Implementation and Analysis of Large Receive Offload in a Virtualized System

Takayuki Hatori and Hitoshi Oi
The University of Aizu, Aizu Wakamatsu, JAPAN
{s1110173,hitoshi}@u-aizu.ac.jp

***Abstract*—System level virtualization has advantages such as easy server consolidation, dynamic reconfiguration, and low power consumption. However, there exists network overhead. In native environment, software level Large Receive Offload (LRO) is effective to improve network receive performance.**

In this paper, we present the performance study of the LRO implemented in a Xen virtualized environment. We implemented LRO in both the physical and the virtual interface and measured the performance in these cases. The receive performance improved 22% with LRO implementation in the virtual interface, whereas no performance improvement appeared with an LRO implemented physical interface. We analyze the performance improvement and describe effectiveness of LRO.

I. Introduction

System level virtualization has advantages such as easy server consolidation, dynamic reconfiguration, and lower power consumption, and is a popular method in building high-performance servers. With the recent hardware support for virtualization like Intel VT and AMD SVM, those advantages have grown by reducing virtualization overhead costs.

These advantages of system virtualization are not free. In particular, we are interested in the overhead of managing concurrent accesses to a network interface. Menon et al. [3] found network receive performance difficult to improve. Their research aimed to improve network performance by utilizing network interface card (NIC) abilities like checksum offloading and TCP segmentation offloading. Willmann et al. [8] suggested Concurrent Direct Network Access (CDNA) architecture. It achieved high network performance on both receive path and transmit path as native environment, however, it required a CDNA aware special NIC. Previous researches have focused on hardware support for virtualization, and more studies must be conducted on software level optimization.

A technology called Large Receive Offload (LRO) [5] was recently proposed. The core idea is to increase network bandwidth at the expense of latency. Gallatin [1] found that LRO was effective to improve receive throughput although it was entirely implemented at the software level.

In this paper, we apply LRO to a Xen para-virtualized environment [7]. In the Xen para-virtualized environment, two network interfaces exist. We implemented LRO in both interfaces and analyzed the performance impact.

The rest of this paper is organized as follows. We begin in Section II with related work. In Sections III and IV, we present descriptions of LRO and Xen as background for our work. In Section V, we describe our LRO implementations onto a virtualized environment. In Section VI, we describe experimental environment. Section VII presents the results and analysis of the experiments. The paper is concluded in Section VIII.

II. Related Work

Willmann et al. [8] suggested CDNA architecture to improve network performance of virtualized environment. It thrust virtualization overhead and complexity resulting from the software level virtualization upon a NIC. Although it achieved 17% higher transmit throughput and 69% higher receive throughput than non-CDNA virtualized system, it required a CDNA aware NIC to assist virtualization.

With respect to network performance improvement in virtualized environment by software level optimization, Menon et al. [3] reported network transmit performance improvement by a factor of 4.4. They also reported 18% improvement in receive performance from non-optimized virtualized system. However, it remained at 61% degradation from native Linux.

These researches indicated difficulty in improving receive performance of virtualized system at the software level.

III. Large Receive Offload

Large receive offload is a technique for increasing in-bound throughput by reducing the CPU utilization. It works by aggregating multiple incoming packets from a single network stream at the Ethernet layer, and delivering them to upper layer as one big packet.

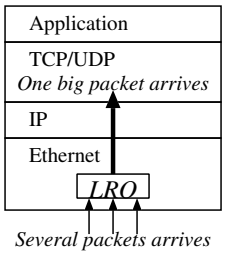

Fig. 1. Network stack with LRO

This technique was first proposed by Grossman [5] who implemented it for a specific NIC. After the introduction of LRO and its performance impact, developers hoped for a generalized LRO. That hope was realized when Themann

produced a generalized patch [4]. It enabled device drivers to use the same LRO implementation with a few modifications.

His patch contains core implementation of packet aggregation. Depending on how device drivers copy a received packet from a NIC onto memory, it offers two different aggregation modes, skb-mode and page-mode. The former is for aggregating multiple socket buffers, each containing one packet. The latter is for aggregating multiple pages, each containing one packet, by creating one socket buffer inside the LRO implementation. Thus, while the page-mode aggregation can reduce the socket buffer consumption compared to skb-mode, it can also waste more memory for smaller packets.

According to benchmarks, even implementing LRO entirely at the software level can improve receive performance [1].

IV. Xen System Level Virtualization

Xen [7] is an open source Virtual Machine Monitor (VMM). It allows multiple operating systems to run on the same machine and to access concurrently to hardware resources. It adopts para-virtualization technology that requires operating systems to be modified.

Xen provides abstracted Virtual Machines (VM), called a domain, for each operating system. Although it is easily manageable for Xen, an abstracted VM is different from underlying physical resources.

There are two kinds of domains called domain0 and domainU in the Xen para-virtualized environment, privileged and unprivileged respectively. The domain0 can access physical hardware, whereas the domainU cannot. To access physical hardware from the domainU, an operating system must access an abstracted VM. The domain0 detects the access, and emulates what the domainU wants to do. In this paper, we use a term driver domain to represent a privileged domain0, and guest domain to represent an unprivileged domainU.

In Xen para-virtualized environment, a physical interrupt is first handled by a Xen interrupt handler, and then delivered to a target VM through an event channel provided by Xen. Xen also provides grant tables for safe memory sharing between VMs. It enables a VM to grant a page with permission to another VM.

To manage safe access to network interfaces and block devices, Xen provides IO channels composed of split device drivers, backend and frontend drivers. The former is used in a driver domain and the latter is used in a guest domain. The split device drivers share a single memory ring to transfer data through the ring. They use the event channel to notify the other side, and the grant table to transfer data through the ring. This ring consists of a fixed number of buffers, and it provides fair share of hardware capacity between VMs.

A. Virtual Network

A simplified Xen virtualized network is represented in Fig. 2. Xen adopts bridge networking to multiplex packets to each VM. A physical interface is connected to a bridge, and the bridge is connected to virtual interfaces. There exists some couples of virtual network interfaces, backend interfaces in a driver domain and frontend interfaces in a guest domain. A packet is exchanged between the couple of virtual interfaces by the split device drivers.

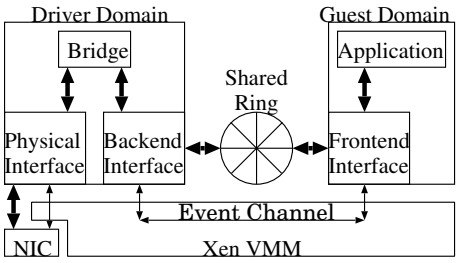

Fig. 2. Virtualized network

In a network IO channel, Xen offers two types of data transfer modes between domains, transferring data by copying and by page swapping. The former copies a packet onto a page granted by a guest domain. The latter exchanges a page containing a packet with a free page granted by a guest domain. The data copying method is useful for a small data transfer because it costs per byte, whereas the page swapping method is useful for a large data transfer because it costs per transfer. In this research, we used copying method because it was widely used with Xen version 3.1.0 we employed.

V. LRO Implementation

In this section, we describe how we implemented LRO to network interfaces. For both implementation on a physical interface and a virtual interface, we used Themann's LRO patch [4]. It provides two methods to aggregate packets (skb-mode and page-mode.) Both of our target interfaces manage a packet as a socket buffer, and thus we implemented skb-mode LRO on both interfaces.

To utilize the patch, it was required to implement some initiation operations, and a function that seeks headers from a received packet. To monitor how much LRO can aggregate packets in average, we also implemented some functions to make the information available from a Linux command ethtool. In addition, depending on where to implement LRO, some difficulties occurred. We will explain these difficulties, and present how we avoided them in the following subsections.

A. Physical Interface

Although this research targets the receive network performance, we had a trouble with a transmit network path. The bridge networking uses a receive path to obtain a packet from an interface, and a transmit path to deliver the packet to an interface. It forces a received packet at a physical interface to pass through both of the receive path and the transmit path.

As mentioned above, we employed skb-mode LRO implementation. This made everything worse. The Linux receive path was capable of aggregated packets by LRO, however, the transmit path was not. Furthermore, Xen's split device drivers for a network IO channel could not transmit skb-mode aggregated packets. In order to carry skb-mode aggregated

packets at the physical interface, we modified the transmit path to reconstruct one big packet by linearizing the aggregated packets. It meant one extra data copy became required for all LRO aggregated packets.

B. Virtual Interface

LRO implementation on a virtual network interface was not complicated. We implemented LRO in a frontend virtual interface used in a guest domain. Thus, it aggregated packets after passing through a bridge and a network IO channel. It meant that we could aggregate after sophisticated operations. However, one technique was required.

It was necessary to isolate loop-backed packets transmitted from virtualized sub-network, such as packets from a driver domain. Those packets had a different structure from packets received at a physical network interface. They maintained data and headers separately, and required two page transfers per packet at the network IO channel. After transferring such a page-divided packet, a backend driver forced to reconstruct a socket buffer to combine a header part with a data part.

The problem was that the reconstructed packet was not linear. It was obliged to treat the packet as a fragmented packet like an over Maximum Transmission Unit (MTU) size packet. The Linux network stack could not recognize skb-mode aggregated socket buffers that contained fragmented packets. In this research, we required loop backed packets for an experimental script. Not to include those packets in the experimental data stream, we created two couples of virtual interfaces, and divided the network streams.

VI. Experimental Environment

We implemented LRO described in Section V on Xen version 3.1.0 para-virtualized system with Linux version 2.6.18 for both a driver domain and a guest domain.

For the receiver machine, we used Dell PowerEdge SC440 with 1.86 GHz Intel Xeon. This machine has one RealTek gigabit PCI NIC and 2 GB memory. A non-virtualized sender machine has 2 GHz AMD Athlon64 X2 with 2 GB memory, and an on-board gigabit NIC.

We wrote simple sender and receiver programs and executed on each machine. These programs communicate over the Ethernet connected by a Logitec gigabit switching hub. We measured the time to transfer 10GB data with the MTU size of 1500 bytes through a TCP session.

Config.	System	LRO
Native	Native	No
Guest-1	Virtualized	No
Guest-2	Virtualized	Physical Interface
Guest-3	Virtualized	Virtual Interface

TABLE I
Experimental configurations

We examined four configurations presented in TABLE I for the receiver machine. Differences were in whether the system was virtualized or not, and in LRO implementation. The receiver processes ran in guest domains of Guest-1, Guest-2, and Guest-3 configurations. It meant a packet passed through a network bridge in the driver domain and then arrived at the guest domain. For Guest-1, Guest-2, Guest-3 configurations, we assigned the driver domain and the guest domain with one virtual CPU each, which guaranteed fair sharing of CPU, and 1 GB and 512 MB memory respectively. For a Native configuration, we assigned full physical resources.

As a profiling tool, we employed Xenoprof [2], and tcpdump for network traffic monitoring. Xenoprof is a profiling tool based on OProfile [6] targeting a Xen virtualized environment. It collects hardware events such as clock cycles and retired instructions. Packet monitoring by tcpdump operated in the sender machine not to make any performance degradation on the receiver machine. We also used Linux commands ethtool and ifconfig to obtain LRO specific information and total network traffic information.

VII. Results

In this section, we present results of our experiments first, and then analyze performance improvement and effectiveness of LRO. All the results were averages of five executions.

Config.	Throughput	LRO Rate
Native	557.0	1.00
Guest-1	505.4	1.00
Guest-2	508.4	1.99
Guest-3	615.3	4.92

TABLE II
Throughput (Mbps) and LRO rate (packets)

TABLE II describes receiver throughput in Mbps and LRO rate, the average number of packets aggregated in one group by LRO, achieved by four configurations defined in Section VI. The non-LRO virtualized environment, Guest-1, achieved throughput of 505 Mbps, which was 91% of the native Linux throughput, 557 Mbps. Guest-2, which had a LRO implemented physical interface, achieved almost the same throughput as Guest-1 with twice the LRO rate. The configuration with a LRO implemented virtual interface, Guest-3, improved upon the non-LRO virtualized environment by 22%, and upon native Linux by 10%, and achieved throughput of 615 Mbps. It was significant that Guest-3 achieved higher throughput than native Linux.

Regarding LRO rate, in both Guest-2 and Guest-3 configurations, LRO could not aggregate efficiently multiple packets, whereas LRO implementation permitted the aggregation of 64 packets. It is considered that the CPU did not wait further packet reception as the CPU was not busy and could handle interrupts for packet reception immediately. Compared to Guest-2, Guest-3 achieved a higher LRO rate. It might be due to the high throughput or a configuration difference, where to aggregate packets, or both.

A. CPU Utilization

In this subsection, we present profiling result collected by Xenoprof, in virtualized environments, and by OProfile, in

a native environment. We profile clock cycles and retired instructions, and estimate CPU utilization. These events were sampled at every 10000 occurrence during the execution of 10GB transfer.

Config.	Clock	Instr.
Native	6.27	6.30
Guest-1	9.71	7.16
Guest-2	8.01	5.36
Guest-3	9.87	7.49

TABLE III
CLOCK CYCLES AND RETIRED INSTRUCTIONS ($\times 10^6$)

TABLE III presents clock cycles and retired instructions consumed across the system. In this table, low consumption of clock cycles and retired instructions of the Guest-2 configuration was distinguished. In particular, Guest-2 consumed fewer retired instructions than native Linux. Besides, compared to non-LRO Guest-1, that achieved nearly the same throughput as Guest-2, consumption of clock cycles and retired instructions decreased 18% and 25% respectively.

Config.	Domain	Clock	Instr.
Guest-1	driver	2.32	1.50
	guest	7.39	5.67
Guest-2	driver	2.09	1.33
	guest	5.92	4.03
Guest-3	driver	3.33	2.27
	guest	6.54	5.23

TABLE IV
BREAKDOWN OF CLOCK CYCLES AND RETIRED INSTRUCTIONS ($\times 10^6$)

TABLE IV presents per VM consumption of clock cycles and retired instructions. Although Guest-2 required extra data copy in a driver domain, required for aggregated packets to pass through a bridge, the driver domain consumed fewer clock cycles and fewer instructions than those of a Guest-1 driver domain. It indicated that LRO implementation in a physical interface could improve CPU utilization of individual VMs, thus, improve across the system. With an LRO aware network bridge and split device drivers, which can manage LRO aggregated packets without any overhead, this improvement in CPU utilization will increase.

In the Guest-3 configuration, as seen in TABLE IV, a guest domain's clock cycles and instructions reduced from the Guest-1 configuration. However, a driver domain consumed more clock cycles and more instructions than Guest-1, and thus, consumption of clock cycles and instructions increased across the system as presented in TABLE III.

Despite the use of the same driver domain as the Guest-1 configuration, we observed that the CPU utilization was lowered in the Guest-3. It is possibly the result of 18% higher throughput than Guest-1, but further investigation is needed to identify the source of this performance degradation.

B. Network Traffic

In this subsection, we present network traffic profiling observed by tcpdump and Linux commands. In this research, we examined a sender-receiver networking model through a TCP session. Traffic from the sender to the receiver was a data packet, and traffic from the receiver to the sender was acknowledgment.

For each received packet, an acknowledgment packet with the identification of the received packet, such as the data segment number is returned to the sender. In order to reduce the acknowledgment traffic, there is a strategy called delayed acknowledgment to aggregate acknowledgments. It permits to aggregate two acknowledgments for packets received within 500 milliseconds. Fig. 3 describes how the delayed acknowledgment works.

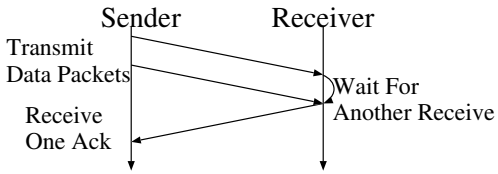

Fig. 3. Delayed acknowledgment

Config.	Phys. Rx.	Phys. Tx.	TCP Rx.
Guest-1	7.42	3.86	7.42
Guest-2	7.42	2.77	3.74
Guest-3	7.43	1.49	1.51

TABLE V
NETWORK TRAFFIC ($\times 10^6$PACKETS)

TABLE V presents total network traffic information observed at the receiver machine. Phys Rx and Phys Tx represents the numbers of packets received at and transmitted from the receiver machine respectively. TCP Rx represents received packets at the TCP layer in a guest domain. We calculate TCP Rx by dividing Phys Rx by LRO rate, the average number of aggregated packets in one group by LRO, presented in TABLE II. Because the TCP layer recognizes aggregated packets by LRO as one big packet, there exists a difference between Phys Rx and TCP Rx.

By comparing transmitted packets from the receiver machine and received packets at the TCP layer, it was clear that delayed acknowledgment did not work well in the Guest-3 configuration. It could aggregate only 1.03 acknowledgments, whereas it could aggregate 1.93 and 1.41 acknowledgments in Guest-1 and Guest-2 configurations, respectively.

The inefficient delayed acknowledgment was a characteristic of LRO implementation in the guest domain. In order to aggregate acknowledgments, it was required to deliver several packets to the TCP layer in a short delay. It was also required that these packets had data segment numbers in serial order. However, when short delayed and serial ordered packets were transferred from a driver domain together at once, they were

aggregated by LRO, which was implemented in the Ethernet layer. In consequence, one big packet arrived at the TCP layer in the guest domain, and it became difficult to receive another packet in time.

Config.	Ack.	Sack.	Sack. Rate(%)
Guest-1	3.54	0.502	14.191
Guest-2	2.47	0.354	14.341
Guest-3	1.26	0.120	9.514

TABLE VI
ACKNOWLEDGMENT TRAFFIC ($\times 10^6$ PACKETS)

TABLE VI describes acknowledgment traffic captured by tcpdump in the sender machine. Ack and Sack are abbreviations of acknowledgments and selective acknowledgments. A selective acknowledgment is an alternative of acknowledgments that informs disordered or lost packets. It forces the sender TCP layer to retransmit the desired packet as soon as possible. Sack rate represents the ratio of Sack packets against the Ack packets. Due to insufficient buffer space for tcpdump, several packets were dropped from capturing. It appeared as the difference between acknowledgments observed in the sender machine and transmitted packets counted in the receiver machine presented in TABLE V.

According to Linux implementation, the selective acknowledgment rate is used to estimate available network bandwidth in the sender side TCP layer for congestion control. Fig. 4 describes a model of congestion control strategy in Linux. As an acknowledgment receives, the estimated bandwidth will grow. Once a selective acknowledgment arrives, it decreases to a slow start thresh, calculated by using maximum estimated bandwidth, until another acknowledgment arrives.

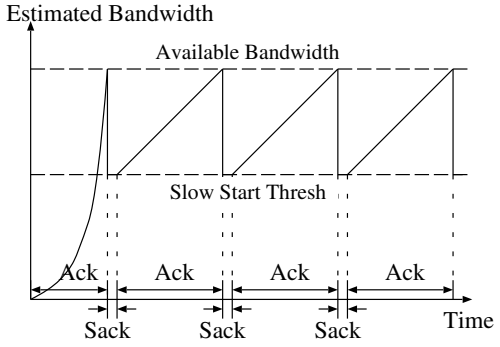

Fig. 4. Model of congestion control strategy

Of note was a low selective acknowledgment rate of the Guest-3 configuration, whereas those of Guest-1 and Guest-2 configurations were equally higher. This low Sack rate led to a high estimated bandwidth on the sender side which in turn resulted in more packets transfer at once. TABLE VI indicated the 40% lower selective acknowledgment rate than in Guest-1 and Guest-2 configurations resulting in the 18% higher throughput in the Guest-3 configuration.

According to RFC 2582 [9] and Linux implementation, a selective acknowledgment is not aggregated by the delayed acknowledgment mentioned before. The unsuccessful delayed acknowledgment increased only acknowledgments. It resulted in depressing the selective acknowledgment rate in the Guest-3 configuration.

Based on the results of the Guest-3 configuration, we can reduce 500 milliseconds delay for every group of aggregated packets at the expense of 2% increase in acknowledgment by preventing the unsuccessful delayed acknowledgment. However, current Linux implementation does not allow us to restrain the delayed acknowledgment and incorporating this option into the TCP layer of the Linux is another topic of our future work.

VIII. CONCLUSION

We implemented LRO on two different interfaces in the virtualized system. We found that LRO in a physical interface reduced the numbers of clock cycles and instructions across the system while achieving the same throughput as the original system. However, we also found that packet aggregation by LRO did not work as we expected. It indicated that LRO in the physical interface would achieve higher throughput in network bound environments.

We found that LRO in the virtual interface achieved a higher throughput than the original virtualized system. We consider that this was the result of fewer acknowledgment aggregations due to the combination of the LRO and the delayed acknowledgment on the network stack.

We identified the overhead in LRO (e.g. extra data copy in the bridge) and the interface problem with the Linux implementation (e.g. the delayed acknowledgment). These are the issues that need be solved to take full advantage of the LRO in the virtualized system.

ACKNOWLEDGMENT

This work is supported in part by grant from the University of Aizu Competitive Research Funding.

REFERENCES

[1] A. Gallatin, “Re: [RFC 0/1] lro: Generic Large Receive Offload for TCP traffic,” July 2007, http://lkml.org/lkml/2007/7/25/313.
[2] A. Menon et al., “Diagnosing Performance Overheads in the Xen Virtual Machine Environment,” *VEE ’05: Proc. 1st ACM/USENIX Int. Conf.*, pp.13–23, 2005.
[3] A. Menon, A. L. Cox and W. Zwaenepoel, “Optimizing Network Virtualization in Xen,” *USENIX-ATC ’06: Proc. Annu. Technical Conf.*, pp.15–28, 2006.
[4] J.-B. Themann, “[RFC 0/1] lro: Generic Large Receive Offload for TCP traffic,” July 2007, http://lkml.org/lkml/2007/7/20/250.
[5] L. Grossman, “Large Receive Offload implementation in Neterion 10GbE Ethernet driver,” *Proc. Linux Symp.*, vol. One, pp.195-200, July 2005.
[6] OProfile, http://oprofile.sourceforge.net/news/.
[7] P. Barham et al., “Xen and the Art of Virtualization,” *SOSP ’03: Proc. 19th ACM Symp.*, pp.164–177, 2003.
[8] P. Willmann et al., “Concurrent Direct Network Access for Virtual Machine Monitors,” *HPCA ’07: Proc. IEEE 13th Int. Symp.*, pp.306–317, 2007.
[9] S. Floyd and T. Henderson, “The NewReno Modification to TCP’s Fast Recovery Algorithm,” 1999, RFC2582.