Build an SR-IOV Hypervisor

Liang-Min Wang
Intel Corp
liang-min.wang@intel.com

Alex Zelezniak
AT&T Labs
alexz@att.com

E. Scott Daniels
AT&T Labs - Research
daniels@research.att.com

Timothy Miskell
Intel Corp
timothy.miskell@intel.com

Li-De Chen
Intel Corp
li-de.chen@intel.com

***Abstract*— IO virtualization is a key technology that enables SDN and NFV applications. There are two mainstream IO virtualization technologies that are used to implement a virtualized network device: VIRTIO and SR-IOV. The VIRTIO technology is a software virtualization technology that renders efficient IO virtualization based upon paravirtualization. In contrast, SR-IOV is a hardware assisted device virtualization that requires support from both the platform and IO devices. Although the packet processing throughput of a virtualized network device leveraging SR-IOV technology outperforms a device leveraging VIRTIO technology, there remains a lack of standardization amongst NIC device vendors with respect to the management of SR-IOV devices. The adoption of SR-IOV technology tends to be limited in embedded environments where ad hoc management routines are typically used. In this paper, we introduce the concept of an SR-IOV hypervisor to depict a trust management interface that can be used to manage SR-IOV devices over enterprise scale infrastructures such as the cloud. In addition, we present implementations of this new management interface through both user-space drivers (DPDK) and kernel-space drivers. Furthermore, for our user-space driver approach, we develop an innovative framework so that users can leverage existing legacy tools to manage SR-IOV devices. We demonstrate how an SR-IOV hypervisor can be deployed into a cloud environment, often containing a myriad of network devices and dynamic VM configurations, which can then be controlled via the management interface.**

***Index Terms*—SDN, NFV, VIRTIO, SR-IOV, DPDK, VFd, DPDM.**

I. Introduction

Software-Defined Networking (SDN) and Network Function Virtualization (NFV) have transformed how network resources are used over the course of the past decade [1-4], revolutionizing not only the traditional networking centric telecommunication industry, but also the manner in which computing and networking resources are deployed within the data center environment. Two key components of SDN and NFV are 1) the separation of the control plane from the traditional data forwarding plane and 2) network virtualization. Although SDN deployments can vary from one implementation to another [1], the essential requirement is a separated control plane, either centralized or distributed, that supports flexible network protocols, i.e. programmability. Kreutz et al. [1] have provided an extensive review of various approaches to realize SDN networks from inception to implementation. One aspect not fully explored as part of this study are IO virtualization technologies, which are key to network virtualization. This is not surprising, because most large-scale applications of network virtualization are implemented through software IO virtualization, e.g. Linux-based VIRTIO [16] and VMWare's VMXNET[17], which involves the adoption and implementation of a common software specification.

By contrast, IO virtualization capabilities and programming interfaces for hardware are vendor dependent. In this paper we focus upon Single-Root IO Virtualization (SR-IOV), a hardware-based IO virtualization technology [14-15][18], which is considered for fast-path IO virtualization, as compared to other software approaches [1][8][18]. Figure 1 presents the test results collected from a simple Virtual Network Function (VNF) running an L2 forwarding application between two VF ports. The VF ports are realized through either VIRTIO technology or SR-IOV. Figure 1(a) demonstrates the throughput generated by the same VNF for different packet sizes. Figure 1(b) shows hardware performance monitor unit (PMU) counters collected for each test [23].

As demonstrated over the test results, a significant throughput difference exists between device drivers based upon VIRTIO and SR-IOV technologies. In particular, for small sized packets, SR-IOV based implementations are shown to exhibit more than three times the throughput than VNFs based upon VIRTIO. The gap between these two IO virtualization technologies can be explained through the collected run-time PMU counters shown in Fig. 1(b), which reveal the key differentiators between the two IO technologies in terms of the number of read-write operations.

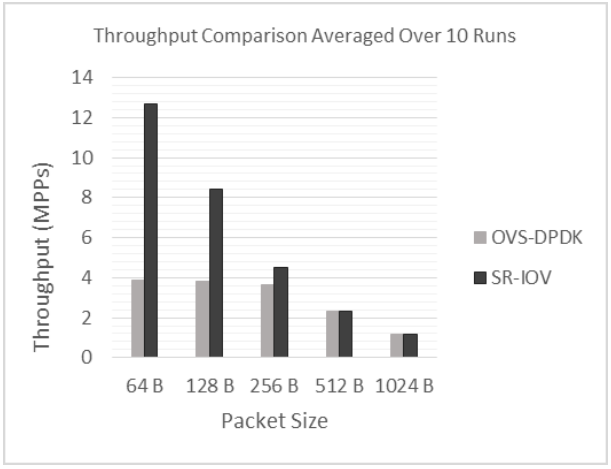
The differences in the PMU counters stems from the fact that, for SR-IOV deployments, guest-to-host memory translation is done in the hardware. In the case of VIRTIO, a backend driver must copy data from shared memory between the host and guest whenever data is forwarded from one VF port to the other. This observation underlines the importance of integrating SR-IOV technology with SDN/NFV implementations. As described, one drawback of integrating SR-IOV into large scale virtualized networks is the potential disparity of hardware implementations amongst NIC vendors. Therefore, a common interface designed to abstract the SR-IOV offloading capabilities of a NIC is a key step towards the standardization of low-level hardware programming. With a standardized interface, it would simplify integrations of SR-IOV devices into SDN based implementations, such as southbound interface described in [1].

978-3-903176-15-7 © 2019 IFIP

For the remainder of the paper, we begin by discussing the gap that exists between kernel network device management interfaces and provide a description of the SR-IOV management routines implemented throughout this work. Moreover, in Section II we describe an SR-IOV hypervisor by means of a VFd (VF daemon) [11] implementation. In Section III and IV we present the design for an SR-IOV hypervisor API realized through both user-space drivers as well as kernel drivers.

We begin this work with a popular open source network application design framework, the Data Plane Development Kit (DPDK). With the SR-IOV management routines provided by DPDK, we develop a Data Plane Device Management library (DPDM) along with a light-weight kernel module [24]. The DPDM framework enables DPDK applications to manage network devices through existing network management utilities such as `ethtool`, `ip`, `ifconfig`, etc. In addition, we describe an innovative encoding scheme allowing for the management of SR-IOV devices via `ethtool` commands. Upon realizing the SR-IOV hypervisor routines, we continue our work by presenting a means of standardization through a comprehensive, kernel sysfs based framework. This sysfs framework builds upon the work presented in [25] by extending the original design and providing a comprehensive specification, as described in [11].

We present a VFd use case in Section V. Subsequently, Section VI concludes the paper with identified challenges along with future work.



(a)Throughput

PMU	SR-IOV	VHOST/VIRTIO
mem_uops_retired.all_stores+ mem_uops_retired.all_loads	239	771/451
cycles	173	565/565

(b) Per Packet PMU

Figure 1 VIRTIO versus SR-IOV

II. VF NETWORK DEVICE MANAGEMENT

A. The Kernel Device Management Interface

Similar to most device drivers, conventional NIC drivers are either built into the kernel stack or are provided as loadable kernel modules, encapsulating hardware resources within the kernel space. As a result, only kernel space APIs can access device hardware resources, e.g. control and status registers, and manage target NIC devices. To constitute a secure interface, i.e. to restrict management routines to the kernel-space, and enable user-space applications to send requests to the respective kernel space device driver, a kernel driver needs to support the device driver `ioctl` mechanism [9 - 10]. As illustrated by Fig. 2, the `ioctl` mechanism enables a common application programming interface to manage NIC devices. All the device management operations are captured by three members of the `net_device` (netdev) data structure, namely `netdev_ops`, `ethtool_ops` and `fwd_ops`. A description of the `net_device` data structure and the associated network device driver can be found in chapter 17 of [9].

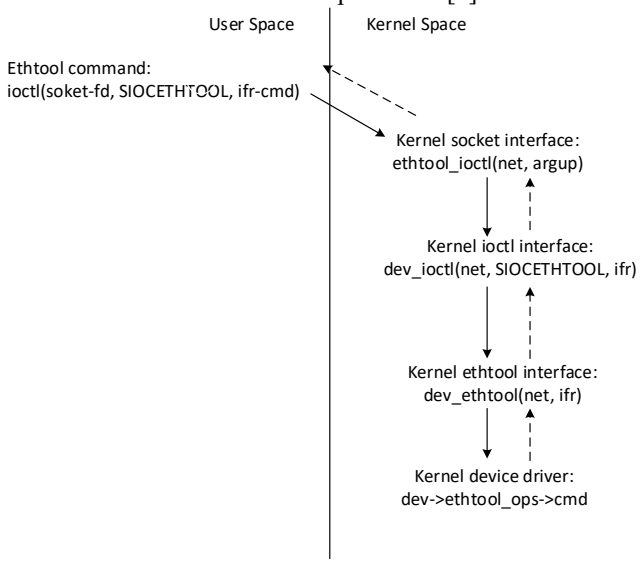


Figure 2 `ethtool` command request traversal between a user-space application and the kernel interfaces

As shown in Fig. 2, an `ethtool` command is passed through the kernel socket interface followed by the `ioctl` kernel interface, and eventually the request reaches the device driver's registered `ethtool_ops` callback function. This mechanism is equally applicable for the other two members, i.e. `netdev_ops` and `fwd_ops`, of the netdev interface. Each NIC kernel device driver can register its own device management routines through one of these netdev device management ops. As a result, a hardware abstraction layer is introduced such that the kernel network stack can share device management with common kernel-space/user-space utilities such as `ethtool`, `ip`, etc. Nevertheless, the existing design focuses on managing devices in its domain. Specifically, the interface provides Physical Function (PF) drivers to manage PF devices and Virtual Function (VF) drivers to manage VF devices, which misinterprets a key design principal of SR-IOV. The design of SR-IOV [14] calls for a single-root (PF) to manage multiple VFs. In SR-IOV, a PF device is capable of accessing all of the NIC resources while a VF device is constrained to limited access of a subset of network device resources. The design makes the Physical Function the master,

trusted device allowing the possibility for the PF device management routine to enforce different policies on each VF device. The existing netdev interface only enables limited VF management capabilities through `netdev_ops`, which at present does not adequately meet the demands of VNF management in the cloud. Therefore, we propose additional APIs to manage virtual functions through PF drivers. The proposed APIs are designed to address

- VLAN Management
- Tx Security and Rx Mode Management
- VF QoS
- VF Statistics

Additional APIs currently in development are also designed to address network traffic mirroring, power management, along with protocol header parsing offloading.

B. The SR-IOV Hypervisor

Given the lack of support from the kernel stack, we decided to implement our SR-IOV hypervisor by means of DPDK-based user-space drivers. Beginning with version 16.11, DPDK has supported new APIs to management VF configurations via the DPDK-based PF driver [11-12]. In the following sections, we provide a description of the SR-IOV hypervisor functions that have already been implemented along with functions that are currently in development.

B.1 VLAN management

As described in [1] [3], VLAN based network virtualization is one technology to partition network traffic and steer packet flows. Within this category, APIs that were implemented include management interfaces for VF Rx VLAN stripping, VF Rx VLAN filtering, VF Tx VLAN insertion and VF Rx traffic steering, i.e. filtering and traffic distribution via Rx queue assignment. This capability allows a control plane application to manage each VF device as an independent hub, which in turn can be used to manage different traffic flows by means of vlan-tagging.

B.2 Tx Security and Rx Mode Management

The VF management APIs within this category are designed to address packet spoofing, wherein a "malicious" VF might impersonate different VLAN IDs or MAC addresses when transmitting packets. The configuration APIs provide both MAC and VLAN anti-spoofing capabilities. As for SR-IOV packet receiving interface, the SR-IOV hypervisor adds support of the same PF Rx mode configuration (such as broadcast, untag, multicast etc.) management.

B.3 VF QoS

VF QoS management is one of the few VF management interfaces supported by the kernel netdev interface and is available via the Linux `ip` command. Specifically, there are four VF settings available: VF MAC-address, VF VLAN tag, VF VLAN QoS and VF Tx rate. The SR-IOV hypervisor provides an extension to what has already been defined in the netdev interface. For example, the existing netdev interface only allows a single VLAN tag to be assigned to a VF device, which is inadequate for most cloud applications. This work provides an API to allow multiple VLAN tag assignments to the same VF.

B.4 VF Queue Management, Statistics and Other Features

The SR-IOV hypervisor interface supports APIs to allow the PF to manage VF Rx/Tx queues. With this functionality, the network manager can enable and disable packet traversal into/out of a virtual device. Other VF management APIs include the ability to query statistics from VF devices, and register handlers for specific VF related events. The latter feature allows the SR-IOV hypervisor to manage VF devices when a critical event occurs.

B.5 Features in Development

Aside from the aforementioned APIs, there are device management features made available by DPDK that can be integrated into the SR-IOV hypervisor design. Among them are two important management features: statistics based power/frequency management and packet header parsing offloading. The x86 architectures provide CPU P-state and turbo-state [20] adjustments which enable applications to adjust CPU frequency for throughput and power consumption. Based on collected statistics, the controller can apply policies in regarding to observed run-time traffic (Rx receiving rate calculated from VF statistics), e.g. increasing core frequency to boost packet processing performance or dropping core frequency to conserve power consumption. Another interesting NIC feature is the dynamic device personality (DDP) [22], which allows run-time update of NIC firmware to support new packet types. As indicated in [6], packet header parsing contributes up to 58% of overhead when implementing a P4 [7] based packet forwarding application. Although this feature is not universally available from all device vendors, it can be significant as headers parsing has become increasingly cumbersome [1].

III. SR-IOV HYPERVISOR VIA USER-SPACE DRIVER

As described, the proposed SR-IOV hypervisor functions are not supported in the existing kernel stack, the initial work is therefore done through DPDK APIs. As described in [11], we implemented VFd which includes a parentless DPDK PF driver process, i.e. a process daemon, along with a set of configuration utilities. When combined, VFd can be used for managing VF configurations in cloud applications. In this section, we describe our attempt to bridge the gap between DPDK user-space device drivers and the kernel netdev interface. This effort began two years ago in an attempt to create a netdev equivalent interface in DPDK[1]. The goal was to create an interface that requires minimal changes to the device management implementation of existing network applications, thus allowing legacy applications to run device management either through kernel driver or DPDK device drivers. Given that the initial set of APIs were not all accepted

[1] http://dpdk.org/dev/patchwork/6564

by the DPDK community, a re-architecture was done to support comprehensive netdev integration. Three NIC PF/VF usage models were analyzed to capture the gap and the need to optimize running both kernel and user-space drivers.

- Bifurcated driver model (BFD): A bifurcated driver approach involves running both the kernel and user-space DPDK drivers. This mixed mode configuration either deploys drivers based upon PF/VF interface or leverages flow-director to separate flows [19]. In this partition, the kernel driver is used for device management, and user-space driver is used to run VF device or selected flow so maximum data throughput can be achieved.
- DPDK Kernel NIC interface model (KNI): The existing DPDK libraries [5] support a set of functions that enable a non-binding kernel module[2] to respond to `ioctl` and `ethtool` requests. As part of this model, the KNI kernel module implements the `ethtool_ops` data structure required by legacy applications which are based upon `ethtool` APIs. This configuration allows legacy utilities, e.g. `ethtool`, `ip`, and `ifconfig`, to work seamlessly with user-space DPDK applications. The drawback of this approach is that all the device drivers that support DPDK libraries must implement a separate set of driver functions in the kernel space. This non-trivial effort would result in duplication of driver functions in the kernel and user-space.
- User-space `ethtool` (USE) API with a proxy kernel module: In this execution model, a new set of device ops is introduced in the user-space driver to support existing netdev interface and a kernel module is devised to serve as a conduit to forward requests from kernel to user-space device drivers. To support this design, an attempt was previously initiated to modify DPDK KNI interface so that the kernel implementation for netdev was removed and replaced by a proxy kernel module[3]. Due to concerns described within the aforementioned link, this effort was put on hold.

Data Plane Device Management

Given the aforementioned development history, we employ a user-space device driver extension methodology. Specifically, a set of user-space drivers are created and attached to existing DPDK device drivers. With this design approach, we implemented DPDM (Data Plane Device Management) [24] libraries without the need to modify the DPDK source code. Furthermore, by incorporating DPDK API revision detection, DPDM has been shown to work with multiple versions of DPDK[4]. The major components of DPDM include:

- A new set of high-level `rte_eth` APIs, created to provide equivalent functions as defined in `ethtool_ops` (`linux/ethtool.h`), and `netdevice_op` (`linux/netdevice.h`).
- A user-space netlink client library.
- A kernel module (VNI) that supports both the netlink and netdev kernel interface.

Figure 3 presents the architectural diagram for DPDM. The communication between the kernel module, VNI, and the user-space libraries is based upon the kernel netlink interface. The kernel netlink interface supports data transferred between the kernel module and the user-space application through a socket. The user-space netlink client interface provides an API that allows an application to register virtual netdev interfaces via packets delivered from the user-space to the VNI. Similar to the kernel device driver, each port can register one interface name through the kernel netdev device registration process. In order to enable each virtual interface to be connected with its proxy driver, i.e. the kernel driver that owns the device, e.g. `igb_uio` or `vfio-pci`, DPDM includes bus information as part of the netlink client data structure during interface registration. Once the virtual netdev devices are registered by VNI, a subsequent request for a netdev op on the virtual netdev interface will be re-directed to VNI. The VNI will then pass the requests through the netlink interface to the user-space netlink client. The netlink client parses the request packet and invokes the respective user-space netdev API via the extended device drivers. The result along with the data is then returned to the kernel through the netlink interface. To support requests from the kernel netlink server, a user-space netlink client need to listen to a dedicated netlink socket as prescribed by kernel netlink protocol. Over DPDM design, such complicated socket communications is simplified through a session open/close API design that can easily be adopted by any network management application [24].

As mentioned in prior sections, due to lack of support of SR-IOV hypervisor functionality, we decided to extend the existing `ethtool –set-priv-flags` option through an innovative encoding scheme. Essentially, this `ethtool` option allows each device driver to define up to 32 one-shot features[5] and to define its own feature string(s). The definition of each bit is private to each NIC device driver, i.e. different NIC device driver may have different function definition of each 32 *priv-flags* bit. With the inherent driver automacy from this `ethtool` option, we extend support of this `ethtool` feature by employing a variable length API encoding scheme. By means of a 32-bit encoding scheme as described in Table 1, it can describe the SR-IOV hypervisor functions through `ethtool –set-priv-flags` feature.

[2] Unlike kernel-space drivers, the KNI is not bound to the NIC as the device driver. With KNI, the target NIC device is bound by a kernel proxy driver, e.g. igb_uio, and the kernel module serves as a conduit for the kernel space API to pass requests from the ioctl system call to the target NIC device.

[3] http://dpdk.org/dev/patchwork/patch/10130

[4] At the time of writing, DPDM has been tested with DPDK 16.11 through the latest release (18.11).

[5] A one-shot feature is a feature that can be either enabled or disabled, and there is no additional parameter can be specified

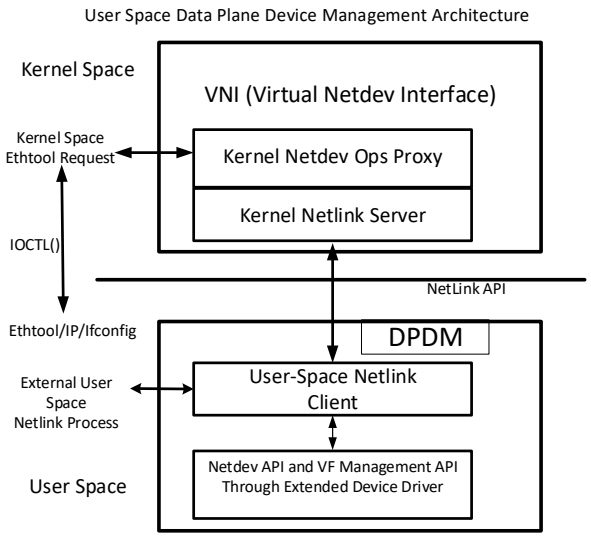


Figure 3 DPDM Architectural Diagram

Priv-Flags Bits	Descriptions
31	Enable VFd API Encoding
30:24	Denote vf index (0..127)
23:21	OP-Code: Seven categories defined: 0: VF feature bit set/reset 1: VF Rx mode management 2: VF statistics 3: VF Rx VLAN filter 4: VF Tx VLAN insertion 5: VF bandwidth (QoS) 6: VF queue bandwidth
20:18	Code sequence (for multi-step ops)
15:0	Feature bits, Rx mode features, 16-bit VLAN

Table 1 VFd API ethtool Encoding

With this encoding scheme, we are able to extend the existing netdev interface to support all SR-IOV hypervisor management routines implemented over DPDK.

IV. SR-IOV HYPERVISOR VIA KERNEL DRIVER

As mentioned the existing kernel netdev interface has limited support of VF management. Therefore, we extended the design in [25], which leverages a sysfs file hierarchy of the form as shown below to provide the ability to program SR-IOV configurations from simple shell scripts.

```
/sys/class/net/<pf-interface>/
device/sriov/<vf-index>/<SysFs File>
```

Table 2 lists the first implementation of this extension over Intel® i40e device [26]. The complete list of all features can be found in [11]

SysFs File	Description
vlan_mirror	List of VLANs to mirror to this VF
trunk	List of VLANs to filter on
allow_untagged	Enable/disable untagged packets
ingress_mirror	Mirror traffic of the specified VFs to this VF
mac_anti_spoof	Enable/disable MAC anti spoofing
vlan_anti_spoof	Enable/disable VLAN anti spoofing
loopback	Enable/disable Tx loopback
mac	Default MAC if not set use random
mac_list	List of additional MACs
promisc	Enable/disable unicast/multicast promisc.
allow_bcast	Enable/disable broadcast
vlan_strip	Enable/disable VLAN stripping
max_tx_rate	Maximum Tx bandwidth
Stats	Rx/Tx statistics

Table 2 VFd API Sysfs Encoding

By means of this interface, we are able to build an SR-IOV hypervisor through network device kernel drivers. However, two important tasks lie ahead. The first requires an increased adoption rate of the sysfs interface by vendors. The second, and more important task, requires OS vendors to incorporate support of the sysfs interface into the in-tree drivers, or by simply adopting this interface as a kernel network device management interface [11][25-26].

V. VFD USE CASE

With a small set of patches, Openstack Neutron is being used to define settings and policy information for VNFs, then communicates these to VFd for actual NIC configuration. Figure 4 illustrates the relationship between Openstack and VFd showing that the configuration for each guest is passed to VFd which pushes the configuration to the NIC. VFd also reacts to *callback* requests from the PF.

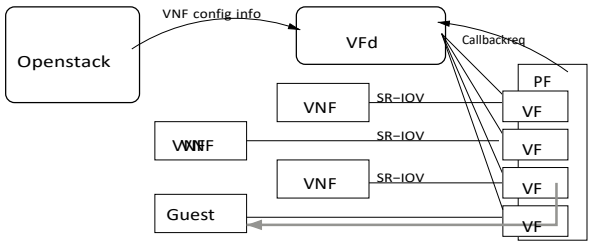


Figure 4 Relationship between Openstack and VFd.

Debugging packet flows for VNFs is often difficult. VFd is being used to configure VF mirrors such that packets to and/or from a VNF are mirrored to a second VF (heavy grey line) allowing traditional analysis tools to be used in the guest.

VI. CONCLUSION

In this paper we presented a set of APIs that provide an SR-IOV device management interface for the control plane within a trusted host environment, and develop this interface into a hypervisor. With the API adoption by DPDK, we are able to create two frameworks: VFd and DPDM. Furthermore, we intend to surface our VF management routines through a sysfs

file structure. This work allows us to explore its application in cloud environments, e.g. Openstack[11]. Based upon results shown in Fig. 1, we believe such an interface can provide a fast-path for most VNFs. However, PCI-SIG SR-IOV may be inadequate for large numbers of VFs, since the existing technology supports at most 256 instances [14]. Although the specification can be modified to extend this number, we foresee that additional VFs will eventually be subject to physical resource limitations. A hybrid deployment of VIRTIO and SR-IOV devices is inevitable. There are two scenarios where both devices can coexist: horizontal expansion or hierarchical expansion. For a horizontal expansion setup, the same PF device can manage SR-IOV instances and serve as a vhost driver for VIRTIO devices. For hierarchical expansion, SR-IOV devices serve as the vhost driver in order to manage multiple VIRTIO devices. Given that the use of nested VMs has not yet been popularized, over the hierarchy expansion use case, SR-IOV devices are primarily deployed in the host environment. One such use case can be found in [22] where SR-IOV is used as an OVS vhost driver for VIRTIO devices, and the proposed SR-IOV hypervisor API can work as part of integrated host management system instead.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable comments on this paper. The authors also wish to show their gratitude to the DPDK team for their support in developing an SR-IOV hypervisor API.

REFERENCES

[1] D. Kreutz, et al., Software-Defined Networking: A Comprehensive Survey, Proceedings of the IEEE, vol. 103, no. 1, Jan 2015.

[2] K. Greene, TR10: Software-Defined Networking, MIT Technology Review 2009,http://www2.technologyreview.com/news/412194/tr10-software-defined-networking.

[3] N. McKeown et al., OpenFlow: Enabling innovation in campus networks, ACM SIGCOMM Comput. Comm. Rev., vol. 38, no. 2, Apr. 2008, pp 69 – 74.

[4] S. Sezer et al., Are We Ready for SDN? Implementation Challenges for Software-Defined Networks, IEEE Comm. Mag., July 2013, pp 36 – 43.

[5] DPDK: Data Plane Development Kit. http://dpdk.org.

[6] P. Li et al., BMAcc: Accelerating P4 Based Data Plane with DPDK, DPDK summit 2017, https://www.slideshare.net/LF_DPDK/lfdpdk17accelerating-p4based-dataplane-with-dpdk

[7] P4, https://p4.org.

[8] Intel, Intel 82576 SR-IOV Driver Companion Guide: Overview of SR-IOV Driver Implementation, 322192-001 revision 1.0, June 2009.

[9] J. Corbet, A. Rubini and G. Kroah-Hartman, Linux Device Driver, 3rd Edition, 2005, O'reilly Media, Inc.

[10] D. P. Bovet and M. Cesati, Understanding the Linux Kernel, 3rd Edision, Oreiley, 2006.

[11] VFd: Virtual Function Daemon. https://github.com/att/vfd.

[12] A. Zelezniak, Implementing an SR-IOV Hypervisor using DPDK, DPDK Summit 2017, https://dpdksummit.com/us/en/past-events.

[13]] E. S. Daniels, Reflections on Mirroring with DPDK, DPDK Summit 2017, https://www.slideshare.net/LF_DPDK/lfdpdk17reflections-on-mirroring-with-dpdk.

[14] PCI-SIG. SR-IOV Specification, https://pcisig.com/sites/default/files/specification_documents/ECN_SR-IOV_Table_Updates_16-June-2016.pdf

[15] Y. Dong, Z. Yu and G. Rose, SR-IOV Networking in Xen: Architecture, Design and Implementation, https://www.usenix.org/legacy/event/wiov08/tech/full_papers/dong/dong.pdf

[16] OASIS Virtual I/O Device TC, Virtual I/O Device (VIRTIO) Version 1.0, http://docs.oasis-open.org/virtio/virtio/v1.0/csprd02/virtio-v1.0-csprd02.pdf

[17] VMWare, Best Practices for Performance Tuning of Latency-Sensitive Workloads in vSphere VMs, VMWare Tech. Paper, https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/vmw-tuning-latency-sensitive-workloads-white-paper.pdf

[18] P. Kutch and B. Johnson, SR-IOV for NFV Solutions: Practical Consideration and Thoughts, Intel® Tech. Brief, Feb. 2017, https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/sr-iov-nfv-tech-brief.pdf.

[19] Getting the Best of Both Worlds with Queue Splitting (Bifurcated Driver), http://rhelblog.redhat.com/2015/10/02/getting-the-best-of-both-worlds-with-queue-splitting-bifurcated-driver.

[20] T. Kidd, Power Management States: P-States, C-States, and Package C-States, Intel Developer Zone, https://software.intel.com/en-us/articles/power-management-states-p-states-c-states-and-package-c-states

[21] S. Su et al., Low overhead VM-to-VM transmission for service function chaining, IEEE NFV-SDN, Nov. 2016. Demo track. http://nfvsdn2016.ieee-nfvsdn.org/program/demonstrations

[22] A. Chilikin and B. Johnson, Flexible and Extensible support for new protocol processing with DPDK using Dynamic Device Personalization, DPDK Summit 2017.

[23] Performance Monitoring Unit, Intel® 64 and IA-32 Architectures Software Developer's Manual, vol. 3, chap. 18.

[24] DPDM, https://github.com/intel/dpdm_lib

[25] Mellanox, MLNX_EN for Linux User Manual, http://www.mellanox.com/related-doces/prod_software/Mellanox_EN_for_Linux_User_Manual_v4_4.pdf

[26] https://sourceforge.net/projects/e1000/files/i40e%20stable/2.7.12