IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received February 28, 2019, accepted April 14, 2019, date of publication April 18, 2019, date of current version April 29, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2911905

ZCopy-Vhost: Replacing Data Copy With Page Remapping in Virtual Packet I/O

DONGYANG WANG AND BEI HUA
School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China
Corresponding author: Dongyang Wang (dyw@mail.ustc.edu.cn)

ABSTRACT Virtualization technology is the core technology of cloud computing. While virtualization technology offers flexibility in many ways, it also introduces additional performance overhead. For network systems, it needs to provide additional virtual switching, the virtual packet I/O, and other functions. Providing these features requires a lot of CPU resources. As cloud services grow, more and more CPU resources are being used to provide virtualization support, which means that fewer CPUs can be sold to tenants. Therefore, cloud service providers have been constantly seeking more efficient virtualization solutions for network systems. In this paper, we identify one of the main reasons for the large consumption of CPU resources in the virtual networking system – copy and propose to use memory page remapping technology to eliminate the copy. Moreover, to adapt to the zero-copy technology, we have integrated the technology with a mature virtual switch software in the host and a userspace protocol stack in the virtual machine. These components together build a more efficient network system in virtual environments. The evaluations with microbenchmarks and macrobenchmark show that our system performs much better than the state-of-the-art solutions.

INDEX TERMS Network I/O virtualization, virtio/vhost, zero-copy vhost, virtual switch.

I. INTRODUCTION

Cloud computing is becoming more and more important in today's world. It brings many benefits such as flexibility, high availability and more. One of the most critical technologies for enabling cloud computing is system virtualization, which refers to creating a virtual machine that can act like a real computer with an independent operating system. In virtualization, the software that creates a virtual machine is called a **hypervisor**, and the **host** is the machine which is used by the virtualization and the **guest** is the virtualized machine.

However, the virtualization technology does not only bring benefits, but also introduce performance overhead. Because it needs another virtualized layer to support the virtualization environment. For the networking system virtualization, it needs to provide virtual switching and virtual packet I/O functions. These layers will consume a lot of physical CPU resources. Unfortunately, in the cloud, physical CPUs are used to be sold to customers by virtualizing them to vCPUs (virtual CPUs). Thus, the efficiency of these virtual layers is related to the interests of cloud service providers. To provide more efficient virtual I/O system and save more CPU resources, we should first profile the bottleneck of the current virtual I/O system. Before we locate where the bottleneck is, let's first look at the overall architecture of the virtualized I/O system.

A virtual I/O system mainly includes two parts, one is the packet I/O system which is used to exchange packets between a VM and the virtual switching system. The other is the virtual switching system which is responsible for switching packets between the VM's ports or the physical ports. We mainly focus on the packet I/O system in this paper. There are three classes of packet I/O virtualization modes, namely the full-virtualization mode, the para-virtualization mode and the hardware-assisted virtualization mode [16]. In the full-virtualization mode, the hypervisor emulates a full function of hardware network interface (NIC), which can be driven by the native driver of the hardware NIC. Thus the guest's operating system does not know it is in a virtualized environment. But this solution incurs very high performance overhead. In the para-virtualization mode, a virtual device is driven by a split-driver, which is split into two parts, one is in the guest OS as the *frontend driver*, the other is in the host acting as the *backend driver*. The two drivers cooperates to exchange packets between the guests and the host. This method has better performance than the full-virtualized

The associate editor coordinating the review of this manuscript and approving it for publication was Edith C.-H. Ngai.

2169-3536 © 2019 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

mode and flexible enough, so it is the preferred solution in the cloud. In the hardware-assisted mode, the SR-IOV [14] allows different VMs to share a hardware NIC via different virtual functions (VFs). This method can supply close to bare-metal performance but it cannot be used in the cloud environment due to inflexibility and does not support migration. This paper focuses on the para-virtualization solution that is commonly used in the cloud. There are multiple implementations of the para-virtualization solution and their design principles are similar. Among them, we choose virtio [1] which is open-source and the default para-virtualization I/O framework for QEMU/KVM [13] as our research object.

As we discussed above, virtio adopts a para-virtualization mode and its driver is split into two parts: the frontend driver (virtio driver) and the backend driver (vhost driver). I/O requests are exchanged between the two drivers via virtio queues, which actually are a piece of shared memory between the two drivers. Each queue contains three rings. One descriptor ring to indicate where the packets locate, one avail ring to notify the other side which descriptors are available and one used ring to notify the other side which descriptors are used. The notification mechanism between the two drivers can be based on virtualized interrupts and VM exit/entry. Because virtualized interrupt and VM exit/entry are expensive operations, the performance of virtio is low when using this method. Another method is to poll the rings instead of interrupts. DPDK [2] virtio/vhost driver [17], [18] use this method to achieve high performance. To the best of our knowledge, DPDK virtio/vhost driver is the highest performance implementation of virtio.

In the backend, each vhost is usually connected to a virtual switch (vSwitch) for packet switching. The most commonly used vSwitch is OpenvSwitch [19]. According to our experiments, the vhost consumes a lot of CPU cycles. Therefore, Improving the vhost's performance can save more CPU resource for the cloud provider.

To test the vhost's performance, we use a simple vSwitch [20] (forwarding by destination MAC address) implementation. We conduct the experiment on a server using nine cores (referring section VII-A for the configuration), one runs DPDK vSwitch (with eight vhost ports), each of the other eight cores runs a DPDK packet-generator [21] using DPDK virtio driver in the guest. The eight frontends communicate with pairs through the DPDK vSwitch. We use eight frontend drivers to prevent the frontend from becoming the bottleneck. The experiment result shows that a core can provide a forwarding rate of 10.08 Mpps (packet per second) for 64-bytes packets and 4.16 Mpps for 1514-bytes packets. Then, we continue to analyze the real cause of performance degradation when transmitting large packets. We measure the time of different processing stages in vhost, and find that 34.7% of the CPU cycles were spent on data copy when transmitting 64-byte packets, and the proportion was 78.5% in the case of 1514-byte packets. Moreover, there are jumbo frames [24] that are much larger than 1514-bytes to be forwarded. More CPU cycles will be consumed when copying such packets. Therefore, the data copy is one major bottleneck in the virtual packet I/O process.

A naive method to eliminate data copy is to use shared memory between the virtual machines. But this method will introduce extra security problem. For example, a VM can access a packet that does not belong to it. Reference [8] uses this method to construct a high performance networking system for virtualization environments. But the high-speed zero-copy data delivery can only work between trusted VMs. Therefore, we propose another zero copy design that can achieve high performance without introducing security issues. Based on the design, we develop a high performance networking system for virtual machines. Our work's key contributions are:

1. A new zero copy design for virtual packet I/O that can achieve high performance packet delivery between VMs while keeping VMs' memory isolated.

2. ZCopy-vhost, which is a zero copy high-performance virtual packet I/O library is implemented based on DPDK.

3. Based on ZCopy-vhost, a high-performance networking system (integrating a virtual switching system and a TCP/IP stack) is developed.

The rest of this paper is organized as follows. Section II introduce the background. Section III explores the design space of the zero-copy solution. Section IV and section V introduce the design and the implementation details respectively. Section VII evaluates the performance of our system. Section VIII introduces related works and section IX concludes.

We note that our zero copy design in this paper first appears in IEEE Conference on Local Computer Networks (2017) [26]. Our initial conference paper consists of the design idea of the zero copy solution and a simple prototype to validate the idea. Now we have constructed a whole networking system (including switching system and the TCP/IP networking stack) for VMs based on the zero copy design and the system can achieve higher performance compared to state-of-the-art approach. Moreover, we have added jumbo frames [24] transmitting support in the design, which can improve transmitting efficiency greatly.

II. BACKGROUND

A. KVM & VIRTIO

Kernel-based Virtual Machine (KVM) is a virtualization module in the Linux kernel that allows the kernel to act as a hypervisor. QEMU [6] is a free and open-source emulator, which generally work with KVM to provide a virtualization platform. It exploits KVM to manage the CPU and memory resources. For a VM, each vCPU is emulated by a QEMU thread and runs on the physical CPU. A memory-backend-file is mapped to QEMU and serve as the VM's memory. So, the hypervisor can access all memory address in the VM. Moreover, if another process want to access the VM's memory, it needs to map to memory-backend-file too.

Virtio is a standard framework for I/O virtualization in KVM. It can support virtualization in network, storage and so

on. For network virtualization, virtio mainly consists a virtio device, a frontend driver (virtio driver) and a backend driver (vhost). The two drivers are reside in the VM and the host respectively. Their job is to complete the packet transmission between the VM and the host. the virtio queue in the virtio device are used to exchange information between the two drivers. The drivers of virtio is constantly developing for high performance. There are two versions of virtio implementation now, the kernel version and the userspace version. The userspace version (such as DPDK virtio/vhost) use polling instead of interrupts to achieve higher performance.

B. COPY IN DPDK VIRTIO/VHOST

Intel DPDK is a Data Plane Development Kit that consists of libraries to accelerate packet processing. It uses many optimizations such as hugepage, polling and batch processing to reduce the overhead of data processing.

To support packet I/O virtualization, DPDK implements a virtio driver and a vhost library. The virtio driver is responsible for receiving packets from the virtio device or transmit packets from the upper application to the virtio device. And the vhost library runs in the host, which is responsible for receiving packets from the virtio device to the host or transmit packets from the host to the virtio device. Usually, a vhost library is integrated with a virtual switch, which is responsible for switching packets between different ports. A virtio device is usually treated as a port of the virtual switch. For the convenience of the narrative, we will refer to the backend process as DPDK-vhost.

DPDK-vhost [17] communicates with QEMU through a UNIX domain socket. To access the VM's memory (mainly for packets, packet descriptors, etc.), DPDK-vhost maps QEMU's memory-backend-file into its address space. Moreover, the QEMU's memory is allocated on huge pages, thus the address translation from guest physical address (GPA) to vhost virtual address (VVA) can be done by adding a constant offset to the GPA (i.e., $VVA = GPA + constant_offset$).

Figure 1 shows that a packet is copied twice when exchanged between two VMs. Let us see why these copies exist in the virtual packet I/O. When vSwitch receives a packet, it should first determine the packet's destination (a VM in this example), and then forwards the packet to the destination. A packet cannot go to the destination without its destination being resolved by the vSwitch. As the virtual switch process runs in the host software, the data copy is necessary to forwarding the packet. AccelNet [7] offloads the virtual switch process to the hardware NIC, but it is not a pure software solution. Reference [8] allocates a piece of shared memory between VMs to receive packets, but it brings insecurity problem, because a VM could access a packet that does not belong to it. Before presenting our solution to eliminating this copy, we would like to introduce some relevant principles of address translation in the VM.

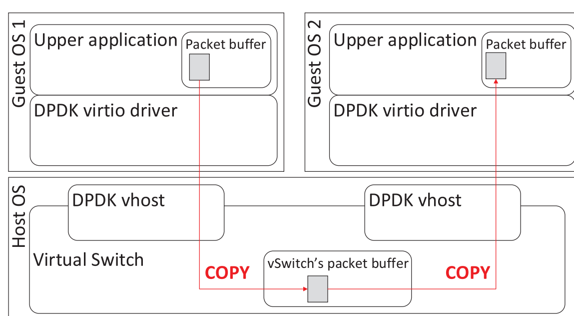


FIGURE 1. Data copy in the DPDK-vhost.

C. ADDRESS TRANSLATION & EXTENDED PAGE TABLE

Traditionally, for a process in the host, virtual address is translated to physical addresses according to the page table of the process. This translation is done by the MMU (memory management unit) hardware. However, when virtual layers are introduced, the translation became complicated. For a process in a VM, it requires two layers of address translation to translate the guest's virtual address to the host's physical address. The first layer of translation is to translate GVA (guest virtual address) to GPA (guest physical address) according to the page table of the process. This page table is managed by the guest OS. The second layer of translation is to translate GPA to HPA (host physical address) according to the EPT (extended page table) of the VM. The extended page table is managed by the hypervisor. accordingly, the MMU is upgraded to support two-layers address translation. As shown in Figure 2, the guest operating system still maintains mappings from GPN (guest page number) to GFN (guest frame number) in the process's page table, and the hypervisor maintains mappings from GFN (guest frame number) to (HFN) host frame number in VM's extended page table. Each VM has an extended page table. MMU translates addresses according to the process's page table and the VM's extended page table.

D. OTHERS

To support virtual switching function, the zero-copy vhost is integrated with Open vSwitch [19], which is a production quality virtual switch. It is also the classic vSwitch in the OpenStack, which is a well-known open source solution for private and public clouds. Some work about the memory management needs to be done to integrate with Open vSwitch.

Meanwhile, a protocol stack is also needed to support running applications. To construct a high performance and stable network system, we choose F-stack [32] as our protocol stack, which is a user space network TCP/IP protocol stack based on DPDK. It has advantages such as high performance and low latency. The TCP/IP stack is ported from the kernel of FreeBSD, which means it is robust. Moreover, F-stack has been deployed in various products in Tencent Cloud [32].

Jumbo frames [24] are Ethernet frames with large size. Conventionally, the payload of an Ethernet frame should not exceed 1500 bytes, which is limited by the MTU (maximum

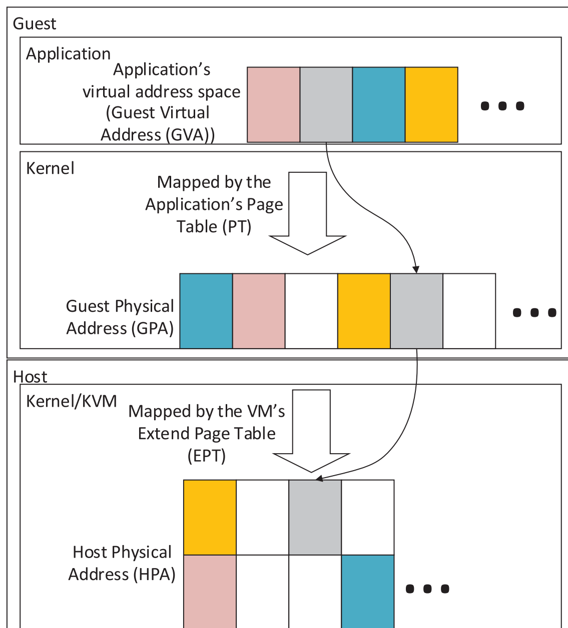


FIGURE 2. Address translation in a VM.

transfer unit). Jumbo frames can carry up to 9000 bytes of payload. This feature can bring in many benefits to computer networking, such as reducing overheads and CPU cycles, improving the end-to-end TCP performance. Thus the feature is supported by more and more hardware devices. Although the feature is naturally supported in traditional virtio/vhost solution, it needs extra care in Zcopy-vhost. Because Zcopy-vhost limits each packet's payload size to 4096 bytes.

III. DESIGN SPACE EXPLORATION

As described in subsection II-B, DPDK vhost needs two copies to transmit a packet from a guest to another. The copies are hard to eliminate because the virtual switching process must be done in the host. Fortunately, page remapping can be used to remove data copy. Instead of copy data from the source process's memory to the destination process's memory, page remapping transmit data by mapping the pages to the destination process, which only involves minor modifications to the page tables. However, modification to the page tables is also a CPU cost operation. Because the page table is managed by the kernel and system calls are needed to enter the kernel. Generally, system calls can consume a lot of CPU. Therefore, in order to amortize the overhead of system calls, batch processing is required. Moreover, it also needs complicated memory management because the page size is fixed and the data may span multiple pages. In order to avoid starting from scratch, we do some changes to DPDK-vhost to meet our needs.

Firstly, In each guest, there exist two packet buffers. One is used to store the packets to be sent and the other is used to receive packets. To switch packets between different guests, DPDK-vhost allocates a block of memory as packet swap area in the host. Since we are going to use page remapping instead of data copy, the packet swap area is no longer necessary. Moreover, to remap the pages in different guests, we need to manage all the pages in all guests' packet buffers. As each guest's packet buffer is mapped into the address space of DPDK-vhost individually, these packet buffers are usually not contiguous in the vhost's address space and in the physical address. However, managing non-contiguous pages is inefficient and complicated. To this end, we remap all the pages in all packet buffers to a piece of memory area that is continuous both in vhost's address space and in physical address.

Secondly, since page remapping requires one-page size of data to be moved at a time, only one packet can be placed in a page. If two or more packets are placed in the same page, they will be switched to the same destination since the page remapping can only move a page size of data at a time. But their destinations may be different. Moreover, to remapping pages, the relationship of GFN (guest frame number) and HFN (host frame number) must be maintained. And the EPT needs to be modified to make the page remapping take effect.

Thirdly, in DPDK, packets are managed by the mbuf struct, which is designed to have an mbuf header followed by a fixed-size area for the packet data. This design only needs one operation to allocate/free the whole memory representation of a packet. However, the design is not suitable for page remapping. As mbuf headers are placed together with packet data, they will also be remapped to the destination if page remapping happens. But the mbuf headers should not be sent out, because they are management struct. Therefore, the mbuf struct should be redesigned to separate mbuf header from packet data.

IV. DESIGN

A. PAGE SIZE FOR A PACKET

Remapping a page need to modify the EPT (extended page table). EPT support three different page sizes on x86_64 CPU [10]. the sizes are 4KB, 2MB and 1GB respectively. Generally, the maximum Ethernet frame size is 1518B and each page can only hold one packet at most. Obviously, the page size of 2MB and 1GB are not suitable for holding a packet. Therefore, We use a 4KB-page to hold a packet. Although this design suffers a 50% packet buffer memory loss compared to DPDK-vhost that uses 2KB buffer for holding a packet, the loss is acceptable in modern computers with large memory since the total packet buffer is small. For the convenience of narration, when we talk about a page, we mean a 4KB page that contains a packet buffer.

However, some Ethernet frames can be large than 4KB. For example, the frame size of jumbo frame [24] can be up to 9000 bytes, which needs at most 3 pages ($3 * 4096 > 9000$) to store it. We chain these pages in the mbuf header. They will be treated as one packet in the switching process, and their remapping process will be done together.

B. PAGE MAPPING

As analyzed in section III, all packet buffers in guests are remapped to a piece of memory area that is continuous both

in vhost's address space and in physical address. we call this memory area host swapping area (HSA). The HSA is managed by pages. Each page holds one packet at most. Pages in HSA cannot be swapped out by the OS. Therefore, for an address in HSA, there exists a constant offset between its VVA (vhost virtual address) and the HPA (host physical address). i.e. $VVA = HPA + constant_offset$.

In the guest, the packet buffer is called GSA (guest swapping area). Each page in GSA is mapped to the HSA. The backend maintains a (GFN, HFN) mapping table for each guest. The mapping relationship is shown in Figure 3. Generally, the GFN can be calculated through the guest physical address in the virtio ring. the backend needs to find which HFN is mapped by the GFN. To provide O(1) lookup time, the mapping table must be realized efficiently. So the HSA and GSA are designed continuous in host physical memory and in guest physical memory respectively. Therefore the mapping table can be realized as an array. For example, the mapping (GFN_x, HFN_x) can be indexed in the array with $index_number = GFN_x - GFN_0$, where GFN_0 is the minimum GFN of the GSA, and the corresponding value will be HFN_x.

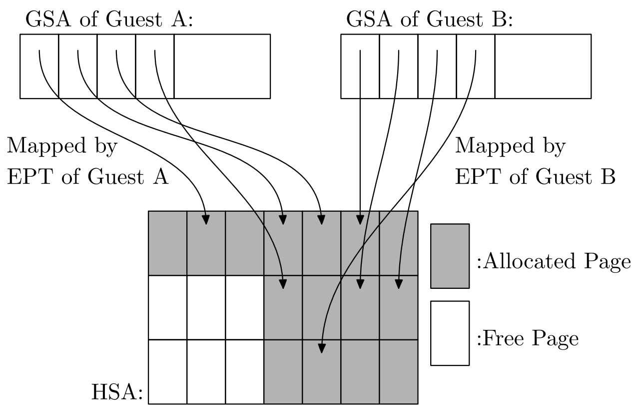


FIGURE 3. Page mapping between GSA and HSA.

1) ZCopy-VHOST RECV PROCESS

When a guest wants to send a batch of packets, it will pass the descriptors of those packets to ZCopy-vhost via vring. For each packet descriptor, Zcopy-vhost obtains the packet's guest physical address (GPA), and translates it into vhost virtual address (VVA) according to the following steps: 1) calculating the guest frame number (GFN) by dividing GPA with 4096; 2) getting the host frame number (HFN) by looking up the guest's page mapping table; 3) calculating the host physical address (HPA) from the HFN; and 4) getting the VVA by adding base_offsset to the HPA. The updating of (GFN, HFN) mapping information is illustrated in Figure 4. After the address is translated, ZCopy-vhost will get a free page from its HSA and exchange pages with the guest. In the guest's page mapping table, page mapping (GFN, HFN_{old}) is replaced by (GFN, HFN_{new}), where the subscript old denotes

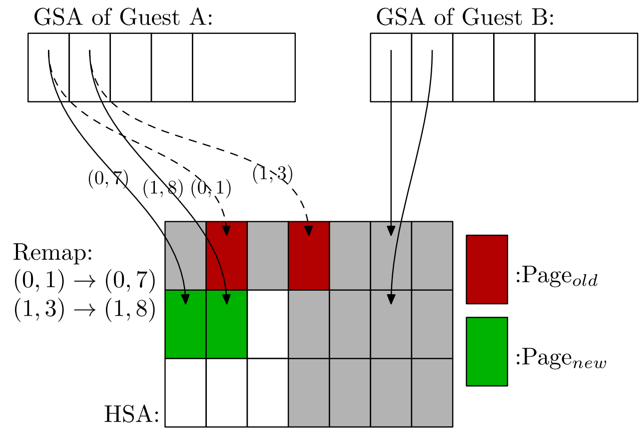


FIGURE 4. Process of receiving two packets from guest A in Zcopy-vhost.

the page holding that packet, and new denotes the free page chosen by Zcopy-vhost. In Figure 4, the dashed lines represent the old mappings, and the tuple *(i, j)* means that the *i-th* page in the GSA is mapped to the *j-th* page in the HSA. After all the packet descriptors are processed, ZCopy-vhost will enter the kernel to modify the EPT to make the new mapping take effect. Finally, it updates vring to notify the guest that it has received the packets. So far, all the packets are received by ZCopy-vhost, and they are inaccessible to the source guest.

2) ZCopy-VHOST SEND PROCESS

When Zcopy-vhost wants to send a batch of packets to a guest, it will get free descriptors from the vring. For each free descriptor, it gets the descriptor's GPA, and then calculates the corresponding GFN, HFN, and VVA as described in the sending process. As shown in Figure 5, after the address is translated, ZCopy-vhost will map the descriptor's GFN to the packet's physical page(the red page shown in Figure 5) and the descriptor's old physical page(the yellow page shown in Figure 5) is reclaimed. After all the packet descriptors are processed, Zcopy-vhost will enter the kernel to modify the EPT. Finally, it updates vring to notify the destination guest

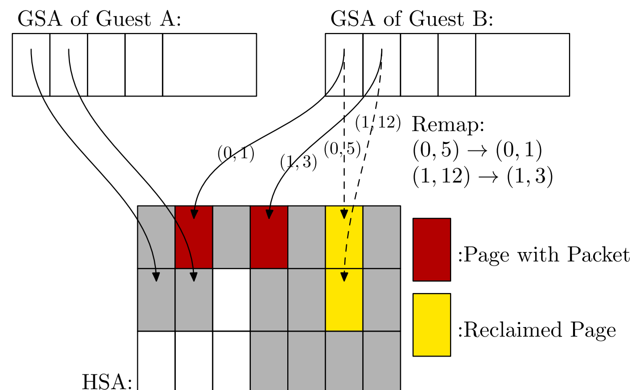


FIGURE 5. Process of transmitting two packets to guest B.

that there are new packets arrived. So far, all the packets are sent to the destination, and pages got from the destination guest are reclaimed.

C. BUFFER MANAGEMENT

DPDK uses struct mbuf to store packets. In this design, the packet is stored after the corresponding mbuf header, which is not suitable for page remapping. Because the mbuf headers which are used to manage packets should not be remapped. Therefore, the packet data must be separated from the mbuf header. To this end, we use two memory pools. One is used to store mbuf headers, and the other is used to store packet data. The memory pool for storing packet data is just the HSA mentioned above, which is divided into 4KB pages. There exists a one to one correspondence between the mbuf header pool and the packet data pool, with the *ith*-mbuf header pointing to the *ith*-packet buffer. The backend can manage the packet buffer pool by managing the mbuf header pool with the corresponding relationship between the two pools. More specifically, when a mbuf header is allocated from the mbuf header pool, which means the corresponding packet buffer is also allocated. When a packet buffer needs to be free, the corresponding mbuf header should be found out depending on the linear correspondence between the two pools, and then the mbuf header is freed.

We exploit the DPDK mempool library as our memory pool implementation, which is an efficient memory management library for fixed-sized objects. It also provides an easy-to-use API to allocate/free objects.

D. OpenvSwitch SUPPORT

OpenvSwitch (OVS) uses the struct *dp_packet* to store packets. For DPDK-OVS (the DPDK version of Open vSwitch), the struct *dp_packet* wraps the DPDK's *rte_mbuf* struct, which means it also stores packet data following the manager struct (i.e., struct dp_packet). So DPDK-OVS's packet buffer management should also be redesigned to support page remapping. Similarly, packet data and *dp_packet* struct are separated by using two memory pools. But the correspondence between the *dp_packet* struct and the packet data is recorded by the *dp_packet* struct. Moreover, besides vhost port, OpenvSwitch also supports many other types of port such as tap port, data copy is also needed to transmit data between these types of port. To distinguish the *zero-copy dp_packet*s (i.e., the packets transferred between vhost ports) from the other *dp_packet*s, a *zcopy_type* field is inserted to the struct *dp_packet*. When packets are transmitted, the *zcopy_type* filed of the *dp_packet* is first identified. For the *zero-copy packet*s, they are processed in zcopy-vhost mode without copy. For other *dp_packet*s, they are processed as they were.

E. USERSPACE STACK

In order to provide a reliable transport service to the upper application, a network stack is needed. The recent emergence of userspace protocol stack technology provides high-performance networking services. Among them, F-stack is ported from the FreeBSD kernel protocol stack and developed based on DPDK, providing high performance while ensuring robustness. F-stack uses two types of struct to store packets, one is DPDK's *rte_mbuf* struct in the driver layer, the other is FreeBSD's *mbuf* struct in the protocol layer. The two types of structs are incompatible with each other, so the packet data are also copied when transferred between the driver layer and the protocol layer. The copy is hard to eliminate because the data should be reserved in the protocol layer for protocol processing such as data retransmission. So we only need to change the *rte_mbuf* to the splitting form of header and data, and keeping the data copy between the driver layer and protocol layer, to avoid dealing with complex protocol processing.

V. IMPLEMENTATION

A. MODIFICATIONS TO DPDK-VHOST

To implement page remapping in DPDK-vhost, the memory management system must be redesigned. There are three key aspects to modify: 1) host swap area (HSA) management, 2) guest swap area (GSA) management for each guest, and 3) the ETP must be modified to make the page remapping take effect.

In the system initialization phase of vhost, we allocate a host swap area (HSA) which is a 2MB-aligned physical-continuous memory. The HSA is divided into 4KB units. Moreover, an mbuf pool is also allocated. Each mbuf in the mbuf pool corresponds to a 4KB unit in HSA. Thus the allocation/free of 4KB units can be implemented by allocating/freeing mbufs in the mbuf pool.

The physical memory of GSA should be located in the HSA. So the allocation process of GSA should be intercepted and then allocate physical memory from the HSA for GSA. When the GSA (4KB aligned) is allocated in the guest (real physical memory has not been allocated yet), the information (guest physical address and length) of the GSA should be told to the DPDK-vhost. As a guest cannot interact with DPDK-vhost directly, we use hypercalls to send this information to KVM, and ask KVM and QEMU to relay the information to vhost. After receiving the GSA's information, vhost allocates real physical memory from HSA for the GSA. It also records the mappings between GSA page number to the HSA page number. To make GSA get real physical memory, vhost enters kernel to modify EPT to map the guest pages in GSA to real physical page frame.

Here we describe the processes in details. First, after the vhost starts, it will create a UNIX socket and listen to it, waiting for QEMU's connection. Then, it allocates HSA for page remapping. When a guest starts, QEMU will establish a connection with vhost, and sends the guest OS's memory information to the vhost.

When a guest application starts, the DPDK virtio driver will allocate a GSA. To map the GSA into the HSA. the GSA's

information needs to be sent to the vhost. First, the GSA's information is sent to KVM by using a hypercall. Information of the GSA includes address and length of the GSA, and MAC addresses of all the virtual NICs that will use the GSA. After issuing the hypercall, the guest will be blocked. Then KVM sends the received information together with the address of the relevant VCPU-structure to QEMU. And QEMU disables the corresponding VCPU-thread, and sends the received information together with QEMU's PID (to distinguish different guests) to vhost.

After receiving the information from QEMU, vhost will create a mapping table for the guest. For each page in GSA, it will allocate a physical page in HSA, and records the mapping in the table. Then it enters the kernel with the address of VCPU-structure and page mappings to be modified through a kernel module we added. The kernel module invokes the KVM module to do EPT modification.

KVM module firstly gets the VCPU-structure, and then gets the EPT corresponding to it, and then modifies the relevant entries in the EPT. After the modification is finished, vhost will notify QEMU to enables the VCPU-thread to run. So far, the VCPU can run with its GSA mapped to HSA.

B. OPTIMIZATIONS TO EPT MODIFICATION

To make the page remapping take effect, vhost need to modify EPT entries in the kernel. The modification of an ETP entry includes some complicated operations, 1) kernel entry/exit; 2) lock operations before modifying EPT; 3) walking multiple levels of page table to find the entries to be modified; and 4) flushing the TLB. According to our test, the time used by these operations is more than that used by copying a large packet. So some optimizations must be made to reduce the overheads.

To amortize the overhead, we use batch processing, which means vhost will modify multiple EPT entries each time it enters the kernel. Because the operation 1 and 4 takes constant time independent of packet numbers, the average overhead of entering kernel and TLB flushing can be largely reduced for a packet. Moreover, For the operation 2, we eliminate the lock operations due to the following facts. EPT is constructed using a lazy evaluation strategy, which means a VCPU-thread does not access EPT unless an EPT violation (i.e., no corresponding EPT entry) occurs. However, in our design, EPT of a GSA is initialized in the allocation phase and maintained solely by vhost. Thus the EPT violation of GSAs will never happen. Therefore the lock operation when modifying EPT for GSA can be eliminated safely.

For operation 3, walking multiple levels of page tables is a time consuming operation. To solve the problem, KVM has already used reverse-map (rmap) mechanism which records the address of the corresponding EPT entry to a given guest page. The rmap is also established by QEMU when EPT violation happens. However, the design does not work for our vhost. Because mapping information of GSA is maintained by vhost rather than QEMU and there will no EPT violation in GSA. Therefore, the old rmap mechanism is not suitable for GSA. Furthermore, in the rmap design, accessing rmap need to acquire lock first, which will degrade the performance. For GSA, the rmap is only accessed by the vhost thread, so the lock operation can be avoided. Therefore, we design a new rmap mechanism for the GSA to accelerate the lookup of EPT entry. To maximum the lookup performance, the rmap is designed as an array, with $(GFN - GFN_0)$ as the index and the address of corresponding EPT entry as the value.

C. TLB FLUSHING

To accelerate the address translation process, TLB (Translation Lookaside Buffer) is used to cache recently-used address translation entries. In the virtual environment, the translation entries from GVA (guest virtual address) to HPA (host physical address) are also cached in TLB. When a page remapping happens and the EPT is modified, the corresponding translation entries in the TLB will be invalid. So after the page remapping, the affected address translation entries in TLB should be flushed.

However, flushing TLB can impact the guest's performance, but the performance of vhost is not affected because its TLB is not flushed. As vhost is easy to become the system bottleneck, we think it is acceptable to improve the vhost's performance by sacrificing some performance of guest. We will evaluate the impact of TLB flushing in section VII.

VI. DISCUSSION & FUTURE WORK

Live migration is needed in many cloud environments. Since ZCopy-vhost is based on para-virtualized solution, supporting live migration is not a complicated problem. On a traditional platform, when live migration begins, KVM records which pages have been modified and continuously transfers the modified pages to the destination host. In our system, when live migration happens, we can also record the remapped pages in KVM, transfer the pages to the remote HSA, and map them to the new virtual machine. We will add migration support for ZCopy-vhost in the future.

For the ZCopy-vhost-OVS, we only modified its virtual packet I/O part, so the other features of OVS such as QoS, Openflow support are preserved.

In fact, the method of page remapping can also be applied to other types of I/O. We will investigate more available scenarios for the method and apply it. For more compatibility, we will consider supporting kernel drivers and kernel protocol stacks in virtual machines in the future.

VII. EVALUATION

In this section, we first prove that ZCopy-vhost performs better than DPDK-vhost, and then we demonstrate that ZCopy-vhost-OVS also outperforms DPDK-OVS. Furthermore, the overhead of each phase in ZCopy-vhost is analyzed. Finally, we use a real-world application, Redis, to evaluate the performance of the whole system.

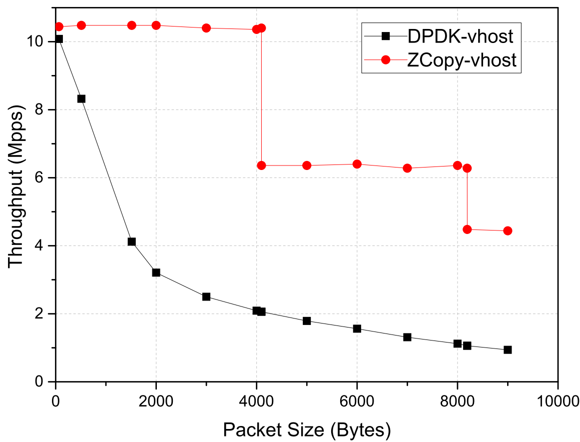


FIGURE 6. Throughput comparison between DPDK-vhost and Zcopy-vhost.

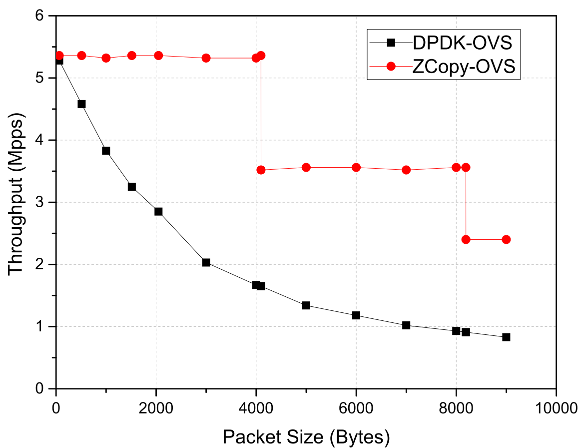


FIGURE 7. Throughput comparison between DPDK-OVS and Zcopy-OVS.

A. EXPERIMENTAL SETUP

We run experiments on a server equipped with a processor (Intel Xeon E5-2690 v3 @ 3.6GHz) and 4 * 16GB of DRAM (DDR4 @ 2133MHz). For the software, we run Ubuntu 14.04.3 with 3.19.8 kernel in the host and Ubuntu 14.04.2 with 3.16.7 kernel in the guest. The QEMU version is 2.5.50 and DPDK version is 17.05.2. Moreover, we integrated Zcopy-vhost with Openvswitch 2.7.0 and F-stack-1.0. A packet generator for DPDK 17.05.2 is used for microbenchmark and Redis is used for macrobenchmark.

We deploy two VMs in the experiments, each with 6 virtual CPU cores and 16GB memory. The virtual CPU cores are pinned to the fixed physical CPU cores. For the vhost, we use one CPU core to do packet switching.

B. THROUGHPUT OF ZCopy-VHOST

In this section, we compare the maximum throughput of ZCopy-vhost with that of DPDK-vhost. To measure the maximum throughput of one core can provide, both solutions use only one core for packet switching. We implemented a simple virtual switch which switches packets based on destination MAC address in the host. Each guest is configured with four CPU cores to send/recv packets. Figure 8 illustrate the configuration of the experiments.

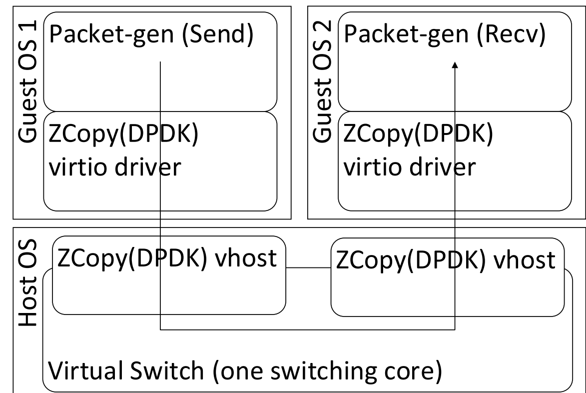


FIGURE 8. Experimental architecture for testing throughput of ZCopy-vhost.

As the throughput is affected by the payload size, packets with different payload sizes ranging from 64B to 9000B are generated. And the burst size is set to 32 for the balance of throughput and latency. For each size of packets, we run experiments five times and the average results are reported. Figure 6 shows the throughput of ZCopy-vhost and DPDK-vhost with different packet sizes.

Three observations can be made from Figure 6. Firstly, Zcopy-vhost performs better than DPDK-vhost in all cases, and the larger the packet size, the more obvious the performance gap. Secondly, the throughput of ZCopy-vhost is almost independent of packet size in three granularities (i.e., 0-4096, 4097-8192 and 8193-9000), whereas the throughput of DPDK-vhost drops significantly with the increase of packet size. This is because page remapping takes constant time for each packet, whereas the time taken by packet copy is proportional to the packet size. Thirdly, when the size of packets is at 4097 or 8193 ($4096 * 2 + 1$), the throughput of zcopy-vhost drops dramatically. This is because the packet occupies one more page when its size reaches these values. So the performance of zcopy-vhost only depends on the number of pages occupied by the packets. For most situations where jumbo frames are not supported, the maximum Ethernet frame size is limited to 1518B. Under this situation, the performance ZCopy-vhost is about 2.5 times that of DPDK-vhost.

C. THROUGHPUT OF OpenvSwitch WITH ZCopy-VHOST

In this section, we measure the maximum throughput of OpenvSwitch (version 2.8.4) with DPDK-vhost (i.e., DPDK-OVS) and Zcopy-vhost separately. DPDK-OVS is a high performance virtual switch that is adopted by many cloud vendors as the default virtual switch. The guests are configured with enough VCPUs so that the guest will not become the bottleneck. The detailed configuration is the same as in section VII-B. The flow table [33] of OVS is configured to switch packets according to their destination MAC address.

We generate packets with different payload sizes ranging from 64 bytes to 9000 bytes. And the result is shown in Figure 7. We can see that Zcopy-vhost can improve the performance of virtual switches, especially when transmitting large packets.

D. IMPACT ON GUEST'S PERFORMANCE

In this section, we evaluate the impact of TLB flushing on the guest's performance. To prevent the backend from being a bottleneck, each VM is configured with one virtual CPU core, which is bound to a physical core. We measure the throughput of ZCopy-vhost and DPDK-vhost with different packet sizes (64B, 521B, and 1514B). Table 1 shows the average throughputs of ZCopy-vhost and DPDK-vhost. The column **Loss of Improvement** is calculated as the ratio of decreased throughput of Zcopy-vhost to the throughput of DPDK-vhost.

TABLE 1. Throughput of a guest on DPDK-vhost and Zcopy-vhost.

Packet Size (Byte)	DPDK-vhost (Mpps)	Zcopy-vhost (Mpps)	Loss of Improvement
64	10.04	4.09	59%
512	8.24	4.09	50%
1514	4.12	4.09	1%

The results show that the maximum throughput of a guest core is limited to 4.09 Mpps due to TLB flushing, which is smaller than that of DPDK-vhost for small packets. However, when packet size is large, the overhead of TLB flushing is similar to the overhead of packet copy.

To sum up, ZCopy-vhost has a great advantage when transmitting large packets.

E. PERFORMANCE BREAKDOWN

In this section, we breakdown the performance of ZCopy-vhost by measuring the time spent in each phase of data transmitting.

In general, three steps are needed to switch packets between two guests: 1) getting packets from source guest; 2) switching packets, i.e., determining the destination for the packet; and 3) sending packets to the destination guest. As the page remapping only affect step 1 and 3, and the page remapping processing for the two steps are similar, we just need to breakdown step 1.

There are roughly five phases to get packets from the source guest: 1) reading packet descriptors from avail vring; 2) allocating pages from HSA for page remapping; 3) remapping page for each packet; 4) entering kernel and modifying EPT; and 5) updating used vring. The average CPU cycles consumed for a packet in different stages is shown in Table 2.

In phase 1 and 5, all operations are performed on the whole batch, which takes constant time independent of the batch size. So the CPU cycles averaged for each packet decrease linearly as the batch size increases. In phase 3, each packet is processed by the same operations, therefore the average time for a packet is almost the same. In phase 2 and 4, the average

TABLE 2. CPU cycles consumed by each packet in different phases under different batch sizes.

Batch Size	1	2	4	8	16	32
Phase 1	185.0	92.5	46.3	23.1	11.6	5.8
Phase 2	144.0	73.5	38.3	22.5	14.3	10.3
Phase 3	32.0	31.0	32.0	28.8	26.9	26.5
Phase 4	310.0	162.0	84.0	47.5	27.6	18.2
Phase 5	30.0	15.0	7.5	3.8	1.9	0.9
Total	701.0	374.0	220.5	125.6	82.3	61.8

cycles for a packet also decrease, but not so severe as that in phase 1 and 5. This is because some operations in phase 2 and 4 are performed on each packet (e.g. modifying EPT for each packet) and some operations are performed independently with batch size (e.g. entering kernel).

The experiments demonstrate the effectiveness of batch processing in ZCopy-vhost. However, when batch size increases too large, the network latency will be higher. In order to balance throughput and latency, we suggest setting batch size to 32.

Table 2 shows the average CPU cycles consumed by each packet in different steps when different batch sizes are used. The CPU cycles are counted on each batch of packets, and then divided by the batch size.

F. REAL-WORLD APPLICATION

In this section, we show the performance of Redis [22], a popular in-memory key-value store, running in the VM. We compare the performance on ZCopy-vhost against DPDK-vhost. The experimental architecture is shown in Figure 9.

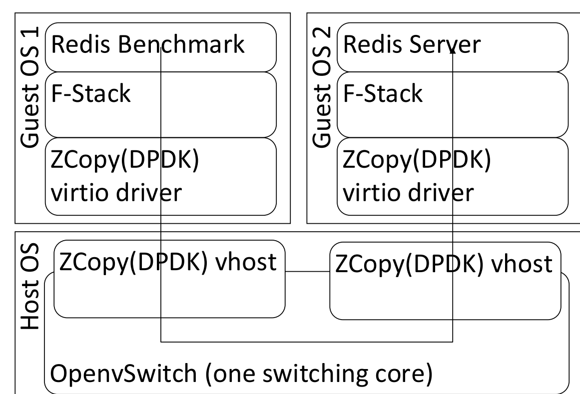


FIGURE 9. Experimental architecture of Redis.

To test the maximum throughput the backend can achieve, we run six Redis (version 3.2.8) instances in one VM, and six Redis benchmark [23] instances in the other VM. For each instance, we issue 1,000,000 of *SET* requests with 100 concurrent connections. The six instance pairs run simultaneously. Payload sizes are set to 64B, 1KB, 64KB respectively. Total throughput is calculated by summing the six instances pairs' throughput. Figure 10 shows the throughput of Redis on both systems. From the figure, we can see

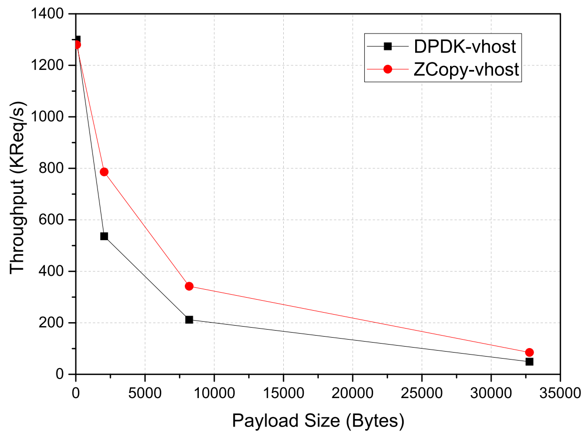


FIGURE 10. The throughput of Redis on both networking systems.

that ZCopy-vhost can provide more throughput than DPDK-vhost, especially when payload size is large. We can attribute the performance improvement to reducing the data copies between guests and the host.

VIII. RELATED WORK

Vhost Data Path Acceleration (vDPA) [28] enables offload of the Vhost vring data path to HW devices in a para-virtualized way without direct pass-through to the guest. vDPA decomposes data path/control path of virtio devices. The data path is pass-through for VRING capable device, while the control path remains to be emulated. The VRING capable device has the ability to enqueue/dequeue VRING and recognize VRING format according to VIRTIO specification. As a result, it can provide near bare metal I/O performance while maintaining live migration ability. But it needs new VRING capable device. More importantly, the queue numbers in the hardware are limited, thus limiting the number of virtio devices, resulting in scalability issues. Finally, it needs to offload software services such as switching, VxLAN tunneling, etc. to the hardware, which is not as flexible as the software solution.

VIRTIO-USER [29] is a new versatile channel for kernel-bypass networks, it is designed for containers to gain better performance by the kernel-bypass virtual switch. It also needs data copy to switch packets between containers and the host.

In accelNet [7] Microsoft Azure offloads the virtual switch to the network and uses SR-IOV to eliminate the virtualization overhead. It also provides the VM live migration ability by turning off hardware acceleration and switching back to its synthetic vNIC. But it does not optimize the software in the VM. What's more, it needs to modify the network card, cannot be deployed with commodity network cards.

NetVM [8] constructs a high performance and flexible networking system based on DPDK. It eliminates the data copy between VMs and the host through shared memory. Therefore, security issues may be introduced because a VM can access packets that do not belong to it. So the benefits can only be gained between trusted VMs.

Many works based on RDMA such as Hyv [30], vRDMA [31] can eliminate the copy between the VM and the host. However, they eliminate the copy based on RDMA networking rather than Ethernet networking. Moreover, RDMA has other problems to be solved when deploying in the public cloud, such as live migration, networking isolation and network security.

IX. CONCLUSION

The paper present ZCopy-vhost, which can eliminate the data copy between the VM and the host in virtual I/O by replacing data copy with page remapping. Evaluations show that ZCopy-vhost can improve the performance of the backend switching system. Furthermore, we integrated ZCopy-vhost with a mature virtual switch, OpenvSwitch, and a user space stack, F-Stack, to build an efficient networking system for the virtual environment. Experiment with real-world application also proves that the page remapping method can save more CPU resources in the backend than the data-copy solution.

REFERENCES

[1] R. Russell, "Virtio: Towards a de-facto standard for virtual I/O devices," *ACM SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 95–103, 2008.
[2] Intel Corporation. *Intel DPDK*. [Online]. Available: http://dpdk.org/
[3] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating two-dimensional page walks for virtualized systems," in *Proc. 13th Int. Conf. Architectural Support Program. Lang. Oper. Syst. (ASPLOS)*, 2008, pp. 1–10.
[4] P. Barham *et al.*, "Xen and the art of virtualization," in *Proc. Symp. Oper. Syst. Princ. (SOSP)*, 2003, pp. 164–177.
[5] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, transparent operating system support for superpages," in *Proc. 5th Symp. Oper. Syst. Design Implement. (OSDI)*, 2002, pp. 89–104.
[6] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX Annu. Tech. Conf.*, 2005, p. 41.
[7] D. Firestone *et al.*, "Azure accelerated networking: SmartNICs in the public cloud," in *Proc. 15th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Renton, WA, USA, Apr. 2018, pp. 51–66.
[8] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High performance and flexible networking using virtualization on commodity platforms," *IEEE Trans. Netw. Service Manage.*, vol. 12, no. 1, pp. 34–47, Mar. 2015.
[9] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 5, pp. 2–13, Dec. 2006.
[10] *IA-32 Intel Architecture Software Developer's Manual*, Intel Corp., Santa Clara, CA, USA, 2001.
[11] Intel Corporation. (2018). *DPDK Mempool Library*. [Online]. Available: http://doc.dpdk.org/guides/prog_guide/mempool_lib.html
[12] Intel Corporation. (2018). *DPDK MBUF Library*. [Online]. Available: http://doc.dpdk.org/guides/prog_guide/mbuf_lib.html
[13] A. K. Qumranet, Y. K. Qumranet, D. L. Qumranet, U. L. Qumranet, and A. Liguori, "KVM: The Linux virtual machine monitor," in *Proc. Linux Symp.*, vol. 1, 2007, pp. 225–230.
[14] PCI Special Interest Group. (2018). *SR-IOV*. [Online]. Available: http://pcisig.com/specifications/iov/single_root
[15] S. Hajnoczi. (2018). *QEMU Internals: Vhost Architecture*. [Online]. Available: http://blog.vmsplice.net/2011/09/qemu-internals-vhost-architecture.html
[16] J. Sahoo, S. Mohapatra, and R. Lath, "Virtualization: A survey on concepts, taxonomy and associated security issues," in *Proc. 2nd Int. Conf. Comput. Netw. Technol. (ICCNT)*, Apr. 2010, pp. 222–226.
[17] Intel Corporation. (2018). *DPDK Vhost Library*. [Online]. Available: http://doc.dpdk.org/guides/prog_guide/vhost_lib.html
[18] Intel Corporation. (2018). *DPDK Vhost Library*. [Online]. Available: http://doc.dpdk.org/guides/nics/virtio.html

[19] B. Pfaff *et al.*, "The design and implementation of open vSwitch," in *Proc. 12th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2015, pp. 117–130.

[20] Intel Corporation. (2018). *Vhost Application*. [Online]. Available: http://doc.dpdk.org/guides/sample_app_ug/vhost.html

[21] DPDK. (2018). *The Pktgen Application*. [Online]. Available: https://pktgen-dpdk.readthedocs.io/en/latest/

[22] (2018). *Redis*. [Online]. Available: https://redis.io/

[23] (2018). *Redis Benchmark*. [Online]. Available: https://redis.io/topics/benchmarks

[24] (2018). *Ethernet Jumbo Frames*. [Online]. Available: http://www.ethernetalliance.org/wp-content/uploads/2011/10/EA-Ethernet-Jumbo-Frames-v0-1.pdf

[25] DPDK Summit 2017. (2018). *Integrating and Using DPDK With Open vSwitch*. [Online]. Available: https://www.dpdk.org/wp-content/uploads/sites/35/2018/06/Integrating-and-using-DPDK-with-Open-vSwitch-.pdf

[26] D. Wang, B. Hua, L. Lu, H. Zhu, and C. Liang, "Zcopy-vhost: Eliminating packet copying in virtual network I/O," in *Proc. Proc. IEEE 42nd Conf. Local Comput. Netw. (LCN)*, Oct. 2017, pp. 632–639.

[27] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "OpenStack: Toward an open-source solution for cloud computing," *Int. J. Comput. Appl.*, vol. 55, no. 3, pp. 38–42, 2012.

[28] DPDK Summit China 2017. (2018). *A Better Virtio Towards NFV Cloud Vhost Datapath Acceleration*. [Online]. Available: https://dpdksummit.com/Archive/pdf/2017Asia/DPDK-China2017-LiangWang-A-Better-Virtio-towards-NFV-Cloud.pdf

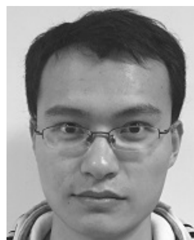
[29] J. Tan *et al.*, "VIRTIO-USER: A new versatile channel for kernel-bypass networks," in *Proc. Workshop Kernel-Bypass Netw.*, Aug. 2017, pp. 13–18.

[30] J. Pfefferle *et al.*, "A hybrid I/O virtualization framework for RDMA-capable network interfaces," *ACM SIGPLAN Notices*, vol. 50, no. 7, pp. 17–30, Mar. 2015.

[31] VMWare Technical Journal. (2018). *Toward a Paravirtual vRDMA Device for VMware ESXi Guests*. [Online]. Available: https://labs.vmware.com/vmtj/toward-a-paravirtual-vrdma-device-for-vmware-esxi-guests

[32] (2019). *F-Stack | High Performance Network Framework Based On DPDK*. [Online]. Available: http://f-stack.org/

[33] The Open Networking Foundation. (2018). *OpenFlow Switch Specification*. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.3.1.pdf



DONGYANG WANG received the B.S. degree from the University of Science and Technology of China, in 2013, where he is currently pursuing the Ph.D. degree. His research interests include networking, system virtualization, and RDMA.

BEI HUA received the bachelor's degree in electronics engineering from the University of Science and Technology of China, in 1990, the master's degree in electronics engineer from Peking University, in 1993, and the Ph.D. degree in computer science from the University of Science and Technology of China, in 2005, where she is currently a Professor with the School of Computer Science and Technology. Her research interests are computer networks and parallel computing.