This is the Pre-Published Version.

The following publication F. Li, J. Cao, X. Wang, Y. Sun, T. Pan and X. Liu, "Adopting SDN Switch Buffer: Benefits Analysis and Mechanism Design," 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), Atlanta, GA, USA, 2017, pp. 2171-2176 is available at https://doi.org/10.1109/ICDCS.2017.255.

Adopting SDN Switch Buffer: Benefits Analysis and Mechanism Design

Fuliang Li[†‡‖], Jiannong Cao[‡], Xingwei Wang[†¶(*)], Yinchu Sun[†], Tian Pan[§], Xuefeng Liu[‡]
[†] School of Computer Science and Engineering, Northeastern University, Shenyang, China
[‡] Department of Computing, Hong Kong Polytechnic University, Hong Kong, China
[§] Department of ICE, Beijing University of Posts and Telecommunications, Beijing, China
[¶] Software College of Northeastern University, Northeastern University, Shenyang, China
[‖]Key Laboratory of Computer Network and Information Integration (Southeast University), Ministry of Education

***Abstract*—One critical issue in SDN is to reduce the communication overhead between the switches and the controller. Such overhead is mainly caused by handling miss-match packets, because for each miss-match packet, a switch will send a request to the controller asking for forwarding rule. Existing approaches to address this problem generally need to deploy intermediate proxy or authority switches to hold rule copies, so as to reduce the number of requests sent to the controller. In this paper, we argue that using the intrinsic buffer in a SDN switch can also greatly reduce the communication overhead without using additional devices. If a switch buffers each miss-match packet, only a few header fields instead of the entire packet are required to be sent to the controller. Experiment results show that this can reduce 78.7% control traffic and 37% controller overhead at the cost of increasing only 5.6% switch overhead on average. If the proposed flow-granularity buffer mechanism is adopted, only one request message needs to be sent to the controller for a new flow with many arrival packets. Thus the control traffic and controller overhead can be further reduced by 64% and 35.7% respectively on average without increasing the switch overhead.**

I. Introduction

SDN enhances network flexibility and scalability by separating control plane from data plane. It is progressively dominating the dynamic management for timely network trouble shooting and fine grained traffic scheduling in data center networks [1-3]. One critical issue in SDN is to reduce the communication overhead between the switches and the controller. Such overhead is mainly caused by miss-match packets that cannot match any forwarding rules of the flow tables. For each miss-match packet, the switch will generate a request message and send it to the controller. After the controller decides how to forward the packet, it will send operation messages back to the switch. According to these operation messages, the miss-match packet and the subsequently arrival packets of this flow can be forwarded. If many new flows arrive simultaneously, they may introduce massive miss-match packets, which will trigger the corresponding number of request messages sent to the controller. The overhead, including the transmission load on the control path and the computation load on the controller will inflate quickly.

Existing studies have tried to reduce the communication overhead by cloning more forwarding rules in intermediate proxy [4] or authority devices [5], which in partly take responsibility of the controller. The switch first requests the intermediate devices for how to forward the miss-match packets, and will not request the controller unless the devices fail to give a response. Although these methods reduce the requests sent to the controller, they don't reduce the requests generated by the switches. Furthermore, additional devices will increase the overall budget. Different from previous works, we utilize the intrinsic buffer of SDN switches to reduce both the size and the number of the request messages generated by the switches. Our methods can substantially reduce the communication overhead and will make a supplement to existing approaches. To our best knowledge, we conduct a first-ever study on reducing the switch-controller communication overhead through SDN switch buffer. The main contributions of this paper are summarized as follows.

(1) We investigate the benefits of adopting SDN switch buffer. Without the buffer, each miss-match packet is entirely included in the request message sent to the controller, while if adopting the buffer, only several header fields are involved, which will shrink the request message size. Experiment results reveal that the switch buffer can reduce 78.7% control path load and 37% controller overhead at the cost of increasing only 5.6% switch overhead on average.

(2) We propose a flow-granularity buffer mechanism for SDN switches. With the default buffer mechanism, every miss-match packet will trigger a request message sent to the controller. While using the proposed buffer mechanism, only one request message is sent to the controller for all the miss-match packets of a flow, which will reduce the number of the request messages. Experiment results show that the proposed buffer mechanism can further reduce 64% control path load and 35.7% controller overhead on average without increasing the switch overhead.

According to OpenFlow switch specifications [6], the buffer is not used by default under the assumption that a flow sets up beginning with several small packets negotiating first. For example, a TCP connection starts with the three-way handshake, which only needs three small packets to establish the connection. In this case, the buffer is not necessary indeed. However, for an UDP connection, one communication end may suddenly send massive packets to another end without negotiation, in which case, buffer becomes inevitable. In addition, forwarding rule of a TCP flow may be kicked out

© 2017 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

from the size limited flow table, occurring during the short time period of data transmission interruption. Large volume of data may be transmitted after that transient time period because the TCP connection is not terminated in actual. Therefore, buffer is also useful for such kind of TCP connections.

The remainder of this paper is organized as follows. Related work is presented in Section II. Benefits of adopting SDN switch buffer are analyzed in Section III. Section IV presents and evaluates the proposed flow-granularity buffer mechanism. We conclude the whole paper in Section V.

II. Related Work

SDN has become a promising network technology and it has been deployed in data center networks [1-3]. However, there are still many issues need to be addressed for SDN.

One critical issue is to reduce the communication overhead between the switches and the controller. In SDN, frequent interactions between switches and controller are necessary for miss-match packets. Kim et al [7] addressed the controller overhead when packets cannot match any rule of the flow tables. Curtis et al. [4] argued that fine-grained control of SDN cannot meet the demands of high performance networks, e.g., micro flows may create excessive load on the controller and the switches. They decreased the communication overhead between switches and controller by cloning forwarding rules in intermediate proxy, which can reduce the need to invoke the control plane for most flow setups. *DIFANE* [5] distributed necessary rules to the intermediate authority switches. Forwarding rules are partly made in data plane, which can reduce the flow setup latency and respond quickly to network dynamics. *Mazu* [8] reduced the latency of generating request messages by redirecting miss-matched packets to a fast proxy that is tasked with generating the necessary messages for the controller, while it reduces the latency of execution of forwarding rules by enabling fast parallel execution of updates. In summary, current studies reduce the requests on controller through utilizing intermediate devices to play a part role of controller. Different from previous works, this study takes full utilization of SDN switch buffer to reduce the communication overhead. Current works reduce the requests sent to the controller, while we try to reduce the size and the number of the requests generated by the switches. That is to say, our method could also be utilized as a supplement to existing approaches.

III. Problem and Experiment Description

A. Problem Description

A flow contains many packets of $\{p_1, p_2, ..., p_n\}$ arriving at $\{t_1, t_2, ..., t_n\}$. If p_1 matches a rule of the flow table, it will be forwarded at a line rate. Otherwise, the switch will generate a *pkt_in* message sent to the controller. After the controller decides how to forward the packet, it will send a pair of control operation messages (*flow_mod* and *pkt_out*) to the switch: *flow_mod* message carries the forwarding rule that will be installed in the switch; *pkt_out* message instructs to directly forward the miss-match packet through a specified interface of the switch. The time of the *flow_mod* message taking effect is t_e. If $t_e > t_2$, p_2 will trigger another *pkt_in* message. In the worst case, if $t_e > t_n$, n *pkt_in* messages will be triggered. When massive packets fails to match any rules of the flow tables simultaneously, a great deal of request messages will be sent to the controller. Moreover, corresponding control operation messages will be sent back to the switch. Such communication overhead needs to be reduced by the following three reasons.

1) Control path may share the same physical links with the data path. When the physical links carry heavy data traffic, control messages may be congested. 2) Even if we reserve the bandwidth or increase the priority for control traffic, we still have the requirements of reducing the control messages to relieve the load on the centralized controller. 3) Concurrent switch activities, i.e., generating control request messages and handling control operation messages will increase the communication delay between the switch and the controller [9]. This is caused by the competition of the limited resources of the switch. So it is important to keep the control traffic at a low level.

TABLE I
Configurations of Experimental Devices

Device Name	CPU	Cores	RAM	NIC
$Host_1$	3.3GHZ	4	4GB	1×100Mbps
$Host_2$	3.3GHZ	4	4GB	1×100Mbps
Open vSwitch	3.3GHZ	4	4GB	3×100Mbps
Floodlight	3.3GHZ	4	4GB	1×100Mbps

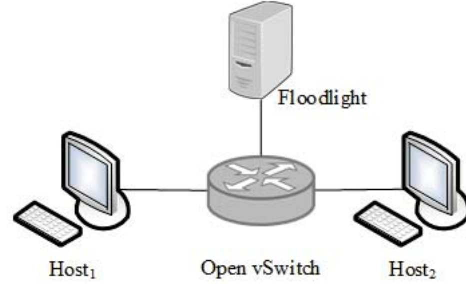

Fig. 1. Topography of the Experimental Platform.

B. Experiment Description

Fig. 1 shows our experiments setup. *Open vSwitch*(OVS) [10] is an open source OpenFlow virtual switch. *Floodlight* [11] is an open source SDN controller. We run *OVS* and *Floodlight* on two commodity PCs respectively. Table I shows the configurations of the experimental devices. $Host_1$ and $Host_2$ connect to OVS with 100Mbps interfaces. We run *pktgen* [12] on $Host_1$ to generate traffic at rates of 5Mbps - 100Mbps with the Ethernet frame size of 1000 Bytes. We run *tcpdump* [13] to listen on the interfaces that are connected to the hosts and the controller respectively.

IV. Benefits of Adopting Switch Buffer

Adopting the default buffer of *OVS*, the switch buffers each miss-match packet, and lets only several header fields instead of the entire miss-match packet include in the *pkt_in* message. In our experiment, the buffer is set to no-buffer, buffer-16 (storing at most 16 packets) and buffer-256 (storing at most 256 packets). As shown in Fig. 1, $Host_1$ sends one packet for each new flow to $Host_2$. To generate new flows, we use *pktgen* [12] to forge source IP addresses and generate 1000 flows (one packet for each flow) at each sending rate. We repeat the experiments at each sending rate for 20 times.

A. Control Path Load

Control path load refers to the control traffic (i.e., *pkt_in* messages sent from the switch to the controller, *flow_mod* and *pkt_out* messages sent from the controller to the switch) on the control path. We analyze the control path load from two directions respectively. As depicted in Fig. 2 (a), we find that the control path load nearly presents a linear relation to the sending rate when the buffer is not utilized. Without buffer, the entire miss-match packet will be included in the *pkt_in* message, resulting in large packets sent to the controller. When the buffer is used, only several bytes of each miss-match packet are included in the *pkt_in* message. Results show that buffer-16 and buffer-256 limit the control path load under 40Mbps. Control path load of buffer-16 gradually increases when the sending rate exceeds 35Mbps, while buffer-256 always generate less control traffic with the mean of 10.86Mbps and the standard deviation of 6.05Mbps. Deep analysis results reveal that using buffer-16, the buffer is exhausted around the sending rate of 35Mbps. So once the buffer is adopted, the buffer size should be correctly set according to the traffic patterns. Fig. 2(b) shows the similar patterns to Fig. 2(a). A *pkt_in* message will introduce a pair of *flow_mod* and *pkt_out* messages. Without buffer, the *pkt_out* message includes the entire miss-match packet, while it mainly contains a specific port number when using buffer. In summary, providing enough buffer space, we can reduce 78.7% of the control path load of one direction on average and 96% on average for another direction.

B. Controller Usages

We measure controller usages (i.e., CPU utilization of the *floodlight* process) to evaluate the load on the controller. As depicted in Fig. 3, controller usages present a gradual growth when the sending rate is lower than 50Mbps. After that point, controller usage of no-buffer is unstable and increases quickly with a standard deviation of 33.41%. While controller usages of buffer-16 (mean of 53.07% and standard deviation of 16.62%) and buffer-256 (mean of 34.59% and standard deviation of 9.87%) are relatively low and stable. Without buffer, the controller needs to capture the header fields of each miss-match packet from the *pkt_in* messages. When the sending rate becomes higher, the controller will handle more concurrently arrival *pkt_in* messages and require much more computing resources as a result. If adopting buffer, the

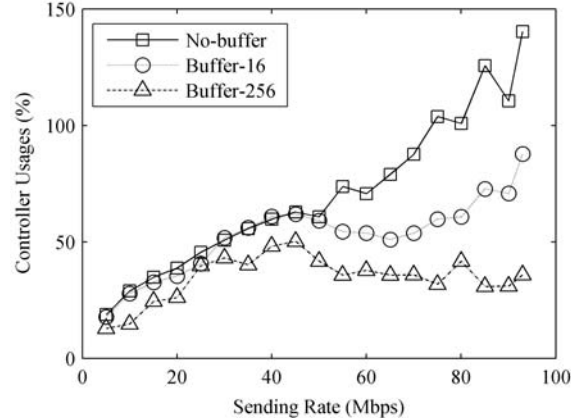

Fig. 3. Controller Usages under Different Sending Rates.

pkt_in message only contains the necessary header fields, which simplifies the process of decision making. Due to the exhaustion of the buffer space, buffer-16 shows a poor performance when the sending rate is high. Note that since the experimental devices are multi cores, controller usages exceed 100% sometimes. In summary, buffer can reduce 37% of the controller overhead on average. And if we set the buffer with enough space, we can keep the controller usages at a relatively stable level.

C. Switch Usages

We use switch usages (i.e., CPU utilization of the switch) to evaluate the load on the switch. We should make sure how much extra load will be added to switch when adopting buffer. As depicted in Fig. 4, switch usages of no-buffer, buffer-16 and buffer-256 present similar patterns, increasing quickly at the beginning, while slowly when the sending rate exceeds 40Mbps. During the whole testing process, buffer-256 (mean of 274.64% and standard deviation of 44.62%) introduces more load to switch than buffer-16 (mean of 263.84% and standard deviation of 51.88%), which behaves similarly to no-buffer (mean of 260.13% and standard deviation of 51.92%). The reason is that buffer related operations cause higher switch CPU utilization than no-buffer. Buffer-16 is exhausted when the sending rate is high, so it performs similarly to no-buffer after that point. In summary, buffer adoption complicates the switch design and its packet process, but we find that it only introduces 5.6% extra load to switch on average. This is a positive indicator to adopt buffer in SDN switches.

V. Flow-granularity Buffer Mechanism

The default buffer only reduces the size of the request messages. It still requires to send *pkt_in* messages for every miss-match packet. We name the default buffer mechanism packet-granularity. For a flow with many miss-match packets, redundant *pkt_in* messages will be sent to the controller. Therefore, in addition to reducing the request message size, we further aim to reduce the number of the request messages.

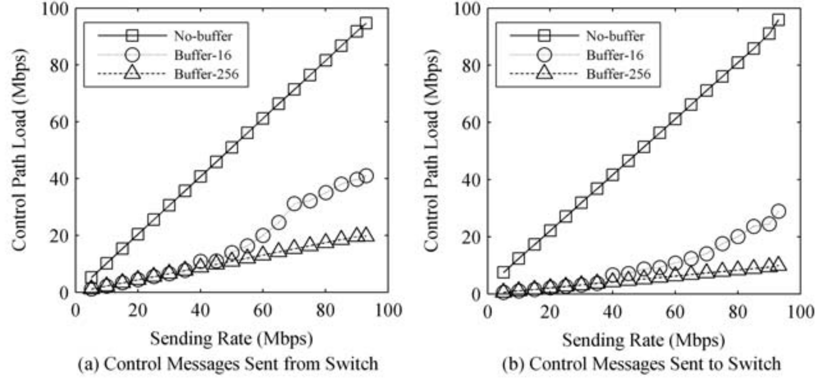

Fig. 2. Control Path Load under Different Sending Rates.

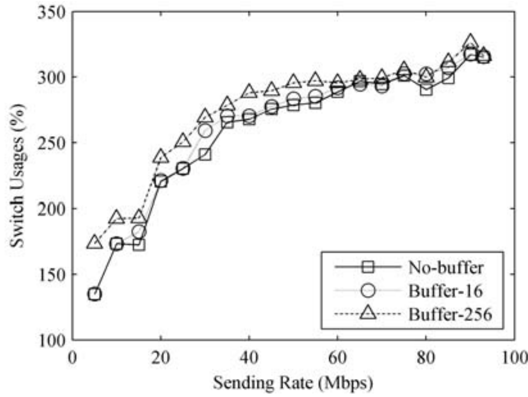

Fig. 4. Switch Usages under Different Sending Rates.

A. Mechanism Description

In this section, we present a flow-granularity buffer mechanism. It buffers all the miss-match packets of a flow and lets only one request message send to the controller. After a timeout period, if the switch doesn't receive the control operation messages, it will send another request message. The flow-granularity buffer mechanism is divided into two parts as depicted in ***Algorithm 1*** and ***Algorithm 2***.

Algorithm 1 describes how to buffer each miss-match packet of a flow. A flow F with n packets arrives a switch. Each packet p_i of F will first match the flow table (line 2). If p_i matches a rule, it will be directly forwarded by the switch (line 3). Otherwise, the switch will extract the *buffer_id* for p_i (line 4∼5). If p_i is the first arrival packet of the flow, the switch cannot get the *buffer_id* from the *buffer_id* map (line 6). Then, the switch buffers p_i and creates a *buffer_id* for p_i (line 7). Note that all the miss-match packets of F share the same *buffer_id*. It is calculated based on the tuple of (*src_ip*, *src_port*, *dst_ip*, *dst_port*, *protocol*), which is usually used to identify a flow. The switch stores the *buffer_id* for p_i (line 8) and sends a *pkt_in* message to the controller (line 9). The *pkt_in* message includes the header of p_i and the *buffer_id*. If p_i can get the *buffer_id* from the *buffer_id* map, it means p_i is not the first arrival packet of F, then the switch directly buffers p_i without triggering a *pkt_in* message (line 10∼11). If the controller doesn't send back control operation messages after a timeout period (line 12), the switch needs to send another *pkt_in* message to the controller (line 13).

Algorithm 1 Buffer Each Miss-match Packet

Input: a flow F with n packets arriving
Output: buffer each miss-match packet of F
1: **for** each arrival packet p_i of F **do**
2:   **if** p_i matches a rule of flow table **then**
3:     the switch forwards p_i;
4:   **else**
5:     *buffer_id* ← getBufferIdFromMap(p_i);
6:     **if** *buffer_id*$= -1$ **then**
7:       *buffer_id* ← bufferFirstPacket(p_i);
8:       storeBufferIdIntoMap(p_i, *buffer_id*);
9:       send a *pkt_in* message including header of p_i and *buffer_id*;
10:     **else**
11:       bufferSubsequentPacket(p_i, *buffer_id*);
12:       **if** timestamp expires **then**
13:         send a *pkt_in* message including header of p_i and *buffer_id*;
14:       **end if**
15:     **end if**
16:   **end if**
17: **end for**

Algorithm 2 describes how to forward the buffered packets of a flow. A *pkt_in* message will introduce a *flow_mod* message and a *pkt_out* message. When receiving the *flow_mod* message, the switch installs the forwarding rule in the flow table (line 1). However, this rule only applies to the subsequent arrival packets of F, not to the already buffered packets of F. When the *pkt_out* message arrives, it carries the *buffer_id* (line 2) and the *out_port* (line 3). The switch uses *buffer_id* to get the first buffered packet of F (line 4), and forwards it through the *out_port* of the switch (line 5). Then, the switch forwards other buffered packets of F one by one (line 7∼8). Meanwhile, corresponding buffer units are released (line 6 and line 9).

B. Performance Evaluation

We implement the proposed buffer mechanism in *Open vSwitch*, and evaluate its performance compared with the default buffer mechanism. In this experiment, the buffer is

Algorithm 2 Forward Each Buffered Packet

Input: *flow_mod* and *pkt_out* messages arriving
Output: Forward each buffered packet of F

1: modify the flow table based on *flow_mod* message;
2: *buffer_id* ← getBufferId (*pkt_out*);
3: *out_port* ← getOutPort (*pkt_out*);
4: firstPacket ← getPacketFromBuffer (*buffer_id*);
5: forward (firstPacket, *out_port*);
6: releaseBufferUnit(firstPacket);
7: **while** (nextPacket ← getPacketFromBuffer(*buffer_id*)) is not null **do**
8: forward (nextPacket, *out_port*);
9: releaseBufferUnit(nextPacket);
10: **end while**

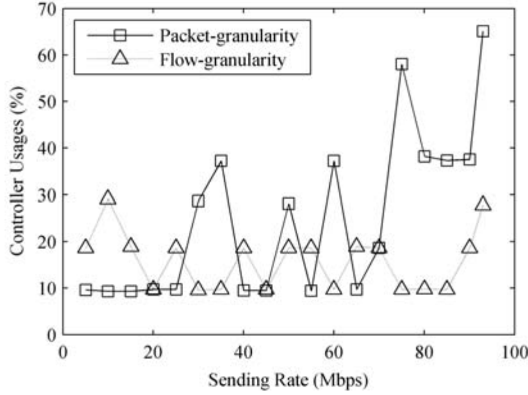

Fig. 6. Controller Usages under Different Sending Rates.

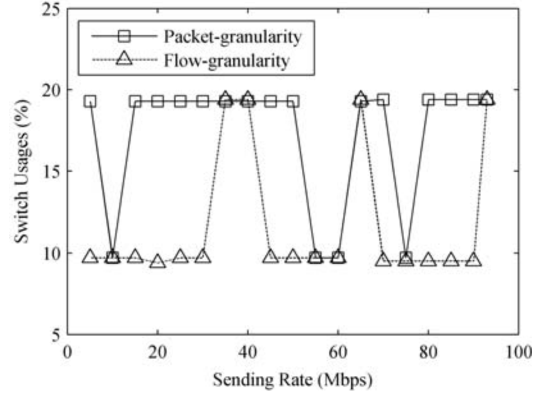

Fig. 7. Switch Usages under Different Sending Rates.

set to 256 for both of the buffer mechanisms. As shown in Fig. 1, $Host_1$ sends 50 flows (20 packets for each flow) to $Host_2$ in cross sequences. We repeat the experiments at each sending rate for 20 times.

1) Control Path Load: As depicted in Fig. 5(a), the proposed buffer mechanism introduces less control traffic sent from the switch to the controller. Using the flow-granularity buffer, control path load is kept at a low and stable level, while for the packet-granularity buffer, control path load increases quickly when the sending rate exceeds 30Mbps. The flow-granularity buffer sends only one *pkt_in* message to the controller for a flow with many miss-match packets, so it can reduce the number of *pkt_in* messages. As shown in Fig. 5(b), the flow-granularity buffer also introduces less control traffic sent to the switch. This is because fewer control request messages introduce fewer control operation messages, i.e., the *pkt_out* messages and the *flow_mod* messages. In summary, flow-granularity buffer can reduce 64% of the control path load of one direction on average and for another direction, the control path load can be reduced by 80% on average.

2) Controller Usages: As depicted in Fig. 6, the proposed buffer limits the controller usages below 30%. While using the default buffer, the controller needs more computing resources (mean of 24.82% and maximum of 65.1%) in most cases, especially when the sending rate is over 70Mbps. The flow-granularity buffer reduces the number of *pkt_in* messages, which effectively decreases the controller overhead by 35.7% on average.

3) Switch Usages: As depicted in Fig. 7, the mean switch usage of the proposed buffer is 11.67%, while the mean value is 17.31% for the default buffer. Considering the error caused by the instability of the experimental devices, they present similar switch usages. That is to say, the flow-granularity buffer mechanism complicates packet processing, but it doesn't introduce extra overhead to the switch compared with the packet-granularity buffer mechanism.

4) Buffer utilization: We calculate the average and maximum number of the buffer units that are used at each sending rate. As depicted in Fig. 8(a), the flow-granularity buffer uses fewer buffer units during the whole testing process. The flow-granularity buffer always uses no more than 5 buffer units, while for the packet-granularity buffer, the buffer utilization presents a rapid growth with the sending rate increasing, and it uses 43 buffer units at the sending rate of 95Mbps. Using the flow-granularity buffer, only one *pkt_in* message is triggered for a flow with many miss-match packets. The *pkt_in* message is given a *buffer_id* and all the buffered packets share this *buffer_id*. After the *pkt_out* message arrives, it instructs to forward the buffered packets with the same *buffer_id*. That is to say, this *pkt_out* message applies to all the buffered packets of this flow. Through this way, the buffer units can be quickly released, leading to a low utilization of the buffer space. While for the packet-granularity buffer, each miss-match packet triggers a *pkt_in* message, which is given an exclusive *buffer_id*. So a *pkt_out* message only applies to its corresponding buffered packet. This causes the buffer units released slowly and leads a high utilization of the buffer space. When the sending rate is high, the packet-granularity buffer mechanism will trigger more *pkt_in* messages in a short time period, which presents an increasing demand on buffer space. In summary, the flow-granularity buffer mechanism can quickly release the buffer units. It improves the efficiency of the buffer utilization by 71.6% on average.

VI. CONCLUSIONS

In this paper, we take a first step towards analyzing and utilizing SDN switch buffer. 1) We first analyze the benefits of adopting switch buffer. Using the default packet-granularity

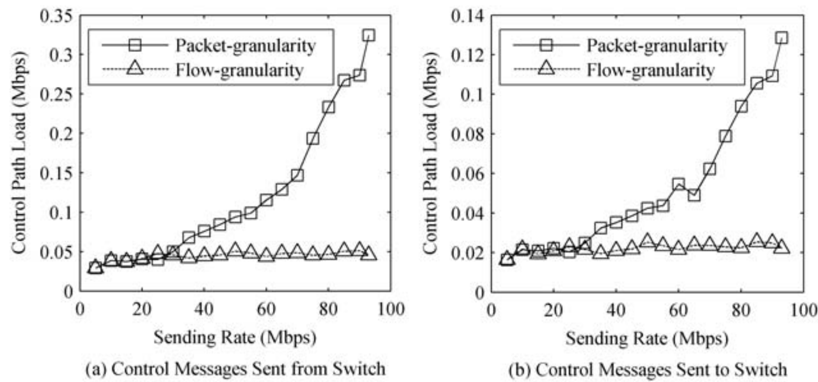

Fig. 5. Control Path Load under Different Sending Rates.

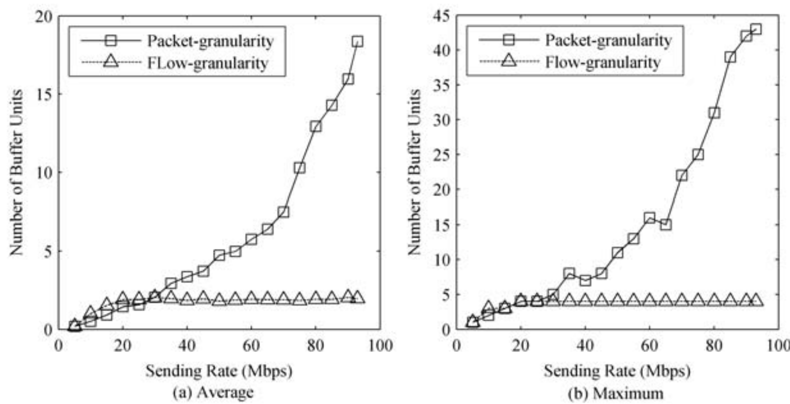

Fig. 8. Buffer Utilization under Different Sending Rates.

buffer, only several header fields instead of the entire miss-match packet are included in the request message, which reduces the request message size. Experiment results show that the default buffer could reduce the load on control path and the controller. 2) Then, we propose a flow-granularity buffer mechanism, which sends only one request message to the controller for a flow with many miss-match packets. Through this way, the number of the request messages is reduced. Experiment results show that the proposed buffer mechanism could further reduce the communication overhead without increasing the switch overhead.

ACKNOWLEDGEMENTS

This work is supported by RGC General Research Fund (GRF) with RGC No. PolyU 152244/15E; the National Natural Science Foundation of China under Grant Nos. 61602105 and 61572123; China Postdoctoral Science Foundation under Grant No. 2016M601323; the Fundamental Research Funds for the Central Universities Project under Grant No. N150403007; CERNET Innovation Project under Grant No. NGII20160126.

REFERENCES

[1] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In NSDI, 2010.
[2] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In CoNEXT, 2011.
[3] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. zUpdate: Updating Data Center Networks with Zero Loss. In SIGCOMM, 2013.
[4] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, S. Banerjee. DevoFlow: Scaling Flow Management for High-performance Networks. In SIGCOMM, 2011.
[5] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable Flow-based Networking with DIFANE. In SIGCOMM, 2010.
[6] OpenFlow Switch Specification. https://www.opennetworking.org/.
[7] E. D. Kim, Y. Choi, S. I. Lee, M. K. Shin, H. J. Kim. Flow table management scheme applying an LRU caching algorithm. In ICTC, 2014.
[8] K. He, J. Khalid, S. Das, A. Akella, E. L. Li, and M. Thottan. Mazu: Taming latency in software defined networks. University of Wisconsin-Madison Technical Report, 2014.
[9] K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, E. L. Li, M. Thottan. Measuring control plane latency in SDN-enabled switches. In SOSR, 2015.
[10] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, M. Casado. The Design and Implementation of Open vSwitch. In NSDI, 2015.
[11] Floodlight. http://www.projectfloodlight.org/floodlight/.
[12] R. Olsson. Pktgen the Linux Packet Generator. In Ottawa Linux Symposium, 2005.
[13] Tcpdump. http://www.tcpdump.org/.