Revisiting virtualized network adapters

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

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

Draft 5 feb 2013. Please do not redistribute, ask a copy to the authors.

Abstract

Network performance on virtual machines (VMs) is of critical importance for the virtualization of packet switching devices and middleboxes, largely used in Software Defined Networks. Nevertheless, most of the work on VM network performance so far has focused on workloads, such as bulk TCP traffic, which cover more classical applications of virtualization.

We expect packet processing boxes to become more and more important over time, so we studied how to deal with rates of millions of packets per second within VMs. In this paper we show that the goal is achievable without subverting existing architectures, and demonstrate a combination of techniques that let two VMs, using a slightly modified `e1000` device, exchange over 600 Kpps / 5 Gbit/s using sockets and 1500-byte frames, and almost 5 Mpps / 25 Gbit/s using the netmap API.

These results are comparable or superior to the state of the art even for paravirtualized devices, and have been achieved with very small modifications to existing device drivers and backends. The ideas we used are easily applicable to other devices and virtualizers, and our code is publicly available.

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*).

Ingenious software solutions [5], and later hardware support [13, 3], have filled the gap for CPU performance. Storage peripherals and bulk network traffic have comparable performance on 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).

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 and experiment with 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 report our results with four techniques, that can be used individually or in combination, to improve the packet processing rate of guests on top of QEMU-KVM while still using an emulated `e1000` device.

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, and iv) adapt the hypervisor to our high speed VALE [18] backend.

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[1], and our approach can be easily applied to other OSes and virtualization platforms.

Certainly this paper contains more engineering than fundamental theoretical contributions. Nevertheless we

[1] We are pushing the relevant changes to QEMU, FreeBSD and Linux.

believe that our experience can help the design and development of systems that are more virtualization-friendly and generally more efficient.

In the rest of this short 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, Virtual Machines (VMs) run on a Host which manages hardware resources with the help of a component on the Host called Virtual Machine Monitor (VMM). 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. Assuming virtualization support [13, 3] in the host CPU[2], most of the code for the guest OS is run directly on the host CPU operating in "VM" mode. Whenever critical resources (peripherals, etc.) need to be accessed, or interrupts must be delivered to the guest OS, a somewhat expensive context switch ("VM exit") is generated, and the VMM puts the CPU in host mode to perform appropriate actions, including emulating peripherals (these components of the VMM are called "backends").

VM exits are not needed for hardware that is directly accessible to the guest, either because it is a dedicated device or because it can expose multiple virtual peripherals; hardware mechanism such as IOMMU [4] help protecting from unauthorized peripheral access from guests. Likewise, it is possible for interrupts to be delivered directly to the guest without causing VM exits[3]. 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.

Apart from these cases, VM exits often constitute the main source of overhead in the guest. In particular, high network packet rates generally result in a correspondingly high rate of VM exits. Part of this work presents techniques to limit them.

[2] 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].

[3] Another way to remove interrupt-related exits is to have the guest poll the memory region, such as buffer descriptors, where peripherals post results. This is used quite often, especially for latency-sensitive applications.

2.1 I/O virtualization

Virtualization of I/O peripherals is described in [22], which also proposes some ways to accelerate the emulation. In order to further improve these early results, research and products have followed three routes:
1) hardware support in the peripherals ("virtual functions" and IOMMU's), so that guest machines can access directly and in a protected way subsets of the device and run at native speed;
2) run-time optimizations in the VMM. As an example, [2] shows how short sequences of code involving multiple I/O instructions can be profitably run in interpreted mode to save some VM exits;
3) reduce some costly operations in peripheral emulation (I/O instructions and interrupts) by designing "virtual" device models more amenable to emulation.

The latter approach ("paravirtualization") produced several different NIC models (vmxnet [23], virtio [19], xenfront [6]), in turn requiring custom device drivers in the guest OS. Synchronization between the guest and the VMM uses a shared memory block, 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 physical device models.

2.2 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), but these solutions do not help for packet processing devices (software routers, switches, firewalls), which must cope with true line rate 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 [16, 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 [20] 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 ex-

its 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.

3 Our proposal

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 [16, 7, 14, 21] 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) or other NIC or PCIe bus limits.

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 on the receive side (see Fig. 1 and 2). 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, so there is still a strong motivation for efficient NIC emulation.

Taking QEMU-KVM and `e1000` as a prototype platform, we then investigated how we could improve the performance of a generic, non paravirtualized network device. The techniques we propose in the following can be easily applied to other systems.

3.1 Adding interrupt moderation

Far from being a new idea, since interrupts are a significant source of uncontrolled system load, our first step was to implement the interrupt moderation registers, which are present in most NICs and used by operating systems, but not always implemented by emulators[4].

This requires modifications only in the emulator, and helps reducing interrupt storms at high packet rates, which can cause receiver livelock and (in a VM) severe reductions of the transmit rate. See the effect in Fig. 1, comparing the lines `itr=0` and `itr=100` (`itr` is the minimum value between interrupts, in 250 ns units).

3.2 Send Combining

Device drivers typically notify the NIC immediately when new packets are ready for transmission. This normally requires an IO or MMIO instruction, moderately inexpensive on real hardware, but causing a VM exit in a VM. To reduce the number of such exits we used an idea similar to what in [22, Sec.3.3] is called "Send Combining" (SC): 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. Transmissions may be delayed by up to the interrupt moderation delay (often in the order of 20..100 μs), but a similar worst case delay already occurs on the receive path due to interrupt moderation. The system administrator can pick appropriate tradeoffs, and both SC and moderation can be tuned to kick in only above a certain packet rate.

Ideally, one should strive for solutions that do not require changes to the guest; the original Send Combining could be 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 the only way to intercept I/O is VM exits. However the driver modifications involved are small (25 lines of code), and the benefits are huge – 3..5x speedups on the transmit side, see Fig. 1.

3.3 Paravirtualized e1000

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. Notifications (called *kicks*) issued to wake up the other peer do cause VM exits, but are only used to start up the communication. 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.

The way Send Combining works is actually very close to paravirtualization: the register write to start transmissions and the completion interrupt are perfectly equivalent to a "kick" in virtio terminology; packet buffers and metadata (descriptor rings) are already in shared memory and accessible without VM exits. Hence all it takes to build a full paravirtualized device is a small region of memory to implement the CSB, and the code to exchange information through it. Overall, this modification required about 90 lines of code in the device driver, and 100 lines in the QEMU side[5]. Two additional PCI registers have been added to the emulated `e1000` 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.

Once again this modification requires changes in the guest OS; once again it would be wonderful to improve performance without guest modifications, but this

[4] QEMU and VMware Player do not implements them; VirtualBox introduce them only in very recent versions.

[5] as of this writing we have only implemented the transmit side of the paravirtualization; the receive code will be smaller as it can reuse a lot of the infrastructure we added.

change is much less intrusive than adding a brand new paravirtualized device to the system, yet it makes our paravirtualized `e1000` perform as well as `virtio`.

3.4 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 [18] 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 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 [17].

General notes

For basic UDP, TCP and latency tests we have used the popular `netperf` program, or other socket-based applications. For very high speed tests, exceeding the data rates achievable with sockets, we have used the `pkt-gen` program part of the netmap [15] framework. `pkt-gen` accesses the network card bypassing the network stack and can sustain line rate packet rates even at 10 Gbit/s and 64-byte frames.

Our host systems run a recent Linux (3.2 or 3.7). QEMU-KVM is the git-master version as of Jan.2013, extended with all the mechanisms described in the previous Section[6]. We use `tap` or VALE as backends.

Our guests are FreeBSD HEAD or Linux 3.7, normally using the `e1000` device 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`).

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).

We repeated each experiment several times, and in many cases there are significant fluctuations (+/-5%) in the results. For brevity we have omitted confidence intervals, but keep these variations in mind when comparing results. Entries not present in the tables of results are missing because not relevant or because of technical issues.

4.1 Effect of NIC improvements

Figure 1 shows how performance changes with various combinations of the parameters. The lines labeled `BASE` correspond to unmodified QEMU and guest.

[6]The CSB is only implemented for packets transmitted from the guest; receive performance, however, is already reasonably good.

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

1 VCPU	UDP8 Kpps	UDP-1460 Kpps	UDP-1460 Mbps	TCP Mbps	TCPw Mbps	TCPRR KTps
TX itr=0 BASE	30	27	314	228	603	17.0
TX itr=100	82	71	831	520	650	4.5
TX itr=100 SC	317	202	2366	1239	1256	4.5
TX itr=0 CSB	711	332	3874	2272	1925	17.2
TX itr=100 CSB	717	326	3810	2226	2067	4.7
RX itr=0 BASE	260	236	2760	1150	1150	
RX itr=100	310	292	3421	1060	1060	
RX itr=100 SC				1943	1950	
RX itr=100 CSB				2140	2150	

2 VCPU	UDP8 Kpps	UDP-1460 Kpps	UDP-1460 Mbps	TCP Mbps	TCP-w Mbps	TCPRR KTps
TX itr=0 BASE	80	72	835	750	730	22.0
TX itr=100	83	74	840	730	815	4.5
TX itr=100 SC	316	196	2291	1850	1730	4.4
TX itr=0 CSB	700	318	3710	2498	2562	18.5
TX itr=100 CSB	710	315	3684	2488	2505	4.4
RX itr=0 BASE	480	403	4717	1250	1230	
RX itr=100	490	418	4890	1250	1230	
RX itr=100 SC				2300	2320	
RX itr=100 CSB				4400	4300	
TX pkt-gen	720	660	7780			
RX pkt-gen	576	517	6040			

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

2 CPU TX = RX	UDP8 Kpps	UDP-1460 Kpps	UDP-1460 Mbps	TCP Mbps	TCPw Mbps	TCPRR KTps
itr=0 BASE	67	64	755	645	700	8.7
itr=100	67	61	715	600	680	6.8
itr=100 SC	275	177	2073	1230	1150	6.4
itr=0 CSB	440	281	3284	1787	1800	7.1

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

2 CPU TX = RX	UDP8 Kpps	UDP-1460 Kpps	UDP-1460 Mbps	TCP Mbps	TCPw Mbps	TCPRR KTps
itr=0 BASE	111	92	1077	830	967	n.a.
itr=100	110	90	1050	1002	1043	1.9
itr=100 SC	540	299	3050	907	899	1.9
itr=0 CSB	629	480	5620	2551	2931	7.2
TX pktgen GG	6300	2700	31530			
RX pktgen GG	4700	2200	25700			
TX pktgen GH	7400	3440	40180			
RX pktgen GH	5400	3310	38660			

Figure 1: Our performance with TAP or VALE.

Guest-to-Host: Receive rates are generally good even in the base case, because the backend submits traffic in batches, and TCP flow control prevents overloads or livelocks. On the transmit side the BASE performance is much worse: only 30 Kpps with one VCPU, and 80 Kpps with two (the extra CPU can absorb the interrupt load, but the sender is still limited by excessive VM exits). **Moderation** by itself slightly helps with one VCPU or in the receive path, where interrupts constitute a huge load. **SC or CSB** (mutually exclusive) really make a difference when combined with moderation, greatly reducing residual VM exits[7] and letting the VM saturate the backend at over 700 Kpps (on bare metal, this guest does about 900 Kpps). The fact that the TAP interface becomes the limiting factor is shown by using `pkt-gen`: with TAP it is barely faster than `netperf`, whereas it becomes almost 10 times faster with VALE.

Guest-to-Guest: communication between two guests (still using TAP) is slightly slower than Guest-Host due to the additional delays and bottlenecks, but has the same response to optimizations. Overall the two guests can exchange up to 440 Kpps and 3.2 Gbit/s with UDP sockets, 1.8 Gbit/s with TCP (without TSO), and slightly more with `pkt-gen`.

VALE: this backend gives a further boost at high pps rates. Guest-to-Guest figures grow up to 630 Kpps and 5.6 Gbit/s with UDP, almost 3 Gbit with unoptimized TCP, and 4.7 Mpps and 25.7 Gbit/s with `pkt-gen`.

As a comparison, Figure 3 shows the best performance achievable with `virtio` and a combination of various related optimizations (VHOST, TSO, which can be added also to our `e1000` emulation). In absence of extra optimizations, our `e1000` performance is comparable to `virtio`, and the use of VALE supports much higher packet rates (presumably VALE could improve the `virtio` performance as well).

4.2 Comparison with other solutions

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 2 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 Fig.2, this is still able to give significant performance improvements, presumably due to a reduction in the number of VM exits on transmissions.

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

[7] There are almost no interrupts with CSB, and the few VM exits occur only when the sender and the IO thread overrun each other.

Linux on QEMU, VIRTIO, no accelerations

	UDP8	UDP1460		TCP	TCPw	TCPRR
	Kpps	Kpps	Mbps	Mbps	Mbps	Ktps
2 CPU TX GH	380	354	4143	3842	3834	28.6
2 CPU TX GG	438	395	4613	3661	3661	15.0
2 CPU RX GG	438	288	3369			

Linux on QEMU, VIRTIO, VHOST

2 CPU	TX GH	853	733	8572	6732	4603	36.1
2 CPU	TX GG	983	844	9865	1071	1090	22.3
2 CPU	RX GG	5	21	250	(*** livelock)		

Linux on QEMU, VIRTIO, VHOST, TSO

2 CPU	TX GH	649	582	6814	7460	22900	32.3
2 CPU	TX GG	812	714	8343	2873	13746	18.6
2 CPU	RX GG	3	10	120	(*** livelock)		

Figure 3: Effect of `virtio`, VHOST and TSO

4.3 Limitations and missing features

Latency[8] is an area where our mechanisms cause some loss of performance, as they try to group events to amortize expensive operations over large sets of packets. The Round Trip Time is generally worse (as shown by the TCP_RR tests), and the extra delay and batched transmissions may have negative effects on TCP.

We plan to add features such as VHOST (should help reducing latency, but has marginal effects on the pps rate) and TSO (which should instead have huge benefits on TCP performance, as shown in Figure 3) to our `e1000` emulation.

5 Conclusions and future work

Some small and simple modifications to hypervisors and device drivers, such as the ones shown in this paper, 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. The work presented here still needs improvements in several areas, as well as more extensive performance evaluation and comparison with more performant platforms, but the results achieved so far are extremely encouraging and useful, especially considering the simplicity of the techniques used. 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.

[8] A bug in the netmap backend discovered during the measurements impacts the latency-related tests. We expect to have it fixed for the final version of the paper. Of course!

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

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

[17] RIZZO, L., AND LETTIERI, G. The VALE Virtual Local Ethernet home page. http://info.iet.unipi.it/∼luigi/vale/.

[18] 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.

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

[20] 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.

[21] SOLARFLARE. Openonload. *http://www.openonload.org/* (2008).

[22] 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.

[23] VMWARE. Performance evaluation of vmxnet3. http://www.vmware.com/pdf/vsp_4_vmxnet3_perf.pdf.