Project Report

A Specialized Hypervisor For Efficient Middlebox Virtualization

Author:
Mihir J. Vegad

Guide:
Prof. Purushottam Kulkarni

A report submitted in partial fulfilment of the requirements

for the degree of Master of Technology in the

Computer Science and Engineering

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay

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

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of the key factor in the rise of the Data center networking in recent years is, the core system infrastructure such as storage, network devices has become software-defined. Network Function Virtualization decouples the network applications from proprietary hardware and virtualize them. These network functions are usually referred as Middleboxes. To run middleboxes on virtual machines, we require a Virtual Machine Monitor or Hypervisor which allow 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 and we want to virtualize them efficiently without compromising the overall performance or resource utilization.

1.1 Middleboxes and Hypervisors

By definition a middlebox is a networking function that transforms, inspects, filters or manipulates network traffic for some purpose other than packet forwarding. [9] NFV has attracted many service providers to virtualize their network. Virtualized network functions would be located in those data centres along with other network nodes. According to a survey, number of middleboxes deployed in varying size data centres are on par with number of L3 routers and number of L2 switches. [3] Wan optimizers, proxies, application layer gateways, load balancers, intrusion detection system, intrusion prevention system are widely used middleboxes in data centres. Middleboxes play an important role in meeting service level agreements(SLAs) with client. Traffic is normally routed through chain of such middleboxes. All the middleboxes perform some entry level task followed by specific middlebox function followed by packet forwarding. To improve middlebox performance, we can target hypervisor as in the transition from traditional network to NFV, hypervisor is still one component which operates in traditional manner.

The sole aim of virtualization is to run many virtual systems on a single physical system efficiently. Hypervisors are softwares that creates or destroys virtual machines and manages physical resources among them. Two different types of hypervisors, Bare metal hypervisor and Hosted hypervisors are widely used in the market. 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 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.

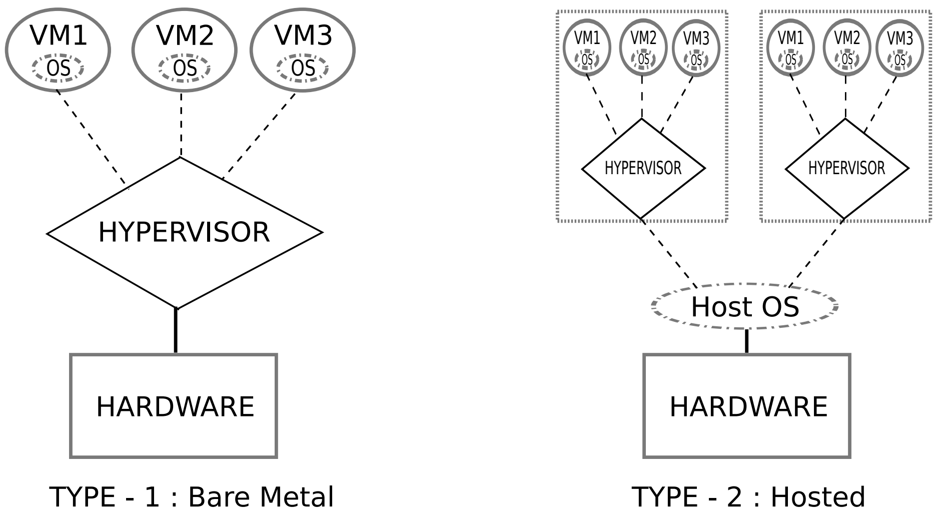

Figure 1.1: Different types of Hypervisor

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 hardware. We have used virtio enabled KVM setup for experiments. Virtio is a standard for network device driver while transmitting or receiving a packet from/to that network device. In this case, network device driver just know that it is running in the virtualized environment.

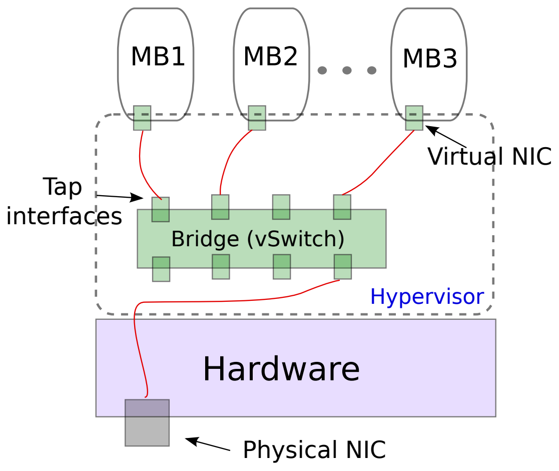

Figure 1.2: packet path from pNIC to middlebox

When a packet is received on the physical network interface card, it is hypervisor's duty to forward it to the specific middlebox hosted on that machine or vice versa. In our KVM setup, how this packet travels from the physical NIC to middlebox or viceversa is very important to understand before we move towards design of the solution. 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. And then at last it is received by the middlebox. This is very brief description of the whole process.

1.2 Motivation: why to focus on middleboxes?

Middleboxes are crucial part of data centre networks as only packet forwarding is not good enough to meet 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.

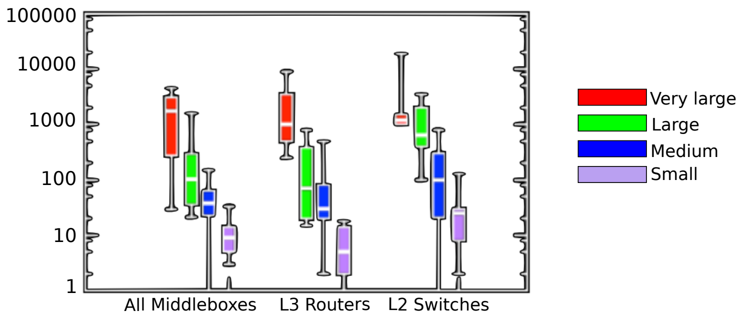

Figure 1.3: MB presence in the data centres [3]

1.3 Problem Description

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 cpu overhead to get maximum performance possible. Some hardware optimization techniques like virtual machine device queue(VMDq), Direct I/O, SR-IOV exist to counter the intervention by hypervisor in high speed networks. But they are costly and they have scalability issues.

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

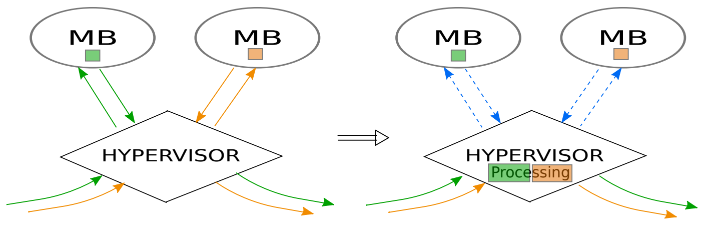

Figure 1.4: MB specific functionalities inside Hypervisor

We will discuss the design in detail in chapter 4.

1.4 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 limitations of the model and the challenges faced.
4. Implemented part of widely used middleboxes like IP/Application layer firewall, IP load balancer, implemented interface between a middlebox and a hypervisor for communication and benchmarked its performance.
5. Documented the work done so far, results achieved and the future work possible.

1.5 Outline of the report

The report is organized in the following manner. In chapter 2, we will discuss several classifications of the middleboxes 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 and challenges faced. In chapter 5, we will discuss the implementation details for a possible solution, experiment setup and results obtained. At last in chapter 6, we will give conclusion and discuss some work which can be explored in future.

Chapter 2

Background

2.1 Middlebox Examples

Some of the widely used middleboxes among the data centres are described here.

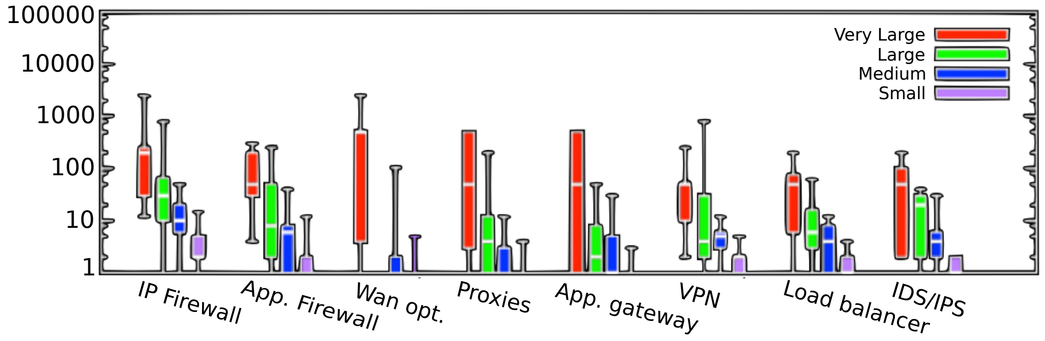

Figure 2.1: Different types of middleboxes in data centres [3]

- **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. [4]

- **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 occuring. [5]

- **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. [9]

- **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.

2.2 Middlebox classification

We have classified the middleboxes based on several parameters to test our proposed solution for all the types of middleboxes.

2.2.1 Based on MB functionality

A middlebox function can be any networking function apart from the routing function.

General MBs

All the middlebox examples discussed in section 2.1 are general network functions which can be used in any of the network as per the requirements. They are usually deployed at the entry point of the networks before actual processing starts. Some examples are firewalls, gateways, load balancers, proxies, intrusion detection/prevention system, traffic shaper.

Application specific MBs

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.

2.2.2 Based on packet processing

Middleboxes can be classified based on how they process the packets to take some action on it. They process the packets at different layers and then take the specified action.

Packet header inspection MBs

Middleboxes belong to this category only inspects some specific field of the packet header and then takes appropriate action specified by the administrator. They are just decision making middleboxes, which just decide whether to drop a packet or to forward it further. Some examples are IP firewall, network monitors, gateways.

Packet header modification MBs

Middleboxes belong to this category modifies some specific field of the packet header and then forwards the packet. It requires more processing compare to the header inspection MBs. Some examples are IP load balancer, NAT, proxies.

Packet body inspection MBs

Middleboxes belong to this category inspects the body of the packet and then takes appropriate action specified by the administrator. They are also decision making middleboxes, which just decide whether to drop a packet or to forward it further. Some examples are application layer firewall, traffic shaper, intrusion detection/prevention system.

Packet body modification MBs

Middleboxes belong to this category modifies the body of the packet and then forwards the packet. It requires more processing compare to the body inspection MBs. Some examples are data compression middleboxes, proxies.

Our model can be applied to any of the middlebox from any category. Second classification is exhaustive classification for middleboxes. The procedure to offload some middlebox functionality remains same for all the middleboxes belonging to the same category. We have experimented on general middleboxes which inspects or modifies the packet header. 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.

- MB Operating System optimizations
- MB Hypervisor optimizations
- MB 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 MB Operating System optimizations

ClickOS[8] 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[2] 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[1] also provides light weight machines to deploy middlebox applications in cloud environment. Unikernels are single purpose appliances, they strips away 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[*] is an example of such a library OS. It is written in OCaml, a functional programming language and it runs on the Xen hypervisor[*]. It also improves scalability and setup latency for the middleboxes.

3.2 MB Hypervisor optimizations

Container[12] 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 hypervior 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[14] 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 efiiciency 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[8] 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 MB Hardware assisted optimizations

Virtual Machine Device queue (VMDq) is part of Intel Virtualization Technology[13] 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[7], 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)[15] 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.[10][11]

3.4 Comparison of all the approaches

Our goal is to achieve performance improvement for the middleboxes runnning on the physical machine in the virtualized environment. Performance of the middleboxes is a very abstact 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 based on the metrics like middlebox setup latency, scalability, multiprocessing support, processor utilization, redundancy among hypervisor and guest os, throughput etc. [Table - 3.1]

We can observe that solutions from each category improves on some of the metrics. They do not affect other metrics. MB 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. MB 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. MB hardware assisted optimizations category solutions offer gains on throughput and cpu utilization. But they do not improve on scalability and portability of VMs.

Our proposed solution is a combination of MB 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.

Metric	**MB OS optimizations**	**MB HYP optimizations**	**MB HW assisted optimizations**
Middlebox setup latency	yes	no	no
Scalability	yes	yes	no
Multiprocessing support	no	yes	no
Throughput	no	yes	yes
Packet processing time	no	yes	yes
Redundancy among hypervisor and guest operating system	no	yes	yes
Reducing the hypervisor intervention	no	yes	yes

Table 3.1: Comparison Table

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 MB specialized Hypervisor

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. First 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 perform the offloaded middlebox functionality and take appropriate action on the packet. Middlebox should inform the hypervisor about which functionality it wants to offload. Middlebox should also be able to dynamically inform the hypervisor about change in its state which may effect the processing of the offloaded functionality. Hypervisor also needs to maintain updated state of the middlebox in order to execute specific offloaded functionality. 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 act on the middlebox state stored on the hypervisor accordingly. Thus the proposed design requires a hook to be added in the packet processing in the hypervisor, an interface for communication on the hypervisor and the middlebox and the state of the middlebox to be stored on the hypervisor. Figure-4.1 represents the overall design we just discussed.

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 Interface between a MB and the Hypervisor

Shared memory

We have used ivshmem, a QEMU virtual PCI device to share space between a middlebox and the hypervisor for communication. It is fairly simple to use[6]. The guest OS can access a POSIX SHM region on the host through the ivshmem device. It emulates memory mapped

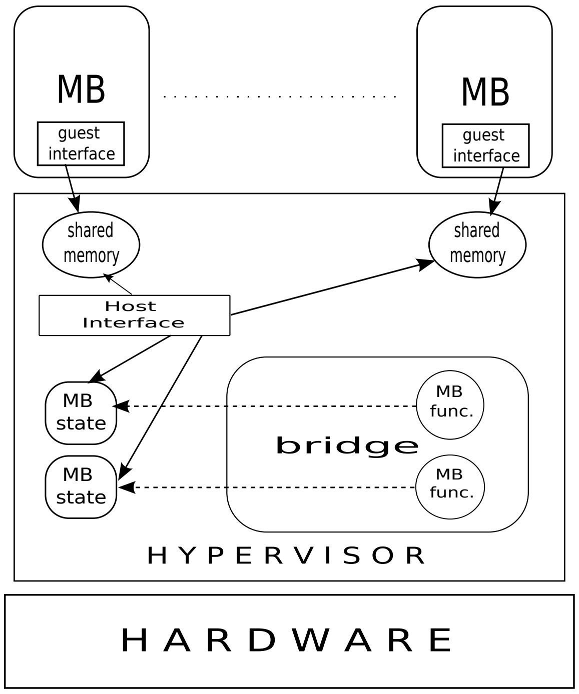

Figure 4.1: Design of MB specialized hypervisor

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 on QEMU command to fire a VM.

$$\$qemu - system - x86\_64\ (...)\ - device\ ivshmem, shm = ivshmem1, size = 1$$

Here, ivshm1 is the name of the POSIX SHM object to use as the shared memory. It is /dev/shm/ivshmem1 here. 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[6] to write on the shared memory from a MB.

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 convey anything to the hypervisor, it does so by writing that information on the shared memory embedded within @s as shown above.

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() systemcall 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[6] as described earlier.

4.2.2 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.

4.2.3 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 that 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 task, 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

In a network set up where middleboxes are virtualized as per the above design, a packet can be processed in three different ways. 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. 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. The cost associated with the other packets is kernel processing on the hypervisor, additional module processing and reduced IRQ processing. 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.

In the next chapter, we will experiment on the middlebox virtualization design that we described here. We will focus on the several metrics like cpu utilization and packet processing time. We will also discuss challenges faced during the experimentation.

Chapter 5

Experiments

5.1 Experimentation setup

5.2 Results

5.2.1 Interrupts generated by packets

5.2.2 Time required to process a packet

5.2.3 Throughput of a client application

5.3 Challenges

Chapter 6

Conclusion and Future work

References

[1] Anil Madhavapeddy, R. M. Unikernels: Library operating systems for the cloud. In *ASPLOS13* (2013).

[2] Avi Kivity, D. L. Osvoptimizing the operating system for virtual machines. In *2014 USENIX Federated Conferences* (2014).

[3] Data centre middleboxes. http://www.cs.cornell.edu/courses/cs5413/2014fa/lectures/26-datacenter-middleboxes.pdf.

[4] Firewall middlebox. https://en.wikipedia.org/wiki/Firewall_(computing).

[5] IDS/IPS middleboxes. https://en.wikipedia.org/wiki/Intrusion_detection_system.

[6] Writing a PCI Device Driver, A Tutorial with a QEMU Virtual Device. http://nairobi-embedded.org/linux_pci_device_driver.html.

[7] Jose Renato Santos, Y. T. Bridging the gap between software and hardware techniques for i/o virtualization. In *USENIX* (2008).

[8] Martins, J., and Ahmed, M. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014).

[9] Middleboxes. https://en.wikipedia.org/wiki/Middlebox.

[10] What is SR-IOV? http://blog.scottlowe.org/2009/12/02/what-is-sr-iov/.

[11] What are SR-IOV pros and cons? http://searchwindowsserver.techtarget.com/answer/What-are-some-benefits-and-limitations-of-Windows-Server-SR-IOV.

[12] Stephen Soltesz, H. P. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors.

[13] Intel VMDq Virtualization Technology. https://www.vmware.com/files/pdf/partners/intel/vmdq-white-paper-wp.pdf.

[14] Xin Jin, J. G. Covisor: A compositional hypervisor for software-defined networks. In *12th USENIX symposium on networked system design and implementation* (2014).

[15] Yaozu Dong, X. Y. High performance network virtualization with sriov. In *IEEE Xplore* (2009).