In-Kernel Fast Path Performance For Containers Running Telecom Workloads

Nishanth Shyamkumar, Piotr Raczynski, Dave Cremins, Michal Kubiak, Ashok Sunder Rajan
Intel Corporation
Oregon, United States; Gdansk, Poland; Shannon, Ireland

Abstract

Virtualization of Telecommunication workloads opens the door to flexible and resource efficient deployment in a cloud infrastructure, for Telecom operators. The layers of virtualization involved in containers while providing advantages, also present a bottleneck for network functions requiring high performance. Relying on third party data-path libraries as a solution result in increased external dependencies, that affect ease of integration at the orchestration layer, and loss of Linux native debugging and diagnostic support.

In this paper, we explore and make the case to use an in-kernel based network pipeline for a 5G User Plane Function, that reduces the CPU core count from 10 to 1 for processing 2 Million Packets Per Second. This is accomplished by implementing AF_XDP zero copy support in an SR-IOV Virtual Function driver. We demonstrate performance comparisons with alternate in-kernel data path mechanisms as part of the evaluation. The proposed solution maintains performance at par with third party poll mode drivers, but importantly maintains operability with Linux tools, scalability and allows Kubernetes orchestration of the User Plane Function running in pods.

Keywords

AF_XDP, Zero copy, SR-IOV, UPF, Containers, In-kernel, Namespace, Telecom, Throughput, Cloud-native

Introduction

Telecommunication(Telecom) workloads are shifting from running on Fixed Function appliances, where the proprietary hardware and software are closely integrated and owned by a vendor, to appliances based on Network Function Virtualization. This allows for utilising commodity hardware to execute the Telecom workload while reducing cost, improving flexibility and openness.

Containers are at the forefront in achieving these cloud principles of workload flexibility, quick deployment, scalability, isolation and efficient resource utilization.

The flexibility from network virtualization comes with a trade-off, as these solutions increase CPU footprint i.e. increasing the number of CPUs required to achieve fast path performance. In a cloud deployment of UPF, the acceptable fast path performance of 2 Million Packets Per Second(MPPS) throughput is achieved using 10 cores [7]. The performance is limited because the data plane packet traverses the host namespace Linux network and container namespace Linux network stack, thus lengthening the Network IO path. The penalty incurred from additional CPU processing reduces bandwidth and increases latency for the user application. As a result, cloud-nativity imposes a high Total Cost Of Ownership(TCO) as more resources need to be reserved to achieve optimal performance.

In this paper, we combine Hardware Network Virtualization technology (SR-IOV) ; and an in-kernel Network bypass mechanism (AF_XDP) , for a container application to effectively utlilise Network IO at 10x lower TCO. The end result is that we achieve 2MPPS performance using 1 core, while also adhering to the cloud principles mentioned above.

The paper is structured into the following sections: 1) Delve into the network performance bottlenecks when utilizing containers 2) Description of the SR-IOV Hardware Virtualization technology 3) Explanation of AF_XDP zero-copy kernel network bypass 4) A brief characterization of the Telecom workload 5) Introduce the Test Setup for the experiments 6) Observation of the values measured and we conclude with a discussion on future work.

Network Performance Bottlenecks

Container sandboxing is managed by the underlying kernel, which ensures resource limits and isolation. Part of this isolation includes the network namespace, which sandboxes the containers such that, they are provided their own network stack and are unaware of any network devices that have not been assigned to them.

The underlying host system uses the root network namespace(also called default namespace), and each container is assigned it's own network namespace [5]. The network namespace allows for isolation and scalability, as each container can be assigned a virtual interface that can be used for communicating with other containers and external hosts. These interfaces can be quickly brought up when a container is started and removed as it is destroyed. To achieve this, the network needs to be virtualized. Network virtualization can be achieved through software or hardware.

A few examples of the network virtualization implementations in software are L2 bridge, MACVLAN, IPVLAN or OpenVSwitch as illustrated in Figure 1. Different Network plugins in Kubernetes such as Cilium, Weave and Calico,

take a hybrid approach. They use L2 bridges/ L3 Routing for intra-host communication and overlays such as VxLAN and IP-over-IP for inter-host communication [10]. These suffer from processing overheads and cannot provide fast path performance for a Telecom workload.

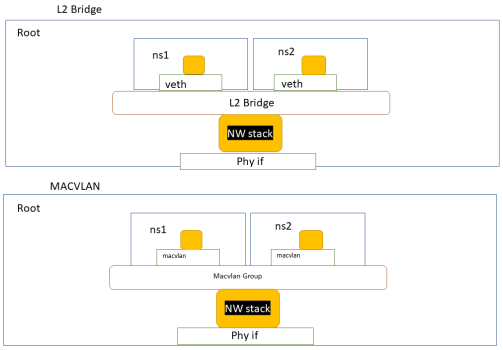

Figure 1: Packet flow from physical to virtual interfaces in an L2 bridge and MACVLAN bridge.

The network virtualization solutions previously mentioned create CPU overhead, due to Netfilter rule matching, Bridge lookup table matching and encapsulation/decapsulation operations. The lengthened Network IO stack that the packet now traverses result in CPU cycles spent in moving packet data, rather than processing the packet payload. This reduces bandwidth and increases latency for the user application.

Secondly, once the container virtual interface receives the packet after the bridging operation, it is transferred to be processed by the standard Linux kernel Network Stack.

Enabling in-kernel Fast Path Performance

Leveraging Hardware Network Virtualization technologies on Network Interface cards(NIC) , such as SR-IOV, allows the negation of the network virtualization logic running in software.

With hardware IO virtualization techniques, the container interface can directly access the NIC resources without a software bridge, eliminating CPU cycles which would have otherwise been spent on parsing and routing packets to the appropriate container interface.

SR-IOV

Single Root I/O Virtualization is a network virtualization solution in hardware that is standardized under the PCI-SIG standard [12].

SR-IOV permits the grouping of a set of network resources in hardware, i.e hardware queues, interrupts and PCI configuration space. This umbrella of resources is called a Virtual Function(VF), which are then assigned to a container, and accessed via a dedicated IO driver. Since the VF is a slice of the network card resource, the number of VF entities determines an upper bound on scalability.

Network Packets arriving on the NIC port are routed to the queues associated with a VF, through layers of filtering and routing present within the NIC hardware pipeline. A few examples of these on an Intel E810-XXVDA4 NIC are L2 MAC filters, 5-tuple filters, Receive Side Scaling and Flow-director. In our test setup, we use Ethernet Flow Director to steer fast path packets to the appropriate hardware queue of the VF.

AF_XDP

AF_XDP is a high-performant network data path mechanism, that is natively supported by the Linux Kernel. It is built on top of XDP and used to direct packets to a user space application [2].

Express Data Path(XDP) is a hook point within the Kernel Network Stack, where an extended Berkley Packet Filter(eBPF) program logic can be executed on a packet. eBPF are programs that run in a privileged kernel context within a sandboxed environment. These programs are verified by the kernel and allow for flexible and programmable logic to run in the kernel at specific hook points [3]. Two modes of support for XDP are Native and SKB mode.

The Native mode requires the hardware network device driver to be XDP-Aware i.e, implement parts of the XDP mechanism in concert with the core Kernel. It increases complexity of the driver, but in return provides high throughput performance capabilities. SKB mode is used for the purpose of flexibility in order to run on any network device, even if its' device driver does not support XDP. In this mode, the XDP mechanism is delegated entirely to the Kernel. However, this mode is not useful as a high performance alternative to the traditional network stack.

From the userspace application point of view, a new socket address family type called AF_XDP is utilized. The socket descriptor binds itself to a single queue on a hardware device. The user-space application leveraging AF_XDP, registers a user space memory region called UMEM, with the Kernel. The UMEM is further divided into equal sized chunks of either 2KB or 4KB, called Frames, which are used to hold data packets and are the basic unit in an AF_XDP operation. The UMEM has a single Fill Ring and Completion Ring associated with it. Each AF_XDP socket that is created gets it's own Rx and Tx descriptor rings. Thus, the 4 descriptor rings house descriptors that can point to frames within the UMEM. They work together to synchronize and transfer control of UMEM frames between user space and kernel space [6].

AF_XDP Zero Copy

When implemented natively through the network device driver, there are two options to send the packet data to the user-space application.

The first option is to copy the packet byte-by-byte from kernel space to user space memory. This operation is CPU-intensive, and degrades the throughput performance.

The second option, is to enable zero copy of packets, where the network hardware device does a Direct Memory Access(DMA) operation of the packet, into and out of system memory representing the respective frame. This eliminates the need for the CPU to be involved in moving the packet into the user space buffer, freeing it to focus on application logic.

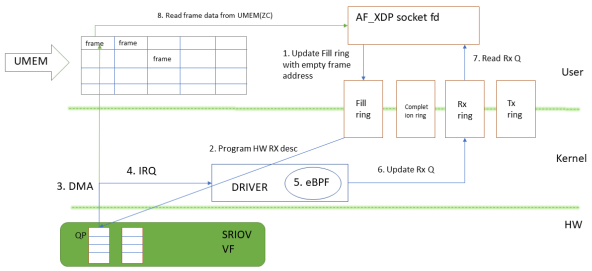

Figure 2: Steps showing the receive side of an application using AF_XDP Zero-copy mechanism.

In Figure 2, the sequence of steps involved in a zero-copy operation on the receive side are depicted. Here we explain the steps involved:

- The user space application opens an AF_XDP socket and binds it to queue 0 of the VF interface. The socket is registered with a UMEM area. The application is ready to receive packets and writes the address of the available empty frame, as an offset from the UMEM base, into the UMEM's Fill Ring descriptor. At this point, control of the frame is transferred to the Kernel.
- The Kernel reads the Fill Ring, converts the frame address into a PCI bus address and fills the HW RX descriptor for the VF queue 0.
- On packet arrival at queue 0, the NIC uses the information in the HW Rx descriptor and initiates a DMA operation to this address.
- The NIC notifies the CPU of packet transfer completion via an interrupt. The CPU schedules the network driver to run on the CPU.
- The driver references the packet and executes the XDP program on this packet in softirq context. A decision is made on the packet, if it is to be redirected to the user space application.
- If the redirect operation is valid, the driver fills in the offset address of the frame into the descriptor of the AF_XDP socket Rx ring. Control of the frame is now handed back to the user-space application.
- The user-space application reads the frame offset address from the AF_XDP Rx ring, dereferences the frame and accesses the packet data.

The two important highlights from an AFXDP_ZC operation are: The kernel space is in control of programming the Network hardware. This embraces the isolation provided by the kernel, improving system security. Secondly, the fast path application binds to a single queue pair. The other queue pairs on the VF device can then direct packets to the Linux Kernel network stack, which is especially useful for handling ICMP control packets such as ARP.

Telecom workload Characterization

The Network Function that is considered in our experiment is a User Plane Function(UPF) that runs in the core of the Telecom Network and provides the gateway to external networks such as the Internet. In Figure 3 , we observe that User Equipment(UE) such as mobile phones are connected to the cell tower/base station through an access network. The base station aggregates data from multiple end user devices and sends it over to the Telecom Core Network through wired infrastructure. This aggregated wired link uses GTP tunneling to communicate with the Mobile Core Control Plane and Data Plane [1]. The interface on the Serving Gateway(SGW) communicating with the base station is the S1u interface. On the other hand, the interface on the Packet Gateway(PGW) connecting to the internet is the SGi interface. In the transition to 5G architecture, the SGW and PGW have been combined into 1 entity called the UPF.

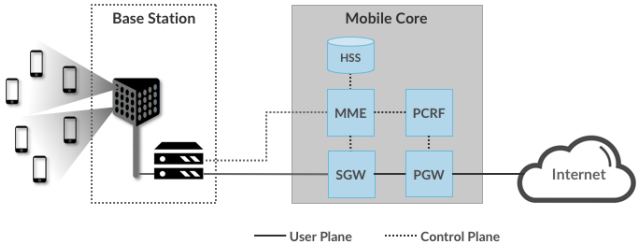

Figure 3: Telecom Network for 4G architecture.[9]

The UPF is divided into a Control Plane component(UPF-C) which takes care of session creation, IP address allocation etc. and a User Plane component(UPF-U) which manages the data path. This is part of the Control User Plane Separation(CUPS) design. The goal of the UPF core is to ensure each connection from an end user device to the Core network(a session flow) is properly managed. The Mobile Core control components(MME, HSS, PCRF) involves features such as mobility management, the authentication of users, ensuring the users data plan limits etc. The User Plane data plane(UPF-U) receives flow-based information from the control plane, and these details are stored as tables in memory.

The data plane component(UPF-U) for each session can be split into an Uplink flow(UL) and a Downlink flow(DL). The UL deals with packets originating at the UE and the DL covers packets that have the UE as the destination. In the UPF-U, for a UL flow, packets arrive at the S1U interface, undergo GTP header decapsulation, and are passed through multiple stages such as Policy Control and Charging, Application Detection and Control, Service Data Flow etc. The packet payload is matched on the values stored in these tables and if all matches succeed, the packet is allowed to exit to the internet through the SGi interface.

With the virtualization of Network Functions and the advent of 5G use cases, there is a shift in deployment strategy for Telecom functions. The Data plane components are disaggregated from the Mobile Core and moved closer to the base stations and customer premises, and is known as

Multi-Edge Access Computing(MEC). This allows for lower latency times and tighter control of locally generated data. The primary concerns for such deployments is having high throughput performance at low operational costs and low power consumption. Our test case is focused on such a scenario of a UPF VNF deployed at a MEC site servicing a load of 20Gbps.

Experimental Test Setup

The experimental test setup consists of a host designated as System Under Test(SUT), that runs the Telecom UPF network function . The Network Function program we use, is compliant with 3GPP Release 13.6 and maintains Control and User Plane Separation. For our test, we simulate the control plane by statically adding flow based information into the data plane. This is done so that our tests focus primarily on the data plane performance.

The UPF is deployed in a container using the Docker platform. For the L2 bridge, MACVLAN and IPVLAN groups, we use Docker's in built support for creating these networks and connecting container interfaces onto them. For SR-IOV use case, the VFs for each network port are created before deploying the UPF container. The VF interfaces are then moved into the container namespace using a software tool called 'Pipework' [8].

In Figure. 4, it can be seen that each UPF application utilizes 2 cores. Core 0 processes uplink packets received on Rx at the S1u interface(UL-Rx) and transmits packets out Tx at the SGi interface(UL-Tx). Symmetrically, Core 1 processes Rx at the SGi interface(DL-Rx), and transmits it to Tx of S1u interface(DL-Tx). Therefore, one UPF application processes both UL and DL flows simultaneously.

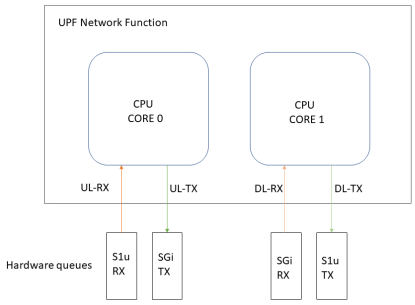

Figure 4: Flows handled by the cores in a UPF application.

The packets for UL and DL data paths are generated from a second host, designated as Traffic Generator. It uses a modified DPDK-Pktgen software which is capable of pushing packets at different rates and for multiple flows. In our test case, we are using 2.4 Million Packets Per Second as the packet rate with 8000 flows.

The SUT and the Traffic generator are directly connected with a 10G SFP+-SFP+ cable on each of the Network interface ports. The traffic generator simulates Access Network packets for the UL flow, while simulating Internet traffic for the DL flow. The Experimental Test Setup is illustrated in Figure 5.

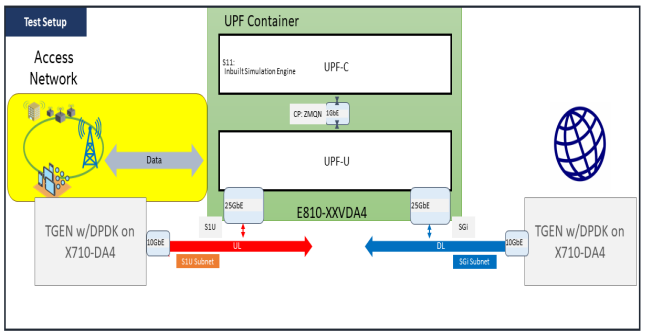

Figure 5: Experimental Test Setup diagram

Further details of the test setup are mentioned below:

Hardware

- SUT
  - CPU: Intel Xeon Platinum 8260M CPU @ 2.40GHz Dual Socket
  - Memory: 96GB/socket, DIMM, DDR4, 2933MT/s
  - NIC: Intel Ethernet Network Adapter E810-XXVDA4 4x25G
- Traffic Generator
  - CPU: Intel Xeon Platinum 8260M CPU @ 2.40GHz Dual Socket
  - Memory: 96GB/socket, DIMM, DDR4, 2933MT/s
  - NIC: Intel Ethernet Network Adapter X710-DA4 4x10G

Software

The two hosts are designated as SUT and Traffic Generator

- SUT
  - OS: Ubuntu 20.04.4 LTS
  - Kernel: 5.15.0-23-generic
  - Hugepages: 2MB
  - Docker version 20.10.14
  - iavf Out-Of-Tree Driver
- Traffic Generator
  - OS: Ubuntu 18.04.5 LTS
  - Kernel: 5.4.5-050405-generic
  - Hugepages: 2MB

The SUT has the following optimizations for performance improvement: 2MB Hugepages are reserved to reduce Translation Lookaside Buffer evictions, and the UPF application is pinned to it's own dedicated core. Both core pinning and hugepages respect the Non-Uniform Memory Access(NUMA) node alignment of the NIC port, which allow for faster main memory access times. As mentioned above, we use an Out-Of-Tree iavf driver with AF_XDP Zero Copy support built into it.

Empirical Observations

This section covers the throughput performance results that we achieved using the in-kernel fast path solution covered in this paper and we compare it to other in-kernel options.

The fields on the X-axis, are represented in a 'NW Virtualization + NW stack' format. So, for example, 'MACVLAN+Linux' implies that the briding of the container virtual interface to hardware resources is done by MACVLAN, and the packet processing is taken care of by the Linux Network Stack.

The unit of measurement for throughput on the Y-axis is 'Megabits per second per core' i.e, Mbps/core. The packet size used is 512 Bytes, the average packet size of a mixed payload, as seen on a Telecom infrastructure [11]. The experiments were run for 50seconds on each run, with pktgen generating at 2.4 MPPS. Each experiment was repeated 5 times and the results averaged out.

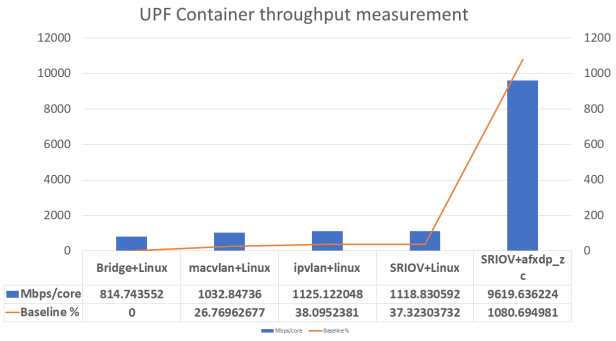

Figure 6: Single core throughput measurements of UPF container.

From the results we infer the following, that the 'L2 bridge + Linux network stack' is the least performant as it involves both Netfilter rule matching and a bridge lookup table. We make this our baseline and the performance improvements for other use cases are displayed as a percentage on the right sided Y-axis in Figure 6.

The 'MACVLAN + Linux' and 'IPVLAN + Linux' show performance improvements over the L2 bridge implementation, as Netfilter matching is not required for these solutions, however a routing table lookup is still needed. Using 'SR-IOV+Linux' to move network virtualization into the hardware increases the throughput to 1118 Mbps/core, but is limited by the Linux network stack, implying that the Linux Network Stack packet processing and data copying are the main causes of throughput throttling.

The performance of the 'SR-IOV+AF_XDP ZC' is at close to 10x of the other scenarios as a result of HW network virtualization and zero-copy mechanism. A single container running a UPF Network Function will transmit close to 20Gbps of data on the UL and DL combined, using 2 cores and 2 VFs. Compared to the baseline case where the same throughput is achieved using 20 cores, we can see a significant reduction in Operational Expenditure costs.

We would like to mention a few caveats about our results. As observed the UPF application utilises 2 cores to move 20Gbps of data, however there is an additional CPU overhead on non-application cores for softirq processing. When the network driver polls the AF_XDP rings(FillQ and TxQ) for new descriptors and doesn't find any entries, it reschedules the NAPI poll. This creates close to 100 percent CPU utilization for the softirq core. The problem can be alleviated by using 'XDP_USE_NEED_WAKEUP' flag when setting up the AF_XDP program. This flag stops the driver from polling the queues and instead entrusts the user space to notify the driver when packets are ready to be consumed by the driver. This greatly reduces the CPU cycles being spent idly thus reducing the load on non-application cores, but the measured values in this paper don't include this use case. Secondly, acceptable packet loss for Telecom providers varies based on Service Level Agreements. We measure 1.1MPPS per core throughput with zero packet loss. For speeds from 1.1MPPS to 2.4MPPS per core we observe less than 0.1 percent packet loss. We are currently exploring the reason for this consistently small packet loss, which may be due to the RX hardware descriptors not being replenished quickly enough, or losses at the Flow-director, Access Control List or possibly invalid queues.

Conclusion

Containers provide the infrastructure that enable cloud-native principles of scalability, isolation, flexibility and quick-deployment. However, for high throughput workloads that run on Telecom infrastructure, containers cannot achieve this bandwidth requirement without increasing the Total Cost of Ownership. Existing In-kernel mechanisms are limited in performance due to the overhead of the CPU cycles being spent in moving the packet from the wire into the application, and vice versa. Third party solutions such as DPDK impose external library dependencies and loss of Linux Kernel support which make integration, support and orchestration in a cloud deployment challenging.

In this paper we propose a solution that involves two principle ideas. The first, is utilizing hardware network virtualization in the form of SR-IOV to route packets to the container virtual interface in hardware, which frees CPU cycles, while maintaining scalability for containers. The second, is to use an in-kernel supported Network Stack bypass mechanism i.e, AF_XDP, running in Zero copy mode. The support for Zero copy comes from the Network driver for the SR-IOV VF. In AF_XDP, privileged operations such as HW queue programming are separated from user space and handled in kernel space. The XDP layer allows for additional programmability which can be exploited based on application requirements. The non-fast path packets can be received on other hardware

queues of the VF and processed by the Linux Kernel Network stack. This is especially useful for ARP resolution and other control packets that hit the interface as part of the ICMP protocol. Finally, we can leverage Linux tools and utilities during development for debugging and diagnostics purposes [4].

From our results, we are able to realize that our approach provides close to 10x throughput improvement per core compared to existing in-kernel options, and similar performance to third-party fast path solutions. Thus, we believe the solution can marry fast path performance with ease of system integration, due to it's Linux-native support.

Future work lies in integrating this solution into a Kubernetes framework using Multus and SR-IOV Container Networking Interface Plugins, and testing the scalability constraints while ensuring the performance profile remains consistent. The caveats mentioned in the previous section are also to be investigated and fixed. Other avenues of research include decomposition of the UPF Virtual Network Function into Micro-services bringing it closer to Cloud-native deployment, while reducing existing permissions/privileges needed for the container application, to enhance security.

References

[1] 3GPP. Ts 29.274 technical specification.

[2] Corbet, J. 2018. Accelerating networking with af_xdp. *lwn.net*.

[3] ebpf documentation. *ebpf.io/what-is-ebpf*.

[4] Høiland-Jørgensen, T.; Brouer, J. D.; Borkmann, D.; Fastabend, J.; Herbert, T.; Ahern, D.; and Miller, D. 2018. The express data path: Fast programmable packet processing in the operating system kernel. *CoNEXT '18* 54–66.

[5] Jain, S. M. 2020. *Linux Containers and Virtualization*. Apress.

[6] Karlsson, M.; Topel, B.; and Fastabend, J. 2017. Af packet v4 and packet zerocopy. *Netdev conference*.

[7] Kumar, D.; Chakrabarti, S.; Rajan, A. S.; and Huang, J. 2020. Scaling telecom core network functions in public cloud infrastructure. *CloudCom*.

[8] Petazzoni, J. Pipework software-defined networking for linux containers. https://github.com/jpetazzo/pipework.

[9] Peterson, L., and Sunay, O. *5G Mobile Networks: A Systems Approach*. CC BY-NC-ND 4.0.

[10] Qi, S.; Kulkarni, S. G.; and Ramakrishnan, K. K. 2020. Understanding container network interface plugins: Design considerations and performance.

[11] Rajan, A. S.; Gobriel, S.; Maciocco, C.; Ramia, K. B.; Kapur, S.; Singh, A.; Erman, J.; Gopalakrishnan, V.; and Jana, R. 2015. Understanding the bottlenecks in virtualizing cellular core network functions. *The 21st IEEE International Workshop on Local and Metropolitan Area Networks*.

[12] Shea, R., and Liu, J. 2012. Network interface virtualization: Challenges and solutions. *IEEE Network* 28–34.