Speeding up packet I/O in virtual machines

Luigi Rizzo, Giuseppe Lettieri, Vincenzo Maffione,
Università di Pisa, Italy

rizzo@iet.unipi.it, http://info.iet.unipi.it/∼luigi/vale/

Abstract

Most of the work on VM network performance has focused so far on bulk TCP traffic, which covers classical applications of virtualization. Following popular wisdom (and perhaps bad initial experience with device emulation), completely new "paravirtualized devices" (Xenfront, virtio, vmxnet) have been designed and implemented to improve network throughput.

We expect virtualization to become widely used also for different workloads: packet switching devices and middleboxes, Software Defined Networks, etc.. These applications involve very high packet rates that are problematic not only for the hypervisor (which emulates network interfaces) but also for the host itself (which switches packets between guests and physical NICs).

In this paper we provide three main results. First, we demonstrate how rates of millions of packets per second can be achieved even within VMs, with limited but targeted modifications on device drivers, hypervisors and the host's virtual switch. Secondly we show that emulation of conventional NICs (e.g., Intel e1000) is perfectly capable of achieving such packet rates, without requiring completely different device models. Finally, we provide sets of modifications for various use cases (acting only on the guest, or only on the host, or on both) and depending on the configuration and use case we can improve the network throughput of a VM by 20 times or more.

These results are important because they enable a new set of applications within virtual machines. In particular, our prototype implementation achieves guest-to-guest speeds of over 1 Mpps with short frames (and 6 Gbit/s with 1500-byte frames) using a conventional e1000 device, and socket-based sender/receivers. This is the same speed that we experience running the OS on bare metal, and a ten-fold speedup over the performance of the original e1000 emulation, and slightly better than virtio. Furthermore, we reach over 5 Mpps when guests use the netmap API.

Our work requires only small changes to device drivers (about 100 lines, both for FreeBSD and Linux version of e1000), similarly small modifications to the hypervisor (we have a qemu prototype available) and the use of the VALE switch as a network backend. Relevant changes are being incorporated in FreeBSD and distributed as patches for QEMU/Linux.

1. INTRODUCTION

Virtualization is a technology in heavy demand to implement server consolidation, improve service availability, and make efficient use of the many cores present in today's CPUs. Of course, users want to exploit the features offered by this new platform without losing too much (or possibly, anything) of the performance achievable on traditional, dedicated hardware (*bare metal*).

Over time, ingenious software solutions [5], and later hardware support [13, 3], have mostly filled the gap for CPU performance. Likewise, performance for storage peripherals and bulk network traffic is now comparable between VMs and bare metal, especially when I/O can be coerced to use large blocks (e.g. through TSO/RSC) and limited transaction rates (e.g., say less than 50 K trans/s).

However a class of applications, made relevant by the rise of Software Defined Networking (SDN), still struggles under virtualization. Software routers, switches, firewalls and other middleboxes, need to deal with very high packet rates (millions per second) that are not amenable to reduction through the usual Network Interface Card (NIC) offloading techniques. The "direct mapping" of portions of virtualization-aware NICs to individual VMs can provide some relief, but it has scalability and flexibility constraints.

We then decided to explore solutions to let VMs deal with millions of packets per second without requiring special hardware, or imposing massive changes to OSes or hypervisors. In this paper we discuss the general problem of network performance in virtual machines, identifying the main causes of performance loss compared to bare metal, and design and experiment with a comprehensive set of mechanisms to fill the performance gap.

Our contribution: in detail, we i) emulate interrupt moderation, ii) implement "Send Combining", a driver-based form of batching and interrupt moderation; iii) introduce an extremely simple but very effective paravirtualized extension for the `e1000` devices (or other NICs), providing the same performance of `virtio` and alikes with almost no extra complexity; iv) adapt the hypervisor to our high speed VALE [20] backend, and v) characterize the behaviour of *device polling* under virtualization.

Some of the mechanisms we propose help immensely, especially within packet processing machines (software routers, IDS, monitors ...). Especially, the fact that we provide solutions that apply only to the guest, only to the host, or to both, makes them applicable also in presence of constraints (e.g., legacy guest software that cannot be modified; or proprietary VMMs).

In our experiments with QEMU-KVM and `e1000` we reached a VM-to-VM rate of almost 5 Mpps with short packets, and 25 Gbit/s with 1500-byte frames, and even higher speeds between a VM and the host. These large speed improvements have been achieved with a very small amount of code, and our approach can be easily applied to other OSes and virtualization platforms. We are pushing the relevant changes to QEMU, FreeBSD and Linux.

In the rest of this paper, Section 2 introduces the necessary background and terminology on virtualization and discusses related work. Section 3 describes in detail the four components of our proposal, whereas Section 4 presents experimental results, and also discusses the limitations of our work.

2. BACKGROUND AND RELATED WORK

In our (rather standard) virtualization model (Figure 1), Virtual Machines (VMs) run on a Host which manages hardware resources with the help of a component on the Host called *hypervisor* or Virtual Machine Monitor (VMM, for brevity). Each VM has a number of Virtual CPUs (VCPUs, typically implemented as threads in the host), and also runs additional IO threads to emulate and access peripherals. The VMM (typically implemented partly in the kernel and partly in user space) controls the execution of the VCPUs, and communicates with the I/O threads.

The way virtual CPUs are emulated depends on the features of the emulated CPU and of the host. The x86 architecture does not lend itself to the *trap and emulate* implementation of Virtualization [1], so historical VMMs (Vmware, QEMU) relied for the most part on binary translation for "safe" instructions, and calls to emulation code for others. A recent paper [5], long but very instructive, shows how the x86 architecture was virtualized without CPU support. The evolution of these techniques is documented in [1]. A slowdown of 2..10 times can be expected for typical code sequences, slightly lower if kernel support is available to intercept

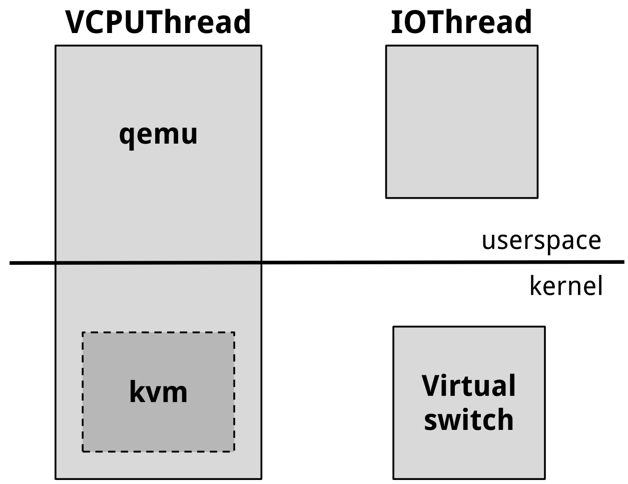

Figure 1: In our virtualized execution environment a virtual machine uses one VCPU thread per CPU, and one or more IO threads to support asynchronous operation. The hypervisor (VMM) has one component that runs in userspace (QEMU) and one kernel module (kvm). The virtual switch also runs within the kernel.

memory accesses to invalid locations.

Modern CPUs provide hardware support for virtualization (Intel VTX, AMD V) [13, 3], so that most of the code for the guest OS is run directly on the host CPU operating in "VM" mode. In practice, the kernel side of a VMM enters VM mode through a system call (typically an `ioctl(.. VMSTART ..)`), which starts executing the guest code within the VCPU thread, and returns to host mode as described below.

2.1 Device emulation

Emulation of I/O devices [25] generally interprets accesses to I/O registers and replicates the behaviour of the corresponding hardware. The VMM component reproducing the emulated device is called *frontend*. Data from/to the frontend are in turn passed to a component called *backend* which communicates with a physical device of the same type: a network interface or switch port, a disk device, USB port, etc.

Access to peripherals from the guest OS, in the form of IO or MMIO instructions, causes a context switch ("VM exit") that returns the CPU to "host" mode. VM exits often occur also when delivering interrupts to a VM. On modern hardware, the cost of a VM exit/VM enter pair and IO emulation is 3..10 μs, compared to the 100-200 ns for IO instructions on bare metal.

The detour into host mode is used by the VCPU thread to interact with the frontend to emulate the actions that the real peripheral would perform on that

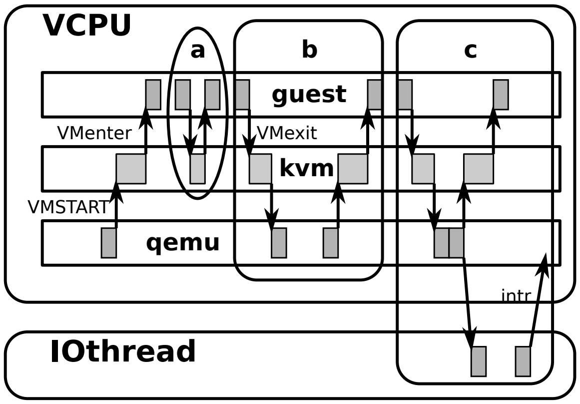

Figure 2: Different ways to emulate IO register accesses. They can be taken care of directly in the kernel component of the hypervisor (case "a"), or in the host component (case "b"), or handed off to the IO thread (case "c") in case of long activities.

specific register access. In some cases these are trivial (such as reading or writing a memory word); other times, a register access may involve copying one or more blocks of data, e.g. network frames, between the emulated device and the actual peripheral on the host.

This emulation can be done in different ways, as illustrated in Figure 2. For frequently accessed peripherals that are a standard part of an architecture, such as interrupt control registers, the kernel side of the VMM can do the emulation directly; this is represented by case "a" in Figure 2).

In other cases, the task can be performed in the user side of the VMM (see case "b"), which gives greater flexibility but also adds cost to the operation. As an example in one of our systems (Intel i5-2450M @ 2.5 GHz, memory @ 1333 MHz) a register access requires 1.2μs when executed by the kernel component of the hypervisor (kvm), and 3.7μs when dispatched to the userspace component.

In the worst case, the VCPU thread may actually even be descheduled, or block waiting for I/O. This can be problematic as I/O accesses often occur under some kind of locking, so changing the execution time from 100 ns to potentially long times may completely disrupt the work of other VCPU threads. It follows that potentially long operations are better served by running them within an independent thread (IOthread), to which the VCPU hands off the request (case "c"). IOthreads are also used to handle events (e.g., I/O completions, timeouts) generated from the host OS.

This description illustrates the fundamental performance issues and tradeoffs in using emulated peripherals within a VM. While regular code runs at full speed, IO instructions consume thousands of clock cycles, and can block other VCPUs for very long times. Also, the latency in performing the actual I/O operation is badly affected by the thread handoffs that are necessary to return quickly from a VM exit.

Hence, a significant challenge in VM environments is the reduction or elimination of VM exits, which are especially frequent when dealing with high packet rate network traffic. Our work is actually providing a number of solutions in this space.

2.2 Reducing VM exits

It is not uncommon that instructions leading to VM exits are close to each other – as an example, within device drivers where multiple registers are accessed at once. A recent proposal [2] shows how short sequences of code involving multiple I/O instructions can be profitably run in interpreted mode to save some VM exits. Implementing this technique is non trivial, but it can be done completely within the VMM without any change in the guest OS.

VM exits are not needed for hardware that is directly accessible to the guest VM: we can give it full control of a physical device (or part of it, if the device supports "virtual functions") as long as we can make sure that the VM cannot fiddle with other reserved parts of the hardware. IOMMU [4] and related technologies (generally referred to as *PCI passthrough*) comes to our help here, as they provide the necessary protections as well as translation between the guest's physical address space and the host physical address space. Again, this approach can be almost completely transparent for the guest OS, and relies only on support in the hardware and the VMM.

2.2.1 *Paravirtualized devices*

Another popular approach to reduce VM exits is to design new device models more amenable to emulation. This approach, called "paravirtualization", has produced several NIC models (vmxnet [26], virtio [21], xenfront [6]), in turn requiring custom device drivers in the guest OS. Synchronization between the guest and the VMM uses a shared memory block, as described in Section 3.5, accessed in a way that limits the number of interrupts and VM exits. One contribution of this paper is to show that paravirtualization can be introduced with minimal extensions into existing NICs.

2.2.2 *Interrupt handling*

Interrupts to the guest frequently cause exits, either directly or indirectly (because they trigger accesses to registers of interrupt controllers and devices). Mechanisms to reduce interrupt rates, as those shown in Section 3.1 and following, help significantly.

Likewise, it is possible for interrupts to be delivered directly to the guest without causing VM exits. As an example, ELI [8] removes interrupt-related VM exits on direct-access peripherals by swapping the role of host and guest: the system is programmed so that all interrupts are sent to the guest, which reflects back those meant for the host.

2.3 High speed networking

Handling 10 Gbit/s or faster interfaces is challenging even on bare metal. Packet rates can be reduced using large frames, or NIC support for segmentation and reassembly (named TSO/GSO and RSC/GRO, respectively). These solutions do not help for packet processing devices (software routers, switches, firewalls), which are not sourcing or terminating connections, hence must cope with true line speed in terms of packet rates.

Only recently we have seen software solutions that can achieve full line rate at 10 Gbit/s on bare metal [18, 7, 12]. In the VM world, apart from the trivial case of directly mapped peripherals, already discussed [8], the problem has not seen many contributions in the literature so far. Among the most relevant:
Measurements presented in [23] show that packet capture performance in VMs is significantly slower than on native hardware, but the study does not include the recent techniques mentioned above, nor proposes solutions. Interrupt coalescing and Virtual Receive Side Scaling have been studied in [10] within Xen; the system used in that paper is limited to about 100 Kpps per core, and the solutions proposed impose a heavy latency/throughput tradeoff and burn massive amounts of resources to scale performance. ELVIS [9] addresses the reduction of VM exits in host-guest notifications: a core on the host monitors notifications posted by the guest(s) using shared memory, whereas inter-processor interrupts are used in the other direction, delivered directly to the guest as in the ELI case.

2.4 State of the art in VM networking

On bare metal and suitably fast NICs, clients using a socket API are generally able to reach 1 Mpps per core, peaking at 2..4 Mpps per system due to driver and OS limitations. Recent OS-bypass techniques [18, 7, 14, 24] can reach much higher rates, and are generally I/O bound, easily hitting line rate (up to 14.88 Mpps on a 10 Gbit/s interfaces) unless limited by NIC or PCIe bus constraints.

Within a VM, at least when emulating regular NICs (e.g., the popular Intel's `e1000`), common VMMs reach rates of about 100 Kpps on the transmit side, and marginally higher rates on the receive side (see Figure 4, 5 and 10). Paravirtualized devices (`virtio` etc.) can help significantly, but support for them is neither as good nor as widespread as it is for popular hardware NICs. This is especially true in the presence of old/legacy systems that cannot be decommissioned: they are often run within a VM due to lack of suitable hardware. So there is still a strong motivation for efficient NIC emulation, which motivates some of the solutions that we propose, requiring none or minimal modifications to the guest OS — much less than adding an entirely new device driver.

Note that our work goes beyond a simple replacement of paravirtualized devices. We address packet rates beyond those supported by the usual virtual switches, so we needed to identify and fix performance bottlenecks in multiple places of the architecture, from the guest operating system to the virtual switch itself.

3. OUR WORK

Taking QEMU-KVM as a prototype platform, we have investigated how to improve the network performance of guests, using both regular (e.g., Intel `e1000`) and paravirtualized network devices. Given the huge number of dimensions in the problem space, we initially focus the presentation on the cases in dire need of improvement: guest to guest communication, high packet rates, non-paravirtualized devices. Other configurations will be discussed and evaluated in Section 4.

For the host OS we used Linux 3.8, with KVM. For the guest OS we used both Linux 3.8 and FreeBSD HEAD. The differences in device driver and network stack architectures helped pointing out different phenomena and also the peculiarities in running in a VMs rather than on bare metal.

We should also keep in mind the constraints that may limit the solution space. Sometimes the guest OS cannot be modified or extended, hence we can only operate on the host side; in other cases the hypervisor is instead proprietary, hence we should look for guest-only solutions. And of course, in absence of such constraints, we should look for performance improvement techniques across the board. We believe that the set of solutions covered in this paper covers nicely the various situations, and can be easily applied to other systems/platforms than the ones we used for our experiments.

3.1 Interrupt moderation on emulated NICs

Interrupt moderation [15] is a widely used technique to reduce interrupt load, and it is normally available on modern NICs and supported by OSes. It struck our attention that the feature is rarely implemented by VMMs: QEMU and VMware Player do not emulate moderation registers; VirtualBox supports them only in very recent versions.

This was surprising since interrupts storms from NICs at high packet rates are a significant source of uncontrolled system load, leading to receiver livelock and severe reductions of the transmit rate.

One reason for the omission may be that modern

Operating Systems employ various interrupt avoidance techniques in software (e.g., the NAPI architecture in Linux [22], device polling in FreeBSD [17]; lazy buffer recovery on the transmit side, see Sec. 3.3). As a consequence, interrupt moderation is mostly useful for (older) OSes that do not adopt these techniques. Nevertheless, support of legacy legacy OS/applications for which real hardware is not available anymore is actually one of the interesting applications of virtualization. Also, having interrupt moderation in the device enables the use of Send Combining (discussed next in Sec. 3.2) on the transmit side which has significant benefits in a VM.

Our first modification was to implement the interrupt moderation registers in QEMU. The (small) change of the code affects only the hypervisor, and can work even if guest OS or the original NIC do not support moderation, as we can enforce the feature on the emulated device.

Hence this makes a good solution for old or not modifiable guests.

On the transmit side, moderation more than doubles the transmit rate for UDP, especially with a single VCPU (on Linux, we go from 29 to 82 Kpps with 64-byte frames, 28 to 70 Kpps with 1500-byte frames). The reason is that on the VM, in absence of moderation, a transmit request (triggered by a write to one of the NIC registers, in turn causing a VM exit) is instantly followed by an interrupt, that causes a burst of VM exits in the interrupt service routine. As a consequence, each interrupt serves exactly one packet. With moderation, one exit per packet is still used in the transmit routine, but the cost of the interrupt routine is now amortized over a larger set of packets.

The effect of moderation is more limited with 2 VCPUs, because in this case the interrupt service routine can run in parallel with the transmit requests, and this gives some chances to serve more than one packet per interrupt. Depending on the OS, we may still see some improvement, but more limited (e.g., Linux goes from 40 to 82 Kpps).

On the receive side, the effect of moderation depends on the arrival pattern of traffic. In many cases, traffic received by the VM already comes in batches because this is how it is passed on by the incoming (physical) interface, virtual switch and/or the backend.

3.2 Send Combining

Device drivers typically notify the NIC immediately when new packets are ready for transmission; this is relatively inexpensive on bare metal, but causes a VM exit in a VM.

To reduce the number of such exits we used an idea similar to what in [25, Sec.3.3] is called “Send Combining” (SC): since the driver knows about pending TX interrupts, it can defer transmission requests until the arrival of the interrupt. At that point, all pending packets are flushed (with a single write on one of the NIC’s registers), so there is only on VM exit per batch.

SC is a trivial (about 25 lines of code in our case), *guest only* modification of the transmit path. However it has no effect if TX interrupts are instantaneous, so it is mostly useful when coupled with moderation, or with device polling, see Section 3.3.2, because it reduces the system load from one VM exit per packet to one VM exit per batch.

On the negative side, send combining may delay transmissions by up to the moderation/polling delay (20..100 μs for moderation, 250..1000 μs for polling), similarly to what happens on the receive path with these techniques. The inconvenience can be reduced by tuning SC (and interrupt moderation) to kick in only above a certain packet rate, and by choosing appropriate tradeoffs between delay and throughput.

The combination SC plus moderation gives a fantastic boost on the transmit path: on Linux, we went from 40 to 354 Kpps with 2 VCPUs, and from 30 to 221 Kpps with 1 VCPU. The reason for these high speedups is the massive reduction of VM exits (by far the most expensive operation), which are now well below one per packet. We are now approaching rates where the virtual switch and the data path within the VMM may become a performance bottleneck.

As a final note in this section, we should mention that unlike our version, the original Send Combining in [25] was implemented entirely in the VMM, because it ran I/O instructions in “binary translation” mode. When using CPU-based virtualization support we do not have this luxury, and we cannot intercept I/O requests before they cause a VM exit.

3.3 Reducing interrupts

Interrupt moderation in the emulated device is not the only way to reduce interrupts. There are other techniques that can be implemented in the guest OS, and lead to similar or better performance improvements. The following ones are not new contributions of this work, but it is important to understand how they relate to virtualization. Especially, the benefits of polling within VMs has not been documented so far.

3.3.1 Lazy tx completion

As long as there are resources available, it is not necessary to interrupt on each transmitted packet to recover the buffer. *Lazy tx completion* recovers completed transmission in batches (we can think of it as moderation based on packet counts rather than time), or opportunistically during subsequent transmissions. This technique is implemented in some device drivers, and also in the netmap [18] framework. Note that it can completely remove transmit interrupts, so it is not compatible with send-combining, which instead relies on interrupts to flush pending transmissions.

3.3.2 FreeBSD's device polling

The FreeBSD's device polling framework [17] completely disables NIC interrupts, and runs the equivalent of the interrupt service routines at every timer tick (0.2 .. 1 ms on most systems) and optionally in the system's idle thread. The framework was designed for a totally different purpose (enabling moderation for devices that do not support it, and preventing livelock under high network load), but turns out to be extremely virtualization-friendly.

In fact, by removing interrupts, polling also removes the many VM exits that are issued to access the interrupt controller registers, so it is even more effective than interrupt moderation. Coupled with send combining, device polling also achieves great rates on the transmit path. As an example, just enabling polling on FreeBSD, with proper parameters, supports receive rates of up 890 Kpps, and to about 450 Kpps on the transmit side.

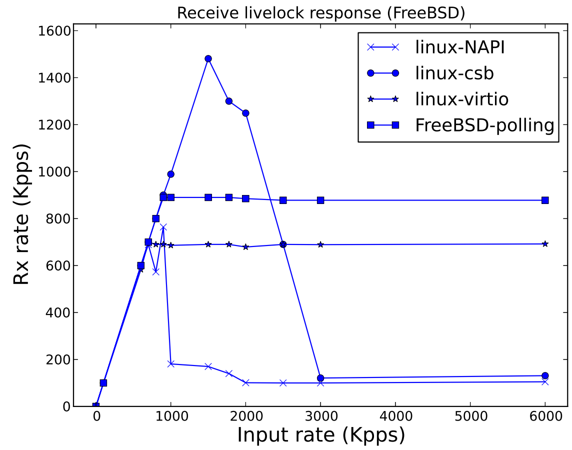

Figure 3: UDP receive throughput for different input traffic. Only Linux-virtio (without VHOST) and FreeBSD polling are able to sustain their peak rate for input rates up to 20 Mpps (the figure is truncated to 6 Mpps for clarity).

3.4 Receive livelock

Receive livelock [15] is a serious problem in high speed network environment. Interrupt moderation, coupled with modern, powerful CPUs, is often sufficient to alleviate the problem.

Within VMs, the problem is aggravated by two factors: the much higher cost of I/O operations, and the fact that emulated hardware can be subject to much higher input rates than the physical devices they emulate. As an example, the `e1000` driver normally manages 1 Gbit/s cards (1.488 Mpps), but the VALE[20] backend presented in Section 3.6 is able to drive it with up to 20 Mpps and 70 Gbit/s. This puts under stress most mechanisms to prevent livelock.

To illustrate the phenomenon, Figure 3.4 shows the receive throughput for on an UDP socket under variable input load (with input coming in bursts of 100 packets). For most configurations (including several not shown) the receive rate drops to very low values, or even to 0, as the input rate grows. The two solutions, in our experiments, that resisted to extreme input rates were Linux virtio (without VHOST) and the FreeBSD polling framework [17]. The latter includes a control mechanism that guarantees a user-configurable amount of CPU time to user processes. As such, even under extreme input loads, the desired fraction of CPU capacity is still available to perform work.

3.5 Paravirtualizing real NICs

Paravirtualized devices (virtio, vmxnet, xenfront) generally reduce the number of VM exits by establishing a shared memory region (we call it *Communication Status Block* or CSB) through which the guest and the VMM can exchange I/O requests without VM exits. On the first packet after an idle interval, the traffic source (either the guest or the IOthread) issues an initial notifications (called *kick*) to wake up the other peer. Kicks from the guest are writes to a NIC register and do cause a VM exit; kicks from the IO thread are interrupts, and cause VM exits too. After a kick, the two peers poll the CSB to look for new work (new packets available) or post results (packet processing completed), and go to sleep when no work is available for a while.

Same as other mechanisms discussed earlier, paravirtualization reduces the number of VM exits to one (or a few) per batch of packets, and becomes more effective as the load increases, a nice scaling property. The key features used by paravirtualization do not differ much from what is already implemented by modern NICs; in fact:

- the hardware already reports packet receptions and transmit completions through status bits in the descriptor ring, which is accessible through shared memory without VM exits;
- the register writes that start transmissions, and the receive interrupts are perfectly equivalent to a "kick" in virtio terminology;

The only missing features to support paravirtualization are copies (which the guest can update without VM exits) of the registers used to indicate new transmissions and new buffers for the receive ring.

Hence all it takes to build a full paravirtualized device is a small region of memory to implement the CSB (which includes the two registers mentioned above, plus some some extra control information), and the code to exchange information through it.

We have implemented paravirtualization support for the `e1000` and `r8169` devices and drivers by adding two PCI registers to the emulated NIC to point to the mapped CSB. The new capability is advertised through the PCI subdevice ID, and the device driver must explicitly enable the feature for the backend to use it.

Overall, this modification required about 100 lines of code in the guest device driver, and approximately the same amount in the frontend. The same CSB structure is used for both NICs.

This is immensely simpler than writing an entirely new device driver (about 3000 lines for virtio-net) and frontend (about 1500 lines). In terms of performance, our paravirtualized NICs perform as well as `virtio`. As such, this is a more viable approach to extend both guests and hypervisors that do not have paravirtualized devices (or have incompatible ones: it is amusing that even among paravirtualized NICs, each manufacturer has gone its own way, leading to the existence of at least three contenders — virtio, vmxnet, xenfront — which replicate similar functionality).

3.6 VALE, a fast backend

With all the enhancements described above, we have pushed our VM to the maximum rate supported by the backends (sockets, tap, host bridges, . . .) used to interconnect virtual machines. This bottleneck used to be hardly visible due to general slowness in the guest and device emulation, but the problem clearly emerges now. Some room for improvement still exists: as an example, the VHOST feature [11] avoids going through the IO thread for sending and receiving packets, and instead does the forwarding directly within the kernel. In our experiments VHOST almost doubles the peak pps rate over TAP for the `virtio` cases, slightly surpassing our best result with CSB (our `e1000` still does not support VHOST).

Nevertheless, reaching our target of several millions of packets per second requires a substantially faster backend. We have then implemented a QEMU backend that uses the VALE [20] software switch, which in itself is capable of handling up to 20 Mpps with current processor technology. Attaching QEMU to VALE pointed out a number of small performance issues in the flow of packets through the hypervisor (mostly, redundant data copies and lookups of information that could be cached). Our initial implementations could "only" reach 2.5 Mpps between guest and host, a value that has been more than doubled in the current prototype.

4. EXPERIMENTAL RESULTS

We have anticipated some performance numbers in the previous Section, but here we provide a more comprehensive comparison of the various mechanism presented. Exploring all possible combinations of components (guest OS; virtual CPUs; NICs; hypervisors; backends; virtual switch; load conditions) would be prohibitively time consuming, so we restrict our analysis to a (still large) number of relevant configurations.

We present now some results on the data rates we achieved between Guest and Host (GH), and between two guests (GG), with different emulators and combinations of features. Our source code containing all QEMU and guest modifications is available at [19].

General notes

For basic UDP, TCP and latency tests we have used the popular `netperf` program. A socket-based program (`netsend` from FreeBSD) has been used to generate rate-limited UDP traffic. For very high speed tests, exceeding the data rates achievable with sockets, we have used the `pkt-gen` program part of the netmap [16] framework. `pkt-gen` accesses the NIC (or the VALE port) bypassing the network stack, acts as a sink or as a source with programmable rate, and can sustain tens of millions of packets per second.

Our host system uses an Intel i7-3930K CPU @ 3.20GHz running Linux 3.8.11, QEMU-KVM is the git-master version as of May 2013, extended with all the mechanisms described in this paper. We use `tap` or VALE as backends.

Our guests are FreeBSD HEAD or Linux 3.8, normally using the `e1000` driver with small extensions to implement Send Combining (SC) and Paravirtualization (CSB). The interrupt moderation delay is controlled by the `itr` parameter (in 250 ns units).

We also ran some tests using other hypervisors (VirtualBox, VMware Player) and/or features (`virtio`, VHOST, TSO) to have some absolute performance references and evaluate the impact of the features we are still missing.

Note: the goal of this paper is to study the behaviour of the system *at high packet rates*, as those that may occur in routers, firewalls and other network middleboxes. Hence, our main focus are streams of UDP or raw packets. TCP throughput is only reported for completeness, but we did not try or want to optimize TCP parameters (buffer and window sizes, path delays, etc.) for maximum throughput but rely on system defaults, for good or bad as they are. For the same reason, we have normally disabled TSO, not because we neglect its importance, but because its use would otherwise hide other bottlenecks.

Notations and measurement strategy

We have used some significant combinations of the many parameters that influence results: the **VMM** (typically QEMU, but we ran some limited tests with VMware Player and VirtualBox); the **backend** (TAP or equivalent, or VALE); the number of **VCPUs per guest**; the **moderation delay** (`itr`); the use of either of the **SC** or **CSB** extensions; **packet size** (8 or 1460 bytes for UDP) or **write() size** for TCP (we used 1460, labeled `TCP`, or netperf defaults, labeled `TCPw`).

Below are full tables of experimental results in a variety of configurations for Linux and FreeBSD. We will provide a detailed discussion of some of the results in the rest of this section.

=== Linux on QEMU-KVM, TAP backend, Guest-Host ===

1 VCPU	UDP8	UDP-1460		TCP	TCPw	TCPRR
	Kpps	Kpps	Mbps	Mbps	Mbps	KTps
TX itr=0 BASE	30	28	326	103	182	11.8
TX itr=250	83	71	827	135	249	14.2
TX itr=250 SC	216	138	1613	1309	1521	4.6
TX itr=0 CSB	353	242	2828	2620	2594	21.2
TX itr=250 CSB	360	239	2794	2590	2757	7.6
RX itr=0 BASE				3370	3327	12.3
RX itr=100 SC				3912	3890	13.4
RX itr=100 CSB				7179	7493	

2 VCPU	UDP8	UDP-1460		TCP	TCP-w	TCPRR
	Kpps	Kpps	Mbps	Mbps	Mbps	KTps
TX itr=0 BASE	45	38	452	239	349	13.5
TX itr=250	84	54	630	329	396	13.5
TX itr=250 SC	291	136	1592	1723	1761	11.3
TX itr=0 CSB	369	190	2223	1993	1959	20.7
TX itr=250 CSB	343	172	2011	1690	1683	10.4
RX itr=0 BASE				3938	4118	
RX itr=100				1250	2894	
RX itr=100 SC				2300	5128	
RX itr=100 CSB				4400	7566	

=== Linux on QEMU-KVM, TAP backend, Guest-Guest ===

1 CPU TX = RX	UDP8	UDP-1460		TCP	TCPw	TCPRR
	Kpps	Kpps	Mbps	Mbps	Mbps	KTps
itr=0 BASE	29	28	321	416	462	7.1 *
itr=250	82	70	827	522	598	6.5
itr=250 SC	221	138	1615	1265	1661	5.1
itr=0 CSB	458	269	3147	2923	2982	11.8 *

2 CPU TX = RX	UDP8	UDP-1460		TCP	TCPw	TCPRR
	Kpps	Kpps	Mbps	Mbps	Mbps	KTps
itr=0 BASE	40	38	439	302	372	5.1
itr=250	82	70	815	578	646	6.1
itr=250 SC	354	163	1913	2174	2185	5.0
itr=0 CSB	409	258	3016	2747	2824	11.0
pkt-gen	400	360	4205			

=== Linux on QEMU-KVM, VALE backend, Guest-Guest ===

1 CPU TX = RX	UDP8	UDP-1460		TCP	TCPw	TCPRR
	Kpps	Kpps	Mbps	Mbps	Mbps	KTps
itr=0 BASE	32	31	363	559	805	7.4 *
itr=250	125	98	1143	792	859	7.1
itr=250 SC	455	186	2173	1856	2450	6.7
itr=0 CSB	1526	480	5616	4190	4206	12.3 *

2 CPU TX = RX	UDP8	UDP-1460		TCP	TCPw	TCPRR
	Kpps	Kpps	Mbps	Mbps	Mbps	KTps
itr=0 BASE	77	66	767	456	548	7.3
itr=250	118	90	1059	813	900	7.0
itr=250 SC	468	286	3350	2144	3130	6.6
itr=0 CSB	1221	447	5226	4380	4849	11.2
pkt-gen	2800					

Figure 4: Top: Performance of the various solutions with TAP or VALE as a backend, and Linux as the guest OS. Lines with an * indicate that the receiver is not able to sustain the transmit rate.

FreeBSD on QEMU-KVM, TAP backend, Guest-Host

1 VCPU	UDP8	UDP-1460		TCP	TCPw	TCPRR
	Kpps	Kpps	Mbps	Mbps	Mbps	KTps
TX itr=0 BASE	17	18	207	181	516	14.6
TX itr=250	64	55	640	267	538	8.9
TX itr=250 SC	237	167	1947	1060	1041	10.1
TX itr=0 CSB	392	247	2886	1783	1534	16.5
TX itr=250 CSB	379	244	2851	1767	1684	7.2

2 VCPU	UDP8	UDP-1460		TCP	TCP-w	TCPRR
	Kpps	Kpps	Mbps	Mbps	Mbps	KTps
TX itr=0 BASE	17	17	193	176	500	14.1
TX itr=250	59	53	625	251	522	12.2
TX itr=250 SC	244	165	1930	1068	1001	11.6
TX itr=0 CSB	400	244	2851	1862	1611	17.3
TX itr=250 CSB	388	243	2844	1746	1658	7.2

FreeBSD on QEMU-KVM, TAP backend, Guest-Guest

1 CPU TX = RX	UDP8	UDP-1460		TCP	TCPw	TCPRR
	Kpps	Kpps	Mbps	Mbps	Mbps	KTps
itr=0 BASE	20	19	224	185	633	7.1
itr=250	54	46	543	357	516	3.8
itr=250 SC	251	180	2109	739	770	3.9 *
itr=0 CSB	475	262	3055	1512	1496	9.5 *

2 CPU TX = RX	UDP8	UDP-1460		TCP	TCPw	TCPRR
	Kpps	Kpps	Mbps	Mbps	Mbps	KTps
itr=0 BASE	16	16	187	157	549	6.6
itr=250	50	48	564	422	531	3.8
itr=250 SC	250	170	1996	704	723	3.8 *
itr=0 CSB	464	268	3131	1350	1327	8.0

FreeBSD on QEMU-KVM, VALE backend, Guest-Guest

1 CPU TX = RX	UDP8	UDP-1460		TCP	TCPw	TCPRR
	Kpps	Kpps	Mbps	Mbps	Mbps	KTps
itr=0 BASE	20	18	205	160	804	7.9
itr=250	91	60	697	547	661	4.3
itr=250 SC	459	268	3126	927	936	5.7 *
itr=0 CSB	838	456	5325	1835	1821	9.9 *

2 CPU TX = RX	UDP8	UDP-1460		TCP	TCPw	TCPRR
	Kpps	Kpps	Mbps	Mbps	Mbps	KTps
itr=0 BASE	20	20	230	183	730	7.3
itr=250	85	65	764	439	621	4.3
itr=250 SC	440	258	3018	906	921	5.0 *
itr=0 CSB	817	449	5248	1684	1690	9.1 *
pkt-gen	5000	1890	22075			

Figure 5: Top: Performance of the various solutions with TAP or VALE as a backend, and FreeBSD as the guest OS. Lines with an * indicate that the receiver is not able to sustain the transmit rate.

Guest-Host (GH) measurements are useful to evaluate separately the transmit and receive path (the host being generally a faster source/sink). Tests between two guests (GG) show the end-to-end behaviour (on the same host).

4.1 Effect of NIC improvements

We show here a few graphs to compare the effect of NIC improvements on the communication speed.

Figure 6 shows how performance changes with various combinations of the parameters, using TAP (grey) or VALE (black) as a backend, and short UDP packets (the results for large UDP frames are similar). As it can be seen, paravirtualization of the `e1000` NIC (column labeled `csb`) reaches the same or better performance than VIRTIO, and is surpassed (when using TAP as a backend) only by VIRTIO+VHOST. The latter is

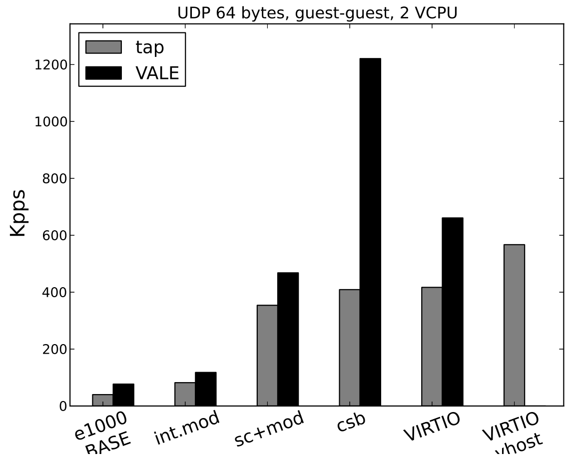

Figure 6: UDP throughput with various configurations.

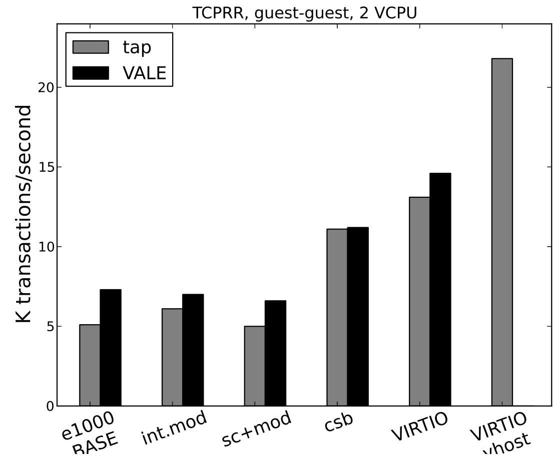

Figure 8: TCP RR rates with various configurations.

favoured by the fact that traffic does not have to go through the IO thread. Send combining together with interrupt moderation also provide significant speedups over the base case.

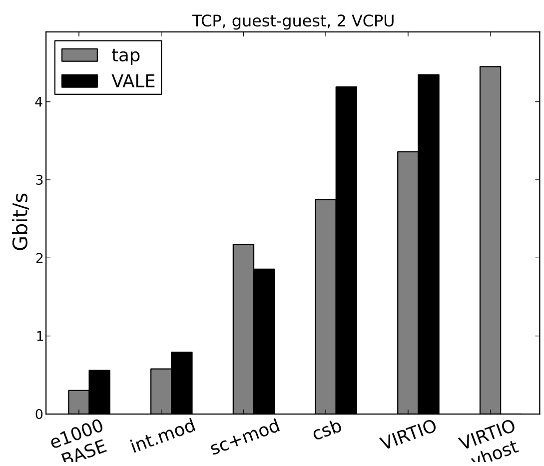

Figure 7: TCP throughput with various configurations (no TSO).

Figure 7 shows the behaviour of the various solutions for TCP traffic, again between two guests, and without using TSO (which is not supported in our e1000 device driver). Once again, the paravirtualized e1000 almost matches VIRTIO+TAP, and is slightly less performant than VIRTIO+VHOST.

4.2 Latency

Figure 8 presents the results of the "request-response" test (TCPRR) in netperf. The test, after establishing a TCP connection, exchanges one-byte packets between the peers, measuring the number of exchanges. Hence, it measures the average latency of the path. In this test, neither interrupt moderation nor send combining are effective, as there is normally only one packet in flight. The paravirtualized NIC slightly improves the latency, although it is marginally slower than VIRTIO, because the interrupt to issue the receive kick is more expensive than the one for VIRTIO. As usual, VIRTIO+VHOST gets a performance boost as it avoid going through the IO thread.

In all these three configurations we see that the VALE switch provides better performance than the TAP device, an effect that is especially visible at high packet rates.

4.3 TCP throughput

The throughput of a TCP flow is especially sensitive to the timing of packets. Adding delay in the connection can limit the throughput, especially with limited window sizes, and similarly, burstiness in the generation of data or acks can affect the way the window opens, and ultimately the throughput of short lived connections.

In Figure 9 we compare the throughput of a connection when the sender and the receiver are placed on the host or on the guest. What is evident here is that the basic `e1000` device is unable to send at high speed (columns Guest→Host and Guest→Guest) but the receive direction has much better performance (column Host→Guest), presumably because packets arrive in bursts. As we bring in interrupt moderation and send combining, the transmit direction starts exhibiting good performance, and the paravirtualized mode performs almost as good as VIRTIO.

4.4 VALE backend

The capacity of the VALE virtual switch, on our test machine, is approximately 20 Mpps with small UDP

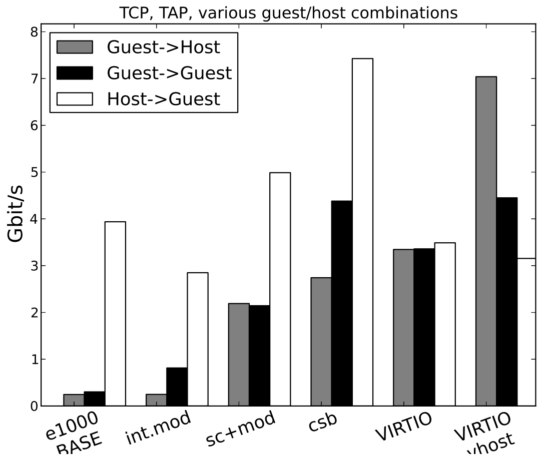

Figure 9: TCP RR rates with various configurations.

frames, and 70 Gbit/s with 1500-byte frames. When used to interconnect VMs, the VALE port is attached to the VMM backend and from there to the frontend. This introduces an extra data copy in each direction, leading to a reduction in the throughput. We can give a reasonable estimate of the throughput of the guest-guest path by running instances of `pkt-gen` within the guest, one in send mode and one in receive mode. Our tests indicate about 5 Mpps with 64-byte frames, and 1.89 Mpps with 1500-byte frames, corresponding to 22.7 Gbit/s. All these numbers are significantly higher than the packet rates we experience using regular sockets. This suggests that the VALE switch is not acting as a bottleneck for the system, unlike the case of TAP, which is limited to approximately 1 Mpps.

4.5 Comparison with other solutions

Linux on VirtualBox and VMware, 2 CPU

	UDP8 Kpps	UDP-1460 Kpps	UDP-1460 Mbps	TCP Mbps	TCPw Mbps	TCPRR Ktps
VBox TX GH	22	23	264	84	633	10.9
VBox TX GG	22	21	244	1121	1255	4.2
VMware TX GH	52	51	590	250	1332	13.2
VMware TX GG	65	64	748	3375	4138	9.2

FreeBSD on VirtualBox, vboxnet, Guest-Host

1 cpu, std	24	24	273	219	666	16.9
1 cpu, SC	24	24	273	232	928	17.0
2 cpu, std	60	58	676	570	690	14.7
2 cpu, SC	225	176	2064	1060	1100	14.7

Backend performance:

pktgen, tx	540	470.0	5500
pktgen, rx	670	270.0	3153

Figure 10: Performance of other VMMs (e1000 device). SC can help even without VMM modifications.

Linux on QEMU, VIRTIO, no accelerations

	UDP8 Kpps	UDP1460 Kpps	UDP1460 Mbps	TCP Mbps	TCPw Mbps	TCPRR Ktps
1 CPU GH tap	342	320	3742	3490	3485	24.3
2 CPU GH tap	334	312	3653	3346	3335	26.7
1 CPU GG tap	424	394	4610	3475	3480	14.5
2 CPU GG tap	417	367	4294	3359	3416	13.8
1 CPU GG vale	981	768	9184	4526	4532	15.3 *
2 CPU GG vale	875	774	9044	4346	3682	14.6 *

Linux on QEMU, VIRTIO, VHOST

	UDP8 Kpps	UDP1460 Kpps	UDP1460 Mbps	TCP Mbps	TCPw Mbps	TCPRR Ktps
1 CPU GH	644	564	6594	7144	7303	37.0
2 CPU GH	642	576	6724	7039	6880	35.8
1 CPU GG	528	477	5572	5893	5868	22.7
2 CPU GG	871	768	8976	4450	4539	21.8 *

Linux on QEMU, VIRTIO, TSO

	UDP8 Kpps	UDP1460 Kpps	UDP1460 Mbps	TCP Mbps	TCPw Mbps	TCPRR Ktps
1 CPU GH tap				11329	21478	
1 CPU GH vhost				13107	26750	
2 CPU GH tap				10809	21293	
2 CPU GH vhost				12687	26519	
1 CPU GG tap				19510	19435	13.7
1 CPU GG vhost				17286	33015	22.6
2 CPU GG tap				16558	18930	13.1
2 CPU GG vhost				12391	26867	21.6

Figure 11: Performance with `virtio` and TSO

The huge performance improvements that we see with respect to the baseline could be attributed to inferior performance of QEMU-KVM compared to other solutions, but this is not the case. Figure 10 shows that the performance of VirtualBox and VMware Player in a similar configuration is comparable with our BASE configuration with QEMU. We hope to be able, in the future, to run comparisons against other VMMs.

We also tried to use Send Combining on an unmodified VirtualBox (without moderation), and at least in the 2-VCPU case, see Figure 10, this is still able to give significant performance improvements, presumably due to a reduction in the number of VM exits on transmissions.

4.6 Missing features

While our performance numbers are excellent for UDP, the fact that we do not yet have support for features such as TSO is severely penalizing for the case of bulk TCP connections. Figure 11 shows that TSO increments the TCP throughput up to 3 times, so it is a worthwhile addition. Most of the support for TSO is already available in the driver and frontend we used, so it will not take long to add the feature also to the VALE switch.

Latency improvements would require moving the network backend from the IO thread to the KVM module, similarly to what is done in VHOST. This is a slightly longer term task, as it is more complex than adding TSO support and the benefits on throughput should be marginal.

5. CONCLUSIONS AND FUTURE WORK

We have shown how some small and simple modifications to hypervisors and device drivers can go a long way into making network performance on virtual machine very close to that of bare metal, in particular for the case of high packet rate workloads. Our extensions to the e1000 (and r8169) drivers can already improve the network performance of guest OSes by a large factor. The use of VALE as a virtual switch permits a much faster interconnection between virtual machines and the host. Our modifications, which are publicly available, will hopefully contribute to deploy high performance SDN components in virtualized environments, and also help the study and development of high speed protocol and applications over virtual machines.

Acknowledgements

This work has been supported by the European Commission in the contexts of project CHANGE and OPENLAB.

6. REFERENCES

[1] Agesen, O., Garthwaite, A., Sheldon, J., and Subrahmanyam, P. The evolution of an x86 virtual machine monitor. *SIGOPS Oper. Syst. Rev. 44*, 4 (Dec. 2010), 3–18.

[2] Agesen, O., Mattson, J., Rugina, R., and Sheldon, J. Software techniques for avoiding hardware virtualization exits. USENIX ATC'12, USENIX Association, pp. 35–35.

[3] AMD. Secure virtual machine architecture reference manual. http://www.mimuw.edu.pl/∼vincent/lecture6/sources/amd-pacifica-specification.pdf, 2005.

[4] Ben-Yehuda, M. Utilizing iommus for virtualization in linux and xen.

[5] Bugnion, E., Devine, S., Rosenblum, M., Sugerman, J., and Wang, E. Y. Bringing virtualization to the x86 architecture with the original vmware workstation. *ACM Trans. Comput. Syst. 30*, 4 (Nov. 2012), 12:1–12:51.

[6] Chisnall, D. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall, 2007.

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

[8] Gordon, A., Amit, N., Har'El, N., Ben-Yehuda, M., Landau, A., Schuster, A., and Tsafrir, D. Eli: bare-metal performance for i/o virtualization. *SIGARCH Comp. Arch. News 40*, 1 (Mar. 2012), 411–422.

[9] Gordon, A., Har'El, N., Landau, A., Ben-Yehuda, M., and Traeger, A. Towards exitless and efficient paravirtual i/o. SYSTOR '12, ACM, pp. 4:1–4:6.

[10] Guan, H., Dong, Y., ma, R., Xu, D., Zhang, Y., and Li, J. Performance enhancement for network i/o virtualization with efficient interrupt coalescing and virtual receive side scaling. *IEEE Transactions on Parallel and Distributed Systems 99*, PrePrints (2013), 1.

[11] Hajnoczi, S. QEMU Internals: vhost architecture. http://blog.vmsplice.net/2011/09/qemu-internals-vhost-architecture.html, 2011.

[12] Han, S., Jang, K., Park, K., and Moon, S. Packetshader: a gpu-accelerated software router. *ACM SIGCOMM Computer Communication Review 40*, 4 (2010), 195–206.

[13] Intel. Intel virtualization technology. *Intel Tech. Journal 10* (Aug. 2006).

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

[15] Mogul, J., and Ramakrishnan, K. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems (TOCS) 15*, 3 (1997), 217–252.

[16] Rizzo, L. Netmap home page. *Università di Pisa, http://info.iet.unipi.it/∼luigi/netmap/*.

[17] Rizzo, L. Polling versus interrupts in network device drivers. *BSDConEurope 2001* (2001).

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

[19] Rizzo, L., and Lettieri, G. The VALE Virtual Local Ethernet home page. http://info.iet.unipi.it/∼luigi/vale/.

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

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

[22] Salim, J. H., Olsson, R., and Kuznetsov, A. Beyond softnet. In *Proceedings of the 5th annual Linux Showcase & Conference - Volume 5* (Berkeley, CA, USA, 2001), ALS '01, USENIX Association, pp. 18–18.

[23] Schultz, M., and Crowley, P. Performance analysis of packet capture methods in a 10 gbps virtualized environment. In *Computer Communications and Networks (ICCCN), 2012 21st International Conference on* (30 2012-aug. 2 2012), pp. 1 –8.

[24] Solarflare. Openonload. *http://www.openonload.org/* (2008).

[25] Sugerman, J., Venkitachalam, G., and Lim, B.-H. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *USENIX ATC 2002* (Berkeley, CA, USA, 2001), USENIX Association, pp. 1–14.

[26] VMWare. Performance evaluation of vmxnet3. http://www.vmware.com/pdf/vsp_4_vmxnet3_perf.pdf.