Exploiting DPDK in Containerized Environment with Unsupported Hardware

Leila Askari, Payam Majidzadeh, Omran Ayoub, Massimo Tornatore
Department of Electronics, Information and Bioengineering
Politecnico di Milano, Milan, Italy
E-mail: firstname.lastname@polimi.it

***Abstract*—Network virtualization is an attractive technique to deploy new network services in an agile and cost efficient way. However, since virtualization imposes additional performance overhead (e.g., disk input/output virtualization overhead), which results in higher latency for service deployment, new virtualization frameworks that accelerate the performance of virtualized network functions are becoming available. Among these frameworks, Intel has proposed a set of libraries to accelerate packet processing and to remove additional delays caused by context switching from kernel space to the user space in computing servers. In their current public releases, DPDK libraries are guaranteed to work only if specific requirements in terms of supported hardware and Network Interface Card (NIC) are satisfied. However, the supported physical NICs are usually found in high performance servers. Therefore, it will be more cost-effective (and useful for research activities) to be able to deploy DPDK using any physical NIC.**

In this study, we aim at demonstrating that DPDK experimental activities can be run also on unsupported hardware (i.e., hardware that is not originally supported by DPDK libraries). To demonstrate this, we propose various component stack and implement different testbed setups to exploit DPDK in our lab environment using generic servers. Results obtained show that, even on quite outdated and baseline equipment available in our lab environment, it has been possible to run DPDK and, using DPDK, we were able to reach the line rate assigning at least two CPU cores to DPDK application and using packet size greater than 256 Bytes.

***Index Terms*—DPDK, container, Unsupported hardware**

I. Introduction

Using Network Function Virtualization (NFV), operators are able to move their network functions from traditional dedicated hardware devices ("middleboxes") to virtual machine (VM) that can be run on commodity servers. These servers can then be even co-located with routers and switches, enhanced with computing capabilities, that allow to agilely instantiate network functions in several network locations (usually called NFV-capable nodes). The resulting network infrastructure must be capable of running network functions while satisfying stringent requirements, such as processing packets of virtualized network functions at network speed. However, NFV has its own disadvantages, mainly in terms of overhead [1] which limits performance. Recently, container technology has gained traction as it introduces less overhead, providing higher performance and easier deployment with respect to classical virtualization technologies.

As Internet continues to evolve in terms of new services (e.g., Augmented Reality that has strict requirement in terms of latency), traditional network devices which consider traditional methods to process packets and data cannot be anymore considered. In other words, the traditional traffic processing methods, using kernel network protocol stack, imposes context switching delay by constantly switching from kernel to user space and results in bottlenecks in data transmission.

To avoid the drawback of this context switching, a possible solution is to move networking functionalities from kernel space to the user space. As an example, Google SNAP [2], as a user space networking system, implements various network functions and enables fast development and deployment of networking features. Another software tool, that is currently attracting lot of attention, is the Intel's Data Plane Development Kit (DPDK). DPDK offers a set of user input output libraries and drivers, and has been demonstrated to enable packet processing at the line rate. DPDK is able to bypass the kernel and hence avoids significant system issues that degrade the performance of packet processing required to run specific network services. However, DPDK libraries only support specific Network Interface Cards (NICs) that are usually found in high performance servers [3]. Therefore, it is more cost-effective (specially for research purposes) to be able to deploy DPDK with any unsupported hardware.

In this paper, we first propose a component stack that allows DPDK to work over unsupported hardware. Then, we implement this framework in various testbed setups to exploit DPDK in different settings allocating different amount of resources (e.g., number of CPU cores) to the DPDK application and considering different packet sizes. We start from deploying DPDK inside a single VM. Then, we provide a testbed setup in which we are able to use DPDK with unsupported physical NIC to perform layer-two and layer-three packet forwarding. We demonstrate that DPDK shows a satisfactory performance in terms of throughput and packet processing when run on a generic hardware platform (e.g., considering unsupported NICs).

The rest of the paper is organized as follows. In Section II we provide an overview of related works. In Section III we introduce background concepts (focusing on DPDK sample applications) we used in our testbed setup. In Section IV we present the component stack that can be used to deploy DPDK on unsupported hardware and we describe the testbed setups considered in our analysis. Section V discusses the results related to our experiments. Finally, in Section VI we conclude the paper.

II. Related Work

DPDK has recently received a lot of attention in literature. Ref. [3] proposes a packet-capture method using DPDK to reduce packet loss and improve packet-processing rate. However, since DPDK has specific requirement in terms of hardware, e.g. NICs, their experiment is implemented on a high performance server. In Ref. [4], a custom transport protocol, is implemented over DPDK to improve the transport protocol performance. The results obtained show performance improvement achieved thanks to DPDK with respect to standard Linux network stack, however, since the protocol is not able to exploit full DPDK capabilities, the line rate is not achieved. Ref. [5] designs and implements a network emulator based on DPDK which is able to emulate packet loss in the network and has higher accuracy with respect to NetEm which is a popular network emulator in Linux. Ref. [6] proposes an evaluation framework for Active Queue Management (AQM) which tries to improve the performance of network. The AQM algorithms are described in P4 language and are compiled with a DPDK-based P4 compiler. Ref. [7] proposes a DPDK-based framework to achieve elastic scaling of computing resources allocated to network functions using a single commodity server while Ref. [8] implements a Deep Packet Inspection (DPI) network function using DPDK and compares its performance with the scenario in which DPI is installed over Linux kernel network stack. Experimental results demonstrate a better packet throughput performance obtained using DPDK. Moreover, Ref. [9] demonstrates a design that improves the VM-to-VM communication performance using port mirroring in DPDK-enabled Open vSwitch (OvS), and Ref. [10] proposes a DPDK-based service chaining framework, which improves the performance of service chain by a factor of two. DPDK is also used to build architectures [11] and frameworks [12] and to perform experiments in a number of works in the literature. As an example, in Ref. [13] a system is implemented to build virtual network functions from reusable loosely-coupled components, implemented as DPDK processes. In this system DPDK is used to provide packet input/output and memory sharing between the components.

However, in all these existing studies, DPDK is implemented over high performance servers and implementation over unsupported hardware is not investigated. In our work, we provide different testbed setups to evaluate the performance of DPDK deployed over unsupported hardware.

III. Background Concepts on Intel's DPDK

DPDK offers libraries designed to accelerate the processing of packet workloads running on different CPU architectures. The key features of DPDK is to receive and send packets within the minimum number of CPU cycles, design fast packet capture algorithms (tcpdump-like) and run third-party fast path stacks. For example, many packet processing capabilities have been optimized for up to a hundred million frames per second using 64-Byte packets with a PCIe NIC [14].

DPDK also comes with some pre-developed sample applications which are designed to test its capabilities and can be used in test environments like ours. Some of these sample applications are [14]:

- *pktgen* (Packet Generator), a software based traffic generator powered by DPDK fast packet processing.
- *testpmd* application, that can be used to test DPDK in a packet-forwarding mode.
- *l3fwd* a simple example of packet processing that performs layer-3 forwarding.

A. Container Technology and DPDK

Container technology is a method for packaging an application and its dependencies in such a way that it can be run isolated from other processes. Container technologies with specific container software (including the popular choices of Linux Container, Docker, Apache Mesos, rkt) have been embraced by major public cloud providers such as Amazon Web Services, Microsoft Azure and Google Cloud Platform. Containers put together software and its dependencies (e.g., libraries, binaries, and configuration files) in a package (the "container") that can be migrated to another machine as a unit. This will allow avoiding incompatibility issues when running the application on different operating system or different hardware [15]. Since containers are more efficient than VMs in terms of resource usage and deployment time, it is more cost effective to implement network functions inside containers. However, containers also use Linux kernel networking stack, hence, they also experience context switching delay. To overcome this drawback, DPDK can be used inside containers to accelerate container networking by bypassing Linux kernel network stack.

IV. Testbed setup

One of the main limitations in deploying DPDK in a physical network is the necessity of using DPDK-supported NICs. This means that, currently, implementing and running network functions with DPDK is restricted to specific hardware. A number of paravirtualized NICs are introduced by DPDK [14] to solve this issue, however, they are only designed for performing tests inside a VM. Therefore, running and testing DPDK applications in a physical network using paravirtualized NICs are challenging tasks as paravirtualized NICs are not designed to be bounded to physical NICs. In this study, we describe how we succeeded in deploying paravirtualized NICs over physical NICs which are not supported by DPDK. More specifically, we test the performance of DPDK applications (as testpmd) in our lab environment using generic hardware and with unsupported common NICs. The source code of our testbed can be found in [16].

We performed our evaluations in three different scenarios. In all the scenarios, a VMware ESXi hypervisor is run on top of our bare metal server and a VM running Ubuntu is installed on top of it. In the first scenario ("single-host"),

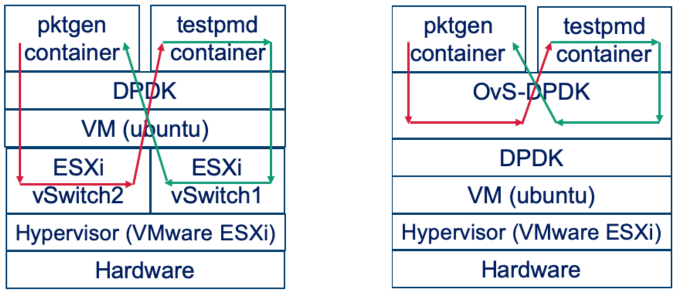

Fig. 1: Single-host scenario

we deploy DPDK in a totally virtualized environment. In the second scenario ("dual-host scenario"), we run DPDK in different physical hosts connected via unsupported NICs. In the third scenario ("DPDK in Chain"), we run two different DPDK applications on one host, and we then connect the two DPDK applications in a "chain of applications" to other two applications running in another host. In this scenario, we aim to mimic the deployment of DPDK applications at the edge of the network.

A. *Single-host scenario*

In the single-host scenario, we compare the line rate achieved by virtualized functions over two testbed setups each relying on a different virtual switch technology to connect the containers running the DPDK applications: i) one based on DPDK-enabled OvS and ii) one based on ESXi vSwitch. As the former natively supports the fastpath DPDK, we will use it as a reference in our evaluations. However, to be able to use unsupported NICs, we introduce the component stack using ESXi vSwitch (the virtual switch of VMware vSphere). Note that, the virtual network adaptor of VMware vSphere (VMXNET3) is defined as one of the supported virtualized network adaptors by DPDK. In the following, we first give a high level description of each testbed setup, and then explain in detail the steps followed to realize them. Note that, we used different containers for each DPDK application to achieve isolation of containers.

1) Single-host Scenario with OvS-DPDK: The setup of this testbed is shown on the right hand side of Fig. 1, and it is inspired from a series of videos from the Intel software github project [17], but with some required modifications to make it comparable with the other testbed setups of our study. The steps followed to realize this configuraton are as follows:

1) Build DPDK and OvS, and allocate system resources by allocating hugepages [1] and inserting user-space IO driver into kernel. Note that, since the CPU we used has Page Size Extension flag, hugepages of size 2M are supported [14] and we used 1024 of hugepages to run each application.
2) Configure and initialize the OvS database, set OvS parameters and launch it.
3) Create OvS bridge and ports and add flows between the virtual ports of the bridge.
4) Two containers are built and two DPDK applications, namely "pktget"and "testpmd"are instantiated providing them access to poll mode drivers (PMDs)[2]. We allocate two CPU cores for each application and 4 memory channels for *testpmd*. Note that, for each application we assigned a CPU core as the master core for command line parsing purposes.
5) Run DPDK applications inside containers that are running in privileged mode and each has access to the host hugepages and two of virtual NICs (vNICs) assigned to the VM. After that, packets are forwarded to port0 of *testpmd* container through the OvS-DPDK. Then, *testpmd* forwards the received packets to its other port that is connected through OvS-DPDK to port1 of pktgen.

2) Single-host scenario with ESXi: In this setup we exploit DPDK in lab environments with unsupported hardware while keeping a fair comparison between OvS with native support of DPDK and ESXi vSwitch technology. The key difference in this scenario with respect to the former one is the use of *ESXi vSwitch* instead of *OvS vSwitch*, and therefore the difference lays in the structure of these two virtual switches. Alongside with the fact that OvS has a native support for DPDK, the boldest differences between these two virtual switches is that OvS runs on top of operating system and DPDK, while ESXi vSwitch runs on top of a hypervisor, with the support of DPDK PMD (VMXNET3 vNIC). A high level scheme of this setup is shown in Fig. 1 to the left. Unlike OvS, ESXi does not support Openflow, hence, we are not able to separate the flows using Openflow. However, it is possible to build two different vSwitches with no uplinks to have separate switches that work in an isolated network with internal access. We build a VM that hosts two Docker containers and the containers communicate with each other through the ESXi vSwitches. Note that, in both vSwitches, the *promiscuous mode*, i.e., a mode that requests the NIC to pass all received traffic to the CPU, should be enabled.

The steps to set up this testbed are very similar to those in *Single-host Scenario with OvS-DPDK*, however, in addition to replacing OvS with ESXi vSwitch, there are three differences: i) after the first step, we need to specify the kernel module through which we launch VMXNET3 PMD to DPDK; ii) containers need to be launched granting them access to the Linux devices (specifically to the PMDs of the NICs allocated to each container); and iii) before running the DPDK applications at the last step, we need to configure containers to be connected to different vSwitches through different vNICs.

B. *Dual-host scenario*

In this section, we illustrate how to enable the same connectivity between applications seen in the single-host setup, but now in two remote hosts interconnected through unsupported

[1]Pages are physical and virtual contiguous blocks of memory, and hugepages are big pages that DPDK relies on to improve performance.

[2]A Poll Mode Driver consists of APIs, provided through the BSDdriver running in user space, to configure the devices and their respective queues

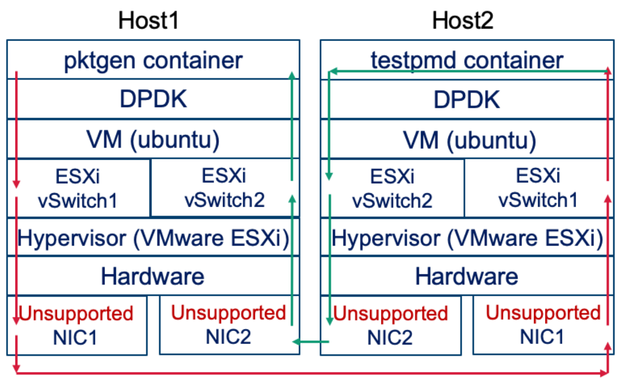

Fig. 2: Dual-host scenario with unsupported hardware

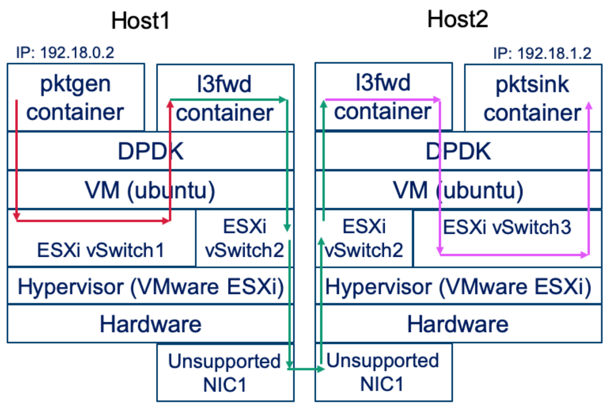

Fig. 3: DPDK in Chain scenario with unsupported hardware

NICs. Fig. 2 shows the high-level scheme of this setup. As shown in the Fig. 2, we launch *pktgen* container in the first host, and *testpmd* container in the second one.

The difference between this setup and the previous ones (Single-host Scenario with OvS-DPDK and Single-host scenario with ESXi) is that in this setup we need to manage the uplinks of the vSwitches, in which we have the connectivity between the ESXi hypervisors. We connect ESXi vSwitch1 from both hosts to the first DPDK-unsupported physical NIC, and ESXi vSwitch2 to the second DPDK-unsupported physical NIC. After that, we follow the same steps as those mentioned in previous section to run *testpmd* on one host and *pktgen* on the other host.

C. DPDK in chain scenario

In this scenario we are going to use two DPDK *l3fwd* containers that operate as virtual routers at the endpoint of each host and communicate with *pktgen*. Fig. 3 illustrates a high level scheme of this setup. Packets are generated in *pktgen* container in host1 with assigned IP address of “192.18.0.2”. ESXi vSwitch1 is connected to *pktgen*’s port 0 (the only port of this container) and to port 0 of *l3fwd* container. At the same time, port2 of this container is connected to ESXi vSwitch2 of host 1, and the vSwitch has a DPDK-unsupported uplink to ESXi vSwitch2 of host2. The second port of this vSwich is connected to port 0 of the container running *l3fwd* function in host 2. These two *l3fwd* containers emulate a virtual router. After that, port 1 of *l3fwd* container in host2 is connected to *pktsink*[3] container with IP address “192.18.1.2”of the same host through ESXi vSwitch3. In this scenario, the ESXi vSwitch2 of each of the hosts has the uplink with “unsupported NICs”, and it simulates the logical connection of two separated networks. ESXi vSwitch1 in host1 and ESXi vSwitch3 in host2 are without uplinks and they just manage the host internal networks.

The steps followed to set up this testbed are identical to those of the dual-host scenario, however, with two main differences. In this scenario, we need to run *pktgen* container with access to just one vNIC and the *l3fwd* container instead needs to have access to two vNICs, in which each port is pinned with queue 0 and is assigned a CPU core.

[3]pktsink have a similar behavior to pktgen however we have used different names since in this setup, the network function is used to sink packets.

V. Experimental Results

In this section, we present some numerical results obtained performing evaluations on the testbeds described in previous sections. All the results presented are obtained averaging the metrics over ten evaluations considering a confidence level of 95% with at most 5% confidence interval. The performance is measured in terms of throughput in Mpps, which represents the number of packets processed and forwarded.

Table I lists the different components used in our testbed setups. Note that, we used an unlicensed version of VMware ESXi to setup our testbeds. In all experiments, traffic is sent from transmitter port of the *pktgen* container at line rate of 1 Gbps, i.e., at the maximum bit rate that can be transmitted from the port, while varying the packet size. For example, transmitting packets of 1024 Bytes at 1 Gbps, 0.12 Mpps are transmitted with a packet arriving every 8350 ns, while for a packet size of 64 Bytes, 1.49 Mpps are transmitted with a packet arriving every 672 ns. In all cases, we plot the throughput, measured in terms of Mpps of each setup and compare it to the line rate.

TABLE I: Hardware and software details

Component	Description
Processor Type	Intel Core i7-6700 CPU@3.4GHz
Motherboard	Asus H110M-A/M.2
Logical Processors	8 logical cores
Memory	32 GB
Hypervisor	VMware ESXI, 6.7.0, 14320388
Operation System	Ubuntu Server 19.04
Memory Allocated to VM	24 GB
Container	Docker version 18.09.7, build 2d0038d
DPDK	dpdk-19.05
vSwitch for ESXi	ESXi Standard vSwitch, 6.7.0
vSwitch for OvS	Open vSwitch 2.6.1
vNIC for ESXi	VMXNET3 and virtio (for OvS)
vNIC for OvS	virtio
pNIC	TP-Link TG-3468 1Gbps
Packet Generator Application	pktgen-dpdk-pktgen-19.10.0

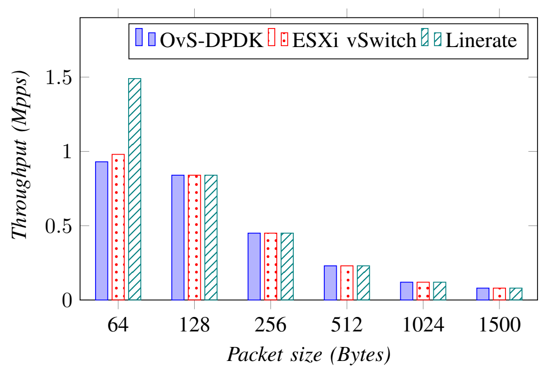

Fig. 4: Single-host scenario, OvS-DPDK vs. ESXi vSwitch

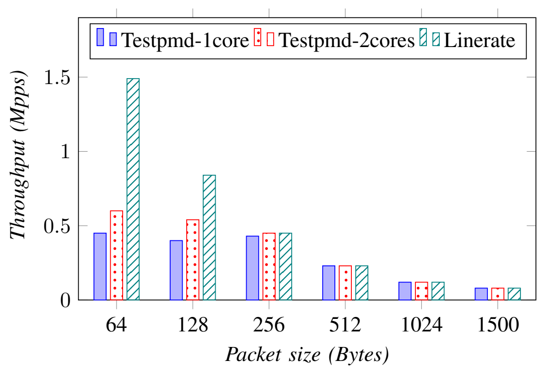

Fig. 6: Throughput comparison of Testpmd with 1 and 2 CPU cores in Dual-host scenario with unsupported hardware

A. Single-host scenario: ESXi vs. OvS

We first compare the two single-host scenarios. Fig. 4 shows the throughput in Mpps obtained for each of the single-host scenarios, i.e., of OvS-DPDK and of ESXi vSwitch, assigning 2 CPU cores to *testpmd* application with respect to packet size (in Bytes). Results show that, for packet sizes equal to 128 Bytes and larger, both setups reach the line rate. Only for a packet size of 64 Bytes, which is the most challenging case due to the high number of packets that need to be processed, the line rate is not reached. We speculate that more CPU cores are needed to be allocated to process this high number of packets. Moreover, we note that ESXi performs slightly better than OvS-DPDK.

We now compare the performance of the single-host scenario with ESXi vSwitch in two cases, when allocating 1 CPU core and when allocating 2 CPU cores for the testpmd application, with the objective of evaluating impact of CPU resources in the performance of our testbed. Figure 5 plots the throughput (in Mpps) for the two cases with respect to packet size. Results show that for packet sizes of 64 and 128 Bytes, assigning 2 CPU cores results in a better performance than assigning only one, showing the importance of assigning two CPU cores to avoid excessive packet loss. For a packet size of 256 Bytes and larger, both core assignments show the same performance and reach the line rate. This means that for such packet sizes, and therefore for such overall number of packets, allocating one core is enough to guarantee optimal performance. Moreover, the experimental results demonstrate that, for 1 Gbps throughput, ESXi vSwitch merged by DPDK is powerful enough to boost the packet processing by reaching line rate speed with packets size higher than 128 Bytes in the case where two CPU cores are assigned to DPDK application, and higher than 256 Bytes when only 1 CPU core is assigned.

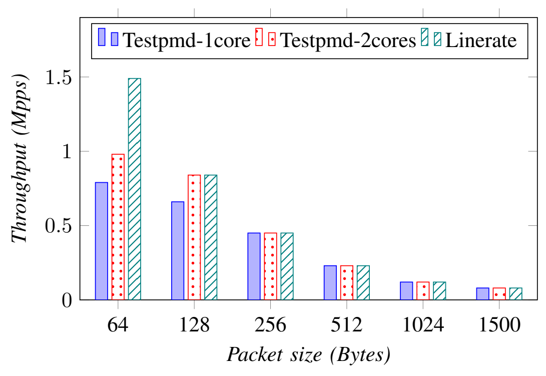

Fig. 5: Comparison of Testpmd with 1 and 2 CPU cores in single-host scenario with ESXi vSwitch

B. Dual-host scenario with ESXi: 1 CPU core vs. 2 CPU cores

We now evaluate the importance of assigning more resources in terms of CPU cores to DPDK applications in a dual-host scenario settings. Fig. 6 plots the throughput in Mpps for the two cases. Results show that line rate can be reached for packet size greater than 128 Bytes when assigning 2 CPU cores, however, by assigning only 1 CPU core, line rate can be reached only for packet size greater than 256 Bytes. This shows that assigning more CPU cores to the DPDK applications in a dual host scenario can improve throughput.

Therefore, one of the bottlenecks in achieving higher throughput is lack of enough CPU resources to process the packets received by DPDK application. Nevertheless, the results obtained demonstrate that a reasonable throughput is achieved in this setup that exploits unsupported NICs.

C. DPDK in chain

As for the DPDK in chain scenario, we performed the same analysis considering 1 CPU core and 2 CPU cores assigned to the DPDK application. In particular, we consider the *l3fwd* sample application, which extracts the necessary information from the IP header of the received packets and performs a look-up in the rule database to figure out where to forward the packets. Results of Fig. 7 show that when assigning 1 CPU core to the *l3fwd* application, the line rate is reached for sizes of packets equal to or greater than 512 Bytes, however when 2 CPU cores are assigned to *l3fwd*, the line rate is also reached at a size of packets equal to 256 Bytes, i.e., for a significant larger number of packets than the case of assigning 1 CPU core to *l3fwd*. For lower packets' sizes, i.e., for the cases of 64 and 128 Bytes, which represent more challenging cases with respect to

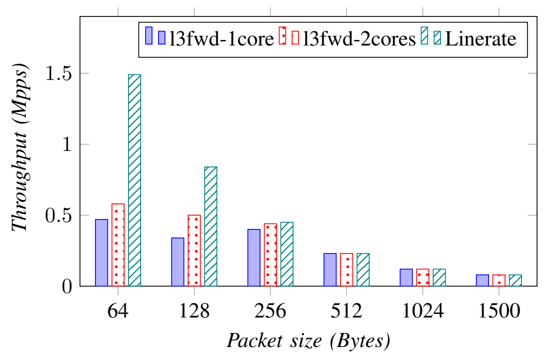

Fig. 7: Throughput comparison of l3fwd with 1 and 2 CPU cores in dual-hosts scenario with unsupported hardware

cases of larger packets' sizes due to greater number of packets that need to be processed, both assignments, with 1 and 2 CPU cores, result insufficient to process all incoming packets and to reach line rate. However, *l3fwd-2cores* processes around 30% more packets than *l3fwd-1core*.

It is worth mentioning that, like the previous testbed setup, a reasonable throughput is achieved using unsupported NICs for implementing packet forwarding functionalities using DPDK applications.

VI. Conclusion

In this paper we investigated solutions to deploy and run DPDK in a lab environment without using the supported NICs defined by DPDK. We presented four different testbed setups ranging from totally virtualized, that works only inside a single host, to running DPDK in chain using unsupported NICs. The first setup builds and runs DPDK enabled OvS, and on top of that using PMD, we run two separated containers. In the second setup, we modeled the same functionality this time on top of ESXi vSwitch (instead of OvS with DPDK support). In the third testbed setup we model the same structure but in two remote physical machines, connected using a direct link and using two unsupported NICs. Finally, we run DPDK in chain in our last testbed setup, by using two layer-3 DPDK applications in the remote hosts.

The results obtained show that using unsupported 1 Gbps NIC assigning two CPU cores to DPDK application in single-host scenario, we can reach to line rate with packets size equal and above 128 Bytes. While by assigning only one CPU core to DPDK application, line rate is achievable only for packets of size 256 Bytes and higher. As for the dual-host scenario, line rate can be reached by assigning two CPU cores to testpmd and having packets of 256 Bytes and higher. Correspondingly, in DPDK l3fwd application, which works as a virtual router, only for packets size of less than 512 Bytes the throughput of testpmd with two CPU core assigned is not equal to the line rate. Nevertheless, by assigning two CPU cores we can reach to line rate with packets size above 256 Bytes. As a future step, we aim at exploring the same testbeds considering 10 Gbps NIC and perform more precise analysis on the resources allocated to containers and host machine using different monitoring tools.

Acknowledgment

The work leading to these results has been supported by the European Community under grant agreement no.761727 Metro-Haul project.

References

[1] H. R. Kouchaksaraei and H. Karl, "Service function chaining across openstack and kubernetes domains," in *Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '19, p. 240–243.

[2] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat, "Snap: a microkernel approach to host networking," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 399–413.

[3] W. Zhu, P. Li, B. Luo, H. Xu, and Y. Zhang, "Research and implementation of high performance traffic processing based on Intel DPDK," in *2018 9th International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*, 2018.

[4] D. Syzov, D. Kachan, K. Karpov, N. Mareev, and E. Siemens, "Custom udp-based transport protocol implementation over DPDK," in *Proceedings. of the 7th International Conference on Applied Innovations in IT*, 2019.

[5] K. Sasaki, T. Hirofuchi, S. Yamaguchi, and R. Takano, "An accurate packet loss emulation on a DPDK-based network emulator," in *Proceedings of the Asian Internet Engineering Conference*, ser. AINTEC '19, p. 1–8.

[6] S. Laki, P. Vörös, and F. Fejes, "Towards an AQM evaluation testbed with P4 and DPDK," in *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, 2019, pp. 148–150.

[7] Z. Shen and Y. Zhang, "An NFV framework for supporting elastic scaling of service function chain," in *2018 IEEE 4th International Conference on Computer and Communications (ICCC)*, Dec 2018.

[8] M. Kourtis, G. Xilouris, V. Riccobene, M. J. McGrath, G. Petralia, H. Koumaras, G. Gardikis, and F. Liberal, "Enhancing VNF performance by exploiting SR-IOV and DPDK packet processing acceleration," in *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, Nov 2015.

[9] L. Wang, T. Miskell, P. Fu, C. Liang, and E. Verplanke, "OVS-DPDK port mirroring via NIC offloading," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020, pp. 1–2.

[10] A. Leivadeas, M. Falkner, and N. Pitaev, "Analyzing service chaining of virtualized network functions with sr-iov," in *2020 IEEE 21st International Conference on High Performance Switching and Routing (HPSR)*, 2020, pp. 1–6.

[11] F. Slyne, D. Coyle, J. Singh, R. Sexton, B. Ryan, R. Giller, M. O'Hanlon, and M. Ruffini, "Experimental demonstration of multiple disaggregated olts with virtualised multi tenant dba, over general purpose processor," in *2020 Optical Fiber Communications Conference and Exhibition (OFC)*, 2020, pp. 1–3.

[12] M. Adeppady, M. K. Singh, and B. R. Tamma, "ONVM-5G: a framework for realization of 5G core in a box using DPDK," *CSI Transactions on ICT*, vol. 8, no. 1, pp. 77–84, Mar 2020. [Online]. Available: https://doi.org/10.1007/s40012-020-00275-7

[13] S. R. Chowdhury, Anthony, H. Bian, T. Bai, and R. Boutaba, "A disaggregated packet processing architecture for network function virtualization," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 6, pp. 1075–1088, 2020.

[14] DPDK.org, "DPDK homepage," https://www.dpdk.org, Nov. 2019.

[15] J. DeMuro, "Containers," https://www.techradar.com, Nov. 2019.

[16] "DPDK-Unsupported-NIC," https://github.com/PayamMajidzadeh/DPDK-Unsupported-NIC.

[17] I. S. Inc., "OvS DPDK lab," http://bit.ly/2sWyolg, Nov. 2019.