Standardized but Flexible I/O for Self-Virtualizing Devices

Joshua LeVasseur Ramu Panayappan Espen Skoglund Christo du Toit
Leon Lynch Alex Ward Dulloor Rao[†] Rolf Neugebauer Derek McAuley

Netronome Systems [†]Georgia Tech
firstname.lastname@netronome.com *dulloor@gatech.edu*

Abstract

Moving device virtualization from the virtual machine monitor (VMM) to the devices improves virtual-machine performance significantly, but it requires support from the devices. PCI and PCI Express (PCIe) devices can provide VMs with direct and secure I/O through the use of multiple functions per card, but at significant cost and inflexibility. One solution to help reduce the costs is the PCIe SR-IOV standard, which introduces lightweight, virtual PCIe functions.

We are currently developing a highly configurable and programmable PCIe networking device which can change its behavior at runtime and which can provide a number of different types of device functions to the host system (e.g., standard NIC, specialized packet capturing devices, or crypto offload engines). We have found the PCIe SR-IOV standard to be too inflexible to support these types of devices, primarily due to its mechanism for configuring virtual functions.

In this paper we propose an alternative approach which does not require additional silicon and provides significantly higher flexibility than SR-IOV. We achieve this by delegating enumeration and configuration of "software configurable virtual functions" to the main device driver for the device. Our solution is compatible with the higher layers of the PCI device stack of modern operating systems and hypervisors so that we can leverage all the existing mechanisms for hot-plugging, discovering devices, loading device drivers, and assigning PCI devices to virtual machines (including providing DMA isolation with IO-MMUs). We present details of a prototype implementation for Linux and Xen.

1 Introduction

I/O virtualization, especially for network devices, incurs significant overheads when performed in software, resulting in lower performance or significantly higher resource utilization [21, 14, 20]. We observe three key technologies which help to reduce these overheads:

1. Self-virtualizing devices are being developed which move virtualization logic from the VMM to the device [9, 5, 6, 12, 25, 19]. In the case of networking they typically perform some form of filtering or even simple Ethernet bridging to de-multiplex incoming traffic to a number of queues, or host endpoints.

2. Several hypervisors allow PCI device functions to be assigned to different virtual machines by virtualizing the PCI configuration space and by giving VMs limited access to parts of the IO address space [13, 11, 7].

3. While a hypervisor can restrict which physical resources a VM has direct access to, it requires advanced IO-MMUs to protect and isolate the host and VMs from arbitrary accesses from the device [10, 4, 1, 2, 24]. Contemporary chipsets typically include such IO-MMUs, which determine based on a device function's PCI ID which host resources the device function has access to.

To tie these three technologies together the host endpoints of self-virtualizing devices need to appear as distinct PCI device functions to both the hypervisor and the IO-MMU. This allows the hypervisor to manage the endpoints using its standard mechanism and setup IO-MMU protection to isolate the host from devices.

SR-IOV [17] is one such mechanism to associate device endpoints with PCI function IDs. It provides an extension to the PCI configuration space enabling the enumeration of relatively lightweight virtual functions which share some configuration information with the physical function representing the actual device. Self-virtualizing devices with SR-IOV support represent their endpoints as virtual functions in the extended PCI configuration space.

In this paper we propose an alternative mechanism to associate device endpoints with PCI IDs. Our approach is motivated by a highly flexible and configurable network processing device which we are currently developing. SR-IOV proved to be too inflexible to support such a device and was relatively immature when we started developing the device. We make use of most of the PCIe features SR-IOV utilizes, however we require less support from the PCI device as host endpoint enumeration is performed in software (and not by the device). We call these host endpoints *software configurable virtual functions* and their enumeration is performed in a manner compatible with and transparent to the host OSes and hypervisors.

In the next section we briefly review SR-IOV and related PCIe technologies and motivate the need for a more flexible mechanism. We also recap PCI device function assignment in hypervisors. In Section 3 we provide details on our alternative approach and in Section 4 we briefly describe our implementation for Linux and Xen.

2 Background

2.1 SR-IOV and PCI Express

PCIe already provides a variety of technologies which SR-IOV builds upon. Standard PCI allows up to eight addressable physical functions per device (so-called multifunction devices). The Alternative Routing-ID Interpretation (ARI) [16] extends the number of addressable functions per device to 256 by merging the device and function ID bits in the PCI Requester ID to a single entity (the device ID is assumed to be 0).

SR-IOV also builds on the Address Translation Services [18], which provide an interface to a chipset's Address Translation and Protection Table (ATPT viz. IO-MMU) to translate DMA addresses to host addresses and defines a Address Translation Cache (ATC) for devices to cache these DMA translations.

Further relevant PCIe technologies are MSI-X for providing a large number of interrupts that can target different cores, interrupt remapping, access control services (ACS), and function-level reset (FLR).

SR-IOV builds upon these existing technologies by adding configuration and management of light weight virtual functions [17]. A physical device may have one or more physical functions (PFs), each of which may provide several virtual functions (VFs). A VF has a significantly simplified configuration space. All VFs associated with a PF share the device type of the PF, however, host software can largely treat them as normal PCI functions. For example, if a hypervisor allows assignment of PCI functions to VMs, it should be able to assign VFs to VMs without major modifications, including setting up IO-MMUs.

A simple SR-IOV device only provides a single PF through which its VFs are managed. We are developing a highly programmable and configurable network processing device which exposes a large number (currently 64) of endpoints to the host. We are currently developing endpoints which act as normal network interfaces, specialized network interfaces tuned for packet capturing, and different crypto endpoints. The type of a given host endpoint can be controlled by software and can be changed dynamically at runtime.

A simple SR-IOV device is not suitable to support such a device since all VFs have to have the same type as the PF they are associated with. From a host software perspective this means that the same device driver will be loaded for all VFs. With host endpoints of the diversity we are developing this is not feasible. An SR-IOV device with several PFs, e.g., one per endpoint type, would be able to accommodate the diversity of endpoints but would significantly increase the complexity of the device itself and the control software on the host. Further, it is unclear from the specification if SR-IOV allows moving VFs between PFs on the same device, i.e., to change the type of a host endpoint, and how complex such an operation would be. Enabling such support would certainly increase device complexity further.

In the Section 3 we describe an alternative approach which builds very much on the same PCIe technologies as SR-IOV, but provides more flexibility, satisfying the requirements for highly configurable devices. Our approach requires less support on the device itself. This flexibility is mainly achieved by looking at device virtualization from host software perspective, which we briefly recap next.

2.2 OS Support and PCI Device Assignment

In a typical system, an OS probes a bus to discover the peripherals installed in the machine, and uses a device-neutral algorithm to identify the devices for loading their respective drivers. PCI provides this via a configuration space that each PCI function participates within. Device probing may be repeated at run-time if the OS and bus support hot-plugging of devices.

In a virtual environment, a privileged entity, e.g., the hypervisor, a host OS for hosted VMMs [8], or a service OS like Xen's Dom0 [7], performs the device discovery. We refer to this entity as *OS/hypervisor* in the remainder of the paper to accommodate native OS execution and the different virtualization environments.

Hypervisors which support assigning PCI functions to VMs have to provide a virtual PCI (vPCI) bus for each VM so that the guest OS in the VM can discover the assigned PCI devices. Typically the following steps are performed:

1. A device is chosen from the PCI bus to be assigned to a VM.
2. The OS/hypervisor arranges for the device's interrupts to be forwarded to the VM.
3. The OS/hypervisor configures the IO-MMU to permit the device to access the VM's memory.
4. The OS/hypervisor attaches the device to a VM's vPCI bus.
5. The guest OS discovers the device on its vPCI bus, and loads the device's PCI driver.

The OS/hypervisor has to intercept all PCI configuration space accesses by the guest OS to the assigned device's configuration space as these represent privileged operations. Most of these access are typically passed through directly to the real configuration space, however, some accesses need to be emulated and modified.

3 Software Configurable VFs

For software configurable virtual functions we borrow heavily from the mechanisms used by a OS/hypervisor to assign physical PCI functions to VMs. As pointed out in the introduction, the endpoints of self-virtualizing devices need to appear to both the hypervisor and the IO-MMU as distinct PCI functions to enable safe hardware based I/O virtualization. SR-IOV achieves this by enumerating virtual functions within the device's configuration space.

Just like SR-IOV's PFs, in our approach a physical PCIe device offers one master or control function which manages a number of virtual functions. The control function is a real PCIe function implemented on the device with its own PCI configuration space and IO resources. However, virtual functions in our approach are only partially implemented on the device: they have device functionality, such as DMA and I/O registers, but they lack the PCI configuration records. Thus the virtual functions are initially hidden from the OS/hypervisor bus-probing logic. We call these virtual functions software configurable virtual functions to distinguish them from SR-IOVs hardware based VFs.

Probing of software configurable virtual functions is deferred until the OS/hypervisor has loaded the device driver for the control function. Once this control driver has been loaded, it assists the OS/hypervisor to enumerate the virtual functions. The control driver first initializes the device, optionally loading a firmware image. It then probes the device to discover configured virtual functions. Note that this discovery process is device specific and can be implemented in a manner suitable for the device. Finally, the control driver activates a *virtual configuration space* in which the virtual functions are enumerated. To the OS/hypervisor the virtual functions appear on the same bus as the control function and the OS/hypervisor can probe them using its existing PCI probe logic. Since the virtual configuration space is managed by the control driver it can place arbitrary virtual device functions in the configuration space, including functions with different device and vendor IDs as well as device types. This provides significantly more flexibility than SR-IOV's VF enumeration.

The virtual configuration space is a software representation of a standard PCI configuration space. The way it is activated is OS/hypervisor specific. However, most OSes provide either hooks for different PCI probing functions or a framework for implementing bus drivers, which is what some hypervisors exploit to implement the vPCI bus used to provide device assignment to VMs. The control driver may utilize the same hooks to activate the virtual configuration space. Once the OS/hypervisor can enumerate virtual functions it can apply its standard mechanisms for device assignment provided that the physical device and virtual functions behave in a certain way. We discuss the details in the following sections.

3.1 Real Device Behavior

Although the software configurable virtual functions lack physical configuration spaces, they still have to participate on the PCIe bus: they need to receive memory-mapped read and write operations, they need to initiate and terminate DMA requests, and they need to raise interrupts.

Read and Write Operations: We use I/O memory addresses to correlate read/write requests with virtual functions. The memory-mapped region of the physical device function is divided into page-aligned blocks assigned to the virtual functions. Since they are page-aligned, they can be assigned to separate VMs while preserving isolation. The host driver assigns the appropriate blocks to the virtual configuration spaces of the virtual functions, so that the OS can make those regions available to the drivers. The PCIe logic block on the card uses the address of the PCIe read/write operation to determine the target virtual function.

DMA: For DMA isolation, each virtual function needs a different PCIe routing number (i.e., a PCI *requester-id*), so that the OS/hypervisor can assign the virtual functions to VMs and prohibit the DMA of one VM from accessing the memory of another VM. Since the card initiates the

DMA requests, it can use any function number available to the card. PCIe permits a total of 256 function numbers via ARI and the physical function is assigned function number 0. Unlike SR-IOV the control driver can assign IDs arbitrarily to virtual functions. The control driver just needs to ensure consistency between the host side virtual configuration space and the assignment of IDs to virtual functions numbers on the device.

Interrupts: The control function has access to many interrupts through the use of PCI MSI-X. The control driver assigns the allocated interrupts to the virtual functions so that each virtual function can raise different interrupts. Interrupt assignment is communicated to the host via the virtual configuration space.

3.2 Virtual Device Behavior

Each virtual function has real device resources published to the OS through the virtual PCI configuration space. This includes the virtual function's memory mapped region, and its interrupt assignment. Additionally, if the OS/hypervisor changes the configuration spaces for the virtual functions, the control driver propagates necessary changes to the device. Normally a device reconfigures itself when it sees PCI configuration space accesses; instead, the control driver forwards the changes via a device-specific protocol.

Memory-mapped Region: Each virtual function has a dedicated block within the control function's memory-mapped region. This memory-mapped block is published within the virtual function's configuration space. While it is possible for the OS/hypervisor to request the device to relocate its memory-mapped region, the virtual function's memory region is immutable since it must reside within the region of the physical function.

Interrupt Assignment: Each virtual function has an MSI interrupt assigned to it. The interrupt could be unique or shared (sharing within a VM is preferable to sharing across VMs). The MSI interrupt configuration resides within the virtual configuration space for the virtual function. Thus, the control driver can intercept all MSI configuration attempts and map them to the real MSI-X interrupt configuration[1].

[1]Our current Xen implementation does not allow virtual function drivers to use MSIs since in Xen the MSI configuration is not passed through the control driver's virtual configuration space. Instead we configure MSI interrupt vectors in the control driver and pass the vector to the virtual function driver via the configuration space.

4 Implementation

We have developed an implementation of software configurable virtual functions for Linux and Xen [3]. Our development is motivated by the flexibility offered by the network processing device we are currently developing: the Netronome Flow Processor (NFP) 3200 (the successor for the Intel IXP network processor). It is a programmable, self-virtualizing device. The NFP 3200 provides 10-gigabit networking with typical network interface card (NIC) functionality, plus offloaded *nanonets* such as bridging for inter-VM communication, flow classification, routing, firewalling, etc. Additionally, the device can change virtual functions on the fly, e.g., providing an arbitrary mix of NICs, cryptography offload units, network bridges, and other device behavior. Thus the hardware requires the flexibility provided by the software configurable virtual functions outlined above.

4.1 Linux and Xen Device Models

Linux represents each device with an in-kernel abstraction, `struct device`, and a user-visible file-system abstraction, `sysfs`. The `sysfs` abstraction permits the user to configure or query the device. It also permits the user to unbind a device from one driver and to bind it to another.

Xen permits assigning devices directly to VMs, and does so by building upon Linux's device model: the user unbinds the device from its Linux driver in domain 0, and binds it to Xen's virtual PCI subsystem for use by Xen's VMs.

To assign a device's virtual function to a VM, the virtual function must have a dedicated `struct device` instance. Since software configurable virtual functions are enumerated on a PCI bus, Linux treats them as normal PCI device and creates a `struct device` for them. All further interactions the kernel has with a device, including driver loading, hot-plugging, and assignment to VMs, "just works".

4.2 Virtual Configuration Space

Linux, as other OSes, provides a flexible bus abstraction allowing device drivers to act as bus drivers. With PCIe each device has a dedicated bus [15]. Thus, in Linux the control driver which gets loaded when the kernel discovers the physical device function can act as a bus driver. All probing functions the kernel performs for devices on this bus are channelled through the control driver. The control driver can then easily enumerate the virtual functions and intercept configuration space accesses for virtual functions. Only when the kernel accesses the control function, they are forwarded directly to the device.

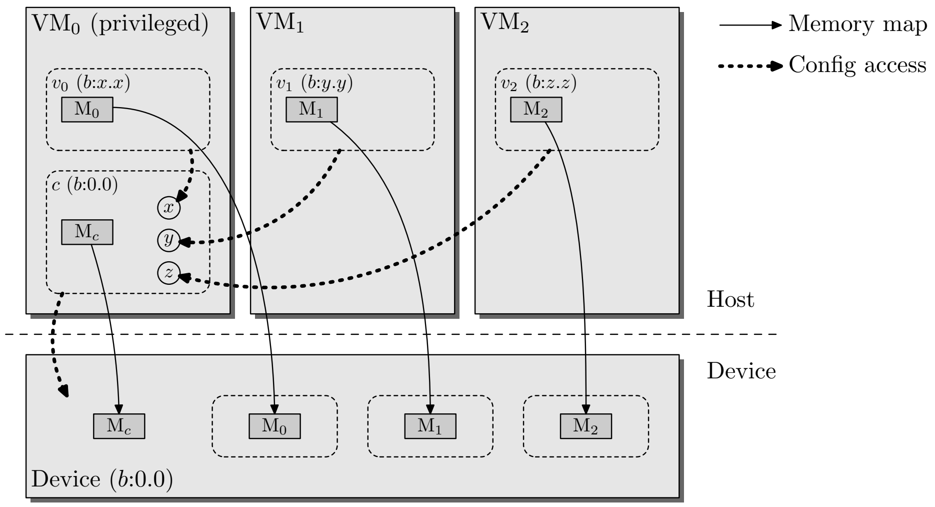

Figure 1: System setup with a privileged control driver (c) and three virtual functions (v_0, v_1, v_2). The control function has been assigned PCI device ID b:0.0, and the virtual functions are assigned device IDs on the same logical bus b. Each device driver has access to some device memory, but only the control driver can access the PCI configuration space of the device. Configuration space accesses from the virtual function drivers are forwarded to the control driver.

With Xen, VMs with assigned devices have the devices enumerated on a virtual PCI bus and configuration accesses to the assigned devices are forwarded to the PCI backend driver, typically to a Linux VM in Domain 0. If the device is enumerated in a virtual configuration space the PCI backend driver would thus automatically forward configuration accesses originating in a VM to the control driver. This scenario is depicted in Figure 1.

4.3 Discussion

As our implementation for Linux and Xen demonstrates, software configurable virtual functions can be supported by an operating system purely within the device driver for the control function, without requiring changes to the rest of the OS/hypervisor. However, it requires the OS to provide a flexible bus abstraction, allowing device drivers to provide bus functionality. While this is true for Linux and *BSD, we are currently investigating the applicability for other OS/hypervisors such as KVM [8] and VMWare ESX server [23], both of which generally support PCI device assignment. We would expect these systems to provide similar flexible device and bus abstractions partly because support for SR-IOV requires support for at least some of the OS/hypervisor features we exploit in the Linux/Xen implementation.

5 Related Work

SR-IOV [17] aims to improve PCIe's support for PCI device assignment to VMs. Like our solution, it builds upon many existing PCIe standards (e.g., address translation and protection). SR-IOV adds to the existing features by reducing the incremental hardware cost of additional functions by introducing virtual functions; and it supports more than 256 virtual functions without using a PCIe switch. Yet SR-IOV requires development of a new silicon logic block. It encodes configuration information within silicon which interferes with flexibility, particularly for programmable devices. The hardcoded configuration information permits a device-neutral driver to probe and discover SR-IOV's virtual functions — in contrast to our software configurable virtual functions — but many self-virtualized devices need a driver presence in the OS/hypervisor to manage global and privileged device resources. Further, software configurable virtual functions can be used alongside SR-IOV's virtual functions provided that the control driver allocates non-overlapping ranges for virtual functions.

Rather than use device assignment, Santos et al. [20] argue that software-based device virtualization has features worth preserving and show how to improve software-based network performance. They also demonstrates that Xen device assignment has lower performance than native I/O due to hypervisor overheads.

Several projects use paravirtual drivers for direct I/O with self-virtualizing devices [12, 9, 25, 19, 22]. It is the most flexible solution: like our solution, it works without SR-IOV support; unlike our solution, it does not require the hypervisor to support PCI device assignment. However, the use of paravirtual drivers requires development and maintenance of additional drivers for each hypervisor *and* guest OS environment, whereas software configurable devices can use the same device driver irrespective of the hypervisor used. Support for seamless VM migration between machines with different hardware configurations while offering accelerated IO access is often cited as another advantage of paravirtual drivers. However, NIC bonding accomplishes the same when using NIC direct I/O [5, 26].

6 Conclusion

In this paper we have proposed a more flexible alternative to SR-IOV which requires less hardware support from the device. The key insight is that SR-IOV makes use of many standard PCIe features and primarily adds a hardware based (virtual) function enumeration mechanism. Our software configurable virtual functions do not require such a hardware-based configuration space; it is implemented by a device driver. This software solution provides more flexibility as it does not have to conform to the restrictions imposed by SR-IOV. In particular, software configurable virtual functions can be enumerated with different device types, which is highly desirable for configurable and programmable PCI devices. We have argued and demonstrated with a sample implementation for Linux and Xen, that our alternative proposal can be implemented in software without requiring changes to the operating system and/or hypervisor.

References

[1] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel virtualization technology for directed I/O. *Intel Technology Journal*, 10(3):179–192, Aug. 2006. doi:10.1535/itj.1003.

[2] AMD. *AMD I/O Virtualization Technology (IOMMU) Specification*, Feb. 2007. Publication 34434.

[3] 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 System Principles*, pages 164–177, Bolton Landing, NY, Oct. 2003. doi:10.1145/945445.945462.

[4] M. Ben-Yehuda, J. Mason, O. Krieger, J. Xenidis, L. Van Doorn, A. Mallick, J. Nakajima, and E. Wahlig. Utilizing IOMMUs for virtualization in Linux and Xen. In *Proceedings of the Linux Symposium*, Ottawa, Ontario, Canada, July 2006.

[5] C. Bestler. Virtualized networking via direct function assignment. Presentation at the Xen Summit, Boston, MA, June 2008.

[6] S. Chinni. Reduce I/O bottlenecks with Intel virtualization technology for connectivity (Intel VT-c). Presentation at the Intel Developer Forum, San Francisco, CA, Aug. 2008.

[7] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, Boston, MA, Oct. 2004.

[8] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. `kvm`: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, Ottawa, Ontario, Canada, June 2007.

[9] G. Law. Accelerated Xen networking. Presentation at the Xen Summit, San Jose, CA, Sept. 2006.

[10] B. Leslie and G. Heiser. Towards untrusted device drivers. Technical Report UNSW-CSE-TR-0303, School of Computer Science and Engineering, UNSW, Mar. 2003.

[11] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 17–30, San Francisco, CA, Dec. 2004.

[12] J. Liu, W. Huang, B. Abali, and D. K. Panda. High performance VMM-bypass I/O in virtual machines. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, May 2006.

[13] S. E. Madnick and J. J. Donovan. *Operating Systems*, pages 549–563 (VM/370 Case Study). McGraw-Hill Book Company, 1974.

[14] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *First ACM/USENIX Conference on Virtual Execution Environments (VEE'05)*, Chicago, Illinois, June 2005.

[15] PCI-SIG. *PCI Express Base Specification Revision 2.0*, Dec. 2006.

[16] PCI-SIG. *Alternative Routing-ID Interpretation (ARI), PCI-SIG Engineering Change Notice*, June 2007.

[17] PCI-SIG. *Single Root I/O Virtualization and Sharing Specification, Revision 1.0*, Sept. 2007.

[18] PCI-SIG. *Address Translation Services*, Mar. 2008.

[19] H. Raj and K. Schwan. High performance and scalable I/O virtualization via self-virtualized devices. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing*, Monterey, CA, June 2007. doi:10.1145/1272366.1272390.

[20] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt. Bridging the gap between software and hardware techniques for I/O virtualization. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 2008.

[21] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 2001.

[22] S. Varley and H. Xu. I/O pass-through methodologies for mainstream virtualization usage. Presentation at the Intel Developer Forum, San Francisco, CA, Aug. 2008.

[23] C. Waldspurger. Memory resource management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 181–194, Boston, MA, Dec. 2002.

[24] P. Willmann, S. Rixner, and A. L. Cox. Protection strategies for direct access to virtualized I/O devices. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 2008.

[25] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel. Concurrent direct network access for virtual machine monitors. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, Phoenix, AZ, Feb. 2007. doi:10.1109/HPCA.2007.346208.

[26] E. Zhai, G. D. Cummings, and Y. Dong. Live migration with pass-through device for Linux VM. In *Proceedings of the Linux Symposium*, Ottawa, Ontario, Canada, July 2008.