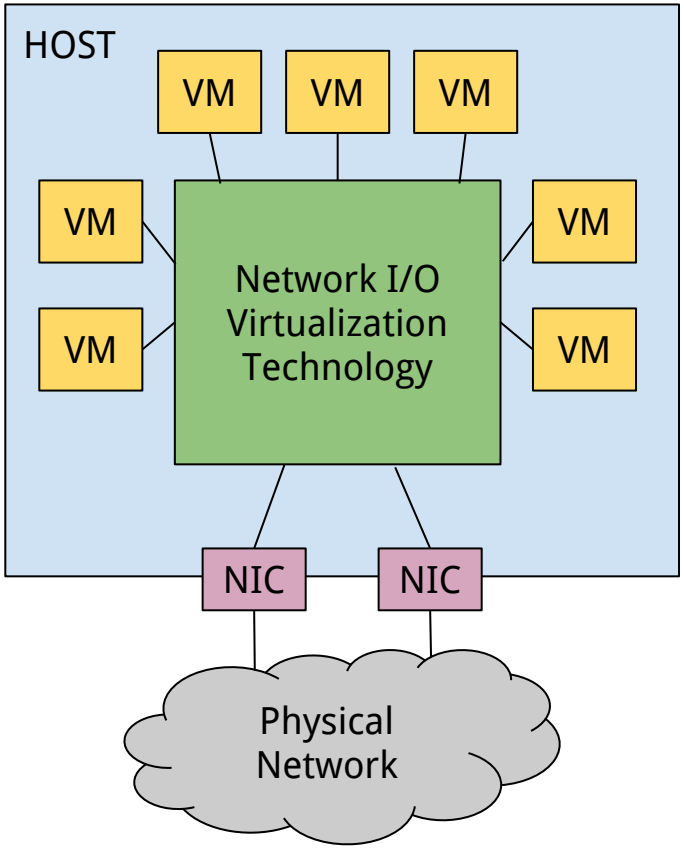
# High-performance TCP/IP networking for bhyve VMs using netmap passthrough

Belgrade, 23rd September 2016, *Vincenzo Maffione*
Università di Pisa


# Introduction

- Network I/O virtualization is about attaching the Virtual Machines to the network of the hypervisor/host.
- Let different VMs on the same host communicate among them and with the external physical network.
- Hot use-cases → Network Function Virtualization



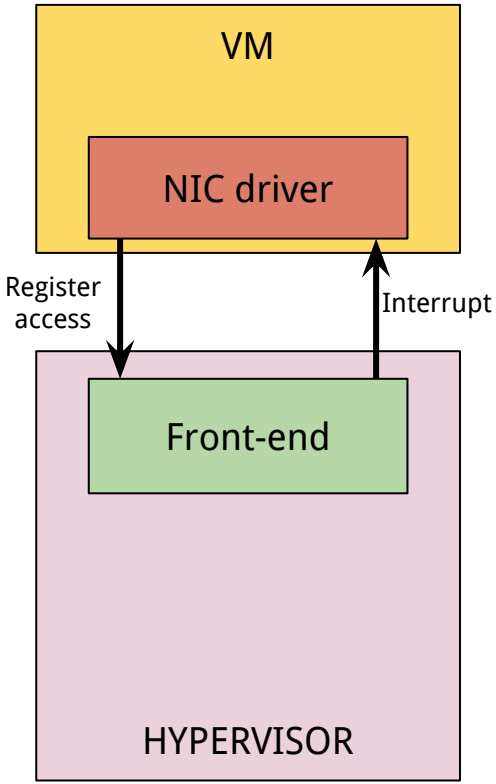
# Outline

1. Traditional VM networking
2. VM networking with Netmap passthrough
3. FreeBSD ptnet implementation
4. Performance evaluation and comparisons
5. Demo

# Traditional VM networking

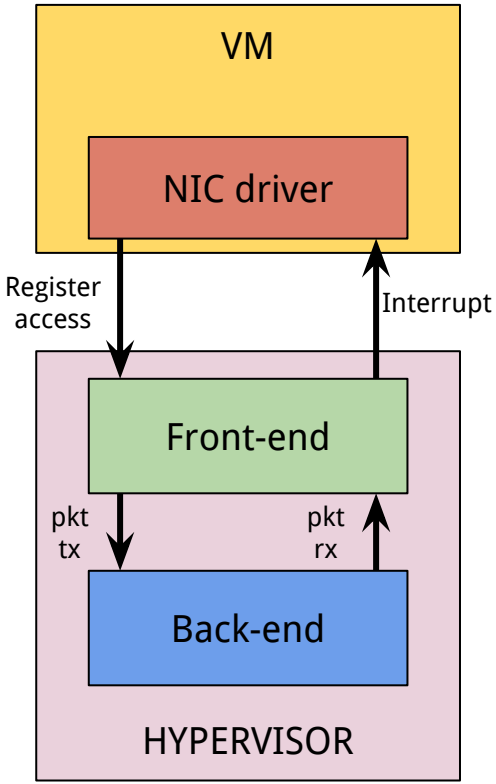

# Traditional VM networking - Front end

- A guest NIC comes with its own device model
  - An emulated commercial NIC (e.g. e1000, r8169)
  - A paravirtualized NIC (e.g. virtio-net, Xen netfront)
- Device model implemented by a front-end module in the hypervisor
- Different model, different ring & descriptor format
  - Virtio-net: VirtQueues, Avail and Used rings, …
  - Xen netfront: I/O rings
  - Commercial NICs: hardware specs Intel, Realtek, … use formats specified in their documentation



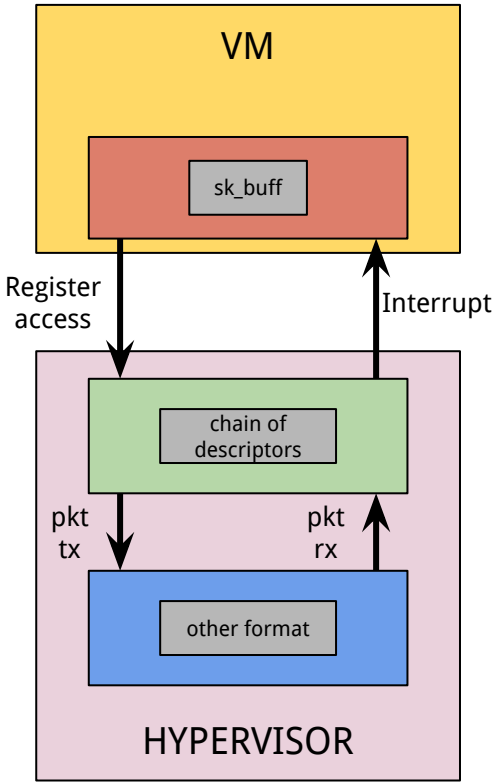
# Traditional VM networking - Backend

- Front-end and back-end interact to transmit and receive packets to/from the host network
- Different back-ends usually available:
  - TAP: inject/receive packets from host TCP/IP stack
  - Socket: packets forwarded through a TCP or UDP socket
  - NAT: backend implements NAT (in user-space) to give a VM internet access
  - Netmap/DPDK: Packets injected/received from an high performance userspace networking framework
- Different back-end different packet representation



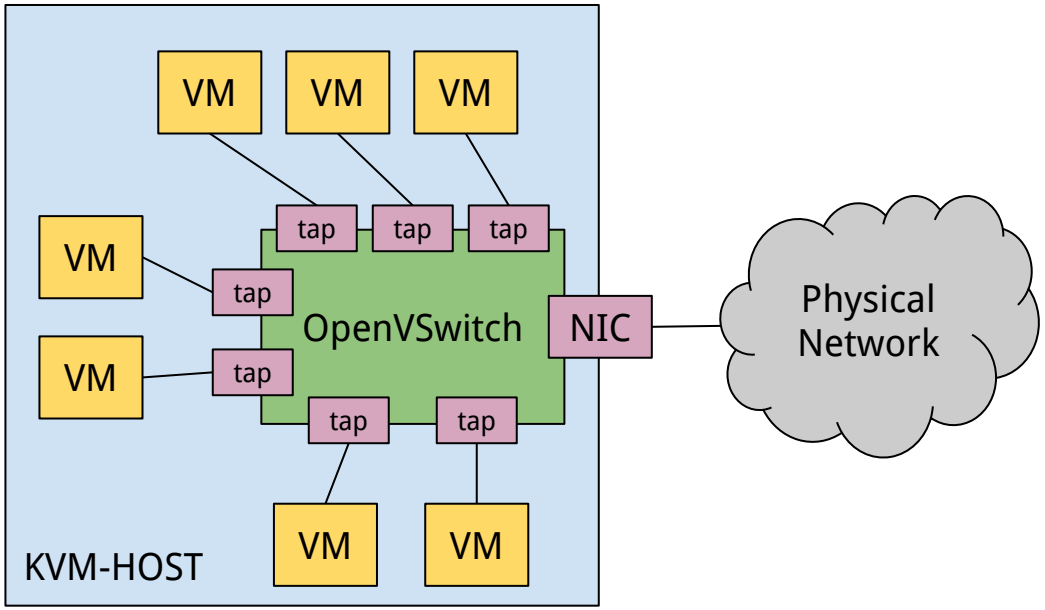
# Traditional VM networking - Packet formats

- The journey of a tx/rx packet is convoluted
- Guest driver uses mbuf/sk_buff
- Front-end uses a list of descriptors in a ring/queue
- Backend may use multiple formats:
  - TAP and sockets uses sk_buffs/mbufs
  - Netmap uses its API (netmap rings and slots)
  - ...
- Packet representation conversions is needed at each step
  - Conversions requires processing
  - Copies may be needed
  - Need driver/front-end synchronization and front-end/back-end synchronization



# A traditional deployment

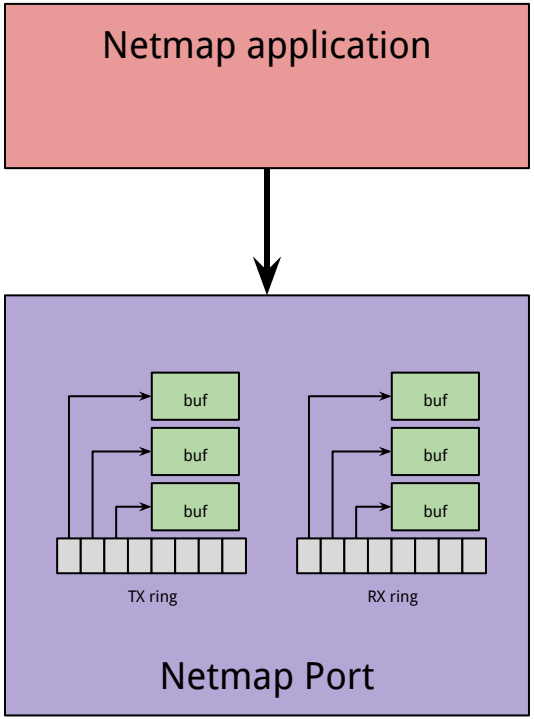
- QEMU-KVM or bhyve hypervisor
- virtio-net front-end
  - With vhost acceleration on QEMU
- TAP backend attached to either
  - OpenVswitch switch instance, or
  - Standard in-kernel L2 bridge
- Many bottlenecks:
  - read/write to TAP interfaces
  - Virtual switch processing
  - driver/front-end conversions and synchronization



# VM networking with Netmap passthrough

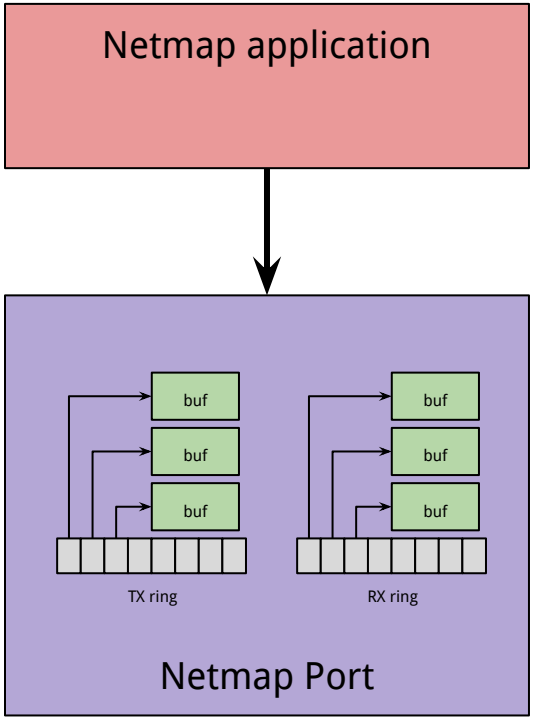

# Towards an alternative approach - Netmap (1)

- Netmap is an API to directly access NIC TX/RX rings
  - It supports TX/RX batching, useful to remove I/O bottlenecks
- A *netmap port* is accessed through the netmap API
  - Hardware-independent rings and buffers are mapped into the userspace application address space
- Various port types, depending on the backing I/O
  - Physical ports (NICs)
  - VALE (virtual L2 switch) ports
  - Pipes
  - Monitors
  - ...



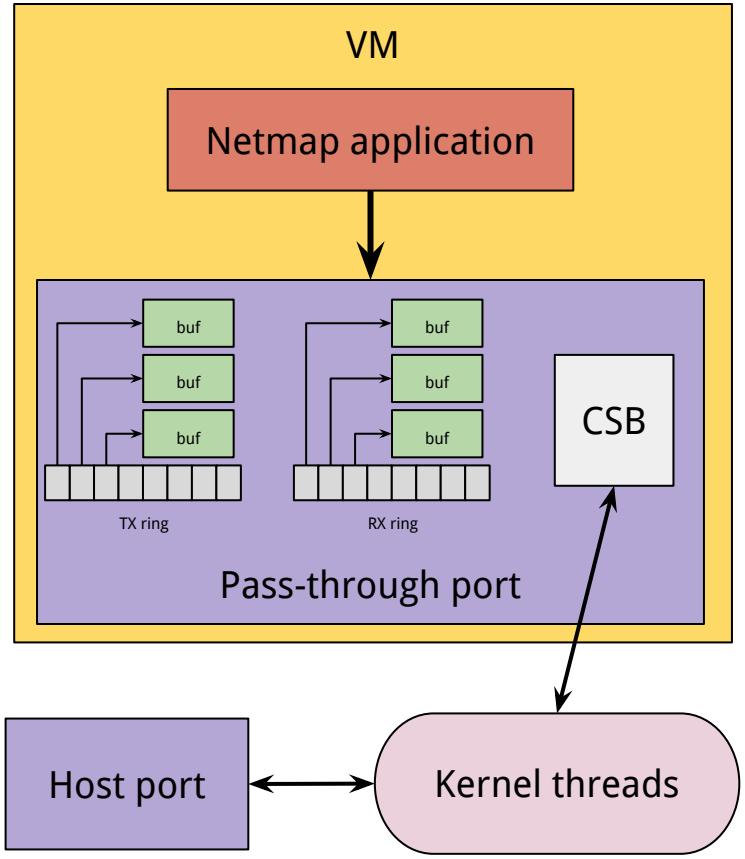
# Towards an alternative approach - Netmap (2)

- Differences with DPDK-based solutions:
  - Support for asynchronous notifications
    - Low CPU usage under low load
    - Notifications dynamically suppressed under high load
  - VALE provides isolation (packet copy) between untrusted VMs, while allowing for high performance
    - up to 20 Mpps between two ports



# Towards an alternative approach - Passthrough

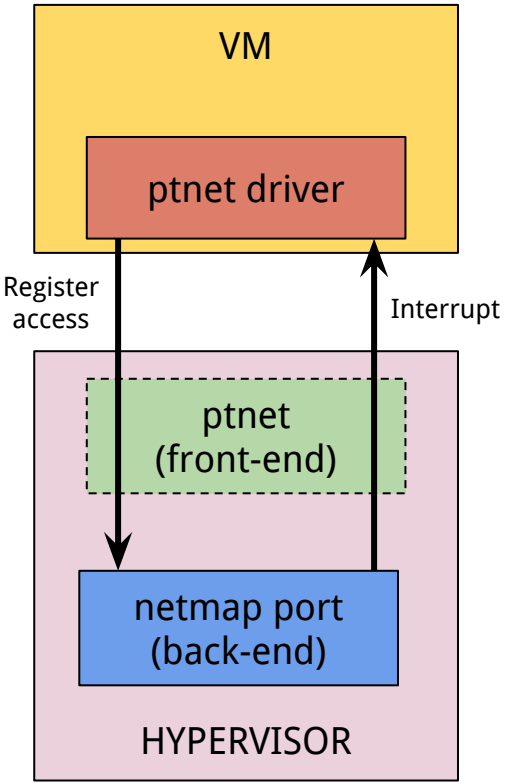
- What if we map netmap rings and buffers of an host port inside a VM?
- The VM could directly access the host port using the netmap API!
- A special pass-through port is used to do the trick
  - TX/RX sync operations (e.g. real synchronization with the NIC hardware) are performed by kernel threads in the host
  - Communication Status Block (CSB) used to synchronize guest ring indices with host ring indices




*[ANCS 2015] - Virtual device passthrough for high speed VM networking*

# The ptnet device - Model

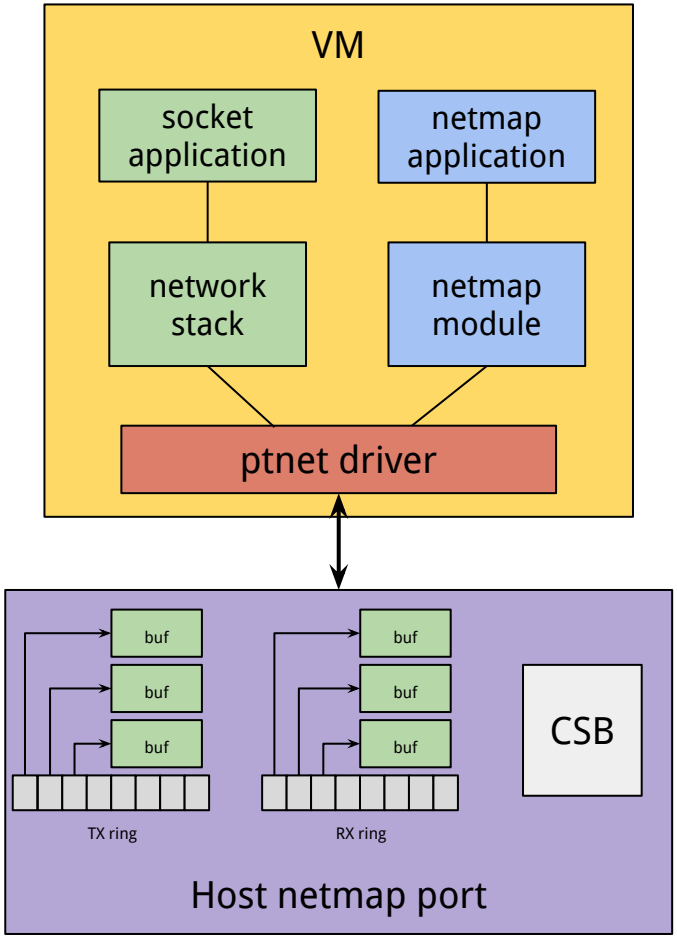
- ptnet is a paravirtualized NIC which uses the netmap API as the underlying device model
- No format conversions necessary between front-end and back-end
- Guest can access the back-end port directly, using netmap passthrough
- The front-end is used only for
  - Configuration: number of available queues
  - Control: start/stop kernel threads
  - Synchronization: kick/interrupt
- … but it's not part of the datapath




*[LANMAN 2016] Flexible Virtual Networking using netmap passthrough*

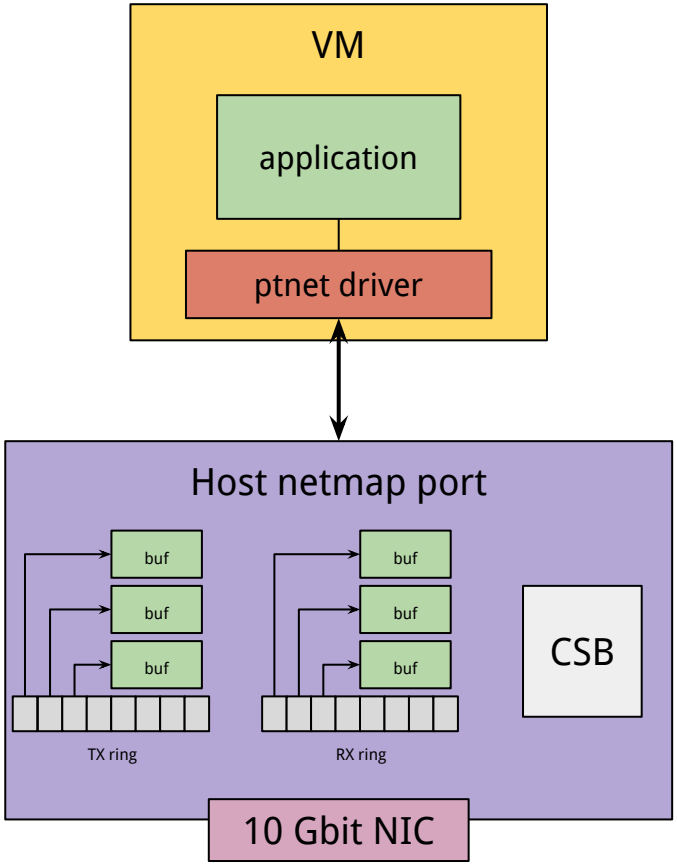
# The ptnet device - Applications

- Support for applications running in netmap mode
  - They directly access the back-end netmap port
- Support for traditional socket applications
  - Conversion between sk_buff/mbuf and netmap slots is performed
  - The driver behaves as an application running in netmap mode
  - TSO and checksum offloadings



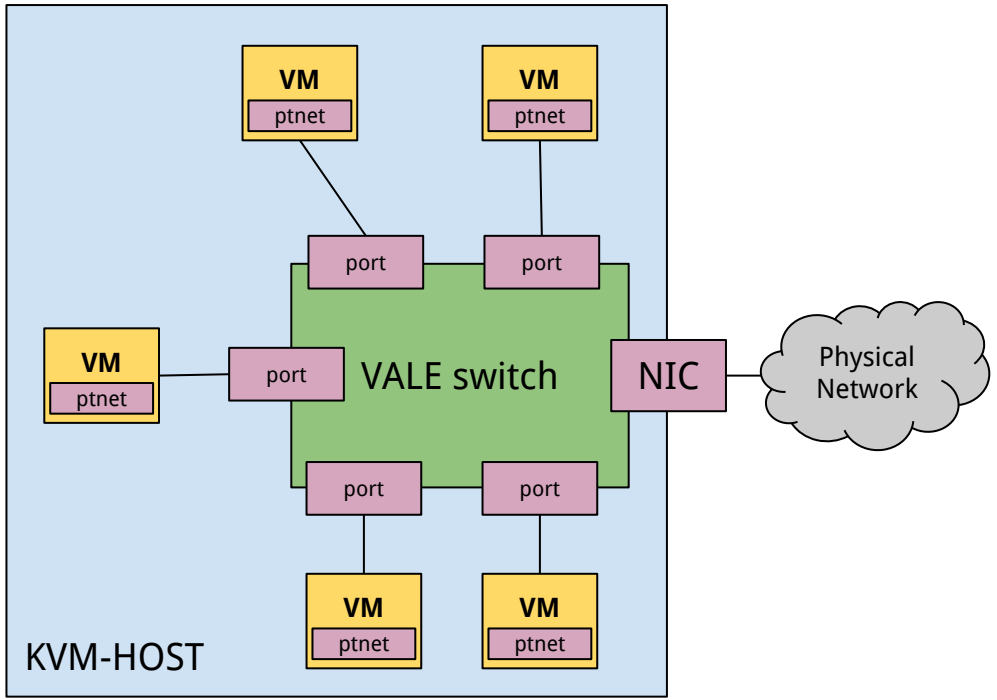
# Ptnet for NIC passthrough

- The VM sees all the queues of the physical NIC
- Netmap performance preserved in the guest
  - 14.88 Mpps TX/RX with a single core
- No PCI passthrough support needed in the hypervisor
  - Netmap code is reused



# Ptnet with VALE: deployment

- VALE as software switch between VMs and the NIC
- VALE ports are passed through to the VMs
- Up to 20 Mpps between different VMs
  - When using the netmap API



# FreeBSD *ptnet* implementation


# FreeBSD implementation - Starting point

- Work presented here → GSoC 2016 project supported by FreeBSD
- Netmap passthrough for FreeBSD was already available at project start, but with some limitations:
  - Only support for netmap applications → No support for socket applications
  - Hacked virtio-net driver to expose the passed-through netmap → The virtio-net interface becomes unusable
  - No support for multi-queue

# FreeBSD implementation - What's new

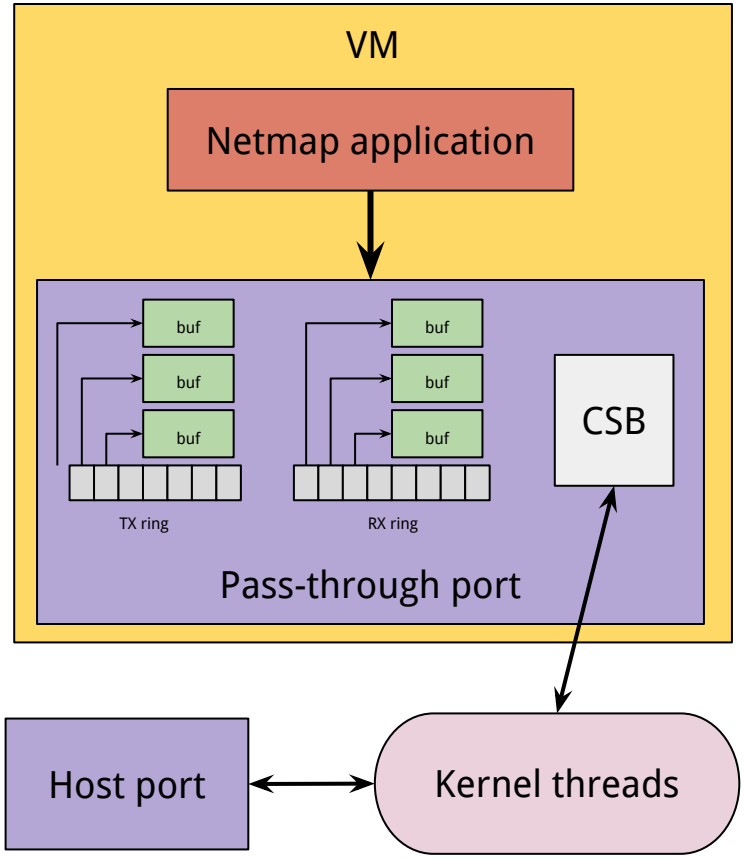
- My contributions, to remove these limitations:
  - Ad hoc FreeBSD multi-queue NIC driver for netmap passthrough → ptnet
  - Bhyve PCI device model for ptnet
  - Reorganization of bhyve networking code to allow multiple net backends
- Some links:
  - Code available at https://github.com/luigirizzo/netmap
  - More info at https://wiki.freebsd.org/SummerOfCode2016/PtnetDriverAndDeviceModel

# FreeBSD guest driver - Device attach

- Routines to probe/attach/detach ptnet PCI device:
  - Setup BARs of the ptnet PCI device
  - Read configuration of the passed-through host port from I/O registers
    - Number of TX/RX queues and per-queue slots
  - Allocate and setup Communication Status Block (CSB) memory for guest-host fast synchronization.
  - Setup MSI-X interrupts, one per queue
  - Setup ifnet (if_t) struct and `ether_ifattach()`
    - For socket applications
  - Setup netmap adapter and `netmap_attach()`
    - For netmap applications

# FreeBSD guest driver - Netmap adapter

- Main callbacks to expose to netmap:
  - Netmap register:
    - Switch the NIC from/to netmap mode
      - NIC stolen to network stack
    - Start/stop kthreads in the host
  - Netmap TXSYNC and RXSYNC:
    - Code completely shared with Linux ptnet driver
    - CSB sync: publish new guest ring indices and read current host indices
    - Kick the host (write to a per-queue I/O register) if necessary
      - The kick wakes up a kthread
      - Kthreads poll the CSB for more work



# FreeBSD guest driver - Network stack

- Basic idea: kernel-space network stack uses the NIC like an user-space netmap application
- Data copied back and forth between `mbufs` and netmap buffers.
- Per-queue `taskqueues` used for deferred receive or transmit work
  - Work deferred on interrupt or when running out of budget
  - Budget used to avoid be greedy on CPU
- Network stack callbacks, e.g.
  - ptnet_ioctl, to manage interface flags:
    - IFF_UP/IFF_DOWN: Switch netmap mode on (off) and (un)set the RUNNING flag
    - IFCAP_POLLING: Switch back and forth from polling mode
      - Need to sync with `taskqueues` when switching on
  - ptnet_transmit:
    - Push mbuf into a per-queue ringbuffer (drbr) and call the ringbuffer drain routine
  - ptnet_poll: polling routine

# FreeBSD guest driver - Offloadings support

- virtio-net header - prepended to each Ethernet frame - is the key to boost TCP performance:
  - TCP Segmentation Offloading
  - TCP/UDP checksum offloading
- Mechanism, in short:
  - VMs on the same host can exchange 32K/64K TCP packets without never performing TCP segmentation or computing TCP checksum
  - If a TSO packet needs to leave the host system, segmentation and checksumming can be offloaded to real NIC hardware
- Header supported by the VALE switch ports
- ptnet supports virtio-net header in the same way as virtio-net does:
  - virtio-net-header processing code copied from if_vtnet.c
  - TODO: this code should be shared between the two drivers
  - A sysctl available to disable the header (e.g. for middleboxes)

# FreeBSD host - vmm.ko extensions

- Some modification to vmm.ko and libvmm were necessary to
  - Make it possible for netmap (kthreads) to intercept guest register write from within the kernel
  - Make it possible for bhyve to map host netmap memory inside bhyve guests
- Adopted code from previous GSoC project
  - https://wiki.freebsd.org/SummerOfCode2015/ptnetmapOnBhyve

# FreeBSD host - ptnet device model for bhyve

- ~400 lines of code
- Device model callbacks:
  - `ptnet_init()`
    - Setup PCI device, including MSI-X and I/O registers
    - Create a netmap port backend in passthrough mode
  - `ptnet_bar_read()` and `ptnet_bar_write()`, to manage I/O register access
    - CSBBAH, CSBBAL: configure CSB physical address
    - VNET_HDR_LEN: specify virtio-net-header len (0 to disable)
    - Read only registers: number of tx/rx rings/slots, …
    - PTCTL, to configure kthreads:
      - per-queue MSI-X interrupt info
      - per-queue kick register info (to allow them to intercept I/O writes)
      - Start (stop) kthreads

# FreeBSD host - bhyve networking reorganization

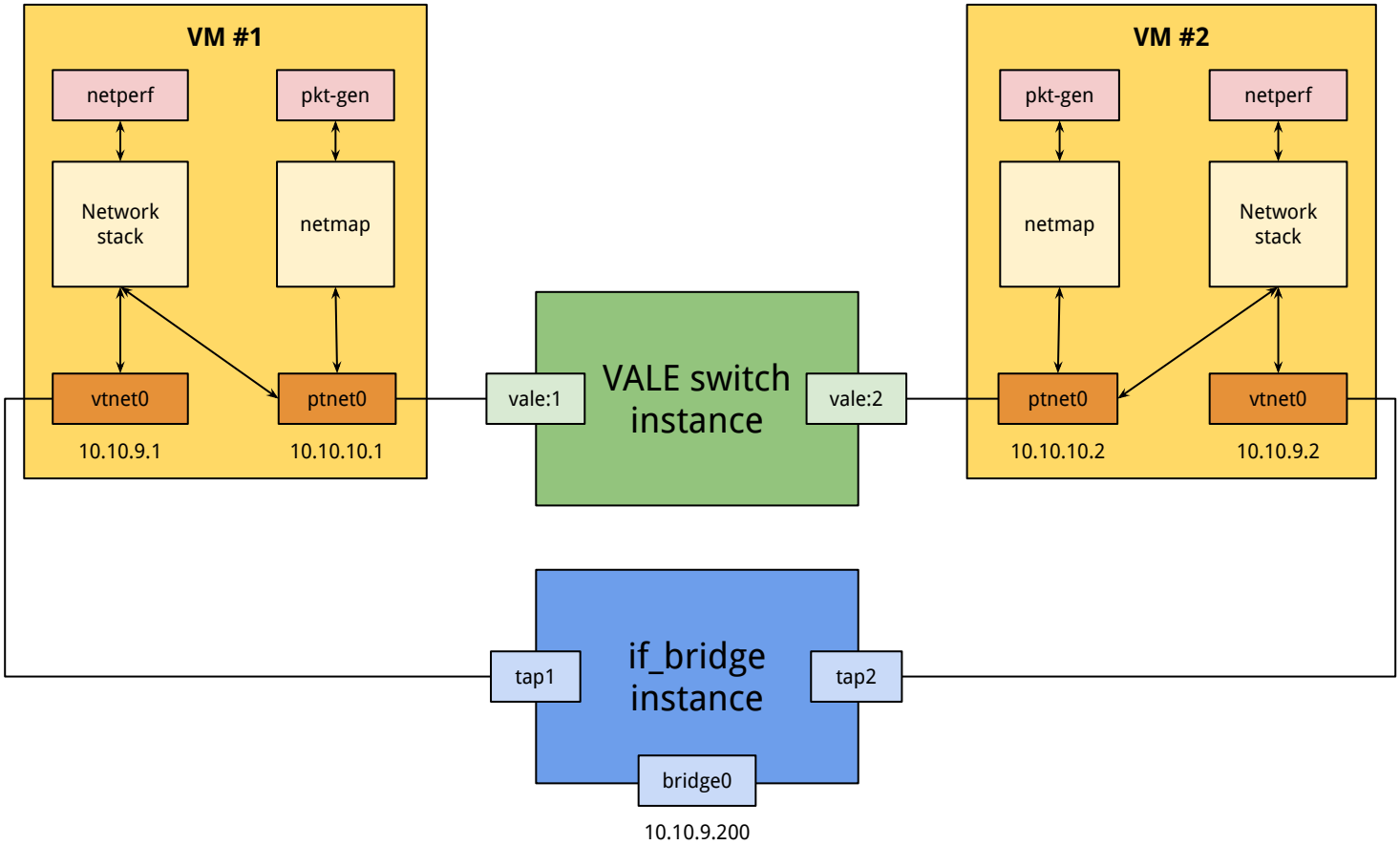
- Adopted previous code of mine to add support for multiple backend and frontends
- Supported combinations:
  - virtio-net + TAP
  - virtio-net + Netmap
  - ptnet + Netmap
- Example:
  - ```
    # bhyve -c 2 -m 1G -A -H -P \
            -s 31,lpc -l com1,stdio \
            -s 0:0,hostbridge \
            -s 1:0,virtio-net,tap1 \          # virtio-net NIC + TAP backend
            -s 2:0,virtio-net,vale0:2 \       # virtio-net NIC + netmap userspace backend
            -s 3:0,ahci-hd,freebsdimg.raw \
            -s 4:0,ptnet,vale1:1 \            # ptnet NIC + ptnetmap kernelspace backend
            -s 5:0,ptnetmap-memdev \          # ptnetmap memory device, needed by the ptnet NIC
            vm1
    ```
    ```

# Performance evaluation and comparisons

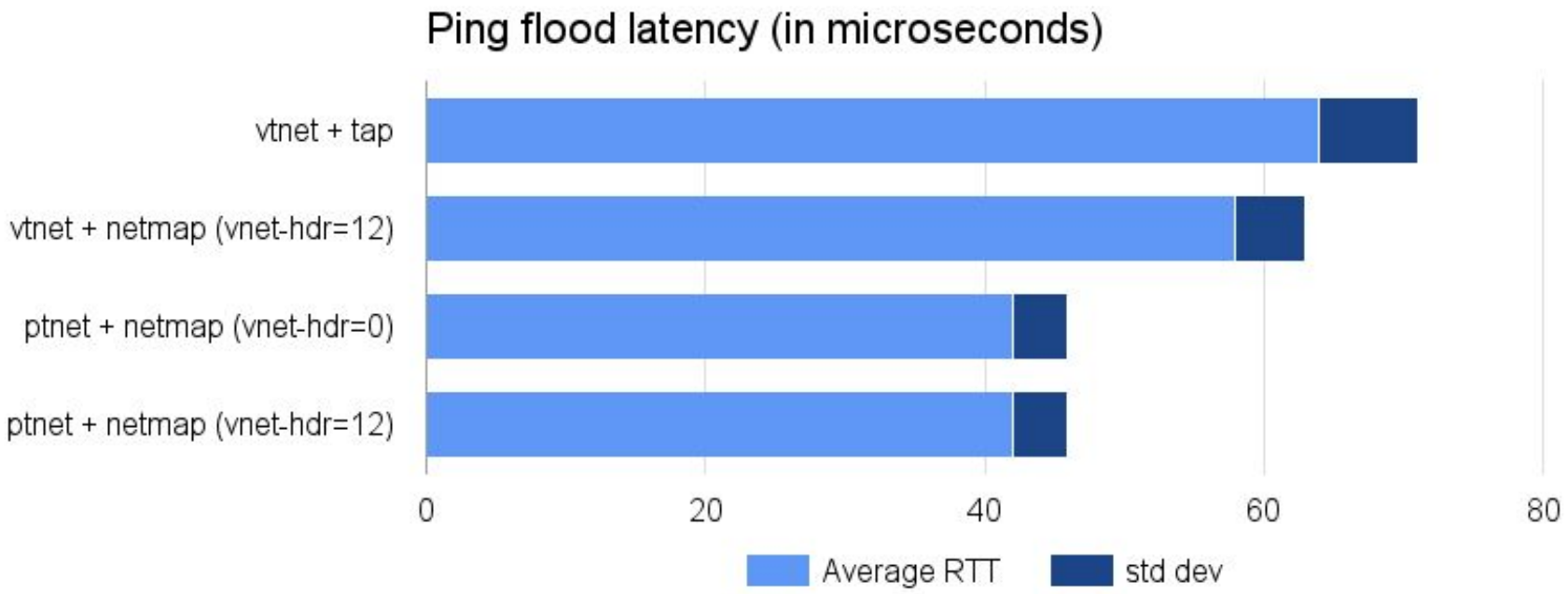

# Test setup

- CPU: Intel CPU 3770K, 4 physical cores, 2 hyperthreads per-core
- Memory: 8 GB DD3 @ 1.33 MHz
- Two bhyve VMs connected in two possible configurations:
    - traditional in kernel bridge, with TAP devices
    - a VALE switch
- Ping utility and Netperf tool used to measure throughput and latency

# Test setup

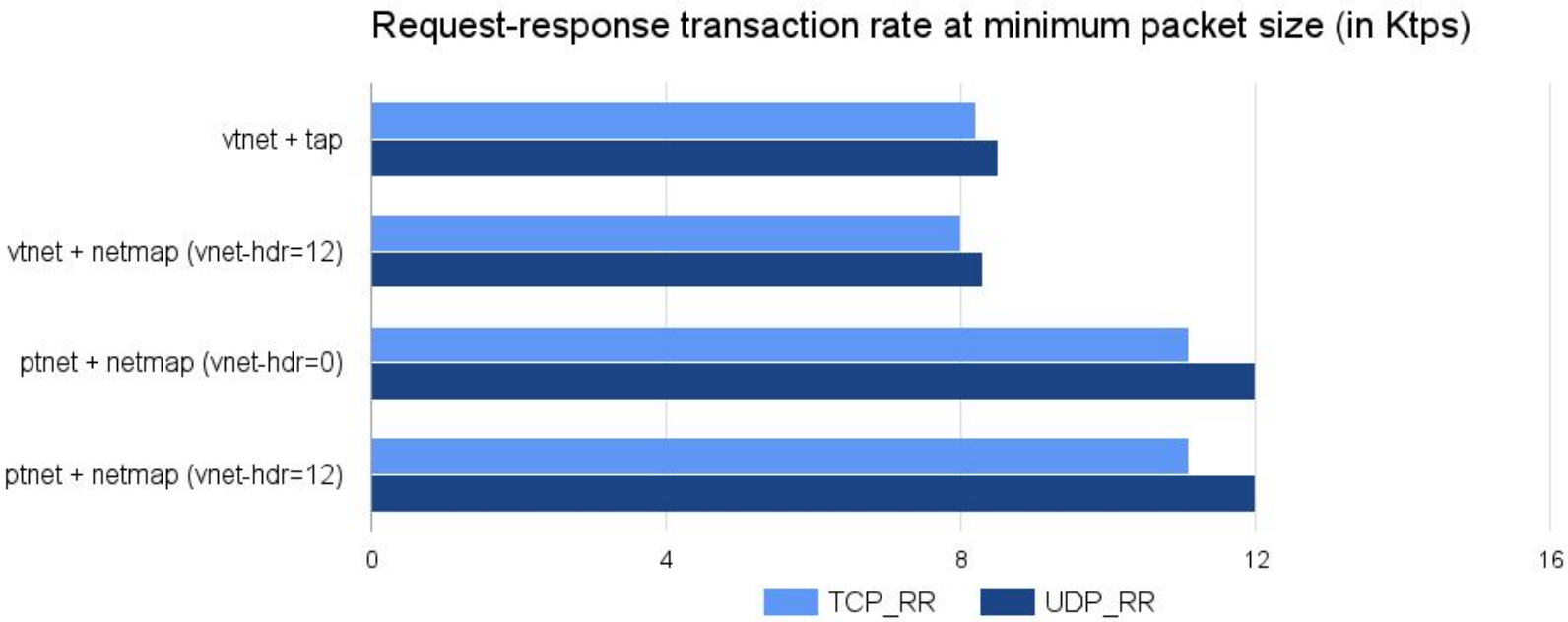

VM #1
netperf
pkt-gen
Network stack
netmap
vtnet0
ptnet0
10.10.9.1
10.10.10.1
VALE switch instance
vale:1
vale:2
VM #2
pkt-gen
netperf
netmap
Network stack
ptnet0
vtnet0
10.10.10.2
10.10.9.2
if_bridge instance
tap1
tap2
bridge0
10.10.9.200

# Performance - Ping round-trip-time



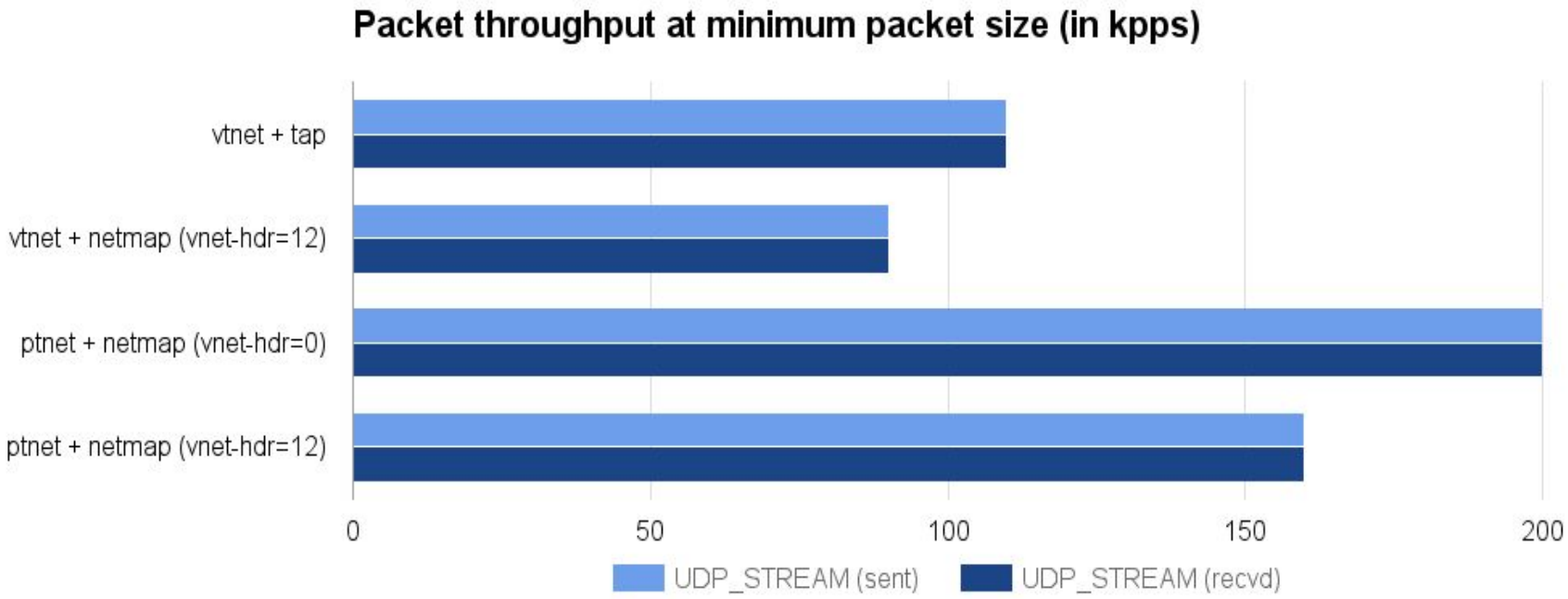

- Results collected with “`ping -fq`”.

# Performance - TCP/UDP request-response latency



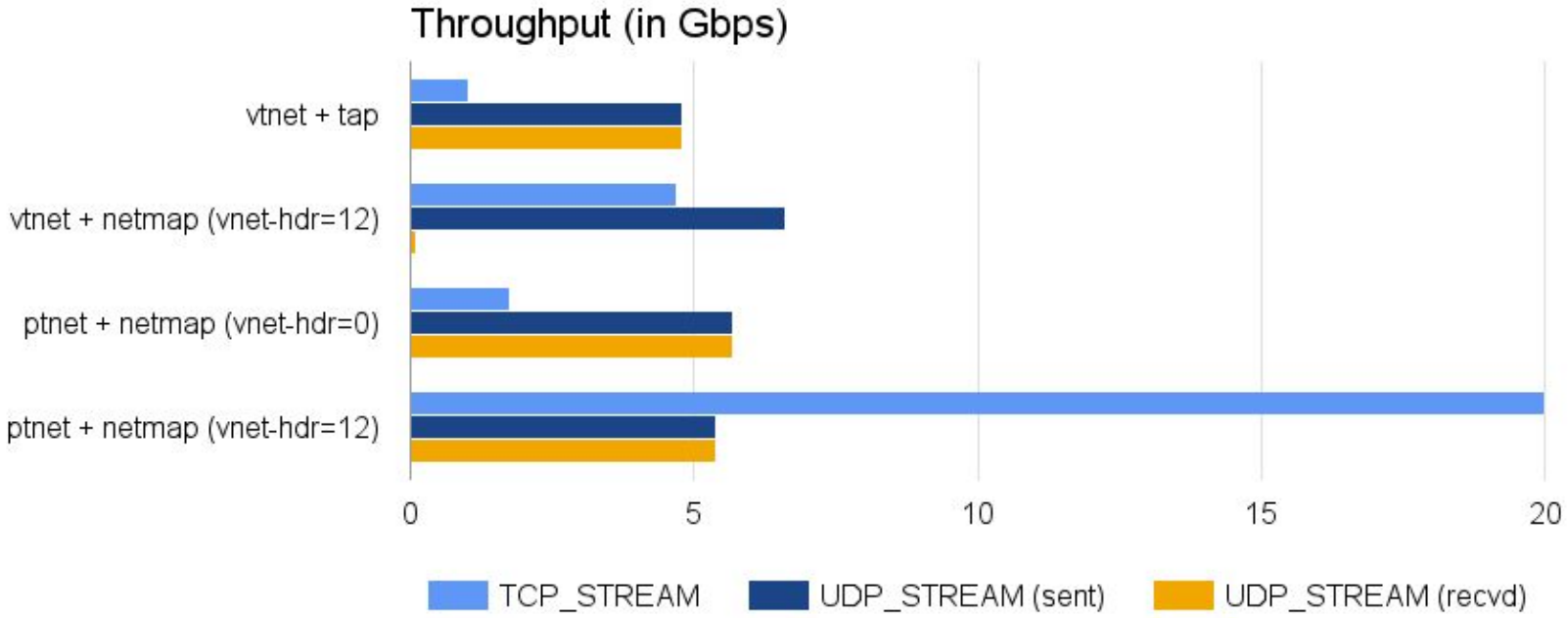

- Latency is low for ptnet, since the packet journey is shorter and simpler!

# Performance - TCP/UDP packet-rate throughput



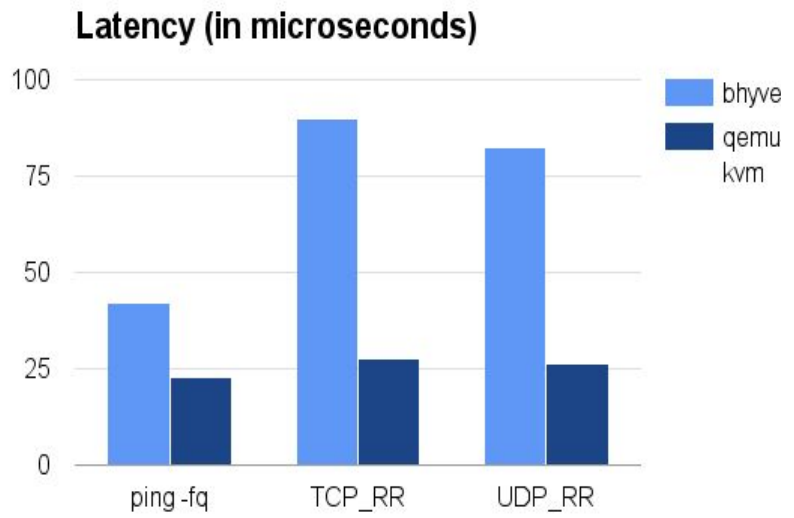

- Performance heavily affected by SP/SC synchronization.

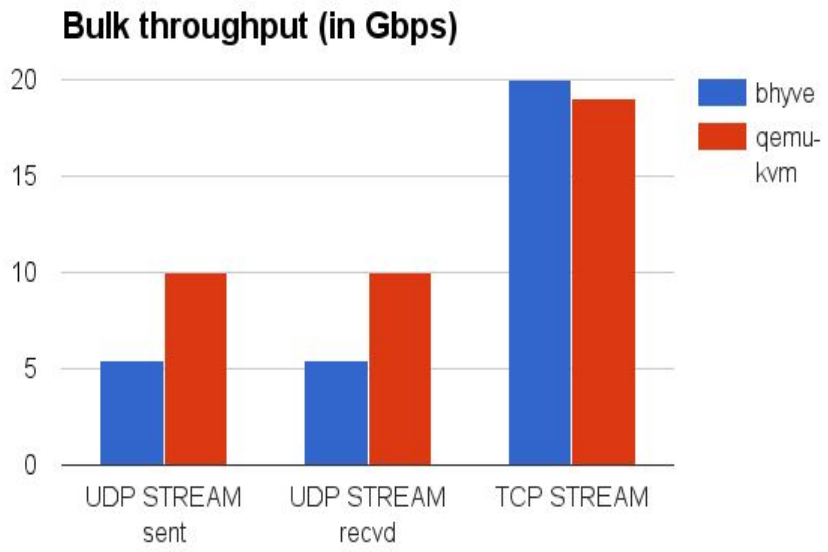
# Performance - TCP/UDP bulk throughput




- TCP works well because of (1) offloadings, (2) reduced latency
- FreeBSD does not support UDP Fragmentation Offloading (UFO)

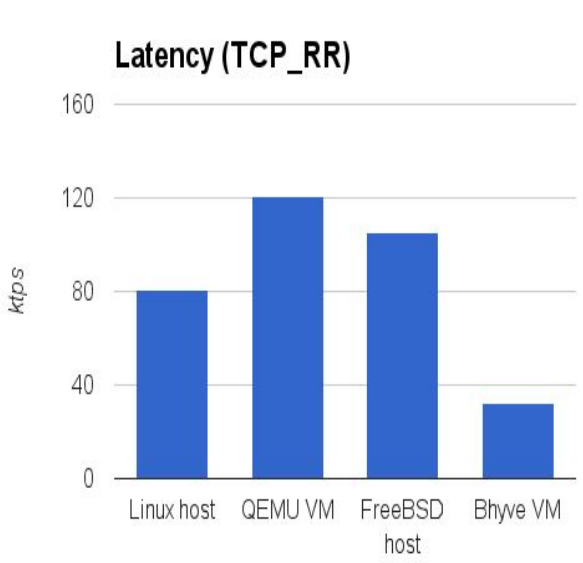
# Performance - bhyve vs QEMU-KVM

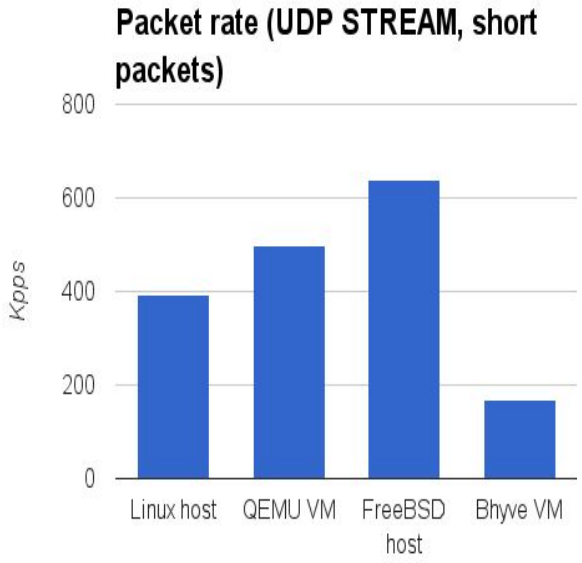


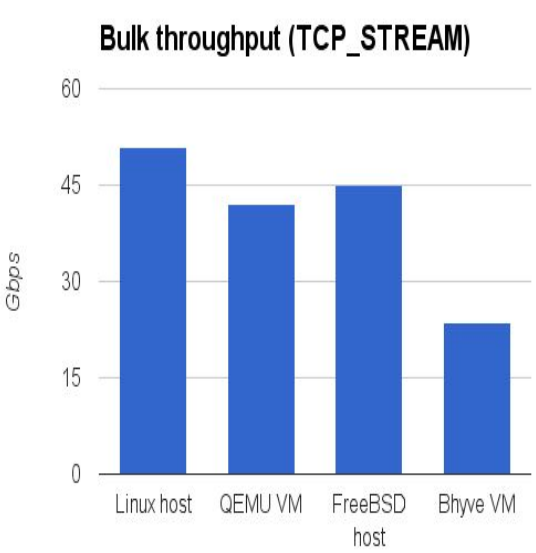




- The only difference is in kick/interrupt implementation, which should not make a big difference
- A latency problem is laying somewhere else

# Performance - Possible bhyve execution overhead?



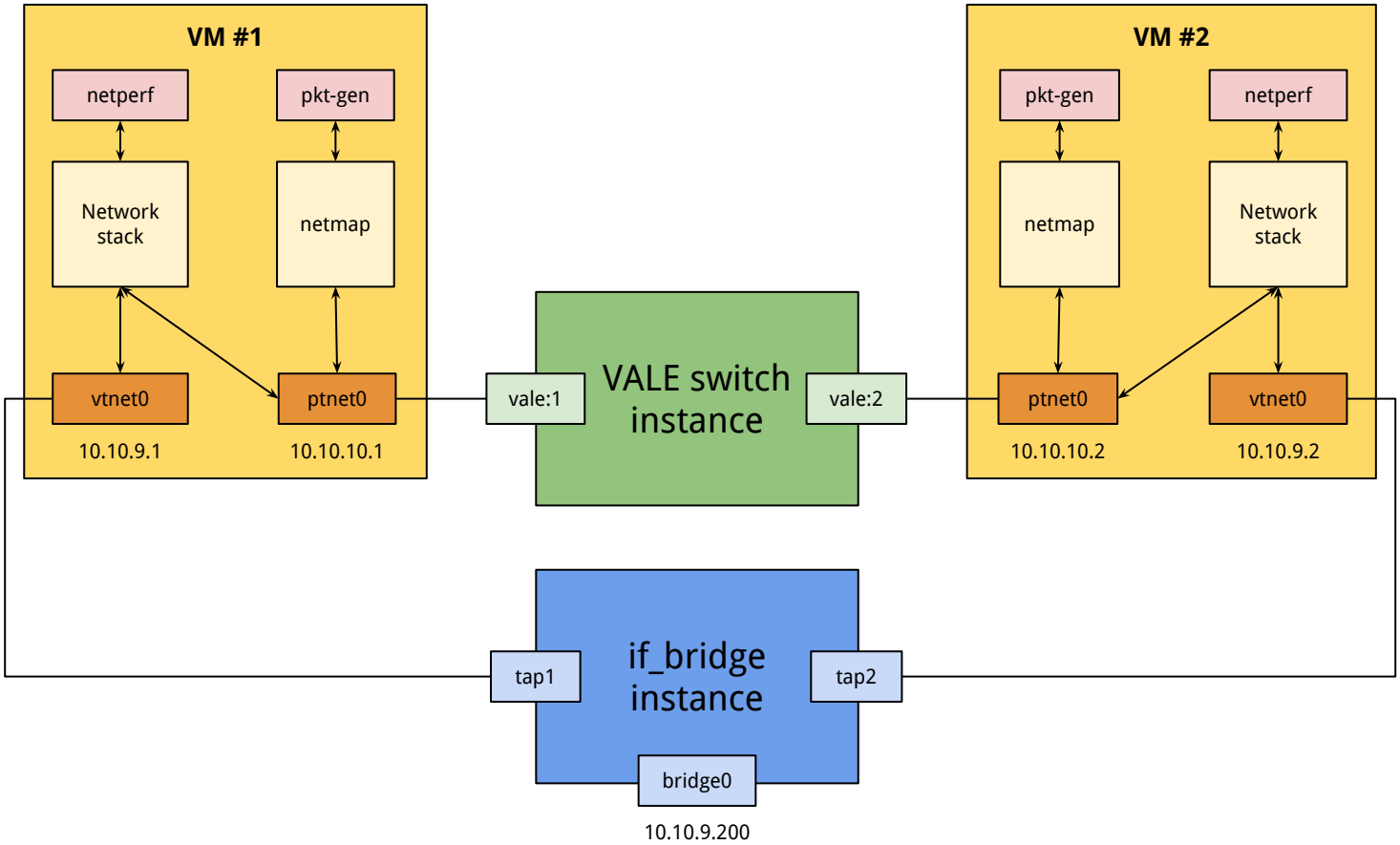




- Netperf tests on the loopback device: physical vs virtualized, no I/O involved.
- Bhyve seems to run code slower
  - HPET virtualization is missing?
  - Overhead in memory virtualization?

# Demo


# Demo setup


VM #1
netperf
pkt-gen
Network stack
netmap
vtnet0
ptnet0
10.10.9.1
10.10.10.1
VALE switch instance
vale:1
vale:2
VM #2
pkt-gen
netperf
netmap
Network stack
ptnet0
vtnet0
10.10.10.2
10.10.9.2
if_bridge instance
tap1
tap2
bridge0
10.10.9.200

# Thanks!

**Contact:**

- **Vincenzo Maffione <v.maffione@gmail.com>**