows (or the forwarding table size constraints on all switches are satisfied), the algorithm terminates. The complete RDSR algorithm is described in Algorithm 2.

Algorithm 2 Complete Description for the RDSR Algorithm

```
1: **Step 1: Same as that in Algorithm 1**
2: **Step 2: Same as that in Algorithm 1**
3: **Step 3: Route Selection with Flow-Table Size Constraint**
4: Assume that the number of required flow entries on $s$ is $T'(s)$
5: for each flow $f \in \Pi$ do
6:   $\widetilde{x}_f = \sum_{p\in\mathcal{P}(f)-\{p^*(f)\}} \widetilde{y}_f^p$
7: end for
8: Rank all the flows in the increasing order of $\widetilde{x}$. Assume that $\widetilde{x}_{f_1} \le \ldots \le \widetilde{x}_{f_r}$
9: for each flow $f \in \Pi$ do
10:   Assume that its route is denoted by $p$
11:   if $T'(s) \ge T(s)$ and $I(f,p,s) = 1, \exists s \in p$ then
12:     Assign the traditional route for flow $f$
13:     for each switch $v \in p$ do
14:       $T'(s) = T'(s) - I(f,p,s)$
15:     end for
16:     continue;
17:   end if
18: end for
```

Now, we discuss the time complexity of the complete RDSR algorithm. The first step mainly solves the linear program. Since the linear program LP_1 contains polynomial number of variables, it takes polynomial times to solve this linear program. The second step uses randomized rounding for route selection, and its time complexity is $\Delta \cdot r$, where Δ is the maximum number of feasible paths for all flows and $r = |\Pi|$. In the third step, the algorithm computes a weight for each flow, and its time complexity $\Delta \cdot r$ too. Then, for each flow, the algorithm will check the states of all switches on its selected route, which takes a time complexity of $\delta \cdot r$, where δ is the maximum hop number of each feasible path. As a result, the total time complexity of RDSR is polynomial.

V. Numerical Evaluation

A. Performance Metrics and Methodology

We evaluate the proposed hybrid switching (HS) through simulations and testbed implementation, and compare it with the most-related, state-of-the-art work of DevoFlow [5]. Both of them try to improve the scalability of a large SDN system, but their design goals are somewhat different: The main goal of HS is to address the problem of limited forwarding-table size and to reduce the overhead of the controller. The main goal of DevoFlow is to reduce the overhead of the controller, while achieving high network performance. Both DevoFlow and HS are flexible in what algorithms to use in re-routing and computing default paths, even though the paper gives an RDSR re-routing algorithm for the load-balancing case study. In fair comparison, we apply RDSR and shortest paths to both DevoFlow and HS. In short, re-routing elephant flows is done through RDSR, and ECMP rules are constructed to follow the available shortest paths. When RDSR is applied for DevoFlow, $I(p,f,s) = 1$, with $s \in p$ and $p \in \mathcal{P}(f)$.

We use the following performance metrics in our numerical evaluation: (1) the maximum number of forwarding rules (or flow entries needed) on any switch at any time during the simulation. To better describe the usage of flow entries, we also observe the cumulative distribution function (CDF) of the flow entries under a fixed number of flows in a network; (2) the maximum throughput of the entire network; (3) the maximum load factor for load balancing, and the load factor CDF of all links; (4) the communication traffic volume to/from the controller. During a simulation run, at each time instance, we measure the maximum and median number of forwarding rules per switch, and the average number of forwarding rules on any switch. We use their largest values over time during the simulation as the first four metrics. The number of routing entries in HS and the number of ECMP rules in DevoFlow are the same at each switch, but each routing entry is shorter and stored in SRAM, whereas each wildcard rule is longer and stored in TCAM. Our evaluation does not consider this "constant" overhead, which is highly dependent on the network size, although such comparison would be in favor of HS. As we continuously increase the number of flows, we measure the maximum throughput that the network can support. The load factor of a link is the traffic load divided by the link capacity. The load balancing metric is the maximum load factor among all links. Moreover, we compute the load factor CDF of all links. The simulator measures the total communication traffic to/from the controller, divided by the time period of simulation, which gives the fifth metric.

B. Simulation Evaluation

1) Simulation Settings: Our simulations are executed on the Mininet [22], which is a widely-used simulator specified for SDN. We choose a two-dimensional HyperX topology [23] in the simulation. The HyperX topology forms a 9×9 grid, and so has 81 access switches, each attached to 16 other switches. All links are 1Gbps, and 20 servers will be attached to each access switch. So, the HyperX topology has 1620 servers. We use power law for the flow-size distribution, where 20% of all flows account for 80% of traffic volume [13]. The average flow intensities of elephant flows and mice flows are set as 4Mbps and 0.25Mbps, respectively. We generate three types of flows: (1) random flows, whose sources and destinations are randomly picked; (2) server flows, which simulate the traffic between random hosts and a number of designated servers, *e.g.*, mail servers and web servers; (3) associative flows, which simulate the traffic between a subnet and a server, *e.g.*, communications between the finance department and the finance database or between a hospital and a datacenter that houses the patient data. Curtis *et al.* [5] have shown that the overhead of the sampling method is less than that of the push-based method for flow statistics collection. For fairness, both HS and DevoFlow use the sampling method and each switch will report the traffic estimation information of the largest 1,500 (elephant) flows at most to the controller.

The simulations are performed under two scenarios. The first scenario has no forwarding-table size (FTS) constraint, assuming that the switches have sufficient space to handle all flows. This hypothetical scenario tests the performance of HS (DevoFlow) when the table size is not a limiting issue. The second scenario has an FTS constraint and tests how well HS (DevoFlow) performs when the table size becomes a problem.

2) Performance Comparison Without FTS Constraint: Our first set of simulations compares HS and DevoFlow in the scenario without FTS constraint. The results are shown in Figures 4-9, where the horizontal axis is the number of flows in the network, ranging from 10×10^4 to 80×10^4. Figure 4 shows the maximum number of flow entries needed by two algorithms. As the number of flows increases, there are more elephant flows as well. As a result, the maximum number of flow entries (or forwarding rules) increase in both HS and DevoFlow. In comparison, the proposed HS solution uses much fewer flow entries than DevoFlow. For example, when there are 50×10^4 flows (about 300 flows per server), HS uses a maximum number of 600 flow entries

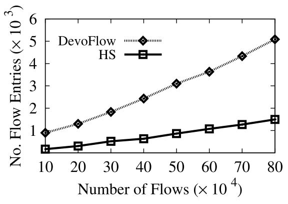


Fig. 4. Maximum number of flow entries vs. number of flows without FTS constraint.

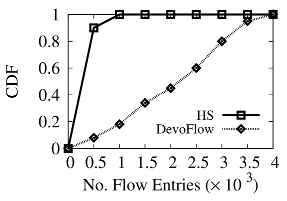


Fig. 5. The CDF of flow entries among all switches without FTS constraint.

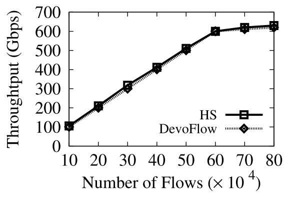


Fig. 6. Network throughput vs. number of flows without FTS constraint.

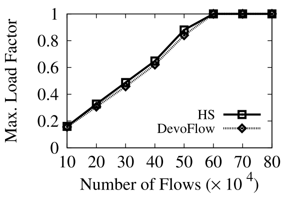


Fig. 7. Max. load factor vs. number of flows without FTS constraint.

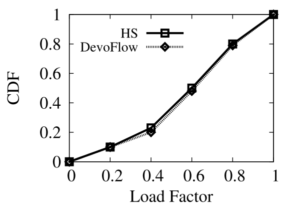


Fig. 8. The CDF of load factors without FTS constraint.

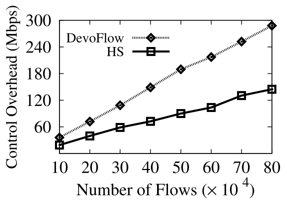


Fig. 9. Control overhead vs. number of flows without FTS constraint.

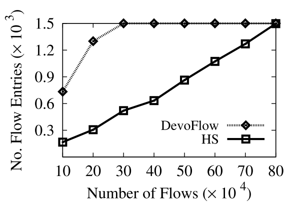


Fig. 10. Maximum number of flow entries vs. number of flows with FTS constraint.

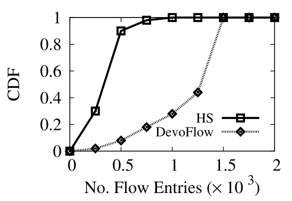


Fig. 11. The CDF of flow entries among all switches with FTS constraint.

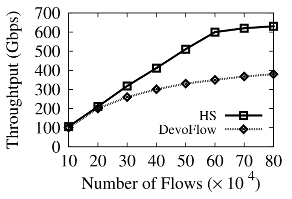


Fig. 12. Network throughput vs. number of flows with FTS constraint.

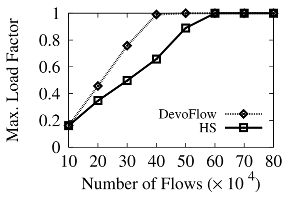


Fig. 13. Max. load factor vs. number of flows with FTS constraint.

on a switch, while DevoFlow uses 3,100. Figure 5 shows the CDF of flow entries under a fixed number (*e.g.*, 60×10^4) of flows. From this figure, over 90% of switches need less than 500 flow entries by HS, while over 80% of switches about 1,000-4,000 flow entries by DevoFlow. More specifically, HS reduces the required flow entries by about 80.5% compared with DevoFlow. That's because our hybrid path deployment mechanism helps to reduce the required flow entries by combining traditional/SDN routing.

Figures 6-8 show that HS and DevoFlow achieve similar network performance, including throughput and load balancing. The reason is that without FTS constraint, both designs can dynamically schedule the elephant flows for efficient routing. Figure 9 shows that HS has much smaller communication overhead at the controller than DevoFlow. As the number of flows increases, DevoFlow deploys more forwarding rules than HS, which results in higher control overhead. For example, when there are 80×10^4 flows, the control overheads of HS and DevoFlow are about 140Mbps and 290Mbps, respectively.

3) Performance Comparison With FTS Constraint: The second set of simulations compares HS and DevoFlow with FTS constraint, where the forwarding table size is set to 1,500 entries [5]. The results are shown in Figures 10-15, where the horizontal axis is the number of flows in the network. In Figure 10, HS needs much fewer flow entries than DevoFlow, whose table is saturated much earlier. As an example, when there are 50×10^4 flows in the network, HS uses

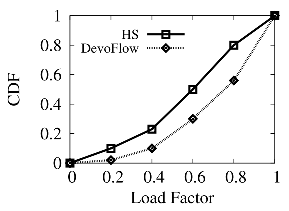


Fig. 14. The CDF of load factors among all links with FTS constraint.

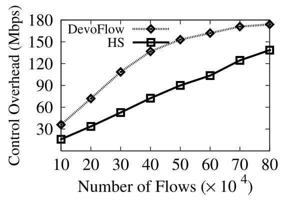


Fig. 15. Control overhead vs. number of flows with FTS constraint.

a maximum number of 800 flow entries in any switch, while DevoFlow uses 1,500 (with the forwarding table becoming full). Under a fixed number (*e.g.*, 60×10^4) of flows, we plot the CDF of flow entries in Figure 11. This Figure shows that, over 90% of switches need less than 500 flow entries by our HS mechanism, while about 70% of switches need 1,000-1,500 flow entries by DevoFlow. This saturation has a performance impact, as shown below.

When the number of flows is less than 30×10^4, the forwarding table is not made full by DevoFlow. In this case, HS and DevoFlow achieve similar network performance, including throughput and load balancing, as shown in Figures 12-14. On the contrary, when the number of flows is more than 30×10^4 and the forwarding table is made full by DevoFlow, HS outperforms DevoFlow because the routing flexibility of the latter is constrained. For example, when there are 80×10^4 flows in the network, HS improves network throughput by 63% when comparing with DevoFlow by Figure 12. From Figure 13, when the number of flows exceeds 60×10^4, the maximum load factor is close to 1 for both DevoFlow and HS. Figure 14 plots the CDF of load factor by different algorithms under the same case with 60×10^4 flows. This figure shows that our proposed HS method can achieve better route performance than DevoFlow. In other words, DevoFlow will make more flows congested compared with HS. As a result, HS can forward more flows than DevoFlow, which is also validated by Figure 12. Since HS deploys a fewer number of forwarding rules than DevoFlow, it incurs smaller control overhead, as shown in Figure 15. When there are more and more flows in a network, due to the constraint of forwarding table size, the numbers of deployed rules on each switch by DevoFlow and HS are very close, and the overheads of two solutions are very close too.

4) Robustness Comparison With FTS Constraint: This section mainly shows the applicability and robustness of our proposed algorithm. We first observe the throughput performance of DevoFlow and HS by changing two network parameters, *e.g.*, the number of available flow entries and the traffic scheme. When the number of flow entries is increasing, the network throughput for both algorithm is also increased, and the increasing ratio of network throughput is much slower, as shown in Figure 16. That's because the SDN network can provide more chance for flow re-rerouting with more flow entries. This figure shows that our HS method can improve the network throughput 15-55% compared with

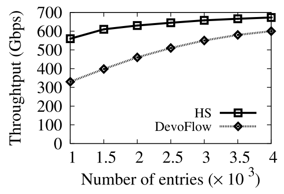


Fig. 16. Network throughput vs. number of entries with FTS constraint.

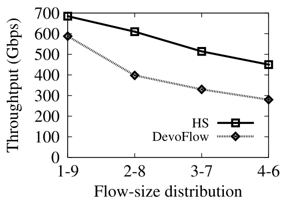


Fig. 17. Network throughput vs. different traffic schemes with FTS constraint.

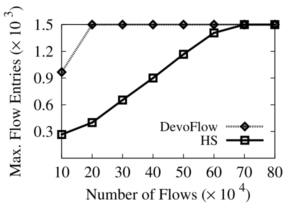


Fig. 18. Maximum number of flow entries vs. number of flows with FTS constraint on fat-tree.

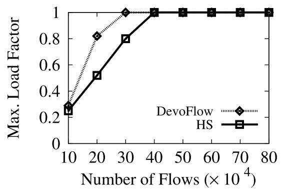


Fig. 19. Max. load factor vs. number of flows with FTS constraint on fat-tree.

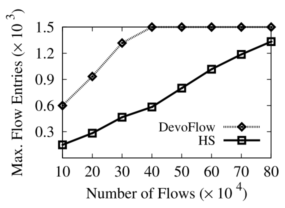


Fig. 20. Max. number of entries vs. number of flows with FTS constraint using ECMP.

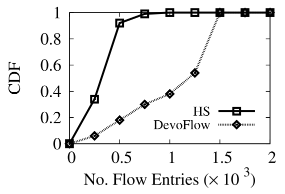


Fig. 21. The CDF of flow entries among all switches with FTS constraint using ECMP.

DevoFlow under a various number of flow entries on each switch. Figure 17 shows the network throughput of two algorithms by changing the network traffic schemes. For example, 1-9 denotes that top 10% of all flows account for 90% of traffic volume. This figure shows that our HS method can perform much better than DevoFlow with a lighter traffic skew (*e.g.*, 4-6 distribution). However, even in an extremely higher traffic skew, HS can improve the network throughput 17% compared with DevoFlow.

We then observe different performance metrics (*e.g.*, maximum number of flow entries and maximum load factor) on another network topology. We adopt a widely used datacenter topology, fat-tree [24], which contains 80 switches and 128 servers. Figure 18 shows that HS needs much fewer flow entries than DevoFlow. As an example, when there are 20×10^4 flows, HS uses a maximum number of 430 flow entries in any switch, while DevoFlow uses 1,500 (with the forwarding table becoming full). Moreover, when the number of flows reaches 70×10^4, HS uses a maximum number of 1,500 flow entries. With more and more flows, Figure 19 shows that the maximum load factor is increasing for both algorithms. Moreover, HS can reduce the maximum load factor by about 25%-30% compared with DevoFlow, except that the maximum load factor is 1 for both DevoFlow and HS with more than 40×10^4 flows.

In the above simulations, we assume that the controller uses the OSPF method for default paths. We finally evaluate the number of needed flow entries when another default path scheme, *e.g.*, ECMP, is adopted on HyperX. In Figure 20, HS needs much fewer flow entries than DevoFlow, whose table is saturated much earlier (with 40×10^4 flows). Even with 80×10^4 flows, HS requires 1,360 flow entries at most on any switch. Under a fixed number (*e.g.*, 60×10^4) of flows, we plot the CDF of flow entries in Figure 21. This figure shows that, over 90% of switches need less than 500 flow entries by our HS mechanism, while about 60% of switches need more than 1,000 flow entries by DevoFlow.

Our evaluation is performed under two scenarios: In the first scenario, we do not set a limit on the number of rules that each switch can hold. As expected, the results show that DevoFlow and HS have similar performance in load balancing, while HS requires much fewer rules due to its hybrid path deployment. In the second scenario, we set a certain limit on the number of rules. This makes sense because many SDN switches are designed to hold all OpenFlow forwarding rules in TCAM. As we vary this limit, the results consistently show that HS outperforms DevoFlow. The reason is that HS requires fewer rules to re-route elephant flows (again thanks to hybrid path deployment). Therefore, HS can re-route more elephant flows and thus have better performance. Our simulation results also reveal the better robustness of HS.

C. Testbed Evaluation

1) Implementation on the Platform: We implement the hybrid switching and flow re-routing solutions on the commodity hardwares. The topology of our SDN platform is illustrated in Figure 22. Our testbed is composed of three parts: a server installed with the controller's software, a set of OpenFlow enabled switches and some terminals. Specifically, we choose Opendaylight, which is an open source project supported by multiple enterprises, as the controller's software. The controller is running on a server with a core i5-3470 processor and 4GB of RAM. We build the forwarding plane of an SDN with 6 H3C S5120-28SC-HI switches, which support the OpenFlow v1.3 standard. We use 4 laptops as terminals, which will generate hundreds of flows in a network. In this figure, the solid link denotes the data link between two switches, and the dashed link denotes the control link between a switch and the controller. By default, the capacity of each data link is 100Mbps.

In the system implementation, each flow is identified as three elements, source IP, destination IP and source port, so that the system is able to generate hundreds of flows in

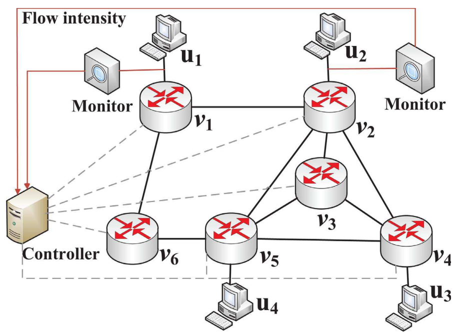


Fig. 22. Topology of the SDN platform. Our platform is mainly composed of three parts: a controller, six OpenFlow enabled switches and four terminals.

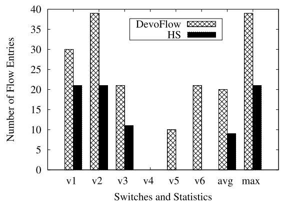


Fig. 23. Number of flow entries needed on different switches.

the network with only four terminals. If we do not include source port for flow identification, tens of terminals are needed so as to generate hundreds of flows, which makes the SDN platform more complex. Under our system framework, each ingress switch should be able to estimate the flow intensity using the packet sampling. However, since the commodity switch's software (*e.g.*, firmware) is not open for us, we cannot add the extra software module on each switch under the current phase. Thus, to sample the packets and estimate the flow intensity, we deploy a traffic monitor between a sender terminal and its ingress switch using the "mirroring" mechanism, so that the monitor can estimate the real-time flow intensity, and report to the controller periodically. Then, the controller records the information of those reported elephant flows, designs the efficient routes for those (elephant) flows, and deploys feasible rules on forwarding tables. We implement two different solutions, HS and DevoFlow, on the controller. Each of two algorithms requires two flow tables. However, our commodity switch is only equipped with one flow table, *i.e.*, the forwarding table. To accommodate this constraint, we deploy rules of both traditional routes and SDN routes in a single forwarding table, but they will be assigned with different priorities. More specifically, the priority of SDN routes is higher than that of traditional routes, so that the controller will first match SDN route rules with higher priority before traditional route rules with lower priority.

2) Testing Results: We will discuss the testing results of both HS and DevoFlow.

Flow entry requirement: In the system, two terminals u_1 and u_2 will totally generate 300 flows, and forward to two terminals u_3 and u_4, respectively. According to the 2-8 distribution rule, there are 60 large flows (or elephant flows) and 240 small flows (or mice flows). The average flow intensities of the elephant flows and the mice flows are 1.5Mbps and 150kbps, respectively. We also set the traditional route for each pair of terminals. For example, the traditional route from u_1 to u_3 is $u_1 - v_1 - v_2 - v_4 - u_3$. We mainly observe the number of required flow entries for SDN routes among all switches by different algorithms. Figure 23 shows that our HS solution needs less flow entries than the DevoFlow method. More specifically, the maximum number of flow entries is reduced from 39 by DevoFlow to 21 by HS. That is, our solution can reduce the maximum number of required flow entries about 46.1% compared with the DevoFlow method. Note that, since v_1 and v_2 are two ingress switches of these flows, they require more flow entries for SDN routes compared with other switches. That's because, the controller will schedule these flows on different paths for load balancing with global traffic information. As a result, other switches may burden a subset of flows, and thus require less flow entries compared with two ingress switches. For the average case, HS and DevoFlow require 7 and 20 flow entries, respectively, on all the switches.

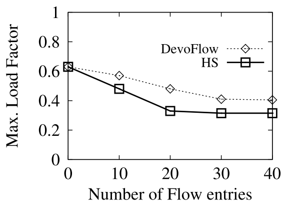


Fig. 24. Max. load factor on a testbed.

Route performance: We then observe the route performance of load balancing by changing the number of flow entries for SDN route rules on each switch. Since the DevoFlow and HS solutions need 39 and 21 flow entries at most for these 300 flows, we change the number of available flow entries from 0 to 40 with interval 10 flow entries. When the number of available flow entries for SDN routes is 0, it means that all the flows will be forwarded by the traditional routes. Accordingly, the maximum load factor is high (*e.g.*, 0.63). With the increase of flow entries, since some flows can be re-routed by setting up flow entries for SDN routes, the maximum load factor is decreased for both two solutions, shown in Figure 24. When there are 10, 20, 30 and 40 flow entries for SDN routes in a switch, our HS solution can reduce the maximum load factors about 15.8%, 31.2%, 23.6% and 22.2%, respectively.

VI. Related Work

SDN was introduced to improve network performance through centralized control [2]. For example, the centralized traffic engineering service of B4 [3] is able to drive links to near 100% utilization, with load balancing among alternative paths. These routing methods try to maximize the network throughput by different ways, such as linear program, route update, etc. However, the limited size of the forwarding table and the communication/computation/management load at the controller place constraint on the scalability of such a centralized design.

To address these problems, the prior art mainly uses wildcards [25]. The Plug-n-Serve system [26] uses wildcards to aggregate multiple flows into a single rule (occupying one forwarding entry). Their use of wildcards is limited to the suffixes of source address. But as each wildcard rule bundles many flows together based on their address adjacency, it partially compromises the SDN's flexibility of differentiating arbitrary individual flows in traffic engineering. Moreover, compression of SDN rules using wildcards is also discussed in [27].

Hedera [28] is designed to re-route large flows in a datacenter network that has a structured topology such as fat tree [24] to provide probabilistic default path selection among multiple alternative choices. DevoFlow [5] is more general for arbitrary network topologies. It combines pre-deployed wildcard rules and dynamically-established exact rules, *with a design goal of reducing the need to involve the controller in setting up paths for new flows*. Limited by the small size of TCAM, the number of wildcard rules is small, and each wildcard rule may have to match numerous flows. In order to differentiate individual flows (so as to measure their individual sizes and re-route them as needed for load balancing), when a new flow matches a wildcard rule, an exact rule specifically for that flow will be created based on the template of the wildcard rule, without involving the controller. Though DevoFlow addresses the problem of limited table size, it installs exact rules on switches along its route path, so that it can not provide per-flow control for much more flows, which has been validated by our simulation in Section V. Moreover, the paper of DevoFlow has no discussion on how to construct the wildcard rules, in particular, ones that can cover all possible flows at all switches throughout the network.

Cohen *et al.* [6] study the effect of forwarding-table size on network utilization. They formulate this problem as an NP-hard optimization problem and present approximate algorithms. They assume that when a switch's forwarding table is full, new flows will simply be dropped. Huang *et al.* [29] consider splittable flows, each of which is allowed to follow multiple paths to improve network utilization. They study joint optimization of rule placement and traffic engineering for QoS provisioning. Since this paper assumes that each flow is split into several sub-flows, the rule multiplexing scheme is only applied among these sub-flows so as to save the flow entries. However, when applied to the unsplittable flows, this scheme can not help to reduce the flow entries. Our paper considers unsplittable flows, such as TCP flows whose window adaptation may be adversely affected if packets of the same flow follow different paths.

VII. Conclusion

In this paper, we have designed a novel hybrid switching mechanism, which integrates traditional switching and SDN switching for the purpose of achieving both scalability and optimal performance. Moreover, a hybrid path deployment method has been presented to reduce the required forwarding rules. As a case of applications, load-balancing re-routing is also studied. Testing and numerical evaluation results demonstrate the superior performance of hybrid switching when comparing with the DevoFlow solution.

Appendix
Proof of Lemma 2

We give two famous lemmas for probability analysis.

Lemma 4 (Chernoff Bound): Given n independent variables: $x_1, x_2, \ldots, x_n$, where $\forall x_i \in [0,1]$. Let $\mu = \mathbb{E}[\sum_{i=1}^{n} x_i]$. Then, $\mathbf{Pr}\left[\sum_{i=1}^{n} x_i \geq (1+\epsilon)\mu\right] \leq e^{\frac{-\epsilon^2 \mu}{2+\epsilon}}$, where ϵ is an arbitrarily positive value.

Lemma 5 (Union Bound): Given a countable set of n events: $A_1, A_2, \ldots, A_n$, each events A_i happens with possibility $\mathbf{Pr}(A_i)$. Then, $\mathbf{Pr}(A_1 \cup A_2 \cup \ldots \cup A_n) \leq \sum_{i=1}^{n} \mathbf{Pr}(A_i)$.

Proof: We first bound the probability with which the link capacities will be violated. Note that, after the linear program procedure in step 1 of the RDSR algorithm, we derive a fractional solution $\widetilde{y_f^p}$ and an optimal result $\widetilde{\lambda}$ for the relaxed LBR-FT problem. Using the randomized rounding method, for each flow $f \in \Pi$, only one of the paths in $\mathcal{P}(f)$ will be chosen as its route path. Thus, the traffic load of link e from flow f could be defined as a random variable $x_{f,e}$, which equals $r(f)$ with possibility $\sum_{e\in p: p\in\mathcal{P}(f)} \widetilde{y_f^p}$.

Definition 1: For each sampled flow $f \in \Pi$ and each link $e \in E$, a random variable $x_{f,e}$ is defined as:

$$x_{f,e} = \begin{cases} r(f), & \textit{with probability of } \sum_{e\in p:p\in\mathcal{P}(f)} \widetilde{y_f^p} \\ 0, & otherwise. \end{cases} \quad (4)$$

According to the definition, $x_{f_1,e}, x_{f_2,e}, \ldots$ are mutually independent. The expected traffic load on link e is:

$$\begin{aligned}\mathbb{E}\left[\sum_{f\in\Pi} x_{f,e}\right] &= \sum_{f\in\Pi} [x_{f,e}] \\ &= \sum_{f\in\Pi} \sum_{e\in p:p\in\mathcal{P}(f)} \widetilde{y_f^p} \cdot r(f) \\ &\leq \widetilde{\lambda} c(e) - g(e) \leq \widetilde{\lambda} c(e) \end{aligned} \quad (5)$$

Combining Eq. (5) and the definition of α in Eq. (3), we have

$$\begin{cases} \dfrac{x_{f,e}\cdot\alpha}{\widetilde{\lambda} c(e)} \in [0,1] \\ \mathbb{E}\left[\sum_{f\in\Pi} \dfrac{x_{f,e}\cdot\alpha}{\widetilde{\lambda}\cdot c(e)}\right] \leq \alpha. \end{cases} \quad (6)$$

Then, by applying Lemma 4, assume that ρ is a arbitrary positive value. It follows

$$\begin{aligned} &\mathbf{Pr}\left[\sum_{f\in\Pi} \frac{x_{f,e}\cdot\alpha}{\widetilde{\lambda}\cdot c(e)} \leq (1+\rho)\cdot\alpha\right] \leq e^{\frac{-\rho^2\cdot\alpha}{2+\rho}} \\ &\quad \Leftrightarrow \mathbf{Pr}\left[\sum_{f\in\Pi} \frac{x_{f,e}}{\widetilde{\lambda}\cdot c(e)} \leq (1+\rho)\right] \leq e^{\frac{-\rho^2\cdot\alpha}{2+\rho}} \end{aligned} \quad (7)$$

Now, we would assume that

$$\mathbf{Pr}\left[\sum_{f\in\Pi} \frac{x_{f,e}}{\widetilde{\lambda}\cdot c(e)} \leq (1+\rho)\right] \leq e^{\frac{-\rho^2\cdot\alpha}{2+\rho}} \leq \frac{\mathcal{F}}{n^2} \quad (8)$$

where $\mathcal{F}$ is the function of network-related variables (such as the number of switches n, etc.) and $\mathcal{F} \rightarrow 0$ when the network grows.

By solving Eq. (8), we have the following result:

$$\rho \geq \frac{\log\frac{n^2}{\mathcal{F}} + \sqrt{\log^2\frac{n^2}{\mathcal{F}} + 8\alpha\log\frac{n^2}{\mathcal{F}}}}{2\alpha}, \quad n \geq 2 \quad (9)$$

Set $\mathcal{F} = \frac{1}{n^2}$. Eq. (8) is transformed into:

$$\mathbf{Pr}\left[\sum_{f\in\Pi} \frac{x_{f,e}}{\widetilde{\lambda}\cdot c(e)} \leq (1+\rho)\right] \leq \frac{1}{n^4}, \quad \text{where } \rho \geq \frac{4\log n}{\alpha} + 2 \quad (10)$$

By applying Lemma 5, we have,

$$\begin{aligned}
&\mathbf{Pr}\left[\bigvee_{e\in E}\{\sum_{f\in\Pi}\frac{x_{f,e}}{\widetilde{\lambda}\cdot c(e)}+\frac{g(e)}{\widetilde{\lambda}\cdot c(e)}\}\leq(2+\rho)\right]\\
&\quad\leq\mathbf{Pr}\left[\bigvee_{e\in E}\sum_{f\in\Pi}\frac{x_{f,e}}{\widetilde{\lambda}\cdot c(e)}\leq(1+\rho)\right]\\
&\quad\leq\sum_{e\in E}\mathbf{Pr}\left[\sum_{f\in\Pi}\frac{x_{f,e}}{\widetilde{\lambda}\cdot c(e)}\leq(1+\rho)\right]\\
&\quad\leq n^2\cdot\frac{1}{n^4}\\
&\quad=\frac{1}{n^2},\quad\rho\geq\frac{4\log n}{\alpha}+2
\end{aligned}\tag{11}$$

Note that the third inequality holds, because in a network of n nodes, the number of links would not exceed n^2. The approximate factor of our algorithm is $\rho+2=\frac{4\log n}{\alpha}+4$. $\square$

REFERENCES

[1] H. Xu, H. Huang, S. Chen, and G. Zhao, "Scalable software-defined networking through hybrid switching," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, May 2017, pp. 1–9.

[2] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Apr. 2008.

[3] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined WAN," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 3–14, 2013.

[4] S. Agarwal, M. Kodialam, and T. V. Lakshman, "Traffic engineering in software defined networks," in *Proc. IEEE INFOCOM*, Apr. 2013, pp. 2211–2219.

[5] A. R. Curtis *et al.*, "DevoFlow: Scaling flow management for high-performance networks," *Comput. Commun. Rev.*, vol. 41, no. 4, pp. 254–265, Aug. 2011.

[6] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz, "On the effect of forwarding table size on SDN network utilization," in *Proc. IEEE INFOCOM*, Apr./May 2014, pp. 1734–1742.

[7] X. Jin *et al.*, "Dynamic scheduling of network updates," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 539–550.

[8] D. Li, Y. Shang, and C. Chen, "Software defined green data center network with exclusive routing," in *Proc. IEEE INFOCOM*, Apr./May 2014, pp. 1743–1751.

[9] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended bloom filter: An aid to network processing," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 4, pp. 181–192, 2005.

[10] J. Moy, *OSPF Version 2*, document RFC 2328, 1997.

[11] Y. Rekhter, T. Li, and S. Hares, *A Border Gateway Protocol 4 (BGP-4)*, document RFC 1771, 2005.

[12] K. Kannan and S. Banerjee, "Compact TCAM: Flow entry compaction in TCAM for power aware SDN," in *Proc. Int. Conf. Distrib. Comput. Netw.*, 2013, pp. 439–444.

[13] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements & analysis," in *Proc. 9th ACM SIGCOMM Conf. Internet Meas. Conf.*, 2009, pp. 202–208.

[14] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proc. ACM SIGCOMM Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2012, pp. 323–334.

[15] *OpenFlow Switch Specification-Version 1.4.0*, Open Netw. Found., 2013.

[16] T. Li, S. Chen, and Y. Ling, "Fast and compact per-flow traffic measurement through randomized counter sharing," in *Proc. IEEE INFOCOM*, Apr. 2011, pp. 1799–1807.

[17] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *Proc. IEEE INFOCOM*, Apr. 2011, pp. 1629–1637.

[18] H. Xu *et al.*, "Incremental deployment and throughput maximization routing for a hybrid SDN," *IEEE/ACM Trans. Netw.*, vol. 25, no. 3, pp. 1861–1875, Jun. 2017.

[19] S. Even, A. Itai, and A. Shamir, "On the complexity of time table and multi-commodity flow problems," in *Proc. IEEE 16th Annu. Symp. Found. Comput. Sci*, Oct. 1975, pp. 184–193.

[20] C.-Y. Hong *et al.*, "Achieving high utilization with software-driven WAN," in *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 15–26, 2013.

[21] P. Raghavan and C. D. Tompson, "Randomized rounding: A technique for provably good algorithms and algorithmic proofs," *Combinatorica*, vol. 7, no. 4, pp. 365–374, 1987.

[22] *The Mininet Platform*. Accessed: May 2016. [Online]. Available: http://mininet.org/

[23] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber, "HyperX: Topology, routing, and packaging of efficient large-scale networks," in *Proc. Conf. High Perform. Comput. Netw., Storage Anal.*, 2009, pp. 1–11.

[24] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 63–74, 2008.

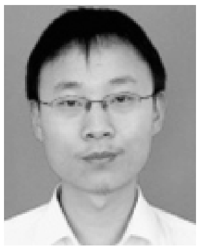
[25] R. Wang, D. Butnariu, and J. Rexford, "OpenFlow-based server load balancing gone wild," *Hot-ICE*, vol. 11, p. 12, Mar. 2011.

[26] N. Handigol, S. Seetharaman, N. McKeown, and R. Johari, "Plug-n-serve: Load-balancing Web traffic using openFlow," *ACM SIGCOMM Demo*, vol. 4, no. 5, pp. 1–2, 2009.

[27] M. Rifai *et al.*, "Too many SDN rules? Compress them with MINNIE," in *Proc. IEEE Global Commun. Conf.*, Dec. 2015, pp. 1–7.

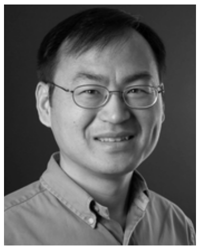
[28] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proc. NSDI*, vol. 10. 2010, p. 19.

[29] H. Huang, S. Guo, P. Li, B. Ye, and I. Stojmenovic, "Joint optimization of rule placement and traffic engineering for QoS provisioning in software defined network," *IEEE Trans. Comput.*, vol. 64, no. 12, pp. 3488–3499, Dec. 2015.



Hongli Xu (M'08) received the B.S. degree in computer science and the Ph.D. degree in computer software and theory from the University of Science and Technology of China in 2002 and 2007, respectively. He is currently an Associate Professor with the School of Computer Science and Technology, University of Science and Technology of China. He has authored over 60 papers and holds about 20 patents. His main research interest is software-defined networks, cooperative communication, and vehicular ad hoc network.

He Huang (M'13) received the Ph.D. degree from the Department of Computer Science and Technology, University of Science and Technology of China, in 2011. He is currently an Associate Professor with the School of Computer Science and Technology, Soochow University, China. His current research interests include crowdsourcing, software-defined networking, privacy preserving for wireless networks, and algorithmic game theory.



Shigang Chen (A'03–M'04–SM'12–F'16) received the B.S. degree from the University of Science and Technology of China in 1993, and the M.S. and Ph.D. degrees from the University of Illinois at Urbana–Champaign in 1996 and 1999, respectively, all in computer science. He was with Cisco Systems for three years. He joined the University of Florida in 2002. He was a CTO with Chance Media Inc. during 2012–2014. He is currently a Professor with the Department of Computer and Information Science and Engineering, University of Florida. He has authored over 160 peer-reviewed journal/conference papers. He holds 12 U.S. patents. His research interests include computer networks, Internet security, wireless communications, and distributed computing. He received the IEEE Communications Society Best Tutorial Paper Award and the NSF CAREER Award. He was an Associate Editor of the IEEE/ACM TRANSACTIONS ON NETWORKING, the IEEE TRANSACTIONS ON VEHICULAR TECHNOLOGY, and a number of other journals. He served in various chair positions or committee members for numerous conferences. He is an ACM Distinguished Member and a Distinguished Lecturer of the IEEE Communication Society.

Gongming Zhao is currently pursuing the master's degree in computer science with the University of Science and Technology of China. His main research interest is software-defined networks.

Liusheng Huang received the M.S. degree in computer science from the University of Science and Technology of China in 1988. He is currently a Senior Professor and a Ph.D. Supervisor of the School of Computer Science and Technology, University of Science and Technology of China. He has authored six books and over 300 journal/conference papers. His research interests are in the areas of Internet of Things, vehicular ad hoc network, information security, and distributed computing.