Università di Pisa
Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea Specialistica in Tecnologie Informatiche

Tesi di Laurea

Towards wire-speed network monitoring using Virtual Machines

Relatore:

Prof. Luca Deri

Controrelatore:

Prof. Gianluigi Ferrari

Candidato:

Alfredo Cardigliano

Anno Accademico 2009/2010

In every conceivable manner, the family is

link to our past, bridge to our future.

— Alex Haley

Dedicated to my parents.

Abstract

Virtualization is becoming a very common trend in the industry today. Improving utilization of power and hardware, reducing server administration costs, simplifying disaster recovery and increasing availability, Virtualization is sweeping away the trend of having one application running on one server.

On the other hand, Virtualization software must be very robust and efficient, without compromising performance. This is especially true for network monitoring applications, specially when single packet capture and analysis is required and running at wire-speed becomes crucial.

Furthermore, in the era of Cloud computing, Virtualization technology plays a key role. A non-intrusive, on-demand, remote network monitoring tool can represent an example of modern Cloud service.

This thesis discusses a method for speeding up network analysis applications running on Virtual Machines, and presents a framework that can be exploited to design and implement this kind of applications. An example of usage is also provided, with Virtual PF_RING, for validation and performance evaluation.

Sommario

La Virtualizzazione sta diventando una tendenza molto comune nell'industria di oggi. Migliorando l'utilizzo dell'energia e dell'hardware, riducendo i costi di amministrazione dei server, semplificando il ripristino in seguito a disastri e aumentando la disponibilità, la Virtualizzazione sta eliminando la tendenza ad avere una singola applicazione in esecuzione su un server.

D'altra parte, il software di Virtualizzazione deve essere molto robusto ed efficiente, senza compromettere le prestazioni. Ciò è particolarmente vero per le applicazioni di monitoraggio di rete, specialmente quando è necessario catturare ed analizzare ogni singolo pacchetto e diventa essenziale andare alla velocità del mezzo trasmissivo.

Inoltre, nell'era del Cloud computing, la Virtualizzazione gioca un ruolo chiave. Uno strumento di monitoraggio di rete a distanza, non intrusivo, su richiesta, può rappresentare un esempio di servizio Cloud moderno.

In questa tesi viene descritto un metodo per migliorare le prestazioni delle applicazioni di analisi di rete in esecuzione su Macchine Virtuali, e viene presentato un framework che può essere utilizzato per la progettazione e l'implementazione di applicazioni di questo genere. Viene dato anche un esempio d'uso, Virtual PF_RING, per la convalida e la valutazione delle prestazioni.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my supervisor Prof. Luca Deri for his support throughout this thesis, for having spent his valuable time helping me with tips and indicating the best direction to follow, for his enthusiasm, and for having believed in me since the very beginning. His efforts are highly appreciated. I could not have imagined having a better supervisor for my master thesis.

I would also like to extend my deepest gratitude to my family, and especially my parents, for the absolute confidence in me and for the constant moral (and economical) support during these years. Without their encouragement I would not have had a chance to be here. I am simply indebted to them.

Pisa, april 2011 *Alfredo Cardigliano*

Contents

1 Introduction **1**

1.1 Overview 1

1.1.1 A long packet journey 3

1.1.2 Packet filtering: the earlier the better 4

1.1.3 Limited network hardware support 4

1.2 Motivation 5

1.3 Scope of work 6

1.4 Requirements 6

1.5 Thesis outline 8

2 Related Work **9**

2.1 Virtualization 9

2.1.1 Virtualization technologies 10

2.1.2 Paravirtualized devices 11

2.1.3 Hardware-assisted virtualization 11

2.1.4 Xen vs KVM 14

2.2 KVM 16

2.2.1 Memory management 17

2.2.2 VirtIO 19

2.3 Packet journey: from the wire to the Virtual Machine 21

2.4 Data reduction: packet filtering 24

3 The vNPlug Framework **29**

3.1 Hypervisor-bypass and isolation 29

3.2 Framework design 31

3.3 vNPlug-Dev: memory mapping and event signalling 32

3.3.1 Zero-copy from kernel-space to guest user-space 32

3.3.2 Two-way event signalling 36

3.4 vNPlug-CTRL: control messages over VirtIO 40

3.4.1 Routing messages 41

3.5 vNPlug interface 42

3.5.1 Host side API 42

3.5.2 Guest side API 43

4 Validation with PF_RING **45**

4.1 PF_RING: a kernel module for packet capture acceleration 45

4.2 Virtual PF_RING 47

4.2.1 Design 49

4.2.2 Performance evaluation 50

4.2.3 Validation 59

5 Final Remarks **63**

5.1 Open issues and future work 64

A Ethernet Basics **67**

B Code snippets **71**

B.1 vNPlug-CTRL 71

B.1.1 Host side 71

B.1.2 Guest side 76

B.2 vNPlug-Dev 77

B.2.1 Host side 77

B.2.2 Guest side 80

Bibliography **83**

List of Figures

2.1 Intel VMDq technology 12
2.2 Intel SR-IOV technology 13
2.3 Xen and KVM design approaches. 15
2.4 QEMU/KVM memory design 18
2.5 Packet journey with Virtio-Net. 24

3.1 Hypervisor-bypass idea 30
3.2 Framework architecture 32
3.3 vNPlug-Dev memory mapping 35
3.4 QDev live cycle 36
3.5 Host-to-guest notifications 39
3.6 Guest-to-host notifications 40
3.7 Control messages routing 42
3.8 Registration of an application and creation of a new mapping . . . 44

4.1 PF_RING design 46
4.2 Virtual PF_RING design 50
4.3 Testbed topology 51
4.4 Number of packets processed per time unit (1x Gigabit Ethernet) . 53
4.5 Percentage of idle time (1x Gigabit Ethernet) 54
4.6 Percentage of Packet Capture Loss 55

4.7 Number of packets processed per time unit (1x VM, 2x Gigabit Ethernet) . . . 56
4.8 Percentage of idle time (1x VM, 2x Gigabit Ethernet) . . . 57
4.9 Number of packets processed per time unit (2x VM, 2x Gigabit Ethernet) . . . 58
4.10 Percentage of idle time (2x VM, 2x Gigabit Ethernet) . . . 59

A.1 Ethernet frame format . . . 67

List of Tables

4.1 Maximum rates for Gigabit Ethernet 52

Chapter 1

Introduction

Virtualization continues to demonstrate additional benefits the more it is used and almost all today's datacenter solutions are based on it. Running network monitoring applications for high-speed networks on a VM (Virtual Machine) is not a simple task, especially when we need to capture and analyse every single packet, ensuring no packet loss and low latency.

The aim of this thesis is to provide a method to achieve near-native performance in packet capture within a VM, using Open Source virtualization solutions and commodity hardware. Considerations, in this chapter, apply to the Kernel-based Virtual Machine, a full virtualization solution for Linux, better described in Chapter 2, where we will see the motivation of this choice.

This chapter presents an overview of the problem, explains the scope and the motivations of this thesis, and identifies its requirements.

1.1 Overview

Virtualization, intended as the technique for the simultaneous execution of more operating systems instances on a single computer, is increasingly gaining popularity for a lot of well-known reasons.

- Improved power and hardware utilization: fewer physical computers and

hardware utilized more efficiently, increasing so energy efficiency.

- High flexibility thanks to the abstraction layer that separates applications and hardware.
- Simplified disaster recovery: failures are quickly repaired, increasing availability. With live migration, a running virtual machine can be moved among different physical computers, without disconnecting the application.
- Automatic load-balancing techniques instantaneously reallocating hardware resources on peak demands.
- Administration costs reduction: less effort is required by administrators, due to automatic disaster recovery and load-balancing software. Furthermore there are fewer physical computers, and the number of specialized installations is dramatically reduced.

On the other hand, virtualization software must be very robust and efficient, without compromising performance. In fact, it is well-known that virtualization involves additional overhead, due to the abstraction level it provides. This is especially true in the case of network monitoring applications for high-speed networks. Analyzing efficiently an high-speed network, by means of applications running on VMs, is a major performance challenge, and this is even more hard if we use commodity hardware.

Packet capture is perhaps the most expensive task for passive monitoring applications, in terms of CPU cycles. Despite many efforts made to speed the networking on VMs up, it still represents a bottleneck and native performances are still away (see [46]), unless you use hardware support, usually not so flexible.

Regarding packet capture with applications running on native general-purpose operating systems and commodity hardware, researchers [15, 20, 17] have demonstrated that performance can be substantially improved by enhancing these systems

for traffic analysis. In fact, the abstraction layers, provided by these systems, produce a series of overheads and multiple memory accesses for packet copies which can be avoided. These results can be exploited and extended in order to improve performance also in the case of virtualized environments, where the level of abstraction is much higher and introduces much more overhead.

1.1.1 A long packet journey

The journey of a packet, from the wire to the application, on a native operating system, can be long and it usually involves multiple copies.

In Chapter 2 we will see that the journey of a packet, from the wire to an application running on a VM, can be longer, because it passes through multiple layers before reaching the application. The number of copies may further increase, and the abstraction layer provided by the hypervisor (the software managing the VMs) introduces additional overhead.

The worst case occurs with fully virtualized network devices. In this case the guest operating system (the operating system running on the VM) runs unmodified using standard drivers for network devices. But this high level of abstraction, provided by the hypervisor, leads to very low performance.

Better performance can be achieved with paravirtualized network devices: the guest operating system communicates with the hypervisor using ad-hoc drivers, reducing packet copies as well as the hypervisor overhead. In Chapter 2 we will see some example reaching one copy, or zero-copy in some smart implementations, from the hypervisor to the guest's kernel-space.

In both cases, the journey of a packet continues on the guest operating system, and needs additional copies, from the guest's kernel-space to reach the monitoring application. The aim of a capture accelerator is to reduce this journey by providing a straight path from the wire to the monitoring application, avoiding the operating system and hypervisor overheads.

1.1.2 Packet filtering: the earlier the better

The common and simplest way to configure VMs networking is to use a virtual bridge. A virtual bridge is a functionality usually provided by the operating system, a way to connect two Ethernet segments together. With this configuration you can link a real network device to a virtual network device. The virtual network device can be used to forward packets to the hypervisor, as we will see in Chapter 2. In this configuration, another issue with packet capture is that, with the real device in promiscue mode (making the device pass all traffic it receives), every VM will receive a copy of each packet crossing the card.

As said before, one of the main reasons of performance loss is due to packet copies, and we should avoid to waste CPU cycles pushing up unnecessary packets. Thus, it becomes important to use efficient packet filtering techniques at the lowest level possible.

An alternative way to virtual bridges is virtual switches, for example Open vSwitch [36]. Open vSwitch operates like an hardware switch, implementing standard Ethernet switching, but also providing high flexibility with filtering support at various layers of the protocol stack and supporting OpenFlow [30]. As mentioned in [35], the problem is that these switches require much CPU to switch packets, and these CPU cycles could be used elsewhere.

So, besides the reduction of the packet journey, designing a network monitoring application it is important to take into account an efficient as-soon-as-possible packet filtering method. A packet capture accelerator should provide support also this way.

1.1.3 Limited network hardware support

In order to boost network I/O and to reduce the burden on the hypervisor, in the last few years, companies as Intel have addressed their efforts in enhancements to processors, chipsets and networking devices by enabling hardware assists to the

virtualization requirements.

As the hypervisor abstracts the network device and shares it with multiple VMs, it needs to sort by destination and then deliver incoming packets. In order to avoid the overhead introduced by this sorting activity, the new generation of commodity network cards support multiple reception queues, by performing data sorting in the adapter. This technology, combined with optimized paravirtualization techniques, allows networking to achieve very high performance. But the number of queues is limited, as well as the sorter ability, resulting not so flexible for packet capture.

Other technologies allow VMs to share a real network device and to bypass the hypervisor involvement in data movement. This is achieved by configuring a single ethernet port to appear as multiple separate physical devices, each one assigned to a VM. Once again, these techniques allow networking to achieve near-native performance, but they are not flexible enough for packet capture, due to limitations on the number of virtualized devices and sorting filters.

1.2 Motivation

In the era of Cloud computing [3], which is the natural evolution of the adoption of virtualization, the efficient execution of network monitoring applications on VMs can also open a new scenario. For instance a company can provide a remote on-demand network analysis service with some considerable advantages:

- no need of expensive and intrusive hardware in place;
- elimination of support and upgrade costs;
- clients can dynamically select services from a set.

But also the use of VMs within an in-place network appliance provides some advantages, the same as described above by talking about benefits of virtualization, provided that they are efficient.

Furthermore, we can consider application domains such as lawful interception. If an xDSL user is intercepted, only a few tenths packets per second need to be captured out of million flowing on the link, where several hundred users are connected. Moreover, we can consider to separate traffic analysis, regarding different flows on the same link, on different VMs. Existing general purposes solutions are not optimized for splitting traffic into flows as efficiently as required by these applications. Thus, a solution which provides a straight path for packets and gives the possibility to apply an efficient packet filtering at the host (the actual machine on which the virtualization takes place), may represent the turning point.

1.3 Scope of work

As explained above, the journey of a packet from the wire to the application is too long. The use of software switches in order to play with flows requires additional overhead. Relatively new hardware support, combined with optimized paravirtualization techniques, provides near-native performance and represents a very good solution for common network connectivity, but it is too limited for packet capture.

The scope of this thesis is to accelerate the packet capture, defining a framework that can be used with little efforts by a network monitoring application running on a VM, to achieve near-native performance.

1.4 Requirements

During the analysis phase, some requirements have been defined. In Chapter 4 these requirements will be used to validate the work.

1. High performance

   The objective of this framework is exactly the following: increase application

performance. This is accomplished either directly, helping the application to collapse the packets path, and indirectly, allowing the applications to use smart tricks, for instance the as-soon-as-possible packet filtering.

2. Flexibility

   The framework must be flexible enough to accommodate various network monitoring application families, without focusing on one application loosing, this way, generality, and providing everything a network monitoring application may need.

3. Dynamic VM reconfiguration

   Applications must be able to start using the framework at runtime, whitout restarting the VM, or the framework itself, or other applications using the framework.

4. Scalability

   The framework must be scalable in terms of number of concurrent applications using it.

5. Ease of use

   The framework must provide a simple interface which leads the design and the development of efficient monitoring applications, requiring limited effort.

6. Open Source

   The proposed solution must be based on Open Source software.

7. Commodity hardware

   The proposed solution must not rely on expensive/exotic hardware to improve the performance. This increases the flexibility and makes easier the applications deployment.

1.5 Thesis outline

This chapter outlines the subject of this thesis, summarizing the key concepts necessary to understand its goals and to identify the requirements.

Chapter 2 provides the knowledge on virtualization technologies, focusing on Open Source solutions, and describes existing software and hardware supports to networking.

Chapter 3 describes the design choices made during the framework definition.

Chapter 3 validates the work redesigning PF_RING, a packet capture accelerator, to run efficiently on virtual environments.

Chapter 5 contains conclusions, open issues and possible future work.

Appendix A contains an overview of the Ethernet communication protocol, that will be used to calculate the theoretical rate of a link.

Appendix B contains code snippets for a better and more pratical understanding of the framework.

Chapter 2

Related Work

2.1 Virtualization

Virtualization can refer to various computing concepts, in this case it is intended as the creation of Virtual Machines. A VM acts as a real computer, but one or many can be concurrently executed on a physical one, each with its own operating system. The hypervisor, also referred to as Virtual Machine Monitor, is the software that manages a VM on the host, providing allocation and access of physical hardware to the VMs. First of all, the CPU allocation, by means of scheduling algorithms, is an example of resource management.

An important property of virtualization is the isolation: even if VMs share the resources of a single physical computer, they remain completely isolated from each other, as if they were separated physical computers. It must be not possible for the software running on a guest to acquire access to the host or to another VM, providing a platform for building secure systems. Furthermore, the fault isolation enhances system reliability, allowing the execution of several critical applications on one physical machine.

2.1.1 Virtualization technologies

Many different virtualization techniques are available today, with different flexibility and performance, depending on the abstraction degree. In fact, a complex abstraction layer provides high flexibility, but also overhead and lower performance.

Full virtualization

With full virtualization, the hypervisor emulates the complete hardware visible to the VM, and it is usually used when the guest's instruction set differs from the host's instruction set. QEMU [7] is an example of machine emulator that relies on dynamic binary translation [38, 39], converting binary code from the guest CPU architecture to the host architecture on the fly, achieving a reasonable speed. This is a very high abstraction level, which makes it possible to run the same VM on hosts with different architectures, but also implies a huge overhead for the hypervisor and very low performance.

Native execution

With native execution, the hypervisor virtualizes a machine with the same instruction set of the physical host. This way the guest operating system can be executed natively, ensuring isolation thanks to some techniques, such as hardware protection and trap mechanisms. In fact, the guest operating system is executed natively in unprivileged mode, and whenever it tries to execute a privileged instruction, which would conflict with the isolation or the resource management of the hypervisor, the hardware protection traps out to the hypervisor invoking an handling routine. This technique reduces overheads allowing to achieve very high performance.

Paravirtualization

When the host architecture does not have hardware support able to cooperate with virtualization techniques, it is possible to achieve high performance through paravirtualization. Paravirtualization requires the guest operating system is aware of being virtualized, forwarding critical instructions which need to be executed to the hypervisor by the hypercall mechanism. This technique can be placed performance-wise between full virtualization and native execution.

2.1.2 Paravirtualized devices

In a fully virtualized environment, the guest operating system is completely unaware of being virtualized, and all the hardware it sees is emulated-real-hardware, with very low performance.

In order to achieve higher performance, avoiding the overhead of emulating real hardware, it is possible to use paravirtualized devices. This way, 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 particular drivers to talk with the hypervisor through a more direct path. Later in this chapter we will discuss of Virtio [42], an interface for paravirtualization (look also at [2]).

2.1.3 Hardware-assisted virtualization

In the last few years companies like Intel and AMD have addressed their efforts in enhancements to processors, chipsets and networking devices to meet the virtualization requirements.

As we saw above in the case of “native execution”, thanks to hardware support, it is possible to improve virtualization achieving very high performance.

Almost all recent processors take advantage of virtualization extensions, such as the Intel Virtualization Technology and the AMD Secure Virtual Machine. The

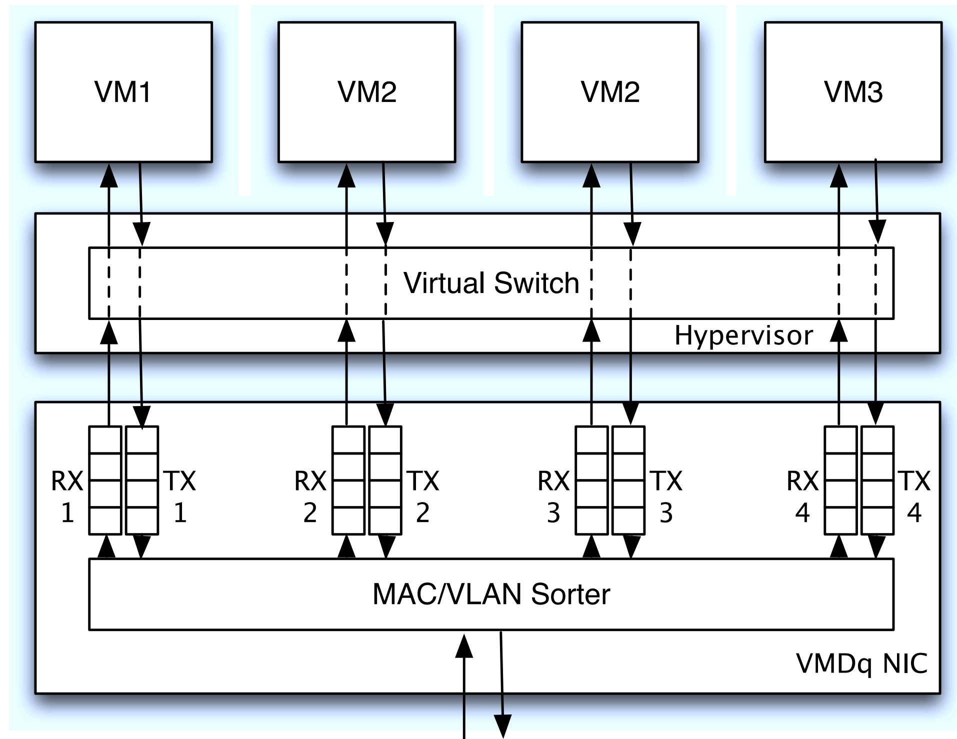


Figure 2.1: Hardware sorting using the Intel VMDq technology.

main feature of these technologies is to allow the guest code to be executed natively, by adding a new unprivileged execution mode, the guest execution mode, in which some instructions or registers accesses trigger traps that can be handled by the hypervisor. Furthermore, they provides hardware acceleration for context switches, and some other kind of enhancements, as we will see later talking about memory management with the Kernel-based Virtual Machine.

Other virtualization extensions concerning I/O devices, such as network devices, have been introduced in order to reduce the burden on the hypervisor.

Network devices

As hypervisor abstracts the network device and shares it with multiple VMs, it needs to sort by destination and then to deliver incoming packets. This sorting consumes CPU cycles and impacts on performance. Through the use of a relatively new generation of commodity NICs (Network Interface Cards), supporting multiple

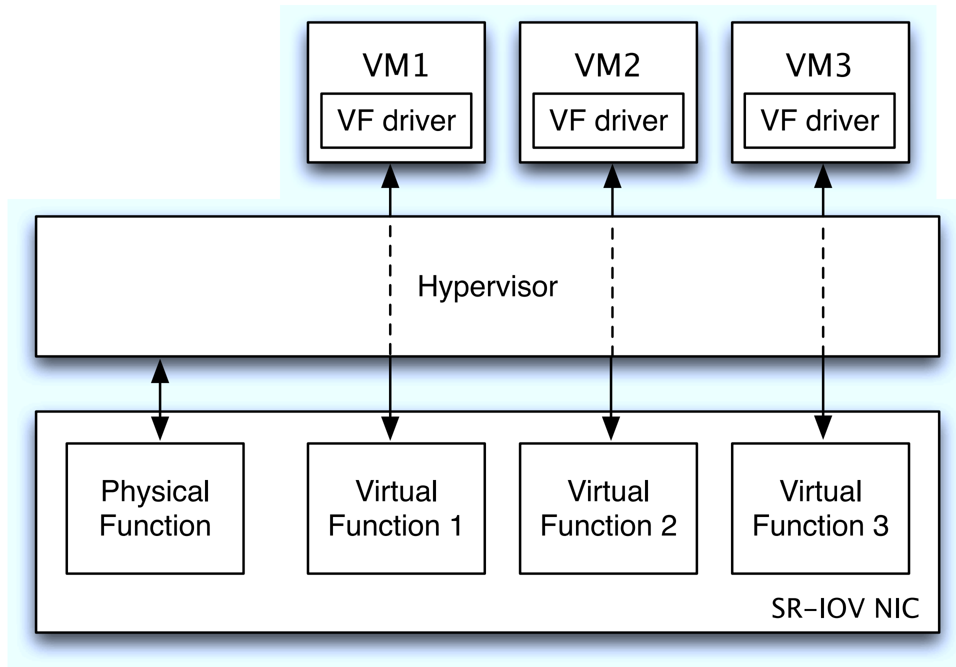


Figure 2.2: Virtual Functions with the Intel SR-IOV technology.

receive queues (such as the VMDq technology [43] depicted in Figure 2.1 on the facing page, part of the Intel Virtualization Technology for Connectivity), it is possible to reduce this overhead, by performing data sorting in the adapter. By using this technology combined with optimized paravirtualization techniques it is possible to achieve very high performance as shown in [41, 45]. The problem of this technology, with respect to our needs in packet capture, is the low flexibility, due to the limited number of queues and the confined ability of the sorter, which places packets based only on MAC (Media Access Control) address and VLAN (Virtual LAN) tag.

Another relatively new Intel technology is the SR-IOV (Single Root I/O Virtualization) [34], a way of sharing a device in a virtualized environment, bypassing the hypervisor involvement in data movement. With this technology a single ethernet port can be configured by the hypervisor to appear as multiple separate physical devices, each one with its own configuration space. As shown in Figure 2.2, the hypervisor assigns each Virtual Function, lightweight PCIe (Peripheral Component

Interconnect Express) functions, to a VM, providing independent memory space and DMA (Direct Memory Access) streams. Memory address translation technologies based on IOMMUs (Input/Output Memory Management Unit) [8, 22], such as Intel VT-d [13] and AMD IOMMU, provide hardware assisted techniques to allow the direct DMA transfers keeping isolation between host and VMs . The hypervisor is still involved either for control functions and to deliver virtual interrupts to the VMs. Once again, this technique allows networking to achieve near-native performance, but it is not so flexible, due to limitations on the number of Virtual Functions and sorting filters.

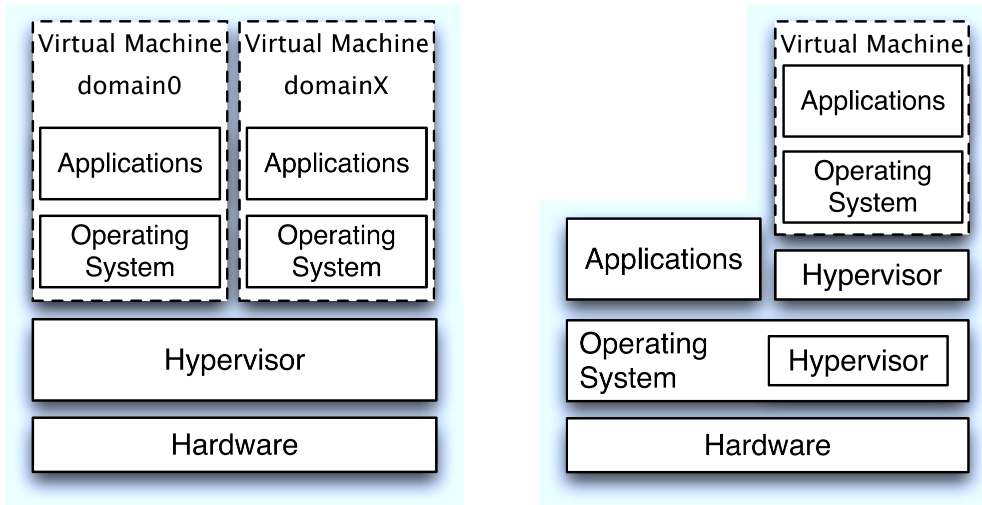
There are several projects [40, 47, 27] following the same approach of the SR-IOV, with different designs of self-virtualized devices for direct I/O. They represent good solutions for common network connectivity but, besides efficiency, we aim to provide more flexibility to traffic monitoring applications.

2.1.4 Xen vs KVM

Since Open Source is required, there are two alternatives: Xen and KVM (Kernel-based Virtual Machine). Let's have an overview of both hypervisors. Figure 2.3 on the facing page shows the differences between the design approaches followed by these hypervisors.

Xen (see [5, 19]) comes as a stand-alone hypervisor, a layer directly above the hardware. It supports either paravirtualization and hardware-assisted virtualization. As said above, paravirtualization requires modified guest operating systems, using a special hypercall interface, allowing Xen to achieve high performance even on architectures which lack of hardware support to virtualization. Hardware-assisted virtualization, instead, allows Xen to run unmodified guest operating systems.

In Xen, the host operating system runs in *domain 0*, a privileged domain that can administer the other domains and allow control over the hardware. In fact, the



(a) Xen design. **(b)** KVM design.

Figure 2.3: Xen and KVM design approaches.

driver architecture is split in a back-end in the *domain 0* and a front-end in the guest domains: I/O requests in guest domains are sent to *domain 0*, which checks the request and executes the necessary operations.

KVM (an overview is available on [26]) is a small and relatively simple hypervisor based on Linux, in the sense that a kernel module makes the Linux kernel itself be an hypervisor. Officially it supports hardware-assisted virtualization only.

KVM, in practice, is split into architecture-independent and architecture-dependent modules, kvm_intel and kvm_amd, loaded according to the underlying hardware. These modules are officially included in the mainline Linux kernel, however they can be compiled separately, without any changes on the latter.

Common hypervisor tasks, like scheduling and memory management, are delegated to the Linux kernel, while the VMs hardware is emulated by a modified version of QEMU [7]. For instance, while Xen uses its own scheduling algorithms, such as the Borrowed Virtual Time or the Simple Earliest Deadline First, KVM relies on the Linux kernel algorithm.

Compared to Xen:

- KVM has the advantage of being into the mainline Linux kernel, while to run

a Xen host you need a supported kernel or to use some commercial solution.

- KVM is a pretty new project, so Xen has had more time to mature, but the former is growing very fast.

- KVM leverages Linux kernel capabilities, such as scheduling and memory management, benefiting from any of its improvements.

- Officially, KVM supports hardware-assisted virtualization only, anyway virtualization extensions are widely available on commodity hardware nowadays.

- KVM is small and relatively simple, therefore understanding it and integrating new functionality would be simpler with respect to Xen.

Therefore KVM has many advantages in our case, so it is where our choice falls upon.

2.2 KVM

As summarized above, KVM turns the Linux kernel into an hypervisor by means of a small kernel module and hardware virtualization extensions (the Intel Virtualization Technology and the AMD Secure Virtual Machine).

The KVM module provides a set of features, such as the creation of a VM, memory allocation, access to virtual CPU registers, injection of virtual CPU interrupt, etc. These features are accessed by means of a character device, which can be used at user-space through the *ioctl* interface.

In fact, with KVM, a VM appears as a normal user-space process. As with a regular process, the scheduler is not aware of scheduling a virtual CPU, and the memory manager allocates discontiguous pages to form the VM address space.

As described in [26], in order to run a VM, a user-space process calls the kernel to execute guest code, until an I/O instruction or an external event occurs. The

kernel causes the hardware to enter guest mode, a new mode added to the existing user mode and kernel mode. The processor can exit guest mode due to:

- an event such as an external interrupt or a shadow page table fault. In this case the kernel performs the necessary operations and resumes guest execution.
- an I/O instruction or a signal queued to the process. In this case the kernel exits to user-space.

In this design, the user-space process is a QEMU process. The upstream QEMU [7] is an Open Source emulator which supports many different guest processors on several host processors and it is able to emulate several I/O devices. KVM uses a modified QEMU, to benefit from the QEMU I/O model and hardware virtualization.

2.2.1 Memory management

The virtualized guest physical memory is part of the user address space of the QEMU process, allocated with a *malloc*-equivalent function. This means the guest sees *malloc*-ated memory as being its physical memory, and as a normal *malloc*-ated memory, there is no physical memory allocated until it is touched for the first time.

Every time a guest operating system makes a change in its page tables, the host hooks it and updates the shadow page tables. The shadow page tables encode the double translation from guest virtual to host physical, and are exposed to the hardware. This mechanism works by means of traps: when the guest tries to set the page table base register, the hardware triggers a trap, handled by the hypervisor. This allows the hypervisor to provide transparent MMU (Memory Management Unit) virtualization, but introduces overhead due to frequent updates to the page table structures [1].

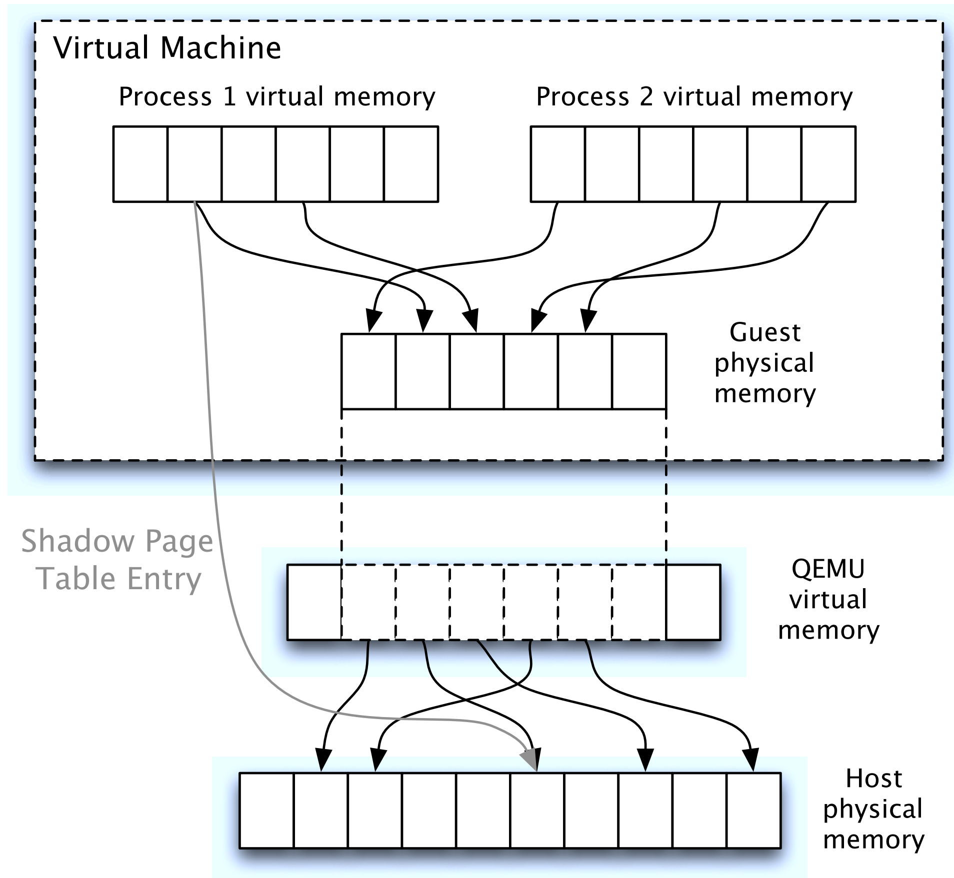


Figure 2.4: QEMU/KVM memory design and Shadow Page Tables.

The second generation of hardware extensions incorporates MMU virtualization (called Extended/Nested Page Tables respectively from Intel and AMD), providing multi-level translation in hardware [10] and eliminating much of the overhead incurred to keep the shadow page tables up-to-date. In fact, using Extended/Nested Page Tables, a first level of page tables mantains the guest virtual to guest physical mapping, while, in an additional level of page tables the hypervisor maintains guest physical to host physical mapping. Both the two levels of page tables are exposed to the hardware. So, when a virtual address gets accessed, the hardware first looks at the guest's page tables the same way of the native execution, then at the nested page tables to determine the corresponding host physical address. The TLB (Translation Lookaside Buffer) is extended with new tags, called Virtual Processor Identification by Intel, which allows the TLB to keep track of which

entry belongs to which guest. Thus, entries of different guests can coexist and a VM switch does not require a TLB flush.

Linux may try to swap the *malloc*-ated memory representing the guest physical one, but every change must be coordinated with these hardware structures. In KVM this is a task of the MMU Notifier.

2.2.2 VirtIO

Virtio is the de-facto standard for paravirtualized devices in the QEMU/KVM environment, which provides a simple and efficient transport mechanism (an overview is available in [42]).

The basic abstraction used by Virtio is a *virtqueue*, that is a queue where buffers are posted by the guest and comsumed by the host. Each buffer is a scatter-gather array consisting of "out" entries, destined to the hypervisor driver, and "in" entries, used by the hypervisor to store data destined to the guest driver. Paravirtualized devices can use more than one queue, for instance a network device uses one queue for reception and one for transmission. Virtio also provides a mechanism to negotiate features, so that back-end supports can be detected by guest drivers.

Virtio actually uses a Virtio-over-PCI implementation, which leverages the pre-existing PCI emulation and gives maximum portability for guests. With this approach, the configuration is also easy using an I/O region.

The *virtqueue* implementation, *virtio ring*, consists of:

- an array of descriptors, where the guest chains together guest-physical address and length pairs. Each descriptor contain also a "next" field for chaining and a flag indicating whether the buffer is read-only or write-only.

- a ring where the guest indicates which descriptors are available for use. This implementation permits an asynchronous use of the queue, so that fast-

serviced descriptors do not have to wait for the completion of slow-services descriptors.

- a ring where the host indicates which descriptors have been used.

When buffers are added/consumed, a notification mechanism is used. In order to avoid useless VM exits, some flags allow the consumer/producer of a *virtqueue* to suppress notifications. For instance, this can be an important optimization of a guest network driver, that is advising that interrupts are not required.

As said before, the guest memory appears in the host as part of the virtual address space of a process. Based on this assumption, this Virtio implementation is publishing buffers from guest to host, relying on the fact that it is simple to map guest physical memory to host virtual memory.

VirtIO-Net

The Virtio network driver uses two separate *virtqueue*s, one for reception and one for transmission. It has some features such as TCP/UDP Segmentation Offload and Checksum Offload.

Segmentation Offload is the ability of some network devices of taking a frame and breaking it down to smaller-sized frames, reducing so the number of packet transfers to the adapter. With Virtio-Net, the guest driver can transfer large frames to the host, reducing the number of calls out from the VM.

Some network devices can also perform hardware checksumming, required in some protocol header (i.e. TCP). Since we have a reliable transmission medium between guest and host, the Virtio-Net driver can forward packets to the back-end without computing the checksum. When a packet is forwarded to the physical network, the checksum gets computed, at that point, by the hardware.

VirtIO-Net allows network devices to achieve higher performance (but far from native) compared to emulated devices. The packet journey is still too long. Packets, before reaching the hypervisor, have to pass through a virtual bridge and a

virtual TAP device (a virtual network device which provides packet reception and transmission for user-space programs).

VHost-Net

VHost-Net is a relatively new back-end for Virtio-Net that accelerates this paravirtualized device. The guest side of the Virtio-Net driver does not require modification.

The VHost-Net support is designed as a kernel module, configured at user-space, with the aim to reduce the number of system calls by moving the *virtqueue* operations into the kernel. It uses *eventfd* (a file descriptor for event notification, better described later) for signaling, and structures the memory paying attention to VM migration.

First of all, this accelerator improves not only latency but also throughput and overhead. Compared to a user-space Virtio-Net back-end, performance are closer to native.

2.3 Packet journey: from the wire to the Virtual Machine

As described in [15], the journey of a packet from the wire to the application, on a native operating system, may be long and it usually involves at least two copies, one from the network card to a socket buffer (a data structure associated to every sent or received packet by the Linux Network stack), and one from the socket buffer to the monitoring application. The journey of a packet from the wire to an application running on a VM may be much longer, because it passes through multiple layers before reaching the application.

Usually, before reaching the VM hypervisor, packets pass through virtual switches [35, 36], or virtual bridges and TAP devices. When a packet reaches the hypervisor

it gets delivered to the guest operating system in different ways, according to the adopted virtualization technique.

The worst case occurs with fully-virtualized network devices. In this case the guest operating system runs unmodified using standard drivers for network devices, taking so the worst performance. Better performance can be achieved with paravirtualized network devices, such as Virtio-Net introduced above.

In order to have a better idea of the packet journey, using Virtio-Net, we shall give the following brief overview. We are not going to describe the journey when the VHost-Net support is used, because it is an optimization that runs at kernel-space and would be inflexible for a packet capture accelerator (anyway it will be considered for performance comparison in Chapter 4).

The journey begins when a packet hits the Ethernet adapter, then:

1. The packet is stored into an adapter's reception queue in a FIFO (First In First Out) manner.

2. If the device uses DMA, which is pretty common nowadays, the driver pre-allocates a socket buffer and initializes a pointer. The device copies the packet directly in kernel memory through DMA.

3. The adapter issues an interrupt to inform the CPU about the event.

4. The interrupt handler can notify the kernel in two ways:

    (a) With older drivers, it enqueues the buffer in a FIFO queue, notifies the kernel, and returns from the interrupt. This queue is unique for all the interfaces, one for each CPU to avoid serialization. The kernel will dequeue the packet later, using a software interrupt which passes the packet to the upper layer.

    This mechanism leads to a phenomena called "congestion collapse" [32], because if the number of received packets is high and for each packet an

interrupt is issued, the CPU spends all the time handling them instead of doing something else.

(b) New drivers, instead, use the NAPI (New API) [44] interface, a mechanism introduced to resolve the "congestion collapse". Instead of issuing an interrupt, the drivers notify the kernel about new packets, register the device on a poll list and disable interrupts. Then the device is polled by means of a software interrupt, and each time its handler is called, it can pass a certain quota of packets to the upper layer. If the queue is empty, then the device is not polled anymore and interrupts are turned on again.

5. At this point, before delivering the packet to the upper-layer protocol handler, a copy is delivered to each registered sniffer. Then, the bridging is handled.

6. The virtual bridge adds the source MAC address to the forwarding database, then looks for the destination MAC address. If the address is found, the packet is forwarded to the corresponding port, otherwise it is flooded to all ports.

7. If a virtual TAP device is connected to the bridge, as in the case of Virtio-Net, it receives a copy of the packet.

8. The QEMU process receives the packet from the TAP device using a *read* system call.

9. Virtio-Net then delivers the packet to the VM, by copying it into a *virtqueue* and sending a notification (the Virtio-over-PCI support emulates an interrupt) to the paravirtualized drivers running on the guest.

The journey continues on the guest operating system following the same path since the beginning.

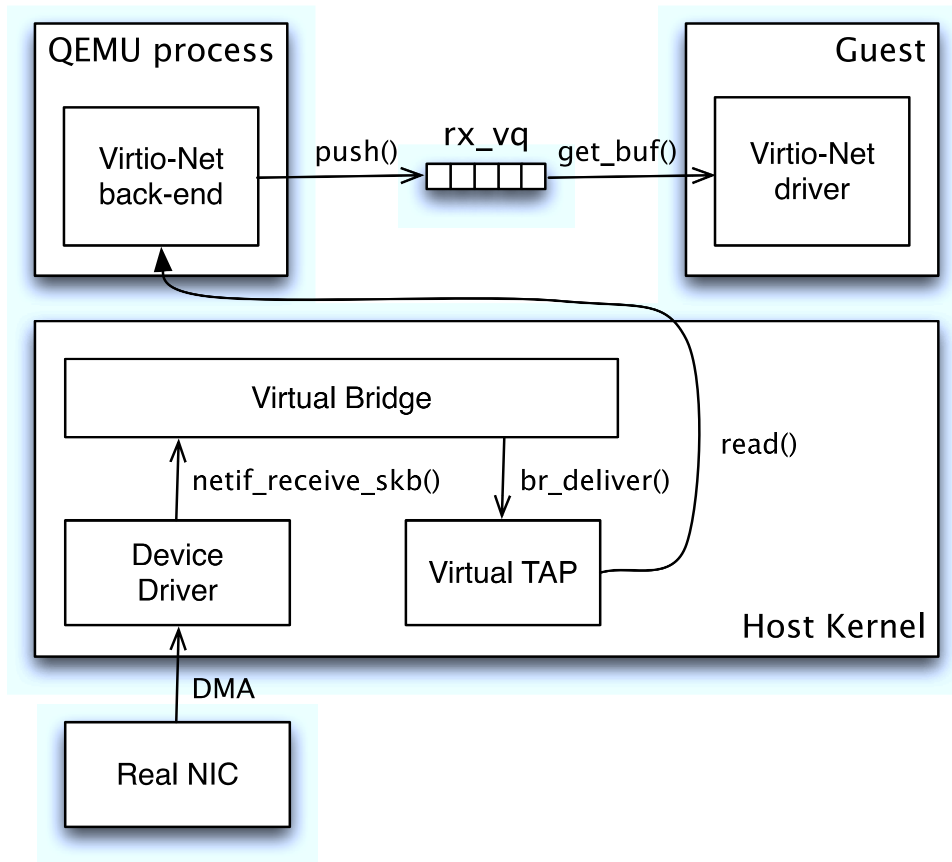


Figure 2.5: Packet journey with Virtio-Net.

A sniffer running on a VM, which misses of virtualization support, is able to receive the packet only when the 5th point on the guest is reached. Therefore, it pays, besides the overhead of the hypervisor (included virtual bridge and virtual TAP), two times the overhead of the operating systems mechanisms.

So it is clear that, in the same way as a packet capture accelerator on a native system creates a straight path at least from the 5th point to the monitoring application, we are at least twice as motivated to create a straight path from the 5th point on the host to the application running on the guest.

2.4 Data reduction: packet filtering

The previous section shows the way QEMU/KVM provides networking to the VMs, passing through a virtual bridge. As we saw, a virtual bridge connects two

Ethernet segments together providing L2 (Data Link layer) forwarding, flooding all the received packets, if no information on the destination are available. This means that, in a typical network monitoring activity, every VM linked to the real device will receive a copy of each packet that crosses the NIC.

As discussed in [14], in application domains such as lawful interception, this represents a major problem. In fact - as mentioned - “network operators usually provide a copy of packets flowing through a link where several hundred users are connected, while the law states that only the traffic of intercepted users can actually be captured and analyzed. This means that if an xDSL user is intercepted, only a few tenths pps (packets per second) need to be captured out of million pps flowing on the link”. This becomes even more evident in our case, where we might consider to split traffic into flows, implementing separate traffic analysis on different VMs.

This example demonstrates that providing a straight path for packets to increase packet capture performance is not enough. The overall system performance can be further improved with packet filtering, by letting the system to “do not waste several CPU cycles just for pushing unnecessary packets to user space that will be later discarded”. Thus, it becomes important to reduce packet copies with an as-soon-as-possible packet filtering.

One possible solution is to replace the virtual bridge with a virtual switch, for example Open vSwitch [35, 36]. Open vSwitch operates like an enhanced L2 switch, implementating standard Ethernet switching, providing high flexibility with full control on the forwarding table, specifying how packets are handled based on L2, L3 (Network Layer), and L4 (Transport Layer) headers, and implementing a superset of the OpenFlow protocol [30].

As mentioned in [35], the problem is that these switches require high CPU utilization to switch packets, and these cycles could be used elsewhere. Designing our framework, we should provide an high degree of flexibility to applications, allowing them to use more optimized packet-filtering solutions (both software and

hardware-based) at the host.

The concept of *packet filtering* was first proposed by Mogul et al. a quarter of a century ago [31]. The idea was the same: moving the filtering to the lowest layer possible reduces the system overhead ("the packet filter is part of the operating system kernel, so it delivers packets with a minimum of system calls and context switches") pushing up only interesting packets ("far more packets are exchanged at lower levels than are seen at higher levels. A kernel-resident implementation confines these overhead packets to the kernel and greatly reduces domain crossing").

So far, an extensive research has been done to refine the filtering model and a large number of solutions have been produced.

BPF (Berkeley Packet Filter) [29], the most widely-used packet filter, was born a few years later to support high-speed network monitoring applications. BPF is implemented as an interpreter able to execute programs. A filter program is an array of instructions that sequentially perform some actions on a pseudo-machine state. Instructions, for instance, can fetch data from the packet, execute arithmetic operations, and test the results, accepting or rejecting the packet based on them.

Enhancements to BPF and many other solutions have been introduced over the years, highlighting the importance of a fast and flexible packet filtering tool. We can see some examples.

BPF+ [6] enhances the performance of BPF with optimizations to eliminate redundant predicates across filters and just-in-time compilation to convert filters to native code.

xPF [23] tries to reduce the context switching overheads of BPF, increasing its computational power and moving more packet processing capabilities into the kernel. In fact xPF allows programs to maintain state across invocations and supports backward jumps. The idea is to use BPF not just to demultiplex packets but as a tool to execute monitoring applications inside the kernel.

FFPF (Fairly Fast Packet Filters) [11] is a monitoring framework which further

increases performance. When multiple applications are executed simultaneously, it tries to avoid packet copies by sharing packet buffers when multiple applications need to access overlapping sets of them, and uses memory mapping to reduce context switches and copies between kernel-space and user-space. FFPF is also extensible using kernel-space library functions, which are precompiled binaries that can be loaded at runtime.

Swift [50] tries to achieve high performance in a way similar to BPF. In fact Swift is also implemented as an interpreter which executes programs, but with a simplified computational model and powerful instructions, allowing common filtering tasks to be accomplished with a small number of instructions.

PF_RING, described in Chapter 4, is a packet capture accelerator which also provides enhanced packet filtering capabilities, and it is extensible through plugins and hardware support.

Chapter 3

The vNPlug Framework

This chapter discusses the design choices which led to the creation of vNPlug, a framework that drives the design, and facilitates the development, of network monitoring applications executed on VMs, allowing designers and programmers to focus on software requirements rather than to deal with details concerning virtual environments in order to achieve high performance.

3.1 Hypervisor-bypass and isolation

As introduced in Chapter 1 and highlighted in Chapter 2, this framework aims to improve the packet capture speed, first of all by reducing the packet journey.

In order to achieve the same result, in the case of a native operating system, the approach followed by PF_RING (see Chapter 4) is to bypass the operating system standard mechanisms using a memory map from kernel-space to the address space of the monitoring application. This straight path reduces the overhead and the number of data copies, allowing PF_RING to achieve high performance, demonstrating this way how a general purpose operating system can be optimized for network monitoring [12].

The operating system bypass approach is adopted in many research project as well as by commercial products, most of all in areas requiring intense I/O activity,

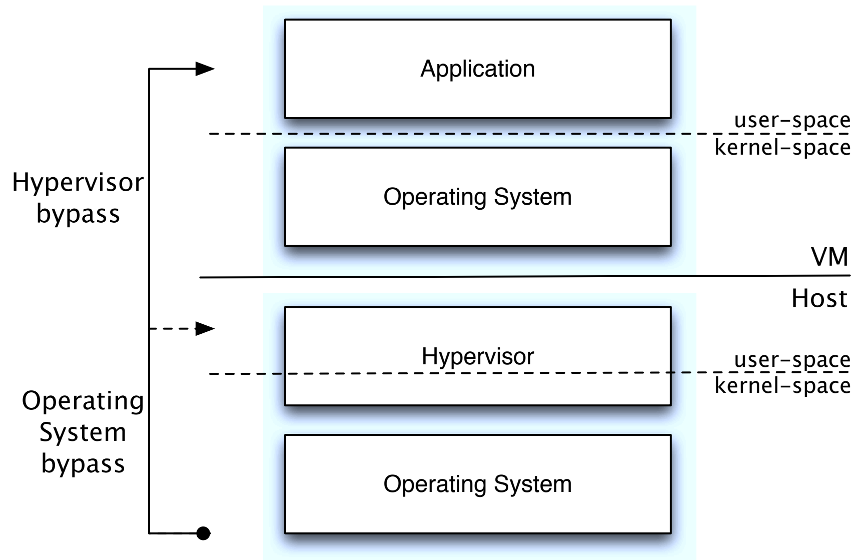


Figure 3.1: Hypervisor-bypass idea.

and where low latency and high bandwidth are vital [48, 4, 37].

With this work, we want to propose a model that extends the idea of operating system bypass to the context of virtual environments, providing a way to create a mapping between the host kernel-space and the guest user-space, where the monitoring application is located. Figure 3.1 depicts the idea of hypervisor-bypass.

The hypervisor involvment 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, becoming a bottleneck for I/O intensive workloads. The hypervisor-bypass approach aims to perform operations that require intensive workloads directly, without the involvement of the hypervisor.

Studies on High-Performance Computing [21, 28] 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.

We shall now focus on isolation: bypassing the hypervisor makes it unaware of the resources allocated by the applications. Since we aim to give to the applications

an high degree of freedom, this might break the isolation property through, even if, designing the framework, we have been pedantic on safety. That's the price for flexibility.

3.2 Framework design

In this section we start with an high-level view of the design, to identify the major components of the framework.

As described in the previous section, this framework should provide a way to create a mapping between the host kernel-space and the guest user-space in order to reduce the packet journey.

Furthermore, it should provide a way for applications to use some features provided by the host side, such as an as-soon-as-possible packet filtering.

These requirements suggest an architecture split in a guest side and a host side, with some kind of control communication to let both sides get coordinate. For instance, the monitor application on the guest side, might want to use the communication channel to instruct its own back-end, on the host side, to filter some kind of packets.

Thus, as shown in Figure 3.2 on the following page, we can identify two main components of the framework. The first component, vNPlug-Dev, is responsible of memory mapping and of an efficient event signalling. The second component, vNPlug-CTRL, is responsible of coordinating the host and the guest side by providing a control communication channel.

Obviously, with this design, the application using the framework, which can be a monitoring application designed for virtual environments or a library for packet capture, is aware of being virtualized. Of course, in the case of a library for packet capture, it is possible to exploit the library abstraction layer in order to run monitoring applications, this time, not aware of virtualization.

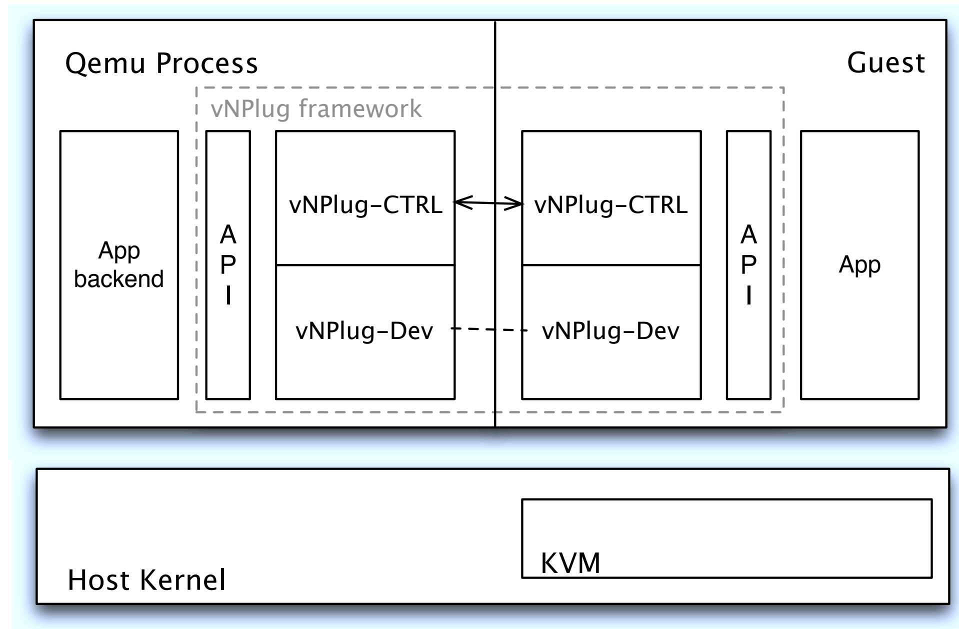


Figure 3.2: vNPlug global architecture.

3.3 vNPlug-Dev: memory mapping and event signalling

This component shall provide a way to map memory from the host kernel-space to the guest user-space, that is where the monitoring application is running. In addition it shall provide an efficient event signalling to notify, for instance, the arrival of a new packet.

3.3.1 Zero-copy from kernel-space to guest user-space

In this section we aim to define the basis for allowing applications to create a mapping between host kernel-space and guest user-space, in order to support zero-copy implementations.

Paging and memory map overview

As we know, the virtual address space of a Linux process is divided into pages, with a lot of advantages. This let processes share the physical memory, while

each one has a linear virtual address space. Thanks to this abstraction, each page can reside in any location of the physical memory, and the same physical memory can be mapped to different pages. Furthermore, it is possible to use a memory protection mechanisms on pages. Each process has its own set of Page Tables, mantained by the operating system, which keep information about the mapping between virtual memory and physical memory, and associate, to each page, some protection flags.

Each Linux process is represented by a process descriptor, plus a memory descriptor. The latter manages information about the process memory, including the Page Tables and the Virtual Memory Areas. A Virtual Memory Area, basically, consists of a range of contiguous addresses in the process virtual address space, and some access rights. For instance, the stack segment corresponds to a Virtual Memory Area. This one-to-one association does not apply to the memory mapping segment (used by the kernel to create anonymous memory mapping or to map files directly to memory) where there may be different Virtual Memory Areas, one for each mapping.

This is what we have behind the scene of the *mmap* system call, that allows an application to ask for a mapping between the process virtual address space and a file or a shared memory object. PF_RING, for instance, uses this functionality to map the kernel-space ring buffer into the user-space library.

The next step is allowing an application, like PF_RING, to map a user-space memory area, which is the result of a *mmap*, into a guest.

Exporting guest memory

As described in Chapter 2, the guest physical memory appears in the host as part of the virtual address space of the QEMU process. Thus, if a guest physical address is given, the mapping in host virtual address results simple. This is the way Virtio works, publishing buffers from the guest to the host.

In order to keep things simple, we want to map contiguous guest physical memory, resulting so contiguous in the virtual memory of the QEMU process. This simplify the applications design, in particular when they need to map a buffer from the host kernel-space.

Even if exporting guest memory is simple, the allocation of contiguous physical memory on the guest becomes difficult, in particular if the application requires a large buffer. Furthermore, even with the assumption of contiguous memory, it is complex to have an efficient access to such memory from the host kernel-space.

Instead, we aim to allow an application - mapping via *mmap* large contiguous buffers from kernel-space (allocated for instance with a single *vmalloc*) into user-space - to map this buffer into the guest. Therefore, we would find another solution, a way to import host memory instead of exporting guest memory.

Virtual PCI devices: importing host memory

Another way is to attach an additional block of memory to the guest by means of a memory region of a virtual device, mapping a virtual memory area of the QEMU process to this memory region. This is supported by the internal API (Application Programming Interface) of QEMU/KVM, which allows a module to emulate a PCI device and create this mapping. The mapped virtual memory area may be the result of a *mmap* on kernel-space memory, for instance.

Inside the VM, the memory region of the virtual device can be accessed by an *ioremap*, and mapped in a virtual memory area of a process via a kernel module which creates a character device with *mmap* support.

This way we have achieved our goal: a mapping between host kernel-space and guest user-space (Figure 3.3 on the next page shows the steps which led to the final mapping). Actually, the framework aims to provide a mapping between the host user-space and the guest user-space, with the possibility, for the application, to create the second mapping, from host kernel-space to host user-space.

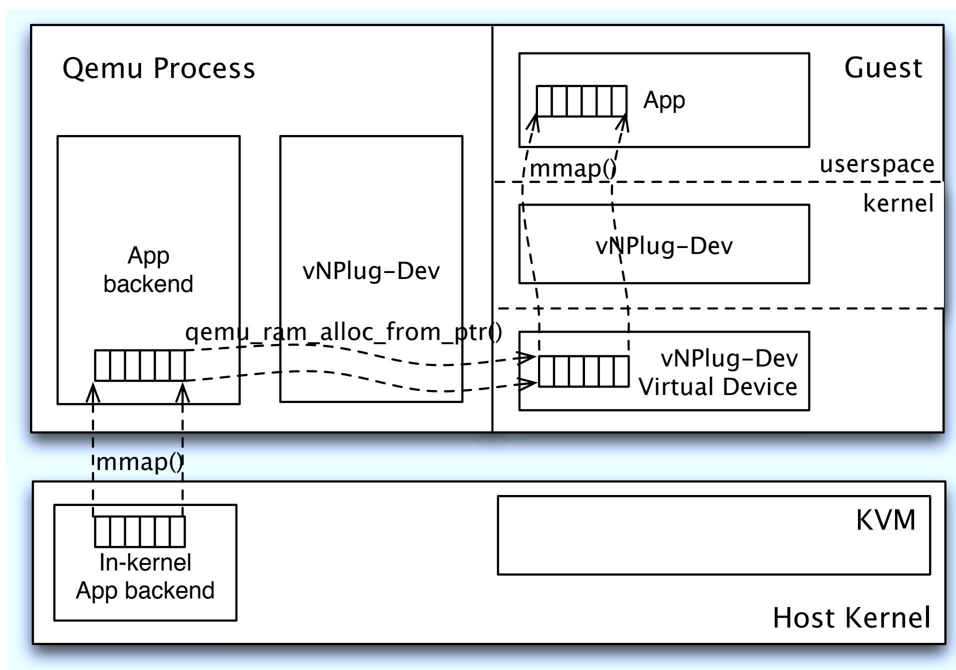


Figure 3.3: Memory mapping using the Virtual Device.

QDev, virtual device management and hotplug

The QEMU internal API provides device creation and configuration through the QDev device model. The latter manages the tree of virtual devices connected by busses, and supports device hot-plug and hot-unplug. Figure 3.4 on the following page shows the live cycle of a QDev device.

The hotplug support allows devices to dynamically attach to or remove from the system, while it is running. The PCI bus supports hotplug, and it is possible to create a QDev-compliant "hotpluggable" PCI device. Furthermore, basic hotplug support is included in all modern operating systems, making "hotplugged" devices get immediately usable, with no extra effort.

The interesting aspect of this is that we can take advantage from the hotplug support to dynamically attach shared memory to the guest, whenever it is required. Without this support, we would have had to attach a device with the shared memory at the VM boot, but this would have been inflexible.

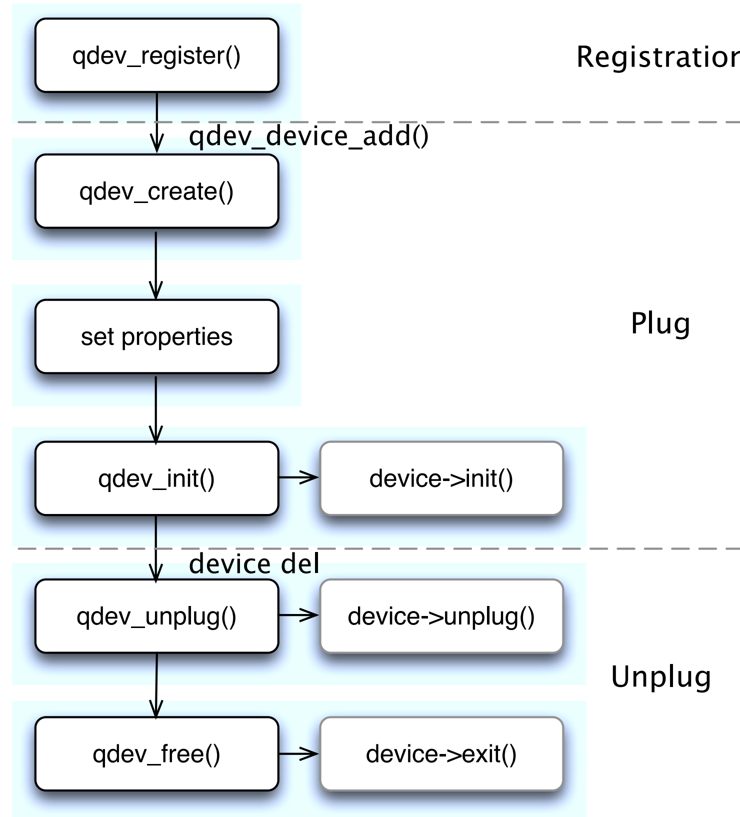


Figure 3.4: QDev live cycle.

3.3.2 Two-way event signalling

As mentioned above, the framework must provide a notification mechanism between host and guest. In fact, applications may need an high performance mechanism to notify, for instance, of a new packet into the receive buffer. It is easy to find other examples of notifications in the opposite direction as well. Thus, the framework aims to create an efficient two-way event signalling mechanism between host and guest.

Host to guest: Message Signaled Interrupts and IRQFD support

Since we are using a virtual PCI device in order to share memory, it is natural to think of a notification mechanism from host to guests that we already have for free: virtual hardware interrupts. There are two alternative ways to use interrupts: IRQs and Message Signaled Interrupts.

With traditional IRQs, a device has an interrupt pin which asserts when it wants to require a service. This way, PCI cards are limited to a small number of interrupts due to the hardware limitation.

Message Signaled Interrupts is an alternative way of requiring interrupts. It enables a device to generate an interrupt, emulating a pin assertion by sending a memory write on its PCI bus, to a special address in memory space. Then the chipset determines which interrupt it has to generate, depending on the data sent with the write request. Message Signaled Interrupts increases the number of available interrupts, allowing a device to allocate up to 32 interrupts with MSI, 2048 with MSI-X. Message Signaled Interrupts also have a lot of other features that significantly increase flexibility.

The QEMU/KVM device model lets a virtual PCI device generate both traditional IRQs and Message Signaled Interrupts, the latter when supported by the guest operating system. Designing our framework we can use both, encouraging the use of the latter which increase flexibility, simplifying the notification mechanism when several events are required, using multiple vectors instead of status registers, and performance, due to the KVM support which will be introduced later.

Given that we have now a way to notify the guest, we need a flexible mechanism for applications to require an interrupt. Both QEMU and KVM have nice support for *eventfd*, a file descriptor for event notification. With this mechanism it is possible to create a file descriptor that can be used to refer to an *eventfd* object from either user-space and kernel-space. Basically an *eventfd* object is represented by a 64-bit kernel-space counter.

An *eventfd* has the following behaviour:

- A *read* on the file descriptor :
  - if the counter has a non-zero value, returns an unsigned 64-bit integer. Then the counter is reset to zero or decremented by 1, according to a flag specified when creating the *eventfd*.

  - if the counter is zero, blocks until the counter become non-zero, or fail if the file descriptor has been set "non-blocking".

- A *write* on the file descriptor adds an unsigned 64-bit integer to the counter.

QEMU provides a simple way to add *eventfd* file descriptors, with associated handlers, to a set of ones that the former polls. Thus, one solution is to let the QEMU process poll an *eventfd*, and to associate an handler which notifies the VM with an IRQ or an MSI vector (one for each *eventfd*, if there are many).

Another solution, even more efficient, keeping the flexibility of using *eventfd* to throw interrupts, concerns the use of the *irqfd* support of KVM. In fact, since virtual interrupts are injected to the guest via KVM, the *irqfd* support allows the latter to directly translate in kernel-space a signal on an *eventfd* into an interrupt, without passing through the QEMU process.

Now, from the interrupt reception, inside the guest kernel, we need a mechanism to notify the user-space application. Here we can get inspired by the *eventfd* approach, using a blocking *read* on the character device (the same previously created for the memory map) in order to wait for notifications.

Figure 3.5 on the next page shows the path of a notification when the *eventfd* is signaled within kernel-space and *irqfd* is used.

Guest to host: IOEventFD support

We want an efficient mechanism for notifications from guest to host and the best candidate is the *ioeventfd* support of KVM, which is also used by VHostNet, the optimized back-end for Virtio-Net described in Chapter 2, as it is the fastest and more flexible way to get notified from the guest.

With the *ioeventfd* support it is possible, while creating the emulated PCI device in QEMU, to register arbitrary addresses of a MMIO (Memory-Mapped I/O) region, in order to trigger an *eventfd* signal, when the guest tries to execute

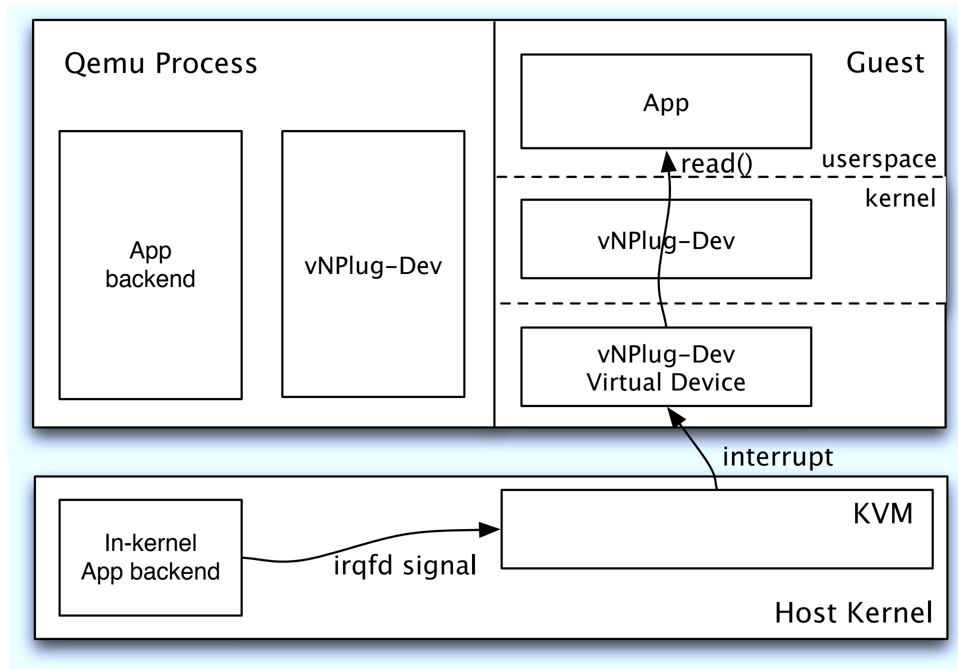


Figure 3.5: A possible configuration for host-to-guest notifications.

a write operation at that address. Moreover, it is possible to assign a different *eventfd* to each different value to match.

This is particularly efficient because, while a normal I/O operation on an emulated QEMU virtual device requires a heavy VM exit, going back to the QEMU user-space to synchronously serve the request by means of a user-space handling routine, this mechanism allows a lightweight exit, long enough to signal an *eventfd* in kernel-space by means of a KVM service routine.

Applications on the host side can use the *eventfd* to get notified, from both user-space or kernel-space.

Instead, on the guest side, we just need to map the MMIO region in user-space, the same way we do for the shared memory, notifying events via normal writes at a known address. Events can be identified depending on the value of the write, setting the relationship between value to match and *eventfd* into the *ioeventfd* support.

Figure 3.6 on the following page shows an example of notification with *ioeventfd*

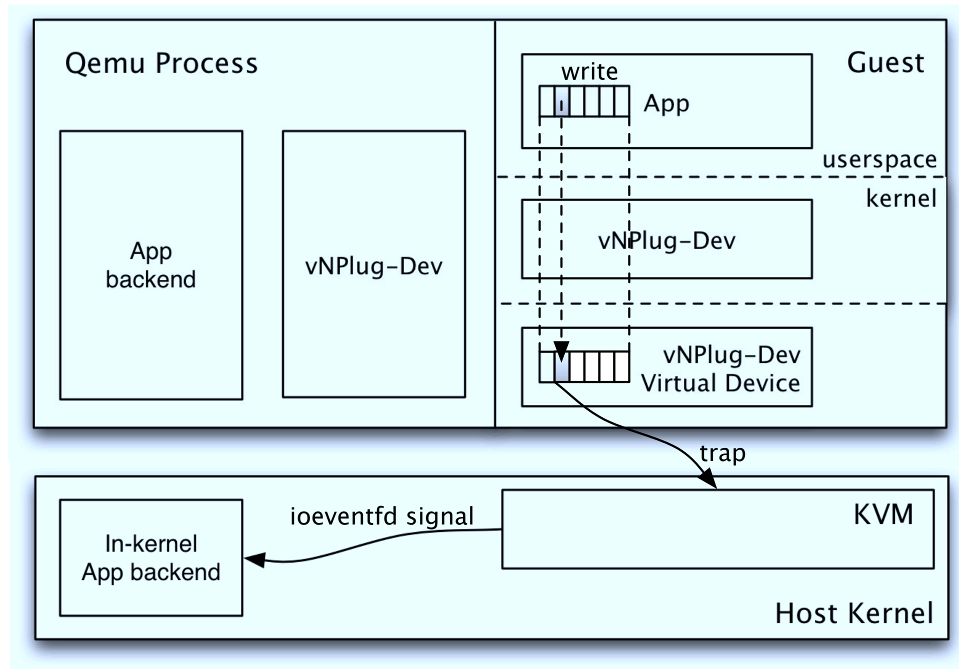


Figure 3.6: Guest to host notifications with in-kernel polling.

and in-kernel polling.

3.4 vNPlug-CTRL: control messages over VirtIO

This component shall provide a communication channel, between the guest side of the monitoring application and its back-end in the host, in order to exchange control messages to coordinate the two sides. First of all, this can be useful for an application on the guest side to require the backend to setup a new shared memory, but we can find a lot of other examples.

Actually, we may consider to use a common network connection. The reason this component has been introduced, in our design, is that we want something more reliable. For instance, with a network connection, a user might unintentionally make changes to the network interface, compromising so the correct behaviour of the framework.

Chapter 2 introduced Virtio, the support for paravirtualized devices, and briefly described its transport mechanism. Besides being efficient and ensuring low re-

sponse times, it requires a little more effort at development time with respect to a network communication. Thus, the choice of this support would fit our requirements.

The two-way communication channel over Virtio can use two *virtqueue*s, one for host-to-guest messages and one for the counter-direction. In order to send and receive messages from the guest user-space, the most convenient solution is to expose common file operations on a character device, such as *read* and *write*.

3.4.1 Routing messages

Through the communication channel, the framework handles the routing of messages between host-side and guest-side of applications. Of course, it may support multiple applications, and each may need multiple virtual devices. This means that each application needs a unique identifier, and the same holds for each virtual device. Furthermore, the framework may need to coordinate the two sides of itself.

Therefore, these considerations suggest to use something similar to a minimal protocol stack. At the bottom, the Virtio transport mechanism takes place, providing a two-way point to point communication between the two sides of the framework, guest side and host side. At the second layer, a framework-level header allows the framework to distinguish between messages addressed to itself and messages addressed to an application. At the third layer, an application-level header allows the framework to identify the application. From the fourth layer on, all is managed by the application, to identify internal operations and to address virtual devices.

Figure 3.7 on the next page shows an example of routing with a control message from an application instance to its back-end. Appendix B.1.1 gives an example of implementation of framework-level and application-level headers.

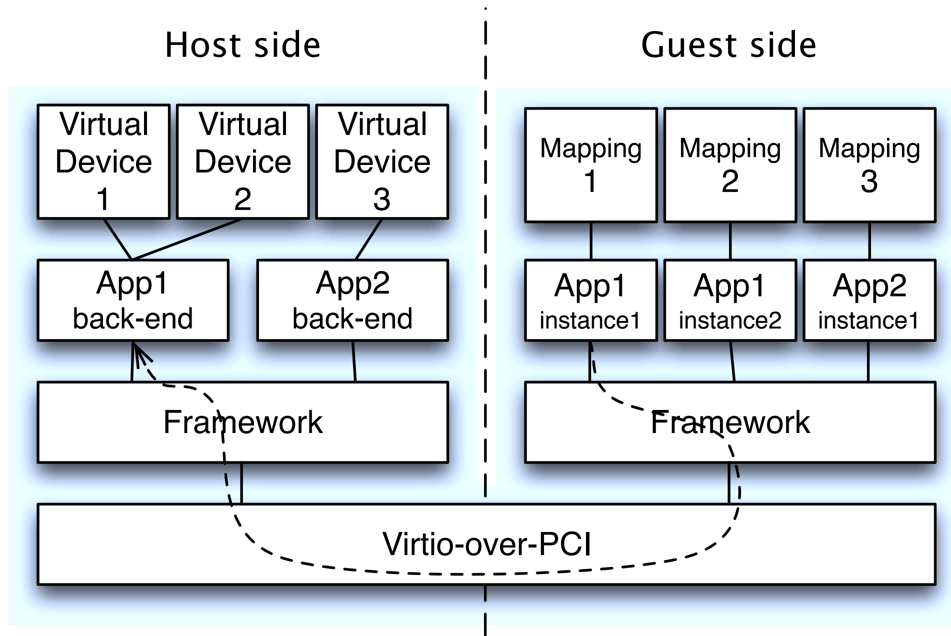


Figure 3.7: Control messages routing using an approach similar to a protocol stack.

3.5 vNPlug interface

In order to simplify and encourage the development of monitoring applications, the framework should provide a simple API, supplying an high level abstraction of the implemented functions. Framework's components get abstracted through two subsets of the interface: the host side API and the guest side API.

3.5.1 Host side API

The features that the host side interface must provide are:

- Registration and unregistration of the application back-end, with a unique identifier.
- Control messages reception and transmission.
- Shared memory creation and removal, given:
  - the user-space virtual address and the size of the memory area to map.

  – the number of host-to-guest and guest-to-host events.

  This functionality must return the virtual device unique identifier, and the *eventfd*s used by the application to signal and wait for events.

3.5.2 Guest side API

The features that the guest side interface must provide are:

- Control messages transmission and reception, using a unique identifier to address the application back-end.
- Shared memory mapping and unmapping in the virtual address space of the application, given the virtual device identifier.
- Event-signalling and event-waiting functionalities.

Figure 3.8 on the following page shows the registration of an application to the framework, and the creation/removal of a virtual device.

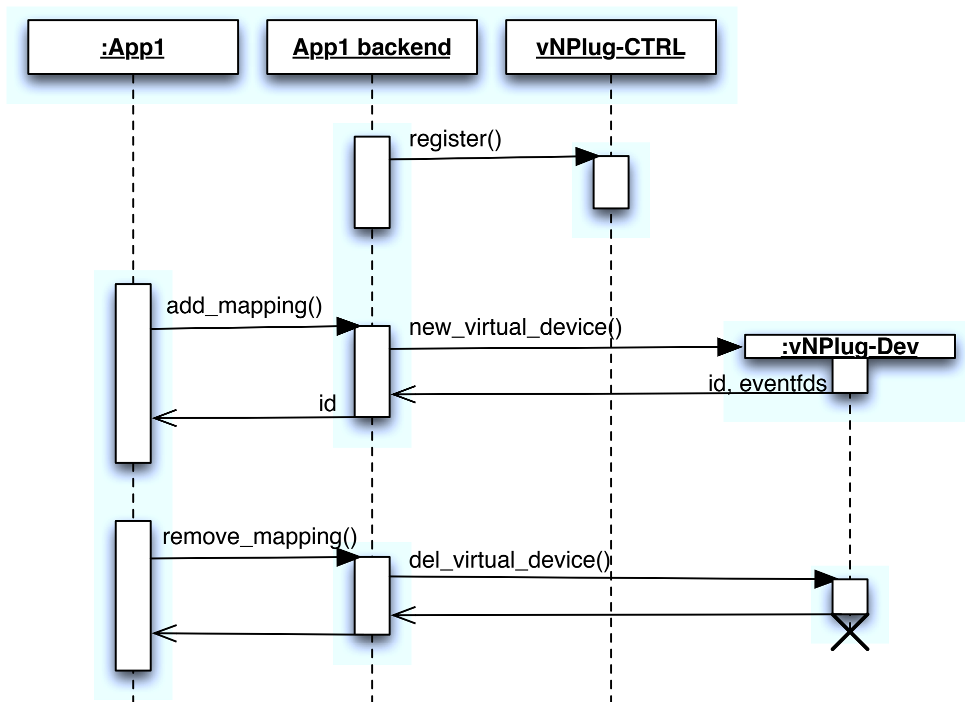


Figure 3.8: Registration of an application and mapping creation/removal.

Chapter 4

Validation with PF_RING

The previous chapter introduced the vNPlug framework. This chapter will present Virtual PF_RING, an application designed on top of vNPlug, which is based on the PF_RING packet capture accelerator that we are going to introduce. Virtual PF_RING represents a use case example for the framework, and allows us to have an idea of the framework performance. Then, the thesis requirements will be validated.

4.1 PF_RING: a kernel module for packet capture acceleration

PF_RING [16] is a kernel-based extensible traffic analysis framework, that significantly improves the performance of packet capture. It reduces the journey of a packet from the wire to user-space, with different working modes, which differ on their level of optimization.

PF_RING can use both standard drivers or PF_RING-aware enhanced drivers. In order to achieve high packet capture performance, it is necessary to modify the implementation of network card drivers, because they are not optimized for packet capture. In fact, usually when a packet is received, a new socket buffer is allocated

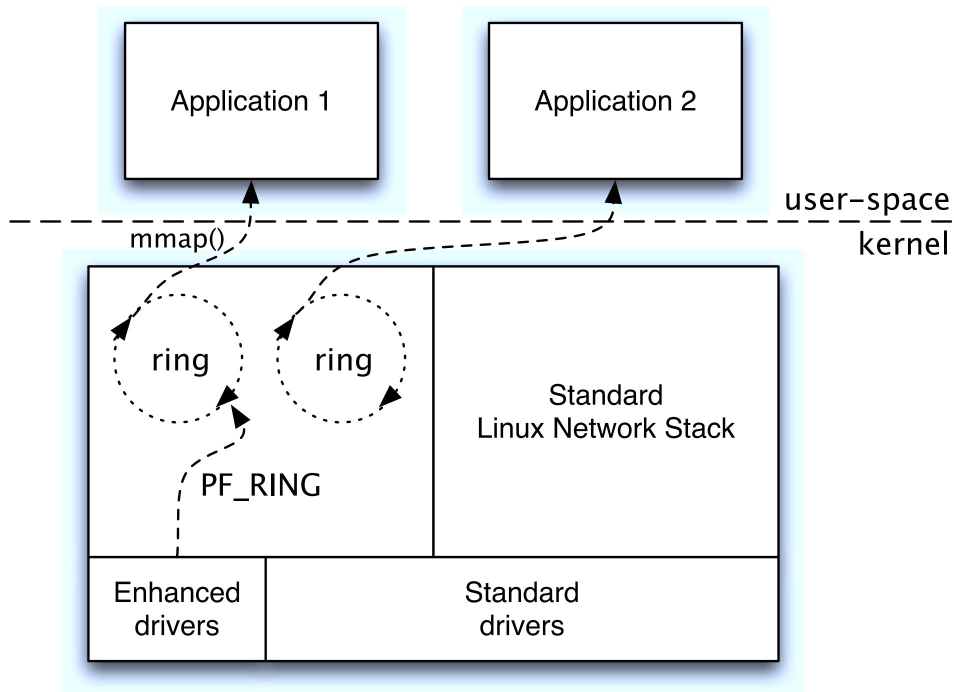


Figure 4.1: PF_RING design.

and queued in kernel structures. Then, if the packet is handled by PF_RING, it gets copied into a ring buffer and discarded by the kernel. An optimized driver can push packets directly into the PF_RING buffer, bypassing the standard kernel path.

PF_RING takes advantage of many other sophisticated mechanisms, to accelerate the packet capture, such as the TNAPI (Threaded NAPI), which polls packets simultaneously from each reception queue, and the DNA (Direct NIC Access), which is a way to map NIC memory and registers to user-space.

PF_RING has a rich support for packet filtering. It also supports hardware filtering using commodity network adapters, of course when supported.

Moreover, PF_RING can implement advanced packet parsing and filtering by means of dynamically-loadable kernel plugins, which can elaborate packets directly at the kernel layer without copying packets to user-space.

The PF_RING framework provides a user-space library that exposes an easy-to-use API to monitoring applications. Through this library, ring buffers are di-

rectly mapped from kernel-space into user-space by using a *mmap*, reducing the number of 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.
- if 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.

Actually, this algorithm uses a slight variant, an "adaptive sleep", to avoid many systems calls when the application consumes packets too quickly. This variant performs an active wait instead of a *poll*, with a sleep interval that is changed according to the incoming packet rate, allowing the kernel to copy several packets into the ring during it.

4.2 Virtual PF_RING

The aim of Virtual PF_RING is to validate the vNPlug framework, and to demonstrate that a packet capture library can be ported in a virtual environment with very little effort using this method.

First of all, the whole vNPlug framework has been developed following the design choices described in Chapter 3. The implementation, on the host side in QEMU, includes:

- the vNPlug-CTRL component. This component registers a virtual PCI device via QDev, and uses the Virtio support with two *virtqueue*s to allow the two-way communication channel. Appendix B.1.1 shows the device creation and the *virtqueue*s initialization.
- the vNPlug-Dev component. This component can be used to register multiple virtual PCI devices via QDev, on demand, at runtime. Each virtual device uses a memory region to map memory from the host, and a memory region

for registers (status/mask for standard IRQ, virtual device identifier, size of the mapped memory, guest to host notifications via *ioeventfd*). An additional region can be used for MSI. Multiple events are supported, in both directions, depending on the applications requirements. Appendix B.2.1 shows the way a new virtual device is added using the QDev support, and the way the *ioeventfd* and *irqfd* supports are set.

- an interface implementing the host-side API. Appendix B.1.1 shows the way applications get registered.

Instead, on the guest side, the implementation includes:

- a kernel module responsible of:
  - the creation of a character device for allowing a user-space process to send and receive messages over Virtio via common *read* and *write* file operations. Appendix B.1.2 contains an example of adding a message to a *virtqueue*.
  - the creation of a character device for each virtual PCI device connected to the VM, in order to: map the shared memory via *mmap*, wait for interrupts via blocking *read*, and signal events by writing to the registers region. Of course hotplug is also supported, by loading the *acpiphp* kernel module.

  Appendix B.2.2 shows the registration of this module inside the Linux kernel.

- a slight user-space library, implementing the guest-side API and mediating between the above mentioned character devices and the applications.

PF_RING has been redesigned, or rather we should say “slightly modified”, to fit the model imposed by the vNPlug framework.

4.2.1 Design

The design of PF_RING lends itself particularly well to be adapted to the vN-Plug framework. In fact, on the host side, it only needs a few enhancements, keeping both the kernel module and the user-space library fully backward-compatible.

The PF_RING library uses a *mmap* to export the ring from kernel-space into user-space. In order to map this memory area into the guest, the same virtual address, returned by the *mmap*, can be passed to the framework. The framework will perform the "dirty work", making the memory area available into the guest user-space.

In order to replace the *poll*, it is possible to use the two-way event signalling support. When an application on the guest-side wants to read a new packet, but no packets are found into the ring, the library on the guest-side informs the host side that it wants to be alerted if a new packet arrives. 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. An algorithm similar to the "adaptive sleep" of the PF_RING library can be used, in order to avoid many *poll*-equivalent calls.

So, Virtual PF_RING, needs one event for each direction. The *eventfds*, created by the framework, can be used to signal an event or to wait for one, directly from the PF_RING module, inside the kenel.

The Virtual PF_RING back-end, on the host-side, is also responsable of registering the application to the framework, and of translating guest-to-host control messages into calls to the PF_RING library.

Instead, on the guest-side, a new and very slight library has been created, which translates each calls to the PF_RING library into control messages, in addition to the memory mapping and event signalling/waiting described above.

Figure 4.2 on the next page depicts the overall design of Virtual PF_RING.

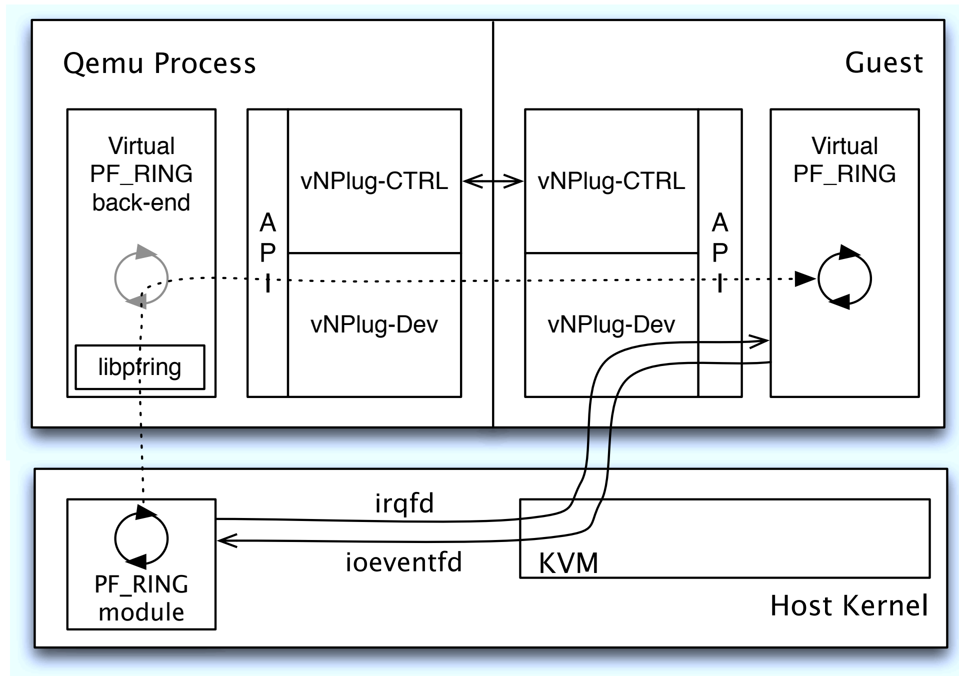


Figure 4.2: Virtual PF_RING design.

4.2.2 Performance evaluation

After the example with Virtual PF_RING, it should be quite clear that the framework provides high flexibility and reduces development of efficient monitoring applications. Now, we want to evaluate its ability to cope with high packet rates, and to compare its performance with pre-existing (software-based) neworking supports.

During the preliminary test phase *pktgen* [33], the Linux packet generator, has been used. *Pktgen* is a testing tool included in the Linux kernel, which can be used to generate packets at high speed, configuring things like: headers content, packet size, delay between packets, number of packets to send, etc. It is not a tool for precise measurements, but it has proved useful testing the framework's correctness.

Performance evaluations have been conducted building a simple testbed (shown in Figure 4.3 on the facing page), which consists of:

- an IXIA [24] 400 Traffic Generator with two Gigabit Ethernet ports

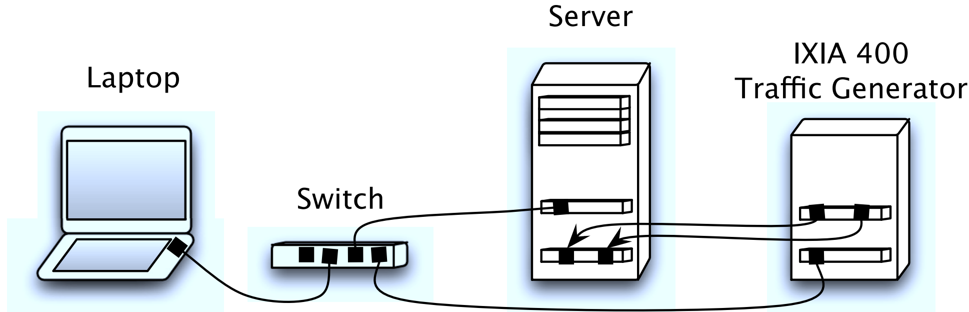


Figure 4.3: Testbed topology.

- a server equipped with:
  - an entry-level Quad-Core Intel Xeon at 2.5GHz
  - 4Gb of memory
  - an Intel 82576 Gigabit Ethernet Controller Dual-Port
  - an Intel 82574 Gigabit Ethernet Controller
  - a Linux 2.6.36 kernel
- a laptop

The traffic generator is connected to the server through the two Gigabit Ethernet ports of the Intel 82576. The laptop has the purpose of managing the traffic generator and the main computer, so it is connected via Ethernet to the management port of the IXIA 400 and to the Intel 82574.

The IXIA 400 is able to generate traffic at wire-rate on each port, and it is possible to configure the precise rate and a lot of other settings, such as the header fields, the packet size, the payload content, etc.

In order to test the framework performance, we will use Virtual PF_RING. The performance of Virtual PF_RING will be compared with the performance of PF_RING running on a native (non virtualized) environment. Virtual PF_RING will be also compared to PF_RING running on a virtual environment, using the Virtio-Net support (described in Chapter 2) with the VHostNet optimization. The

Table 4.1: Maximum rates for Gigabit Ethernet.

Packet size in bytes	Max pkts/s
64	1,488,095
128	844,594
256	452,898
512	234,962
1024	119,731

aim is to reason at the worst case for the framework, and at the best case without the framework (in any case without hardware support for networking). The same server is used to obtain performance results for both the native and the virtualized environment.

The device driver used on the server on the host-side is the *igb*, developed by Intel, which is included in the Linux kernel. Note that, although PF_RING supports optimized drivers to bypass the standard operating system's mechanisms described in Chapter 2, they were not used because we do not want to rely on particular supports and we want to reason at the worst case.

All the VMs we will use have a single virtual CPU and 512Mb of memory, and run Linux with a 2.6.36 kernel.

Before evaluating performance, it is worth mentioning the most important parameter, the packet size. Packet size is relevant because, at wire-rate, the smaller the size, the higher the number of packets, then the higher the overhead due to handling. Gigabit Ethernet is built on top of the Ethernet protocol, better described in Appendix A, keeping compatibility with pre-existing Ethernet standards, like Fast Ethernet. The maximum theoretical rate can be calculated using the frame size and the IFG (Inter Frame Gap). Table 4.1 shows the maximum rates with respect to the frame sizes that will be used during the tests.

Another aspect it is worth to note, is that, thanks to the framework's design, it is possible to use efficient packet filtering techniques within the host (in kernel-space or even in hardware), further increasing performance in a real case. In

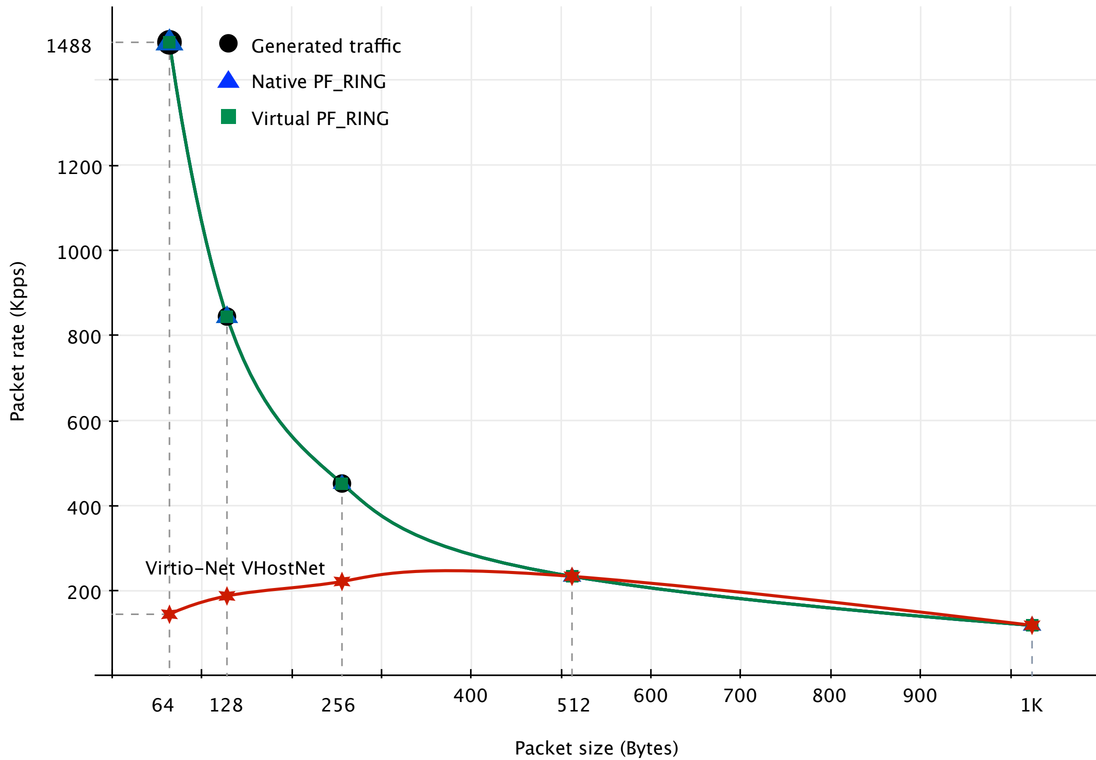


Figure 4.4: Number of packets processed per time unit (1x Gigabit Ethernet).

fact, through the efficient communication channel provided by the vNPlug-CTRL component, Virtual PF_RING is able to instruct the PF_RING module to set a varity of efficient filters.

In order to evaluate performance we will use *pfcount*, a simple application running on top of PF_RING, which captures packets, updates some statistics, and then discards them without doing any analysis.

As a first test, we want to see the performance when a single application instance is processing one Gigabit of traffic, with several packet sizes.

First of all we want to compare the number of packets processed per time unit. Results, in Figure 4.4, show that Virtual PF_RING, like PF_RING in a native environment, is able to process packets at wire-rate, for every packet size, up to the maximum rate for Gigabit Ethernet which is 1.488 million packets per second. Instead we can observe that, without the framework, using PF_RING in a virtual environment with the Virtio-Net support, it is possible to capture all

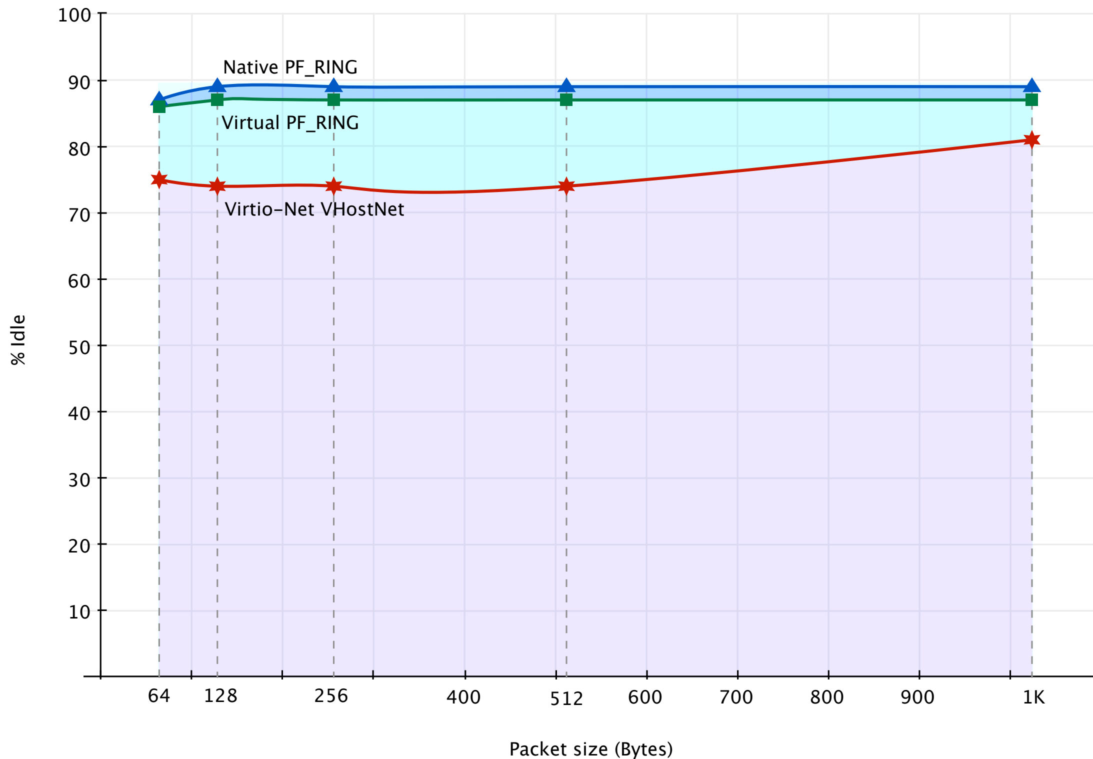


Figure 4.5: Percentage of idle time (1x Gigabit Ethernet).

packets crossing the wire only up to a few hundred thousand of packets per second, then performance gets even worse (it seems to suffer from a problem similar to the "congestion collapse" phenomena described in Chapter 2, but at a different layer).

Another aspect it is worth considering to evaluate the system overhead is the percentage of CPU idle time. Figure 4.5 shows that Virtual PF_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 more overhead (and we should remember that fewer packets per second are processed).

Now let us see Figure 4.6 on the facing page, which contains a more friendly representation of the results: the percentage of packet loss. The reliability of a network monitoring solution depends, first of all, on the percentage of packets it is able to process. In some critical applications, providing wire-speed processing with no packet loss it is going to get mandatory. This figure highlights Virtual

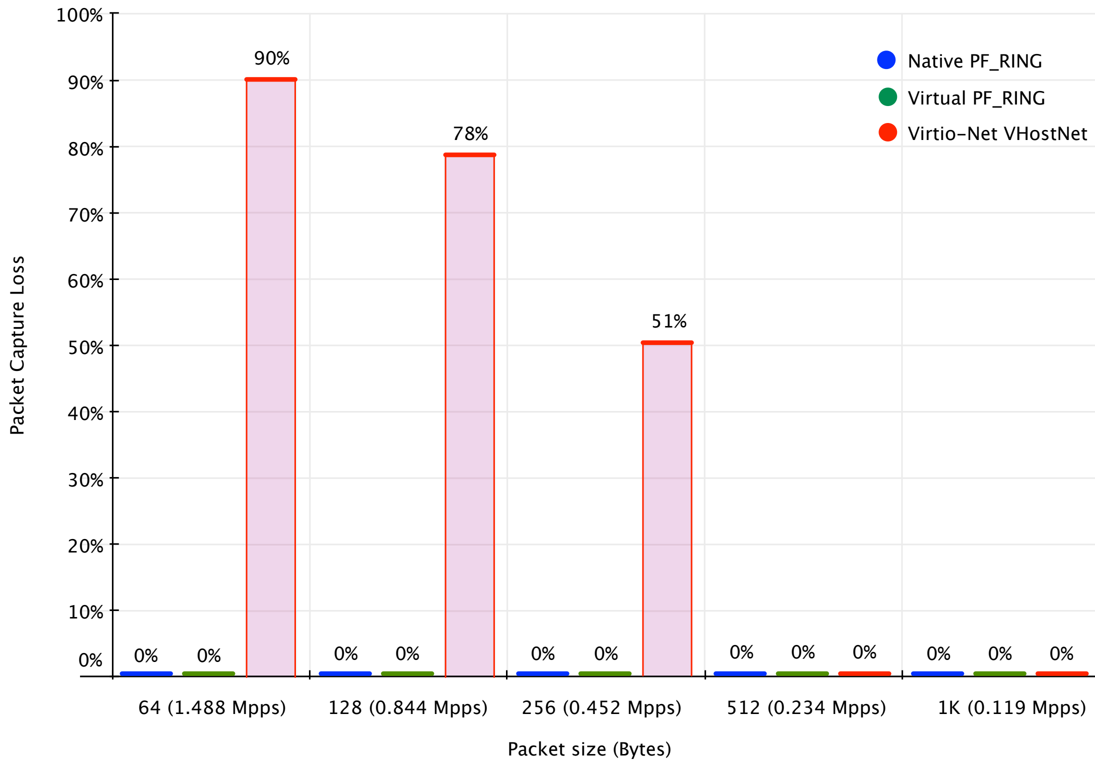


Figure 4.6: Percentage of Packet Capture Loss.

PF_RING, like native PF_RING, experiences no packet loss in any case, even with the highest rate possible. Instead, using Virtio-Net, the percentage of packet loss can reach 90% for high rates.

A second test has been performed to gain an insight into the scalability, evaluating what happens with two instances of the *pfcount* application, each one processing one Gigabit of traffic on a different interface. Both application instances run on the same VM.

Looking at the Figure 4.7 on the next page, that shows the sum of the number of packets processed per time unit by the two application instances, we can see that Virtual PF_RING, like native PF_RING, is able to process up to nearly two million packets per second with no loss (with an average of one million per instance). When the packet rate on the wire further increases, with 64-byte packets at wire-speed, both start to lose packets, but native PF_RING processes about half a million more.

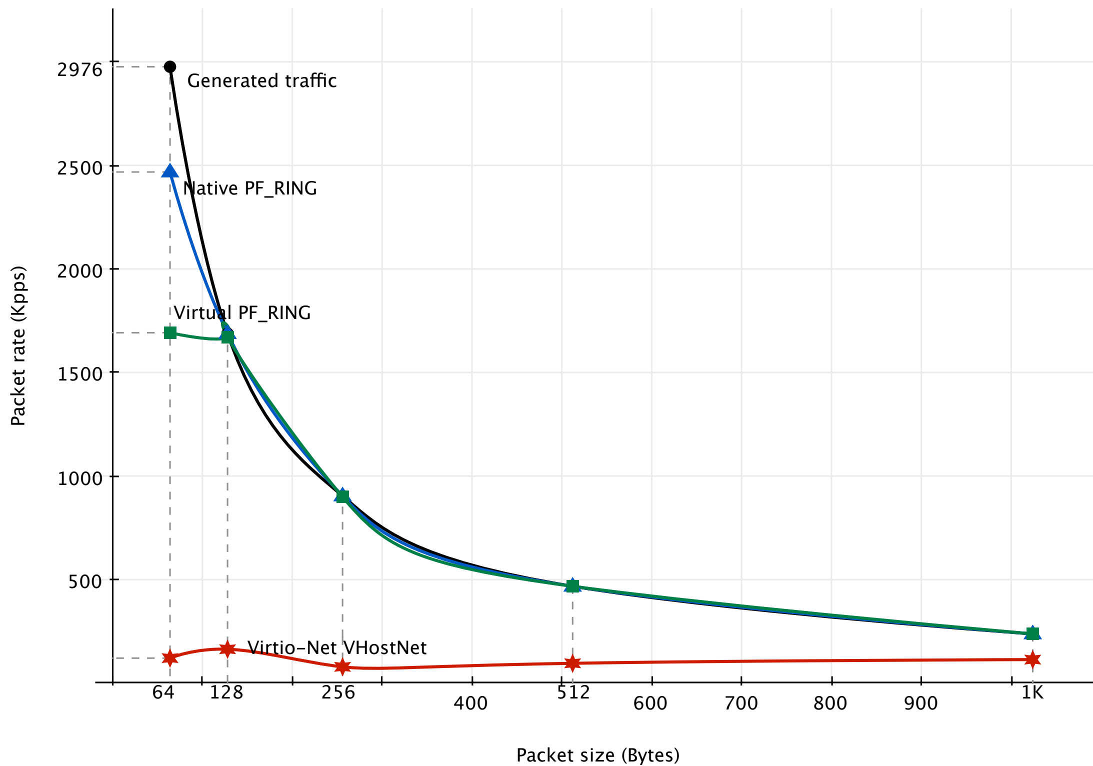


Figure 4.7: Number of packets processed per time unit (1x VM, 2x Gigabit Ethernet).

This does not mean that Virtual PF_RING is unable to scale as much as the native PF_RING, it may be due to the way a virtual machine is executed. In fact, while the two instances of *pfcount* of the native solution can run concurrently on different cores of the SMP, we know that a virtual CPU, where the two application instances of the virtual solution are scheduled on, is itself scheduled as a normal process 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. This means that even with large packets, there is a high percentage of loss.

Figure 4.8 on the facing page, which shows the percentage of CPU idle time, supports our hypothesis about the scalability. In fact, it shows that Virtual PF_RING keeps the CPU relatively idle, with an higher percentage compared to the native PF_RING. The solution based on Virtio-Net continues to require

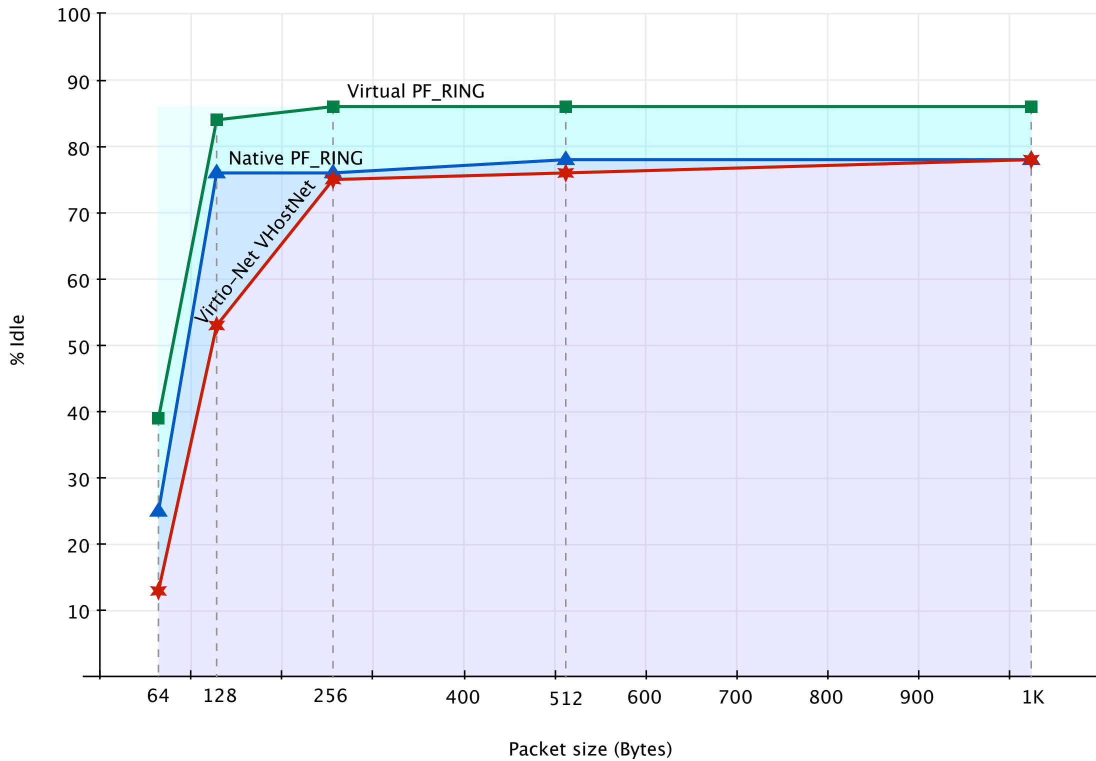


Figure 4.8: Percentage of idle time (1x VM, 2x Gigabit Ethernet).

more overhead, even with a very low percentage of captured packets.

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

In this case, as shown in Figure 4.9 on the next page, the sum of the number of packets processed per time unit by the two application instances follows the same performance as the previous test for lossless packet rates. Instead, in this case, for 64-byte packets at wire-speed, the capture rate of Virtual PF_RING is really close to the capture rate of the native PF_RING. This, once again, supports 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 are similar to the previous ones.

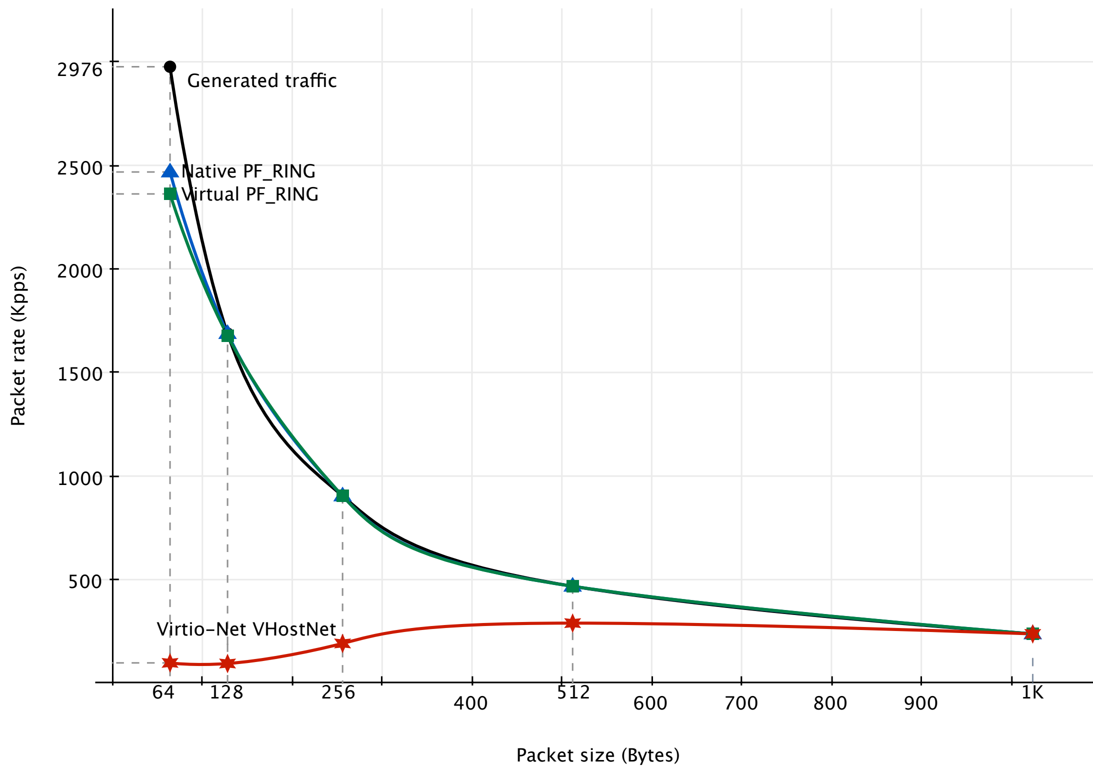


Figure 4.9: Number of packets processed per time unit (2x VM, 2x Gigabit Ethernet).

Figure 4.10 on the facing page shows the percentage of CPU idle time. As one would guess, this time Virtual PF_RING requires more overhead than the native PF_RING. The solution based on Virtio-Net continues to require much overhead, even with very poor performance.

Latency would be another significative measure which assumes considerable relevance in some critical applications, where an immediate response time is required.

The most precise counter available on x86 architecture is the TSC (Time Stamp Counter), a 64-bit counter that increases at each clock cycle and can be used for accurate latency comparisons. The *rdtsc* instruction returns the TSC value, storing its high part in register *EDX* and its low part in register *EAX*.

Since a KVM guest does not access the host TSC - because it would experience issues during live migration or when the virtual CPU is scheduled on multi-core SMPs - the guest-visible TSC differs from the host TSC for an offset and may experience drifts. Nowadays there are several attempts to improve TSC virtualization.

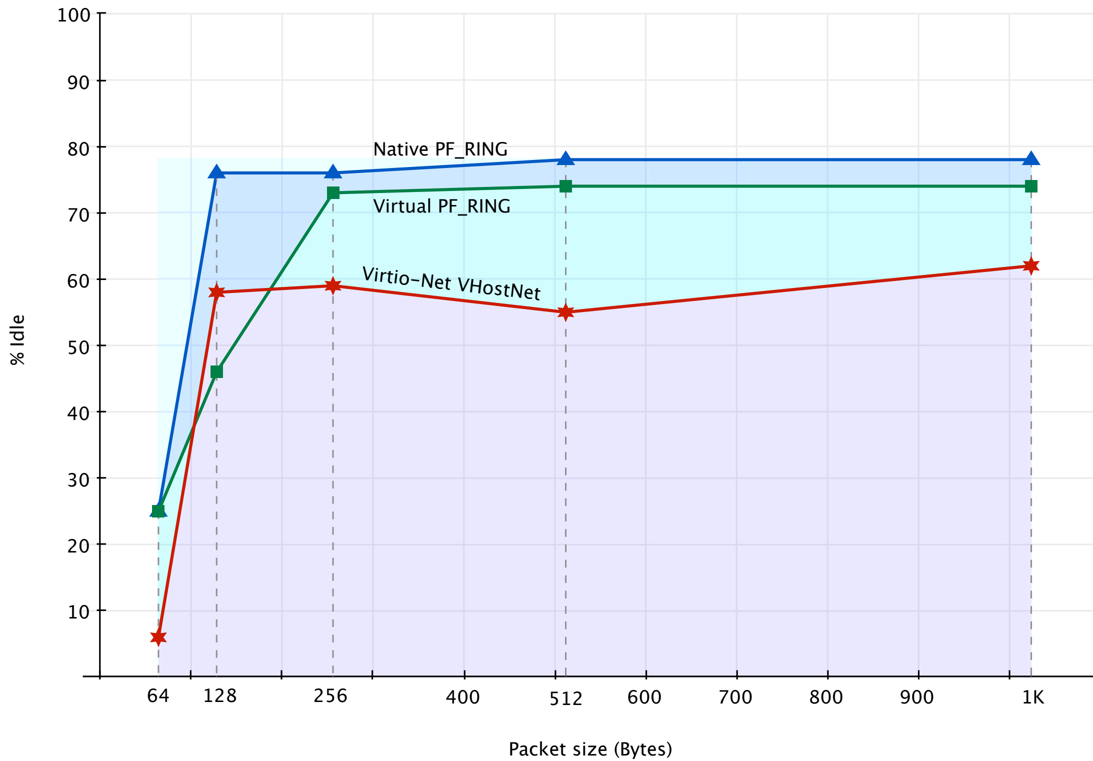


Figure 4.10: Percentage of idle time (2x VM, 2x Gigabit Ethernet).

Latency comparison, in our case, requires fine-grained time measurement, but tests have proved it is not possible to get a perfectly synchronized time. Some tests have been evaluated, printing the TSC when a packet is received by the *igb* driver, and when it is processed by *pfcount*, trying to adjust results with averages and offsets, but it was impossible to stay accurate and significant.

However, we can assume that the latency with Virtual PF_RING is lower than with Virtio-Net, probably close to the native PF_RING, as the path of the packet is drastically cut down.

4.2.3 Validation

In order to validate the proposed method, we shall look at the requirements defined during the analysis, reported in Chapter 1, to see if they are satisfied.

1. High performance

   In the previous section, the performance of a packet capture library, based on

the framework, has been evaluated. Results demonstrate that it is possible to achieve very high performance, with no packet loss even at wire-rate with the smallest packets possible on Gigabit Ethernet. The percentage of CPU idle time demonstrates that also the overhead is optimal, because it is similar to the native case. An aspect it is worth to note, is that, with this method, it is possible to use efficient packet filtering techniques within the host, further improving performance and CPU utilization in a real case.

2. Flexibility

   The framework provides applications an high degree of freedom, with a very flexible API. In fact, the API supplies simple but effective tools, everything a monitoring application may need to design an efficient infrastructure for packet capture:

   - it allows applications to share, at run-time, multiple buffers of arbitrary size between host-side and guest-side;
   - it provides an efficient event signalling support, which is also flexible thanks to the *eventfd* mechanism (events can be signalled/waited from both user-space and kernel-space);
   - it provides a reliable and efficient communication channel to coordinate host-side and guest-side.

3. Dynamic VM Reconfiguration

   Applications can register to the framework at run-time. New virtual devices can be dynamically plugged in, mapping new memory, without compromising the behaviour of the VM, or of other applications, or of other virtual devices.

4. Scalability

   Actually, the number of concurrent applications is not limited, as well as the number of virtual devices. Each virtual device has an independent sup-

port. The only support which is shared among applications, on the same VM, is the control communication channel, which should not produce efficiency problems. So, as verified in the previous section, the only limitation is imposed by resources availability.

5. Ease of use

   The framework simplifies the development of efficient monitoring applications running on VMs, because it provides, through a very simple API, a set of flexible tools for memory mapping, event signalling and control communications, hiding what is behind the scene of a virtual environment. The Virtual PF_RING example showed that designing and developing an application using the framework requires very little effort.

6. Open Source

   The framework is based on an Open Source virtualization solution: Linux and QEMU/KVM.

7. Commodity hardware

   The framework does not rely on specialised monitoring hardware, thanks to its design this choice is left to applications. During the performance evaluation phase, described in the previous section, only commodity hardware has been used, achieving anyway very good results.

Chapter 5

Final Remarks

There have been many efforts in recent years to improve network performance on VMs, both with hardware and software solutions. In previous chapters we spoke about paravirtualization, self-virtualized devices for direct I/O, and various hardware supports. However, none of these address the problem of using VMs for high-performance network monitoring.

This thesis has presented the idea of hypervisor-bypass, which allows packets to follow a straight path from the kernel to an application running on a VM, avoiding the involvement of the hypervisor and the intervention of the operating system standard mechanisms (completely on the guest-side, partially on the host-side).

Our performance evaluations showed that this method can significantly improve performance, which are close to native under most circumstances, while current virtualization approaches have proven ill suited to be used with network monitoring applications when wire-rate packet capture is required.

The fact that the hypervisor, in order to maximize performance, lets a guest access some resources directly, is also visible in the case of the CPU. With native execution, which is the virtualization technique used by KVM, a VM can execute all non-privileged instructions natively without intervention of the hypervisor. Instead, the execution of a privileged instruction will trigger a trap, allowing the hypervisor to make sure that the execution can continue without compromising

isolation and system integrity.

Since most CPU-intensive activities of a monitoring application seldom use privileged instructions, they can achieve near-native performance even when executed on a VM. This means that, as clearly shown by the results in this study, removing bottlenecks in packet capture, it is possible to collapse the gap between performance achieved by virtualized network monitoring applications running in VMs and performance achieved by applications running natively, opening new scenarios.

In regard to commercial solutions, Endace [18], leader in high-speed packet capture and analysis, recently introduced its virtualized solution, advertised as "100-Percent Packet-Capture". The Endace DOCK Network Monitoring Platform allows multiple simultaneous analytics applications to be all hosted within the same platform, including also 3rd party applications. This solution leverages specialized network monitoring hardware, designed specifically for packet capture. Mike Riley, chief executive officer at Endace, promoting the new virtualized solutions, said *"By separating hardware from software, organizations can fundamentally change the way that they measure and manage their networks. Organizations need the ability to run their own proprietary applications in a managed environment with the ability to work with the best application vendors in the market. No organizations should be beholden to a single application vendor. The days of the point solution are gone."*

5.1 Open issues and future work

The current framework implementation already represents an efficient and flexible solution, providing all the supports described in Chapter 3 during the design phase. The framework, togheter with Virtual PF_RING, gives us a complete packet capture solution, which benefits of the improvements made to PF_RING (just at the price of keeping the slight translation layer up-to-date).

An issue is the live migration, in fact 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. Developing the framework we focused on basic functionalities without attention to live migration, so in future we will work on such support.

As already discussed, also isolation remains an open issue, because the use of resources by applications, which have an high degree of freedom, is not directly controlled by the framework.

Furthermore, it would be interesting to carry out more detailed performance evaluations, to look for other possible improvements, and to test the framework in VMs with multiple virtual CPUs investigating on scheduling and resource management. It would also be interesting to evaluate the framework's performance on a 10 Gigabit network in order to evaluate its scalability this way.

Appendix A

Ethernet Basics

Ethernet is made up of a number of components which operate in the lower two layers of the OSI model: the Data Link layer and the Physical layer.

Ethernet separates the Data Link layer into two distinct sublayers: the LLC (Logical Link Control) sublayer and the MAC (Media Access Control) sublayer. The IEEE 802.2 standard describes the LLC sublayer, and the IEEE 802.3 standard describes the MAC sublayer and the Physical layer.

The MAC layer uses the CSMA/CD (Carrier Sense Multiple Access with Collision Detection) to send packets around an Ethernet network. A packet on an Ethernet link is called frame, and consists of the following fields (see Figure A.1):

- Preamble (8 bytes)

  The Preamble field contains a synchronization pattern consisting of of 7 in-

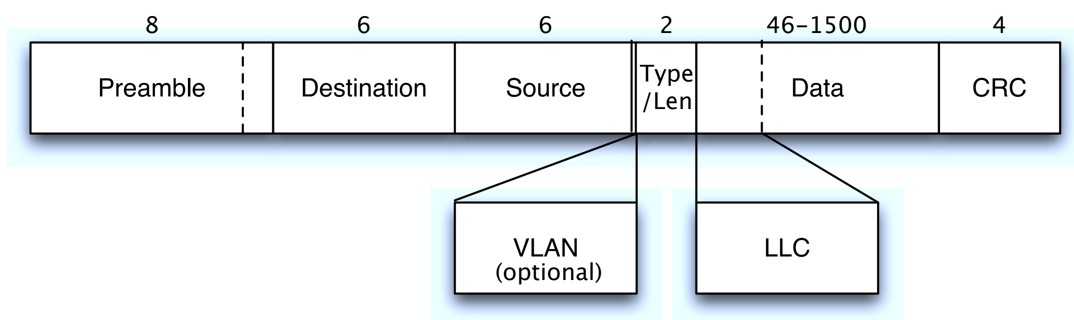


Figure A.1: Ethernet frame format.

stances of 0101 0101, and the Start of Frame Delimiter 0101 1101.

- Destination address (6 bytes)

  The Destination address field contains the address of the Ethernet interface card to which the frame is directed.

- Source address

  The Source address field contains the address of the station that is transmitting the frame.

- Optional: VLAN Tag (4 bytes)

  This optional field contains the IEEE 802.1Q VLAN tag, indicating VLAN (Virtual LAN) membership.

  The first 2 bytes of the tag are the TPID (Tag Protocol Identifier) value, 0x8100, which is necessary to identify the presence of the VLAN tag (it is located in the same place as the Type/Length field). The remaining 2 bytes contain the TCI (Tag Control Information), Quality of Service and VLAN ID.

- Type/Length field (2 bytes)

  The Type/Length needs to be interpreted. The Ethernet II specification uses this Type field to identify the upper layer protocol, for example a value of 0x0800 signals that the frame contains an IPv4 datagram. The IEEE 802.3 specification replaces the Type field with the Length field, used to identify the length of the Data field. The protocol type in IEEE 802.3 frames is moved to the data portion of the frame.

  Since both formats are in use, in order to allow frames to coexist on the same Ethernet segment, Type values must be greater than or equal to 1536. With this convention, since the maximum length of the data field of an IEEE

802.3 frame is 1500 bytes, it is possible to determine whether a frame is an Ethernet II frame or an IEEE 802.3 frame.

- Data Field (46-1500 bytes)

  The Data field contains the information received from the upper layer.

  The LLC field is present in IEEE 802.3 frames, within the Data field, providing information for the upper layer. It consists of the DSAP (Destination Service Access Point) field, the SSAP (Source Service Access Point) field and the Control field.

  The need for additional protocol information led to the introduction of the SNAP (SubNetwork Access Protocol) header, an extension to the LLC field, indicated by the SSAP and DSAP addresses with a value of 0xAA. The SNAP header is 5 bytes, 3 bytes for the organization code assigned by IEEE, 2 bytes for the type set from the original Ethernet specifications.

- CRC Field.

  The CRC (Cyclic Redundancy Check) field contains a cyclic redundancy check to detect corrupted data within the entire frame.

The minimum size of a packet, without considering the preamble which is added by the card, is 64 bytes.

Calculating the maximum number of packets crossing a link per time unit, we have to consider the Preamble, but also the IFG (Inter Frame Gap), the minimum gap between frames (12 bytes).

Appendix B

Code snippets

This appendix contains code snippets, extracted from the framework sources, for a better and more pratical understanding of the implementation.

B.1 vNPlug-CTRL

B.1.1 Host side

Listing B.1: Registration of an application back-end to the framework

```
static struct vNPlugCTRLClientInfo vapp_client_info = {
    .id          = VNPLUG_CLIENT_ID_VAPP,
    .name        = "vapp",
    .msg_handler = vapp_ctrl_msg_handler,
};

static void vapp_register(void)
{
    vnplug_ctrl_register_client(&vapp_client_info);
}

device_init(vapp_register);
```

Listing B.2: Control message headers, framework layer and application layer

```
#define VNPLUG_CTRL_MSG_TYPE_FWD 0
#define VNPLUG_CTRL_MSG_TYPE_LOG 1

struct vnplug_ctrl_msg_fwk_hdr {
    /* message type */
    uint32_t type;
};

struct vnplug_ctrl_msg_app_hdr {
    /* application id */
    uint32_t        id;

    /* payload length*/
    uint32_t        payload_len;

    /* message data */
    char            payload[0];
};
```

Listing B.3: Registration of a virtual device and initialization of the two-way communication channel over Virtio

```
VirtIODevice *vnplug_ctrl_init(DeviceState *dev)
{
    VirtIOvNPlugCTRL *v;

    v = (VirtIOvNPlugCTRL *) virtio_common_init(
        "vnplug-ctrl",
        VIRTIO_ID_VNPLUG_CTRL,
        sizeof(struct vnplug_ctrl_virtio_config),
        sizeof(VirtIOvNPlugCTRL));

    v->vdev.get_config   = vnplug_ctrl_get_config;
    v->vdev.set_config   = vnplug_ctrl_set_config;
    v->vdev.get_features = vnplug_ctrl_get_features;
    v->vdev.set_features = vnplug_ctrl_set_features;
    v->vdev.bad_features = vnplug_ctrl_bad_features;
    v->vdev.reset        = vnplug_ctrl_reset;
    v->vdev.set_status   = vnplug_ctrl_set_status;

    v->h2g_vq = virtio_add_queue(
        &v->vdev,
        VNPLUG_CTRL_VQ_SIZE,
        vnplug_ctrl_handle_h2g);
    v->g2h_vq = virtio_add_queue(
        &v->vdev,
        VNPLUG_CTRL_VQ_SIZE,
        vnplug_ctrl_handle_g2h);

    register_savevm(dev, "vnplug-ctrl", -1, VNPLUG_CTRL_VM_V,
        vnplug_ctrl_save, vnplug_ctrl_load, v);
    v->vm_state = qemu_add_vm_change_state_handler(
        vnplug_ctrl_vmstate_change, v);
```

```
    return &v->vdev;
}

static int vnplug_ctrl_init_pci(PCIDevice *pci_dev)
{
    VirtIOPCIProxy *proxy = DO_UPCAST(
        VirtIOPCIProxy, pci_dev, pci_dev);
    VirtIODevice *vdev;

    vdev = vnplug_ctrl_init(&pci_dev->qdev);

    virtio_init_pci(
        proxy,
        vdev,
        PCI_VENDOR_ID_VNPLUG_CTRL,
        PCI_DEVICE_ID_VNPLUG_CTRL,
        PCI_CLASS_VNPLUG_CTRL,
        0x00);

    proxy->nvectors = vdev->nvectors;
    return 0;
}

static PCIDeviceInfo vnplug_ctrl_info = {
    .qdev.name  = "vnplug",
    .qdev.size  = sizeof(VirtIOPCIProxy),
    .init       = vnplug_ctrl_init_pci,
    .exit       = vnplug_ctrl_exit_pci,
    .qdev.props = (Property[]) {
        DEFINE_VIRTIO_COMMON_FEATURES(
            VirtIOPCIProxy,
            host_features),
        DEFINE_PROP_END_OF_LIST(),
    },
```

```
    .qdev.reset = virtio_pci_reset,
};

static void vnplug_ctrl_register_devices(void)
{
    pci_qdev_register(&vnplug_ctrl_info);
}

device_init(vnplug_ctrl_register_devices);
```

B.1.2 Guest side

Listing B.4: Virtio device driver: sending a control message on the guest-to-host virtqueue

```
static int32_t vnplug_ctrl_virtio_send_msg(
    struct vnplug_ctrl_info *vi,
    struct scatterlist sg[],
    struct vnplug_ctrl_msg_fwk_hdr *hdr,
    void *payload, uint32_t payload_size,
    void *ret_payload, uint32_t ret_payload_size)
{
    int32_t err;
    uint32_t out = 2, in  = 1;  /* number of "out"/"in" entries */

    sg_init_table(sg, out + in);

    /* "out" entries */
    sg_set_buf(&sg[0], hdr, sizeof(*hdr));
    sg_set_buf(&sg[1], payload, payload_size);

    /* "in" entries */
    sg_set_buf(&sg[out + 0], ret_payload, ret_payload_size);

    err = vi->g2h_vq->vq_ops->add_buf(vi->g2h_vq,
        sg, out, in, vi);

    if (err < 0)
        return err;

    vi->g2h_vq->vq_ops->kick(vi->g2h_vq);

    return 0;
}
```

B.2 vNPlug-Dev

B.2.1 Host side

Listing B.5: Example of adding a new virtual device using QDev

```
DeviceState *qdev;
QemuOpts *opts;

opts = qemu_opts_create(&qemu_device_opts, NULL, 0);

qemu_opt_set(opts, "driver", "vnplug-dev");
qemu_opt_set_uint(opts, "backend_events", 1);
qemu_opt_set_uint(opts, "guest_events", 1);
qemu_opt_set(opts, "msi", "on");
qemu_opt_set(opts, "irqfd", "on");
qemu_opt_set(opts, "ioeventfd", "on");
qemu_opt_set_uint(opts, "vma_size", buffer->info->tot_mem);
qemu_opt_set_ptr(opts, "vma_ptr", buffer->ptr);
qemu_opt_set_ptr(opts, "client_info_ptr", client);

qdev = qdev_device_add(opts);
```

Listing B.6: Setting up the irqfd support of KVM

```
static int set_irqfds(struct vNPlugDev *vndev, int on) {
    int i, err;

    for (i = 0; i < vndev->backend_events; i++) {
        err = kvm_set_irqfd(
            vndev->dev.msix_irq_entries[i].gsi,
            vndev->backend_eventfd[i],
            on);

        if (err < 0)
            return err;
    }

    return 0;
}
```

Listing B.7: Setting up the ioeventfd support of KVM

```
static int set_ioeventfds(struct vNPlugDev *vndev, int on) {
    int i, err;

    for (i = 0; i < vndev->guest_events; i++){
        err = kvm_set_ioeventfd_mmio_long(
            vndev->guest_eventfd[i],
            vndev->reg_base_addr + REG_OFFSET_IOEV,
            i,
            on);

        if (err < 0)
            return err;
    }

    return 0;
}
```

B.2.2 Guest side

Listing B.8: vNPlug-dev device driver: registration of the PCI driver, initialization of the character devices

```
static const struct file_operations vnplug_fops = {
    .owner      = THIS_MODULE,
    .read       = vnplug_read,
    .write      = vnplug_write,
    .mmap       = vnplug_mmap,
    .open       = vnplug_open,
    .release    = vnplug_release,
};

static struct pci_device_id vnplug_pci_ids[] __devinitdata = {
    {
        .vendor     = PCI_VENDOR_ID_VNPLUG_DEV,
        .device     = VNPLUG_DEV_ID,
        .subvendor  = PCI_ANY_ID,
        .subdevice  = PCI_ANY_ID,
    },
    { 0, }
};

static struct pci_driver vnplug_pci_driver = {
    .name           = VNPLUG_DEVICE_NAME,
    .id_table       = vnplug_pci_ids,
    .probe          = vnplug_pci_probe,
    .remove         = vnplug_pci_remove,
};

static int __init vnplug_init_module(void)
{
    int ret;
```

```
/* vNPlug-Dev */

vnplug_major = register_chrdev(0, VNPLUG_DEVICE_NAME, &vnplug_fops);

if (vnplug_major < 0) {
    ret = vnplug_major;
    goto exit;
}

vnplug_class = class_create(THIS_MODULE, VNPLUG_DEVICE_NAME);
if (IS_ERR(vnplug_class)){
    ret = -ENOMEM;
    goto clean_major;
}

ret = pci_register_driver(&vnplug_pci_driver);

if (ret < 0)
    goto class_destroy;

/* vNPlug-CTRL */

vnplug_ctrl_major =
    register_chrdev(0, VNPLUG_CTRL_DEVICE_NAME, &vnplug_ctrl_fops);

if (vnplug_ctrl_major < 0){
    ret = vnplug_ctrl_major;
    goto unregister;
}

ret = register_virtio_driver(&vnplug_ctrl_virtio_driver);

if (ret) {
    goto clean_ctrl_major;
```

```
    }

    return ret;

clean_ctrl_major:
    unregister_chrdev(vnplug_ctrl_major, VNPLUG_CTRL_DEVICE_NAME);
unregister:
    pci_unregister_driver(&vnplug_pci_driver);
class_destroy:
    class_destroy(vnplug_class);
clean_major:
    unregister_chrdev(vnplug_major, VNPLUG_DEVICE_NAME);
exit:
    return ret;
}

module_init(vnplug_init_module);
module_exit(vnplug_exit_module);

MODULE_DEVICE_TABLE(pci, vnplug_pci_ids);
```

Bibliography

[1] K. Adams and O. Agesen. "A comparison of software and hardware techniques for x86 virtualization". In: *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems.* ACM. 2006, pp. 2–13. ISBN: 1595934510.

[2] Z. Amsden et al. "VMI: An interface for paravirtualization". In: *Ottawa Linux Symposium.* Citeseer. 2006.

[3] M. Armbrust et al. "Above the clouds: A berkeley view of cloud computing". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28* (2009).

[4] P. Balaji et al. "High performance user level sockets over Gigabit Ethernet". In: *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on.* IEEE. 2002, pp. 179–186. ISBN: 0769517455.

[5] P. Barham et al. "Xen and the art of virtualization". In: *Proceedings of the nineteenth ACM symposium on Operating systems principles.* ACM. 2003, pp. 164–177. ISBN: 1581137575.

[6] A. Begel, S. McCanne, and S.L. Graham. "BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture". In: *ACM SIGCOMM Computer Communication Review* 29.4 (1999), pp. 123–134. ISSN: 0146-4833.

[7] F. Bellard. "QEMU, a fast and portable dynamic translator". In: *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*. 2005, pp. 41–46.

[8] M. Ben-Yehuda et al. "Utilizing IOMMUs for virtualization in Linux and Xen". In: *Proceedings of the 2006 Ottawa Linux Symposium*. 2006.

[9] Christian Benvenuti. *Understanding Linux Network Internals*. O'Reilly, 2005.

[10] R. Bhargava et al. "Accelerating two-dimensional page walks for virtualized systems". In: *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. ACM. 2008, pp. 26–35.

[11] H. Bos et al. "FFPF: fairly fast packet filters". In: *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation-Volume 6*. USENIX Association. 2004, pp. 24–24.

[12] L. Braun et al. "Comparing and improving current packet capturing solutions based on commodity hardware". In: *Proceedings of the 10th annual conference on Internet measurement*. ACM. 2010, pp. 206–217.

[13] S. Muthrasanallur G. Neiger G. Regnier R. Sankaran I. Schoinas R. Uhlig B. Vembu J. Wiegert D. Abramson J. Jackson. "Intel Virtualization Technology for Directed I/O". In: (2006).

[14] L. Deri. "High-speed dynamic packet filtering". In: *Journal of Network and Systems Management* 15.3 (2007), pp. 401–415. ISSN: 1064-7570.

[15] L. Deri and F. Fusco. "Exploiting commodity multicore systems for network traffic analysis". In: *Unpublished. http://luca. ntop. org/MulticorePacketCapture. pdf* ().

[16] L. Deri et al. "Improving passive packet capture: beyond device polling". In: *Proceedings of SANE*. Vol. 2004. Citeseer. 2004.

[17] L. Deri et al. "Wire-speed hardware-assisted traffic filtering with mainstream network adapters". In: *Advances in Network-Embedded Management and Applications* (2011), pp. 71–86.

[18] Endace. *Endace, world leaders in high-speed packet capture and analysis solutions.* 2011. URL: http://www.endace.com.

[19] K. Fraser et al. "Safe hardware access with the Xen virtual machine monitor". In: *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS).* Citeseer. 2004.

[20] F. Fusco and L. Deri. "High speed network traffic analysis with commodity multi-core systems". In: *Proceedings of the 10th annual conference on Internet measurement.* ACM. 2010, pp. 218–224.

[21] W. Huang et al. "A case for high performance computing with virtual machines". In: *Proceedings of the 20th annual international conference on Supercomputing.* ACM. 2006, pp. 125–134. ISBN: 1595932828.

[22] 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.

[23] S. Ioannidis et al. "xPF: packet filtering for low-cost network monitoring". In: *High Performance Switching and Routing, 2002. Merging Optical and IP Technologies. Workshop on.* IEEE. 2002, pp. 116–120. ISBN: 488552184X.

[24] IXIA. *IXIA, the Leader in Converged IP Testing.* 2011. URL: http://www.ixiacom.com.

[25] Alessandro Rubini Jonathan Corbet Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition.* O'Reilly, 2005.

[26] A. Kivity et al. "kvm: the Linux virtual machine monitor". In: *Proceedings of the Linux Symposium.* Vol. 1. 2007, pp. 225–230.

[27] J. LeVasseur et al. “Standardized but flexible I/O for self-virtualizing devices”. In: *Proceedings of the First conference on I/O virtualization*. USENIX Association. 2008, pp. 9–9.

[28] J. Liu et al. “High performance VMM-bypass I/O in virtual machines”. In: *Proceedings of the annual conference on USENIX*. Vol. 6. 2006.

[29] S. McCanne and V. Jacobson. “The BSD packet filter: A new architecture for user-level packet capture”. In: *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USENIX Association. 1993, pp. 2–2.

[30] N. McKeown et al. “OpenFlow: enabling innovation in campus networks”. In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74. ISSN: 0146-4833.

[31] J. Mogul, R. Rashid, and M. Accetta. “The packer filter: an efficient mechanism for user-level network code”. In: *Proceedings of the eleventh ACM Symposium on Operating systems principles*. ACM. 1987, pp. 39–51. ISBN: 089791242X.

[32] J.C. Mogul and K.K. Ramakrishnan. “Eliminating receive livelock in an interrupt-driven kernel”. In: *ACM Transactions on Computer Systems (TOCS)* 15.3 (1997), pp. 217–252. ISSN: 0734-2071.

[33] R. Olsson. “pktgen the linux packet generator”. In: *Linux symposium*. Citeseer. P. 11.

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

[35] J. Pettit et al. “Virtual Switching in an Era of Advanced Edges”. In: ().

[36] B. Pfaff et al. “Extending networking into the virtualization layer”. In: *Proc. HotNets (October 2009)* ().

[37] I. Pratt and K. Fraser. “Arsenic: A user-accessible gigabit ethernet interface”. In: *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE.* Vol. 1. IEEE. 2001, pp. 67–76. ISBN: 0780370163.

[38] M. Probst. “Dynamic binary translation”. In: *UKUUG Linux Developer's Conference.* Vol. 2002. 2002.

[39] M. Probst. “Fast machine-adaptable dynamic binary translation”. In: *Proceedings of the Workshop on Binary Translation.* Vol. 9. Citeseer. 2001.

[40] H. Raj and K. Schwan. “High performance and scalable I/O virtualization via self-virtualized devices”. In: *Proceedings of the 16th international symposium on High performance distributed computing.* ACM. 2007, pp. 179–188.

[41] K.K. Ram et al. “Achieving 10 Gb/s using safe and transparent network interface virtualization”. In: *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments.* ACM. 2009, pp. 61–70.

[42] R. Russell. “virtio: towards a de-facto standard for virtual I/O devices”. In: *ACM SIGOPS Operating Systems Review* 42.5 (2008), pp. 95–103. ISSN: 0163-5980.

[43] R. Hiremane S. Chinni. “Virtual Machine Device Queues: An Integral Part of Intel Virtualization Technology for Connectivity that Delivers Enhanced Network Performance”. In: (2007).

[44] J.H. Salim, R. Olsson, and A. Kuznetsov. “Beyond softnet”. In: *Proceedings of the 5th annual Linux Showcase & Conference.* 2001, pp. 18–18.

[45] J.R. Santos et al. “Bridging the gap between software and hardware techniques for i/o virtualization”. In: *USENIX 2008 Annual Technical Conference on Annual Technical Conference.* USENIX Association. 2008, pp. 29–42.

[46] J. Shafer. “I/O virtualization bottlenecks in cloud computing today”. In: *Proceedings of the 2nd conference on I/O virtualization*. USENIX Association. 2010, pp. 5–5.

[47] J. Shafer et al. “Concurrent direct network access for virtual machine monitors”. In: *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE. 2007, pp. 306–317.

[48] P. Shivam, P. Wyckoff, and D. Panda. “EMP: zero-copy OS-bypass NIC-driven gigabit ethernet message passing”. In: (2001).

[49] J. Sugerman, G. Venkitachalam, and B.H. Lim. “Virtualizing I/O devices on VMware workstation’s hosted virtual machine monitor”. In: *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. 2001, pp. 1–14.

[50] Z. Wu, M. Xie, and H. Wang. “Swift: a fast dynamic packet filter”. In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association. 2008, pp. 279–292.