A Framework for Comparative Evaluation of High-Performance Virtualized Networking Mechanisms

Gabriele Ara[1](✉), Leonardo Lai[1], Tommaso Cucinotta[1], Luca Abeni[1], and Carlo Vitucci[2]

[1] Scuola Superiore Sant'Anna, Pisa, Italy
{gabriele.ara,leonardo.lai,tommaso.cucinotta,luca.abeni}@santannapisa.it
[2] Ericsson, Stockholm, Sweden
carlo.vitucci@ericsson.com

Abstract. This paper presents an extension to a software framework designed to evaluate the efficiency of different software and hardware-accelerated virtual switches, each commonly adopted on Linux to provide virtual network connectivity to containers in high-performance scenarios, like in Network Function Virtualization (NFV). We present results from the use of our tools, showing the performance of multiple high-performance networking frameworks on a specific platform, comparing the collected data for various key metrics, namely throughput, latency and scalability, with respect to the required computational power.

Keywords: Kernel bypass · DPDK · Netmap · NFV · Containers · Cloud computing

1 Introduction

Over the last decade, many applications shifted from centralized approaches to distributed computing paradigms, thanks to the widespread availability of high-speed Internet connections. As a result, cloud computing services experienced a stable growth in the past few years, both in sheer size and the number of services provided to their end-users. Their success is mostly due to their high level of flexibility in resource management, especially for those applications that may be subject to significant service demand variations over time.

Cloud systems also gained the interest of network operators, intending to replace traditional physical networking infrastructures with more flexible cloud-based systems. To achieve this goal, highly specialized networking devices will be progressively replaced with equivalent software-based implementations that can be dynamically instantiated and relocated inside a cloud-based infrastructure, called Virtualized Network Functions (VNFs). This approach represents the core idea behind Network Function Virtualization (NFV), which has gained popularity in recent years. Given the nature of the services usually deployed in

D. Ferguson et al. (Eds.): CLOSER 2020, CCIS 1399, pp. 59–83, 2021.
https://doi.org/10.1007/978-3-030-72369-9_3

NFV infrastructures, these systems must be characterized by high performance in terms of throughput and latency among VNFs. These services are typically deployed in long service chains; for this reason, it is imperative to maintain the cost of individual components interactions as small as possible, to avoid high end-to-end costs across the whole chain. These requirements are so tight that the NFV industry is now considering Operating System (OS) containers to deploy VNFs in cloud infrastructures, rather than traditional Virtual Machines (VMs), following the rise of popularity of container solutions like LXC or Docker. These solutions exhibit similar performance as deploying VNF applications directly on the host OS [7,9], by partially sacrificing isolation among virtualized components.

Thanks to containers' superior performance, the research focus is now into further reducing communication overheads. Many high-performance I/O frameworks have been developed in the past decade to reduce by several orders of magnitude the cost for user-space application to send and receive packets with respect to traditional networking stacks.

1.1 Contributions

This paper shows the characteristics of a benchmarking framework for comparing system performance when adopting high-performance I/O solutions to interconnect VNF components deployed in a private cloud infrastructure using OS containers. In particular, this tool eases the creation of a virtual network infrastructure using software-based networking solutions or even leveraging special features in network devices that support the Single-Root I/O Virtualization (SR-IOV) specification. It can then be used to deploy on that infrastructure a set of benchmarking applications that measure system performance under various working conditions. We present experimental results collected using this framework and compare the performance of various virtual switching solutions (either software-based or hardware-accelerated) when subject to synthetic workloads.

This work constitutes an extended version of the paper already appeared in [4]. Details on this will follow at the end of Sect. 5.

2 Background

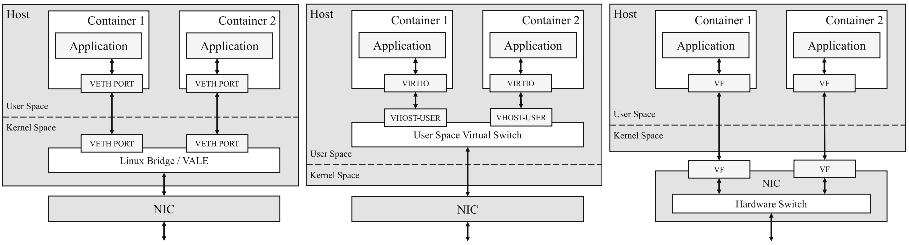
Application components or services deployed in OS containers inside a cloud infrastructure can choose among several network primitives to communicate with each other or with the external world. Usually, these primitives use network virtualization techniques to provide a set of gateways to exchange data over a virtual network infrastructure. Choosing the right communication primitives to use when connecting multiple encapsulated components may significantly impact the overall application performance and latency, but some of them require a special set-up from the infrastructure provider that limits the flexibility in deployment typical of cloud environments.

In this work, we focus on the following solutions: **(i)** *kernel-based networking*, **(ii)** *kernel-based networking with network stack bypass* (e.g. Netmap), **(iii)** *software-based user-space networking*, **(iv)** *hardware-accelerated user-space networking*. In the following, we summarize the main characteristics of each of these techniques when adopted in NFV scenarios to interconnect OS containers within a private cloud infrastructure. We will focus on the performance attained when adopting each solution on general-purpose computing machines running Linux.

2.1 Kernel-Based Networking and VNFs

Most operating systems, including Linux, provide abstractions that can be used to create and assign virtual Ethernet ports to VMs or OS containers. Each virtual port has no corresponding hardware interface; they are purely implemented in software as endpoints for networked communications within the same host by emulating the behavior of real Ethernet ports. Typically, these virtual ports are created as directly connected pairs: this means that each port in the pair is always directly connected with the other one as if connected by a virtual cable.

Using standard techniques provided by the Linux kernel, containers or other virtualized environments can be interconnected by assigning one end of the virtual Ethernet pair each. This operation usually hides the selected port from the host networking devices[1], allowing applications in the VM or container to send packets to the virtual port on the other end of the connection.



(a) Kernel-based solution. (b) Using DPDK with *vhost-user*. (c) SR-IOV support.

Fig. 1. Different approaches to inter-container networking. Adapted from [4].

To connect more than two virtualized environments to the virtual network or to connect a VM or container to the actual physical Network Interface Controller (NIC), a virtual implementation of a L2 switch is required, to forward packets

[1] OS containers in Linux achieve isolation employing cgroups and namespaces. With these tools, virtual Ethernet ports assigned to a container will no longer be visible or accessible outside the assigned cgroup/namespace, but it will still be part of the host network stack. In this sense, OS containers do not introduce any overhead when encapsulated applications exchange packets over the virtual network.

from each virtualized environment to the desired destination, be it another virtual port or the outside world. For this purpose, the Linux kernel implements a virtual switch called "*linux-bridge*". It allows VNFs to communicate on the same host with other containerized VNFs or with other hosts via forwarding through actual Ethernet ports present on the machine, as shown in Fig. 1a.

Virtual Ethernet ports can be accessed via blocking or nonblocking system calls, for example using the standard POSIX Socket API, exchanging packets via `send()` and `recv()` (or their more general forms `sendmsg()` and `recvmsg()`). With this approach, at least two system calls are required to exchange each UDP datagram over the virtual network; therefore, overheads grow proportionally with the number of packets exchanged. In addition, each packet traverses various network stack layers in the Linux kernel, to be properly processed and delivered.

The recent introduction of batch system calls in the kernel API enables partial amortization of the cost of a system call over a *burst* of packets. The `sendmmsg()`/ `recvmmsg()` system calls handle multiple packets in a single call, reducing the number of system calls required to exchange huge traffic volumes. However, this only reduces the ratio between the number of packets and the number of system calls needed to exchange each packet over the local virtual network, but packets still need to traverse the whole kernel network stack, going through additional copies, even when transmitted locally on a machine.

2.2 Bypassing the Kernel's Networking Stack

Several solutions can be adopted inside the Linux kernel to bypass (entirely or partially) the standard networking stack, in favor of more efficient pipelines designed for high-performance data plane operations.

The most straightforward solution is to partially bypass the networking stack using raw sockets instead of regular UDP sockets and implementing networking and transport-level encapsulation in user-space. This approach is often taken in combination with zero-copy APIs and memory-mapped I/O to transfer data quickly between a single application and the virtual Ethernet port, partially reducing the time needed to send a packet [21]. This way, part of the high-level processing required on the kernel side can be skipped, leaving the kernel the only burden of forwarding raw Ethernet frames from an Ethernet port to another one, at the expense of handling upper networking stack layers (UDP, IP, etc.) inside the user-space application itself. For this purpose, some efficient user-space implementations of the network stack exist [13,26]. Finally, applications using raw sockets require exclusive access to the virtual network interface, preventing other applications in the same virtualized environment to access it; this is not a relevant problem in most NFV scenarios, since each container usually encapsulates exactly one VNF application.

Another solution is to change the *linux-bridge* component into another in-kernel software switch that is more optimized for the traffic expected from the VNFs. A representative example of this solution is Open vSwitch (OVS) [18], a flexible general-purpose virtual switch implemented as a kernel module focused on high-performance scenarios. The implementation of OVS is optimized to handle traffic generated by virtualized environments, employing caching techniques throughout its implementation, especially in its packet classifier.

Another advantage of using a replacement for *linux-bridge* inside the kernel is that VNF applications do not need to be rewritten or customized for different sets of APIs or system calls. However, there are situations in which this approach cannot achieve the required performance levels. If we analyze the overhead required for UDP or raw sockets, about 50% of total processing time is spent on the system calls [21]. This consideration indicates that `send()`, `recv()`, and similar APIs are not efficient mechanisms to exchange data between the user-space application and the kernel. This consideration can lead to two distinct approaches to tackle this problem: **(i)** redesign the way user-space applications interact and exchange data with the Linux kernel itself; **(ii)** bypass the Linux kernel entirely and build new APIs and communication mechanisms in user-space, so that there is no need to pay the cost of executing a system call at all. The former is the approach taken by Netmap, while the latter is the one of many solutions that rely on kernel bypass techniques described in Sect. 2.3.

Netmap [22] is a networking framework for high-performance I/O developed for FreeBSD and Linux. Netmap has a custom APIs allowing applications to send and receive multiple packets per system call, without any need for data copies between user and kernel space[2]. Netmap achieves high performance removing three main packet processing costs [21], namely system call overheads (amortized over large packet bursts), per-packet dynamic memory allocation (pre-allocating fixed-size packet buffers and descriptors during interfaces initialization phase), and expensive data copies (providing user-space applications direct access to in-kernel packet buffers). These features are provided by leveraging standard memory mapping and protection mechanisms for device registers and other kernel memory areas to enforce protection among processes.

FreeBSD already includes Netmap kernel support by default since version 11, while Netmap can be installed on Linux by patching a set of standard NIC drivers and loading some additional custom modules[3]. The driver patches introduce a new mode for the various network device drivers in the Linux kernel, called Netmap mode. Unlike the NIC default operating mode, in which packets are exchanged from and to each NIC through the standard kernel's networking stack, devices in Netmap mode no longer communicate with the default networking stack. Rather, their ring buffers are connected to Netmap-defined ring buffers, implemented in a shared memory area. Netmap data structures provide device-independent yet efficient access to data, providing a representation that closely resembles NICs' typical ring-based internal structures [21].

An application that wants to leverage Netmap features can either use a modified version of libpcap [20], which supports network devices in Netmap mode, or directly use Netmap's custom API. In the latter case, the application

[2] However, data copies across multiple processes are still required for security reasons, especially when interacting components do not trust each other, like VNFs.

[3] https://github.com/luigirizzo/netmap.

first obtains a reference to Netmap's in-kernel data structures from user-space, including packet buffers; it can then start filling them with packets to send or consuming the received packets. Synchronization between user and kernel space is achieved using either blocking system calls (using either `select()` or `poll()` to send or receive packets), or non-blocking ones (using `ioctl()` for both sending and receiving operations). The non-blocking alternative checks if there are empty packet buffers for new outgoing packets (for send operations) or if there are packets ready to be processed (for receiving ones) [22].

Notice that, contrary to traditional `sendmsg()`/`recvmsg()` and similar system calls, Netmap uses system calls only as synchronization mechanisms, no data copies are issued between user and kernel space during the execution of each system call. Also, Netmap provides other features to achieve high performance for both local and remote communications, including support for multiple hardware queues, and zero-copy data transfer with supported interfaces [22].

2.3 Inter-container Communications with Kernel Bypass

Significant performance improvements over traditional networking between containers can be achieved also bypassing the kernel entirely. This removes the costs associated with system calls, context switches and unneeded data copies as much as possible. Various I/O frameworks undertake such approach, recurring to a set of kernel bypassing techniques to exchange batches of packets among applications without requiring a single system call. Typically, they require using different kinds of virtualized or para-virtualized network interfaces that can be managed from the user-space.

One notable example of these kinds of ports is introduced by the *virtio* standard [24]: it defines a new kind of para-virtualized ports which rely on shared memory to achieve high-performance networking among applications running on the same host (even across containers), a fairly common situation in NFV scenarios. These interfaces expose "virtual queues" for incoming/outgoing packets that can be shared among different guests on the same hosts or connected to software implementations of network switches, allowing the implementation of efficient host-to-guest and guest-to-guest communications. While *virtio* interfaces are typically implemented by hypervisors (e.g. QEMU, KVM), a user-space implementation of the *virtio* specification, called *vhost-user*, has been defined.

Notice that while *virtio* ports can effectively improve significantly same-host communication performance with respect to fully virtualized Ethernet ports, they cannot be used to directly access the physical network without any user-space software implementation of a network switch, which is necessary to achieve both dynamic and flexible communications among independently deployed VNFs.

Data Plane Development Kit (DPDK)[4] is an open source framework for fast packet processing implemented entirely in user-space, characterized by a

[4] https://www.dpdk.org/.

high portability across multiple platforms. Initially developed by Intel for its own family of network devices, it now provides a flexible high-level programming abstraction, called Environment Abstraction Layer (EAL) [1], that provides applications an efficient access point to low-level resources from user-space without depending on specific hardware devices. Data Plane Development Kit (DPDK) uses various techniques to reduce the gap between applications and network interfaces, including non-blocking access to packet rings, batch packet transfers between memory and interfaces, and the use of resident huge pages of memory to hold memory buffers.

Other than several physical interfaces from multiple vendors, DPDK supports *virtio*-based networking via its own implementation of *vhost-user* interfaces. Hence, DPDK APIs can be used to exchange data efficiently both locally and with applications residing on remote hosts, in complete transparency for the user applications: for local communications, *vhost-user* ports can be used, while for remote ones the efficient user-space implementation of real Ethernet device drivers provided by DPDK can be leveraged. For this reason, DPDK has become extremely popular over the past few years to develop high-performance networking applications.

2.4 High-Performance Switching Among Containers

High-performance virtual networking infrastructures can be implemented by employing a combination of software and/or hardware tools. There are essentially three main ways to achieve this goal: **(i)** by assigning each container a virtual Ethernet port and connecting each of port to an efficient implementation of an in-kernel software switch (Fig. 1a); **(ii)** by assigning each container a *virtio* port, using *vhost-user* to bypass the kernel, and then connect each port to a software implementation of a virtual switch running in user-space on the same host (Fig. 1b); **(iii)** by leveraging special capabilities of certain NIC devices that allow concurrent access from multiple applications and that can be accessed in user-space by using DPDK drivers (Fig. 1c). The virtual switch instance used on each host (either software or hardware) is then connected to the physical network via the actual NIC interface present on the host.

Many software implementations of L2/L3 switches are available, each implementing their own packet processing logic responsible for packet forwarding. Some of them can be used in combination with DPDK, Netmap or other networking frameworks to improve the performance over standard networking APIs. For these reasons, performance may differ significantly across implementations.

A common characteristic of most software virtual switches is a non-negligible amount of processing power required to achieve very high network performance. On the other hand, special NIC devices that support the SR-IOV specification allow traffic offloading to a hardware switch embedded in the NIC itself, which applications can access concurrently without interfering with each other.

Below, we briefly describe the most common software virtual switches in the NFV industrial practice, and the characteristics of network devices compliant with the SR-IOV specification.

VALE [23] is an implementation of an efficient virtual Ethernet switch that can be used instead of the default host networking stack to connect applications that use ports in Netmap mode on the same host or to connect virtual Netmap ports with the physical NIC present on the host. In principle, VALE acts like a traditional L2 learning switch, associating each port with a list of L2 addresses by inspecting the source field of each incoming Ethernet frame. VALE is specialized to manage Netmap's ring buffers and it implements a multi-stage forwarding process that leverages packet batching and cache prefetching instructions to speed up memory accesses. While it does not support zero-copy of data from one port to another, even on the same host, for isolation purposes between different applications [23], it does not require any data copy between user and kernel space (thanks to Netmap API design).
DPDK Basic Forwarding Sample Application[5] is a sample application provided by DPDK that can be used to connect DPDK-compatible ports, either virtual or physical, in pairs: this means that each application using a given port can only exchange packets with a corresponding port chosen during system initialization. For this reason, this software does not perform any packet processing operation, hence it cannot be used in real use-case scenarios.
Open vSwitch (OVS)[6] is an open source virtual switch for general-purpose usage with enhanced flexibility thanks to its compatibility with the *OpenFlow* protocol [18]. Recently, OVS has been updated to support DPDK and *virtio*-based ports, which accelerated considerably packet forwarding operations by performing them in user-space rather than within a kernel module [2]. This is the preferred solution when the focus is on data-plane performance, rather than deploying OVS as an alternative to the default network stack inside the Linux kernel (see Sect. 2.2).
FD.io Vector Packet Processing (VPP)[7] is an extensible framework for virtual switching released by the Linux Foundation Fast Data Project (FD.io). Since it is developed on top of DPDK, it can run on various architectures and it can be deployed in VMs, containers or bare metal environments. It uses Cisco VPP that processes packets in batches, improving the performance thanks to the better exploitation of instruction and data cache locality [5].
Snabb[8] is a packet processing framework that can be used to provide networking functionality in user-space. It allows for programming arbitrary packet processing flows [17] by connecting functional blocks in a Directed Acyclic Graph (DAG).

While not being based on DPDK, it has its own implementation of *virtio* and some NIC drivers in user-space, which can be included in the DAG.

Single-Root I/O Virtualization (SR-IOV) [8] is a specification that allows a single NIC device to appear as multiple PCIe devices, called Virtual Functions

[5] https://doc.dpdk.org/guides/sample_app_ug/skeleton.html.
[6] https://www.openvswitch.org.
[7] https://fd.io/.
[8] https://github.com/snabbco/snabb.

(VFs), that can be independently assigned to VMs or containers and move data through dedicated buffers within the device.

VMs and containers can directly access dedicated VFs and leverage the L2 hardware switch embedded in the NIC for either local or remote communications (Fig. 1c).

Using DPDK APIs, applications within containers can access the dedicated VFs bypassing the Linux kernel, removing the need of any software switch running on the host; however, a DPDK daemon is needed on the host to manage the VFs.

3 Proposed Framework

This section presents the framework we realized for the purpose of evaluating and comparing the performance and efficiency of different virtual networking solutions. The framework can be easily installed and configured on any desired number of interconnected general-purpose servers running an Ubuntu-based Linux distribution; it can be used to instantiate and deploy a number of OS containers, each running a custom high-performance benchmarking application. This application, also developed for this framework, serves the dual purpose to generate/consume synthetic network traffic, simulating real NFV applications, and to collect statistics to evaluate system performance in the given configuration.

The purpose of this framework is to carry out a number of experiments from multiple points of view, depending on the investigation focus, while varying testing parameters (e.g. packet size, sending rate, etc.) and system configuration. Each test defines which networking solution is to be used to interconnect the benchmarking applications, how many instances for each machine should be instantiated, and what are the characteristics of the network traffic that should be generated. After each individual distributed test is done, the framework collects and stores the system performance measured by each benchmarking application and moves on to the next configuration in the list. This way, multiple tests can be performed consecutively, without any additional user intervention. When all tests are finished, a summary of the collected statistics is presented to the user.

The software is open-source and it is freely available on GitHub, under a GPLv3 license, at: https://github.com/gabrieleara/nfv-testperf . It can be conveniently extended by researchers or practitioners, should they need to write further customized testing applications. Figure 2 depicts the software architecture of the framework, which includes a number of software tools, both readily available or custom-made, and Bash scripts. The latter ones are used to install system dependencies, configure and customize installation, set up and run performance evaluations, and collect statistic data.

The framework dependencies include the DPDK framework (including its Basic Forwarding Sample Application), Netmap (and its own virtual switch, VALE), and the other user-space virtual switches described in Sect. 2.4: the user-space implementation of OVS (compiled with DPDK support), VPP, and

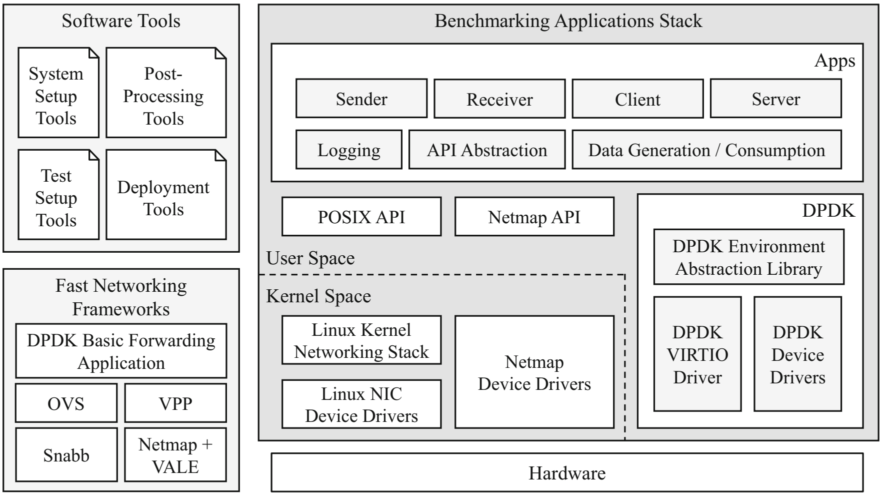


Fig. 2. Main elements of the proposed framework. Adapted from [4].

Snabb. Each virtual switch is configured to act as a simple learning L2 switch, with the only exception represented by the DPDK Basic Forwarding Sample Application, which does not have this functionality. In addition, OVS, VPP, and VALE can be connected to physical Ethernet ports to perform tests for inter-machine communications.

Figure 2 shows the internal structure of the custom benchmarking application included in the framework. This can be configured to act either as traffic generator and/or consumer (depending on the kind of VNF application that is emulated) to evaluate system performance from the following points of view:

Throughput: Many VNFs generate or consume huge volumes of network traffic per second: for this reason, it is of utmost importance to evaluate the maximum forwarding performance provided by each networking solution, varying system parameters, in relationship with the required computational resources. For this purpose, the benchmarking application can be configured to act as a pure sender or pure receiver application, to generate/consume unidirectional traffic (from each sender to a designated receiver application).

Latency: In general, in NFV infrastructures it is crucial to strive for the minimum latency possible for individual interactions, in order to reduce the end-to-end latency between components across long service chains. For this purpose, a client/server application pair is used to generate bidirectional traffic to evaluate the average round-trip latency for each packet when multiple packets are transmitted in bursts over the virtual network infrastructure. To do so, the server application will send back each packet it receives to its corresponding client.

Scalability: Evaluations from this point of view are orthogonal with respect of the two previous dimensions, in particular with respect to throughput: since full utilization of a computing infrastructure is achieved only when multiple VNFs are deployed on each host, it is extremely important to evaluate how the networking performance of multiple concurrent applications are affected when increasing the number of applications deployed on the each host. For this purpose there are no dedicated applications: multiple instances of each designated application can be deployed concurrently to evaluate how that affects global system performance.

The benchmarking applications are implemented in C and they are built over a custom API that masks the differences between POSIX, DPDK, or Netmap frameworks; this way, they can be used to evaluate system performance using each of the approaches described in Sect. 2 to realize the virtual network infrastructure. When POSIX APIs are used to exchange packets, raw sockets can also be used rather than regular UDP sockets to bypass partially the Linux networking stack, building Ethernet, IP and UDP packet headers in user-space.

Each application emulates a specific kind of VNF application, namely a sender, a receiver, a server, or a client application. In each case, the application accepts a number of parameters that determine the kind of traffic that is generated/consumed, including the sending/receiving rate, packet size, burst size, etc. To maximize application performance, the applications are developed to use always non-blocking APIs and measurements of elapsed time are performed by checking the TSC register instead of less precise timers provided by Linux APIs.

During each test, each application is deployed within a LXC container on the targeted machines and automatically connected to the other designated application in the pair, according to the provided configuration. The Linux distribution that is used to realize each container is based on a simple *rootfs* built from a basic *BusyBox* and it contains only the necessary resources to run the benchmarking applications. Depending on the networking solution selected for the current test, the framework takes care of all the setup necessary to interconnect the deployed applications with the desired networking technology, being it *linux-bridge*, VALE, another software-based virtual switch (using *virtio* and *vhost-user* ports), or a SR-IOV Ethernet adapter; again, each scenario is depicted in Fig. 1. In any case, deployed applications use polling to exchange network traffic over the selected ports. For tests involving multiple hosts, only OVS, VPP, or VALE can be used among software-based virtual switches to interconnect the benchmarking applications; otherwise, it is possible to assign to each container a dedicated VF and leverage the embedded hardware switch in the SR-IOV network card to forward traffic from one host to another.

The proposed framework can be easily extended to include more low-level networking frameworks, alongside DPDK and Netmap's APIs, or more virtual switching solutions that can be used to interconnect the containerized applications. From this perspective, the inclusion of other *virtio*-based virtual switches is straightforward, and it does not require any modification of the existing test applications. In contrast, other low-level networking frameworks not considered in this work that rely on custom port type/programming paradigms may require the extension of the API abstraction layer to adapt it to the new low-level components. Other high-level testing applications generating or consuming different types of synthetic workloads can also be easily introduced on top of the existing API. Further details about the framework's extensibility can be found at https://github.com/gabrieleara/nfv-testperf/wiki/Extending.

4 Experimental Results

This section reports experimental results obtained with the framework just introduced above. The goal of the experiments is to test the functionality of the framework and compare the performance of the various virtual switching solutions described in this paper.

We performed all experiments on two identical hosts: the first has been used for all local inter-container communications tests, while both hosts have been used for multi-host communication tests (using containers as well). The two hosts are two Dell PowerEdge R630 V4 servers, each equipped with two Intel® Xeon® E5-2640 v4 CPUs at 2.40 GHz, 64 GB of RAM, and an Intel® X710 DA2 Ethernet Controller for 10 GbE SFP+ (used in SR-IOV experiments and multi-host scenarios). The two Ethernet controllers have been connected directly with a 10 Gigabit Ethernet cable. Both hosts are configured with Ubuntu 18.04.3 LTS, Linux kernel version 4.15.0-54, DPDK version 19.05, OVS version 2.11.1, Snabb version 2019.01, VPP version 19.08, and Netmap for Linux (September 2020). To maximize results reproducibility, the framework carries out each test disabling CPU frequency scaling (governor set to performance and Turbo Boost disabled). Finally, the various components of the framework have been configured to avoid using hyperthreads simultaneously.

4.1 Testing Parameters

The framework's configuration depends on the parameters used to instantiate the containers containing the benchmarking applications, set up the virtual network,

Table 1. List of parameters used to run performance tests with the framework. Adapted from [4].

Parameter	Symbol	Description
Test dimension	D	The test evaluates *throughput* or *latency* performance
Hosts used	L	The test performs only communications on a single host (shown as "*local*") or between different hosts ("*remote*")
Containers set	S	The number of container pairs deployed on the host for the test duration; this is expressed as "$NvsN$"—e.g. "$1vs1$" means that there are two containers in a pair, while "$4vs4$" means four pairs of containers are deployed
Virtual switch	V	The virtual switch used to connect the containers; can be one among *linux-bridge*, *basicfwd* (for the Basic Forwarding Sample Application), *ovs*, *snabb*, *sriov*, *vpp*, *vale*
Packet size	P	The size of each packet, in bytes; includes the content of the whole Ethernet frame
Sending rate	R	The desired packet sending/receiving rate, expressed in packets per second
Burst size	B	The number of packets that are grouped in each burst

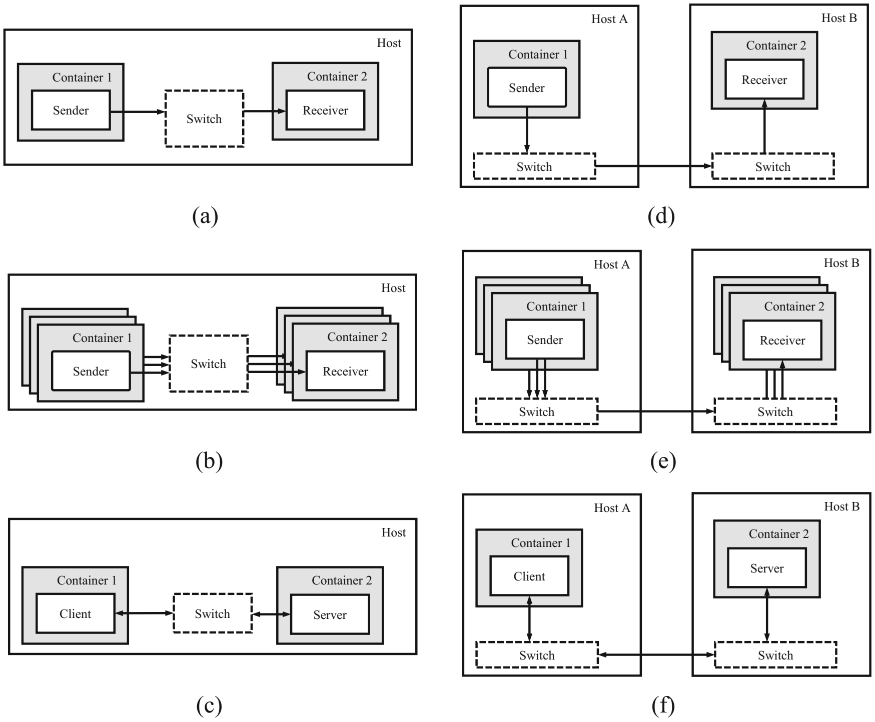


Fig. 3. Different testing scenarios used for our evaluations. In particular, (a), (b), and (c) refer to single-host scenarios, while (d), (e), and (f) to scenarios that consider multiple hosts. From [4].

and instruct the applications to generate traffic with specific characteristics. The number of test cases is significant; hence, we show only relevant results for each different perspective.

To identify each test, we use the following notation, where a tuple uniquely identifies each test in the following form:

$$(D,\ L,\ S,\ V,\ P,\ R,\ B)$$

Table 1 describes each parameter in detail. Then, referring to the results of multiple tests at the same time, we omit from this notation the parameters that are free to vary within a predefined set of values. For example, to show some tests performed while varying the sending rate, the R parameter may not be included in the tuple.

For each test, the framework automatically deploys the applications on one or both hosts, grouped in pairs (i.e. sender/receiver or client/server), and it runs the desired test for a fixed amount of time. The scenarios that we considered in our experiments are summarized in Fig. 3. Each test uses only a fixed set of parameters and runs for 1 minute. Upon completion, we compute the average value of the desired metric (either throughput or latency), after discarding a certain number of values from the beginning and the end of the experiment; the discarded values are related to initial warm-up and shutdown phases. The resulting statistics are therefore calculated only over values related to steady-state conditions of the system.

4.2 Kernel-Based Networking

Table 2. From [4]. Maximum throughput achieved for various socket-based solutions: ($D = \textit{throughput}$, $L = \textit{local}$, $S = 1vs1$, $V = \textit{linux-bridge}$, $P = 64$, $R = 1M$, $B = 64$).

Technique	Max throughput (kpps)
UDP sockets using `send/recv`	338
UDP sockets using `sendmmsg/recvmmsg`	409
Raw sockets using `send/recv`	360
Raw sockets using `sendmmsg/recvmmsg`	440

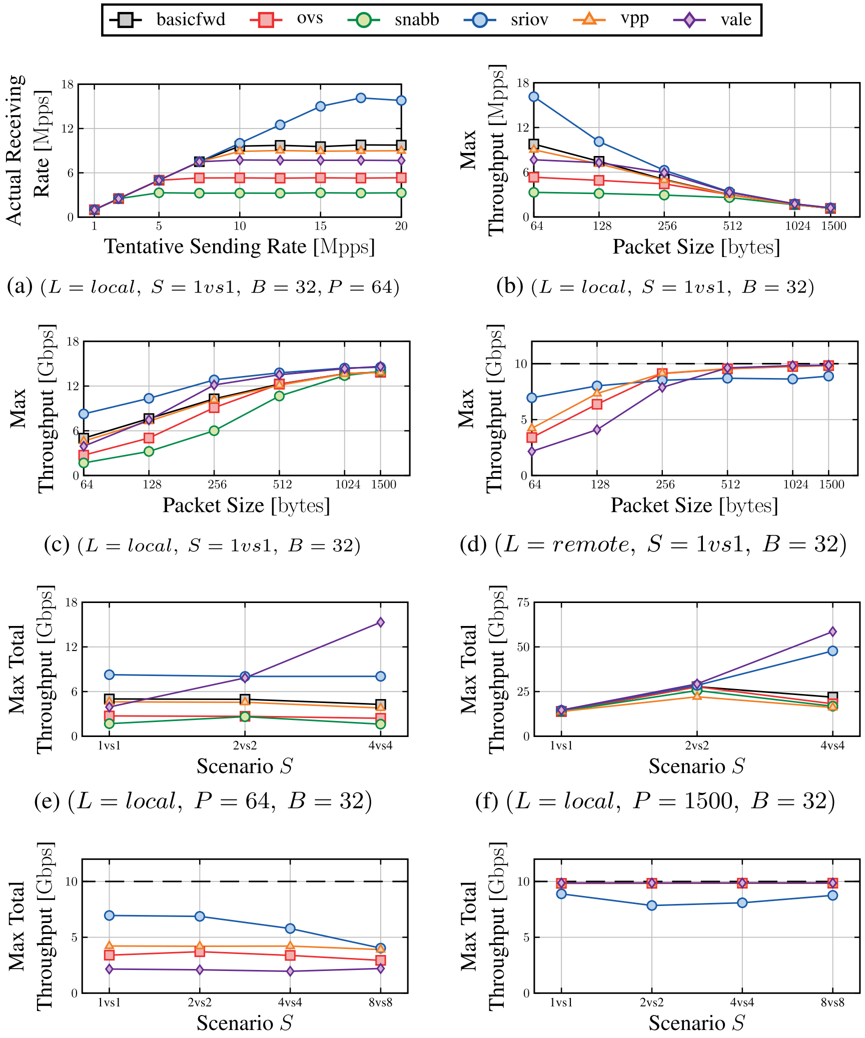
In this section, we show the performance achieved using standard POSIX system calls and Linux kernel's networking stack. Table 2 reports the maximum throughput achieved using POSIX socket APIs and *linux-bridge* to interconnect a pair of sender and receiver application, both deployed on the same host (Fig. 3a). With this configuration, the maximum throughput is achieved when most of the networking stack is bypassed, using raw sockets, with a maximum throughput of 0.440 Mpps. As we will show in the following sections, using Netmap or techniques that entirely bypass the Linux kernel, it is possible to achieve well over 2 Mpps in similar set-ups. Given their inferior performance compared to the other frameworks, in all the results that will follow standard POSIX system calls and *linux-bridge* will not be considered anymore.

4.3 Throughput Evaluations

This section evaluates throughput performance between two applications in a single pair, deployed either on the same host or multiple directly connected machines, varying the desired sending rate, packet, and burst sizes using high-performance networking frameworks. In all our experiments, we noticed that varying the burst size from 32 to 256 packets per burst did not affect throughput performance; thus, we will always refer to the case of 32 packets per burst in further reasoning, if not explicitly indicated otherwise. In all our experiments, we considered 1 Gbps exactly equal to 10^9 bits per second.

Same-Host Throughput Results. First, we deployed a single pair of sender and receiver applications on a single host (Fig. 3a), and we connected them each time with one among the various high-performance frameworks available:

$$(D = \textit{throughput},\ L = \textit{local},\ S = 1vs1)$$



(a) $(L = local,\ S = 1vs1,\ B = 32, P = 64)$

(b) $(L = local,\ S = 1vs1,\ B = 32)$

(c) $(L = local,\ S = 1vs1,\ B = 32)$

(d) $(L = remote,\ S = 1vs1,\ B = 32)$

(e) $(L = local,\ P = 64,\ B = 32)$

(f) $(L = local,\ P = 1500,\ B = 32)$

(g) $(L = remote,\ P = 64,\ B = 32)$

(h) $(L = remote,\ P = 1500,\ B = 32)$

Fig. 4. Throughput performance obtained varying system configuration and virtual switch used to connect sender/receiver applications deployed in LXC containers.

In these tests, we varied the packet sending rate from 1 to 20 Mpps and the packet size from 64 to 1500 bytes. The two applications are configured to generate and consume each exchanged packet's content, respectively, simulating an actual application's behavior.

Figure 4a shows that each networking solution matches the desired throughput in each of our tests until a certain plateau is reached, which varies depending on the capabilities of each virtual port/switch combination. In our evaluations, this maximum throughput strongly depends on the packet size; thus, from now on, we consider only the maximum achievable throughput for each networking framework when the size of each packet varies in our desired range.

The maximum receiving rates achieved in our tests between two containers on the same host are shown in Figs. 4b and 4c. Each plot shows the achieved receiving rate (y-axis), expressed respectively in Mpps and Gbps, as a function of the size of each packet (x-axis), for which a logarithmic scale has been used. In both figures, the behavior shown by each solution is similar: with an increase in the packet size, the throughput in terms of Mpps decreases progressively, while in terms of Gbps it grows logarithmically with the packet size (Fig. 4c). From these results, we can see that while the maximum throughput in terms of Mpps is achieved with the smallest of the selected packet sizes (64 bytes), in terms of Gbps it is better to use the biggest packet size (1500 bytes).

From both figures, it is clear that the maximum performance is attained by offloading network traffic to the SR-IOV device, exploiting its embedded hardware switch. Instead, the second-best solution varies depending on the packet size selected: while for smaller packet sizes both the Basic Forwarding Sample Application and VPP dominate the other solutions, as the packet size increases over 128 bytes, VALE outperforms them both, almost resulting en-par with SR-IOV offloading. The overall high performance of the Basic Forwarding Sample Application is expected since it does not implement any actual switching logic. The minimal performance gap between VPP and the latter solution indicates that the batch packet processing features that characterize VPP can effectively distribute packet processing overheads among incoming bursts of packets. Finally, OVS and Snabb, which lack similar optimizations, obtain inferior performance with respect to the other solutions. Comparing its performance with other evaluations present in literature that used Snabb only to connect directly two ports without a switching component in-between [3], we were able to conclude that its internal L2 switching component represents the major bottleneck for Snabb.

In general, these figures show that the efficiency of each port/switch pair has a more significant impact when smaller packets are exchanged over the virtual network, and hence a higher number of packets is processed by each virtual switch: the performance gap among the various solutions is very small for packets whose size is 1 kB and beyond. From this, we can conclude that for bigger packet sizes, the system's major bottleneck becomes the capability of the CPU and the memory subsystem to move data from one CPU core to another, which is mostly equivalent for any implementation. Given also the slightly superior performance achieved by SR-IOV, especially for smaller packet sizes, we also concluded that its hardware switch is more efficient at moving a large number of packets between CPU cores than the software implementations that we tested.

Note that the authors of VALE reported a performance peak of 27 Mpps in [14], however the sender/receiver application they used is simpler than ours, that scans through every byte of sent and received packets, calculating a very simple CRC, for the purpose of emulating better the effect on the overall experiment of possible limits arising from the limited memory bandwidth available.

Therefore, the performance attainable with our framework is expected to be lower, albeit more representative of what would be achievable by a realistic application that has to prepare the packets to send and process the received ones. Additionally, in our experimentation, a non-particularly fast CPU was used, clocked at 2.4 GHz, while in the experiments in [14] a 4 GHz CPU was used. Finally, they employed VMs rather than OS containers to perform the experiment, In that configuration, Netmap applications can leverage some optimizations that forward system calls to a pool of threads running on the host machine [15], which may lead to increased performance compared to bare-metal deployments.

Multiple Hosts. We repeated these evaluations deploying the receiver application on a separate host (Fig. 3d), using the only virtual switches able to forward traffic between multiple hosts[9]:

$$(D = throughput,\ L = remote,\ S = 1vs1,\ V \in \{ovs, sriov, vpp, vale\})$$

Figure 4d shows the maximum receiving rates achieved for a burst size of 32 packets. In this scenario, results depend on the exchanged packets' size: for smaller packet sizes, the dominating bottleneck is still represented by the CPU for all software-based virtual switches, while for bigger packets, the Ethernet line rate limits the total throughput achievable by any virtual switch to only 10 Gbps. From these results, we concluded that when the expected traffic is characterized by relatively small packet sizes (up to 256 bytes), deploying a component on a directly connected host does not impact system performance negatively when using OVS, VPP, or VALE. Also, we noticed that in this scenario, there is no clear best virtual switch with respect to the others: while SR-IOV is more efficient for smaller packet sizes, software-based virtual switches perform better for bigger ones.

4.4 Throughput Scalability Evaluations

The scalability of system performance is a critical factor in NFV since the full utilization of system resources can be achieved only by deploying multiple components on each host. That is why we repeated all our throughput evaluations deploying multiple application pairs on the same host (Fig. 3b), up to 4 sender/receiver pairs:

$$(D = throughput,\ L = local,\ S \in \{1vs1, 2vs2, 4vs4\})$$

From our previous evaluations, we highlighted that the throughput capability of most networking solutions varies greatly depending on the size of the packets exchanged over the local network. For this reason, we show in Figs. 4e and 4f the relationship between the number of application pairs deployed simultaneously and the maximum total throughput achieved (i.e. the maximum sum

[9] The Basic Forwarding Sample Application does not implement any switching logic, while Snabb was not compatible with our selected SR-IOV Ethernet controller.

of throughput values registered simultaneously by all sender/receiver pairs) for packets of 64 and 1500 bytes respectively.

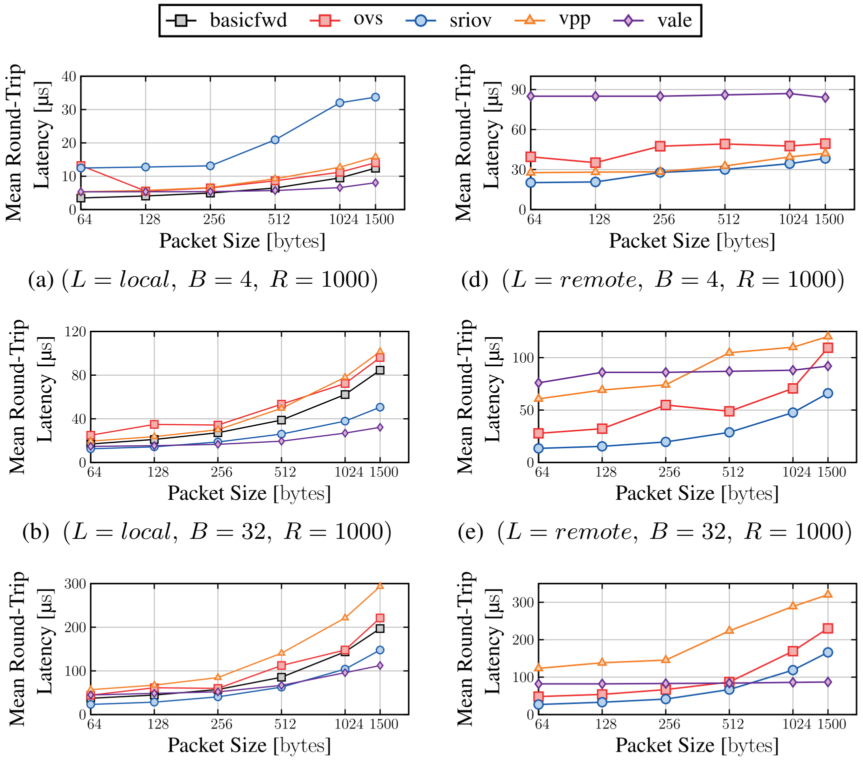
Figure 4e highlights a significant difference between VALE and the other network solutions considered in this work. While most virtual switches do not achieve higher total throughput when increasing the number of application pairs transmitting 64 bytes per packet, VALE's throughput increases almost linearly with the number of application pairs. Most virtual switches have a fixed amount of processing power at their disposal, distributed among all the packet flows traversing them[10]. On the other hand, VALE operates directly inside the Linux kernel: each sender process is responsible for forwarding its packets to their destination. In a scenario where N processes send packets simultaneously over the local network, VALE can use virtually N times the processing power than with a single sender. The effectiveness of this approach is evident when the scalability of the system is taken into account, albeit it does consume part of the processing power of each process to do packet processing operations. The resulting performance is only penalized when only one packet flow is present on the system ($1vs1$), while the other software/hardware virtual switches cannot keep up once more network flows are added.

The situation is slightly different when we increase the packet size, up to 1500 bytes per packet. Figure 4f shows that for bigger packets, SR-IOV also shows a similar almost-linear behavior with the increase of the number of participants: in this case, VALE and SR-IOV can sustain 4 senders with only a per-packet performance drop of about 0% and 17.8%, respectively. On the contrary, *virtio*-based switches can still only distribute the same amount of resources over a more significant number of network flows.

From these results, we concluded that the number of packets mostly represents the major limitation of our SR-IOV NIC exchanged on the local network. In contrast, the most significant limitation of *virtio*-based switches is the capability of the CPU to move data from one application to another, which depends on the overall amount of bytes exchanged. Finally, VALE is affected by the same limitation of the other software-based virtual switches, but thanks to its distributed implementation it is possible to sustain higher traffic volumes without consuming an unreasonable amount of processing power (see also Sect. 4.6 for more details about performance and processing power).

Repeating scalability evaluations on multiple hosts ($L = remote$), we deployed up to 8 application pairs ($S = 8vs8$) transmitting data from one host to the other one (Fig. 3e). Figures 4g and 4h show that the selected packet size strongly influences the outcome. Similarly to single-flow remote test results, the system's major bottleneck is represented by the limited throughput of the Ethernet line rate when bigger packets are used (256 bytes and above). When exchanging smaller packets, the CPU becomes unable to efficiently move big numbers

[10] This limitation corresponds to the processing power reserved for each worker thread they spawn for software virtual switches, while for SR-IOV devices, it is an intrinsic characteristic of their hardware implementation.



(a) $(L = local,\ B = 4,\ R = 1000)$

(d) $(L = remote,\ B = 4,\ R = 1000)$

(b) $(L = local,\ B = 32,\ R = 1000)$

(e) $(L = remote,\ B = 32,\ R = 1000)$

(c) $(L = local,\ B = 128,\ R = 1000)$

(f) $(L = remote,\ B = 128,\ R = 1000)$

Fig. 5. Average round-trip latency obtained varying system configuration and virtual switch used to connect client/server applications deployed in LXC containers. Plots on the left refer to tests performed on a single-host, while plots on the right involve two separate hosts.

of packets from and to the NIC, especially when software-based solutions are adopted.

4.5 Latency Performance Evaluations

For the latency dimension, we evaluated the round-trip latency between a pair of client and server applications, depending on the network traffic and the network infrastructure, to estimate the per-packet processing overhead introduced by each different solution. In particular, our focus is to evaluate how each technology distributes its packet processing costs over multiple packets when increasing the number of packets in each burst. To carry out these experiments, we deployed a single pair ($S = 1vs1$) of client/server applications on either a single ($L = local$) or multiple hosts ($L = remote$). In each test, benchmark applications use a relatively low packet sending rate, enough so that there can be no interference between the processing of a burst of packets and the following one.

First, we deployed both the client and the server on the same host (Fig. 3c), varying the burst size from 4 to 128 packets per burst and the packet size from 64 to 1500 bytes:

$$(D = latency,\ L = local,\ S = 1vs1,\ R = 1000)$$

In all our tests, the performance that we registered for Snabb was considerably worse than the ones achieved by other solutions; for example, the minimum average latency registered for Snabb is about 64 µs, even when other solutions in similar working conditions averaged well under 40 µs. Since this behavior is repeated in all our tests, regardless of which network parameters are applied, Snabb will not be discussed further.

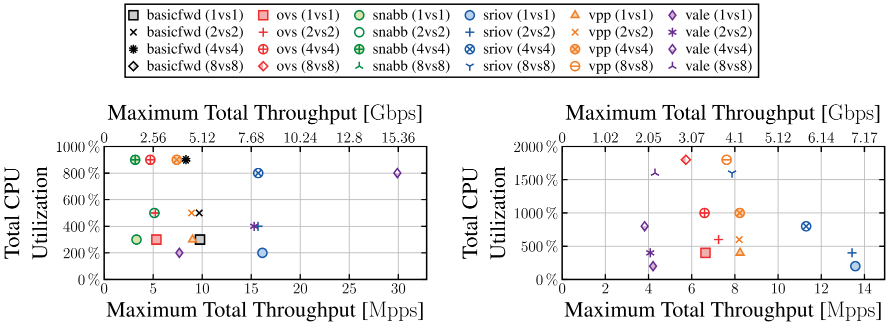
Figures 5a to 5c show that SR-IOV is the only solution that cannot provide single-digit microsecond round-trip latency even for small packet and burst sizes, achieving at best about 12.4 µs. Increasing the burst size to 32 packets per burst improves its performance, enabling SR-IOV to outperform all *virtio*-based virtual switches, although VALE remains the most lightweight solution. From these results, we inferred that the variation of the burst size has a lower influence on the SR-IOV and VALE performance; for this reason, they are both suitable solutions that can be used with more bursty traffic.

We repeated the same evaluations by deploying the server application on a separate host ($L = remote$, Fig. 3f). In this new scenario, SR-IOV unsurprisingly always outperforms OVS and VPP, as shown in Figs. 5d to 5f; in fact, software-based virtual switches introduce two new levels of indirection with respect to directly offloading all network traffic to the NIC: the two software instances, each running in their respective hosts, perform additional packet processing operations that contribute to the overall latency of each packet exchanged to the ones already performed in hardware by the very same SR-IOV device. On the other hand, VALE's performance is less subject to change when varying packet or burst sizes, achieving consistently around 85 µs round-trip latency on average.

4.6 Performance and Computational Requirements

Finally, we compared the cost of these high-performance networking solutions in terms of computational power required. For this purpose, during our throughput evaluations we configured the sender and receiver applications to measure the consumed computational power, by comparing the CPU time executed by the process against the actual time.

For most of these scenarios the overall CPU utilization can be obtained by some simple considerations. First of all, both DPDK and Netmap achieve maximum performance when applications continuously poll the network driver. For this reason, each sender/receiver application consumes exactly 100% of the CPU time when running at maximum capacity. Software-based user-space virtual switches (i.e. DPDK Basic Forwarding Sample Application, OVS, Snabb,



(a) $(L = local,\ B = 32,\ P = 64)$

(b) $(L = remote,\ B = 32,\ P = 64)$

Fig. 6. Total CPU utilization vs maximum throughput measured varying the number of network flows and type of virtual switch used to connect sender/receiver applications deployed in LXC containers.

and VPP) are each implemented as a user-space process which also continuously polls network drivers for maximum performance. Hence, they each consume an entire CPU for each worker thread they spawn. On the contrary, SR-IOV and VALE do not require additional CPUs, albeit for two very different reasons: SR-IOV does not utilize any CPU at all, since it is an hardware offload solution; VALE runs inside each sender/receiver process, moving packets whenever an application executes a system call, as mentioned in Sect. 4.4.

Experimental results confirm these formulations, as shown in Fig. 6. Since for each of our tests the total CPU utilization for each solution depends only on the number of participants required by the test (e.g. $1vs1$, $2vs2$, and so on), we show only the maximum throughput performance registered during each test with the associated CPU utilization. It is implicit that given a certain system configuration, sending a smaller number of packets per second would result in worse throughput performance, but it would not affect CPU utilization when polling techniques are used. Figure 6 shows the clear advantage in terms of CPU costs of SR-IOV and VALE against the other switching technologies: in each system configuration, they always require one less CPU than the others for local and two less CPUs for multi-host communications.

It must be noted that most of these solutions support some form of interrupt coalescing techniques, similarly to the Linux New API (NAPI) [25], to reduce the overall CPU utilization at the cost of reduced throughput/latency performance. For example, Netmap API provides blocking access to NICs, allowing processes to suspend while waiting for the device to be ready again, thus reducing CPU utilization [22]. DPDK can also be used in combination with interrupts [11], 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 greater latency when interrupts must be disabled to revert to polling mode, when the first packet of a burst is received.

As a final note, both OVS and VPP support multi-threaded packet forwarding by spawning multiple worker threads on separate cores and assigning each worker thread a subset of the virtual switch ports. When using this mode, benchmarking performance is directly influenced by the placements of the various applications. If the same worker thread manages multiple sender applications, performance is the same as shown in Figs. 4e and 4f, at the cost of consuming additional CPUs if assigning their receivers to other continuously polling worker threads. On the other hand, when spreading each sender application to a separate worker thread, both OVS and VPP can scale the total traffic linearly with network flows. The placement of receiver applications is entirely irrelevant from this perspective. However, it must be noted that using this mode has a considerable cost in terms of computational requirements since each worker thread consumes the totality of a CPU core, imposing a hard limit on the number of VNFs that can be deployed on a single machine.

5 Related Work

The proliferation of different technologies to exchange packets among applications deployed in virtualized environments has created the need for new tools to evaluate virtual switching solutions' performance with respect to throughput, latency, and scalability. For this reason, various works in the research literature addressed the problem of network performance optimization for VMs and containers, often analyzing the problem from different points of view.

A comparison among standard POSIX sockets and more modern *kernel bypass* frameworks like DPDK and Remote Direct Memory Access (RDMA)[11] focusing on round-trip latency between two directly connected hosts [11] showed that both DPDK and RDMA significantly outperform POSIX UDP sockets. In the study, DPDK and RDMA were the only ones able to achieve single-digit microsecond latency, with the drawback that applications must continuously poll the physical devices for incoming packets, leading to high CPU utilization.

Another work [14] compared qualitatively and quantitatively common high-performance networking setups, including SR-IOV, Snabb, OVS (with DPDK), and Netmap, measuring throughput and relative CPU utilization when deploying two VMs on either a single or two multiple directly connected hosts. Their evaluations concluded that, in their setups, Netmap could reach up to 27 Mpps (when running on a 4 GHz CPU), outperforming SR-IOV, due to the limited bandwidth of its hardware switch.

A previous comparison among high-performance networking technologies [10] analyzed the performance of three different frameworks: DPDK, Netmap, and PF_RING[12]. The analysis showed that two major bottlenecks may limit networking performance between two hosts: CPU capacity and NIC maximum transfer rate. Characteristics of network traffic (like packet or burst sizes) can influence whether one or the other represents the dominating bottleneck: when the

[11] http://www.rdmaconsortium.org.

[12] https://www.ntop.org/products/packet-capture/pf_ring/.

per-packet processing cost is kept low, the NIC maximum transfer rate is what imposes a cap on performance; on the other hand, as processing cost increases the CPU becomes increasingly more loaded, until it reaches a maximum packet processing rate. DPDK achieved the highest throughput in terms of packets per second, independently from the burst size; on the contrary, Netmap reached its highest throughput only when at grouping least 128 packets in each burst, and even then, it could not reach performance similar to DPDK or PF_RING.

The authors of a more recent work addressed the scalability of various virtual switching solutions against the number of VMs deployed on the same host [19], comparing VPP and OVS against SR-IOV. From their evaluations, they concluded that SR-IOV could sustain a more significant number of VMs with respect to its software-based counterparts, achieving almost linear throughput scalability for the number of VMs. Both OVS and VPP were only able to scale the total throughput up to a certain plateau, which depended on the number of CPU resources reserved for each virtual switch: the global resources allocated to each virtual switch were distributed equally among VMs, with a per-VM performance degradation that increased with the number of parallel VMs.

The same authors of this paper presented a preliminary work [3] that compared various virtual switching techniques based on *kernel bypass* for inter-container communications. That evaluation was limited to techniques that bypass the Linux kernel and only a single unidirectional packet flow on a single host. The results indicated that offloading traffic to the SR-IOV interface was the most suitable among the tested solutions. Another work comparing a much broader number of test cases and working conditions has also been presented by the same authors [4], deploying benchmarking applications on either one or multiple machines and presenting a new set of tools useful to repeat the experiments in other scenarios conveniently.

This paper constitutes an extension of our prior work [4] just described above. In this paper, we provided a more in-depth description of high-performance networking frameworks, including also Netmap, that does not rely entirely on *kernel bypass* mechanisms. We described how the original framework has been extended to support also Netmap and its virtual switch VALE as suitable inter-container communication mechanisms. With the new extended framework, we now provided a broader comparative analysis of the obtained networking performance, including VALE in our experiments. Finally, we also analyzed the computational requirements of each of the networking solutions included in the framework, which was missing from [4].

6 Conclusions and Future Work

This paper presented an extension to a software framework for the comparative evaluation of virtual networking solutions commonly used in the NFV industrial practice. With the extended framework, we evaluated the performance and computational demands of interconnecting VNFs, both on a single or multiple hosts. Results obtained on our reference machine show that SR-IOV or Netmap's virtual switch, VALE, obtain superior performance against the competition when networking on a single host; this is true not only in terms of throughput, latency, and scalability of the virtual network but also in terms of the processing power needed to perform packet forwarding at high rates. For inter-host communications, the CPU's limitations or the limited throughput of the underlying physical layer represent the major bottleneck for each of the tested solutions, at least on our reference hardware.

In the future, we plan to support additional virtual networking solutions, like SoftNIC [12] or PF_RING, as well as other virtual switches, like FastClick [6] or BESS [12]. Finally, we plan to implement support for one or more high-performance networking frameworks to real state-of-the-art NFV applications, like the ones developed for the OpenAirInterface project [16], and evaluate the potential performance improvements of these frameworks for real industrial applications, rather than emulating their behavior.

Acknowledgements. The authors would like to thank professor Giuseppe Lettieri from the University of Pisa for the timely help provided during the integration of Netmap in the framework.

References

1. Impressive packet processing performance enables greater workload consolidation. White Paper, Intel (2012). https://media15.connectedsocialmedia.com/intel/06/13251/Intel_DPDK_Packet_Processing_Workload_Consolidation.pdf
2. Open vSwitch enables SDN and NFV transformation. White Paper, Intel 2015. https://networkbuilders.intel.com/docs/open-vswitch-enables-sdn-and-nfv-transformation-paper.pdf
3. Ara, G., Abeni, L., Cucinotta, T., Vitucci, C.: On the use of kernel bypass mechanisms for high-performance inter-container communications. In: Weiland, M., Juckeland, G., Alam, S., Jagode, H. (eds.) ISC High Performance 2019. LNCS, vol. 11887, pp. 1–12. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34356-9_1
4. Ara, G., Cucinotta, T., Abeni, L., Vitucci, C.: Comparative evaluation of kernel bypass mechanisms for high-performance inter-container communications. In: Proceedings of the 10th International Conference on Cloud Computing and Services Science. SCITEPRESS - Science and Technology Publications (2020)
5. Barach, D., Linguaglossa, L., Marion, D., Pfister, P., Pontarelli, S., Rossi, D.: High-speed software data plane via vectorized packet processing. IEEE Commun. Mag. **56**(12), 97–103 (2018)
6. Barbette, T., Soldani, C., Mathy, L.: Fast userspace packet processing. In: Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS 2015, USA, pp. 5–16 (2015)
7. Barik, R.K., Lenka, R.K., Rao, K.R., Ghose, D.: Performance analysis of virtual machines and containers in cloud computing. In: 2016 International Conference on Computing, Communication and Automation (ICCCA). IEEE, April 2016
8. Dong, Y., Yang, X., Li, J., Liao, G., Tian, K., Guan, H.: High performance network virtualization with SR-IOV. J. Parallel Distrib. Comput. **72**(11), 1471–1480 (2012)
9. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and Linux containers. In: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), March 2015

10. Gallenmüller, S., Emmerich, P., Wohlfart, F., Raumer, D., Carle, G.: Comparison of frameworks for high-performance packet IO. In: ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), May 2015
11. 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), July 2018
12. Han, S., Jang, K., Panda, A., Palkar, S., Han, D., Ratnasamy, S.: SoftNIC: A software NIC to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015
13. Jeong, E., et al.: mTCP: a highly scalable user-level TCP stack for multicore systems. In: 11th USENIX Symposium on Networked Systems Design and Implementation, pp. 489–502 (2014)
14. Lettieri, G., Maffione, V., Rizzo, L.: A survey of fast packet I/O technologies for network function virtualization. In: Kunkel, J.M., Yokota, R., Taufer, M., Shalf, J. (eds.) ISC High Performance 2017. LNCS, vol. 10524, pp. 579–590. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67630-2_40
15. Maffione, V., Rizzo, L., Lettieri, G.: Flexible virtual machine networking using netmap passthrough. In: 2016 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN). IEEE, June 2016
16. Nikaein, N., Marina, M.K., Manickam, S., Dawson, A., Knopp, R., Bonnet, C.: OpenAirInterface. ACM SIGCOMM Comput. Commun. Rev. **44**(5), 33–38 (2014)
17. Paolino, M., Nikolaev, N., Fanguede, J., Raho, D.: SnabbSwitch user space virtual switch benchmark and performance optimization for NFV. In: IEEE Conference on Network Function Virtualization and Software Defined Network, November 2015
18. Pfaff, B., et al.: The design and implementation of Open vSwitch. In: 12th USENIX Symposium on Networked Systems Design and Implementation, pp. 117–130 (2015)
19. 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 - ICPE 2018. ACM Press (2018)
20. Rizzo, L.: Netmap: A novel framework for fast packet I/O. In: 2012 USENIX Annual Technical Conference (USENIX ATC 12), Boston, MA, pp. 101–112. USENIX Association (2012)
21. Rizzo, L.: Revisiting network I/O APIs: the Netmap framework. Queue **10**(1), 30 (2012)
22. Rizzo, L., Landi, M.: Netmap: memory mapped access to network devices. ACM SIGCOMM Comput. Commun. Rev. **41**(4), 422 (2011)
23. Rizzo, L., Lettieri, G.: VALE, a switched ethernet for virtual machines. In: Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies - CoNEXT 2012. ACM Press (2012)
24. Russell, R., Tsirkin, M.S., Huck, C., Moll, P.: Virtual I/O Device (VIRTIO) Version 1.0. Standard, OASIS Specification Committee (2015)
25. Salim, J.H., Olsson, R., Kuznetsov, A.: Beyond Softnet. In: Annual Linux Showcase & Conference, vol. 5, pp. 18–18 (2001)
26. Yasukata, K., Honda, M., Santry, D., Eggert, L.: StackMap: low-latency networking with the OS stack and dedicated NICs. In: USENIX Annual Technical Conference (USENIX ATC 16), Denver, CO, pp. 43–56, June 2016