IEICE TRANS. INF. & SYST., VOL.E99–D, NO.2 FEBRUARY 2016

PAPER

vCanal: Paravirtual Socket Library towards Fast Networking in Virtualized Environment

Dongwoo LEE[†], **Changwoo MIN**[†], ***Nonmembers***, ***and*** **Young Ik EOM**[†a)], ***Member***

SUMMARY Virtualization is no longer an emerging research area since the virtual processor and memory operate as efficiently as the physical ones. However, I/O performance is still restricted by the virtualization overhead caused by the costly and complex I/O virtualization mechanism, in particular by massive *exits* occurring on the guest-host switch and redundant processing of the I/O stacks at both guest and host. A para-virtual device driver may reduce the number of *exits* to the hypervisor, whereas the network stacks in the guest OS are still duplicated. Previous work proposed a socket-outsourcing technique that bypasses the redundant guest network stack by delivering the network request directly to the host. However, even by bypassing the redundant network paths in the guest OS, the obtained performance was still below 60% of the native device, since notifications of completion still depended on the hypervisor. In this paper, we propose vCanal, a novel network virtualization framework, to improve the performance of network access in the virtual machine toward that of the native machine. Implementation of vCanal reached 96% of the native TCP throughput, increasing the UDP latency by only 4% compared to the native latency.

key words: *device virtualization, multicore processor, para-virtual library, concurrent queue, polling-based notification*

1. Introduction

Recently, virtualization has become a mature technology where virtual machines (VM) have achieved nearly bare-metal performance through several improvements in software and hardware technologies. Leveraging these improvements, many data centers and enterprises, as well as individuals, have widely applied virtualization to their workloads. Nevertheless, the use of virtualized systems is limited for workloads requiring high I/O performance, since they suffer from unacceptable overhead mostly induced by the virtualization layer; typically, the I/O path in a virtualization layer is implemented in a much more costly and complex way than the native I/O path, such as through a *trap-and-emulate* mechanism. Para-virtualization [1]–[3] reduces the emulation overhead by exploiting awareness of the virtualization environment, but frequent context-switches still remain, such as *exit*, [4], for passing the I/O request to the hypervisor. Therefore, the virtualization layer limits the I/O performance of VMs more significantly with high-performance devices. For example, state-of-the-art network interface cards (NICs) reach throughput over 40 Gbps, but `virtio` performs below 1 Gbps.

Manuscript received June 10, 2015.
Manuscript revised October 11, 2015.
Manuscript publicized November 11, 2015.

[†]The authors are with the School of Information and Communication Engineering, Sungkyunkwan University, Suwon, Korea.
a) E-mail: yieom@ece.skku.ac.kr

DOI: 10.1587/transinf.2015EDP7224

Though modern devices supporting *direct assignment* enable direct issuing of I/O requests from guests to the corresponding devices and thus curtail most of the virtualization overhead, there are a few limitations. Since they require additional hardware extensions [5], [6], the hardware cost would be higher and the legacy hardware cannot benefit from it. Moreover, since VMs directly access the physical NICs without support of the hypervisor, it becomes difficult to manage the network configuration of each VM by QoS policy and to migrate the VM for load balancing. In many virtualization applications, the ease of management is also important, thus enhancement to the performance of software-based I/O virtualization is still required.

Many previous studies [5], [7]–[11] found that the overhead of software-based I/O virtualization is mainly caused by *exits* for communication between the guest and the host. To make matters worse, the I/O stack, e.g., TCP/IP layers in the kernel, is duplicated at both the guest and the host; the I/O stack in the guest is obviously unnecessary because the guest has virtual devices only. Though ELVIS [11] achieved an exitless virtual I/O with the para-virtual device driver, the duplicated network stack harms the performance of the virtual network. This redundancy is caused by the absence of awareness on the virtualized environment in the guest application.

In this paper, we determine *exits* and the redundant network stack as major sources of virtual network overhead and introduce a novel network virtualization system, vCanal, which improves network performance by providing a virtualization-aware socket library. vCanal can bypass the redundant network stack in the guest, and enable the guest-host communication without *exits* by exploiting multicore architecture. It is a software-only approach that requires no hardware support. This paper makes the following specific contributions:

- We introduce a *para-virtual socket library* which improves the performance of virtual networks by reducing virtualization latency. Many previous studies tried to reduce the overhead of a para-virtualized device driver in several ways: by enhancing the back-end driver [12], by applying lock-free mechanisms [13], or by delivering data with zero-copy processes [14]. In contrast, we reduced the network virtualization overhead by revisiting the communication mechanism between the guest and host network services.
- vCanal does not require modification of the guest application or kernel. Since the socket library is aware

of the virtualization environment, vCanal can easily be applied to the existing system by simply replacing the existing socket library. In addition, vCanal is an orthogonal mechanism to the para-virtualized device, thus it can be selectively applied for network intensive workloads; if the workload requires computing power rather than the network throughput, the typical event-driven network process can be used.

- With vCanal, the overhead of network access in the VM can mostly be removed by eliminating *exits* and bypassing the network stack in the guest. To make the design practical, we developed two techniques: *communication channel* and *hybrid address space*. The communication channel is a concurrent queue structure which enables the network requests of the guest to be forwarded to the host with very low overhead, while hybrid address space enables the host network service to directly access buffers or data structures in the guest without the costly address translations and needless copies.
- In order to validate performance improvements achieved by the vCanal system, we ran popular network benchmarks and compared the results with those of traditional para-virtualized I/O framework. We got nearly bare-metal throughput and latency; the vCanal improved TCP throughput up to 169%, and decreased UDP latency up to 38% compared with `virtio` on the 10Gbps NIC.

The rest of this paper is organized as follows. Section 2 provides a description on the three conventional network virtualization technologies along with their limitations. Section 3 then elaborates on the design of the vCanal system. Section 4 validates our design by presenting experimental results, including throughput improvements, latency reductions, and the impact of each design issue. Limitations of vCanal is discussed in Sect. 5 and some related work is introduced in Sect. 6. Finally, we conclude our paper in Sect. 7.

2. Network Virtualization Backgrounds

Network virtualization is conventionally implemented by one of the following three methods: emulation, para-virtualization, and direct assignment. In this section, we give an overview of them to help understand the design of the vCanal system.

Emulation: Device emulation is the most traditional and straightforward method by which the hypervisor emulates common NICs, such as Realtek RTL8139 and Intel PRO/1000 (e1000). An unmodified guest OS accesses the network in the same way that native OS accesses real NICs. The hypervisor catches this request and emulates the behaviors of the corresponding device. Although there is no need to modify the guest OS or to install any special device drivers when using the emulation method, the cost is significantly high since the hypervisor should process each and every hardware command through software emulation. Therefore, even though the host machine is equipped with NICs of sufficiently high performance, the emulation process becomes a bottleneck which decreases the performance of network access in virtualized environment

Para-virtualization: Many hypervisors currently mitigate the overhead from device emulation by providing awareness of the virtualization environment to the device driver. Instead of emulating the popular devices, the para-virtualized device driver [1]–[3] communicates with the back-end driver in the emulation process (e.g., QEMU [15] in KVM) or the separated service domain (e.g., Dom0 in Xen) in order to deliver the network request to the host. The back-end driver can process those requests with better performance than emulation, since it reduces the number of interventions with the hypervisor by coalescing network requests. Nevertheless, network access in virtualized environment is still slower than that in the native environment since *exits* are still generated during the notification and the completion. Many previous studies [5], [7]–[10] show the context switch to be a major source of device virtualization overhead. In addition, since the virtual machine only accesses para-virtualized device drivers to deliver the network request, I/O stacks such as the TCP/IP layer in the guest OS are obviously unnecessary and only increase the virtualization overhead; network packets are handled twice on both the guest and the host because the host should rebuild or modify the packets with the configuration of the physical network, whereas the guest OS build packets with the configuration of virtualized network.

Direct assignment: Recently, state-of-the-art NICs have virtualization features such as PCI pass-through [16], [17] or SR-IOV [5]. By support of the hardware, the guest OS can directly access the physical device, significantly improving network performance of the VM. But, despite their advantages, they still encounter some problems: (1) the legacy NICs should be replaced with new NICs which provide those hardware features, thus the cost becomes relatively high; (2) since the VM bypasses the hypervisor while it accesses the physical hardware, the hypervisor cannot provide any abstraction for the underlying hardware, and so, the device I/O from the guest OS may be out of the hypervisor's control. For example, it may be hard to apply the network QoS to the VM; (3) absence of hardware abstraction also makes VM migration difficult, since the destination machine should be equipped and configured identically to the source machine. Although these NICs can provide high performance networks for the VMs, migrating VM or applying device QoS in cloud service might be more important for service management.

3. vCanal Design

The design of vCanal aims to provide fast networking in virtualized environment towards bare-metal performance and scalability through a software-only approach exploiting multicore architecture, instead of using special hardware or modifying the guest OS. In this section, the rational behind the design decisions and key techniques of vCanal system will be described in detail.

3.1 System Overview

Our novel network virtualization framework, vCanal, illustrated in Fig. 1, mainly consists of three components: libCanal, vCanal service, and communication channel. These components are designed to reduce the virtualization overhead induced by frequent context-switches and redundant network processing. Instead of incurring costly context-switches by traps to the hypervisor, guest applications directly issue network requests to the host using libCanal. Issuing a network request needs only to enqueue the request message into the communication channel. The communication channel is the per-VM concurrent queue shared between the guest and the host. vCanal service schedules the polling thread to decide which channel should be serviced and how many requests should be processed. vCanal service checks whether new requests are in the communication channel by the polling thread, and passes them to one of the network threads for further processing. After the request is processed, notification is also delivered directly to the guest. Therefore, the virtualization latency is significantly decreased since there are no costly context-switches to the hypervisor during network processing. Further details of the design of each component are presented in the rest of this section.

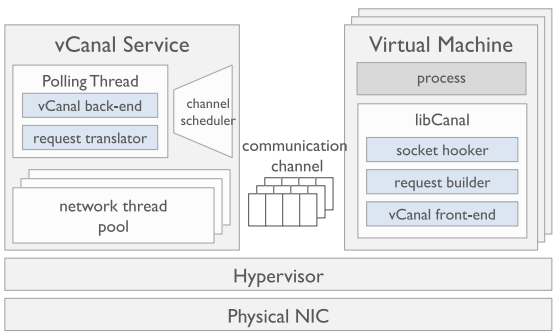

Fig. 1 The architecture and main components of the vCanal system.

3.2 Paravirtual Socket Library

In native systems, when an application wants to access the network to transfer data, it ordinarily uses network socket interfaces. Because the socket library is the wrapper of the corresponding system calls, the actual network processing is performed in the kernel. The TCP/IP layer in the kernel builds packets and passes them to the NIC for physical transmission. On the other hand, in virtualization environment, the guest application sends the network requests to the paravirtualized network driver as does the native application, as shown in the top of Fig. 2. However, the paravirtualized network driver delegates the network processing to a separated domain or emulation process, instead of delivering it directly to the physical NIC. This incurs the costly context-switches since the paravirtualized driver triggers an *exit* or uses hypercall in order to inform the host of the request. If the application heavily accesses the network, this kind of context-switch occurs very frequently, causing significant degradation of the network performance.

To address the issues on the overhead of the context-switches, the socket library is the best place for eliminating redundant network processing in the guest kernel, since it is the first layer where the network request of application passes through. The `libCanal`, which is the guest-side socket library in our architecture, has awareness of virtualization by defining the new socket family, `AF_VSOCKET`, to the existing socket library. Therefore, by indicating the socket family at initialization, vCanal can easily be applied to the existing systems without modifying the guest applications. With this socket, the guest application directly delivers its network requests to the host as illustrated at the bottom of Fig. 2, by simply queueing the request to the communication channel. After the I/O request is processed in the host, the completion notification is given to the guest by just storing the return value to the result field in the INR, whereas the paravirtualized driver requires the virtual interrupt which incurs *exits*. Moreover, since all physical net-

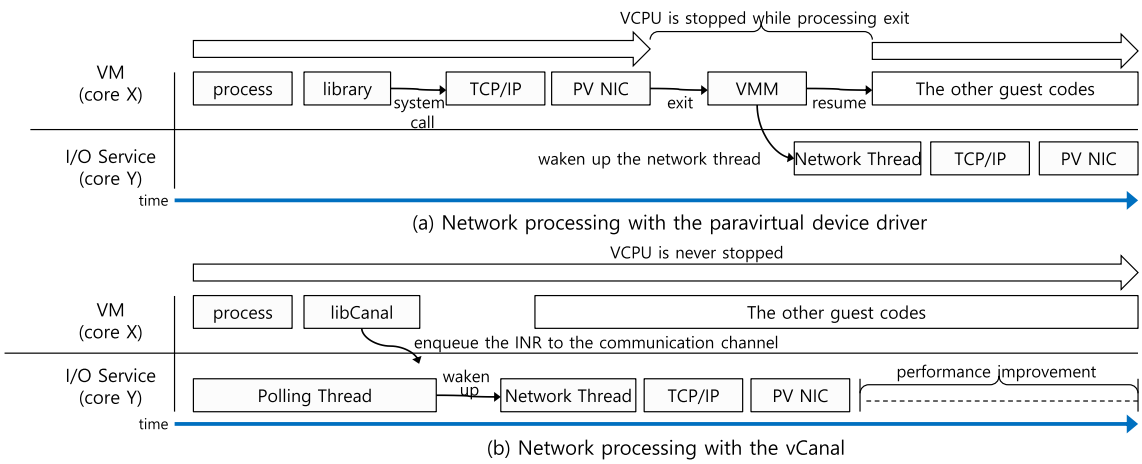

Fig. 2 Comparison of the paravirtualized device driver (a) with the network process of vCanal (b).

work requests are issued in the vCanal service that is pinned on a certain core, we can assign interrupt affinity to that core in order to eliminate *exits* triggered by external interrupts. So, there are no *exits* during the completion notification and thus vCanal can completely eliminate the overheads induced by *exits*.

Though libCanal enables the guest application to directly access the host network service, intervention of the guest kernel is still required for system initialization and guest-host interface. For these purposes, a special device driver termed the *helper driver* was implemented. In the case of initialization, the helper driver allocates a few memory pages that is needed to establish the communication channel during VM boot up, and notifies it to the vCanal service. This notification can be done by an *exit*, in the case of KVM, and it can also be implemented by hypercall within Xen. The main reason for using the guest memory space for the communication channel is to protect the system against an untrusted VM. It is important especially for the environment that VM isolation should be retained. If the communication channel is established in the host, it may become a vulnerability, allowing the guest to access the host address space outside of its boundary. After establishing the communication channels the helper driver is used for interfacing between libCanal and the hypervisor, in the two cases: guest polling control and guest-host address translation. More details about the helper driver for these two operations will be presented in Sect. 3.4 and Sect. 3.5, respectively.

3.3 Intermediate Network Request

In vCanal, the file descriptor of the socket established in the host is delivered to the guest when the guest requests to create the socket with vCanal socket since the network requests issued by the guest application should be actually processed in the host service thread. In order to provide the way for the guest and host to communicate without any conflicts, we introduce the concept of intermediate network request (INR). The INR is a kind of message representing the network request in a socket-independent structure. The INR includes information on the socket function type, function parameters, return value (originally uninitialized and filled after I/O completion), and flag for the request state. When the guest application issues a function call for controlling a TCP connection, libCanal intercepts it and builds the INR with the arguments of the corresponding function call. For example, if `socket()` is issued, the type of the INR is set to SOCKET and the parameter fields are filled with the same values of the parameters of the original function call. Afterward, libCanal sets the flag of the INR to `INR_REQUESTED` in order to mark that the request is not yet completed, and passes the INR to the host. In the host, vCanal service checks the type of the INR, and actually issues the `socket()` function with the parameters retrieved from the INR. After the socket is established, the return value field of the INR is set to the file descriptor of the established socket, and the flag is changed into `INR_COMPLETED`. Even the request is processed in the host and the guest can retrive the result from the return value field in the INR, the guest cannot directly use the file descriptor obtained from the host since the same descriptor is already being used (or may be used later) for opening files in the guest application. To address this problem, libCanal makes a dummy descriptor by opening *helper driver*, and creates a mapping between the dummy descriptor in the guest and the socket descriptor in the host. Afterward, the guest application issues function calls with the dummy descriptor, but libCanal builds the INR with the host descriptor mapped for the dummy descriptor.

In addition, since the network request should actually be processed with network settings of the host whereas all guests have their own network settings, the network request cannot be directly processed in the host. Especially, when all guests use the same TCP ports for special purpose such as HTTP connection or FTP transfer, it may conflict with other VM connections. In order to reconfigure network parameters of the INR to those suitable for host network settings, the vCanal service uses a hash structure which is indexed by the IP address and TCP ports. It checks whether the INR contains the IP address or TCP port parameters, and then convert them as suitable to the host network through the hash table. The number of TCP ports doesn't exceed 64K, and the guest IP address is usually configured as same subnet mask with the host, consequently we only care about few bits of subnet ID. Therefore, we can maintain the hash structure without wasting much memory. In case of TCP connection, this translation is taken once at the initialization of connection, because the file descriptor is used for the network access after establishing the connection.

3.4 Polling-Based Request and Notification

In vCanal, network requests are delivered through the communication channel without any notification to the hypervisor. Instead, polling thread is pinned onto a certain core for leveraging the multicore architecture, and polls the communication channel. Since many VM servers are currently equipped with over 16 cores, it is an affordable way to dedicate one core for improvement of the network performance. The pinned core continuously executes the polling thread to check if new requests are in the communication channel, and, if so, the vCanal back-end dequeues them and forwards it to the request translator, so as to transform the request into the proper socket type which can be handled by the host. Then, the polling thread selects an available thread in the network thread pool and makes the thread process the requests with the underlying native NIC driver. With multiple VMs, the polling thread should serve all channels. However, just rounding all channels may be inefficient and would harm the scalability of network access in virtualized environment. Therefore, a proper way to select the channel is required. The channel scheduler schedules the vCanal service for each channel. A similar scheduling policy to the fine-grained I/O scheduling of ELVIS [11] was adopted, classifying the sockets by two characteristics: throughput-

intensive and latency-sensitive. We have no device layer in vCanal, and so, we classify the socket functions into three types: TX-related, RX-related, and CMD-related. TX-related functions such as `send()` and `sendto()` access the NIC immediately after being issued, but RX-related functions such as `recv()` and `recvfrom()` should wait until the NIC receives data; the network thread goes to sleep after RX-related functions are issued. Therefore, we gives higher priority to the RX-related functions in order for them to be served first, but its time slice is shorter than TX-related functions so as to improve the throughput of the workloads which send massive data through the network. We gives highest priority to the CMD-related functions such as `socket()`, `bind()`, `listen()` and `shutdown()` since these functions do not access the physical NIC, and thus they might be processed in relatively short time. After finishing the CMD-related functions, the VM yields the vCanal service for other VMs.

We also adopt the polling mechanism for the notification to the guest. ELVIS pointed out that polling in the guest is inefficient, and proposed *exitless interrupt* for a scheme of the guest to host notification. However, since it leverages Intel's x2APIC [18] for exitless interrupt, it is not a complete software solution and cannot be applied on other architectures. In order to mitigate the performance degradation induced by continuous guest polling, we apply a hybrid approach that hypervisor can notify the guest either by polling mode or interrupt mode. At first, libCanal polls `return value` field of the request for a while. If polling exceeds the pre-defined time, libCanal switches-off the polling mode and requests the helper driver to change the notification mode into the interrupt mode. Then, the helper driver registers a special interrupt handler, and informs the hypervisor that the guest should be notified by an interrupt instead of polling. After all, it places the process into an inactivated state to wait an event on a sleep queue. It is difficult to wait the precise period in the virtualized systems due to the jitter of timer interrupt. For this reason, we apply an idea of *futex* in Linux, which repeats polling a specific number of times (default is 100 times) instead of waiting a specific period.

3.5 Communication between the Guest and Host

The communication channel is the per-VM queue structure shared between the libCanal (on the guest side) and the vCanal service (on the host side). It is used to pass network requests to the host without any context-switches. Since access to the communication channel can occur simultaneously at different cores, the access should be synchronized. Although the synchronization can be simply done by the locking mechanism, lock-based synchronization might incur massive contentions on the multicore processor. To solve this problem, the concurrent queue mechanism [19] was adopted for the communication channel. In the vCanal system, each thread in the guest could access the network through the channel of its VM simultaneously, but the polling thread in the vCanal service can serve one channel at a time. Thus, the communication channel can be optimized better with the concurrent queue algorithm designed for multiple enqueuer and single dequeuer.

After a request is delivered to the vCanal service through the communication channel, it can be processed as a native network access. However, there are two technical challenges on processing the request of the guest in the host: socket handover and pointer variable translation. At first, to provide the socket identifier to each guest, vCanal service manages the established sockets with an array in the host and returns only an index of them to the guest application. When an application accesses the network through the socket, the libCanal re-translates the corresponding socket identifier to the index of a host socket and builds the INR with it. The most important part of socket handover is processing `select()` system call. Since established sockets in the host can be classified with the array index (actually, it is used for socket identifier in the guest), `select()` system call can also be processed by delivering the `fd_set` in the guest to the host. Though the `fd_set` is the parameter passed by pointer variables, vCanal service can efficiently pass the pointer variable with *hybrid address space*. More details about the hybrid address space are described in the rest of this section.

The guest applications and vCanal service lie on different contexts, thus the host is unable to directly access the parameters of the socket function which include the pointer variables, such as data buffers. A simple solution is having the host service make another copy for each pointer parameter. However, copying the data on every request causes a lot of memory contention, thus network access in the VM performance may be significantly decreased. Another solution is for the vCanal service to directly access the guest parameters to achieve zero-copy parameter passing. The guest parameter cannot be accessed directly on the host side, thus the address of the guest parameter should be translated to the host address because the host has the virtual address of the guest process memory space.

Translating the guest address to the host address is performed in three steps: 1) translating the GVA (guest virtual address) to the GPA (guest physical address) in the guest kernel, 2) translating the GPA to the HPA (host physical address) in the host kernel, and 3) translating the HPA to the HVA (host virtual address) in the host service. The guest physical address space is part of the host virtual address space, thus translating the GPA to HVA can easily be done by arithmetic calculation. In contrast, the only way to retrieve the mapping between the GVA and GPA is to refer the page table in the guest kernel. However, it is inefficient to translate the address with kernel intervention for every request. To reduce this translation overhead, we propose a *hybrid address space* for efficient address translation. It contains the mapping information between the virtual address and the physical address as shown in Fig. 3, and is managed by a radix tree similar to the page table. The libCanal first finds the mapping in the hybrid address space and uses it for the INR. In the case that the mapping doesn't exist, lib-

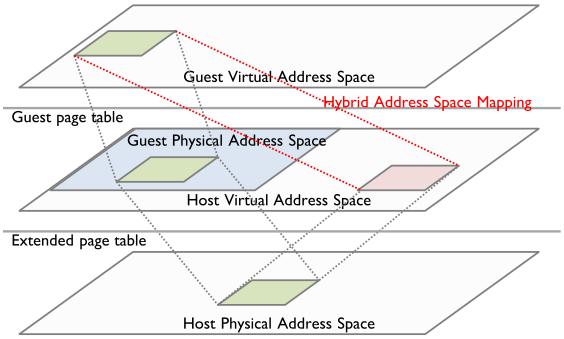

Fig. 3 The relation among three address spaces in virtualized systems. Translating GVA to HVA requires three times of address translation without hybrid address space.

Canal translates the address through the helper driver and stores it to the mapping table for the shared address space for later use. In general, applications frequently reuse their buffer and data structures, consequently allowing the hybrid address space to maintain the mapping with low overhead.

4. Evaluation

To validate the vCanal design, we implemented the vCanal system in the KVM hypervisor. Though our design was implemented in KVM with the processor virtualization support, the only part that depends on the hypervisor was arrangement of the shared memory between the guest and the host; vCanal can also be implemented in Xen without any conscious effort. In this section, we analyze the impact of vCanal's exitless network request and the elimination of redundant network layer. How our optimizations contributed to the performance improvements was also evaluated.

4.1 Experimental Setup

The experiments were performed on two systems which are directly connected with fiber cable. They are used as a client and server for the network benchmark test. Two systems were equally equipped with dual-socket processor with Intel Xeon E5540 CPUs running at 2.53 GHz (4 cores/socket), DDR3 16GB of main memory, and Intel X520-DA2 10Gbps NIC. The client was configured with the virtualization environment using Linux 3.5 for the host and guest, and it used native kvm-tools v2 for the emulation process.

We compared vCanal against the following five I/O virtualization configurations:

Baseline: This configuration is used to measure the performance of the unmodified KVM with *virtio*. All physical cores were assigned to the VM as a typical case where the VM running performance-oriented application dedicates all the cores to itself. The guest and host made no changes for network configuration: setting the Maximum Transmission Unit (MTU) as a default size of 1500bytes and disabling the SR-IOV feature of the 10Gbps NIC.

Baseline + Affinity: The kvm-tools creates a thread per device and the thread continues running on a CPU while performing the I/O request. Since this implementation does not consider the host scheduler, a negative influence on performance may be observed if a virtual processor and I/O thread contend for a single core. To examine this problem, the configuration was set similarly to the Baseline but the cores were explicitly distributed to the VM and the network thread; in our experimental setup, 7 cores were dedicated to the VM and 1 core to the network thread. We also set IRQ affinity to the core executing the network thread, and thus the other cores were not disturbed by the interrupts.

ELVIS: This configuration measured the performance of ELVIS, distributed by the open-source project. To acquire fairness with Baseline + Affinity, 7 cores were also assigned to the VM and 1 core to the I/O thread. The SR-IOV feature was also disabled.

vCanal: This configuration was set for analyzing vCanal performance including all optimizations: hybrid address space and IRQ affinity. The core distribution was the same as for ELVIS; 7 cores assigned to the VM and 1 core to the vCanal service.

SR-IOV: The SR-IOV feature of the NIC was enabled and the guest directly accesses it through the `ixgbe` driver.

4.2 Throughput

Improvements in throughput with vCanal were examined with the following two benchmarks:

Netperf TCP-stream: This is a network benchmark that opens a single TCP socket to the remote Netperf server, and calls as many `send()` functions as possible in a given time. We measured the throughput with the default settings of Netperf except benchmark duration (in our experiments, it is configured as 5 minutes), to avoid the influence on the results by small changes of the system or network state.

ApacheBench: In this benchmark, the VM becomes an HTTP server and has two concurrent threads. *ApacheBench* in the remote system repeatedly sends requests to the VM with 4KB pages and the VM receives and processes those requests. With this benchmark, we measured the aggregated requests in a given time.

As shown in Fig. 4, Baseline achieved 3.1 Gbps on the Netperf TCP-stream; it only took throughput by 57% in comparison with the native environment. On ApacheBench, Baseline also performed 49% of the aggregated request rates compared to the physical environment. Just assigning the affinity to Baseline was not helpful to the TCP throughput, as Baseline + Affinity was found to have 52% of the throughput and 48% of the aggregated request rates on the Netperf TCP-stream and ApacheBench, respectively. Although the IRQ affinity reduced the number of *exits* induced by external interrupts, it was observed that the affinity rather harms the performance of network access in virtualized environment. The major source of performance degradation of the affinity is the inter-processor-interrupt (IPI) that occurs during I/O completion notification. The interrupts from the physical NIC were concentrated on a certain core with the affinity. However, in order to inform the guest OS that the

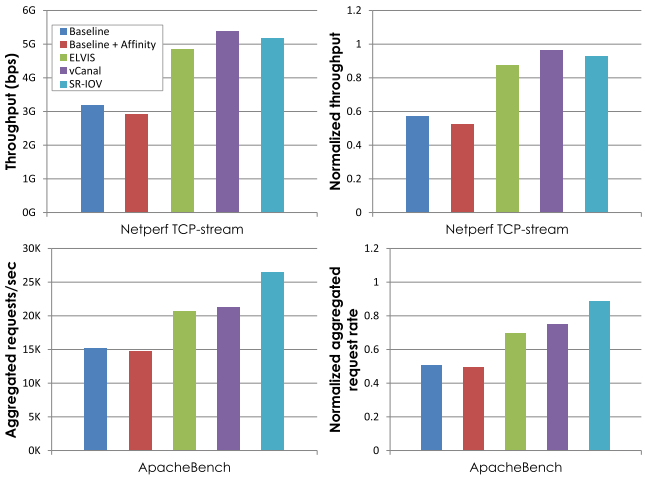

Fig. 4 Comparing throughput of vCanal to other I/O virtualization methods, configured as baseline, baseline + affinity, and etc. Graphs on the left show the result of each benchmark. Graphs on the right show the same result normalized by the native throughput.

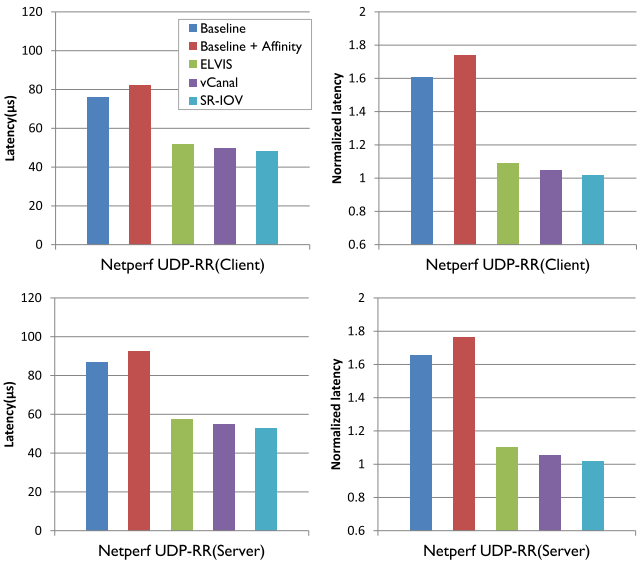

Fig. 5 The latency result of Netperf UDP-RR on both client and server. Graph on the left shows the average latency. Graph on the right is the same measurement normalized to the native UDP latency

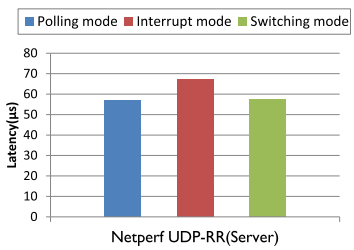

Fig. 6 Comparison of the latency on executing Netperf UDP-RR (server) when vCanal is configured to use three different notification modes

requests were finished, the network thread should send the IPI to the other cores which run VCPU. The IPI is a costly operation and triggers an *exit* to the VM, so the actual performance cannot benefit from the affinity.

In contrast, vCanal achieved a TCP throughput of 5.3 Gbps on the Netperf TCP-stream. This result was nearly close to the native performance and vCanal increased the throughput by 69% in comparison with Baseline. ApacheBench also showed that vCanal achieved 44% increment of the aggregated request rates compared to Baseline. The host notifies the guest of the completion not by sending the IPI but by writing the result on the request message. Therefore, the guest is not interrupted by IPI *exits*, and performance improvement could be observed. ELVIS also increased the throughput of both benchmarks, but their increments are restricted by the redundant TCP/IP layer in the guest.

SR-IOV was found to have lower throughput in the Netperf TCP-stream than vCanal, even though it directly accesses the physical device, since the hypervisor spent a long time for handling the external interrupts due to extreme requests for the network resources. On the contrary, vCanal showed less throughput on ApacheBench, where the reason is that vCanal uses the scheme of switching between the polling mode and the interrupt mode for the guest-to-host notification. Moreover, ApacheBench required more computing power than the Netperf TCP-stream, and thus the overhead of the interrupt mode and the dedicated I/O core incurs performance degradation.

4.3 Latency

We measured network latency reduction using **Netperf UDP-request-response** (UDP-RR). This benchmark sends a UDP packet to the server and measures the time until the sever replies its response. The same benchmark duration as for the Netperf TCP-stream was employed. As shown in Fig. 5, vCanal was observed to reduce latency by 38% (client) and 41% (server) in comparison with the Baseline. Baseline + Affinity also shows higher latency than Baseline for both cases, because the IPI processing took time in the host. Though vCanal outperforms ELVIS for both the TCP throughput and UDP latency, the performance improvements for the TCP connection was found to be relatively higher than for the UDP. This is because the TCP connection has more complicated process than UDP. That is, the guest gets more advantages for the TCP connection by bypassing the entire network stack of the kernel.

Even the switching scheme is applied to the receiving process of vCanal, the benchmark with the VM used as a server showed similiar results to that with the VM used as a client. With our analysis, this is because the UDP packets are repeatedly delievered in a very short time, and thus its notification is mostly conducted by polling. To determine the influence of the switching scheme on the network latency, we configured the vCanal to execute only with the polling or interrupt mode, and compared them to the result of the vCanal using the switching mode. As shown in Fig. 6, though the interrupt mode incured 16% additional latency

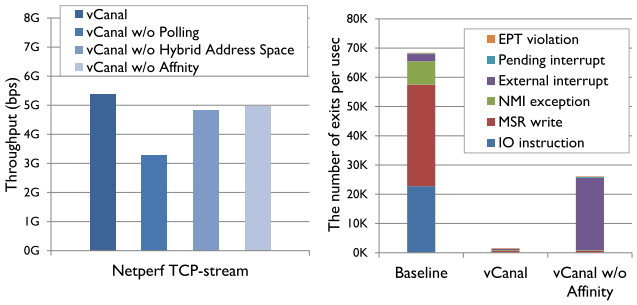

Fig. 7 Comparing the impact of vCanal components. Graph on the left is measurements of TCP throughput for each configuration. Graph on the right shows the number of exits

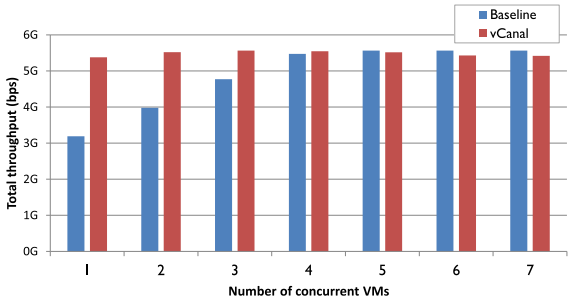

Fig. 8 Total throughput of the Netperf TCP-stream benchmark, when multiple VMs concurrently access the network device through the single I/O core.

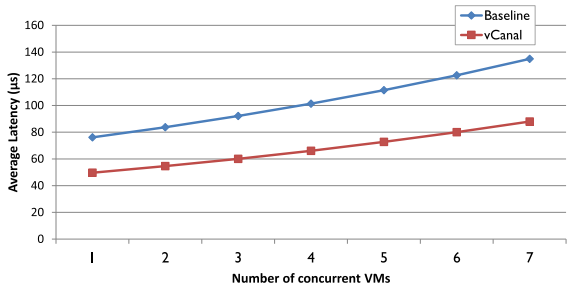

Fig. 9 Average latency of the Netperf UDP-RR benchmark, when multiple VMs concurrently access the network device through the single I/O core.

due to the overhead of processing the interrupt for notifying the completion to the libCanal, it is considered meaningful to apply vCanal to real-world workloads since the notification mode is rarely switched to the interrupt mode when the packet is received frequently.

4.4 Impact of vCanal Optimizations

To verify the effectiveness of vCanal's component, we measured the network throughput using Netperf TCP-stream with four configurations: vCanal, vCanal without Polling, vCanal without Hybrid Address Space Table, and vCanal without Affinity. These configurations disabled design options of vCanal framework as follows relatively: instead of polling, exit-based notification was applied to the case of vCanal without Polling; instead of using hybrid address space, the host translated the address every time in the case of vCanal without Hybrid Address Space; the default settings for thread scheduling and IRQ distribution was used in the case of vCanal without Affinity.

Figure 7 shows the results for all configurations. The results clearly show that the exitless polling-based notification to be the major contributor of performance improvements, increasing TCP throughput by 57%. The hybrid address space and the affinity also improved TCP throughput by 8% and 4%, respectively. The number of *exits* during network processing was further measured to verify how the exitless notification contributed to the performance improvement. The results show that the Baseline suffered from massive *exits* mostly due to three *exit* reasons: I/O instructions, MSR writes, and interrupts. These *exits* were induced by the hardware request, EOI (end-of-interrupt), and completion notification, respectively. Specifically, in order to validate improvement of affinity, we compared the result of `vCanal` to the result of vCanal without Affinity. vCanal reduced approximately 84% of *exits*, whereas the VM still suffer from external interrupts in the case of vCanal without affinity. Though a few *exits* are still remaining even in the case of applying the affinity optimization, it may be induced by other reasons such as timer interrupt. Since those exits are typically generated during the VM is even in an idle state, the performance of network access is not influenced.

4.5 Scalability

To verify that vCanal can serve multiple VMs using a single I/O thread running on a dedicated core, we performed some experiments using $1 \sim 7$ VMs. In all the configurations, we used one core per VM and one additional core to run the I/O thread. As shown in Fig. 8, the single I/O thread (used by both Baseline and vCanal) was saturated at a throughput of around 5.5Gbps, leading to a plateau for 4 VMs and more, where both vCanal and Baseline could scale no longer. This scalability problem would not be an issue for less throughput-intensive workloads (as we demonstrate below), and can also be mitigated by applying the NIC which has higher performance. Moreover, we believe that with further research, the I/O capacity of the single I/O core can be significantly increased, thereby significantly increasing the number of guests which could be served by a single I/O core.

We also measured vCanal's latency improvement as shown in Fig. 9. We can see that vCanal reduced latency by 35µs compared to Baseline when only a single VM was running. With multiple VMs, vCanal reduced the average latency per VM by 39µs. This improvement was possible because vCanal's single I/O thread combined with exitless notifications, as opposed to multiple threads for serving each VM, significantly reduced the time it takes to detect and handle the I/O requests sent by the guests. We can see that, compared to the TCP stream benchmark we previously analyzed, UDP Request-Response did not saturate the I/O core and scaled very well.

5. Limitations

vCanal paravirtualizes socket library instead of the network device driver, and thus it has the advantage of eliminating the redundant network process of the guest kernel by directly delivering the request to the hypervisor from the user-applications. However, vCanal has limitations on compatibility in comparison to the paravirtualized network device driver. First of all, vCanal socket cannot provide some functionalites that the host cannot support. For example, vCanal cannot be applied when the Linux-server hosts the Windows VM executing the applications using Winsock APIs, since Winsock APIs has many functions which is not provided by Berkeley Socket. In contrast, the paravirtualized network device driver can support all socket APIs since it is not coupled with the type of socket API. Moreover, vCanal also have problems on error handling; errors, which occur during the network process in the host, cannot be resolved by the VM directly due to the limited privileged level of the guest OS. Nevertheless, vCanal is still useful since it is an orthogonal solution to the existing solutions. In trusted environment, vCanal is selectively applied by `AF_VSOCKET` to the guest application which requires high-level network performance, and it can provide the guest application with native-level performance.

6. Related Work

An efficient virtual I/O framework has received a lot of interests as a major area of research for virtualization. Here, we present some studies that had the same motivation as our work, to enhance the performance of virtualization by reducing *exits* to the hypervisor.

VMware products use a software technique [20] which inspects the guest code to detect back-to-back pairs of instructions which incur*exits*, and handles the pair of instructions at the time when the first one *exits*. However, this technique can only be applied to the hypervisor using binary translation. For SplitX [8] method, a hardware extension similar to the inter-processor-interrupt (IPI) was proposed for VMs environment, where run on dedicated cores and the hypervisor runs on a different cores, in parallel. It enabled exitless communication between cores which could be used for notifications between the guest and the host, but is not available on commodity processors. In contrast, vCanal does not require any new hardware extensions. ELI [7] achieved nearly bare-metal performance by handling interrupts in the guest directly, without hypervisor intervention. However, its application is very limited because ELI works only for direct device assignment. Moreover, it only focuses on the *exits* induced by interrupts. ELVIS [11] proposed an I/O core similar to vCanal service, but the request is passed to the para-virtualized device driver to successfully achieve exitless I/O with a software-only approach. Though vCanal outperformed ELVIS for network devices by eliminating the redundant network stack in the guest, ELVIS can be adopted more generally for block storage devices.

To eliminate redundant processing, socket-outsourcing [21] and VMM-bypass network [22] is invested the guest application which has the awareness [23] of the virtualization. Nevertheless, they only succeeded in improving the performance up to about 60% of the native device, since their notifications of completion were still engaged by the hypervisor.

7. Conclusion

In this paper, massive switches and redundant I/O stacks were identified as the major contributors in slowing down the network in virtualized systems. We proposed a novel system, vCanal, which eliminates the duplicated network layer with a para-virtual socket library, providing the guest application with awareness of virtualization. vCanal also significantly reduced the number of *exits* by directly passing the requests to the host through an I/O-dedicated core with a polling mechanism. We described our prototype implementation details, in this paper and the results of the evaluation show that vCanal improves throughput up to 69% and reduced latency 38% on average.

Acknowledgments

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2010-0020730)

References

[1] R. Russell, "virtio: Towards a De-facto Standard for Virtual I/O Devices," ACM SIGOPS Operating Systems Review, vol.42, no.5, pp.95–103, 2008.
[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," ACM SIGOPS Operating Systems Review, vol.37, no.5, pp.164–177, 2003.
[3] J. Sugerman, G. Venkitachalam, and B.H. Lim, "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual machine monitor," USENIX Annual Technical Conference, pp.1–14, 2001.
[4] K. Adams and O. Agesen, "A Comparison of Software and Hardware Techniques for x86 Virtualization," ACM SIGPLAN Notices, vol.41, no.11, pp.2–13, 2006.
[5] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High Performance Network Virtualization with SR-IOV," Journal of Parallel and Distributed Computing, vol.72, no.11, pp.1471–1480, 2012.
[6] R. Hiremane, "Intel Virtualization Technology for Directed I/O (Intel VT-d)," Technology@ Intel Magazine, vol.4, no.10, 2007.
[7] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir, "ELI: Bare-Metal Performance for I/O Virtualization," ACM SIGARCH Computer Architecture News, vol.40, no.1, pp.411–422, 2012.
[8] A. Landau, M. Ben-Yehuda, and A. Gordon, "SplitX: Split Guest/Hypervisor Execution on Multi-Core," USENIX Workshop on I/O Virtualization, 2011.
[9] J. Liu and B. Abali, "Virtualization Polling Engine (VPE): Using Dedicated CPU Cores to Accelerate I/O Virtualization," ACM International Conference on Supercomputing, pp.225–234, 2009.

[10] H. Raj and K. Schwan, "High Performance and Scalable I/O Virtualization via Self-Virtualized Devices," ACM International Symposium on High Performance Distributed Computing, pp.179–188, 2007.
[11] N. HarEl, A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger, and R. Ladelsky, "Efficient and Scalable Paravirtual I/O System," USENIX Annual Technical Conference, pp.231–242, 2013.
[12] M. Tsirkin, "vhost-net and virtio-net: Need for Speed," tech. rep., KVM Forum, 2010. http://www.linux-kvm.org/wiki/images/8/82/Vhost_virtio_net_need_for_spe%ed_2.odp.
[13] J.C. Mogul and K. Ramakrishnan, "Eliminating Receive Livelock in an Interrupt-Driven Kernel," ACM Transactions on Computer Systems, vol.15, no.3, pp.217–252, 1997.
[14] A. Menon, A.L. Cox, and W. Zwaenepoel, "Optimizing Network Virtualization in Xen," USENIX Annual Technical Conference, pp.15–28, 2006.
[15] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," USENIX Annual Technical Conference, pp.41–46, 2005.
[16] E. Zhai, G.D. Cummings, and Y. Dong, "Live Migration with Pass-through Device for Linux VM," Ottawa Linux Symposium, pp.261–268, 2008.
[17] B.A. Yassour, M. Ben-Yehuda, and O. Wasserman, "Direct Device Assignment for Untrusted Fully-Virtualized Virtual Machines," tech. rep., IBM Research Report H-0263, 2008.
[18] Intel 64 Architecture x2APIC Specification, Intel Corporation, 2008.
[19] J. Giacomoni, T. Moseley, and M. Vachharajani, "FastForward for Efficient Pipeline Parallelism: a Cache-Optimized Concurrent Lock-Free Queue," ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp.43–52, 2008.
[20] O. Agesen, J. Mattson, R. Rugina, and J. Sheldon, "Software Techniques for Avoiding Hardware Virtualization Exits," USENIX Annual Technical Conference, pp.35–35, 2011.
[21] H. Eiraku, Y. Shinjo, C. Pu, Y. Koh, and K. Kato, "Fast Networking with Socket-outsourcing in Hosted Virtual Machine Environments," ACM Symposium on Applied Computing, pp.310–317, 2009.
[22] J. Liu, W. Huang, B. Abali, and D.K. Panda, "High performance vmm-bypass i/o in virtual machines," USENIX Annual Technical Conference, pp.29–42, 2006.
[23] A. Gordon, M. Ben-Yehuda, D. Filimonov, and M. Dahan, "VAMOS: Virtualization Aware Middleware," USENIX Workshop on I/O Virtualization, 2011.

Dongwoo Lee received his B.S. degree in the Department of Computer Engineering of Sungkyunkwan University, Korea in 2010 and M.S. degree in the Department of Mobile Systems Engineering from Sungkyunkwan University in 2012. He is currently a Ph.D. candidate in the Department of IT Convergence of Sungkyunkwan University. His current research interests include virtualization, cloud computing, and storage systems.

Changwoo Min received his B.S. and M.S. degrees in Computer Science from Soongsil University, Korea in 1996 and 1998, respectively, and he received a Ph.D. degrees in the Department of Mobile Systems Engineering of Sungkyunkwan University, Korea in 2014. He was a software engineer for Samsung Electronics in Korea. His current research interests include virtualization, storage systems, and mobile platforms.

Young Ik Eom received his B.S., M.S., and Ph.D. degrees in the Department of Computer Science and Statistics of Seoul National University, Korea in 1983, 1985, and 1991, respectively. From 1986 to 1993, he was an Associate Professor at Dankook University in Korea. He was also a visiting scholar in the Department of Information and Computer Science at the University of California, Irvine, from Sep. 2000 to Aug. 2001. Since 1993, he has been a professor at Sungkyunkwan University in Korea. His research interests include virtualization, operating systems, cloud systems, and system securities.