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

Roberto Bonafiglia, Ivano Cerrato, Francesco Ciaccia, Mario Nemirovsky, Fulvio Risso

Politecnico di Torino, Italy
Agenda

• Goals
• Motivation
• Technologies overview
  – Execution environments (KVM, Docker)
  – Virtual switches (OvS, OVDPDK)
• Experimental results
• Conclusions
Goals

• Analyze the performance of different virtualization technologies \textit{when used to implement NFV}
  
  – Virtual switches:
    • Open vSwitch (OvS)
    • DPDK-based OVS (OVDPDK)
  
  – Execution environment:
    • Kernel-based Virtual Machine (KVM)
    • Docker container

• Provide an initial evaluation of the overhead they introduce in an NFV environment
  
  – Throughput
  
  – Latency
Why new evaluations?

- NFV is heavily based on cloud computing technologies
  - VNFs executed in virtual machines or lightweight containers
  - Paths implemented through vSwitches
- However, many differences exist between cloud computing and NFV:
  - Computed-bound tasks (cloud) vs network I/O intensive (NFV)
  - Packets have to traverse several VNFs: the vSwitch may have to handle the same packet multiple times
  - Existing hardware accelerators not appropriate
    - Generic Receive Offload (GRO) / TCP segmentation offload (TSO)
    - Some VNFs (e.g., NAT) need to work on each single Ethernet frame, not on TCP/UDP segments
- Then, NFV services may behave differently in a cloud environment
Networking with KVM

- KVM: kernel module that transforms Linux in an hypervisor
- Guest operating system executed within QEMU
  - Process running in user-space in the host
    - VM memory: part of the QEMU memory
    - Each vCPU corresponds to a different QEMU thread in the host
Networking with the KVM – cont’d

• The guest operating system accesses to the virtual NICs through the *virtio-net* driver
  – Driver optimized for the virtualization context

• Each vNic is associated to a *vhost* kernel-process in the host
  – Vhost is connected to the vNIC on one side, to a TAP interface on the other side
    • TAP interface in turn connected with the vSwitch
  – Vhost waits for interrupts, and moves pkts between the vNIC and the TAP
Networking with Docker containers

- Lightweight virtualization mechanism
  - Does not run a complete operating system
  - All containers share the same kernel
  - Just a set of namespaces to limit the resources visible by a user-space process
    - If no user-space process is running in a container, it is not associated with any thread in the host
Networking with Docker containers – cont’d

- Each container corresponds to a different network namespace
  - It is only aware of the vNICs in such a namespace
  - It has its own network stack

- Packets can traverse the namespace boundary through veth interfaces
Open vSwitch I/O model

- OvS consists of a kernel module and a user-space daemon
  - Kernel module: it is a cache of the last matched rules
  - User-space daemon:
    - Contains the full forwarding table of the switch
    - Can be programmed through OpenFlows
- Both the kernel and the user-space daemon work in interrupt mode
Open vSwitch I/O model – cont’d

- No thread associated with the kernel module
  - It is a callback invoked by the software interrupt raised when a packet is available on a network interface (NIC, tap, etc)
    
    - OvS executed in the context of the *ksoftirq* kernel thread in case of packets from a physical NIC
    - OvS executed in the context of the sender process, in case of virtual NIC
      - The vhost kernel thread in case of VMs
      - The user-space process in case of Docker
DPDK-based OvS I/O model

- It consists of a number of user-space threads
- Continuously iterates on the physical and virtual NICs
  - Polling mode (vs. interrupt mode in the vanilla OvS)
- Thanks to DPDK:
  - OvS accesses to the physical NICs by bypassing the operating system
  - Zero-copy data transfer between physical NICs and the vSwitch
• The processing of the packet is split in multiple threads
  – Ksoftirq
  – Vhost
  – QEMU process
Packet flows – the “container” case

- The VNF is linux bridge
  - The data-plane is a kernel-level callback executed in the context of the thread providing the packet to the switch

- A single thread executes all the operations associated with a packet
  - No parallelization
  - Poor performance
• Running in a container a VNF that is actually a process, parallelization can be achieved in Docker as well
Testbed

• Physical machines: Intel Core i7-4770 @3.40GHz (4+4 cores), 32 GB of memory, fedora 20
• Physical network interfaces: 10 Gigabit Ethernet Intel X540-AT2
• Variable number of VNFs chained through a vSwitch
  – KVM VNF: linux bridge
    • Each VM has a single virtual CPU (i.e., it corresponds to 1 thread in the host)
  – Docker VNF: simple libpcap-based bridge
• Performance slightly better with VMs, tends to be equivalent with more VNFs in the chain

• Docker: linux bridge vs libpcap-based switch
  • Bare metal throughput: linux bridge outperforms libpcap-based switch
  • Several chained dockers: linux bridge is much worse (3Mbps with 8 chaines containers)

• Throughput inversely proportional to the length of the chain
  – More VNFs means more contention on the physical resources (CPU, cache, TLB)
Multiple chain

- The graph shows the results for 64B packets
- Using multiple chains reduce the overall throughput
- Having multiple VMs in parallel (this graph) or in sequence (previous slide) produces comparable results
• Results obtained by installing DPDK inside the VM
  – The L2-fw application provided with DPDK has been used
  – Running traditional Linux bridge w/o DPDK did not provide enhancements wrt standard OvS

• More cores to the switch results in better performance if 
  (#cores_switch + #VMs) <= numbers of cores of the server
Latency

- As per the throughput, also latency worsens sensibly when exceeding the #CPU cores (with OVDPPDK)

- The behavior for VM and Docker results linear with the number of elements in the chain
**Conclusions**

- Docker provides acceptable results only in case of VNFs associated with a specific process
  - Containers are unsuitable for VNFs implemented as kernel callbacks
- Due to the low overhead introduced by containers, a user-space program in Docker provides better latency and almost the same throughput wrt kernel callback-based VNFs within VMs
- OvDPDK provides better performance than OvS, when used with DPDK-based VNFs within VMs
  - Polling working model
    - This limits its usage in servers running a reduced number of VNFs
  - Zero copy between the physical NICs and the switch
Questions?