©2016 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Flexible Virtual Machine Networking Using Netmap Passthrough

Vincenzo Maffione, Luigi Rizzo, Giuseppe Lettieri
Dipartimento di Ingegneria dell'Informazione
Università di Pisa, Italy
Email: v.maffione@gmail.com, {rizzo,g.lettieri}@iet.unipi.it

***Abstract*—The rising interest in Network Function Virtualization (NFV) and SDN paradigms requires Virtual Machines (VMs) to operate with diversified networking workloads, from traditional, bulk TCP transfers to novel ones featuring extremely high packet rates. In response, researchers have explored and proposed new solutions for high performance VM networking, including optimizations to virtual network adapters (such as VirtIO) to support high speed bulk traffic, and alternative frameworks for userspace networking and physical or virtual passthrough.**

To date, we are still missing a comprehensive solution that supports such extreme workloads across multiple operating systems and hypervisors, while at the same time addressing other requirements such as ease of configuration, operating system independence, scalability and isolation.

In this paper we present *ptnet*, an approach to VM networking that provides high performance for both traditional TCP/IP and high packet rate applications. *ptnet* leverages the features of the `netmap` **framework (including virtualization and passthrough support), and defines a simple yet performant network device model that can be easily supported in different operating systems and hypervisors. We prove the effectiveness of our approach by comparing *ptnet*'s performance with one of the state of the art VM networking solutions, namely VirtIO on Linux and QEMU/KVM. *ptnet* is available under a BSD license as part of the netmap distributions on** `github`**.**

I. Introduction

The strong industry interest for Cloud Computing and Data Center solutions, together with recent developments of Network Function Virtualization (NFV) and Software Defined Networking (SDN) architectures, has led to explore different approaches for Virtual Machine networking.

Platforms must simultaneously support high performance, multi threaded, bulk TCP communication, and high rate packet processors. At the same time, they must provide isolation between VMs belonging to different tenants, scalability, compatibility with different operating systems and virtualization platforms, and ease of configuration.

Current solutions address only a subset of these requirements, as we discuss next.

Specialized virtual network devices (`virtio-net`, `vmxnet`) that address bulk TCP communication, often deliver high performance only on specific OSes and hypervisors. As an example, one of the most successful open-source solutions for VM networking, VirtIO [14], is available for different hypervisors (QEMU, bhyve, Xen, VirtualBox) and operating systems (Linux, FreeBSD, Windows), but the most performant and feature-rich implementation is for QEMU running on Linux with KVM enabled.

Applications that require extremely high packet rates normally relies on hardware support in the NIC (exporting multiple NIC instances to virtual machines) and in the hypervisor (PCI passthrough). These features give the guest OS direct access to hardware resources, and let it use "kernel bypass" solutions such as DPDK [8]. At the same time, this approach introduces a scalability problem (as the number of virtual NICs that can be exposed is limited, and they all go through the PCI bus which is a significant bottleneck), and dependencies on specific hardware.

In the past we have worked extensively in this space, addressing high packet rates, device independence and isolation. Our netmap framework [9], high performance VALE switch [11], hypervisor enhancements for high packet rates [12] and virtual passthrough solutions [7] led to excellent performance in terms of packet rates, complete device independence and extreme simplicity in terms of configuration. However, they do not match state of the art solutions such as VirtIO when dealing with bulk TCP traffic.

In this work we present *ptnet*, a solution for VM networking that uses the netmap data format as the underlying virtual device model. This lets us build very simple device drivers that natively support multi queue operation, and leverage our previous `netmap` passthrough work [7], to communicate with hypervisors at rates of tens of millions of packets per second. The contributions of this work, compared to [7], are (i) a new device model for paravirtualized devices, (ii) efficient access to passthrough ports even for legacy applications, (iii) a performance evaluation of our solution comparing it to that of VirtIO, and finally (iv) an actual implementation of both guest OS and hypervisor support of *ptnet*.

We show that *ptnet* can reach or beat VirtIO networking in terms of performance and flexbility, with the additional capability of supporting fast `netmap` applications (10..100 Mpps). Finally, considering that `netmap` is currently available under Linux, FreeBSD and Windows, we argue that the effort of supporting *ptnet* on other platforms than Linux/QEMU - with the same performance – will be lower compared to VirtIO.

A *ptnet* prototype for Linux and QEMU-KVM is already available as opensource as part of the `netmap` distribution on `github.com/luigirizzo/netmap`. Work is in progress

on a FreeBSD and bhyve version.

In the rest of this paper, Section II provides some background about `netmap` and its passthrough virtualization technology, as well as VirtIO; Section III describes *ptnet* architecture and implementation, and a comparison with VirtIO networking; Section IV provides experimental evaluation of *ptnet*.

II. Background

We briefly describe two representative mechanisms used in VM networking, namely VirtIO and the `netmap` framework.

A. VirtIO and I/O paravirtualization

VirtIO [14] is a widely deployed standard for I/O paravirtualization, similar to other paravirtualized devices, where the guest (VM) device driver is aware of running inside a VM, packets are exchanged with the hypervisor through shared memory Virtqueues (VQs), and the device is just a wrapper around the VQs.

A Virtqueue contains: (i) a ring to exchange I/O buffers for read/write operations; (ii) ring indices for synchronization; (iii) control flags. The VirtIO networking device (*virtio-net*) can have one or more pairs of VQs, each pair providing a TX and RX ring.

In the datapath, packets and the state of the queues can be read through shared memory. Emulated I/O registers (which cause a trap into the hypervisor) are only used for guest-to-host notifications. Host-to-guest notifications use hypervisor-specific interrupt injection mechanisms (usually MSI-X interrupts, as they have less overhead than traditional PCI interrupts).

Even with hardware assisted virtualization, nowadays universally available, notifications are extremely expensive (in the order of 1 μs), since they involve VM enter/exits; thus VirtIO has mechanisms to amortize the cost of notifications over multiple I/O operations. Both the guest device driver and the hypervisor suppress notifications while they are actively polling the VQ, and enable them only before going to sleep because no more work is pending. This strategy is usually very effective at reducing notifications under heavy load. An analisys of these kinds of producer-consumer synchronization mechanisms is available in [10].

B. Netmap

`netmap` [9] is a framework for high performance network I/O. It exposes a hardware-independent API that lets userspace application to interact in an efficient but protected way with NIC rings to send and receive Ethernet frames. Applications are in charge of performing packet processing (e.g. playing with TCP/IP headers) in user-space, bypassing the in-kernel network stack.

When applications start using a NIC through the `netmap` API (figure 1), the NIC is said to switch to `netmap` mode: the NIC datapath is intercepted by `netmap` and cannot be used directly by the network stack. When no more applications are using the NIC through the `netmap` API, the NIC switches back to normal mode.

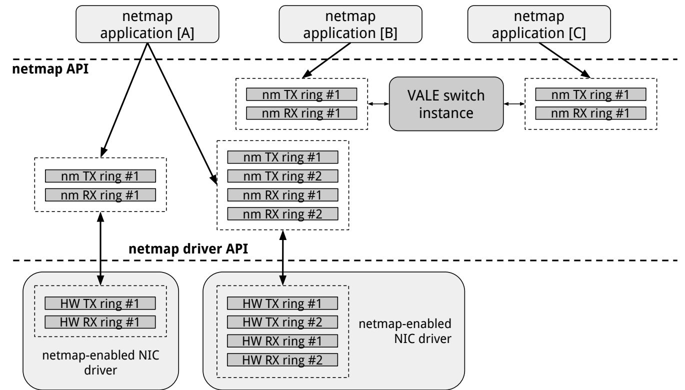

Figure 1: The netmap framework. In the example, application A works on two hardware NICs, while applications B and C communicate through a VALE switch (packet buffers are not shown).

In `netmap` mode, rings are always accessed in the context of application system calls, and NIC interrupts are only used to notify them about NIC processing completion. This is different from what happens with traditional network stacks, where interrupts usually schedule a kernel thread to perform receive/transmit packet processing asynchronously w.r.t. applications.

The performance boost of `netmap` over traditional socket API comes from three sources: (i) batching, since it is possible to send/receive hundreds of packets with a single system call and/or under the same lock; (ii) preallocation of packet buffers, which saves from the cost of dynamic allocation; (iii) memory mapping of packet buffers in the application address space, to avoid a packet copy across the user-space/kernel-space boundary. These techniques represent a considerable leap from traditional in-kernel network stacks, and are they key to achieve impressive performance - e.g. line-rate on 10 Gbps NICs (14.88 Mpps), with minimum-size packets, using just a fraction of the capacity of a CPU core.

Compared to other user-space networking frameworks such as DPDK [8] and PF_RING-DNA [5], `netmap` has full support for blocking I/O, does not rely on busy wait, does not require exclusive use of the NIC, and presents a considerably richer set of port types (NICs, virtual switch, pipes, monitors, passthrough) and operating systems and hypervisor support.

C. Netmap passthrough

`netmap` has been widely extended to support virtualization. We have built a fast netmap-based software switch (VALE [11]), supporting communication and isolation between ports at rates of up to 20 Mpps; netmap *pipes* for point to point communication at over 100 Mpps; `netmap` adapters for paravirtualized drivers; and support in hypervisors such as QEMU and `bhyve`, to allow `netmap` applications to run in a VM at up to 5 Mpps.

The large speed gap between native netmap ports (VALE/pipes), and netmap application running in the guest is caused by the different data format between the netmap ports and the devices (either emulated or paravirtualized) made available to the VMs. These differences require format conversions, and especially data copies, that at the speeds of interest introduce significant slowdowns.

To overcome these limitations, `netmap` passthrough [7] (*ptnetmap*) has been introduced as a technique to completely avoid hypervisor device emulation in the packet datapath, unblocking the full potential of `netmap` also for VM environments.

With ptnetmap, a `netmap` port on the host – a VALE port, an hardware NIC, etc. – is exposed to the guest in a protected way, so that `netmap` applications running in the guest can directly access the rings and packet buffers of the host port, avoiding all the extra overhead involved in the emulation of network devices.

The passthrough is transparent to guest `netmap` applications: they don't need modifications to run on ptnetmap ports.

The txsync/rxsync system calls issued on a ptnetmap port don't operate directly on the corresponding host port. Instead, these requests are forwarded to some kernel threads running in the host – one per ring in the current architecture – which take care of actually handling them.

To make request forwarding possible, ptnetmap guest driver needs to exchange information with the host kernel threads, so that the can synchronize guest rings' indices (as seen by the guest `netmap` application) with the rings' indices of the host port.

To all intents and purposes, guest driver and kernel threads form a producer-consumer system (one per ring).

Similarly to VirtIO paravirtualization (section II-A):

- A shared memory data structure called Communication Status Block (CSB) is used to exchange ring indices and notification suppression flags.
- I/O registers and MSI-X interrupts are used to let guest and host `netmap` wake up each other on CSB updates.
- Notifications are suppressed when not needed, i.e. each time the producer or the consumer is actively polling the CSB to check for more work. From an high-level perspective, the system tries to dynamically switch between polling operation under high load, and interrupt-based operation under lower loads, which is the same idea introduced by Linux NAPI [2], [15].

III. PTNET

The original ptnetmap implementation [7] uses a slightly modified version of e1000/virtio-net guest drivers and hypervisor device emulators, where only the notification part of those devices is used, while the datapath (e.g. e1000 rings or virtio-net VQs) is completely bypassed. This approach requires minimal modifications to the drivers and hypervisors, but also prevents the use of such ports as regular network interfaces for legacy applications running on the guest. This limitation is inconvenient and would prevent adoption by upstream communities (FreeBSD, Linux, QEMU, bhyve, etc.).

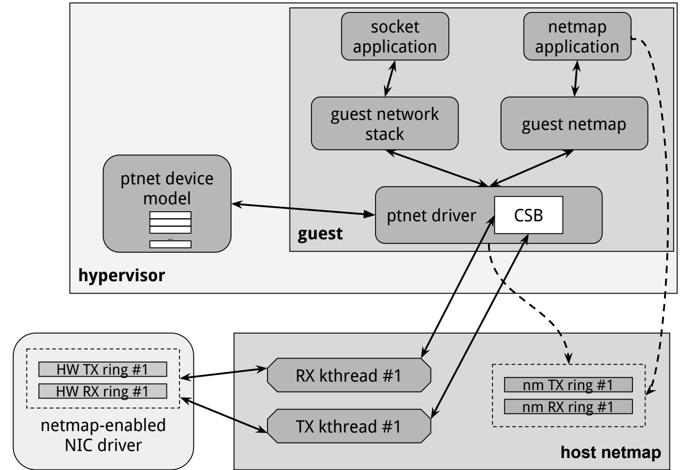

Figure 2: The *ptnet* driver used by a VM to passthrough an hardware interface of the host. Both guest `netmap` applications and the *ptnet* driver (when used by the network stack) access the mapped `netmap` rings.

The key contribution of this paper is the design of a new paravirtualized device model, *ptnet*, which tries to be both efficient and compatible with the legacy network stack. In order to preserve the efficiency of ptnetmap even in legacy mode, *ptnet* uses the netmap API as the native communication format with the hypervisor, thus replacing Virtqueues or specific device models.

A guest OS can use a *ptnet* interface through either the `netmap` API applications or the traditional socket API, as shown in figure 2. If a passed-through host `netmap` port has multiple TX/RX queues, the corresponding ptnet interface will have the same configuration.

The introduction of a new device model does not limit the adoption of this solution, since *ptnet* drivers are distributed together with `netmap`, and hypervisor modifications are needed in any case to map `netmap` host memory in the guest.

A CSB – cleaned up and extended to support an arbitrary number of rings – is used for producer-consumer synchronization and notification suppression, as explained in section II-C.

When used by guest network stack, the *ptnet* device driver acts as any other guest `netmap` application, accessing the `netmap` rings and buffers. Note that the passed-through host `netmap` port is always open in `netmap` mode, while the guest *ptnet* interface is not in `netmap` mode when used by the network stack.

We have developed a version of *ptnet* driver for Linux and the associated device model emulation for QEMU. The ptnetmap support for the host (i.e. the kernel threads) has been modified to support multiple rings and the new CSB, but otherwise still works and performs as described in [7].

A. The ptnet device model

ptnet uses the netmap API as the underlying device model, replacing hardware-oriented models (such as e1000) or Virtqueues. The *ptnet* CSB is laid out as an array of structures, one per ring. Each structure contains the following information:

- head/cur pointers, reflecting the status of the ring as seen by the guest. These are written by the guest and read by the host.
- hwcur/hwtail pointers, reflecting the status of the ring seen as seen by the host. These are written by the host and read by the guest.
- Two flags to suppress guest-to-host and host-to-guest notifications.

A small number of device registers are used to read configuration (number of rings and descriptors per ring, device MAC address, acknowledged features), or to write configuration (CSB physical address, wanted features). A command register is available to start or stop kernel threads. A full description of the register layout is not reported here for space reasons.

For each ring, a dedicated *kick* register is used for guest-to-host notifications. Using different registers is important for performance since each register is associated to a different kernel thread. Using a single register could cause lots of unnecessary wake ups.

A similar approach is used for host-to-guest notifications: a different MSI-X interrupt vector is allocated for each ring, so that different guest applications listening on different rings do not suffer from spurious wake-ups.

B. Paravirtualized offloadings

Hardware NICs commonly support TCP/IP offloadings, the most useful ones being TCP Segementation Offloading (TSO) and TCP checksumming offloading. When used together, the network stack can pass to the NIC driver a TCP segment much bigger than the interface MTU, usually 64KB long, with the TCP checksum not computed. The driver will then program the NIC to perform TCP segementation and checksum computation in hardware.

Performance improvements are twofold: (i) segementation and checksum processing is done faster in the hardware and the CPU can be used for other purposes; (ii) the network stack is traversed less times, since packets are bigger, and so fixed overheads are amortized over more bytes.

When coming to I/O paravirtualization, offloadings can be exploited rather than being emulated. When the device emulator receives a large unchecksummed TCP segment, no segmentation or checksum computation is performed. Instead, the packet is directly pushed into the host network stack (or into a VALE switch), where it pops up again as a big unchecksummed TCP segment.

If the packet is directed to a paravirtualized network device of another guest on the same host, similarly, it is received by the guest driver and injected into the guest stack as is, without further processing. Checksum is unnecessary because the journey of the packet from one guest to another only goes through memory copies.

From an high level perspective, this strategy allows TCP segmentation and checksumming to be performed lazily, and only if necessary - i.e. when the packet needs to go on a physical link.

VirtIO network devices use this technique, which is the key to achieve huge throughputs in terms of TCP (or UDP) bulk transmission. Skipping checksum computation also improves latency.

In order to implement paravirtualized offloadings, however, guest and hypervisor needs to exchange some metadata for each packet, including: (i) offset and length to possibly perform checksum; (ii) Maximum Segment Size (MSS) to possibly perform segmentation. The VirtIO standard [13] defines an header to contain such information, to be prepended to each Ethernet frame.

The *ptnet* device supports paravirtualized offloadings using this standard header, which is already supported by the VALE switch and QEMU. This opens the doors for high performance TCP/IP networking and ensures full interoperability between *ptnet* and virtio-net devices.

C. Comparison with VirtIO networking

As VirtIO networking is the most successful and commonly deployed VM networking solution, we use it as a reference to compare with our solution, taking into account features, flexibility, performance and code reuse. Performance comparison is reported in section IV.

The two solutions are equivalent in terms of features and flexibility when used to build an intra-host virtual LAN between VMs, also considering that they both use virtio-net headers. On most hypervisors (QEMU/bhyve/Xen), VirtIO uses standard in-kernel bridge and TAP interfaces to connect VMs together (VirtualBox uses an equivalent custom bridge implementation). With *ptnet*, the VALE switch is used to provide inter-VM connectivity.

When a guest wants eclusive access to an host physical interface, with *ptnet* we can directly passthrough that interface, which can then be used without the need of a software switch.

VirtIO does not support this kind of passthrough and exclusive access: the interface can be attached to an in-kernel bridge, and the guest can connect to the same bridge using a TAP interface.

Differently from PCI passthrough techniques [6], *ptnet* does not require hardware support and consistent hypervisor modification to passthrough an host physical interface: the `netmap` software is reused to provide most aspects of this feature.

VirtIO networking has recently been extended (on Linux/QEMU) to support multi-queue, which required modifications to the hypervisor and the TAP driver (in addition to the extension of the virtio-net guest driver). These modification, however, are not currently available on other platforms (FreeBSD, bhyve, Windows, etc.). With *ptnet*, instead, there is no need to modify hypervisor or backend drivers: `netmap`

already support multi-queue ports from its inception, and the *ptnet* guest driver is able to use all the queues exposed by the pass-through interface. Multi-queue support, as a consequence, comes for free together with `netmap`.

Mixing together the concepts of device passthrough and paravirtualization, *ptnet* is an hybrid solution that allows for more flexibility than VirtIO with small hypervisor modifications (~1000 code locs for QEMU).

IV. Experimental Evaluation

We now complete the comparison started in section III-C with performance measurements in terms of throughput and latency. Our tests machine has an Intel Core i7-3770K CPU at 3.50GHz (4 core / 8 threads), 8GB RAM DDR3 at 1.33GHz, We run Linux 4.4.5 with a recent QEMU (git master 888ea9, Feb 2016) as hypervisor, extended with ptnetmap support. Each guest VM has 1 vCPU and runs Linux 4.4.5.

For VirtIO tests, we always enable the vhost-net [4] optimization, which accelerates VirtIO networking by performing device emulation directly within the kernel.

A. TCP/IP tests

For TCP/IP tests, we use the popular `netperf` tool, run between two VMs connected through a software switch: a VALE switch for *ptnet*, and an in-kernel Linux bridge for vhost-net. These tests document the latency and throughput of communications going through standard sockets and the regular network stack. Each single experiment lasts 10 seconds and is repeated 10 times. Unless stated differently the worst standard deviation is under 2%, so we only report the averages.

Latency tests: These experiments measure the Request/Response transaction rates at different message sizes, for both UDP and TCP, which give a measure of the round-trip time. The results, presented in Figure 3, show that *ptnet* performs better than vhost-net at smaller message sizes, while the two mechanisms are equivalent for very large messages. *ptnet* gains over vhost-net because the passthrough mechanism does not require device emulation. On the other hand, *ptnet* requires one copy for each transmitted/received packet[1] (so four more copies than vhost-net for each transaction), and this reduces the gap at larger message sizes.

Streaming tests: With these experiments we measure the throughput of the communication path, which is expected to grow with the message size as the size-independent cost of system calls and protocol processing is amortized more. For the TCP_STREAM tests we focused on the common case of large messages, with the transmitter sending maximum sized TSO packets (64 KiB each). In this configuration *ptnet* achieves about 20 Gbps, while vhost-net is able to reach 26 Gbps. The performance gap is due to the additional packet copies made by *ptnet*, that increase the CPU load, eat memory bandwidth and also increase the latency in the communication.

[1] In the netmap architecture, buffers are provided by netmap, not by applications, and so an additional copy is needed between OS packet representation (i.e. `sk_buffs` for Linux) and netmap buffers. We are currently investigating the use of netmap indirect buffers to avoid these copies.

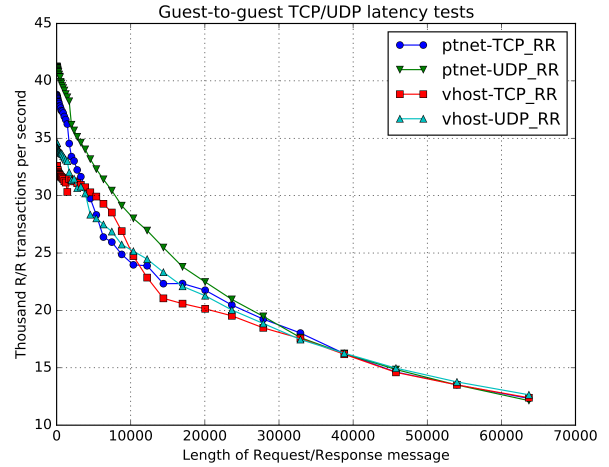

Figure 3: Guest-to-guest latency tests, with *ptnet* performing better than VirtIO because no device emulation is involved, even with the (four) additional packet copies per transaction.

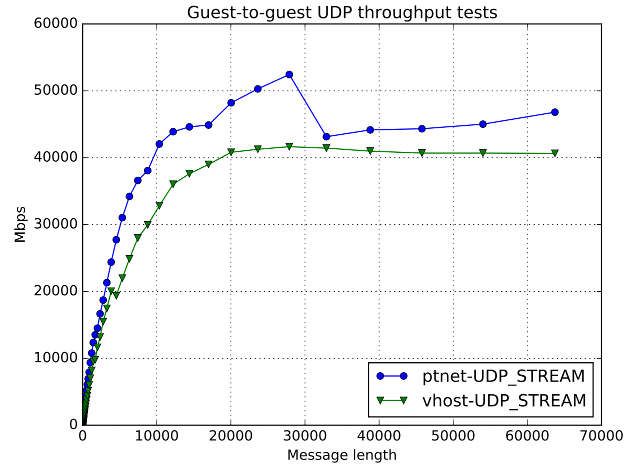

Figure 4: Guest-to-guest UDP throughput measured at the transmitter. Despite doing more copies, *ptnet* transmits faster.

Results for UDP_STREAM are shown in Fig. 4. We only report the transmit throughput because at high rates the lack of flow control causes the receiver to enter a livelock regime, where 50-80% of the received UDP packets are dropped at the receive socket buffer, before netperf can read them.

Throughput for both VirtIO and *ptnet* stops growing beyond 20-30 KiB, when the size-independent cost stops dominating the cost of copies. In spite of the additional copies, however, *ptnet* performs better at all sizes. The drop for messages larger than 30 KiB coincides with an increase in the standard deviation of the results (about 15%), and is related to the speed mismatch between sender and receiver as discussed in [10]. In particular, we are in a "Fast Consumer" regime, described in section IV-B.

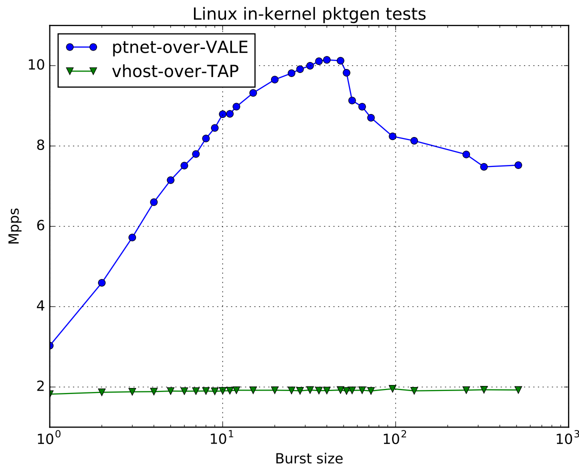

Figure 5: Transmission tests using Linux pktgen with variable batch size. VirtIO is not able to exploit the batching, while *ptnet* makes good use of netmap batching API.

B. Linux in-kernel pktgen tests

Recent improvements in the Linux kernel [1], [3] try to leverage on batching to improve throughput, without artificially introducing latency. Since [1], the kernel can give the NIC driver a batch of packets ready for transmission, so that the driver can notify the whole batch to the NIC, if possible.

To measure the effectiveness of this approach, which can be used with traditional socket applications, we run the Linux in-kernel packet generator in a guest, sending batches of minimum-sized packets (60 bytes) at maximum speed through a *ptnet* (over VALE) or virtio-net device (over TAP), with variable burst size, and measuring the packets received on the host. Results are shown in figure 5.

The VirtIO API has limited support for batching, and therefore using batch transmission with virtio-net is not really effective at improving throughput - only guest-to-host notifications are delayed.

On the contrary, *ptnet* does a good job when driven with batches, since the batches are preserved when going through the passed-through netmap port. The throughput nicely increases with the batch sizes, reaching its optimium at about 50. After that, throughput decreases. As explained in [10], the decrease happens because for larger batches the producer-consumer systems enters a Fast Consumer (FC) regime. The consumer (ptnetmap TX kthread) becomes more efficient w.r.t. the producer (pktgen guest thread), and so the producer slows down because it spends more time notifying the consumer.

C. Tests with netmap applications

In the same scenario of section IV-A, we used the Netmap pkt-gen tool over *ptnet* to measure guest-to-guest throughput at minimum packet size (60 bytes). We measured ~19.5 Mpps when using virtio-net headers and ~21 Mpps when not using them. Note that disabling virtio-net headers is a reasonable choice for VMs implementing middlebox functionalities (e.g. NFV nodes), since virtio-net headers are mostly useful if the VM is the endpoint of TCP or UDP streams.

Other tests with *ptnet* on different network configurations (e.g. with hardware ports, pipes, etc.) are not reported here for space reasons, but the performance numbers are the same as reported in [7].

V. Conclusion

This work presents *ptnet*, a solution for VM networking, comparing it with the current state of the art - VirtIO networking. We have seen that ptnet and VirtIO performance are comparable with traditional socket applications, whereas *ptnet* also supports guest applications which are able to benefit from the Netmap API.

Finally, we explained how *ptnet* can be easy ported to other platforms with similar performance, since most of its implementation is already shipped within Netmap. In the short term, we plan to develop a *ptnet* driver for FreeBSD and the device emulation model for the bhyve hypervisor. This is particularly interesting for the FreeBSD project, since it does not currently have an high performance VM networking solution such as vhost-net.

Acknowledgment

This paper has received funding from the European Union's Horizon 2020 research and innovation programme 2014-2018 under grant agreement No. 644866. This paper reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] Bulk network packet transmission. https://lwn.net/Articles/615238/.
[2] Linux Net:NAPI ("New API"). http://www.linuxfoundation.org/en/Net:NAPI.
[3] Qdisc bulk dequeueing. https://lwn.net/Articles/615240/.
[4] Vhost architecture. http://blog.vmsplice.net/2011/09/qemu-internals-vhost-architecture.html.
[5] L. Deri. PFRING DNA page. http://www.ntop.org/products/pf_ring/dna/.
[6] B.-Y. et al. Utilizing iommus for virtualization in linux and xen. In *Proceedings of the Linux Symposium*, 2006.
[7] S. Garzarella, G. Lettieri, and L. Rizzo. Virtual device passthrough for high speed vm networking. In *Proceedings of ACM/IEEE ANCS 2015*, pages 99–110, 2015.
[8] Intel. Intel data plane development kit. *http://edc.intel.com/Link.aspx?id=5378*, 2012.
[9] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC'12*, Boston, MA. USENIX Association, 2012.
[10] L. Rizzo, S. Garzarella, G. Lettieri, and V. Maffione. A Study of Speed Mismatches Between Communicating Virtual Machines. *IEEE/ACM ANCS*, 2016.
[11] L. Rizzo and G. Lettieri. VALE, a Switched Ethernet for Virtual Machines. *ACM CoNEXT*, 2012.
[12] L. Rizzo, G. Lettieri, and V. Maffione. Speeding up packet I/O in virtual machines. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '13, pages 47–58, Piscataway, NJ, USA, 2013. IEEE Press.
[13] R. Russel, M. Tsirkin, and C. Huck. http://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.html.
[14] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.
[15] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond softnet. In *Proceedings of the 5th annual Linux Showcase & Conference*, volume 5, pages 18–18, 2001.