On the use of kernel bypass mechanisms for high-performance inter-container communications

Gabriele Ara[1], Luca Abeni[1], Tommaso Cucinotta[1], and Carlo Vitucci[2]

[1] Scuola Superiore Sant'Anna, Pisa, Italy
[2] Ericsson, Stockholm, Sweden

Abstract. In this paper, we perform a comparison among a number of different virtual bridging and switching technologies, each widely available and commonly used on Linux, to provide network connectivity to co-located LXC containers for high-performance application scenarios.

1 Introduction

Information and Communication Technologies (ICT) have gone quite a long way, with continuous advances in processing and networking technologies, among others, that resulted into a great push towards distributed computing models. This is witnessed by the widespread diffusion of public cloud computing services, along with private cloud models being employed in nearly every sector/industry. These allow for a greater degree of flexibility in resource management paving the way for on-demand, cost-effective distributed computing solutions progressively replacing traditional dedicated infrastructure management.

In the domain of network operators, recent technological trends led to replacing traditional physical networking infrastructures sized for the peak hour with software-based virtualized network functions (VNFs). These are instantiated on-demand and "elastically", so as to provide the required level of service performance. The new paradigm of Network Function Virtualization (NFV) [11] relies on a flexible general-purpose computing infrastructure managed according to a private cloud paradigm, to provide networking functions at reduced operational costs. As an example, in the Radio Access Network (RAN), a number of different options have been proposed [1] for splitting the functionality among the distributed unit (DU) that needs to stay close to the antenna, vs. the centralized unit (CU) that can be off-loaded to a closeby private cloud data center.

In this context, a number of network functions need high-performance and low end-to-end latency, where a key role is played by the communication overheads experienced by the individual software components participating in each deployed VNF. Such requirements are so tight that NFV has already focused on lightweight virtualization solutions based on operating system (OS) containers, rather than traditional virtual machines (VMs). Moreover, in order to reduce even further the per-packet processing overheads, and at the same time allow for the maximum flexibility in packet processing by the VNF, plenty of experimentation is being done on the use of user-space networking, as opposed to traditional TCP/IP based management of network packets within an OS kernel or hypervisor. While using these kernel bypass[3] techniques, among which a prominent position in industry is played by the Data Plane Development Kit (DPDK) [7],one of the key functionality that needs to be preserved is the virtual switching among multiple containers within the same host. This need can occasionally be dropped in presence of hardware support for virtual switching, such as with Single-Root of I/O Virtualization (SR-IOV) [4], letting different containers/VMs use dedicated virtual functions of the same NIC.

Paper contributions and structure. In this paper, experimental results are presented comparing different virtual switching and user-space packet processing solutions. The focus is on the maximum achievable throughput for small packets exchanged among containers deployed onto the same host. This highlights the difference in per-packet processing overheads of the compared solutions.

The remainder of this paper is structured as follows. Section 2 introduces basic terminology used throughout the paper, and describes the basic elements of the compared networking/switching technologies. Section 3 describes the testbed we used for the comparative evaluation of the considered technologies, and presents experimental results gathered on a multi-core platform running Linux. Section 4 reviews related literature on the topic, discussing our contribution in relation to prior art. Finally, Section 5 contains a few concluding remarks, pointing out future research lines on the topic.

2 Overview of compared solutions

This section presents various background concepts about containers on Linux and the various virtual switching solutions that are compared later in Section 3.

Containers

Differently from traditional machine virtualization, allowing multiple operating systems to coexist on the same host, virtualizing the available hardware via full emulation or para-virtualization, containers are a lightweight virtualization abstraction realized directly within a single operating system, by recurring to proper kernel-level encapsulation and isolation techniques.

The Linux kernel supports containers by proper configuration of control groups (affecting resource scheduling and control) and namespaces via user-space tools such as Docker or LXC. Namespaces allow to isolate (and virtualize) system resources: a process running in a namespace has the illusion to use a dedicated copy of the namespace resources, and cannot access resources outside of it. The Linux kernel provides different kinds of namespaces, one for each different hardware or software resource that needs to be isolated/virtualized. For example, the network namespace encapsulates all the resources used by the kernel network stack, including network interfaces, routing tables, iptables rules, etc...

[3] See for example: https://lwn.net/Articles/629155/

Network connections among processes running inside and outside a namespace are generally implemented by using a virtual Ethernet pair, i.e., two software network interfaces (there is no physical NIC attached to them) connected point-to-point, so that packets sent to one of the two interfaces are received by the other, and vice-versa. Hence, if one of the two endpoints is inside the namespace and the other one is outside, it is possible for the applications running in the namespace to communicate with the external world. This is shown in Figure 1a.

Socket API

Traditionally, the external endpoint of a virtual Ethernet pair is attached to a Linux software bridge or some other kind of virtual switch, while the internal endpoint is accessed via simple blocking system calls exchanging a packet per call, such as when using `send()` or `recv()`. As a result, to exchange a UDP packet, at least two system calls are needed, along with various user-space to kernel space switches, data copies, and scheduling decisions. Therefore, when exchanging small packets, the overheads associated to each system call needed by the sender and the receiver grow to prohibitive values. On the other hand, when exchanging large packets, these overheads are amortized over the large amounts of data exchanged per call, but this time performance bottlenecks are due mostly to how many times data is copied to go from the sender to the receiver.

The typical way to mitigate the first issue is by recurring to batch APIs, with which a single system call sends or receives multiple packets, as possible with `sendmmsg()` and `recvmmsg()`. The second issue, instead, is mitigated by having the application and the underlying hardware reduce any need for copying data, exploiting memory-mapped I/O, scatter-gather primitives or using zero-copy APIs, such as the `MSG_ZEROCOPY` flag with standard `send()`.

Use of `virtio`

To improve the networking performance between containers, system calls, context switches and data copies should be avoided as much as possible. For example, if two containers are co-located on the same physical node, they can exchange data by using shared memory buffers. This requires to bypass the in-kernel networking stack or to use a user-space TCP/IP stack. The latter allows user programs to be developed by using standard networking APIs, where the system is able to use the appropriate optimizations when possible.

This can be achieved by replacing virtual Ethernet pairs with para-virtualized network interfaces based on the virtio standard [19, 18]. These use “virtual queues” of received and transmitted packets, that can be shared among different guests and the host, allowing for the implementation of efficient host/guest communications. While virtio virtual NICs are generally implemented by hypervisors such as qemu/kvm, thanks to the recent “vhost-user” introduction they can also be used in containerized environments. This way, vhost services [23] can be used to move packets across virtual NICs. These can be implemented either in kernel

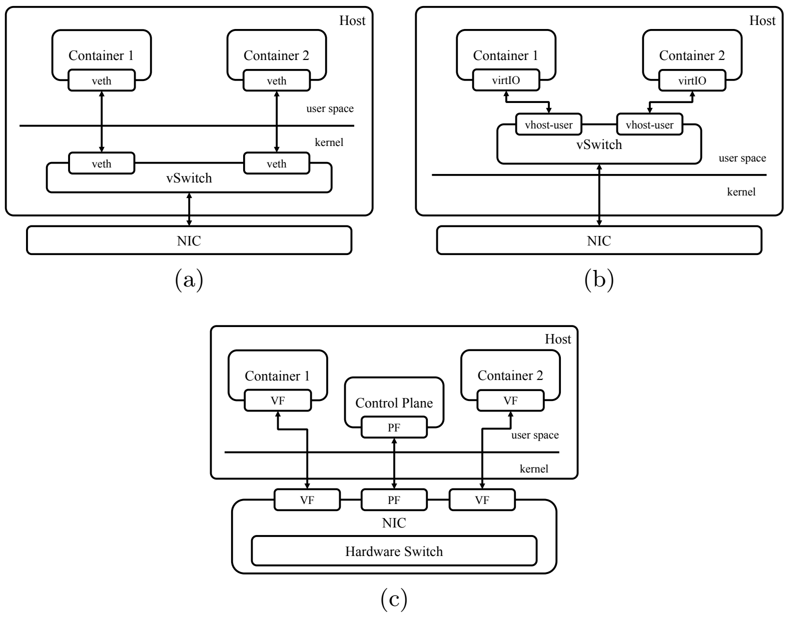


Fig. 1: Different approaches to inter-container networking: (a) kernel-based solution; (b) using DPDK with vhost in user mode; (c) using SR-IOV support.

space (using a kernel thread created by the vhost-net module) or in user-space, using a daemon [12] to implement the vhost functionalities. See also Figure 1b.

DPDK

The simplest way to use virtio NICs in containers to enable the memory sharing optimizations is through a commonly used kernel bypassing framework: the Data Plane Development Kit (DPDK). While DPDK has been originally designed only for kernel bypassing (implementing the NIC drivers in user space), it now provides some interesting features such as the virtio and vhost drivers. The virtio driver is able to connect to virtio-net virtual interfaces, while the vhost driver can be used to implement vhost functionalities in user space (the so called "vhost-user"). For example, virtual switches can use the DPDK vhost driver to implement virtio-net endpoints connecting different VMs.

When using containers instead of full VMs, the vhost driver can be used to implement virtual NICs for the virtio-user driver; as a consequence, a DPDK-based virtual switch running in the host can create virtio-net interfaces which DPDK-based applications running in the containers can connect to. Different software components (described below) can be used as the vhost-user user-space daemon. Clearly, the inter-container networking performance also depends on the software component used as a vhost-user daemon, as shown in Section 3.

Virtual switches based on virtio

The following is a list of the most commonly adopted solutions in industry for inter-container communications using virtual switches. For each of these scenarios, two vhost-user ports are connected with each other via a virtual switch application running in user space.

Open vSwitch (OVS) [10] is an open-source software-only virtual switch that can be used with Linux-based hypervisors (e.g., KVM) or container platforms [13]. Since it is a full fledged open-source switching solution, `ovs` is our reference to evaluate performance of generic virtual switching solutions between vhost-user ports.

DPDK Testpmd can be used to test the functionality of the DPDK PMD driver and can be configured to act as a simple bridge between pairs of virtual or physical ports. Albeit configurable, the default pairing between ports is performed on their ordering (ports 0 and 1, ports 2 and 3, and so on), which means that this software does not contain any actual switching functionality nor packet inspection programmed in it. This is why we will use it as a reference for the maximum performance achievable using PMD driver and vhost-user ports.

Snabb [22] is a simple and fast packet networking toolkit that can be used to program user-space packet processing flows [12]. This is done by connecting functional blocks in a directed acyclic graph, each block performing a specific action or representing a custom driver for an interface. In our tests, we configured it statically as a simple bridge between the two vhost-user ports, and thus it represents the maximum achievable performance by an actual configurable virtual switch solution that performs minimal packet processing on forwarding.

FD.io Vector Packet Processing (VPP) [9] is an extensible framework that provides switching and routing functionalities released in the context of the Fast Data IO (FD.io) Linux Foundation project and that is an open-source implementation of Cisco's VPP technology. The core idea behind VPP is to process more than one packet at a time, taking advantage of instruction and data cache localities to achieve lower latencies and higher throughput [3]. This switch will be used to compare its performance with the one achieved by OVS, which does not have this optimization.

Single Root I/O Virtualization (SR-IOV)

SR-IOV [4] is a specification that allows a single PCIe device to appear as multiple physical PCIe devices. This is achieved by introducing the distinction between Physical Functions (PFs) and Virtual Functions (VFs): the former allow for using the full list of features of the PCIe device, while the latter are "lightweight" functions that have only the ability to move data between an application and the device. VFs can be individually exposed in passthrough to VMs or containers, which can access directly the hardware device, without any need for virtual switching. Most SR-IOV devices contain a hardware layer-2 switch able to forward traffic among PFs and VFs. This is depicted in Figure 1c. SR-IOV devices can either be accessed trough OS drivers, or via DPDK ones, which allow an application to gain complete control over a VF to access it from user space. In this case, a Testpmd application must run on the host to handle configuration requests from applications, to assign or release VFs and configure the control plane associated with the hardware switch, which will then perform all forwarding operations in hardware.

3 Experimental results

In this section we introduce the experimental testbed used for evaluating the considered technologies, along with our testing application and environment.

Platform description and set-up

Experiments were performed on a Dell PowerEdge R630 V4 server equipped with two Intel® Xeon® E5-2640 v4 CPUs at 2.40 GHz, 64GB of RAM and an Intel® X710 DA2 Ethernet Controller for 10 GbE SFP+ (used in SR-IOV experiments). The machine is configured with Linux kernel version 4.15.0, DPDK version 18.11, OVS version 2.11.0, Snabb version 2018.09 and VPP version 19.04. To maximize reproducibility of results, our tests have been run without using hyperthreads and disabling CPU frequency scaling (governor set to `performance` and Turbo Boost disabled).

Testing application

To evaluate the various local networking alternatives described in Section 2, a testing application has been developed, composed of a sender and a receiver exchanging packets at a configurable rate. Each of these programs is deployed on a separate container pinned down on a different CPU of the same CPU package (same NUMA node). The sender program sends packets of a given size in bursts of a given size, matching a desired sending rate. The three parameters can be controlled via command-line options. The receiver program continuously tries to receive packets, measuring the received packet rate.

In a realistic scenario, when deciding what networking paradigm to adopt, different trade-offs would be possible: for example, traditional `send()` and `recv()` system calls can be used on raw sockets to bypass the networking stack, but not the kernel completely. Or a user-space TCP/IP stack implementation can be used to allow using higher-level networking protocols without giving up on the advantages coming from direct access to the NIC.

Table 1: Maximum throughput achieved for various socket-based solutions, using packets of 64 bytes and bursts of 64 packets.

Technique	**Max Throughput** (kpps)
UDP sockets using `send`/`recv`	227
UDP sockets using `sendmmsg`/`recvmmsg`	276
Raw sockets using `send`/`recv`	471
Raw sockets using `sendmmsg`/`recvmmsg`	594

Therefore, the benchmark application comes in various incarnations that use UDP sockets, bypass the networking stack by using raw sockets, or bypass the kernel completely so that:

1. virtio-net is used instead of virtual Ethernet pairs to enable optimizations (based on shared memory buffers) in inter-container communications;
2. DPDK is used to bypass the kernel completely (both in-kernel device drivers and networking stack);
3. polling techniques are used for sending/receiving network packets.

The first version of the benchmark application uses kernel sockets to exchange data; it can be configured to send packets one at a time (using `send`/`recv`) or in bursts of various sizes (using `sendmmsg`/`recvmmsg` to perform a single system call per burst) and it can be set to use raw sockets instead of UDP sockets, to bypass part of the network stack during kernel processing. In any case, the two containers communicate through a simple Linux bridge (further benchmarking with alternative bridging techniques is planned in future work).

The other application variant uses DPDK and the PMD driver to bypass the kernel when exchanging data. This allows it to be used both with vhost-user ports or with SR-IOV virtual functions, depending on the test.

Results

Using the described setups, we measured the receiving rate at various UDP packet sizes (from 64 to 1500 bytes) and packet sending rates (from 1 to 20 Mpps) in steady state conditions. Moreover, packets are sent in bursts (of 16, 32, and 64 packets per burst).

The first result that we want to point out is that performance achieved without using kernel-bypass techniques are much lower than the ones achieved using DPDK, either via virtio or offloading to SR-IOV. For example, using packets of 64 bytes all DPDK solutions were able to achieve over 5 Mpps, while traditional sockets were not even able to reach 1 Mpps. Table 1 reports the maximum throughput achieved using the traditional socket APIs and show that bypassing the networking stack can improve the performance from 276 kpps to 594 kpps; however, even with optimizations like sending packets in batches with `sendmmsg` and a burst size of 64 packets, the achieved performance is significantly impaired when a virtual Ethernet port is used to connect the two containers, and it is even lower with smaller burst sizes.

Switching to the higher-performance alternatives relying on kernel bypass, Figure 2 summarizes the major results from our experimentation, reporting the achieved throughput (on the Y axis) with respect to the desired packet sending rate (on the X axis), using bursts of 16 packets and packet sizes of 64 and 512 bytes. Note that our tests using a burst size of 32 and 64 packets achieved no measurable difference in the results, compared to the showed results, referring to a burst size of 16 packets.

In the plots, each data point is obtained from a 1-minute run, by averaging the obtained per-second receive rate for 20 seconds, discarding as many initial samples for each run, to ensure we skip the initial warm-up phase. Note that the standard deviation among the 20 averaged values was below 2.5 % (and around 0.5% on average) for all the runs.

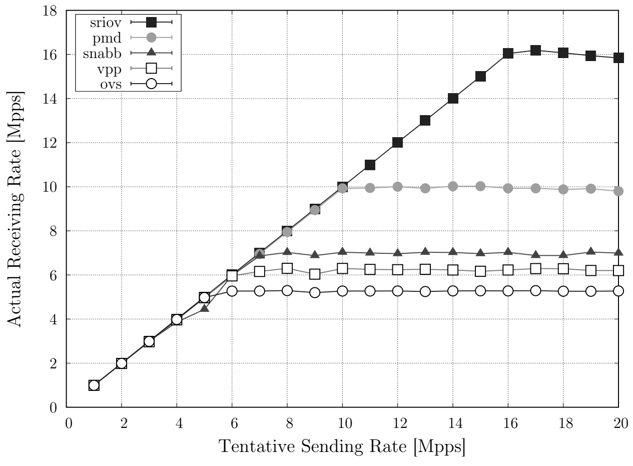
In all our tests, each networking solution is able to achieve the desired throughput up to a certain maximum value, after which any additional packet sent is dropped before reaching its destination. The maximum throughput achieved with each packet size and networking technology is summarized in Figure 3.

Among the solutions that use kernel-bypass techniques, maximum performance is achieved offloading traffic to the SR-IOV enabled Ethernet controller, taking advantage of its hardware switch. The second best performance is achieved by DPDK Testpmd application used as virtual bridge; while this result was expected among virtio-based solutions, it cannot be used in a realistic scenario because the application itself is not a virtual switch, it just forwards all the packets received from a given port to another statically assigned port. Snabb resulted as the best among software switches during our tests; however, it was configured to act as a simple bridge between the two virtual ports, limiting to the very minimum the amount of processing performed on packet forwarding.

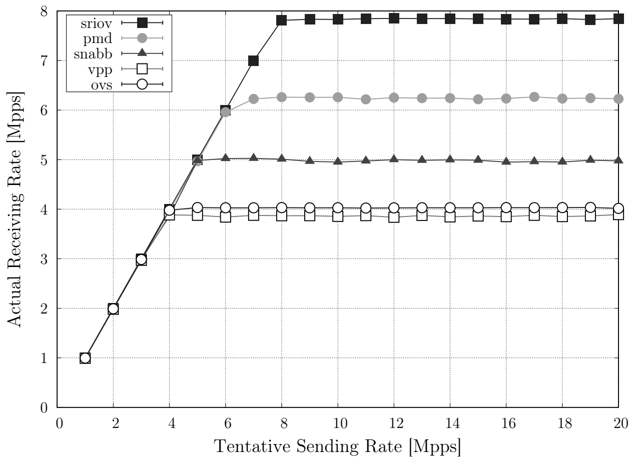
Between the two considered complete virtual switches, VPP and OVS, the former is able to achieve a higher throughput with respect to the latter when the packets are smaller than 512 bytes. This is probably thanks to the batch packet processing features included with VPP, which make it more cache-friendly and lead to higher network performance on larger amounts of packets.

Final remarks

Results indicate clearly that if the focus of the NFV application being developed is to achieve maximum performance, SR-IOV is the way to go, as it can clearly outperform even Testpmd, which is just a mock bridge application. This however requires custom configuration of containers to access the device in passthrough and introduces some limits to the portability of applications, even if DPDK framework provides a sufficiently generic API to access either virtual or physical devices with little effort on a great variety of platforms. If it is not possible to use a SR-IOV enabled interface, the two most promising alternatives based on vhost-user are Snabb and VPP. It is worth noting that Snabb configuration



(a)



(b)

Fig. 2: Receive rate obtained at varying sending rate, with a packet size of 64 bytes (a) and 512 bytes (b).

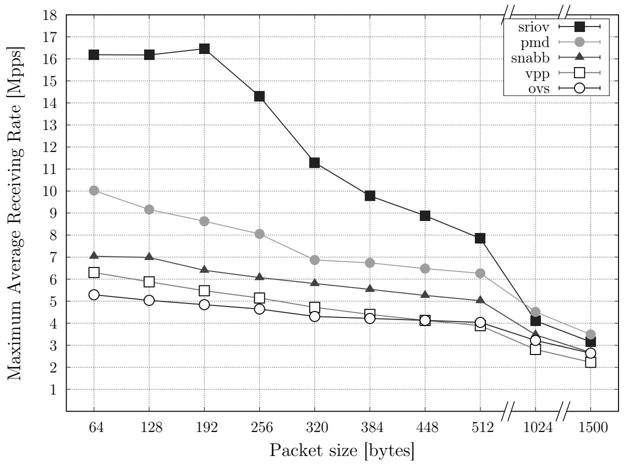


Fig. 3: Maximum performance achieved by each switching technology. Notice that the last two entries on the x axis are not in scale with the rest of the axis.

must be programmed manually in the Lua language, while VPP accepts a simple configuration file on startup.

4 Related Work

Several works appeared in prior research literature addressing how to optimize the networking performance for virtual machines and containers. For example, some authors investigated [2] packet forwarding performance achievable in virtual machines using different technologies to implement the virtual NICs and to connect them to the host network stack or physical NIC.

Many solutions exist to greatly reduce the overheads due to interrupt handlers, like interrupt coalescing and other optimizations available on Linux through New API (NAPI) [21], including hybrid interrupt-handling techniques that switch dynamically between interrupt disabling-enabling (DE) and polling (NAPI) depending on the actual traffic on the line [20].

Comparisons among Remote Direct Memory Access (RDMA) [15], DPDK and traditional sockets already exist [5], mainly focusing on the achievable minimum round-trip latencies between two different machines. These works show how both RDMA and DPDK can outperform sockets, achieving much smaller latencies for small UDP packets, at the cost of forcing applications to operate in poll mode, leading unavoidably to high CPU utilizations. Also, authors point out that DPDK can actually be used in combination with interrupts, saving energy, but before sending or receiving packets the program must switch back to polling mode. This reduces CPU utilization during idle times, at the cost of a greater latency when interrupts must be disabled to revert to polling mode, when the first packet of a burst is received.

Another survey among common networking set-ups for high-performance NFV exists [8], accompanied by a quantitative comparison addressing throughput and CPU utilization of SR-IOV, Snabb, OVS with DPDK and Netmap [16]. Authors highlight how the different solutions have remarkable differences in security and usability, and they show that, for local VM to VM communications, Netmap is capable of reaching up to 27 Mpps (when running on a 4GHz CPU), overcoming SR-IOV due to its limited internal switch bandwidth that becomes a bottleneck. While in this paper we provide a thorough performance analysis for many of the networking solutions described in that work, some of them have not been included; in particular, we did not evaluate NetVM [6] and Netmap (along with its associated virtual switch, VALE [17]), because our focus has been here on solutions commonly adopted in current NFV industrial practice. However, their inclusion in the comparison is among our planned future work.

In another interesting work [14], VPP, OVS and SR-IOV are compared with respect to scalability in the number of VMs on a single host. Other works exist in the area, but a complete state of the art review is out of the scope of this paper.

Differently from the above mentioned works that mostly deal with networking performance for virtual machines, in this paper we focused on industrially viable solutions for high-performance networking for Linux containers. Also, this paper extends our preliminary work [24], where a very basic comparison among communication techniques for co-located Linux containers was done, with a strong focus on motivational arguments for the research in the context of NFV.

5 Conclusions and Future Work

In this paper, we compared various switching technologies for inter-container communications, with a focus on high-performance and poll-based APIs employing batch-based packet processing in user-space. In our experimentation, the best performance has been achieved by SR-IOV, followed by Snabb, among DPDK based virtual switching solutions. These all outperform what is achievable with traditional socket-based APIs and a Linux virtual bridge.

In the future, we plan to extend the evaluation by: considering also the latency dimension, along with the throughput one considered in this work; comparing the various solutions when transmitting through a real network, in addition to the local communications considered here; considering additional works in the comparison, such as Netmap, VALE and NetVM; and finally evaluating how the various solutions impact the performance of a realistic NFV use-case, in addition to the low-level benchmark considered so far.

References

1. 3rd Generation Partnership Project; Transport requirement for CU-DU functional splits options; R3-161813. In: 3GPP TSG RAN WG3 Meeting 93 (August 2016)

2. Abeni, L., Kiraly, C., Li, N., Bianco, A.: On the performance of KVM-based virtual routers. Computer Comm. **70**, 40–53 (2015). https://doi.org/10.1016/j.comcom.2015.05.005
3. Barach, D., Linguaglossa, L., Marion, D., Pfister, P., Pontarelli, S., Rossi, D.: High-Speed Software Data Plane via Vectorized Packet Processing. IEEE Communications Magazine **56**(12), 97–103 (2018)
4. Dong, Y., Yang, X., Li, J., Liao, G., Tian, K., Guan, H.: High performance network virtualization with SR-IOV. Journal of Parallel and Distributed Computing **72**(11), 1471–1480 (2012)
5. Géhberger, D., Balla, D., Maliosz, M., Simon, C.: Performance evaluation of low latency communication alternatives in a containerized cloud environment. In: IEEE 11th International Conference on Cloud Computing (CLOUD). pp. 9–16 (2018)
6. Hwang, J., Ramakrishnan, K.K., Wood, T.: Netvm: High performance and flexible networking using virtualization on commodity platforms. IEEE Transactions on Network and Service Management **12**(1), 34–47 (March 2015)
7. Intel Corporation: Data Plane Development Kit (DPDK) (February 2019), http://www.dpdk.org
8. Lettieri, G., Maffione, V., Rizzo, L.: A Survey of Fast Packet I/O Technologies for Network Function Virtualization. In: High Performance Computing. pp. 579–590. Springer International Publishing, Cham (2017)
9. LF Projects, LLC: Vector Packet Processing (VPP) (February 2019), http://fd.io/technology
10. Linux Foundation Collaborative Project: Open vSwitch (OVS) (February 2019), https://www.openvswitch.org
11. NFV Industry Specif. Group: Network Functions Virtualisation. Introductory White Paper (2012)
12. Paolino, M., Nikolaev, N., Fanguede, J., Raho, D.: SnabbSwitch user space virtual switch benchmark and performance optimization for NFV. In: Proc. of the IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN 2015). pp. 86–92 (November 2015)
13. Pfaff, B., Pettit, J., Koponen, T., Jackson, E., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., Amidon, K., Casado, M.: The Design and Implementation of Open vSwitch. In: 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15). pp. 117–130. Oakland, CA (2015)
14. Pitaev, N., Falkner, M., Leivadeas, A., Lambadaris, I.: Characterizing the performance of concurrent virtualized network functions with OVS-DPDK, FD.io VPP and SR-IOV. In: Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering. pp. 285–292. ACM (2018)
15. RDMA Consortium: Architectural Specifications for RDMA over TCP/IP (February 2019), http://www.rdmaconsortium.org
16. Rizzo, L.: Netmap: a novel framework for fast packet I/O. In: 21st USENIX Security Symposium (USENIX Security 12). pp. 101–112 (2012)
17. Rizzo, L., Lettieri, G.: VALE, a switched ethernet for virtual machines. In: CoNEXT 2012 - Proceedings of the 2012 ACM Conference on Emerging Networking Experiments and Technologies (December 2012)
18. Russell, R., Tsirkin, M., Huck, C., Moll, P.: Virtual I/O Device (VIRTIO) Version 1.0. Standard, OASIS Specification Committee (2015)
19. Russell, R.: VIRTIO: Towards a De-facto Standard for Virtual I/O Devices. ACM SIGOPS Operating Systems Review **42**(5), 95–103 (2008)
20. Salah, K., Qahtan, A.: Implementation and experimental performance evaluation of a hybrid interrupt-handling scheme. Computer Comm. **32**(1), 179–188 (2009)

21. Salim, J.H., Olsson, R., Kuznetsov, A.: Beyond softnet. In: USENIX (ed.) Proceedings of the 5th Annual Linux Showcase & Conference. vol. 5, pp. 165–172 (November 2001)
22. SnabbCo: Snabb (February 2019), https://github.com/snabbco/snabb
23. Tsirkin, M.S.: vhost-net and virtio-net: Need for Speed. In: Proc. of the KVM Forum (May 2010)
24. Vitucci, C., Abeni, L., Cucinotta, T., Marinoni, M.: The Strategic Role of Inter-Container Communications in RAN Deployment Scenarios. In: ICN 2019: The Eighteenth International Conference on Networks (March 2019)