S2H: Hypervisor as a Setter within Virtualized Network I/O for VM Isolation on Cloud Platform

Ye Yang, Haiyang Jiang, Guangxing Zhang, Xin Wang, Yilong Lv, Xing Li, Serge Fdida, Gaogang Xie

S2H: Hypervisor as a Setter within Virtualized Network I/O for VM Isolation on Cloud Platform

Ye Yang, Haiyang Jiang, Guangxing Zhang, Xin Wang, Yilong Lv, Xing Li, Serge Fdida, Gaogang Xie

***Abstract*—Virtualized Network I/O (VNIO) plays a key role in providing the network connectivity to cloud services, as it delivers packets for Virtual Machines (VMs). Existing para-virtualized solutions accelerate the virtual Switch (vSwitch) data transfer via memory-sharing mechanism, that unfortunately impairs the memory isolation barrier among VMs. In this paper, we categorize existing para-virtualized solutions into two types: VM to vSwitch (V2S) and vSwitch to VM (S2V), according to the memory-sharing strategy. We then analyze their individual VM isolation issues, that is, a malicious VM may access other ones' data by exploiting the shared memory. To solve this issue, we propose a new S2H memory sharing scheme, which shares the I/O memory from vSwitch to Hypervisor. The S2H scheme can guarantee both VM isolation and network performance as the hypervisor acts as a "setter" between VM and vSwitch for packet delivery. To show that S2H can be implemented easily and efficiently, we implement the prototype based on the *de-facto* para-virtualization standard vHost-User solution. Extensive experimental results show that S2H not only guarantees the isolation but also holds the comparable throughput with the same CPU cores configured, when comparing with the native vHost-User solution.**

***Index Terms*—Virtualized network I/O, memory isolation, memory-sharing mechanism, cloud platform.**

I. Introduction

Cloud computing has become a popular paradigm for service provision, due to its ability of providing flexibility, dedicated execution and isolation to a vast number of services. These benefits are achieved thanks to advanced network virtualization techniques, which provide each tenant a Virtual Machines (VM) with its own network topology and traffic control strategy [1]. The VM is an independent operating system running inside the hypervisor (also known as Virtual Machine Monitor, VMM) with isolated running environment, and can flexibly reuse the resources on the physical server. To realize network virtualization, a software virtual switch (vSwitch) is run to provide packet exchange and traffic control

Y. Yang, H. Jiang, G. Zhang are with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China (e-mail: yangye@ict.ac.cn; jianghaiyang@ict.ac.cn; guangxing@ict.ac.cn).

X. Wang is with the Department of Electrical and Computer Engineering, Stony Brook University, Stony Brook, NY 11794 USA (e-mail: x.wang@stonybrook.edu).

Y. Lv and X. Li are with Alibaba Group, Hangzhou 311121, China (e-mail: lvyilong.lyl@alibaba-inc.com; lixing.lix@alibaba-inc.com).

S. Fdida is with the LIP6 Laboratory, Sorbonne University, 75006 Paris, France. (e-mail: serge.fdida@sorbonne-universite.fr).

G. Xie is with the Computer Network Information Center, Chinese Academy of Sciences, Beijing 100190, China (e-mail: xie@cnic.cn).

Y. Yang and G. Xie are also with the School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing 100049, China

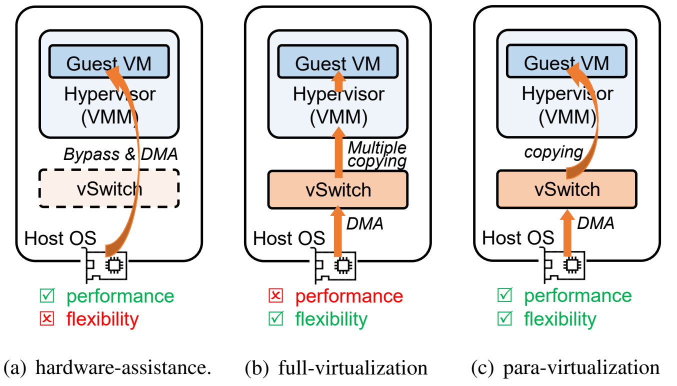


Fig. 1. The types of VNIO solutions. Figs. (b) and (c) are both software based solutions. The arrows represent the packet transferring path. It can be seen that software-based para-virtualization achieves the balance between performance and flexibility, and thus has being accepted by the industry. This paper also focuses on para-virtualized VNIO.

for these high-density deployed VMs. As its most crucial part, the Virtualized Network I/O (VNIO) technology permits the delivery of packets through different I/O paths connecting physical Network Interface Controllers (pNICs) to VMs.

There are mainly two types of VNIO solutions: hardware-assisted and software-based. Hardware-assisted solutions are shown in Fig.1(a). They attain "bare-metal" performance by using Single Root I/O Virtualization (SR-IOV [2]–[4]) that can bypass the virtualization layer so that multiple VMs directly access a single pNIC via different Virtual Functions (VFs)[1]. However, these solutions lose the flexibility enabled by virtualization (*e.g.*, memory overcommitment support [5], VM live migration [6], *etc.*) and face the risks of I/O channel attack [7]. For these reasons, hardware-assisted solutions are not widely adopted in enterprise cloud computing services [8]–[11].

In contrast, software-based solutions can support full-virtualization (see Fig.1(b)) and achieve a high-level of flexibility, but at the expensive cost of reducing system performance due to multiple times of packet copying. To alleviate this crucial issue, para-virtualization (see Fig.1(c))is proposed for reducing times of packet copying and more efficiently transferring the I/O data, with the sharing of memory between VMs and vSwitch. Through providing a good trade-off between performance and flexibility, para-virtualized VNIO has been supported by all well-known vSwitches [12] and widely adopted in many real-world cloud platforms, *e.g.*, Google's

[1]Virtual function (VF) is a kind of pNIC virtualization technology, details can be seen on: http://doc.dpdk.org/guides/howto/lm_bond_virtio_sriov.html

Andromeda [10] and Alibaba Cloud [11] all adopt virtio based para-virtualized VNIO [13]. Nonetheless, the shared memory in para-virtualization goes against strict VM isolation and creates potential risks, *e.g.*, a malicious tenant may escape from its private VM environment and gain access to the shared memory that belongs to other VMs. This isolation issue significantly challenges the security and stability of cloud computing services.

In this paper, we categorize existing para-virtualized VNIO solutions into two types of scheme, *i.e.*, VM to vSwitch (V2S) and vSwitch to VM (S2V), according to their memory-sharing mechanisms. In the V2S scheme, VMs share their private memory with a virtual Switch (vSwitch) process, that launches several Polling Mode Driver (PMD) threads for the Packet Delivery (PD) tasks. As the user-space vSwitch process has the privilege to access all VMs' whole memory, the isolation between VMs and the host is broken. On the contrary, in the S2V scheme, vSwitch allocates a piece of monolithic I/O memory to share with all VMs and the PD procedure is completed inside each VM. A malicious VM may cross its boundary and access other ones' packets, which violates the isolation among VMs.

These isolation issues brought by insecure memory-sharing mechanisms have already been noticed, but the reinforcement solutions are all at the expense of a significant degradation in performance. For example, the community has reported a type of Direct Memory Access (DMA) attack under V2S scheme [14]–[16], and they proposed a solution called vIOMMU [15], [17], [18] to restrict the memory access by reinforcing the address translation procedures. Unfortunately, this solution can only prevent illegal memory access during PD operations, and the insecure shared memory still exists. To make matters worse, the system performance after using vIOMMU will be severely degraded to $\sim$20%. Other memory access protection mechanisms, such as the hardware-based software guard extensions (SGX) [19], are rarely used in the data path of the systems and the use case in Network Function Virtualization (NFV) scenarios has shown that these hardware-based protection mechanisms also severely reduce performance [20].

To effectively guarantee VM isolation under the premise of ensuring performance, we propose a new memory-sharing scheme for para-virtualized VNIO called S2H, which exploits the hypervisor to transfer I/O data between VMs and vSwitch. Compared with the S2V and V2S schemes where vSwitch and VM communicate directly, S2H uses the hypervisor as an intermediate "setter" [2] that isolates VM and vSwitch. Since the hypervisor is maintained by the service provider and has access to the VM memory by default, it is more reliable to use it for memory sharing and packet delivery. On the aspect of performance, in order to maintain the advantage of high throughput brought by memory-sharing, some more innovations in concurrent memory access and scalability are required. First, an efficient framework for memory sharing and access is needed to support the transfer of packets between vSwitch and a number of hypervisor processes. Second, we need to efficiently schedule the concurrent PD procedures, which are distributed in hypervisor processes, to improve the VNIO scalability.

[2]The "setter" in volleyball sport is responsible for passing the ball, and we use it here to describe that hypervisor delivers I/O data between VM and vSwitch.

This paper is based on the conference version [21] while the quality is substantially improved. For example, we propose a more comprehensive of literature review (in Section VII). Moreover, we put forward a more in-depth discussion on the memory-sharing mechanisms and security issue in existing VNIO solutions (in Section II). In addition, we describe in more detail the design and implementation of S2H, and improve the scheduling strategy (in Section III and IV). With these enhancements completed, we open source the code of S2H on github: https://github.com/ictyangye/secure-vhost.git. Lastly, we conduct a more thorough experimental evaluation of our proposed S2H solution. More various scenarios are added for performance test and comparison (in Section V).

Thus the main contributions of this paper are summarized as follow:

- We classify the memory-sharing mechanisms of existing para-virtualized VNIO solutions into S2V and V2S, and analyze how they violate the memory isolation. To guarantee the isolation without degrading the system performance, we propose a new S2H scheme that adopts hypervisor processes for sharing memory with vSwitch and completing PD procedures.
- To efficiently deliver packets via the shared host I/O memory between the vSwitch and hypervisor processes, we take advantage of the Data Plane Development Kit (DPDK) memory management in hypervisor processes for concurrent memory access, and well design the conflict-free packet delivery pipeline for high-speed packet processing.
- To support scalability and run a large number of VMs on a single server, we propose a "batch-grained" scheduling strategy, which schedules PD procedures on limited CPU cores according to batch processing workload instead of uncontrollable time slices. This kind of scheduling strategy brings more efficiency and flexibility.
- We implement S2H prototype based on the vHost-User architecture as it is the *de-facto* para-virtualization standard which has been widely adopted in commercial products. We show that S2H can be simply realized through adding less than 1000 lines of code to the existing solutions. Extensive evaluations are conducted in diversified scenarios and settings including data-path performance, inter-VM performance, application performance, and scalability. The results show that S2H can achieve comparable throughput with 9% more latency than those of the native vHost-User architecture, while effectively guaranteeing VM isolation.

The rest of this paper is organized as follows: Section II analyzes the memory-sharing mechanisms and their isolation issues in existing para-virtualized VNIO solutions. In Section III, we propose the new S2H scheme to resolve these isolation issues. Section IV elaborates our prototype design and implementation. The performance of the proposed S2H scheme is evaluated in Section V. Section VI discusses related technical issues. Related studies are presented in Section VII. Finally, Section VIII concludes this paper.

II. Background and Motivation

In the virtualization technology, the software that creates and runs VMs is called VMM or hypervisor. The host is the physical server on which a hypervisor runs one or more VMs. Each VM is called guest VM. Generally, a hypervisor contains both user-space processes and kernel module. For example, in the QEMU/Kernel-based Virtual Machine (KVM) [22], [23] implementation, a hypervisor layer consists of independent user-space QEMU processes and a KVM module. Each VM is actually virtual CPU (vCPU) thread(s) launched by the corresponding QEMU process and interacts with the KVM module.

VNIO is an important building block of virtualization, and it is responsible for delivering traffic among VMs and pNICs. In a full-virtualization VNIO solution, the hypervisor serves as a dividing layer between the host and the guest VM. The hypervisor emulates a full function of pNIC, that can be driven by the native driver in a guest VM. The full-virtualization is flexible, but the device emulation causes significant performance overheads [24]. Authors in [25] proved that optimizing software interrupts for reducing VM-exit in device emulations can greatly improve the performance of full-virtualization. So in order to break the bottleneck in device emulations and software interrupts for better I/O performance, various para-virtualization solutions are proposed [13].

A. Para-virtualized I/O Data Path

As shown in Fig.2(a), a para-virtualized VNIO solution consists of two parts: a front-end driver in the guest VM and a back-end driver in the host. The front-end handles the virtual device emulation in the guest Operating System (OS), while back-end performs I/O data transfer between host and guest VM. In this paper, we focus on the back-end as the I/O operations mainly happen there.

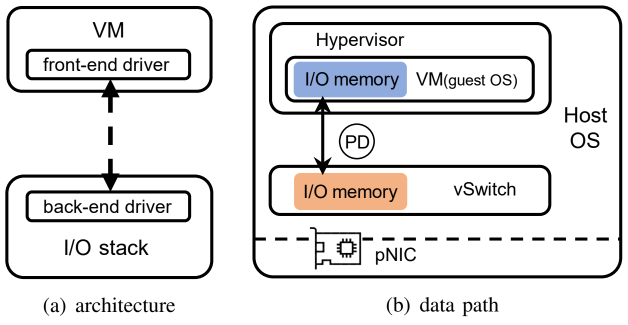


Fig. 2. Para-virtualized VNIO

Fig. 2(b) illustrates typical para-virtualized VNIO environment, with one VM running on the host. As it shows, the back-end consists of a PD procedure and vSwitch. The PD procedure exchanges I/O data between the two blocks of I/O memory residing in the VM and the vSwitch, respectively, while vSwitch is responsible for forwarding traffic among virtual and physical ports. For example, if a packet arrives at a port in the pNIC, the vSwitch component first captures and holds the packet in the I/O memory (the orange colored block in Fig. 2(b)). The vSwitch then parses the packet, looks for its forwarding information and finally notifies the PD procedure to copy the packet from the host to a particular VM.

It is worth noting that, these two blocks of I/O memory in Fig. 2(b) belong to different memory spaces, i.e., host memory space and VM memory space[3]. When the back-end (vSwitch) used to run in the kernel-space, it can access VM memory and do packet copying by default. But in the last decade, to improve performance, vSwitch has been transferred from the kernel-space (known as bridge in Linux kernel) to user-space to exploit high-performance drivers like DPDK and netmap [26]. As a result, the vSwitch and the VM are isolated from each other's memory accesses. To implement the memory copying operations in the PD procedure, a memory-sharing mechanism is needed, i.e., either sharing the I/O memory from vSwitch to VM (S2V), or from VM to vSwitch (V2S), which introduces isolation issues.

B. Memory-sharing Mechanisms

The shared memory largely boosts the I/O performance, as the PD procedure can directly access both of the source and destination memory addresses during memory copying. However, there is a trade-off between performance and isolation. As the motivation of this work, we classify and analyze the user-space memory-sharing mechanisms in para-virtualized VNIO solutions.

According to Fig. 2, the memory-sharing mechanism consists of two blocks of I/O memory (on VM and host sides), three participants (VM, hypervisor, and vSwitch), and a PD procedure. As the PD procedure needs the privilege to access the I/O memory on both sides, the choice of I/O memory-sharing schemes depends on the location where the PD procedure performs, vice versa.

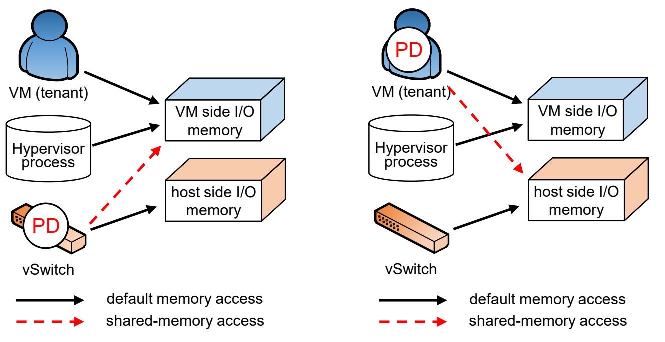
Depending on the locations of the shared memory and PD procedure, we categorize the existing para-virtualization solutions into 2 schemes, S2V shown in Fig. 3(a) and V2S shown in 3(b). In each figure, the PD procedure and three participants (VM, hypervisor, and vSwitch) are on the left, while the two blocks of I/O memory are on the right. The arrow lines present the memory access from the participants to the I/O memory. In particular, a solid line indicates that the memory access is granted by default, while a dashed line indicates that the access is implemented via the memory-sharing mechanism.

Fig. 3(a) illustrates the **V2S scheme**, which is followed by virtio vHost-User [27], Andromeda [10] and ELVIS [8]. Taking the virtio vHost-User solution in QEMU/KVM implementation as an example, VM and QEMU hypervisor are granted the access to the VM's own memory, while the vSwitch is granted the access to the I/O memory on both

[3]The host memory space here indicates the memory space of vSwitch process. As the vSwitch is deployed directly on the host OS and we need to distinguish its memory from VM's memory, we introduce the concept of host memory space.

sides. For vSwitch, the access to the host side I/O memory is granted by default, and the one to the VM side I/O memory is implemented by the V2S memory-sharing mechanism via a series of *mmap()* operations. Several threads (one in default, more with the increase in the VM count or workload) are created in the vSwitch process to deliver packets for all VMs.

Fig. 3(b) illustrates the **S2V scheme**, which is followed by NetVM [28], IVSHMEM [29], [30] and ClickOS [31]. Compared with the V2S scheme in Fig. 3(a), VM is granted the access to the block of I/O memory on the host side. Meanwhile, the PD procedure is moved from the host side to the VM side. Taking the NetVM VNIO solution as an example, vSwitch allocates a block of memory and shares it with the VM. A virtual PCI (vPCI) device is created in the VM, and the device memory is redirected to the block of shared-memory. Therefore, each VM can complete its PD procedure via the virtual device.



(a) The V2S scheme (b) The S2V scheme

Fig. 3. Memory-sharing mechanisms of existing para-virtualization solutions

Both the S2V scheme and the V2S scheme commonly have the isolation issue either among VMs or between the VM and the host. The issue is critical in cloud computing environment, where the per-host VM density is sufficiently high [32]. Taking the V2S virtio vHost-User as an example, the shared memory should be ideally restricted within the I/O memory region in the VM. However, we cannot predict the I/O memory addresses if they are dynamically allocated at run-time. As a result, the VM's whole memory is shared with vSwitch in the practical usage of virtio. The user-space vSwitch process, granted the access to all VMs' whole memory, can easily become the Achilles' heel of the mechanism. Further, existing vSwitch implementations are not reliable. Open vSwitch (OVS) [33], [34], the most popular vSwitch project, has security vulnerabilities reported in the Common Vulnerabilities and Exposures (CVE) database [35], [36]. With a compromised vSwitch process, hijacker can arbitrarily access and even overwrite any piece of memory in VMs via I/O operations. The community has already noticed the security issue, and a type of Direct Memory Access (DMA) attack has been reported [14]–[16]. The issue also exists in NetVM and IVSHMEM S2V schemes. Malicious tenants inside the VM could rewrite the vPCI device driver or exploit existing vulnerabilities and software bugs in the driver to access the host I/O memory without any restrictions [37].

To solve these isolation problems, the current related works are devoted to simply restrict the memory access under existing architectures. For example, to reinforce memory address translation functions, vIOMMU was proposed to prevent DMA attacks [15], [17], [18]. Unfortunately, these reinforcement attempts have severely degraded the VNIO performance (to $\sim$20% of the original performance) and therefore cannot be adopted. Other memory access protection mechanisms, such as the hardware-based SGX [19], are rarely used in the data path of the systems and the practice in NFV scenarios has shown that these hardware-based protection mechanisms also severely reduce performance [20].

Rethinking the two schemes shown in Figs. 3(a) and 3(b), we can conclude that because packets need to be delivered between vSwitch and VM, it is natural to make them share memory with each other. But both schemes have isolation issues and cannot be simply reinforced. It is noteworthy that, until now the hypervisor has not participated in either memory-sharing or PD procedure. However, if the hypervisor process can complete the PD procedure, it has two advantages. First, the hypervisor process can access VM's whole memory by default without memory-sharing. Taking the QEMU/KVM virtualization as an example, each time when booting a VM, a QEMU process needs to be launched with command line parameters that present the properties of a VM. These VM properties include the memory size, the devices' information, *etc*. The QEMU process then emulates devices and allocates the memory for the VM according to these parameters. Therefore, each QEMU can access the memory of its corresponding VM. The second benefit is, compared with the uncontrollable behavior inside the individual VMs, compromising a hypervisor process is known to be much more difficult as it is typically well maintained and monitored by the platform provider [38]. This motivates us to re-examine the existing strategies, and design a new memory-sharing mechanism and architecture with the use of hypervisor to facilitate the memory access.

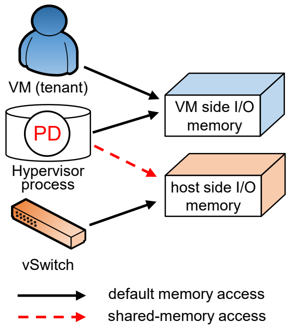
III. The S2H Memory-Sharing Scheme

We propose a new memory-sharing scheme (see Fig. 4(a)) which shares the host-side I/O memory from vSwitch to Hypervisor (S2H) process. Significantly different from S2V and V2S schemes, neither VM nor vSwitch process in the S2H scheme can access the I/O memory that does not belong to itself. Instead, the hypervisor is granted to access the blocks of I/O memory on both sides. Consequently, the PD procedure is moved to the hypervisor layer, and a hypervisor process launches a dedicated PD thread for the corresponding VM.

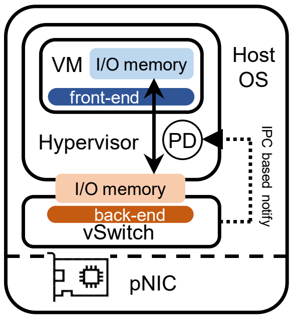
Fig. 4(b) shows the S2H VNIO architecture, where the hypervisor containing a PD procedure works as a "setter" between the front-end and the original back-end. If the PD procedure follows the existing communication protocol, neither front-end driver nor back-end (the part in vSwitch process) needs any modification.

A. Isolation and Security

The main goal of this work is to resolve the VM isolation issues to improve the security of para-virtualized VNIO. We



(a) S2H scheme



(b) S2H VNIO architecture

Fig. 4. The S2H memory-sharing scheme and the VNIO architecture

first analyze the isolation capability of the proposed S2H scheme from two perspectives, isolation among VMs and isolation between a VM and the host.

Isolation among VMs: In the S2V scheme, a malicious tenant could exploit existing vulnerabilities and software bugs in the vNIC driver to acquire or modify other tenants' I/O data via out-of-boundary memory access on the host side. In the proposed S2H scheme, no I/O memory is directly shared with the VM. Instead, the PD procedure in the hypervisor works as a barrier to prevent the unauthorized memory access. As a result, S2H has better isolation among VMs.

Isolation between VM and host: In the V2S scheme, taking the virtio vHost-User as an example, the host-side user-space vSwitch process is granted the access to all VMs' whole memory. If a vSwitch process is compromised, a hijacker can arbitrarily access and even overwrite any piece of memory in VMs via I/O operations. In contrast, in the proposed S2H scheme, no I/O memory is shared from VM to the host, so it has a better isolation between a VM and the host.

TABLE I
THE MEMORY RANGE THAT CAN BE ACCESSED BY VIRTUALIZATION COMPONENTS. ("VM" IN THE TABLE REPRESENTS ONE VM'S WHOLE MEMORY, "HOSTBUF" DENOTES HOST-SIDE I/O MEMORY. THE PARTS HIGHLIGHTED IN ORANGE INDICATE MEMORY ACCESS VIA SHARED MEMORY.)

components / schemes	VM	hypervisor	vSwitch
V2S	VM	VM	hostbuf+all VMs
S2V	VM+hostbuf	VM+hostbuf	hostbuf
S2H	VM	VM+hostbuf	hostbuf

To illustrate the improved security owing to better memory isolation, we show the memory range that each component can access under different memory-sharing mechanisms in Table I. In the table, we can see the proposed S2H minimizes the shared memory size. Comparing with the S2V and V2S schemes, S2H improves the security on two aspects. On the one hand, according to existing security issues, the most common DMA attacks [14] and buffer overflow attacks [36], [39] on VNIO require direct manipulation of sensitive memory addresses. As the proposed S2H can isolate the memory spaces of VM and vSwitch, these well-known attack schemes cannot work. On the other hand, the S2H introduces little security risks while providing better isolation. Hypervisor is the foundation of the virtualization environment and is commonly well maintained by service providers. Comparing with VM operating systems or vSwitch software, the hypervisor is more secure since its size is relatively small and the exported attack surfaces for guest domains are considerably less [38]. Last but not least, in the proposed S2H scheme, we only add the PD procedure (memory copying workload) to hypervisor. That means the introduced code is very low and executes as a separate thread which is independent of the existing code in hypervisor. So the newly added PD procedure is easy to control and maintain.

B. *Design Challenges*

Despite of the potential of providing better isolation and security, the realization of the proposed S2H scheme faces two major challenges. First, as a "setter" process between the front-end driver and the back-end component (vSwitch process), the PD procedure in the hypervisor adds the overhead to the VNIO processing. The sharing of memory between the vSwitch process and the concurrent hypervisor processes is more complex and will affect the performance. Secondly, the CPU resources that can be occupied by PD are very limited. Cloud service providers prefer to assign most of computing resources to VMs and leave very few for VNIO processing. In the Google cloud, no more than two physical CPU cores per physical server are assigned to the PD procedures [10]. Therefore, efficiency and scalability are the main design challenges to realize S2H. We will enhance the efficiency of S2H from two perspectives.

- **Sharing memory among concurrent processes.** As each VM has a dedicated PD procedure in its parent hypervisor process, the shared memory will be accessed by multiple threads that belong to different processes. The block of shared memory is divided into small pieces to storing I/O data and notifications belonging to different VMs, which are occupied, modified and freed by the concurrent PD threads and vSwitch processes. We need an efficient memory sharing framework to deal with the potential conflicts and contention from concurrent processes.
- **Thread scheduling.** As PD procedures run concurrently on limited CPU resources, scheduling these threads with a granularity of time slice is inefficient and will increase the competition, as well as context switching [40]. We need an efficient scheduling mechanism with the granularity that can be properly set according to the working mode of PD procedures.

IV. DESIGN AND PROTOTYPING

To address the above challenges of realizing S2H, in this section, we will elaborate our proposed innovations on efficient memory sharing among concurrent processes and scalable scheduling of PD threads. We introduce our designs in the context of a prototype system. However, our sharing and scheduling strategies are general and can be used to guide the design and implementation over other VNIO architectures.

To demonstrate the function and prove its simplicity in realization, we prototype the S2H system over the vHost-User

(V2S) architecture, as it is a state-of-the-art design and has been widely used in the production environment. The virtio *de-facto* standard that vHost-User follows is also supported by the QEMU/KVM virtualization platform, as well as the kernels of most operating systems such as Linux and Windows.

In this section, we first introduce the basic vHost-User platform and our prototype implementation, and then describe the principles and realization of our proposed memory sharing and thread scheduling mechanism.

A. vHost-User and Basic Platform

On the vHost-User architecture, the user-space QEMU process and the KVM kernel module provide VMs with the virtual execution environment. A corresponding QEMU process is created for each VM. OVS [33], [34], [41], as the back-end in vHost-User, handles the packet classification and forwarding. For high-speed packet processing, OVS is compiled into the Data Plane Development Kit (DPDK) framework [42], which possesses unique features, such as user-space network driver and efficient memory management. When the OVS-DPDK is running, it launches one PMD thread by default, to look up packet destination in flow tables and implement the PD tasks on all physical and virtual ports. The number of PMD threads in OVS-DPDK can be increased as the workload goes up, and each PMD thread must be bound to one dedicated CPU core.

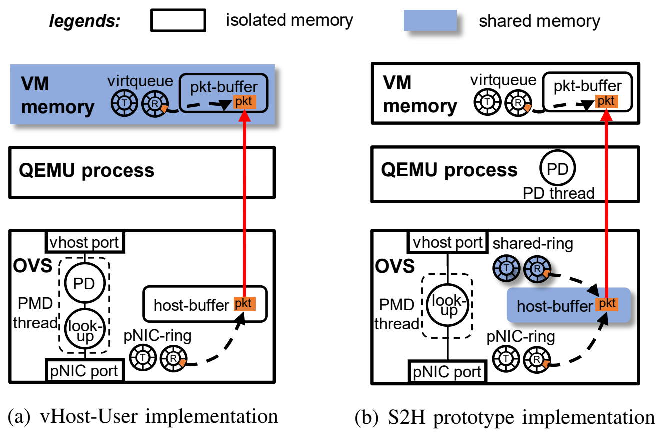


Fig. 5. vHost-User and S2H implementation. The data structures in shadow (blue) color are allocated as the shared memory. The dashed arrows show that the descriptors in rings refer to packets in buffers, and the solid arrows indicate the packet copying paths.

The data exchange of vHost-User architecture is shown in Fig. 5(a). As a typical V2S scheme, the QEMU process does not participate in either the memory-sharing or the PD procedure. On the contrary, OVS is granted the access to the memory both in the host and in VM. According to the function, a PD procedure can be divided into two parts: the notification path for the transmission of packet descriptors and the data path for packet transmission. There are also two types of memory infrastructure involved. In the first type, rings are used to buffer packet descriptors, e.g. the pNIC-ring stores packet descriptors in the host-side memory and the *virtqueue*, as part of virtio standard, stores descriptors on the VM side. As the second type, packet buffers are applied to store packets on both the host side and the VM side. The host-buffer is a block of memory managed by DPDK and used for storing packets received from all OVS's ports, while pkt-buffer is allocated by the VM driver and used for receiving packets from hosts (e.g. skb_buff in Linux kernel).

Both notification path and data path are bidirectional. We take the traffic-in-VM as an example. After pNIC receives packets and stores them in the host-buffer with DMA, the OVS PMD thread can poll the pNIC-rings to access the packets according to the descriptors. It decides which VM is the destination based on the matching of the parsed packet header with the entries in flow table. Then, the PMD thread finds the available pkt-buffer addresses from *virtqueue* of the destination VM and copies packets from the host-buffer to the pkt-buffer. After the packet copying is completed, the PMD thread updates the *virtqueue* of the receiving direction to notify VM for getting packets. The procedure in the opposite direction is similar.

B. S2H Prototype

We modify the vHost-User architecture and implement S2H as shown in Fig. 5(b). The key of S2H architecture is that each QEMU process acts as a "setter" and runs a PD thread between the corresponding VM and OVS to provide the VM isolation. Since each VM has its own QEMU process to complete the PD procedure, OVS needs to share the memory with all QEMU processes on this server. To keep the data path unchanged and avoid additional overhead, we use the host-buffer as the shared memory, and OVS directly exposes the host-buffer to all QEMU processes. On the notification path, due to the need of transmitting packet descriptors between QEMU and OVS, a pair of separated shared-rings is allocated and shared between each QEMU and OVS process. Each time when OVS needs to send packets to VM, its PMD thread only needs to copy the packet descriptors from pNIC-ring to the particular shared-rings to be accessed by the corresponding QEMU process, according to the flow table lookup results.

To efficiently complete tasks on both data path and notification path, we create a dedicated PD thread running in the polling mode for each QEMU. The major workload of the PMD thread on each vhost-port in vHost-User solution is transferred to the PD thread of corresponding QEMU process. As a result, the data path workload remains about the same as that there is no extra packet copying added, comparing with the primitive solution in Fig. 5(a). Meanwhile, as shown in Fig. 5(b), only one extra packet descriptor copying operation is added to the PMD thread of OVS on the notification path. As the descriptor data structure only contains the packet address information, the increased overhead is negligible. Now we also take traffic-in-VM as an example. After pNIC receives packets and stores in the host-buffer via DMA, the OVS PMD thread can poll the pNIC-rings to get the descriptors for the access of packets in the host buffer. Then after the flow table lookup, it copies the packet descriptors from pNIC-rings to the destination QEMU's shared-rings and the PMD thread completes its job. In QEMU, the PD thread polls shared-rings and obtains packet descriptors pointing to the host-buffer.

The following steps are the same as those of in vHost-User—getting available pkt-buffers, copying packets and updating the *virtqueue*.

Compared to the vHost-User, the heaviest workload, packet copying in the PD procedures, is undertaken by the PD threads in QEMU processes. The work done by the PMD threads in OVS is reduced to only the lookup in the flow table and the copying of packet descriptor. As the workload changes, we should allocate most of the CPU resources used by the OVS PMD threads in the native vHost-User to the PD threads of QEMUs. We also design a thread scheduling mechanism for these threads to run on limited CPU resources, which will be described in Section IV-D.

The native vHost-User is modified to prototype the proposed S2H. We only add less than 1000 lines of codes into QEMU to implement the PD thread and make few changes to OVS. As the interface to the *virtqueue* from the PD thread remains unchanged from that of vHost-User, the components inside VM (such as the kernel, VM driver and *virtqueue*) have no need to be modified and are compatible with the configurations in native vHost-User. Though the prototype implementation is based on QEMU/KVM virtualization platform, S2H can also be adopted to other virtualization platforms (e.g. Xen, VMware), as a general VNIO architecture.

C. Memory Sharing with Concurrent Access

As the foundation of VNIO, the shared memory is used to exchange packets between VMs and the host. For example, in vHost-User, each VM shares its memory space (blue in Fig. 5(a)) to the OVS process. It is very efficient and there is no conflicts during the shared memory access as it is essentially a single-producer single-consumer (SPSC) issue. But in S2H, the shared memory (blue in Fig. 5(b)) consists of two parts: the public host-buffer part and the private shared-ring part. The public shared host-buffer is a block of memory, that will be operated (allocate, access and free) by the PD threads of different QEMU processes and the OVS PMD thread simultaneously. So it will lead to a multi-producer multi-consumer (MPMC) issue. For each pair of shared-rings, it is newly added and only shared privately between the corresponding QEMU process and the OVS process.

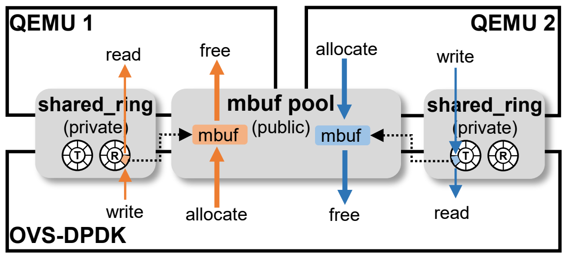


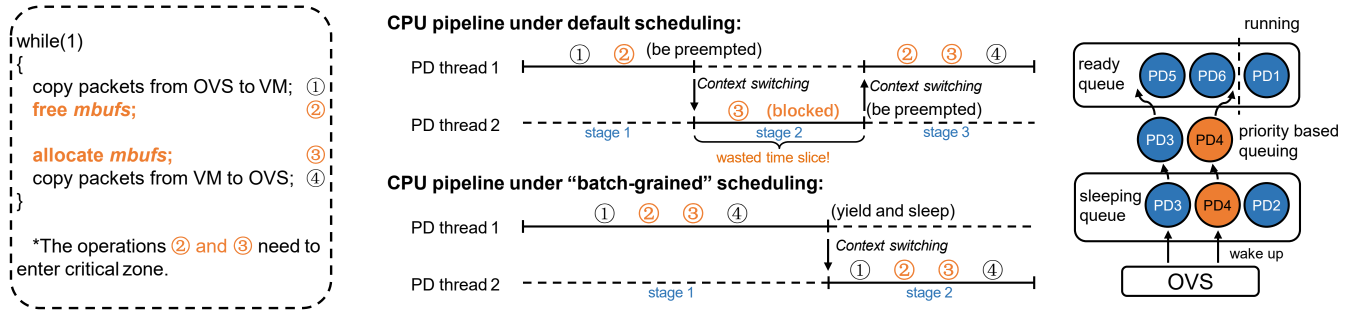
Fig. 6. Concurrent shared memory access in S2H.

We first introduce how we build these two parts of shared memory. For the public shared host buffer, as the it is in fact the *mbuf pool* managed by OVS-DPDK in the native vHost-User, we directly let OVS-DPDK share the *mbuf pool* with all QEMU processes as shown in Fig. 6. To let these QEMU processes have access to the *mbuf pool*, we add "-mem-file = /path-to-rte_config" to the command line parameters for each QEMU process to find and map the *mbuf pool* to its own address space. Once a QEMU process starts and the initialization is completed, the *mbuf pool* has been accessible by to its PD thread. For shared-ring, OVS will create a separate small piece of memory for each QEMU process. Then, each QEMU process attaches to this block of private shared memory. Both types of shared memory are constructed by calling *mmap()* functions to map the files into the process memory space.

To solve the MPMC issue in the public shared host buffer, we take the advantage of DPDK memory management to make the PD threads of all QEMU processes access this block of memory efficiently. As the access to the shared *mbuf pool* is in the form of reading or writing packets, we need a unified memory management and also an efficient conflict-free packet processing pipeline. As the DPDK memory management is designed for processing packets in high speed to efficiently solve the MPMC issue, we implement the same data structures in QEMU as those in DPDK for access the *mbuf pool*. With the same memory management, both QEMU and OVS-DPDK can allocate or free *mbuf* (*mbuf* is a data structure used for storing single packet, like skb_buffer in linux kernel). So we design a conflict-free packet delivery pipeline as shown in Fig. 6. The allocation and free of *mbuf* are performed by the most suitable process according to the direction of the PD procedure. For example, the packet sender process (whether it is OVS-DPDK or QEMU) is responsible for allocating *mbuf* from *mbuf pool*, and the receiver process needs to free it after fetching the packet. Combined with the efficient DPDK memory management, packets are able to be transferred among processes via *mbufs* at high speed.

In the two kinds of shared memory, host-buffer (*mbuf pool*) is consistent with that in vHost-User, but the private shared-ring is newly added. As an intermediate, temporary storage area for the transfer of descriptor, its size may affect the S2H VNIO performance. We construct performance evaluation with varying traffic loads, and find that when the shared-ring size is 4 times greater than the batch size, the system can completely deal with traffic bursts and achieve a reasonable performance. However, if the ring size continues to grow, the throughput will not rise any more. Thus, we set the shared-ring size to be 4 times of the batch size.

Besides the performance concerns, the shared memory in S2H also needs to support other features in VNIO like reconnection and live migration, which are crucial in practical environment. We propose the following schemes for realizing these features under our shared memory implementation. For reconnection, according to the order of shared memory initialization procedure, we let QEMU processes do the *munmap* and free operations on the *mbuf pool* immediately after OVS goes down. When the OVS restarts and successfully allocates the new *mbuf pool*, then all QEMU processes attach to it again. For the live migration, because the shared memory is allocated by OVS, a VM cannot be migrated to another server unless all the packets in the shared memory are processed. As a result,



(a) Working mode of PD threads.

(b) CPU pipeline under default scheduling and "batch-grained" scheduling.

(c) SLA case.

Fig. 7. The "batch-grained" scheduling. (a) shows the main loop of each PD thread, and (b) uses CPU task pipeline to show the difference between the default Linux "SCHED_OTHER" scheduling policy and "batch-grained" scheduling strategy. Obviously, "batch-grained" scheduling avoids blockage and brings flexibility. In (c), we use an SLA case to show that "batch-grained" scheduling also brings flexibility and can make complex QoS strategy easily be implemented in S2H without modifying the architecture, which is not available in vHost-User.

in S2H, before QEMU starts the migration, we need to ensure that all packets referred in the shared-ring have been delivered to VM.

D. Scalability Support

Scalability is an important feature of the cloud platform, which contains two aspects: scalability among VMs and scalability inside a VM. For cloud service providers, the cost is one of their biggest concerns. In a multi-tenant scenario, the biggest scalability issue among VMs is how to use limited CPU resources to support the PD procedures of a large number of VMs. For a single VM, how to increase the receiving and sending queues in vNIC is the key to improve the VM network performance and achieve the scalability inside the VM.

In native vHost-User, OVS PMD threads naturally support these two types of scalability. As mentioned before in Section IV-B, centralized PMD threads poll all the *virtqueues* of each VMs in sequence and do PD on the corresponding ports. The number of PMD threads is equal to the number of CPU cores that is required to complete PD procedures. For multi-queue, OVS needs to add multiple *virtqueues* of a single VM to the polling queue of the PMD threads. It works well for the two types of scalability, and the only limitation is that the PDs of all VMs are executed sequentially and are not flexible enough.

In S2H, the situation is the opposite. As the PD procedures are distributed in PD threads of different QEMU processes, it brings the nature of flexibility. The out-of-order execution of different VMs' PD procedures can support some SLA and QoS strategies of service providers that were not available in traditional architectures. For multi-queue, because the interaction between the PD thread and *virtqueue* remains the same, we only need to increase the number of shared-rings to the same number as *virtqueues*. But as each VM has a dedicated PD thread to perform its PD procedure, the number of PD threads would be relative large. We need to bind multiple PD threads on limited CPU cores and schedule the PD threads on the same core to achieve scalability among VMs.

But binding many PD threads on the same CPU core will bring serious performance issue. By default, these PD threads bound to the same CPU core are managed under the Linux scheduling policy "SCHED_OTHER", which is a kind of completely fair scheduling (CFS) policy. Under the CFS policy, each PD thread is allocated with a fixed time slice to run its workload. After the time slice is used up, the core will be preempted by other PD threads [43]. As the scheduling granularity is too small, frequent context switching among different PD threads will result in huge performance overhead. In addition, PD threads may be preempted when operating the shared data structure and then lock each other. Usually, this situation happens when the PD thread enters the critical zone. For example, as shown in Fig.7(a), allocating and freeing *mbuf* requires the threads to enter critical zone, which are common in the main loop of PD threads. We use the CPU task pipeline under default scheduling in 7(b) to show this issue. Each "stage" in this figure represents the time period of a PD thread occupying the CPU to execute workload until the context switching. In stage 1, the PD thread 1 is still in the critical zone when being preempted by PD thread 2. In stage 2, PD thread 2 will be blocked when it tries to enter the critical zone to allocate *mbufs*. As a result, the time slice is wasted in stage 2. In stage 3, PD thread 1 completes its work and leaves the critical zone, then PD thread 2 will be able to continue its workload after it preempts CPU in the next stage.

We propose "batch-grained" scheduling strategy to avoid blocking and make it more CPU-friendly. For efficiency, the scheduling granularity needs to be well designed: how long each PD thread can occupy CPU and when it can yield the CPU. We use one batch as granularity. First, we set all PD threads' scheduling policy to "SCHED_FIFO". Because the threads under "SCHED_FIFO" will not be preempted unless they yield the CPU by themselves. That gives us the ability to determine the granularity of the schedule by workload rather than uncertain time slice. As the main loop of each PD thread contains polling *virtqueue* and shared-ring to transfer one batch of packets, we allow each QEMU's PD thread to run one loop before it goes to sleep and yields the CPU for other threads. The pipeline is shown in Fig. 7(b), it can be seen this kind of scheduling strategy avoids the blocking caused by context switching in the critical zones and also makes the context switching less frequent.

Besides efficiency, the "batch-grained" scheduling also pro-

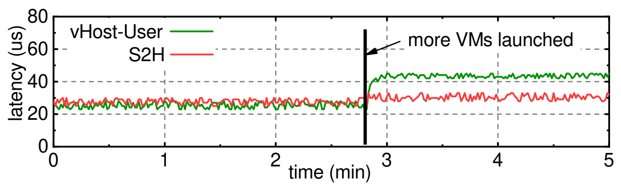


Fig. 8. Single VM's latency during different network conditions. After 3 more VMs are launched and share the same CPU core for PD, the test VM under vHost-User suffers from 50% higher latency while under S2H it almost keeps the same.

vide more flexibility that can be used to design more complex SLA policies. Its details can be found in our another work [44], [45]. Here we show it can be easily implemented in S2H without modifying the architecture. The logical is shown in Fig. 7(c), each PD thread is set with a priority and sleep time threshold. OVS wakes up the corresponding PD thread based on the sleep time and whether there are enough packets to send. But the woken-up PD threads do not immediately occupy the CPU, they stay in a ready queue based on arrival time and priority. For example, the orange colored PD thread indicates that it has higher priority, while the blue color indicates lower priority. When the PD4 with the highest priority is woken up, it will be inserted into the head of the ready queue, but the PD3 with the lowest priority will be placed at the tail. After the PD thread gets the CPU and completes its job, it will go to sleep and wait for the OVS to wake up again. The strategy in this example makes S2H flexible to implement differentiated latency guarantee. A simple test result is shown in Fig. 8, the test VM is set with a fixed sleep time threshold and highest priority. When the background traffic gets larger, the test VM under S2H can maintain the original latency of 30us, while the latency under original vHost-User is increased by 50% to 45us.

TABLE II
THE NUMBER OF TIMES OF CONTEXT SWITCHING ON THE PD CORE. NO PACKET MEANS THE CPU CORE IS DOING CONTINUOUS CONTEXT SWITCHING. IN THE REMAINING CASES, THE CORE DOES PDS FOR FULL LOAD.

	2VM	4VM	8VM
no packet	1.512M/sec	1.524M/sec	1.454M/sec
64-byte	0.196M/sec	0.206M/sec	0.194M/sec
512-byte	0.045M/sec	0.043M/sec	0.043M/sec
1518-byte	0.017M/sec	0.017M/sec	0.017M/sec

The main overhead of "batch-grained" scheduling comes from the context switching among PD threads. As the scheduling only changes the order of threads waiting on the PD cores, it does not affect the processing capacity of the PD cores itself. It is the context switching in thread scheduling that affects PD cores processing capacity. We test the context switching times on a single PD core which serves different number of VMs with different sizes of packets to show how it affects on the performance. The results of *perf* are shown in table II. The overhead of context switching is independent of the number of VMs. As the packet size increases, the effect of context switching becomes smaller. That is because each context switching only happens after each batch process. The time used for one context switching is fixed, while the time used for copying a batch of 1518-byte packets is far much more than copying a batch of 64-byte packets. So, when transferring 1518 byte packets, there will be much less number of times of context switching per second. The worst case happens when copying 64-byte packets, and nearly 10% of the CPU is used for context switching, while the rate with 1518-byte packets drops to less than 1%. This overhead is acceptable for better isolation and flexibility.

In addition to these benefits, there is still a problem that may occur when we bind so many "SCHED_FIFO" scheduled PD threads to the same CPU core. The core cannot be scheduled to any other threads with the default scheduling policy, even they belong to the kernel. For example, some system calls, like page fault interrupts, would run a small piece of codes on each core of the physical server platform. This problem can be easily solved by isolating the PD cores in the *grub* file.

V. EXPERIMENTAL RESULTS AND EVALUATION

In Section IV, we have illustrated that S2H can provide two types of isolation — isolation among VMs and isolation between a VM and the host, which greatly reduces security risks. As the use of hypervisor and newly added shared-ring may introduce additional overheads, in this section, we will evaluate the performance of S2H and compare it with that of native vHost-User.

The performance evaluation and comparisons are from three perspectives. As our goal is to measure VNIO performance, we firstly evaluate and compare the VNIO data path performance of S2H with that of vHost-User. But considering the PD capacity in VNIO data path is ultimately reflected in the application performance inside the VM, so we deploy various types of applications (e.g., IP lookup, TCP, Nginx and VM-to-VM communication) inside the VM to compare the application experience under the two architectures. Besides, to measure the scalability of single VM and the entire physical server, we compare the performance in the multi-queue and multi-tenant scenarios under the two architectures.

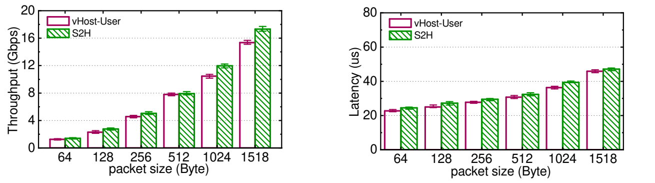
In the experimental setup, we use a server with two Xeon CPU E5-2640 v3 2.60GHz (2x8 cores and 2 logical cores in each physical core) for running the whole cloud platform for the two architectures. The other configurations of the server are as follows: 128GB DDR4 memory at 1866MHz, one Intel 82599ES 10-Gigabit Dual Port NICs, ubuntu 16.04.1 (kernel 4.8.0) as both host OS and guest OS, QEMU 2.10, DPDK 17.11.2 and OVS-2.9.2. Every VM is allocated with 2GB memory and one logical core for all tests.

We use TestCenter from Spirent [46] as traffic generator for the test of packet forwarding and use qperf on a directly connected server with the same configuration for the evaluation of the TCP performance. In all test scenarios, the VM configurations in the two architectures are consistent, and we repeat 10 times of running experiments to eliminate the accidental errors.

A. VNIO Datapath Performance

We first compare VNIO datapath performance by measuring forwarding rate of VM on S2H with that of vHost-User. The

DPDK driver is used inside a VM to ensure that the VM internal network processing will not become a bottleneck. To evaluate the performance of data path with PD running on a single core on vHost-User, OVS launches a PMD thread, which is bound to one logical core. For S2H, as the packet copying by the PD threads in QEMUs take most of the workload originally taken by the PMD thread on vHost-User, we also assign them with a logical core. However, although the OVS PMD thread in S2H only has very lighted look-up workload, it still needs to consume a little CPU resources. So in our implementation, S2H requires a little more CPU resources than vHost-User in any case, because its architecture decouples flow table lookup and PD procedure. The experimental configuration follows RFC 2544 [47] — throughput and latency are evaluated with zero packet loss. Different sizes of packets (64, 128, 256, 512, 1024, 1518 bytes) are generated by TestCenter and forwarded back via the VM internal DPDK l2fwd program.



(a) Throughput. 11% throughput improvement on average.

(b) Latency. 2–9% latency increase.

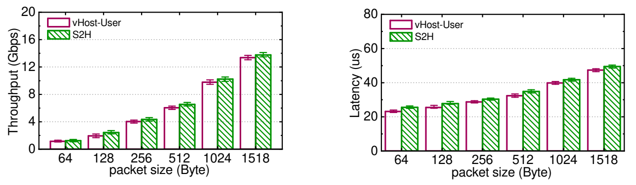
Fig. 9. Datapath Performance comparison with different packet sizes. DPDK l2fwd is used as the forwarding program inside a VM.

Fig. 9(a) shows the datapath throughput of S2H and that of vHost-User. S2H achieves up to 14% throughput improvement with 1024-byte packet and 11% average throughput improvement compared to vHost-User. The throughput improvement of S2H is mainly due to two reasons, 1) a running PD thread cannot be preempted by any other threads thus avoiding the overhead due to unnecessary context switching, and 2) the little CPU resources used by OVS PMD thread undertakes some flow table lookup workload. In Fig. 9(b), the latency of S2H increase by 2–9% under different packet sizes, up to $2\mu s$ compared to those of vHost-User. The increase of latency is caused by the additional workload on the notification path. As mentioned in Section IV, we add one packet descriptor copying operation during each packet delivery. The increase of latency is constant and independent of the packet size.

B. Performance of Applications

For a more realistic study, we consider applications and the kernel protocol stack deployed in the VM of S2H and vHost-User. The performance of the applications including IP lookup, TCP, Nginx and VM-to-VM communication are evaluated.

IP lookup. Running Network NFV applications on the cloud platform is a common trend. To evaluate the performance of NFV in both architectures, we use VM with DPDK driver to enable high-performance IP lookup. Fig. 10(a) shows the throughput of IP lookup on S2H and vHost-User, respectively. S2H achieves up to 20% throughput improvement

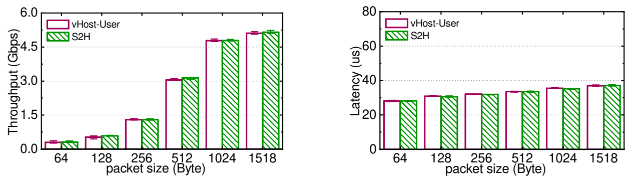


(a) Throughput. 9% throughput improvement.

(b) Latency. 4–9% latency increase.

Fig. 10. Performance comparison of IP lookup with the varying packet sizes. SAIL [48] with DPDK driver is used as IP lookup algorithm and FIB (Forwarding Information Base) contains *600k* entries from a real backbone router.

with 128-byte packet and 9% for average improvement compared to that using vHost-User. Fig. 10(b) shows the latency of IP lookup in S2H and vHost-User. The latency of S2H are 4–9% higher than that of vHost-User. The performance differences in the two implementations is caused, similarly, by CPU resources and the operations of shared-ring on S2H, as illustrated in VNIO datapath performance. There is a 10–15% decline of throughput compared to that on the datapath due to the workload of IP lookup in VMs.

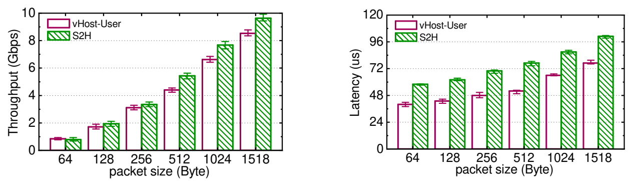


(a) TCP bandwidth.

(b) TCP latency.

Fig. 11. VM's TCP bandwidth and latency comparison. Linux kernel driver are used inside the VM.

TCP stream. TCP performance is an important indicator to measure the stability of VNIO. TCP bandwidth and latency are tested by qperf [49]. The qperf client runs in a VM and the qperf server runs in another direct-connected physical server. Fig. 11(a) shows the TCP bandwidth of S2H and vHost-User. S2H achieves up to 4% throughput improvement with 128-byte packet and 2% improvement on average compared to vHost-User. Fig. 11(b) shows the latency of TCP packet transmission by VM using S2H and vHost-User. The latency is relatively stable and almost the same in the two implementations, as the performance in this case is mainly affected by packet processing in the VM kernel protocol stack.

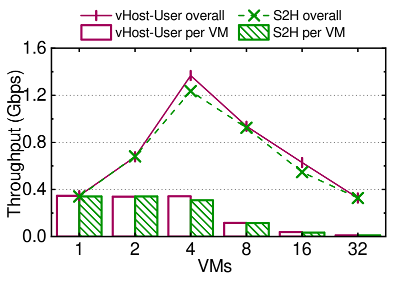


(a) VM-to-VM throughput.

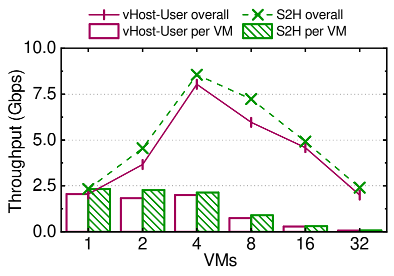
(b) VM-to-VM latency.

Fig. 12. VM-to-VM performance comparison. The service chain is formed by two VMs on both S2H and vHost-User.

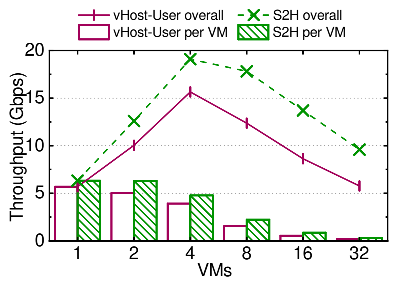
Nginx. As a high-performance HTTP server and reverse proxy, Nginx [50] has been widely deployed on the cloud platform. We further evaluate the performance of Nginx on



(a) Throughput with 64 Byte



(b) Throughput with 512 Byte



(c) Throughput with 1518 Byte

Fig. 13. Scalability comparison of S2H and vHost-User. Sub-figures (a) to (c) show the overall and per VM throughput of S2H and vHost-User. Both implementations achieve peak performance with 4 VMs, where S2H achieves 20% more throughput than vHost-User with 1518-byte packets.

S2H and vHost-User. The Nginx 1.10.3 is deployed in one VM of S2H and vHost-User respectively. We use the Apache Bench running on another directly connected server to test the throughput and response time of Nginx on VM for HTTP requests. The throughput is 9733 responses per second for S2H and 9431 for vHost-User. The average latency per response is 208ms in S2H and 205ms in vHost-User. The performance of Nginx application on the two architectures are very close. The main reason is that the bottleneck appears at the VM kernel protocol stack and the Nginx application, which are configured the same for both architectures.

VM-to-VM. VM-to-VM communication is not very common on cloud tenant platforms. But in some other scenarios like NFV and distributed computing, VM-to-VM performance significantly affects the performance of user's applications. We test the VM-to-VM performance by using two VMs to form a service chain. The traffic generated by TestCenter is sent to one VM (say VM1) through pNIC, and then forwarded to another VM (say VM2) through DPDK l2fwd program. Finally, VM2 uses l2fwd program to forward it back to the TestCenter. As shown in Fig. 12(a), S2H achieves up to 10% throughput improvement with 128-byte packets and 7% for the average improvement compared to vHost-User. Due to the overhead of context switching, the S2H throughput in 64-byte packets is slightly worse than vHost-User. In Fig. 12(b), the latency of S2H increases by 22–30% under different packet sizes. The high latency in S2H VM-to-VM test is caused by the multiple times of packet descriptor copying during each packet's PD procedure and the synchronization overhead among shared *mbuf pool*. To alleviate the heavy overhead of S2H during the VM-to-VM communication, we will use a VM-to-VM fast-path to bypass the vSwitch and reduce times of memory copying. We will discuss our strategy in Section VI.

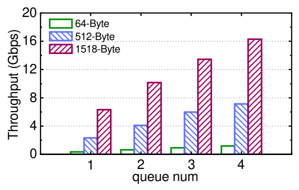
C. Scalability

Multi-tenant and multi-queue scenarios of S2H are evaluated in this section.

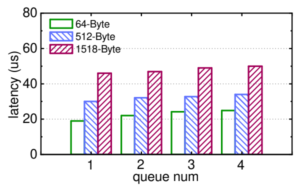
Multi-tenants scalability. Scalability in multi-tenant scenario is evaluated by the network I/O performance with the growing number of VMs on a physical server. To simulate a real cloud platform environment, we run up to 32 VMs on a server. These VMs are all configured with kernel drivers and set the forwarding rules using *iptables* [51], which can forward traffic back to the TestCenter. Two logical cores are assigned to PD procedures in both architectures to undertake the heavy packet copying workload.

The results are shown in Fig. 13. S2H achieves good scalability, especially in the case of large packets. Both S2H and vHost-User achieve the maximum overall throughput when running 4 VMs, where the throughput of S2H is 25% higher than that of vHost-User. As shown in the figure, the throughput per VM and the total throughput decrease with the number of running VMs being increased to 8 or more due to the competition of computing, memory and virtual network I/O resources. This is an inherent issue in OVS and it has nothing to do with this work.

Comparing these figures, it can be seen that the throughput of two architectures have different sensitivity with different packet sizes. The throughput of S2H is slightly worse than that of vHost-User when delivering 64-byte packets. But as packet size increases, S2H has an advantage over vHost-User. When running 8 VMs, S2H's total throughput is about 40% higher than that of vHost-User with 1518-byte packets and 20% higher with 512-byte packets. This is caused by the context switching of PD threads as illustrated in Section IV.



(a) Throughput. Each additional queue increases VNIO throughput by 0.76 times the throughput of a single queue.



(b) Latency. The number of queues has no effect on the latency as multi-queue does not introduce additional overhead.

Fig. 14. Multi-queue performance. Kernel driver and iptables are used for Network Address Translation (NAT) forwarding inside VM.

Multi-queue scalability. Multi-queue is a crucial feature to improve the packets processing ability inside a VM. We evaluate the scalability of multi-queue in the S2H architecture by continuously adding the number of queues in a single VM. The PD procedure is bound to one logical core, and the VM running kernel iptables NAT forwarding is given enough resources (one core for each queue).

Fig. 14(a) shows the throughput of a single VM with the number of queues varying from 1 to 4. Each time one more queue is added, the VM throughput increases by an average of 0.76 times of the single queue performance. The throughput increment decreases as the packet size goes down, for about 0.8 times of the single queue performance with 64-byte packets and only 0.67 times with 1518-byte packets. This is because the PD thread has more packets to deliver in one batch as the number of queues increases, and that greatly reduces the impact of context switching overhead, especially when the packet size is small. The latency of the architecture with different numbers of queues is shown in Fig. 14(b). Since the multi-queue scheme increases the processing capability of VM without introducing additional overheads on both data path and notification path, the latency does not change as the number of queues increases.

D. Summary of Performance Evaluation

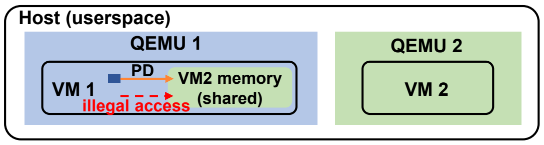
Compared with the native vHost-User, the S2H prototype system achieves up to 11% improvement on data path throughput, but suffers from 2-9% latency increase. These performance fluctuations are negligible, and has no effect on the network applications where the processing bottleneck is inside the VM kernel protocol stack rather than VNIO. S2H also shows good scalability in multi-tenant scenarios. But as limited by the context switching overhead, S2H is at a little performance disadvantage compared to vHost-User in the case of delivering small packets. So in a word, the S2H prototype is performance-friendly, which can maintain the high efficiency as the native vHost-User while guaranteeing memory isolation.

VI. Discussions for Real-World Deployment

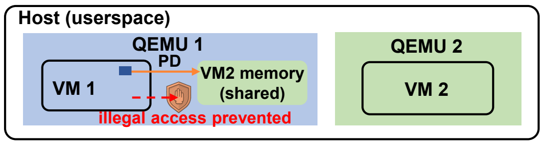
In addition to the implementation challenges mentioned in Section IV, there are still some practical issues in the deployment of S2H architecture, such as VM-to-VM fast-path and adaptation to different virtualization platforms. We discuss the issues that may encounter in a practical environment in this section.

VM-to-VM fast-path. The native vHost-User suffers from the double packet copying during the VM-to-VM communication. To improve the performance, some radical approaches have been proposed. In vhost-pci [52]–[54], one VM maps another VM's whole memory where it can bypass vSwitch to directly perform PD between these two VMs. For example, with the driver modified, one VM can copy packets from another VM's buffer to its own buffer. This method, however, also causes the problem of memory isolation and compatibility. As shown in Fig.15(a), the VM1 is granted to access VM2 memory for direct PD. But that also makes the tenant in VM1 has the privilege to "legally" access VM2's whole memory, which brings significant security issue that we have illustrated in Section II-B.

For S2H, additional overhead is incurred for VM-to-VM communication. The CPU cores used for PD need to operate the newly added shared-rings for several times and suffer from frequently context switching. Inspired by vhost-pci, we consider that the S2H design (placing of PD procedure into the QEMU process) can work well with the idea of building a fast-path in the VM-to-VM communication. As shown in Fig.15(b), the QEMU process of the VM1 can map and access the whole memory of the VM2. When sending packets, the QEMU1 process directly copies packets from the VM1's memory to VM2's memory. After the packet copying is completed, the QEMU1 process updates the VM2's *virtqueue* to notify VM2 to receive packets.



(a) Vhost-pci fast-path solution.



(b) S2H VM-to-VM fast-path solution.

Fig. 15. Compared with vhost-pci directly sharing VM2 memory to VM1, the fast path we designed for S2H uses QEMU to share VM2 memory for fast PD among VMs achieves higher isolation and security.

Compared with vhost-pci, the biggest advantage in our proposed solution is that the shared VM2 memory is invisible to VM1. The illegal memory access is not able to achieve because the memory access from VM1 to VM2 will be isolated and prevented by the virtualization platform (see the shield in Fig.15(b)). On the other hand, the security benefits of using QEMU to perform PD has been discussed in Section III: the higher privilege and more trusty; the illegal memory access is much easier to monitor in hypervisor. Therefore, our proposed fast-path solution for S2H will improve the VM-to-VM performance by reducing the times of packet copying to one without leading to the isolation issue.

Adaptation to other platforms. Although the S2H prototype system is developed based on QEMU/KVM virtualization platform, our proposed memory-sharing mechanism is a platform-independent general solution. For any virtualization platforms, the hypervisor can be divided into two parts: the user-space process and the kernel module. The kernel module is responsible for virtual memory management(VMM). The user-space hypervisor process is the corresponding main process of each VM on the host OS and can be easily found. No matter what the virtualization platform is, the key to realize S2H is to implement PD procedure and shared memory in the user-space hypervisor process. Taking Xen virtualization platform as an example, Dom0 in Xen implements the function similar to the combination of centralized vSwitch and QEMU process in QEMU/KVM. All of its guest VMs run inside separate DomU processes which manages vCPUs [60]. So if we want to adapt S2H design on Xen platform, we can also place the PD thread inside each DomU process as a separate thread.

VII. Related Work

VNIO technique is now a hot topic, and a number of architectures have been proposed. As shown in Table III,

TABLE III
RELATED WORKS. BLACK DOTS REPRESENT GOOD, WHILE HOLLOW DOTS DENOTE POOR.

VNIO solutions	Memory-sharing mechanism	Shared memory location	PD location	Performance	Isolation
vHost-User [27]	V2S	VM	vSwitch	●	○
Andromeda [10]	V2S	VM	vSwitch	●	○
ELVIS [8]	V2S	VM	vSwitch	●	○
Xen1 [55]	V2S	VM	Dom0	●	○
NetVM [28]	S2V	vSwitch	VM	●	○
IVSHMEM [29], [30]	S2V	vSwitch	VM	●	○
clickOS [31]	S2V	vSwitch	VM	●	○
Xen2 [56]	S2V	Dom0	VM	●	○
Xen3 [57]	V2S	VM	Dom0	○	●
vIOMMU [15], [17]	V2S	VM	vSwitch	○	●
Zcopy-vhost [58]	N/A (page flipping)	N/A	vSwitch	○	●
Hyper-switch [59]	N/A (kernel-space VNIO)	N/A	Hypervisor(kernel)	○	●

existing solutions mostly trade off isolation for improved performance based on either S2V or V2S memory-sharing mechanisms. On the contrary, the isolation issue has not been tackled until recent years. Some solutions were respectively proposed to resolve the issue in V2S and S2V schemes on both QEMU/KVM platform [15], [17] and Xen platform [57].

For V2S, existing solutions all adopt the address checking mechanism during the memory access. Authors in [15], [17], [57] proposed vIOMMU solution, which is an emulated IOMMU device and can limit the memory accesses within specific regions during the memory address translation. As the permission checking procedure requiring frequently inter-process communication is commonly very time-consuming, these solutions can even cause the performance loss as high as 80% [15].

For S2V, some existing works can isolate the host side shared memory at the VM level. For example, authors in [28], [37] proposed a scheme to divide shared memory into separate pieces for each VM to store packets. The hardware features like VF and flow classification in pNIC are required to implement such complex DMA operation, so that pNIC can place packets to different pieces of shared memory according to their destination VM. This scheme relies too much on hardware and does not need software vSwitch any more, which makes it more like hardware-assisted VNIO and inflexible.

How to implement user-space VNIO without shared memory has also been the focus of some works. Authors in [58] proposed a VNIO architecture named Zcopy-vhost. Instead of packet copying, the architecture completes the PD procedure by Extended Page Table (EPT) page-flipping technique. However, the architecture also has efficiency issues, as each page-flipping operation is based on a series of system calls.

The hardware-assisted approach for improving security is also a trend. The Trusted Execution Environment (TEE), represented by Intel SGX [19] in x86 and TrustZone [61] in ARM, encrypts data in memory to improve security. Although these techniques have not been adopted in VNIO yet, the use case in NFV scenarios shows that these techniques degraded the performance by 90% [20]. Besides TEE, Intel also proposed a low-overhead CPU feature called Memory Protection Keys (MPK) [62] [19] to restrict the memory pages that one thread can access for enhancing memory isolation. In this way, malicious threads cannot access the private memory of other threads although they are inside the same process. But it has nothing to do with the isolation issue in S2V and V2S because the potentially malicious thread is exactly the worker thread which needs to complete the PD procedure and access the shared memory. For S2H, MPK can be used to isolate PD thread in the QEMU process, so that the newly added PD thread is completely independent of the existing components in the native QEMU. That will make S2H closer to commercial deployment.

In addition to these security reinforcement solutions, there is another work similar to our design that uses hypervisor to implement PD for VMs. Unlike our focus on isolation, authors in hyper-switch [59] placed the *whole* OVS data plane into the *kernel-space* part of hypervisor on the Xen platform to improve the performance of *kernel-space* VNIO. In terms of performance, the kernel-space data path in hyper-switch puts it at a disadvantage to S2H on leveraging the acceleration of high-performance user-space components. So it is obvious that the performance of hyper-switch cannot be compared with the OVS-DPDK accelerated S2H solution [63]. In terms of isolation and security, placing the whole OVS data plane into the *kernel-space* part of the hypervisor really can guarantee VM memory isolation, but it also introduces a large number of codes into kernel along with the attack surfaces. Even a vulnerability in OVS data plane may cause the host OS kernel to be attacked. Therefore, the scheme in S2H that only puts the PD procedure as a separate thread into the user-space part of hypervisor will be more secure and efficient.

In summary, in order to improve the VM isolation, these existing solutions restrict or remove the memory-sharing mechanisms. In contrast, we consider the efficient memory-sharing is critical for the performance. We therefore propose an S2H solution with a new memory-sharing mechanism that exploits hypervisor as a barrier for the VM isolation. To maintain the high performance brought by memory sharing mechanism, we also use the innovations in concurrent memory accesses and efficient thread scheduling to reduce the overhead introduced by S2H.

VIII. CONCLUSIONS

Para-virtualized VNIO is an enabling technology in the context of cloud computing. Existing para-virtualized VNIO solutions often pursue the performance at the expensive cost of VM isolation. In this work, we classified existing para-virtualized solutions into S2V and V2S schemes according to the memory-sharing mechanism and then analyzed their isolation issues. To solve this problem, we proposed a new S2H scheme, which shares the host-side I/O memory to the hypervisor. In order to adopt the S2H scheme in the VNIO design, we implemented an efficient shared memory access which exploits the DPDK memory management to address the MPMC issue. In addition, we proposed a "batch-grained" scheduling strategy for PD threads to ensure network performance in multi-tenant scenarios. We integrated the *de-facto* software-based VNIO standard, vHost-User architecture into our prototype S2H system and evaluated its performance. The prototype exhibited good trade-off between the isolation and the performance. It can achieve the VM isolation with the comparable throughput and less than 9% latency increase compared to the techniques based on the native vHost-User. The results also demonstrated the effectiveness of the proposed concurrent shared memory access and scheduling strategy in ensuring the scalability.

References

[1] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. J. Jackson, A. Lambeth, R. Lenglet, S. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang, "Network virtualization in multi-tenant datacenters," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 203–216, USENIX Association, 2014.

[2] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan, "High performance network virtualization with SR-IOV," in *Proceedings of the 16th International Conference on High-performance Computer Architecture (HPCA)*, pp. 1–10, IEEE Computer Society, 2010.

[3] R. L. Solomon and T. E. Hoglund, "Paravirtualization acceleration through single root i/o virtualization," 2012. US Patent 8,332,849.

[4] M. F. Aris Leivadeas and N. Pitaev, "Analyzing service chaining of virtualized network functions with SR-IOV," in *Proceedings of the 21st IEEE International Conference on High Performance Switching and Routing (HPSR)*, pp. 1–6, IEEE, 2020.

[5] B.-A. Yassour, M. Ben-Yehuda, and O. Wasserman, "Direct device assignment for untrusted fully-virtualized virtual machines," 2008.

[6] E. Zhai, G. D. Cummings, and Y. Dong, "Live migration with pass-through device for linux VM," in *Proceedings of the Ottawa Linux Symposium (OLS)*, pp. 261–268, 2008.

[7] R. E. B. Asvija and M. B. Bijoy, "Security in hardware assisted virtualization for cloud computing - state of the art issues and challenges," *Computer Networks*, vol. 151, pp. 68–92, 2019.

[8] N. Har'El, A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger, and R. Ladelsky, "Efficient and scalable paravirtual i/o system.," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 231–242, USENIX Association, 2013.

[9] Y. Kuperman, E. Moscovici, J. Nider, R. Ladelsky, A. Gordon, and D. Tsafrir, "Paravirtual remote I/O," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 49–65, ACM, 2016.

[10] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, *et al.*, "Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization," in *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, USENIX Association, 2018.

[11] X. Zhang, X. Zheng, Z. Wang, H. Yang, Y. Shen, and X. Long, "High-density multi-tenant bare-metal cloud," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, p. 483–495, ACM, 2020.

[12] T. Zhang, L. Linguaglossa, M. Gallo, P. Giaccone, L. Iannone, and J. Roberts, "Comparing the performance of state-of-the-art software switches for NFV," in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies (CoNEXT)*, pp. 68–81, ACM, 2019.

[13] 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.

[14] "CVE-2018-1059." https://www.cvedetails.com/cve/CVE-2018-1059/, 2018.

[15] N. Amit, M. Ben-Yehuda, D. Tsafrir, and A. Schuster, "vIOMMU: efficient iommu emulation," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 73–86, 2011.

[16] P. Stewin and I. Bystrov, "Understanding DMA malware," in *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 21–41, Springer, 2012.

[17] P. X. Jason Wang, "Vhost and vIOMMU." http://www.linux-kvm.org/images/c/c5/03x07B-Peter_Xu_and_Wei_Xu-Vhost_with_Guest_vIOMMU.pdf, 2016.

[18] E. Auger, "vIOMMU/ARM: full emulation and virtio-iommu approaches." https://www.linux-kvm.org/images/8/8e/Viommu_arm.pdf, 2017.

[19] Intel, "Intel 64 and IA-32 architectures software developer's manual," *Volume 3A: System Programming Guide, Part*, vol. 1, no. 64, p. 64, 64.

[20] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska, "S-nfv: Securing nfv states by using sgx," in *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pp. 45–48, ACM, 2016.

[21] Y. Yang, H. Jiang, Y. Liang, Y. Wu, Y. Lv, X. Li, and G. Xie, "Isolation guarantee for efficient virtualized network i/o on cloud platform," in *Proceedings of the 22nd IEEE International Conference on High Performance Computing and Communications (HPCC)*, pp. 344–351, IEEE, 2020.

[22] QEMU, "QEMU: Open source processor emulator." https://wiki.qemu.org/Main_Page.

[23] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux symposium*, vol. 1, pp. 225–230, Dttawa, Dntorio, Canada, 2007.

[24] K. H. anel Bourguiba and G. Pujolle, "Packet aggregation based network I/O virtualization for cloud computing," *Computer Communications*, vol. 35, no. 3, pp. 309–319, 2012.

[25] L. Rizzo, G. Lettieri, and V. Maffione, "Speeding up packet I/O in virtual machines," in *Proceedings of the Symposium on Architecture for Networking and Communications Systems (ANCS)*, pp. 47–58, IEEE Computer Society, 2013.

[26] L. Rizzo, "netmap: A novel framework for fast packet I/O," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 101–112, USENIX Association, 2012.

[27] openvswitch document, "Dpdk vhost user ports." https://docs.openvswitch.org/en/latest/topics/dpdk/vhost-user/.

[28] J. Hwang, K. K. Ramakrishnan, and T. Wood, "Netvm: high performance and flexible networking using virtualization on commodity platforms," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 445–458, USENIX Association, 2014.

[29] A. C. Macdonell *et al.*, *Shared-memory optimizations for virtual machines*. University of Alberta Edmonton, Canada, 2011.

[30] C. MacDonell, "Nahanni, a shared memory interface for kvm." http://www.linux-kvm.org/images/e/e8/0.11.Nahanni-CamMacdonell.pdf, 2021.

[31] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pp. 459–473, USENIX Association, 2014.

[32] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The eucalyptus open-source cloud-computing system," in *Proceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, pp. 124–131, IEEE, 2009.

[33] "Open vswitch." http://www.openvswitch.org/.

[34] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, *et al.*, "The design and implementation of open vswitch," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 117–130, USENIX Association, 2015.

[35] "Openvswitch CVE vulnerability statistics." https://www.cvedetails.com/vendor/12098/Openvswitch.html, 2017.

[36] "CVE-2016-2074." https://www.cvedetails.com/cve/CVE-2016-2074/, 2016.

[37] S. Sreenivasamurthy and E. L. Miller, "SIVSHM: Secure inter-vm shared memory," Tech. Rep. UCSC-SSRC-16-01, University of California, Santa Cruz, May 2016.

[38] Y. Cheng, X. Ding, and R. H. Deng, “DriverGuard: Virtualization-based fine-grained protection on I/O flows,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 16, no. 2, p. 6, 2013.

[39] “CVE-2019-14835.” https://www.cvedetails.com/cve/CVE-2019-14835/, 2019.

[40] Z. Wei, J. Hwang, S. Rajagopalan, K. K. Ramakrishnan, and T. Wood, “Flurries: Countless fine-grained NFs for flexible per-flow customization,” in *Proceedings of the 12th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pp. 3–17, ACM, 2016.

[41] J. V. M. Runkai Yang, Xiaolin Chang and V. B. Misic, “Performance modeling of linux network system with open vswitch,” *Peer Peer Netw. Appl.*, vol. 13, no. 1, pp. 151–162, 2020.

[42] “Data plane development kit.” https://www.dpdk.org.

[43] J. Bouron, S. Chevalley, B. Lepers, W. Zwaenepoel, R. Gouicem, J. Lawall, G. Muller, and J. Sopena, “The battle of the schedulers: Freebsd ule vs. linux cfs,” tech. rep., 2018.

[44] Y. Ye, J. Haiyang, W. Yulei, L. Yilong, L. Xing, and X. Gaogang, “C2QoS: CPU-Cycle based network QoS strategy in vswitch of public cloud,” in *Proceedings of the 17th IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pp. 438–444, IFIP/IEEE, 2021.

[45] Y. Yang, H. Jiang, Y. Wu, C. Han, Y. Lv, X. Li, B. Yang, S. Fdida, and G. Xie, “C2QoS: Network QoS guarantee in vswitch through CPU-cycle management,” *Journal Systems Architecture*, vol. 116, p. 102148, 2021.

[46] “Spirent testcenter products.” https://www.spirent.com/products/testcenter.

[47] S. Bradner, “Benchmarking methodology for network interconnect devices,” *RFC2544*, 1999.

[48] T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy, “Guarantee IP lookup performance with FIB explosion,” in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pp. 39–50, ACM, 2014.

[49] “qperf.” https://pkgs.org/download/qperf.

[50] “Nginx.” http://nginx.org/.

[51] “iptables.” https://linux.die.net/man/8/iptables.

[52] W. Wang, “Design of vhost-pci.” http://www.linux-kvm.org/images/5/55/02x07A-Wei_Wang-Design_of-Vhost-pci.pdf, 2016.

[53] M. A. E. Jun Nakajima, “Scalable and high-performance virtual switching using pre-switch.” http://schd.ws/hosted_files/ons2016/36/Nakajima_and_Ergin_PreSwitch_final.pdf, 2016.

[54] W. Wang, “Github link of vhost-pci.” https://github.com/wei-w-wang/vhost-pci, 2017.

[55] A. Menon, A. L. Cox, and W. Zwaenepoel, “Optimizing network virtualization in xen,” in *Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 15–28, USENIX Association, 2006.

[56] J. R. Santos, Y. Turner, G. J. Janakiraman, and I. Pratt, “Bridging the gap between software and hardware techniques for i/o virtualization.,” in *Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 29–42, USENIX Association, 2008.

[57] K. K. Ram, J. R. Santos, and Y. Turner, “Redesigning xen’s memory sharing mechanism for safe and efficient I/O virtualization,” in *Proceedings of the 2nd Conference on I/O Virtualization*, pp. 1–1, USENIX Association, 2010.

[58] D. Wang, B. Hua, L. Lu, H. Zhu, and C. Liang, “Zcopy-vhost: Eliminating packet copying in virtual network I/O,” in *Proceedings of the 42nd IEEE Conference on Local Computer Networks (LCN)*, pp. 632–639, IEEE Computer Society, 2017.

[59] K. K. Ram, A. L. Cox, M. Chadha, and S. Rixner, “Hyper-switch: A scalable software virtual switching architecture,” in *Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 13–24, USENIX Association, 2013.

[60] 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,” in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 164–177, ACM, 2003.

[61] “TrustZone.” https://developer.arm.com/technologies/trustzone.

[62] P. Soyeon, L. Sangho, X. Wen, M. Hyungon, and K. Taesoo, “libmpk: Software abstraction for intel memory protection keys (intel MPK),” in *Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 241–254, USENIX Association, 2019.

[63] W. Tu, Y. Wei, G. Antichi, and B. Pfaff, “revisiting the open vswitch dataplane ten years later,” in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pp. 245–257, ACM, 2021.