

Project Report

Offloading Middlebox Services To Hypervisors For Efficient Network Function Virtualization

submitted in partial fulfilment of the requirements

of the degree of

Master of Technology

By:
Mihir J. Vegad
143050073

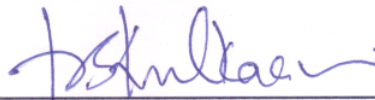
Supervisor:
Prof. Purushottam
Kulkarni



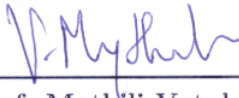
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
2016

Dissertation Approval

This dissertation entitled “Offloading Middlebox Services To Hypervisors For Efficient Network Function Virtualization”, submitted by Mihir J. Vegad (Roll No: 143050073) is approved for the degree of Master of Technology in Computer Science and Engineering from Indian Institute of Technology Bombay.



Prof. Purushottam Kulkarni
Dept. of CSE, IIT Bombay
Project guide



Prof. Mythili Vutukuru
Dept. of CSE, IIT Bombay
Examiner



Prof. Varsha Apte
Dept. of CSE, IIT Bombay
Examiner



Chairperson

Declaration of Authorship

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission.

I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Date: 24/06/2016

Place: IIT Bombay, Mumbai



Mihir J. Vegad

Roll No: 143050073

Acknowledgement

I would like to thank my guide, **Prof. Purushottam Kulkarni** for giving me the opportunity to work in this field. I really appreciate the efforts which he put in throughout the project, to understand the work done by us and then to guide us to the next step. During this process, I learned a lot and overall it has created strong base for me in the field of NFV/SDN/Virtualization. I would also like to thank fellow SYNERG mates for extending their support whenever it was required.

Abstract

In a set up where a server hosts several guest machines with the help of the hypervisor, an incoming packet on the server generates two interrupts to the processor. First interrupt is to sort the packet among the guests and the second interrupt is to deliver the packet to the guest. In high speed networks, a guest often cannot process the packet close to the line rate due to these interrupts. For example, in a 10Gbps network, throughput achieved at the guest is 4Gbps at maximum. Some optimizations are suggested as a solution for the problem by reducing the intervention of the hypervisor in the packet processing for guests. But those optimizations are costly to implement and they also come at some cost to scalability and portability of the guests. For example, SR-IOV[1] needs special hardware support and it binds the guests to the physical machine. DPDK[2] is a software based solution but it needs to modify the guest operating system. Our proposed solution can work as software extensions on any guest and host platform. In this report, we show how we can specialise the hypervisor for a middlebox such that most of the traffic for the middlebox bypasses the actual middlebox processing as it gets processed at the hypervisor only. We improve the average CPU utilization by almost 75 percentage for the firewall middlebox and almost 50 percentage for the load balancer middlebox. These results suggest that in a high speed network, our design can stop the servers from getting saturated and allows the guests to process the packets close to the line rate. Thus we improve the overall I/O virtualization performance of the middleboxes.

Contents

References	1
1 Introduction	1
1.1 Middleboxes	1
1.2 Hypervisors	2
1.3 Motivation	3
1.4 Problem Description	3
1.5 Organization Of The Work	4
1.6 Outline Of The Report	4
2 Background	5
2.1 Network I/O Path In Virtualized Network	5
2.2 Middlebox Examples	6
2.3 Middlebox Classification	8
3 Related Work	9
3.1 Middlebox Operating System Optimizations	9
3.2 Middlebox Hypervisor Optimizations	10
3.3 Middlebox Hardware Assisted Optimizations	11
3.4 Comparison Of All The Approaches	11
4 Design of Middlebox Specific Hypervisor Extensions	13
4.1 Overall Design	13
4.2 Implementation	14
4.2.1 Shared Memory	14
4.2.2 Middlebox Side (Guest) Interface	14
4.2.3 Hypervisor Side (Host) Interface	15
4.2.4 Middlebox State Stored At The Hypervisor	16
4.2.5 Offloaded Middlebox Functionalities	16
4.3 Different Packet Traversal Paths Analysis	17
5 Experimental Evaluation	18
5.1 Experimentation Setup	18
5.2 Experiment Details	18
5.3 Overall Cost Of Specialized I/O Virtualization	18
5.3.1 Analysis Of Network Interrupts	19
5.3.2 Analysis Of Packet Processing Time	19
5.3.3 Analysis Of CPU Utilization	21
5.4 Challenges	22
6 Conclusion	24

List of Figures

1.1	MB presence in the data centres [3]	1
1.2	Different types of Hypervisor	2
1.3	KVM-QEMU combination virtualization	2
1.4	Offloaded middlebox functionalities inside the Hypervisor	3
2.1	packet path from pNIC to middlebox	5
2.2	virtio network device architecture	6
2.3	Different types of middleboxes in data centres [3]	7
4.1	Design Of Middlebox Specialized Hypervisor	14
4.2	Different packet traversal paths	17
5.1	No. of interrupts generated for 875 incoming UDP packets of size 1500 bytes at 100Mbps line rate out of which 12 packets got dropped	19
5.2	Firewall Middlebox	20
5.3	Load Balancer Middlebox	20
5.4	Firewall Middlebox	21
5.5	Load Balancer Middlebox	22

List of Tables

2.1	Middlebox classification based on the packet processing	8
3.1	Comparision between different I/O virtualization techniques for middleboxes . .	12
4.1	Acronyms used for the middlebox commands	15
4.2	Middlebox commands for Firewall / Load balancer services	15

Chapter 1

Introduction

One of the key factors in the rise of the data center networking in recent years is, the core system infrastructure such as storage, network devices have become software-defined. Various network functions such as firewall, load balancer, gateway, proxies were used to installed as hardware elements in the traditional networks. These network functions are usually referred as Middleboxes. Now, administrators use Network Function Virtualization which decouples the network functions from proprietary hardware and virtualize them. To run middleboxes on the virtual machines, we require a Virtual Machine Monitor or Hypervisor which allows multiple commodity virtual machines to share conventional hardware in a safe and resource managed manner. In the era of virtualization and Software Defined Networking, Data centres are being flooded with various middleboxes[3] and we want to virtualize them efficiently without compromising the overall performance or resource utilization.

1.1 Middleboxes

By definition a middlebox [4] is a networking function that transforms, inspects, filters or manipulates network traffic for some purpose other than packet forwarding. NFV has attracted many service providers to virtualize their data center networks. Middleboxes are crucial part of such data centre networks as only packet forwarding is not good enough to meet the customer requirements, other functionalities like Quality of Service/Quality of Experience, load balancing, security are needed to make customer experience complete. This fact is also supported by a survey done over 57 enterprise data centres, whose size varied from less than 1K hosts to more than 100K hosts. [3] The survey shows that number of middleboxes deployed in data centres are on par with number of routers and number of switches in the data centre. WAN optimizers, proxies, application layer gateways, load balancers, intrusion detection system, intrusion prevention system are widely used middleboxes in the data centres. Middleboxes play an important role in meeting the service level agreements(SLAs) with the clients. All the middleboxes perform some entry level task followed by specific middlebox function followed by packet forwarding. Their presence in large number in data centres indicates that overall performance of a data centre is very much related to the performance of the middleboxes.

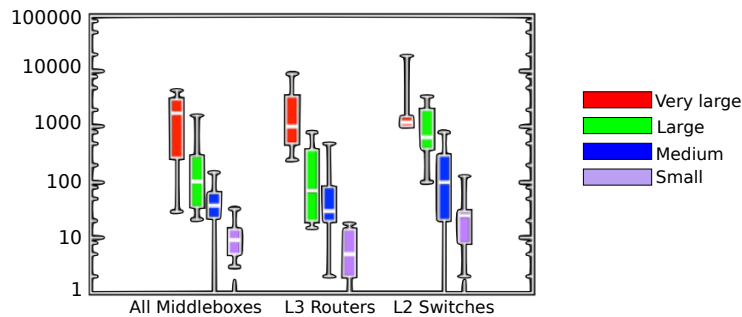


Figure 1.1: MB presence in the data centres [3]

1.2 Hypervisors

The aim of virtualization is to run many virtual machines on a single physical machine efficiently. Hypervisors are softwares that create or destroy virtual machines and manages physical resources among them. Two different types of hypervisors are used for virtualization: Bare metal hypervisor and Hosted hypervisors. Bare metal hypervisor runs directly on the hardware and manages virtual machines. Hosted hypervisor runs as an application on the operating system, usually known as host operating system. Oracle VM Server for x86, Xen server, KVM, Hyper-v are examples of bare metal hypervisors. Vmware workstation/player, virtual box are example of hosted hypervisors. We will focus on Bare metal hypervisors (Type-1) as they are widely used in data centres. A hypervisor mainly provides virtual hard disk, virtual network interface card, and life cycle management facility to virtual machines. We will discuss what more a hypervisor can offer to middleboxes, apart from this traditional stuff. Mostly used hypervisors

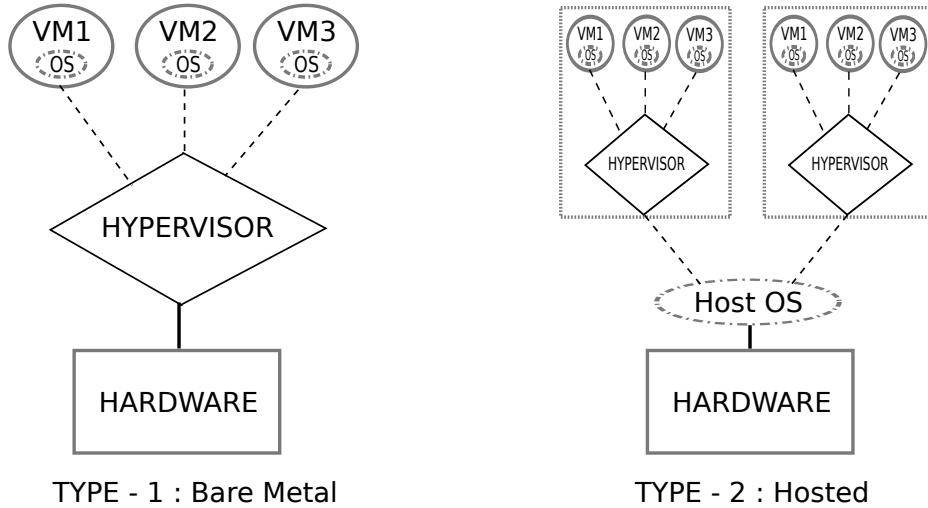


Figure 1.2: Different types of Hypervisor

in data centre virtualization are vSphere from VMware, Xenserver from Citrix, hyper-v from Microsoft and KVM from Redhat. Different hypervisors are used as per the requirements of the data centre administrator. We will focus on Kernel Virtual Machine(KVM) as a hypervisor in our experiments. KVM virtualizes the processor and the memory of the system. QEMU is a 'Quick Emulator' which is used to arbitrate hardware resources such as disk and network interface card. QEMU is a type-2 hypervisor by itself, it executes the virtual CPU instruction using host operating system on the physical CPU. But when QEMU combines with KVM, the combination becomes type-1 hypervisor. In that combination, QEMU is used as a virtualizer, and it achieves near native performance by executing guests code directly on the hardware.

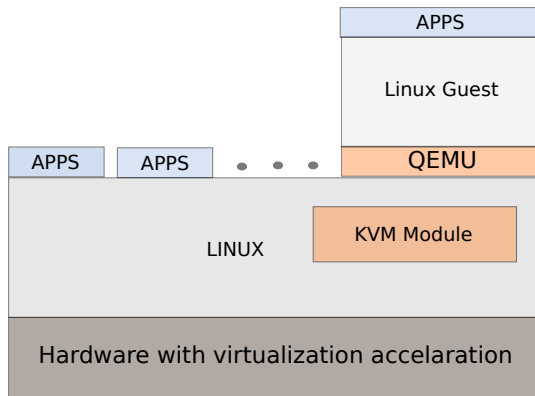


Figure 1.3: KVM-QEMU combination virtualization

1.3 Motivation

Virtualization aims to maximize the resource utilization at the cost of some added processing by the hypervisor for virtual machines. As we have seen earlier, Any virtual machine doing any I/O requires intervention from the hypervisor. A designated CPU core has to handle these interventions every time. After this, a designated CPU core handling a specific virtual machine will take care of the rest of the processing. On a high speed network when all the CPU cores are busy, the above mentioned processing takes toll on CPUs. In such a situation, somehow we need to reduce the CPU overhead to get maximum performance possible. Some hardware optimization techniques like Intel virtual machine device queue(VMDq)[5], Direct I/O[6], SR-IOV[1] exist to counter the intervention by hypervisor in high speed networks. These techniques improve network I/O to provide throughput close to the line rate. To enable these services dedicated hardwares are needed. Intel VMDq service is supported by only few high-end Intel Ethernet controller cards[5]. Due to this fact, they face scalability issues. They also have portability issues. For example, SR-IOV binds the virtual machines to the physical machines. DPDK[2] provides software based solution to the problem but it needs to modify the guest operating system and it requires DPDK libraries to be installed on the machine. These observations motivated us to come up with some hardware independent and easy to use solution which improves the overall network I/O for middlebox virtualization.

1.4 Problem Description

We suggest that when the hypervisor intervenes for the first time in the incoming packet processing, it should be intelligent enough to decide whether further processing by a virtual machine(middlebox) is required or not. A middlebox installed in the high speed network do not be require to process every incoming packet if the hypervisor does some pre-programmed processing to bypass the middlebox. By doing this, We aim at reducing the no. of interrupts to the CPU core serving the specific middlebox and also some processing for the first interrupt. Here, we propose a design for middlebox virtualization in which part of the middlebox functionality is offloaded to the hypervisor as shown in the figure-1.4.

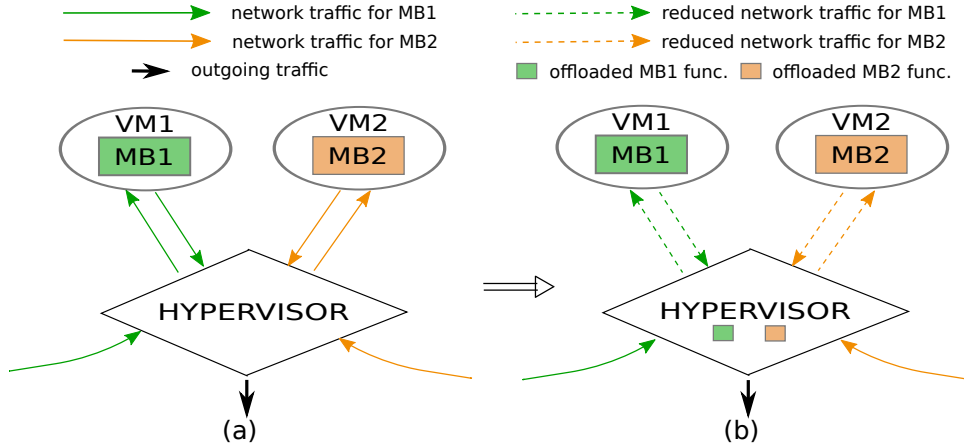


Figure 1.4: Offloaded middlebox functionalities inside the Hypervisor

Part (a) of the figure-1.4 shows VM1 and VM2 processes the network traffic respectively for middlebox1 and middlebox2 which comes through the hypervisor. In part (b), we can see that some offloaded middlebox1/middlebox2 functionality reside in the hypervisor. As a result of that, a portion of the incoming traffic is processed at the hypervisor and only remaining traffic is forwarded to the middleboxes. In chapter 4 and 5, we will discuss the design and the implementation aspects of the proposed solution in detail.

1.5 Organization Of The Work

1. Explored various types of middleboxes and their presence in data centres. Explored virtualization and hypervisor. Specifically, explored KVM as a hypervisor in detail.
2. Explored research done so far to optimize middlebox performance while running on physical machine and compared them based on different metrics. Built the problem definition based on the observations made in this study.
3. Proposed a possible solution to solve the problem, designed different components of the solution and analysed the challenges faced.
4. Implemented part of widely used middleboxes like firewall, load balancer, implemented interface between a middlebox and a hypervisor for communication. Measured several performance metrics by doing experimental evaluation of the system.
5. Documented the work done so far, results achieved and the future work possible.

1.6 Outline Of The Report

The report is organized in the following manner. In chapter 2, we will discuss several classifications of the middleboxes and network I/O path in detail. In chapter 3, we will discuss related work done in this field and compare them on several metrics to see how it relates to our approach. In chapter 4, we will discuss design of a possible solution, cost analysis of the suggested design. In chapter 5, we will discuss the implementation details for a possible solution, experimentation setup, results obtained and the challenges faced. At last in chapter 6, we will give conclusion and discuss some work which can be explored in the future.

Chapter 2

Background

2.1 Network I/O Path In Virtualized Network

KVM is the virtualization infrastructure which transforms linux kernel into a hypervisor. KVM requires the processor to support hardware virtualization extension like Intel VT-x or AMD-V for x86 processors. To create, edit, start or stop KVM based virtual machines, GUI based tool named Virtual Machine Manager is available.

There is a concept of full-virtualization in which the hypervisor emulates actual physical devices like network interface cards for the guest. It allows us to virtualize any operating system we want. But research done in this area has proven that this implementation is inefficient as well as complex. There is a concept of para-virtualization in which the guest's device driver just knows that it is running in a virtual environment and cooperates with the hypervisor. It helps guests to achieve high network and disk performance. Virtio is a virtualization standard for network and disk device drivers which provides para-virtualization with KVM. We have used Virtio enabled KVM-QEMU combination for middlebox virtualization.

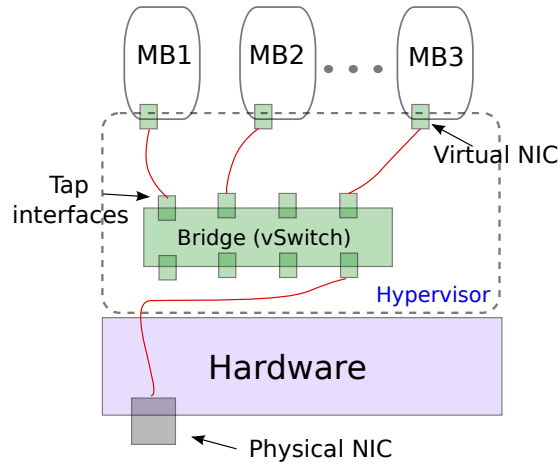


Figure 2.1: packet path from pNIC to middlebox

When we create a virtual machine, we can configure which type of networking we want for that machine. We opt for bridged networking in our experiments. The above explains how packet reaches to middlebox from pNIC in this scenario. On a KVM virtio enabled linux physical machine, each VM(middlebox) will be configured with one vNIC. The VM will be exposed to the internet by pNIC of host. As shown in the figure above, a linux bridge(virtual switch) associates the vNIC of a VM to pNIC of the host. VM's vNIC is associated to the virtual tap interface of the host and that tap interface is added to the linux bridge. On one of the interface of bridge, we have connected pNIC(eth0). Tap interfaces forwards the raw ethernet frames to virtual machines. This figure gives a high level view of how packets travel from pNIC to vNIC and vice versa.

When a packet is received on pNIC, It is forwarded to the hypervisor bridge. Bridge forwards it to the tap interface for destination middlebox. Tap interface forwards it to vNIC. We can consider vNIC as virtio network device. Receiveq and transmitq are the two mandatory virtqueues associated with the virtio network device. Receiveq consists empty buffers for receiving packets. Transmitq consists of empty buffers for outgoing packets in transmission order. Virtqueues reside in the guest memory. Each virqueue can support maximum 64K buffers at a time. As we can see in the figure, each virtqueue consists of three parts.

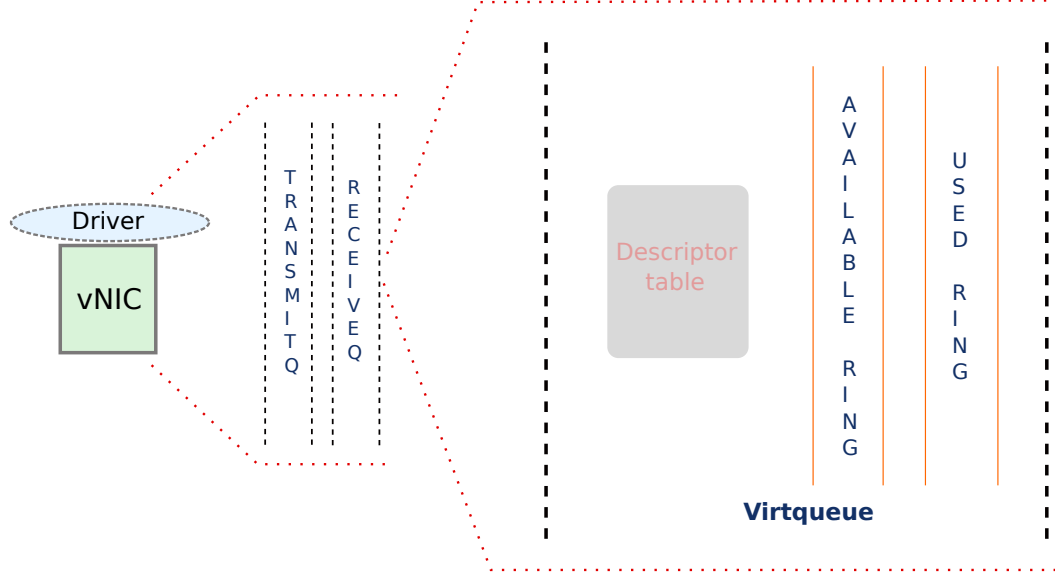


Figure 2.2: virtio network device architecture

1. **Descriptor table:** It keeps track of the buffers used by driver for the network device. Each descriptor contains physical address where the buffer starts, length of the buffer or buffer chain and next field to chain the buffers.
2. **Available ring:** It keeps track of the descriptors, which driver is offering the device. It can only be written by driver and read by the device.
3. **Used ring:** It keeps track of the buffers returned by the device once it processes it. It is only written by the device and read by the driver.

Transmitq and receiveq of the devices are made up of above mentioned three components. In case of packet transmission, outgoing packet is added as a buffer to transmitq. Driver adds the descriptor table entry for the packet and also to available ring. Then driver notifies the device about new entry. Once the device is done with packet transmission, it makes the buffer entry to used ring. So, it again returns the buffer to pool of empty buffers. In case of receiving the packet, packet is copied into a buffer in the receiveq and that is the only difference between transmitting the packet and receiving the packet from the device. Once device processes the packet, it makes buffer entry to used ring to make it free. Now we exactly know how a packet travels from the physical NIC to a middlebox.

2.2 Middlebox Examples

General Middleboxes

Based on a survey[3] done over the data center networks, which were containing varying number of hosts, some of the widely used middleboxes are as described below.

- **Firewalls:** A firewall is a network security system that monitors and controls the incoming and outgoing traffic based on predetermined security rules set by administrator. [7]

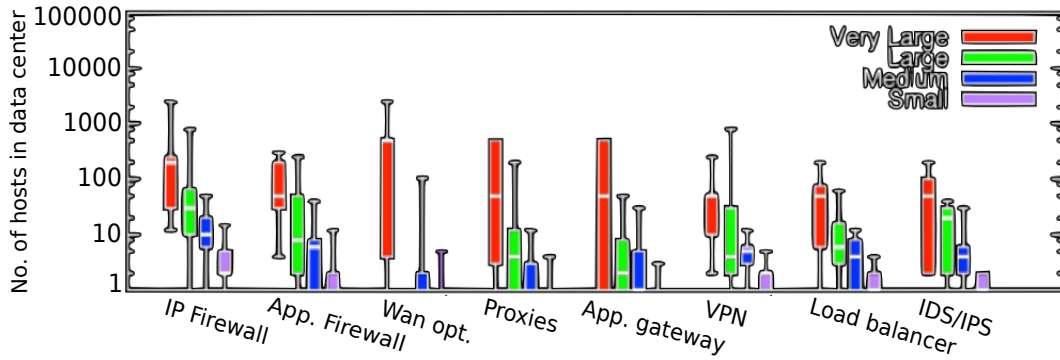


Figure 2.3: Different types of middleboxes in data centres [3]

- **IDS/IPS:** IDS is a software application that monitors the network traffic for malicious activities or policy violation and produces a report to administrator. IPS is proactive approach of monitoring network traffic and identifying malicious activities and prevent them from occurring. [8]
- **Load Balancer:** Load balancer distributes the incoming requests among a set of resources available. Efficient load balancer leads to optimized resource utilization, maximized throughput and minimized response time.
- **Wan optimizer:** It improves bandwidth consumption and latency between dedicated end points. It coordinates to cache and compress traffic that traverses the internet. [4]
- **Network Address Translators:** It serves network traffic to multiple private machines which are exposed to internet through a public host. It modifies the 4-tuple address fields of the packets to ensure that it reaches to the correct host.
- **Traffic shaper:** Traffic shaper is also known as a rate limiter. It limits the amount of bandwidth for specific traffic.
- **Proxies:** A proxy sits in between client and server, to simplify the client's requests to servers and to provide anonymity to clients. There are various types of proxies based on what specific task they do in addition to the basic task. The simplest proxies which passes the requests and responses unmodified are also known as **Gateways**.
- **VPN :**A virtual private network establishes a private network across public networks. It allows user to send and receive data across public networks maintaining the policy and security enforced by the private network.
- **Wire:** A simple middlebox which sends packets from its input interface to output interface. It is generally used to give a performance baseline.

Application Specific Middleboxes

Some network functions can be application specific. Those middlebox functions are defined as small modules of the application which results from modularization of the application. Below are some of the examples of such middleboxes.

- **IP Multimedia Subsystem:** It is an architectural framework to standardize the methods to deliver IP multimedia services to the user mobiles. We will describe several functions which belongs to IPMS architecture. HSS(Home Subscriber Server) is a function which contains master user database and its functionality involves authentication and authorization of the user. SLF(Subscriber Location Function) is responsible to map user address when multiple HSSs are used.

- **GSM network architecture:** GSM architecture includes several components such as base-station, controller, MSC, AuC, HLR, VLR etc. VLR(Visitor Location Register) contains selected information from the HLR that enables the selected services for the individual subscriber to be provided. It can be implemented as a separate entity. EIR(Equipment Identity Register) decides whether a given mobile equipment maybe allowed onto the network. SMSG is the gateway for messages being sent to MEs.
- **EPC architecture:** The Evolved Packet Core is the latest evolution of the 3GPP core network architecture. It contains four network elements: the serving GW, the PDN GW, the MME and the HSS. HSS is same as what we have seen in the IPMS architecture. The gateways transport the IP data traffic between the user equipment(UE) and the external networks.

2.3 Middlebox Classification

There is no bound on the middlebox functionality. Hence there is no bound on the number of middleboxes that can exist. Based on the application requirements, any sub module of the application can run as a middlebox. We aim to propose a solution which covers all the middleboxes. This classification helps us to do that as it groups the middleboxes based on how they process the packets. Any middlebox can be classified in one of the following category. A middlebox which,

- Category 1: Inspects the packet header
- Category 2: Modifies the packet header
- Category 3: Inspects the packet body
- Category 4: Modifies the packet body

We classify the general middleboxes mentioned in the previous section in these four categories.

Category 1	Category 2	Category 3	Category 4
IP firewall, network monitors, gateways	IP load balancer, NAT, protocol spoofing	App. firewall, traffic shaper, IDS/IPS	Data compression, proxies

Table 2.1: Middlebox classification based on the packet processing

This classification is exhaustive classification for the middleboxes. Approach for our proposed solution remains same for all the middleboxes belonging to the same category. Thus we can cover the whole range of middleboxes. We have experimented for category 1 and category 2 middleboxes. We will see it in detail in chapter 5.

Chapter 3

Related Work

Over the past few years, much work has been done in the area of the customization of the guest operating system, customization of the hypervisors which hosts the middleboxes and customization of the underlying hardware to improve the middlebox performance. It is very important to understand the work done so far, before we move ahead with our proposed solution. We can classify the techniques studied so far into three categories.

- Middlebox Operating System optimizations
- Middlebox Hypervisor optimizations
- Middlebox Hardware assisted optimizations

We will briefly discuss some techniques which fall inside these categories. And we will also state how it relates to or contrast with our proposed solution.

3.1 Middlebox Operating System Optimizations

ClickOS[9] proposes replacement of traditional guest operating system with some optimized, light weight operating system. Linux as a guest operating system in VMs is actually over provisioning for middlebox applications. Middlebox uses very few operating system services in addition to network connectivity. Traditional Linux operating system takes around 128MB space on guest VM and also takes around 5 second time to boot, that is very slow in context of middlebox setup time. So, they came up with minimalistic operating system for virtual machines. ClickOS virtual machines which runs MiniOS are very small in size that is only around 5 MB and also very fast to setup, only 30 msec boot time. MiniOS has a single address space, no kernel/user space separation and a cooperative scheduler. Basically ClickOS are single core machines. They have used Xen as a hypervisor for experiments. It also improves scalability of the middleboxes. With traditional full fledged OS running on the VMs, number of tenants supported on a physical machine are very small. But with MiniOs running on ClickOS machine the number increases drastically. This technique improves the scalability and the setup time for middleboxes but it does not address redundancy among middlebox functionalities. Though it can create new ClickOS VM quickly for scaling, ClickOS VMs do not have symmetric multiprocessing support.

OSv[10] also offeres optimized operating system for middleboxes in cloud environment. VMs in cloud usually runs linux as an OS. When we say middlebox is running on a VM, it often means that only single application (middlebox application) runs on that VM. In this case, managing multiple VMs on hypervisor only means that managing multiple applications on the hypervisor. Hypervisor provides features like isolation among VMs, hardware abstraction, memory management. In this case hypervisor almost act as an OS for middlebox applications. So, when we use traditional operating system to run middlebox applications, the above mentioned features become redundant. It affects overall performance of the host machine. OSv is a guest OS specially designed to run a single application on a VM in the cloud environment. It has very small OS

image, dedicates more memory to the application and lesser boot time. OSv supports various hypervisors and processors with minimal change in architecture specific code. For 64-bit x86 processors, it supports KVM, Xen, VMware and Virtual Box hypervisors. It also improves scalability of middleboxes just like ClickOS. But it supports Symmetric multi-processing which is an advantage over ClickOS. It also gives throughput and latency improvement for middleboxes.

Unikernels[11] also provides light weight machines to deploy middlebox applications in cloud environment. Unikernels are single purpose appliances, they cut down functionalities of general purpose system at compile time. It is inherently suitable for middleboxes. It takes into consideration the idea of library OS, in which an application links against separate OS service libraries and unused services from the library are eliminated from the final image by the compiler. For an example, virtual machine image of a DNS server can be as small as 200KB. Mirage OS[12] is an example of such a library OS. It is written in OCaml, a functional programming language and it runs on the Xen hypervisor[13]. It also improves scalability and setup latency for the middleboxes.

3.2 Middlebox Hypervisor Optimizations

Container[14] modifies the hypervisor to eliminate redundant features among the hypervisor and the guest OS. Actually it drops the idea of traditional hypervisor in virtualization. It modifies the host operating system to support isolated execution environments for applications while running on the same kernel. It improves resource utilization among guests and lowers per guest overhead. It also improves the overall performance of the system. I/O related workload, server type workload performs better on container based system compared to hypervisor based system. It also scales well compared to hypervisor setup. But still most of the data centres prefer to use hypervisor based system. Hypervisor based system can support multiple kernels but by design container based system can not support the same. Container also does not have support for VM migration. Hypervisors are the industry standard for virtualization.

CoVisor[15] is a hypervisor, which uses a completely different approach to host middlebox applications in a software defined network. It uses basic concept of network hypervisor that is to manage virtual networks on a physical network. Along with usual middlebox hosting challenges, it deals with some SDN specific challenges as well. For example, middlebox applications used by different vendors may be built by using different SDN APIs. Covisor makes it possible to run any middlebox irrespective of what SDN API is used. Covisor provides facility to assemble multiple middlebox applications as per the administrator configuration. Each middlebox can be used independently, in parallel or sequential with other middlebox, or conditionally. Administrator can provide abstract virtual topology for each middlebox. It restricts each middlebox's view of the physical network. Configuration file provided by administrator includes policies to assemble middleboxes, mapping for each middlebox's virtual network components to actual physical components and access control limitation for each middlebox. Covisor was tested with respect to its composition efficiency and devirtualization efficiency. It gives satisfactory results on metrics like policy compilation time, Rule updation time and total devirtualization time. Covisor is still in development phase and as most of the data centres use traditional hypervisors, switching to Covisor needs lots of modification to existing data centre architecture.

We already discussed ClickOS[9] phase-I in the first section. ClickOS phase-II falls under this section. ClickOS authors ran ClickOS machines on Xen hypervisor for experiments. They modified the Xen hypervisor to achieve throughput and better resource utilization. In traditional Xen hypervisor networking, when a packet is received on physical NIC, it traverses through network driver(dom0), software switch(dom0), virtual interface(dom0), netback driver(dom0) and netfront driver(guest machine) before getting processed by a middlebox. ClickOS technique modifies memory grant mechanism, netback driver, software switch and netfront driver to increase the middlebox throughput. Modified version of Xen is still not capable of handling a chain of middleboxes. Longer the chain is, lower the throughput. Even after all the modifications, hypercalls done by Xen for each packet transmission remains the bottleneck in this case.

3.3 Middlebox Hardware Assisted Optimizations

Virtual Machine Device queue (VMDq)[5] is part of Intel Virtualization Technology which improves high speed network performance and reduces CPU utilization. As packets are received on the network adapter, a layer 2 classifier in the network adapter determines for which virtual machine each packet is destined for based on the MAC address and VLAN tags. The hypervisor's bridge just route the packet to the specific VM, avoiding the work of sorting every incoming packet. it focuses on the receive side network I/O to reduce the CPU utilization. Without VMDq the receive only throughput on the 10Gbps line was 4 Gbps. Experiments show that with VMDq, throughput became more than double which is 9.2 Gbps. But this feature is hardware dependent and supported by only some of the ethernet controller cards. This feature is available only in Intel 82575 Gigabit Ethernet Controller and Intel 82598 10 Gigabit ethernet controller. It also requires some hypervisor enabling. It is costly and it also has scalability issue.

Direct I/O[6], is a technique in which a hardware device supports multiple logical interfaces. Guest virtual machines can bypass the virtualization layer and access the logical interfaces securely. It gives CPU performance close to the CPU performance without virtualization. But direct I/O differs from basic dedicated driver model and lacks the key advantages of it. It avoids full support for guest VM transparent services like live migration and traffic monitoring. In order to enable these services, additional support in the hardware device required. Direct I/O model is also difficult to apply to virtual appliance models of software distribution which rely on the ability to execute on the arbitrary hardware platforms. For this direct I/O has to include device drivers for a large variety of devices which increases the complexity and maintainability.

Single Root I/O virtualization(SR-IOV)[1] aims at removing major Virtual Machine Manager intervention for performance data movement such as the packet classification and address translation. SR-IOV is ancestor of Direct I/O which offloads memory protection and address translation using IOMMU. A device which supports SR-IOV can create multiple light weight instances of PCI function entities, also known as Virtual Functions(VFs). Each VF can be assigned to guest for direct access but still shares the major device resources. General architecture of SR-IOV devices contains a VF driver, a PF driver and an SR-IOV manager. The VF driver runs in the guest OS, the PF driver runs in host OS to manage PF and the SR-IOV manager in VMM. Now communication between them goes through the SR-IOV manager which makes this design independent of VMM. SR-IOV provides hypervisor bypass by attaching a VF to VM and sharing a single physical NIC. But it requires hardware support for achieving the goal. VM portability is also an issue with SR-IOV supported devices. The hypervisor should be capable of moving VMs between SR-IOV and non SR-IOV platforms in case of VM migration from one server to another. A server receiving traffic on switch ports has no way to distinguish the virtual function traffic. It may result in switching problem or some confusing situation.[16][17]

3.4 Comparison Of All The Approaches

Our goal is to achieve performance improvement for the middleboxes running on the physical machine in the virtualized environment. Performance of the middleboxes is a very abstract term. Based on the analysis of above mentioned techniques, We can narrow it down to several metrics. Let's compare broad categories of the middlebox optimization techniques with our approach based on the metrics like middlebox setup latency, scalability, processor utilization, throughput and packet processing time. [Table - 3.1]

We can observe that solutions from each category improves on some of the metrics. They do not affect other metrics. Middlebox OS optimizations category solutions improves middlebox setup latency and scalability. They do not focus on improving metrics like throughput, processor utilization for the middlebox host. Middlebox hypervisor optimizations category solutions improves on metrics like throughput and scalability of the middleboxes. They do not improve on metrics like packet processing time by the middleboxes. Middlebox hardware assisted optimizations category solutions offer gains on throughput and CPU utilization. But they affect the scalability and the portability of the VMs. Our proposed solution is a combination

of middlebox operating system and hypervisor optimization categories. Our technique focuses on improvement of the performance metrics like CPU utilization, throughput, packet processing time without affecting scalability or portability of VMs.

Category	Metrics
Middlebox Operating System optimizations	middlebox set up latency, scalability
Middlebox Hypervisor optimizations	scalability, throughput, packet processing time
Middlebox Hardware Assisted optimizations	throughput, packet processing time, CPU utilization
Our Approach	packet processing time, CPU utilization, throughput, hardware/software platform independent

Table 3.1: Comparision between different I/O virtualization techniques for middleboxes

In later chapters, we will see proposed design of our solution, implementation part and cost analysis of the proposed scheme. We will also discuss different challenges faced throughout the process.

Chapter 4

Design of Middlebox Specific Hypervisor Extensions

In this chapter, we will discuss how to provide such intelligence to the hypervisor in order to bypass the actual middlebox processing to reduce the CPU processing overhead and to improve the packet processing time. We will also discuss several metrics to be observed in order to make this design worthy for a virtualized middlebox.

4.1 Overall Design

As per definition, middlebox performs a specific network function and in case of virtualized middlebox, network function is an application running on a virtual machine. We want to execute a part of the middlebox functionality into the hypervisor. We need to design three components in order to do that. First is the way in which the middlebox conveys the hypervisor, what to offload. Second is where to execute the offloaded functionality in the hypervisor. Third is, how to execute the offloaded functionality. Let us discuss them in detail.

1. **Communication Between a Middlebox And a Hypervisor:**
Middlebox should inform the hypervisor about which functionality it wants to offload. Middlebox should also be able to dynamically inform the hypervisor about the change in its state. We have developed an interface between a VM and a hypervisor using some available POSIX shared memory mechanism. The interface accepts several kind of commands from the middlebox and updates the middlebox state stored on the hypervisor accordingly.
2. **Hook In The Hypervisor To Execute The Offloaded Functionality:**
Second task is to find a point in the hypervisor where we can fit in this functionality. It should be a point from where all the network traffic passes. Virtual bridge in the hypervisor in one such point. All the packets arrive at the bridge and then sorting is performed to forward each packet to the destination middlebox. In the bridge processing when sorting is done, we also execute the offloaded functionality. Thus we bypass the middlebox processing on the guest.
3. **Execution of The Offloaded Functionality:**
Each offloaded middlebox functionality is implemented as a kernel module in the hypervisor and for each of them, respective middlebox state is stored on the hypervisor. Whenever a packet arrives for the middlebox with the offloaded functionality, hypervisor uses this stored state to take an appropriate action on the packet. The hypervisor always maintains the updated state of the middlebox in order to execute the offloaded functionality correctly.

Thus the proposed design requires a hook to be added in the packet processing in the hypervisor, an interface for communication between the middlebox and the hypervisor and the state of the middlebox to be stored on the hypervisor. Figure-4.1 represents the overall design we just discussed.

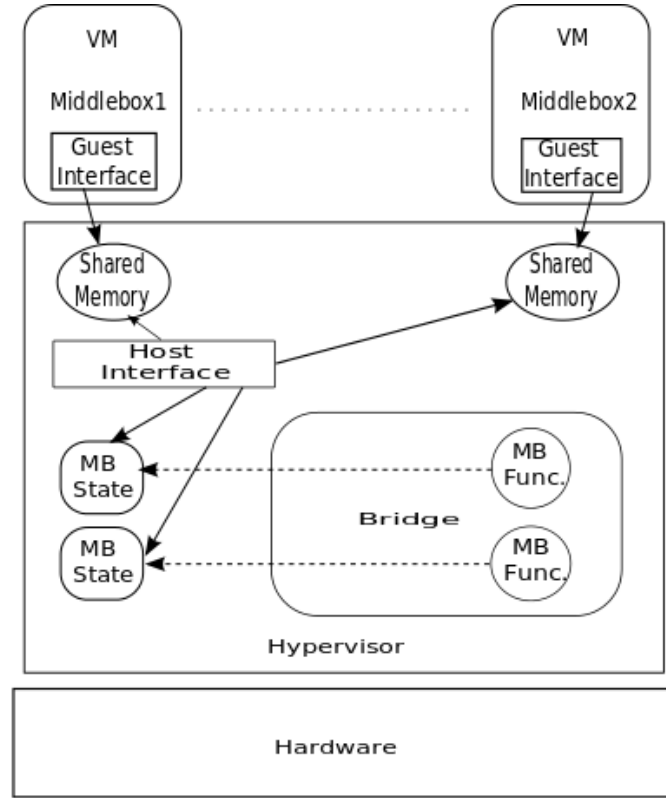


Figure 4.1: Design Of Middlebox Specialized Hypervisor

4.2 Implementation

We will describe each component shown in the figure-4.1 in detail. We will discuss implementation details of each component and interconnection between the components.

4.2.1 Shared Memory

We have used `ivshmem`[18], a QEMU virtual PCI device to share space between a middlebox and the hypervisor for communication. It is fairly simple to use[18]. The guest OS can access a POSIX SHM region on the host through the `ivshmem` device. It emulates memory mapped I/O access on a physical device, the host SHM region appears as a MMIO region to the guest OS. We can configure this device in the QEMU command to fire a VM.

```
$qemu -system -x86_64 .... -device ivshmem,shm=ivshmem1,size=2
```

Here, `ivshmem1` is the name of the POSIX SHM object to use as the shared memory. It is `/dev/shm/ivshmem1` in above example. The size parameter defines size of the POSIX SHM object in MB. It must be a power of two as a restriction of PCI memory regions. We have used the PCI device driver developed by Siro Mugabi[18] to write on the shared memory from a MB.

4.2.2 Middlebox Side (Guest) Interface

On the guest side, we will first install the `ivshmem` driver module, `ivshmem_driver`. Then, we will create a helper binary file for reading / writing into the POSIX shared memory. We can use simple commands like,

```
./a.out -r, to read from the shared memory.
```

```
./a.out -w "@hello@", to write hello to the shared memory
```

```
./a.out -help, to get more help about all the commands.
```

We have used `"@-@"` as the separators to parse a specific command from the shared memory. In case, a middlebox wants to convey anything to the hypervisor, it does so by writing that information on the shared memory embedded within `@s` as shown above.

4.2.3 Hypervisor Side (Host) Interface

On the hypervisor, we need to access the PCI device's MMIO data region to get the commands from the middlebox. We have written a program readSHM.c, which uses mmap() system-call to memory map the ivshmem device's MMIO data region into its own virtual address space. Now, the process will have direct access to the shared region by means of pointer referencing. As the process reads a command from the shared memory, it makes a system call to pass that command to the kernel space. We have implemented a separate system call to parse the command and to update the middlebox state stored in some kernel data structure accordingly. We have implemented specialization of the hypervisor for Firewall and Load balancer middleboxes. Let's see what commands do middlebox passes and how the system call interprets them in our case.

Acronym	Stands for
R	To register a middlebox for hypervisor services
F	Firewall services
L	Load balancer services
A	To enable services for a registered middlebox
D	To disable services for a registered middlebox
X	To cancel the middlebox registration
I	To show current status of the IP firewall filters
M	To show current status of the MAC firewall filters
T	To show current status of the APP. firewall filters
s	srcIP / srcMAC / srcPort
d	dstIP / dstMAC / dstPort
t	Type of service required
p	Transport / Network / MAC layer protocol

Table 4.1: Acronyms used for the middlebox commands

We have used above described acronyms to build the middlebox commands. Most of them are self-explanatory. But more explanation is needed for some of the acronyms like I, M or T. They are implemented as an integer type value, they can take any decimal value between 0 to 15. These decimal numbers can be represented by 4-digit binary values. There are different types of filters associated with each bit of those binary values as firewall services. One's in the corresponding binary representation of the decimal value suggest that filter associated to that particular bit is set. Zero's suggest that filter associated to that particular bit is not set. Now, let's see the command formats.

Purpose	Command
To register a middlebox for firewall services	R F <MAC address> <IP address>
To register a middlebox for load balancer services	R L <MAC address> <IP address> <type> <srcIP> <dstIP>
To add firewall services for a reg. middlebox	A <MAC address> {<I/M/T> value} <s value> <d> <t value> <p value>
To remove firewall services for a reg. middlebox	D <MAC address> {<I/M/T> value} <s> <d> <t> <p>
To cancel registration of a middlebox	X <MAC address>

Table 4.2: Middlebox commands for Firewall / Load balancer services

In the command to register a middlebox for load balancer services, type field can take binary value, 0 or 1. If it is zero, it indicates that traffic coming for that middlebox need to be redirected to the specified destination irrespective of the source. If it is one, it indicates that traffic only from the specified source should be redirected to the specified destination. Now, let's see few examples of different commands from middleboxes.

For example, to register a middlebox having 52:34:56:00:12:22 as the MAC address and 10.129.26.115 as the IP address for firewall services, the command will be,

R F 52:34:56:00:12:22 10.129.126.115

To enable firewall services for the above registered middlebox, on source MAC address and MAC layer protocol for incoming packets, the command will be,

A 52:34:56:00:12:22 M 5 s 12:13:14:15:16:17 p 8

To disable firewall services for the above registered middlebox, on the source MAC address for incoming packets, the command will be,

D 52:34:56:00:12:22 M 4 s

To implement the host side interface, we need a program to continuously read from the POSIX shared memory. It is approximately 100 lines of C code. We need a system call to parse the command and transfer the data from user space to kernel data structures. It is approximately 450 lines of C code. On the guest side interface, we can use already available implementation of ivshmem device driver[18] as described earlier.

4.2.4 Middlebox State Stored At The Hypervisor

Middlebox state is required at the hypervisor to keep track of the services it has registered for. It is also required to store the information required to execute the offloaded middlebox functionality. For each type of service provided by the hypervisor, we keep an object for all the middleboxes which registered for that service. The object contains all the required fields needed by the hypervisor to provide a specific service. For example, firewall functionality structure contains source IP/MAC/Port address, destination IP/MAC/Port address, ipfilters, macfilters, tcpfilters, tos and protocol as members. The host interface reads a command from the middlebox from the shared memory and parses it as shown in above section. In addition to that, it also updates the object belongs to that middlebox accordingly as shown in fig-4.1. We have added two structures to the /include/linux/syscalls.h header file for the firewall functionality and the load balancer functionality respectively.

4.2.5 Offloaded Middlebox Functionalities

We have implemented each offloaded middlebox functionality as a loadable kernel module. In order to make middleboxes use those hypervisor services, we just need to insert the module corresponding to the service. Once the module is inserted, middleboxes communicate with the hypervisor as shown in the previous section. A firewall performs various tasks such as some complex decision making tasks, filtering of the packets, generating alert for security breach etc. We offloaded a light weight filtering task of the firewall to the hypervisor. We have implemented a kernel module, 350 lines of C code for filtering task. A load balancer also performs several tasks such as decision making for intelligent load balancing among the servers, guarantees always availability of the application, add or remove servers to the pool etc. We offloaded simple forwarding functionality of the load balancer which is performed once all the decision making is done. We have implemented a kernel module for that too, around 200 lines of c code. Both these modules use the stored middlebox state to take appropriate action on the packets.

It was very important to find a hook in the hypervisor processing to fit these module processing. We insert this processing in the bridge forwarding part of the hypervisor at the point where the first interrupt is generated for the incoming packet, in br_handle_frame_finish() function inside /net/bridge/br.input.c file. As a packet passed to the hypervisor bridge module,

br_handle_frame is the first function to be executed. It does initial processing like initial validity checks on the frame, separating ethernet control frames and data frames etc on the incoming frame. Then the frame is passed to br_handle_frame_finish, where the actual forwarding process begins. So, our modules get executed before any kind of sorting or decision making performed by the hypervisor bridge module.

4.3 Different Packet Traversal Paths Analysis

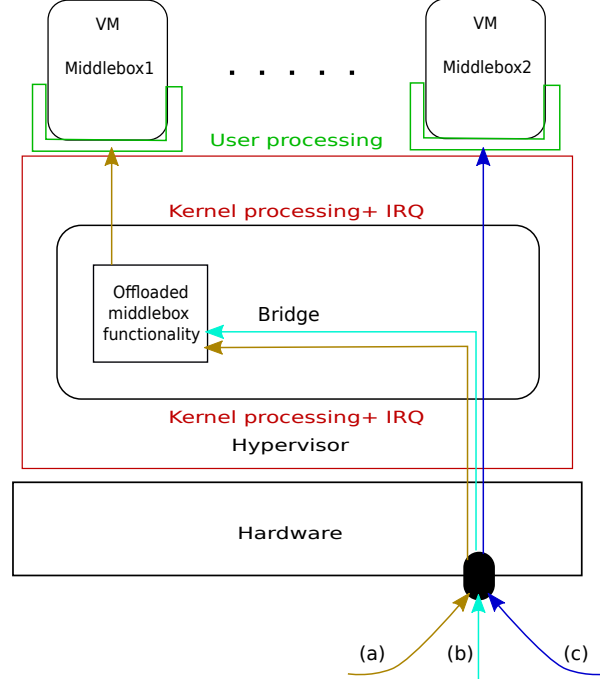


Figure 4.2: Different packet traversal paths

In a network set up where middleboxes are virtualized as per the above design, a packet can be processed in three different ways as described in figure-4.2. Any middlebox for which none of the functionality is offloaded to the hypervisor, packets are processed in normal way. It includes kernel processing on the hypervisor, IRQ processing, guest kernel processing and finally the middlebox application processing. For example, path (c) for middlebox2 in the above figure. Any middlebox for which some of the functionality is offloaded to the hypervisor, packets can be processed in two ways. First of all, every packet for such a middlebox is processed by the hypervisor module. Then only some of the packets will be forwarded to the middlebox for further processing according to the middlebox state stored on the hypervisor. Appropriate action will be taken on the other packets by the module itself, thus the middlebox is bypassed for those packets. The cost associated with the first kind of packet processing is kernel processing on the hypervisor, additional module processing, IRQ processing, guest kernel processing and middlebox application processing excluding the offloaded functionality processing. For example, path (a) for middlebox1 in the above figure. The cost associated with the other packets is kernel processing on the hypervisor, additional module processing and reduced IRQ processing. For example, path (b) for middlebox1 in the above figure. Clearly, offloading the middlebox functionality to the hypervisor is advantageous only if the average CPU utilization and the average packet processing time for the design with the hypervisor module is less compare to normal design without the module. It largely depends on the which middlebox functionality is offloaded to the hypervisor. It also depends on the division of the incoming traffic among three ways described here. It is very important to decide how much of the middlebox functionality should be offloaded to the hypervisor. It can be decided based on some intelligent assumptions or prior experience or may be on trial and error basis.

Chapter 5

Experimental Evaluation

In this section, we showcase performance of the virtualized set up having the hypervisor specialized for the middleboxes. Our analysis focus on the processing of the received packets on the hypervisor. We analyse the time taken to process a packet in such a set up and cost of offloading the middlebox functionality to the hypervisor in terms of CPU utilization.

5.1 Experimentation Setup

Our experiment setup includes the virtual machines running on the 64 bit, x86 hardware connected to the other machines through a gigabit 8-port d-link switch. The server has four 3GHz Intel CPUs with 8GB of memory and Intel I217-V gigabit ethernet network interface card. During the experiment we assume that all the four CPUs are busy serving the middleboxes running on it. To generate network traffic, we used iperf[19] unidirectional UDP packet streams. TCP traffic generates ack packets in response to the data packets, so it would be difficult to observe the processor cost only on the receive path of the packets. All the experiments are done for the UDP traffic but we can expect the same results for the TCP traffic at I/O virtualization layer. KVM-4.1.6, QEMU-2.2.0 and libVirt API is used as the hypervisor for the experiments. Virtual machines have virtual NICs which follow the virtio standards and connect to the hypervisor virtual bridge as the tap interfaces. We use function_graph tool of the function tracer[20] to measure the time taken to process each packet on the hypervisor as well as on the guest machine. We use htop[21] to measure the CPU utilization for processing the packets at gigabit line rate.

5.2 Experiment Details

In order to do the experiments, we run two 64-bit, 3GHz, 2GB virtual machines on the host machine. We assume that one of the virtual machine runs the firewall application. Thus it becomes the firewall middlebox. And the other virtual machine runs the load balancer application. Thus it becomes the load balancer middlebox. We offload filtering functionality of the firewall middlebox and matching and forwarding functionality of the load balancer middlebox to the hypervisor. We use Iperf UDP packets of the size 52 bytes and 1500 bytes as the workloads. Turn by turn, We send these loads to both of the middleboxes at gigabit line rate. We focus on the metrics like no. of interrupts generated due to incoming packets, average packet processing time and average CPU utilization. We observe these metrics for all possible packet processing flows discussed in the previous chapter. Let's analyse the results of the experiments in detail.

5.3 Overall Cost Of Specialized I/O Virtualization

In the first part, we log number of times CPU gets interrupted in order to process a packet. In second part, we discuss total time required to process an incoming UDP packet until it is

consumed by the middlebox application. The last part compares the processor utilization to process the stream of UDP packets with the specialized I/O and without it.

5.3.1 Analysis Of Network Interrupts

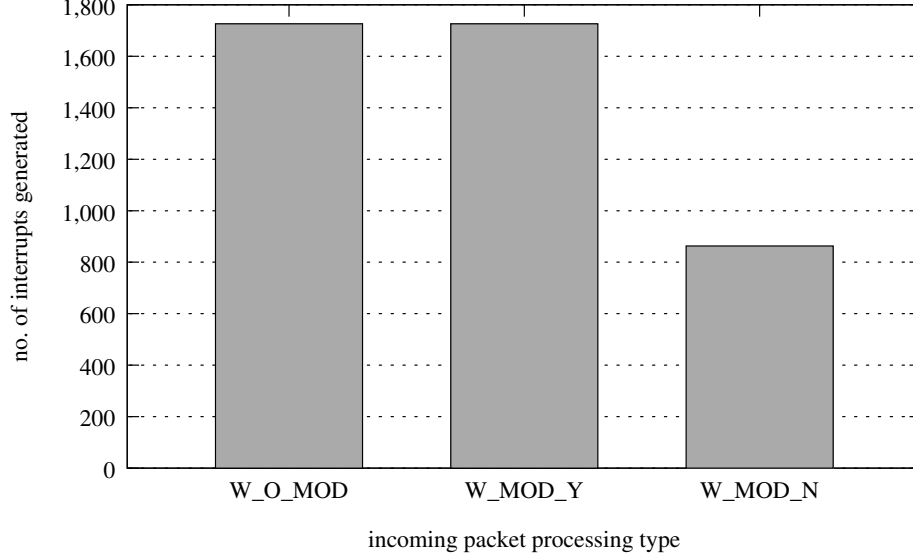


Figure 5.1: No. of interrupts generated for 875 incoming UDP packets of size 1500 bytes at 100Mbps line rate out of which 12 packets got dropped

When a packet arrives from the network, it generates a software interrupt. All the CPU cores are busy serving the middleboxes. One of the designated CPU core needs to serve the interrupt. It generates another interrupt for the CPU core serving the destination middlebox virtual machine for the packet. Then the first interrupted core gets back to work again until next packet arrives. In high speed networks, interrupt generating rate will also be high. Server processes the packets at much lower rate than line rate. In our network set up, physical network interface card is attached to a bridge port. Each bridge port will have rx_handler which in turn calls br_handle_frame function of /net/bridge/br.input.c as the entry point of the bridge processing in the hypervisor[22]. The network interface code __netif_receive_skb calls the rx_handler. So, that is the entry point of the packet processing and that is where the first interrupt is generated. In this experiments, we log the number of calls to the __netif_receive_skb for a given UDP packets stream and thus we keep the count of number of interrupts generated. We do this on the hypervisor as well as on the guest. In normal virtualized set up, two interrupts will be generated per received packet as we just discussed. In our specialized virtualized set up, we bypass the middlebox processing for the packets which are processed by the offloaded middlebox module at the hypervisor. Thus each incoming packet which can be handled by the hypervisor module generates only single interrupt. The above graph shows no. of interrupts generated for 863 incoming UDP packets of size 1500 bytes at 100Mbps line rate. W_O_MOD stands for processing when the hypervisor module is disabled. W_MOD_Y stands for processing when the hypervisor module is enabled and the packet is forwarded to the middlebox as the module does not able to handle the packet. W_MOD_N stands for processing when the hypervisor module is enabled and module takes appropriate action on the packet. Figure-5.3 shows that number of interrupts generated for incoming packets are reduced by 50 percentage.

5.3.2 Analysis Of Packet Processing Time

We compare the average packet processing time with the offloaded hypervisor module and without it. Here, we define the packet processing time as time required for the execution of the two interrupt subroutines. Execution of the first interrupt subroutine includes offloaded middlebox functionality processing. The x-axis in the above graphs is UDP packet size. The

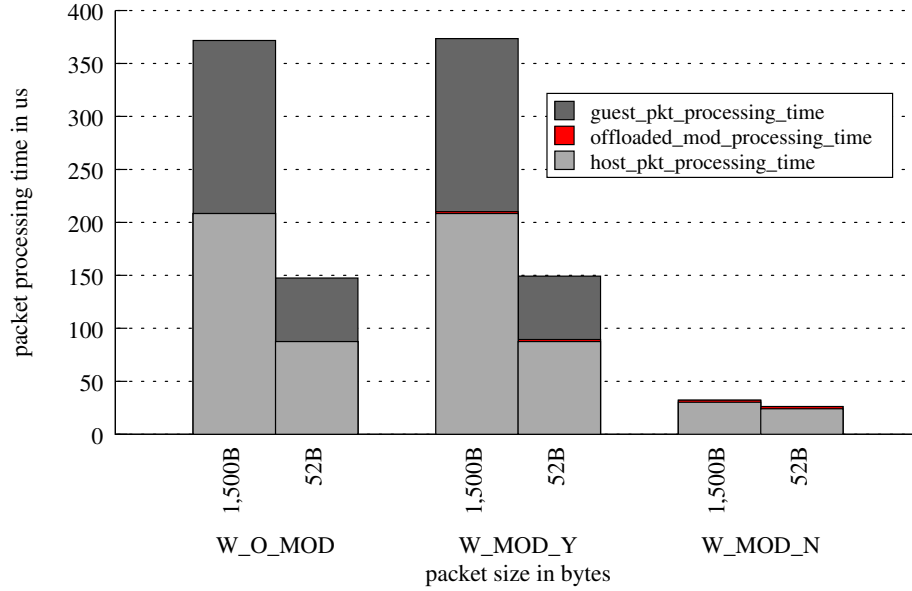


Figure 5.2: Firewall Middlebox

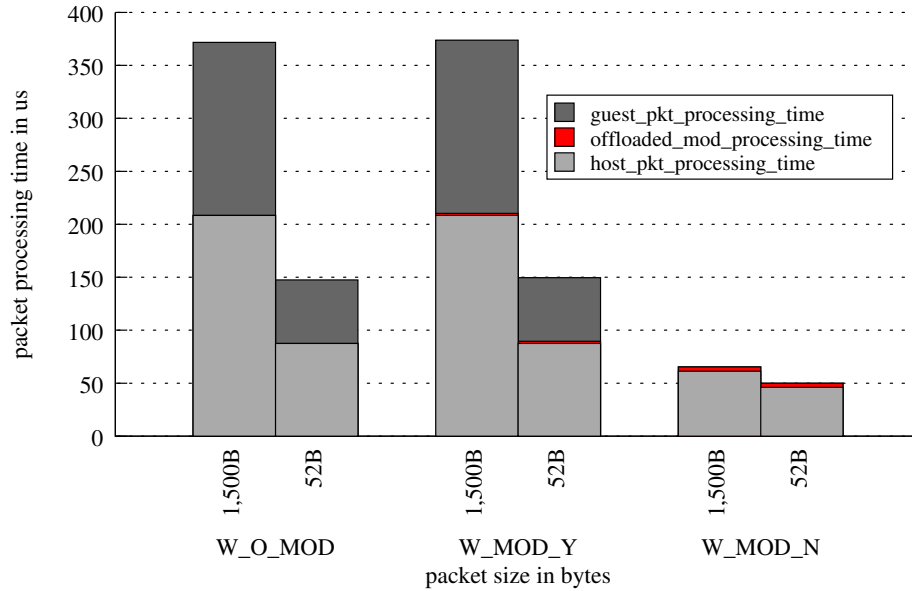


Figure 5.3: Load Balancer Middlebox

graphs shows results for 1500 bytes and 52 bytes UDP packets. 1500 bytes packets depict a maximum size Ethernet packet and 52 bytes packets depict a TCP acknowledgement packet. The larger the packet size the larger the chunk size of data for delivery. Thus packet processing time will be more. We can see this in graphs shown above. Important thing to observe is, we have not included the actual middlebox application processing time in case of W_O_MOD and W_MOD_Y. It completely depends on the middlebox application. But in case of W_MOD_Y, it should definitely be less than W_O_MOD as in first case some part of the application is already processed at the hypervisor as shown in the graph. If the processing time in W_MOD_Y case is much greater than the processing time in W_O_MOD case then it suggest that we have not offloaded the functionality in the correct way. We can see that in W_MOD_N case, the packet processing time for the load balancer middlebox is more than the packet processing time for the firewall middlebox. It depends on the action taken on the packet once the module is executed and the time taken by the offloaded module. For firewall middlebox, the module drops the packet at the very beginning of the host processing according to the middlebox state stored. For load balancer middlebox, the module modifies the destination IP address of the

packet and process it normally to forward it. Thus second action requires more time compare to the first one. For firewall or load balancer middlebox, for 1500 bytes packets, W_O_MOD packet processing time contains 208.33us of host processing and 163.46us for guest processing and for 52 bytes packets, 87.58us of host processing and 59.96us for guest processing. For firewall, for 1500 bytes packets, W_MOD_N packet processing time contains 32.23us of host processing and for 52 bytes packets, 26.15us of host processing. This processing includes 1.8us time of the offloaded firewall functionality. For load balancer, for 1500 bytes packets, W_MOD_N packet processing time contains 65.40us of host processing and for 52 bytes packets, 50us of host processing. This processing includes 2us of the offloaded load balancer functionality. These results show that for firewall middlebox, W_MOD_N packet processing time is 91.33 percentage and 82.28 percentage lower than the W_O_MOD packet processing time for 1500 bytes and 52 bytes packets respectively. For load balancer middlebox, W_MOD_N packet processing time is 82.41 percentage and 66.11 percentage lower than the W_O_MOD packet processing time for 1500 bytes and 52 bytes packets respectively. Figure-5.2 and figure-5.3 represent these results. It suggest that filtering functionality for the firewall middlebox and the address matching and forwarding functionality of the load balancer middlebox is worthy to be offloaded at the hypervisor.

5.3.3 Analysis Of CPU Utilization

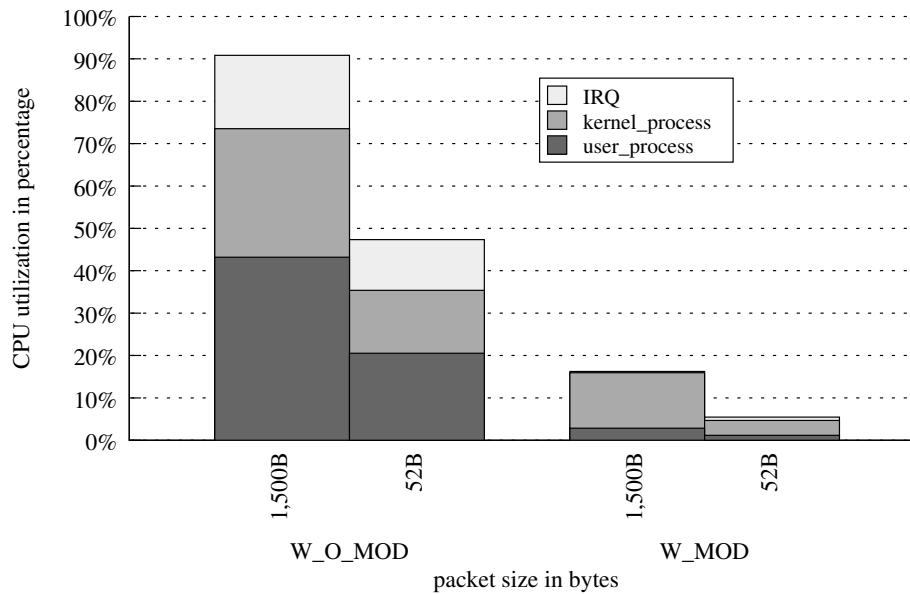


Figure 5.4: Firewall Middlebox

We compare the CPU utilization to process a stream of UDP packets with the hypervisor module and without it. Here, we can distribute the CPU utilization in three parts: user process utilization, kernel process utilization and interrupt request service utilization. This measurements are done at the hypervisor. so the user process utilization includes the guest kernel process utilization and the guest user process utilization. In fact, there was no other user process running during the experiments. Here, also x-axis for the graphs is UDP packet size. At 1Gbps line rate, in case of 52 bytes UDP packets, the CPU buffer becomes the bottleneck and it drops lots of packets. This is why we can clearly see the difference between CPU utilization for both the packet sizes in above graphs. These results show that for firewall middlebox, W_MOD CPU utilization is 82.13 percentage and 88.49 percentage lower than the W_O_MOD CPU utilization for 1500 bytes and 52 bytes packets respectively. For load balancer middlebox, W_MOD CPU utilization is 54.37 percentage and 50.68 percentage lower than the W_O_MOD packet processing time for 1500 bytes and 52 bytes packets respectively. Figure-5.4 and figure-5.5 represent these results. In W_MOD case, we completely bypass the guest processing. It explains the drop of approximately 94 percentage of user process utilization between two categories. For

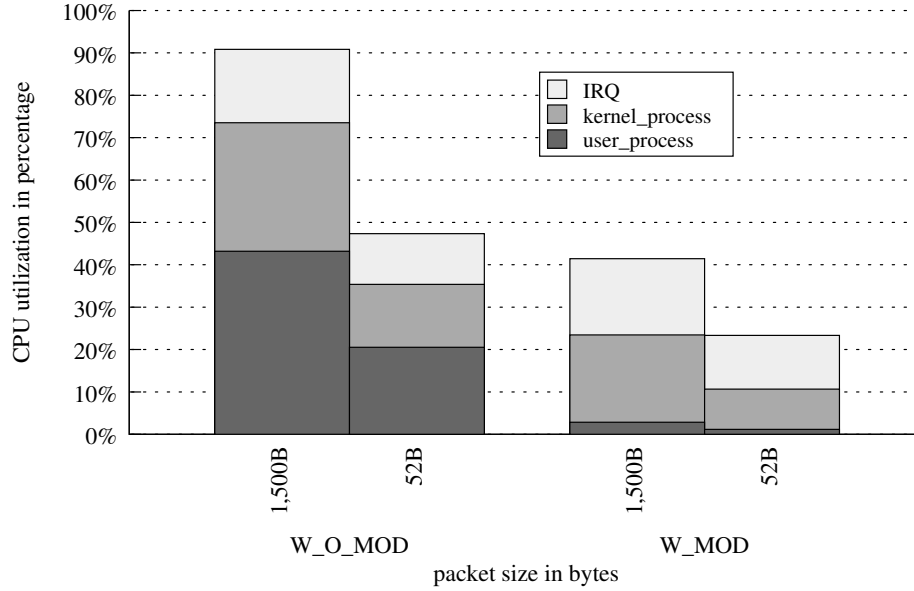


Figure 5.5: Load Balancer Middlebox

firewall middlebox, the module drops the packets at the very beginning of the interrupt request processing. It explains drop of approximately 96 percentage of the IRQ utilization between two categories in the firewall graph. For load balancer middlebox the packet header gets modified as a part of offloaded load balancer functionality and it follows the normal bridge forwarding procedure. It explains approximately 5 percentage increase of the IRQ process utilization in the load balancer graph. Kernel process utilization also reduces for both the middleboxes but not as significantly as the user process utilization. The real impact of this improvement can be understood in case of the high speed network such as 10Gbps. Intel VMDq[5] paper shows that at 10Gbps line rate, throughput achieved is only 4Gbps at maximum. In such a scenario, the amount of reduction in the CPU utilization shown in the above graphs can improve the throughput significantly.

Now let's address some of the challenges in the design and the implementation of this specialized I/O virtualization for the middleboxes.

5.4 Challenges

Which middlebox functionality to offload ?

The choice of the middlebox functionality which is being offloaded to the hypervisor is absolutely critical to the performance of the system. For any general or application specific middlebox, the choice made is correct or not that depends on the two metrics we just discussed: packet processing time and CPU utilization. If the system does not improve on these metrics then it indicates that the choice is not correct. We can use trial and error kind of approach here. Initial choice is made by intuitions, intelligent guesses or assumptions. Then we can fine tune the choice until it meets the expectation or discard it if it is not working out. For example, in our experiments we guessed that if the packets which are going to be filtered at the firewall middlebox, can be filtered at the hypervisor then it will improve the performance of the system and it did exactly the same.

Centralised vs distributed communication interface

We can see in the previous chapter that each middlebox running on the server set up a POSIX shared memory with the hypervisor. Further communication between the hypervisor and the middlebox takes place through that shared memory. We have used this approach in our experiments. Generally in a data center, administrator will manage the middlebox func-

functionalities. In this case only one shared memory segment is required between the administrator and the hypervisor. Administrator can communicate through that for any of the middlebox running on the hypervisor. First kind of implementation completely isolates all the middlebox communications to the hypervisors. Network administrator can choose any of the option based the network requirements.

Middlebox functionality must be tempered

It is very important to understand that here we are offloading the middlebox application which means that the offloaded functionality should not be performed again at the middlebox for the packets which travel to the middlebox. Otherwise performance of system might suffer for all those packets. This design need slight modification to the actual middlebox application to disable the offloaded functionality. In our experiments, we have the offloaded functionality module installed at the hypervisor but we are not running any middlebox application in the guests machines as it was not necessary to obtain the desired results. How complex it may get completely depends on what functionality is being offloaded.

Performance of the system depends on network traffic

We have distributed the incoming packets in two categories when the hypervisor module for a middlebox is installed: `W_MOD_Y` and `W_MOD_N`. We have described them in detail in the previous section. If most part of the incoming packets belongs to the first category then this design does not produce any noticeable gain compare to the normal design. In fact, implementing the offloaded functionality module and communication interface between the middlebox and the hypervisor may become overhead in such a scenario. Thus, along side which middlebox functionality is being offloaded, what kind of network traffic a middlebox consumes also decide the performance of the newly implemented system.

Chapter 6

Conclusion

We designed and implemented a specialized hypervisor on top of a traditional one for efficient middlebox virtualization. We have classified the middleboxes in several categories. We compared several other techniques for middlebox virtualization on different performance metrics. For a firewall middlebox virtualization, we offloaded the filtering functionality to the hypervisor and the average packet processing time for the packet processed at the hypervisor reduces to 32 us from 370 us plus middlebox application processing time. The average CPU utilization improves almost by 75 percentage. For the load balancer middlebox, we offloaded the address matching and forwarding part to the hypervisor and the average packet processing time for the packets processed at the hypervisor reduces to 65 us from 370 us plus middlebox application processing time. The average CPU utilization improves by almost 50 percentage. These results also suggest that the throughput of the system has also been improved. Most of the previous techniques do not improve on these metrics at this scale except hardware assisted optimizations for virtualization. But they are costly and they have issues like scalability and portability. Our design is not that generic in approach as the hypervisor needs to get configured for each different kind of middlebox. We have provided a solution to do that and once it is done, we can see major improvement in the performance of the middlebox without any major issues.

In addition to the work done so far, here we make few suggestions for the future steps, to fine tune the design that we suggested and to test it in the more robust manner.

Future work

- To thoroughly test the suggested design, it needs to be implemented for some application level middleboxes in a complete set up of the application network. Then the set up should be tested for all the scenarios that we have mentioned earlier.
- While implementing the interface between the middlebox and the hypervisor, we have used the polling mechanism. The interface continuously checks the shared memory to dynamically receive a new command from the middlebox. Instead, we can implement some event based mechanism such that when a new command is received from the middlebox, interface signals the hypervisor to read it from the shared memory.
- The experiments we have done are good enough to showcase the impact of the design. But in order to support these results more concretely the experiments should be performed for high speed networks such as 10Gbps network, in which servers often get saturated due to higher line rates.

References

- [1] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan. High performance network virtualization with sr-iov. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–10, Jan 2010.
- [2] Data Plane Development Kit. <http://dpdk.org/>.
- [3] Data centre middleboxes. <http://www.cs.cornell.edu/courses/cs5413/2014fa/lectures/26-datacenter-middleboxes.pdf>.
- [4] Middleboxes. <https://en.wikipedia.org/wiki/Middlebox>.
- [5] Intel VMDq Virtualization Technology. <https://www.vmware.com/files/pdf/partners/intel/vmdq-white-paper-wp.pdf>.
- [6] Jose Renato Santos, Yoshio Turner, G. Janakiraman, and Ian Pratt. Bridging the gap between software and hardware techniques for i/o virtualization. In *USENIX 2008 Annual Technical Conference, ATC'08*, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.
- [7] Firewall middlebox. [https://en.wikipedia.org/wiki/Firewall_\(computing\)](https://en.wikipedia.org/wiki/Firewall_(computing)).
- [8] IDS/IPS middleboxes. https://en.wikipedia.org/wiki/Intrusion_detection_system.
- [9] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473, Seattle, WA, April 2014. USENIX Association.
- [10] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. Osv—optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72, Philadelphia, PA, June 2014. USENIX Association.
- [11] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13*, pages 461–472, New York, NY, USA, 2013. ACM.
- [12] MIRAGE OS. <https://mirage.io/>.
- [13] Xen Hypervisor. <http://www.xenproject.org/developers/teams/hypervisor.html>.
- [14] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, EuroSys '07*, pages 275–287, New York, NY, USA, 2007. ACM.

- [15] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. Covisor: A compositional hypervisor for software-defined networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 87–101, Oakland, CA, May 2015. USENIX Association.
- [16] What is SR-IOV? <http://blog.scottlowe.org/2009/12/02/what-is-sr-iov/>.
- [17] What are SR-IOV pros and cons? <http://searchwindowsserver.techtarget.com/answer/What-are-some-benefits-and-limitations-of-Windows-Server-SR-IOV>.
- [18] Writing a PCI Device Driver, A Tutorial with a QEMU Virtual Device. http://nairobi-embedded.org/linux_pci_device_driver.html.
- [19] iPerf-The network bandwidth measurement tool. <https://iperf.fr/>.
- [20] ftrace-Function Tracer. <https://www.kernel.org/doc/Documentation/trace/ftrace.txt>.
- [21] htop - an interactive process viewer for Unix. <http://hisham.hm/htop/>.
- [22] Department of Communications Nuutti Varis, Aalto University School of Electrical Engineering and Finland Email: firstname.lastname@aalto.fi Networking P.O.Box 13000, 00076 Aalto. Anatomy of a linux bridge, december. 2012.
- [23] Anat Bremler-Barr, Yotam Harchol, David Hay, and Yaron Koral. Deep packet inspection as a service. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 271–282, New York, NY, USA, 2014. ACM.
- [24] Hesham Mekky, Fang Hao, Sarit Mukherjee, Zhi-Li Zhang, and T.V. Lakshman. Application-aware data plane processing in sdn. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 13–18, New York, NY, USA, 2014. ACM.