Performance Analysis of Packet Capture Methods in a 10 Gbps Virtualized Environment

Michael J. Schultz and Patrick Crowley
Washington University in Saint Louis
Department of Computer Science and Engineering
Saint Louis, MO 63130-4899
Email: {mjschultz, pcrowley}@wustl.edu

***Abstract*—Network speeds are increasing and processor core counts rise while processor clock rates stagnate. This has led to both packet processing applications distributing their workload over several cores and virtualization of physical systems also using multiple cores. However, these two concepts are at odds with each other as both must take full advantage of multi-core systems for desirable performance.**

In this paper, we look at the performance considerations of dealing with 10 Gbps traffic rates in worst case loads using a bare-metal system and a virtual appliance model and several difference packet capture methods. We also discuss potential ideas to improve the performance of these virtual systems.

***Index Terms*—network monitoring; performance evaluation; throughput; virtualization**

I. Introduction

Network speeds have increased to 10 Gbps and with the IEEE 802.3ba standard approved [14] it may not be long before 100 Gbps networks are deployed in an enterprise setting. Meanwhile, in commodity CPUs, processor frequency scaling has been replaced by processor core scaling. Networking appliances have taken advantage of multi-core processors by distributing the network load to several processor cores on commodity servers to keep pace with the demands of a modern network. However, this proliferation of processor cores has also enhanced another factor: virtualization.

Virtualization allows several logically distinct computing appliances to exist on a single physical piece of hardware. Instead of having a physical machine dedicated to a firewall appliance and a second machine for an intrusion detection appliance, a network administrator can purchase a single physical machine and instantiate two virtual machines (VMs) on that physical hardware. With processor core counts continuing to rise, server consolidation makes financially sense and improves resource utilization.

However, if we couple the trend of distributing network processing over multiple cores with the trend of virtualizing network appliances the outcome may not be as good as expected. To better understand this problem we configured a physical machine to receive and process packets of several sizes using various packet capture methods. We then repeated the same experiments on a VM. Concretely, this paper makes the following contributions.

- A clear explanation of an experimental design that can be used to evaluate network processing workloads in a virtualized environment
- The results of a controlled, multi-factor experiment that suggest the following guidelines: single-core 10 Gbps performance is sustainable on bare-metal for maximum sized packets on 2.2 GHz processor, guest operating system throughput can achieve about 1 Gbps without specialized software, and on bare-metal a kernel module can outperform state-of-the-art user-space receive methods.
- A discussion of future directions in virtualized networking research

II. Virtualization Background

Most modern Intel and AMD processors have an extended instruction set that enables hardware virtualization support (VT-x [15] and AMD-V [5]). With support enabled the processor can execute an instruction that puts the processor into a special 'unprivileged' mode of operation. In this mode, if the instruction stream takes an action that would normally cause system state (special purpose registers, access special memory regions, etc.) to change a trap exception is generated. This trap exception fills out the reason for the trap in shared memory between the guest and the host and allows the host to take control of the processor. Since the host runs in 'privileged' mode it can modify system state and will emulate and process the reason the guest operating system caused the trap.

In a performance critical piece of software, like a high-speed network device, this trap-decode-emulate cycle can take too much time. To help alleviate this burden, a special type of device driver has been created called a "paravirtualized driver."

Paravirtualized drivers are a step towards achieving higher performance in virtual machine I/O. Both Xen [17] and KVM's virtio [24] drivers are paravirtual. This means that instead of the host system emulating a complete network device (typically Realtek's RTL8139) and its quirks it provides a software-only device driver optimized to run in virtual environments. For the virtio paravirtualization driver, it presents an interface to programmers who then can implement arbitrary devices that interact with the guest system. The virtio network driver uses a simple two queue system (transmit and receive). However—as we will see in this paper—there are problems when the driver is heavily loaded as the notification system between the host and guest is "primitive" [24].

	Bare-Metal	**Virtual Machine**
Linux Kernel	`3.2.1`	`3.2.1`
Operating System	CentOS 5.4	CentOS 5.5
Processor Clock Rate	2.27 GHz	2.27 GHz
Number of Cores	8	4
Private Cache Size	256 KiB	64 KiB
Shared Cache Size	8 MiB	–
Memory	12 GiB	6 GiB
Control NIC	Broadcom NetXtreme II	Realtek RTL8139
Data Bit Rate	10 Gbps	∞
Data NIC	Intel 82599EB	Qumranet Virtio
Data Driver	`ixgbe`	`virtio_net`

TABLE I
SUMMARY OF BARE-METAL AND VIRTUAL MACHINE CONFIGURATIONS.

III. EXPERIMENTAL DESIGN

One goal of this paper is to provide a clear performance baseline for future studies. To achieve this goal this section explains our system configuration, testing infrastructure, and our experiments.

A. System Configuration

In an effort to stay present with the mainline Linux kernel this study uses kernel 3.2.1 in both the bare-metal and virtual environment. The kernel is configured to enable KVM, KVM support for Intel processors, and virtio paravirtualization driver support [24] settings. Our machine specifications are summarized in Table I and detailed below.

The bare-metal system uses CentOS 5.4 with the custom kernel described above. Virtualization support is provided by KVM [4] (distributed with the kernel) and uses libvirt [1] (version `0.8.2`) as the virtualization controller. The virtual machine (VM) uses CentOS 5.5, again using the custom kernel version 3.2.1 described above instead of the default.

The physical hardware contains two Intel Xeon E5520 quad-core processors with VT-x enabled and hyper-threading disabled giving us eight physical cores that can use hardware accelerated virtualization. Each core has a clock rate of 2.27 GHz, a private L2 cache size of 256 KiB, and a processor-shared L3 cache size of 8 MiB; total system memory is 12 GiB. Unfortunately, our processor does not have I/O MMU virtualization (VT-d) available, preventing us to perform experiments using the PCI single-root I/O virtualization (SR-IOV) standard [20].

Attached to the physical hardware are two networking devices. The first device is a Broadcom NetXtreme II 1 Gbps network card that is used purely for control messages (SSH sessions, start/stop commands). This is connected to the 100 Mbps control network and uses the `bnx2` driver. The second device is an Intel 82599EB 10 Gbps network card and is used purely for the generated data traffic described in Section III-E. This network card is connected to a 10 Gbps data network and is capable of using MSI-X [21], VMDq [7], and SR-IOV (though again our chipset does not support SR-IOV). Linux kernel 3.2.1 ships with the `ixgbe` driver version `3.6.7-k` which we will use for most of our experiments, unless otherwise noted.

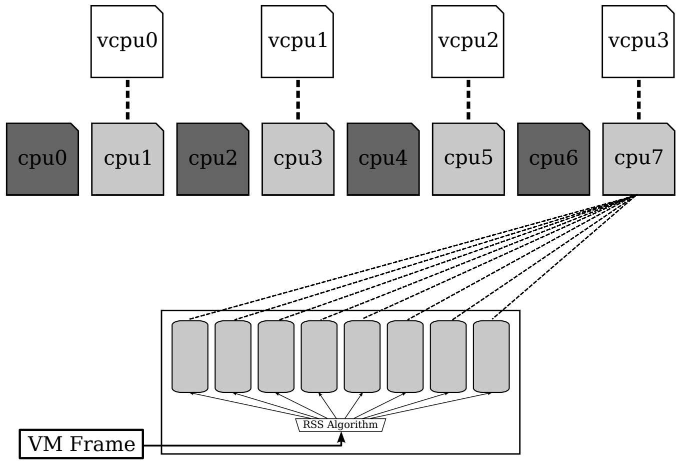

Fig. 1. Configuration of interrupt device receive queues, interrupt bindings, and virtual processor bindings.

During our virtualization experiments, the VM is configured with four virtual processors that use hardware virtualization (thus they also have a clock rate of 2.27 GHz), but certain "privileged" operations will be emulated by KVM. The VM is configured with 6 GiB of memory. As with the physical hardware, the VM has two network cards. The first device is a fully virutalized Realtek RTL8139 network card (the default virtual network card for most hypervisors). This card is connected to the control network, so its performance does not matter. The second device is the Qumranet Virtio network device connected to the data network. As mentioned in Section II the "virtio" driver is paravirtualized and is limited only to the amount of data that the processor can share between the host and guest systems.

Physical to virtual bindings can be seen in Figure 1. CPUs that are even numbered exist on the first physical processor; similarly odd numbered CPUs exist on the second physical processor. Thus, the four virtual CPUs at the top of the Figure are all bound to a single physical processor. The physical network card uses receive side scaling (RSS) to distribute incoming frames over distinct queues. Since experimental traffic is destined for a single machine, all receive queues have interrupts bound to `cpu7` to prevent inter-processor interrupts.

Virtual networking is done using a bridged host network. Simply put this creates a software switch ("bridge") in the host operating system that is connected to both the host data interface and the guest data interface. This gives the virtual data interface a *logically* direct connection to the control and data networks. The logical breakdown of receiving a packet on a bridged network can be seen in Figure 2. Traffic that is destined for the VM must go through the host system's bridge code before being pushed through a virtual network device for reception in the guest system.

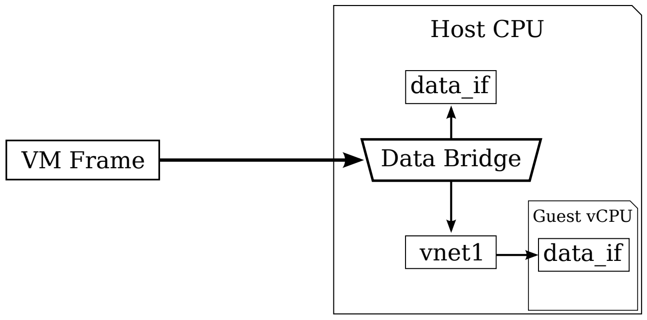

Fig. 2. Logical processing that must happen when a processor receives a packet at the software bridge.

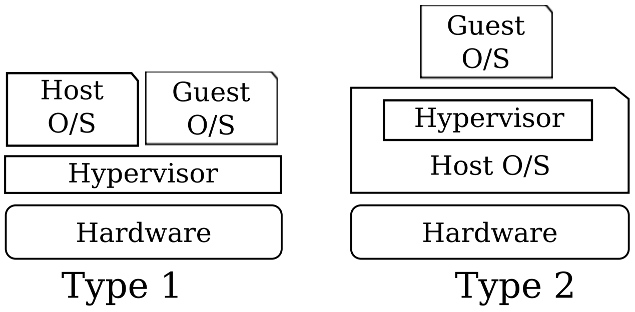

Fig. 4. Two hypervisor types. Our experiments use KVM (type 2).

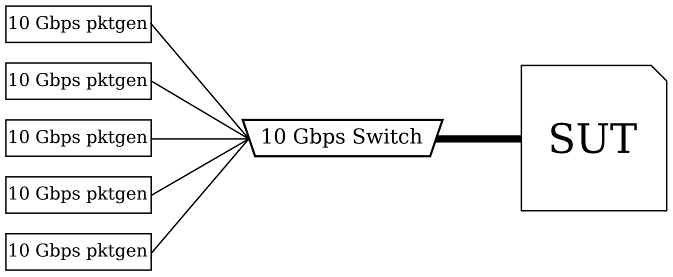

Fig. 3. Experimental setup for generating packets and sending them to the system-under-test (SUT).

B. Testing Infrastructure

All tests are performed using the Open Network Laboratory (ONL) [30]. Using ONL enables us to test the system in a reproducible fashion without unknown external factors.

Generating packets at 10 Gbps without specialized hardware is challenging, though easier than receiving packets at the same rate. To ensure there are enough packets for the system-under-test (SUT) to receive, we use a 10 Gbps network switch to aggregate multiple Linux kernel packet generators [18], as can be seen in Figure 3. Further details of the packet generation will be discussed in Section III-E.

Experimental runs are completely automated once the ONL topology for the experiment is loaded. For each packet capture method the software is built and processing is started on the SUT, then the packet generators are started. During the run, every 10 seconds the average throughput for that interval is emitted to a log file. On the completion of a test the packet generators are stopped, the packet processing software is stopped, the log file is gathered and stored in a unique file on a centralized host, and then the SUT is rebooted. Performance data is processed *post facto*.

C. Packet Processing Methods

One factor of interest to us is the differences between various software packet processing methods on bare-metal and virtual machines. As such, we have selected a kernel-space based packet processing method and two user-space packet processing methods. For the kernel-space packet processor we are using the Passive Network Appliance (PNA) [25]. The PNA has been previously developed by the authors to facilitate fast processing of packets on commodity hardware and has been proven to handle a constant load of about 1 Gbps in an operation setting. It also supports both 'null' monitoring—capturing a packet, but performing only header field extraction—and 'flow' monitoring—capturing a packet and inserting its features into the hash table.

For the user-space packet processing methods we use the Linux standard `PF_PACKET` capture methodand Fusco and Deri's `PF_RING` capture method [12]. In both cases a special user-space version of the PNA software was built to keep difference between the kernel- and user-space versions minimal (e.g. the kernel-space implementation uses the Linux `sk_buff` structure while the user-space implementation uses parameter passing for the data we need). With the processing code in place, the differences between all three implementations lies solely in the interactions with the device driver and kernel networking stack.

Since `PF_PACKET` is the default kernel implementation, we use the ubiquitous `lipcap` library (version 1.1.1) to capture packets [2]. During packet processing, the PCAP callback handler passes the packet data and true packet length to the user-space PNA software for processing.

For `PF_RING` based processing, we use the `pfring` library distributed with the sources [10]. As with the libpcap, the pfring library provides a callback handler that passes in the packet data and length which we simply pass on to the user-space PNA implementation.

Since the `PF_RING` kernel module supports a mode of direct access to the network card ("`transparent_mode=2`") it ships with a modified version of the `ixgbe` driver, so the bare-metal `PF_RING` experiments use this modified driver based on the 3.6.7 version. In the virtual experiments, there is no specialized driver and `PF_RING` must use the default mode of operations.

D. Operating System

As with the packet processing method, we are also interested in how bare-metal performance compares to VM performance. We have previously mentioned the hardware specifications for our bare-metal system and that we are using the KVM hypervisor for our virtual platform in Section III-A.

Factor	Level 1	Level 2	Level 3
Hardware Platform	Bare-metal	Virtual (KVM)	
Processing Method	Kernel Module	`PF_PACKET`	`PF_RING`
Packet Monitor	Null	Flow	
Flow Distribution	Single	Multiple	
Packet Size	64 bytes	256 bytes	1518 bytes

TABLE II
SUMMARY OF EXPERIMENTAL FACTORS AND TESTING LEVELS VARIED DURING OUR TESTING.

Beyond the specific configuration of the VM, there is a distinction between types of virtual machines [13]. KVM is defined as a "type 2" hypervisor, meaning that the host operating system boots as it normally would and the hypervisor (or virtual machine montior) is contained within the host system, as seen on the right of Figure 4. This type also includes products like Oracle's VirtualBox [19]. The other method of virtualization is a "type 1" hypervisor where the hardware is directly controlled by the hypervisor which starts a special host operating system to interact with the hardware, as seen on the left of Figure 4. This hypervisor type is used by Xen [9], VMware's ESX and ESXi [28].

E. Experiments

Beyond the three packet processing methods and two platforms we are testing, the other factors we have varied in our experimentation are: amount of processing to be done, input packet distribution, and input packet size. Table II shows a summary of all our experimental factors and their levels. Section IV presents our evaluation of these factors.

To vary the amount of processing done we use both a "null" monitor and a "flow" monitor. The null monitor is simple. It allows the Linux kernel to do the smallest amount of processing needed to deliver the packet our processing stage, at which point we capture the packet length for performance tracking and discard it. It also decodes some packet header fields, all of which are constant time operations. The flow monitor performs all the above steps and additionally inserts the flow features into a d-left hash table [29]. We define a flow record as the five-tuple consisting of the source and destination network address, transport protocol, and source and destination transport port numbers.

Since our goal is stress testing the packet processing methods, we are interested in the difference between a heavy, but computationally simple load and a heavy, but computationally intense load. This is done by using two flow distribution types: 'single' and 'multi.' The 'single' flow distribution maintains the exact same flow five-tuple for every packet sent to the system-under-test. In the flow monitor example this means that every packet will hash to the same table entry, update the feature counts, and finish processing. The 'multi' flow distribution makes every packet received by the SUT distinct, so each packet belongs to a different flow five-tuple. Specifically, each packet generator instance generates packets using a separate network address range and a port combination the does not repeat until over four billion packets have been send by that host. Since a 10 Gbps network can only send a maximum of 14.8 million packets per second, this ensures each packet belongs to a different flow.

Many modern network cards (including ours) implement TCP segment offloading which allows the hardware to coalesce packets in the same TCP flow before sending them to the operating system. To avoid any chance of this influencing our experiments the packets we generate are all UDP packets, so there is no point at which the hardware or software can try to artificially improve performance.

Our final factor is the input packet size. Again, since we are stress testing the system we want to know how it performs in different situations. For the input packet sizes we have selected 64 (min), 256, and 1518 (max) byte sized packets. Note that generating 64 byte packets at high rate on commodity hardware does not saturate the 10 Gbps link; however, each packet generator instance is capable of generating approximately 600 Mbps of data and the aggregation of that gives an input rate of 3 Gbps which is much higher than the SUT can handle in that scenario.

IV. EVALUATION

To evaluate our system, we used a full factorial design with the five factors at each of the levels listed in Table II [16]. Each experiment was replicated five times with each replication running for 240 seconds. The system-under-test was rebooted after every test to ensure any residual state is cleared out prior to beginning the next test. Throughput samples were taken every 10 seconds with the first two and last two discarded as they may include samples taken before all the packet generators have warmed up or cooled down. Some replications may also include fewer data points if the system-under-test did not reach a stable state when testing began (i.e. the system had not finished booting).

The main packet processing in all experiments is bound to a single core in both the bare-metal and virtual machine runs. However, each machine does have other cores available that idle most of the time and can perform whatever bookkeeping tasks that the operating system needs to run (e.g. control network connections for data collection).

Our first set of experiments determines the bare-metal performance of our platform. Figure 5 shows the throughput values for this set of experiments. Though not shown on the graph all the 95% confidence intervals are all within 20% of their respective averages and most are withing 5%. This clearly shows very poor throughput performance in the minimum sized packet case with the `PF_PACKET` capture method at 330 Mbps, below both `PF_RING` (391 Mbps) and the PNA kernel module (1056 Mbps) in the null monitor case at this packet size. In the flow monitor case none of the methods showed strong statistical significance in their differences.

As packet sizes scale up to 256 and 1518 bytes per packet the PNA kernel module is within 200 Mbps of line rate in all cases. This means that time it takes to receive a maximum sized packet is longer than the time it takes to insert the flow record into our hash table.

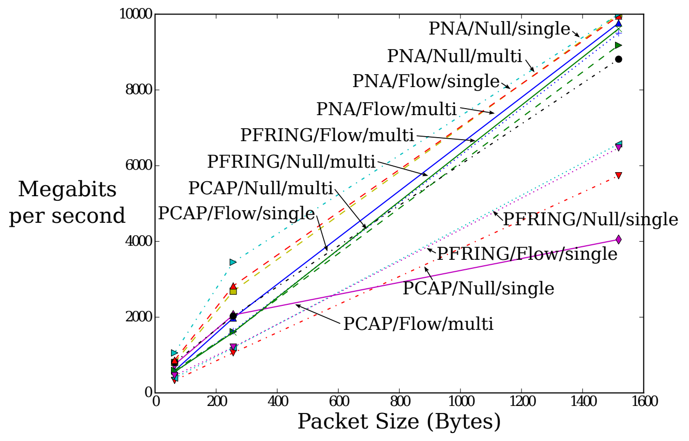

Fig. 5. Bit rates for our experiments running on bare-metal.

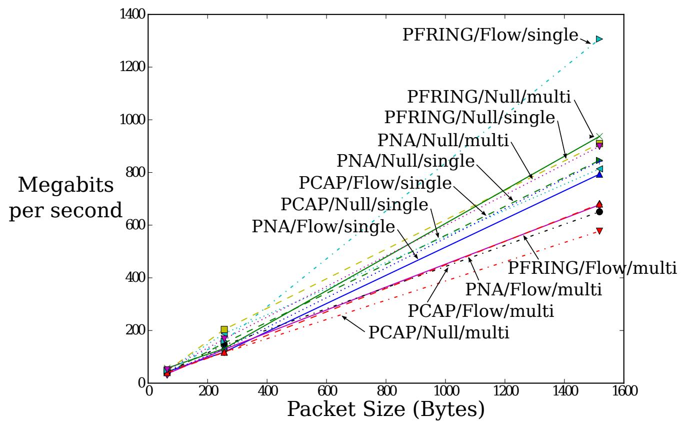

Fig. 6. Bit rates for our experiments running on a virtual machine.

Our next set of experiments looked at the performance of VMs in packet capture, the results of which are shown in Figure 6. To better show the details, the y-axis of Figure 6 only goes to 1400 Mbps instead of 10,000 Mbps as used in the bare-metal results. Unfortunately, in all cases the throughput results are so small that there is no statistically significant difference among the points. Even the “best” performing method (PF_RING with 1518 byte packets and a single flow), which has an average throughput of 1306 Mbps comes with a 95% confidence interval of $\mp$846 Mbps!

This result demonstrates the complexity of virtualization. As shown earlier in Figure 2 and discussed in Section III-A when a packet is destined for a virtual machine it must first be received by the host operating system, processed through a software bridge, pushed to the hypervisor, and then pulled into the kernel of the guest operating system, at which point several of the packet reception stages occur a second time. Even with the paravirtualized virtio-net driver used by KVM, the packet still must be received by the host, pushed into the user-space virtio driver then shared with the guest system. Though the implementation is different, Xen uses a similar system for paravirtualized networking.

Throughput	**64 bytes**	**256 bytes**	**1518 bytes**
10 Gbps	14880.95	4528.98	812.74
1 Gbps	1488.09	452.90	81.27

TABLE III
SUMMARY OF BARE-METAL AND VIRTUAL MACHINE CONFIGURATIONS.

Figure 7 shows the actual packet rates for both the bare-metal and virtual machine experiments. For reference Table III shows the maximum kilopacket per second values expected at 10 Gbps and 1 Gbps for the three packet sizes used. We only label the highest performing VM experiment, but recall that there is not a statistically significant difference between any of the VM experiments.

This figure clearly shows that the host operating system is failing to deliver packets at a rate that is even close to proportional to the input rate with each packet size showing a rate of approximately 70 kpps. The bare-metal number also show that, at 10 Gbps, between 64 byte and 256 byte sized packets there is also a processor performance bottleneck. In most cases while as the packet sizes scale up to 1518 bytes the system can perform at or near network speed.

V. DISCUSSION

During our experiments we found several interesting pieces of information. First, the Linux kernel version can have a major impact on performance. Though not shown in any of our results, using older kernel versions could bring our bare-metal throughput down from about 10 Gbps to 2.5 Gbps. This is our prime reason for keeping the kernel up-to-date. Similarly, the actual device driver used in the kernel can have a major effect. When using the ixgbe 2.0.62-k2 driver distributed with Linux kernel 2.6.34 we had peak performance numbers of 2.5 Gbps (as stated above), but moving to the ixgbe 3.6.7 version improved our throughput to about 5 Gbps.

We also observed that our initial scheme of binding all receive interrupts to cpu7 and binding the virtual packet processor to vcpu3 (which is also bount to cpu7) resulted in throughput performance in the 10 Mbps range (for 64 byte packets). We believe this to be caused by contention on the CPU between the host operating system using NAPI to poll for packets and the guest operating system also using NAPI to poll for packets (or disabling and reenabling interrupts through the trap-decode-emulate cycle of virtualization). As such we put the host receive and address space switching portion on one CPU and the actual packet processing on a second CPU. While technologies such as SR-IOV should help by reducing the host operating system's burden, this is still an area that should be explored.

As will be discussed in the following section, Dong et al. have begun exploring the interrupt emulation path for SR-IOV aware systems [11]. We believe this to be an important aspect of high-performance virtual systems as interrupts are frequent and modify system state, meaning they must frequently take the trap-decode-emulate path. Hardware support for such systems may be in the near future, but the software

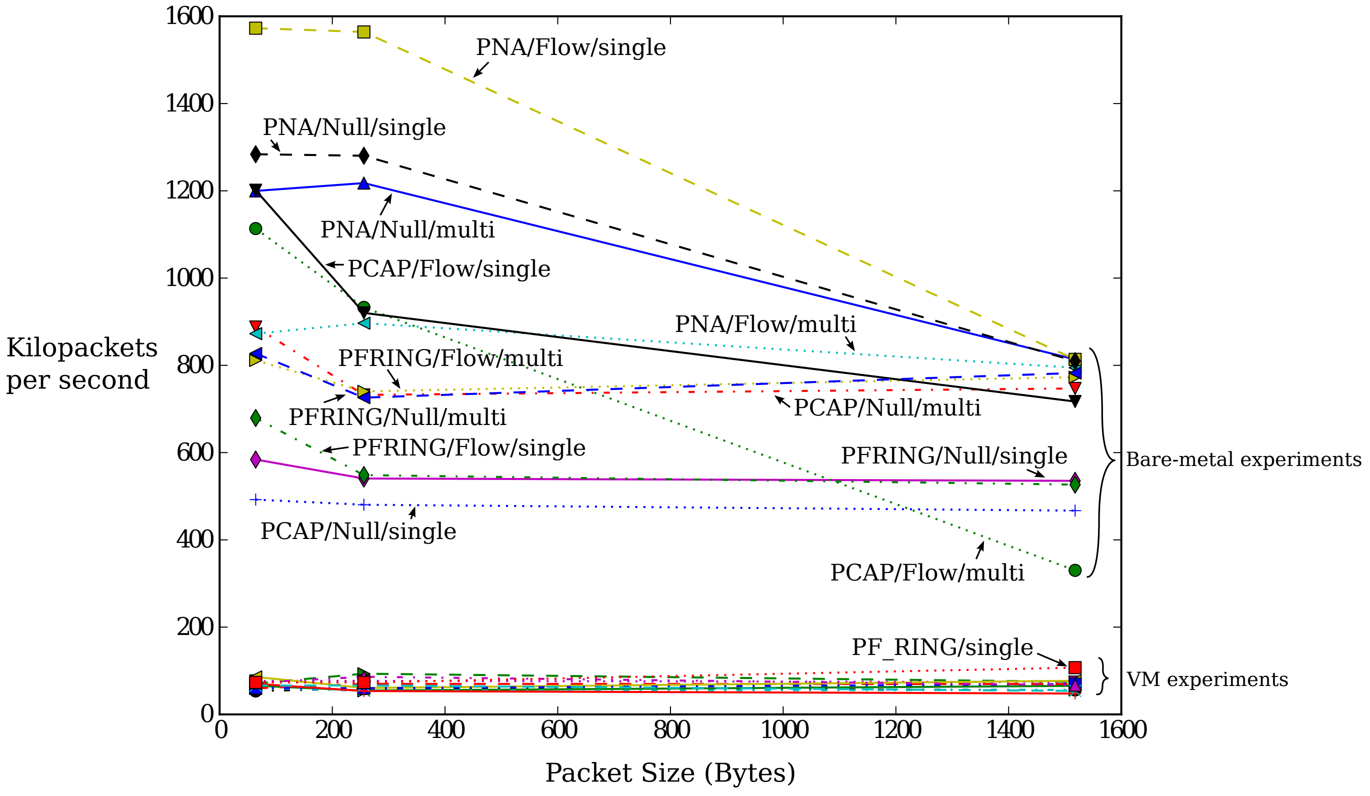

Fig. 7. Packet rates for both the bare-metal and virtual machine experiments.

aspects of interrupts should still be further explored to reduce unnecessary overheads.

VI. Related Work

Recent papers have looked at both network virtualization and improving the performance of network processing in a virtual environment. Note that these are distinct ideas. "Network virtualization" research turns the physical network into a software definable network for administrative purposes. These proposals have consequences in virtual machine network performance because they provide more direct methods of capturing packets for processing [8][22][27]. Our goal is to achieve high performance networking in virtual machines; network virtualization's goal is to better use network resources.

In the area of high performance networking in a virtual environment there are proposals focused on improving the path from initial frame reception to final processing on the VM. Prior to Intel and AMD providing a virtual execution environment, Sugerman et al. explored the use of VMware Workstation's Virtual Machine Monitor [26]. Their work explores handling I/O devices on a 733 MHz processor with a 100 Mbps network. The analysis focuses on the transmit side, though "TCP receive yields similar results and conclusions" [26]. Their results show host side processing overhead consumes most of the time during packet transmission. VM performance matches native performance if the processor is sufficiently powerful and data sizes are exceed 512 bytes.

A later document by VMware evaluates the ESX and ESXi hypervisor in a 10 Gbps network environment [3]. This paper focuses on the packet processing performance for 1518 byte frames and jumbo (9000 byte) frames. At these packet sizes they are able to achieve 5.4 Gbps in a single virtual machine. Aggregating the throughput of multiple VMs allows them to achieve network speeds of 9 Gbps. Unfortunately, due to resource constraints we were not able to test their platform.

Hardware techniques to aide in virtual machine performance exist in the form of Intel's Virtual Machine Device queues (VMDq) [7]. VMDq allows the network card to maintain multiple hardware queues with separate interrupt request line to the processor (which can be tied to a specific core). This prevents the network card from raising an interrupt on one core, running through the software bridge code, and then raising an inter-processor interrupt to the core running the VM. This moves the software switch to the network card, however the host operating system still handles the interrupt, some amount of packet processing, and moving the packet from host virtual address space to guest virtual address space.

More recently memory chipsets and the PCI bus have added virtualization support with the I/O MMU and single-root I/O virtualization (SR-IOV) [20]. These technologies allow I/O devices to provide performance critical PCI bus access directly to the VM and enable the I/O MMU to be aware of the guest's address space. Combining these allows a guest operating system to directly interact with the network card by providing it with DMA addresses in its virtual memory space so the host system does not need to perform translations. By doing this, the host operating system no longer has to receive the packet, process it, and re-map it to the guest's address space; the host only needs to forward the interrupt to the appropriate guest operating system [23].

Recognizing this interrupt bottleneck, Dong et al. analyze and improve the interrupt performance overheads between the host and guest systems [11]. By taking advantage of SR-IOV and looking at the aggregate throughput of 8 VMs they are able to achieve a throughput of 9.57 Gbps in the virtual environment while reducing the CPU utilization that interrupt emulation introduces. Our hardware platform does not support the PCI SR-IOV specification, so our experiments cannot reflect these differences. We plan on purchasing SR-IOV aware systems and re-running our experiments in the future.

In late 2011, Cardigliano et al. extended the `PF_RING` [12] packet capture method used in this paper to be virtual machine aware with `vPF_RING` [6]. This extension uses the virtio framework to effectively bypass the hypervisor by using `mmap()` to provide direct access to packet memory to the paravirtualization driver, then using the event channels to map memory into the guest system. This allows `vPF_RING` to achieve about 1.8 Gbps on their testing platform. Due to time and compatibility problems we were not able to get the `vPF_RING` system to work in our environment.

We believe our work complements each of these by providing a baseline for single core packet processing performance in VMs, rather than aggregate performance.

VII. Conclusions

With network speeds and processor core counts continuing to increase while processor clock rates stagnate we see a proliferation of products that provide either high performance networking or good virtualization. Both of these take advantage of multiple cores to achieve the desired performance levels, however seldom do the two meet. High performance virtual network appliances will become important in the near future as network administrators move from physical appliances like firewalls and intrusion detection systems to systems that can host two or more virtual network appliances.

In this paper we looked at how well a physical machine can perform in several scenarios to stress test it in a 10 Gbps environment, then ran the same tests against a virtual machine. The results from the virtual machine experiments were—while quite low—not unsurprising given the amount of overheads incurred in moving a packet through the system.

Based on our results we developed these guidelines.

- Single-core 10 Gbps performance is sustainable on bare-metal for maximum sized packets on 2.2 GHz processor
- Guest operating system throughput can achieve about 1 Gbps without specialized software
- On bare-metal a kernel module can outperform state-of-the-art user-space receive methods

As hardware support becomes accessible, virtual machine throughput will improve but network speeds will also continue to increase. It is still important that software support be in place to ease the burden for virtual machines.

References

[1] libvirt virtualization API. http://libvirt.org/.

[2] Tcpdump/libpcap public repository. URL http://www.tcpdump.org/.

[3] 10gbps networking performance. Technical Report PS-071-PRD-01-01, VMWare, November 2008. URL http://www.vmware.com/pdf/10GigE_performance.pdf.

[4] KVM: Kernel based virtual machine, February 2012. http://www.linux-kvm.org/.

[5] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual*, volume 2: system programming edition, December 2011.

[6] A. Cardigliano, L. Deri, J. Gasparakis, and F. Fusco. vpf_ring: Towards wire-speed network monitoring using virtual machine. In *Internet Measurement Conference*, 2011.

[7] S. Chinni and R. Hiremane. Virtual machine device queues. White Paper, 2007.

[8] N. M. K. Chowdhury and R. Boutaba. Network Virtualization: State of the Art and Research Challenges. *IEEE Communications Magazine*, 47(7):20–26, July 2009.

[9] Citrix Systems, Inc. The xen hypervisor. URL http://xen.org/.

[10] L. Deri. PF_RING, May 2011. URL http://www.ntop.org/PF_RING.html.

[11] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan. High performance network virtualization with sr-iov. In *High Performance Computer Architecture*, pages 1–10, 2010.

[12] F. Fusco and L. Deri. High speed network traffic analysis with commodity multi-core systems. In *Internet Measurement Conference*, pages 218–224, 2010.

[13] R. P. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, Cambridge, MA 02138, February 1973.

[14] IEEE. IEEE P802.3ba 40Gb/s and 100Gb/s Ethernet Task Force. URL http://www.ieee802.org/3/ba/.

[15] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3b: system programming guide, part 2 edition, May 2011.

[16] R. Jain. *The Art of Computer Systems Performance Analysis*. Wiley Professional Computing, 1991.

[17] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in Xen. In *Proceedings of the Annual Technical Conference on USENIX*. USENIX, 2006.

[18] R. Olsson. pktgen the linux packet generator. In *Ottawa Linux Symposium*, 2005.

[19] Oracle. Virtualbox. URL https://www.virtualbox.org/.

[20] PCI-SIG. Single Root I/O Virtualization.

[21] PCI-SIG. MSI-X, June 2003. URL http://www.pcisig.com/specifications/conventional/msi-x_ecn.pdf.

[22] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending networking in the virtualization layer. In *Workshop on Hot Topics in Networks (HotNets)*, 2009.

[23] S. Rixner. Network virtualization: Breaking the performance barrier. *ACM Queue*, 6(1), January/February 2008.

[24] R. Russell. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Operating Systems Review*, 42(5), 2008.

[25] M. J. Schultz, B. Wun, and P. Crowley. A Passive Network Appliance for Real-Time Network Monitoring. In *Architectures for Networking and Communications Systems*, pages 239–249, October 2011.

[26] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*. USENIX, 2001.

[27] D. Unnikrishnan, R. Vadlamani, Y. Liao, A. Dwarkaki, J. Crenne, L. Gao, and R. Tessier. Scalable Network Virtualization Using FPGAs. In *Symposium on Field Programmable Gate Arrays (FPGA)*, 2010.

[28] VMware. Vmware esx and esxi info center. URL http://www.vmware.com/products/vsphere/esxi-and-esx/.

[29] B. Vöcking. How asymmetry helps load balancing. *Journal of the ACM*, 50(4), July 2003.

[30] C. Wiseman, J. Turner, M. Becchi, P. Crowley, J. DeHart, M. Haitjema, S. James, F. Kuhns, J. Lu, J. Parwatikar, R. Patney, M. Wilson, K. Wong, and D. Zar. A remotely accessible network processor-based router for network experimentation. In *ANCS*, pages 20–29, New York, NY, USA, 2008. ACM.