# An Efficient Label-Based Packet Forwarding Scheme in Software Defined Networks

Yeim-Kuan Chang,Yi-Tsung Huang and Yu-To Chen
Department of Computer Science and Information Engineering
National Cheng Kung University, Taiwan

***Abstract*—Software-Defined Networking (SDN) attracts the attention of many researchers in the academic and industry. OpenFlow protocol is a solution of SDN. In OpenFlow version 1.3, there are 40 match fields. This makes the design of switch more complicated. Casado et al. describe a new network architecture, called fabric network [2]. They separate the network elements into edge switches and core switches. Edge switches tag a label onto packet header, and core switches use this label to perform lookup. In this paper, we propose a scheme called Path_label switching. In this scheme, core switch only uses one field to perform lookup. In order to reduce the number of rules in edge switches, we encode the output port number of the last hop into MPLS field. We also propose a scheme to minimize the number of rules in fat-tree topology. In our scheme, the number of rules in core and aggregation switches is the same as the number of ports. We use Mininet to construct the experiments. We will show that our proposed scheme needs much less number of rules than other methods in leaf-spine and fat-tree topologies.**

***Keywords—Software-Defined Networking, OpenFlow, Label switching***

## I. Introduction

Software Defined Networking (SDN) is a new network architecture and attracts the attention of many researchers and the industry. For example, Google uses SDN to improve their internal network between data centers [1]. A key concept of SDN is to decouple the control plane from the data plane. In SDN, control functions are removed from the forwarding elements (routers or switches), and centralized in a centralized SDN controller. SDN controller has an entire network view for easy management of the network.

OpenFlow is a famous solution for SDN. SDN controller uses OpenFlow to communicate with OpenFlow switch and configure flow tables in OpenFlow switch. OpenFlow provides many fields that you can use to match packets. However, the number of match fields in OpenFlow grows rapidly. In OpenFlow version 1.0, it only has 12 fields. In OpenFlow version 1.3, it has 40 fields. This makes OpenFlow switch design becomes complicated.

Casado et al. describe that the complex network logic should exist only at the edge switches (i.e., ingress and egress switches), while the core switches should be kept as simple as possible [2]. Authors advocate label-based forwarding scheme as the basis for SDN network architecture. An intelligent edge switch routes traffic in and out of a label-based core. There are many methods using this idea ([3-7][17][21][22]). In this thesis, we propose a label-based forwarding scheme. We give each path in the network a unique label which is called Path_label. We encode Path_label in MPLS label of packet header. Core switch uses Path_label as forwarding label. In order to reduce the number of rules in edge switch, we also encode the output port number of egress switch in MPLS label. We also propose a scheme to minimize the number of rules in fat-tree topology.

The rest of this paper is organized as follows. In section II, we introduce background knowledge of SDN and related technology. Section III introduces the proposed scheme, Path_label switching, and the optimization for fat-tree topology. The experimental simulation and comparisons with shadow MAC [3] and EncPath [12] are shown in section IV. Section V concludes the paper.

## II. Related Wor

### A. *Software Defined Networking (SDN)*

Software-Defined Networking (SDN) is an emerging network architecture [8]. SDN decouples the control plane and data plane enabling the network control to become directly programmable and separates the control plane from switch to a centralized controller. The centralized controller is a key element of SDN and controls the entire network. Furthermore, it maintains a global view of the network and simplifies network management.

Besides OpenFlow, other solutions for SDN include P4 [9] and POF [10]. OpenFlow protocol is used to communicate between SDN controller and OpenFlow switch. The SDN controller uses OpenFlow protocol to control and manipulate flow tables of OpenFlow switches. The OpenFlow switch specification defines the flow table control messages and the behavior of an OpenFlow switch. In OpenFlow version 1.3, an OpenFlow switch consists of three main components: multiple flow tables, one group table, and a secure channel to a controller. A Flow entry contains of two parts: match fields and a set of actions. Match fields are packet header fields from L2 to L4. An action associated with the match fields defines how matched packets should be processed

As the version of OpenFlow is updated, the match fields become more and more. In OpenFlow 1.0, there are only 12 fields. In OpenFlow 1.3, there are 40 fields. This requires OpenFlow switch hardware to support lookups over hundreds of bits. It does not simplify switch hardware. Casado et al. [2] advocate label switching and propose a “fabric network” to solve the limitations of current SDN. The authors were inspired by MPLS. They separate the network into edge and core to put more intelligence into the edge and keep the core simpler. Figure 1 shows the architecture of fabric network [2].

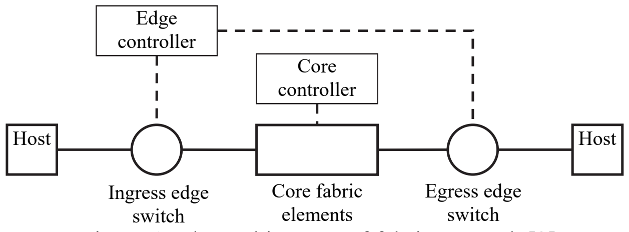


Figure 1. The architecture of fabric network [2].

The edge switch is responsible for complex network services, such as network security, while the core fabric only provides basic packet routing. Forwarding elements are different in core from edge switches. Core fabric is not required to use end-host addresses (e.g., IPv4 or IPv6) for forwarding. And core fabric is only responsible for delivering packets.

Shadow MACs [3] uses L2 addressing to implement the fabric network. A scalable label-based forwarding architecture was implemented on existing commodity switch by using virtual destination MAC addresses (i.e., shadow MACs) as forwarding label. Unlike traditional MAC addresses, shadow MACs are not associated with particular network interfaces. Rather, shadow MACs are unique values and used to represent paths. The key idea is to treat each packet's destination MAC address as an opaque forwarding label. The controller gives each path in the network a unique shadow MAC address. Controller then installs the rules in the L2 forwarding tables of each core switch along the path. This design allows core switches to use their large L2 forwarding tables to implement the label-based forwarding scheme. The ingress edge switches select the appropriate shadow MAC based on flow information (i.e., packet header) and changes packet's destination MAC address to shadow MAC. The egress switches rewrite the packet's destination MAC address to destination host's MAC address.

EncPath [12] uses the entire network view to encode the end-to-end path information (i.e., outgoing port numbers) into packet header. This scheme significantly reduces the flow table size and the number of control messages. In this scheme, the number of flow entries in a switch is linearly proportional to the number of ports. The authors find that OpenFlow supports arbitrary netmask in IP and MAC addresses where the ones and zeros can be inserted in any octets of netmask. They use $IP_{TTL}$ as a hop counter to point to the right address octet that contains the outgoing port number. EncPath can encode outgoing port numbers in IP or MAC address of packets. Below, we use EncPath address in IP address as an example.

This scheme has a path length challenge. When EncPath encodes the path in IP address, it can only encode the path length shorter than 8 hops. The authors provide a scalable mechanism to avoid the long path challenge. When the path length is longer than 8 hops, controller inserts a flow entry in the 8th switch to rewrite the IP address of packets again. The 8th switch rewrites the IP address of packets to the subsequent output port numbers. After this configuration, another challenge is how the 8th switch recognizes the packets belonging to this long path. The authors use the combination of MAC address, $IP_{TTL}$, and EncPath information as a flow-ID which is used by the 8th switch to identify the targeted flow.

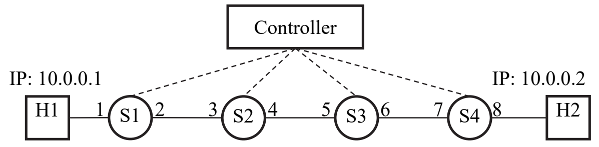


Figure 2. A linear topology.

In this paper, we focus on two commonly used data center topology, fat-tree [16] and leaf-spine. Fat-tree is a three layer network topology consisting of three types of switches, core, aggregation and edge. When a fat-tree is built by using K-port switches, there are K pods, each containing two layers (i.e., aggregation and edge) of K/2 switches. Each edge switch directly connects to K/2 hosts and K/2 aggregation switches. There are $K^2/4$ core switches. Each core switch connects to K pods. Each aggregation switch of a pod connects to K/2 consecutive core switches [16]. In general, a fat-tree built by K-port switches supports $K^3/4$ hosts. Figure 3 shows an example of fat-tree for K=4.

Leaf-spine is a two layer network topology consisting of two types of switches, leaf and spine. Every spine switch connects to every leaf switch in a full-meshed topology. Hosts only connect to leaf switches. We denote a leaf-spine topology as (S, L, H), where S is the number of spine switches, L is the number of leaf switches, and H is the number of hosts per leaf switch. Figure 4 shows an example of leaf-spine topology, where (S, L, H) is (3, 4, 2).

## III. PROPOSED SCHEME

The proposed scheme called *Path Label Switching* (*PathLS*) is a label-based switching mechanism that uses Path_label to make forwarding decision. Notice that, in traditional MPLS packet forwarding scheme, core routers need to swap labels, and egress routers need to lookup the label forwarding information base. However, in PathLs, labels only be tagged in ingress routers, and egress routers extract first 6 bits of MPLS field to determine the output port. We design two additional instructions for edge switches and core switches, and add one label table in core switches. Ingress edge switches tag incoming packets with Path_label provided by controller, and the egress edge switches remove Path_label from outgoing packets. Controller is responsible for computing Path_label that will be broadcast incrementally to all switches. The core switches use labels in the packet headers for switching by looking up the label table.

The Path_label in PathLS represents the path of a flow and it will be used by core switches for packet forwarding decision. We use Figure 2 as an example. In this topology, the path from H1 to H2 is S1->S2->S3->S4. Excluding the ingress switch S1, we use a label to represent the path S2->S3->S4 to forward packets from H1 to H2.

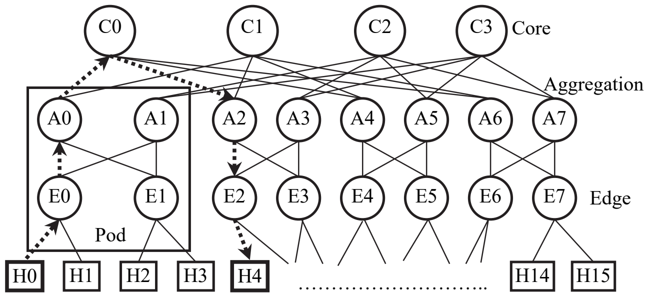


Figure 3. A fat-tree topology of K=4.

One challenge for a label-based switching scheme is to determine how a label will be tagged onto a packet. We find that OpenFlow version 1.3 [11] supports the MPLS label field. So, we choose to push Path_label on the MPLS label field. Figure 5 shows the format of MPLS header. There are four fields in MPLS header, namely 20-bit *Label*, 3-bit *TC* (Traffic Class for QoS), 1-bit *S* (flag to indicate Bottom-of-Stack), and 8-bit *TTL* (Time-to-Live). TC is not used in our scheme. When S is 1, it signifies that the current MPLS label is the last in the MPLS stack. In our mechanism, this TTL filed will only be used in the fat-tree topology. How to use TTL will be described in subsection III.E.

When packets with Path_label reach at egress switches, egress switches should remove Path_label from packets and use the destination IP to lookup the flow tables. In this situation, egress switches may have many flow entries in flow tables, but we want egress switch to have only one rule in our mechanism. In order to achieve this goal, we encode the output ports of the egress switches into the first 6 bits of MPLS's Label field. Path_label can be encoded in the remaining 14 bits of MPLS's Label field, as shown in Figure 5. When the network topology is fat-tree, we have another way to encode the Path_labe into MPLS's Label field that will be described in subsection III.C

### *A. Extending OpenFlow*

We extend OpenFlow by adding a label table in OpenFlow switches. Label table will only be used in core switches because edge switches do not use Path_label for lookups. We also add two instructions, *Goto_label_table* and *Read_output*. Switch extracts Path_label from packet header and directs it to abel table for lookups. The implementation of this instruction is different in the general topology and fat-tree topology because the length of Path_label is different. Egress switch gets the output port from the first six bits of the MPLS's Label field and removes Path_label from packet header. Then the egress switch transfers the packet to the extracted output port.

### *B. Control Planes*

We implement the proposed label switching in Ryu [14] with four major components, *OpenFlow API*, *Topology viewer*, *Path_computing*, and *Path_label Information Base* (*PIB*).

OpenFlow API and Topology viewer built-in components. We modify it so that Ryu can send Flow-Mod message with two new instructions to switches. Ryu uses Link-Layer Discovery Protocol (LLDP) to get a complete network

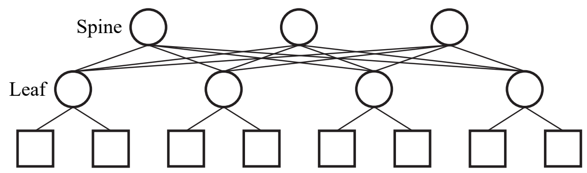


Figure 4. An example of leaf-spine topology.

Original

| Label | TC | S | TTL |
|---|---|---|---|

Modified

| Egress->Host | Path_label | 0 | 1 | 1 |
|---|---|---|---|---|

0 6 20 23 24 31

Figure 5. Original and modified MPLS header field.

topology view. We save the network topology as a graph. We use a Python data structure named dictionary to save this information. As for host discovery, we will collect this information when a host starts sending a new flow to a destination.

Component PIB is implemented by Python dictionary to store the path and the corresponding Path_label. The pair of key and value in dictionary correspond to the path and Path_label. When Ryu receives a Packet-In message, it will extract the destination IP from the payload of the Packet-In message. Ryu then triggers component Path_computing to compute the shortest path from source to destination. We use Breadth-first Search to compute the shortest paths. After computing, Ryu will check whether the path exists in PIB or not. If it does not exist, Ryu will give it a new Path_label and add new entry into PIB.

### *C. Data Plane*

Open vSwitch (OVS) [13] is a well-known software switch. We extend OVS to support our mechanism. The three major component of the existing OVS are OpenFlow API, userspace flow table pipeline, and the kernel flow table. OpenFlow API is used to process OpenFlow protocol, like Packet-In, Packet-Out, Flow-Mod, and so on. We extend Ryu and OVS to support our two instructions, *Goto_label_table* and *Read_output*. The implementation of *Goto_label_table* is different in general topology and fat-tree topology because the definition of Path_label are different. In general topology, the length of Path_label is 14 bits, and it is 8 bits in fat-tree topology. So OVS has two way to extract the Path_label from MPLS label. The flow table pipeline contains one or more flow tables, each with flow entries that specify how the matching packets should be processed. To speed up packet processing, the kernel flow table caches the actions of the active flows from the userspace flow table so that active flows can be directly processed in the kernel.

We add a label table in the userspace of OVS to perform Path_label lookup, and this table is only added in core switches. We use an array of pointer to implement label table. In general topology, because the length of Path_label is 14 bits, the size of label table is $2^{14}$. The length of Path_label is 8 bits in fat-tree topology, so the size of label table is $2^{8}$. The value of Path_label is the index of the label table, and the associated action is stored in the entry. When the entry points to NULL,

Table 1. The flow table and label table in S2.

(a) The flow entry in the flow table

| Match field | Action |
|---|---|
| Ether_type=MPLS | *Goto_label_table* |

(b) The Path_label in the label table

| Path_label | Action |
|---|---|
| 1 | Output(4) |

Table 2. The flow entry in the flow table of S4.

| Match field | Action |
|---|---|
| Ether_type=MPLS | *Read_output* |

Table 3. Label table size comparison in fat-tree (K=4).

| | Path_label | Optimization |
|---|---|---|
| Core switches | 24 | 4 |
| Aggregation switches | 26 | 4 |

it means that the label table does not have this Path_label. The SDN controller installs Path_label and associated actions in label table by sending Flow-Mod messages.

Procedure in Edge Switch. Edge switch receives the packet and perform flow table lookup. If the packet is not matched, edge switch sends a Packet-In message to the controller to request the flow entry. If the packet is matched and the Ether_type is MPLS (i.e., the packet contains Path_label) as in the case of egress switch, edge switch performs Read_output. If the packet is matched and the Ether_type is not MPLS as in the case of ingress case, edge switch tags the corresponding Path_label on it and transfers to the output port.

Procedure in Core Switch. Core switch receives the packet and perform flow table lookup. If the Ether_type of the packet is MPLS, it means that the packet has Path_label. Then core switch directs the packet to label table to perform lookup and transfers the packet to the output port. If the Ether_type of the packet is not MPLS, core switch performs other packet processing. For example, core switch receives the LLDP packet used by the controller to collect the network topology. Core switch then sends LLDP packet to the controller via a Packet-In message.

## *D. Procedure for Path_label Switching*

We will use Figure 2 to describe the procedure of Path_label Switching. Initially, switches do not have any flow entry in their flow tables. After switches connect to Ryu, switches insert two flow entries in their flow tables. One is *table-miss*, and the other is for LLDP packet. The *table-miss* flow entry specifies how to process packets unmatched by other flow entries in the flow table (i.e., send packets to controller via Packet-In messages). First, Ryu uses the Topology viewer to get the network topology and saves it as a graph. H1 starts sending messages to H2. S1 receives the first packet of the flow and performs a lookup in its flow table. When the packet is not matched, S1 buffers the packet and sends a Packet-In message containing the packet header to Ryu. Ryu receives the Packet-In message and extracts the destination IP of the packet. Ryu then computes the shortest path from H1 to H2 and checks whether the path exists in the PIB or not. If the path doesn't exist, Ryu will create a

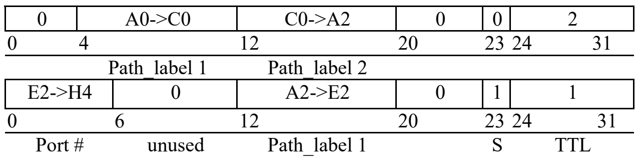


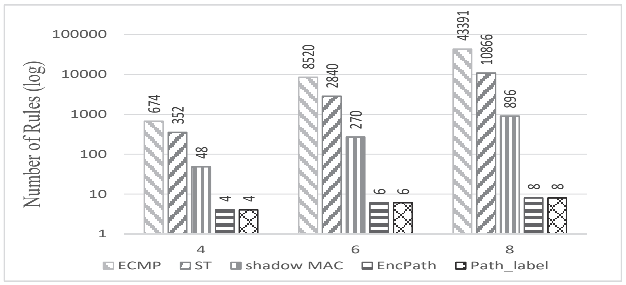
Figure 6. Stacked MPLS headers.

Path_label for it and add new entry in PIB. Ryu sends Flow-Mod messages with flow entries to the switches on the path in order. Ryu also sends a Barrier message to the egress switch, S4, to make sure the path is set up. By assuming the value of Path_label is 1, MPLS label is 0010-0000-0000-0000-0001 because the most significant six bits of MPLS label are for the output port 8 (i.e., 001000) of S4 to H2 and the remaining fourteen bits are 00-0000-0000-0001. The ingress switch, S1, receives the Flow-Mod message and inserts the flow entry in its flow table. The action applied on the packet is to add Path_label in packet header and send it out from port 2. The core switch, S2, receives two Flow-Mod messages. One is for inserting the flow entry in its flow table, and the other is for inserting Path_label in its label table, as shown in Table 1. The flow entry in flow table of S3 is the same as S2 except Output = 6. The egress switch, S4, receives a Flow-Mod message and a Barrier message. S4 inserts a flow entry in its flow table, as shown in Table 2. The action of this flow entry indicates that S4 gets the output port from the packet header. S4 then removes the Path_label from the packet header and transfers to the output port. After S4 inserts the flow entry, it will send a Barrier-Reply message to Ryu. The Barrier-Reply message is used to tell Ryu that S4 finishes inserting the flow entry in its flow table. After Ryu receives the Barrier-Reply message from S4, it will send a Packet-Out message to S1. S1 then performs the lookup for the first packet buffered in S1. Finally, the packet can then follow the path computed by Ryu to reach H2.

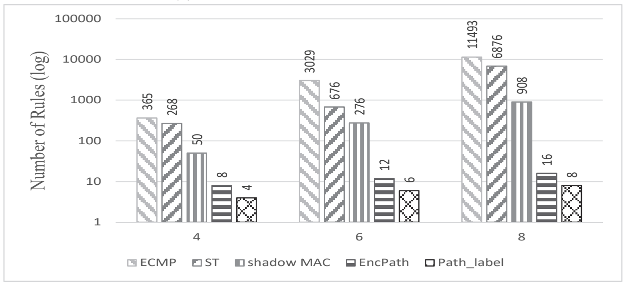
## *E. Optimization for fat-tree*

A fat-tree, as mentioned in subsection II.F, is a three-layer network topology consisting of three types of switches, core, aggregation and edge. To minimize the number of rules in core and aggregation switches, we combine our method with EncPath. We modify the size of Path_label from fourteen bits to eight bits, and the value of Path_label is the port number of switch. We use the TTL field to indicate the number of 8-bit Path_label stored in the MPLS header. TTL field value can be 2 or 1. When TTL is 2, bits 4-11 and 12-19 of Label field store two Path_label. When TTL is 1, only bits 12-19 of MPLS label store a Path_label. The flag bit S perfroms the same function as before. After switch performs lookup in label table, switch should decrease TTL by one. When TTL becomes 0 and also flag S is 0, switch should pop MPLS label from MPLS stack.

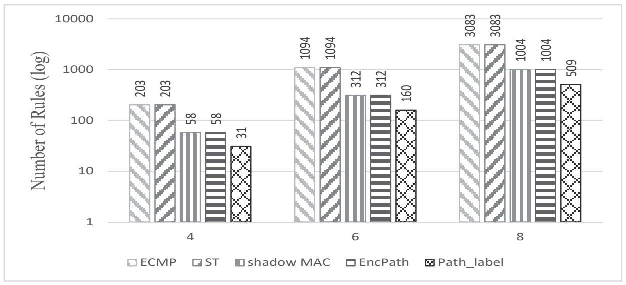
Figure 3 shows an example to explain the optimization in fat-tree topology. The path from H0 to H4 is E0->A0->C0->A2->E2. We then encode the outgoing port numbers into MPLS headers, as shown in Figure 6. The output port from A0 to C0 of switch A0 and output port from C0 to A2 of switch C0 are encoded in bits 4-11 and 12-19 of the upper MPLS's Label field in MPLS stack. The TTL of the upper MPLS's Label field is set to 2. The output port from A2



(a) Number of rules in core switch.



(b) Number of rules in aggregation switch.



(c) Number of rules in edge switch.

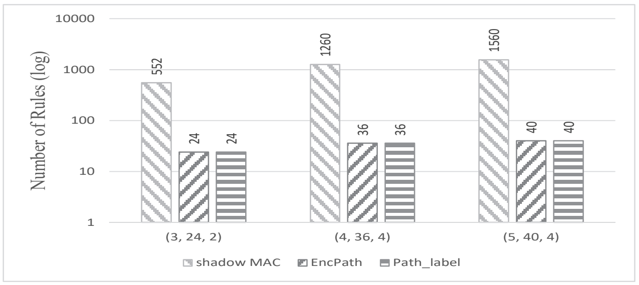
Figure 7. Number of rules in switches (fat-tree).

to E2 of switch A2 is encoded in bits 12-19 of the bottom MPLS's Label field in MPLS stack. And the output port from E2 to H4 is encoded in bits 0-5 of the bottom MPLS's Label field. The TTL of the second MPLS label is set to 1. Assume that all the outgoing port numbers of the path (H0->H4) are 1, H0's IP is 10.0.0.1, and H4's IP is 10.0.0.5. The values of two MPLS's Label field values are 257 (0000-0000-0001-0000-0001) and 16385 (000001-000000-0000-0001), respectively. The flow entries in the flow tables of A0, C0, and A2 are the same. As mentioned in subsection III.C, the implementation of *Goto_label_table* in fat-tree topology is different from general topology.

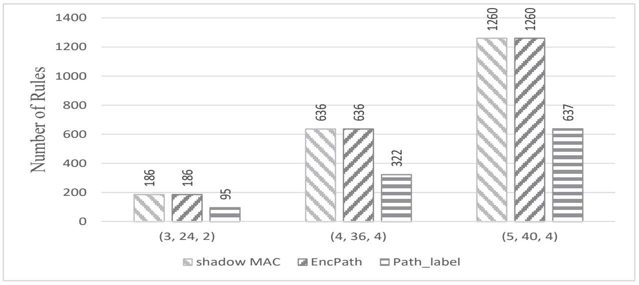
The optimization for fat-tree topology can make the number of rules in core switches and aggregation switches equal to the number of ports. Table 3 shows the label table size comparison in fat-tree topology which is built by 4-port switches.

## IV. EXPERIMENTAL RESULTS

In order to validate our scheme, we use Mininet [15] to construct the network topology. Mininet is a network emulator which creates a network of virtual hosts, switches, controllers, and links. We use Open vSwitch, a well-known software switch, as our OpenFlow switch and Ryu as our controller.



(a) Number of rules in spine



(b) Number of rules in leaf

Figure 8. Number of rules in switches (leaf-spine).

In order to generate all the possible paths, we use the built-in CLI command *pingall* in Mininet to generate ICMP packets from every host to all other hosts. We build three different sizes of two common data center topologies, leaf-spine and fat-tree, and then compute the number of rules in switches.

Figure 7 shows results of *PathSL* compared with EncPath, shadow MAC, ECMP and ST in fat-tree topology. Riplpox [18] is used to run Equal-Cost Multi-Path (ECMP) and Spanning Tree (ST) routing algorithms. Riplpox is a simple data center controller built on POX. In Path_label and EncPath, the number of rules in a core switch is the same as the number of ports which is smaller than the other three methods. In Path_label, the number of rules in an aggregation switch is the same as the number of ports, and it is half of that needed in EncPath. Figure 7 (c) shows the number of rules in an edge switch. Our scheme is 15% to 50% of the other methods. A summary of the comparison is shown in Table 4, where K is the parameter of fat-tree topology. We also write a program to analyze how many rules in switches of a fat-tree topology designed by Microsoft [20], where K=20. In this topology, there are 500 switches and 2000 hosts. We compare our scheme with shadow Mac and EncPath as shown in Table 5.

Because riplpox cannot run in leaf-spine topology, we only compare our scheme with shadow MAC and EncPath. Results in Figure 8 (a) show the number of rules in spine switch. In Path_label, the number of rules in spine switch is the same as the number of leaf switches. Figure 8 (b) shows the number of rules in leaf switch. In Path_label, the number of rules is 50% of the others. The comparison is shown in Table 6, where S is the number of spine switches, L is the number of leaf switches, and H is the number of hosts per leaf switch.

### A. Path Setup Latency

We measure path setup latency from H1 to H2. We use Mininet to build a linear topology as in Figure 2. In each

Table 4. Comparison of the number of rules (fat-tree).

| | Shadow MAC | EncPath | Path_label |
|---|---|---|---|
| Core | $(K^4-K^3)/4$ | K | K |
| Aggregation | $(K^4-K^3+K^2-2K)/4$ | 2K | K |
| Edge | $(K^4-K^2-2K)/4$ | $(K^4-K^2-2K)/4$ | $(K^4-4K)/8+1$ |

Table 5. Number of rules in fat-tree topology (K=20).

| | Shadow MAC | EncPath | Path_label |
|---|---|---|---|
| Core | 38000 | 20 | 20 |
| Aggregation | 38090 | 40 | 20 |
| Edge | 39890 | 39890 | 19991 |

Table 6. Comparison of the number of rules (leaf-spine).

| | Shadow MAC | EncPath | Path_label |
|---|---|---|---|
| Spine | L(L-1) | L | L |
| Leaf | $(2L-1)H^2-H$ | $(2L-1)H^2-H$ | $LH^2-H+1$ |

experiment, H1 will ping H2, and we use the round-trip time (RTT) of the first packet as path setup latency. We run simple_switch_13 that is a built-in application in Ryu. The simple_switch_13 makes switches act as a learning switch. When the switch receives a packet of a new flow, it sends a Packet-In message to request flow entry. If the controller knows where the destination host is, it will send a Flow-Mod message to the switch. If controller does not know where the destination host is, controller will send a Packet-Out message to the switch to broadcast the packet. When the packet reaches at the next switch, the above steps will be repeated. Figure 9 shows the results on path setup latency. The reason why our proposed scheme is better than simple_switch_13 is the number of Packet-In messages. In Path_label, controller only receives one Packet-In message and uses its network global view to establish a path, while controller needs as many as the number of switches to establish a path in simple_switch_13.

## V. CONCLUSIONS

In this paper, we proposed a label-based forwarding scheme, named Path_label switching. Controller has global network view to compute the shortest paths. We give each path in the network a unique Path_label. Path_label and the output port number of egress switch are encoded into MPLS headers. Core switch uses Path_label to perform lookup and forward packets. We also propose a scheme to minimize the number of rules in fat-tree topology. As a result, the number of rules in core switch and aggregation switch are equal to the number of ports. We use Mininet to build two commonly used data center topologies, leaf-spine and fat-tree. The number of rules needed in switches are less than other existing methods. We use Mininet to build a linear topology, and measure the path setup latency from a source to a destination. We compare our scheme with a built-in application in Ryu. This application acts as a traditional scheme with OpenFlow. As a result, the path setup latency is shorter than traditional scheme.

## VI. REFERENCES

[1] S. Jain , A. Kumar , S. Mandal , J. Ong , L. Poutievski , A. Singh , S. Venkata , J. Wanderer , J. Zhou , M. Zhu , J. Zolla , U. Hölzle , S. Stuart , A. Vahdat, "B4: Experience with A Globally-Deployed Software Defined WAN", in SIGCOMM, pp. 3-14, 2013.

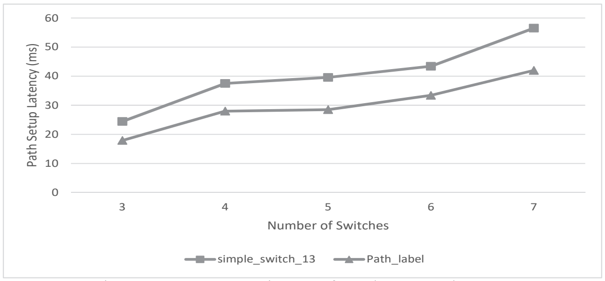


Figure 9. Comparison of path setup latency.

[2] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian, "Fabric: A Retrospective on Evoloving SDN", in HotSDN, pp. 85-90, 2012.

[3] K. Agarwal, C. Dixon, E. Rozner, and J. Carter, "Shadow MACs: Scalable Label-switching for Commodity Ethernet", in HotSDN, pp. 157-162, 2014.

[4] A. Schwabe and H. Karl, "Using MAC Address as Efficient Routing Labels in Data Centers", in HotSDN, pp. 115-120, 2014.

[5] H. Farhadi and A. Nakao, "Rethinking Flow Classification in SDN", in IEEE Int'l Conference on Cloud Engineering, pp. 598-603, 2014.

[6] A. Hari, T. V. Lakshman, and G. Wilfong, "Path Switching: Reduced-State Flow Handling in SDN Using Path Information", 11th ACM Conf. on Emerging Networking Experiments and technologies, pp. 1-7, 2015.

[7] C. Filsfils, N. Kumar Nainar, C. Pignataro, J. Camilo Cardona, and P. Francois, "The Segment Routing Architecture", IEEE Global Communications Conference, pp. 1-6, 2015.

[8] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J.Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks", ACM SIGCOMM Computer Communication Review, vol.38, no.2, pp.69-74, 2008.

[9] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-Independent Packet Processors", ACM SIGCOMM CCR, pp. 87-95, 2014.

[10] H. Song "Protocol-Oblivious Forwarding: Unleash the Power of SDN through a Future-Proof Forwarding Plane", HotSDN, pp. 127-132, 2013.

[11] OpenFlow Foundation, "OpenFlow Switch Specification Version 1.3.0".

[12] A. AlGhadhban and B. Shihada, "Energy Efficient SDN Commodity Switch based Practical Flow Forwarding Method", in Network Operations and Management Symposium (NOMS), pp. 784-788, 2016.

[13] Open vSwitch. Available: http://openvswitch.org

[14] Ryu SDN Framework. Available: https://osrg.github.io/ryu

[15] B. Lantz, B. Heller, and N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks", the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, pp. 19:1-19:6, 2010.

[16] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, Commodity Data Center Network Architecture", in SIGCOMM, pp. 63-74, 2008.

[17] K. Kannan and S. Banerjee, "Compact TCAM: Flow Entry Compaction in TCAM for Power Aware SDN", in Distributed Computing and Networking, pp. 439-444, 2013.

[18] Riplpox. Available: https://github.com/MurphyMc/riplpox

[19] C. Hopps, "Analysis of an Equal-Cost Multi-Path", RFC 2992, Internet Engineering Task Force, 2000.

[20] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall, "Augmenting Data Center Networks with Multi-Gigabit Wireless Links", in Proceedings of ACM SIGCOMM, pp.38-49, 2011.

[21] Y. Ren, T.-H. Tsai, J.-C. Huang, C.-W. Wu, Y.-C. Tseng, "Flowtable-Free Routing for Data Center Networks: A Software-Defined Approach," GLOBECOM 2017.