# Virtual device passthrough for high speed VM networking

Stefano Garzarella
Università di Pisa, Italy
stefanogarzarella@gmail.com

Giuseppe Lettieri
Università di Pisa, Italy
g.lettieri@iet.unipi.it

Luigi Rizzo
Università di Pisa, Italy
rizzo@iet.unipi.it

**Draft copy, March 2015. Please do not redistribute. The full paper will appear in IEEE/ACM ANCS 2015, May 2015, Oakland, CA**

## ABSTRACT

Supporting network I/O at high packet rates in virtual machines is fundamental for the deployment of Cloud data centers and Network Function Virtualization.

Historically, SR-IOV and hardware passthrough were thought as the only viable solution to reduce the high cost of virtualization. In previous work [11] we showed how even plain device emulation can achieve VM-to-VM speeds of millions of packets per second (Mpps), though still at least 3 times slower than bare metal. In this paper we present architectural solutions to fill this gap.

Our design, called `ptnetmap`, implements a virtual passthrough network device based on the netmap framework. `ptnetmap` allows VMs to connect to any netmap port (physical devices, software switches, netmap pipes), conserving the speed and isolation of the native netmap system, and removing the constraints of hardware passthrough.

Our work includes two key features not present in previous related works with comparable performance: we provide a high speed path also to untrusted VMs, and do not require dedicated polling cores/threads, which is fundamental to achieve an efficient use of resources. Besides these features, our speed is also beyond previously published values. Running on top of `ptnetmap`, VMs can saturate a 10 Gbit link at 14.88 Mpps, talk at over 20 Mpps to untrusted VMs, and over 70 Mpps to trusted VMs.

`ptnetmap` extends the netmap framework, and currently support Linux and QEMU/KVM. Support for FreeBSD and bhyve is under active development.

## Keywords

virtual machines; netmap; passthrough; NFV

## 1. INTRODUCTION

Virtual Machines (VMs) are one of the most popular tools for building "cloud" services through the composition of multiple independent components. Running services in VMs has a higher cost (especially for I/O) compared to simpler isolation mechanisms, such as separate processes or containers [6]. However, the various components may have conflicting constraints on their required libraries, platforms, kernels. This often makes VMs or separate physical machines the only viable alternative, VMs provide much higher flexibility and ease of management, as well as significant cost reductions, due to the availability of systems with large amounts of memory, powerful I/O, (relatively) large number of cores, and native CPU support for virtualization.

VMs do introduce performance bottlenecks, especially for I/O intensive workloads which are common in networking services involving firewalls, software routers and middleboxes. The techniques used to support virtualization in fact have a hard time dealing with the very high packet rates that may appear on today's fast network interfaces. As a matter of fact, even conventional operating systems running on physical servers ("bare metal") suffer at high packet rates, and in recent years we have seen a number of techniques that try to bypass the entire operating system (commonly called "OS-bypass", such as [5, 2]) or at least the network stack and device driver while still using some OS services (this is called "network stack bypass", as in [9, 13]) to achieve better throughput.

Adopting the same solutions within VMs has historically relied on the use of a technique called hardware passthrough, where the host OS gives exclusive control of a physical device to the guest operating system, which in turn can run its own OS-bypass or network-bypass code on top of the physical device. Hardware support in modern CPUs, chipsets and Network Interface Cards (NICs) can provide proper memory protection (IOMMU) and the illusion of independent I/O ports for each VMC (SR-IOV).

As described in more detail in Section 2, hardware passthrough has drawbacks, such as having to pass all communications (even VM-to-VM) through a (relatively) slow PCIe bus, and binding the VM to a specific device hence making migration difficult [14].

A couple of recent works try to address the performance problem. In [11] we have shown how VMs can communicate very high speed (our current numbers are around 8 Mpps) using standard device emulation. [11] uses the netmap API to communicate with a software switch called VALE [10], and adopts a number of ideas to amortize or eliminate VM exits and some data copies. While extremely fast, our previous work is still 2..4 times slower (or less resource efficient, when the bottleneck is in the hardware) than the best bare-metal bypass techniques [10].

netvm [4] instead leverages the DPDK framework [5] to achieve line rate speed between VMs and 10 Gbit/s interfaces, but at the price of high CPU overhead (DPDK requires active polling to detect events) and flexibility (the use of huge pages to maximize performance prevents the use of the high speed channel with untrusted VMs).

In this paper we present a virtual device passthrough mechanism called `ptnetmap` that uses the netmap API as the "device" model exported to VMs. This makes us completely independent from the hardware, enabling communication at memory speed with physical NICs, software switches, or high-performance point-to point links (netmap pipes).

The two most important contributions of this work are the ability to provide high throughput even to untrusted VMs, and the presence of a proper synchronization framework, that does not require on always-active threads as it is the case for DPDK-based solutions. These features do not impair performance: `ptnetmap` can saturate 10 Gbit NICs (14.88 Mpps with 64-byte frames), and achieve over 20 Mpps between untrusted VMs and over 70 Mpps between trusted VMs.

In the rest of the paper, Section 2 presents the problem in more detail; Section 3 describes the architecture of `ptnetmap`. Section 4 describes the implementation of our system, while Section 5 gives a detailed performance evaluation. Related work is addressed in Section 6.

Our code is part of the netmap framework, and will be distributed with it, together with the (small) modifications for hypervisors and guest operating systems. We currently support Linux and QEMU/KVM, but FreeBSD and bhyve support is in the works.

# 2. BACKGROUND

In this Section we present some background information on the implementation of the network path in VMs, and on the netmap framework that we use to develop our system.

## 2.1 VM networking, standard approach

A "guest" virtual machine normally runs within one or more CPU threads, talking to the virtualization support software (hypervisor) through system calls that in turn access the true or emulated peripherals physically managed by the host system, and assigned to the VM.

As shown in Figure 1, the network device driver in the VM normally interacts with a *frontend* component in the host, which emulates the hardware the guest device driver expects to talk to. Popular examples are the Intel E1000 family,

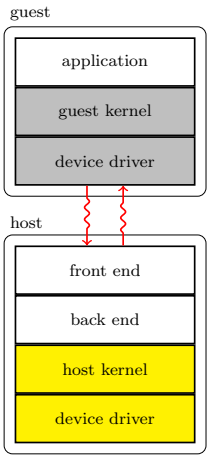


**Figure 1: Standard configuration for the network path in a VM**

or the `virtio-net` device. In this arrangement, data are transferred through in-memory queues and buffers while a set of (emulated) PCI registers are used for the control path, and emulated interrupts dispatch notifications from the host to the guest.

Guest's accesses to the emulated PCI registers trigger "VM exits", i.e. switch of the CPU from guest to host execution mode, where the code in the frontend and backend is run. These exits are extremely expensive (up to several microseconds) because they not only need to save (and later restore) a substantial amount of CPU state, but also because they often trigger subsequent system calls to emulate the desired effect, such as sending or receiving packets. The transfer of data packets to the actual network port on the host (a physical device, a `tap` port on a virtual switch, a `netmap` port) is done through the help of a device-specific *backend* component on the host. Between the frontend and the backend, it is normally necessary to perform a data format conversion, which involves touching every packet and typically also one data copy. While the cost of this conversion is not substantial, it has severe performance implications, especially when running at very high packet rates and with fast backends.

Optimizations do exist, as described in detail in [11], to reduce the number of VM exits and possibly system calls, by placing the frontend and backend in the kernel, and using additional threads to replace exits and interrupts with in-memory communication. However even these optimizations cannot eliminate the need for an additional copy of data and descriptors.

As a reference number, even on bare metal, a typical application running on top of a `socket` or a `tap` interface peaks at 1-2 Mpps and is CPU bound when using short packets. Moving the application to a VM, the additional conversion work adds to the CPU load and reduces the throughput significantly, even with the best implementations [11]. Is it easy to imagine how the performance decays badly with higher packet rates. Consider for instance the case when the guest VM (hence the frontend) uses netmap and transfers packets in batches, and the backend attaches to a VALE port us-

ing netmap. Individually each of these components is able to handle approximately 20 Mpps (at full CPU utilization), but cascading the two ports and adding the conversion and other VM-related overheads brings the throughput down to only a few Mpps.

## 2.2 VM networking, hardware passthrough

To eliminate the cost of device emulation, a solution called hardware passthrough (HPT) has emerged in the past few years, with hardware supporting it. First, the host OS gives the guest VM direct access to some of its hardware peripherals. If the device exports multiple "Virtual Functions" (e.g. multiple logical network interfaces within a single physical device, a functionality called SR-IOV), the design scales reasonably well as each VM gets access to one VF. Protection is implemented using an IOMMU [3], which is used to translate physical memory addresses in requests issued by the VFs into actual physical memory addresses. With this combination of features, the guest OS will be able to access and program the peripherals (VFs) without VM exits, and the host will be able to enforce protected access to memory. All I/O involving the device occurs within the guest OS, including the dispatching of interrupts. The host is only involved in setting up initial access to the device, and then for memory protection, through the use of the IOMMU .

Note that VFs require actual hardware resources, such as separate register sets and buffers, and logic in the device to arbitrate access to the shared blocks, such as the PCIe bus, and the TX/RX units in a NIC.

## 2.3 Accelerating I/O: the Netmap framework

The device driver, both in the host and the guest OS, is a significant bottleneck in the performance of the network stack, and several solutions in recent years have been designed to provide speedier access to the NICs. In this work we leverage the features of the netmap framework, because it is extremely flexible and efficient, and it is a perfect tool to implement our virtual passthrough mechanism.

The netmap framework [9], which includes the VALE [10] software switch, provides high speed network *ports* to clients, which are normally userspace applications (but can also be within the operating system's kernel). Logically, each port is a set of transmit and receive queues and associated packet buffers, which clients `mmap()` to support zero copy I/O (Figure 2, top). The (in-kernel) backend for a port can be a device driver for a physical NIC, a port of a software switch, which enforces isolation between clients through memory copies; a high performance, shared memory channel called netmap pipe (see Figure 2, bottom). Other types of ports are also present, e.g. for mirroring, logging etc.

### 2.3.1 Data access

Clients access netmap by creating a file descriptor with an `open("/dev/netmap")`, binding it to a specific device or port with an `ioctl(fd, NIOCREGIF, ...)`, and then use `mmap()` to access the shared memory. From this point on, data transfers occur through the queues using `ioctl()` for non-blocking I/O, and `select(), poll(), kqueue(), epoll()` for blocking I/O.

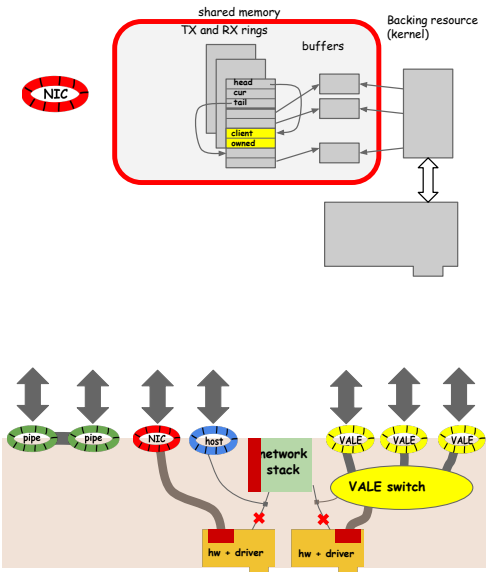


**Figure 2: Top: a netmap port is associated to transmit and receive queues and buffers in shared memory, which are also accessed by different types of backends (in the figure, a hardware NIC). Bottom: the various types of ports supplied by the netmap framework**

Each queue (and its buffers) is logically divided in two regions: one owned by the client, the other owned by the kernel. The boundary between the two regions is marked by two pointers, `head` and `tail`, as shown in Figure 2, top. A third pointer, `cur`, is used for notifications. In detail:

**head** is only updated by the client, and points to the first queue slot owned by the client. It is advanced when the client is done with the buffer, e.g. because it has consumed an incoming packet or filled a buffer for transmission, and is used to pass ownership to the kernel.

**tail** is only updated by the kernel *during a system call*, and points to the first slot owned by the kernel. Is it used to notify the client of newly available buffers, e.g. newly received packets or completed transmissions.

**cur** is only updated by the client, and is used to indicate when the clients want to be notified on a blocking system call. Normally, `head` and `cur` are moved together, and blocking system calls return whenever the client-owned region is non empty. However, at times the client may want to hold some buffers without releasing them (e.g. because it still has to complete processing), or it may need a minimum amount of buffers to complete processing. In these cases, `cur` can be moved ahead of `head`, and the kernel will wake up the client when `tail` moves past `cur`.

The kernel guarantees mutual exclusion with the client by only reading and updating the pointers during system calls,

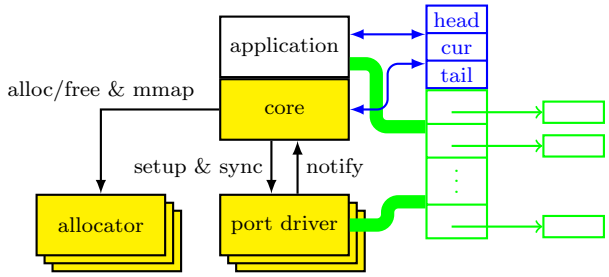


**Figure 3: Internal architecture of netmap. The colored block are part of the kernel, whereas white blocks are in userspace.**

and only accessing the slots and buffers of the queue portion it owns, based on the pointer values observed during the last system call. This gives the application simple semantics, similar to other legacy system calls and free from memory ordering issues.

The netmap API is extremely efficient not only because of memory sharing, which saves some data copies, but also because the pointers can be advanced by any amount (compatibly with the available space), so batching of packet processing is possible in both directions.

The memory region holding queues and buffers can be private to a port (i.e., shared only between client and kernel), or shared with other netmap ports and clients. Private regions are used when the client attached to the port is not trusted, or we want to isolate the port from others in the system. As an example, this is the approach used in the VALE switch, which copies data across its ports. Shared regions are instead used when the isolation is not necessary, either because the clients are trusted, or because it provides no advantage (e.g. point to point links). Traffic between ports attached to the same region can be done by simply swapping buffer pointers, which permits extremely low overhead communication.

## 2.4 Netmap internals

Internally, netmap is implemented by three main blocks, as shown in Fig. 3: the *core* implements all the system calls issued by the applications, and the logic to interact with all kinds of netmap ports; a *port driver* interacts with each type of netmap port (devices, switches, pipes etc.), and a *memory allocator* creates the memory regions used for queues and buffers. All these components reside in the OS kernel, but the distinction is important because in the design of `ptnetmap` some of these functions need to be split between the host and the guest VM.

The allocator is only involved at bind time, when memory regions are created and mapped. Subsequent data transfers only require the interaction between the core and the port driver, through a couple of functions:

- `sync()`, which is used to synchronize the state of the kernel owned parts of the port queues with the underlying device;
- `notify()`, which is used to notify the core that the queues need updating (typically issued as a result of an hardware interrupt).

Note that the core is the only code accessing the queue pointers (in blue), while the port code only accesses the queue slots and buffers (in green). The mutual exclusion rules outlined above allow for the port code to be executed asynchronously with the application. This is also important for the `ptnetmap` implementation, where the port driver runs in a completely different domain from the core and has no access to the running state of the user application.

# 3. PTNETMAP ARCHITECTURE

While in principle the performance of a guest with an HPT device can be very close to that of bare metal, the sharing of resources in the physical device (and the bus) can become a severe bottleneck. On top of this, HPT makes migration difficult as it binds the VM to specific hardware (though this problem can be addressed with a bit of cooperation in the guest OS, see [14]).

To overcome these limitations, we designed a Virtual Passthrough solution, called `ptnetmap`, focused on networking. The NIC "device" exported to the VM is not a piece of hardware but a software port implemented by the netmap [9] framework. Protection is inherited from the underlying netmap port, so we can restrict untrusted VMs to use ports of a software switch [10], or attach trusted or mutually cooperating VMs to ports corresponding to physical devices (with hardware access still mediated by the host OS) or to shared-memory, high performance channels called netmap pipes.

The advantage of `ptnetmap` over HPT are in the fact that the VM is not bound to a specific piece of hardware, so it can be migrated seamlessly to hosts with different backing devices. Also, communication between VMs does not need to go through the PCIe bus, and can in fact run at memory speed, achieving surprisingly high packet and data rates.

At a high level, `ptnetmap` is extremely simple to understand. The difficulty in its implementation comes in building a datapath that avoids the data copies generally involved in peripheral emulation, and in constructing an efficient path to dispatch asynchronous notifications.

The idea of netmap passthrough is to give the netmap core running in the guest kernel direct access to a netmap port instantiated on the host, as shown in Fig. 4. This is a fundamental feature that permits transferring data and descriptors between the host and the guest without expensive copies or descriptor conversions. At the same time, the guest is unaware of the exact nature (physical port, pipe, switch port) of the underlying netmap port. This direct access is achieved by letting the allocator on the guest forward its request to the allocator on the host, as shown in the left part of Figure 4.

The architecture of the control path is slightly more complex. A requirement of the netmap architecture is that ownership of the ring pointers is controlled through system calls. As a consequence, only the guest can access those fields.

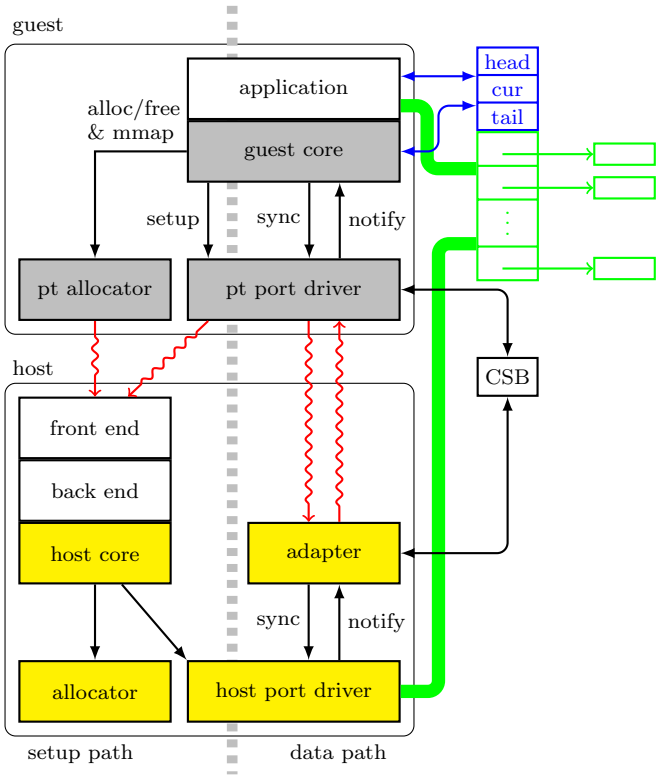


**Figure 4: Netmap passthrough architecture**

Each ring does have a set of (internal) pointers, which are visible only within the kernel components both on the host and the guest. As a consequence, the `sync()` call from specific port driver in the guest ("pt port driver") is passed directly to the host's kernel, either through VM exits or through a shared memory block ("CSB" in the figure). In the host, an "adapter" module (associated with a thread which is woken up on demand) will process these requests, and communicate with the specific host port driver to access the shared rings and buffers, Return information are posted into the CSB and possibly trigger a notify to the pt port driver in the guest, which will in turn update the queue pointers at the next opportunity.

This setup preserves the netmap semantics, since the queue pointers (in blue) are only accessed by the guest netmap core, synchronously with the guest application. Asynchronous updates by the host port will only address the netmap owned portion of the shared queues, as it is already the case for the non-passthrough setup.

In `ptnetmap`, the per-packet cost of transmission and reception becomes exactly the same for a netmap application running in the guest and a netmap application running directly on the host. The per-batch cost, which is paid whenever a netmap system call is issued, is different in the two settings. The actions of the pt port driver are extremely simple, and since it does not access the queue (unlike other port drivers), it may run extremely quickly. On the other hand, sometimes (especially at low data rates), the pt port driver may require a VM exit, which has a big impact especially in the latency of the system call. The actual queue processing is done by a thread associated to the adapter and running in the host's kernel. This thread is *woken up on demand* so it does not consume CPU resources when there is no traffic.

## 3.1 Mitigating VM exits

The techniques to mitigate VM exits and interrupts to the guest VM are described in detail in [11], but we summarize them briefly here. The key idea is to have one thread on each side of the communication that can be activated on demand. In our case, these are the application on the guest, and the kernel thread in the adapter on the host. activated on demand. The threads share a block of memory (CSB) to exchange messages and information on their status.

When one the application wants to notify the other thread, it posts the message in the CSB, and checks (in the CSB) whether the other thread is active; if not, it executing a VM exit to actually wake up the thread and have it process the message. Similarly, in the other direction, the host kernel thread posts the message in the CSB, checks the status of the guest thread, and if necessary issues a guest interrupt, which eventually wakes up the guest thread. Double checks and memory barriers are used to make sure that events are not lost due to data races.

## 3.2 Busy wait versus proper signalling

We conclude this section with a brief discussion of a commonly used approach in building high performance services, namely the use of busy wait instead of proper signalling support.

A critical problem in building a distributed system is notifying components of events. In our context, these events can be packet arrivals or transmit completions, and they are reported by the NIC by setting the content of internal registers and at the same time by triggering an interrupt. The hardware involved is managed by the host OS, but the notifications should be delivered to processes running on the guest OS.

A simple way to detect the event is for the recipient to spin, actively polling the resource (device register, memory location etc.) where the event is reported. The obvious advantages of this approach are that i) none or little[1] system support is necessary, and ii) event detection can be almost immediate. The price to pay, however, is high: a full CPU is burned to spin on the resource even in absence of load, and in case the system has other activities to perform we have no way to implement an asynchronous notification mechanism.

The alternative approach is to build a proper chain of handlers that will react to the interrupts reported by the hardware, and dispatch the information to the processes involved. Conventional device drivers work (and netmap, too) use this technique, which requires a substantial amount of additional code in the system, both to implement a notification delivery mechanism (e.g., creating a file descriptor that can be used to dispatch notifications), and to make sure that

[1] if the resource is not directly accessible, such as a device register for a userspace program, some minimal OS support in the form of a non blocking system call may be necessary.

notifications are not lost due to races. Additionally, notifications may incurs a significant latency (several microseconds). Even in the best case, waking up and starting a sleeping thread, possibly on a different processor, requires the scheduler to perform a global notification to all cores in the system; the newly started thread will work from a cold cache, thus incurring substantial latency to bring data to memory. Finally, especially with high speed network devices, interrupts are typically throttled (a mechanism called interrupt moderation) to avoid overloading the system, and this results in additional latency in the order of 20..50 $\mu$s.

The advantage of proper signalling is however that the system's load grows proportionally with the amount of traffic, and it is possible to have asynchronous notifications thus supporting energy efficient solutions. We should note that having proper signalling only adds flexibility to a system, and does not prevent from running it in polling mode whenever there is a strong reason to.

For these reasons, both netmap and `ptnetmap` include a complete signalling chain.

# 4. IMPLEMENTATION

In this Section we give some details on the implementation of `ptnetmap`. Our system requires some extensions to the netmap code to implement the additional functions, plus some small changes in the guest device driver and the hypervisor.

**netmap:** With reference to Figure 4, the netmap code (which runs both in the host and guest OS) has been extended with two simple additional components:

- **pt allocator:** an allocator for passthrough ports, which simply issues requests (through emulated PCI registers) to the backend for the memory region to use, instead of allocating it locally;
- **adapter** a module that runs in the host's kernel, providing the hypervisor access to the internal data structures associated to a netmap port. It registers with the hypervisor's kernel part (e.g. KVM) to receive notifications on accesses to specific PCI registers.

Additionally, the core is extended to handle the additional requests to set the port in passthrough mode and create the adapter.

**guest device driver:** The netmap-related part of the guest device driver is what we call the "port driver". This code is logically part of netmap, and the required modifications involve just exchanging notifications with the adapter on the host kernel, rather than operating directly on the physical device to send or receive packets and reclaim buffers.

**Hypervisor:** the hypervisor needs only small extensions in the frontend for the device and the netmap backend, enabling the use of an additional set of PCI registers that support the pt allocator and port driver requests.

The current prototype has been developed on QEMU/KVM and Linux, using the `e1000` device driver to support the passthrough mode. Given the minimal amount of modifications in the guest and hypervisors, porting to other systems which already support netmap (such as `bhyve` and FreeBSD) will be trivial.

Note that a VM using a `ptnetmap` backend can dynamically switch between the standard mode described in Section 2.1, and the passthrough mode. As a consequence, guest applications using sockets will continue to work, of course with the performance constraints imposed by the guest OS.

## 4.1 Operation

We briefly summarize here the runtime operations involved with the use of `ptnetmap`.

The device initialization (executed when the guest device driver is loaded) is not critical for performance, and it only informs the guest and the host of the mutual availability of the `ptnetmap` mode.

When the guest NIC operates in standard mode, the passthrough datapath remains dormant and the guest device driver will talk to the regular frontend as shown in Figure 1. The backend will still use the netmap API, thus potentially allowing much higher speed than a standard `tap` backend.

When the guest switches the interface in netmap mode, the netmap port on the guest is created, attached to the memory region created on the host, and the notification paths are set to forward requests through the adapter module in the host.

These actions will be undone when the interface is switched back to standard mode.

The notifications between the guest and host port driver occur with the help of the kernel thread described in Section 3.1, represented by the thick dashed line in Figure 5. The thread is in charge of multiple tasks:

- it acts as an interrupt handler on the host, reacting to incoming packets or completed transmissions;
- it performs actions on the host on behalf of the guest VM, so that kernel exits can return quickly without waiting for completion (as an example, transmissions through a VALE switch requires a data copy and can be time consuming);
- it implements the VM exit and interrupt mechanism described in Section 3.1, which relies on the presence of a thread that actively polls the CSB when there is traffic.

In principle these task could be offloaded to other entities or removed, but the impact on performance is huge, and since the thread sits idle when there is no traffic, the impact on system resources is limited. We note that the last task, when successful essentially removes all guest-host notifications (and this normally happens under high load, when it is most needed).

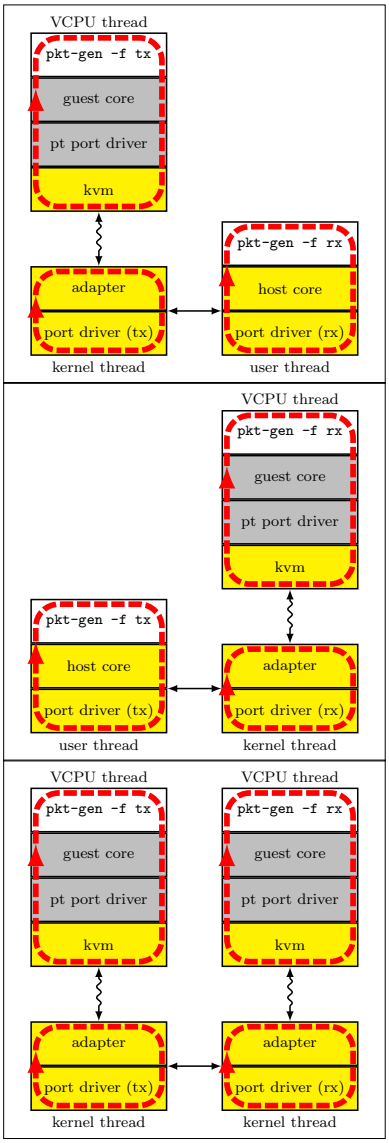


**Figure 5: Threads in the guest-host path (top), host-guest path (middle) and guest-guest path (bottom).**

# 5. PERFORMANCE EVALUATION

This Section presents experimental results on the performance of `ptnetmap`. Our test machine is equipped with an Intel Core i7-3770K CPU at 3.50GHz (4 core / 8 threads), 8GB RAM DDR3 at 1.33GHz and Intel 82599ES dual-port 10Gbps NIC. When needed, we use a second machine running netmap as a traffic source/sink over a 10 Gbit/s NIC.

The host OS is Linux 3.17.6, and we use QEMU-KVM as our hypervisor (git master c79805 - Aug 2014), extended to support `ptnetmap`. The guest VMs run Linux 3.12, and use an e1000 driver, modified to connect to the host using out virtual passthrough mode when the guest switches the interface in netmap mode.

## 5.1 Performance metrics

The three metrics of interest in `ptnetmap` are throughput, latency and CPU efficiency. We should start with a couple of mandatory warnings (at the risk of stating something obvious to researchers working on high speed systems).

First, there are huge tradeoffs between these metrics, and it is easy to achieve high values in one of them sacrificing one or the others. In particular, latency and throughput are often conflicting goals (e.g., larger batch sizes improve throughput at the expense of latency); and relaxing on CPU efficiency usually improves performance in the other dimensions (e.g., busy wait normally improves latency; or, oversizing parts of the system can help saturate other parts such as fixed-speed links, thus hiding inefficiencies).

Second, especially at high loads, the interactions between components may lead to inverse proportionality (such as, making one faster actually slows down the entire system, see Section 5.3.3 for an example), or even unstable behaviours. Hence, it is important to evaluate the system under different load patterns.

With this in mind, it does make sense to discuss the peak performance values and how to maximise them because they represent the boundary of the operating region, but individual metrics should not be considered in isolation without understanding the tradeoffs involved.

## 5.2 Comparison with other systems

It is customary to compare systems with similar one, but to the best of our knowledge, the only two system which are comparable in performance to `ptnetmap` are our previous work on QEMU [11], which `ptnetmap`improves significantly, and one recent proposal, netvm [4]. Other proposals for software switches exist, but they are all based on classical architectures described in Section 2.1, and barely reach 2 Mpps.

Netvm features are discussed in detail in Section 6. In terms of performance, it has been evaluated by the authors only on physical interfaces, with throughput capped to 14.88 Mpps and an RTT in the 60-70 $\mu$s range, probably also constrained by the hardware. On the other hand, the architectural choices of netvm (based on DPDK) severely limit its applicability compared to `ptnetmap`.

## 5.3 Throughput

Measuring communication costs (hence throughput) at very high speed is non trivial because of the highly non linear behaviour with varying input patterns, and also the steep memory bandwidth and latency changes that occur when the working set exceeds the cache size.

Apart from these effects, the cost of communication has a very weak dependency on the packet size (especially when using a zero-copy datapath), and the dominant components are per-packet and per batch costs. For this reason, we will show our results with variable batch sizes, and express our performance in packets per second[2] using minimum size

[2]we use pps as a metric following the common practice for reporting throughput. Its inverse, time-per-packet, would be a better metric to use as it is additive and eases predicting

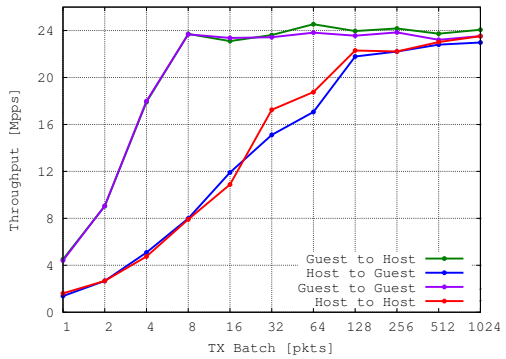


**Figure 6: Throughput across a VALE switch, with different placements of the source and sink, and 60 byte packets. In these experiments the queue size is 1024 slots.**

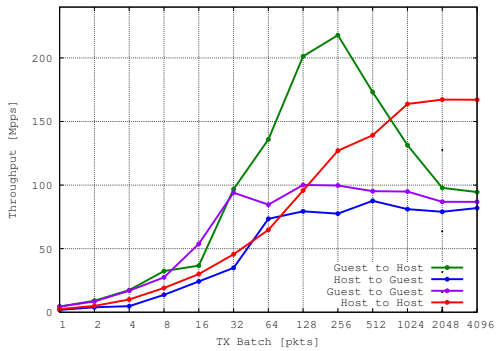


**Figure 7: Throughput across a netmap pipe, with different placements of source and sink, and 60 byte packets. In this case the queue size is set to 2048 slots to emphasize the behaviour described in Section 5.3.3.**

packets.

For throughput measurement, we use the netmap's `pkt-gen` program, which is a general purpose sender/receiver with configurable packet size, rate, batch size and number of threads. `pkt-gen` normally uses `poll()` to do I/O, hence it blocks when there are no slots in the queue to send or receive. We use two instances of `pkt-gen` (one sender and one receiver) with a single thread each, running on the host or the guest. The sender runs in unlimited rate mode, minimum size Ethernet frames (60-byte packets excluding CRC), and variable TX batch size. VALE ports and pipes are configured to use a single queue with 1024, 2048 or 4096 slots. NICs are configured with 4 queues and 2048 slots.

In these tests results are very stable and repeatable, with variations of less than 2% on physical ports and VALE ports, and less than 5% on netmap pipes (we will explain the larger variations in this case). For simplicity we only report the average values in our experiments, taken from runs lasting 10 s each.

**IMPORTANT NOTE:** by itself, `pkt-gen` does not touch the payload of the packets, so it is an excellent tool to measure the overhead of just the communication channel, without adding unnecessary operations. As an example, this feature lets us identify clearly the performance difference between netmap pipes and VALE ports, as well as the location of the bottleneck in the various combinations of sender and receiver.

Real world, data touching applications as data source and sink would be up to one order of magnitude slower than `ptnetmap`, and their use would transform the experiments in a benchmark of the source/sink themselves, and hide most details on the behaviour of `ptnetmap`.

### 5.3.1 *Throughput over VALE switch*

Figure 6 shows the throughput when source and sink communicate across a VALE switch. VALE transfers packets by executing a data copy to guarantee isolation between the ports, so in this experiment, the bottleneck is the thread (on the sender side) that executes the data copy.

Our reference curve is the throughput between two ports on the Host (red in the Figure); in this case the bottleneck thread is the client application on the sender side. As expected, the curve climbs up as the system call time is amortized over larger batches.

When the receiver is moved within a VM (the Host-Guest case, blue curve) there is an additional thread involved in the data transfer (see Figure 5, middle), but it operates on the receive side, so the bottleneck remains unchanged and throughput is very similar to the Host-Host case.

Moving the source on the Guest instead yields a higher throughput for comparable burst size, and also a modest throughput increase. This is not surprising: the sending thread (now on the Guest) only has to notify the port adapter in the host, while the expensive work – copying packets – is performed by the kernel thread on the host-transmit side (see Figure 5, top). This thread (which is the bottleneck) runs entirely in the kernel, and the time it takes to wakeup is much lower than that of the `poll()` system call used in the Host-Host case. This explains both the slightly higher throughput, and the steeper curve at small batch sizes.

In the Guest-Guest case, the interaction involves two additional kernel threads besides the sender and the receiver (see Figure 5, bottom). The bottleneck thread is the same as in the Guest-Host case, hence we have similar throughput.

### 5.3.2 *Throughput over netmap pipes*

Figure 7 shows the throughput of a sender-receiver pair communicating over netmap pipes. Since this time there is no data copy involved, the absolute throughput is much higher than on VALE ports, and in fact dominated by the speed of memory, the effectiveness of the cache, the overhead of system calls and interference of OS activities related to the scheduling of threads. Because of the latter, different runs

performance with multiple components.

have larger variations, around 5%, than in the VALE case. We have run this experiment with a larger queue (2048 slots) to emphasize some peculiar behaviours of the system.

The baseline throughput between two host ports exceeds 100 Mpps for large enough batches. This is because the per-packet cost is now extremely small (pipes only swap packet descriptors between the sender and receiver queue) and it takes much larger batches to make the system call overhead negligible with respect to the per-packet cost. Keep in mind that a data-touching workload will significantly reduce these values, as the pipe (and `ptnetmap`) are not the bottleneck anymore.

Moving the sender on a VM (Guest-Host case), the behaviour is similar to that on VALE ports: the maximum throughput increases much faster with the burst size, and to higher values (green curve in Figure 7). The peak throughput in this case is much higher, because the per-batch cost is the limiting factor for throughput, and splitting the work between the two threads greatly reduces it. The large drop of performance with even higher batch sizes will be explained in detail in the next section.

The remaining two cases, in which the receiver is on the guest, give no surprises. apart from a slightly lower peak performance than with the host-based receivers due to more delay in releasing buffers in the receive thread (the receiver in the guest has to notify a thread in the host adapter, which in turn wakes up the sender).

### 5.3.3 *Short queue regime*

An interesting phenomenon (in the Guest-Host case) is the performance drop as the batch size grows. It is worth discussing it as it appears many times in high performance systems. Remember how in this case the sender is extremely fast, as the total work is divided between the guest and the host-tx thread. When the batch size is small, the sender is the bottleneck and throughput grows with the batch size. Above some threshold, the bottleneck in the system switches from the sender to the receiver, and the sender will periodically block because the transmit queue fills up. The blocking in turn causes additional load also on the receiver, which not only has to read input packets but also wake up the sending thread, and this explains the drop as the batch size grows.

Figure 8 shows how the total queue size impacts the phenomenon. As expected the peak throughput increases with the maximum queue size because both sender and receiver benefit from it, and the dip is less severe because larger queues allow a higher receive throughput.

Note that a fast receiver is less subject to blocking than a fast sender: the receiver normally returns whatever amount of packets is queued, and at low speeds it will simply become less efficient (system call costs are amortized on shorter batches). Blocking will only occur when the sender's rate is less than one packet per receive system call, a rate so low that does not cause visible performance problems.

### 5.3.4 *Throughput over physical ports*

Our last throughput test involves physical ports. Here the results (see Figure 9), are unsurprising, with the guest able

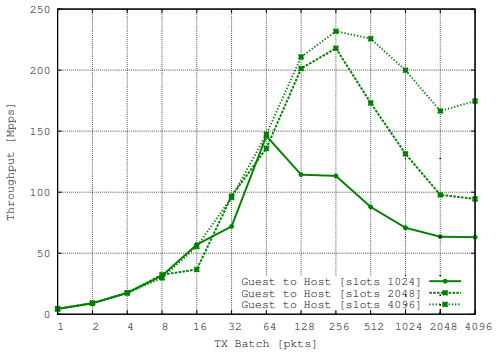


**Figure 8: Throughput on pipes, Guest to Host, for variable queue size.**

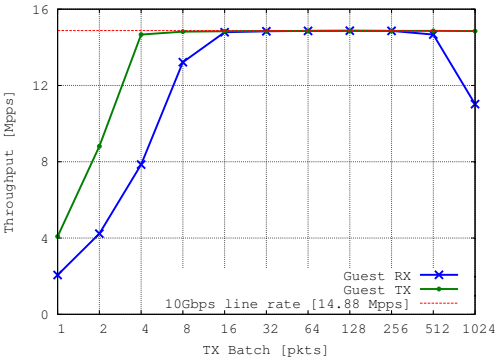


**Figure 9: Throughput on physical port (Intel 10Gbps NIC) from the guest in `ptnetmap` mode.**

to reach full line rate even with modest batch sizes. This result matches the performance of other passthrough solutions [4] and of netmap on bare metal.

## 5.4 Cascaded VMs

To evaluate how `ptnetmap` behaves with multiple VMs on the same host, we ran some experiment by chaining VMs as shown in Figure 10. Each VM runs a simple program (called `bridge`, also part of the netmap suite) that passes packets from one port to the other, *doing an actual memory copy*. We chose not to use zero-copy in the application running in the VM to make sure that the application itself creates a sufficient load on the system, otherwise the results might have been misleading, with all CPU resources potentially consumed by the kernel threads associated to each `ptnetmap` port. At the same time, we made the application sufficiently simple so that it would not, by itself, be too slow to stress the system adequately.

The experiment is run with a transmit batch size of 32 packets, queues with 2048 slots, and $B$ (varying between 1 and 4) `bridge` instances. The Host has 4 physical cores (8 with hyperthreading). The baseline for our experiment is when all `bridge` programs, together with the source and sink, run on the Host. In this case the total number of active threads

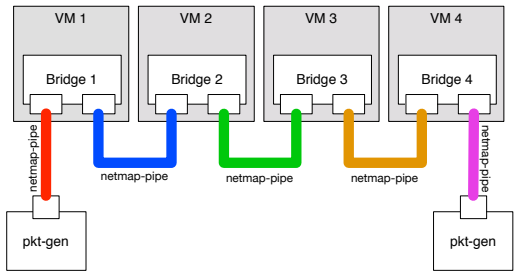


**Figure 10: Topology for the experiments with cascaded VMs. The "bridge" application on each VM copies packets from one port to another. All VMs, and the source and sink run on the same host and are connected through netmap pipes.**

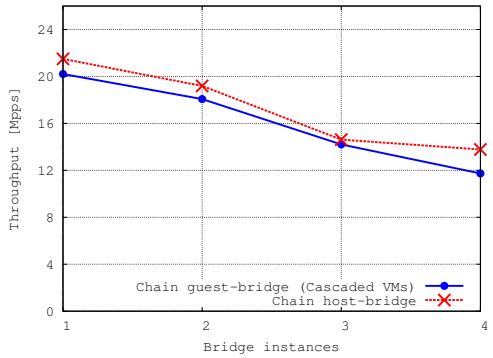


**Figure 11: Throughput on cascaded VMs.**

is $2+B$, and the throughput is shown by the top (red) curve in Figure 11.

The absolute performance, around 20 Mpps, is in line with what we see with another packet-copy solution, namely VALE ports. The slight degradation is expected and due to the delay that is incurred in releasing packets across the pipes. Fine tuning the batch sizes and the application would surely yield much better and stable throughput, but that was not the goal of experiment: instead, our goal was to see how, without any special tuning, the system degrades when applications are moved to VMs.

The second set of measurements runs the `bridge` applications in different VMs. Due to the additional kernel thread[3] that `ptnetmap` creates for each passthrough port, now the total number of threads is $2+3B$, which largely exceeds the number of available cores. Fortunately (actually, by design!) the kernel threads are activated on demand, so they do not consume too many additional CPU cycles. As a result, the performance degradation when running the cascade on the VMs is very modest, in the order of 15%. This is a witness of the goodness of our architecture, and also of its efficient use of resources.

## 5.5 Latency

[3]we ignore the NAPI threads that each linux kernel runs.

Latency is another parameter that must be considered with extreme care. It is extremely easy to reduce it significantly by replacing all notification mechanisms with busy wait loop; but in this way, the efficiency at low load plummets and the approach is simply not acceptable in large scale deployments.

Since `ptnetmap` uses several thread handoffs to dispatch notifications, it is expected that latency will suffer, especially at the beginning of a communication when the kernel thread are idle and the mitigation technique of Section 3.1 is not effective.

For the latency measurement we use `pkt-gen` program in ping mode: the sender transmits the packet and blocks on a `poll()` waiting for a response; the receiver waits for a packet (also using a `poll()`, and immediately bounces it back. Each packets carries a timestamp generated by the sender, which can easily measure the Round Trip Time upon receiving the response.

The use of blocking system calls is the normal behaviour for CPU-conscious applications. However, the absolute numbers we derive would not tell us much on how far we are from an ideal (in terms of latency) situation, where threads use non blocking calls and busy wait to exchange messages. For this reason we ran a second set of experiments where `pkt-gen` is modified to use non-blocking `ioctl()` to send and receive, and the kernel threads are forced in an active state, thus avoiding all interrupts, VM exits and process wakeups.

The results of the two sets of experiments are shown in Table 1, in the **Blocking** and **Busy-wait** columns, respectively. The results are interesting and encouraging, but not surprising.

In the busy-wait case, the latency is less than 400 ns per hop (remember messages travel back and forth), marginally higher in the VALE cases which has an extra copy and additional locking in the path. This is the latency that `ptnetmap` clients can actually experience when the system is kept streaming (e.g. when a sufficient amount of traffic is flowing through the port.

In the efficient, **Blocking** case, which represents the worst case scenario, we incur an average of about 25 $\mu$s in the Guest-Guest case, a time that is dominated by the cost of two VM exits and two VM interrupts. The value is in the same range of a `ping` between two physical machines connected by 10 Gbit/s network cards, so we are not particularly sure that is should classified as an unacceptably large latency. In any case it is worth noting that `ptnetmap` does give the user a choice between a latency (between VMs) comparable to that between physical hosts with efficient CPU usage, and a much lower one while streaming or while desired, if the user has CPU cycles to spare.

## 5.6 CPU efficiency

We would like to conclude with a few remarks on CPU efficiency.

Systems such as DPDK [5] and derived ones completely give

| | | Round Trip Time [usec] | | | |
|---|---|---|---|---|---|
| | | **Blocking** | | **Busy-Wait** | |
| Configuration | Backend | Min | Avg | Min | Avg |
| Host - Host | pipe | 2.6 | 4.6 | 0.5 | 0.8 |
| Host - Host | vale | 3.3 | 5.2 | 0.8 | 1.2 |
| Host - Guest | pipe | 7.3 | 14.2 | 1.1 | 1.6 |
| Host - Guest | vale | 8.3 | 14.2 | 1.2 | 1.6 |
| Guest - Host | pipe | 9.1 | 14.5 | 1.5 | 1.6 |
| Guest - Host | vale | 9.6 | 14.6 | 1.5 | 1.7 |
| Guest - Guest | pipe | 21.4 | 24.5 | 1.6 | 2.1 |
| Guest - Guest | vale | 21.4 | 24.6 | 1.6 | 2.1 |

**Table 1: Round Trip Time between two clients in different configurations.**

up on CPU efficiency at low loads. This choice allows significant simplifications in the implementation (see Section 3.2), with advantages on the latency (and possibly throughput) side. While it is in theory possible for those systems to build a proper notification support mechanism, it is not obvious that the result may be better than what exists in various OS kernels (such as Linux and FreeBSD) or hypervisors such as KVM, which have benefited from a huge amount of optimization work over the years. As a consequence, it is prudent to exclude systems based on busy wait from the reasoning on CPU efficiency.

Regarding systems such as `ptnetmap`, which use proper synchronization, a comparison is extremely difficult for two reasons. First, reliable measurements of the CPU utilization is extremely elusive, because depending on the architecture, functions can be shifted between components (kernels, hypervisors, user threads etc.) in a way that makes it difficult to account for the total amount of cycles used for a communication.

Furthermore, depending on the input load, systems such as `ptnetmap`, virtio and others, that use interrupt and VM-exit moderation techniques, exhibit an all-or-nothing behaviour, because the packet processing time is comparable or often much smaller than the time it takes to start a sleeping thread. Hence, when the event rate is low enough to let threads go to sleep, the average CPU usage will be extremely low. Conversely, when the inter-event time approaches the wakeup time, CPU utilization quickly ramps up to 100%, and remaining stable for a wide range of input loads.

Under these circumstances, the two parameters of interest are the maximum throughput of the system, which is an indirect measurement of the efficiency of use of resources, and the event rate that marks the switch between the two regimes of operation. We have measured the former in Section 5.3, whereas for the latter we postpone the investigation to future work, both because of lack of space, and because the threshold itself (which we can estimate in one event every 10-20 $\mu$s, according to our latency measurements) is most of the time inherited by the underlying operating system.

# 6. RELATED WORK

Achieving very high network performance (1 Mpps and more) in Virtual Machines relies on three groups of techniques: performance enhancement techniques also used on bare metal; hardware support in CPUs and peripherals; and software solutions adopted in hypervisors to overcome hardware limitations.

## 6.1 OS and network stack bypass

The first group typically addresses inefficiencies in the OS's network when processing traffic at the high packet rates that firewalls, software routers, DNS servers etc. may have to deal with. Proposals in this group try to provide an alternate, more efficient path for packets between applications and the physical link. One approach, "OS-bypass", relies on an almost complete bypass of the operating system for network communication. The hardware is directly controlled by the application program using custom libraries [5, 2] which provide efficient transmit and receive functions. High throughput is generally achieved by direct mapping of packet buffers and batched I/O. At the speeds of interest, even the misses in the virtual to physical address translation buffers (TLB) are prohibitively expensive, hence systems often use huge memory pages (e.g. 1 GByte) to reduce the number of TLB entries in use. A feature that is frequently missing in OS-bypass solutions is the ability to receive interrupts. This is for two reasons: first and foremost, when the device is controlled by the application, the OS has no good way access the device in response to interrupts. Secondly, as discussed in Section 3.2, busy wait permits lower latency in responding to events. For this reason, OS-bypass techniques normally rely on additional processes which are continuously active monitoring the hardware.

netmap [9] belongs to the "network-bypass" family of solutions, and differs from OS-bypass as it relies heavily on the operating system for dealing with the hardware. In netmap, slightly modified device drivers provide a high speed data path to the hardware (or to the VALE [10] virtual switch, or to high speed netmap pipes), but resource management, synchronization, protection are still implemented by the operating system. As a result of the tight integration with the OS, netmap can use blocking I/O functions, resulting in a much more efficient use of resources. A similar approach, targeted to socket-based applications, is used by the OpenOnload [13] framework.

In terms of absolute performance, most OS-bypass and network bypass solutions (DPDK, DNA, netmap) can drive the NIC *to its limits* even with just one core. Ideally, this means 14.88 Mpps on a 10 Gbit/s interface with 64-byte frames, but several 10 Gbit/s NICs have hardware limitations and are unable to reach line rate with small frames. At 40 Gbit/s, we have measured 45 Mpps with netmap [1].

## 6.2 Hardware support

The hardware support for fast networking in VMs comes from three places. First, generic CPU support for virtualization helps running the VM at native speed at least when not accessing (pseudo)-hardware resources. VM exits when accessing I/O devices are expensive, but their cost can be mitigated with the solutions indicated in Section 3.1, and with CPU support that permits injecting interrupts directly in the VM without requiring an exit. Second, IOMMU permits protected direct access to hardware resources from VMs, remapping memory access generated from I/O devices. This permits, for instance, a safe implementation of hard-

ware passthrough. Third, NICs (and peripherals, in general) can ease their use in VMs through a mechanism (called SR-IOV) which allows a single physical device to export multiple independent instances (Virtual Functions, VF) of the device itself, each associated with hardware resources. This is a key feature to implement hardware passthrough in a scalable way, as each VF can be dedicated to a VM with zero overhead. The fast frameworks of Section 6.1 can be run on top of a VF, but VFs share the same I/O bus and output link, so the actual communication capacity even between VMs is severely limited.

## 6.3 Virtualization solutions

Fast networking support for VMs generally relies on the techniques shown in Section 3.1 to mitigate some of the drawbacks of device emulation. Solutions such as virtio [12] are normally used in most virtualization systems, though they are often bottlenecked by the upper (guest network stack) and lower (host switch) connections.

Research on software switches has been mostly focused on the flexibility of the control plane, such as Open vSwitch [7], and performance on large TCP flows [8]. The VALE software switch [10] has shown that extremely high speed packet switching is possible without exceeding complexity, and its integration with hypervisors can reach extremely high speeds, as shown in [11]. `ptnetmap` clearly builds extensively on this previous work of ours.

A very related work is netvm [4], which implements a zero-copy userspace software switch on top of DPDK. VMs attach to netvm using a virtual passthrough mechanism, using DPDK as the communication API. The authors show that netvm is capable of supporting 14.88 Mpps (line rate) on 10 Gbit interfaces and between VMs, and a round trip time for VMs in the 40-70 $\mu$s range, not much higher than that of bare metal. Because of the use of DPDK as the underlying transport, netvm lacks interrupts, and depends critically on the use of shared memory and huge pages, hence it cannot be used directly between untrusted VMs, and instead relies on a much slower channel through the ordinary network stack for those cases.

# 7. CONCLUSIONS AND FUTURE WORK

We have presented a solution for efficient network access from Virtual Machines through the use of a Virtual Passthrough mechanism.

Unlike hardware passthrough, we do not depend on the presence of specific hardware. Our architecture is also fundamentally different from other proposals in two aspects: we make a high speed data path available also to untrusted VMs, and we do not require busy-wait loops for synchronization. Implementing these two features (while preserving high speed operation) is way more challenging than simpler shared-memory, active polling systems proposed in the past, but it is the only way to develop truly scalable and efficient solutions for network function virtualization.

Our performance numbers are not penalized by our architecture: we still manage to reach line rate (14.88 Mpps) on 10 Gbit/s NICs, and go even much further in local communication, reaching over 20 Mpps between untrusted VMs and well over 70 Mpps between trusted VMs.

Our code has been developed as an extension of the netmap framework, and is publicly available. Currently we support Linux guests and KVM hosts, but given the relatively small size of the changes we expect a FreeBSD guest support to be available soon. Work is in progress on a bhyve host implementation.

# 8. REFERENCES


[1] Chelsio. T5 netmap performance on freebsd. http://www.chelsio.com/wp-content/uploads/resources/FreeBSD-T5-Netmap.pdf, 2014.

[2] L. Deri. PFRING DNA page. http://www.ntop.org/products/pf_ring/dna/.

[3] M. B.-Y. et al. Utilizing iommus for virtualization in linux and xen.

[4] J. Hwang, 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)*, 2014.

[5] Intel. Intel data plane development kit. *http://edc.intel.com/Link.aspx?id=5378*, 2012.

[6] P. B. Menage. Adding generic process containers to the linux kernel. *2007 Linux Symposium, Ottawa*, pages 45–58.

[7] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenkerz. Extending networking into the virtualization layer. In *ACM SIGCOMM HotNets, October 2009*.

[8] K. K. Ram, A. L. Cox, M. Chadha, S. Rixner, T. W. Barr, R. Smith, and S. Rixner. Hyper-switch: A scalable software virtual switching architecture. *2013 USENIX Annual Technical Conference*, pages 13–24, 2013.

[9] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC'12*, Boston, MA. USENIX Association, 2012.

[10] 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*, CoNEXT '12, pages 61–72, New York, NY, USA, 2012. ACM.

[11] 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.

[12] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008.

[13] Solarflare. Openonload. *http://www.openonload.org/*, 2008.

[14] E. Zhai, G. D. Cummings, and Y. Dong. Live migration with pass-through device for linux vm. In *2008 Linux Symposium, Ottawa, CA*, pages 261–270, 2008.