Transparent Optimization of Inter-Virtual Network Function Communication in Open vSwitch

Transparent Optimization of Inter-Virtual Network Function Communication in Open vSwitch

Mauricio Vasquez Bernal*, Ivano Cerrato*, Fulvio Risso*, David Verbeiren†
*Politecnico di Torino, Dept. of Computer and Control Engineering, Torino, Italy
†Intel Corporation NV/SA, Kontich, Belgium
*{mauricio.vasquez, ivano.cerrato, fulvio.risso}@polito.it; †david.verbeiren@gmail.com

***Abstract*—This paper proposes an architecture that can optimize inter-VM communication in an NFV environment through the creation of direct channels between virtual machines. Particularly, our prototype can transparently optimize the data transfer between virtual machines running DPDK applications by dynamically recognizing the existence of point-to-point connections in the traffic steering rules, reverting back to the traditional VM-to-switch-to-VM approach when the optimization is no longer possible. This paper demonstrates the huge advantages of this architecture and the possibility to implement it with localized modifications mainly in Open vSwitch, without touching the applications inside the VMs.**

***Keywords*-NFV; Open vSwitch; DPDK; performance; inter-VM communication**

I. Introduction

Network Function Virtualization (NFV) [1] transforms many network functions (e.g., NAT, firewall) in software images executed on standard high-volume servers. Complex services can be delivered by rearranging multiple Virtual Network Functions (VNFs) in arbitrary *graphs* (Figure 1(a)), with multiple VNFs often executed on a single physical server. Usually, VNFs are instantiated as virtual machines (VMs)[1], while the traffic steering is carried out by a virtual switch (vSwitch) that classifies and forwards the packets according to specific rules sent through OpenFlow [3] messages, as shown in Figure 1(b).

Figure 1(a) shows a generic graph that contains both point-to-point (*p-2-p* in this paper) and point-to-multipoint links. While the latter require the vSwitch to classify and send each packet to the proper next VNF, p-2-p links, which are definitely more common in current service graphs, could be implemented by a direct communication path, hence taking the vSwitch out of that portion of the data plane. This, may result in higher throughput and lower latency, as well as in lower resource consumption thanks to the CPU saved by avoiding a further pass in the vSwitch.

Starting from this consideration, this paper proposes an architecture, called "*direct VM2VM*", that optimizes inter-VNF communications by setting up a direct connection between two VMs, hence allowing the traffic to bypass the vSwitch in case of p-2-p links. Our architecture differs from other existing proposals (e.g., [4] and [5]) because it has the capability to accelerate *transparenty* and *dynamically* the packets exchange between the VMs, and it is integrated in a *widespread* vSwitch.

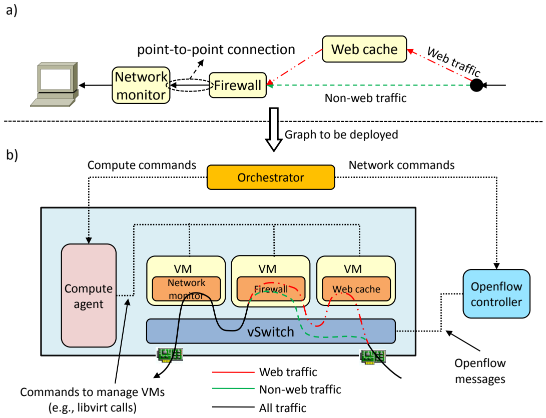


Figure 1. Traffic crossing several VNFs: (a) the "abstract" service graph; (b) its implementation on a server.

Transparency refers to the possibility for an application to exploit the advantages of the *direct VM2VM* technology without even knowing it is there, and for an OpenFlow controller to attach to a vSwitch without noticing it has been modified. In fact, most of the extensions needed by this technology are kept in the vSwitch, with minimal modifications in few other components.

Dynamicity refers to the capability to either establish a direct VM-to-VM channel or return to a traditional VM-to-vSwitch-to-VM path on the fly, based on the run-time analysis of the graph(s) that is being instantiated or modified.

Finally, the *direct VM2VM* technology has been integrated in a *widespread* vSwitch, namely Open vSwitch (OvS) [6]; particularly, it extends the version of OvS based on the Data Plane Development Kit (DPDK) [7], which exploits the optimized packet processing capabilities of that framework to achieve high throughput on standard high-volume hardware. For the same reason, this paper focuses on VMs that *execute*

[1] Although also lightweight containers such as Docker [2] can be used to run VNFs, they are not considered in this paper. Then, in the following, the terms VNF and VM will be used interchangeably.

DPDK-based network applications; in fact, we expect that in a near future NFV applications will leverage the power of optimized frameworks such as DPDK for most of the low-level packet processing tasks.

This paper is structured as follows. Section II analyzes the related works, while Section III provides an overview of the technologies exploited in our work. Section IV presents the prototype architecture, while experimental results are shown in Section V. Finally, Section VI concludes the paper and draws our future plans.

II. Related work

Several works aim at optimizing the communication between VMs executed on the same server, often through the creation of a direct channel between such VMs.

At the best of our knowledge, the closest works to our *direct VM2VM* architecture are [8] and [4] proposed by Intel, and `ptnetmap` [5]. Particularly, while Intel is working on a traffic bypass mechanism based on an extension of Virtio [9], `ptnetmap` allows netmap-based applications [10] (running in VMs) to transparently use different types of port; hence they can be connected to physical NICs, to the VALE [11] vSwitch or to a netmap pipe, which is a direct channel between two netmap-based applications.

Although these proposals present similarities with our work (e.g., both of them create a direct channel between VMs), important differences exist with respect to the *direct VM2VM* architecture. First, our proposal is integrated into a widespread vSwitch (OvS) and is transparent to other components used in an NFV environment (e.g., OpenFlow controller, hypervisor). Second, it is able to accelerate the packets exchange between VMs by dynamically creating direct channels between them after recognizing the existence of point-to-point connections in the traffic steering rules. Third, our proposal is able to revert back to the traditional VM-to-switch-to-VM approach when the optimization is no longer possible.

Although also architectures such as ClickOS [12] (based on VALE and Click [13]), NetVM [14] (based on DPDK) and [15] aim at improving performance of virtualized services, they are orthogonal to our proposal. In fact, they optimize the data exchange between the VMs and the vSwitch, without considering direct channels between chained VNFs; hence, in these works all the packets leaving a VM enter into a vSwitch that classifies and forwards them through the proper (physical or VM) port.

Finally, it is worth mentioning some works that are not designed to operate in an NFV context, although they create direct channels between communicating VMs. XenSocket [16] defines a new socket that exploits a shared memory to bypass the network stack, but requires applications to be modified; XWay [17] gets the same result by modifying the internals of the TCP stack and hence supports unmodified applications. This also applies to XenLoop [18], which intercepts packets at the network layer and can send them via shared memory. Works described in [19] and [20] focus instead on High Performance Computing (HPC) and propose libraries that implement a message-passing paradigm based on memory shared between VMs residing on the same server.

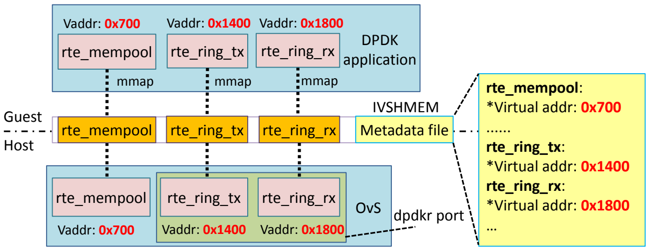


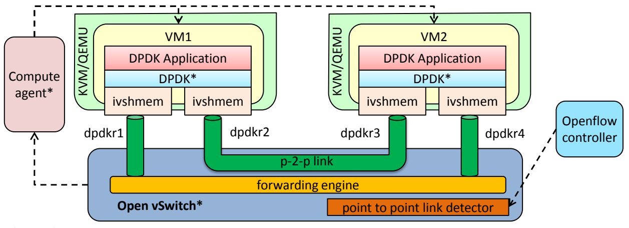
Figure 2. Sharing DPDK data structures between OvS and VMs.

III. Background

Among the several types of ports supported by OvS, `dpdkr` is considered the fastest one. It consists of a pair of DPDK queues (`rte_rings`) that contain pointers to packets; packets are in fact stored in a piece of memory (`rte_mempool`) allocated in huge pages, shared between OvS and the entities that exploit `dpdkr` ports. Consequently, `dpdkr` ports exchange packets in a zero-copy fashion. Moreover, this port does not have any notification mechanism, and hence entities connected to its ends (i.e., VNF and OvS) operate in polling mode.

A `dpdkr` port can connect the vSwitch to DPDK applications executed inside VMs. However, OvS exports to applications the `dpdkr` port as two `rte_rings` (RX and TX); hence applications have to explicitly write/read packets to/from such rings, and do not have any concept of network interface. `rte_rings` are provided to the VM through the Inter-VM Shared Memory (`ivshmem`) technology [21], a standard interface for the KVM hypervisor [22] that is used to share memory between the host and the guest operating systems. The memory region to be shared is exposed to the guest as a PCI Base Address Registers (BAR); then, applications can `mmap` [23] it into their own virtual address space.

DPDK includes a library [24] to create `ivshmem` devices; particularly, this library "inserts" in the device the data structures forming the `dpdkr` port to be shared between OvS and the VMs (e.g., `rte_rings`), and also some information about those data structures, such as their virtual address in the virtual memory of OvS (Figure 2). This information is used by DPDK in the guest OS to `mmap` the shared structures at the *same* virtual address used by OvS, which allows the application and OvS to exchange pointers to packets and de-reference them without any additional translation, which is a crucial factor in high performance environments.



* Modified to support transparent inter-VNF communication

Figure 3. Overall *direct VM2VM* architecture.

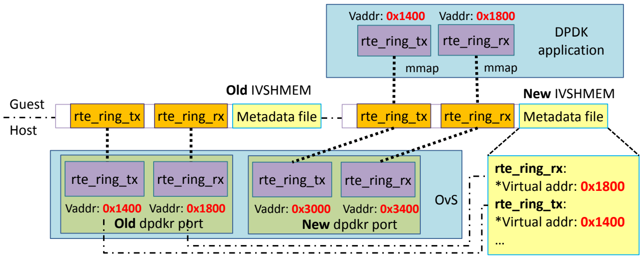


Figure 4. `ivshmem` device for port remapping.

IV. Direct VM2VM architecture

The DPDK-based applications we consider run inside VMs that are connected to OvS through `dpdkr` ports terminated in the forwarding engine of the vSwitch. This module handles packets according to the content of its forwarding table, which can be configured with OpenFlow `flowmods` [3]. All the connections among VMs are implemented in this way, regardless of the nature of the connection itself, i.e., p-2-p or point to multipoint.

In our proposal, shown in Figure 3, p-2-p links are implemented using two modified `dpdkr` ports connected directly to each other and detached from the OvS forwarding engine. Although OvS is no longer involved in moving packets exchanged by VMs, the two modified ports are still exported by OvS as standard `dpdkr` ports. This keeps the compatibility with external entities (applications, compute/network agents, OpenFlow controller), as they can continue to issue commands involving those ports as they usually do (e.g., get statistics, turn them on/off, etc.), without noticing any change in their actual implementation.

A. Detecting p-2-p links

We extended OvS with a new *p-2-p link detector* module (Figure 3), which analyses each rule (i.e., `flowmod`) received by the vSwitch in order to dynamically detect the creation or destruction of a p-2-p link between two `dpdkr` ports. In the current implementation, this operation requires a time $O(N)$ where `N` is the number of forwarding rules installed, but this algorithm could be replaced with a more efficient version in the future.

When a new p-2-p link is detected, OvS creates two `dpdkr` ports mapped on the same piece of memory, which contains a pair of `rte_rings` and that will be shared by both the communicating VMs (the `rte_ring` used as TX in one VM, has to be used as RX in the other VM, and vice versa). This way they are directly connected, and then packets can be exchanged without the intervention of the OvS forwarding engine[2].

B. Handling the new `ivshmem` device in the host

The `rte_rings` forming a `dpdkr` port are provided to the VM as part of an `ivshmem` device, together with a metadata file indicating, for each data structure, its address in the virtual memory of OvS (Section III).

As shown by comparing Figure 2 and Figure 4, we modified the metadata file included in the new `ivshmem` device so that it contains the virtual addresses of the *old* `rte_rings` used by OvS, and not those used for the new rings contained in the device itself. This allows DPDK to `mmap` the new `rte_rings` at the same virtual address of the old ones, so that the application can continue to work without realizing that the `rte_rings` are changed (more details in Section IV-C).

After being created, the new `ivshmem` device has to be connected to the proper VM. Since OvS does not know which VM is attached to a specific port (it just knows ports and the rules used to forward packets among them), for this operation the vSwitch has to rely on an external component. In our prototype we adopted two strategies: (*i*) a meaningful message is printed on the console with an example of the proper command that can be issued (manually) to attach the new `ivshmem` device to a VM, and (*ii*) we extended the compute agent to receive requests from OvS and plug the new `ivshmem` device into the proper VMs by interacting with QEMU. Particularly, in our prototype we exploit the compute agent of the Universal Node [25], an NFV node defined as part of the FP7 UNIFY project [26], although other solutions such as the OpenStack Nova [27] agent can be exploited as well.

The above procedure is executed both when a new p-2-p link is recognized, and in case a p-2-p link does not exist anymore and then the direct channel between two VMs must be destroyed.

C. Handling the new `ivshmem` device in the guest: the remapping process

The *remapping process* consists in recognizing, then changing, dynamically and transparently, the pair of `rte_rings` an application is using. Then, it allows existing DPDK-based applications using `dpdkr` ports to support our technology without any modification, except for the necessity to be recompiled with our modified DPDK.

[2] It is worth mentioning that, when a VM is created, its `dpdkr` ports are connected to the forwarding engine of OvS. In fact, at that time no `flowmod` involving those ports has still been received, then the vSwitch cannot know whether they will be used or not in a p-2-p link.

Vanilla DPDK does not recognize when a new `ivshmem` device is hotplugged in the VM. Hence, our prototype extends DPDK in order to register an handler (through the `lib_udev` library) that is executed each time a new `ivshmem` device is connected to the PCI bus. In this handler, DPDK identifies the old `rte_rings` to be removed (thanks to the virtual addresses specified in the metadata file), and marks them as "to be remapped". The remapping process is in fact not actually done in this handler, since its execution is asynchronous with respect to the application, which may be accessing the `rte_rings` to send/receive packets during the execution of the handler itself.

Algorithm 1 Remapping process in the VM.

```
 1: procedure sendPackets (rte_ring *ring, list *pkts) {receivePackets is equivalent}
 2:   if ring $\in$ toBeRemapped then
 3:     for all r $\in$ toBeRemapped do
 4:       munmap(r.virtualAddress)
 5:       mmap(r.virtualAddress,r.device)
 6:       r.mapped $\leftarrow$ true
 7:       toBeRemapped.remove(r)
 8:     end for
 9:     while not ring.usable do
10:       {do nothing}
11:     end while
12:   end if
13:   {send/receive packets as usual}
14: end procedure
```

The remapping of the `rte_rings` used by an application is then done by DPDK when such an application transmits or receives packets, as shown in Algorithm 1. To this purpose, we extended DPDK so that, before transmitting/receiving packets on an `rte_ring`, it checks whether such a ring has to be remapped or not (line `2`). If so, according to lines `3-8`, it remaps all the `rte_rings` contained into the new `ivshmem` device. This requires to unmap the old `rte_rings` and to `mmap` the corresponding new ones at the same virtual addresses just released (lines `4-5`).

In order to avoid packet loss and reordering in this transient, we defined a synchronization mechanism between DPDK in the guest and OvS, based on two flags inserted in the `rte_ring` structure: `mapped` and `usable`. As shown in line `6`, DPDK sets the former as soon as an `rte_ring` has been remapped. At this point OvS, which was blocked on such a shared flag: *(i)* copies the (pointers to) packets present in the old `RX rte_ring` into the new one; *(ii)* handles the packets already inserted by the application in the old `TX rte_ring`; *(iii)* notifies DPDK in the guest that the new `rte_ring` can be used, by means of the `usable` flag. At this point (line `9-11`) the application can finally transmit/receive the packets.

As a final remark, the remapping process is exactly the same both in case a p-2-p channel is going to established or destroyed.

D. Ports and flows statistics

As already mentioned, in order to keep the *direct VM2VM* transparent to external entities, OvS allows them to issue the same commands (e.g., get statistics, turn the port on/off) on all `dpdkr` ports, regardless of the actual port implementation.

Particularly, the possibility to expose statistics related to ports and flows implementing a p-2-p link requires to further extend the DPDK framework. The vSwitch is in fact not able to count statistics related to p-2-p links by itself, as it is not involved in moving packets flowing through these connections.

Then, each time a packet is sent through a direct channel, DPDK increases the counters associated with that OpenFlow rule and port, which are stored into the `rte_rings` used to transmit the packet. Then, when OvS needs to export statistics related to p-2-p links, it just reads the proper values from the specific `rte_ring`.

V. Experimental results

We characterized our *direct VM2VM* prototype on an Intel Xeon E5-2690 v2 @ 3 GHz (ten physical cores plus hyperthreading), 64 GB RAM, Ubuntu 15.04, equipped with two 10G Intel 82599ES NICs. Our code is based on OvS 2.4.9 and DPDK 2.1.0, and it is available at [28].

In the tests, we compare our solution with traditional connections implemented through the forwarding engine of OvS, both from the point of view of the maximum throughput achieved and the latency introduced by service chains implemented with the two approaches[3], and from the point of view of the number of CPU cores required to achieve the maximum performance. Finally, we report the time required by our prototype to detect a p-2-p link and create the direct path between the two VMs involved.

During the tests, each VM had two `dpdkr` ports and ran a single core DPDK application that simply moved packets from one port to another. Notably, thanks to the transparency of the *direct VM2VM* technology, the same forwarding VM has been used unchanged in all tests.

A. Throughput

Figure 5 reports the throughput obtained with chains of growing length, when traversed by bidirectional traffic consisting of 64B packets sent at the maximum speed.

Particularly, Figure 5(a) refers to the scenario shown in Figure 6(a), in which the first and the last VM of the chain act as traffic source/sink; this test validates our approach without the NICs and PCI-e bus bottlenecks. Figure 5(b) refers instead to the scenario depicted in Figure 6(b), in which traffic is delivered/drained to/from the chain through the 10Gbps NICs; in this case the maximum theoretical throughput is 20Gbps.

[3] With the term *chain*, we indicate a graph in which VMs are connected only through p-2-p links.

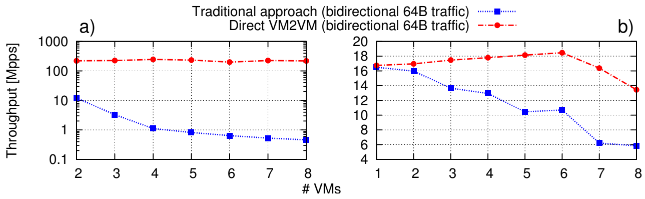


Figure 5. Throughput: (a) internal traffic; (b) traffic source/sink connected through physical NICs.

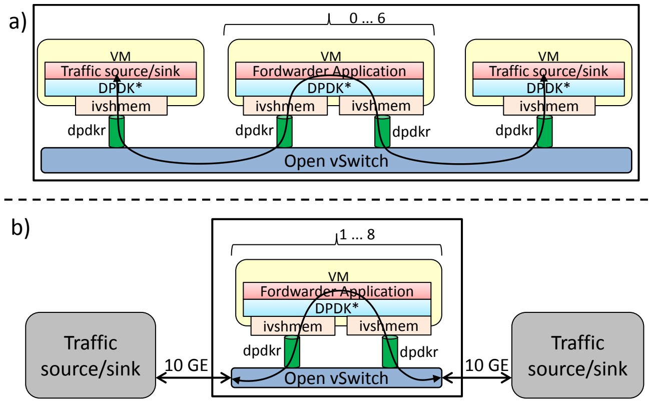


Figure 6. Test setup: (a) internal traffic; (b) traffic source/sink connected through physical NICs.

Both tests show that a chain of VMs exploiting our *direct VM2VM* approach provides better throughput than the same chain implemented using the traditional approach. Moreover, the throughput is almost constant when the *direct VM2VM* is used, while it presents mostly a decreasing trend when all the connections are implemented through the forwarding engine of OvS.

B. Latency

Figure 7 shows the latency measured in the test scenario depicted in Figure 6(b), tuning the TX speed in order to avoid packet loss in the chain. Particularly, the picture reports the median value of the samples; the margin of error is of $\pm 0.4\mu$s in the worst case at the 95% confidence level and hence not visible in the graph.

According to the graph, latency introduced by both the approaches is almost the same until 5 chained VMs, then the values start to diverge, resulting in an improvement of about 80% in case 8 chained VMs with the *direct VM2VM* architecture.

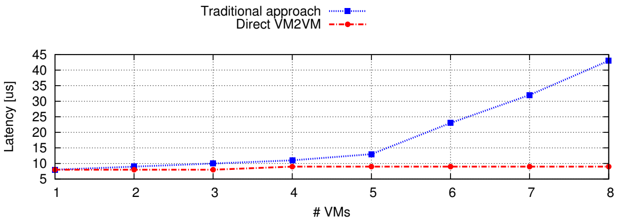


Figure 7. Latency when physical NICs are involved.

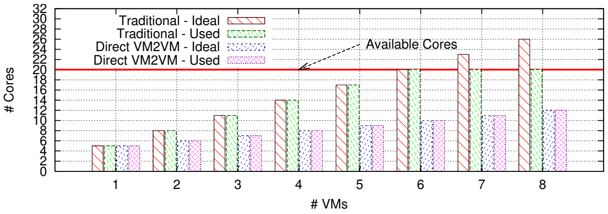


Figure 8. Cores ideally required vs cores used during the tests.

C. Number of CPU cores

Figure 8 compares the *direct VM2VM* proposal with the traditional approach in terms of CPU cores, and reports both the number of cores ideally needed to maximize performance, and the number of cores actually used to get the throughput of Figure 5(b). According to the graph, the *direct VM2VM* prototype always requires the same, or a smaller, number of CPU cores than the traditional approach, resulting in the possibility to consolidate more VMs on the same physical server. Moreover, while we were always able to use the ideal number of cores with our prototype, this was not possible with vanilla OvS (in case of 7 and 8 chained VMs).

To understand the reported numbers, it is worth mentioning that OvS ideally requires one polling core per each receiving queue of (physical and `dpdkr`) ports connected to it. Particularly, the same polling core takes care of receiving all the packets from a specific receiving queue, processing and finally transmitting them through the proper output port. Then, in case the number of cores assigned to the vSwitch is lower than the number of receiving queues, the same core takes care of handling packets entering from multiple queues, potentially resulting in performance degradation.

If we consider the test scenario depicted in Figure 6(b), where all the ports have been configured with a single receiving queue, the traditional approach requires one core per physical port, and three cores per each running VM (one per `dpdkr` port plus one core running the VM itself); in other words, it requires a number of cores that is $(2+3*\#VMs)$. Instead, with the *direct VM2VM* approach, `dpdkr` ports between VMs are no longer connected to the forwarding engine of OvS, and then they do not require any dedicated core in the vSwitch. Then, this solution just needs two cores for the physical ports, one core for a `dpdkr` port of the first VM and one core for a `dpdkr` port of the last VM of the chain, plus one core per VM. This results in a number of cores that is $(2+2+\#VMs)$.

D. Direct channel setup time

Figure 9 reports the time needed to establish a direct channel between two VMs, from the moment in which OvS

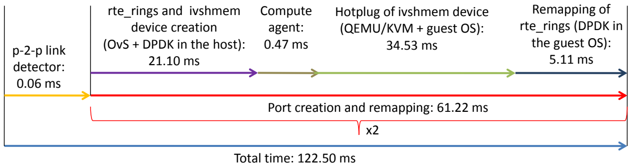


Figure 9. Time required to establish a direct connection.

receives a new rule (flowmod) that triggers the creation of a p-2-p link, to the moment in which the forwarding application starts to use the new direct dpdkr port. This time depends on different components such as DPDK, OvS, compute agent, guest OS and the QEMU/KVM hypervisor. Results show that the (by far) dominant contributors are OS-level components, namely the time needed by QEMU/KVM to plug the ivshmem device and the guest OS to recognize it. Instead, the weight of the p-2-p link detector module is negligible, questioning the necessity of a more optimized algorithm.

VI. Conclusion

This paper proposes an architecture that is able to optimize inter-VNF communications by bypassing the vSwitch in case of p-2-p connections between VMs. Our architecture can accelerate the packets exchange between the VMs *transparently*, i.e., without modifying the applications and by keeping the compatibility with all the services (e.g., OpenFlow controller) deployed in an NFV environment, and *dynamically*, i.e., it can optimize direct paths when those are detected and revert back to the traditional VM-to-switch-to-VM communication when the optimization is no longer possible. Our extensions have been integrated in a *widespread* vSwitch, bringing the advantages of this technology to a broad set of use cases.

Our tests confirm the goodness of the approach and the possibility to implement this idea by touching a limited number of components, namely OvS, DPDK and (optionally) the compute agent. Future work will explore the possibility to extend the usage of direct paths also when accessing to physical NICs, e.g., through SR-IOV.

Acknowledgments

This work was conducted within the framework of the FP7 UNIFY project, which is partially funded by the Commission of the European Union. Study sponsors had no role in writing this report. The views expressed do not necessarily represent the views of the authors employers, the UNIFY project, or the Commission of the European Union.

References

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

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

[3] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[4] J. Nakajima, M. Ergin, Y. Jiang, K. Murthy, J. Tsai, W. Wang, H. Xie, and Y. Zhang. KVM as the NFV hypervisor, kvm forum 2015. http://events.linuxfoundation.org/sites/events/files/slides/Jun_Nakajima_NFV_KVM%202015_final.pdf.

[5] S. Garzarella, G. Lettieri, and L. Rizzo, "Virtual device passthrough for high speed vm networking," in *Architectures for Networking and Communications Systems (ANCS), 2015 ACM/IEEE Symposium on*. IEEE, 2015, pp. 99–110.

[6] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, "Extending networking into the virtualization layer," in *Proceedings of the 8th ACM Workshop on Hot Topics in Networks (HotNets-VIII)*, October 2009.

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

[8] J. Nakajima, J. Tsai, M. Ergin, Y. Zhang, and W. Wang. Extending KVM models towards high-performance NFV, kvm forum 2014. http://www.linux-kvm.org/images/1/1d/01x05-NFV.pdf. [Online]. Available: http://www.linux-kvm.org/images/1/1d/01x05-NFV.pdf

[9] R. Russell, "Virtio: Towards a de-facto standard for virtual i/o devices," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 95–103, Jul. 2008. [Online]. Available: http://doi.acm.org/10.1145/1400097.1400108

[10] L. Rizzo, "Netmap: a novel framework for fast packet i/o," in *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342821.2342830

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

[12] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, 2014, pp. 459–473.

[13] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, "The click modular router," in *Proceedings of the seventeenth ACM symposium on Operating systems principles*, ser. SOSP '99. New York, NY, USA: ACM, 1999, pp. 217–231.

[14] J. Hwang, K. K. Ramakrishnan, and T. Wood, "Netvm: High performance and flexible networking using virtualization on commodity platforms," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, 2014, pp. 445–458.

[15] I. Cerrato, G. Marchetto, F. Risso, R. Sisto, and M. Virgilio, "An efficient data exchange algorithm for chained network functions," in *High Performance Switching and Routing (HPSR), 2014 IEEE 15th International Conference on*. IEEE, 2014, pp. 98–105.

[16] X. Zhang, S. McIntosh, P. Rohatgi, and J. L. Griffin, "Xensocket: A high-throughput interdomain transport for virtual machines," in *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, ser. Middleware '07. New York, NY, USA: Springer-Verlag New York, Inc., 2007, pp. 184–203. [Online]. Available: http://dl.acm.org/citation.cfm?id=1516124.1516138

[17] K. Kim, C. Kim, S.-I. Jung, H.-S. Shin, and J.-S. Kim, "Inter-domain socket communications supporting high performance and full binary compatibility on xen," in *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '08. New York, NY, USA: ACM, 2008, pp. 11–20. [Online]. Available: http://doi.acm.org/10.1145/1346256.1346259

[18] J. Wang, K.-L. Wright, and K. Gopalan, "Xenloop: A transparent high performance inter-vm network loopback," in *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, ser. HPDC '08. New York, NY, USA: ACM, 2008, pp. 109–118. [Online]. Available: http://doi.acm.org/10.1145/1383422.1383437

[19] W. Huang, M. J. Koop, Q. Gao, and D. K. Panda, "Virtual machine aware communication libraries for high performance computing," in *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2007, November 10-16, 2007, Reno, Nevada, USA*, 2007, p. 9. [Online]. Available: http://doi.acm.org/10.1145/1362622.1362635

[20] F. Diakhaté, M. Pérache, R. Namyst, and H. Jourdren, "Efficient shared memory message passing for inter-vm communications," in *Euro-Par 2008 Workshops - Parallel Processing, VHPC 2008, UNICORE 2008, HPPC 2008, SGS 2008, PROPER 2008, ROIA 2008, and DPA 2008, Las Palmas de Gran Canaria, Spain, August 25-26, 2008, Revised Selected Papers*, 2008, pp. 53–62. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00955-6_7

[21] C. Macdonell. Nahanni, a shared memory interface for kvm. [Online]. Available: http://www.linux-kvm.org/images/e/e8/0.11.Nahanni-CamMacdonell.pdf

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

[23] "mmap," http://man7.org/linux/man-pages/man2/mmap.2.html.

[24] "Ivshmem library," http://dpdk.org/doc/guides/prog_guide/ivshmem_lib.html.

[25] I. Cerrato, A. Palesandro, F. Risso, M. Suñé, V. Vercellone, and H. Woesner, "Toward dynamic virtualized network services in telecom operator networks," *Computer Networks*, vol. 92, pp. 380–395, 2015.

[26] "Unify: unifying cloud and carrier network," 2013. [Online]. Available: http://www.fp7-unify.eu/

[27] "openstack nova," http://docs.openstack.org/developer/nova/index.html.

[28] Direct VM2VM optimzation in OvS for DPDK enabled applications. https://github.com/netgroup-polito/directvm2vm.