A New Network I/O Acceleration Architecture for Virtualized Systems Supporting Large-Scale, Dynamic Cloud Infrastructures

---

Ph.D. thesis

Anna Fischer

October 2012

Abstract

In response to increasing demand for high-performance I/O in virtualized infrastructures, a variety of virtualization-aware network devices have been developed over the last years. However, due to the lack of a standard hardware or software interface for integrating these into a virtualized system and exposing their capability set to an application running inside a virtual machine, it is a challenge for today's cloud computing infrastructure providers to take advantage of high-performance network devices. Existing approaches tie the virtual machine to a particular hardware interface which significantly limits the dynamic and flexible nature of virtual machines in the cloud. This research investigates an I/O architecture which allows advantage to be taken of a variety of network devices by assigning them as "network acceleration engines" (NAEs) to the virtual machine. The main benefit of the proposed design is that the virtual machine remains decoupled from the underlying real network device. We analyse the design and implementation of today's hardware and system software interfaces and our work reveals challenges for using those efficiently and flexibly on a virtualized system. As a result, we propose enhancements to both hardware and software interfaces in order to provide a high-performance I/O path which can be deployed in dynamic, large-scale environments like cloud computing infrastructures. We implement this NAE framework for three different network devices from two different vendors and results of our initial prototype demonstrate that we can efficiently provide a common, vendor-independent virtual network interface on top of a variety of network hardware.

Declaration

I hereby declare that this thesis is my own work and has not been submitted in substantially the same form for the award of a higher degree elsewhere.

Acknowledgements

I would like to thank my colleagues and friends at HP Labs in Bristol, UK, for the support during the four years of doing my PhD. Without their support, I would not have been able to go through with my ideas and finish off writing this thesis.

Most of all I would like to thank my first manager at HP Labs, Peter Toft, and the manager of the Automated Infrastructure Lab, John Manley, for offering me to start the PhD and helping me with funding and with finding a suitable topic. I would like to thank Peter Toft, in particular, for encouraging me to initiate PhD research whilst working at HP Labs and for supporting me all the way.

I would like to thank my colleagues and friends at HP Labs (in no order!), Aled Edwards, Eric Deliot, Alistair Coles, Patrick Goldsack, Simon Shiu, Dirk Kuhlmann, Andrew Farrell, Li Chen, Nigel Edwards, Lawrence Wilcock, Boris Balacheff, Maher R., David Plaquin, Rycharde Hawkes, Ange, Graeme Proudler, Paul Murray, Paul Vickers, Marco C.-M., Keith Harrison, Adrian Baldwin, Natalie Curtis and Richard Stock for all technical discussions and for creating a great work environment that I felt comfortable in. Most of all I would like to thank Aled Edwards for his guidance, his ideas and for passing on some of his excellent knowledge over the last six or seven years.

I furthermore thank Chris Dalton for all his support during the last year when I was writing up and finishing off my PhD thesis. Without his support I would not have been able to finish what I started four years earlier. I appreciate this very much.

Last but not least I would also like to mention a special thanks to my colleague and good friend Adrian Shaw for all those endless technical discussions, and for actually reading my final thesis and giving valuable feedback for the PhD viva.

I hope that, for each and every one of you, I will be able to return the favour one day.

Table of Contents

1. Introduction 8
1.1. Problem Statement 8
1.2. Solution Summary 10
1.3. Contributions 14
2. Background 17
2.1. Virtualization and I/O Virtualization 17
2.2. Network I/O Virtualization 20
2.3. Study of Previous Work in Directly Relevant Areas 32
3. Requirements Analysis 47
3.1. A Hardware/Software Interface to Support Virtualization 47
3.2. Virtualized Infrastructures Supporting Cloud Computing 50
3.3. Initial Design Principles 52
4. Hardware Design Evaluation 57
4.1. Overview 57
4.2. Intel 82599 and Intel 82576 59
4.3. Mellanox ConnectX 61
5. Network Acceleration Engine (NAE) Architecture 65
5.1. Virtualized System Architecture Overview 65
5.2. NAE Guest API 67
5.3. NAE Network Control Domain API 93
5.4. NAE Hypervisor API 95
5.5. NAE Hardware API 99
5.6. Prototype Overview 103
5.7. NAE Guest API 108
5.8. Network Control Domain API 110
5.9. NAE Hypervisor API 115
5.10. NAE Hardware API 117
6. Performance Evaluation 122
6.1. Overview 122
6.2. Test Setup 125
6.3. Test One: Gigabit Ethernet Performance 127
6.4. Test Two: Ten Gigabit Ethernet Performance 133
7. Contributions 144
8. Conclusion 149
8.1. Achievements 149
8.2. Future Work 153
9. Bibliography 155

Table of Figures

Figure 1 Overall System Architecture 12
Figure 2 Basic Virtualized Platform Architecture...18
Figure 3 Para-virtualized I/O model...23
Figure 4 PCI SR-IOV Network Card..30
Figure 5 Overview of NAE system APIs...66
Figure 6 Virtual Control Path memory space layout 70
Figure 7 Hardware-based Virtual Control Path 76
Figure 8 Concept of Circular Buffer 80
Figure 9 Arrangement of I/O Channels...84
Figure 10 Multi-channel I/O System Architecture...87
Figure 11 System Architecture including Virtual Network Switch 89
Figure 12 Multi-channel I/O using SW-based and HW-based VDP 91
Figure 13 Overview of NAE VCC API 93
Figure 14 System Architecture Overview of the Implementation 104
Figure 15 Implemented NAE System Components 106
Figure 16 Virtual I/O Device Implementation..109
Figure 17 NAE System Components Function Hooks 111
Figure 18 Test Set-up 125
Figure 19 Throughput in Mbps 128
Figure 20 CPU Usage on TX..129
Figure 21 CPU Usage on RX..130
Figure 22 Latency on TX...131
Figure 23 Latency on RX...132
Figure 24 Throughput in Mbps 134
Figure 25 CPU Usage on TX..136
Figure 26 CPU Usage on RX..138
Figure 27 Latency on TX...141
Figure 28 Latency on RX...142

List of Tables

Table 1 Description of Virtual Control Path Fields 71
Table 2 Simple TX Data Descriptor 100
Table 3 Simple RX Data Descriptor 102

1. Introduction

1.1. Problem Statement

Over the last years virtualization has become commodity and as a result platform developers are trying to build systems that are better suited to running virtualized workloads. Major architectures like x86 and ARM have been enhanced to provide better virtualization support. In parallel to this, system design is moving towards more specialized (but modular) hardware components and one can find various hardware "accelerators" on most of today's current platforms. This can, for example, be in the form of specialized cores or graphics processing units (GPUs), but also can be found recent generations of intelligent I/O devices that might have their own processing engine and which, in some cases, have been enhanced for running in a virtualized environment.

For cloud computing infrastructure providers it is important to have a platform that is easily manageable and can be run at low cost and at large scale, so it is common to choose commodity hardware with little specialization. However, as competition in the market is fierce, performance will play a bigger role in the future and infrastructure providers need to take advantage of hardware acceleration in their virtualized infrastructure to differentiate their offerings. This is particularly significant as High-Performance Computing (HPC) applications are moving into the cloud as well (Kimball 2012).

One of the most important areas in virtualized infrastructures supporting cloud computing environments is the underlying networking and a critical part of this is network I/O virtualization. Network I/O devices have evolved significantly to better support system virtualization. For example, there are fully self-virtualizing network controllers, some using a standard low-level hardware interface (for example PCI SR-IOV compliant devices[1]) and

[1] PCI Single-Root I/O Virtualization (SR-IOV) is an extension to the PCI standard and enables a PCI device to provide multiple "virtual" PCI devices to the host. We explain PCI and PCI SR-IOV in detail in later sections.

some using proprietary APIs, like (LeVasseur, et al. 2008). There are other technologies like HP's Virtual Connect running, for example, Broadcom or Emulex network chipsets and aiming at providing network resource partitioning support to virtualized platforms. And we are seeing programmable network adapters which are based on FPGAs[2] that can be used for more powerful network processing on endpoints, as for example proposed in (Wun and Crowley 2006).

With all these different technologies and little standardization between them in this space, infrastructure providers have difficulties deploying them in the automated fashion necessary when operating at large scale. Furthermore, it is not clear which technology is best suited for deployment given any specific workload. Also, there are no common management stacks for different hardware-based network accelerators.

In order to address these problems, we set out to analyse the network I/O path of virtualized systems. As part of this we identify problems and limitations of network I/O architectures one can find on current systems. In particular, we focus on the software/hardware interface involved in network I/O processing. Following such an evaluation we propose the development of a new network I/O architecture to better take advantage of today's hardware accelerators. We investigate a secure and efficient network I/O path design that satisfies the requirements of a cloud computing infrastructure provider: it needs to be manageable at a large scale and support an automated, dynamic deployment scenario in virtual machine environments.

Rather than take the traditional approach and expose a full hardware device interface directly to a guest operating system, our solution proposes the use of the advanced capabilities offered by modern I/O devices as so-called "network acceleration engines" (NAEs). Instead of "blindly" handing over a device to a virtual machine, the idea is to be able

---

[2] Field-programmable gate arrays (FPGAs) are highly customizable integrated circuits that can be programmed by a user after manufacturing. They can be shipped at significantly lower volume and at significantly lower cost than a dedicated ASIC and can be re-configured with any logical function a user can think of implementing with the logic blocks provided by the FPGA.

to be more selective about what hardware resources to make accessible and in what way to make them accessible to a virtual machine. In this architecture the NAE capabilities are controlled by the privileged virtualization layer and their hardware APIs are not directly exposed to applications running within virtual machines. This makes it easier to migrate applications across different hardware platforms and virtual machines do not need to be aware of the specifics of the hardware accelerators available on the platform. As part of this work we propose a "network acceleration engine API" that enables the control of a variety of network hardware accelerators and exposes their capability set to applications running inside virtual machines. This allows a common management software stack that is vendor-independent and covers a wide range of network I/O devices.

1.2. Solution Summary

Our architecture explores new ways of taking advantage of hardware acceleration for network I/O on virtualized systems. Traditional, early I/O virtualization architectures did not incorporate hardware acceleration at all. Instead, they virtualized in software by emulating I/O devices. Such an architecture, as we demonstrate later, introduces significant processing overhead into the virtualized platform. That performance impact is unacceptable for many of today's applications which would like to move into virtualization-based environments. Therefore better hardware integration is a must for the next generation of virtualized cloud computing infrastructures. Current approaches which try to integrate intelligent network I/O devices, like multi-queue NICs[3], PCI SR-IOV adapters or network FPGAs, expose vendor-specific hardware interfaces to the virtual machine which wants to take advantage of the device. In such a design the virtual machine is tied to a particular real device and needs to run a specific driver for that device. This so-called *direct device assignment* introduces

[3] A multi-queue NIC is a network card that can provide multiple, simultaneously running transmit and receive queues in hardware. Using those queues the network card can handle multiple packet streams simultaneously and the host processor can, for example, assign individual processor cores to each queue. Such a design enables highly parallel network packet stream processing.

serious challenges for infrastructure providers. Exposing vendor-specific hardware interfaces directly to virtual machines is undesirable in a large-scale environment for various reasons. The biggest issue in this context is certainly portability. Large-scale cloud computing infrastructures typically run across a huge set of heterogeneous platforms where the underlying hardware is not necessarily identical. Therefore, if virtual machines are migrated between platforms, they have to be able to run across a different set of hardware resources, and, in particular, network I/O devices. This is only possible, if the virtual machine is not tied to a particular, device-specific hardware interface.

In this context, it is the goal of this work to investigate the “ideal” I/O architecture for virtualized systems taking into account the limitations of previous approaches as mentioned above. We want to provide improved performance compared to purely software-based I/O approaches while offering more portability and flexibility than direct device assignment approaches. In our solution we suggest to not use a vendor-specific API to access virtualized hardware resources and instead we propose a new, secure interface for accessing and managing “network acceleration engines” (NAEs) which achieves high I/O performance for virtual machines. The NAE API has the following characteristics:

- It is vendor-independent and works across a wide set of network I/O devices
- It is modular, so that over time new features can be added and so that vendors can still implement their value-add feature allowing differentiation from other network I/O devices
- It defines reasonable generalized features, providing common data structures across different device types
- It covers typical networking capabilities like TCP/UDP processing offloading, hardware-accelerated VLAN tagging, rate control, multicast and broadcast handling, packet switching and packet rewriting

- It is architecture-independent, so it can, for example, be used on powerful x86-based servers or workstations, but also on ARM-based mobile platforms

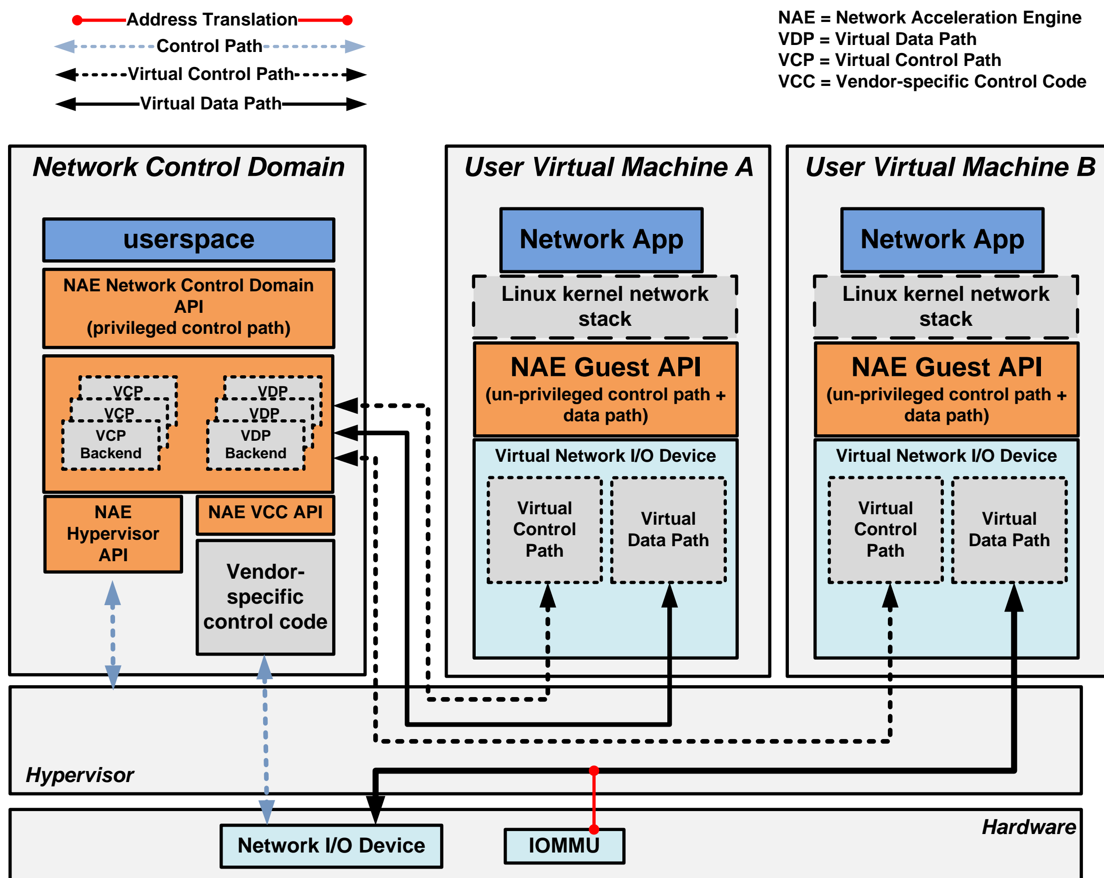


Figure 1 Overall System Architecture

Figure 1 shows the overall high-level design of the Network Acceleration Engine framework. On the virtualized system the NAE API is split into two main parts. One part sits in the privileged driver domain (in the figure called *Network Control Domain*), and one part sits in the unprivileged virtual machine (called *user virtual machine*) wanting to access the network I/O device. This “split” virtual I/O device architecture is very commonly used in today’s virtualized systems as it allows some form of control of the virtual I/O device from the privileged driver domain. Typically, the unprivileged user virtual machine issues I/O requests that are then checked by the privileged network control domain before being passed to hardware. There is also a more direct path between the user virtual machine and the hardware network I/O device, as shown for user virtual machine A in the figure above.

Despite being a direct path not involving the network control domain on data transfer, this virtual I/O path is still controlled by the network control domain. For example, the network control domain handles its set-up and ensures that its main properties cannot be changed by the unprivileged user virtual machine.

One of the key design ideas of the proposed virtualized system design is to clearly separate the data path and the control path for network I/O: the NAE API explicitly categorizes functions into control path functions and data path functions. We found this separation of information flow is a key to a clear and efficient design giving the following important advantages:

- It makes it easier to unify device access across different device types and define the minimal common data structures that are required to be implemented across different vendors.
- Control operations and data transfer operations have very different characteristics and service requirements and separating them out makes it easier to provide the best design and implementation for both of them.
- It allows having a software-based or hardware-based virtual data path (VDP) and potentially switching between those at runtime.

The proposed architecture allows the application running inside the virtual machine to directly access the (virtualized) data path of the network I/O device. Ideally, one always wants to provide a hardware-accelerated VDP to the user virtual machine. Such a configuration is shown for user virtual machine B in Figure 1 and for the remaining sections of this thesis we focus on how to achieve that. However, our design makes it also possible to run a software-based VDP emulated in the network control domain (as shown for user virtual machine A in Figure 1). This might be required if we run more user virtual machines than we can provide dedicated hardware-based VDPs for. In such a scenario, we can load-balance user virtual machines across a limited number of hardware-accelerated VDPs while

remaining user virtual machines can still use the network through a software-based VDP (which potentially runs at reduced performance). Our generalized, vendor-independent interface allows to efficiently use network resources in such a dynamic and flexible configuration. This is crucial for cloud computing infrastructures and one of the major differentiators of our solution.

Through the NAE API network I/O devices can safely be shared between multiple virtual machines as each virtual machine gets its own virtual network I/O device with its own virtual control path and its virtual data paths. The configuration of the virtualized data path is managed via the virtual control path from the user virtual machine. However, it is strictly controlled by the privileged part of the NAE API sitting in the network control domain. This ensures that virtual machines cannot control hardware resources they have not been granted access to. For each real device used as hardware-based network I/O accelerator, there exists vendor-specific code implementing access to the device's low-level control functions. Typically, vendor-specific code is only required for device hardware initialization and teardown. Furthermore, there might be device features that need additional, more complex logic that cannot be feasibly implemented on the device itself and therefore has to be implemented by vendor-specific control software. This code sits underneath the NAE Network Control Domain API and is invisible to higher-level code. In Figure 1 this code is pictured under the NAE vendor-specific control code (VCC) API.

1.3. Contributions

In this section we aim to give a brief summary of the major contributions of this thesis. We separate them into high-level achievements and more low-level technical contributions. The high-level statements explain how and why we think that our work influences and advances the current state-of-the-art of virtualization technologies used in data centres backing up large-scale cloud computing infrastructures. Those aspects also show how our work can be

used and what benefits it will give to cloud computing infrastructure providers. The more low-level technical contributions of our work consist of newly developed methodologies that are unique and differentiate our work from any prior art in this research area.

The high-level contributions:

- We analyse the network I/O path of a virtualized system and identify limitations of network I/O architectures found on current platforms. In particular, we analyse the hardware design of today's I/O devices and how their implementation fits the deployment on virtualized systems.
- We propose the development of a new secure and efficient I/O architecture that satisfies the requirements of virtualized cloud computing infrastructure: it needs to be manageable at a large scale and support an automated, dynamic deployment across a heterogeneous set of hardware platforms. This I/O architecture enables unprecedented VM mobility and portability while maintaining high-performance network I/O.
- We design a virtualized data path with a more generalized virtual I/O interface exposed to VMs. Such an I/O interface enables novel ways of load-balancing platform resources backing up the virtualized data path. That way we can switch between a hardware-based and software-based virtual I/O path, transparently to the VM, enabling significantly better resource management opportunities for cloud infrastructure providers.

The low-level, technical contributions:

- We develop the principle of separating information flow into "control" and "data" parts and use this principle consequently across the whole system architecture, spanning across all software-based and hardware-based components involved on the network I/O path. Many technical benefits result from this strict separation principle.

- We define a very simple but efficient virtual data path interface consisting of a small set of data structures and functions that can be feasibly implemented by hardware and by software. For this we investigated a minimal set of data structures and functions an efficient I/O path needs to provide in order to transfer data from virtual machines to the network, and vice versa. This furthermore has to advantage that resources backing up the virtual data path can be switched easily – between hardware-based and software-based resources, and also between different physical machines (simplifying virtual machine migration).
- We investigate the implementation of current network I/O devices and show their limitations when using them in a virtualized system where their resources are to be shared between multiple virtual machines concurrently. We prove that some devices lack sufficient isolation of hardware-based resources and therefore do not enable secure sharing of a single I/O device between multiple virtual machines.

2. Background

2.1. Virtualization and I/O Virtualization

System virtualization has been a hot topic in both academic and industrial research over the last years. System virtualization allows running multiple, potentially different operating systems on a single platform. System virtualization creates one or more so-called *virtual machines* (VMs). Virtual machines look like individual, complete real platforms to a user. Virtual machines consist of virtual resources. For example, they have virtual CPUs, virtual memory, virtual network cards, virtual storage devices, virtual GPUs, and so on. Virtual resources can be backed up by real hardware or they can be completely "virtual" which means that they are software-based, emulated devices. They can also be created as a mixture of the two. For example, a platform can have just a single CPU, but it can be virtualized into two virtual CPUs which share the real CPU. Those virtual CPUs are then multiplexed by software onto the real CPU.

Within each virtual machine the user can run their own operating system with its own set of applications. Virtual machines are isolated from each other in that applications running in one virtual machine cannot normally access resources from other virtual machines. Isolation properties of virtual machine solutions vary significantly and some solutions provide stronger isolation than others. Virtualization solutions introduce a new component into the system: the so-called *hypervisor* or *virtual machine monitor* (we use these terms interchangeable for the rest of the thesis). The hypervisor runs at a higher privilege on the platform than any operating system or application. Therefore the hypervisor is the most important component on the platform controlling virtual machine resources and real hardware backing up those resources. It has full control over any resource and it can apply

access restrictions on resources that shall be made accessible to virtual machines. Figure 2 pictures a basic virtualized platform and its main components.

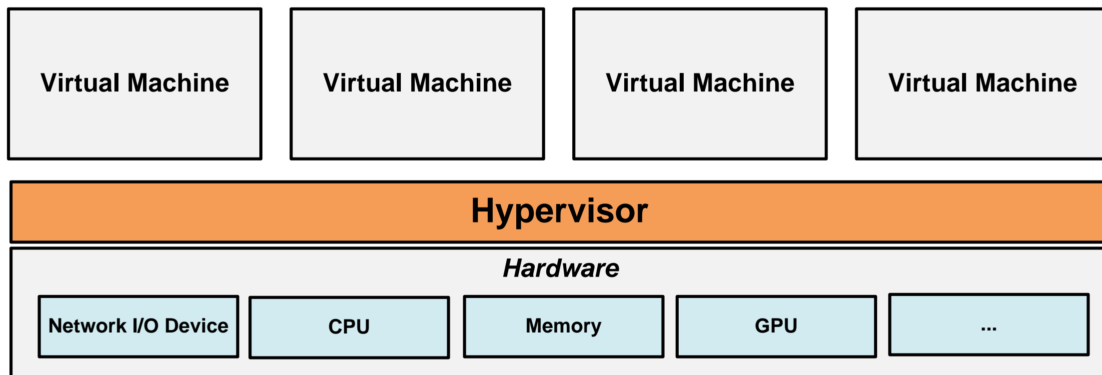


Figure 2 Basic Virtualized Platform Architecture

Virtualization introduces significant overhead into a system (for example, to run multiple operating systems concurrently and to multiplex shared hardware resources between two or more virtual machines) and in order to address this limitation platform developers started to introduce hardware-based assists on the platform to enhance performance. In this context, CPU and memory virtualization assists came onto the market first. While CPU and memory virtualization have been researched extensively right from the beginning and performance improvements have been made through early hardware-based assists, I/O path virtualization and I/O device virtualization has only recently caught the interest of the industry (Chen and Bozman 2009) (Rubens 2010) and the research community. Various performance evaluations and real-world use cases, as described in later sections, show that the I/O path has developed to be the bottleneck of a highly utilized virtualized platform.

I/O virtualization takes care of getting data into and out of virtual machines. This can, for example, be storage I/O, network I/O, graphics I/O, mouse and keyboard I/O, and so on. Different I/O devices often use different ways to virtualize I/O. This is mainly because those devices have different characteristics and requirements, and the I/O virtualization mechanism needs to be well suited to the actual device and the service it provides to the virtual machine. The work presented in this thesis focuses on optimizing network I/O

virtualization. Cloud infrastructures are highly networked environments and typically most services require network connections to other virtual machines hosted in the same cloud infrastructure or even on the Internet. The software architecture of today's highly distributed cloud services often even requires networking between internal service components. For example, the front-end web server might access its underlying database through a network connection. In summary, networking between virtual machines represents one of the most critical and fundamental building blocks of cloud infrastructures. It needs to be secure and efficient. This means that performance should ideally be as close to performance of a native, non-virtualized system as possible. Otherwise applications might not function anymore when running across a virtualized infrastructure, for example, because they have been designed to only run over low-latency network connections. The other reason for an efficient solution is that quite often service providers are bound to run their infrastructure conforming to certain service level agreements (SLAs) meaning a certain performance level needs to be achieved constantly. Security is important, because cloud infrastructures are by design multi-tenant networks and it must be ensured that network traffic between different, potentially competing, parties is sufficiently isolated. Furthermore, the design of a network I/O virtualization solution needs to take into account that virtual machines are highly mobile and dynamic components. For example, their (virtualized) resources can be adjusted on-the-fly and their location within the infrastructure can change frequently. As a more concrete example, a virtual machine might, at run-time, request additional virtual network cards attached to different (virtual) networks. This means its network identity changes whilst the virtual machine is online. On the other hand a virtual machine can migrate to a different physical location meaning its own network identity remains the same, but it might have to re-discover peers around it. The virtual machine's network stack and the underlying virtual I/O architecture need to handle these types of events.

I/O virtualization approaches can be divided into software-based I/O virtualization and hardware-based I/O virtualization. In software-based I/O virtualization, a software component handles passing data into and out of the virtual machine. That software component usually runs in the so-called *driver domain* or *host operating system* (we use those terms interchangeable from now on during this thesis). The driver domain is a privileged virtual machine running on top of the hypervisor and has some form of privileged access to system hardware or other attached devices in order to initialize and configure them. Every virtualized platform has a driver domain in some form, even if it might be called differently for various virtualization technologies. We use the term driver domain in this thesis for all of these privileged virtual machines owning real hardware.

2.2. Network I/O Virtualization

2.2.1. Software-based, Emulated I/O Devices

In some virtualization solutions a software component in the driver domain emulates a real I/O device. This technique of I/O virtualization, so-called *device emulation* and best described in (Sugerman, Venkitachalam and Lim 2001), has been the first and initially most popular solution to provide virtual I/O capabilities to virtual machines. It is also used when sharing a single real device between multiple virtual machines – in that case there typically needs to be an additional software-based multiplexing component that distributes I/O between the real device and its attached virtual devices[4]. Device emulation is used in most of today's virtualization solutions. For example, it is used in Oracle's VM VirtualBox (Oracle n.d.) and all VMware products (VMware Inc. n.d.). Linux's kernel-based system virtualization

[4] For network I/O this software-based multiplexing component is usually a so-called *virtual switch* or *virtual bridge*. It has multiple virtual ports to which the real device and all virtual devices are connected. Virtual switch concepts are described in more detail in (VMware Inc. 2007), (Tseng, et al. 2011) and (Pfaff, et al. 2009). A virtual switch can provide a variety of packet processing services and many advanced virtual switch implementations are available on the market today. Virtualization solution vendors can significantly differentiate their product with smart and efficient packet processing functions.

(KVM) hypervisor (Kivity, et al. 2007) and the QEMU open-source machine emulator and virtualizer (Bellard 2005) also implement a device emulation layer. In this context, the QEMU machine emulator can be enhanced with kernel-based virtualization acceleration through KVM. In that case KVM takes care of virtualizing CPU, memory and interrupts of real devices while QEMU takes care of emulating virtual I/O devices. Here the standard configuration can emulate a well-known Intel e1000 network interface card (NIC) which is then attached to a virtual machine. The virtual machine then loads the standard Intel e1000 device driver as if it was running on a real e1000 NIC. The virtual machine does not know that it runs on an emulated device rather than real hardware meaning it communicates with the emulated device as it would communicate with hardware. This, and also the fact that everything is processed in software on the main CPU, makes emulation a very expensive and inefficient I/O virtualization technology. On the other hand, this approach allows using existing drivers which have been developed for the real hardware-based device in the virtual machine. Mainstream operating systems, like Windows and Linux, already include a large set of device drivers by default and when emulating a real device, as described above, those public device drivers can immediately be used without any modification. Therefore, this approach allows to easily migrate legacy operating systems into virtual machine instantiations. Most virtualization solutions emulate well-known and well-distributed network devices. That means many existing operating systems by default provide drivers for those. Apart from this, architecturally, emulated devices enable virtual machine migration: they do not tie a virtual machine to a particular real device, but instead they just tie them to a software model of a device which can be moved relatively easily to any other platform. The main component that has to be migrated in this approach is the state of the emulated network card plus open network connections. The state of the network card is available completely in software and therefore reasonably easy to capture and re-instantiate on the target platform. Open network connections are maintained by the operating system and will be migrated in the

same way that other operating system state is migrated. This type of purely software-based virtual machine migration is implemented in most virtualization solutions, like for example (Inc., VMware vMotion n.d.), and has already been studied extensively with regard to performance and cost (Voorsluys, et al. 2009) (Nelson, Lim and Hutchins 2005), portability (AMD 2007) and service availability (Travostino, et al. 2006) (Bradford, et al. 2007). Next to KVM/QEMU, the most well-known virtualization solution using software-based emulation is VMware with their VMware Workstation (Inc., VMware Workstation 2012) and Server (Inc., VMware vSphere 2012) products. VMware has been the pioneer of bringing software-based system virtualization to a large consumer and business market. However, various studies, like (Mei, et al. 2010) and (Nakajima and Stekloff 2006), show that device emulation is not suitable for applications requiring high-performance I/O and strong isolation between virtual resources. Emulation puts a high CPU load on the driver domain as all I/O requests need to be processed in software. Running the emulated device also always requires a context switch to the driver domain when I/O requests need to be processed. Frequent context switches like this are expensive. There are further issues with current device emulation as noted in (Nakajima and Stekloff 2006). For example, multiple concurrently running emulated devices are not sufficiently isolated when running as user processes in the same driver domain. Furthermore, it is difficult to account I/O resource utilization of shared real devices to individual emulated devices. Due to all these issues VMware and most other virtualization solution vendors have all enhanced their products with hardware-based virtualization assists (VMware Inc. 2009) and with a recent software-based approach called *para-virtualization*.

2.2.2. Para-virtualized I/O Devices

Para-virtualization tackles the issue of low I/O performance that emulated devices struggle with. In an architecture based on so-called para-virtualized I/O, the virtual machine runs a para-virtualized device driver (also called the *frontend* interface) which is not under the illusion of communicating with real hardware, but instead it is designed to know about and

understand the underlying virtualization layer on which the virtual machine runs. By knowing about it, the device driver can use more efficient mechanisms to communicate with the virtualized device. Para-virtualization has been introduced initially for the Xen hypervisor (Barham, et al. 2003) that originally just provided para-virtualized device drivers for block I/O and network I/O, but then over time added para-virtualization support for other system components and I/O devices. On top of that, other virtualization layers introduced para-virtualization in various aspects of the system, for example (Magenheimer, et al. 2008), (Dowty and Sugerman 2009) and (Youseff, et al. 2006). A para-virtualized device driver running in a virtual machine communicates with a backend interface sitting in the driver domain. In order to ensure good I/O performance, the para-virtualized device driver and the backend interface use advanced memory sharing and event notification mechanism provided by the virtualization layer. These efficient inter-domain communication mechanisms are hypervisor-specific and define, to a large extent, the performance of para-virtualized I/O.

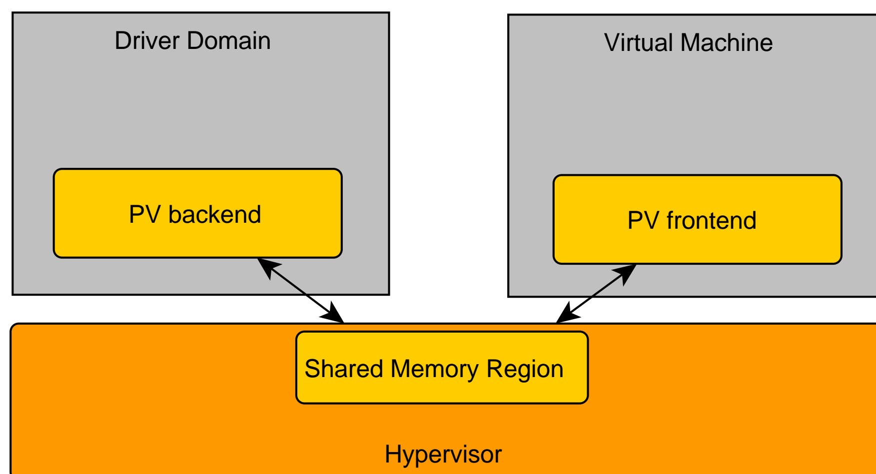


Figure 3 Para-virtualized I/O model

The use of para-virtualized I/O devices can significantly improve performance for virtual machines; however, the deployment of para-virtualization comes at a significant cost. The biggest drawback of para-virtualization is that the virtual machine operating system has to

be modified explicitly to run on a particular hypervisor. Main operating system data structures need to be made aware of the para-virtualized infrastructure. Furthermore, para-virtualized drivers for Xen do not work on Linux KVM or any VMware products, and vice versa. Every hypervisor needs its own set of para-virtualized drivers. This can potentially be a large number of drivers, because a full-blown operating system needs drivers for network I/O, storage I/O, consoles, graphics I/O and so on. Those para-virtualized drivers need to be developed for each individual operating system that is supposed to run inside virtual machines. Overall, porting effort for operating systems and device drivers to work in a virtualized environment based on para-virtualization is significant. Therefore, many of today's modern hypervisors do not require the whole operating system to be para-virtualized any more. Instead, it is possible to run a vanilla operating system kernel inside a virtual machine (details on this approach will be explained in the following section). On top of this it is possible to run *optional* para-virtualized device drivers to enable improved I/O performance for certain devices. This type of hybrid virtualization seems to have found a lot of support in the industry as it provides a reasonable compromise for real-world deployments.

In (Russell, virtio: towards a de-facto standard for virtual I/O devices 2008) Rusty Russell from IBM introduces the so-called *virtio* I/O virtualization framework. virtio was originally developed in 2008 for Linux KVM and lguest, an open-source Linux virtualization solution (ozlabs.org n.d.). virtio implements a standardized virtual I/O bus between a para-virtualized device driver (the frontend) running inside the virtual machine and the backend running inside the driver domain. The virtio architecture follows a layered approach implementing a transport protocol[5] and a set of I/O device drivers which can be loaded on the virtual I/O bus on top of the chosen transport protocol. This includes a network driver (virtio-net), a block device driver (virtio-blk) and a console device driver (virtio-console). The idea of the

[5] At the moment virtio only supports a single transport protocol on the virtual I/O bus, which is PCI, but a memory-mapped I/O transport for non-PCI architectures like ARM is being developed as well.

standardized virtual I/O bus is that anyone can reasonably quickly develop a device driver virtualizing an I/O device and give a virtual machine access to it. The backend can use a real device to process requests (for example, a network card) or it can pass requests to a software-based emulation layer that virtualizes an I/O device (for example, QEMU implements various virtio backend interfaces). That way, various other virtio device drivers have been proposed and partly implemented. For example, a SCSI driver (virtio-scsi), a serial console driver (virtio-serial) and many others. Internally, virtio implements transferring of data buffers between the frontend and the backend by using ring buffers residing in memory regions that are shared between the virtual machine and the driver domain. These are the so-called *virtqueues* and their implementation is hypervisor-specific. Frontend and backend can both place data buffers into the virtqueues and signal to the other end that new data is ready to be processed. Event signaling like this is transport-specific. As an example, the backend can signal the arrival of new data to the frontend by injecting an interrupt into the virtual machine while the frontend indicates new data to the backend by writing a particular register of the associated virtual I/O device. A virtio specification has been drafted and gives more details on its operation and implementation in (Russell, Virtio PCI Card Specification v0.9.4 DRAFT 2012).

There have been various performance studies indicating that virtio I/O devices achieve better I//O performance than software-based emulated I/O devices. We will do our own evaluation as part of this thesis in later sections. As virtio is one of the most popular standards in I/O virtualization, it is worth comparing a prototype implementation of our I/O architecture against it.

virtio is a great approach that signals a move away from doing I/O virtualization by emulating real hardware devices in software which is seriously inefficient. As virtualization becomes more and more commodity, it makes sense to develop I/O bus infrastructures that are designed purely for the usage on virtualized platforms. One limitation of virtio is that it

cannot integrate with hardware-based I/O acceleration, if there are real devices on the platform that can offer such features. This is a restriction we want to overcome when designing our proposed NAE architecture.

Para-virtualization can significantly improve the I/O path between a virtual device running in the virtual machine and its, typically software-based, backend component running in the driver domain. As briefly explained before, this can be done through improved memory sharing and event notification mechanism between the driver domain and the virtual machine. However, it does not cover the full I/O data path. The full I/O data path further includes a connection between the backend component in the driver domain and the real device that is involved in I/O processing. This bit of the I/O path is not at all covered by para-virtualization approaches and therefore still presents a challenge. I/O still needs to be multiplexed in software, so that a single real device can potentially serve multiple backend components. A backend component of a para-virtualized device driver can be significantly more efficient than an emulated device component. However, running a software-based backend component still introduces non-negligible processing overhead in the driver domain. Therefore, overall, para-virtualized I/O cannot reach the performance numbers of direct I/O techniques. We present related performance studies in the following sections.

2.2.3. Hardware-assisted I/O Virtualization

System virtualization introduces a significant processing overhead slowing down applications running inside virtual machines. But still, promises of reduced costs and increased flexibility for computer users made system virtualization become a mainstream technology. With these prospects, vendors have started adding support for virtualization to various parts of the platform and to I/O devices. Mainstream hardware-based virtualization started with CPU extensions as introduced for x86, described in (AMD Virtualization n.d.) and (Intel Virtualization Technology (Intel VT) n.d.), and ARM, described in (Virtualization Extensions n.d.), which makes it significantly easier to run a hypervisor underneath an *unmodified* guest

operating system. It makes it possible to more easily “trap” into the hypervisor when the virtual machine makes an attempt to access privileged resources. A hypervisor trap is a hardware-based mechanism to intercept privileged instructions executed whilst running in unprivileged mode as used by virtual machines. In this case, when a virtual machine attempts to execute such a “trapped” instruction, the hardware automatically routes execution to pre-defined trap exception handlers controlled by the hypervisor. Within the exception handler the hypervisor can then handle the trapped instruction, for example, it can run code emulating the instruction. When it has finished processing the trapped instruction, control is given back to the virtual machine. For both ARM and x86, hardware-assisted CPU virtualization works by introducing an additional, privileged CPU mode. That mode is used to run the hypervisor at a higher privilege level while operating system and applications run at their usual privilege levels, for example, system mode and user mode. Using this approach operating systems and applications do not need to be modified to run above a hypervisor. In fact, they are unaware that they are running on a hypervisor-based platform. This type of virtualization solution is also called *full-virtualization*.

Platform vendors furthermore introduced support for virtualization in their platform I/O memory management units (MMUs), like for example (Abramson, et al. 2006), (Advanced Micro Devices, Inc. 2009) and (Goodacre 2010). This simplifies allowing safe and controlled direct I/O access from virtual machines to hardware resources on the I/O bus. The I/O MMU takes care of address translation when a virtual machine accesses an I/O device directly. This approach allows running an unmodified operating system in a virtual machine which has direct access to I/O devices. The I/O MMU furthermore enforces access control, so that virtual machines can only access I/O device regions that they are allowed to access, and I/O devices can only access system memory regions they have been granted access to. When a virtual machine has direct access to the full real I/O device interface, we call this I/O

virtualization mechanism *direct device assignment* or *direct I/O* or also *pass-through I/O*. In the following parts of this thesis, we use these terms interchangeably.

With I/O MMU virtualization in place it is reasonably straightforward to directly assign devices to a virtual machine and allow a direct I/O path between the virtual machine and the device. However, with traditional I/O devices this means a single device can only be used by a single virtual machine. If a network interface card has multiple physical ports, then these could be assigned to different virtual machines. However, that is still not enough as we typically run many more virtual machines on one physical machine than we have network interface card ports. So-called *self-virtualizing* I/O devices have been developed to overcome this issue. These provide virtualization assists implemented in silicon and can provide *multiple virtual device interfaces* to the platform. Various forms of self-virtualizing devices exist. Some just provide some form of partitioning of hardware resources that can be individually assigned to virtual machines, like (Chinni and Hiremane, Virtual Machine Device Queues 2007), which can multiplex network packets into separate hardware-based packet queues which can then be directly handed to a virtual machine. The advantage of this, compared to software-based approaches, is that there is no intermediate data copy to the driver domain. Others provide a more PCI-like, but still proprietary, virtual device interface to the virtual machine, like for example (LeVasseur, et al. 2008). With more and more vendors showing interest in developing virtualization-aware devices, a unified standardization effort started and was reasonably quickly supported by all major industry players. It is explained in the following section.

2.2.4. Introducing PCI Single-Root I/O Virtualization

A big move forward in standardizing I/O virtualization came with a series of specifications by the PCI Special Interest Group (SIG). PCI stands for Peripheral Component Interconnect and is probably the most popular I/O bus used by x86-based platforms. Devices on the PCI bus are directly addressable by the processor. The first PCI specification (now called

Conventional PCI) and the second-generation PCI-X standard have later been replaced by the PCI 3.0 specification introducing the PCIe (also called *PCI Express*) standard. PCI is used as a motherboard-level interconnect. Many onboard or plug-in I/O devices in servers, laptops and workstations use PCI. PCI devices consist of one of more device *functions* with each having their own so-called *configuration space*. Each function can be identified by a 8-bit PCI bus ID, a 5-bit device ID and a 3-bit function ID. PCI devices are mapped into I/O port and memory-mapped address spaces dynamically when the platform boots. System software (BIOS, hypervisor or operating system) programs the device with address mappings by writing into specific registers of their device configuration space. This mechanism enables a reasonably dynamic system initialization where devices on the PCI bus can be mapped into system memory depending on overall resource utilization.

As PCI is such an important, widespread interconnect, technology advances in the PCI I/O bus specification have a significant impact on industry trends and future directions. When the PCI SIG introduced a new standard in the space of I/O virtualization, technology in this area could advance significantly as most, if not all, hardware and software vendors started building support around the same specification.

The PCI I/O Virtualization (IOV) specifications consist of a new PCI Address Translation Service (ATS), PCI Single-Root IOV (SR-IOV) and Multi-Root IOV (MR-IOV). Most significantly ATS and SR-IOV, as described in (PCI-SIG n.d.), define a standardized way of providing multiple PCI virtual functions (VFs) on top of a PCI physical function (PF). A virtual function is typically a lightweight version of a physical function, but it has its own PCI configuration space in hardware and its own PCI device functions and hardware resources which are independent of the physical function and isolated from other virtual functions on the same hardware. The PCI SR-IOV specification covers how virtual functions are discovered on the PCI bus, so that they seamlessly integrate into existing PCI bus hierarchies. PCI SR-IOV requires support from the hypervisor and the driver domain operating system in order to

initialize the virtual functions. Most major network interface card vendors developed and sell PCI SR-IOV capable devices for both 1 Gigabit Ethernet and 10 Gigabit Ethernet. For example, Intel's 82576 and 82599 Ethernet controllers, Solarflare's Solarstorm device, Mellanox's ConnectX-2 controller and QLogic's 3200 Series were the first devices available on the market. All mainstream hypervisors and operating systems also support PCI SR-IOV by now. Performance evaluations, for example (Worley 2010), (Liu, Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support 2010) and (Dong, Yang, et al. 2009), show that virtual I/O using PCI SR-IOV devices can potentially reach performance numbers close to those achieved by direct hardware access on a non-virtualized system. We describe some further evaluations in the following sections elaborating why direct I/O access using PCI SR-IOV from virtual machines is "only" close to native I/O performance measured on non-virtualized systems, and not the same.

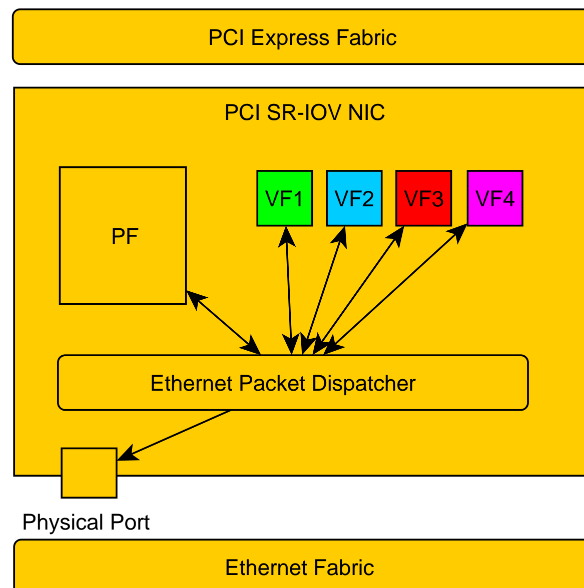


Figure 4 PCI SR-IOV Network Card

PCI IOV is a big step towards standardizing hardware-based I/O virtualization. Obviously PCI IOV only covers virtualization of PCI-based I/O devices and does not deal with virtualization of other I/O bus infrastructures. Furthermore, PCI IOV does not cover the integration of the self-virtualizing device into the bigger system, for example, it does not specify the

hardware/software interface sufficiently to allow hypervisor-independent and OS-independent standards. It does not specify how virtual device interfaces should be integrated into system software, where they are configured and how they are assigned to virtual machines. PCI IOV is typically used in pass-through I/O mode where a particular PCI function (physical or virtual) is dedicated completely to a virtual machine. The virtual machine has full control over that hardware-based PCI resource. As explained previously, in such a pass-through I/O configuration, a device-specific interface is exposed directly to the virtual machine which introduces hardware-dependencies for applications running inside the virtual machine and makes virtual machine migration incredibly difficult. Even though PCI is a common, vendor-independent interface, its abstraction level is too low to make it practical for a generic hardware/software interface used in virtual machine environments. The PCI function does not directly interface with the operating system. Instead, each device vendor needs to supply a software-based, device-specific driver to control and operate the PCI function. These device drivers are typically complex and provide a large part of the actual functionality of the device. In today's platform architecture, the device driver represents the vendor-specific, device-specific hardware/software interface. In closed, commercial operating systems this interface is proprietary and even undocumented for the public. The driver then implements a vendor-independent, device-independent software interface to the operating system, so that applications running in the operating system can take advantage of the device without knowing details of the hardware implementation and its interface to the driver. Here it is clear that PCI itself does not truly provide a vendor-independent and device-independent interface that can be used to expose I/O devices to virtual machines in a generic way. As a result, any solution that virtualizes purely at the PCI level, like PCI IOV, cannot be easily used on its own as an I/O virtualization approach. Therefore PCI IOV in its current form is not practical for dynamic, large-scale environments.

2.3. Study of Previous Work in Directly Relevant Areas

The research presented in this thesis is trying to address issues in various areas of network I/O virtualization. At first we want to improve network I/O performance for virtual machines. Then, we want to better integrate powerful network hardware into the virtualized platform, so that it is easier to take advantage of their sophisticated capabilities. Finally, we want to do all this while maintaining mobility and dynamic management properties of traditional virtual machine environments. Previous work exists in all these areas and we study the most relevant below. We analyse existing approaches and outline their individual benefits and shortcomings. This review will lead to a definition of requirements and potential mechanism for designing our new Network Acceleration Engine framework.

2.3.1. Improving Network I/O Performance

Various mechanisms for providing high-performance I/O to virtual machines have been subject to academic research initiatives, but also various industry solutions. Research around software-based I/O virtualization mainly focuses on providing faster data transfer to and from virtual machines, for example by reducing the number of data copy operations. In (Zhang, et al. 2010) the field of I/O virtualization and existing approaches for optimizing I/O performance are described and analyzed. The authors recognize the limitations of current hardware-based I/O virtualization mechanisms which directly expose the hardware interface to the virtual machine. Direct access interfaces like this make virtual machine migration very difficult. The paper also evaluates common performance optimizations for software-based I/O approaches and proposes additional enhancements on the virtualized I/O path. The main source of overhead they identify is the memory copy operation when moving data from the driver domain to virtual machines and vice versa. It is also considered that hardware needs to be integrated more efficiently to achieve better I/O performance, and in this context a new virtual device driver interface is proposed which reduces I/O access instructions issued

from the virtual machine and avoids unnecessary trapping into the hypervisor code and context switches due to device interrupts. The study addresses individual critical performance problems that are specific to a particular virtualization layer. For our goal of enabling high-performance I/O in a very dynamic and mobile environment, it lacks the context of a broader solution that looks at the challenges of presenting a virtualized hardware interface to the virtual machine in order to enable virtual machine migration and automated configuration of network capabilities as required in dynamic, large-scale cloud computing infrastructures. Individual techniques specific to a particular virtualization layer as presented in this paper are not sufficient to fully support the requirements of infrastructures we target with our work.

In (Huang and Baldine 2012) the authors look at 10 Gigabit network devices implementing PCI SR-IOV support as well as other network processing offloading functions, like TCP Segmentation Offload (TSO) or Large Receive Offload (LRO). Offload capabilities like this are used to move parts of the operating system's network stack processing to hardware. The study compares KVM with direct device assignment of a PCI SR-IOV device to container-based virtualization using a software-based bridged network connection and KVM using a software-based bridged network. They furthermore compare all approaches to a native, non-virtualized Linux environment. Such a comparison analyzes the I/O path while taking into account different types of virtualization. The authors show that offloading capabilities like TSO and LRO can significantly improve performance by reducing load on the main CPU. During bandwidth tests they discover that direct assignment of a PCI SR-IOV device in KVM does not perform as well as a non-virtualized Linux environment with the same device. The performance drop results from KVM's approach to interrupt virtualization causing a significant number of context switches to the hypervisor on the I/O data path in order to handle interrupts from the real device.

Other work, most recently (Abel, et al. 2012), recognizes this problem as well. In the work described in their paper the authors design and implement an I/O architecture where interrupt delivery is implemented in a way that the hypervisor is circumvented, if the target virtual machine for the interrupt is also the one currently running on a particular CPU. Their so-called ELI architecture achieves I/O performance superior to previous pass-through I/O technologies. These evaluations underline the importance of a design supporting virtualization in all subsystems involved in I/O access. The authors argue for architectural support from the underlying platform. They outline that hardware-based interrupt virtualization support is not yet fully optimized on x86-based platforms, even though their system has virtualization support in CPU, MMU and I/O MMU. Currently, it seems as if interrupt virtualization is the biggest problem area for I/O virtualization which has only very recently (this year, 2012) caught the eyes of researchers and developers. In this context ELI shows some very promising results and a first step in the right direction. The design of the Network Acceleration Engine architecture should build on ideas like ELI and look deeper into how interrupts can be handled in a more efficient way.

In (Santos, Janakiraman, et al. 2007) the authors show the performance drop of Xen's para-virtualized network device drivers compared to natively running, non-virtualized device drivers running on Linux where CPU saturation prevents Xen-based systems achieving line rate on a 10 Gigabit link. Their work proposes several optimizations to improve network I/O performance for Xen, mainly focusing on the receive data path as that is where a more significant bottleneck is identified. Again, the main source of overhead they identify is the copy of the packet buffer from Xen's driver domain to the destination virtual machine and this is what their optimizations target. The first optimization moves the copy operation from the driver domain into the virtual machine which improves CPU cache performance and eliminates the driver domain as the bottleneck when serving multiple virtual machines. It also improves accountability for resource management. Furthermore in (Santos,

Janakiraman, et al. 2007) the authors describe the enhancement of the receive I/O path with support for multi-queue network interface cards. With these devices, packets are multiplexed in hardware and placed into individual, hardware-based queues which are assigned to virtual machines. That way the NIC places packet buffers directly into virtual machine memory which avoids the expensive memory copy in the driver domain. The third optimization they propose is a memory mapping caching mechanism allowing the recycling of memory buffers in the virtual machine without having to constantly map and unmap memory buffers for packet reception. On top of these enhancements on the receive I/O path, the authors describe further modifications to both backend and frontend network drivers within the driver domain and virtual machine. Their approach significantly reduces CPU overhead and parts of their work have been proposed to be implemented in the Xen hypervisor under the codename "Netchannel2". Netchannel2 is very specific to how Xen handles I/O transfers between virtual machines and their approach cannot easily be transferred to other hypervisors. Multi-queue NICs are well suited to improving network performance in virtualized systems and the authors incorporate them in Netchannel2 without exposing vendor-specific details to the virtual machine which seems a very suitable approach potentially supporting future work on enabling virtual machine migration on a multi-queue device. However, their work is only applicable to Intel-based multi-queue NICs that are VMDq-capable (Chinni and Hiremane, Virtual Machine Device Queues 2007). Ideally, we would want to present a more generic framework which allows the inclusion of all sorts of I/O devices which are capable of exposing individual packet queues to virtual machines directly.

The same authors present similar work on improving Xen network I/O virtualization performance in (Santos, Turner, et al. 2008). Here they analyze the different network I/O virtualization approaches Xen provides and underline the benefit of keeping a driver domain on the data path rather than allowing direct I/O access from a user virtual machine to the

real hardware. The paper analyzes the performance of the direct I/O access mechanism in Xen which introduces a 31% overhead compared to non-virtualized Linux due to changes in the para-virtualized Linux DMA interface. The authors then look into Xen's para-virtualized device driver interface and here network I/O performance significantly drops due to data copy and packet processing overhead in the driver domain. Another significant source of overhead is Xen's inter-domain memory sharing mechanism which requires expensive hypervisor intervention on the data path. The authors propose similar optimizations as already explained in (Santos, Janakiraman, et al. 2007), but on top of this, they underline the importance of decoupling driver domains that are responsible for I/O processing from other driver domains on the same platform. As the driver domain which takes care of network packet processing significantly adds to the performance overhead of the virtualized I/O path, it needs to be a stripped-down OS with a kernel optimized for network packet processing rather than a general purpose OS kernel. This includes optimizing interrupt processing for network interface cards as well as packet filtering functions carried out in the driver domain. These findings underline how important it is to implement the right system design to realize optimal I/O processing. Various components involved in the I/O data path need to be optimized for I/O processing. Other research, for example (Fraser, et al. 2004), (Le, et al. 2009), (Levasseur, et al. 2004), (Anderson, Moffie and Dalton 2007) and (Swift, Bershad and Levy 2003), states the importance of isolating device drivers into their own virtual machines running purpose-built minimal OS kernels. The authors continue their work on Xen network I/O virtualization in (Ram, et al. 2009) where they propose several further enhancements to increase network throughput of virtual machines to full 10 Gb/s line rates. Once again, in this paper, the authors focus on implementation optimizations specific to Xen. In particular, they enhance the para-virtualized driver interfaces and these improvements cannot be easily applied to other hypervisor solutions.

Various research efforts recognized the poor performance of software-based device virtualization and the security and reliability issues of approaches directly assigning I/O devices to virtual machines. In (Xia, Lange and Dinda 2008) the authors propose an intermediate solution which allows the virtual machine to mostly interact with the real device directly while preventing it from programming the device illegally. The authors claim that their solution provides performance close to directly assigned I/O devices. The “Virtual Passthrough I/O” (VPIO) technology described has been developed with the requirements of running an unmodified guest operating system and allowing secure and efficient I/O access from a virtual machine. The authors aim to support I/O devices which do not have any sort of virtualization support in hardware. VPIO works by creating a software model of the real device which is maintained in the hypervisor. This model is driven by guest/device interactions (here in particular I/O port read/write operations) which trap into the hypervisor where a so-called “device model manager” (DMM) runs which verifies I/O requests from the virtual machine and modifies the related software device model accordingly. The DMM decides whether a virtual machine is allowed to carry out a certain I/O operation on a real device and it also context-switches a real device between multiple, concurrently active software models. That way this approach provides safe, concurrent access to I/O devices for virtual machines. Every virtual device assigned to a virtual machine has its own software device model. The DMM furthermore controls DMA operations by intercepting the initialization and setup process of DMA transfers. That way it can verify and potentially block DMA requests if the associated DMA addresses are not accessible by or permitted for the virtual machine. In this context the DMM takes care of address translation as the virtual machine uses virtual memory addresses while the real device operates on physical memory addresses. In this work the authors use AMD’s CPU virtualization extensions to intercept I/O port reads and writes from the virtual machine. These so-called *VM Exits* are expensive operations which limit the performance and scalability of this

approach. Therefore the authors try to limit I/O port interceptions as much as possible. For example, some I/O port reads do not need to be directly intercepted and instead the DMM can read from real device registers to recognize changes that need to be applied to the software model. In this paper the authors move a step in the right direction by asking vendors to supply a simple, inexpensive software device model for their device. This would greatly simplify the virtualization of the I/O path. It would help to operate a variety of devices through a software-based API which does not need to know exact details of device internals. Ideally, software models of different devices should have a comment set of interfaces, so that all devices can be controlled through a single, vendor-independent API. If every software model needs to be controlled individually, then the approach the authors propose here does not necessarily help large-scale infrastructure providers who manage a large, heterogeneous set of I/O devices. In this context, the research proposed in this thesis goes exactly this step further. By trying to support legacy devices which do not have any form of virtualization support, there are significant technical challenges that VPIO is faced with and does not seem to successfully overcome them. Firstly, because the real device cannot maintain state for each virtual device, the DMM needs to maintain this state in software and every time a context switch occurs, the real device needs to be programmed with the new state information. Typically this involves changing device register values, but potentially more expensive operations like resetting the device are required. The authors show in (Xia, Lange and Dinda 2008) that context-switching a device can be very costly. A further limitation of supporting legacy devices is specific to networking operation: when the real device receives a packet from the network, the virtual machine that is currently running and whose software device model is loaded might not be the valid receiver of the packet. However, interrupts and DMA writes from the real device would go to the currently running virtual machine. As the real device does not have a notion of multiple receivers, the only way to deal with this problem is to always route the device interrupt to the DMM which then

selects the right receiver virtual machine for a particular packet and then initiates a context switch. But still the challenge here is that the receiver can only be identified after a DMA write into host memory has taken place (in order to read the destination MAC address). Therefore, that packet might have to be copied over to the correct receiver virtual machine, or potentially the memory can be re-mapped. Such a solution severely impacts receive-side I/O performance, but it also limits the system's isolation properties as virtual machines can potentially see network traffic of other virtual machines that are using the same real device. This dilemma makes clear that ideally I/O devices which are shared between multiple virtual machines must have minimal virtualization support. For example, they need to be able to hold state, configuration and identity information of multiple, isolated, simultaneously running (virtual) I/O paths. With this minimal hardware support, costs of context switches between virtual machines can be significantly reduced. In this thesis we want to further analyze the minimal set of functions and properties a real device needs to provide in order to enable efficient and secure I/O virtualization.

(Gordon, et al. 2012) proposes "ELVIS" – a mechanism for efficient para-virtualized I/O which eliminates the majority of expensive hypervisor involvements (VM Exits) on the critical I/O path. The authors underline the advantage of para-virtualized I/O over pass-through I/O: in some cases there is no real device that can back up a virtual device, for example, when a virtual disk is stored as a file on the host's filesystem. The authors furthermore state that pass-through I/O requires more expensive hardware and significantly hinders virtual machine migration. As a result, para-virtualized I/O seems to be a better fit for real-world applications. ELVIS promises to improve I/O performance of para-virtualized devices by providing a so-called *exit-less* I/O data path. With ELVIS, virtual machines and the hypervisor run on distinct CPU cores. It uses shared memory areas between hypervisor and virtual machines. When a virtual machine wants to notify the hypervisor about an event, it writes a notification into this area which is constantly polled by the hypervisor running concurrently

on a different CPU core. In the other direction, when the hypervisor wants to signal an event to the virtual machine, it uses inter-processor interrupts (IPIs) in combination with a previously developed exit-less interrupt (ELI) technique (Abel, et al. 2012) to deliver the notification directly to the virtual machine. Previous approaches used expensive, explicit context switches to send a notification from a virtual machine to the hypervisor. Traditional notifications from the hypervisor to the virtual machine on an x86 platform are expensive as well, because, due to hardware limitations (specific to x86 platforms), virtual interrupts cannot actually be delivered to a virtual machine while it is running. The virtual machine must be stopped by the hypervisor before a virtual interrupt can be injected. In (Abel, et al. 2012) the authors show significant performance improvement compared to traditional para-virtualized I/O devices. However, the paper lacks a comparison with pass-through I/O approaches and therefore it is difficult to put the presented performance results in the overall context of I/O virtualization. The results furthermore demonstrate the problem of scalability when using software-based I/O processing: when running more than three virtual machines, the component processing all I/O requests from virtual machines becomes overloaded and cannot process I/O fast enough. While it is potentially possible to involve more CPU cores in I/O processing, it is very likely that software-based I/O processing remains the bottleneck of such a system architecture. From this we can conclude that off-loading certain capabilities to other hardware components is desirable in order to release workload from the main CPU.

2.3.2. Hardware Integration for Improved Network Processing

Hardware-accelerated network processing is an active research field in non-virtualized system architectures. One approach here is to offload network processing to various hardware-based processing units which belong to the platform but are not part of the main CPU complex. This frees up the host CPU for application processing while network processing is accelerated by executing it on suitable hardware. In (Wun and Crowley 2006)

Wun et al. claim that it can be greatly beneficial for network performance if network processing can be offloaded from the main CPU to a set of so-called “micro engines” which just take care of fast, parallel network I/O processing. The authors evaluate the performance benefits if parts of, or even the whole, network stack are implemented in hardware rather than in software as part of the OS. The authors give valuable insights into how I/O performance can be improved by distributing processing onto dedicated hardware components. From these results it is clear that it might be desirable to also enable this on a virtualized system. However, their work looks at using specialized network processors and does not take into consideration the issues of portability across different platforms. Our work, on the other hand, has the goal of focusing on using mainstream network interface cards and generalizing the hardware/software interface to be vendor-independent.

In (LeVasseur, et al. 2008) LeVasseur et al. propose a new way of assigning virtualization-aware devices to virtual machines. The self-virtualizing device partitions its resources into multiple virtual functions which can then be assigned directly to virtual machines. Instead of implementing a complete virtual device in silicon, as the PCI SR-IOV standard proposes, this solution only implements some parts on the real device itself. The virtual configuration space on the other hand is emulated in software within the control device driver. That way it is possible to provide a standard PCI device to a virtual machine, but it leaves the vendor more flexibility about the configuration layout of the virtual device. For example, virtual devices on the same real device can have different PCI device types which would not be possible with a standard PCI SR-IOV infrastructure. The design proposed by the authors has similar goals to our work, in particular the provision of a more standardized virtualized I/O device to a virtual machine while still offering high I/O performance by taking advantage of hardware-based acceleration. However, their I/O infrastructure virtualizes at the PCI level which restricts this solution to particular platforms. Furthermore this research does not look into unifying the data path across different device types which means that ultimately one

has to run different device drivers for different hardware device types inside the virtual machine. Therefore, our research builds on some of the same ideas as described by the authors, but we want to go a step further and allow unified access from virtual machines to a variety of real devices.

In (Dong, Jiang and Tian, SR-IOV support in Xen 2008) Dong et al. introduce a new I/O virtualization architecture for Xen which integrates PCI SR-IOV devices and makes them accessible to user virtual machines. The design relies on PCI SR-IOV compliant self-virtualizing devices and exposes the real hardware interface of the PCI virtual function to the user virtual machine which then runs a vendor-specific device driver in order to control and access the virtual function. For executing privileged operations the virtual functions need to communicate with the physical function, and the authors propose three possible ways of doing this: inter-domain communication, a virtual mailbox mechanism implemented as port-based or memory-mapped I/O access, or a virtual mailbox mechanism implemented in silicon on the real device. They recommend the second approach, because it could be implemented as a vendor-independent and hypervisor-independent infrastructure. The authors note that the PCI SR-IOV specification does not define a communication mechanism between virtual function and physical function which means that at the moment this is vendor-specific and varies significantly between PCI SR-IOV compliant devices which can make it more difficult to integrate these devices into managed virtualized environments. Therefore, in our work, we want to investigate a solution that enables intercepting requests from the virtual machine to the device that require higher privileges in a way that this I/O control path can be generalized across different devices. That way, we do not rely on hypervisor-specific or device-specific solutions for configuring real device resources that require coordination between various unprivileged parties accessing that device.

Dong et al. go into more detail in (Dong, Yu and Rose, SR-IOV Networking in Xen: Architecture, Design and Implementation 2008) describing Xen's approach for exposing PCI

SR-IOV compliant devices to virtual machines that do not use para-virtualization (in Xen terms also called hardware-based virtual machines, or HVMs). In this paper the authors analyze system-level requirements for using PCI SR-IOV compliant devices on a Xen-based platform. They mention the security risks and difficulties with performance isolation when directly assigning real devices to user virtual machines: faulty or malicious drivers can disrupt other user virtual machines or even the driver domain, if there is no virtualization-aware I/O MMU in place which takes care of I/O access control and interrupt virtualization. The solution proposed by the authors assigns the PCI virtual function directly to the user virtual machine, but PCI configuration space accesses to the VF are trapped by Xen and handled in the driver domain. A master device driver sits in the driver domain and controls the real device, including the physical function and potentially some shared virtual function resources. The architecture relies on advanced virtualization support in the platform I/O MMU. The authors claim that their solution achieves better I/O performance than software-based I/O virtualization approaches, and it achieves a potentially more secure design than original PCI pass-through approaches by using a self-virtualizing device architecture, hardware-based memory protection and interrupt virtualization to control access to the real device from different user virtual machines. However, their I/O architecture locks a particular user virtual machine into using a particular hardware which makes virtual machine migration particularly difficult. In our work, we build on some of the ideas presented in this paper to improve I/O performance, but on top of that we want to look into ways of generalizing the interface presented to a virtual machine. That way, we aim to provide high network performance while maintaining virtual machine mobility.

Network interface card virtualization is also a relevant topic for embedded systems research as stated in (Rauchfuss, Wild and Herkersdorf 2010). In this work the authors propose a new network I/O virtualization architecture that supports real-time services and differentiated services sharing a single card. I/O access to the NIC is maintained through a set of virtualized

network interfaces. These do not constantly have real hardware resources assigned, but instead the real device only maintains a fixed, small set of virtual network interface contexts with dedicated hardware resources in silicon while the others are maintained in software and swapped in and out from system memory on-demand. Hardware resources are assigned to virtual network interfaces dynamically depending on load and requirements of the network service running on top. The authors investigate the hardware interface for exposing multiple virtualized network interfaces on a single real device. They use separate receive and transmit packet queues that can be directly accessed via DMA operations, and packet queues for hardware-based virtual interfaces are pre-filled with DMA descriptors. This work by Rauchfuss et al. is relevant to the approach proposed in this thesis, because the authors also investigate the hardware design of network devices and how it suits I/O virtualization. The solution they present differs from ours in that at any given time they expect to have just a single virtual NIC which is hardware-accelerated while we would like to enable multiple hardware-accelerated virtual NICs running simultaneously on the platform. With today's powerful I/O devices this is a realistic goal, in particular in the server market which is the main target of our work. In this context, we also expect to be able to take advantage of hardware-based virtualization capabilities, for example, PCI SR-IOV, which are not yet commodity in embedded systems.

2.3.3. Maintaining Virtual Machine Mobility

In (Kadav and Swift 2009) Kadav et al. investigate the challenges of migrating virtual machines that have direct access to real network hardware. This is the case when using typical pass-through I/O access methods as provided by Xen and KVM and explained in previous sections. In such a setup, it is difficult for the hypervisor to migrate the state of the network device as it is hardware-specific and the destination platform where the VM is to be migrated to might not have the same hardware. To overcome this problem the authors suggest the utilization of so-called "shadow device drivers" which sit between the vendor-

specific device driver interface and the operating system kernel interface and intercept function calls to track kernel objects for network packet transfers and kernel calls for configuring network device settings. When the virtual machine is migrated to another platform which uses a different real network device, then the shadow device driver layer will take down the “old” device driver, initialize the “new” device driver in the virtual machine and connect it to the existing, intercepted kernel resources. The shadow device driver also masks out features on specific hardware, so that only features that are common across all devices are enabled and available to the virtual machine. This work is a rather practical-oriented approach: it does not change the core of the I/O infrastructure to overcome difficulties with existing pass-through I/O technologies, but instead it introduced another layer on top of it. This means that an implementation can be realized very quickly and existing approaches can be used without intrusive changes. The research proposed in this thesis, however, looks at the “core” of the problem which is the virtualized I/O path itself and how it interfaces with the real hardware, and we show how existing interfaces can be improved to provide better performance and portability. In our opinion, a redesign of the internal I/O infrastructure and its interface to the virtual machine and to I/O hardware is required to truly fulfill the potential of high-performance I/O solutions deployed on future virtualized systems.

A more recent approach to live migration of PCI pass-through devices is suggested by Pan et al. (Pan, et al. 2012). This approach focuses on enabling migration of the actual hardware state from one platform to another, which is challenging, because, in traditional approaches using pass-through I/O, the hypervisor does not have access to hardware state. In this new approach, the authors use an additional “self-emulation” layer in the hypervisor to track access to hardware registers, so that those can be stored and restored on virtual machine migration. The solution furthermore uses shared memory between the hypervisor and the device driver to exchange information about what hardware state has to be preserved after

migration. The hypervisor also uses this shared memory to indicate to the device driver that a migration operation has taken place. The evaluation presented in this paper indicates that their technology has superior performance compared to other existing approaches. The proposed "self-emulation" layer is a promising idea in order to track access to hardware registers. However, the proposed solution does not consider virtual machine migration across different hardware platforms and devices of different vendors and capability sets. In order to achieve this, one would need to at first define a unified, virtualized device interface that is exposed to virtual machines instead of the hardware interface of the real underlying device. That way, it would be possible to track hardware I/O access for a variety of devices in the same way and potentially enable migration across a heterogeneous infrastructure. The work in this thesis tries to investigate this idea.

3. Requirements Analysis

3.1. A Hardware/Software Interface to Support Virtualization

The review of prior art in the area of network I/O virtualization shows that the hardware/software interface used by today's systems is not optimized for the sort of virtual environments we target with this work. Dynamic, large-scale cloud computing infrastructures require a more flexible, well-performing approach to exposing powerful network hardware to unprivileged virtual machines. It is clear that, as network I/O devices evolve and hardware support for virtualization increases throughout the whole platform, it is desirable to take advantage of these capabilities as much as possible. However, due to a lack of standardization and common interfaces between software and hardware in a virtualized system, it is not easy to deploy these technologies in an automated fashion.

This work focuses on analysing the hardware and software stack of a virtualized system when processing network I/O. We aim to investigate the interaction between and integration of hardware and software to provide some insight into how to best implement efficient and secure network I/O virtualization in order to fulfil requirements of dynamic, large-scale cloud computing infrastructures.

One of the challenges in this is to clearly define which I/O path functions should be processed in hardware and which functions should be processed in software. Various approaches aim to offload certain capabilities of the I/O path to hardware. For networking, this includes functions like network protocol processing offload functions, for example, TCP Segmentation Offload (TSO) and Large Receive Offload (LRO). We also need to consider data movement operations which can be offloaded from the main CPU onto network hardware by using capabilities like Direct Memory Access (DMA). DMA plays an important role on virtualized systems with high I/O load as it allows for I/O devices to directly put data into the

right memory areas without involvement of the main CPU. In that case, virtual machines can access data more quickly while other parts of the virtualized platform, for example other virtual machines and also the driver domain, remain reasonably unaffected by the high I/O load imposed by a particular virtual machine. Overall, prior art shows that there is significant benefit in taking the main CPU off the critical I/O data path as much as possible and let the specialized hardware handle I/O processing, if available.

Another serious challenge is the design of a device-independent and vendor-independent hardware/software interface that is sufficiently secure and efficient to be used in cloud computing infrastructures. In this work we aim to find the optimal design in terms of performance, security and mobility. But furthermore, our interface design needs to be practical and feasible, so that hardware and software vendors can follow our proposed guidelines and implement better solutions in the short to medium term. In order to propose such a more generic interface, we investigate at what level we can define common boundaries across different devices and, even more important, across different vendors. We evaluate today's hardware and software components with the goal of identifying the common data structures and mechanisms that most, if not all, vendors and devices use today, or potentially could use in the future. If these mechanisms differ in some ways between current devices, then we need to design a new hardware/software interface which is generic enough to cover these different mechanisms, if such an interface design is feasible. Assuming that hardware implementations and software/hardware interfaces of different vendors differ significantly, we want to develop a mechanism to map those existing interfaces onto our new API that exposes hardware capabilities to virtual machines as so-called *network acceleration engines*. Such a prototype implementation built on existing hardware can show the potential of our proposal. In this case, current hardware support for our I/O architecture will not be ideal, but we hope that we can still show an I/O architecture which introduces low overhead and is flexible enough to be deployed in an automated

fashion at a large scale. Looking into the future, it is the goal of our work to propose a new, standardized software/hardware interface to best support I/O virtualization in cloud computing infrastructures. In this context, we aim to outline to vendors and system software developers how an ideal realization of the NAE API would look like.

The core part of the analysis is the investigation of the integration of the following components:

- Network I/O device (hardware interface)
- Hypervisor (thin, privileged software interface)
- Driver domain (privileged software interface)
- Virtual machine (unprivileged software interface)

At the hardware level, the challenge is to design an interface with the right control points to be efficiently used in a virtualized system. Ideally, different vendors need to agree on common data structures allowing multi-vendor management stacks on top of network devices. In order to support and encourage such a development we investigate the minimal set of data structures that is required to implement virtualized network I/O on different types of network I/O devices.

Within the virtualization layer (the hypervisor) there needs to be sufficient low-level support for various functions on the I/O path. For example, the hypervisor needs to support functions that require privileged access to platform resources in such a way that they can be executed in a safe and efficient manner on the critical I/O data path. As we have seen in the analysis of prior art, this is very challenging in itself and one of the main limitations of current technologies and system architectures. The hypervisor needs to be able to efficiently and safely expose particular hardware resources, like device registers, I/O memory areas and device interrupts to a virtual machine. Furthermore, it needs to enable efficient communication paths and context switching capabilities between different virtual machines

running on the system, the involved driver domain(s) and potentially also the hypervisor itself.

Another challenge introduced by virtualization is dividing control of a hardware device and its resources between the privileged driver domain and unprivileged virtual machines trying to access the device. The system architecture needs to support isolation between virtual machines sharing the same I/O device, but potentially some control can be given to the virtual machine. A common example of this would be to enable the virtual machine to schedule its own workloads better over the virtualized I/O resources it has been granted access to. There needs to be a clear definition of what the virtual machine can do with its own (virtualized) hardware resources and it must be guaranteed by the virtualization layer that the virtual machine cannot access hardware resources which it does not own.

The driver domain operating system needs to provide sufficient infrastructure in kernel space and in user space to control hardware resources. Ideally, both driver domain and virtual machine can cooperate to some extent in order to more efficiently process network I/O, in particular for packet movement and interrupt delivery events. A well-designed interface at this level is critical from a performance point-of-view, but an efficient implementation is challenging as we have seen when looking at prior art solutions.

3.2. Virtualized Infrastructures Supporting Cloud Computing

Cloud computing, virtualization, big data and network convergence are said to be the biggest technology trends of today's IT landscape driving the demand for significantly improved network I/O performance (Lane and Bonina 2012). Cloud computing infrastructures have to deal with many technical challenges. Cloud infrastructure providers incorporate many critical technology components in order to address these. Virtualization enables to build a very dynamic and cost-efficient environment that can be adjusted to customer needs, functional requirements and current load of the infrastructure.

Cloud infrastructures are by nature *multi-tenant* platforms in that different customers share the same physical infrastructure and the cloud infrastructure provider needs to ensure that sufficient isolation between customers can be guaranteed. Agreements between provider and customer usually cover performance isolation and data isolation. In the context of network I/O virtualization this means that data flows of different customers need to be sufficiently separated and network utilization of different customers' applications should not affect each other. Any I/O architecture needs to satisfy this fundamental requirement.

One of the main features of virtualized infrastructures is that they allow moving workload around in a very dynamic fashion. Through virtualization, applications can be moved to platforms in different physical locations and with potentially different capabilities. This occurs without the application noticing that it has moved. Potentially the application can be moved without any downtime, if the virtualization software supports so-called *live migration*. Virtual machine migration is one of the key technologies which cloud infrastructure providers need to be able to take advantage of in order to run a cost-efficient infrastructure and fulfill customer demands. On most cloud platforms, customers come and go. The number of customers might vary depending on various circumstances. Furthermore, demands of individual customers might change over time. It is critical that a cloud computing infrastructure is sufficiently flexible to accommodate all these requirements – virtual machine migration is a key technology enabler in this context. However, virtual machine migration presents a serious challenge for I/O architectures. In particular, tight hardware integration must be thought through very carefully, so that the I/O path remains reasonably hardware-independent and does not restrict mobility of the virtual machine. A restrictive I/O architecture potentially annuls the benefits of virtualization technology.

Cloud computing is offered as a service on top of large-scale, highly-networked infrastructures. Cloud infrastructures may span across different physical locations. Various cloud infrastructure providers have multiple data centers spread around the world which are

linked together. For a user running applications on the cloud, various technical details of the underlying infrastructure are hidden. This includes, for example, physical network topologies and the physical location of servers. Furthermore, details of the exact platform hardware are transparent to the application running within virtual machines on the cloud infrastructure. This is an important characteristic of cloud computing environments, as these large-scale, multi-data center infrastructures potentially consist of significantly heterogeneous underlying hardware. Running over a large set of hardware platforms which are not identical presents a serious challenge for I/O architectures. At first, in order to stay transparent to the application running in the virtual machine, it is critical to design an I/O path that is sufficiently generic in order to cover heterogeneous platforms. The application shall see a similar interface, no matter what hardware it runs on. Different hardware should be able to expose different capabilities (only that way can we take advantage of more powerful hardware), but the configuration interface and the instantiated I/O path shall remain the same. Secondly, cloud infrastructure management stacks need a common API to configure and control a platform and its I/O devices in an automated fashion. When operating at this scale, it is impossible to control hardware through individual, vendor-specific APIs.

3.3. Initial Design Principles

The main part of this thesis is to design and develop the Network Acceleration Engine framework which can be seen as a new system architecture and its associated I/O infrastructure. The NAE framework enables virtual machines to access a heterogeneous set of hardware-based network accelerators. The fundamental requirement for a platform supporting the NAE framework is that it must be a *virtualized* system. With this we mean that it runs a hypervisor or an architecturally similar virtualization layer. After analysing technologies used in prior art and their results, we furthermore decide that we need to keep

the instantiation of a so-called *driver domain* on the platform. We call it the *network control domain* as we are looking into processing network I/O. The network control domain is required, because we expect that we need to keep some control components outside of the virtual machine in order to manage shared device resources and carry out privileged operations. Furthermore it is required in order to implement a common management software stack on all platforms which are part of the cloud infrastructure. That way cloud infrastructure providers can easily manage their hardware at a large scale and in an automated fashion. We introduce the concept of the network control domain in more detail in later sections.

Another design principle of the proposed architecture is that, while we want to decouple the virtual machine from the hardware interface of the underlying network accelerator, we expect that some vendor-supplied, device-specific code needs to reside somewhere on the platform. This is required because, as we explained before, the software/hardware interfaces of today's devices are implemented by vendor-supplied, device-specific code. This code is very complex and therefore it would be very challenging to completely rewrite it and integrate it natively into a new I/O architecture. Any vendor would find that such a major rework task of their drivers is unfeasible which would make adopting and supporting the NAE framework unattractive to them. Therefore, we aim to integrate these vendor-supplied functions into the new NAE framework as smooth as possible and with just minimal code changes. We deploy these pieces on the platform hidden from the management software stack and from virtual machines.

As we have seen from the analysis of prior art in this research area, it is challenging to provide improved I/O performance whilst maintaining flexibility. In order to do both, a new I/O architecture is required covering every bit of the I/O path from low-level hardware functions to system software running in virtual machines. We have seen that many previous approaches successfully improve I/O performance on the data path. Various promising

results came out of those research efforts and those ideas and technologies should ideally also be deployable as part of our proposed I/O framework. Therefore, it is important to note that it is not our goal to significantly improve performance of the I/O path compared to existing and upcoming direct device assignment mechanisms – as we have analysed in the previous sections, many of the newer approaches in this field already achieve very good I/O performance numbers. Instead, the main goal of this work is to be able to provide an efficient I/O path that is flexible enough to be used in highly dynamic cloud computing environments. In this context, it is important to evaluate and understand what processing can happen in software and what processing should be done in hardware. The right design is challenging, but critical. It must enable taking advantage of hardware acceleration while maintaining sufficient hardware independence. A virtualized I/O path exposed to virtual machines running on cloud computing infrastructures needs to provide a sufficiently generic interface.

One of the biggest hurdles that previous approaches did not overcome is the design of a vendor-independent and device-independent I/O interface, so that virtual machines can efficiently use advanced capabilities of real I/O devices without being tied to a particular hardware interface. A major challenge in designing such an interface is that different devices ship with different device drivers containing a significant amount of code in order to initialize a device, program a device and initiate data transfers through the device. Different vendors have different approaches to how their devices are to be configured. Trying to unify this control/configuration interface across all devices and vendors would be unfeasible, if not impossible. On the other hand, the data transfer interface of many I/O devices is actually reasonably comparable between different vendors. Apart from moving data into and out of device memory, there is not much more a data transfer interface needs to do. We analyse this further in one of the next sections, but, in summary, we can capture that the control interfaces of devices vary significantly while the data transfer interfaces do not.

Looking at these two types of interfaces on a device, it is important to also look at their requirements for providing a safe and efficient I/O path from virtual machines to real devices, and vice versa. The data transfer interface needs to be as fast as possible. It represents the critical path in order to enable high-performance I/O. The configuration/control interface, on the other hand, does not need to be that fast. It mainly implements operations that are carried out infrequently, potentially at device start-up or device reset, and every now and then during device operation.

As a result of these findings, we suggest that it is best to clearly separate I/O interfaces into control functions and data transfer functions. In the following, we will call those *control path functions* and *data path functions* respectively. More precisely, we call them *virtual* control path and *virtual* data path as they are exposed to and accessed by virtual machines. Furthermore, concluding from these findings, when we talk about providing a unified device interface to virtual machines, we will not unify the control path and data path of different real devices in the same way. In particular, when unifying the control path between different real devices, we can consider software-based approaches, because the control path is not as critical to I/O performance as the data path and because unifying the control path between real devices of different vendors would be a significant effort. Instead, we decide to have that software-based control path backed up by a smaller portion of vendor-specific control code which is hidden behind the NAE API and invisible to applications running inside virtual machines. The data path on the other hand is easy to unify across devices, as we state above and analyse further in the next sections. Furthermore, it must be fast and therefore more direct between the virtual machine and the real device. Ideally there should not be any software-based intermediate component involved on the data path. As part of this thesis, we would like to investigate further what type of software-based processing components should be on the data path, if any, and how much impact on performance they have.

This separation of control flow from data flow is one of the key design principles of our proposed I/O architecture.

In the following section we look a bit more in detail into current I/O devices and how their hardware architecture relates to our proposed Network Acceleration Engine framework. We aim to evaluate how their design fits I/O virtualization approaches, for example, how their design suits virtualized systems where functions and resources of the device are exposed to virtual machines. We also want to investigate how their data path functions and control path functions are implemented and how they are typically made accessible to system software. Such an evaluation of current hardware is important in order to get a first idea of whether or not we can feasibly build an I/O architecture satisfying the requirements we list above. Furthermore, we want to design our new I/O virtualization architecture in a way that it can deal with existing hardware capabilities and therefore we need to understand how current hardware works. However, the main goal of the evaluation of existing devices is to show where their design and implementation can be improved in order to better support I/O operations on virtualized platforms leading to a better hardware design which can feasibly be implemented by all vendors.

4. Hardware Design Evaluation

4.1. Overview

As system virtualization has developed into a mainstream technology, many new I/O devices have been - at least partly - designed with virtualization in mind. For example, with PCI, vendors invested in multi-function devices which provide *multiple physical functions* (PFs) on a single device, and these can be used independently of each other, for example by different user virtual machines. PCI SR-IOV, as specified in (PCI-SIG n.d.), extends this approach by providing *multiple virtual functions* (VFs) per physical function. In this case, virtual functions operate independently of each other and can also be assigned individually to different user virtual machines. Furthermore, within a single physical or virtual function, most devices can isolate network traffic onto *individual packet queues* which can, in some cases, like for example when using Intel's VMDq technology as described in (Chinni and Hiremane, Virtual Machine Device Queues 2007), be associated with different user virtual machines. Looking at these trends in I/O device technology, it is clear that vendors care about providing isolated, independent data channels on a single piece of hardware. Vendors can also differentiate their hardware by the amount of independent data channels they can provide in silicon. Mostly, the idea is: the more the better.

When comparing the actual I/O data channel implementations of different devices and vendors, our findings indicate that they do not differ significantly from a design point-of-view. Typically bi-directional data flow is implemented with a send and receive data queue. Data queues are implemented as circular buffers: for the send queue, the buffer is written by software and read by hardware; for the receive queue, the buffer is written by hardware and read by software. The circular buffer resides in system memory. As explained in the previous section, it typically does not contain the actual data that has to be transferred to or from hardware. Instead, it contains so-called data descriptors which contain information

about the actual data which has to be sent or has been received. Most importantly data descriptors describe where in system memory the actual data can be found, for example storing a memory address. Furthermore, if the device is designed so that it can deal with data of different sizes, then the data descriptor also conveys the size of the data. The circular buffers used for implementing the I/O channels contain data descriptors rather than the actual data in order to improve scalability: the descriptors reside in a virtually contiguous memory space, but the actual data can be held anywhere scattered around in system memory. The actual data transfer between the I/O device and the memory address specified in the data descriptor is realized through direct memory access (DMA) transactions.

Device configuration is typically done through a set of device registers that can be modified by privileged software. Most privileged software controls this register set by mapping it as a whole into contiguous system memory, and then it can modify configuration registers by writing into that memory space, and it can read configuration register by reading from that memory space. This mechanism is called *memory-mapped I/O (MMIO)*. Memory-mapped I/O is common across different architectures, like, for example, x86 and ARM, and therefore it seems to be a good choice for device configuration.

Not all device settings are suitable to be exposed via a memory-mapped I/O space though. Once MMIO space is mapped, it has a fixed size, and so it is not suitable if configuration options are of a varied size. As an example, a common feature is to store a list of multicast addresses on the device, and then program the device to deliver all packets to those addresses to the platform (while any other packets to other multicast addresses will be dropped). In that case, the list of multicast addresses could be quite long and contain a large number of entries. If we wanted to store all these in MMIO space, then we would have to reserve a significant amount of memory for that. Instead, most vendors have a more flexible mechanism of configuring the device by sending configuration commands to the hardware. Some devices we have analysed and used for our prototype allocate an additional interrupt

for this mechanism. We describe more details about this in the following sections. With the proposed NAE architecture, all configuration commands issued from the user virtual machine go through the software-based virtual control path. Configuration commands are part of the NAE API and they are unified across all hardware-based network accelerators.

4.2. Intel 82599 and Intel 82576

The Intel 82576 controller operates at 1 Gigabit per second while the 82599 controller operates at 10 Gigabit per second. Both devices are otherwise very similar in design which leads to the assumption that both devices use the same core building blocks in silicon. The 82576 was one of the first (if not the first) PCI SR-IOV compatible network cards, and even though its resources and capabilities are limited, it significantly shaped the software support for PCI SR-IOV, because it was used for most early system software development. The 82599 came out later and was more or less a faster version of the 82576 with more internal resources and some additional features.

Both Intel controllers have a very clear and simple design for the data path. The data path consists of a configurable number of I/O channels each of which are implemented through circular buffers as described in the previous section. Every I/O channel consists of two circular buffers: one representing the send queue and one representing the receive queue. Every circular buffer is controlled by a head and a tail pointer. Hardware writes the head pointer when new data has been sent or received, and software writes the tail pointer when data is available to be sent or when received data has been processed. As head and tail pointer values are potentially modified every time a packet is sent or received, access to those pointer values need to be fast and efficient. Also, it is clear that access to those needs to be part of the data path functions, and therefore, in a virtualized system, access to those needs to be possible directly from the user virtual machine. Both Intel controllers expose the head and tail pointer values as device configuration registers as part of the device's memory-

mapped I/O space. Unfortunately, those registers (four of them per I/O channel, for example, two for the receive queue and two for the send queue) are mixed together with other device configuration registers. Also, registers are mixed together across all I/O channels. For example, all receive queue tail pointers are stored in sequence one after the other at a fixed offset in MMIO configuration space. And all receive queue head pointers are stored together at a different offset. This design makes isolation of the control and operation of I/O channels very difficult, as access to individual registers cannot be separated out that easily. Control path registers do not necessarily need to be accessed directly from user virtual machines and therefore this issue is not critical for this type of register. However, for example, head and tail pointers of circular buffers are data path registers and therefore access to them really needs to be designed differently.

With the early introduction of PCI SR-IOV on both Intel controllers, the existing system infrastructure did not support communication between the privileged physical function (PF) and its unprivileged virtual function (VF). Even today, there is no standard way of achieving such a communication implemented in any hypervisor or operating system. The Intel controller supports sending of messages between PF and VF. Messages are 64 bytes long, and device control registers exposed through the device's MMIO space are used to synchronize sending and receiving messages via a so-called *mailbox* capability. Through the mailbox, both PF and VF can send messages and notify the other end about the transmission or reception of a message. Notification is realized by the hardware generating an interrupt. This additional interrupt needs to be managed per device on top of all interrupts associated with data queues. The mailbox itself is implemented as a shared device register which can be accessed by both PF driver and VF driver. The mailbox capability is used for the following operations:

- The PF can report general errors, and for example, link status information to the VF.

- The VF can send requests to the PF which require privileged access to the hardware. This includes, for example, the configuration of VLAN tagging or multicast receive filters. A VF could not program that itself as these resources are shared between all VFs which are instantiated on the same underlying hardware.

As the mailbox capability is a control path feature, an implementation does not necessarily need to enable very fast communication between PF and VF. This channel is purely used to configure device resources, and it is never accessed as part of a data path operation (when sending or receiving packets). Therefore, when running the Intel 82576 and the Intel 82599 controllers in our proposed NAE architecture, we do not expose the hardware-based mailbox and instead integrate the functions into our software-based virtual control path as part of the NAE API. These control functions are fundamental network control functions and not in any way specific to Intel hardware. Therefore a vendor-specific approach to configuring these resources is not desirable and not necessary. Network hardware that adapts the NAE architecture does not need to implement a hardware-based communication path between PF and VF. Instead a single mechanism can be used for all devices.

4.3. Mellanox ConnectX

The Mellanox ConnectX is a network controller supporting Infiniband and Ethernet. It operates at 10 Gigabit per second. Firmware controls whether a device is used in Infiniband mode or in Ethernet mode. Infiniband puts different requirements on the underlying hardware than Ethernet and as a result the ConnectX hardware is designed quite differently from the Intel cards which support only Ethernet transport.

One of the most significant capabilities of the ConnectX is that it has a memory management unit (MMU) on the controller. That is programmable from privileged software to create memory translation and protection tables which control access to all hardware resources. Once the MMU is configured, hardware can directly read from and write to permitted

addresses in system memory. The MMU is a powerful tool in this context, because it can control access for individual applications or virtual machines to specific ConnectX hardware resources at a very fine granularity. For example, it can provide isolation between individual packet queues that are, for example, assigned to different user virtual machines.

Another differentiator of the ConnectX is the huge amount of simultaneous I/O data channels it can support. The card used for our experiments supports 16 million I/O data channels (each consisting of a packet send queue and a packet receive queue). As with other network cards, the data channels are implemented as virtually-contiguous, circular buffers residing in system memory. Using those, data is transferred between the ConnectX and system memory through DMA transactions. Here, the ConnectX does not differ from the Intel cards, underlining that the virtual data path can be very easily unified across significantly different hardware.

Due to its Infiniband-compatible nature, the ConnectX design includes a quite complex, but flexible event delivery infrastructure. Rather than a simple design supporting one or two interrupt vectors per queue (one for reporting a send event and one for reporting a receive event, as most standard Ethernet cards do), the ConnectX implements an event notification mechanism based on an additional set of event and completion queues. Those queues are implemented very similar to the usual data queues: they are implemented as virtually-contiguous, circular buffers residing in system memory. The ConnectX we use for testing supports up to 16 million completion queues which can be associated with data queues. Multiple data queues can share completion queues, or they can be assigned individually. Here it becomes more apparent why the ConnectX does not just use interrupt vectors associated with I/O channels in a one-to-one mapping: in order to support such a huge number of simultaneous I/O channels, it would need a significant number of interrupt vectors per network card. Completion queues report on what data the hardware has already processed, while event queues also report generic hardware events, like, for example, link

errors. Event queues can be configured to link to interrupts, and so the device can, for example, generate a PCI MSI-X interrupt for particular events.

This design of the ConnectX event delivery infrastructure has both advantages and disadvantages for usage in a virtualized system. If a virtual machine requires direct access to an I/O data channel, then it also needs access to the associated completion queue which is written by hardware in order to indicate what data transfers have completed. As hardware reports completion of data transfers to a completion queue, access to that queue needs to be given to the virtual machine as well. Or, as an alternative, it would be possible to keep access to the completion queue restricted to the network control domain which then reports completion of a data transfer to the virtual machine via the virtual control path. The advantage of the latter solution is that less hardware information needs to be directly exposed to the virtual machine. Also, it would be quite simple to unify reporting of completion events across different hardware: hardware-based completion reporting stays in control of the hardware-specific, vendor-supplied control driver that is part of our proposed NAE I/O architecture and resides in the network control domain, and reporting to the virtual machine is done via the software-based, generalized virtual control path introduced by the proposed NAE I/O infrastructure. However, there might be a performance hit with this approach, since reporting of data transfer completion needs to be routed through the network control domain. Depending on the workload, this might have non-negligible impact on latency. Checking on data transfer completion is a data path feature and therefore it is time-critical. Hence, from a performance point-of-view, the former solution might be more suitable. In this case, the virtual machine has direct access to the completion queue and no intervention from the network control domain is required. As the completion queue is implemented as circular buffer used for DMA transactions, it will be straight forward to expose it to the virtual machine: it can be exposed and configured exactly like the data queue used on the hardware-based virtual data path. For this solution the MMU on the

controller is significant, because it allows isolating completion queues from each other and making sure that only virtual machines (or applications) which have been granted access to a particular completion queue can use it. As the ConnectX has an internal MMU, it does not rely on the system to provide memory protection (an I/O MMU) for individual hardware resources.

The ConnectX uses so-called configuration commands to control more complex device features that cannot be easily set by writing simple one-value registers. For this the ConnectX maintains a command register which is held in the MMIO configuration space. The command register can be written by privileged system software and it is used to, for example, initialize the firmware of the device, query device parameters, configure the MMU, set up and partition hardware resources like, for example, packet queues, and start or stop device operation. The command register includes a device-specific command code and command input and output parameters. When a command is executed on the device, the device can write results back as output parameters immediately, or post results on an event queue. When deployed in the NAE architecture, this privileged command interface is not directly exposed to the virtual machine, but instead it stays under the control of the device-specific, vendor-supplied control module that is part of the NAE I/O infrastructure. The virtual machine can configure its portion of the real device through the virtual control path as provided by the NAE API.

5. Network Acceleration Engine (NAE) Architecture

5.1. Virtualized System Architecture Overview

In our proposed architecture the virtualized system consists of a hypervisor, one or more *privileged* driver domains which own particular hardware, and one or more *unprivileged* user virtual machines. One or more of the privileged driver domains own networking hardware, and so we call these *network control domains*. A system which is connected to the physical network requires at least one real network device and one associated network control domain. For now we assume that our virtualized system runs a single network control domain which controls all networking hardware. However, the design also allows running multiple network control domains, for example one network control domain per real network device.

If a user virtual machine wants to take advantage of hardware acceleration provided by the NAE architecture, then we will assign it a *virtual network I/O device* which looks like a typical network I/O device to applications running inside the virtual machine. This virtual I/O device resides in virtual machine memory and is the “frontend” of the virtual I/O path which connects a user virtual machine to a particular real device and its associated privileged virtual control path running in the network control domain. As the NAE virtual I/O device looks like any other networking device to the OS, users can use standard OS tools and networking tools to configure the virtual I/O device. Figure 5 shows a very rough structure of our virtualized system and its components, and where the different parts of the Network Acceleration Engine API sit. A more detailed description of all components and their connections will be given in the following sections.

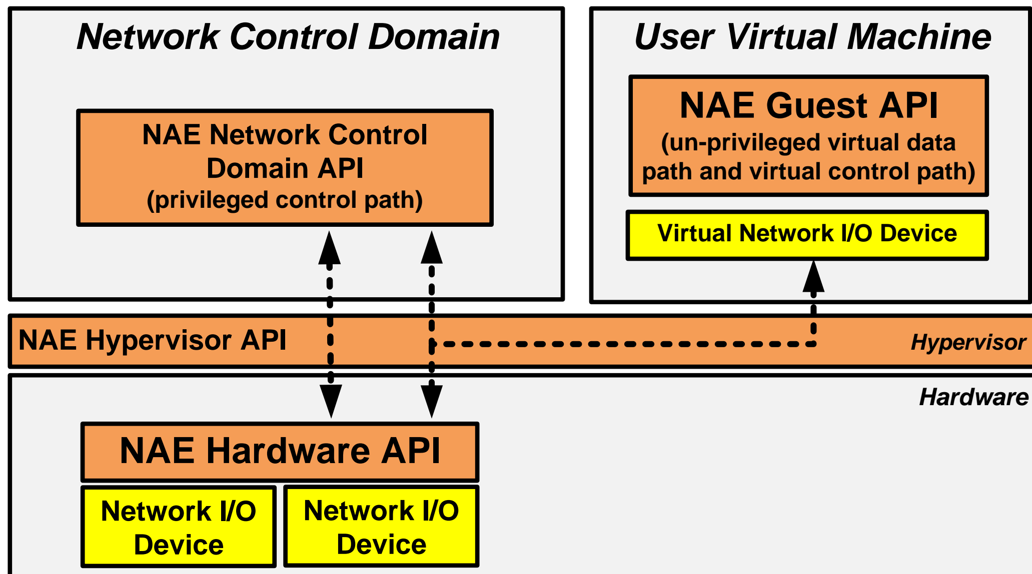


Figure 5 Overview of NAE system APIs

Every virtual network I/O device can support multiple NAE I/O channels. A channel is used to push data to and from the user virtual machine. Channels are bidirectional and can be hardware-based or software-based. A NAE I/O channel consists of at least two network packet queues – one queue for incoming packets and one queue for outgoing packets. The I/O channel design is described further in the upcoming sections.

The virtualized I/O path is clearly separated into a *virtual data path* and a *virtual control path*. This separation of information flow is important, because control operations and data transfer operations have very different characteristics and service requirements, and by separating them out it is easier to provide the best design and implementation for both of them. The NAE API clearly categorizes functions into control path functions or data path functions. Furthermore, the decoupling of control and data path allows plugging in different types of virtual data paths under a single network I/O device which is under the control of a single virtual control path. For example, with this design we can switch the virtual data path between hardware-based and software-based instantiations without any disruption of network applications running inside the user virtual machine. In order to do that, we need a clearly defined virtual data path API which can be implemented by both software and hardware components. This capability makes our architecture incredibly flexible which is

significant for implementing cloud infrastructure features such as virtual machine migration or resource load balancing. A more detailed use case is described in the section on the *Software-based Virtual Data Path*.

5.2. NAE Guest API

5.2.1. Overview

An important goal of the network acceleration engine API is to make the guest OS network I/O path more suitable for deployment on virtualized systems. Most importantly we want to provide an I/O infrastructure capable of virtual machine migration across different hardware platforms, and we want to develop a system design which enables unified access to different network hardware from both the user virtual machine and the network control domain. Such a generalized, virtualization-aware API will significantly enhance portability and manageability of virtual machines and their applications running on cloud infrastructures. The major goal here is to provide a vendor-independent network device configuration and control interface, and to make all network functions which are required in a virtualized infrastructure accessible to the user virtual machine.

The virtual network I/O device assigned to the user virtual machine runs a simple and light-weight device driver which is part of the NAE framework. In contrast to a traditional network device driver, the NAE driver does not contain any code which is tied to a particular hardware or vendor-specific implementation. Most traditional network drivers contain a lot of complex code to program firmware, and all their configuration functions are device-specific. With NAE we remove a lot of this complexity in the virtual machine which improves portability and maintainability.

A virtual device consists of mandatory properties and optional features. Mandatory fields are present on every virtual device and can be read or written easily through the guest API. On a virtual network device, for example, mandatory properties include a primary MAC

address, and the number of I/O data channels it supports. Every virtual I/O device needs to define a field containing a feature bitmask. We use a simple bitmask to discover features of the NAE-compliant virtual device. If a virtual device supports a specific feature, then it will set the related bit in its feature bitmask and the driver running within the user virtual machine can then enable and use that feature, if it has the capabilities to do so. In theory, it is possible that a particular driver does not support certain features, and in that case it will simply not configure or use them. This can, for example, be the case if a user virtual machine runs an old driver which has not yet been updated with a new feature a particular I/O device supports. In that case, the feature remains unused. The capability set advertised through the virtual control path needs to be implemented on both the underlying I/O device and the device driver residing in the user virtual machine.

The NAE guest API allows the guest OS to control capabilities of the network acceleration engine it owns. Typical network functions that need to be controlled or customized from within the guest OS are setting checksum offload capabilities, TCP/UDP processing offload capabilities, bandwidth or QoS controls, VLAN tagging and multicast filters. It should also be able to configure MTU sizes of the virtual device and enable/disable promiscuous mode, reset the virtual device into a known good state and query status and statistics of the device, for example, link status and speed, packet counters and so on.

Most of these functions require coordination of resources controlled by the privileged control path. This is crucial on a virtualized system, because physical resources are shared between multiple virtual machines. On a non-virtualized system, a network driver completely owns the underlying hardware and therefore no further coordination is required. On a virtualized system, however, resources are shared and, if the system supports the NAE API, the privileged control path controls and coordinates hardware resource allocation between multiple, possibly concurrently running, virtual machines. In particular, if the virtual machine wants certain bandwidth limits configured for its network traffic, then this needs to

be coordinated with other bandwidth restrictions that might need to be applied in hardware. These can, for example, be traffic regulations programmed in by other virtual machines, or bandwidth controls that are applied to some or all virtual machines as requested by the virtualized infrastructure management. Typically in an environment where resources are shared between different, potentially competing parties it is essential that all resources are strictly controlled. Another network-specific resource that needs to be shared and coordinated between virtual machines is VLAN membership information. Multicast Ethernet address filters need to be controlled by the privileged control path as well, as multicast filters on the real device are limited in size and only a privileged component should have access to program these. Through the NAE API all these functions are standardized across all supported network hardware which makes our solution a well-suited approach for large-scale infrastructures requiring automated network configuration.

The NAE guest API is split into a virtual control path (VCP) and a virtual data path (VDP). Both parts require very different capabilities from the underlying I/O infrastructure design and implementation: the VDP needs to have a simple and straight-forward design, so that it can be used to access different network hardware. It needs to be fast and efficient, but it also needs to be safe. For example, a VDP assigned to one virtual machine should not in any way interfere with the VDP assigned to another virtual machine, and a virtual machine should always just be granted access to its own VDP and associated resources. The VCP should be able to expose more complex features, so that most, if not all, hardware features can be accessed and controlled from within a user virtual machine. Access to the VCP does not necessarily need to be fast though, because control and configuration functions are not used frequently during I/O data transfer. Because of these very different requirements the NAE I/O infrastructure clearly separates control path functions from data path functions.

5.2.2. Virtual Control Path

5.2.2.1. Virtual Configuration Space

The Virtual Control Path is designed as a virtualized memory space for reading and writing configuration values. This configuration space is mandatory and standardized for all NAE-compatible virtual devices. It includes a simple, defined set of register values that can be read or written. The virtual configuration space defines fields that identify a particular device and device type, and advertise its features. This includes hardware capabilities like network processing offload functions. While we propose a generalized virtual device interface, it is an important goal of our solution that individual vendors can still differentiate with their hardware by advertising their complete high-performance network processing capability set through the VCP.

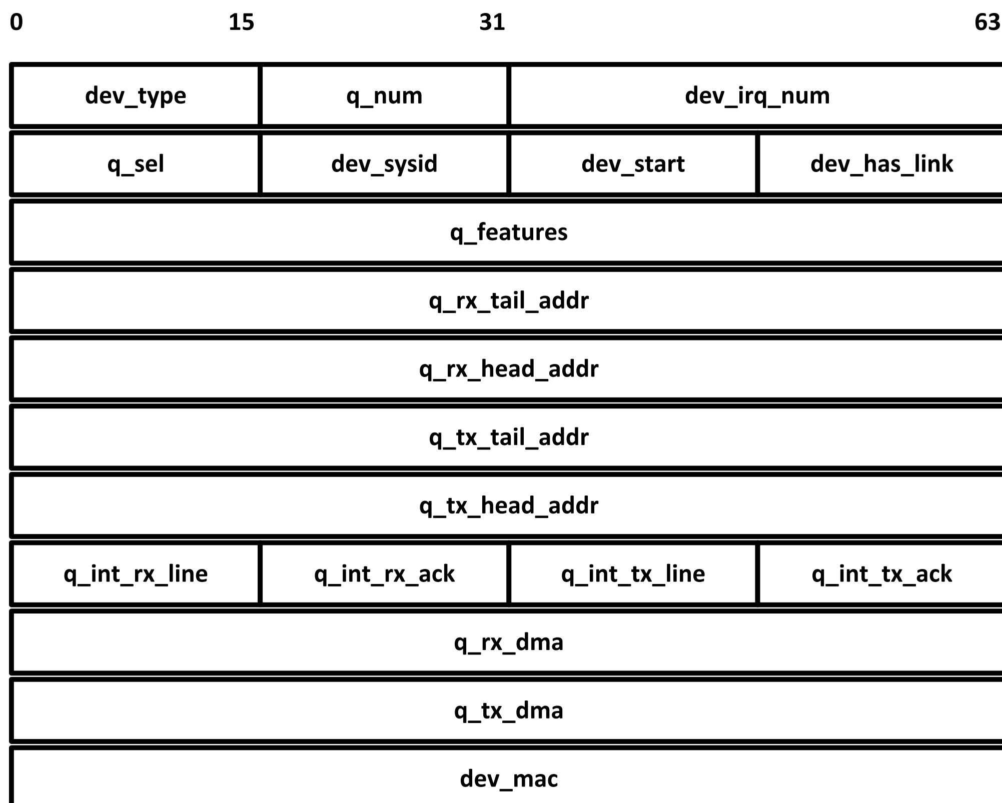


Figure 6 Virtual Control Path memory space layout

The VCP is also used to communicate live device statistics and link status information. Most importantly the configuration space exposes information about the virtual data path and its configuration. A virtual device advertises its interrupt capabilities and the VCP is used for

setting up the event notification mechanism providing an appropriate communication channel between the real device and the guest OS. The configuration space furthermore exposes the number of supported simultaneous I/O channels (send/receive packet queues) per VDP. We do not expose the full hardware interface to the guest OS as existing approaches do, but instead we select the minimum information required for the guest OS to safely access (a portion of) the real device. Figure 6**Error! Reference source not found.** shows the layout of a minimal required configuration space and the following table describes the mandatory fields for each NAE-compatible virtual device:

Table 1 Description of Virtual Control Path Fields

Field name	Size (bits)	Description
dev_type	16	ID describing the device type
q_num	16	value indicating the number of supported I/O channels
dev_irq_num	32	value indicating the number of supported interrupts
q_features	64	bitmask indicating what features are supported
q_sel	16	value written by driver to indicate which I/O channel to program
dev_sysid	16	ID identifying the device in the system
dev_start	16	value written by driver to start/stop the device
dev_has_link	16	value indicating link status
q_rx_tail_addr	64	value written by driver to define RX tail pointer of currently selected I/O channel
q_rx_head_addr	64	value written by driver to define RX head pointer of currently selected I/O channel
q_tx_tail_addr	64	value written by driver to define TX tail pointer of currently selected I/O channel
q_tx_head_addr	64	value written by driver to define TX head pointer of currently selected I/O channel
q_int_rx_line	16	value written by driver indicating RX IRQ number
q_int_rx_ack	16	value written by driver acknowledging RX interrupt
q_int_tx_line	16	value written by driver indicating TX IRQ number
q_int_tx_ack	16	value written by driver acknowledging TX interrupt
q_rx_dma	64	value written by driver defining RX DMA address
q_tx_dma	64	value written by driver defining TX DMA address
dev_mac	64	primary MAC address

One important property of the virtual configuration space is that I/O channel information does not need to be repeated even if the device supports multiple simultaneous I/O

channels. Instead, the configuration space contains a *selector* field (called *q_sel* in the table above) where the driver writes the index of the I/O channel it wants to modify or view, before it can write or read the individual I/O channel properties, like for example addresses of head and tail pointers, or DMA addresses used for storing the circular buffers. Using this mechanism, it is easy to support a large number of simultaneous I/O channels without increasing configuration space memory. The number of simultaneous I/O channels is only limited by the number of bits available in the index (here, we use 16-bit).

When the virtual device starts up in the virtual machine, the driver at first reads the device type, the number of supported I/O channels and the number of supported interrupts. Ideally, it should support two interrupts per I/O channel, for example, one for the send queue and one of the receive queue. That way we can tune interrupt rates independently for transmit and receive operations. However, that is not strictly mandatory, and interrupt notifications can be shared across multiple queues. The driver can determine the allocation by reading the *q_num* and *dev_irq_num* fields.

The driver then configures each I/O channel individually. It starts by reading the I/O channel's feature bitmask from the associated field. Each I/O channel has its own feature bitmask, so that in theory we can support a device with multiple I/O channels which have varying capabilities. Here, the *features* field is 64-bit long, which allows for a significant number of features that a device can support. It is important that the proposed system design enables most, if not all, hardware features to be exposed to virtual machines – this is one of the key goals of the Network Acceleration Engine architecture. While we provide a generalized, hardware-independent and vendor-independent interface to the virtual machine, we need to make sure that vendors can still sufficiently differentiate their network hardware from competitors and this typically is done through better performance or a superior feature set. In traditional approaches, a lot of hardware features cannot be exposed at all. For example, when a virtual machine is configured with a software-based data path

that does not utilize any hardware capabilities, then powerful network hardware is hidden to the end user. This is especially the case in most of today's cloud infrastructures where powerful hardware is simply white-boxed in order to enable virtual machines to be easily migrated across the whole infrastructure. One of the major goals of the NAE design is to make powerful network hardware visible and usable to applications running in virtual machines, and the VCP design is crucial for enabling exactly this.

The driver then reads the I/O memory addresses of the queue head and tail pointers that are used to set up the associated virtual data path. It also reads the virtual device's primary MAC address. When the virtual data path initializes, the driver allocates DMA memory to use for storing the circular buffers, and then advertises the DMA addresses to the privileged control path by writing their values into the *q_rx_dma* and *q_tx_dma* fields. The fields *q_int_rx_line* and *q_int_tx_line* are written by the driver after interrupts have been set up within the virtual machine, and the driver advertises this information to the privileged control path to confirm which IRQ numbers are used. The fields *q_int_rx_ack* and *q_int_tx_ack* can be written by the driver to acknowledge an interrupt.

The layout outlined here gives an idea of mandatory fields that are required to configure a simple virtual data path. There are further properties that need to be negotiated and configured through the VCP. For example, per I/O channel we want to be able to include VLAN configuration, potentially outbound rate limits, and filters for incoming multicast traffic.

5.2.2.2. Event Notification Mechanism

Every I/O channel needs an event delivery mechanism which enables fast and efficient event notification between the guest OS and the virtual network I/O device. This is required because event notification delivery itself is a feature used on the data path. For example, when processing network traffic, there are two significant events affecting the data path:

- New data has arrived from the network and needs processing from the driver

- Outgoing data has been processed and sent off onto the network by the virtual device, and the associated memory buffers can be re-used for new data

For both of these events, the virtual device needs to be able to send a notification to the driver, so that the driver can react and start processing.

The event notification mechanism is set up on initialization of the VDP and data path events are associated with one or more packet queues. If events are shared between multiple queues, then the driver needs to have a way to find out which of the queues is affected. It can either walk through all queues and check, or the virtual device can indicate to the driver in some out-of-bound mechanism which queue the event is for. For example, it can advertise that through a register in the virtual configuration space.

As mentioned in the previous section, ideally, we would like to have two events per I/O channel, for example, one for the send queue and one for the receive queue. Having independent events for this makes it easier to optimize the associated operation. For example, we can independently adjust the frequency of event generation. Most of today's I/O devices support a sufficiently large number of interrupts to make this design feasible. For example, the PCI Message Signaled Interrupts Extended (MSI-X) technology allows a device to support up to 2048 independent interrupts. MSI-X is supported by the majority of PCI-based network devices. Hardware support for multiple interrupts per I/O channel is only required when the interrupts are directly routed through to the virtual machine without any interception from the network control domain. If the real device does not support enough interrupts, then the NAE architecture makes it possible to multiplex a limited number of shared, hardware-based interrupts in software in the network control domain, and then inject virtual interrupts associated with a particular event into the virtual machine. This is not as efficient as using hardware-based interrupts directly routed through to the virtual machine's virtual CPU, but it is a reasonable workaround for a situation where the real device does not support enough interrupts.

Events are not only used on the virtual data path though. The privileged control path can also send event notifications to the virtual control path to indicate link errors, hardware failures, firmware errors or other unexpected events that need to be reported to the user virtual machine. For this, the NAE infrastructure includes further event types that are not used as part of data path I/O processing. In that case, the VCP typically reserves an additional interrupt which is controlled and processed by the VCP instead of the VDP. A single interrupt can be used to indicate different events, and on the VCP event processing is not necessarily performance-critical as these events should only happen very infrequently and will not interfere with high-performance data transfer operations.

5.2.2.3. Hardware-Based Virtual Control Path

The configuration space implementing the virtual control path is part of the NAE architecture and is standardized across all virtual network I/O devices. Hence it is hardware-independent and vendor-independent. The virtual device's configuration space is a simple and space-limited I/O memory region. Presenting the VCP in this way means that it can be easily implemented in hardware by vendors who would like to natively support the NAE API. In that case VCP information flow from a virtual machine would not go to the network control domain, but instead directly to the virtualization-aware, NAE-compatible network I/O device.

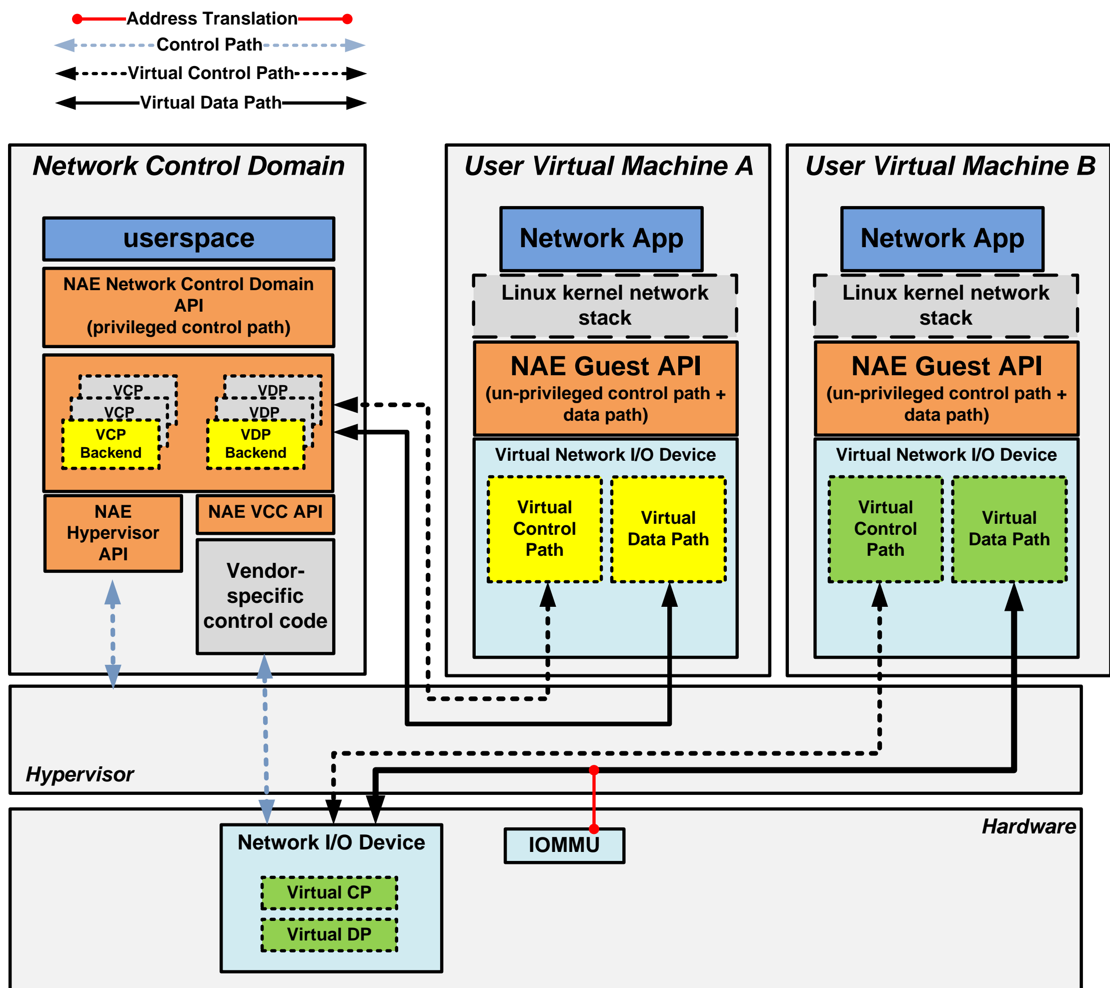


Figure 7 Hardware-based Virtual Control Path

For example, user virtual machine B in Figure 7 connects to a network I/O device that supports a hardware-accelerated VCP. For that virtual machine there is no NAE device emulation required, and instead the hardware can handle requests from the VCP of the virtual machine directly. In all cases there still needs to be vendor-specific control code in the network control domain, for example, for setting up device resources and initializing and tearing down the VCPs and VDPs.

The virtual control path is used to configure various device operations. When the underlying real device is shared across multiple user virtual machines, then the privileged control path needs to ensure that multiple virtual control paths connecting to the same device are sufficiently isolated. Some control path operations, like, for example, configuring VLAN tagging, multicast filters or bandwidth limits, might need involvement of the privileged

control path. But rather than invoking the software-based privileged control path for each operation, it should ideally be possible to allocate certain hardware-resources, for example, a certain bandwidth, multicast filters or VLAN groups, to each hardware-based VCP on VCP initialization. This is done through the privileged control path. From then on, the hardware-based VCP is in control of its allocated resources and they can be accessed and configured directly from the user virtual machine owning the particular VCP without involving the privileged control path. Other control path operations, like setting DMA addresses of circular buffers, do not necessarily need privileged control path intervention and therefore can always be easily routed directly to the real device. In order to reduce complexity of the hardware-based virtual control path, the NAE architecture allows implementing only a subset of the virtual control path in hardware. For example, a virtual control path can consist of directly mapped, hardware-based configuration space registers and emulated, software-based configuration space registers. Implementing configuration registers in hardware helps accelerating control path operations as there is a more direct path to the real device needing configuration, and furthermore it helps reducing CPU load in the network control domain, because less processing has to be done in software.

5.2.3. Virtual Data Path

5.2.3.1. Initialization and Configuration

The virtual data path is implemented using one or more I/O data channels. Those data channels can be hardware-based or software-based. A data channel consists of a receive queue and a send queue, for example, a data channel is always bidirectional. A virtual data path can have multiple I/O channels, and different I/O channels can have varying capabilities. All channels are initialized and controlled by the virtual device's virtual control path. Typically, the VDP is initialized when the virtual device is activated, so that data can flow as soon as possible. However, it can potentially be re-configured at runtime. By default a virtual

I/O device only has a single I/O data channel. If it has multiple data channels, then it needs to support mechanisms to multiplex network traffic across multiple channels. This would need to be supported by both the real device and the driver running in the user virtual machine.

I/O data channels might be added or removed dynamically at runtime as well. When the user virtual machine requests a certain number of channels or capabilities of the virtual data path, the request is validated by the privileged control path code running in the network control domain which then eventually allocates resources for the VDP either in software or in hardware. Resources for the VDP are strictly controlled and limited by the network control domain in order to enforce performance isolation between virtual devices belonging to different virtual machines.

A data channel is characterized by a set of feature bits which indicate to the driver in the user virtual machine what capabilities it has. The feature bits can be read through the virtual control path as explained in the previous section. A feature is only enabled if both the driver and the I/O channel support it. Features of the I/O channel characterize how it can move data to and from the virtual machine. For example, the feature bitmask advertises whether or not the underlying network device (either a hardware-based real device or a software-based emulated device) can do TCP processing offloading. If it can, then the network stack running inside the virtual machine does not need to do it and therefore the virtual machine's compute resources can be used for doing other processing. Hardware-based network protocol offloading (like, for example, TCP segmentation offloading or UDP fragmentation offloading) can significantly improve network performance and reduce load on the system's CPU.

The send queues and receive queues need to be designed to enable fast data transfer operations between a hardware-based real device and the virtual machine. The direct memory access (DMA) capabilities of today's computer systems allow I/O devices to access

system memory without involving the main processor. In this way, DMA operations enable fast I/O without putting significant load on the CPU. In the NAE I/O architecture we develop send and receive queues as DMA-capable data queues. Instead of putting data buffers directly into a continuous queue, we use so-called *data descriptor queues*. A data descriptor describes a data buffer's properties and where it is located in system memory. All data descriptors are stored in a virtually contiguous, circular buffer which we then call the send queue or receive queue. That means, the send and receive queues do not actually contain real data, but just information about where to find the real data. This mechanism is commonly used in hardware-based I/O channel design and it has the advantage that the actual data does not need to be stored anywhere in a contiguous memory region, but instead it can be spread around anywhere in system memory. For the design of the NAE I/O infrastructure, using this technology has further advantages in that it is used in most, if not all, network hardware, and data descriptor formats can easily be unified across different devices.

Circular buffers, also called *rings*, are used in various I/O channel implementations. Not just for I/O channels between hardware and software, but they are, for example, also used between virtual machines. In particular, software-based para-virtualized network interfaces, like virtio, use virtually contiguous circular buffers to transfer data between the network control domain and the user virtual machine. In this case, the circular buffer resides in a shared memory region that can be accessed from both sides.

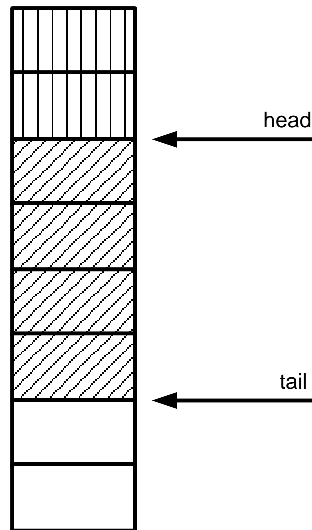


Figure 8 Concept of Circular Buffer

Figure 8 shows the basic idea of a circular buffer, or ring. One side of the I/O channel maintains the head pointer, and the other side maintains the tail pointer. In our NAE I/O infrastructure, the head pointer is maintained by hardware (or emulated hardware) while the tail pointer is maintained by the driver residing inside the virtual machine. The driver fills the ring with empty data descriptors that are ready to be used by hardware. It uses the tail pointer to indicate which descriptors it has prepared and can be taken into hardware ownership. Hardware on the other hand processes all descriptors that software has prepared, and it advances the head pointer to indicate to software which descriptors in the ring have already been successfully processed and can therefore be taken back into software ownership. Both pointers run through to the end of the ring, and then start back again at position zero. This is where the name circular buffer comes from. Despite just using the head pointer, hardware can also mark descriptors explicitly to indicate that they have been processed and can be re-used.

One of the key advantages of the NAE-compatible VDP is that its interface to the virtual machine is independently of whether or not there is a real device connected to the I/O data channels. It is hidden to the virtual machine whether it runs on a hardware-based virtual data path or a software-based, emulated virtual data path. Providing a unified VDP like this

to the virtual machine is crucial in enabling features like virtual machine migration and dynamic load balancing that should be key capabilities of cloud infrastructures.

5.2.3.2. Software-based Virtual Data Path

Even though a hardware-based virtual data path usually provides the best I/O performance, there might be reasons for not having the virtual data path directly connected to hardware, but instead use a virtual data path that is backed up by a software-based VDP running in the network control domain. Such a network I/O path is demonstrated with the configuration of user virtual machine A shown in Figure 7. A setup like this could be used if, for example, some operation needs to be carried out on the data that cannot be done in hardware on a particular platform where the virtual machine runs on. Another reason for such a configuration could be that the platform does not actually have any hardware accelerators installed, so obviously there are no hardware capabilities that we can take advantage of and there is no hardware present that could implement a VDP. In a dynamic, large-scale environment this could easily happen when virtual machines migrate between different platforms with heterogeneous hardware configurations. For these use cases, this architecture allows implementing a software-based virtual data path. This emulated VDP runs in the network control domain and is configurable by the virtual control path from within the user virtual machine. In this case, because there is no real hardware backing up the network processing on the virtual data path, there is no vendor-specific control code required at all. Instead we provide generic control code that allows setting up the software-based virtual data path from user space in the network control domain.

This path typically lowers performance of the system when processing network I/O from the user virtual machine, because purely software-based emulation of the VDP can be CPU-intensive and can be stressing resources in the network domain. Therefore a configuration like the one which user virtual machine A demonstrates in Figure 1 needs to be thought

through carefully and only chosen if a hardware-based virtual data path is not feasible or unpractical.

In the NAE I/O architecture the user virtual machine has *unified access to the VDP*, independently of whether or not the VDP is backed up by hardware resources, or completely emulated in software. This is one of the most significant advantages of the proposed design compared to existing approaches where different drivers have to be plugged in and synchronized if the underlying data path were to change. In fact, none of today's existing approaches enable smooth switching between a hardware-based data path and a software-based data path. On a NAE-compliant virtualized system, we can have a set of hardware-based VDPs and a set of software-based VDPs, and we can enable switching between those depending on, for example, application needs, available system resources, fair sharing policies, or other QoS requirements. One use case for this would be a cloud infrastructure where a platform can support 64 hardware-based virtual data paths, but it runs over 100 user virtual machines. Some of those user virtual machines might not need a high-performance network connection at all while others need it permanently. The cloud infrastructure provider could charge hardware-based VDPs differently to software-based VDPs: hardware-based VDPs could be priced higher than software-based VDPs as their performance is higher and their availability is typically limited. With a NAE-compatible virtualized system a cloud infrastructure provider could maximise resource utilization by allowing flexible resource allocation policies. VDPs can be associated with a virtual machine for a lifetime, or they can be more dynamic, depending on the availability of hardware resources and application needs. If a virtual machine needs a hardware-based VDP, because it requires high-performance network connectivity, then it will get a dedicated hardware-based VDP. However, if the virtual machine does not necessarily require a fast virtual data path, then the cloud infrastructure provider could offer a "best-effort" VDP instantiation. For example, if there are unused hardware-based VDPs on the platform where the virtual

machine is hosted, then one of those unused VDPs is assigned to the VM. However, if another VM requiring a high-performance network path is migrated to the platform, then the hardware-based VDP will be de-assigned from the original virtual machine, and then dedicated to the newly migrated virtual machine instead. Hardware-based VDPs can be allocated based on any agreed fair sharing policy, or a priority scheme. The original virtual machine will potentially experience a drop in network I/O performance when the hardware-based VDP is removed, but it will be able to keep all existing network connections alive as a software-based VDP will be smoothly attached to the virtual machine at the same time. The transition between hardware-based VDP and software-based VDP is transparent to the guest OS and any applications running on top of it, because the interface to the VDP remains exactly the same.

The architecture also allows running a hardware-accelerated VDP in parallel to a software-based VDP. In that case the VDP implements some logic that decides what data is passed down which part of the virtual data path. This logic is programmed through the NAE virtual control path. More details on this are explained in the following section.

5.2.3.3. Multi-Channel I/O

Being able to provide multiple simultaneous I/O data channels on a network device is crucial for providing high-performance network I/O. Having just a single queue in one direction means that processing of network connections can only happen sequentially. On top of this, as computer systems evolve towards platforms with more and more CPU cores that allow parallel processing, it is important to support this design trend also for the system's I/O infrastructure. Most modern network devices support multiple send and receive queues. When a network card supports multiple receive queues, packet processing can be easily spread across multiple CPU cores, each of them serving a receive queue. This mechanism is generally known as receive-side scaling (RSS). Various techniques and algorithms exist to

spread network traffic equally across receive queues. The main goal of receive-side scaling is to improve network I/O performance.

The main advantage of multiple send queues is that, for example, traffic can be split into different isolated traffic "classes". Network cards in the wireless area commonly support multiple transmit queues and server-based wired cards also follow this trend. In this context, video and voice traffic can be separated from, for example, Internet data traffic or other low priority background traffic. Each of these different traffic classes uses its own send queue meaning that network packets from different classes do not affect each other. Such a design is important to ensure fast packet delivery for high priority traffic classes like voice or video data.

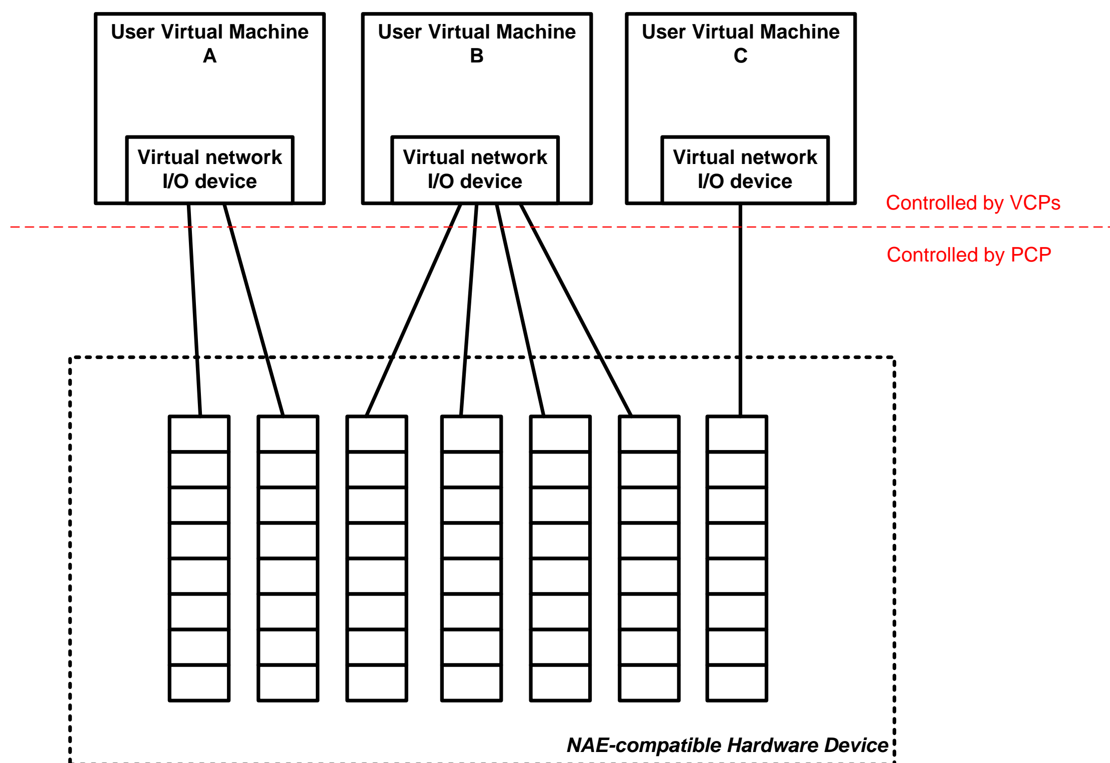


Figure 9 Arrangement of I/O Channels

Figure 9 show a system with a real network device which supports seven I/O channels. The system runs three user virtual machines. User virtual machine A has a virtual network device assigned which supports two simultaneous hardware-based I/O channels, user virtual machine B's virtual network device supports four simultaneous hardware-based I/O

channels and user virtual machine C is configured with a virtual device using just a single hardware-based I/O channel. The NAE architecture allows resources of the underlying real device to be used by the virtual machine in a flexible manner, so that a single real device can be used by multiple virtual machines at the same time. As you can see in this example, the number of I/O channels the virtual device supports, are independent of the number of I/O channels the real device supports.

When introducing a multi-channel I/O architecture into the system, complexity is added because for every packet on the virtual data path there needs to be a mechanism to decide which channel to use. When receiving packets from the network, this mechanism needs to sit inside the real device in order to multiplex packets onto different I/O channels which are (potentially) assigned to different virtual machines. Most of today's devices can multiplex packets by destination Ethernet address when I/O channels are associated with different virtual devices which have different Ethernet addresses. When multiple data channels are assigned to a single virtual device, like for example for user virtual machines A and B, then the goal of such a configuration is typically to take advantage of load balancing. Network connections to the same virtual machine can be spread out to multiple I/O channels which can be processed simultaneously by multiple CPU cores. That way, overall throughput can be significantly increased. Network hardware supports load balancing across multiple I/O channels through technologies like Receive-Side Scaling (RSS) where a single connection is spread across multiple receive queues. For example, Mellanox's ConnectX network controller looks at IP, TCP and UDP headers and takes those as input into a hash function which returns the index of the receive queue to place the packet on. That way, slightly varying packet headers will assign packets destined to the same virtual network device to different receive queues. The privileged control path (PCP) is in control of hardware-based receive queues and their configurations.

On the transmit path there needs to be a mechanism in the virtual machine to decide which send queue a packet has to be placed on. There are various strategies for spreading packets across multiple send queues and current operating systems have support for this in the network stack. For example, Linux has a so-called *traffic control* infrastructure allowing the scheduling of network connections across multiple send queues. Scheduling policies can be configured by the administrator of the system. Typically connections are identified by packet header information, for example, IP addresses, TCP ports and UDP ports. That way, different connections can be routed out onto the network through different send queues. The most important use cases for this are potentially

- the enforcement of rate limits on certain network traffic, for example, some send queues are processed faster or more frequently than others,
- traffic prioritization – for example, high priority traffic goes on a faster processing send queue, and potentially packets on that queue can further be marked with traffic prioritization tags like IEEE 802.1p Class of Service (CoS) bits, and
- the implementation of fair sharing policies, for example, all send queues might be processed in a round robin fashion and therefore all connections get a fair share of the overall bandwidth.

As send queues in the NAE architecture are ideally implemented in hardware, as shown in Figure 9, the real device itself needs some mechanism to schedule the sending of packets from all its configured send queues. While scheduling policies across multiple I/O channels of a virtual device is controlled by the virtual control path inside the virtual machine, scheduling across I/O channels of the real device is controlled by the privileged control path. This is important because typically the I/O channels of a real device are shared between multiple virtual machines. The hardware itself might support various scheduling schemes. For example, Mellanox's ConnectX supports a strict priority scheme, a simple round robin

scheduler, a weighted round robin scheduler and a weighted fair queuing (WFQ) scheduling algorithm. These policies are configured through the privileged control path.

Through the NAE I/O architecture it is by design possible to provide a virtual device having a mixture of hardware-based and software-based I/O channels. Figure 10 shows an example of a system with user virtual machine that is configured with a virtual network device that supports two simultaneous I/O data channels.

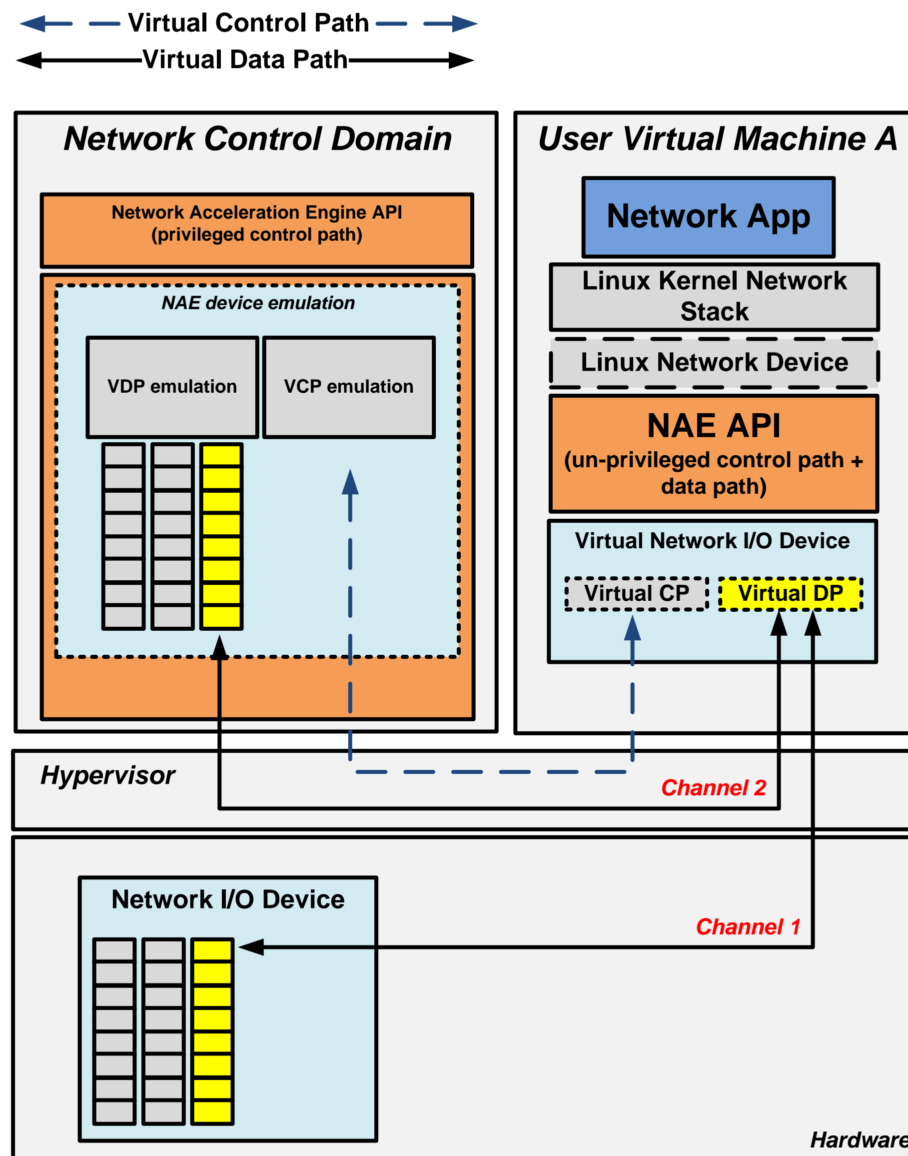


Figure 10 Multi-channel I/O System Architecture

Channel 1 is a hardware-based I/O channel while channel 2 is a software-based I/O channel. The channel resources are marked in yellow. This makes it visible that the real network I/O

device installed on the system has a number of I/O channels, but only one of them is used by user virtual machine A. In the network control domain, the NAE device emulation layer emulates resources that are not provided in hardware. In the figure, that particular network control domain supports three software-based I/O channels, and one of them (the yellow one) is assigned to user virtual machine A. Therefore, the VDP of the virtual machine here uses a combination of hardware-based and software-based I/O channels. More details on the design of the software-based VDP, and its use cases, were given in the previous section.

As with a purely software-based or purely hardware-based multi-channel device, there has to be a mechanism to multiplex packets across multiple queues. One of the most significant differences of a mixed (software-based and hardware-based) I/O channel design is that the location of the VDP backend is different depending on whether the channel is backed up by software emulation or by hardware: it is either located in the network control domain, or on the real device. This does not matter for network operation or compatibility (a software-based VDP looks just as a hardware-based VDP to the user virtual machine), but it matters for the logic of packet routing through the system. Potentially, the real network device has its own packet switch function which allows forwarding of packets between all of its I/O data channels. That packet switch function is programmed through the privileged control path from the network control domain. Typically, the network control domain would also run a packet switching function to forward network traffic between all software-based I/O channels. Traditionally, this software-based packet switching function is called a “virtual network switch”, and most of today’s virtualization solutions incorporate such a capability. Examples of virtual network switches are OpenVSwitch, VMWare’s vNetwork Distributed Switch, IBM’s Distributed Virtual Switch 5000V, and Cisco’s Nexus 1000V.

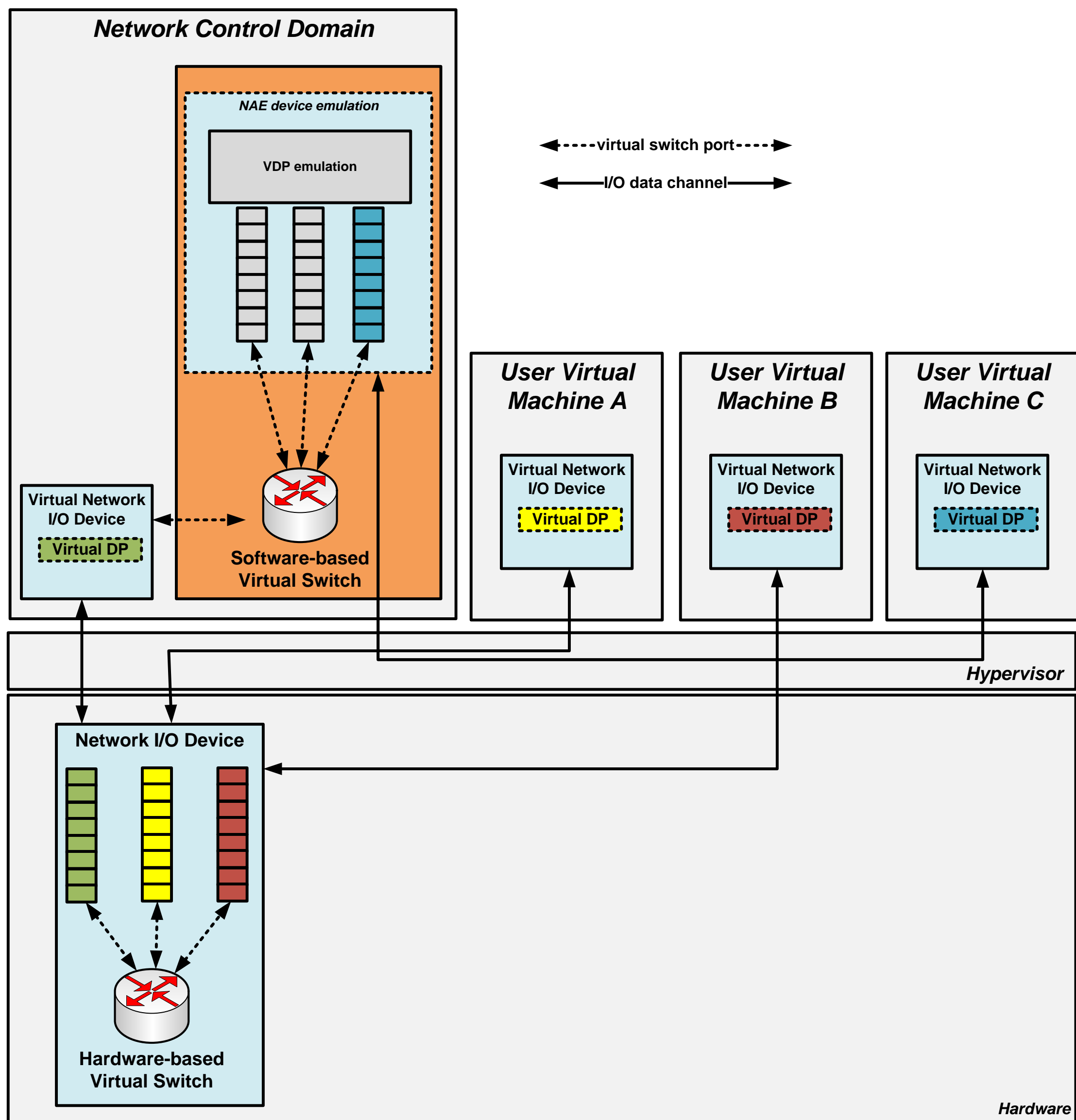


Figure 11 System Architecture including Virtual Network Switch

Figure 11 shows a simplified view of a platform with a hardware-based network I/O device which is able to forward network traffic between its I/O data channels, and a network control domain which runs a virtual network switch to forward packets between its emulated I/O channels. The network I/O device connects the whole system to the physical network infrastructure. In this example, one I/O channel (the green one) is assigned to the network control domain (in this case, it is merely just another virtual machine), so that the network control domain has connectivity to the external world just like user virtual machine A and user virtual machine B which are configured with hardware-based I/O channels. The network control domain then connects this virtual data path to its own, software-based

virtual network switch as one of the virtual network ports. This virtual network switch multiplexes packets between that virtual network port and all software-based I/O channels. For example, user virtual machine C is configured with a software-based I/O channel (marked in blue in the figure) and its VDP backend also plugs into the virtual switch in the network control domain. In this configuration, if user virtual machine B wanted to communicate with user virtual machine C, traffic would have to flow through the hardware-based virtual switch on the network I/O device and then also through the software-based virtual switch in the network control domain. This is not ideal, as the I/O path involves many components processing data. However, as resources in hardware are always limited, it is important to be able to provide a software-based virtual data path and to be able to connect it to hardware-based VDPs on the same system.

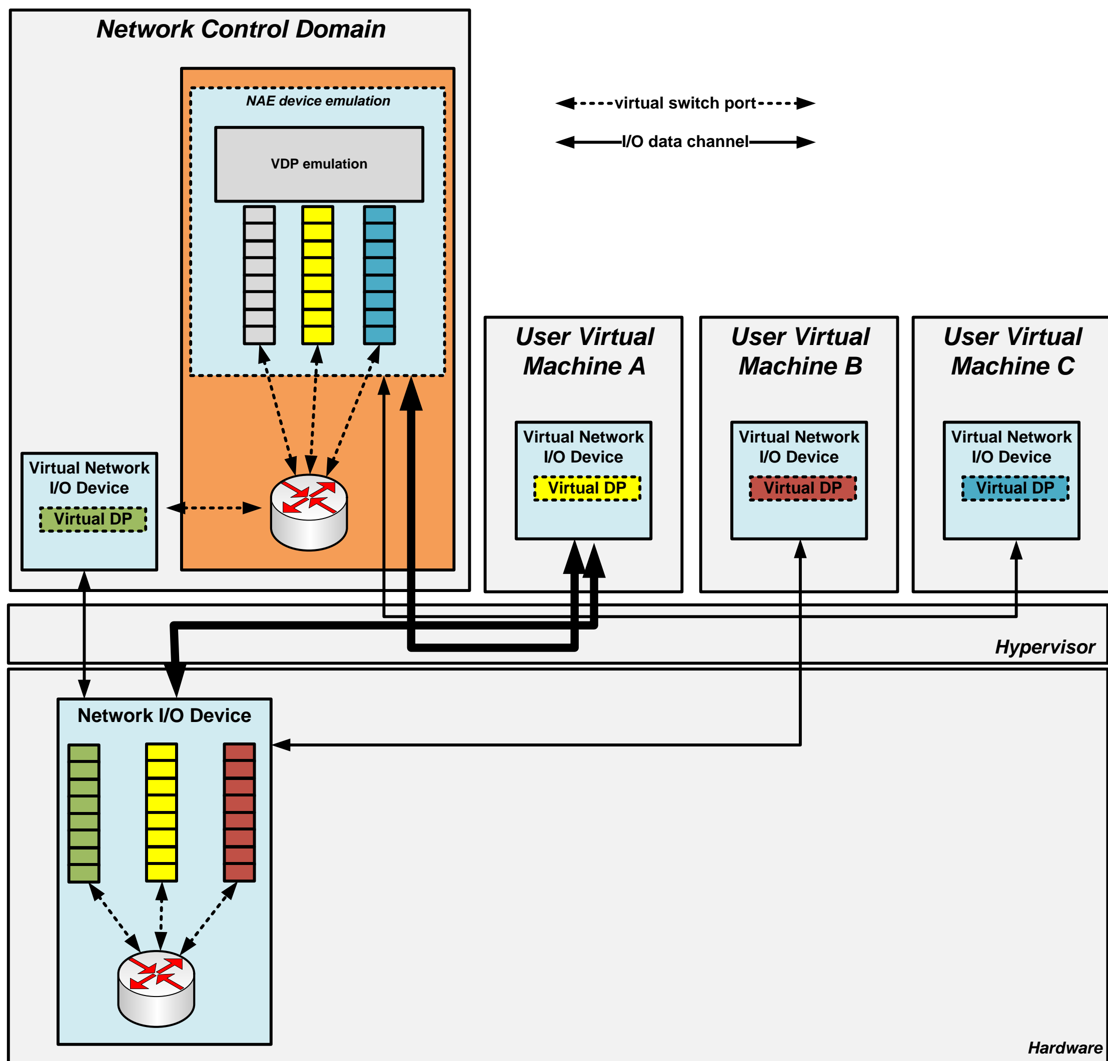


Figure 12 Multi-channel I/O using SW-based and HW-based VDP

Figure 12 shows a slightly modified system configuration as user virtual machine A has now a virtual device assigned which uses a virtual data path that consists of a software-based I/O channel and a hardware-based I/O channel. In this configuration, if user virtual machine A wants to communicate with user virtual machine B, they can send data using their software-based I/O channels and packets would be multiplexed by the software-based virtual switch in the network control domain without having to traverse the hardware-based virtual switch on the network I/O device. A purely software-based I/O path like this connecting two co-located virtual machines can be implemented feasibly to provide a fast and efficient network connection.

One of the biggest challenges in such a multi-channel configuration is to design the logic for routing packets through the system. For example, if a user virtual machine wants to open a connection to another endpoint, then the guest OS network stack needs to decide on which I/O channel that particular endpoint can be reached. Multiplexing between I/O channels can be programmed through the virtual control path. Locations of endpoints can be learnt through traditional discovery on all I/O channels, but they can also be configured through the privileged control path. In that case, the privileged control path (PCP) might have the knowledge of endpoint locations through various mechanisms. For example, either because it runs discovery protocols itself, or maybe because endpoint locations have been programmed in through the management interface. The PCP can then pass location information to all the connected virtual control paths on the system which can then program their forwarding tables accordingly. The PCP also programs the hardware-based virtual switch and its forwarding tables. While the NAE architecture allows a flexible design like this, various issues might arise. In particular, the PCP might not want to expose information about whether or not a VDP is software-based or hardware-based, and more importantly it might not want to expose whether or not the endpoint of any connection resides on the same platform or not. Cloud infrastructure providers would be very hesitant about any solution revealing details on the underlying infrastructure and specifically the placement of virtual machines. The NAE architecture supports this requirement. Through the NAE I/O infrastructure, the VDP looks exactly the same to the user virtual machine independently of whether it is backed up by a software-based emulation layer or by a real device. This is the case for each individual I/O channel, too. Therefore distributing information about on which channel a particular endpoint can be reached does not give away any sensitive, infrastructure-specific information.

5.3. NAE Network Control Domain API

5.3.1. Privileged Control Path

The privileged control path (PCP) sits partly in kernel space and partly in user space of the network control domain. The network acceleration engine API does not directly interface with the hardware device. It is assumed that there always needs to be device-specific code controlling a particular hardware accelerator, and so that control code is vendor-specific. The vendor-specific control code (VCC) sits underneath of the NAE API and is therefore invisible to high-level management software stacks. The NAE API defines the privileged control path interface which is vendor-independent and controls a wide range of network I/O devices. Vendor-specific control code is plugged underneath this API and implements device-specific functions that shall be exposed through NAE.

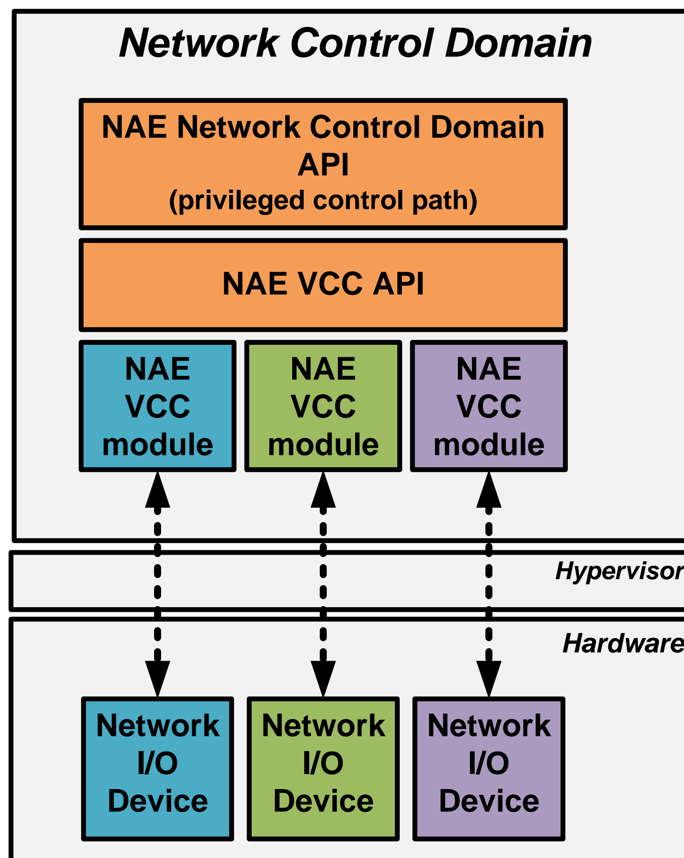


Figure 13 Overview of NAE VCC API

Figure 13 visualizes the NAE VCC API and individual VCC modules that control different devices. Here we have three VCC modules and three different network I/O devices that are used as network accelerators. Vendor-specific control code is required, because it would be too expensive to unify the privileged control path between different network I/O devices as their hardware capabilities and their configuration logic implemented in silicon vary significantly. Mostly device firmware is quite simple, and intelligence is put into software as complex device drivers. Instead of taking that code from device drivers into the network acceleration engine API, it is best to leave vendor-specific control code underneath a generalized API. It is important to make the virtualized device interface in the user virtual machine vendor-independent, but it is acceptable to keep vendor-specific control code in the network control domain as this part does not need to be migrated across different hardware platforms.

5.3.2. Vendor-specific Control Code

Vendor-specific control code (VCC) resides in kernel space of the network control domain. Each VCC is developed as a loadable kernel module, just as today's device drivers are implemented. The NAE API provides a set of virtual function tables consisting of various function stubs defining what capability sets can be implemented by VCC modules. Some of those functions are mandatory. For example, functions to start and stop a real device. Other functions might be optional. For example, functions to control specific device features. If a real device does not implement a particular feature, then those function calls can stay unassigned. The NAE VCC API resides completely in kernel space. Mostly, the VCC API implements control path functions which are not necessarily performance-critical. However, the VCC module might also be involved in interrupt processing, and in that case it is important that there is a fast and efficient path between the VCC module and the hypervisor. The VCC API defines functions for a VCC module to register for real device interrupts being routed to itself rather than directly to the virtual machine. Furthermore it provides functions

to inject an interrupt into a virtual machine. Typically, when an interrupt is generated by the real device, and then routed to the VCC module, then the VCC module can take some action (for example, reading or writing a register to acknowledge the interrupt or update some statistics), and then inject a virtual interrupt into the virtual machine.

The VCC API furthermore communicates information about real device registers to the virtual machine via the virtual control path. For example, this is required when certain registers need to be made directly accessible from the virtual machine. This is the case for some registers that need to be accessed as part of a data path operation. Exposing registers like this through a common API across all devices makes access to hardware from the virtual machine device-independent and vendor-independent.

Link status information also stays in control of the VCC module. The virtual machine cannot directly check the physical state of the attached link, and instead, the VCC API defines a function that allows the virtual machine to check the link state via the virtual control path. Different vendors implement link status checking in different ways. For example, some devices generate a notification to the system when the link state changes, while others need to be continuously, explicitly polled. With the VCC API, details on how the link state is checked is hidden in the VCC module, and a generalized, vendor-independent function of the VCC API allows status information to be sent to the virtual machine.

5.4. NAE Hypervisor API

The NAE I/O infrastructure requires that the hypervisor exposes an API to control interrupt mappings, DMA address translation mappings (the I/O MMU interface) and memory-mapped I/O bindings for a user virtual machine. These three control interfaces are crucial for building a virtual I/O path with good performance.

The hypervisor API needs to provide a mechanism allowing other components in the virtualized system to raise an interrupt in a user virtual machine. That way, components of

the Privileged Control path, like, for example, the VCC module, can signal an interrupt to the virtual machine. This is, for example, required when the VCC module has detected a hardware event that is not directly routed to the virtual device. Then the VCC module needs a way of notifying the virtual machine about the event. In traditional virtualized systems, only the hypervisor has direct access to interrupt routing functions. It provides capabilities like the emulation of virtual interrupt controllers, or it directly programs hardware-based virtual interrupt controllers, if available on the platform. With the NAE API the VCC module can also indicate to the hypervisor that it wants to keep processing certain interrupts from the real device itself, while others might be directly routed to the user virtual machine. If an interrupt is routed directly to the user virtual machine, then the hypervisor simply programs the user virtual machine's interrupt controller accordingly. If an interrupt is routed to the VCC module, the VCC module registers with the system as the owner of the real device interrupt, but then it asks the hypervisor for the allocation of a virtual interrupt assigned to the associated virtual device residing in the user virtual machine. It can then handle the real device interrupt itself, and inject a virtual interrupt into the user virtual machine, if desired. In this design, virtual interrupts and real interrupts are completely decoupled which provides a more flexible I/O path satisfying a variety of use cases.

The NAE hypervisor API allows other components on the privileged control path to access the memory management unit interface in order to program memory mappings for user virtual machines and in particular their virtual devices which make use of hardware-based network acceleration engines. The VCC module advertises certain registers that have to be mapped into virtual device registers, so that they can be accessed directly by the user virtual machine. The hypervisor API allows building of a complete virtual I/O memory space: parts of it can be mapped directly to real device I/O memory, and parts of it can be emulated in software. The hypervisor itself implements a virtual memory management unit for the user virtual machine, or uses the system's hardware-based, virtualization-aware memory

management unit, if available. Most of today's platforms have a hardware-based, virtualization-aware MMU. For example, Intel's Extended Page Table (EPT) capability described in (Intel Virtualization Technology (Intel VT) n.d.) and AMD's Rapid Virtualization Indexing (RVI) described in (AMD Virtualization n.d.) provide a second stage of address translation in the memory management unit speeding up memory accesses from virtual machines. Newer ARM platforms supporting the ARM v7-A architecture with the virtualization extensions, as available on today's Cortex-A15 and Cortex-A7 processors, also include a virtualization-aware MMU. (Goodacre 2010) and (Mijat and Nightingale 2011) describe more details on ARM's hardware-based acceleration for hypervisors.

The hypervisor is responsible for controlling direct memory access (DMA) policies on the virtualized system. DMA is used by I/O devices to transfer data directly to and from system memory without involving the main CPU. DMA is typically controlled by the operating system, but on a virtualized system, DMA shall also be enabled for user virtual machines, and it might be desired that a single I/O device is shared between multiple user virtual machines. In that case, a single I/O device would directly access memory regions (read and write) of different user virtual machines. When direct I/O access like this was originally developed it was considered as an insecure way of accelerating I/O, because initially there was no way to ensure that a malicious I/O device could not write memory it was not allowed to access. In such a configuration, a buggy or fraudulent I/O device could easily crash the whole system, the network control domain or other user virtual machines. Various technologies have then been developed to enable secure, direct I/O access for virtual machines. As described by Willmann (Willmann, Rixner and Cox 2008) (Willmann, Rixner and Cox 2008), hardware-based, virtualization-aware system I/O MMUs and purely software-based methods can both provide sufficient control of I/O accesses to isolate user virtual machines from each other. The NAE API builds on any of these technologies, depending on what the hypervisor can provide. Hardware-based, virtualization-aware system I/O MMUs

are present in most of today's platforms, for example, Intel's VT-d (Abramson, et al. 2006) and AMD's I/O MMU (Advanced Micro Devices, Inc. 2009), have been on the market since 2006. A virtualization-aware system I/O MMU allows the creation of multiple DMA protection domains having a subset of physical memory assigned. Then one or more I/O devices can be associated with a protection domain. Address translation tables are then used to restrict access to protection domain's physical memory from I/O devices which are not associated with that particular protection domain. This hardware-based mechanism enforces DMA isolation between protection domains. On a virtualized system, protection domains can represent the memory of a user virtual machine, the network control domain and also the hypervisor itself. Through the NAE hypervisor API a protection domain can be associated with a user virtual machine and I/O devices used as network acceleration engines by that user virtual machine can be given controlled access to that protection domain. Apart from protecting the virtualized system from buggy or malicious I/O devices, the I/O MMU is also used for address translation. In particular, the virtualization-aware I/O MMU can translate virtualized memory addresses used by the user virtual machine (often called guest physical addresses, or GPAs) to the system's physical memory addresses (often called host physical addresses, or HPAs) as used by the real I/O device. This is an important hardware feature of the platform enabling DMA transfers between the virtual machine and the I/O device without modifying the virtual machine's operating system. The NAE hypervisor API exposes this capability on the privileged control path.

The hypervisor is responsible for another important feature of a virtualization-aware I/O MMU: interrupt virtualization. For example, on an Intel system without interrupt virtualization, PCI Message Signalled Interrupts (MSIs) are issued as DMA writes to system memory. The interrupt attributes, such as vector and destination processor, are encoded in the target address and the data of the DMA write operation. This mechanism violates DMA isolation properties between protection domains of the system. With hardware-based

interrupt virtualization, as for example introduced by Intel's VT-d (Intel 2011), the DMA write request only contains a message identifier which can be mapped to a unique entry in a table defining all interrupt attributes for a particular I/O device. The DMA request contains a device identifier (also called a requester ID) of the I/O device generating the interrupt, and that is used by hardware for remapping message identifiers as described above.

5.5. NAE Hardware API

Today's I/O devices present a very low-level interface to a platform. For example, many I/O devices standardize at the PCI level, and then they need a significant amount of device-specific control code running on the platform in order to present a more suitable I/O interface to the operating system. In particular, for network I/O, the device-specific control code is used to create more generic abstractions of send and receive queues which can then be used by the operating system to send or receive network traffic. Hence, with today's hardware APIs, there is practically no common interface between different I/O devices, even if they all do mainly the same thing: send and receive network packets. The standardized API is provided completely by software and resides inside the operating system. Such a platform design limits portability, and this is the major restriction we want to overcome with the proposal of the NAE hardware API: we want to design a common, potentially minimal, hardware API that all network I/O devices support. However, the NAE framework does not limit differentiation of each device and its capabilities from competitor's devices - one of its major goals is to allow exposing most, if not all, device features to the virtual machine.

One of the key design decisions of the NAE framework is to split control path resources from data path resources. This is most critical also for the hardware API design. An I/O device might have a set of resources that can be shared across multiple virtual machines. In that case, the design needs to enable the clear partition and isolation of hardware resources, and restrict access to each partition individually. Typically a system can enforce resource

isolation like this using a memory management unit (for I/O memory access) and an I/O memory management unit (for DMA access). Therefore an I/O device needs to be designed so that its resources can be partitioned at the granularity matching the capabilities of today's memory management units. The system's MMUs are controlled by the hypervisor, as explained in the previous section.

For a network device, one of the most critical resources that needs to be clearly partitioned is an I/O data channel. The NAE hardware API defines a hardware-based I/O channel as a bidirectional channel consisting of a send and a receive packet queue. In the NAE architecture, packet queues are implemented as circular buffers (or also called ring buffers) containing data descriptors as explain in the section describing the Virtual Data Path. These can very easily be implemented in hardware. The circular buffer itself is held in system memory. The device has to keep track of the following critical values:

- the system memory address of where the circular buffer is stored,
- the packet queue tail pointer and
- the packet queue head pointer.

The device reads data descriptors from the specified system memory address to find out the location of the actual data buffers that need to be transferred to or from the device. The format of data descriptors is defined by the NAE hardware API, and all NAE-compatible I/O devices need to support that descriptor format. A simple data descriptor used for sending data contains the following fields:

Table 2 Simple TX Data Descriptor

Name	Bits	Description
header_addr	64	DMA address for packet header
data_addr	64	DMA address for packet data
command	16	Command code defining hardware operation

length	16	Length of the data buffer
status	32	Status code indicating success/failure

The send data descriptor splits a packet into a packet header part and a packet data part. Packet header and data do not necessarily need to be split, and not all hardware might be able to provide a packet splitting function. In that case, the header address can be set to zero and hardware just uses the data address. Splitting the packet header and data can enable a powerful I/O architecture where packet header and packet data can use different I/O paths in the system. This can, in particular, be useful for virtualized systems where packet headers could be passed to an intermediate processing component in the network control domain which makes a decision about how the data shall be forwarded. The data itself can then be transferred directly to the final destination (for example, the user virtual machine) without having to traverse the intermediate processing component in the network control domain. This design allows the use of packet headers in forwarding decisions or for enforcing firewalling rules in the network control domain without putting much strain on the system by also transferring the whole data part to the intermediate processing component. That way, we can implement complex intermediate processing components on the data path while still providing good performance as data parts are moved directly to the endpoint.

The send data descriptor contains a *command* field which is used to indicate to the device what function should be used for the associated data buffer. This can be used, for example, to instruct the device to do a checksum calculation of the data and place that in the packet header. It could also be used for having the device add a VLAN tag to the packet. Further command bits can be used to indicate to the device that it should process TCP segmentation or UDP fragmentation. TCP and UDP offloading are very common hardware functions to accelerate network I/O.

The *length* field in the send data descriptor indicates the size of the actual data. It might be that some devices only support a fixed data buffer size. In that case this field can be hard-coded by the driver to a specific value. Hardware reports back whether or not transmission was successful by writing a status code into the *status* field of the descriptor.

The simple data descriptor format for receiving data contains the following fields:

Table 3 Simple RX Data Descriptor

Name	Bits	Description
header_addr	64	DMA address for packet header
data_addr	64	DMA address for packet data
length	16	Length of the data buffer
status	32	Status code indicating success/failure

As with the send descriptor, the receive data descriptor also defines DMA addresses for packet data and packet header separately. The *length* field is written by the device and defines the length of the received data buffer. The *status* field is also written by the device and indicates to the driver whether or not a packet has been received successfully. It can also be used to mark a data descriptor as "used", for example, meaning it points to a valid data buffer in system memory. When hardware marks used descriptors like this, then the driver does not necessarily need to look at the head pointer to know which descriptors can be processed. More complex data descriptors can be used for more complex hardware functions. For example, the descriptor can include an additional field with a VLAN tag or a Quality-of-Service (QoS) tag. It is important to note that data descriptors do not need to be vendor-specific. Different devices carry out the same network operations on data buffers and the information required to carry out these functions is the same for each device. For example, every DMA transaction needs to know the location and the size of the data. There

needs to be a way to tell the device which function to carry out, and there needs to be a way for the device to report back on success or failure of the data transfer operation.

Next to the data path functions, every I/O device has a set of control path functions which are used to configure the device. Ideally, hardware should provide a separate virtual control path for each individual data path resource (like an I/O channel) that can be assigned to a virtual machine. This is not absolutely critical for the NAE architecture as the virtual control path can be emulated in software without significant loss of performance. It is critical that all hardware resources accessed on the data path can be sufficiently partitioned so that resources assigned to different virtual machines are completely isolated from each other. This is required so they can be accessed directly from a virtual machine. Control path resources on the other hand do not necessarily need to be directly accessed from virtual machines. Instead they can stay in control of the privileged control path in the network control domain that can then multiplex resource configuration into individual, software-based virtual control paths which are then exposed to virtual machines. The control path is used to configure the data path. In the NAE hardware API, the hardware-based control path is not strictly defined, simply because the hardware interface is hidden behind the vendor-specific control code API. If a hardware vendor wanted to expose its hardware-based control path directly to a virtual machine, then it would have to implement the NAE-compatible virtual control path as described in section Virtual Control Path. This implementation is feasible as the VCP is designed to eventually be implemented completely in hardware. Implementation

5.6. Prototype Overview

The goal of the prototype implementation is to demonstrate the major principles behind the network acceleration engine framework: we want to show that this solution introduces low overhead into the virtualized platform and lets a user application running inside a virtual

machine take advantage of hardware-based network acceleration while decoupling it from the underlying hardware details.

The prototype furthermore outlines details of possible implementations of the main components of the network acceleration engine architecture. It shows where they sit in a modern computer system and how hardware and software interact in such a system architecture. Most importantly, the prototype proves that an implementation of the proposed approach is feasible.

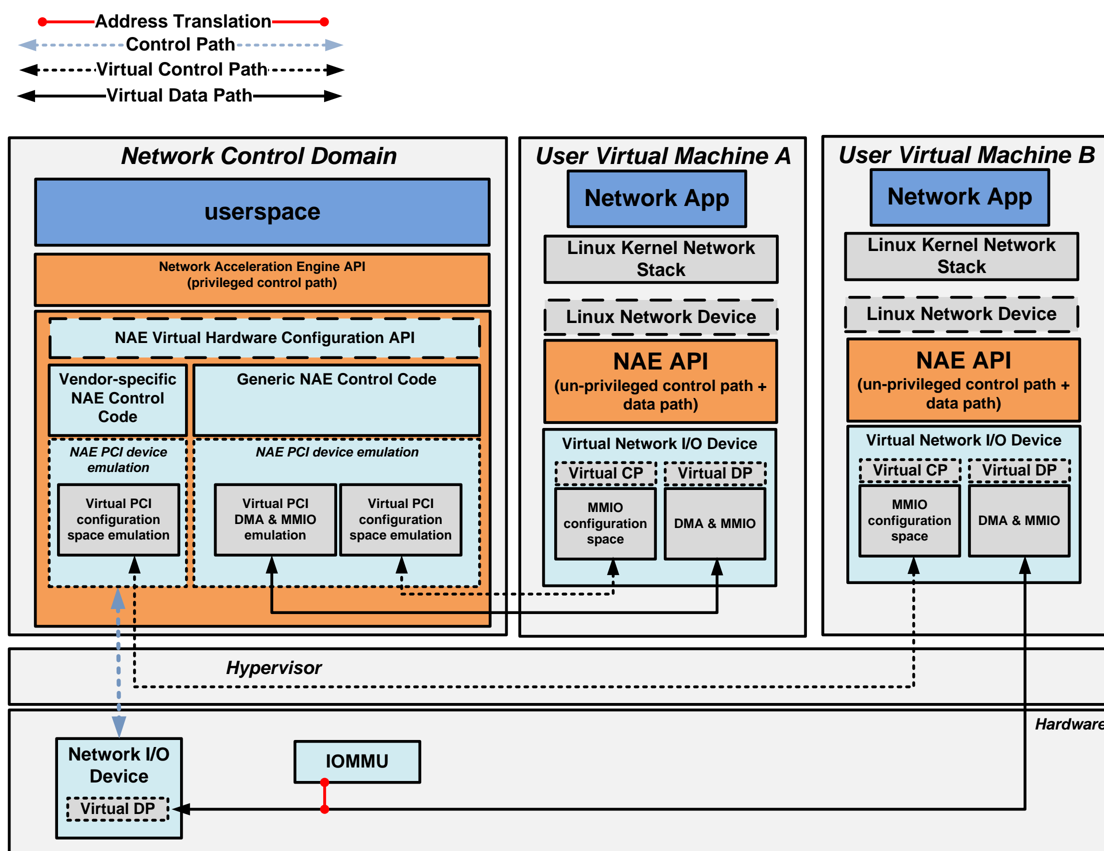


Figure 14 System Architecture Overview of the Implementation

Figure 14 shows details of the network acceleration engine architecture that we are going to target for the prototype implementation. The figure shows two user virtual machines and the network control domain running on top of the hypervisor. The two user virtual machines use different types of network acceleration engines as explained below. Most importantly though, network applications running within the user virtual machines are unaware of exactly what network hardware they run on. The NAE API introduces changes at device and

device driver level in the kernel of the user virtual machine, but leaves the kernel network stack and the interface to network applications untouched. This means that existing applications will benefit from the NAE architecture automatically without having to be adapted.

The implementation of the network acceleration engine architecture is based on a completely virtualized I/O infrastructure that spans hypervisor, user virtual machines, network control domain and the underlying hardware. The guest operating system is configured with a virtual network I/O device looking like a standard Linux network device interface to the guest operating system's network stack. It exposes a configuration interface for controlling device settings to the OS, and offers DMA capabilities for data transfer to and from the device. As shown in Figure 14, these represent the virtual control path (VCP) and virtual data path (VDP) respectively. The two user virtual machines in Figure 14 demonstrate two different configuration options of virtual network I/O devices. Both user virtual machines use I/O memory read/write calls and DMA operations at the lowest level of the NAE virtual network I/O device to communicate on the VCP and VDP, however, the end-points of those channels vary: user virtual machine B interfaces with network hardware while user virtual machine A uses software-based network I/O virtualization.

In the prototype implementation we mainly focus on building a platform that supports a design as used by user virtual machine B in Figure 14. While it is important to provide a virtual I/O path design as used by user virtual machine A in Figure 14, for example, for legacy systems which do not have enough hardware resources to serve all their currently running user virtual machines with a hardware-based virtual data path, the configuration used by user virtual machine B is certainly preferred in most circumstances as it provides the best possible performance. Therefore we focus on the design of the design of the I/O path of user virtual machine B in the prototype implementation as it best demonstrates the value of the

network acceleration engine framework where the I/O path is enhanced with hardware-based capabilities.

In the configuration of user virtual machine B the performance-critical data path is directly connected to the virtualization-aware hardware device while the (unprivileged) control path is connected to the privileged part of the NAE API sitting in the network control domain. In the network control domain, the NAE framework provides a network device emulation layer implementing device functions in software. For user virtual machine B it implements a virtual device configuration space as back-end for the virtual control path.

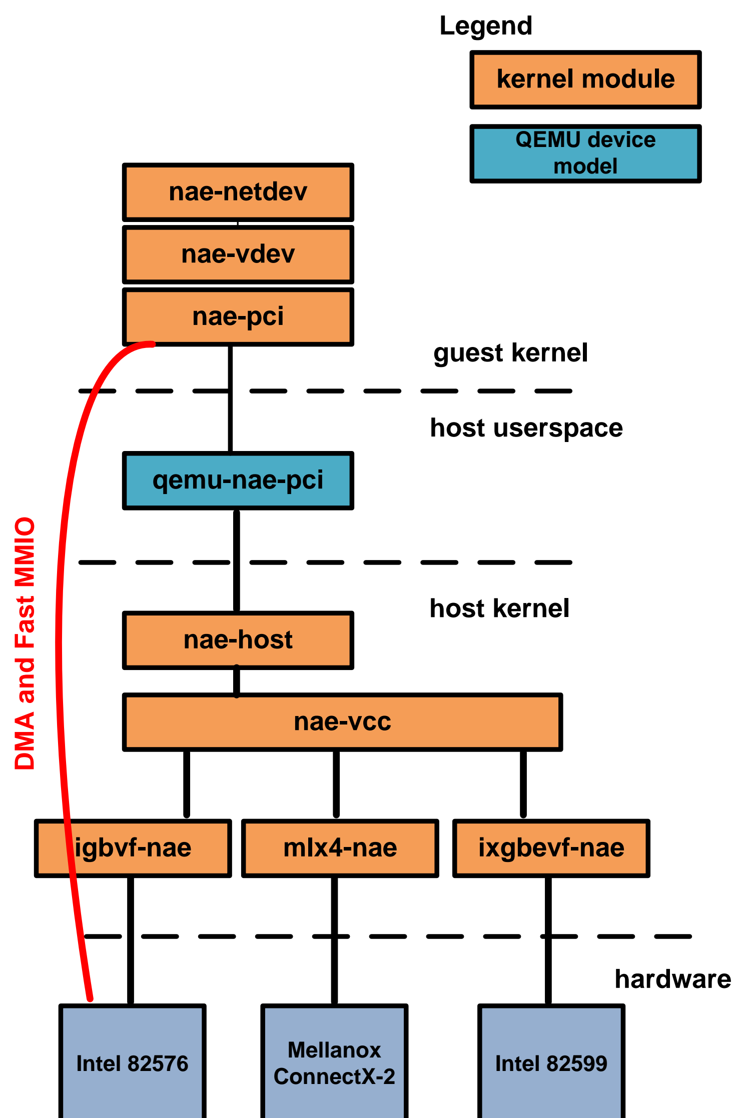


Figure 15 Implemented NAE System Components

Figure 15 shows an overview of all major NAE components we have developed for the prototype. Within the guest OS we run a very slim PCI driver at the very bottom of the stack, here called *nae-pci*. It takes care of mapping PCI BAR[6] memory into the guest kernel's virtual address space, so that these memory-mapped I/O (MMIO) regions can be accessed by kernel modules of the higher layers. The PCI driver also installs interrupts as advertised by the virtual device and exposes them to upper layer kernel modules. We use PCI in the prototype purely because we run KVM on an x86-based system where PCI is the best supported I/O bus infrastructure. On ARM-based systems, for example, we can easily replace the thin PCI layer with a simple driver for an ARM-based I/O bus. The next layer up is the *nae-vdev* kernel module which provides virtual control path and virtual data path interfaces. For the VCP it provides functions to access the MMIO-based configuration space, and for the VDP it provides functions to allocate and control ring buffer resources which are used for transferring data to and from the virtual device. It also connects those operations to the I/O event delivery infrastructure. I/O events are implemented as interrupts generated by the virtual device.[7]

At the highest layer of the NAE infrastructure in the guest kernel we provide the *nae-netdev* kernel module which provides a Linux network device interface to applications. This kernel

[6] PCI Base Address Registers (BARs) are part of the standardized PCI configuration space. The operating system (or sometimes firmware) programs the PCI BARs with system memory address addresses. These are the addresses at which the device's memory regions are mapped.

[7] In theory, this "virtual I/O device layer" is not network I/O specific but it could also be used for storage I/O. We believe that it is quite generic in that it provides a commonly used data transfer mechanism (the circular buffer) and a flexible control interface. In this context it is very similar to virtio (Russell, virtio: towards a de-facto standard for virtual I/O devices 2008) which uses the same virtual I/O bus mechanisms to serve both virtual storage and virtual network interfaces for virtual machines. In practice though, we have not analysed the different requirements of storage I/O, so our particular implementation of this layer might not be optimized for storage-based data transfers.

module registers a Linux network device and makes it available to the operating system. It exposes functions for packet transmission and reception and connects these to the underlying ring buffer operations provided by *nae-vdev*. It furthermore implements functions to make the NAE virtual device accessible and configurable by standard Linux tools like *ethtool* and *ifconfig,* and then translates these configuration commands into I/O reads and writes to the virtual configuration space.

5.7. NAE Guest API

5.7.1. Virtual Control Path

The virtual control path resides in the guest OS kernel and is provided to the user virtual machine for configuring the virtual device. Our implementation uses Linux as a guest operating system. In particular, it runs a customized Linux kernel based on the net-next development tree of Linux 3.3.0 (git repository of Linux net-next development tree n.d.). When the user virtual machine boots up, the VCP starts up first. It initializes the MMIO-based configuration space which is the frontend for communicating with the privileged control path residing in the network control domain. The configuration area within the virtual I/O memory space has a fixed size in each implementation. In our implementation it is 1024 bytes. This seemed to be a reasonable size as it is big enough to contain a large set of configuration options while it is small enough to not consume too much memory[8]. When the VCP initializes, it reads a device type from the configuration that indicates what kind of device it is running on. The type indicates a class of devices and some of their hardware access properties – it is not a vendor-specific ID. The VCP then reads the maximum number of queues and interrupts supported by the virtual device. Finally, it starts to initialize and start the virtual data path.

[8] We have not scientifically deduced the size for the configuration space in the prototype implementation but rather chose a size that seemed acceptable from a memory consumption point-of-view. The size can be very easily adjusted for other implementations, so we found it was not important to choose it very wisely at this point for a prototype implementation.

5.7.2. Virtual Data Path

Figure 16 shows the design of the virtual I/O device exposed to the user virtual machine. The virtual data path (VDP) can be split into three major parts. The primary I/O data transfer mechanism is DMA. We implement the majority of the VDP on top of the Linux DMA API. Through this the real hardware device can directly write to and read from virtual machine memory without involvement of the host CPU. This enables the guest OS and the real device to exchange data directly while bypassing the hypervisor and the network control domain. The Linux DMA API is to a large extent independent of the underlying I/O bus implementation, and so our virtual I/O device implementation can run on PCI-based systems and on systems that sit on a completely virtualized (potentially emulated or para-virtualized) I/O bus infrastructure.

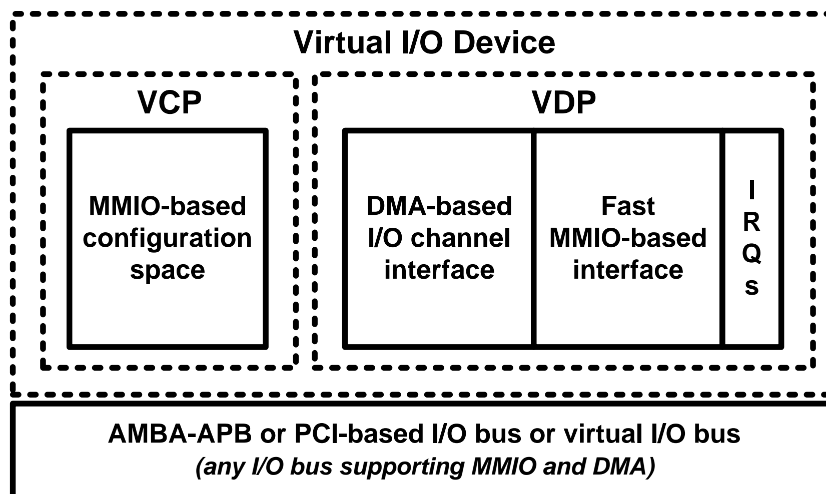


Figure 16 Virtual I/O Device Implementation

Next to the DMA interface the NAE virtual device supports a *Fast MMIO-based (FMMIO) interface*. This interface is used by the guest OS to directly access memory-mapped I/O registers on the real device. The FMMIO interface is configured at device initialization through the virtual control path and it allows fast register reads and writes when the user virtual machine needs to communicate with the real device for carrying out a performance-critical data path operation. For example, the FMMIO space is used to expose head/tail pointer registers of the hardware-based packet queues on the real device. This information

is used on most packet queue accesses when sending or receiving data and so the FMMIO space needs to be accessible efficiently from the user virtual machine without involving the network control domain or the hypervisor in order to keep data path latency as low as possible. FMMIO space access is fast but also safe and flexible: we do not expose any real device registers directly but instead they are mapped into a user virtual machine's virtual I/O memory space region created and controlled by the privileged control path and the hypervisor. Through this indirection the user virtual machine can only access memory regions it is allowed to access. Furthermore, it does not need to know anything about the real device's register layout. Instead, we can map different network hardware into a similar data structure that we expose in a unified way to the user virtual machine. As a result, the user virtual machine has direct (but controlled) access to data path controls of the real device. In order to safely implement the FMMIO interface the real network device needs to follow the minimal design suggestions we define in the NAE hardware API which is covered in the following sections.

Next to the DMA interface and the FMMIO interface the virtual device implements an event delivery infrastructure based on interrupts. Interrupts are configured through the VCP.

5.8. Network Control Domain API

5.8.1. Privileged Control Path

The Privileged Control Path (PCP) resides in the network control domain. The PCP provides the backend for the VCP residing in the user virtual machine, and it controls real devices and their vendor-specific control code (VCC). For our prototype we implement parts of the PCP in user space of the network control domain and parts of it in kernel space.

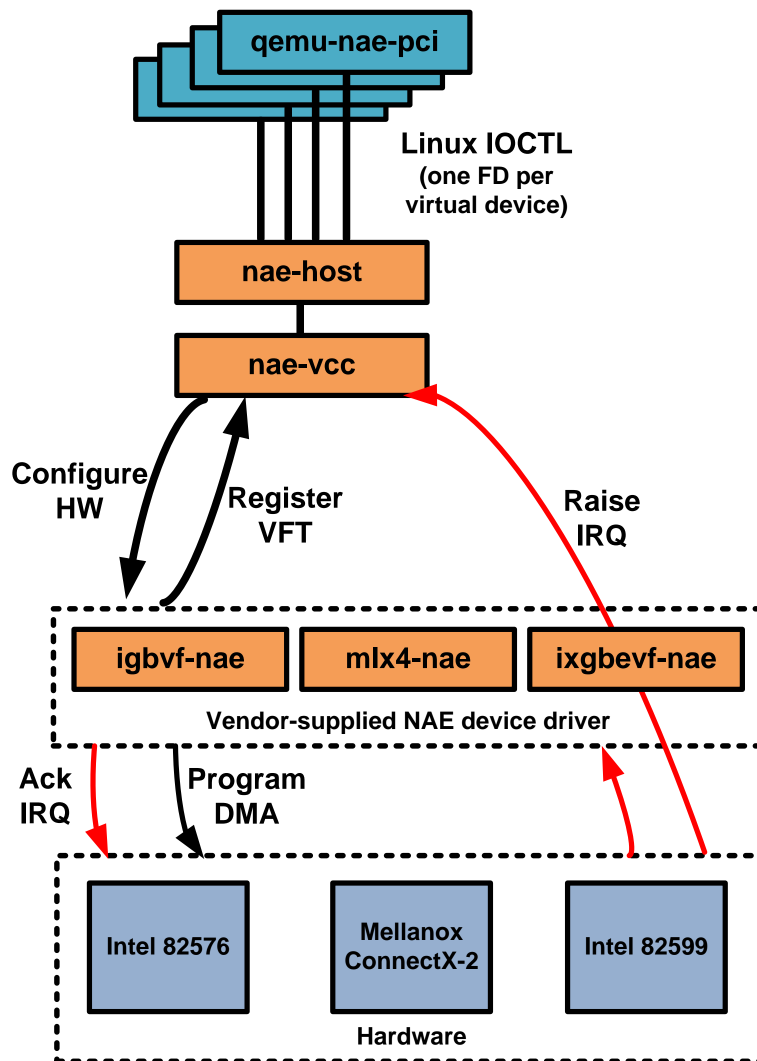


Figure 17 NAE System Components Function Hooks

Figure 17 shows an overview of the implemented NAE components within the network control domain. We have implemented a QEMU device model for NAE (here called *qemu-nae-pci*). The device model is instantiated whenever a virtual machine with a NAE virtual device is created. If multiple virtual devices are requested, then multiple NAE device models run at the same time as part of QEMU. The NAE device model talks to NAE kernel components in order to initialize and configure the real devices which have capabilities that are being exposed to the virtual machine. *qemu-nae-pci* communicates with the main NAE host kernel module (here called *nae-host*) by issuing Linux IOCTL commands. The *nae-host* kernel module exposes a character device at */dev/nae-host* which user space processes can access in order to send IOCTL commands. When a NAE virtual device is initialized, *qemu-nae-*

pci opens a connection on */dev/nae-host*. For each virtual device a new file descriptor is created, so that multiple concurrent connections between user space and the kernel module can exist. The NAE host kernel module coordinates with the *nae-vcc* kernel module which implements the NAE vendor-specific control code API. The *nae-vcc* module is the interface to which vendor-supplied NAE device drivers need to register. When registering such a driver, the driver uses Virtual Function Tables (VFTs) to advertise its capability set as described in section Vendor-specific Control Code. The functions advertised through the VFT implement the NAE API for controlling the real device and its resources. Once advertised, the *nae-vcc* module can then call out into vendor-supplied control code during operation of the virtual device. In the prototype we have implemented NAE device drivers for three different network cards. The first is the *igbvf-nae* kernel module which controls Intel's 82576 1 Gb/s Ethernet controller, the second is the *mlx4-nae* kernel module which controls Mellanox's ConnectX server adapter, and the third is the *ixgbevf-nae* kernel module which controls Intel's 82599 10 Gb/s Ethernet controller. All three kernel modules are based on existing open-source Linux device drivers. They have been modified to remove unused code (mostly datapath functions which have been moved into the guest virtual device driver) and to add additional functions which implement the interface to the *nae-vcc* kernel module. Most of the device-specific code which is part of the control path, like for example programming DMA, writing device-specific registers and booting or resetting the device, is kept mainly unmodified, but wrapped into functions advertised to the NAE framework, so that *nae-vcc* can call out into that code at any time once the device driver is registered. There might also be datapath functions that remain in control of the VCC driver: for example, the VCC driver might be involved in interrupt handling. This is not desirable in all cases, but it is convenient in some. We will go into more details on interrupt processing in the following sections. When the VCC driver is involved in interrupt processing, the physical interrupt from the real device is delivered directly to the VCC driver which acknowledges the interrupt and then

passes it on to the hypervisor through the NAE API which will then raise the interrupt in the associated virtual machine. The following sections give more insights into the PCP components.

5.8.1.1. NAE Device Model for QEMU

In our prototype we develop a new device model for QEMU. We call it the NAE device model for QEMU and it is visualized in Figure 14 and Figure 17 as *qemu-nae-pci* component. It emulates the MMIO-based configuration space that is exposed to the virtual device residing inside the user virtual machine. Hence the NAE device model represents the virtual device's control path backend. As explained previously it is used by the user virtual machine for configuring the virtual device. The NAE device model works by intercepting I/O read and write operations to the memory space, and then passes the associated configuration commands or queries on to further PCP components residing in the network control domain's kernel space. For example, the NAE device model interacts with the NAE kernel module for configuring interrupts for the user virtual machine and for configuring the system's I/O MMU which allows the user virtual machine to directly access some parts of the real device. It furthermore initiates the configuration and set-up of the previously introduced *Fast MMIO space*, and instructs the hypervisor through communication via the *nae-host* module to install the correct access controls on that particular memory-mapped I/O space.

5.8.1.2. NAE Kernel Modules and Integration with KVM

The *nae-host* kernel module manages kernel-level virtual device information. It creates a data structure for each virtual device that is created by the NAE device model code in QEMU. *nae-host* is the main interface for the QEMU component to issue device commands and request real device information. It then cooperates with further NAE components, especially the NAE vendor-specific control code kernel module, called *nae-vcc*, which controls real I/O

devices. It is also tightly coupled with KVM. While we use KVM in our prototype implementation, the NAE framework does not rely on a particular hypervisor. However, it requires that the hypervisor exposes an API to control interrupt mappings, DMA address translation mappings (the I/O MMU interface) and memory-mapped I/O bindings for a user virtual machine. These three control interfaces are crucial for building a virtual I/O path with good performance.

nae-host interacts with the NAE device model in QEMU through the Linux IOCTL interface. For example, when the user virtual machine writes to its virtual configuration space to indicate that it wants the virtual device to start up, then the NAE device model intercepts that I/O write in QEMU and passes the appropriate IOCTL command down to the NAE host kernel module. There is an individual path for IOCTL commands per virtual device, as shown in Figure 17.

5.8.2. Vendor-specific Control Code

Vendor-specific control code can register itself and its real I/O device to the NAE host kernel module by implementing functions of the NAE VCC API. In the prototype implementation these are exposed through Linux virtual function tables (VFTs). The VCC API defines a set of functions that a device vendor has to implement in their own VCC kernel module. Example functions are hooks to start and stop the device, assign interrupts, read capability information or hardware details like pointers or offsets to registers that might be exposed to the user virtual machine. Vendor-specific control code is significantly simpler in the NAE framework than in a legacy environment: it just needs to boot up the device into an operating state, but it does not need to allocate any data path resources and functions. All of this sits inside the user virtual machine.

In the prototype implementation we use the Intel 82576 network card and implemented a NAE VCC driver for it. The VCC driver is a modification of the Intel-based *igbvf* Linux device driver which is supplied by Intel for running the 82576 device on a hypervisor-based system.

Instead of running a whole data path on the device like the *igbvf* driver does, our VCC driver only initializes the hardware at system boot time. Then it registers with the NAE host kernel module and advertises its hardware resources and capabilities. Once it has a virtual device associated with (some of) its hardware resources, then it continuously receives requests through the NAE host kernel module to configure the real device or hand out device information. The VCC module still manages hardware link status information as well and passes that up to the guest through the NAE API if requested. The VCC module might also still take care of basic interrupt management. We have also implemented VCC modules for Intel's 82599 and Mellanox's ConnectX network controllers. They are visualized as *ixgbevf-nae* and *mlx4-nae* in Figure 17.

5.9. NAE Hypervisor API

5.9.1. Interrupt Routing

Interrupts are managed through the hypervisor API. In our case they are controlled by KVM. When our NAE device model for QEMU, *qemu-nae-pci,* requests interrupt configuration for a particular virtual device, then the VCC module which serves that virtual device will advertise its interrupt support and configuration. Interrupts which are assigned to the virtual device in question are then programmed into KVM. A design advantage of the NAE I/O architecture compared to existing approaches is that interrupts can be routed in a flexible way. In traditional hypervisor-based approaches interrupts from I/O devices are either all directly routed to the user virtual machine or they are all routed through the hypervisor. With the NAE I/O infrastructure on the other hand it is possible to configure each virtual device interrupt individually as described in the following section.

Depending on the virtual device configuration, interrupts from the real I/O device might not be directly passed through to the user virtual machine, but instead they might be routed to the associated VCC module at first, so that it can handle device-specific operations like

writing a register to acknowledge the interrupt before the interrupt is then raised in KVM and then inside the user virtual machine. Routing of interrupts through the VCC module and then to KVM is implemented completely within kernel space of the network control domain. KVM then raises the appropriate interrupt in the user virtual machine's virtual CPU. Efficient routing of interrupts is important from a performance point-of-view and especially in order to offer low-latency services where interrupts need to be reported to the user virtual machine very quickly. The VCC module for a particular network device can also advertise that it is not interested in processing certain interrupts, and in that case those interrupts go directly to the user virtual machine's virtual CPU. Flexibly configurable interrupt routing can provide significant performance improvements for network I/O as different network applications have different requirements on the event delivery mechanism of the underlying I/O infrastructure. For example, these requirements can depend on the type of network traffic or Quality-of-Service guarantees (for example, high-bandwidth vs. low-latency).

5.9.2. DMA and Memory-mapped I/O

The NAE hypervisor API includes functions that configure the system I/O MMU and program it with DMA regions that are exposed to a particular user virtual machine and can be accessed by a particular real I/O device. The NAE architecture requires a hardware-based I/O MMU on the system for translating DMA addresses used by the user virtual machine into physical addresses used by the real I/O device without any intervention by the hypervisor or the network control domain. It is also used for protecting the network control domain and other user virtual machines from real I/O devices which issue invalid or malicious DMA requests. Strictly speaking, the NAE architecture requires a memory management unit on the I/O path from the user virtual machine to the real I/O device. This does not necessarily need to be a system-wide I/O MMU as most modern systems provide, but instead the I/O device itself might contain a MMU, as, for example, the Mellanox ConnectX which we use for our design evaluation. In that case, the device's MMU needs to be programmed by both

hypervisor and VCC module in cooperation – as the MMU is device-specific, vendor-specific control code is required to program it.

The hypervisor furthermore needs to provide a mechanism to safely expose memory-mapped I/O regions to user virtual machines. As part of the NAE hypervisor API we create virtual memory regions made up of emulated memory-mapped I/O space and directly mapped real I/O device memory. A virtual memory region is exposed to a user virtual machine by advertising it as I/O memory space of the virtual network device. The directly mapped real device memory is used for the *Fast MMIO space* described previously, and it is crucial for providing a high-performance data path. This approach requires that the hypervisor can take arbitrary memory regions of the system and construct a new virtual memory region which is to be exposed to the user virtual machine. It needs to be able to control access to that virtual memory region and provide different traps (or no traps at all) depending on which parts of the virtual memory region is accessed. The user virtual machine is unaware of the details of the underlying memory regions. We have implemented these capabilities for our prototype as part of QEMU/KVM.

5.10. NAE Hardware API

5.10.1. Data Path and Control Path Separation

The NAE architecture requires a clear separation of control path operations and data path operations. This isolation of information flow needs to be enforced all the way down to the hardware-level APIs. Data path functions can be directly assigned to user virtual machines while control path functions are owned by the privileged control path residing in the network control domain.

In practice we found that the majority of today's network hardware is not designed with sufficient separation between configuration functions and I/O data transfer functions. For example, it is required that device registers controlling different packet queues can be

assigned to different user virtual machines. With today's virtualization layer capabilities this means that these registers may not reside in each other's page boundaries as memory protection today only works on per-page granularity. This forces a hardware design where all control registers for a particular packet queue (or similar data path resource/function) are grouped together within a memory region and isolated from control registers for other packet queues. Many of today's network devices group most, if not all, control registers together, even if they control data path resources that are independent of each other and therefore could well be assigned to different user virtual machines. Such an implementation makes it challenging to efficiently use these control registers in a virtualized system.

Regarding system design our research shows that there clearly is a need for enabling more fine-grained memory I/O accesses. Vendors do not necessarily want to use up 4K bytes[9] of I/O memory for each virtual data path that is supposed to be exposed to a user virtual machine. However, with currently available hardware (x86 and ARM) it is very expensive to allow access control within individual memory pages which makes it practically unusable for real workloads. A current trend in system design enables even bigger page sizes (up to 1 Gigabyte on 64-bit x86) and will make fine-grained I/O memory access control even more of a challenge.

5.10.2. Data Movement

The NAE architecture relies on a DMA-capable device for best I/O performance. The current implementation expects that the device uses some form of (producer/consumer) ring buffer mechanism to transfer data between the hardware and the virtualized system. This means that the device and NAE virtual device driver residing in the user virtual machine share a memory region reserved for DMA transactions. The device posts one or more data buffers to this memory region when it receives a network packet, and it retrieves one or more data buffers from this region when the user virtual machine has advertised that it has network

[9] 4K is the default memory page size of today's systems for both ARM and x86 architectures.

packets to send out. Both parties advertise processing of data buffers by advancing the value of the head/tail pointers (sometimes also called producer/consumer pointers) of the associated ring buffer.

Today’s network hardware uses vendor-specific or device-specific ways of posting/retrieving data buffers from network applications. However, our research reveals that the core I/O transfer mechanism is sufficiently similar in order to unify data path access across different network hardware. As explained in previous chapters, most network I/O devices and their device drivers maintain a list of data descriptors in order to communicate details about their associated list of data buffers. As of today, different network devices use different descriptor formats to advertise information about the data buffers to be sent or have been received. The driver needs to be aware of the descriptor format used by the device; otherwise no DMA transfers can be carried out. With the NAE hardware API we would like to make a step towards unifying the data transfer interface between network hardware. A *minimum send descriptor format* that we have implemented in our prototype consists of the DMA address for transferring the data buffer and the size of the data buffer to send out. Also there needs to be a status field which is written by the device to report back on whether or not the transmission has been successful. The send descriptor might furthermore contain a command field to indicate to the device what offload functions should be carried out on this data buffer (for example, VLAN tagging or checksum calculation etc.). The *minimal receive descriptor format* contains a DMA address for data buffer transfer. Again it might furthermore contain a field indicating to the driver what offload functions have been applied to the associated data buffer. Devices that do not support certain offload functions will ignore the related descriptor fields. We have implemented this simple hardware data path on top of the Intel 82576, the Intel 82599 and the Mellanox ConnectX Ethernet controllers. The implementation shows that it is possible to unify the core parts of the data path of different network hardware. Such a generalized, high-performance I/O path builds the basis

of high-performance network communications for virtual machines which can be migrated across heterogeneous hardware platforms.

Our prototype uses network devices not fully optimized for the NAE architecture. This is purely because we do not have the capabilities and facilities to design and implement hardware conforming to the proposed I/O architecture ourselves, and therefore we rely on devices that are out on the market today and available to us for testing and evaluation. Even so, our evaluation shows promising I/O performance results. If hardware vendors were moving towards a NAE-compatible interface design, then we can expect even greater advantages compared to traditional approaches.

We believe that the developed prototype implementation shows valid results even though we cannot demonstrate performance of the ideal, proposed hardware/software interface due to a lack of NAE-compatible hardware. As mentioned above, today's network I/O devices do not sufficiently separate control registers of different virtual data paths. Good isolation here is critical for performance as virtual data path registers need to be accessible directly from user virtual machines without compromising security between different user virtual machines using the same I/O device. We have worked around this issue during the performance evaluation by only assigning a single virtual data path per I/O device to virtual machines, so that different user virtual machines cannot harm each other.

Furthermore, as mentioned above, we do not have a common hardware-based virtual data path interface between different I/O devices due to being restricted to using legacy network devices for our prototype implementation. In order to overcome this issue, we have introduced a translation function between different vendors' hardware-based resources. In particular, this only affects the hardware-based data buffer descriptor format which varies between vendors. The user virtual machine's virtual device driver needs to be made aware of the currently used data descriptor format in order to use DMA, and for this purpose we are using a simple translation function that is only invoked once at boot time of the virtual

machine. It queries the network control domain for the current data descriptor format through the virtual control path before initializing the virtual data path with the retrieved information. As the translation function is only called once at boot time, it does not in any way affect the performance results we have gathered whilst the virtual machine is up and running. In a follow-up implementation that runs on fully NAE-compatible hardware this translation function can simply be removed.

Despite these two workarounds we had to introduce due to the use of legacy hardware, we believe that the prototype implementation is well suited for proving the performance advantage of the NAE I/O architecture compared to purely software-based approaches. It furthermore proves that the virtual data path of different vendors' devices can be easily unified - even if the hardware is not NAE-compatible. We have implemented interfaces to work with three different network devices of two different vendors showing the principle of hardware independence which is one of the most critical advantages of our proposed approach.

6. Performance Evaluation

6.1. Overview

In the following sections we describe results of a performance analysis of our NAE prototype implementation. In this context we want to show how our approach compares to existing solutions with regard to network throughput and latency experienced by virtual machines running on a NAE virtualized I/O path.

It is important to reiterate that the main goal of the NAE architecture is not to significantly improve I/O performance of the virtual data path. It is also important to note that the NAE virtual data path can be enhanced with prior art technologies that are purely focusing on improving efficiency of the virtual data path like, for example, Exit-less Interrupt (ELI) (Abel, et al. 2012). Various other research activities focus on individual improvements of the I/O data path. These are not competitive technologies to the NAE design, but instead, some, like ELI, can be used on top of the NAE infrastructure and further improve the NAE virtual data path implementation. The value-add of our proposed I/O architecture is to enable a virtual data path that is vendor-independent and device-independent while maintaining reasonable performance. This means that various further performance improvements can be implemented on the virtual data path and by using the NAE framework these will be automatically exposed to the virtual machine through a generic interface that supports features of dynamic cloud infrastructures, like virtual machine migration.

The main goal of the performance evaluation presented in the following sections is to prove that the NAE design provides I/O performance comparable to existing approaches that do not offer hardware-independence, such as traditional direct device assignment mechanisms. An evaluation like this should also show that our NAE design does not have any significant flaws that impact network performance. Ideally, we also want to show that our architecture

performs better than purely software-based approaches. Otherwise we would not be able to demonstrate the benefit of using hardware-based network accelerators.

For our performance evaluation we carry out tests that measure network bandwidth and network latency as experienced by virtual machines running on a NAE-enabled virtual I/O path. Bandwidth and latency are the most important performance metrics for any network-based application. Bandwidth indicates how fast an application can send and receive as much data as possible over the network. Latency measures how quick data can travel from a source to a destination over the network. An ideal I/O path implementation achieves high bandwidth and low latency, but for any implementation it is challenging to do both. However, these two metrics clearly characterize how any network-based application can potentially perform. Some applications might be more latency-sensitive (for example, voice connections and gaming applications) and others might require mainly high bandwidth (for example, video streaming). Of course there are also applications that require a mixture of both and different applications have different I/O access patterns. However, as we are proposing a new I/O architecture rather than a specific improvement on the I/O data path that affects specific network functions, like, for example, TCP Segmentation Offload (TSO) and Large Receive Offload (LRO), we think that this evaluation is sufficient for demonstrating the relevance of the NAE I/O infrastructure. On top of our architecture it will then be possible to enable further, network-specific, individual performance improvements that cover particular application requirements.

Whilst typical network metrics like bandwidth and latency are critical for any performance evaluation involving network traffic, for us it is also very important to measure what impact a particular solution has on overall system utilization. In particular, the most relevant metric in this context is CPU utilization, because it shows how much compute power the system requires for moving data from virtual machines to the network, and vice versa. If it requires a lot of CPU resources, then it cannot use those resources for other tasks running on the

system at that time. For example, high CPU utilization would limit the number of virtual machines that can be run concurrently, simply because at some point the system would run out of CPU resources. Low CPU utilization numbers indicate that the technical solution is scalable and does not restrict the system with regards to other (potentially CPU intensive) applications that must be run whilst virtual machines stress the network I/O path. We think that CPU utilization figures are a good performance metric for evaluating an I/O virtualization framework and therefore we analyze these values as well in detail in the following sections.

The following tests demonstrate a simple use case where a virtual machine that uses a NAE I/O path communicates over the network to another endpoint. We have two different test setups: at first we measure performance at 1 Gb/s (Gigabit per second) speed and then we measure performance in the same way at 10 Gb/s speed. In the future, setups that only run at 1 Gb/s speed potentially are not relevant any more, but in today's data center environments one can still find them frequently. When running at Gigabit speed, the network is more likely to be the bottleneck as today's platforms are powerful enough to process network I/O at sufficiently high rates. Nonetheless, as it is a frequently occurring real-world setup, it is important to evaluate I/O performance of our prototype in that configuration. When moving to 10 Gb/s speed, the bottleneck will more likely be on the virtualized platform and therefore it is more likely that we can see performance differences of our prototype compared to other technologies. As Ethernet speed is constantly increasing and we can see 40 Gb/s and even 100 Gb/s Ethernet devices entering the market, we put slightly more emphasis on the analysis of our prototype when used with faster networks.

We evaluate all relevant properties of the I/O path on the platform where the virtual machine runs. This should give us a good indication of the I/O performance of a virtualized platform implementing our NAE prototype. We leave tests evaluating platforms running multiple virtual machines concurrently to future research activities following this thesis.

However, we try to give already some indications on scalability issues during the following sections.

6.2. Test Setup

We use two HP z600 workstations for extensive network performance tests of the NAE prototype implementation. Both workstations are configured with a 10 Gb/s Ethernet controller and a 1 Gb/s Ethernet controller. We connect the 10 Gb/s ports together through a 10 Gb/s ProCurve 5406zl Ethernet switch. The switch does not connect to any further networks or other machines in order to prevent disruption of tests from cross traffic flowing through the switch. We connect the two 1 Gb/s ports together with a standard cross-over Ethernet cable. The direct connection here ensures that there is no external bottleneck on the network path.

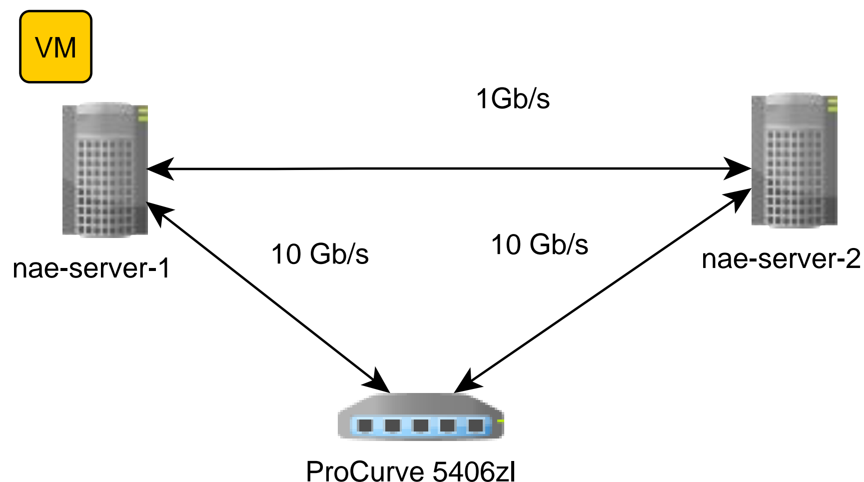


Figure 18 Test Set-up

The HP z600 workstation has a dual-core Xeon CPU (model E5640) running at 2.66GHz. Both workstations are configured with 6 GB of memory. The memory configuration is not significant in the context of our performance evaluation. It is simply set to a value high enough to not create any memory-related bottlenecks on the network I/O path.

The first workstation runs the prototype implementation of the NAE framework. We call it *nae-server-1*. We analyze some of these details in the 10 Gb/s testing section. The second workstation does not run any system virtualization but native Linux (RedHat Enterprise Linux 6). We call that *nae-server-2*. This second workstation is only used as endpoint for our test runs, and it represents the source or sink of network traffic respectively. We do not measure anything on that workstation and we ensure that it is never the bottleneck of our communication path. The first workstation is the main testing machine. It runs a customized 64-bit Linux kernel implementing our NAE prototype APIs with a RedHat Enterprise Linux 6 distribution.

We compare our NAE framework prototype implementation with two other approaches commonly used on virtualized systems:

- virtio: virtio is a software-based I/O virtualization technology. With virtio, networking for virtual machines is emulated in a QEMU-based device model running inside the network control domain as user space process. The virtio device model can then be connected to a kernel-based Linux network bridge attached to a real I/O device, so that the virtual machine can communicate with other endpoints on the physical network.
- PCI pass-through (PCI PT): PCI pass-through allows attaching a real I/O device directly to a virtual machine. The virtual machine sees the full hardware device interface and runs a device-specific driver. In this scenario the virtual machine has full control over the real device.

In section 2.2.1 and 2.2.3 we have already given more details of both virtio and PCI pass-through technologies. We have selected these two approaches, because they are the most commonly used technologies for software-based and hardware-based I/O virtualization approaches in today’s real-world deployments. virtio is a well-developed and already highly-optimized, stable implementation of a reasonably generic software-based virtual I/O interface. It can be used across various hypervisor and guest operating systems. One could

say it is the best and most universal purely software-based solution currently on the market. PCI pass-through currently is the standard solution for providing high-performance I/O access to virtual machine. Most, if not all, current hypervisors implement some form of PCI pass-through support. It typically achieves the highest I/O throughput one can get from today's virtualization solutions. By comparing our I/O framework against these well-known solutions that are potentially the best in their type of solution (software-based and hardware-based, respectively), we think we can provide a very valuable and relevant evaluation of our concept and implementation.

6.3. Test One: Gigabit Ethernet Performance

6.3.1. Test Description

In a first test we evaluate the performance of the network acceleration engine framework on a 1 Gb/s Ethernet link. For this we use the two directly connected 1 Gb/s Ethernet ports on the z600 workstations. Those are Intel 82576 controllers and on *nae-server-1* the port is controlled by our *igbvf-nae* kernel module which we described previously and pictured in Figure 17. For the first test we run a single virtual machine on *nae-server-1*. The virtual machine runs the network benchmark applications. It has two Gigabytes of memory assigned to it, and it is configured with a single dual-core virtual CPU. We use *netperf* for measuring throughput, and the Linux *ping* tool for measuring latency between the endpoints of the communication. In this test we establish a connection from the virtual machine to the *nae-server-2* that also runs the benchmark tools. Hence, the following measurements cover a network path from the virtual machine on the first workstation over the network cable to the second workstation, and vice versa. We measure CPU utilization in the network control domain residing on *nae-server-1*.

In order to better understand the following performance measurements on our virtualized system, it is important to give more insights into how QEMU/KVM works in our test

environment. We use the latest public development version of *qemu-kvm* which is versioned 1.0.50. We configure QEMU in “iothread” mode which means that QEMU will start a separate thread for each virtual CPU of the virtual machine and those can execute guest code in parallel. Next to those VCPU threads QEMU also starts an additional *iothread* per virtual machine. That thread handles processing of I/O events related to that virtual machine from different components within QEMU/KVM.

6.3.2. Bandwidth

All three approaches saturate the 1Gb/s link at around 942Mb/s on both transmit and receive operations. The exact throughput results are show in Figure 19. This means that on a 1Gb/s link, the network connection represents the bottleneck restricting the performance to at most 942Mb/s. Therefore, when looking purely at the throughput, we would not see any significant differences between the three competing approaches in this test scenario.

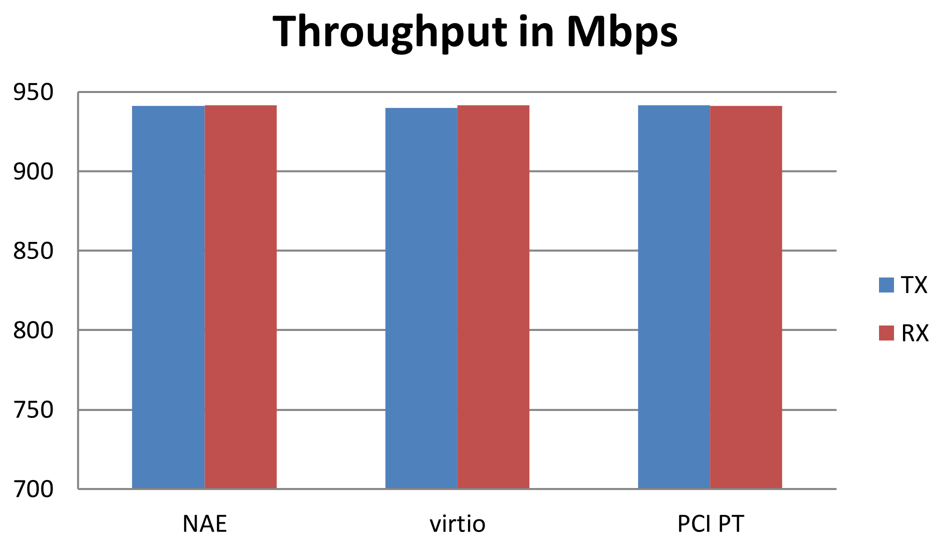


Figure 19 Throughput in Mbps

When we look at the computing resources used during the network tests, then we can see the first differences between the three technologies. We measure CPU utilization within the network control domain with the Linux tool *top*. Figure 20 shows CPU utilization measurements from the transmit operation (TX) while Figure 21 shows the results from the receive operation (RX) when running the *netperf* throughput measurement tool.

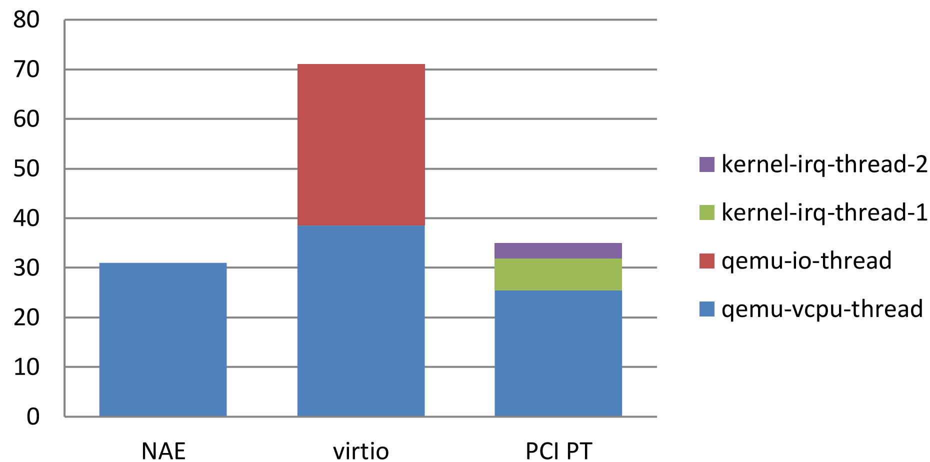


Figure 20 CPU Usage on TX

In Figure 20 we can see that the NAE framework has the lowest overhead within the network control domain. Its overhead comes purely from running the virtual machine as a QEMU VCPU thread. virtio on the other hand uses more than double of the compute resources than the NAE framework uses. The QEMU VCPU thread runs at 38% CPU utilization while it only runs at 31% with NAE. The increased overhead in virtio is very likely to be due to different packet processing overhead within the virtual machine: for NAE the virtual machine device driver simply issues DMA operations which are carried out by the real device without involving the VCPU while virtio's device driver sits on top of a purely software-based QEMU device model which is involved in packet processing and therefore requires VCPU time. On top of that the virtio configuration runs a *qemu-io-thread* which takes care of handling I/O events for virtual machines. In that case, network packets are processed by QEMU in user space before they are passed to the Linux kernel-based network bridge through a Linux *tap* interface (Universal TUN/TAP device driver n.d.). The *tap* interface implements a user space to/from kernel space memory copy of the full network packet. This accounts for most of the CPU overhead of the *qemu-io-thread* when using virtio. The PCI pass-through technology runs the main QEMU VPCU thread only at 25.5% CPU utilization. This overhead is lower than for NAE on the transmit path. However, the PCI pass-

through approach runs two kernel threads which handle interrupts from the real device and these show significant overhead at 6.5% and 3% respectively. These kernel threads are part of the KVM kernel module which installs them to receive interrupts from the real I/O device indicating transmit or receive operations (such as, packet transmitted and packet received). When an interrupt is received by a kernel thread, then the related PCI pass-through device is found and the interrupt is passed on to the virtual machine via QEMU/KVM. Overall, the NAE configuration shows the lowest computing overhead in the network control domain for this configuration.

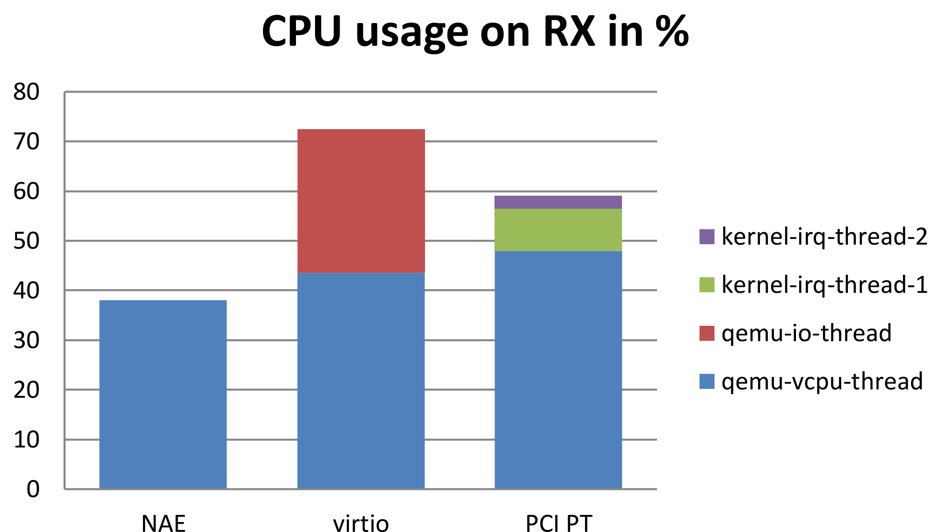


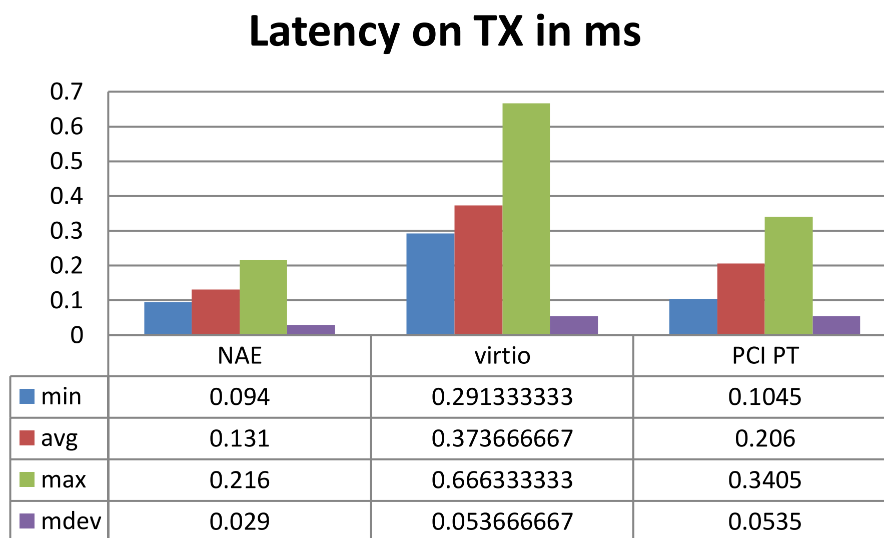
Figure 21 CPU Usage on RX

The compute resource consumption on the receive path looks mostly similar to the transmit path. It is shown in Figure 21. The QEMU VCPU thread only runs at 38% CPU utilization when using the NAE framework while it runs at 43.5% with virtio and at 48% with a PCI pass-through configuration. The virtio approach also runs the *qemu-io-thread* as in the transmit case which runs at an additional 29% CPU utilization. That makes virtio use about 72% CPU utilization while the NAE prototype only uses 38%. PCI pass-through does not use the *qemu-io-thread* either but again runs two kernel threads for handling interrupts from the real device. On the receive path these run at 8.5% and 2.5% CPU utilization respectively. Overall

that makes the PCI pass-through approach use 59% of the CPU which is about 55% more overhead than the NAE framework.

6.3.3. Latency

We measure latency with the Linux *ping* tool. Figure 22 shows latency results for the transmit path (TX) while Figure 23 shows latency results for the receive path (RX). The ping tool gives us the minimum latency incurred (min), the average latency incurred (avg), the maximum latency incurred (max), and then the standard deviation (mdev) as shown in both figures.



	NAE	virtio	PCI PT
min	0.094	0.291333333	0.1045
avg	0.131	0.373666667	0.206
max	0.216	0.666333333	0.3405
mdev	0.029	0.053666667	0.0535

Figure 22 Latency on TX

The NAE framework shows by far the lowest latency with on average 0.131ms on the transmit path and 0.1485ms on the receive path. The PCI pass-through approach has on average 0.206ms on the transmit path and 0.294ms on the receive path. This is nearly double the latency of the NAE prototype implementation. The increased latency comes from different IRQ routing paths between NAE and PCI pass-through: in the NAE architecture, IRQ routing stays within the kernel at all times and interrupts are directly passed through to the virtual machine when they arrive from the real device, while for PCI pass-through, IRQ handling is done within a slightly delayed kernel worker thread.

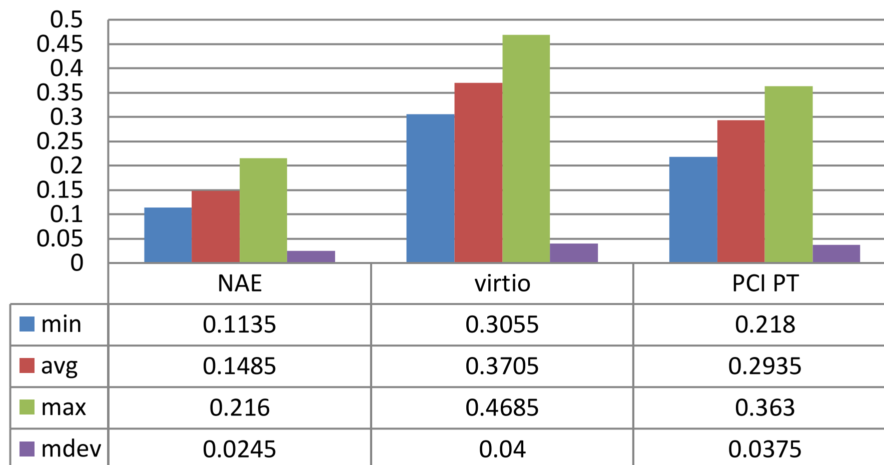


Figure 23 Latency on RX

While the NAE implementation of IRQ routing outperforms the PCI pass-through implementation in this scenario, we appreciate that there is a significant benefit in using kernel worker threads for interrupt processing. Most importantly, for improved scalability it is best to run as little code as possible in interrupt context, and threaded IRQ processing achieves exactly that. Due to the flexible interrupt routing design of the NAE hypervisor API, we can actually configure some individual interrupts to be processed directly in the kernel as described above (for example, for virtual devices used by low-latency network services) while others are processed with slightly delayed but more scalable kernel worker threads.

virtio shows the most significant latency with 0.374ms on the transmit path and 0.371ms on the receive path. This is nearly three times the latency than what we measure when running the NAE framework. For virtio, interrupts are handled by the QEMU user space process and packets need to traverse the Linux kernel-based network bridge in the network control domain before they can be delivered to the virtual machine. This adds additional context switch overhead and significant packet processing delay on both receive and transmit paths. Packets also need to be copied twice, for example, on receive they need to be copied from the network card to the network control domain, and then further on to the virtual machine. On the transmit path there are also two copies which move data in the other direction.

6.4. Test Two: Ten Gigabit Ethernet Performance

6.4.1. Test Description

For the 10 Gb/s throughput and latency tests, we use Intel's 82599 controller on *nae-server-1*. When running the NAE prototype, the adapter is controlled by our *ixgbevf-nae* kernel module, as described in previous sections. For the test we run a single virtual machine on the first z600 workstation. The virtual machine is configured with two Gigabyte of system memory and we assign a single dual-core virtual CPU to it. That way, the virtual machine can freely use all compute resources of the z600 workstation. We do not pin VCPUs to physical CPUs.

The second z600 uses a 10 Gb/s Mellanox ConnectX controller which is not virtualized in the test, because *nae-server-2* only acts as source or sink of network traffic. It runs four hardware-based receive and transmit queues (spanning across all four cores of the machine) which enable fast transmission and reception of network packets and ensure that this workstation is never the bottleneck of the connection. We also measure CPU utilization on this second workstation to make sure that *nae-server-2* is not restricting the network performance on *nae-server-1*.

When running bandwidth and latency benchmarks, we measure the network I/O path from the virtual machine running on *nae-server-1* through the ProCurve Ethernet switch to *nae-server-2*, and vice versa. The ProCurve switch has a simple configuration in all test runs. There are no bandwidth restrictions on the switch ports, no VLANs are used, and the switch itself forwards network traffic by looking purely at layer-2 Ethernet header information.

6.4.2. Bandwidth

Figure 24 shows the overall best achieved throughput of all three approaches. We measure throughput with *iperf* and use its TCP bulk data transfer test. As the z600 has a dual-core CPU and we assign a dual-core virtual CPU to the virtual machine, all approaches achieve the

best performance when running two simultaneous TCP streams. We can configure that by calling *iperf* with the *–P 2* command line option. The *iperf* server running on *nae-server-2* can potentially run up to four simultaneous TCP threads as it is has four CPU cores and is configured with four hardware-based receive and four hardware-based transmit queues. Both the NAE prototype and PCI pass-through can achieve just over 8Gb/s when sending TCP bulk data traffic while virtio achieves about 6.5Gb/s. The receive performance for TCP is significantly lower for all three approaches. PCI pass-through achieves slightly over 4Gb/s while the NAE prototype achieves slightly over 3Gb/s. virtio again shows the lowest performance at just over 2Gb/s.

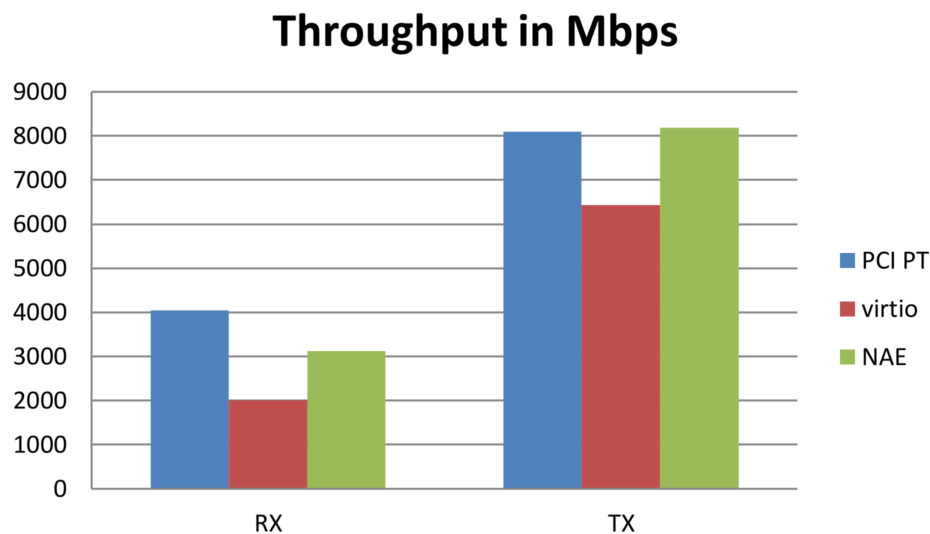


Figure 24 Throughput in Mbps

The low TCP receive performance in our tests mainly comes from head-of-line blocking issues on the receiver side: we are only using a single TX queue and a single RX queue on *nae-server-1*, and so here the receiver cannot quickly enough process TCP ACK packets as data in the same queue is being served. The sender then assumes the packet has been dropped and has to initiate a re-transmission. Due to this pattern in TCP processing, packet loss in a TCP connection causes a significant drop in overall TCP throughput.

While this clearly limits TCP receive performance, it does not prevent a fair test between all three approaches. virtio only supports a single TX and a single RX queue, and so we

configure the NAE prototype with only a single I/O channel, and also the native *ixgbevf* driver used for the PCI pass-through configuration is configured with just a single queue pair. Head-of-line blocking is a well-studied TCP problem preventing higher bandwidth connectivity when using a single TCP stream with a single input/output queue, as described in (Scharf and Kiesel 2006). Using multiple simultaneous TCP streams across multiple receive queues solves this problem and enables achieving higher TCP bandwidth when the receiver is stressed with processing TCP packets. Receive-side Scaling (RSS) could for example enhance TCP bandwidth when using multiple TCP connections simultaneously: it spans TCP connections across multiple receive queues which are typically processed by multiple CPU cores. RSS or similar multi-queue technologies *can* be implemented on top of all the evaluated approaches, and therefore should not be a differentiator for any of them in our evaluation. The native, non-virtualized driver for the Intel 82599 network controller, called *ixgbe*, implements multiple receive queues and RSS. However, the virtual function driver *ixgbevf* we use in the PCI pass-through configuration does not implement any of these features. The virtio implementation we use also only supports a single receive queue at any time. On the other hand, our NAE prototype implements multiple receive and transmit queues. For evaluation purposes, however, we restrict the NAE prototype to run with a single send queue and a single receive queue. That way we should be able to present a fair comparison.

While achieving the bandwidth numbers illustrated in Figure 24, we have recorded the CPU load each approach puts on the system when sending and receiving data. Figure 25 shows CPU utilization results when sending TCP bulk data. We have measured CPU load with the Linux *mpstat* tool. *mpstat* gives slightly more detailed CPU utilization data than *top*. In particular, it can separate out measurements per individual CPU core, and it can break up exactly in which part of the system processing overhead is occurring. The percentage given

in this figure adds up CPU utilization on both cores of the physical machine, so the maximum CPU load that any workload can achieve is 200% (if both cores are utilized for 100%).

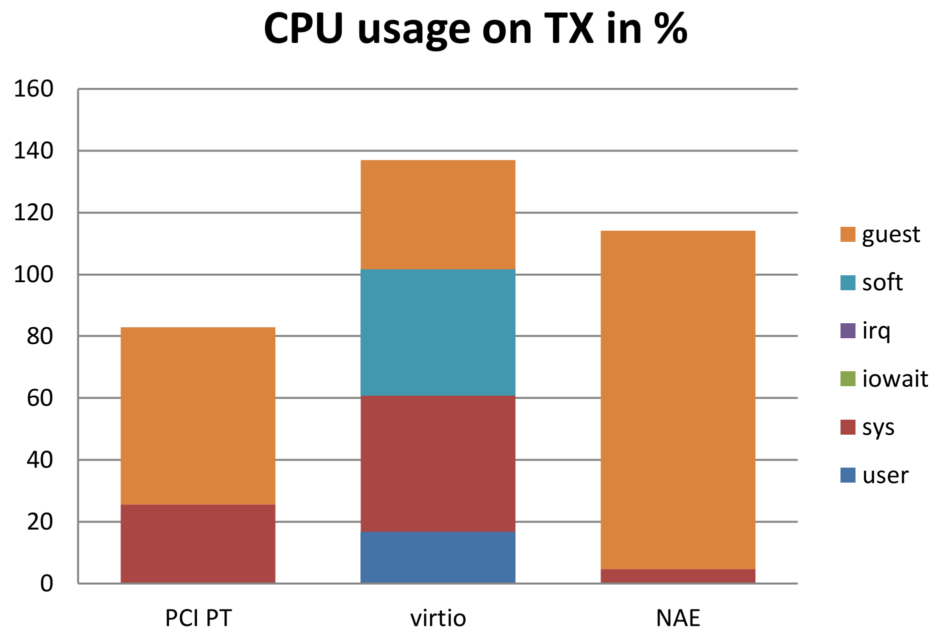


Figure 25 CPU Usage on TX

The PCI pass-through configuration shows the lowest CPU overhead. About 25% CPU resources are used by the system (as illustrated by "sys" in the figure) which indicates that this is processing occurring in kernel space. Kernel space processing here excludes hardware and software interrupt processing. These are listed explicitly as "irq" and "soft" resources. The PCI pass-through configuration introduces a significant amount of kernel processing. This mainly comes from the virtualization layer KVM and the kernel-based threaded IRQ handlers explained in previous sections. Even though that is code processing interrupts, it does not count as "irq" or "soft" time, because the kernel worker threads do not run in interrupt context or soft interrupt context. *ixgbevf* uses interrupts for completion notification on both the receive path and the transmit path, and therefore there are two kernel worker threads running all the time. On top of that PCI pass-through adds about 57% CPU utilization while running in guest mode (as indicated as "guest" in the figure). This overhead comes from the native device driver running inside the virtual machine processing packets.

The NAE framework uses about 4% CPU resources in kernel space which is the lowest of all approaches. That overhead comes from running the virtualization layer KVM. The NAE prototype runs guest code at about 109% CPU utilization which is the highest amount of guest code processing out of all approaches. The NAE guest device driver is very simple compared to a usual driver for a hardware-based network device. For example, it runs a simple data path as explained in the previous sections, and it does not need any hardware-specific control code, for example, to program firmware. Therefore, it is expected that running the NAE guest driver introduces low overhead. However, when transmitting packets it is optimized to re-fill the circular buffer as quick as possible and it can do that without waiting for the real I/O device to raise an interrupt indicating that it has successfully sent data. This is possible because each data descriptor is marked by hardware when it has been processed, so the NAE guest driver can simply look at descriptors instead of waiting for an interrupt. As the NAE guest driver is frequently doing this, the achieved throughput is very high, but the CPU load is also reasonably high. It is important to note though that the results here are showing an implementation difference rather than a significant design difference. With NAE, however, it is configurable through the virtual control path in what way the circular buffer is checked and re-filled. Different applications will have different preferences on this capability.

virtio shows the largest CPU overhead. About 16% is used as code is executed in user space. virtio runs the *qemu-io-thread* in user space, as explained in the previous section, which processes packets sent by the virtual machine and passes them to the kernel-based *tap* device which then transmits packets across the Linux bridge onto the physical network to *nae-server-2*. “sys” time is highest for virtio with about 44%. Most of that is due to additional network processing in the kernel when packets are passed through the *tap* device, the Linux bridge and the actual network device transmitting packets onto the physical network. virtio also shows a significant time spent in soft interrupt context (shown as “soft” time in the

figure): 40% CPU utilization is used when processing software interrupts. The high percentage here comes from the fact that a significant amount of packet processing in the network stack occurs in software interrupt context. On top of that virtio uses about 35% CPU for running guest code. This can be accounted to the virtio device driver in the virtual machine and the user virtual machine's network stack processing TCP packets. virtio implements the circular buffer containing data descriptors using shared memory between the QEMU device model and the device driver residing in the user virtual machine. The virtio guest OS driver uses writes to a memory-mapped I/O register to indicate to the other side when it has put data into the send queue. The driver can check about data having successfully been sent by reading from the shared memory region rather than waiting for a notification via interrupt.

Figure 26 shows CPU overhead the different approaches incur when receiving TCP bulk data streams. As before, the numbers shown in the figure are CPU load added up from both CPU cores, so the maximum percentage any workload can achieve is 200%.

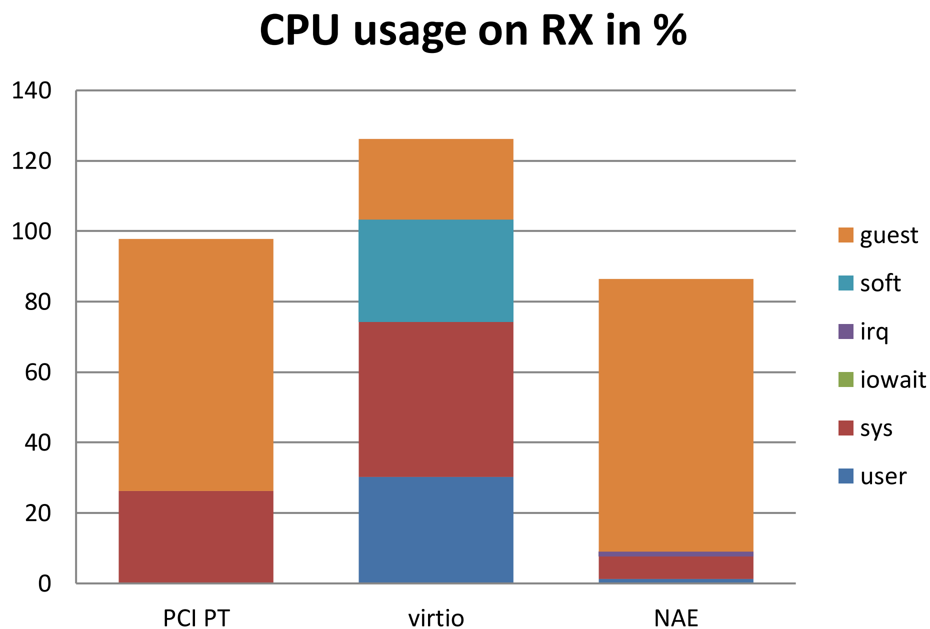


Figure 26 CPU Usage on RX

The PCI pass-through configuration uses about 26% of "sys" resources which indicates time spent processing data in kernel space. As on the transmit path, this includes running the

virtualization layer KVM and processing interrupts in the two previously introduced kernel-based worker threads. On top of that there are further 71% CPU load for running guest code which can be accounted to the native *ixgbevf* driver and the TCP network stack running in the virtual machine processing packets. The *ixgbevf* driver uses a mix of interrupt-based notification and polling on the receive path, as most new Linux network device drivers do. The Linux kernel provides an API for combining interrupts with polling on the receive path which is called NAPI (New API). The idea is that, by default, a device generates interrupts when incoming data has arrived that needs processing by the device driver. However, when too many interrupts are generated (which put a high load on the CPU), then the driver switches off interrupt generation on the device and changes into a polling mode. Polling is typically more efficient when there is a lot of data coming in. Once all data has been processed and there is not enough incoming bulk data to handle, the driver switches off polling mode and enables interrupts on the device again. It has been shown, for example in (Salah and Qahtan 2008), that this combination of interrupt-based notification and polling on the receive path achieves more efficient overall system performance by using less CPU resources during high-volume data transfers. All three evaluated approaches take advantage of the Linux NAPI infrastructure. The *ixgbevf* driver additionally implements a proprietary interrupt mitigation mechanism: it runs a timer constantly checking on hardware-based packet counters and the number of generated interrupts, and it changes the rate of interrupt generation on the device accordingly. For example, when the device is receiving a lot of bulk data, then it switches to a lower rate for interrupt generation. A lower rate is more efficient by creating less work on the CPU, but, on the other hand, it increases latency. For our TCP bulk data tests, the *ixgbevf* driver would mainly use polling, as virtio and also the NAE prototype guest OS driver do as well. However, due to head-of-line blocking issues in our configuration, the driver frequently switches back to interrupt-based notification when it

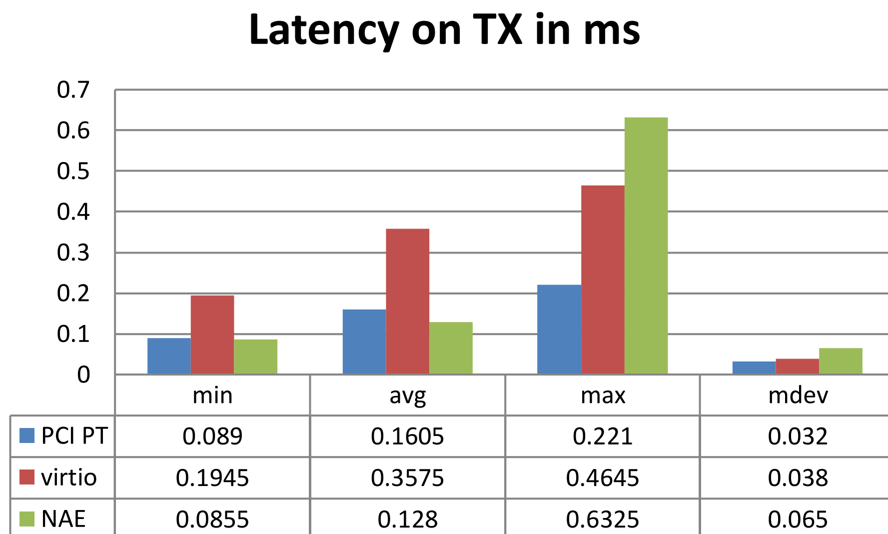
has processed all packets in the circular buffer and waits for the network stack to process TCP connections.

The NAE prototype uses about the same amount of CPU resources while running guest code: the “guest” mode uses about 71%. As with the native *ixgbevf* driver, this mainly accounts for packet processing in the guest OS device driver and TCP connection processing in the user virtual machine’s network stack. NAE only uses 6% of “sys” resources which is significantly lower kernel space processing than the PCI pass-through approach. This shows that on the data path the NAE architecture only imposes very little overhead in the network control domain (this is true for both the receive path and the transmit path), and the majority of the processing overhead can be accounted to the user virtual machine. This significantly simplifies accounting of CPU resources to virtual machines which makes it easier to implement Quality-of-Service policies on virtualized systems.

virtio shows the biggest overhead on the receive path. virtio uses 22% CPU utilization when running guest code, which is the lowest out of all three approaches, and about 44% CPU utilization when running kernel space code, which is the highest out of all approaches. As on the transmit path, packets received from the network need to be processed by the physical device driver in the network control domain at first, then they are passed to the kernel-based Linux bridge and then to the *tap* device which then sends packets up to the QEMU virtio device model running within the *qemu-io-thread* which then finally transmits packets to the user virtual machine. User space processing consumes 30% CPU utilization as shown in Figure 15 which indicates major overhead from running the *qemu-io-thread*. As on the transmit path, virtio shows again significant time spent in software interrupt context: on the receive path that shows as about 28% of CPU utilization which comes from significant overhead of processing occurring in the network control domain’s network stack.

6.4.3. Latency

Latency results are shown in Figure 27 and Figure 28. The results here indicate latency incurred on the network I/O path of the virtualized system on *nae-server-1* - this is where the three approaches differ when processing packets. Figure 27 shows latency measurements when sending packets while Figure 28 shows latency when receiving packets. In both cases, the NAE prototype implementation shows the lowest latency which indicates that the event notification mechanism of the NAE architecture is most efficient. Higher latency for the PCI pass-through implementation shows that the routing of interrupts from I/O devices to virtual machines is not ideal in today's most commonly used I/O device virtualization approach. For the NAE prototype, on the other hand, the achieved low latency numbers reveal that packet processing is fast and efficient. This proves that, even when generalizing the data path and deploying the NAE framework on non-optimized hardware, we can create a well-performing network I/O path to and from the virtual machine. The fast buffer re-fill / polling mechanism implemented as part of the NAE prototype probably contributes positively to our solution's low latency figures.



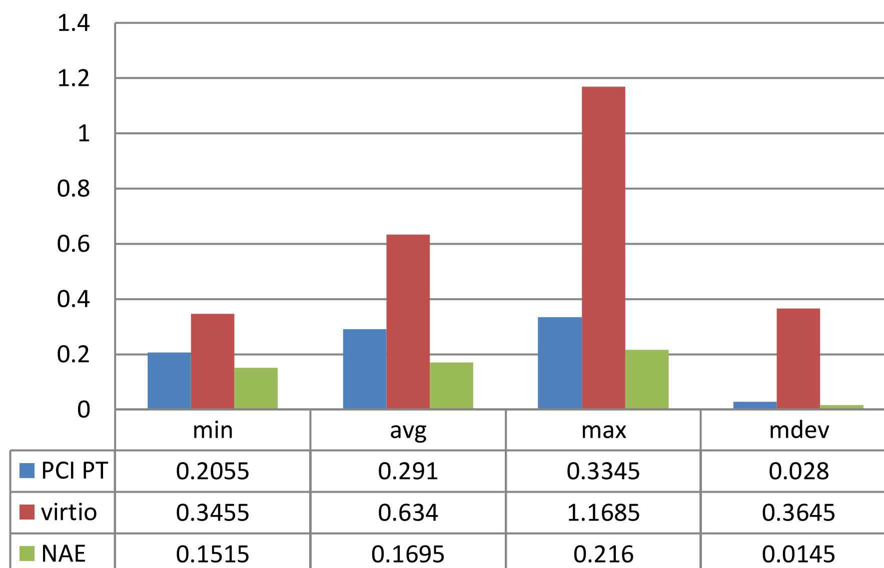
	min	avg	max	mdev
PCI PT	0.089	0.1605	0.221	0.032
virtio	0.1945	0.3575	0.4645	0.038
NAE	0.0855	0.128	0.6325	0.065

Figure 27 Latency on TX

As illustrated in Figure 27, virtio shows by far the highest average latency on the transmit path. The average round trip time is more than twice as high as when running the PCI pass-

through configuration, and nearly three times as high as when running the NAE prototype. virtio packet processing involves a significant number of components running in the network control domain, ranging from the device driver of the real I/O device, the Linux bridge, the *tap* device and the virtio QEMU device model. Having a larger number of components in the I/O path introduces latency. On top of this, the repeated context switches between user space and kernel space are also costly and therefore add to the delay on the I/O path. In these tests it seems unexpected that the NAE prototype actually shows the highest maximum latency. It is expected that this maximum value is not significant and is a result of a random single high latency measurement experienced during tests.

On the receive path, the NAE architecture shows 0.1695ms latency on average which is the lowest of all approaches, while PCI pass-through shows 0.291ms latency on average. virtio recorded a latency of 0.634ms. Here, as on the transmit path, the latency is introduced by having a larger number of packet processing components on the data path in the network control domain.



	min	avg	max	mdev
PCI PT	0.2055	0.291	0.3345	0.028
virtio	0.3455	0.634	1.1685	0.3645
NAE	0.1515	0.1695	0.216	0.0145

Figure 28 Latency on RX

Furthermore, on the receive path, virtio uses the Linux *eventfd* mechanism when the virtio QEMU device model wants to indicate to the virtual machine that new data (for example,

incoming network packets) has arrived which needs processing by the guest OS. Here the virtio QEMU device model running in user space uses an allocated event file descriptor (eventfd) to send a notification to KVM that an interrupt needs to be injected into the virtual machine. *eventfd* is a generic Linux user space to kernel space communication mechanism. The *eventfd* is processed by KVM using kernel-based worker threads. In summary, packet data is moved all the way from kernel space to user space and to the virtual machine, and an event notification then populates from user space to kernel space and then to the virtual machine. This process introduces a significant amount of delay for packet delivery. The analysis of latency in the virtio configuration clearly shows that a more direct I/O path between the network hardware and the virtual machine is required.

In summary our performance evaluation covers thorough tests of the two most important network metrics: throughput and latency. With these two metrics it is possible to derive how more complex applications running on a virtualized infrastructure using the presented approaches would perform. As indicated previously, the virtual data path can potentially be enhanced with more application-specific features to improve the performance of particular applications – even though that applies to all three presented approaches and therefore is not a great differentiator for any of the solutions. We have done a large enough number of test runs for all approaches to be confident of the reported performance numbers. The recorded values were stable and the environment was sufficiently restricted and the setup was simple. It is expected that a real world deployment will be more complex and potentially experience more challenges that we have not covered in our tests. However, we believe that the tests we have presented in this thesis are close enough to potential real world use cases to give a good first evaluation of the potential of the NAE I/O architecture and demonstrate that it can reach the performance of purely hardware-based direct device assignment mechanisms whilst maintaining hardware independence.

7. Contributions

In this work we have presented a new virtual I/O architecture spanning the complete system infrastructure. We have provided an I/O architecture covering enhancements to all critical components involved in processing network I/O on a virtualized platform. In particular, we proposed APIs for the hypervisor, the network control domain, the underlying network hardware and the guest operating system. The design of such a complete architecture that allows the implementation of an efficient virtualized I/O path is an important value-add for cloud infrastructure providers. The concept and design principles we have presented in this thesis help to emphasise that various system components need to be optimized in order to provide good I/O performance while providing a vendor-independent and device-independent interface to the virtual machine. Furthermore our work presented in this thesis makes clear that improving I/O virtualization needs to be looked at from an architecture point-of-view in order to enable an ideal solution. The ideal virtual I/O path demands the development of a new system architecture.

We have developed the complete virtualized system architecture in a prototype implementation. We did not implement a software-based virtual data path, but we implemented the hardware-based virtual data path, consisting of the complete I/O path from the network hardware, the hypervisor components, the network control domain components (vendor-specific control code and virtual device emulation), to the virtual machine including virtual machine device drivers. We implemented a complete hardware-based virtual data path and a complete software-based virtual control path. As explained previously, this is the most ideal configuration set-up a VM can get, because it provides good performance whilst maintaining independence of hardware device specifics. The implementation covering the complete virtual I/O path taking advantage of hardware-based acceleration capabilities is a significant contribution in itself, because it shows that such an I/O architecture can be implemented – even on current hardware not optimized for our

solution. The implementation furthermore allows evaluating the performance of our proposed approach and it allows estimating the required effort for network hardware vendors and system software developers to adopt our I/O architecture. Hence it is a significant contribution that needs to be recognized as one of the biggest and most time-consuming contributions of this thesis.

As part of this research work we have developed a simple, but very important, concept to be applied to virtualized network I/O: the separation of information flow into the two categories 'data' and 'control'. This concept itself is not new. But there exists no previous work that applies this principle as a fundamental and architectural building block on virtualized platforms and uses it consistently throughout the whole system. This separation of information flow is very important and leads to another contribution of our work: we create a virtualized I/O data path that can access a variety of network devices whilst providing a unified, generic I/O interface to virtual machines. The separation of information flow is significant in this context, because it allows the creation of a very narrow I/O interface that can be unified across different hardware devices more easily. The wider the interface, the harder it is to unify. This is simply because different features are implemented differently by hardware vendors. The analysis of the control path, which can be very complex, has shown this.

In this context, the most relevant engineering task of this work has been to find out exactly what is required on the I/O data path in order to transfer data between network hardware and virtual machines. This should be the minimum set of functions and data structures that need to be implemented to provide a well-performing virtual data path. We have shown that this minimum set can be unified across different vendors and different devices. In fact, as shown in our hardware evaluation, most vendors already implement this minimal set in the same way, or at least very similarly. This makes it very simple for us to provide a unified, generalized I/O data path running efficiently on top of a variety of network hardware. The

exact composition of this minimal virtual data path is presented in section 5.5. The definition of this minimal virtual data path is one of the major contributions of this work as it demonstrates how simple it can be to provide an efficient I/O data path that is vendor-independent and device-independent. It makes clear that it is realistic to build such an I/O architecture we propose, and the performance evaluation of our implementation shows that it is also very competitive from a performance point-of-view.

In this thesis we present an I/O virtualization framework that introduces changes to various components of the system architecture. If one wants to go ahead and implement a virtualized platform according to the complete Network Acceleration Engine I/O framework, then hardware and system software interfaces have to be adapted to conform to the principles we described above.

First of all, network hardware vendors potentially need to change how they design their devices in terms of allowing clear partitioning of device resources. In most cases, network hardware is already fulfilling this requirement to a very large extent. For example, many vendors already offer separate I/O channels (packet queues) that are designed to be usable independently and in parallel. However, as we have shown in section 4, many vendors have not implemented data path resources with sufficient isolation to be used in a virtualized system. Separation at hardware level is very important for critical data path resources, because they need to be made accessible directly to (untrusted) user virtual machines, so that the virtual data path can provide maximum performance through hardware acceleration. In summary, we can conclude that minor, but critical, changes have to be made to certain network devices in order to comply with the NAE API.

Secondly, the system architecture of a virtualized platform needs to be adapted to move vendor-specific control code to a (managed) network control domain, so that user virtual machines can run without any hardware dependency. Vendor-specific control code can never be completely removed from the system – it is required for any real-world

deployment. However, it is important to keep these pieces of control code out of the user virtual machine, because only then we can fulfil the requirements of dynamic cloud computing infrastructures. Vendor-specific control code already exists today (the “device driver”), but the NAE framework makes it significantly simpler, because the data path code sits in the user virtual machine. Furthermore, the NAE API puts vendor-specific control code under a common management API and integrates it more tightly with the virtualization layer (e.g. the hypervisor).

Thirdly, an system complying with the NAE API implements a software-based virtual data path and a hardware-based virtual data path that can be switched on-the-fly and transparently to the virtual machine. As mentioned previously, we define a very small set of data structures and functions to realize a high-performance hardware-based virtual data path. This is not only important for being able to unify the data path between different network hardware, but it is also very beneficial for implementing an efficient software-based virtual data path: the simplicity of our virtual data path design ensures that a virtual data path can be implemented with little effort. This significantly contributes to a feasible realization of our I/O architecture. The small set of data structures and functions we defined to develop a virtual data path furthermore allows smooth and efficient swapping of real resources backing up the VDP. These can be software-based and hardware-based. The type of data structures and functions we have selected to construct the VDP are generic enough to be implemented by software and a wide set of hardware devices. What is most important in this context is the fact that we have just a limited set of resources forming the virtual data path: this means that switching between hardware-based and software-based resources is reasonably simple, because only a small set of data (which has the same structure independently of whether it is hold in hardware or in software) has to be managed during a switch.

The simple and clear interface of the virtual data path can also potentially be very significant for virtual machine migration and its technical solution. We have not developed a working prototype system implementing virtual machine migration, but we think that, because the virtual data path only consists of a small set of data structures and functions, it can potentially be moved to a different physical machine more easily. It should be feasible to keep track of a small set of data structures and it should be efficient to migrate them across to another system.

8.

8. Conclusion

8.1. Achievements

We have shown that it is possible to provide a generalized, hardware-accelerated virtual I/O path that can match the performance of traditional direct device assignment approaches. The benefit of our proposed NAE I/O path is that VMs can take advantage of hardware-based network acceleration while maintaining sufficient hardware independence. Such a flexible, generalized I/O path design enables to offer important capabilities including mobility and portability of VMs across a heterogeneous set of hardware platforms. This is a critical solution enabler for cloud infrastructure providers working towards bringing HPC applications into the cloud.

Apart from demonstrating the feasibility of implementing a high-performance I/O path, we furthermore show that a common VDP design makes it possible to load-balance hardware resources more dynamically as we can easily and smoothly switch between a hardware-based VDP and a software-based VDP. No existing virtual I/O architecture can offer such a feature adding significant value to cloud environments where resources have to be continuously balanced.

In the following we list the main achievements of our work and how it contributes to advancing the state-of-the-art in the area of high-performance network I/O virtualization:

- We discovered and demonstrated that today's network hardware is not fully optimized yet for being used on virtualized systems. Our hardware design evaluation shows how current network devices differ and where they lack sufficient support to be used as network accelerator engines on virtualized systems within large-scale infrastructures.

- As a result of our evaluation, we outlined how hardware vendors can design interfaces that are more suitable for virtualized systems. The proposed NAE hardware API enables more efficient partitioning and sharing of hardware resources by providing a better suited hardware abstraction: we suggest modeling an I/O data channel based on simple, device-independent DMA-capable circular buffers as the main resource to provide a hardware-based, high-performance data path.
- We have developed a complete working prototype system implementing the NAE I/O architecture on top of existing hardware and conducted an extensive performance evaluation based on this prototype. The presented results show that the proposed network acceleration engine architecture performs similarly to existing direct I/O approaches, like PCI pass-through, and significantly better than existing, purely software-based I/O approaches.
- Lastly, our research reveals challenges of using today's hypervisor solutions for efficiently virtualizing network I/O. Here, our proposed network acceleration engine I/O architecture demonstrates that, in particular, it can be beneficial if the hypervisor provides a more flexible way of handling interrupts and memory-mapped I/O access to devices. While we implement these capabilities as an extension to the existing Linux-based hypervisor KVM, it would be desirable that hypervisors offer such a flexible virtual data path configuration as part of their core capabilities. Overall, we also conclude that today's hypervisors need to provide a more open, standardized API enabling other trusted, kernel-based components to get involved on the I/O path in an efficient manner.

The performance evaluation presented in the previous chapter outlined various aspects about how to best design a virtualized network I/O path in order to achieve high I/O performance. The major takeaways can be summarized with the following findings:

- Our proposed I/O architecture demonstrates that it can be more beneficial to provide a more flexible way of handling interrupts and memory-mapped I/O access to devices. In particular, a more flexible and optimized feature configuration such as, for example, interrupt handling and fast circular buffer re-fill which can be tuned for each individual network device can significantly improve performance.
- While allowing more flexibility on the virtual data path is a key design principle of the network acceleration engine architecture, our performance evaluation furthermore proves that efficient network processing requires a more direct I/O path between hardware and virtual machines. This is something that purely software-based approaches cannot provide and therefore they struggle to keep up with hardware-accelerated I/O path designs. Ideally, the virtualized system architecture needs to be able to provide as few components as possible on the I/O data path. This is particularly important when the application running inside the user virtual machine requires low-latency networking. On top of that, the design of the I/O data path needs to enable packet transmission and reception with as few context switches as possible. This affects both types of context switches: (1) between user space and kernel space, and (2) between network control domain and user virtual machine – even though context switches of type (2) are significantly more expensive than of type (1).
- Even though we completely remove hardware dependencies from the virtual machine, the main network processing overhead can be accounted to the virtual machine itself, the device driver and the network stack running inside the user virtual machine. For cloud computing infrastructures (or, in general, for multi-tenant platforms running over a shared physical infrastructure) it is very important that the utilization of compute resources can be clearly accounted for and charged to the correct individual user. The more processing happens inside the network control

domain, where resources are shared, the more difficult it is to clearly separate out which virtual machine needs to be charged for using CPU and memory resources. For example, when network processing happens in the Linux bridge in the network control domain, it is difficult to calculate how much time is spent on packet forwarding for each open network connection. With the NAE architecture, on the other hand, the majority of processing overhead is incurred inside the virtual machine, and we can simply charge for time spent running the virtual machine plus resources used on the real I/O device which is exposed to the virtual machine as network accelerator. A design like this enables simple, fair resource sharing algorithm on the virtualized system.

Overall, we believe that this work contributes significantly to enabling high-performance network I/O in virtualized, large-scale infrastructures. Our initial analysis revealed that existing approaches which tie the virtual machine to a particular vendor-specific and device-specific hardware interface are not suitable for large-scale cloud computing infrastructures. In order to take full advantage of high-performance network I/O devices in these environments, a new approach needs to be designed and implemented. Our hardware and system software analysis makes clear that there are flaws in both hardware interface and system software design which result in a virtualized platform design where high-performance I/O devices can only be achieved when restricting virtual machine mobility and portability. We have designed and implemented the network acceleration engine I/O architecture to overcome exactly these challenges. Our suggested new hardware API provides unified access to a variety of network devices. We believe that with this work we can significantly simplify the use of specialized, powerful network hardware across large-scale, cloud-based environments.

8.2. Future Work

While we implemented a complete system architecture enabling a vendor-independent virtualized network I/O path, we have not yet tackled the implementation of all hooks required for virtual machine live migration. Our generalized, hardware-independent design forms a major building block for virtual machine migration capabilities. However, there is further work required, mainly in the NAE Guest API and the NAE Network Control Domain API, to allow taking a snapshot of the virtual device state at any time, move it across to a new platform, and then be able to re-instantiate the virtual device again on the new platform without losing any data that might be currently in flight as the virtual machine might have several network connections open.

The performance evaluation we present is based on a platform running a single user virtual machine, and it would be valuable to furthermore show how the proposed NAE I/O architecture performs when running multiple user virtual machines concurrently. We give some indication of scalability already in our results section – the design is expected to scale well over a larger number of user virtual machines sharing a fixed number of real I/O devices. However, running performance tests over a larger-scale set up might reveal further challenges in critical components like the hypervisor and potentially also the network control domain's operating system. Furthermore, real I/O devices might have scalability limitations themselves when partitioning their resources that we have not yet shown in the evaluation of our prototype.

We show the benefits of a hardware-accelerated virtual data path and a software-based virtual control path. The proposed network acceleration engine API design furthermore covers running a software-based virtual data path and we describe a set of use cases outlining why providing a software-based virtual data path is essential for deploying the NAE framework in a real world scenario and in particular in large-scale cloud computing infrastructures. In this context, one area requiring further research is the development of a

set of policies which can be implemented in the network control domain in order to decide whether a virtual machine gets a hardware-accelerated VDP or a software-based VDP. As mentioned previously, one can think of a variety of policies based on fair sharing of available hardware resources, or potentially users can be charged more for having a dedicated hardware-accelerated VDP. Additionally, the implementation of fast switching between software-based VDP and hardware-based VDP might bring up further challenges we have not yet analyzed in detail. In particular, the actual VDP implementation needs to be changeable while the user virtual machine has active network connections established. A majority of the mechanisms used here will overlap with what is also required in order to enable virtual machine live migration, as described above.

The work presented in this thesis focuses on network I/O. We have, on purpose, kept such a narrow focus in order to build an I/O path that is efficient and optimized for the traffic that is passing through it. Also, such a narrow focus kept the implementation effort feasible and made sure that we could sufficiently evaluate the performance of our new I/O architecture. However, for future investigations it would be interesting to see how the NAE I/O architecture can be applied to other types of I/O transfers like, for example, storage I/O or even graphics I/O. It is expected that the data path of storage I/O devices can be unified in the same way as we have unified it across network devices. Unifying access to the data path of graphics hardware might be harder to accomplish as hardware interfaces of graphics devices are often proprietary and only closed-source software binaries are available to access them. Each type of I/O device will need a full hardware design analysis and a feasibility study to demonstrate the applicability of something like the NAE I/O architecture.

9. Bibliography

Abel, Gordon, et al. “ELI: bare-metal performance for I/O virtualization.” *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems.* New York, NY: ACM, 2012. 411-422.
Abramson, Darren, et al. “Intel Virtualization Technology for Directed I/O.” *Intel Technology Journal* (Intel) 10, no. 03 (August 2006).
Advanced Micro Devices, Inc. *AMD I/O Virtualization Technology (IOMMU) Specification.* Advanced Micro Devices, Inc., 2009.
AMD. *Live Migration with AMD-V Extended Migration Technology.* Sunnyvale, CA: AMD, 2007.
AMD Virtualization . AMD. http://sites.amd.com/us/business/it-solutions/virtualization/Pages/virtualization.aspx (accessed 2011 йил 16-November).
Anderson, Melvin J., Micha Moffie, and Chris I. Dalton. “Towards Trustworthy Virtualisation Environments: Xen Library OS Security Service Infrastructure.” Hewlett-Packard Development Company, L.P., 2007.
Auernhammer, Florian, and Patricia Sagmeister. “Architectural support for user-level network interfaces in heavily virtualized systems.” *Proceedings of the 2nd conference on I/O virtualization.* Pittsburgh, PA, USA: USENIX Association, 2010.
Barham, Paul, et al. “Xen and the Art of Virtualization.” *19th ACM Symposium on Operating Systems Principles.* Bolton Landing, New York, USA: ACM, 2003.
Bellard, Fabrice. “QEMU, a Fast and Portable Dynamic Translator.” *Proceedings of USENIX Annual Technical Conference.* USENIX Association, 2005. 41–46 .
Ben-Yehuda, Muli, et al. “Utilizing IOMMUs for Virtualization in Linux and Xen.” Ottawa, Canada: Linux Symposium, 2006.
Bradford, Robert, Evangelos Kotsovinos, Anja Feldmann, and Harald Schioeberg. “LiveWide-Area Migration of Virtual Machines Including Local Persistent State.” *3rd international conference on Virtual execution environments 2007.* San Diego, CA: ACM, 2007. 169 - 179 .
Chen, Gary P., and Jean S. Bozman. *Optimizing I/O Virtualization: Preparing the Datacenter for Next-Generation Applications.* Framingham, MA, USA: IDC, 2009.
Chinni, Shefali, and Radhakrishna Hiremane. *Virtual Machine Device Queues.* Intel Corporation, 2007.
Chinni, Shefali, and Radhakrishna Hiremane. *Virtual Machine Device Queues.* Intel Corporation, 2007.
Dong, Yaozu, Xiaowei Yang, Xiaoyong Li, Jianhui Li, Kun Tian, and Haibing Guan. “High Performance Network Virtualization with SR-IOV.” Bangalore, India: IEEE, 2009.
Dong, Yaozu, Yunhong Jiang, and Kun Tian. “SR-IOV support in Xen.” *Xen Summit 2008.* Boston, MA, 2008.
Dong, Yaozu, Zhao Yu, and Greg Rose. “SR-IOV Networking in Xen: Architecture, Design and Implementation.” *First Workshop on I/O Virtualization .* San Diego, CA, USA: USENIX Association, 2008.
Dowty, Micah, and Jeremy Sugerman. “GPU Virtualization on VMware’s Hosted I/O Architecture.” *ACM SIGOPS Operating Systems Review* (ACM) 43, no. 3 (2009).
Fraser, Keir, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. “Safe Hardware Access with the Xen Virtual Machine Monitor.” *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS).* 2004.
git repository of Linux net-next development tree. http://git.kernel.org/?p=linux/kernel/git/davem/net-next.git;a=summary (accessed 2013 йил 14-January).

Goodacre, John. “Hardware accelerated Virtualization in the ARM Cortex Processor.” *Xen Summit Asia.* Seoul, Korea, 2010.
Gordon, Abel, Nadav Har'El, Alex Landau, Muli Ben-Yehuda, and Avishay Traeger. “Towards Exitless and Efficient Paravirtual I/O.” *The 5th Annual International Systems and Storage Conference.* Haifa, Israel: ACM, 2012.
Huang, Shu, and Ilia Baldine. “Performance evaluation of 10GE NICs with SR-IOV support: i/o virtualization and network stack optimizations.” *Proceedings of the 16th international GI/ITG conference on Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance .* Springer-Verlag Berlin, Heidelberg, 2012. 197-205.
Inc., VMware. *VMware vMotion.* VMware Inc. https://www.vmware.com/products/datacenter-virtualization/vsphere/vmotion.html (accessed 2012 йил 10-September).
—. *VMware vSphere.* VMware Inc. 2012. http://www.vmware.com/products/datacenter-virtualization/vsphere/overview.html (accessed 2012 йил 10-September).
—. *VMware Workstation.* VMware Inc. 2012. http://www.vmware.com/products/workstation/overview.html (accessed 2012 йил 10-September).
Intel. *Intel Virtualization Technology for Directed I/O Architecture Specification Rev 1.3.* Intel, 2011.
Intel Virtualization Technology (Intel VT). Intel. http://www.intel.com/technology/virtualization/technology.htm (accessed 2011 йил 16-November).
Kadav, Asim, and Michael M. Swift. “Live Migration of Direct-Access Devices.” *ACM SIGOPS Operating Systems Review* (ACM) 43, no. 3 (July 2009).
Kimball, Brady. *The State of HPC Cloud.* 2012 йил 11-September. http://www.hpcinthecloud.com/hpccloud/2012-09-11/the_state_of_hpc_cloud.html (accessed 2012 йил 21-September).
Kivity, Avi, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. “kvm: the Linux Virtual Machine Monitor.” Ottawa, Ontario: Linux Symposium, 2007.
Lane, Katherine, and Jolene Bonina. *Emulex Survey Reveals I/O Perfect Storm.* Emulex. 2012 йил 25-September. http://www.emulex.com/company/media-center/press-releases/2012/sep25/print.html (accessed 2012 йил 25-September).
Le, Michael, Andrew Gallagher, Yuval Tamir, and Yoshio Turner. “Maintaining Network QoS Across NIC Device Driver Failures Using Virtualization.” *8th IEEE International Symposium on Networ k Computing and Applications.* Cambridge, MA, 2009.
LeVasseur, Joshua, et al. “Standardized but Flexible I/O for Self-Virtualizing Devices.” *First Workshop on I/O Virtualization (WIOV).* San Diego, CA, USA: USENIX, 2008.
Levasseur, Joshua, Volkmar Uhlig, Jan Stoess, and Stefan Götz. “Unmodified device driver reuse and improved system dependability via virtual machines.” *Proceedings of the 6th Symposium on Operating Systems Design and Implementation.* 2004. 17-30.
Liu, Jiuxing. “Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support.” *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010: 1-12.
—. “Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support.” *2010 IEEE Symposium on Parallel & Distributed Processing (IPDPS), .* Atlanta, GA, USA, 2010. 1-12.
Magenheimer, Dan, Chris Mason, Dave McCracken, and Kurt Hackel. “Paravirtualized Paging.” *First workshop on I/O virtualization WIOV.* Berkeley, CA, USA: USENIX Association, 2008.
Mei, Yiduo, Ling Liu, Xing Pu, and Sankaran Sivathanu. “Performance Measurements and Analysis of Network I/O Applications in Virtualized Cloud.” *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing .* IEEE Computer Society , 2010. 59-66.

Mijat, Roberto, and Andy Nightingale. *Virtualization is Coming to a Platform Near You.* Whitepaper, ARM Limited, 2011.
Nakajima, Jun, and Daniel Stekloff. “Improving HVM Domain Isolation and Performance.” *Xen Summit.* 2006.
Nelson, Michael, Beng-Hong Lim, and Greg Hutchins. *Fast Transparent Migration for Virtual Machines.* Palo Alto, CA: VMware Inc., 2005.
Oracle. *Oracle VM VirtualBox.* Oracle. https://www.virtualbox.org/ (accessed 2012 йил 21-September).
ozlabs.org. *Lguest: The Simple x86 Hypervisor.* ozlabs.org. http://lguest.ozlabs.org/ (accessed 2012 йил 10-10).
Pan, Zhenhao, Yaozu Dong, Yu Chen, Lei Zhang, and Zhijiao Zhang. “CompSC: Live Migration with Pass-through Devices.” *Workshop on Virtual Execution Environments (VEE'12).* London: ACM, 2012.
PCI-SIG. *PCI-SIG I/O Virtualization (IOV) Specification, Single Root IOV.*
Pfaff, Ben, Justin Pettit, Teemu Koponen, Keith Amidon, Martin Casado, and Scott Shenker. “Extending networking into the virtualization layer.” *8th ACM Workshop on Hot Topics in Networks (HotNets-VIII).* New York City, New York: ACM, 2009.
Ram, Kaushik Kumar, Jose Renato Santos, Yoshio Turner, Alan L. Cox, and Scott Rixner. “Achieving 10 Gb/s using Safe and Transparent Network Interface Virtualization.” *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE).* Washington, DC.: ACM, 2009.
Rauchfuss, Holm, Thomas Wild, and Andreas Herkersdorf. “A network interface card architecture for I/O virtualization in embedded systems.” *Proceedings of the 2nd conference on I/O virtualization* (USENIX Association), 2010.
Rubens, Paul. *The Growing Pool of I/O Virtualization Technology Options.* serverwatch.com. 2010 йил 23-March . http://www.serverwatch.com/trends/article.php/3872371/The-Growing-Pool-of-IO-Virtualization-Technology-Options.htm (accessed 2013 йил 18-January).
Russell, Rusty. *Virtio PCI Card Specification v0.9.4 DRAFT.* 2012 йил 7-February.
Russell, Rusty. “virtio: towards a de-facto standard for virtual I/O devices.” *ACM SIGOPS Operating Systems Review - Research and developments in the Linux kernel* 42, no. 5 (2008): 95-103.
Salah, K., and A. Qahtan. “Experimental performance evaluation of a hybrid packet reception scheme for Linux networking subsystem.” *International Conference on Innovations in Information Technology (IIT).* Al Ain, 2008. 84-88.
Santos, Jose Renato, G. John Janakiraman, Yoshio Turner, and Ian Pratt. “Netchannel 2: Optimizing Network Performance.” *Xen Summit.* 2007.
Santos, Jose Renato, Yoshio Turner, G John Janakiraman, and Ian Pratt. “Bridging the Gap between Software and Hardware Techniques for I/O Virtualization.” *USENIX Annual Technical Conference.* Boston: USENIX Association, 2008.
Scharf, Michael, and Sebastian Kiesel. “Head-of-Line Blocking in TCP and SCTP: Analysis and Measurements.” *Globecom 2006.* San Francisco, CA, USA, 2006.
Sugerman, Jeremy, Ganesh Venkitachalam, and Beng-Hong Lim. “Virtualizing I/O Devices on VMware Workstation’s Hosted Virtual Machine Monitor.” Boston, Massachusetts: USENIX Association, 2001.
Swift, Michael M., Brian N. Bershad, and Henry M. Levy. “Improving the reliability of commodity operating systems.” *Proceedings of the nineteenth ACM symposium on Operating systems principles.* Bolton Landing, NY, USA: ACM, 2003. 207-222.
Travostino, Franco, et al. “Seamless Live Migration of Virtual Machines over the MAN/WAN.” *Future Generation Computer Systems - IGrid 2005* (Elsevier Science Publishers B. V.) 22, no. 8 (2006): 901-907.

Tseng, Hui-Min, Hui-Lan Lee, Jen-Wei Hu, Te-Lung Liu, Jee-Gong Chang, and Wei-Cheng Huang. “Network Virtualization with Cloud Virtual Switch.” *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems .* Washington, DC: IEEE Computer Society, 2011. 998-1003 .
Universal TUN/TAP device driver. kernel.org. http://www.kernel.org/doc/Documentation/networking/tuntap.txt (accessed 2013 йил 14-January).
Virtualization Extensions. ARM. http://www.arm.com/products/processors/technologies/virtualization-extensions.php (accessed 2011 йил 16-November).
VMware Inc. VMware Inc. http://www.vmware.com/ (accessed 2012 йил 21-September).
VMware Inc. *Configuration Examples and Troubleshooting for VMDirectPath.* Palo Alto, CA: VMware Inc. , 2009.
VMware Inc. *VMware Virtual Networking Concepts.* Palo Alto, CA: VMware Inc., 2007.
Voorsluys, William, James Broberg, Srikumar Venugopal, and Rajkumar Buyya. *Cost of Virtual Machine Live Migration in Clouds: A Performance Evaluation.* cloudbus.org, 2009.
Willmann, Paul, Scott Rixner, and Alan L. Cox. *Protection Strategies for Direct Access to Virtualized I/O Devices.* Boston, Massachusetts: USENIX, 2008.
Worley, Steve. *Effect of SR-IOV Support in Red Hat KVM on Network Performance in Virtualized Environments.* IBM, 2010.
Wun, B., and P. Crowley. “Network I/O Acceleration in Heterogeneous Multicore Processors.” *14th IEEE Symposium on High-Performance Interconnects*, 2006: 9-14.
Xia, Lei, Jack Lange, and Peter Dinda. “Towards virtual passthrough I/O on commodity devices.” *First conference on I/O virtualization WIOV.* Berkeley, CA: USENIX Association, 2008.
Youseff, Lamia, Rich Wolksi, Brent Gorda, and Chandra Krintz. “Paravirtualization For HPC Systems.” *International conference on Frontiers of High Performance Computing and Networking .* Berlin: Springer, 2006. 474-486.
Zhang, Binbin, et al. “A Survey on I/O Virtualization and Optimization.” *2010 Fifth Annual ChinaGrid Conference (ChinaGrid)*, 2010: 117-123.