Assessing the Performance of Virtualization Technologies for NFV: a Preliminary Benchmarking

Assessing the Performance of Virtualization Technologies for NFV: a Preliminary Benchmarking

R. Bonafiglia*, I. Cerrato*, F. Ciaccia¶, M. Nemirovsky†, F. Risso*
*Politecnico di Torino, Dip. Automatica e Informatica, Italy
¶Barcelona Supercomputing Center (BSC), Spain
†ICREA Researcher Professor at BSC, Spain

***Abstract*—The NFV paradigm transforms those applications executed for decades in dedicated appliances, into software images to be consolidated in standard server. Although NFV is implemented through cloud computing technologies (e.g., virtual machines, virtual switches), the network traffic that such components have to handle in NFV is different than the traffic they process when used in a cloud computing scenario. Then, this paper provides a (preliminary) benchmarking of the widespread virtualization technologies when used in NFV, which means when they are exploited to run the so called virtual network functions and to chain them in order to create complex services.**

***Keywords*—NFV; service function chain; performance evaluation; KVM; Docker; Open vSwitch**

I. Introduction

Network Function Virtualization (NFV) is a recent network paradigm with the goal of transforming in software images, those network functions that for decades have been implemented in proprietary hardware and/or dedicated appliances, such as NAT, firewall, and so on. These software implementations of network functions, called Virtual Network Functions (VNFs) in the NFV terminology, can be executed on high-volume standard servers, such as Intel-based blades. Moreover, many VNFs can be consolidated together on the same server, with a consequent reduction of both fixed (CAPEX) and operational (OPEX) costs for network operators.

Recently, the European Telecommunication Standard Institute (ETSI) started the Industry Specification Group for NFV [1], with the aim of standardizing the components of the NFV architecture. Instead, the IETF Service Function Chain [2] working group takes into account the creation of paths among VNFs; in particular, they introduce the concept of Service Function Chain (SFC), defined as the sequence of VNFs processing the same traffic in order to implement a specific service (e.g., a comprehensive security suite). Hence, a packet entering in a server executing VNFs may need to be processed in several VNFs (of the same chain) before leaving such a server. Moreover, several chains are allowed on a single server, which process different packet flows in parallel.

NFV is heavily based on cloud computing technologies; in fact, VNFs are typically executed inside Virtual Machines (VMs) or in more lightweight virtualized environments (e.g., Linux containers), while the paths among VNFs deployed on a server are created through virtual switches (vSwitches). While these technologies have been well tested/evaluated for the cloud computing environment, such a study is still missing when used in the case of NFV. In our opinion, an evaluation of virtualization technologies when exploited in the NFV domain is required, since cloud computing and NFV differ both in the amount and in the type of traffic that has to be handled by applications and vSwitches. This difference is due to the following reasons. First, traditional virtualization has to deal most with compute-bounded tasks, while network I/O is the dominant factor in NFV (the main operation of a VNF is in fact to process passing traffic). Second, a packet may need to be handled by several VNFs before leaving the server; this adds further load to the vSwitch, which has to process the same packet multiple times. Finally, common techniques to improve network I/O such as Generic Receive Offload (GRO) and TCP Segmentation Offload (TSO) may not be appropriate for NFV, since some VNFs (e.g., L2 bridge, NAT) need to work on each single Ethernet frame, and not on TCP/UDP segments.

The contribution of this paper is the preliminary benchmarking of VNFs chains deployed on a single server and based on the most common virtualization components, highlighting their bottlenecks under different scenarios and conditions. Particularly, we consider KVM [3] and Docker [4] as execution environments, and Open vSwitch (OvS) [5] and OVDPDK [6] as vSwitches to steer the traffic among them in order to implement the chain(s).

The rest of this paper is organized as follows. Section II provides an overview of the related works, while Section III details the technologies considered in our benchmarking. Tests are detailed in Section IV, while Section V concludes the paper and provides some remarks for the future.

II. Related work

Several works available in literature evaluate the performance of virtualization components, both in the computing virtualization side (i.e., VMs, containers) and in the network virtualization side (i.e., vSwitches). This section provides an overview of the works that are mostly related to the objective of this paper.

From the virtual networking side, [7] and [8] provide a deep evaluation of the behavior of OvS, analyzing how several factors, such as CPU frequency, packets size, number of rules and more, influence the performance of the vSwitch itself. Moreover, [7] measures the throughput obtained when OvS is exploited to interconnect a single VM both with the physical network and with a second VM executed on the same server. However, both the papers do not evaluate OvS when used to cascade several VNFs, as well as they do not consider Docker containers and OVDPDK [6].

In [9], Casoni et al. measure the performance of chains of Linux containers (LXC) [10] interconnected through different technologies: the VALE vSwitch [11], the Linux bridge and the Virtual Ethernet interfaces (`veth`), missing some other widespread technologies such as OvS.

From the computing virtualization side, [12] studies the overhead introduced by the KVM hypervisor [3] when accessing to the disk and to the network. However, it does not evaluate such overhead in a VNF scenario, in which a packet has to potentially traverse several VNFs before leaving a server. This scenario is again not considered in [13], although it provides a comparison between LXC containers and KVM-based VMs. Xavier et al. [14] evaluates networking performance of several virtualization technologies: the Xen hypervisor [15], LXC, OpenVZ [16] and Linux-VServer [17]. However, the tests provided in the paper focus on High Performance Computing workloads, which can be consistently different from those experienced on a server running chains of VNFs.

III. Background

This paper provides an initial evaluation of the overhead introduced by VNF chains deployed on a single server. Different technologies are considered for what regards both the virtual environment running the VNFs, and the vSwitch interconnecting such functions together and with the physical network, in order to identify their performance and compare such technologies among each other.

Virtual Machines (VMs) are the main virtualization technology used to execute VNFs, since they provide strong isolation among VNFs themselves. However, lightweight containers are gaining momentum, since they still provide a form of isolation while having lower resources demand. As environment to execute VNFs, in this paper we consider KVM-based VMs [3] and Docker [4] containers.

Interconnections among VNFs are implemented through a vSwitch, exploiting its capabilities to perform traffic steering based on multiple criteria such as the port ID and L2-L7 protocol fields. Although several vSwitches are available, in this paper we focus on the most widespread ones, namely Open vSwitch (OvS) [5] and OVDPDK [6], a variant of OvS based on the Intel Data Plane Development Kit (DPDK) technology [18].

The remainder of this section provides an overview of the OvS I/O model, as well as it gives some insights on KVM-based VMs and Docker containers, mainly focusing on the way in which they connect to the vSwitch.

A. KVM-based virtual machines and networking

Figure 1 provides an overview of the components involved in the network I/O of VMs running in the KVM hypervisor.

Before detailing how VMs send/receive network packets, it is worth mentioning that KVM is just a kernel module that transforms the Linux kernel into an hypervisor, i.e., it provides to the Linux kernel the capability of running VMs. A VM is then executed within QEMU, which is a Linux user-space process that exploits KVM to execute VMs. For instance, the VM memory is part of the memory assigned to the QEMU process, while each virtual CPU (vCPU) assigned to the VM corresponds to a different QEMU thread in the hypervisor.

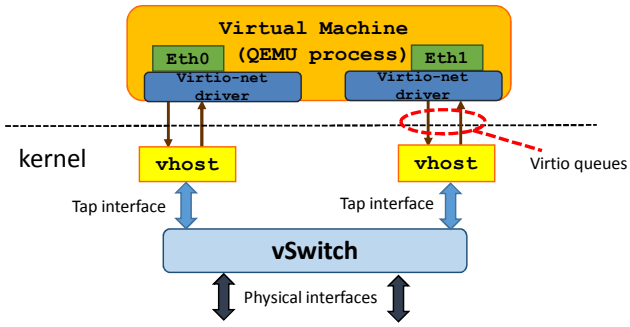


Fig. 1. Networking of a KVM-based virtual machine.

As shown in Figure 1, the guest operating system (i.e., the operating system executed in the VM) accesses to the virtual NICs (vNICs) through the `virtio-net` driver [19], which is a driver optimized for the virtualization context. Each vNIC is associated with two `virtio queues` (used to transmit and receive packets) and a process running in the Linux kernel, namely the `vhost` module in the picture. As shown, `vhost` is connected to the `virtio queues` on one side, and to a `tap` interface on the other side, which is in turn attached to a vSwitch. `vhost` works in interrupt mode; particularly, it waits for notifications from both the VM and the `tap` and then, when such an interrupt arrives, it forwards packets from one side to the other.

As a final remark, the transmission of a batch of packets from a VM causes a *VM exit*; this means that the CPU stops to execute the guest (i.e., the vCPU thread), and run a piece of code in the hypervisor, which performs the I/O operation on behalf of the guest. The same happens when an interrupt has to be "inserted" in the VM, e.g., because `vhost` has to inform the guest that there are packets to be received. These *VM exits* (and the subsequent *VM entries*) are one of the main causes of overhead in network I/O of VMs.

B. Docker containers and networking

Docker containers are a lightweight virtualization mechanism that, unlike VMs, do not run a complete operating system: all the containers share the host's kernel[1]. Containers represent a way to limit the resources visible by a running userland process; hence, if no process is running in the container, such a container is not associated with any thread in the host. Such this limitation of resources (e.g., CPU, memory) is achieved through the `cgroups` feature of the Linux kernel, while isolation is provided through the Linux namespaces, which give to processes running in the container a limited view of the process trees, networking, file system and more.

As shown in Figure 2, each container corresponds to a different network namespace; this means that it is only aware of those interfaces inserted into its own namespace, as well as it has a private network stack. According to the picture, packets can traverse the namespace boundary by means of `veth` pairs,

[1] In other words, on the physical machine there is a single kernel, with a single scheduler, a single memory manager and so on.

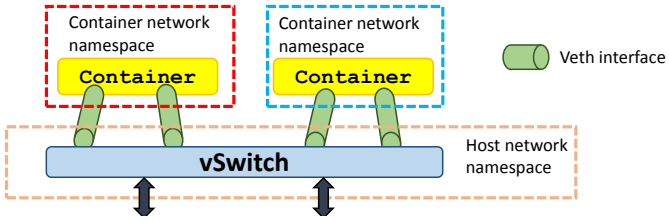


Fig. 2. Networking of a Docker container.

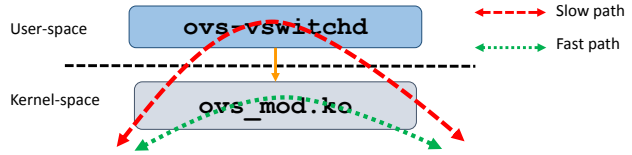


Fig. 3. Open vSwitch data-plane architecture.

which are in fact a pair of interfaces connected through a pipe: packets inserted in one end are received on the other end. Hence, by putting the two ends of the `veth` in different namespaces, it is possible to move packets from one network namespace to another. According to the picture, a vSwitch connects all the `veth` interfaces in the host namespace among each other and with the physical network.

C. Open vSwitch I/O model

OvS is a widespread vSwitch, whose data plane architecture is depicted in Figure 3; as shown, it consists of a user-space daemon and a kernel-space module, respectively referred as `ovs-vswitchd` and `ovs-mod.ko` in the picture.

The kernel module implements the fast path; it acts in fact as a cache for the user-space component, and includes all the last matched rules for increasing forwarding efficiency. In particular, when a packet enters into the vSwitch, it is first processed in the kernel module, which looks for a cached rule matching such a packet. In case of positive result, the corresponding action is executed, otherwise the packet is provided to the user-space.

The user-space daemon (that implements the slow path) contains in fact the full forwarding table of the vSwitch. By default, it implements the traditional MAC learning algorithm, although it can also be "programmed" by an external controller through the Openflow protocol[2]. Then, after that the first packet of a flow has been handled in user-space, the corresponding rule is offloaded to the kernel module, which will be able to handle all the subsequent packets of the same flow, thus increasing sensibly the performance.

Interesting, there are no threads associated with the `ovs-mod.ko` kernel module; the in-kernel code of OvS consists in fact of a callback invoked by the software interrupt raised when a packet is available on a network interface. For instance, the OvS code (in the hypervisor) is executed in the context of the `ksoftirq` kernel thread in case of packets coming from the physical NIC, while it is executed in the context of the `vhost` associated with the "sender" vNIC in case of packets coming from the VM (Figure 1). In case of packets coming from Docker (Figure 2), OvS runs in the context of the process executed in the container. This is possible because a single kernel exists, which is shared among the host and the containers.

[2]The capability to program the switching table of OvS from an external entity is exploited in NFV to create chains of VNFs.

D. OVDPDK I/O model

OVDPDK is a version of OvS based on DPDK, a framework proposed by Intel that offers efficient implementations for a wide set of common functions, such as NIC packet input/output, memory allocation and queuing.

Unlike the standard OvS, OVDPDK simply consists of a user-space process (with a variable number of threads) that continuously iterates over the physical NICs and the `virtio queues`. The `vhost` module shown in Figure 1 is in fact integrated in the vSwitch, so that the `tap` interface, which introduces overhead in the data path of packets to/from VMs, is removed. Furthermore, thanks to DPDK: *(i)* the vSwitch accesses to the physical NICs without the intervention of the operating system; *(ii)* packet transfer between the physical NICs and the vSwitch is done with zero-copy.

As a final remark, OVDPDK can also exchange packets with the VMs through shared memory: this solution removes the `vhost` at all, and does not require the `virtio-net` driver in the VMs. Although this configuration allows the vSwitch to move packets in a zero-copy fashion among VMs, we decided of not using it in our preliminary benchmarking of service chains. In fact, due to a design choice of DPDK, the same memory would be shared among all the VMs deployed, thus weakening the isolation among VMs themselves.

IV. Performance Characterization

This sections details our benchmarking of VNFs chains implemented on a single server. As already mentioned, we carried out several tests using different technologies for what regards both the virtualization environment used to run VNFs (KVM and Docker), and the vSwitch that properly steers the traffic among them in order to create the chain(s) (OvS and OVDPDK).

Particularly, our tests focus on measuring the throughput and the latency obtained when packets (of different size) flow through a server hosting one or more chains of different length. Each measurement lasted 100 seconds and was repeated 10 times; results are then averaged and reported in the graphs shown in the following of this section. Some graphs are provided with a bars view and a points-based representation of the maximum throughput. The former representation is referred to the left y axis, which provides the throughput in Mpps, while the latter is referred to the right y axis, where the throughput is measured in Gbps.

A. Hardware and software setup

As shown in Figure 4, our test environment includes a server that runs a vSwitch and a variable number of VNFs; this server is connected to a sender and to a receiver machine through two point-to-point 10Gbps Ethernet links. All the physical machines are equipped with an Intel Core i7-4770 @3.40GHz (4+4 cores), 32GB of memory, and run Fedora

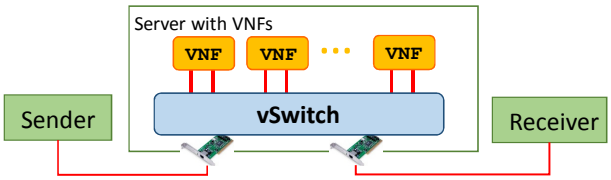


Fig. 4. Physical set up used for the tests.

TABLE I. BARE-METAL THROUGHPUT OF THE APPLICATIONS USED IN OUR MEASUREMENTS.

	Linux bridge	libpcap-based bridge	DPDK-L2fw
64B [Mpps (Gbps)]	2.39 (1,14)	1.29 (0.66)	8.78 (4.39)

20 with kernel 3.18.7-100.fc20.x86_64. The phsyical network interfaces are instead Intel X540-AT2.

As already mentioned, VNFs are executed either in KVM-based VMs[3], or in Docker containers. According to Figure 4, each VNF is equipped with two vNICs connected to the vSwitch, which can be either OvS or OVDPDK. In order to steer the traffic among VNFs, the vSwitch is configured with Openflow rules matching the input port of packets, while the action is always the forwarding of packets through a specific port. In case of multiple service chains deployed on the server, the traffic entering from the physical port is split so that some packets enter in one chain, other packets in another chain and so on. Traffic splitting is based on the source MAC address of packets, which are properly generated so that they are equally distributed among all the available chains.

To measure the overhead introduced by different vSwitches and virtualization engines when used for NFV, we kept the VNFs as simple as possible. Particularly, our VNFs simply receive packets from one interface and forward them on the other; such applications, in a non-virtualized environment (i.e., when used to directly connect two physical interfaces), provide the throughput shown in Table I.

The throughput is measured using unidirectional traffic flowing from the sender to the receiver machine, respectively running a packet generator and a packet receiver based on the `pfring/dna` [20] library, which allows to transmit/receive packets at 10Gbps.

B. OvS-based chains: virtual machines vs Docker containers

This section evaluates the throughput achieved when a single chain of VNFs is deployed on a server running OvS as vSwitch[4]. Measurements have been repeated with VMs and Docker containers, chains of growing length and packets of different size; results are reported in Figure 5. Those graphs have been obtained by executing the Linux bridge within VMs, while each container runs a simple user-space bridge based on `libpcap` [21]. In fact, with the Linux bridge inside containers, we got an unacceptable throughput of 3Mbps with 8 chained VNFs.

[3]In this case, each VM is associated with a single vCPU, i.e., the whole VM corresponds to a single thread in the hypervisor. Moreover, the guest operating is Ubuntu 14.04 with kernel 3.13.0-49-generic, 64 bit.

[4]Note that, due to the small number of rules inserted in the vSwitch, the packet processing is entirely done in the kernel module (Section III-C).

This poor result is a consequence of two factors: *(i)* the Linux bridge data plane is in fact a kernel-level callback executed in the context of the thread that provides packets to the vSwitch (similarly to OvS); *(ii)* all Docker containers share the kernel with the host (as detailed in Section III-B). As a consequence, the `ksoftirq` kernel thread that processes the packet received from the physical NIC is also in charge of executing *(i)* the OvS code that forwards the packet in the first container; *(ii)* the data plane part of the Linux bridge launched in the first container (but running, in fact, in the host kernel); *(iii)* again OvS, and so on, until the packet leaves the server through the physical NIC. In other words a single kernel thread executes all the operations associated with the packet, resulting in no parallelization at all and hence in really poor performance, especially in case of long chains.

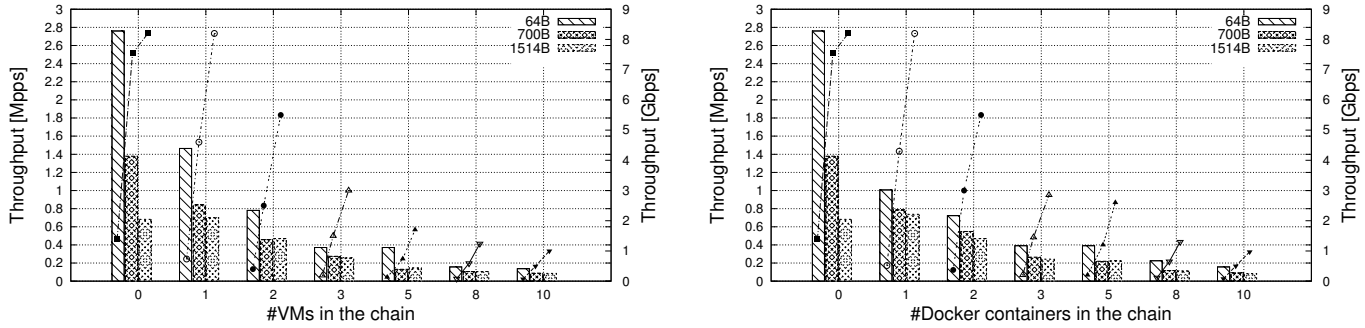
Instead, in case of VMs, the vCPU thread (Section III-A) executes the guest kernel and hence the Linux bridge, while `vhost` executes the OvS code, thus providing high parallelization in packets processing and then acceptable performance. In case of Docker, such a parallelization can be achieved by running a VNF that is actually a process (e.g, our simple `libpcap`-based bridge). This way, the OvS code between VNFs is executed in the context of this process, and the `ksoftirq` is just involved at the beginning of the service chain.

A comparison between Figure 5(a) and Figure 5(b) shows that chains implemented with the two virtualization technologies (when executing the proper application) present different throughput only when there is a single VNF per chain (and 64B packets), while they are almost equivalent in the other cases. However, according to Table I, the Linux bridge provides nearly twice the throughput of the `libpcap`-based bridge, when executed in a non-virtualized environment. The fact that we got almost the same throughput by running these two applications respectively in VMs and containers, shows how the overhead introduced by full virtualization is higher than the overhead due to lightweight virtualization mechanisms.

Figure 5 also shows that the throughput is inversely proportional to the number of chained functions, regardless of the virtualization environment. In fact, more VNFs executed on the same server results in more contention on the CPU resource, as well as on the cache(s) and the Translation Lookaside Buffer (TLB). Even, with longer chains, the probability that a packet is processed on different physical cores is higher, with a consequent increasing in the number of experienced cache misses. Moreover, although OvS is executed on several threads, there is just one instance of the forwarding table, which requires synchronization for example to increment the counters associated with the matched rules.

Figure 6 reports instead the throughput obtained when 64B packets are equally distributed among multiple chains of VMs. As evident, more chains result in lower throughput measured on the receiver machine, for the same reasons stated above in case of a single chains of different lengths (i.e., more CPU contention, more cache misses, etc.). Similar results have been achieved with containers, although they are not reported in the paper due to space constraints.

We can then conclude that Docker containers are well suited to run VNFs only in case of applications associated with



(a) (b)

Fig. 5. Throughput of a single chain (of growing length) with VNFs deployed in: (a) KVM-based virtual machines; (b) Docker containers.

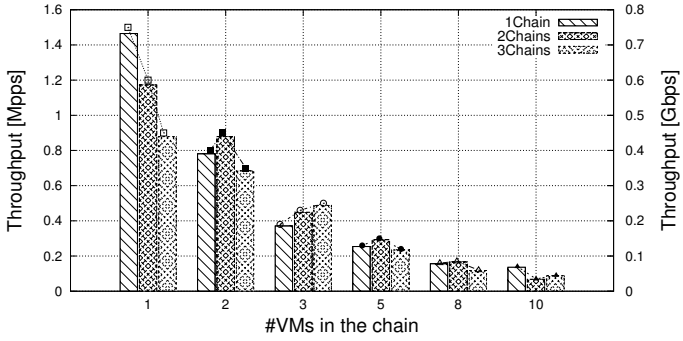


Fig. 6. Throughput with several chains implemented with VMs and OvS.

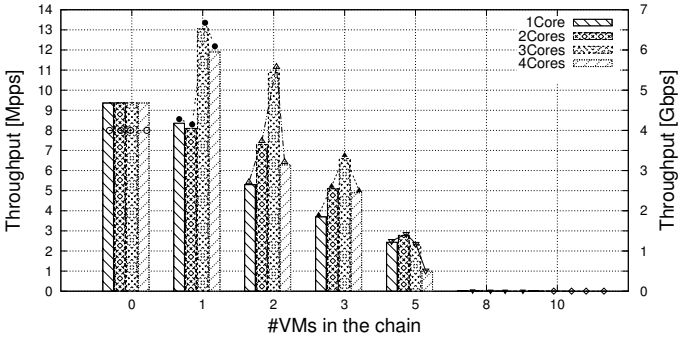


Fig. 7. Throughput with VMs chained through OVDPDK.

a specific thread/process. In this case, they provide almost the same performance of KVM-based VMs, with the advantage of requiring less resources (e.g., memory), but with the drawback of providing less isolation.

C. KVM-based virtual machines: OVDPDK vs OvS

This section evaluates the performance of a chain of VNFs interconnected through OVDPDK, and compares such results with those achieved in case of OvS. Particularly, in order to compare the two switching technologies, we consider VNFs executed in KVM-based VMs. Before analyzing the results, it is worth remembering that OVDPDK is entirely executed in user-space and works in polling, i.e., it continuously iterates over the available NICs. Moreover, the `vhost` module (Figure 1) is replaced with a software layer integrated in OVDPDK, so that VMs directly communicate with the vSwitch.

As a first test, we consider again the Linux bridge as VNFs, obtaining a throughput (not shown in the paper due to space constraints) that is just slightly better than those achieved with OvS and depicted in Figure 5(a); e.g., 1.66Mpps (0.85Gbps) with 64B packets traversing a singe VNF. In fact, although OVDPDK provides several enhancements with respect to OvS, such improvements are mitigated by the high overhead introduced by VMs.

Then, we installed DPDK in VMs as well, in order to execute DPDK-based VNFs and exploits the DPDK acceleration both in the guest and in the hypervisor. In particular, results shown in Figure 7 are gathered using the L2-forwarded application provided with the DPDK library (and 64B packets). The test has then been repeated by changing the number of polling threads (and hence CPU cores) associated with the vSwitch, in order to evaluate its effect on the performance of the chain. Note that, since the L2-forwarder works in polling, during this test we always have to dedicate a CPU core to the vCPU thread associated with each VM.

According to the picture, assigning more cores to the vSwitch results in better performance in case the number of cores required by the whole chain (VMs + vSwitch) does not exceed the number of cores of the server. When this happens, performance degrades until becoming unsustainable with 8 chained VMs. In fact, at this point some cores are shared among many polling threads and, for instance, it may happen that the operating system assigns the CPU to a VNF with no traffic to be processed, penalizing others that would actually have work to do.

For the sake of clarity, Figure 7 only reports the throughput obtained with 64B packets. However, our measurements also reveal that a chain implemented with OVDPDK and the DPDK L2-forwarder within VMs, provides a throughput of 10Gbps in case of 700B and 1514B packets. Of course, such results are obtained only in case the chain does not require more cores than those available on the server.

A comparison between Figure 7 and Figure 5(a) reveals how chains implemented through DPDK-based components (i.e., OVDPDK and VMs running DPDK applications) outperform chains not based on the DPDK library. In fact, DPDK-

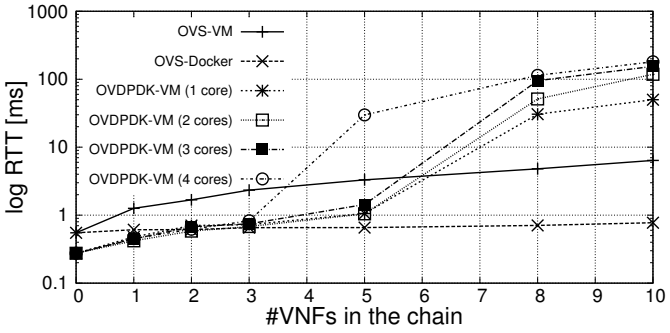


Fig. 8. Latency introduced by service chains implemented through different technologies.

based modules works in polling and optimize the data transfer among each others, exploiting zero-copy as much as possible. However, as already stated above, the polling working model has the drawback of providing unacceptable throughput when the cores required exceed the number of cores available on the server.

D. Latency

In networking, the latency introduced by the system is as important as the throughput achieved. Hence, this section reports the latency measured with a single chain implemented with the technologies discussed above. In particular, for each scenario, we executed 100 `ping` between the sender and the receiver machines; results have been averaged and reported in Figure 8.

As expected, DPDK-based chains (OVDPDK + DPDK-L2forwarder in VMs) provide better results, provided that the number of cores required does not exceed the number of cores of the physical machine on which such a chain is deployed. At this point, similarly to what already discussed in case of throughput, latency becomes definitely unacceptable.

Figure 8 also shows how Docker containers running a user-space process (i.e., the simple `libpcap`-based bridge) introduce smaller latency than VMs. This difference in performance is again a consequence of the higher overhead introduced by full virtualization with respect to lightweight containers.

V. Conclusion

This paper provides a performance analysis of service chains implemented through different virtualization technologies, selected among those representing the state of the art in the NFV domain.

From the several tests carried out, we can draw the following conclusions. First, Docker containers provide acceptable results only in case of VNFs associated with a specific process, while they are definitively unsuitable for VNFs implemented as callbacks to be executed in the kernel. However, due to the low overhead introduced by lightweight virtualization, a user-space program in Docker provides better latency and almost the same throughput of callback-based VNFs (e.g., Linux bridge) run within VMs kernel. Second, OVDPDK (when used with VMs running DPDK-based applications) provides much better performance than OvS. For instance, this is due to the exploitation of zero-copy when transferring packets to/from physical NICs, and to the polling model implemented by DPDK-based processes. However, the polling model requires at least a dedicated core per VM, hence limiting the usage of OVDPDK in servers with a reduced number of VNFs.

Acknowledgment

The research described in this paper is part of the SECURED project [22], co-funded by the European Commission under the ICT theme of FP7 (grant agreement no. 611458).

References

[1] "Etsi nfv," http://www.etsi.org/technologies-clusters/technologies/nfv.

[2] Internet Engineering Task Force (IETF), "Service Functions Chaining (SFC) working group," 2014. [Online]. Available: https://datatracker.ietf.org/wg/sfc/documents/

[3] "Kvm," http://www.linux-kvm.org.

[4] "Docker," https://www.docker.com/.

[5] "Openvswitch," http://openvswitch.org/.

[6] "OVDPDK," https://github.com/01org/dpdk-ovs.

[7] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "Performance characteristics of virtual switching," in *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, Oct 2014, pp. 120–125.

[8] Y. Zhao, L. Iannone, and M. Riguidel, "Software switch performance factors in network virtualization environment," in *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*, Oct 2014, pp. 468–470.

[9] M. Casoni, C. Grazia, and N. Patriciello, "On the performance of linux container with netmap/vale for networks virtualization," in *Networks (ICON), 2013 19th IEEE International Conference on*, Dec 2013, pp. 1–6.

[10] "Lxc," https://linuxcontainers.org/.

[11] L. Rizzo and G. Lettieri, "Vale, a switched ethernet for virtual machines," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, ser. CoNEXT '12. New York, NY, USA: ACM, 2012, pp. 61–72. [Online]. Available: http://doi.acm.org/10.1145/2413176.2413185

[12] B. Zhang, X. Wang, R. Lai, L. Yang, Z. Wang, Y. Luo, and X. Li, "Evaluating and optimizing i/o virtualization in kernel-based virtual machine (kvm)," in *Network and Parallel Computing*, ser. Lecture Notes in Computer Science, C. Ding, Z. Shao, and R. Zheng, Eds., vol. 6289. Springer Berlin Heidelberg, 2010, pp. 220–231.

[13] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," vol. 28, p. 32.

[14] M. Xavier, M. Neves, F. Rossi, T. Ferreto, T. Lange, and C. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, Feb 2013, pp. 233–240.

[15] "Xen," http://www.xen.org/.

[16] "OpenVZ," https://openvz.org/.

[17] "Linux-VServer," http://linux-vserver.org.

[18] "Dpdk," http://dpdk.org/.

[19] R. Russell, "Virtio: Towards a de-facto standard for virtual i/o devices," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 95–103, Jul. 2008.

[20] "Ntoppfring," http://www.ntop.org/products/pf_ring/.

[21] S. McCanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture," in *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USENIX Association, 1993, pp. 2–2.

[22] "Security at the network edge (SECURED)," http://www.secured-fp7.eu/.