Implementation and Performance Analysis of STT Tunneling using vNIC Offloading Framework (CVSW)

Ryota Kawashima
Dept. of Computer Science and Engineering
Nagoya Institute of Technology
Nagoya, Japan
e-mail: kawa1983@nitech.ac.jp

Hiroshi Matsuo
Dept. of Computer Science
Nagoya Institute of Technology
Nagoya, Japan
e-mail: matsuo@nitech.ac.jp

Abstract—**Network Virtualization Overlays (NVO3) provides multi-tenancy services in cloud datacenters with existing networking equipment. IP tunneling is an essential technology to logically separate each virtual traffic, in particular, Stateless Transport Tunneling (STT) is considered to achieve better performance using TCP Segmentation Offload (TSO) feature. Currently, there is no openly available implementation of STT, and its implementation and performance characteristics have not been studied in academic field so far. We have therefore implemented STT protocol and conducted performance evaluation by comparing with VXLAN protocol. In practice, the STT implementation has been done using a virtual NIC offloading framework, co-virtual switch (CVSW). CVSW is a software component that extends virtual NICs and provides high-level packet processing framework such as OpenFlow Match-Action. In this paper, we describe implementation details of STT and performance evaluation results from various perspectives. The results showed that the actual performance of STT was almost equal to non-tunneling VM-to-VM communication and was two-times higher than that of VXLAN. Furthermore, we clarify the high-performance nature of STT is brought from both byte-stream characteristic of TCP and Generic Receive Offload (GRO) feature rather than widely believed TSO.**

Index Terms—**SDN, NVO3, STT, VXLAN, OpenFlow, datacenter, NIC offloading**

I. Introduction

The notion of Software-Defined Networking (SDN) has begun to be introduced in cloud datacenters that provide multi-tenancy services. Various network virtualization techniques have been proposed to achieve logically separated tenant networks and practically Edge-overlay or Network Virtualization Overlays (NVO3)[1] approach has gained attentions so far.

Under the overlay-based model, high-functional edge (virtual) switches and L2-in-L3 tunneling protocols play a core role in network virtualization. Specifically end-to-end virtual switches establish IP tunneling and then Ethernet frames of virtual machines' are encapsulated by the tunneling protocol, such as VXLAN[2], NVGRE[3], and STT[4]. These protocols add similar *outer* Ethernet and IP headers to the original frames, however, L4/tunnel headers vary from protocol to protocol and their characteristics can cause measurable performance differences of virtual networks.

Stateless Transport Tunneling (STT) has been proposed by Nicira, Inc. (acquired by VMware, Inc.) for high-performance virtual networks. The unique characteristic of STT is that it uses a *pseudo*-TCP header identified by protocol number (6) at IP layer, which makes STT packets look like TCP packets to physical NICs. As a result, STT packets can be processed with NIC offloading technology such as TCP segmentation offload (TSO)[5]. However, there has been no openly available implementation and rigorous evaluation of STT so far, even though its specification has been released as RFC draft[4].

In this paper, we first explain implementation details of STT[1] and analyze its performance characteristics evaluated from various points of view. For STT implementation, we used our previously proposed virtual NIC offloading framework, Co-Virtual Switch (CVSW)[6], to ease the development of the protocol. CVSW is a high-functional virtual NIC driver (based on *virtio_net* driver of Linux kernel) that provides OpenFlow[7] *Match-Action* packet processing and VXLAN tunneling. The main advantages of CVSW use for STT implementation are as follows; (i) driver-level implementation simplifies the tunnel protocol development (ii) CVSW does not depend on both hypervisor platforms and underlying physical NIC devices. The former reason is derived from avoiding kernel's protocol stack to be modified. The later is achieved by widely supported virtio[8] mechanism because para-virtualized network driver (*virtio_net*) needs not emulate any physical NIC device. Therefore, CVSW is a preferable framework to implement and evaluate STT protocol.

We have conducted performance evaluation of the implemented STT by comparing with two VXLAN implementations (CVSW-based and Open vSwitch[13]-based). In practice, VM-to-VM throughput of TCP/UDP communications with varied packet sizes and various offloading effects have also been eval-

[1]Our implementation can be available at GitHub.
https://github.com/sdnnit/cvsw_net (cvsw-nvo3 branch)

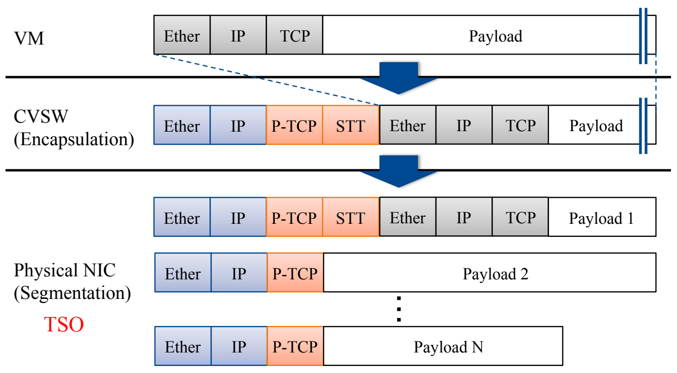

Fig. 1. STT's encapsulation and segmentation flows with TSO feature

uated on 40GbE network. As a result, the actual throughput of STT was about 90% higher at most than MTU-adjusted VXLAN communication. Furthermore, our analysis clarifies that the cause of the STT's high-performance results come from both *byte-stream* characteristics of TCP and Generic Receive Offload (GRO)[9], rather than widely believed TSO.

The rest of this paper is organized as follows. Section II introduces fundamental protocol overviews of STT. Then, brief description of CVSW architecture is explained in Section III and implementation details of STT in CVSW architecture is illustrated in Section IV. In Section V, we evaluate actual performance of the implemented STT by comparing with native and VXLAN-based VM-to-VM communications. Section VI discusses related work, and finally, Section VII concludes this study and gives future work.

II. Stateless Transport Tunneling (STT)

STT[4] is one of the major L2-in-L3 tunneling protocols for network virtualization purpose. The main characteristic of STT is that STT packets are handled as TCP packets by physical NICs, which enables the NICs to apply TCP Segmentation Offload (TSO) feature to large STT packets. Generally, large payload data is split at the TCP protocol layer of the kernel in order to limit single TCP segment size up to Maximum Segment Size (MSS) + 20 bytes. TSO performs this process at the underlying physical NIC instead of the TCP protocol layer to reduce CPU load of the physical server.

Figure 1 shows a sequence of encapsulation and segmentation of a VM's Ethernet frame at the sender side. First, the VM can create large Ethernet frames if TSO feature of the virtual NIC is enabled. Then, a tunnel endpoint system such as virtual switches or CVSW (in this case) encapsulates the frame with STT and *pseudo*-TCP (P-TCP) headers. Finally, the underlying physical NIC divides the large frame into the multiple frames that have consecutive sequence values in P-TCP headers. Note that the counterpart tunnel endpoint system has to reassemble the P-TCP segments before decapsulation because each P-TCP segment except the first one does not include STT and inner headers.

Currently, implementation of STT protocol can only be available in some production systems, such as VMware NSX (Nicira NVP)[10] and Stratosphere SDN Platform (SSP)[11], and thorough evaluation of STT including *how* offloading methods effect its performance or *what* causes the performance differences from other protocols like VXLAN has not been performed yet in academic field. Therefore, we have implemented STT protocol as open source and its performance characteristic is evaluated in this paper.

III. Co-Virtual Switch (CVSW) Framework

In this section, we describe the overview architecture of the Co-Virtual Switch (CVSW) framework[6] that we have developed so far. CVSW is a high-functional software component that resides in para-virtualized network driver (*virtio_net*) of virtual machines, and it performs VM-dedicated packet manipulation based on OpenFlow.

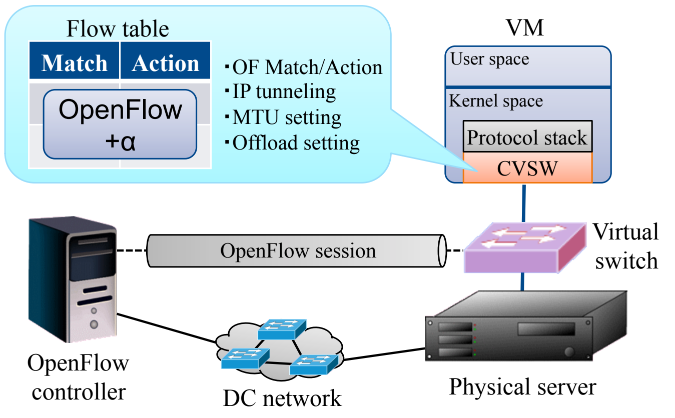

Fig. 2. Overview architecture of CVSW-enabled system

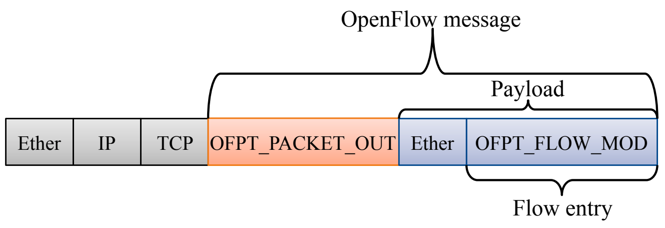

Fig. 3. Packet structure of CVSW message

Figure 2 illustrates the overview architecture of a CVSW-enabled system. First, CVSW works in the kernel space of the virtual machine (VM) as a part of network driver (vNIC). Like existing OpenFlow-enabled switches, CVSW has its own flow table that consists of *Match* and *Action* directives for flow-oriented packet processing. CVSW currently supports all of packet modification actions defined as OpenFLow version 1.0. In addition, VXLAN encapsulation/decapsulation and vNIC configuration (MTU size and NIC offloading) are supposed. The flow table of CVSW can be managed by OpenFlow controllers via the OpenFlow-enabled virtual switch in a way that the controller sends `OFPT_FLOW_MOD` message following an Ethernet header as a payload of `OFPT_PACKET_OUT` message (See figure 3).

CVSW implementation framework is useful for supporting tunneling protocols in that encapsulation/decapsulation process can be implemented independently of TCP/IP protocol stack in both VM and host kernels, and protocol behaviors can be tested on virtual machine environment.

IV. Implementation of STT with CVSW

We describe implementation details of STT protocol as a function of CVSW in this section. Entire CVSW's functionality is implemented in the widely used *virtio_net* para-virtualization driver such that the driver's interface is not changed, and therefore, any modification o f VM's kernel or the underlying hypervisor is not necessary.

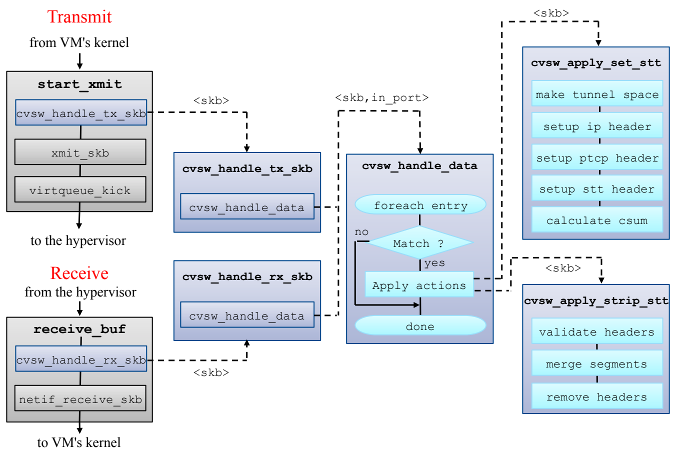

Fig. 4. Flow chart of STT processing on CVSW framework

Figure 4 shows a basic flow chart of STT encapsulation/decapsulation as a part of CVSW's processing. In transmission process, the kernel of the VM calls `start_xmit` function of the driver to pass a `skb` (socket buffer) containing transmitting Ethernet frame. In the case of normal `virtio_net` driver, the `skb` is directly passed down to the underlying hypervisor, however, CVSW-enabled driver performs flow matching in `cvsw_handle_data` function before passing down the `skb`. The flow matching is conducted using `flow_table` entry list as shown below.

```
struct cvsw_flow_entry
{
    /* Priority of this entry */
    __u16 priority;
    /* The number of instructions */
    __u16 nr_insts;
    /* Flow match conditions */
    struct cvsw_match match;
    /* Array of instructions */
    struct cvsw_instruction *instructions;
    /* Pointers to next and previous entries */
    struct list_head list;
};

/* List of cvsw_flow_entry (sorted by priority) */
static LIST_HEAD(flow_table);
```

If matching entry is found, its `instructions` are applied to the `skb` in order. For STT encapsulation, `cvsw_apply_set_stt` function is invoked as an instruction. In this function, buffer space for outer headers (72 bytes) is first ensured. Next, 3-layers of protocol headers (IP, P-TCP, and STT) are inserted into the buffer space. Finally, the checksum filed in the P-TCP header is set with proper value.

In receiving process, `receive_buf` function is called when Ethernet frames arrive, and the corresponding `skb` is passed to `cvsw_handle_data` function. For STT decapsulation, `cvsw_apply_strip_stt` function is called as an instruction. In this function, not only decapsulation of outer headers but also reassembling of P-TCP segments is performed because segmented P-TCP data except the first one does not include inner headers at all (See figure 1). CVSW provides generic reassembling/defragmentation mechanism based on hash table as shown below, and the original STT frame is restored in the `skb` member of `tun_fragment` structure. After the reassembling process, the entire STT frame is decapsulated and the original data is passed to the TCP/IP protocol stack of the VM's kernel.

```
struct tun_fragment
{
    /* P-TCP's identifier */
    __be32 id;
    /* Offset for the next segment */
    off_t next_idx;
    /* Total STT frame size after reassembling */
    size_t frame_size;
    /* Ressembled skb */
    struct sk_buff *skb;
    /* Pointers to next and previous entries*/
    struct hlist_node list;
};

/* Hash list of tun_fragment */
static DEFINE_HASHTABLE(frag_hash, 12);
```

V. Performance Evaluation

We have explained overview architecture and implementation detail of STT with CVSW framework so far. In this section, we evaluate actual performance of the implemented STT protocol in 40GbE environment[12] by comparing with existing VXLAN protocol implemented in Open vSwitch (OVS)[13][2]. In addition, throughput of *optimal* model that does not encapsulate VMs' Ethernet frames was also measured to indicate maximum results in our environment.

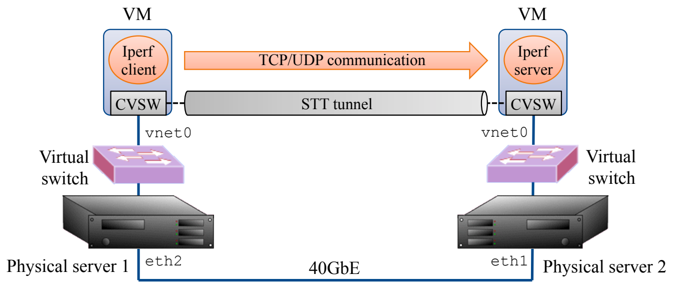

Fig. 5. Experiment environment

[2]MTU size of vNICs were set to 1450 bytes for VXLAN encapsulation.

TABLE I
MACHINE SPECIFICATIONS

VM 1 (Sender)

OS	CentOS 6.5 (2.6.32)
CPU	1 core
Memory	2 GBytes
vNIC	virtio-net
MTU	1420 bytes
Offload	TSO, UFO, GSO, GRO

VM 2 (Receiver)

OS	CentOS 6.5 (2.6.32)
CPU	1 core
Memory	2 GBytes
vNIC	virtio-net
MTU	1420 bytes
Offload	TSO, UFO, GSO, GRO

Physical server 1

OS	CentOS 6.5 (2.6.32)
VMM	KVM
vSwitch	Open vSwitch 2.1.2
CPU	Core i7 (3.60 GHz)
Memory	64 GBytes
MTU	1500 bytes
Offload	TSO, GSO, GRO
Network	40GBASE-SR4

Physical server 2

OS	CentOS 6.5 (2.6.32)
VMM	KVM
vSwitch	Open vSwitch 2.1.2
CPU	Core i7 (3.40 GHz)
Memory	32 GBytes
MTU	1500 bytes
Offload	TSO, GSO, GRO
Network	40GBASE-SR4

Figure 5 shows the experimental network environment and table I gives machine specifications. In the experiments, actual throughput of VM-to-VM running on different physical servers was measured under `performance` mode[3] using Iperf[14] such that the Iperf client continuously sends TCP or UDP to the Iperf server for a minute. Note that the flow tables of both CVSW and OVS were set in advance in the experiments.

A. *TCP communication*

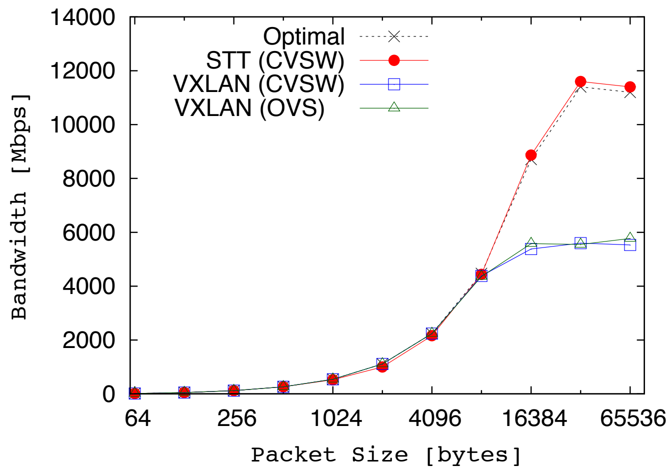

Fig. 6. Throughput of single TCP communication

In this experiments, performance of single TCP connection between end-to-end VMs was evaluated using *Optimal*, *STT*, and *VXLAN* protocols with various packet sizes. The result shows that the throughputs of these three protocols are almost the same for 64–8192 packet sizes. For larger packet sizes, throughputs of *Optimal* and *STT* went up to about 12Gbps at maximum, while that of *VXLAN* peaked under 6Gbps. Considering outer header size of *VXLAN* is smaller than that of *STT*, it is reasonable to consider that this performance gap between *VXLAN* and the others was cased by offloading effects. In addition, the performance gap only occurs for large packets, which indicates the bottleneck of *VXLAN* is packet segmentation or reassembling process. Therefore, we evaluate the actual effects of various offloading functions next.

B. *Offloading effects*

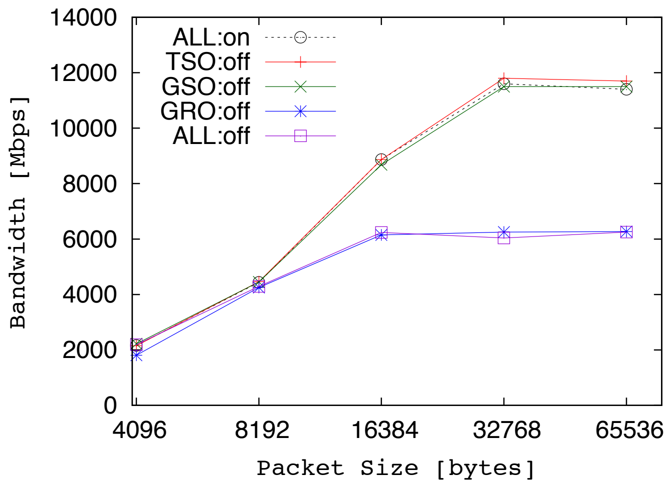

Fig. 7. Effects of offloading types (TSO, GSO, GRO) for STT

We evaluated the performance effects of offloading features, TSO, GSO[15], and GRO, for STT protocol[4]. TSO (TCP Segmentation Offload) is a NIC level function to perform TCP segmentation for large Ethernet frames including TCP/IP headers into acceptable length of ones. On the other hand, GSO (Generic Segmentation Offload) and GRO (Generic Receive Offload) emulate the hardware-level segmentation or reassembling by software at the kernel. GSO supports segmentation for various protocols including TCP and UDP, by contrast, GRO reassembles received packets.

Figure 7 shows the result of offloading effects. First, the dashed line (*ALL:on*) is a result when every offloading feature was enabled and this is the same with the STT's result in figure 6. It is remarkable that TSO and GSO did not effect actual throughput of VM-to-VM communication. This was because the TCP segmentation was not a bottleneck process in the communication under the current computing power. The result indicates that the bottleneck of STT communication is intensive software interruptions (*softirq*) for received packets considering GRO reduces the number of interruptions by reassembling multiple received packets before the interruption. In addition, the throughput of STT without GRO was approximately the same with that of VXLAN shown in the previous figure. We discuss this phenomenon later in this section.

C. *UDP communication*

Finally, we evaluated the performance of UDP communication. The throughputs of every model was almost the same

[3]/sys/devices/system/cpu/cpu*/cpufreq/scaling_governer

[4]We did not evaluate LRO (Large Receive Offload)[16] because it did not work in our environment.

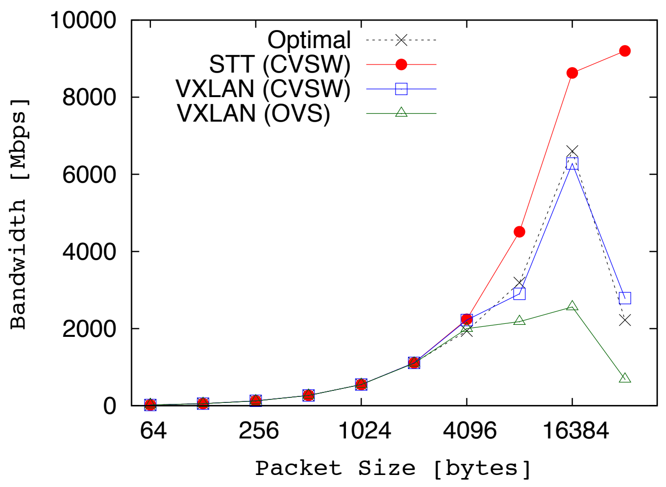

Fig. 8. Throughput of single UDP communication

when packet size was small, however, performance of STT is outstanding for lager packets because GRO feature works for *pseudo*-TCP. The throughput of VXLAN was apparently below *Optimal* for large packets. Considering packet drop rate of *VXLAN* (72%) was higher than that of *Optimal* (27%) when packet size was 16384 bytes, defragmentation process at the physical server before decapsulation could cause buffer overflow of the physical NIC.

D. Discussion

We have evaluated fundamental performance of STT protocol and found the key of its high-speed communication is GRO. Here, we discuss why STT can take advantage of GRO feature and VXLAN cannot, even though GRO can be applied to UDP-based flows.

To show how *outgoing/incoming* packets are processed by GRO, we measured packet length during ten-second VM-to-VM TCP communication at several interfaces, `vnet0`, `eth2` (Tx), and `eth1` (Rx). Figure 9–11 show the number of packets was handled for each packet size. In figure 9, length of most packets was 64K bytes for both protocols because TSO feature of the vNIC was enabled. After GSO processing at the kernel of physical server, lengths of VXLAN packets were changed as shown in figure 10 to fit into MTU size. Large STT packets are also divided by TSO feature of the physical NIC, however, the difference that who divides the packets does not effect VM-to-VM performance according to the above results. At the receiver side, you can see only STT packets were reassembled and GRO feature did not work for VXLAN packets as shown in figure 11. Since GRO can reduce the number of software interruption for received packets, its effect can make a clear difference in actual throughput.

To clear why GRO does not work for VXLAN packets, we analyzed packet structure of VXLAN at the sender machine and the figure 12 illustrates packet structure transition during encapsulation and segmentation processes. We have found that the segmentation process of GSO is applied to the *inner* L4 protocol by `skb_udp_tunnel_segment` function in the kernel to prevent IP fragmentation that is not recommended by the VXLAN's specification[2]. As a result, original large Ethernet frames from the VM are divided into independent

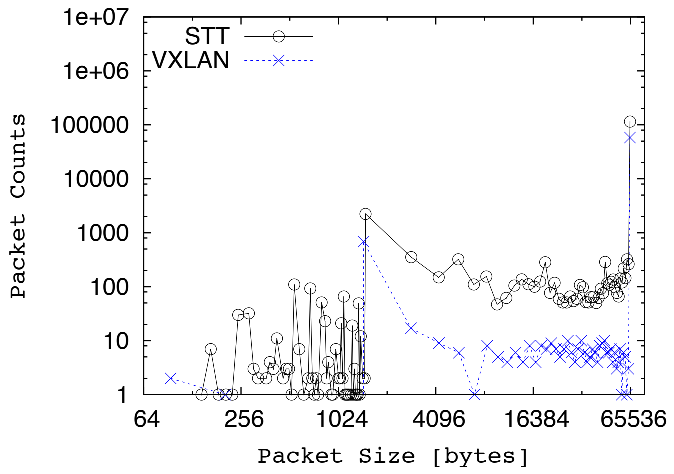

Fig. 9. Tx: Packet size statistic at the `vnet0` (before GSO)

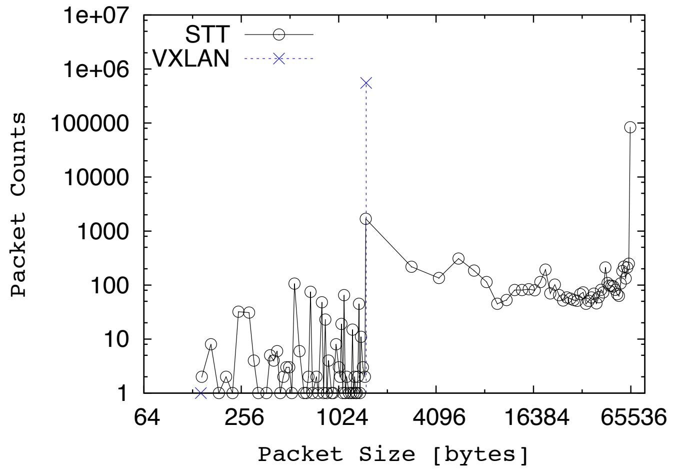

Fig. 10. Tx: Packet size statistic at the `eth2` (after GSO)

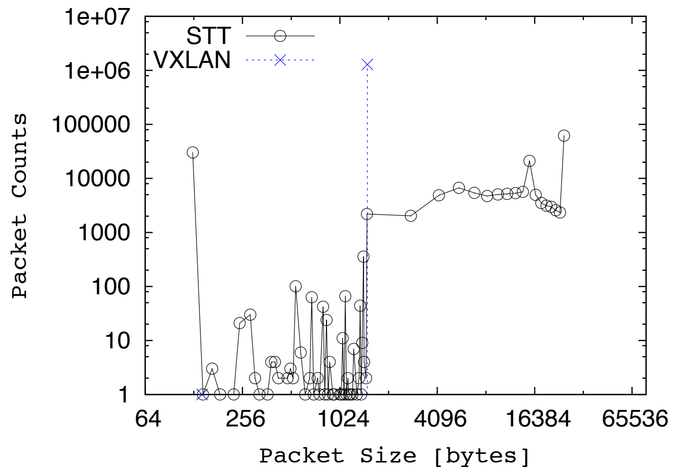

Fig. 11. Rx: Packet size statistic at the `eth1` (after GRO)

multiple frames and outer headers are added to each frame. Considering *message-oriented* characteristic of UDP protocol, there is no need to integrate multiple independent UDP packets by GRO. On the other hand, TCP has *byte-stream* characteristic, which allows reassembling of multiple packets. This is why STT protocol has higher performance than VXLAN.

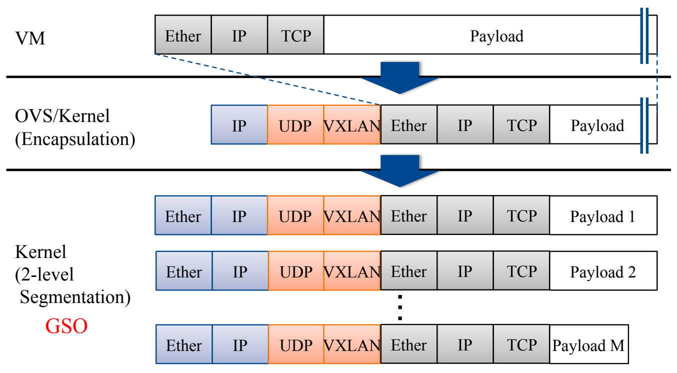

Fig. 12. VXLAN's encapsulation and segmentation flows with GSO

VI. Related Work

VMware has reported the performance of STT implemented in their products[17]. Their result showed that the throughput of STT was comparable to Optimal (9.3Gbps) and they have stated that the use of TSO feature enables high-performance communication. However, analysis of how or which offloading feature brings the improvement of tunnel performance has not written in the report.

Performance of VXLAN has also been reported in [18] provided by VMware. In their report, aggregated throughput of five-sessions per VM running on VMware vSphere was up to about 9Gbps using 10GbE NICs. However, it is unclear about the performance of single session and maximum throughput without the bandwidth limitation.

VII. Conclusion

Edge-overlay based network virtualization has continued to spread in multi-tenant datacenter networks under the SDN paradigm. Various tunneling protocols have been proposed so far, and especially Stateless Transport Tunneling (STT) has been regarded as fastest protocol by taking advantage of TCP Segmentation Offload (TSO) feature. However, actual throughput of STT and the cause of its high-performance have not been analyzed in academic field.

In this paper, we implemented STT protocol using our developed CVSW framework to evaluate. CVSW is a high-level packet processing mechanism that resides in virtual NIC drivers, and it allows developers to concentrate on STT's encapsulation/decapsulation processes. Our evaluation results showed that actual throughput of STT can double compared to VXLAN for large packet size. In addition, we found that the cause of this result was brought from the effect of Generic Receive Offload (GRO) rather than TSO. That is, *byte-stream* characteristics of TCP is a key to reassemble received packets with GRO feature.

STT protocol can cause compatibility problem with packet inspection tools by regarding STT packets as incorrect TCP packets. Therefore, we are planning to explore yet another tunneling protocol based on new connection-less L4 protocol that has *byte-stream* characteristic.

References

[1] M. Lasserre, F. Balus, T. Morin, N. Bitar, and Y. Rekhter, "Framework for DC Network Virtualization", Internet draft, 2013.
[2] M. Mahalingam, D. Dutt, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, "Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks", RFC 7348, 2014.
[3] M. Sridharan, A. Greenberg, N. Venkataramiah, Y. Wang, K. Duda, I. Ganga, G. Lin, M. Pearson, P. Thaler, and C. Tumuluri, "NVGRE: Network Virtualization using Generic Routing Encapsulation", Internet draft, 2014.
[4] B. Davie, Ed. and J. Gross, "A Stateless Transport Tunneling Protocol for Network Virtualization (STT)", Internet draft, 2014.
[5] Offloading the Segmentation of Large TCP Packets, http://msdn.microsoft.com/en-us/library/windows/hardware/ff568840(v=vs.85).aspx
[6] R. Kawashima and H. Matsuo, "Virtual NIC Offloading Approach for Improving Performance of Virtual Networks", The Transactions of Institute of Electronics Information and Communication Engineers B (IEICE), vol.J97-B, no.8, pp.639–647, 2014 (Japanese).
[7] N. McKeown, T. Andershnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turneron, and H. Balakris, "OpenFlow: Enabling Innovation in Campus Networks", Newsletter ACM Computer Communication Review, Vol. 38, Issue 2, pp. 69-74, April 2008.
[8] R. Russel, "virtio: Towards a De-Facto Standard For Virtual I/O Devices", ACM SIGOPS Operating Systems Review, Vol. 42, Issue 5, pp.95–103, 2008.
[9] JLS2009: Generic receive offload, http://lwn.net/Articles/358910/
[10] VMware NSX, http://www.vmware.com/products/nsx/
[11] Stratosphere SDN Platform, http://www.stratosphere.co.jp/
[12] Mellanox, "Performance Tuning Guidelines for Mellanox Network Adapters", http://www.mellanox.com/related-docs/prod_software/Performance_Tuning_Guide_for_Mellanox_Network_Adapters_v1.10.pdf
[13] Open vSwitch, http://openvswitch.org/.
[14] Iperf, http://iperf.sourceforge.net/
[15] gso — The Linux Foundation, http://www.linuxfoundation.org/collaborate/workgroups/networking/gso
[16] Leonid Grossman, "Large Receive Offload implemen- tation in Neterion 10GbE Ethernet driver", Proc. of the Linux Symposium, vol.1, pp.195-200, Ottawa, CA, July 2005.
[17] T. Koponen et. al, "Network Virtualization in Multi-tenant Datacenters", http://download3.vmware.com/software/vmw-tools/technical_reports/network_virt_in_multi_tenant_dc.pdf, VMWare, Technical Report TR-2013-001E, 2013.
[18] VMware, "VXLAN Performance Evaluation on VMware vSphere 5.1", Technical Papers, http://www.vmware.com/files/pdf/techpaper/VMware-vSphere-VXLAN-Perf.pdf, 2013.