vPF_RING: Towards Wire-Speed Network Monitoring Using Virtual Machines

Alfredo Cardigliano [1]
Luca Deri [1 2]
[1] ntop, [2] IIT-CNR
Pisa, Italy
{cardigliano, deri}@ntop.org

Joseph Gasparakis
Intel Corporation
Shannon, Ireland
joseph.gasparakis@intel.com

Francesco Fusco
IBM Research
Rüschlikon, Switzerland
ffu@zurich.ibm.com

ABSTRACT

The demand of highly flexible and easy to deploy network monitoring systems has pushed companies toward software based network monitoring probes implemented with commodity hardware rather than with expensive and highly specialized network devices. Deploying software probes under virtual machines executed on the same physical box is attractive for reducing deployment costs and for simplifying the management of advanced network monitoring architectures built on top of heterogeneous monitoring tools (i.e. Intrusion Detection Systems and Performance Monitoring Systems). Unfortunately, software probes are usually not able to meet the performance requirements when deployed in virtualized environments as virtualization introduces severe performance bottlenecks when performing packet capture, which is the core activity of passive network monitoring systems.

This paper covers the design and implementation of vPF_RING, a novel framework for efficiently capturing packets on virtual machines running on commodity hardware. This solution allows network administrators to exploit the benefits of virtualization such as reduced costs and centralized administration, while preserving the ability to capture packets at wire speed even when deploying applications in virtual machines. The validation process has demonstrated that this solution can be profitably used for multi-gigabit network monitoring, paving the way to low-cost virtualized monitoring systems.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—DNS; C.2.3 [**Network Operations**]: Network monitoring.

General Terms

Measurement, Performance.

Keywords

Virtualization, Packet Capture, Passive traffic monitoring.

1. INTRODUCTION AND MOTIVATION

In the past years, one of the industry trends is to optimize rack space and power consumption while simplifying administration by migrating physical servers onto Virtual Machines (VMs).

In the context of network monitoring, the idea of running multiple monitoring systems on independent VMs deployed on the same physical box is definitely appealing. By running software probes in virtualized environment network administrator can delegate certain tasks (i.e. performance management) to third persons, each having full access and control over specific virtual machines. VMs are often used for implementing monitoring on demand: namely activate monitoring facilities on specific network locations whenever certain network conditions happen (e.g. security alert). In addition, virtualization provides substantial deployment benefits in all the cases where multiple monitoring applications require access to the entire (or a subset of) the network traffic for performing different analysis tasks. By running heterogeneous monitoring software on the same box the complexity of deploying systems responsible to simultaneously dispatch the traffic towards multiple network analysis boxes can be completely avoided.

Since traffic splitting or dispatching is usually implemented by deploying advanced hardware based multi-port network taps and management networks, virtualization in the context of network monitoring allows to substantially reduce the deployment costs of advanced multi-probe monitoring architectures.

Unfortunately, running multiple monitoring systems in virtualized environments is desirable but not yet a common practice mostly due to the severe performance bottlenecks that virtualization introduces in application performing network monitoring. The most critical performance bottleneck introduced by virtualization when used for network monitoring is caused by the inefficient implementation of packet capture, which is the most important building block for most network monitoring applications. In fact, passive network monitoring systems strongly depends on packet capture, which is the process of accessing the stream of packets flowing on a network link. The packet stream is captured for performing several tasks, including network troubleshooting, traffic accounting, security breaches detection, and performance monitoring. Depending on the number of monitoring systems and on

IMC'11, November 2–4, 2011, Berlin, Germany.

their nature, the same packet have to be captured several times. For instance an intrusion detection system (IDS) and a traffic accounting application might need the same packets for accomplishing their respective monitoring tasks. Depending on the criteria these systems use for classifying traffic [1], they might be interested in capturing all packets independently from their nature and content, or only a subset of packets that match specific filtering rules (e.g. all UDP packets sent by server 192.168.1.1) usually specified using BPF filters [2] or hardware-assisted filtering rules supported by modern network adapters [21].

The following figure shows the packet capture performance (packets are just captured and not processed) of a single VM running on a Kernel-based Virtual Machine (KVM) [3] while capturing packets using VirtIO-Net [4] on a quad-core Xeon system.

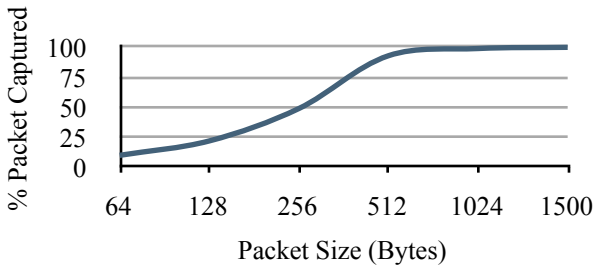

Figure 1. Packet Capture Rate at 1 Gbit using KVM

Since packet capture performance under a KVM virtualized host is poor for small packets and acceptable only for medium/large packets, KVM virtual machines are not suitable for running network monitoring applications.

In the past years, the authors have developed PF_RING [5] [45], a Linux kernel module implementing a framework that can be profitably used for simplifying the development of efficient monitoring applications requiring high speed packet capture. PF_RING substantially accelerates packet capture under the Linux operating system and for this reason is widely used on specific field such as for accelerating IDS/IPS and for passively monitoring network traffic using flow-based tools. Unfortunately, the framework does not provide packet capture acceleration under virtualized environments. In this work, we introduce virtual PF_RING (vPF_RING), an high performance packet capture solution optimized for virtualized environments that solves the performance bottlenecks present in KVM when used for deploying multiple monitoring applications on the same physical box. To improve the packet capture performance the authors capitalize on vNPlug, a novel framework that provides hypervisor bypass.

The rest of the paper is structured as follows. In section 2, we discuss how networking has changed with the advent of virtual machines. In section 3, we cover the design of vNPlug a framework that implements support for hypervisor-bypass. In section 4, we describe how vNPlug has been successfully used to create vPF_RING, an extension of PF_RING for virtualized environments. Finally in section 5, vPF_RING performance is evaluated.

2. RELATED WORK

2.1 Networking in Virtual Machines

Virtualization is appealing for the industry as it simplifies administrative tasks, while reducing costs and increasing scalability [29] [40]. In the past few years there have been many efforts to improve network performance on VMs [33] [34], both with hardware [37] and software solutions [36]. However, with the only exception of Endace, which offers the virtual DAG capture device (vDAG [18]), there are no other companies addressing the problem of using VMs for high-performance network monitoring but just for general network operations. A popular solution for bypassing the hypervisor during network operations is to map the network adapters directly inside the VM such as VMware VMDirectPath [31] [43]. We believe that this solution is sub-optimal for traffic monitoring because:

- Each VM would need a physical network port, thus increasing:
  - The operational costs due to the need of private per-VM ports.
  - The number of monitoring ports.
  - The complexity required during live VM migration and reconfiguration, being each VM bound to a specific network port on which traffic to be analyzed is received.
- If multiple VMs running on the same host need to analyze the same traffic with different goals (e.g. run a VM with an IDS and another VM with a NetFlow probe), it is necessary to use specialized hardware for duplicating the packets to be dispatched to each adapter.
- As packet capture is a costly activity in terms of CPU cycles required, capturing the same packet multiple times on various VMs running on the same hosts is more expensive than capturing the packet once and dispatching it to multiple VMs especially when having high throughputs and small packets.
- As hardware adapters are accessed directly by VMs, it is necessary to install native drivers into the guests, adding an extra constrain to live migration with respect to virtualized adapters.
- Physical and virtual IRQ sharing, and device assignment dependencies [43] can jeopardize the performance of the system and make this solution effective only on specialized servers.

Paravirtualization [19] has been the first attempt to reduce the overhead of emulating real network devices. By implementing paravirtualization, the guest operating system is aware of being virtualized, and cooperates with

the hypervisor to virtualize the underlying hardware. In other words, the guest uses custom drivers that use a direct path for communicating with the hypervisor. Taking the example of VirtIO-Net [4], the paravirtualized network device in KVM, the burden on the hypervisor is reduced, and some optimizations, such as the VHost-Net support, attempt to reduce the number of system calls thus improving latency. Unfortunately the packet journey is not reduced as packets flow through virtual bridges and virtual TAP devices, and twice through the operating system.

Recently, network equipments manufactures have introduced technologies for enhancing networking on virtualized environments. For example, some Intel server class network interface cards (NICs) implement the Intel VMDq (Virtual Machine Device Queues) technology [8]. To abstract the network device and share it across multiple VMs, the hypervisor has to implement a software network switch, which usually introduce severe performance penalties. As shown in Figure 2, VMDq-aware NICs implement all this in hardware thus preventing the in-software switching overhead. By combining this technology with optimized paravirtualization techniques, it is possible to achieve high networking performance in VMs [9] [10]. Unfortunately packet capture cannot benefit from it, as VMDq basically partitions the NIC into several virtual NICs but it does not feature mechanisms for:

- Accelerating packet capture.
- Capturing packets from all virtual queues, rather than from just the one assigned to the VM.

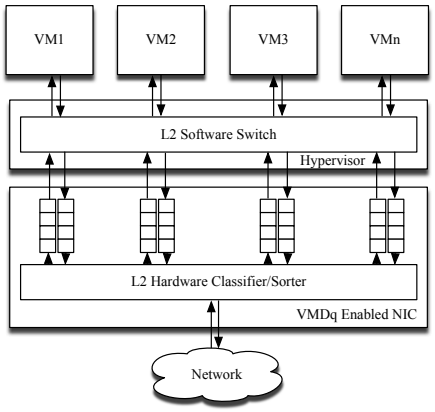

Figure 2. Intel VMDq Technology

Along the path of VMDq, the Single Root I/O Virtualization technology [11] is a way of sharing a device in a virtualized environment, bypassing the hypervisor involvement in data movement. As depicted in Figure 3, with this technology a single Ethernet port can be configured by the hypervisor to appear as multiple independent devices, each one with its own configuration space. The hypervisor assigns each Virtual Function (VF, lightweight PCIe functions) to a VM, providing independent memory space and DMA streams. PCIe PF (Physical Functions) is the physical device seen by the hypervisor on the host. Memory address translation technologies based on IOMMUs [12] [13], such as Intel VT-d [14] and AMD IOMMU, provide hardware assisted techniques to allow direct DMA transfers bypassing the hypervisor, while keeping isolation between host and VMs [30]. There are several projects following the same approach of the SR-IOV, with different designs of self-virtualized devices for direct I/O [15] [16] [17]. All these techniques represent good solutions for common connectivity but, besides efficiency, they do not provide assistance in accelerating packet capture.

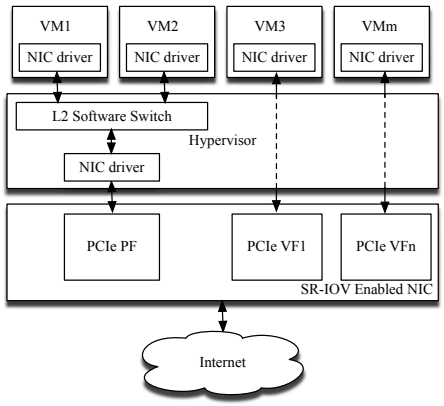

Figure 3. SR-IOV Technology

2.2 Packet Filtering

The previous section has shown various alternatives to implement efficient networking inside VMs. With paravirtualization, packets pass through virtual bridges and virtual TAP devices if hardware support such as VMDq is not present. Using VMDq-aware network devices or self-virtualized devices [11] [15] [16] [17], packet filtering still happens in software as hardware filtering used by VMDq usually provides only MAC-address filtering and thus in-NIC bridging that is disabled when the NIC is set in promiscuous mode. The result is that both with/without VMDq, packet filtering has to happen in software as modern NICs do not provide assistance for filtering packets when used in virtualized environments.

Early packet filtering is necessary to prevent packets from being discarded late in their journey to the VM, after that they have passed through several components. This is because filtering on guest OS means that the packet has already reached the VM and thus that in case of packet not satisfying any filter it would result in wasted CPU cycles.

Another side effect of in-VM filtering, is that all received packets need to be copied to each VM, whereas in case of early filtering, just the packets matching the filters will be forwarded to VMs.

For specific application domains such as lawful interception, filtering at the VM level represents a major performance issue. This is because network operators

usually provide a shadow copy of all packets flowing through a link where several hundred users are connected, but only a small portion of them belong to users that need to be intercepted; the result is that most packets will be discarded except those belonging to the targets (i.e. those users that are being intercepted). This problem is even more visible if the same input traffic needs to be passed to various VMs, each performing a different type of analysis. For this reason early packet discard on the physical machine is very important as it avoid VMs to be over flood with packets that will be discarded later on. Lawful interception is a good example where the physical host receives all packets, and it both filters and dispatches to the VMs only those packets matching the individual filters set by each VM.

A possible solution to the problem is to replace virtual bridges with virtual switches such as Open vSwitch [26] [27]. Open vSwitch implements standard Ethernet switching, while providing high flexibility with full control on the forwarding table by implementing a superset of the OpenFlow protocol [28]. However, as mentioned in [26], the problem is that these switches cause high CPU utilization when switching packets, so latency and overhead of the paravirtualization solutions increases. On the contrary, hardware based OpenFlow switches [42], can potentially offload the VMs from packet filtering. This approach allows the load on VMs introduced by packet filtering to be reduced [29] [41], but limits the flexibility offered by virtualized environments (e.g. the migration of VM across physical machines is compromised) and limits filtering to what is offered by the switches.

In order to implement efficient VM filtering, it is necessary to discard packets in the physical machine as close as possible to the physical NIC. This way, only packets matching the filtering rules will continue their journey to the VMs, whereas others will be dropped immediately as depicted in Figure 4. In a nutshell, early packet filtering in the context of network monitoring is a key requirement for achieving a high performance.

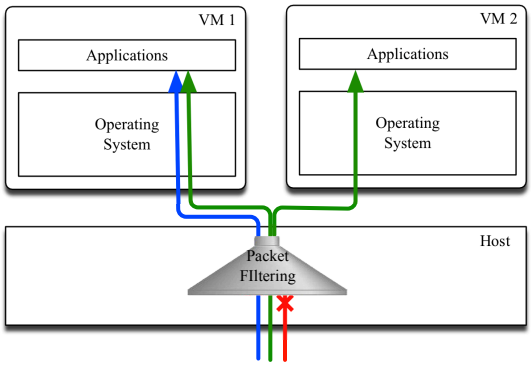

Figure 4. Early Packet Filtering

Discarding packets on the physical machine leveraging on the PF_RING filtering support would be very useful in security when IDS (Intrusion Detection Systems) are used.

We have developed a PF_RING module, part of the PF_RING code distribution, for the popular snort IDS named PF_RING DAQ (Data AcQuisition library) [46]. This module is responsible for receiving packets from PF_RING and dispatching them to snort. For each packet, snort emits a verdict that can be drop/pass and also white/black-list. In the latter case, it would be very desirable to have a snort instance running on a VM able to set filtering rules inside the PF_RING kernel module running on the physical host by means of this DAQ module. The advantage is that unwanted packets/flows are discarded by PF_RING and never hit the VM. This would be a great advantage of early packet discarding that we want to offer inside the vPF_RING framework.

2.3 PF_RING and Operating System Bypass

PF_RING is a kernel-based extensible traffic analysis framework, that significantly improves the performance of packet capture. It reduces the journey of captured packets from wire to user-space, and features a flexible packet filtering system and an extensible plugin-based architecture for adding new functionality at runtime.

PF_RING can use both vanilla Linux drivers and PF_RING-aware drivers. The main difference is that the latter can push captured packets directly to the PF_RING kernel module thus reducing the packet journey with respect to Linux native packet capture. PF_RING supports a rich set of packet filtering mechanisms that allow users to specify actions (e.g. dump a packet to disk), whenever incoming packets match the filtering rules. It also supports hardware filtering and packet steering capabilities when packet capture happens on modern network adapters, such as the Intel 82599 [21] and the Silicom PE210G2RS [22]. With those adapters packets are filtered inside the NIC without any assistance from the main CPU as it happen with software packet filtering. Hardware and software packet filtering, allow efficient applications processing packets directly inside the kernel to be easily implemented. Kernel based packet processing is more efficient than user-space packet processing as packets do not have to be copied from kernel to user-space in case they don't match any configured filter.

The PF_RING framework provides a user-space library that exposes an easy-to-use API for implementing monitoring applications. As depicted in Figure 5, through this library, ring buffers on which PF_RING is based, are directly mapped from kernel-space into user-space by using *mmap()*, reducing overheads and the number of data copies.

When an application wants to read a new packet, the library checks the ring:

- If there are new packets available, they get processed immediately.
- When no packets are found, a *poll()* is called in order to wait for new packets. When the *poll()* returns, the library checks again the ring for new packets.

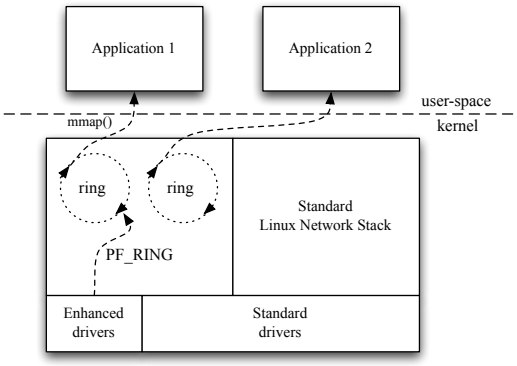

Figure 5. PF_RING Architecture

In order to reduce the number of *poll()* calls and thus a continuous *poll()*-wake up-*poll()* transition, the PF_RING kernel module implements a dynamic polling mechanism that can be configured by packet capture applications. The *poll()* system call returns when at least X packets are available, where X can range from one to several thousand, or when the call times out, usually this is set to 10 msec. This mechanism allows CPU cycles to be preserved for those applications that do not need to process packet immediately, but it also enables low-latency applications to be implemented setting X to one.

As described above, the approach followed by PF_RING is to create a straight path for packets bypassing the operating system standard mechanisms by means of a memory-map from kernel-space to the address space of the monitoring application. With this solution, system calls other than the *poll()* are completely avoided. The operating system bypass approach is adopted in many research projects [23] [24] as well as commercial products such as those manufactured by companies such as Endace and Napatech, most of all in areas requiring intense I/O activity, and where low latency and high bandwidth are vital.

2.4 Hypervisor Bypass

The hypervisor involvement in all the VM I/O accesses ensures isolation and system integrity, but it also leads to longer latency and higher overhead compared to native I/O accesses in non-virtualized environments, thus becoming a bottleneck for I/O intensive workloads.

In this paper, we propose a model that extends the PF_RING's operating system bypass approach to the context of virtual environments, thus creating a direct mapping between the host kernel-space and the guest user-space. This approach aims to perform operations that require intensive workloads such as packet capture using a direct VM-to-physical host path, without the involvement of the hypervisor except during the setup phase. It is worth to note that as in native PF_RING, this does not mean that the hypervisor is completely bypassed in all operations, but just for those that are computationally expensive such as packet capture while it is still used for implementing packet polling. In this view, hypervisor overhead does not affect packet capture performance because this component is fully bypassed when packets are read from the PF_RING ring sitting on the host.

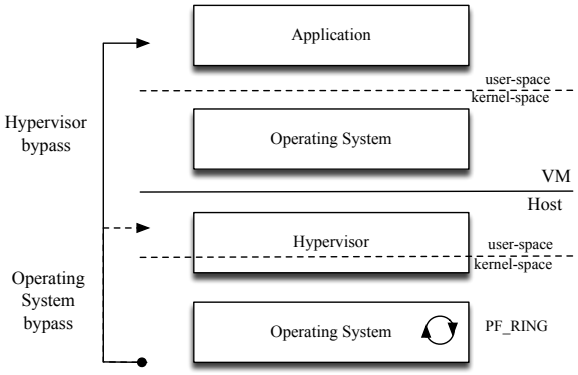

Figure 6. Hypervisor Bypass

The hypervisor-bypass approach is not a novel idea: self-virtualized devices for direct I/O, such as SR-IOV [35] capable ones, are an example. There are also some studies in the context of High Performance Computing (HPC) [6] [7] that have demonstrated that the hypervisor-bypass method can represent a very good solution in order to remove bottlenecks in systems with high I/O demands, especially those equipped with modern low latency and high bandwidth network interconnects.

3. vPF_RING DESIGN PRINCIPLES

In this section, we present the design and implementation of Virtual PF_RING (vPF_RING), that is based on vNPlug, a framework implementing the hypervisor-bypass, also developed by the authors. Although the work presented on this paper addresses general issues that are not dependent on a specific virtualization framework, the authors focus only on KVM as it leverages Linux kernel capabilities, such as scheduling and memory management. KVM is a small and relatively simple software, present out-of-the-box on the majority of Linux distributions, contrary to other similar solutions such as Xen that is not integrated into the mainstream kernel. Proprietary solutions such as VMware [38], which is widely accepted in the industry, have not been taken into account due to their license restrictions and because of the source code not being open and available [20].

KVM implements a kernel-based virtual machine on top of the Linux kernel, and exploits a modified version of QEMU [25] for emulating I/O devices. Implemented as kernel module, KVM supports native code execution by exploiting hardware virtualization extensions such as Intel VT and AMD Secure Virtual Machine. Common tasks, such as scheduling and memory management, are delegated to the Linux kernel. VMs run as conventional user-space

processes making Linux unaware of dealing with a virtual system.

vPF_RING, described later in section 3.2, does not strictly depend on KVM but it mostly relies on PF_RING APIs. Instead, as described in the next section, the vNPlug framework has been designed on top of KVM for implementing the hypervisor-bypass approach (mapping memory, exchanging control messages, notifying events). Porting vNPlug to another hypervisor such as Xen, requires a complete code rewrite, contrary to the vPF_RING code that should not be modified.

3.1 vNPlug Framework

The vNPlug framework exploits the hypervisor-bypass approach for achieving high packet capture performance in virtualized environments. It has been designed to be general enough for being used by every monitoring application and not just vPF_RING. For instance, the Linux native socket type PF_PACKET is quite similar to PF_RING as both use memory mapped buffers to exchange packets between kernel and user-space. Porting PF_PACKET on top of vPF_RING-vNPlug is thus just a matter of time as it does not seem to have any technical challenge being the two socket types pretty similar.

The framework follows a paravirtualization-like design, guests are aware of being virtualized and consequently the architecture is logically split in a guest and an host side block.
The framework is logically divided into two main components. The first component, *vNPlug-Dev*, is responsible for:

- Mapping memory between the host kernel-space and the guest user-space.
- Implementing an efficient event notification that is necessary for VM/Host communications.

The second component, *vNPlug-CTRL*, is responsible for coordinating the host and guest side of applications by means of a control communication channel. The channel is required, for example, when an application needs to instrument its host-side back-end for filtering specific packets.

As can be seen, applications built on top of the framework can access physical host resources that are usually not available in virtualized environments. In case of vPF_RING, applications executed under VMs can capture packets not only from VM's network interfaces, but also from physical host interfaces. This feature can be offered by building vPF_RING on top of the vNPlug framework.

vNPlug is implemented as a QEMU patch on the host side, and a Linux kernel module (vnplug.ko), based on both vNPlug-Dev and vNPlug-CTRL components, on the guest OS.

3.1.1 vNPlug-Dev

The original PF_RING maps kernel ring buffers to user-space via memory-map. vNPlug-Dev allows to further memory-map these virtual memory areas to virtual machines. The initial memory-mapping happens through the hypervisor, whereas all packets are exchanged directly between the VM and the PF_RING sitting on the host without any hypervisor support. This mapping is performed dynamically attaching additional blocks of memory via virtual PCI devices whenever a vPF_RING is created. Inside the VM, these memory regions can be accessed by *ioremap()*, and mapped in virtual memory areas via the vnplug.ko kernel module that creates character devices that can be memory-mapped. Figure 7 depicts the vNPlug-Dev architecture.

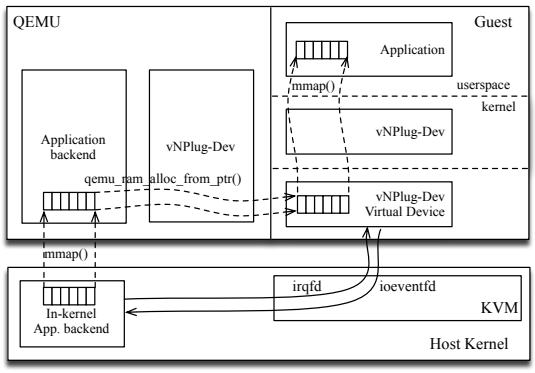

Figure 7. vNPlug-Dev Architecture

vNPlug-Dev is dynamic by design as it allows dynamic memory-mapping to take place by means of virtual PCI devices. Therefore, the number of rings is not limited as it happens, for instance, in BSD systems where packets are captured from a limited number of statically allocated capture devices (/dev/bpfX).

The PCI hotplug support allows devices to be dynamically attached and removed from a running system. Even if hotplug is rarely used in practice, basic hotplug support is provided by the majority of modern operating systems, making hot-plugged devices immediately usable with limited effort. By exploiting the hotplug, it is possible to dynamically attach memory mappings to guests whenever necessary, making vPF_RING a very flexible system that does not have any limitation in terms of functionality and flexibility with respect to native PF_RING.

The event signaling functionality of the framework takes advantage of the *irqfd* and *ioeventfd* supports of KVM in order to provide a two-way notification mechanism, from host-to-guest and from guest-to-host. Both of them are based on the *eventfd* file descriptor for event notification, that is quite powerful yet flexible as it can be used from both user-space and kernel-space in order to signal/wait events.

Using the *irqfd* support it is possible to send interrupts to the VM without passing through the QEMU process, which

is responsible to emulate the device on which interrupts are dispatched. In fact, since virtual interrupts are injected to the guest via KVM, the *irqfd* support allows the latter to directly translate a signal on an *eventfd* into an interrupt, thus ensuring efficiency. At the same time, the MSI (Message Signaled Interrupt) support ensures flexibility, by using multiple vectors that simplifies the notification mechanism when several events are required. On the guest side, the framework has been inspired by the *eventfd* approach that uses a blocking *read()* on a character device for notifying user-space applications that an interrupt has been received.

ioeventfd is used to register arbitrary addresses of a MMIO (Memory-Mapped I/O) region belonging to a virtual device, along with a unique value and an *eventfd*. On the guest side, these MMIO regions are mapped in user-space. Whenever the guest OS writes a value to such MMIO region, if the written value matches the registered value, then an event is triggered on the corresponding *eventfd*. This mechanism is quite efficient as it allows a lightweight exit (long enough to signal an *eventfd* in kernel-space by means of a KVM service routine), while a normal I/O operation on an emulated virtual device requires a costly VM exit.

3.1.2 vNPlug-CTRL

The component implements a message based communication channel that allows control messages to be exchanged between the guest side of the monitoring application and its back-end. For instance it can be used by an application to request the back-end to setup a new memory-mapping, or to filter packets.

The vNPlug-CTRL component has been introduced for having a control channel totally independent from network communications, and, as such, not susceptible to unintentional network configuration changes.

As depicted in Figure 8, the vNPlug-CTRL component implementation is based on the VirtIO interface for paravirtualization that is efficient and ensures low response times, but required a little more effort at development time compared to a network communication implementation. The two-way communication channel over VirtIO uses two

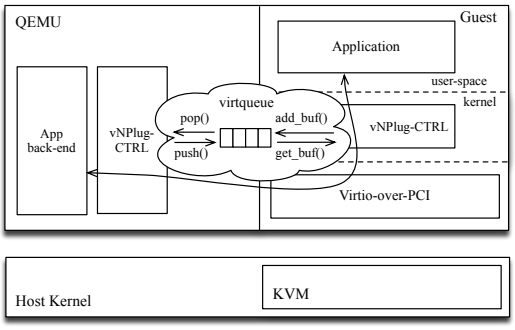

Figure 8. vNPlug-CTRL component.

virtqueue's, one for host-to-guest messages and one for the opposite direction. In order to send and receive messages from the guest user-space, the framework exposes common file operations (*read* and *write*) on a character device.

Through this communication channel, the framework routes messages between the host-side and guest-side of applications. As multiple applications are supported, each with multiple virtual devices, the framework uses a minimal and yet efficient protocol stack, depicted in Figure 9. At the bottom of the stack, the VirtIO transport mechanism takes place, providing a two-way point-to-point communication channel between the two sides of the framework: guest and host side. At the second layer, a framework-level header allows the framework to distinguish between messages addressed to itself and those addressed to an application. At the third layer, an application-level header allows the framework to identify the application to which such message has to be delivered. From the fourth layer on, all is managed by the application, in order to identify internal operations and address virtual devices.

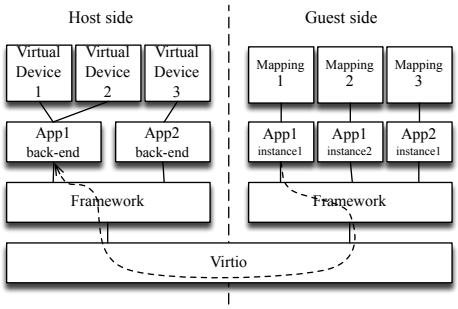

Figure 9. vNPlug-CTRL message routing

3.1.3 vNPlug API

In order to simplify the development of monitoring applications, the framework provides a simple API, that implements an abstraction layer on top of the implemented functions. Framework's components get abstracted through two subsets of the interface: the host side API and the guest side API.

The main features the interface provides:

- Host Side
  - Registration and unregistration of the application back-end.
  - Control messages reception and transmission.
  - Virtual devices, for memory-mapping, creation and tear-down.
- Guest Side
  - Control messages transmission and reception.
  - Shared memory-mapping and unmapping in the virtual address space of the application.
  - Event signaling/waiting functionalities.

3.2 vPF_RING

vPF_RING is an extension of PF_RING for virtualized environments built on top of vNPlug. The design of original PF_RING lent itself particularly well to be adapted to the vNPlug framework. In fact, on the host side, it only needed a few enhancements, keeping both the kernel module and the user-space library fully backward-compatible with the original version. As the PF_RING library uses memory-mapping for exporting the packet capture ring from kernel-space into user-space, the virtual memory address returned by *mmap()* can be used by the framework to map it into the guest. In a nutshell, PF_RING is responsible for making this memory area available to the guest user-space.

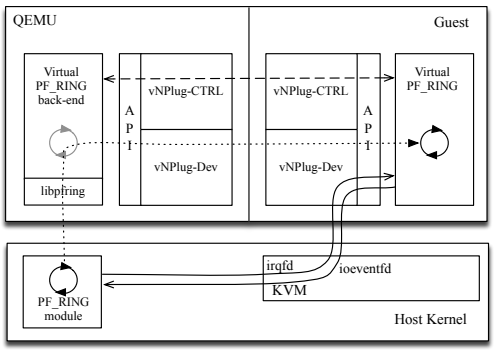

Figure 10. vPF_RING design

The two-way event signaling support of the framework has been used for replacing the *poll()* calls used by PF_RING applications for being waken-up when new incoming packets are available. When an application on the guest-side has to read a new packet, but no packets are ready to be read, the library on the guest-side informs the host side. This way, the host-side knows that if there are unread packets, or when a new one arrives, it has to send an interrupt to the guest-side that is waiting for packets. Furthermore an algorithm similar to the adaptive sleep of the PF_RING native library is used, in order to avoid many *poll*-equivalent calls.

A new and thin library has been created on the guest-side for:

- Translating each call to the PF_RING library into control messages over the communication channel provided by the framework.
- Memory-mapping and event signaling/waiting mechanisms just described.

The vPF_RING back-end on the host-side, is also responsible of translating guest-to-host control messages into calls to the PF_RING library. It allows monitoring applications running on guests to:

- Read packets from kernel via memory-map and not through read() system calls as it happens with VirtIO.
- Access host network interfaces in addition to guest network interfaces.
- Setup packet capture filters directory on the host PF_RING, thus implementing early packet filtering.
- Seamlessly develop applications, that can run unchanged both on physical or virtualized environments, easing the move towards a virtualized monitoring environment.

In a nutshell vPF_RING has been designed to be transparent to application developers, both in terms of features and packet capture speed. The only visible difference is the device name from which packets are captured. With native PF_RING it is possible to capture packets just from physical interfaces. Using vPF_RING, it is possible to capture packets from both the VM's virtual Ethernet device, and the physical host interface. In the former case, vPF_RING operates as PF_RING when capturing packets from a host adapter (in this case from the VM virtual adapter). In the latter case, vPF_RING is not capturing from the VM's interface but from the host's physical interface. As vPF_RING's API is unchanged with respect to PF_RING, a special device naming convention has been used in order to instruct the framework to capture packets from the host interface. This is because host interfaces are not visible to the VM via standard Linux commands such as ifconfig, and also because interface name present on both the VM and host might be the same (e.g. eth0). For this reason in vPF_RING the following naming convention has been used: interface names with a "host:" prefix indicate host interface. For instance when a VM opens "eth0" it means that it wants to open the virtual VM eth0 interface; instead "host:eth0" means the eth0 physical host interface.

vPF_RING honors all PF_RING capture extensions. For instance applications can capture traffic from a specific RX queue of a multi-queue adapter when using PF_RING-aware driver [32], and specify filtering and packet steering rules in hardware on adapters such as Intel 82599 [21]. On one hand, these are interesting features to have as for instance a VM having to analyze HTTP traffic, can capture traffic on RX queue X on which it has configured a hardware filter that sends to such queue only HTTP packets. On the other hand, like most kernel bypass technologies (e.g. the same PF_RING), must be used properly as they circumvent some protection mechanisms, such as the insulation of the VM from host environment.

4. vPF_RING VALIDATION

vPF_RING validation and performance evaluation tests have been performed on a simple test bed, where an IXIA 400 traffic generator has been used for sending packets to a server powered by an Intel Xeon X3440, running Linux kernel 2.6.36 and equipped with a dual Intel 82576 Gigabit Ethernet controller. The IXIA 400 traffic generator is connected to the server via the two gigabit ports, and can generate network traffic at configurable rates, including the wire-rate, on both port regardless of the packet size. For 10

Gigabit tests we have used a home-grown tool named *pfsend* with PF_RING DNA (Direct NIC Access) [47] for reproducing traffic at wire speed previously captured on a network backbone. This has allowed us to test vPF_RING under various conditions and with both synthetic and real network traffic. For the tests described later on this section, have been used forged packets in order to evaluate this work with different packet rates and sizes.

The performance of vPF_RING has been compared with the performance of native PF_RING 4.6.x running on a physical (non virtualized) host and PF_RING running on a virtual KVM environment (using the VirtIO-Net support with the VHost-Net optimization). vPF_RING performance has also been positionedagainst VMware ESXi (using VMXNET 3, the latest version available of the VMware paravirtualized network device). All the VMs used during the evaluation have a single-core virtual CPU and also run Linux kernel version 2.6.36.

The device driver used on the server on the host-side is the *igb*, developed by Intel, which is included in the Linux kernel. It is worth to remark that, although PF_RING supports PF_RING-aware optimized drivers to bypass the standard operating system's mechanisms, we decided not to use them in order to evaluate our work on the worst case (i.e. without any packet capture acceleration exploiting specific network cards features). This is because we want to compare native VirtIO-Net agains vPF_RING, without accounting any vPF_RING performance benefit due to these optimized drivers.

Before describing the evaluating results, it is important to understand how the packet size affects the benchmarks. This parameter is relevant because the maximum packet rate that can be injected on a link depends on the packet size. As shown in Table 1, at wire-rate, small packet sizes corresponds tohigher packet rates. The packet capture performance is affected by the packet rate, which can be as high as 1.4 Million of packets per seconds (Mpps) when the packet size is 64 bytes (minimum packet size) on Gigabit links, 14.880 Mpps on 10 Gigabit.

Table 1. Maximum Packet Rates

Line Speed	Rates Per Frame Size (Kpps)				
	64 Byte	128 Byte	256 Byte	512 Byte	1024 Byte
1 Gigabit	1488	844	452	234	119
10 Gigabit	14880	8445	4528	2349	1197

Another aspect worth to mention, is that with vPF_RING it is possible to use efficient packet filtering techniques within the host (in kernel-space or even in hardware), to further increase the performance. In fact, through the efficient communication channel provided by the vNPlug-CTRL component, vPF_RING is capable to instrument the PF_RING module for setting a variety of efficient filters. However, as we are interested in evaluating our work in the worst case scenario, packet filtering has not been used.

Benchmarks have been done using *pfcount*, a simple packet capture application implemented on top of the PF_RING API. The application captures packets, updates some statistics, and then discards packets without doing any further processing.

In the first test we evaluate the packet capture performance when a single instance of pfcount processes the traffic injected at wire rate with different packet sizes on a single Gigabit link.

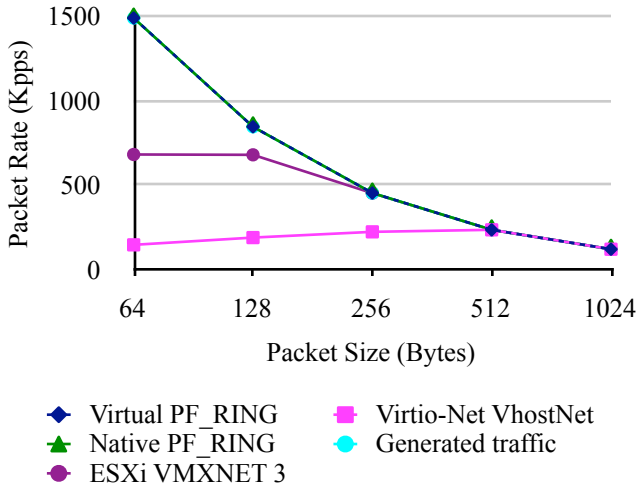

Figure 11. Packet Capture Rate (1 Gbit)

In Figure 11 we show that vPF_RING, similar to PF_RING on a native environment, is able to process packets at wire-rate (without packet loss), for every packet size, up to the maximum rate (1.488 Mpps per port).

From the same figure we can observe that by using PF_RING in a virtual environment with the VirtIO-Net support (i.e. without the assistance of our framework), it is possible to efficiently capture without packet loss only medium/large packets, when packet rates are not more than a few hundred Kpps. In fact, with small packets severe packet drops can be observed. Results are slightly better when using PF_RING on a VMware ESXi virtual environment, but we can still notice severe packet drops for high rates.

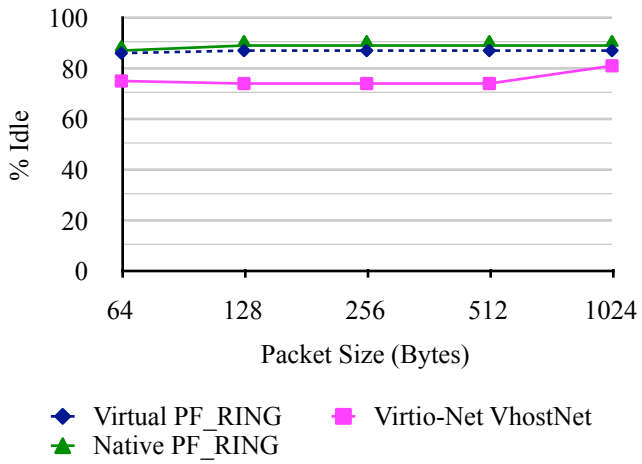

Figure 12. Idle CPU % During Capture on Host as Reported by top (1 Gbit)

In addition to packet capture, we evaluate the percentage of CPU idle time as reported by the *top* command utility. In this way, we can have an indication of the free CPU cycles available for packet processing. Figure 12 shows that vPF_RING can cope with high packet rates while keeping the CPU relatively idle, almost the same percentage as the native solution. Instead, with the VirtIO-Net support, there is an higher overhead even if fewer packets per second are processed.

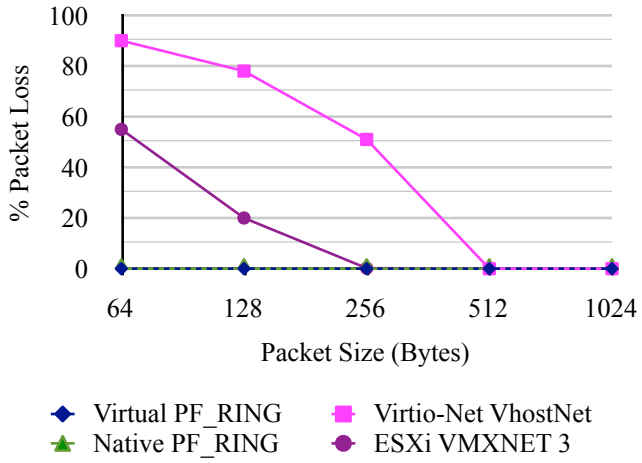

Figure 13. Packet Loss Percentage (1 Gbit)

Figure 13 depicts the packet loss percentage that pfcount reports when using different capture mechanisms. The result highlights that both vPF_RING and PF_RING allows packets to be captured without observing any packet loss for all packet sizes, including the minimum packet size corresponding to the highest packet rate (1.4 Mpps for 64 byte packets). On the contrary, when using VirtIO-Net, the packet loss percentage is significant (as high as 90% in the case of 64 bytes packets), making it unsuitable for applications where 100% packet capture is required. A lower packet loss percentage can be observed when VMware ESXi is used; however also this solution cannot guarantee no packet loss.

A second test has been performed to evaluate the performance when two instances of the *pfcount* application, running on the same VM, process the traffic injected on two different Gbit interfaces.

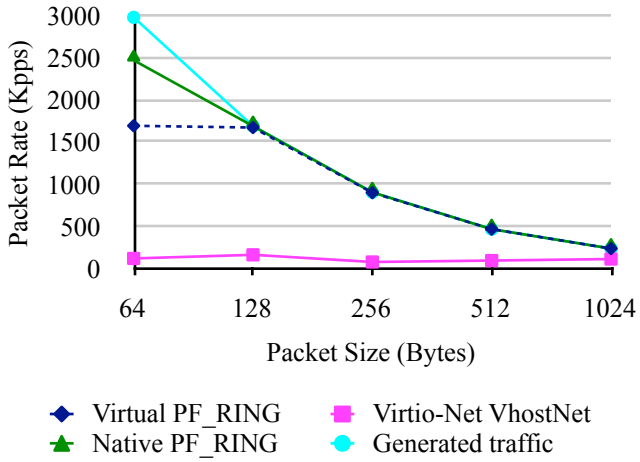

Figure 14. Total Captured Packets By Two pfcount Instances Running On The Same VM

Figure 14 shows the aggregated packet capture rate that is achieved by running the two *pfcount* instances. Both vPF_RING and PF_RING are able to process up to nearly two million packets per second without packet loss (with an average of one million per instance). When the packet rate on the wire increases further (with 64-byte packets at wire-speed) both capture mechanisms lose packets. However native PF_RING processes about half a million more than vPF_RING.

As the virtual machine where the two instances of pfcount have limited CPU resources, this result does not necessarily mean that vPF_RING offers a worse scalability than the native PF_RING. In fact, while the two instances of *pfcount* of the native solution can run concurrently on different cores of the same processor, we know that a virtual CPU, where the two application instances of the virtual solution are scheduled on, is itself scheduled as a normal thread by the host operating system.

Regarding the virtual solution without the framework, using the VirtIO-Net support, performance are similar or even worse to the previous, with up to one hundred thousand packets per second processed by each application instance. The conclusion is that even with large packets, packet loss is pretty severe.

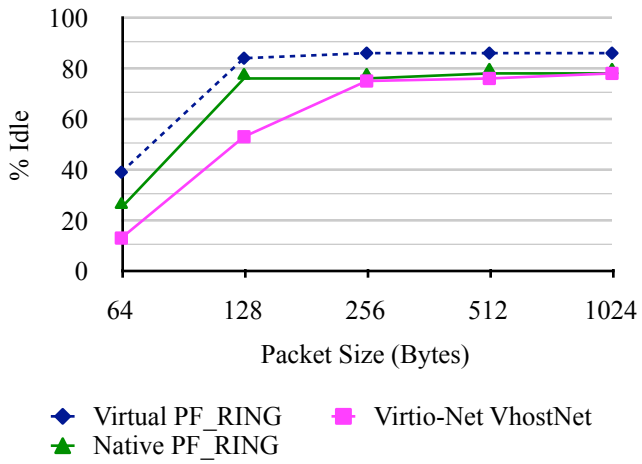

Figure 15. Idle CPU % When Two pfcount Instances Are Running On The Same VM

Figure 15 depicts the percentage of CPU idle time, and it confirms that vPF_RING keeps the CPU relatively idle, even more that native PF_RING. This is because the native PF_RING is more efficient than the virtual version, thus it consumes packets more quickly hence calls *poll()* much more often that contributes to reduce the idle time. Instead the solution based on VirtIO-Net requires more CPU time even with a very low percentage of captured packets.

Another test has been conducted for evaluating the performance of two instances of the application, each one processing one Gigabit of traffic on a different interface, but this time each running on a different VM.

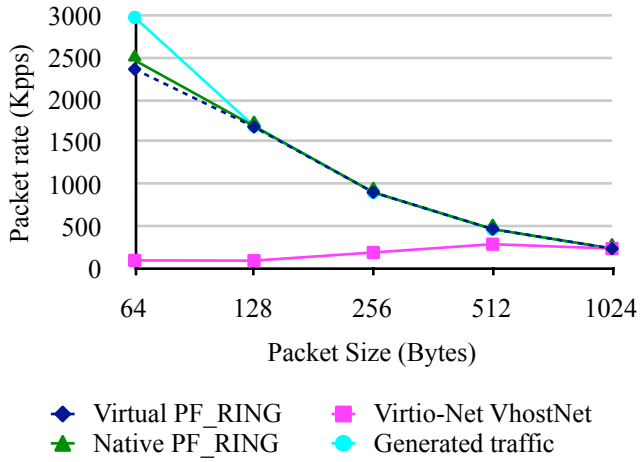

Figure 16. Total Captured Packets by Two pfcount Instances Running on Different VMs

As shown in Figure 16, the total number of captured packets by both application instances has that same trend as in the previous test. The only difference is that in this case for 64-byte packets the capture rate of vPF_RING is basically the same of the native PF_RING. This, once again, confirms our hypothesis about scalability. In fact, in this case we have two virtual CPUs scheduled on the host, one for each VM, and on each virtual CPU an application instance is scheduled.

The solution based on VirtIO-Net, this time, seems to scale for large packets but, at high rates, performance is similar to the one observed in the previous tests.

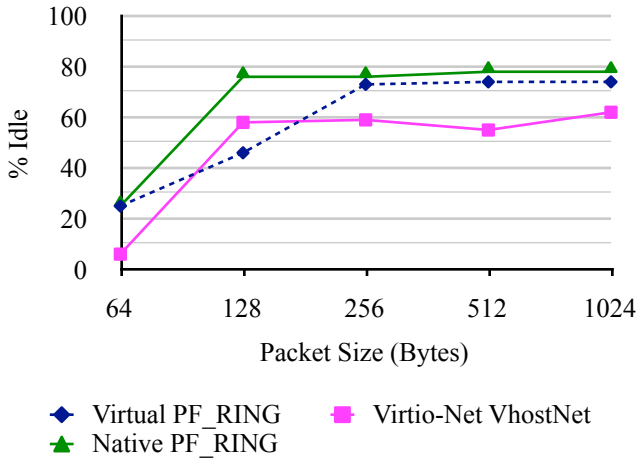

Figure 17 Idle CPU % When Two pfcount Instances Are Running On Different VMs

Figure 17 shows the percentage of CPU idle time. As one would guess, vPF_RING overhead is higher than the native PF_RING. The solution based on VirtIO-Net still requires many more CPU cycles, even if its packet capture performance is lower.

Another series of tests has been performed in order to compare the packet capture performance offered by vPF_RING when capturing from a 10 Gigabit link, to the performance provided by the native PF_RING. *pfsend* on top of PF_RING DNA has been used to generate traffic at wire speed. An Intel 82599 based Gigabit Ethernet interface has been used as a capture device. The server used is still an Intel Xeon X3440 running Linux kernel 2.6.36. The device driver used for these tests, on the host-side, is a PF_RING-aware version of the *ixgbe*, which is able to copy packets directly to PF_RING my means of Linux NAPI packet polling.

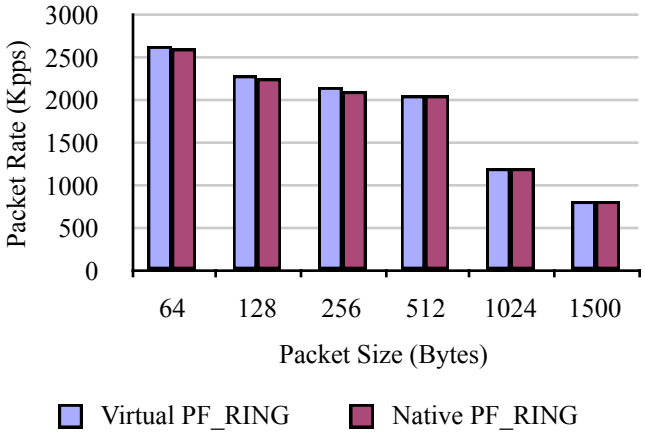

Figure 18. Packet Capture Rate (10 Gigabit)

In the first of these tests, we evaluated the performance with a single application instance. Figure 18 shows that vPF_RING is able to match the packet capture performance offered by the native PF_RING.

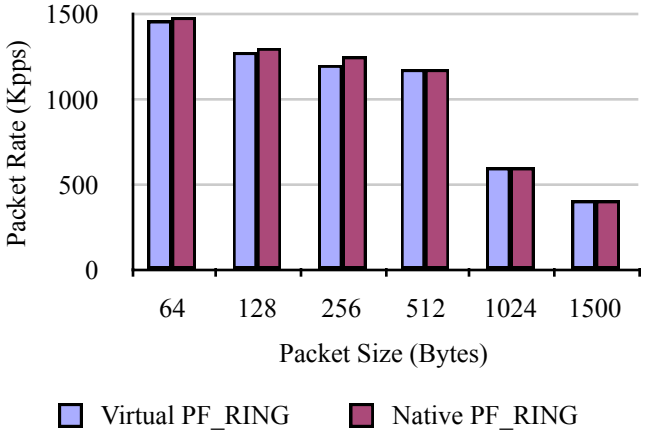

Figure 19. Captured Packets By Each Of The Two pfcount Instances Running On Different VMs (10 Gigabit)

A second test has been performed to evaluate the scalability, with two instances of *pfcount* capturing packets from the same interface, balancing the traffic across applications by means of RSS (Receive-Side Scaling) queues. In the virtual case, each *pfcount* instance is running on a different VM. As shown in Figure 19, also in this case, packet capture performance offered by vPF_RING is close to the one offered by the native PF_RING.

In order to further evaluate the scalability, another test has been conducted with four instances of *pfcount*. As in the previous test, the *pfcount* instances capture packets from

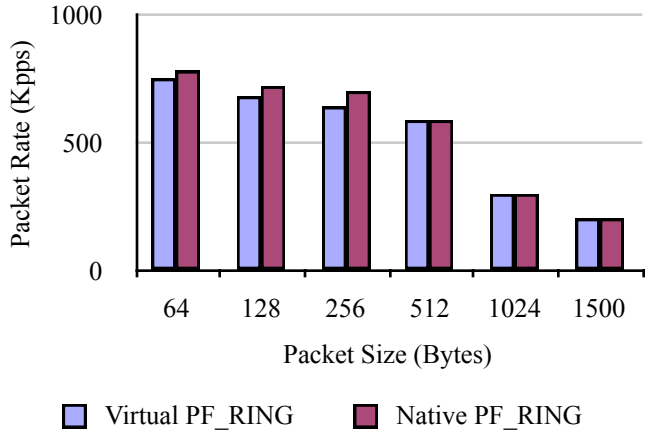

Figure 20. Captured Packets By Each Of The Four pfcount Instances Running On Different VMs (10 Gigabit)

the same interface. As depicted in Figure 20, vPF_RING offers packet capture performance comparable to the one provided by the native PF_RING.

In summary using vPF_RING has no performance penalty relative to native PF_RING (Figures 11,12). Relative to PF_RING in a VM:

- vPF_RING is more than an order of magnitude faster with respect to the performance achieved by vanilla KVM. This means that thanks to vPF_RING it is finally possible to effectively perform traffic monitoring inside KVM-based VMs.
- For all packet sizs, vPF_RING and PF_RING have comparable performance (Fig 14, 18).

5. OPEN ISSUES AND FUTURE WORK

The work described on this paper is an efficient and flexible solution to effective packet capture on VMs. Nevertheless there are a few areas where extra work is needed.

The main issue is live VM migration, as the hypervisor does not have knowledge of the resources allocated by the applications. This is in contrast to traditional device virtualization approaches, where the hypervisor is involved and it can suspend all the operations when live migration starts. While developing the framework we mostly focused on achieving high packet capture performance and we did not consider additional virtualization features, such as live migration. In the future we plan to address the issue for further increasing the flexibility offered by our solution.

Furthermore, it would be interesting to perform more detailed tests, look for further performance improvements, and evaluate the framework on VMs with multiple virtual CPUs investigating on scheduling and resource management.

6. FINAL REMARKS

In the past few years there have been many efforts to improve network performance on VMs, both with hardware and software solutions. However, none of the available solution addresses the problem of using VMs for high-performance network monitoring.

This paper used a well-known approach named hypervisor-bypass, which allows packets to follow a straight path from kernel to VMs, thus avoiding per-packet overhead due to the hypervisor and system calls. This mechanism has been successfully applied for implementing vPF_RING, a kernel-based extensible traffic analysis framework developed by the authors. The validation phase has confirmed that it can drastically improve packet capture performance, often achieving packet capture rates and CPU usage close to those that can be obtained on bare hardware. This reducing the dropped packet rate up to 90% on Gigabit links with respect to preexisting open source software solutions, 55% with respect to commercial solutions such as VMware (or even more on faster links).

The outcome is that it is now possible to efficiently run multiple VMs on commodity hardware, each monitoring the same traffic for different purposes, without packet loss and with plenty of CPU cycles available for processing the captured traffic.

7. CODE AVAILABILITY

This work is distributed under the GNU GPL license and is available at the ntop home page http://www.ntop.org/products/pf_ring/vpf_ring/.

8. ACKNOWLEDGEMENTS

Our thanks to Silicom Ltd. that has greatly supported this research work and provided network equipment used during tests.

9. REFERENCES

[1] F. Baboescu and G. Varghese, Scalable packet classification, Proc. of ACM Sigcomm, 2001.

[2] S. McCanne and V. Jacobson, The BSD Packet Filter: A New Architecture for User-level Packet Capture, Proc. of USENIX Conference, 1993.

[3] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, kvm: the Linux virtual machine monitor, Proc. of 2007 Ottawa Linux Symposium, July 2007.

[4] R. Russell, VirtIO: Towards a De-Facto Standard for Virtual I/O Devices, SIGOPS Operating Systems Review, Vol. 42, Issue 5, July 2008.

[5] L. Deri, Improving Passive Packet Capture: Beyond Device Polling, Proc. of SANE 2004, 2004.

[6] W. Huang et al., A case for high performance computing with virtual machines, Proc. of the 20th annual international conference on Supercomputing, 2006.

[7] J. Liu et al., High performance VMM-bypass I/O in virtual machines, Proc. of USENIX annual conference, 2006.

[8] R. Hiremane and S. Chinni, Virtual Machine Device Queues: An Integral Part of Intel Virtualization Technology for Connectivity that Delivers Enhanced Network Performance, Intel Corporation, White Paper, 2007.

[9] K.K. Ram et al., Achieving 10 Gb/s using safe and transparent network interface virtualization, Proc. of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, 2009.

[10] J.R. Santos et al., Bridging the gap between software and hardware techniques for i/o virtualization, Proc. of USENIX 2008 Annual Technical Conference on Annual Technical Conference, 2008.

[11] PCI-SIG, Single Root I/O Virtualization and Sharing Specification, Revision 1.0, 2007.

[12] M. Ben-Yehuda et al., Utilizing IOMMUs for virtualization in Linux and Xen, Proc. of the 2006 Ottawa Linux Symposium, 2006.

[13] M.D. Hummel et al., Address translation for input/output (I/O) devices and interrupt remapping for I/O devices in an I/O memory management unit (IOMMU), US Patent 7'653'803, 2010.

[14] S. Muthrasanallur et al., Intel Virtualization Technology for Directed I/O, Intel Corporation, 2006.

[15] J. LeVasseur et al.,Standardized but flexible I/O for self-virtualizing devices, Proc. of the First conference on I/O virtualization, USENIX Association, 2008.

[16] H. Raj and K. Schwan, High performance and scalable I/O virtualization via self-virtualized devices, Proc. of the 16th international symposium on High performance distributed computing, ACM, 2007.

[17] J. Shafer et al., Concurrent direct network access for virtual machine monitors, Proc. of IEEE 13th International Symposium on High Performance Computer Architecture, 2007.

[18] Endace, OSm 4.2 vDAG (Virtualized DAG), http://www.endace.com/endace-operating-system-for-network-monitoring-osm.html, June 2011.

[19] L. Youseff at al., Paravirtualization for HPC Systems, Proc. of Workshop on Xen in High-Performance Cluster and Grid Computing, 2006.

[20] K. Adams and O. Agesen, A Comparison of Software and Hardware Techniques for x86 Virtualization, International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2006.

[21] L. Deri et al., Wire-Speed Hardware-Assisted Traffic Filtering with Mainstream Network Adapters, Proc. of NEMA 2010 Workshop, October 2010.

[22] Silicom Ltd., PCIe Packet Processor Server Adapter PE210G2RS, http://www.silicom-usa.com/default.asp?contentID=2144, 2010.

[23] A. Biswas, A High Performance Real-time Packet Capturing Architecture for Network Management Systems, Masters Thesis, Concordia University, 2005.

[24] L. Degioanni and G. Varenni, Introducing Scalability in Network Measurement: Toward 10 Gbps with Commodity Hardware, Proc. of IMC '04, 2004.

[25] F. Bellard, QEMU, a Fast and Portable Dynamic Translator, Proc. of the USENIX Annual Technical Conference, FREENIX Track, 2005.

[26] J. Pettit et al., Virtual Switching in an Era of Advanced Edges, 2nd Workshop on Data Center – Converged and Virtual Ethernet Switching (DC-CAVES), Sept. 2010.

[27] B. Pfaff et al., Extending networking into the virtualization layer, Proc. of HotNets, October 2009.

[28] N. McKeown et al., OpenFlow: enabling innovation in campus networks, ACM SIGCOMM Computer Communication Review 38.2 (2008),.

[29] A. Greenhalgh et al., Flowstream Architectures, Proc. of WowKiVS 2009 Conference, 2009.

[30] Intel Corporation, 82599 10GbE Controller Datasheet, Revision 2.3, April 2010.

[31] Intel Corporation and VMware Inc., Enabling I/O-Intensive Applications for Server Virtualization, White Paper, 2009.

[32] Luca Deri et al., High Speed Network Traffic Analysis with Commodity Multi-core Systems, Proc. of IMC 2010, November 2010.

[33] B. Plaff et al., Extending Networking into the Virtualization Layer, Proc. of 8th HotNets Workshop, October 2009.

[34] N. Chowdhury and R. Boutaba, Network virtualization: state of the art and research challenges, IEEE Communications Magazine, July 2009.

[35] Y. Dong at al., SR-IOV networking in Xen: architecture, design and implementation, Proc. of WIOV'08, 2008.

[36] S. Rixner, Network Virtualization: Breaking the Performance Barrier, ACM Queue Magazine, Vol. 6 Issue 1, Jan./Feb 2008.

[37] D. Unnikrishnan et al., Scalable network virtualization using FPGAs, Proc. of ACM FPGA '10, 2010.

[38] E. L. Haletky, VMware ESX Server in the Enterprise: Planning and Securing Virtualization Servers, ISBN 0132302071, 2008.

[39] J. Wiegert et al., Challenges for Scalable Networking in a Virtualized Server, Proc. of 16th ICCCN Conference, 2007.

[40] N. Niebert et al., Network Virtualization: A Viable Path Towards the Future Internet, Strategic Workshop, 2007.

[41] R. Sherwood et al. FlowVisor: A Network Virtualization Layer. Technical Report Openflow-tr-2009-1, Stanford University, 2009.

[42] R. Sherwood et al., Carving research slices out of your production networks with OpenFlow. ACM SIGCOMM Computer Communication Review, 2010.

[43] VMware Inc.,Configuration Examples and Troubleshooting for VMDirectPath, Technical Note, 2010.

[44] L. Deri, nCap: Wire-speed Packet Capture and Transmission, Proc. of E2EMON Workshop, 2005.

[45] L. Braun et al., Comparing and Improving Current Packet Capturing Solutions Based On Commodity Hardware, Proc. of. IMC '10, November 2010.

[46] Russ Combs, Snort 2.9 Essentials: The DAQ, http://vrt-blog.snort.org/2010/08/snort-29-essentials-daq.html, August 2010.

[47] ntop, PF_RING DNA, http://www.ntop.org/products/pf_ring/dna/, September 2011.