This is the Pre-Published Version.

The following publication F. Li, J. Cao, X. Wang, Y. Sun, T. Pan and X. Liu, "Applying Buffer to SDN Switches: Benefits Analysis and Mechanism Design," in IEEE Transactions on Cloud Computing, vol. 9, no. 1, pp. 54-65, 1 Jan.-March 2021 is available at https://doi.org/10.1109/TCC.2018.2846620.

Applying Buffer to SDN Switches : Benefits Analysis and Mechanism Design

Fuliang Li, *Member, IEEE,* Jiannong Cao, *Fellow, IEEE*
Xingwei Wang, Yinchu Sun, Tian Pan, and Xuefeng Liu

***Abstract*—Software-Defined-Networking (SDN) is progressively dominating the dynamic management for timely network trouble shooting and fine grained traffic scheduling in data center networks. 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.

***Index Terms*—Software Defined Networking, data center networks, communication overhead, switch, buffer.**

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 network infrastructure [3, 6, 28], which is the foundation for building today's cloud computing services. 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. The reasons of keeping the control traffic at a low level includes two aspects. On the one hand, it can reduce the bandwidth consumption of the control path, as well as relieve the load on the controller. This is especially valuable for the SDN applications with only one controller. In addition, for some SDN applications, the control path shares the same physical links with the data path. The control messages may be congested if the control path carries too much control traffic, even though a certain percentage of the bandwidth is reserved for the control messages. On the other hand, existing studies have proved that the control messages compete for the limited resources of the switch, which will increase the communication delay between the switch and the controller [9]. Therefore, it is critical to reduce the switch-controller control traffic. The challenge is that it is hard to quantify the impact of the control traffic on performance. Alternatively, In this paper, we observe the performance changes after the control traffic is reduced by introducing the buffer mechanism.

Existing studies have tried to reduce the communication overhead by cloning more forwarding rules in intermediate proxy [10] or authority devices [15], 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

Fuliang Li is with the School of Computer Science and Engineering, Northeastern University, Shenyang, China, and the Department of Computing, Hong Kong Polytechnic University, Hong Kong, China (e-mail: lifuliang@cse.neu.edu.cn).

Jiannong Cao is with the Department of Computing, Hong Kong Polytechnic University, Hong Kong, China (e-mail: jiannong.cao@polyu.edu.hk).

Xingwei Wang is the corresponding author, with the School of Computer Science and Engineering, Northeastern University, Shenyang, China (e-mail: wangxw@mail.neu.edu.cn).

Yinchu Sun is with the School of Computer Science and Engineering, Northeastern University, Shenyang, China (e-mail: qinqinmuji@163.com).

Tian Pan is with the Department of Information and Communication Engineering, Beijing University of Posts and Telecommunications, Beijing, China (e-mail: platinum127@gmail.com).

Xuefeng Liu is with the School of Electronic Information and Communications, Huazhong University of Science and Technology, Wuhan, China (e-mail: csxfliu@gmail.com).

©2018 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.

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. We measure the performance impact on a variety of metrics under different buffer size settings, including control path load, controller overhead and flow setup delay, etc. 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. And if there is enough buffer space, controller delay, the switch delay and the flow setup delay can also be reduced by 58%,87% and 78% respectively on average.

(2) We propose a flow-granularity buffer mechanism for SDN switches. Assuming that a new flow contains many packets arriving concurrently. 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 this flow, which will reduce the number of the request messages. We evaluate the proposed buffer mechanism through a comparison analysis with the default one. 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. In addition, the proposed buffer mechanism can improve the buffer utilization by 71.6% on average. And Although the proposed buffer mechanism introduces complexity to process the miss-match packets, it does not significantly increase the flow setup delay, while reducing the flow forwarding delay by 18% on average.

One may doubt that buffer is not necessary 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 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. According to OpenFlow switch specification 1.5.1 [26], the buffer is well defined. However, until now, it is not well studied at all. Therefore, we analyze the effects of the default buffer mechanism and propose a new buffer mechanism in this paper. The default buffer mechanism can reduce the size of the request message, and the OpenFlow protocol just needs to be configured correctly. The proposed buffer mechanism can further reduce the number of request messages, and the OpenFlow protocol needs to be extended mainly from two aspects: 1) how to buffer each miss-match packet; 2) and how to forward each buffered packet.

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 discuss the adoption of SDN switch buffer in Section V and conclude the whole paper in Section VI .

II. Related Work

SDN has become a promising network technology and it has been deployed to a wide range of network environment, including campus networks [1], enterprise networks [2], data center networks [3], cellular networks [4], satellite networks [5], etc. 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 [13] addressed the controller overhead when packets cannot match any rule of the flow tables. Song et al. [30] proposed and developed a control path management framework to enhance SDN reliability, including several control path reliability algorithms and a novel control message classification and prioritization system. Curtis et al. [10] 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* [15] 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 sent to the controller through deploying intermediate devices, which cache more forwarding rules and play a part role of the controller. Different from previous works, this paper takes full utilization of switch buffer to reduce the size of the request message, as well as the the number of the requests generated by the switches. Actually, the buffer mechanism could also be utilized as a supplement to existing approaches, because it takes effect before the request messages sent out. This paper proves the value of adopting switch buffer and propose a new buffer mechanism to further reduce the switch-controller communication overhead.

Fine-grained control and real-time sensitivity applications require quick adaptation to the changes of network topologies or traffic patterns [6, 7]. Therefore, researchers also pay attention to understand SDN switch performance, which is more or less related to or affected by the communication overhead between the switches and the controller. Rotsos et al. [11] proposed an open framework to evaluate OpenFlow switch implementations. They found that the switching performance

depends on applied actions and firmware. Huang et al. [12] investigated three commercial OpenFlow switches and pointed out that control path delays and flow table designs affect switching performance. Tai et al. [14] uncovered the source of forwarding latency caused by forwarding rule insertions on data plane. Xu et al. [31] proposed two cost-optimized flow statistics collection schemes using wildcard-based requests, which could reduce the bandwidth overhead and switch processing delay. He et al. [8, 9] studied four SDN switches and showed the latencies underlying the generation of request messages and execution of forwarding rules. Different from previous works, we try to analyze the benefits of adopting switch buffer through an in-depth measurement study, which could make a complementary exploitation of SDN switches. Observed from the analysis results, we find that switch buffer can reduce the communication overhead between the switches and the controller.

The root cause of communication overhead stems from the size limitation of SDN flow tables. Rules in flow tables have to be updated adapting to network dynamics, i.e., rules for inactive flows will be kicked out and replaced by rules for active flows. As a result, packets cannot always match the rules of the flow tables. Therefore, optimizing flow table utilization (such as caching more rules, updating rules quickly, etc.) can reduce the requests sent to the controller indirectly. Luo et al. [16] shrank the flow table size and provided practical methods to achieve fast flow table updates. Li et al. [17] proposed an efficient flow-driven rule caching algorithm to optimize the SDN switch cache replacement. *CacheFlow* [18] spliced long dependency chains to cache smaller groups of rules while preserving the per-rule traffic counts. *FlowShadow* [19] achieved fast packet processing and supports uninterrupted update by caching microflows. Zhang et al. [32] propose a delay-guaranteed approach called D^3G to reduce the latency of chained services while obtain fairness across all the workloads, by means of designing a latency estimation algorithm and a feedback scheme. Yan et al. [29] proposed a new rule caching scheme, as well as an adaptive cache management method. The mechanism can reduce the cache miss rate by one order of magnitude and the control path bandwidth usage by a half. Different from these studies, this paper exploits the benefits of switch buffer to directly reduce the communication overhead. Buffer has been adopted to improve performance and achieve QoS guarantee in legacy switches [20, 21]. However, it receives little notice in SDN.

III. Problem and Experiment Description

In this section, we first describe the problem and then show the measurement methodology.

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

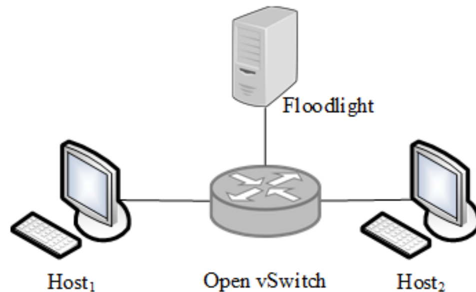


Fig. 1: Topography of the Experimental Platform.

B. Experiment Description

Fig. 1 shows our experiments setup. *Open vSwitch*(OVS) [22] is an open source OpenFlow virtual switch. *Floodlight* [23] 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* [24] on $Host_1$ to generate traffic at rates of 5Mbps - 100Mbps with the Ethernet frame size of 1000 Bytes. We run *tcpdump* [25] to listen on the interfaces that are connected to the hosts and the controller respectively.

In this paper, we utilize switch buffer to reduce the communication overhead from the following two ways: (1) reduce

the size of the request messages; (2) reduce the number of the request messages. We observe the benefits of adopting the default buffer mechanism and the proposed buffer mechanism from many metrics, including:

Control path load: control traffic on the control path.

Controller usages: CPU utilization of the *floodlight* process.

Switch usages: CPU utilization of the switch.

Flow setup delay: the time consumption starting from the first packet of a flow entering the switch to the packet leaving the switch.

Controller delay: the time consumption starting from the *pkt_in* message leaving the switch to the *flow_mod* or *pkt_out* message arriving at the switch.

Switch delay: the difference between the flow setup delay and the controller delay.

Buffer utilization: the number of buffer units used to store the miss-match packets.

IV. Benefits of Adopting Switch Buffer

Adopting the default buffer of *OVS*, the switch buffers each miss-match packet, and can let only several header fields instead of the entire miss-match packet include in the *pkt_in* message. Note that the actual length of the data field in the message depends on how to configure the parameter of the *pkt_in* message. It is easy to implement if the controller want to see the entire packet (considering the security applications). 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 1000 new flows to $Host_2$ at each sending rate. Each flow includes one packet. To generate new flows, we use *pktgen* [24] to forge source IP addresses. 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

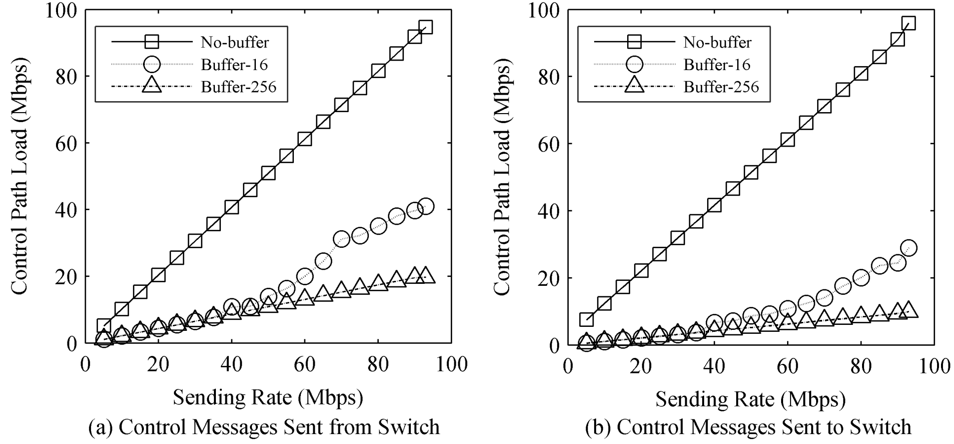
We measure controller usages (i.e., CPU utilization of the *floodlight* process) to evaluate the load on the controller. As depicted in Fig. 3, controller usage tends to be a linear growth when the sending rate is lower than 50Mbps. After that point, controller usage of no-buffer presents an approximate exponential variation 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 *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.

D. Flow Setup Delay

Flow setup delay is used to evaluate the responsiveness of switches to arrival flows. As shown in Fig. 5, when sending rate is lower than 70Mbps, the flow setup delay variations present similar patterns across different buffer sizes. Without



(a) Control Messages Sent from Switch

(b) Control Messages Sent to Switch

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

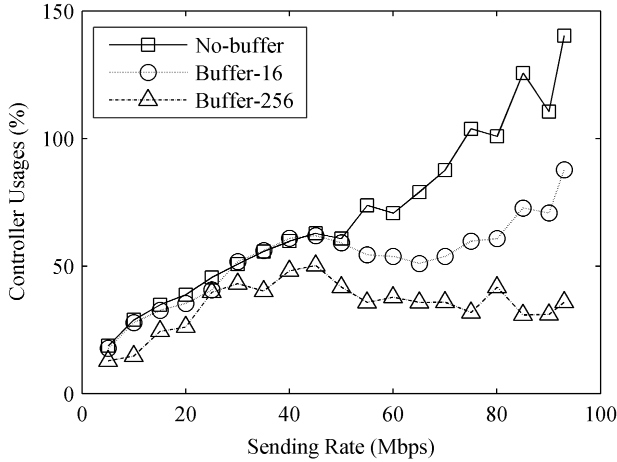


Fig. 3: Controller Usages under Different Sending Rates.

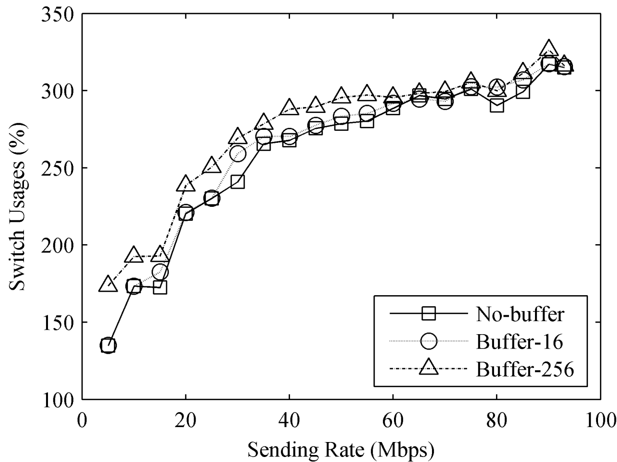


Fig. 4: Switch Usages under Different Sending Rates.

buffer, the mean is 5.28ms and the standard deviation is 8.74ms. When the sending rate exceeds 70Mbps, the delay of no-buffer is quite variable with the maximum of 30.46ms. The mean delay of buffer-16 is 1.98ms. It presents a similar pattern to no-buffer with a standard deviation of 1.85ms. The delay of buffer-256 maintains a stable variation with a mean of 1.17ms, a standard deviation of 0.37ms and the maximum of 5.35ms. Normally, buffering the first miss-match packet and forwarding the buffered packet of a flow introduce buffer related operations, which will increase the flow setup delay of the flow. But we find that the flow setup delay can be reduced by 78% on average when using buffer-256. We next explain the reasons and give the effective condition of buffer on reducing the flow setup delay.

The flow setup delay of a new flow is mainly composed of five parts: the delay of buffering the first miss-match packet, marked as T_{buffer}; the delay of generating the *pkt_in* message by the switch, marked as T_{pkt_in}; the controller delay defined in Section IV.E, marked as $T_{s \to c \to s}$; the delay of processing the *pkt_out* message by the switch, marked as T_{pkt_out}; the delay of releasing the buffered packets according to the *pkt_out* message, marked as $T_{release}$. Without buffer, each miss-match packet is entirely included in the *pkt_in* message and the *pkt_out* message. These control messages compete for the limited bus bandwidth between the ASIC and switch CPU, the bandwidth of the control path, and the computing resource of the controller, which greatly affects the generation of the *pkt_in* messages, the generation of the *pkt_out* messages and the processing of the *pkt_out* messages [8]. Correspondingly, the values of T_{pkt_in}, $T_{s \to c \to s}$ and T_{pkt_out} will be larger than adopting buffer. Because when using buffer, only several bytes of each miss-match packet are included in the *pkt_in* message, and the *pkt_out* message carries even less information than the *pkt_in* message. As a result, less control traffic competes for the limited resources. However, buffer adoption brings in T_{buffer} and $T_{release}$. So the effective condition of buffer on reducing the flow setup delay can be expressed as that the buffer delay ($T_{buffer} + T_{release}$) is less than the additional and variable part of the request delay (T_{pkt_in}, $T_{s \to c \to s}$ and T_{pkt_out}) caused by competing the limited resources. Our experimental results show that when the sending rate is high, buffer can reduce the flow setup delay obviously. This is easy to explain according to the given effective condition. Without buffer, high throughput means that more large control messages compete

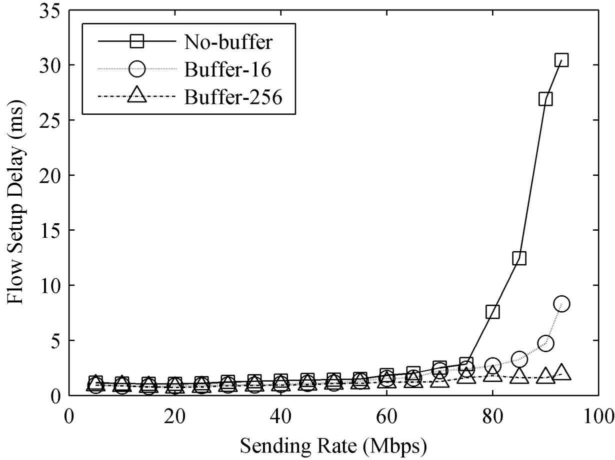


Fig. 5: Flow Setup Delay under Different Sending Rates.

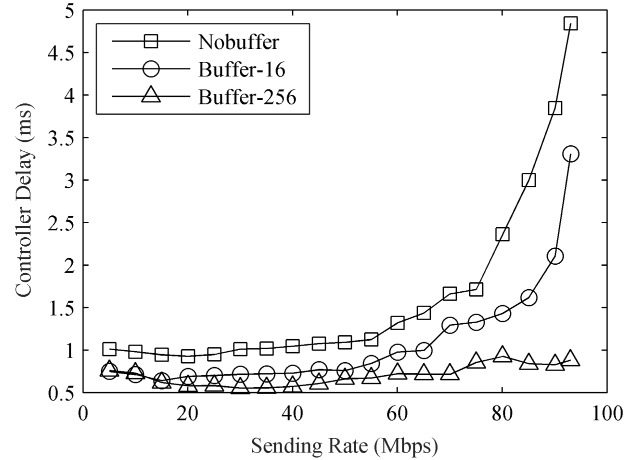


Fig. 6: Controller Delay under Different Sending Rates.

for the limited resources. This needs more extra delay to generate and process the control messages. We also find that buffer-256 performs better than buffer-16, this is because buffer-16 is over utilized when sending rate is high, which complicates the buffer operations and introduces more buffer delay than buffer-256 with enough buffer space.

In summary, in addition to reducing the communication overhead, adopting a large enough buffer can also reduce the flow setup delay. However, the flow setup delay is affected by many factors, such as flow patterns, line rate and processor capabilities, etc. So it is difficult to quantitatively evaluate the flow setup delay under different conditions. Given that, we conduct a qualitative analysis of how the buffer benefits the flow setup delay. The experimental result has shown another positive incentive to adopt SDN switch buffer.

E. Controller Delay

We record the timestamp (t_1) when the *pkt_in* message is sent from the switch, and the timestamp (t_2) when the *flow_mod* or *pkt_out* message arrives at the switch. The difference ($t_2 - t_1$) is roughly considered as the controller delay. As shown in Fig. 6, the controller delay of no-buffer is always higher than that of buffer-16 and buffer-256. The controller delay of no-buffer (mean of 1.65ms, maximum of 4.84ms and standard deviation of 1.1ms) presents an obvious increase beginning at the sending rate of 60Mbps. The controller delay of buffer-16 (mean of 1.11ms and standard deviation of 0.66ms) follows the same trend with no-buffer, while the controller delay of buffer-256 (mean of 0.70ms and standard deviation of 0.12ms) keeps stable during the whole testing process. When the buffer is not adopted, the controller should first capture the required information from the *pkt_in* message, which includes the entire miss-match packet. After the decision is made, it also needs to encapsulate the entire miss-match packet into the *pkt_out* message. These operations are more time consuming than adopting the buffer, which lets some necessary header fields of each miss-match packet include in the *pkt_in* message. Buffer-16 cannot provide available buffer units when the sending rate is high, so it behaves poorly compared with buffer-256. In summary, if we provide enough buffer space, we can reduce 58% controller delay on average.

F. Switch Delay

In our study, the flow setup delay consists of switch delay and controller delay. The switch delay refers to the generation time of the *pkt_in* message and execution time of the *pkt_out* message. Execution time of the *flow_mod* message, i.e., rule installation time has been studied in an existing work [8]. As depicted in Fig. 7, when sending rate is lower than 75Mbps, switch delay presents no differences among the three buffer sizes. After that point, switch delay of no-buffer increases quickly. It reaches to 25.07ms when the sending rate is 95Mbps. Switch delay of buffer-16 (mean of 0.87ms and standard deviation of 1.18ms) outperforms no-buffer. Switch delay of buffer-256 (mean of 0.47ms and standard deviation of 0.27ms) always keeps at a low and stable level. Without buffer, large *pkt_in* messages are sent out, while at the same time, *flow_mod* messages and large *pkt_out* messages are sent back. The going out and coming in control messages compete for the limited switch resources, especially the bus bandwidth between the ASIC and switch CPU [8], which will delay the control messages to take effect. This situation will cause obvious consequence when the control traffic exceeds a threshold. In summary, the switch delay can be reduced by 87% on average by adopting a large enough buffer.

G. Buffer Utilization

Aforementioned analysis results reveal that when buffer is exhausted, performance metrics like control path load, controller usages and flow setup delay, etc., start to increase gradually or sharply. Therefore, avoiding buffer exhaustion is crucial. As a basis, buffer utilization should be investigated first. As depicted in Fig. 8, when the sending rate exceeds 30Mbps, buffer-16 is exhausted. After the buffer is exhausted, the buffer can still work in the full utilization condition. When a *pkt_out* message arrives, the buffered packet will be forwarded and corresponding buffer units will be released. New miss-match packets will be buffered when there are

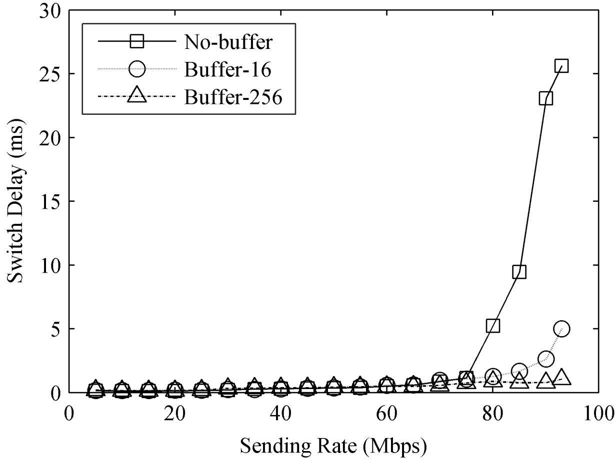


Fig. 7: Switch Delay under Different Sending Rates.

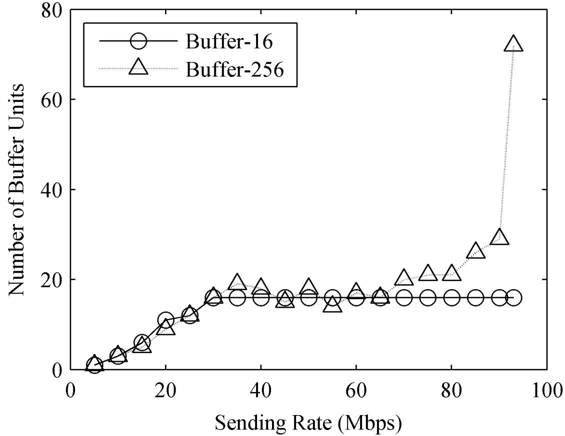


Fig. 8: Buffer Utilization under Different Sending Rates.

available buffer units. However, when the sending rate exceeds a threshold, the replacement of the buffer units cannot catch up with the speed of the coming miss-match packets. Each miss-match packet will be entirely included in the *pkt_in* message, causing the performance to decrease quickly. In our study, buffer-256 is sufficient to meet the requirements of the experimental flow patterns. For buffer-256, when the sending rate is over 70Mbps, more buffer units are needed. And no more than 80 buffer units can meet the maximum sending rate. That is to say, if each packet size is 1000 Bytes on average, a 80 KBytes buffer is satisfied for a 100Mbps SDN switch interface. This is a positive indicator for considering adopting the SDN switch buffer.

V. Flow-granularity Buffer Mechanism

We have proven the benefits of adopting switch buffer. However, 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 new flow with many packets, the first arrival packet triggers a *pkt_in* message, while the subsequent packets of this flow arrive before the control operation messages are sent back and take effect. These subsequent miss-match packets of the flow continue to trigger redundant *pkt_in* messages sent to the controller. Therefore, in addition to reducing the request message size, we further aim to reduce the number of the request messages.

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). In the OpenFlow specification [26], *buffer_id* is used to identify a packet buffered at the switch and sent to the controller by a *pkt_in* message. A *pkt_out* message including a valid *buffer_id* removes the corresponding packet from the buffer and processes the packet by the actions of the message. 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). Operations of the proposed buffer mechanism are a bit more complex than the default buffer mechanism. Therefore, it requires to extend the OpenFlow protocol. If the controller does not 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 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

Algorithm 1 Buffer Each Miss-match Packet

Input: a flow F with n packets arriving
Initialization: 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 Forward Each Buffered Packet

Input: *flow_mod* and *pkt_out* messages arriving
Initialization: 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
```

default buffer mechanism. In this experiment, the buffer is set to 256 for both buffer of the mechanisms. As shown in Fig. 1, $Host_1$ sends 50 flows to $Host_2$. Each flow includes 20 packets. During the test, we first send out 5 flows (i.e., 100 packets) in cross sequences. Then, another 5 flows will be sent to $Host_2$ in the same way. The test will not terminate until all the 50 flows are sent out. We repeat the experiments at each sending rate for 20 times.

1) Control Path Load: As depicted in Fig. 9(a), the proposed buffer mechanism introduces less control traffic sent from the switch to the controller. Using the flow-granularity buffer, control path load (mean of 0.045Mbps and standard deviation of 0.005Mbps) is kept at a low and stable level, while for the packet-granularity buffer, control path load (mean of 0.123Mbps and standard deviation of 0.009Mbps) 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. 9(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. 10, 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. 11, the two buffer mechanisms present similar switch usage patterns. Since the experiment flows do not burden the switch, the switch utilization is low. 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 experiment platform which is built based on software switch, the flow-granularity buffer mechanism doesn't introduce extra overhead to the switch compared with the packet-granularity buffer mechanism, although the new buffer complicates the packet processing.

4) Flow Setup Delay: Fig. 12(a) shows the flow setup delay variations of the buffer mechanisms under different sending rates. Using the proposed buffer, the mean is 2.05ms and the standard deviation is 0.46ms, while using the default buffer, the mean is 1.53ms and the deviation is 0.69ms. Although the flow-granularity buffer reduces the number of *pkt_in* messages, it introduces extra operations to the switch, which delays the generation of *pkt_in* messages. So the packet-granularity buffer performs better than the flow-granularity buffer in most cases. However, when the sending rate is high, the competition for the bandwidth and computing resources increases, and the benefits of reducing the number of *pkt_in* messages start to appear. For example, when the sending rate exceeds 80Mbps, the flow-granularity buffer outperforms the packet-granularity buffer.

We also use flow forwarding delay (starting from the first packet of a flow entering the switch to the last packet of that flow leaving it) to compare the proposed buffer mechanism to the default one. As depicted in Fig. 12(b), the flow-granularity buffer behaves similarly to the packet-granularity buffer in most cases. However, when the sending rate exceeds 80Mbps, the flow-granularity buffer presents its advantages. For example, the forwarding delay of the packet-granularity buffer is 54.71ms at the sending rate of 95Mbps, while the value is 34.23ms for the flow-granularity buffer. There are two reasons explaining the differences. 1) When the sending rate is high, packets of a flow arrive almost at the same time, which means that the number of the miss-match packets increases, and the packet-granularity buffer introduces more control messages competing for the limited resources. As a result, the decision making and taking effect process is delayed. 2) For the packet-granularity buffer, each buffered packet of a flow can be forwarded only after its corresponding *pkt_out* message is sent back. However, for the flow-granularity buffer, the switch sends only one *pkt_in* message for a flow, and it forwards all the buffered packets of that flow immediately when the unique *pkt_out* message arrives.

In summary, the flow-granularity buffer introduces complexity to process the miss-match packets, but it doesn't significantly increase the flow setup delay, while reducing the flow forwarding delay by 18% on average. We also find that the flow-granularity buffer mechanism performs better than packet-granularity buffer mechanism when the sending

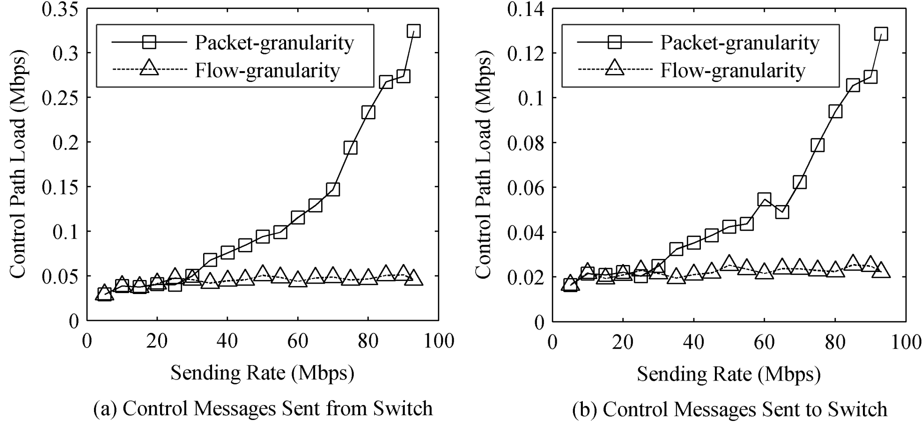


Fig. 9: Control Path Load under Different Sending Rates.

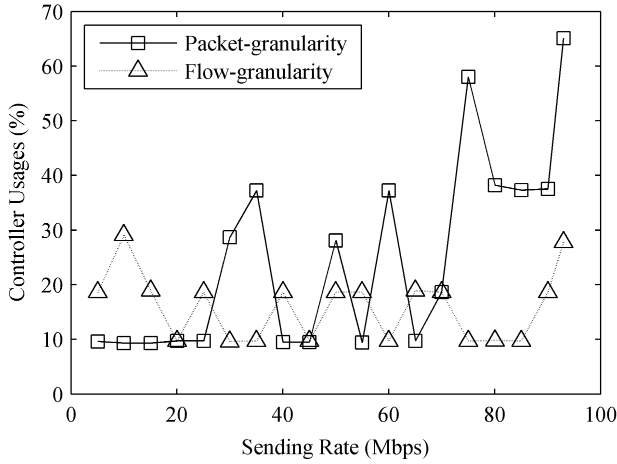


Fig. 10: Controller Usages under Different Sending Rates.

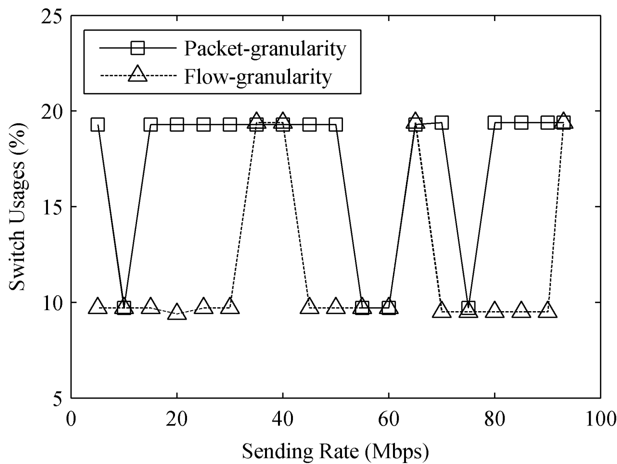


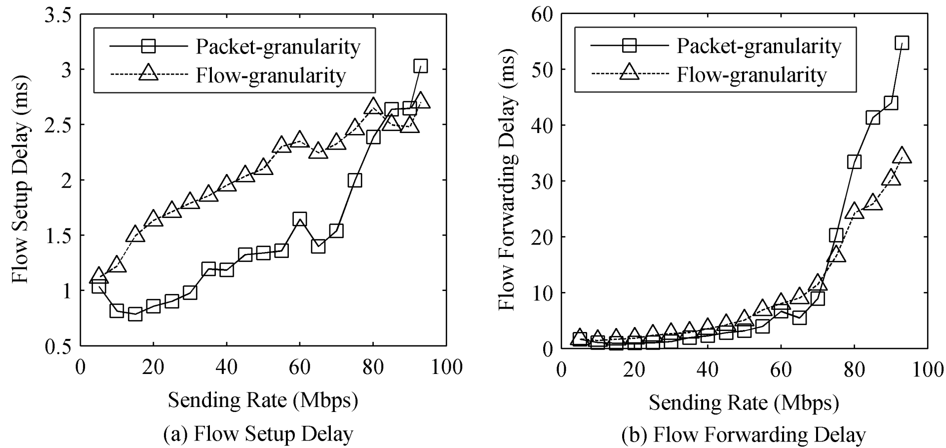
Fig. 11: Switch Usages under Different Sending Rates.

rate exceeds a threshold. For example, when the sending rate reaches to 95Mbps, the proposed buffer mechanism can reduce 10.8% of the flow setup delay and 37.4% of the flow forwarding delay.

5) Buffer utilization: We calculate the average and maximum number of the buffer units that are used at each sending rate. As depicted in Fig. 13(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 (standard deviation of 5.57) 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. Note that due to the experiment conditions, the buffer utilization of the packet-granularity buffer presents differences under different sending rates as shown in Fig. 8 and Fig. 13.

VI. Discussion of Adopting SDN Switch Buffer

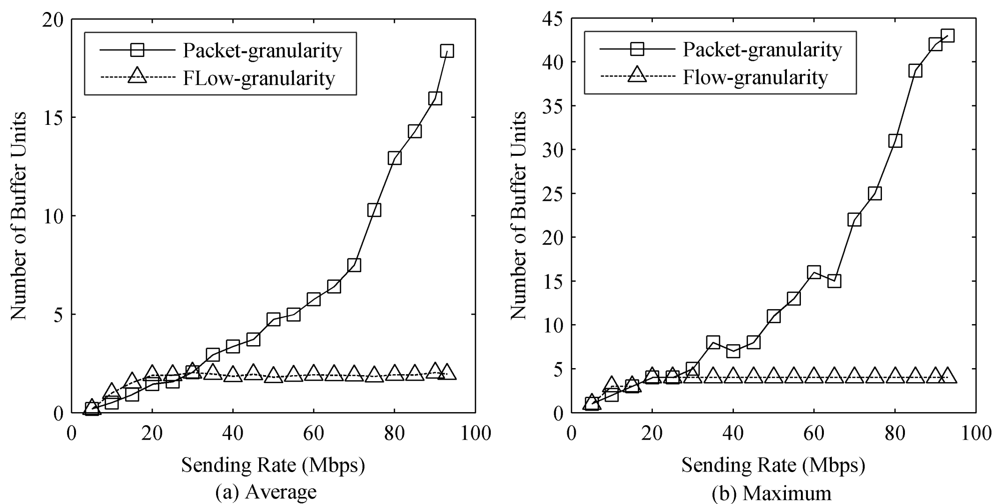
In OpenFlow switch specifications [26], buffer is not utilized by default. This paper evaluates the value of buffer adoption on UDP flows. We may consider that a TCP flow setup begins with several small packets negotiating first. So the buffer might not be necessary for TCP flows. To address the doubts, this section discusses how the buffer benefits both of the UDP flows and the TCP flows.



(a) Flow Setup Delay

(b) Flow Forwarding Delay

Fig. 12: (a) Flow Setup Delay and (b) Flow Forwarding Delay under Different Sending Rates.



(a) Average

(b) Maximum

Fig. 13: Buffer Utilization under Different Sending Rates.

A. Buffer for UDP Flows

UDP is a connectionless oriented protocol, i.e., packets of UDP flow transmit directly without the process of connection establishment. Without buffer, each miss-match packet will trigger a request message, which includes the entire packet. Massive packets of a new UDP flow may arrive around the same time. Many request messages for this flow will be sent to the controller, causing heavy load on the control path and to the controller.

In this study, we use UDP flows to evaluate the benefits of switch buffer and the efficiency of the proposed buffer mechanism. Results reveal that for UDP flows, using buffer could obviously reduce the communication overhead. Therefore, a good switch buffer mechanism is really needed to apply for UDP flows. Existing studies have found that although TCP still dominates network traffic in terms of packets and bytes, UDP now often accounts for the largest fraction of flows in a given link [27]. So using UDP flows to evaluate the effectiveness of the switch buffer is convincing. If switch buffer benefits UDP flows, it also benefits the mix of TCP and UDP flows.

B. Buffer for TCP Flows

TCP is a connection oriented protocol, which adopts a three-way handshake process to establish a connection. In general, the first few packets of TCP flows which are used to establish the connection are very small (e.g. TCP SYN, TCP ACK) and no data will be transferred until the connection is established. After the connection is established, subsequent large packets of this flow arrive successively, and the corresponding forwarding rule has taken effect. These packets sent after the connection establishment can match the rule of the flow table and will no longer be treated as the packets of a new flow. From this view, switch buffer is not necessary for TCP flows. However, buffer can also apply to TCP flows for the following reasons.

1) For some TCP connection flows, the flows may present inactive for a transient time period. The rules may be kicked out from the size limited flow tables due to the inactivity, but the connections are not terminated in actual. After that transient period, corresponding ends may restore large volume of data transmission. In this case, switch buffer is useful.

2) The main objective of this study is to reduce the

communication overhead through adopting the switch buffer. Experiment results show that using the default buffer, the communication overhead is obviously reduced. Meanwhile, the flow setup delay is decreased and kept at a stable level. The proposed buffer can also reduce the flow setup delay when the sending rate is high. Therefore, buffering TCP flows can at least gain the benefits from reducing the flow setup delay.

VII. Conclusion and Future Remarks

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 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, as well as decrease the flow setup delay. 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.

In this paper, we evaluate the benefits of the default and the proposed buffer mechanisms on the Open vSwitch with 100Mbps Ethernet. In the future, we will further evaluate the benefits of buffer adoption through commodity SDN switches with Gigabit Ethernet. In addition, we can design egress scheduling mechanisms combing with the ingress buffer mechanism proposed in this paper to provide QoS guarantee for different applications.

Acknowledgements

The authors would like to thank the anonymous reviewers for their constructive comments. This work is supported by the National Natural Science Foundation of China under Grant Nos. 61602105 and 61572123; the Program for Liaoning Innovative Research Term in University under Grant No. LT2016007; the Fundamental Research Funds for the Central Universities Project under Grant No. N171604006; CERNET Innovation Project under Grant No. NGII20170121. The corresponding author is X.W. Wang.

References

[1] 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, 2008, 38(2): 69-74.

[2] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker. Rethinking Enterprise Network Control. IEEE/ACM Transactions on Networking (ToN), 2009, 17(4): 1270-1283..

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In Proceedings of the 7th USENIX Symposium on Networked System Design and Implementation (NSDI), 2010: 19-19.

[4] X. Jin, L. E. Li, L. Vanbever, and J. Rexford. SoftCell: Scalable and Flexible Cellular Core Network Architecture. In Proceedings of the 9th ACM conference on Emerging networking experiments and technologies (CoNEXT), 2013: 163-174.

[5] L. Bertaux, S. Medjiah, P. Berthou, S. Abdellatif, A. Hakiri, P. Gelard, F. Planchou, M. Bruyere. Software defined networking and virtualization for broadband satellite network. IEEE Communications Magazine, 2015, 53(3): 54-60.

[6] T. Benson, A. Anand, A. Akella, and M. Zhang. Microte: Fine grained traffic engineering for data centers. In Proceedings of the 7th ACM COnference on emerging Networking EXperiments and Technologies (CoNEXT), 2011: 8.

[7] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic Scheduling of Network Updates. ACM SIGCOMM Computer Communication Review, 2014, 44(4): 539-550.

[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 Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR), 2015: 25.

[10] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, S. Banerjee. DevoFlow: Scaling Flow Management for High-performance Networks. ACM SIGCOMM Computer Communication Review, 2011, 41(4): 254-265..

[11] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, A. W. Moore. Oflops: An Open Framework for Openflow Switch Evaluation. In International Conference on Passive and Active Network Measurement. Springer Berlin Heidelberg, 2012: 85-95.

[12] D. Y. Huang, K. Yocum, A. C. Snoeren. High-fidelity Switch Models for Software-defined Network Emulation. In Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking (HotSDN), 2013: 43-48.

[13] E. D. Kim, Y. Choi, S. I. Lee, M. K. Shin, H. J. Kim. Flow table management scheme applying an LRU caching algorithm. In International Conference on Information and Communication Technology Convergence (ICTC), 2014: 335-340.

[14] D. Tai, H. Dai, T. Zhang, B. Liu. On Data Plane Latency and Pseudo-TCP Congestion in Software-Defined Networking. In Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems (ANCS), 2016: 133-134.

[15] M. Yu, J. Rexford, M. J. Freedman, J. Wang. Scalable Flow-based Networking with DIFANE. ACM SIGCOMM Computer Communication Review, 2010, 40(4): 351-362.

[16] S. Luo, H. Yu. Fast incremental flow table aggregation in SDN. In IEEE International Conference on Computer Communication and Networks (ICCCN), 2014: 1-8.

[17] H. Li, S. Guo, C. Wu, J. Li. FDRC: Flow-driven rule caching optimization in software defined networking. In IEEE International Conference on Communications (ICC), 2015: 5777-5782.

[18] N. Katta, O. Alipourfard, J. Rexford, D. Walker. Infinite cacheflow in software-defined networks. In Proceedings of the third workshop on Hot topics in software defined networking (HotSDN), ACM, 2014: 175-180.

[19] Y. Wang, D. Tai, T Zhang, B. Xu, L. Jin, H. Dai, B. Liu, X. Wu. Flowshadow: a fast path for uninterrupted packet processing in SDN switches. In Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for networking and communications systems (ANCS), 2015: 205-206.

[20] N. Beheshti, Y. Ganjali, R. Rajaduray, D. Blumenthal, N. McKeown. Buffer sizing in all-optical packet switches. In Optical Fiber Communication Conference and the National Fiber Optic Engineers Conference, 2006: 3.

[21] P. Eugster, K. Kogan, S. Nikolenko, A. Sirotkin. Shared memory buffer management for heterogeneous packet processing. In IEEE 34th International Conference on Distributed Computing Systems (ICDCS), 2014: 471-480.

[22] 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 Proceedings of the 7th USENIX Symposium on Networked System Design and Implementation (NSDI), 2015: 117-130.

[23] Floodlight. http://www.projectfloodlight.org/floodlight/.

[24] R. Olsson. Pktgen the Linux Packet Generator. In Ottawa Linux Symposium, 2005.

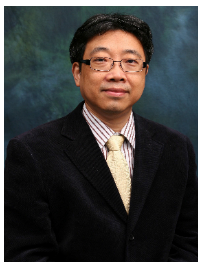
[25] Tcpdump. http://www.tcpdump.org/.

[26] OpenFlow Switch Specification. https://www.opennetworking.org/software-defined-standards/specifications/.

[27] Analyzing UDP usage in Internet traffic. https://www.caida.org/research/traffic-analysis/tcpudpratio/.

[28] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. zUpdate: Updating Data Center Networks with Zero Loss. ACM SIGCOMM Computer Communication Review, 2013, 43(4): 411-422.
[29] B. Yan, Y. Xu, H. J. Chao. Adaptive Wildcard Rule Cache Management for Software-Defined Networks. IEEE/ACM Transactions on Networking, 2018.
[30] S. Song, H. Park, B. Y. Choi, T. Choi, H. Zhu. Control path management framework for enhancing software-defined network (SDN) reliability. IEEE Transactions on Network and Service Management, 2017, 14(2): 302-316.
[31] H. Xu, Z. Yu, C. Qian, X. Y. Li, Z. Liu, L. Huang. Minimizing Flow Statistics Collection Cost Using Wildcard-Based Requests in SDNs. IEEE/ACM Transactions on Networking, 2017, 25(6): 3587-3601.
[32] Y. Zhang, K. Xu, H. Wang, Q. Li, T. Li, X. Cao. Going fast and fair: Latency optimization for cloud-based service chains. IEEE Network, 2018, 32(2): 138-143.

Fuliang Li received the B.S. degree in computer science from the Northeastern University, Shenyang, China in 2009, and the Ph.D. degree in computer science from the Tsinghua University, Beijing, China in 2015. He is currently an assistant professor at the School of Computer Science and Engineering, Northeastern University, Shenyang, China. He was a postdoctoral fellow with Department of Computing at Hong Kong Polytechnic University, Hong Kong during 2016-2017. He published 20 Journal/conference papers, including journal papers such as IEEE/ACM TON, Computer Networks, Computer Communications, Journal of Network and Computer Applications, and mainstream conferences such as IEEE INFOCOM, IEEE ICDCS, IEEE GLOBECOM, IEEE LCN, IEEE CLOUD, and IFIP/IEEE IM, etc. His research interests include network management and measurement, mobile computing, software defined networking and network security. He is a member of the IEEE.



Jiannong Cao received the BSc degree from Nanjing University, Nanjing, China, in 1982, and the MSc and PhD degrees from Washington State University, Pullman, WA, in 1986 and 1990, all in computer science. He is currently a chair professor of Distributed and Mobile Computing of the Department of Computing and the director of University Research Facility in Big Data Analytics at the Hong Kong Polytechnic University, Hong Kong. His research interests include parallel and distributed computing, wireless networks and mobile computing, big data and cloud computing, pervasive computing, and fault tolerant computing. He has served as chairs and members of organizing and technical committees of many international conferences, including PERCOM, INFOCOM, SMARTCOMP, ICMU, ICPP, MASS, ICPADS, IWQoS, ICDCS, DSN, SRDS, ICNP, and RTSS. He has also served as an associate editor and a member of the editorial boards of many international journals, including IEEE TPDS, TCC, TC, IEEE Network, ACM TOSN, Elsevier Pervasive and Mobile Computing Journal (PMCJ), Springer Peer-to-Peer Networking and Applications, and Wiley Wireless Communications and Mobile Computing. He is a fellow of the IEEE.

Xingwei Wang received the B.S., M.S., and Ph.D. degrees in computer science from the Northeastern University, Shenyang, China in 1989, 1992, and 1998 respectively. He is currently a professor at the School of Computer Science and Engineering and the head of College of Software, Northeastern University, Shenyang, China. He has published more than 100 journal articles, books and book chapters, and refereed conference papers. He has received several best paper awards. His research interests include future Internet, cloud computing, mobile computing and mobile social network.

Yinchu Sun received the B.S. degree in computer science from the Northeastern University, Shenyang, China in 2014. He is currently working toward the M.S. degree in the Northeastern University, Shenyang, China. His research interests include network management and measurement, software defined networking.

Tian Pan received the B.S. degree from Northwestern Polytechnical University, Xi'an, China, in 2009 and the Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 2015. He is currently a postdoctoral fellow researcher in the Department of Information and Communication Engineering, Beijing University of Posts and Telecommunications. His research interests include router architecture, network processor architecture, network power efficiency, and software-defined networking.

Xuefeng Liu received his M.S. and Ph.D. degree from Beijing Institute of Technology, China, and University of Bristol, UK, in 2003 and 2008, respectively. He is currently an associate Professor in Huazhong University of Science and Technology. His research interests include wireless sensor networks and in-network processing.