National Technical University Of Athens
Department Of Electrical
And Computer Engineering

COMPUTER SCIENCE DIVISION
COMPUTING SYSTEMS LABORATORY

Co-existing Scheduling Policies Boosting I/O Virtual Machines

DIPLOMA THESIS

Dimitris Aragiorgis

Supervisor: Nectarios Koziris
Associate Professor of N.T.U.A.

Athens, July 2011

NATIONAL TECHNICAL
UNIVERSITY OF ATHENS
DEPARTMENT OF ELECTRICAL AND
COMPUTER ENGINEERING
COMPUTER SCIENCE DIVISION
COMPUTING SYSTEMS LABORATORY

Coexisting Scheduling Policies Boosting I/O Virtual Machines

DIPLOMA THESIS

Dimitris Aragiorgis

Supervisor: Nectarios Koziris
Associate Professor of N.T.U.A.

Approved by the committee on 29th of July 2011.

....	....	....
Nectarios Koziris Associate Professor NTUA	Nikolaos Papaspyrou Assistant Professor NTUA	Konstantinos Sagonas Associate Professor NTUA

Athens, July 2011

....

Dimitris Aragiorgis
Electrical and Computer Engineer N.T.U.A.

Copyright © Dimitris Aragiorgis, 2011
(2011) National Technical University of Athens. All rights reserved.

Abstract

Deploying multiple Virtual Machines (VMs) running various types of workloads on current many-core cloud computing infrastructures raises an important issue: The Virtual Machine Monitor (VMM) has to efficiently multiplex VM accesses to the hardware. We argue that altering the scheduling concept can optimize the system's overall performance.

Currently, the Xen VMM achieves near native performance multiplexing VMs with homogeneous workloads. Yet having a mixture of VMs with different types of workloads running concurrently, it leads to poor I/O performance. Taking into account the complexity of the design and implementation of a universal scheduler, let alone the probability of being fruitless, we focus on a system with multiple scheduling policies that coexist and service VMs according to their workload characteristics. Thus, VMs can benefit from various schedulers, either existing or new, that are optimal for each specific case.

In this paper, we design a framework that provides three basic coexisting scheduling policies and implement it in the Xen paravirtualized environment. Evaluating our prototype we experience 2.3 times faster I/O service and link saturation, while the CPU-intensive VMs achieve more than 80% of current performance.

Keywords Paravirtualization, Service-oriented VM containers, I/O, Resources Utilization, Multi-Core Platforms, Scheduling, SMP, Multiple GigaBit NICs.

Acknowledgements

This thesis would not come to an end without the help and effort of many people. I would like to show my respect and gratitude to my supervisor A. Nanos who patienlty walked me through this period of time giving me great hints an helping me with any issues that arose, not to mention the great effort we gave together in order to get this work published in a paper submitted to VHPC'11.

Many thanks to N. Kozyris for giving me the opertunity to work in CSLab and providing all resources needed to complete my thesis.

Last but not least, the most credit goes to my father, who throughout his life tried to communicate me his wisdom an knowlegde, something that absolutely contributed to my achievements. Furthermore he provided me all the infrastructure needed for the testbed and walked me through the evaluation process.

Dimitris Aragiorgis
Athens, 25th July 2011

Contents

Contents v

List of Figures ix

List of Tables xi

1 Introduction 1

1.1 Virtualization: Virtues and Features 1
1.2 I/O and Scheduling in a Virtualized Environment 2
1.3 Motivation 2
1.4 Thesis Contributions 3

2 Background 5

2.1 Basic Platform Components 5
2.1.1 Core Components 5
2.1.2 Device Related Components 8
2.2 What is Virtualization? 12
2.2.1 Why Virtualize? 14
2.2.2 Known Virtualization Techniques 15
2.2.3 Which to choose and why? 17
2.3 Common Virtualization Platforms 17
2.3.1 QEMU 17
2.3.2 KVM 18
2.3.3 Xen 19
2.4 Virtualization Issues 20

2.4.1 Virtualizing CPU 20
2.4.2 Virtualizing Memory 21
2.4.3 Virtualizing Devices 33
2.5 Xen Internals in Paravirtualized Mode 38
2.5.1 Time–keeping 39
2.5.2 Xenstore 40
2.5.3 Event Channel Mechanism 41
2.5.4 I/O Rings 41
2.5.5 The Split Driver Model 43
2.6 Network I/O Path 44
2.6.1 In Native OS 44
2.6.2 In a ParaVirtualized Environment 49
2.7 Scheduling 51
2.7.1 Linux's CFS 52
2.7.2 Xen's Credit Scheduler 53
2.7.3 CPU pools 55

3 Related Work 58
3.1 HPC Setups 58
3.1.1 Software Approaches 58
3.1.2 Hardware Approaches 58
3.2 Service–oriented Setups 58

4 Design and Implementation 60
4.1 Monitoring Tool 60
4.2 The *no-op* Scheduler 62
4.3 Credit Optimizations for I/O service 67
4.3.1 Time-slice allocation 68
4.3.2 Anticipatory concept 68
4.4 CPU pools 69

5 Towards Distinctive Scheduling Policies and Evaluation 71
5.1 Exploring Vulnerabilities of Current Default Xen's Setup 72
5.2 The Driver Domain Pool 75
5.3 Resources distribution vs. VM Performance 77
5.4 Decoupling vCPUs based on workload characteristics . . 77

6 Conclusions and Future Work 80

Bibliography 82

List of Figures

2.1 Motherboard Schematic 6
2.2 System-level IA-32 architecture overview 9
2.3 Generic Motherboard Components 11
2.4 Call Gates 23
2.5 Guest/Shadow Page Tables 26
2.6 I/O rings: Xen's implementation for interdomain data exchange 42
2.7 Xen's network split driver model 43
2.8 Linux high-level network stack architecture 45
2.9 Internet protocol array structure 46
2.10 Socket buffer data structure 47
2.11 Netfront-Netback interaction 49

5.1 Overall Performance of the Xen Default Case 73
5.2 Monitoring Tool for the Default Setup: msecs lost per MB transmitted 74
5.3 Monitoring Tool Results (msec lost per MB transmitted): (a) default Setup; (b) 2 pools Setup 76
5.4 Overall Performance: default Setup; 2 pools Setup 76
5.5 Overall Performance vs. Physical Resources Distribution to VM pool 77
5.6 Overall Performance: default Setup; 2 pools Setup; 3 pools Setup 79

List of Tables

2.1 Subset of Privileged Instructions 7

5.1 VM Misplacement effect to individual Performance . . . 78

Chapter 1

Introduction

1.1 Virtualization: Virtues and Features

Currently, modern technology, features powerful platforms that can be grouped together with high–performance interconnects and provide HPC infrastructures that are necessary mostly for scientific applications which require great computing power. On the other hand, these infrastructures can be used in terms of Cloud Computing where more service oriented requirements take place. The system is considered as a black box, that should offer the QoS requested. For instance a client can ask for a system with 4 CPUs, 4GB Memory, 1Gbps NIC. To dedicate such a machine to a modern multi-core platform would be a waste of resources. To find one that fulfill these requirements would be impractical. Here intervenes the Virtualization. It allows the co-existence of multiple machine instances in a single container, while guaranteeing isolation and security. Machines with limited demands can run concurrently utilizing the systems resources more efficiently and trying to make the most of them. Moreover, the ability to create and destroy instances of machines on demand, live adding or removing resources, live migration for load balancing, easier charge of service providing are the predominant reasons why virtualization is attractive and desirable.

There are numerous virtualization platforms that have been designed the past few years [1]. Among them most common are VMware ESX, KVM with QEMU, and Xen. The first two affiliate the principles of full Virtualization while the latter adopts the ParaVirtualization scheme. Both have their pros and cons: Synoptically, Full– allows unmodified OS

[1] http://en.wikipedia.org/wiki/Comparison_of_platform_virtual_machines

to run but introduces overhead while emulating the exported hardware to the VM; Para– runs a virtualization–aware kernel but is more lightweight that the other way.

1.2 I/O and Scheduling in a Virtualized Environment

Recent advances in virtualization technology have minimized overheads associated with CPU sharing when every vCPU is assigned to a physical core. As a result, CPU–bound applications achieve near-native performance when deployed in VM environments. However, I/O is a completely different story: intermediate virtualization layers impose significant overheads when multiple VMs share network or storage devices. Numerous studies present significant optimizations on the network I/O stack using software or hardware approaches.

These studies attack the HPC case, where no CPU over-commitment occurs. However, in service-oriented setups, vCPUs that belong to a vast number of VMs and run different types of workloads, need to be multiplexed. In such a case, scheduling plays an important role.

Just like the OS has to multiplex processes and tries to offer a fair time–sharing among them, the VMM must be able to multiplex numerous VMs and provide them a time–slice to eventually execute their applications. The optimal scheduling policy should provide the desired Quality of Service to each VM while trying to fully utilize the host resources.

1.3 Motivation

Currently, the Xen VMM with the Credit scheduler does a good job in time-sharing of VM with similar CPU–intensive workloads. But in case of VMs with different workloads running concurrently, I/O gets neglected mostly because of the Credit's algorithm and of the fact that in ParaVirtualization techniques, not only the VM making the I/O participates in the transaction. Specifically, both this VM and the driver domain need to be scheduled in so that the VM can communicate with the rest of the world. Additionally VMs that coexist in a VM container usually run various applications that have different characteristics. CPU–bound, Memory–bound, random I/O, heavy I/O, low latency, real-time applications, etc. Taking into account the complexity of the design and

implementation of a universal scheduler, let alone the probability such an attempt to be fruitless, we focus on a system with multiple scheduling policies that coexist and service VMs according to their workload characteristics. Thus, VMs can benefit from various schedulers, either existing or new, that are optimal for each specific case.

1.4 Thesis Contributions

Scheduling Concept The main contribution of this work is the evaluation of the scheduling concept where different scheduling policies co–exist and contradicting VMs are decoupled from each other and VMs with similar workload characteristics are multiplexed by the corresponding policy which is tailored to their needs. With our framework we take the first step towards this concept and divide the VMs in two different types; CPU– and I/O– intensive VMs. Implementing this in the ParaVirtualized environment of Xen VMM we point out how I/O VMs can benefit from the isolation provided. We experience network link saturation in contrast to less than 40% utilization of the current case. On the other hand, CPU operations sustained more than 80% of default performance.

Monitoring Tool In a ParaVirtualized I/O path, as mentioned earlier, participate not only the VM that makes the I/O operation but the driver domain as well. So this is the part where the scheduler interferes. To explore the time lost between the actual moments each domain gets scheduled in an serves the I/O request, we build a monitoring tool that traces down the transactions carried out between these domains. It is build on top of Xen's event channel mechanism based on wall clock timestamps. It is a helpful tool that can be triggered on demand while adding negligible overhead. Eventually data are processed and the average msec lost per MB transmitted is calculated.

SMP aware no-op Scheduler In order to decouple the driver domain from the other domains, we take advantage the modular implementation of Xen's scheduling, and build a primitive ,,no-op" SMP-aware scheduler that bounds every newly created vCPU to an available physical core, if any, without preempting it and so avoiding any context switch and frequent scheduling calls. This scheduler can apply to any domain with any number of vCPUs.

Credit Scheduler Optimization for I/O service We evaluate the benefit I/O domains can gain when the time–slice offered by the scheduler gets decreased from 30ms to 3ms, and how it gets affected depending of the type of I/O (random / heavy) and the size of packets transmitted.

An anticipatory scheduling concept proposal To take advantage the high probability, that an I/O operation follows another in the near future, we propose an anticipatory scheduling concept based on the current scheduling algorithm and implemented with multi-hierarchical priority set that lets the VM sustain for a bit longer the boost state and be temporarily favored among others.

A profiling mechanism proposal After having proved that the co-existing scheduling policies can benefit I/O performance and resources utilization we have to examine how such a scenario can be automated or adaptive. How to implement the VM classification and the resources partitioning? Upon this we consider the following design dilemma; the profiling tool should reside in the driver domain or in the Hypervisor? The former is aware of the I/O characteristics of each VM while the latter can keep track of their time-slice utilization. Either way such a mechanism should be lightweight and its actions should respond to the average load of the VM and not to random spikes.

Chapter 2

Background

In this work we address the way the hypervisor in a virtualized environment affects the I/O performance of Virtual Machines and the overall resources utilization. Specifically we choose to focus on the scheduling mechanism the hypervisor uses to multiplex the Virtual CPUs in order to provide them a timeslice to run. To this end in this chapter we are going to mention the components that participate in our study. Specifically we give the big picture about virtualization, we refer to some hardware details including the features modern platforms provide , how the operating system takes advantage of them, and how they are eventually exported in a virtualized environment. We mention the different ways of virtualization techniques, how QEMU, KVM and Xen address the challenges that arrise and conclude with some implementation details regarding Xen and Linux internals needed for the understanding of this work.

2.1 Basic Platform Components

Figure 2.1 depicts a generic Intel platform. In the following subsections we will refer to the system components our work deals with so that the rest it can be more understandable.

2.1.1 Core Components

CPU

Every architecture suports a set of intructions (ISA). The final code that is to be executed on metal is a series of these instructions. How this

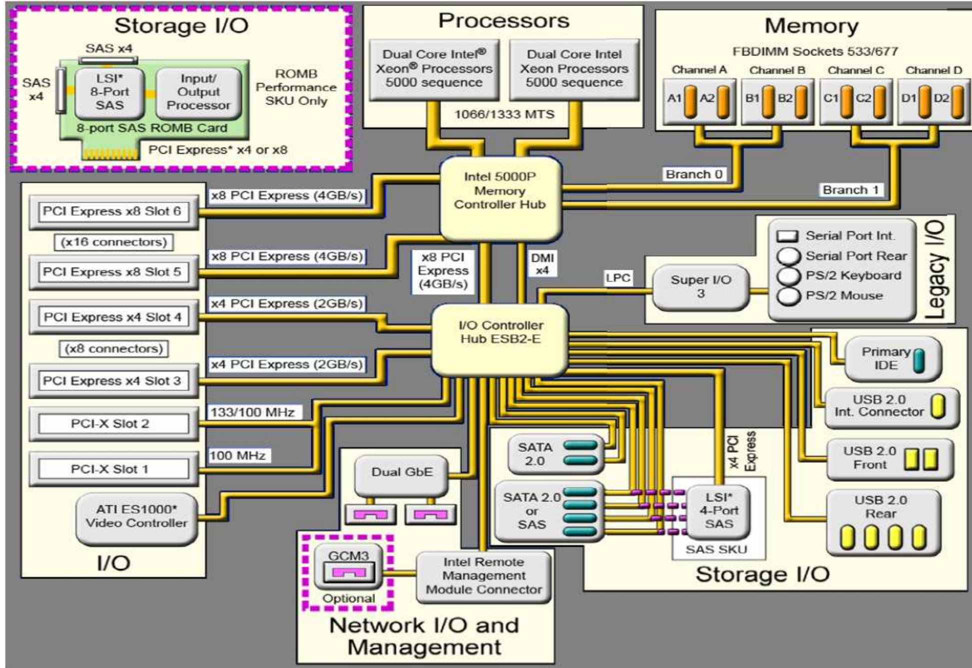


Figure 2.1: Motherboard Schematic

code (source code ⇒ compiler ⇒ linker ⇒ assembler ⇒ binary code) is produced is out of the scope of this work.

The CPU is responsible for executing binary code streams, i.e. instructions belonging to its ISA. ISA includes a special set of privileged instructions (Table 2.1). The CPU itself features different privilege levels (Privilege Rings 0-3). Specifically in every execution context the CPU is set to a privilege level (CPL) and in case it tries to execute a more privileged instruction a General Protection (GP) Fault gets raised. To overcome this a system call must be invoked instead, which is done via call gates in hardware level.

CPL is found in the CS register - descriptor selector (which points to a segment descriptor and eventually to a memory segment where the actual code is located). We will get more into this in the following section discussing memory virtualization.

The important thing is that during bootstrap the kernel code is set to be in the highest privilege level and is the one who can determine the privileges of the rest execution codes (processes). To this end it is used to say that the kernel resides in the inner privilege ring while the rest (user processes etc.) reside in the outer ring.

Some other instructions like IN and OUT require a lower privilege level

Privileged Level Instructions	
Instruction	Description
LGDT	Loads an address of a GDT into GDTR
LLDT	Loads an address of a LDT into LDTR
MOV *Control Register*	Copy data and store in Control Registers
INVD	Invalidate Cache without writeback
INVLPG	Invalidate TLB Entry
RDTSC	Read time Stamp Counter

Table 2.1: Subset of Privileged Instructions

but still user mode code cannot execute them. This kind of instructions usually are needed by device drivers, therefor it is used to say that device drivers reside in the middle privilege rings, between kernel and user processes.

The transition from user level to kernel level is done via call gates and invoking system calls. This means that the kernel, who has the ultimate privilege will service the user process demand without letting it interfere in system level components (page tables, control registers, memory, etc.) and compromizing system security and isolation between other processes. That means that the kernel acts like a supervisor who multiplexes the hardware accesses of the multiple processes running concurrently in a Computing System (one OS + many applications).

MMU

The main memory (DRAM) exports a continuous address space to the system. Every access to the memory cells is done via the physical address. CPU is unaware of the physical memory arrangement. It references logical (virtual) addresses. Logical addresses are included in machine language instruction (generated by compilers) and consist of a segment and an offset. The Memory Management Unit (MMU) is responsible for virtual-to-physical transalation. Specifically it includes:

- A segmentation unit that translates logical into linear.
- A paging unit that translates linear into physical.
- A radix tree ,the page table, encoding the virtual-to-physical translation. This tree is provided by system software on physical memory, but is rooted in a hardware register (the CR3 register)

- A mechanism to notify system software of missing translations (page faults)
- An on-chip cache (the translation lookaside buffer, or TLB) that accelerates lookups of the page table
- Instructions for switching the translation root in order to provide independent address spaces
- Instructions for managing the TLB

Modern commodity hardware provide hardware support for page table walk. The OS needs to load the CR3 with the physical address of the root page directory and every linear to physical translation is done by page walker.

Figure 2.2 shows a system-level architecture overview and specifically the registers state and memory structs an OS has to keep up to date.

System Bus

The system bus (Front Side Bus) connects the CPU to the rest of components. Addresses (physical addresses calculated by the MMU) and data related to the CPU are passed over the system bus.

2.1.2 Device Related Components

APIC

Advanced Programmable Interupt Controller is a component that multiplexes all external interupts (originated from the devices) and propagates them to the CPU interupt pin. It lets the CPU know where the interupt came from by writing the corresponding vector in a special register. Depending on that the OS looks up the Interrupt Descriptor Table and finds the proper Interupt Service Routing to execute (that acknowlegdes the interrupt, defers the actual handler who propably does DMA and exits).

North Bridge

The North Bridge (Memory Controller Hub) is the system's component that brings high-speed hardware together. Specifically, then north brigde is connects CPUs (via FSB), DRAM (via Memory Bus), PCIe (via PCIe

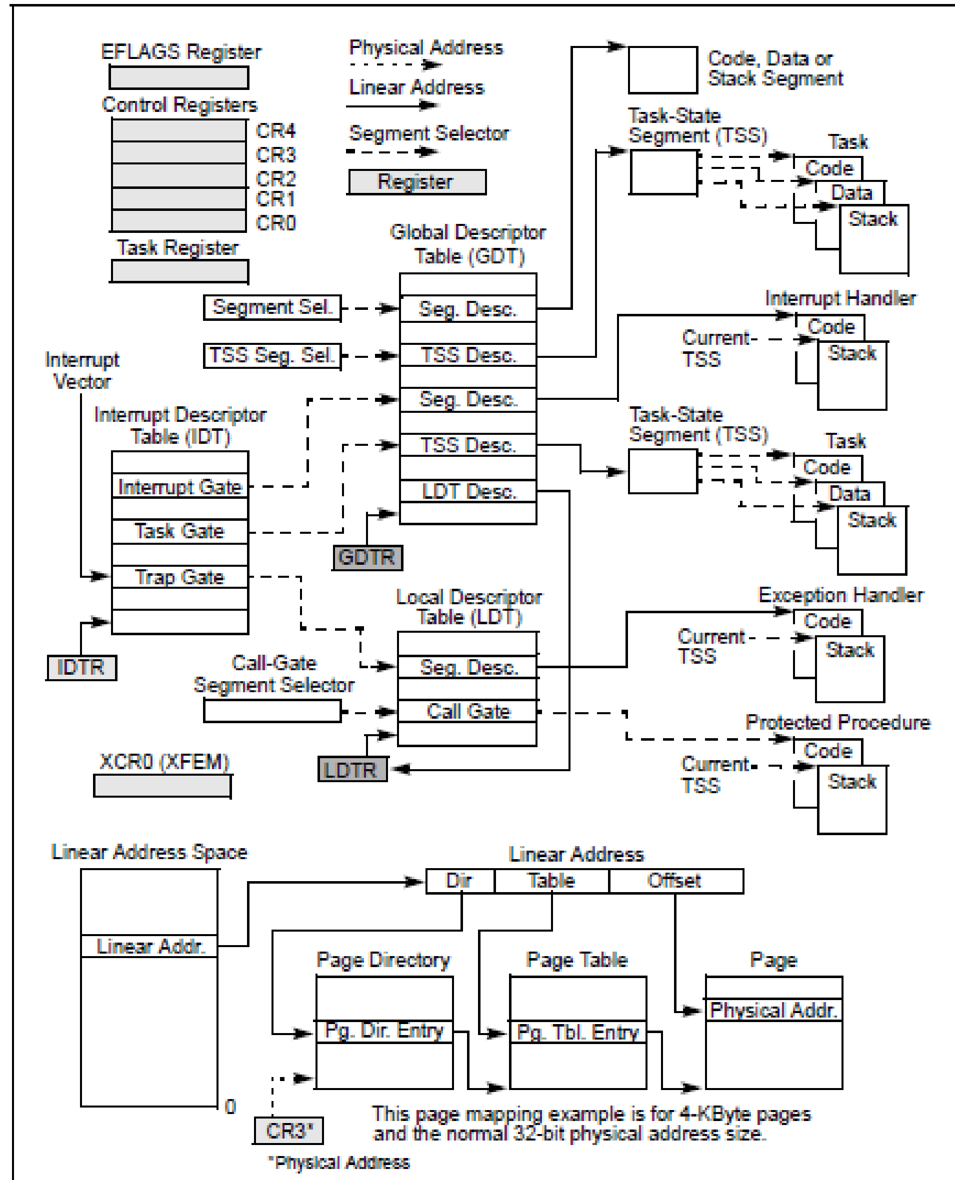


Figure 2.2: System-level IA-32 architecture overview

Root Ports), and South Bridge (via internal bus). It is responsible for all the logic that results to correct accesses. In example when the CPU references an address that its memory is mapped to a PCI configuration space it should not access the main memory but the corresponding PCI address space.

PCI express Conceptually, the PCIe bus is like a high-speed serial replacement of the older PCI/PCI-X bus, an interconnect bus using shared address/data lines. During device detection, PCI configuration space (registers and memory of the device) get mapped to physical address space (memory mapped I/O). This addresses get mapped to kernel virtual address space by the OS. From this point the CPU can reference these virtual addresses and get access to the device registers.

South Brigde The South Bridge (I/O controller Bus) brings low-speed hardware together such as PCI, USB, ISA, IDE, BIOS and other legacy devices. This is out of the scope of this work so we wont get into details.

IOMMU An input/output memory management unit (IOMMU) is a memory management unit (MMU) that connects a DMA-capable I/O bus to the main memory. Like a traditional MMU, which translates CPU-visible virtual addresses to physical addresses, the IOMMU takes care of mapping device-visible virtual addresses (also called device addresses or I/O addresses in this context) to physical addresses. An example IOMMU is the graphics address remapping table (GART) used by AGP and PCI Express graphics cards.

The advantages of having an IOMMU, compared to direct physical addressing of the memory, include:

- Large regions of memory can be allocated without the need to be contiguous in physical memory the IOMMU will take care of mapping contiguous virtual addresses to the underlying fragmented physical addresses. Thus, the use of vectored I/O (scatter-gather lists) can sometimes be avoided.
- For devices that do not support memory addresses long enough to address the entire physical memory, the device can still address the entire memory through the IOMMU. This avoids overhead associated with copying buffers to and from the memory space the peripheral can address.

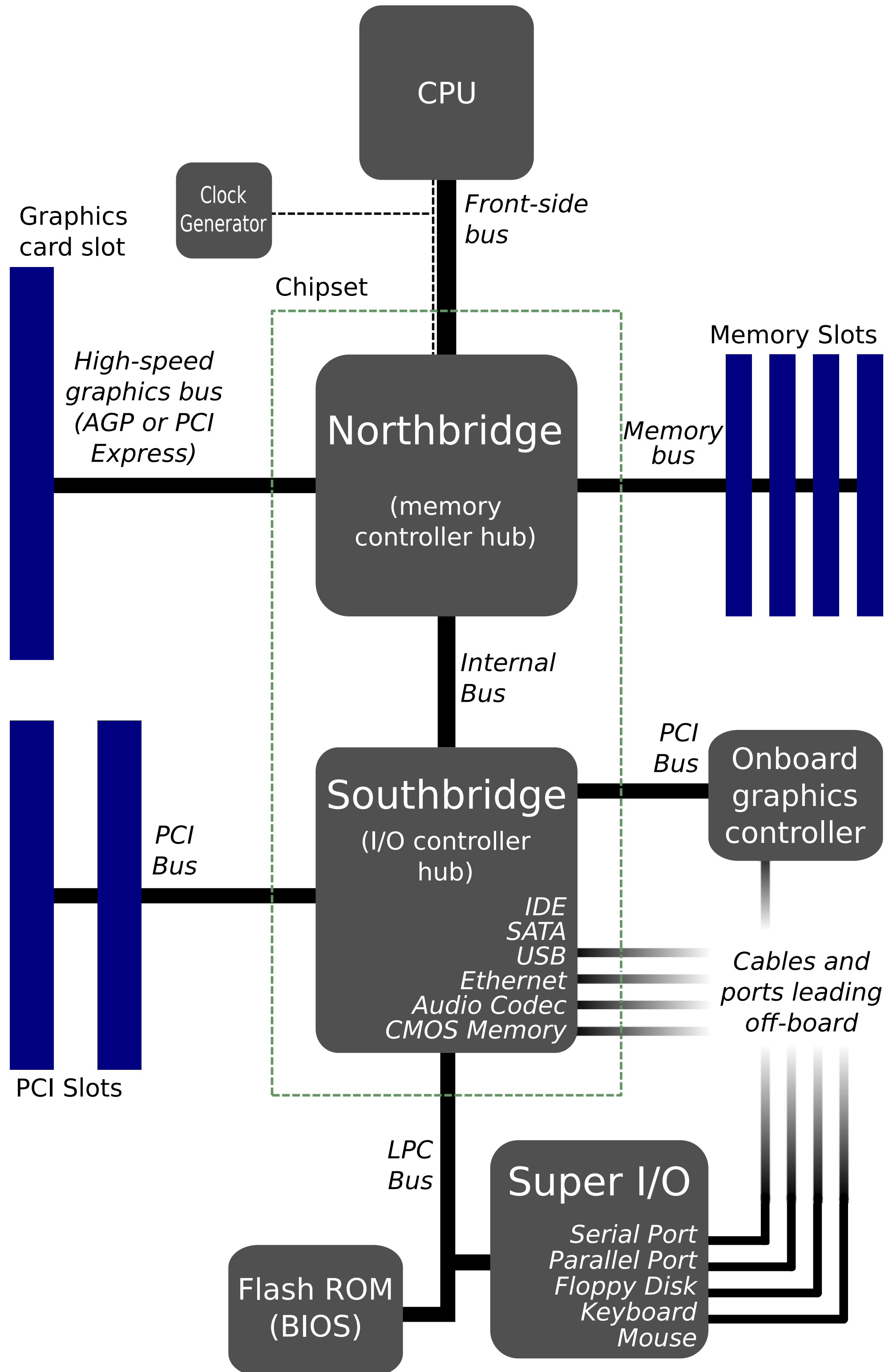


Figure 2.3: Generic Motherboard Components

- Memory protection from malicious or misbehaving devices: a device cannot read or write to memory that hasn't been explicitly allocated (mapped) for it.
- With virtualization, guest operating systems can use hardware that is not specifically made for virtualization. Higher performance hardware such as graphics cards use DMA to access memory directly; in a virtual environment all the memory addresses are remapped by the virtual machine software, which causes DMA devices to fail. The IOMMU handles this remapping, allowing for the native device drivers to be used in a guest operating system.
- In some architectures IOMMU also performs hardware interrupt remapping, in a manner similar to standard memory address remapping.

2.2 What is Virtualization?

With Virtualization basicaly a system pretends to be more of the same system. It must not be confused with emulation where a system pretends to be another one.

One can argue that modern operating systems feature some sort of virtualization; each process is not aware of other running and can access the resources (CPU, memory, devices) as it was alone in the system. So from the process's point of view it feels having a dedicated system serving its demands. This to apply, certain mechanisms take place: processes are allowed to run on the CPU and eventually get preempted according to the scheduling policy; they have the illusion of flat memory address space (virtual memory); they can use the sockets API to access a network device without having to worry about other applications sharing the same device.

In terms of Virtualization, whole machines (resources, OS and applications) are considered by the Virtualization platform as processes are by the OS. They should not be aware of co-existing machines and isolation and fairness among them should be guaranteed.

CPU Virtualization

Virtualizing a CPU is, to some extent, very easy. A process runs with exclusive use of it for a while, and is then interrupted. The CPU state

is then saved, and another process runs. After a while, this process is repeated.

This process typically occurs every 10ms or so in a modern operating system. It is worth noting, however, that the virtual CPU and the physical CPU are not identical. When the operating system is running, swapping processes, the CPU runs in a privileged mode. This allows certain operations, such as access to memory by physical address, that are not usually permitted. For a CPU to be completely virtualized, Popek and Goldberg in [1] put forward a set of requirements that must be met. They began by dividing instructions into three categories:

Privileged instructions are defined as those that may execute in a privileged mode, but will trap if executed outside this mode.

Control sensitive instructions are those that attempt to change the configuration of resources in the system, such as updating virtual to physical memory mappings, communicating with devices, or manipulating global configuration registers.

Behavior sensitive instructions are those that behave in a different way depending on the configuration of resources, including all load and store operations that act on virtual memory.

In order for an architecture to be virtualizable all sensitive instructions must also be privileged instructions. Intuitively, this means that a hypervisor must be able to intercept any instructions that change the state of the machine in a way that impacts other processes.

I/O Virtualization

An operating system requires more than a CPU to run; it also depends on main memory, and a set of devices. Virtualizing memory is relatively easy; it can just be partitioned and every privileged instruction that accesses physical memory trapped and replaced with one that maps to the permitted range. MMU performs these translations, typically based on information provided by an operating system.

On the other hand most other devices are not designed with virtualization in mind, thus supporting virtualization could be more compicated. When an operating system is running inside a virtual machine, it does not usually know the host-physical addresses of memory that it accesses. This makes providing direct access to the computer hardware difficult,

because if the guest OS tried to instruct the hardware to perform a direct memory access (DMA) using guest-physical addresses, it would likely corrupt the memory, as the hardware does not know about the mapping between the guest-physical and host-physical addresses for the given virtual machine. The corruption is avoided because the hypervisor or host OS intervenes in the I/O operation to apply the translations; unfortunately, this delays the I/O operation.

IOMMU can solve this problem by re-mapping the addresses accessed by the hardware according to the same (or a compatible) translation table that is used to map guest-physical address to host-physical addresses.

2.2.1 Why Virtualize?

The basic motivation for virtualization is the same as that for multitasking operating systems; computers have more processing power than one task needs. As so vitrualization made it possible to take advantage of unused computing resources the modern platforms can provide.

Virtualization allows a number of virtual servers to be consolidated into a single physical machine, without losing the security gained by having completely isolated environments. It allows providers to supply customers their own virtual machines with desired requirements without demanding new physical machine taking up rack space in the data center.

A virtual machine gets certain features, like cloning, at a very low cost.

Another big advantage is migration. A virtual machine can be migrated to another host if the hardware begins to experience faults, or if an upgrade is scheduled. It can then be migrated back when the original machine is working again.

Power usage also makes virtualization attractive. An idle server still consumes power. Consolidating a number of servers into virtual machines running on a smaller number of hosts can reduce power costs considerably. A virtual machine is more portable than a physical one. One can save the state of a virtual machine onto a USB flash drive, transport it easily and then just plug it in and restore.

Finally, a virtual machine provides a much greater degree of isolation than a process in an operating system. This makes it possible to create virtual appliances: virtual machines that just provide a single service to a network. A virtual appliance, unlike its physical counterpart, doesnt take up any space, and can be easily duplicated and run on more nodes

if it is too heavily loaded (or just allocated more runtime on a large machine).

2.2.2 Known Virtualization Techniques

Although x86 is difficult to virtualize, it is also a very attractive target, because it is so widespread. A few solutions that overcome its limitations have been proposed.

Binary Rewriting

One approach, popularized by VMWare, is binary rewriting. This has the nice benefit that it allows most of the virtual environment to run in userspace, but imposes a performance penalty.

The binary rewriting approach requires that the instruction stream be scanned by the virtualization environment and privileged instructions identified. These are then rewritten to point to their emulated versions.

Performance from this approach is not ideal, particularly when doing anything I/O intensive. Aggressive caching of the locations of unsafe instructions can give a speed boost, but this comes at the expense of memory usage. Performance is typically between 80-97% that of the host machine, with worse performance in code segments high in privileged instructions.

A virtualization environment that uses this technique is implemented like a debugger. It inserts breakpoints on any jump and on any unsafe instruction. When it gets to a jump, the instruction stream reader needs to quickly scan the next part for unsafe instructions and mark them. When it reaches an unsafe instruction, it has to emulate it.

Paravirtualization

Paravirtualization systems like Xen rather than dealing with problematic instructions, simply ignore them.

If a guest system executes an instruction that doesnt trap , then the guest has to deal with the consequences. Conceptually, this is similar to the binary rewriting approach, except that the rewriting happens at compile time (or design time), rather than at runtime. The environment presented to a Xen guest is not quite the same as that of a real x86

system. It is sufficiently similar, however, in that it is usually a fairly simple task to port an operating system to Xen.

From the perspective of an operating system, the biggest difference is that it runs in ring 1 on a Xen system, instead of ring 0. This means that it cannot perform any privileged instructions. In order to provide similar functionality, the hypervisor exposes a set of hypercalls that correspond to the instructions.

A hypercall is conceptually similar to a system call . An obsolete implementation is using interrupt 80h for system calls and 82h for hypercalls. Currently hypercalls are issued via an extra layer of indirection. The guest kernel calls a function in a shared memory page (mapped by the hypervisor) with the arguments passed in registers. This allows more efficient mechanisms to be used for hypercalls on systems that support them, without requiring the guest kernel to be recompiled for every minor variation in architecture. Newer chips from AMD and Intel provide mechanisms for fast transitions to and from ring 0. This layer of indirection allows these to be used when available.

Hardware Assisted Virtualization

x86 hardware is notoriously difficult to virtualize. Some instructions that expose privileged state do not trap when executed in user mode, e.g. popf. Some privileged state is difficult to hide, e.g. the current privilege level, or cpl.

Now, both Intel and AMD have added a set of instructions that makes virtualization considerably easier for x86. AMD introduced AMD-V, formerly known as Pacifica, whereas Intels extensions are known simply as (Intel) Virtualization Technology (IVT or VT). The idea behind these is to extend the x86 ISA to make up for the shortcomings in the existing instruction set.

Summarily the virtualization extensions support:

- A new guest operating mode the processor can switch into a guest mode, which has all the regular privilege levels of the normal operating modes, except that system software can selectively request that certain instructions, or certain register accesses, be trapped.

- Hardware state switch when switching to guest mode and back, the hardware switches the control registers that affect processor operation modes, as well as the segment registers that are difficult

to switch, and the instruction pointer so that a control transfer can take effect.

- Exit reason reporting when a switch from guest mode back to host mode occurs, the hardware reports the reason for the switch so that software can take the appropriate action.

2.2.3 Which to choose and why?

In some cases, hardware virtualization is much faster than doing it in software. In other cases, it can be slower. Programs such as VMWare now use a hybrid approach, where a few things are offloaded to the hardware, but the rest is still done in software.

When compared to paravirtualization, hardware assisted virtualization, often referred to as HVM (Hardware Virtual Machine), offers some trade-offs. It allows the running of unmodified operating systems. This can be particularly useful, because one use for virtualization is running legacy systems for which the source code may not be available. The cost of this is speed and flexibility. An unmodified guest does not know that it is running in a virtual environment, and so cant take advantage of any of the features of virtualization easily. In addition, it is likely to be slower for the same reason.

Nevertheless, it is possible for a paravirtualization system to make some use of HVM features to speed up certain operations. This hybrid virtualization approach offers the best of both worlds. Some things are faster for HVM-assisted guests, such as system calls. A guest in an HVM environment can use the accelerated transitions to ring 0 for system calls, because it has not been moved from ring 0 to ring 1. It can also take advantage of hardware support for nested page tables, reducing the number of hypercalls required for virtual memory operations. A paravirtualized guest can often perform I/O more efficiently, because it can use lightweight interfaces to devices, rather than relying on emulated hardware. A hybrid guest combines these advantages.

2.3 Common Virtualization Platforms

2.3.1 QEMU

QEMU [2] is a fast machine emulator using an original portable dynamic translator. It emulates several CPUs (x86, PowerPC, ARM and Sparc)

on several hosts (x86, PowerPC, ARM, Sparc, Alpha and MIPS). QEMU supports full system emulation in which a complete and unmodified operating system is run in a virtual machine and Linux user mode emulation where a Linux process compiled for one target CPU can be run on another CPU.

QEMU is a machine emulator: it can run an unmodified target operating system (such as Windows or Linux) and all its applications in a virtual machine.

QEMU also integrates a Linux specific user mode emulator. It is a subset of the machine emulator which runs Linux processes for one target CPU on another CPU. It is mainly used to test the result of cross compilers or to test the CPU emulator without having to start a complete virtual machine.

2.3.2 KVM

After adding virtualization capabilities to a standard Linux kernel and taking advantage of the advent of virtualization extensions of modern hardware architecture every virtual machine under this model is a regular Linux process scheduled by the standard Linux scheduler. Its memory is allocated by the Linux memory allocator, with its knowledge of NUMA and integration into the scheduler. A normal Linux process has two modes of execution: kernel and user. kvm adds a third mode: guest mode (which has its own kernel and user modes, but these do not interest the hypervisor at all).

The division of labor among the different modes is:

- Guest mode: execute non-I/O guest code
- Kernel mode: switch into guest mode, and handle any exits from guest mode due to I/O or special instructions.
- User mode: perform I/O on behalf of the guest.

KVM consists of two components:

- A device driver for managing the virtualization hardware; this driver exposes its capabilities via a character device /dev/kvm
- A user-space component for emulating PC hardware; this is a lightly modified qemu process

2.3.3 Xen

Xen sits between the OS and the hardware, and provides a virtual environment in which a kernel can run. The main difference is that the OS kernel is evicted from ring 0 and replaced by the hypervisor; in x86 OS is put in ring 1, while in x86_64 it is sharing ring 3 along with the application.

The purpose of a hypervisor is to allow guests to be run. Xen runs guests in environments known as domains, which encapsulate a complete running virtual environment. When Xen boots, one of the first things it does is load a Domain 0 (dom0) guest kernel. Domain 0 is the first guest to run, and has elevated privileges. In contrast, other domains are referred to as domain U (domU) - the U stands for unprivileged.

The privileged Domain

Domain 0 is very important to a Xen system. Xen does not include any device drivers by itself, nor a user interface. These are all provided by the operating system and userspace tools running in the dom0 guest.

The most obvious task performed by the dom0 guest is to handle devices. This guest runs at a higher level of privilege than others, and so can access the hardware. Part of the responsibility for handling devices is the multiplexing of them for virtual machines and to provide each guest with its own virtual device.

The dom0 guest is also responsible for handling administrative tasks. While Xen itself creates new domU guests, it does so in response to a hypercall from the dom0 guest. This is typically done via a set of Python tools (scripts) that handles all of the policy related to guest creation in userspace and issue the relevant hypercalls.

Domain 0 provides the user interface to the hypervisor. The two dmons xend and xenstored running in this domain provide important features for the system. The first is responsible for providing an administrative interface to the hypervisor, allowing a user to define policy. The second provides the back-end storage for the XenStore.

Unprivileged domains

An unprivileged domain (domU) guest is more restricted. A domU guest is typically not allowed to perform any hypercalls that directly access

hardware, although in some situations it may be granted access to one or more devices.

Instead of directly accessing hardware, a domU guest typically implements the front end of some split device drivers. At a minimum, it is likely to need the XenStore and console device drivers. Most also implement the block and network interfaces. Because these are generic, abstract, devices, a domU only needs to implement one driver for each device category.

2.4 Virtualization Issues

In this section we summarily refer to the issues that arrise using the system components (mentioned earlier) in a Virualization environment and what is needed to overcome them. Who will now multiplex the the various Computing Systems (many OS + many applications per OS) that coexist in a Virtualized Environment? This role is assigned to the Hypervisor or Virual Machine Monitor (VMM) and must provide isolation and security among the VMs.

2.4.1 Virtualizing CPU

CPU as mentioned features a protection rings that the native operating systems take advantage of and provide security between kernel and user space and isolation between processes. This is not the case and will not suffice in a Virtualized Environment.

Emulation - QEMU

Emulation is one possible way to achieve a Virtualized Environment. But how can it be achieved? Lets assume that we start an unmodified kernel as a user process. That means that it will reside in outer privilege level meaning that when it tries to run a privileged instruction it will cause a General Protection Fault. Having a user level process (qemu) trapping this exception and emulating the instruction. This of course has overhead which leads to poor performance.

Hardware Assisted Virtualization - KVM

With the advent of CPU vitrualization extentions (Intel VT-x and AMD-V) CPU emulation is bypassed. But how is this achieved? Virtualization extentions provide a set of instructions that help store and restore the whole CPU state (when switching from one VM to another) plus implement an extra ring level (-1) that is called root mode. To this end the guest operating system may still reside in the ring level it is designed to run in. Transitions between root and non-root mode is done via special instructions (VMLAUNCH, VMRESUME) or when VM violates its privileges which is trapped by the VMM residing in ring level -1.

The community focused on how the linux kernel could play the role of VMM. The outcome of this effort is the kvm module that takes advantage those extensions and allows unmodified OS reside inside a native linux system. Management of VMs is done via ioctl call on a character device the module exposes to the host kernel.

Paravirtualization - Xen

The main idea is that the VMM is a software layer between hardware and OS. It provides multiplexing and isolation among different VMs. The VMM resides in most inner privilege ring (0) while VM kernels are deprivileged to ring 1. What differs it from the cases above is that the kernels are modified in order to reside in privilege ring 1. The hypervisor exposes an interface (hypercalls) to the guests that is equivalent to the system calls a native kernel exposes to applications.

2.4.2 Virtualizing Memory

In protected mode, the architectures provide a protection mechanism that operates at both the segment level and the page level. This protection mechanism provides the ability to limit access to certain segments or pages based on privilege levels (four privilege levels for segments and two privilege levels for pages). For example, critical operating-system code and data can be protected by placing them in more privileged segments than those that contain applications code. The processors protection mechanism will then prevent application code from accessing the operating-system code and data in any but a controlled, defined manner.

The processor uses privilege levels to prevent a program or task operating at a lesser privilege level from accessing a segment with a greater priv-

ilege, except under controlled situations. When the processor detects a privilege level violation, it generates a general-protection exception (GP).

Segments and Protection Every memory access is done via an segment selector and an offset (logical address). The selector points to a segment descriptor located in the GDT or LDT. From the segment descriptor, the processor obtains the base address of the segment in the linear address space. The offset then provides the location of the byte relative to the base address. This mechanism can be used to access any valid code, data, or stack segment, provided the segment is accessible from the current privilege level (CPL) at which the processor is operating. The CPL is defined as the protection level of the currently executing code segment.

In order the paging mechanism to provide isolation between user and supervisor code and data, four segments need to be defined: code and data segments at privilege level 3 for the user, and code and data segments at privilege level 0 for the supervisor. Usually these segments all overlay each other and start at address 0x0 in the linear address space. This flat segmentation model along with a simple paging structure can protect the operating system from applications, and by adding a separate paging structure for each task or process, it can also protect applications from each other.

Code segments can be either conforming or nonconforming. A transfer of execution into a more-privileged conforming segment allows execution to continue at the current privilege level. A transfer into a nonconforming segment at a different privilege level results in a general-protection exception (GP), unless a call gate or task gate is used.

Execution cannot be transferred by a call or a jump to a less privileged code segment, regardless of whether the target segment is a conforming or nonconforming code segment. Attempting such an execution transfer will result in a general-protection exception.

The architecture also defines two system segments the Task State Segment and LDT. The TSS defines the state of the execution environment for a task. It includes the state of general-purpose registers, segment registers, the EFLAGS register, the EIP register, and segment selectors with stack pointers for three stack segments (one stack for each privilege level). The TSS also includes the segment selector for the LDT associated with the task and the base address of the paging structure hierarchy.

All program execution in protected mode happens within the context of a task (called the current task). The segment selector for the TSS for

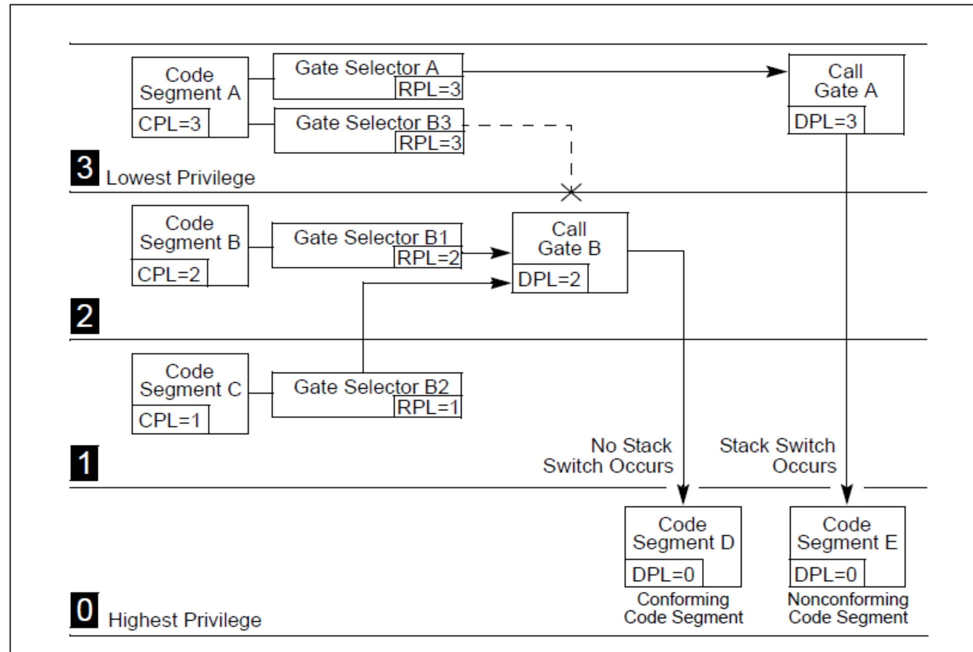


Figure 2.4: Call Gates

the current task is stored in the task register. In switching tasks, the processor performs the following actions:

1. Stores the state of the current task in the current TSS.
2. Loads the task register with the segment selector for the new task.
3. Accesses the new TSS through a segment descriptor in the GDT.
4. Loads the state of the new task from the new TSS into the general-purpose registers, the segment registers, the LDTR, control register CR3 (base address of the paging-structure hierarchy), the EFLAGS register, and the EIP register.
5. Begins execution of the new task.

Called gates (call gates, interrupt gates, trap gates, and task gates), mentioned earlier, provide protected gateways to system procedures and handlers that may operate at a different privilege level than application programs and most procedures.

For example, a CALL to a call gate can provide access to a procedure in a code segment that is at the same or a numerically lower privilege level

(more privileged) than the current code segment. To access a procedure through a call gate, the calling procedure supplies the selector for the call gate. The processor then performs an access rights check on the call gate, comparing the CPL with the privilege level of the call gate and the destination code segment pointed to by the call gate. If access to the destination code segment is allowed, the processor gets the segment selector for the destination code segment and an offset into that code segment from the call gate. If the call requires a change in privilege level, the processor also switches to the stack for the targeted privilege level. The segment selector for the new stack is obtained from the TSS for the currently running task.

External interrupts, software interrupts and exceptions are handled through the interrupt descriptor table (IDT). The IDT stores a collection of gate descriptors that provide access to interrupt and exception handlers. The linear address for the base of the IDT is contained in the IDT register (IDTR).

To carry out privilege-level checks between code segments and data segments, the processor recognizes the following three types of privilege levels:

- Current privilege level (CPL) – The CPL is the privilege level of the currently executing program or task. It is stored in bits 0 and 1 of the CS and SS segment registers. Normally, the CPL is equal to the privilege level of the code segment from which instructions are being fetched.

- Descriptor privilege level (DPL) –The DPL is the privilege level of a segment or gate. It gets compared with the CPL and the RPL of corresponding segment selector when the currently executing code segment attempts to access it; different types have different interpretations:

  - Data segment – if its DPL is 1, only programs running at a CPL of 0 or 1 can access it.
  - Nonconforming code segment (without using a call gate) – if its DPL is 0, only programs running at a CPL of 0 can access it
  - Call gate – same as data segment.
  - Conforming code segment and nonconforming code segment (accessed through a call gate) – if its DPL is 2, programs running at a CPL of 0 or 1 cannot access the segment.

  - TSS – same as data segment.

- Requested privilege level (RPL) – The RPL is an override privilege level that is assigned to segment selectors. It is stored in bits 0 and 1 of the segment selector. The processor checks the RPL along with the CPL to determine if access to a segment is allowed. The RPL can be used to insure that privileged code does not access a segment on behalf of an application program unless the program itself has access privileges for that segment.

Privilege levels are checked when the segment selector of a segment descriptor is loaded into a segment register. The checks used for data access differ from those used for transfers of program control among code segments.

The privileged instructions (such as the loading of system registers) are protected from use by application programs. They can be executed only when the CPL is 0 (most privileged).

IOPL (in EFLAGS register) indicates the I/O privilege level of the currently running program or task. The CPL of the currently running program or task must be less than or equal to the IOPL to access the I/O address space. This field can only be modified by the POPF and IRET instructions when operating at a CPL of 0.

Software Techniques

Software-based techniques maintain a shadow version of page table derived from guest page table (gPT). When the guest is active, the hypervisor forces the processor to use the shadow page table (sPT) to perform address translation. The sPT is not visible to the guest.

To maintain a valid sPT the hypervisor must keep track of the state of gPT. This include modifications by the guest to add or remove translation in the gPT, guest versus hypervisor induced page faults (defined below), accessed and dirty bits in sPT; and for SMP guests, consistency of address translation on processors.

Software can use various techniques to keep the sPT and gPT consistent. One of the techniques is write-protecting the gPT. In this technique the hypervisor write-protects all physical pages that constitute the gPT. Any modification by the guest to add a translation results in a page fault exception. On a page fault exception, the processor control is transferred

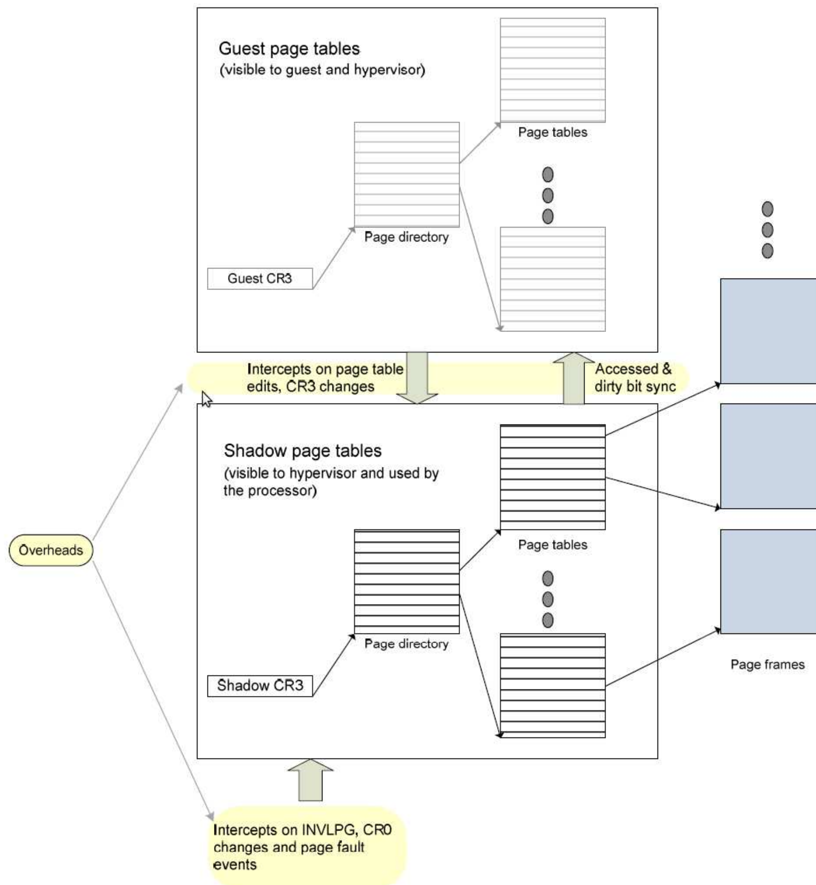


Figure 2.5: Guest/Shadow Page Tables

to the hypervisor so it can emulate the operation appropriately. Similarly, the hypervisor gets control when the guest edits gPT to remove a translation; the hypervisor removes the translation from the gPT and updates the sPT accordingly.

A different shadow paging technique does not write-protect gPT but instead depends on processors page-fault behavior and on guest adhering to TLB consistency rules. In this technique, sometimes referred to as Virtual TLB, the hypervisor lets the guest add new translations to gPT without intercepting those operations. Then later when the guest accesses an instruction or data which results in the processor referencing memory using that translation, the processor page faults because that translation is not present in sPT just yet. The page fault allows the hypervisor to intervene; it inspects the gPT to add the missing translation in the sPT and executes the faulting instruction. Similarly when the guest removes a translation, it executes INVLPG to invalidate that translation in the TLB. The hypervisor intercepts this operation; it then removes the corresponding translation in sPT and executes INVLPG for the removed translation.

Both techniques result in large number of page fault exceptions. Many page faults are caused due to normal guest behavior; such those as a result of accessing pages that have been paged out to the storage hierarchy by the guest operating system. We call such faults guestinduced page faults and they must be intercepted by the hypervisor, analyzed, and then reflected into the guest, which is a significant overhead when compared to native paging. Page faults due to shadow paging are called hypervisor-induced page faults. To distinguish between these two faults, the hypervisor traverses the guest and shadow page tables, which incurs significant software overheads.

When a guest is active, the page walker sets the accessed and dirty bits in the sPT. But because the guest may depend on proper setting of these bits in gPT, the hypervisor must reflect them back in the gPT. For example, the guest may use these bits to determine which pages can be moved to the hard disk to make room for new pages.

When the guest attempts to schedule a new process on the processor, it updates processors CR3 register to establish the gPT corresponding to the new process. The hypervisor must intercept this operation, invalidate TLB entries associated with the previous CR3 value and set the real CR3 value based on the corresponding sPT for the new process. Frequent context switches within the guest could result in significant hypervisor overheads.

Shadow paging can incur significant additional memory and performance overheads with SMP guests. In an SMP guest, the same gPT instance can be used for address translation on more than one processor. In such a case the hypervisor must either maintain sPT instances that can be used at each processor or share the sPT between multiple virtual processors. The former results in high memory overheads; the latter could result in high synchronization overheads. It is estimated that for certain workloads shadow paging can account for up to 75% of overall hypervisor overhead [1].

QEMU The qemu process runs mostly like a normal Linux program. It allocates its memory with normal malloc() or mmap() calls. However, just like a normal program doing a malloc(), there is no actual physical memory allocated at the time of the malloc(). It will not be actually allocated until the first time it is touched.

Once the guest is running, it sees that malloc()'d memory area as being its physical memory. If the guest's kernel were to access what it sees as physical address 0x0, it will see the first page of that malloc() done by the qemu process.

After that the qemu process does all the shadow paging manipulation discussed earlier.

Hardware Assisted Techniques

Both AMD and Intel sought solutions to these problems and came up with similar answers called EPT and NPT. They specify a set of structures recognized by the hardware which can quickly translate guest physical addresses to host physical addresses *without* going through the host page tables. This shortcut removes the costly two-dimensional page table walks.

The problem with this is that the host page tables are what we use to enforce things like process separation. If a page was to be unmapped from the host (when it is swapped, for instance), it then we *must* coordinate that change with these new hardware EPT/NPT structures.

AMD's Nested Paging For example AMD adds Nested Paging to the hardware page walker. Nested paging uses an additional or nested

[1] See http://developer.amd.com/assets/NPT-WP-1%201-final-TM.pdf for more information.

page table (NPT) to translate guest physical addresses to system physical addresses and leaves the guest in complete control over its page tables.Unlike shadow paging, once the nested pages are populated, the hypervisor does not need to intercept and emulate guests modification of gPT.

Nested paging removes the overheads associated with shadow paging. However because nested paging introduces an additional level of translation, the TLB miss cost could be larger.

Under nested paging both guest and the hypervisor have their own copy of the processor state affecting paging such as the CR0, CR3, CR4, EFER and PAT.

The gPT maps guest linear addresses to guest physical addresses. Nested page tables (nPT) map guest physical addresses to system physical addresses.

Guest and nested page tables are set up by the guest and hypervisor respectively. When a guest attempts to reference memory using a linear address and nested paging is enabled, the page walker performs a 2-dimensional walk using the gPT and nPT to translate the guest linear address to system physical address.

When the page walk is completed, a TLB entry containing the translation from guest linear address to system physical address is cached in the TLB and used on subsequent accesses to that linear address.

AMD processors supporting nested paging use the same TLB facilities to map from linear to system physical addresses, whether the processor is in guest or in host (or hypervisor) mode. When the processor is in guest mode, TLB maps guest linear addresses to system physical addresses. When processor is in host mode, the TLB maps host linear addresses to system physical addresses.

In addition, AMD processors supporting nested paging maintain a Nested TLB which caches guest physical to system physical translations to accelerate nested page table walks. Nested TLB exploits the high locality of guest page table structures and has a high hit rate.

Unlike shadow-paging, which requires the hypervisor to maintain an sPT instance for each gPT, a hypervisor using nested paging can set up a single instance of nPT to map the entire guest physical address space. Since guest memory is compact, the nPT should typically consume considerably less memory than an equivalent shadow-paging implementation.

With nested paging, the hypervisor can maintain a single instance of nPT which can be used simultaneously at one more processor in an SMP

guest. This is much more efficient than shadow paging implementations where the hypervisor either incurs a memory overhead to maintain per virtual processor sPT or incurs synchronization overheads resulting from use of shared sPT.

KVM KVM as mentioned is a qemu process that takes advantage of the Virtualization Extentions of the hardware with the corresponding module. To this end, kvm as hypervisor does all the necessary to create and maintain the NPT/EPT explained previously. Taking this feature into account another issue arrises.

Host page tables are what we use to enforce things like process separation. If a page was to be unmapped from the host (when it is swapped, for instance), it then we *must* coordinate that change with these new hardware EPT/NPT structures.

The solution in software is something Linux calls MMU notifiers. Since the qemu/kvm memory is normal Linux memory (from the host Linux kernel's perspective) the kernel may try to swap it, replace it, or even free it just like normal memory.

But, before the pages are actually given back to the host kernel for other use, the kvm/qemu guest is notified of the host's intentions. The kvm/qemu guest can then remove the page from the shadow page tables or the NPT/EPT structures. After the kvm/qemu guest has done this, the host kernel is then free to do what it wishes with the page.

To sum up a day in the life of a KVM physical page is: A day in the life of a KVM guest physical page:

Fault-in path:

1. QEMU calls malloc() and allocates virtual space for the page, but no backing physical page

2. The guest process touches what it thinks is a physical address, but this traps into the host since the memory is unallocated

3. The host kernel sees a page fault, and allocates some memory to back it.

4. The host kernel creates a page table entry to connect the virtual address to a host physical address, makes rmap entries, puts it on the LRU, etc...

5. MMU notifier is called, which allows KVM to create an NPT/EPT entry for the new page.

6. Host returns from page fault, guest execution resumes

Swap-out path: Now, let's say the host is under memory pressure. The page from above has gone through the Linux LRU and has found itself on the inactive list. The kernel decides that it wants the page back:

1. The host kernel uses rmap structures to find out in which VMA (vm_area_struct) the page is mapped.

2. The host kernel looks up the mm_struct associated with that VMA, and walks down the Linux page tables to find the host hardware page table entry (pte_t) for the page.

3. The host kernel swaps out the page and clears out the page table entry

4. Before freeing the page, the host kernel calls the MMU notifier to invalidate page. This looks up the page's entry in the NPT/EPT structures and removes it.

5. Now, any subsequent access to the page will trap into the host ((2) in the fault-in path above)

Paravirtualization

In Paravirtualization we have modified OS so it is possible the guest kernel to cooperate with the hypervisor who is still responsible for VM isolation. Virtualizing memory is highly architecture dependent. For example x86 has a hardware managed TLB; TLB misses are serviced automatically by the processor by walking the page table structure in hardware. Thus to achieve the best possible performance, all valid page translations for the currnet address space should be present in hardware-accessible page table. Another challenge is that the absence of tagged TLB which imposes complete TLB flush when switching address spaces.

Xen Xen as mentioned is a thin software level (VMM) between hardware and virtual machines. Taking x86 constraints into account, Xen 1. lets guest OSes be resposible for alocating and managing the hardware page tables, with minimal involvement from hypervisor to ensure

safety and isolation and 2. Xen itself exists in a 64MB section at top of every address space, thus avoiding a TLB flush when entering and leaving the hypervisor (with hypercalls).

Each time a guest OS requires a new page table, perhaps because a new process is being created, it allocates and initializes a page from its own memory reservation and registers it with Xen. At this point the OS must relinquish direct write privileges to the page-table memory: all subsequent updates must be validated by Xen. This restricts updates in a number of ways, including only allowing an OS to map pages that it owns, and disallowing writable mappings of page tables.

Segmentation is virtualized in a similar way, by validating updates to hardware segment descriptor tables. The only restrictions on x86 segment descriptors are: (i) they must have lower privilege than Xen, and (ii) they may not allow any access to the Xenreserved portion of the address space.

The approach in Xen is to register guest OS page tables directly with the MMU, and restrict guest OSes to read-only access. Page table updates are passed to Xen via a hypercall; to ensure safety, requests are validated before being applied.

To aid validation, we associate a type and reference count with each machine page frame. A frame may have any one of the following mutually-exclusive types at any point in time: page directory (PD), page table (PT), local descriptor table (LDT), global descriptor table (GDT), or writable (RW). Note that a guest OS may always create readable mappings to its own page frames, regardless of their current types. A frame may only safely be retasked when its reference count is zero. This mechanism is used to maintain the invariants required for safety; for example, a domain cannot have a writable mapping to any part of a page table as this would require the frame concerned to simultaneously be of types PT and RW.

The type system is also used to track which frames have already been validated for use in page tables. To this end, guest OSes indicate when a frame is allocated for page-table use – this requires a one-off validation of every entry in the frame by Xen, after which its type is pinned to PD or PT as appropriate, until a subsequent unpin request from the guest OS. This is particularly useful when changing the page table base pointer, as it obviates the need to validate the new page table on every context switch. Note that a frame cannot be retasked until it is both unpinned and its reference count has reduced to zero – this prevents guest OSes from using unpin requests to circumvent the reference-counting mechanism.

As far as physical memory is concerned, the initial memory allocation,

or reservation, for each domain is specified at the time of its creation; memory is thus statically partitioned between domains, providing strong isolation. A maximumallowable reservation may also be specified: if memory pressure within a domain increases, it may then attempt to claim additional memory pages from Xen, up to this reservation limit. Conversely, if a domain wishes to save resources, perhaps to avoid incurring unnecessary costs, it can reduce its memory reservation by releasing memory pages back to Xen.

2.4.3 Virtualizing Devices

To interact with the external I/O devices of a system usually specific software - kernel modules are needed. The so called device drivers operate in higher prividlege level than application and either on the same or lesser level compared to kernel. In native OS the every device is serviced by its own driver. In case of a Virtualized Environment where multiple OSs coexist this cannot be the case. The hypervisor must multiplex guest accesses to hardware. It cannot allow the guest OS to have direct access to hardware.

When virtualizing an I/O device, it is necessary for the underlying virtualization software to service several types of operations for that device. Interactions between software and physical devices include the following:

- *Device discovery: a mechanism for software to discover, query, and configure devices in the platform.*

- *Device control: a mechanism for software to communicate with the device and initiate I/O operations.*

- *Data transfers: a mechanism for the device to transfer data to and from system memory. Most devices support DMA in order to transfer data.*

- *I/O interrupts: a mechanism for hardware to be able to notify the software of events and state changes.*

In the following we refer to the various techniques used to virtualize devices.

Emulation - QEMU

I/O mechanisms on native (non-virtualized) platforms are usually performed on some type of hardware device. The software stack, commonly a driver in an OS, will interface with the hardware through some type of memory-mapped (MMIO) mechanism, whereby the processor issues instructions to read and write specific memory (or port) address ranges. The values read and written correspond to direct functions in hardware.

Emulation refers to the implementation of real hardware completely in software. Its greatest advantage is that it does not require any changes to existing guest software. The software runs as it did in the native case, interacting with the VMM emulator just as though it would with real hardware. The software is unaware that it is really talking to a virtualized device. In order for emulation to work, several mechanisms are required.

The VMM must expose a device in a manner that it can be discovered by the guest software. An example is to present a device in a PCI configuration space so that the guest software can "see" the device and discover the memory addresses that it can use to interact with the device.

The VMM must also have some method for capturing reads and writes to the device's address range, as well as capturing accesses to the device-discovery space. This enables the VMM to emulate the real hardware with which the guest software believes it is interfacing.

The device (usually called a device model) is implemented by the VMM completely in software (see Figure 2). It may be accessing a real piece of hardware in the platform in some manner to service some I/O, but that hardware is independent of the device model. For example, a guest might see an Integrated Drive Electronics (IDE) hard disk model exposed by the VMM, while the real platform actually contains a Serial ATA (SATA) drive.

The VMM must also have a mechanism for injecting interrupts into the guest at appropriate times on behalf of the emulated device. This is usually accomplished by emulating a Programmable Interrupt Controller (PIC). Once again, when the guest software exercises the PIC, these accesses must be trapped and the PIC device modeled appropriately by the VMM. While the PIC can be thought of as just another I/O device, it has to be there for any other interrupt-driven I/O devices to be emulated properly.

Emulation also facilitates the sharing of platform physical devices of the same type, because there are instances of an emulation model exposed

to potentially many guests. The VMM can use some type of sharing mechanism to allow all guest's emulation models access to the services of a single physical device. For example, the traffic from many guests with emulated network adapters could be bridged onto the platform's physical network adapter.

Since emulation presents to the guest software the exact interface of some existing physical hardware device, it can support a number of different guest OSs in an OS-independent manner. For example, if a particular storage device is emulated completely, then it will work with any software written for that device, independent of the guest OS, whether it be Windows*, Linux*, or some other IA-based OS. Since most modern OSs ship with drivers for many well-known devices, a particular device make and model can be selected for emulation such that it will be supported by these existing legacy environments.

Paravirtualization

Another technique for virtualizing I/O is to modify the software within the guest, an approach that is commonly referred to as paravirtualization. The advantage of I/O paravirtualization is better performance. A disadvantage is that it requires modification of the guest software, in particular device drivers, which limits its applicability to legacy OS and device-driver binaries.

With paravirtualization the altered guest software interacts directly with the VMM, usually at a higher abstraction level than the normal hardware/software interface. The VMM exposes an I/O type-specific API, for example, to send and receive network packetsin the case of a network adaptor. The altered software in the guest then uses this VMM API instead of interacting directly with a hardware device interface.

Paravirtualization reduces the number of interactions between the guest OS and VMM, resulting in better performance (higher throughput, lower latency, reduced CPU utilization), compared to device emulation.

Instead of using an emulated interrupt mechanism, paravirtualization uses an eventing or callback mechanism. This again has the potential to deliver better performance, because interactions with a PIC hardware interface are eliminated, and because most OS's handle interrupts in a staged manner, adding overhead and latency. First, interrupts are fielded by a small Interrupt Service Routine (ISR). An ISR usually acknowledges the interrupt and schedules a corresponding worker task. The worker task is then run in a different context to handle the bulk of the work associated

with the interrupt. With an event or callback being initiated directly in the guest software by the VMM, the work can be handled directly in the same context. With some implementations, when the VMM wishes to introduce an interrupt into the guest, it must force the running guest to exit to the VMM, where any pending interrupts can be picked up when the guest is reentered. To force a running guest to exit, a mechanism like IPI can be used. But this again adds overhead compared to a direct callback or event. Again, the largest detractor to this approach is that the interrupt handling mechanisms of the guest OS kernel must also be altered.

Since paravirtualization involves changing guest software, usually the changed components are specific to the guest environment. For instance, a paravirtualized storage driver for Windows XP* will not work in a Linux environment. Therefore, a separate paravirtualized component must be developed and supported for each targeted guest environment. These changes require apriori knowledge of which guest environments will be supported by a particular VMM.

Sharing of any platform physical devices of the same type is supported in the same manner as emulation. For example, guests using a paravirtualized storage driver to read and write data could be backed by stores on the same physical storage device managed by the VMM.

Paravirtualization is increasingly deployed to satisfy the performance requirements of I/O-intensive applications. Paravirtualization of I/O classes that are performance sensitive, such as networking, storage, and high-performance graphics, appears to be the method of choice in modern VMM architecture. As described, para-virtualization of I/O decreases the number of transitions between the client VM and the VMM, as well as eliminates most of the processing associated with device emulation.

Paravirtualization leads to a higher level of abstraction for I/O interfaces within the guest OS. I/O-buffer allocation and management policies that are aware of the fact that they are virtualized can be used for more efficient use of the VT-d protection and translation facilities than would be possible with an unmodified driver that relies on full device emulation.

Xen Xen first came up with the idea of split driver model (see later section for more details). A frontend driver resides in the guest and is all responsible to create the data to be transmitted, then propagates them to the backend driver who eventually passes them to the native driver.

In Xen the backend driver as long as the native driver (both kernel modules) reside in the privileged domain (dom0).

KVM Rusty Russel[3] introduced this kind of model in KVM under the name Virtio. In this case the backend driver is included in the qemu process (user space).

Direct Device Assignment

There are cases where it is desirable for a physical I/O device in the platform to be directly owned by a particular guest VM. Like emulation, direct assignment allows the owning guest VM to interface directly to a standard device hardware interface. Therefore, direct device assignment provides a native experience for the guest VM, because it can reuse existing drivers or other software to talk directly to the device.

Direct assignment improves performance over emulation because it allows the guest VM device driver to talk to the device in its native hardware command format eliminating the overhead of translating from the device command format of the virtual emulated device. More importantly, direct assignment increases VMM reliability and decreases VMM complexity since complex device drivers can be moved from the VMM to the guest.

Direct assignment, however, is not appropriate for all usages. First, a VMM can only allocate as many devices as are physically present in the platform. Second, direct assignment complicates VM migration in a number of ways. In order to migrate a VM between platforms, a similar device type, make, and model must be present and available on each platform. The VMM must also develop methods to extract any physical device state from the source platform, and to restore that state at the destination platform.

Moreover, in the absence of hardware support for direct assignment, direct assignment fails to reach its full potential in improving performance and enhancing reliability. First, platform interrupts may still need to be fielded by the VMM since it owns the rest of the physical platform. These interrupts must be routed to the appropriate guestin this case the one that owns the physical device. Therefore, there is still some overhead in this relaying of interrupts. Second, existing platforms do not provide a mechanism for a device to directly perform data transfers to and from the system memory that belongs to the guest VM in an efficient and secure manner. A guest VM is typically operating in a subset of the real physical address space. What the guest VM believes is its physical memory really is not; it is a subset of the system memory virtualized by the VMM for the guest. This addressing mismatch causes a problem for DMA-capable

devices. Such devices place data directly into system memory without involving the CPU. When the guest device driver instructs the device to perform a transfer it is using guest physical addresses, while the hardware is accessing system memory using host physical addresses.

In order to deal with the address space mismatch, VMMs that support direct assignment may employ a pass-through driver that intercepts all communication between the guest VM device driver and the hardware device. The pass-through driver performs the translation between the guest physical and real physical address spaces of all command arguments that refer to physical addresses. Pass-through drivers are device-specific since they must decode the command format for a specific device to perform the necessary translations. Such drivers perform a simpler task than traditional device drivers; therefore, performance is improved over emulation. However, VMM complexity remains high, thereby impacting VMM reliability. Still, the performance benefits have proven sufficient to employ this method in VMMs targeted to the server space, where it is acceptable to support direct assignment for only a relatively small number of common devices.

There are hardware devices that can be directly assigned to guests. One of these is the SR-IOV network adapter which has a number of virtual interfaces that can be explicitly exported to the guest. This reduces the overhead by much, but still cannot achieve bare metal performace due to host interception to all interrupts which can be bypassed with proposals like ELI[4].

2.5 Xen Internals in Paravirtualized Mode

As mentioned before, once the hypervisor has booted it starts the dom0 kernel. All other can then be started on demand. To keep track the different domain that coexist in the system Xen uses basicaly two main structs: `domain` and `vcpu` [2]. It maintains linked lists of these elements to be able to traverse every domain and its own vCPUs.

The `domain` structure contains some info about available memory, event channels, scheduling, privileges etc. The `vcpu` structure has info about its state, pending events, time-keeping, scheduling, affinity etc.

Two most important structs exported to each domain at start of day are: `start_info` and `shared_info` (the second included in the first).

[2] `domain` and `vcpu` structures defined in xen/include/xen/sched.h

The former contains vital info about vitual address allocated to the domain, machine page number of– and event channel for– the console and the Xen-store mechanism, etc. The latter ows info about pending and masked events, details needed for timekeeping, etc. Pages containing those structures are the first mapped into the domains address space and make it possible to continue booting. Linux references them as `xen_start_info` and `HYPERVISOR_shared_info`.

2.5.1 Time–keeping

In a virtualized environment a domain is not always running; can be preempted by another and forced to leave the processor. As a result it will miss timer interrupts and therefore is unable to perform valid and precise time-keeping. Before we continue it is wise to introduce the following definitions of time:

Wall time is the actual time of day, common for all VMs. It is useful for the various timers existing in OS.

Virtual time is the total period of time each VM is actually running on a processor. It is needed for OS internal process scheduling.

System time is the actual time that has passed since the VM booted.

Each VM uses its own `shared_info` to save data needed for time–keeping. In the listing below the most important points are mentioned. When a VM boots it saves the current wall time. Its system time is periodically calculated; at the time of the update the TSC value is temporarily put aside to provide better precision when requested.

To get the current wall time from the guest OS perspective we have to "consult" Xen; therefore `xen_read_wallclock()` is used. It gets the wall clock in boot time and the time the last update of the system time occured, extracts from the current TSC value the time since then and finally returns a timespec with secs and nsec passed since EPOCH. Details of how auxiliary functions c alculate those numbers are omitted.

```
per_cpu(xen_vcpu,cpu) =
                &HYPERVISOR_shared_info->vcpu_info[cpu]
xen_vcpu = &get_cpu_var(xen_vcpu);

/* the last updated system time in nsec */
xen_vpcu->time.system_time
```

```
/ the TSC when last update of system time occured */
xen_vcpu->time.tsc_timestamp

/* the wall time during boot */
&HYPERVISOR_shared_info->wc_sec|nsec

/* with the help of variables and structures above */

/* returns timespec since EPOCH */
xen_read_wallclock()

 /* adds wc and nsec since boot */
 pvclock_read_wallclock(wc, time)

  /* returns nsec since boot */
  pvclock_clocksource_read()

   /* returns the time of last system update
    * in nsec extracting the values from
    * vcpu_time_info struct (xen_vcpu->time) */
   pvclock_get_time_values()

   /* returns nsec passed since last update
    * calculating the tsc delta between current
        * value and one stored in tsc_timestamp */
   pvclock_get_nsec_offset()
```

The outcome is that we have a tool in hands that gives us in nsec granularity the current time which is common in every VM in the system.

2.5.2 Xenstore

The Xenstore is a mechanism to communicate vital information between the domains. Its main responsibility is for the driver domain to be able to export details about the backends of available devices. So the VMs could initilize their frontends and perform any I/O.

The implementation is done with a shared page, whose frame number is communicated within the `start_info` and then mapped to the domain's own virtual address space.

Xenstore structure is much alike a filesystem (i.e. a directory tree with different entries in the leaves). Every entry/value has a uniqueue key to reference to. All entries are accessible from all domains.

A common xenstore transaction is when setting up the networking on a VM boot. The VM after init it communicates the frontend details (ring-ref, eventchannel configuration options, etc.) in /local/domain/1/device/vif/ and notifies backend in the driver domain, who in its turn reads the store and afterwards exports the backend details (handle, configuration options, etc.) in /local/domain/0/backend/vif.

2.5.3 Event Channel Mechanism

In native oparating systems real hardware uses interrupts for asynchronous notification. In a paravirtulized environment all interrupts are handled by the hypervisor. In order to propagate them the event channel mechanism is introduced. It is also used for bidirectional interdomain notification.

Event channels mechanism is implemented with two bitmasks, one for events pending and one in events maksed and a flag if there are any events pending or not, all stored in `shared_info` structure of each domain. Once an event is delivered to a domain (e.g via `notify_remote_via_irq()` in case of interdomain notification) the coresponding bit in the pending bitmask is set, regardless if the domain currently sleeping. Every time a vCPU is scheduled in or its OS is switching from kernel to user mode or vice versa, checks for pending events.

If there are any then the corresponding handlers are invoked. The binding between handler and event for interdomain notification is done during init phase (e.g. of netfront dirver) via `bind_interdomain_evtchn_to_irqhandler()`.

2.5.4 I/O Rings

An I/O ring is a shared memory region, i.e. a circular buffer with fixed size entries that is used for data exchange between domains, usually between frontend and backend.

According to figure 2.6:

1. We have this circular buffer that both VMs can address it.
2. Additionally we have four pointers; request/response–producer/consumer.
3. We suppose that these pointers move clockwise on the ring.
4. All pointers are initilized a place 0.

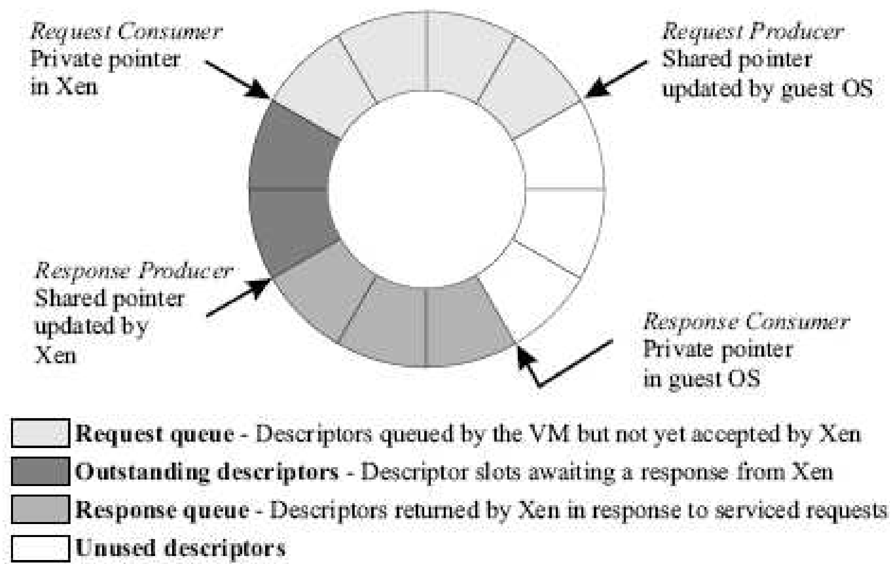


Figure 2.6: I/O rings: Xen's implementation for interdomain data exchange

5. Suppose 10 requests are produced. They are pushed in the ring and the request-producer pointer is increased by 10.

6. The other end removes gradually requests from the tail; request-consumer is increased. It replaces them with responces; responces-producer is increased.

7. The responces are removed from the tail and responces-consumer increased.

8. The ring is full if the request-producer and the responce-consumer overlap.

9. There are no more requests if the request-consumer and the request-producer overlap.

10. There are no more responces if the responce-consumer and the responce-producer overlap.

Detailed data types and useful macros for I/O ring manipulation defined in include/xen/interface/event_channel.h. One of which is `RING_PUSH_REQUESTS_AND_CHECK_NOTIFY` macro used by the split driver model; it pushes requests in the ring and

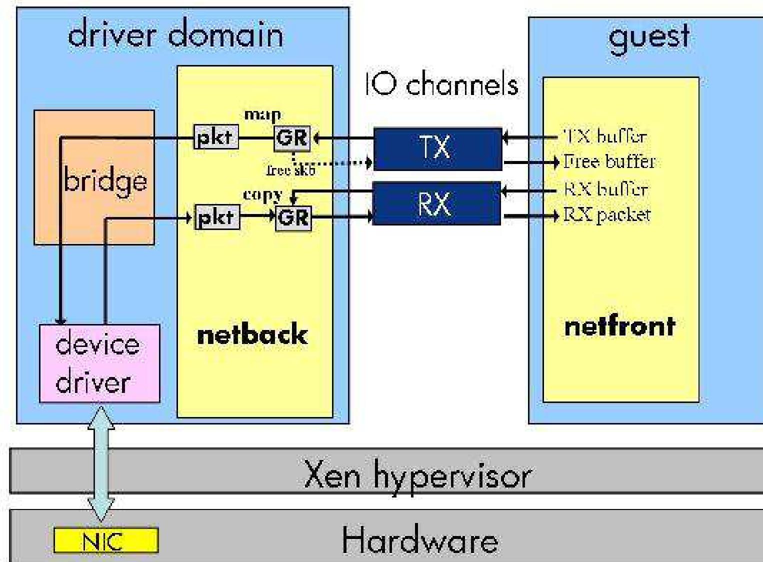


Figure 2.7: Xen's network split driver model

checks if the other end is currently processing any. If so it will continue until it consumes the newly inserted ones. If not then it must be notified; that is done via the `notify_remote_via_irq()` and the event channel mechanism as mentioned before.

2.5.5 The Split Driver Model

As long as in a paravirtualized environment a VM cannot access the hardware directly because its privileges are insufficient to execute I/O commands. Therefore the split driver model is introduced. Based on the fact that only the driver domain can access the hardware via its native drivers (who eventually invoke certain hypercalls) we have the following concept: the driver domain runs an extra driver, the backend who sits between the actual driver and the one running in the VM, the frontend. The latter exports a simple network API to the VM's kernel and talks to backend who eventually executes the I/O requested.

To implement it we make use of the I/O rings and the event channel mechanism mentioned earlier. Both ends are initialized using the Xenstore for communicating configuration options.

An actual split driver is referred in detail below: the netfront–netback drivers (see also Figure 2.7).

2.6 Network I/O Path

2.6.1 In Native OS

In this section we break down every component that participates in the network I/O path in a native OS. The basic networking stack [3] in Linux uses the four-layer model known as the Internet model. At the bottom of the stack is the link layer. The link layer (Ethernet, SLIP, PPP) refers to the device drivers providing access to the physical layer, which could be numerous mediums, such as serial links or Ethernet devices. Above the link layer is the network layer (IP, ICMP, ARP), which is responsible for directing packets to their destinations. The next layer, called the transport layer (TCP, UDP), is responsible for peer-to-peer communication (for example, within a host). While the network layer manages communication between hosts, the transport layer manages communication between endpoints within those hosts. Finally, there's the application layer (HTTP, FTP, SMTP), which is commonly the semantic layer that understands the data being moved.

Core network architecture

Figure 2.8 provides a high-level view of the Linux network stack. At the top is the user space layer, or application layer, which defines the users of the network stack. At the bottom are the physical devices that provide connectivity to the networks (serial or high-speed networks such as Ethernet). In the middle, or kernel space, is the networking subsystem that we focus in this section. Through the interior of the networking stack flow socket buffers (`sk_buff`) that move packet data between sources and sinks.

Overviewing shortly the core elements of the Linux networking subsystem (see Figure 2.8), *a*) at the top we find the system call interface. This simply provides a way for user-space applications to gain access to the kernel's networking subsystem. *b*) Next is a protocol-agnostic layer that provides a common way to work with the underlying transport-level protocols. *c*) Next are the actual protocols, which in Linux include the built-in protocols of TCP, UDP, and, of course, IP *d*) Next is another agnostic layer that permits a common interface to and from the individual device drivers that are available, *e*) followed at the end by the individual device drivers themselves.

[3] http://www.ibm.com/developerworks/linux/library/l-linux-networking-stack/

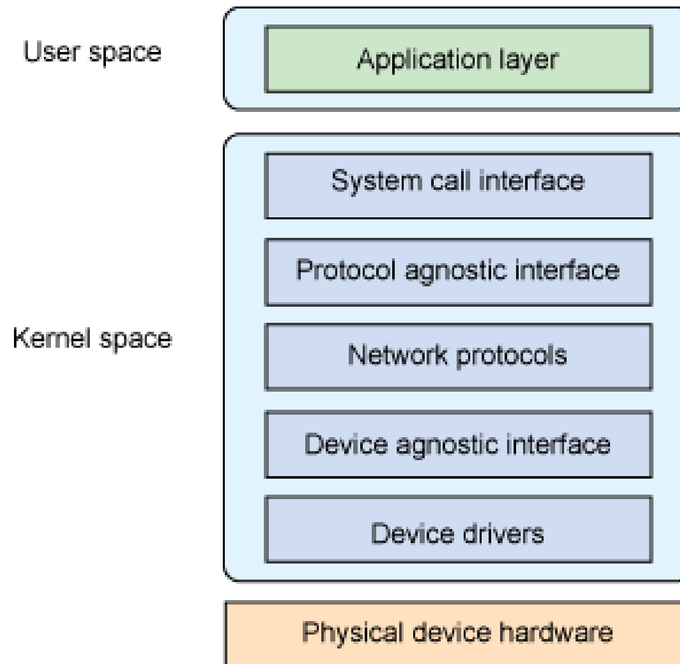


Figure 2.8: Linux high-level network stack architecture

System call interface

The system call interface can be described from two perspectives. When a networking call is made by the user, it is multiplexed through the system call interface into the kernel. This ends up invoking `sys_socket()`[4], which then further demultiplexes the call to its intended target. The other perspective of the system call interface is the use of normal file operations for networking I/O.

Protocol agnostic interface

The sockets layer is a protocol agnostic interface that provides a set of common functions to support a variety of different protocols.

Communication through the network stack takes place with a socket. The socket structure in Linux is `sock`[5].

The networking subsystem knows about the available protocols through a special structure that defines its capabilities. Each protocol maintains

[4] `sys_socket()` defined innet/socket.c
[5] `sock` structure is defined in include/net/sock.h

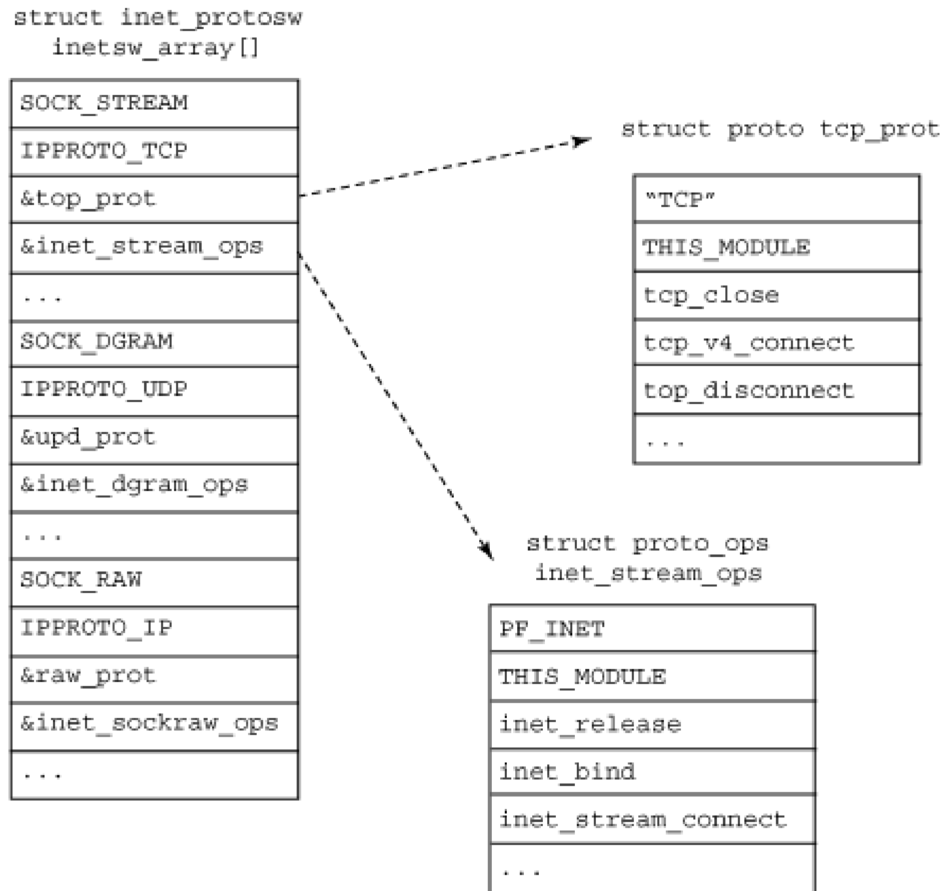


Figure 2.9: Internet protocol array structure

a structure called `proto`[6].

Network protocols

The network protocols section defines the particular networking protocols that are available (such as TCP, UDP, and so on). These are initialized at start of day in `inet_init()`[7] (as TCP and UDP are part of the inet family of protocols).

The `proto` structure defines the transport-specific methods, while the `proto_ops` structure defines the general socket methods.

Data movement for sockets takes place using a core structure called the socket buffer (`sk_buff`). An `sk_buff` contains packet data and also state

[6] `proto` structure is defined in include/net/sock.h
[7] `inet_init()` is defined in net/ipv4/af_inet.c

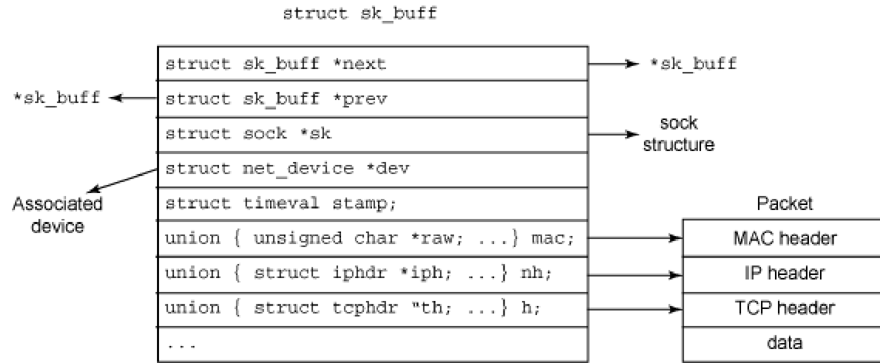


Figure 2.10: Socket buffer data structure

data that cover multiple layers of the protocol stack. Each packet sent or received is represented with an `sk_buff`[8](see Figure 2.10).

As shown, multiple `sk_buff` may be chained together for a given connection. Each `sk_buff` identifies the device structure (`net_device`) to which the packet is being sent or from which the packet was received. As each packet is represented with an `sk_buff`, the packet headers are conveniently located through a set of pointers (`th, iph, mac, header`). Because the `sk_buff` are central to the socket data management, a number of support functions have been created to manage them. Functions exist for `sk_buff` creation and destruction, cloning, and queue management.

Socket buffers are designed to be linked together for a given socket and include a multitude of information, including the links to the protocol headers, a timestamp (when the packet was sent or received), and the device associated with the packet.

Device agnostic interface

Below the protocols layer is another agnostic interface layer that connects protocols to a variety of hardware device drivers with varying capabilities. This layer provides a common set of functions to be used by lower-level network device drivers to allow them to operate with the higher-level protocol stack.

First, device drivers may register or unregister themselves to the kernel through `register_netdevice()` or `unregister_netdevice()`. The

[8]`sk_buff` structure is defined in include/linux/skbuff.h

caller first fills out the net_device structure and then passes it in for registration. The kernel calls its init function (if one is defined), performs a number of sanity checks, creates a sysfs entry, and then adds the new device to the device list (a linked list of devices active in the kernel)[9].

To send an sk_buff from the protocol layer to a device, dev_queue_xmit() is used. This function enqueues an sk_buff for eventual transmission by the underlying device driver[10]. The device structure contains a method, hard_start_xmit(), that holds the driver function for initiating transmission of an sk_buff.

Receiving a packet is performed conventionally with netif_rx(). When a lower-level device driver receives a packet (contained within an allocated sk_buff), the sk_buff is passed up to the network layer through netif_rx(). It queues then the sk_buff to an upper-layer protocol's queue for further processing through netif_rx_schedule() [11].

Recently, a new application program interface (NAPI) was introduced into the kernel to allow drivers to interface with the device agnostic layer (dev). NAPI can yield better performance under high loads by introducing a polling mechanism, and thus avoid taking an interrupt for each incoming frame.

Device drivers

At the bottom of the network stack are the device drivers that manage the physical network devices[12]. At initialization time, a device driver allocates a net_device structure and then initializes it with its necessary routines. One of these routines, dev->hard_start_xmit(), defines how the upper layer should enqueue an sk_buff for transmission. This routine takes an sk_buff. The operation of this function is dependent upon the underlying hardware, but commonly the packet described by the sk_buff is moved to a hardware ring or queue. Frame receipt, as described in the device agnostic layer, uses netif_rx() or netif_receive_skb() for a NAPI-compliant network driver. A NAPI driver puts constraints on the capabilities of the underlying hardware. After a device driver configures

[9] net_device structure is defined in include/linux/netdevice.h. The various functions are implemented in net/core/dev.c.

[10] The network device being defined by the net_device structure or sk_buff->dev reference in the sk_buff structure

[11] dev_queue_xmit() and netif_rx() are defined in net/core/dev.c

[12] Examples of devices at this layer include the SLIP driver over a serial interface or an Ethernet driver over an Ethernet device

Figure 2.11: Netfront-Netback interaction

its interfaces in the dev structure, a call to `register_netdevice()` makes it available for use[13].

2.6.2 In a ParaVirtualized Environment

In this section we break down every component that participates in the network I/O path in a Xen paravirtualized setup. Having already mentioned in detail in the previous section the native I/O path we take under the microscope how the split driver model sits between the the user application and TCP/IP stack discussed in native I/O path.

At init phase both drivers bind an irqhandler to the event channel previously allocated; `xennet_interrupt()` for the netfront and `netif_be_int()` for the netback. Some structs have also been initialized. net_device, a basic kernel data structure necessary for networking (encapsulated in the generic device structure), references netfront_info and backend_info. The latter includes a xen_netif.

In the rest of this section we will describe the netfront-netback interaction which is also depicted in Figure 2.11. Taking the tx packet flow and

[13] Drivers specific to network devices found in `drivers/net`

assuming that we have an sk_buff ready to transmit (after having followed the same procedure as in native OS????) the next things will occur:

1. `xennet_start_xmit()` is invoked. It takes care of fragmentation and eventually pushes requests in the tx ring and notifies netback if necessary.

2. `xennet_tx_buf_gc()` is invoked for pending responses in the tx ring.

3. when the driver domain checks for pernding events the irqhandler `netif_be_int()` is fired. It adds netif in net schedule list and invokes the bottom half handler which eventually schedules the `net_tx_action()` tasklet.

4. `net_tx_action()` perform two things: it invokes `net_tx_submit()` and the skb follows the flow discussed in 2.6.1; invokes `net_tx_action_dealloc()` to perform some memory cleanup (grand pages etc.) and in its turn call `make_tx_response()`.

5. the corresponding responses are pushed in the tx ring and netback notifies the netfront if necessary.

After that we reach a phase identical to the one that appears in the rx data flow. To this end, before we mention the netfront responce to it, we describe shortly rx procedure, and specifically after having acquired the `sk_buff` from the lower levels of native I/O path (hardware, native driver, demux/filter/iptables) and before the netfront gets notified:

1. `netif_be_start_xmit()` is invoked.

2. The `sk_buff` is enqueued in the `rx_queue` of netback.

3. netback's bottom half handler is invoked.

4. The tasklet `net_rx_action()` is scheduled.

5. It performs some memory arrangements and pushes the responses (i.e. the incomming data) in the rx ring via `make_rx_response()` and notifies the netfront if necessary

So on both cases we reach the phase of netfront getting notified via an event. Specifically:

1. when it checks for pending events, the irqhanlder `xennet_interrupt()` is fired.

2. it checks for responces in the tx ring with `xennet_tx_buf_gc()` [14] and cleans them up.

3. then is schedules `xennet_poll()` napi call which checks for incomming resposes in the rx ring.

2.7 Scheduling

In a preemptive multitasking system someone needs to decide which task is going to run and for how long. This resposibility lays on a specific system code: the scheduler. Based on an algorithm it picks a task from the pool of runnable ones, let it run on the processor for some time and then preempts it, chooses another and so forth. In this way the available task are being multiplexed in a way that gives the impression of running in parallel. In the case of a native operating system the scheduler multiplexes the various existing processes while in a virtualized environment it manages Virtual Machines. Of course for the latter a native system scheduler should run on top in order the processes residing in the VM to gain an actual timeslice.

Timeslice represents how long a task can run on the processor until it is preepmted. Too long a timeslice causes the system to have poor interactive performance - loss of “concurrency”. Too short a timeslice causes significant amounts of preocessor time to be wasted on the overhead of context-switch. I/O-bound process do not need longer timeslices (although they do like to run often), whereas processor-bound processes ccrave long timelices (to keep their caches hot).

Priority is a value assigned to each task. All tasks with equal priorities should be treated the same. A task with higher priority is favored among others and is prefered to run.

The nice value is a control over the timelice allocated to each task. A task with large nice value is considered to be nice to others; it gets smaller timeslice or gets lower priority.

The behavior of the scheduler that determines what runs when is called policy. In the case of a unified scheduler, the policy in a system must

[14] Eventually `netif_wake_queue()` is called, which allows upper layers to call the device `hard_start_xmit()` routine, used for flow control when transmit resources are available.

attempt to satisfy two conflicting goals: fast process response time (low latency) and maximal system utilization (high throughput).

In the following sections we will refer with detail to the current scheduler of linux kernel, the Completely Fair Scheduler, as well as to the stable scheduler the Xen hypervisor uses, the Credit Scheduler.

Priority & Timeslice - Pathological Problems

Schedulers have two common concepts: process priority and timeslice. Timeslice is how long a process runs; processes start with some default timeslice. Processes with a higher priority run more frequently and (on many systems) receive a higher timeslice. In Linux the priority is exported to user-space in the form of nice values. Here we mention some design concerns about those values:

- Mapping nice values onto timeslices requires a decision about what absolute timeslice to allot to each nice value. 0-20 nice maps to 100-5 ms.
- Relative nice values and their mapping to timeslices. Nicing down a process by one has wildly different effects depending on the starting nice value.
- Timeslice must be some interger multiple of the timer tick. It defines the floor of the timeslice as well as limits the difference between two timeslices.
- Boosting freshly woken-up processes can provide unfair amount of processor time, at the expense of the rest of the system.

Some easy to apply solutions to the problems above are making nice values geometric instead of additive and mapping nice values to timeslices using a measurement decoupled from the timer tick. But still assigning absolute timeslices yields a constant switching rate but variable fairness. CFS assigns each process a proportion of the processor. Thus yields constant fairness but a variable switching rate.

2.7.1 Linux's CFS

Currently Linux uses the Complete Fair Scheduler (CFS). Linux aiming to provide good interactive response and desktop performance, optimizes

for process responce, thus favoring I/O-bound processes over processor-bound ones without neglecting the latter.

CFS is based on a simple concept: Model process scheduling as if the system had an ideal, prefectly multitasking processor. In such a system each preocess would receive 1/n of the processor's time, where n is the number of runnable processes, and we'd schedule them for infinitiley small durations, so that in any measurable period we'd have run all n processes for the same amount of time.

Such a perfect multitasking is impossible. CFS will run each process for some amount of time, round-robin, selecting next the process that has run least. Rather than assign each process a timeslice, CFS calculates how long a process should run as a function of the total number of runnable processes. Instead of using the nice value to calculate a timeslice, CFS uses it to weight the proportion of the process is to receive.

CFS defines a targeted latency. Based on that each process is assigned a timeslice proportional to its weight divided by the total weight of all runnable processes so that none can experience a larger response time. To the other end CFS defines a lower boundary for a timeslice called minimum granularity.

Each task stores its virtual runtime. That is the total amount of time it has used the processor normalized by the number of runnable processes. Its units are ns; that is decoupled from the timer tick.

CFS picks the next process to run with the smallest virtual runtime. It uses red-black tree (self balancing binary tree) to sort processes depending on their virtual runtime. In such a structure the left most leaf (which is also cached) is the process with the least virtual runtime.

Linux supports three scheduling policies: normal, round-robin and fifo. The first one is where the most processes belong to and is hanlded by the CFS described earlier. The two other are real-time policies; fifo means that if a process belonging to this policy becomes runnable it gets picked up at once and continues running until it yields the processor. round-robin means that each runnable process in this group gets a fixed timeslice and are circularly multiplexed until they block. Of course any normal processes are getting stalled until non runnable real-time exists.

2.7.2 Xen's Credit Scheduler

Xen implements a modular scheduling concept; it executes a generic scheduler code which invokes scheduler specific functions that belong to

the exported interface of currently running scheduler. In the remaining of this section we try to break down its important parts.

From now on when we refer to Credit we imply any scheduler and its specific implementations. To start with we have to mention that there are two main per cpu structs; `scheduler` and `schedule_data`. The first contains the Credit interface and has a field `sched_data` pointing to Credit private data with active domains, number of cpus, etc. The second and very important has the currently running vpcu, the main scheduling timer, and a `sched_priv` field pointing to the Credit physical cpu data. It contains the runqueue for the corresponding cpu as well as the Credit ticker.

It is wise to stay a bit on these two timers. When s_timer goes off it raises a `SCHEDULE_SOFTIRQ` which makes the generic `do_schedule()` to be invoked; in its turn calls Credit `csched_schedule()` which acts based on its algorythm; finally it returns a `task_slice` containing the `vcpu` to run next and the amount of time the scheduler should allocate for it. As so the `s_timer` is set to go off when this amount of time expires. Currently this time is 30ms. On the other hand the `ticker` is a scheduler specific timer that is ticking every 10ms basically for accounting purposes.

Having covered the ticking part of the scheduling we can continue mentioning the Credit algorithm in short:

a) As discussed earlier every physical core has a local run-queue of vCPUs eligible to run.

b) The scheduler picks the head of the run-queue to execute for a time-slice of 30ms at maximum.

c) Every time accounting occurs (every 10ms) credits are debited to the running domain.

d) New allocation of credits occurs when all domains have their own consumed.

e) When a vCPU is woken up it gets inserted to a cpu's run-queue after all vCPUs with greater or equal priority.

f) vCPUs can be in one of 4 different priorities (ascending): IDLE, OVER, UNDER, BOOST. A vCPU is in the OVER state when it has all its credits consumed. BOOST is the state when one vCPU gets woken up.

g) When the local run-queue is empty or full with OVER / IDLE vCPUs, Credit migrates UNDER / BOOST vCPUs from neighbouring run-queues to empty one (load-balancing).

Credit's Shortcomings Credit is a general purpose scheduler that tries to perform adequately for the common case. It has mechanisms to boost temporarily I/O domains and a policy that lets a CPU bound domain occupy a physical core for an enough perriod of time before preempt it. But still it has a few shortcomings that become more apparent when different types of workloads and especially intensive ones co-exist. This is the main reason why we in this work we propose coexisting scheduling policies in a VM container that are tailored to the workloads' needs and service each domain ideally.

Some of the cases that the Credit falls short are the following:

a) In case a VM yields the processor before accounting occurs, no credits are debited [5]. This gives the running VM an advantage over others that consume credits and fall eventually to OVER state just by running for a bit longer.

b) BOOST vCPUs are favored unless they have their credits consumed. In case of a fast I/O, a vCPU consumes negligible credits (see (a)) and as a result CPU-bound vCPUs get eventually neglected.

c) Once a CPU bound VM gets scheduled in it does not get preempted until it consumes all of its time-slice. As a result I/O service, even if data is available to receive or transmitt, gets stalled.

This kind of shortcomings that are expected in a general purpose scheduler we are trying to attack with our proposed concept of co-existing scheduling policies that we are presenting in the following chapters.

2.7.3 CPU pools

The concept of CPU pools is simple: every pool consists of a set of physical cores and a scheduler running on top of them. During boot the primary pool is created running the default scheduler, acquiring all available physical cores and hosting the driver domain. Other pools can created and destroyed on demand as long as some rational requirements are satified:

a) To remove one core from an existing pool, it must be ensured that there are remaining ones that can service the domains running on this pool. If vCPU are pinned to this core then the operation will fail.

b) To add a new core to an existing pool, it must be first in the free list.

c) pool0 cannot be destroyed and dom0 cannot be moved.

Pools manipulation are done via hypercall `HYPERVISOR_sysctl()` with the `XEN_SYSCTL_cpupool_op` command. According the desired operation following subcommands are implemented:

XEN_SYSCTL_CPUPOOL_OP_CREATE
XEN_SYSCTL_CPUPOOL_OP_DESTROY
XEN_SYSCTL_CPUPOOL_OP_INFO
XEN_SYSCTL_CPUPOOL_OP_ADDCPU
XEN_SYSCTL_CPUPOOL_OP_RMCPU
XEN_SYSCTL_CPUPOOL_OP_MOVEDOMAIN
XEN_SYSCTL_CPUPOOL_OP_FREEINFO

CPU pools and scheduling are obviously interrelated; when a cpu pool operation is to be performed methods of the involved schedulers should be invoked. Being obvious that on creating/destroying a cpu pool the init/deinit methods of the corresponding schedulers should be invoked, below we demonstrate two basic features of cpu pools that a more complicated and worth mentioning; the operation of adding a core to an existing pool and the opoeration of moving a domain from a cpupool to another.

1. XEN_SYSCTL_CPUPOOL_OP_ADDCPU:
   (a) Firstly it checks if the given CPU is available (in free list)
   (b) Eventually `schedule_cpu_switch()` is invoked which deals with the scheduler specific calls of old and new scheduler interface: specifically, `alloc_pdata()` for the new core, `alloc_vdata()` for the idle vcpu, `tick_suspend()` for the old scheduler, change the per cpu variables cpupool, scheduler and schedule_data, `tick_resume()` for the new scheduler and `insert_vcpu()` for the idle vcpu; finally `free_vdata()` and `free_vdata()` for the old scheduler.
   (c) The cpu is added in the valid mask of the pool.
2. XEN_SYSCTL_CPUPOOL_OP_MOVEDOMAIN:

(a) dom0 cannot be move to another pool besides pool0.

(b) if the targeted pool does not own any physical core then abort.

(c) decrease the number of domains the old pool services.

(d) `sched_move_domain()` is invoked. New scheduler's `alloc_domdata()` for the moving domain, `alloc_vdata()` for each vcpu of the moving domain, `domain_pause()`, find the first valid cpu of the new pool, migrate the vcpu timers to this cpu and destroy_vcpu for the previous scheduler, update domain affinity, free_domdata and finally `domain_unpause()`

(e) increase the number of domains the new pool services

For further details and the actual implementation refer to the xen/common/cpupool.c, xen/include/xen/sched-if.h and xen/include/public/sysctl.h where the basic structures and functions are defined.

Chapter 3

Related Work

3.1 HPC Setups

Recent advances in virtualization technology have minimized overheads associated with CPU sharing when every vCPU is assigned to a physical core. As a result, CPU–bound applications achieve near-native performance when deployed in VM environments. However, I/O is a completely different story: intermediate virtualization layers impose significant overheads when multiple VMs share network or storage devices [6, 7].

3.1.1 Software Approaches

[8, 9, 10, 11]

3.1.2 Hardware Approaches

[12, 13, 14, 15]

3.2 Service–oriented Setups

Cherkasova [16] is evaluating and comparing the Credit Scheduler and the two other previously used; BVT and SEDF.

Ongaro et al. [5] examine the Xen's Credit Scheduler and expose its vulnerabilities from an I/O performance perspective. The authors evaluate two basic existing features of Credit and propose run-queue sorting

according to the credits each VM has consumed. Contrary to our approach, based on multiple, co-existing scheduling policies, the authors in [5] optimize an *existing*, *unified* scheduler to favor I/O VMs.

Finally, Hu et al. [17] propose a dynamic partitioning scheme using VM monitoring. Based on run–time I/O analysis, a VM is temporarily migrated to an isolated core set, optimized for I/O. The authors evaluate their framework using one I/O–intensive VM running concurrently with several CPU–intensive ones. Their findings suggest that more insight should be obtained on the implications of co-existing CPU– and I/O–intensive workloads. Based on this approach, we build an SMP-aware, static CPU partitioning framework taking advantage of contemporary hardware.

As opposed to [17], we choose to bypass the run-time profiling mechanism, which introduces overhead and its accuracy cannot be guaranteed. Specifically, we use a monitoring tool to examine the bottlenecks that arise when *multiple* I/O–intensive VMs co-exist with multiple CPU–intensive ones. We then deploy VMs to CPU-sets (pools) with their own scheduler algorithm, based on their workload characteristics. In order to put pressure on the I/O infrastructure, we perform our experiments in a modern multi-core platform, using multi-GigaBit network adapters. Additionally, we increase the degree of overcommitment to apply for a real-world scenario. Overall, we evaluate the benefits of coexisting scheduling policies in a busy VM container with VMs running various types of workloads. Our goal is to fully saturate existing hardware resources and get the most out of the system's performance.

Chapter 4

Design and Implementation

In order to examine the overall I/O behavior and to optimize the scheduling effect on the resources utilization few things needed to be developed from scratch and some other to be modified. In following sections we present in detail our designs and implementations that allowed us to evaluate (see Chapter 5) the concept of co-existing scheduling policies and point out that such a idea can benefit I/O–intensive VMs without having much of negative effect on the performance of others that run CPU-intensive workloads.

4.1 Monitoring Tool

The paravirtualized I/O path is discussed in section 2.6.2. We build on top of event channel mechanism and build a monitoring tool aiming to measure the time lost between domU notification and dom0 handling and vice versa. To implement it we choose to place time–stamps just before the event notification occurs and just after the event handler has fired. To this end, we add a function `add_cookie()`; it gets the current wall-clock time (see Section 2.5.1), the name of the interface, the function running and appends it to the existing array of cookies. As a result finally we should have two arrays, one extracted from the *netfront* and one from the *netback* that should have 1-to-1 relationship; when the one "sends" the other "receives". But that was not the case. On heavy I/O our monitor seemed to miss events. The actual fact was that we were counting also the times when a vCPU received an event on a channel that was already pending. To change that we had to slightly modify the `notify_remote_via_irq()` to make it known via its return value. Digging the hypercalls invoked, we noticed that

`HYPERVISOR_event_channel_op` for the `EVTCHNOP_send` case should propagate the return value of `evtchn_set_pending()`. So at the end the "send" event will be monitored with the following code:

```
RING_PUSH_REQUESTS_AND_CHECK_NOTIFY(&np->rx, notify);
if (notify) {
 xen_read_wallclock(&ts);
 r = notify_remote_via_irq(np->netdev->irq);
 if ( r == 0 ) {
  if (np->monitor_device && is_trigger_set())
   add_cookie(dev->name, ts, FN_NAME);
 }
}
```

The whole cookies table is displayed via the *proc_fs*. For this tool to apply in a more dynamic way we added two options;a trigger module and some module parameters to the *netfront* driver. In the boot command line of the domU we provide parameters for whether the domain will support this tool (`monitor_domain`) and if so a list of its network interfaces (`eth[]`) that will be monitor-able. At the init stage of the driver, for every newly created `netfront_info` instance, is defined whether it can be monitored or not according to the booting parameters. The matching between actual interfaces and the parameter list is done via the device's handle extracted from the Xenstore.

```
xenbus_scanf(XBT_NIL, dev->nodename, "handle", "%li", &handle);

netfront_info->monitor_device = monitor_domain & eth[handle];
```

The *netfront* then in `talk_to_netback()` propagates the monitoring feature for the specific device via Xenstore. After the init phase, the `frontend_changed()` callback is fired and *netback* in its turn in `connect_rings()` gets aware which `xen_netif` instance to monitor.

```
xenbus_scanf(XBT_NIL, dev->otherend, "monitor-device",
                 "%d", &netif->monitor_device) ;
```

To that we add the trigger module; echoing 1 to a file in `/proc` initialized the cookies array depending the size stated by a parameter in `cookies` module previously described. After that monitoring is on (see `is_trigger_set()` in first code block).

So we end up having a really flexible and capable tool. But what does it actually measures? What conclusion can be extracted from the cookies manipulation? Supposing an I/O intensive domU whose network interface is under monitoring for enough time, a useful aggregate value can

be calculated; Avg msec lost per MB transmitted. And that is because it is the only common among different setups. The split-driver model does not act identical for all the cases. It can batch requests before notifying the other-end. For instance, on a busy VM container (in terms of over-commitment) we have less events for the same amount of data transmitted. It is measured that in case of only one I/O VM 1000000 cookies are filled in 5 secs while when other coexist it might take more than a minute.

To sum up we have a tool that monitors the latency between event delivery and handling. To this end we can calculate the average time lost per MB transmitted. This number is nothing but the time needed for a vCPU that gets woken up from an event and becomes runnable to the actual time that it gets running. And that applies both to events from/to dom0 and domU. With other words we can express the effect of scheduling to I/O path and performance.

4.2 The *no-op* Scheduler

The main reason for developing a scheduler that actually doesn't perform any scheduling algorythm for time–sharing, is the need for maximum computing power for the driver domain who manages I/O in a paravirtualization environment. The way to achieve near native performance is to dedicate to dom0 exclusive cores and never preempt it in favor other. Such a scheduler should apply to any domain if that is desirable (and of course resources available). Within this concept, in the rest of this section we present in detail our implementation of this SMP-aware *no-op* Scheduler.

As discussed in 2.7.2 Xen's scheduler design is modular; one can easily add a new scheduler. The additions to be made are the following:

Add our new scheduler in xen/common/schedule.c:

```
extern const struct scheduler sched_dimara_def;
extern const struct scheduler sched_io_def;
static const struct scheduler *schedulers[] = {
    &sched_sedf_def,
    &sched_credit_def,
    &sched_credit2_def,
    &sched_dimara_def,
    &sched_io_def,
    NULL
}
```

and define the new scheduler ID in xen/include/public/domctl.h:

```
#define XEN_SCHEDULER_DIMARA    7
#define XEN_SCHEDULER_IO        8
```

Obviously it is not enough. The new scheduler has to implement some methods. Our primitive *no-op* scheduler consists of the following ones:

```
const struct scheduler sched_dimara_def = {
    .name           = "dimara Scheduler",
    .opt_name       = "dimara",
    .sched_id       = XEN_SCHEDULER_DIMARA,
    .sched_data     = &_dimara_priv,
    .init_domain    = dimara_dom_init,
    .destroy_domain = dimara_dom_destroy,
    .insert_vcpu    = dimara_vcpu_insert,
    .destroy_vcpu   = dimara_vcpu_destroy,
    .sleep          = dimara_vcpu_sleep,
    .wake           = dimara_vcpu_wake,
    .adjust         = dimara_dom_cntl,
    .pick_cpu       = dimara_cpu_pick,
    .do_schedule    = dimara_schedule,
    .init           = dimara_init,
    .deinit         = dimara_deinit,
    .alloc_vdata    = dimara_alloc_vdata,
    .free_vdata     = dimara_free_vdata,
    .alloc_pdata    = dimara_alloc_pdata,
    .free_pdata     = dimara_free_pdata,
    .alloc_domdata  = dimara_alloc_domdata,
    .free_domdata   = dimara_free_domdata,
};
```

In order to make the implementation of previous methods possible the following fundamental structs have to be introduced:

```
struct dimara_pcpu {
    struct dimara_vcpu *run; /* the vCPU attached to it */
};

struct dimara_vcpu {
    struct dimara_dom *sdom;
    struct vcpu *vcpu; /* the vCPU bellonging to */
    int oncpu;  /* is it attached to a pCPU? */
    int sleep;  /* is it currently sleeping? */
};

struct dimara_private {
```

```
    spinlock_t lock;
    uint32_t ncpus; /* #CPUs the scheduler owns */
    cpumask_t cpus; /* which CPUs the scheduler owns */
    cpumask_t not_occupied; /* which CPUs are not occupied */
};
```

We present below how all these structures are related to the basic stuctures discussed in Section2.7.2.

```
/* generic structures represending vCPUs and Domains */
struct vcpu *vc;
struct domain *dom;

/* the structure containing scheduler specific methods */
struct scheduler *ops;

/* scheduler specific structures respresenting
 * vCPUS, Domains, physical cores and scheduler's data */
struct dimara_vcpu *svc;
struct dimara_dom *sdom;
struct dimara_pcpu *spc;
struct dimara_private *prv = DIMARA_PRIV(ops);


/* the cpu index of current execution context */
cpu = smp_processor_id();

/* the cpu index of which the vCPU was previouly running on */
processor = vc->processor

/* the vcpu currently running on this cpu */
struct vcpu *curr =
  per_cpu(schedule_data, vc->processor).curr;

/* macros for referencing scheduler specific structures*/
#define DIMARA_PRIV(_ops)    \
  ((struct dimara_private *)((_ops)->sched_data))

#define DIMARA_PCPU(_c)      \
  ((struct dimara_pcpu *)per_cpu(schedule_data, _c).sched_priv)

#define DIMARA_VCPU(_vcpu)   \
  ((struct dimara_vcpu *) (_vcpu)->sched_priv)
```

In the rest of this section we describe in short how individual methods

are implemented and present their important parts via code snippets.

At start of day schedulers init() method is invoked. It allocates and initilizes scheduler specific structures and most important defines all CPUs available:

```
memset(prv, 0, sizeof(*prv));
cpus_setall(prv->not_occupied);
ops->sched_data = prv;
spin_lock_init(&prv->lock);
```

Its basic concept is the following: It keeps track of the available physical CPUs in the system, occupied or not. This info is kept in a bitmask in the scheduler's private data. For every newly created vCPU, `alloc_vdata()` is invoked and the scheduler attaches it to the first in the non-occupied pCPU list and lets it run without scheduling it out. If all pCPU are occupied, then the vCPU gets inserted in a "waiting list". If a pCPU becomes free after `destroy_vpcu()`, the first in the list gets serviced [1]

`alloc_vdata()` method does the following:

```
svc->oncpu = -1;
spin_lock_irqsave(&prv->lock, flags);
if ( is_idle_vcpu(vc) ) goto out;
cpus_and(avail, prv->cpus, prv->not_occupied);
if ( cpus_empty(avail) ) {
   printk("no cpu available!!!\n");
   goto out;
}
cpu = vc->processor;
if ( cpu_isset(cpu, avail) ) {
        printk("putting it to cpu %d \n", cpu);
        spc = DIMARA_PCPU(cpu);
        spc->run = svc;
        cpu_clear(cpu, prv->not_occupied);
        svc->oncpu = cpu;
}
out:
spin_unlock_irqrestore(&prv->lock, flags);
svc->sdom = dd;
svc->vcpu = vc;
return svc;
```

while `free_vdata()` invoked by `destroy_vcpu()` in case the vCPU intented to be removed is currently running on a CPU, it should be added in the not occupied bitmask:

[1] not yet implemented.

```
if ( svc->oncpu != -1 ) {
        spc = DIMARA_PCPU(svc->oncpu);
        spc->run = NULL;
        spin_lock_irqsave(&prv->lock, flags);
        cpu_set(svc->oncpu, prv->not_occupied);
        spin_unlock_irqrestore(&prv->lock, flags);
}
xfree(svc);
```

The `pick_cpu()` invoked by `vcpu_migrate()` should give the processor the vCPU previously run if it is available, otherwise one from the non-occupied list[2].

```
return vc->processor;
```

`insert_vcpu()` is invoked by `schedule_cpu_switch()`. All it needs to do is make the vCPU runnable, because previously `alloc_vdata` has placed it in its right pCPU as mentioned before.

```
svc->sleep = 0;
```

`wake()` should prevent the awaken vCPU from sleeping and raise a schedule SOFTIRQ to the corresponding pCPU so that it will be scheduled in immediately:

```
svc->sleep = 0;
cpu_raise_softirq(cpu, SCHEDULE_SOFTIRQ);
```

`sleep()` should do the opposite. It puts the vCPU to sleep and raises a schedule SOFTIRQ to the CPU it was previously running in case something else should be scheduled in (e.g. an idle vcpu that services a xen tasklet):

```
svc->sleep = 1;
if ( curr == vc )
        cpu_raise_softirq(processor, SCHEDULE_SOFTIRQ);
```

Finally, the `do_schedule` is fast and simple; gets the processor id it is currently running on, which is the one received the timer SOFTIRQ discussed in section 2.7.2; it schedules in the corresponding vCPU (if any) unless a tasklet work needs to be scheduled, in which case the *idle* vCPU takes its place. The code implementing all this follows:

```
dimara_schedule(const struct scheduler *ops,
        s_time_t now, bool_t tasklet_work_scheduled)
{
    const int cpu = smp_processor_id();
```

[2] not yet implemented

```
    struct dimara_vcpu *snext;
    struct task_slice ret;
    struct dimara_pcpu *pcpu = DIMARA_PCPU(cpu);
    struct vcpu *curr =  per_cpu(schedule_data, cpu).curr;

    snext = pcpu->run;
    /* Tasklet work (which runs in idle VCPU context)
         * overrides all else.
*/
    if ( tasklet_work_scheduled ||
              (snext == NULL) || snext->sleep )
        snext = DIMARA_VCPU(idle_vcpu[cpu]);
    ret.migrated = 0;
    ret.task = snext->vcpu;
    ret.time = (is_idle_vcpu(snext->vcpu))  ?
                                 -1 : MILLISECS(30);
    return ret;
}
```

To sum up we build an SMP-aware *no-op* scheduler that will be useful for every domain that needs to be running continuously. That can apply to the driver domain, some other stub-domains or even to some domU that are excessively busy or run real-time applications.

4.3 Credit Optimizations for I/O service

The Credit scheduler is designed to provide completely fair time-sharing to VMs that exhaust their allocated time-slice, i.e. CPU–intensive VMs. As discussed n section 2.7.2 in detail, Credit has also a boosting mechanism that temporarily favors early woken vCPUs; it inserts them in front of others in its runqueues. That applies to I/O VMs that get woken up from upcoming events. Adding credits debit to this concept reveals its shortcomings; Currently the scheduler debits credits to the domain owning the currently running vCPU. This accounting occurs every 10ms. It means when VM yields the processor before 10ms pass, i.e. light I/O case, no credits are debited to it which means that it gets favored before one that should run bit longer. In the case of VMs with heavy I/O, vCPUs get really often woken up from events so they get boosted all the time, and as a result other vCPUs trailing in the runqueues get neglected. On the other hand if there is always CPU work to be done an I/O VM should wait the other to exhaust their 30ms time-slice before it can actually run, although it might be boosted. Additionally it prevents

migrations of a vCPU to another core if it is kind of “cache hot”. This has a lot of meaning for a memory-bound VM, yet it does not absolutely apply to an I/O VM. All this make Credit fall short for the extreme case such a VM container with busy VMs running different types of workloads. Though, without deserting the Credit’s concept, in the following sections we argue that there can be few minor alternatives found that will benefit the service of exclusive I/O VMs.

4.3.1 Time-slice allocation

The easiest adjustment concerns the time-slice allocation and the accounting period. We argue that reducing the maximum time a VM can continuously run before being preempted from 30ms down to 3ms, while accounting will occur every 1ms would result to better I/O performance. That is due to the fairness it will provide among all vCPUs because credits will be debited unconditionally to every domain; ftp servers with heavy I/O would be equally treated just like web-servers performing fast and random I/O.

The price to pay is that VMs will suffer context switch an migration way more often; that is not a problem when scheduling is targeted to exclusive I/O VMs, because themselves are used to block waiting for data or devices to become available and yield the processor before the time-slice expires and because they are not prone to cache effects as much as CPU/Memory–bound applications. This adjustment in order to remain useful, accounting and load-balancing must be guaranteed to be easy and fast for the common case which applies totally to a system with medium overcommitment as VM containers in this work discussed (¡10 vCPUs/pCPU).

4.3.2 Anticipatory concept

Currently Credit boost I/O VMs only temporarily; as soon as they get woken up they obtain the BOOST priority and get inserted in front of others in the runqueues so that will be eventually picked sooner to run. But just after it run its priority get degraded to UNDER. In this way we do not take advantage the high probability of I/O transaction in the near future from the same VM. We argue that in current algorithm an anticipatory concept can be added. What we want to achieve is an I/O VM to remain boosted for an amount of time in which we predict to participate in a I/O operation.

Such a concept to apply following things should be implemented: *a*) There has to be a multi–hierarchical priority set. Supposing a descending one: BOOST1, BOOST2, BOOST3, UNDER, OVER. *b*) When inserting a vCPU in a runqueue it is placed after all others with the same priority. The scheduler picks always the first in the queue to run next. *c*) If a VM gets woken up (via the `wake()`) the scheduler annotates it a BOOST1 priority if it was in UNDER or upgrades it by one if already boosted. *d*) If a VM is put to sleep (via the `sleep()`), i.e waiting for I/O the scheduler will degrade its priority just by one. *e*) Every time an accounting occurs, BOOST priorities get gradually degraded . Thus a vCPU reaches UNDER priority if it consumes all of its allocated time–slice.

To sum up, the aforementioned modifications will ensure temporary boosting needed for I/O VMs running for low latency applications, as well as a sustained boost needed for applications with random and fast I/O such as busy web-servers. Heavy I/O such as ftp servers will be put aside in favor of the previously mentioned.

4.4 CPU pools

Currently, CPU pools manipulation is done via a driver domain's user level interface. Hypercalls' API is implemented in python and exported by xend daemon to the dom0 user. At the moment of this writing, we have in mind our future plan of a more dynamic partitioning of CPUs implemented with a driver running in dom0, which monitors VMs' behavior and classifies them according to their workload characteristics. Finally depending on its findings it could create, destroy pools with their corresponding schedulers, move domains among them, change the distribution of physical resources to each pool, etc. This to apply the hypercalls needed for accessible in kernel mode. Only the following slight additions should be made this to work:

in include/xen/interface/xen.h:

```
#define __HYPERVISOR_sysctl                35
#define __HYPERVISOR_domctl                36
```

in arch/x86/include/asm/xen/hypercall.h

```
#include <xen/interface/sysctl.h>
#include <xen/interface/domctl.h>

static inline long
HYPERVISOR_sysctl(struct xen_sysctl *u_sysctl)
```

```
{
        return _hypercall1(int, sysctl, u_sysctl);
}

static inline long
HYPERVISOR_domctl(struct xen_domctl *u_domctl)
{
        return _hypercall1(int, domctl, u_domctl);
}
```

Summarily we have designed from scratch a sort of tracing tool implemented for the Xen's event channel mechanism that is intended to measure the time lost between event sending and handling that occurs between netback and netfront split drivers used for domU networking.

Moreover we have designed a new modular scheduler to add it to the existing ones of Xen. It is a SMP-aware *no-op* scheduler that binds every newly created vCPU to a pCPU, if any available, so that it can offer maximum computing power and near native parformance to the domain beign serviced.

We have recognize the shortcomings in I/O service of Xen's current default scheduler, Credit and proposed two optimization based on its own concept: reducing the timeslice allocated for each vCPU; an anticipatory concept that takes advantage the high propability an I/O intensive domain has to receive or transmit data in the near future.

Finally we have ported hypercalls for pool manipulation in kernel so that drivers are able to access them. This can be useful for future implementations that include real time VM profiling and dynamic destribution of resources to each pool.

Chapter 5

Towards Distinctive Scheduling Policies and Evaluation

Having already described all tools used for our final implementation in chapter 4 we are able to present the steps followed towards distinctive scheduling policies, its benefits, the issues that arise as well as explain based on the theoretical background discussed in chapter 2 the results experienced, while evaluating our prototype on the Xen virtualization platform.

Testbed

Our testbed consists of an 8-core Intel Xeon X5365 @ 3.00 GHz platform as the VM container, running Xen 4.1-unstable with linux-2.6.32.24 pvops kernel , connected back–to–back with a 4-core AMD Phenom @ 2.3 GHz via 4 Intel 82571EB GigaBit Ethernet controllers.

Testcases

In the following experiments we emulate network traffic and CPU/Memory-bound applications for I/O– and CPU–intensive VMs respectively using generic tools (dd, netcat and bzip2). We measure the execution time of every action and calculate the aggregate I/O and CPU throughput. To explore the platform's capabilities we run the same experiments on native Linux and evaluate the utilization of resources. Our results are normalized to the maximum throughput achieved in the native case.

We have to mention that our emulation falls short implementing different cases of I/O. We narrow it down to heavy I/O probably performed by a ftp or stream server. Such a workload does not suffer from the "time uncertainty" of a random I/O (i.e. of a web-server). Thus it is easier to profile and most important to recreate similar test-cases that could allow you interpret your results without doubting about different execution schemes.

Moreover, we do not focus on a per VM performance but on the system's overall performance and its resources utilization. In most our experiments we increase the number of VMs in the VM container in order to explore the effect of stressing the scheduler as far as the degree of overcommitment is concerned. It is more than obvious that adding busy VMs to the system will decrease each one performance. But that is indifferent to us. What we are trying to succeed is to sustain the performance, the container can provide, to accepted levels. Our results can apply to different VM containers: one can host 30 VMs with some requirements and another can host 50 VMs with other. The latter will have only more limited requirements.

Our basic I/O–intensive application is:

```
dd if=/dev/zero bs=$BS count=$COUNT | netcat -q 0 $IP $PORT
```

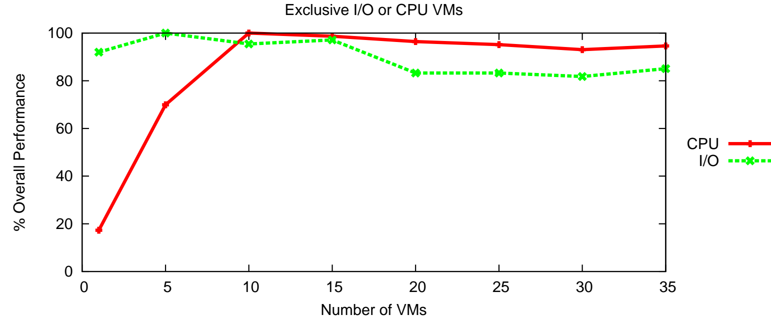
while our CPU/Memory bound is:

```
sudo dd if=/dev/urandom of=/dev/shm/file.img count=8 bs=1M
let i=$TIMES
while [ $i -ne 0 ]; do
    bzip2 -c /dev/shm/file.img > /dev/null
        let i--
done
```

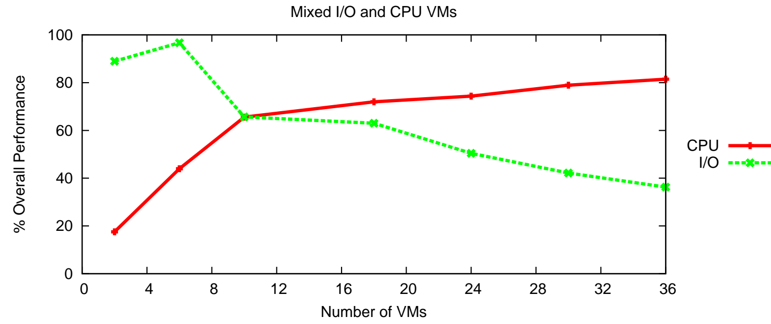
5.1 Exploring Vulnerabilities of Current Default Xen's Setup

In this section we show that, in a busy VM container, running mixed types of workloads leads to poor I/O performance and under-utilization of resources.

We measure the network I/O and CPU throughput, as a function of the number of VMs. In the default setup, we run the vanilla Xen VMM, using its default scheduler (Credit) and assign one vCPU to the driver



(a) CPU or I/O VMs (exclusive)



(b) CPU and I/O VMs (concurrently)

Figure 5.1: Overall Performance of the Xen Default Case

domain and to each of the VMs. We choose to keep the default CPU affinity (any). All VMs share a single GigaBit NIC (bridged setup).

To this end, we examine two separated cases:

Exclusive CPU– or I/O–intensive VMs. Figure 5.1(a) shows that the overall CPU operations per second are increasing until the number of vCPUs becomes equal to the number of physical CPUs. This is expected as the Credit scheduler provides fair time-sharing for CPU intensive VMs. Additionally, we observe that the link gets saturated but presents decreasing performance in the maximum degree of overcommitment. This is attributed to:

a) bridging all network interfaces together,

b) the fact that the driver domain is scheduled in and out repeatedly.

Concurrent CPU– and I/O–intensive VMs. Figure 5.1(b) points out that when CPU and I/O VMs run concurrently we experience a significant negative effect on the link utilization (less than 40%). Taking into

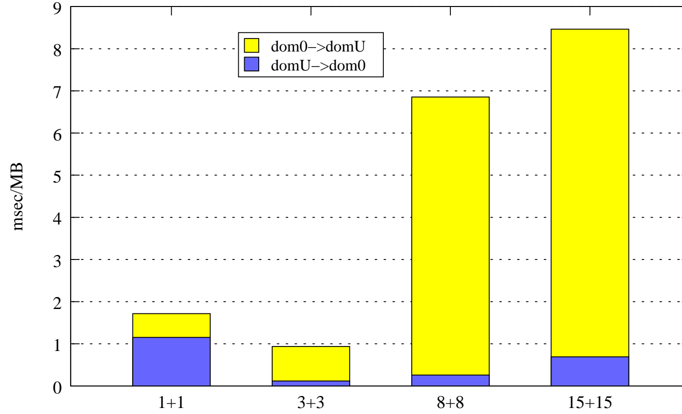


Figure 5.2: Monitoring Tool for the Default Setup: msecs lost per MB transmitted

account the theoretical background we expect this kind of behavior. The reasons justifying it are the following:

a) The fact that the driver domain is scheduled in and out repeatedly making it impossible to service I/O requests adequately.

b) This is deteriorated by the fact that CPU–intensive VMs, once they are scheduled in, exhaust their allocated time-slice and as a result I/O–intensive VMs including the driver domain are stalled waiting in the runqueues despite the fact that they may have been woken up or having pending I/O transactions.

c) The time-slice allocated is 30 msec. The vCPU can still block long before it expires and yield the CPU, something oftenly occurs in VMs performing I/O. This is not the case for CPU–intensive VM, which try to make the most of it. So the longer the time-slice the worst impact previously described on I/O intensive VMs.

To investigate this apparent suboptimal performance , we use the monitoring tool discussed in detail in section 4.1 . We run the same experiments as before, i.e. concurrent CPU– and I/O– intensive VMs, but monitor only the one of the latter.

Figure 5.2 plots the delay between domU event notification and dom0 event handling (dark area) and vice-versa (light area). We observe that the total time lost is increasing proportionally to the degree of overcommitment. As supposed earlier this is an artifact of vCPU scheduling: the awaken vCPU gets stalled until it receives a time-slice to process requests, leading to poor I/O performance.

We notice that the domU $\rightarrow$ dom0 direction is far less "time consuming" than the other way. The actual case is that we have a ratio 1/100 between the number of total events in both directions. That is happening for two reasons:

a) domU is transmitting to the client (emulating the ftp/stream server discussed earlier) so I/O requests towards this direction can be batched from the domU. On the other hand TCP ACK packets arriving from the client arrive in a more random way. dom0 still notifies domU even if only a few are available, as a result this must be repeated plenty of times.

b) dom0 is waking up way more often due to request arriving from other I/O guests or from incoming Ethernet packets. Thus it batches pending requests and process them more efficiently.

Although there is a significant difference between those two directions, trying to eliminate at first just one, it could still increase performance. This attempt leads us to the following section.

5.2 The Driver Domain Pool

To eliminate time lost in domU $\rightarrow$ dom0 data path discussed in the previous section, we decouple the driver domain from all VMs. We take advantage of the pool concept of Xen, we launch the *no-op* scheduler mentioned in section 4.2 on a separate pool running exclusively the driver domain. VMs are deployed on different pool and suffer the Credit scheduler policy. To explore the effect of this kind of setup we make use of our monitoring tool. The results confirm our hypothesis:

Figure 5.3 depicts the effect of decoupling the driver domain from the others. We should emphasize on the following things:

a) domU $\rightarrow$ dom0 event handling latency is eliminated (dark area). In maximum degree of overcommitment 0.691 msec are decreased to 0.011 msec.

b) dom0 $\rightarrow$ domU event handling latency is reduced by 45% (light area).

c) this behavior is expected because of the continuous awareness of the driver domain. It is always scheduled in, and can process domU requests and incoming packets way more efficiently and on time.

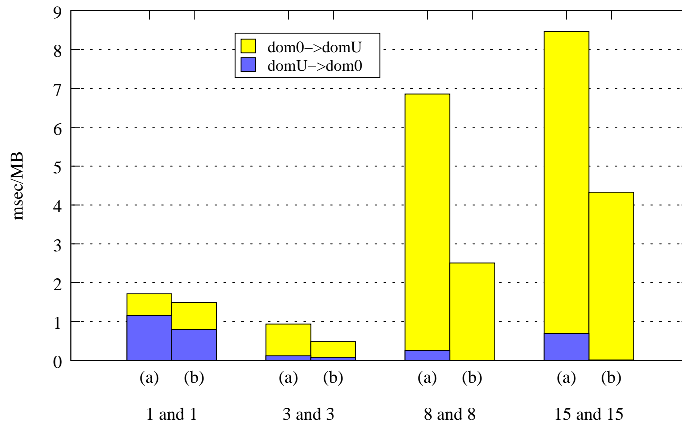
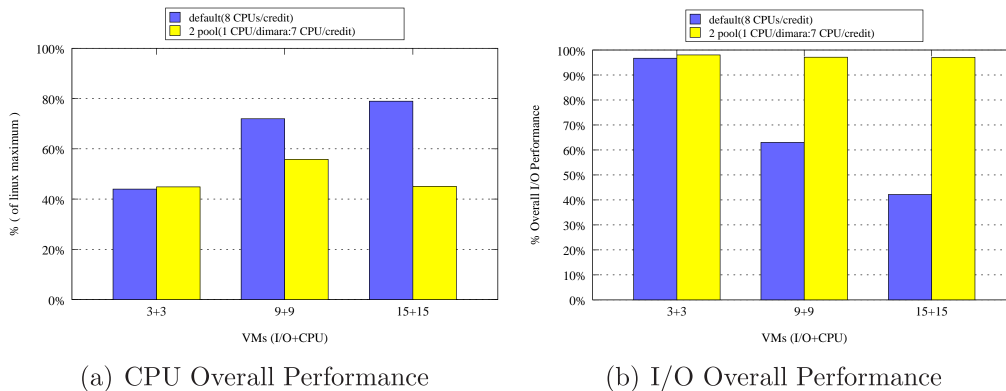


Figure 5.3: Monitoring Tool Results (msec lost per MB transmitted): (a) default Setup; (b) 2 pools Setup



(a) CPU Overall Performance

(b) I/O Overall Performance

Figure 5.4: Overall Performance: default Setup; 2 pools Setup

To explore the overall effect of adding the driver domain pool we run our familiar test cases with CPU and I/O VM running concurrently. Figure 5.4 allows us to draw following conclusions:

a) the apparent improvement of the I/O performance: we achieve Gbps saturation in contrast to less than 40% utilization of the Default case.

b) CPU utilization is deteriorated by 43%. This can be explained taking into account the Credit scheduler algorithm discussed in section 2.7.2 in detail. I/O VM get notified way more frequent because of the advanced and continuous performance of the driver domain. They get boosted, get inserted in front of CPU domain in the runqueues and eventually steal time-slices from them. Thus CPU domains get neglected and finally to leads suboptimal CPU utilization.

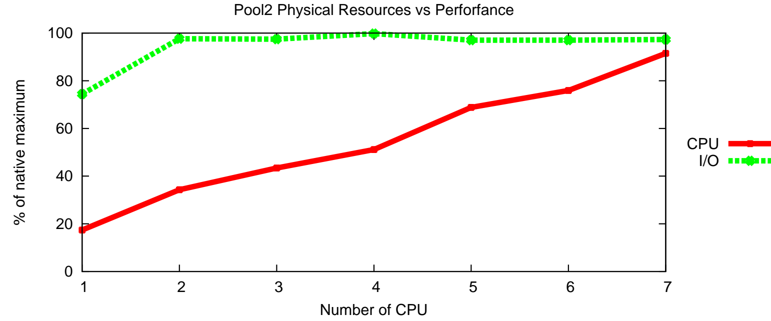


Figure 5.5: Overall Performance vs. Physical Resources Distribution to VM pool

5.3 Resources distribution vs. VM Performance

Adopting the previous concept of decoupling the driver domain from the VMs we dedicate a distinctive pool with one physical core we try to eliminate the negative effect to the CPU–intensive VMs. Therefore we experiment with physical resources distribution and specifically we evaluate the system's overall performance when allocating a different number of physical CPUs to the aforementioned second pool running exclusive 15 I/O– or CPU– intensive VMs. Figure 5.5 plots the resources utilization versus the number of physical cores allocated to the pool. Each time the remaining cores stay unused. Results are normalized to the maximum performance experienced. We observe that with one CPU, the GigaBit link is under-utilized probably due to the single runqueue existing, whereas with two CPUs link saturation is achieved. On the other hand, cutting down resources to the CPU-intensive VMs does not have a negligible effect; in fact it can shrink up to 20% when only one core is used.

5.4 Decoupling vCPUs based on workload characteristics

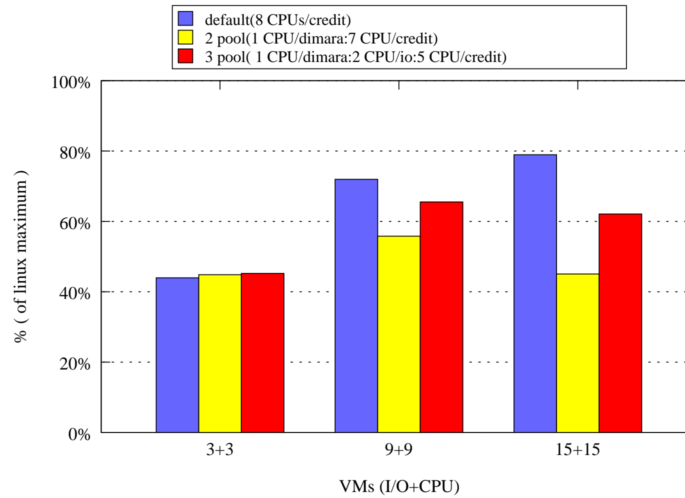
Taking all this into consideration we obtain a platform with 3 pools: *pool0* with only one CPU dedicated to the driver domain with the *no-op* scheduler; *pool1* with 2 CPUs servicing I/O intensive VMs (running potentially an I/O–optimized scheduler); and *pool2* for the CPU-intensive

VMs that suffer the existing Credit scheduling policy. Running concurrently a large number of VMs with two types of workloads we experience GigaBit saturation and 62% CPU utilization, as opposed to 38% and 78% respectively in the default case (Fig. 5.6, third bar).

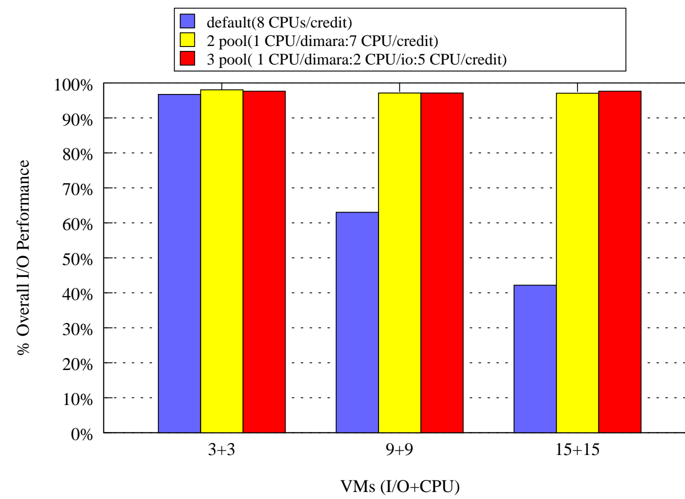
In addition to that, we point out that there is no overall benefit if a VM finds itself in the "wrong" pool, albeit a slight improvement of this VM's I/O performance is experienced (Table 5.4). This is an artifact of Credit's fairness discussed in previous sections and background.

	Misplaced VM	All other
CPU	-17%	-1.3%
I/O	+4%	-0.4%

Table 5.1: VM Misplacement effect to individual Performance



(a) CPU Overall Performance



(b) I/O Overall Performance

Figure 5.6: Overall Performance: default Setup; 2 pools Setup; 3 pools Setup

Chapter 6

Conclusions and Future Work

In this work we examine the impact of VMM scheduling in a service oriented VM container and argue that co-existing scheduling policies can benefit the overall resource utilization when numerous VMs run contradicting types of workloads. VMs are grouped into sets based on their workload characteristics, suffering scheduling policies tailored to the need of each group. We implement our approach in the Xen virtualization platform. In a moderate overcommitment scenario (4 vCPUs/ physical core), our framework is able to achieve link saturation compared to less than 40% link utilization, while CPU-intensive workloads sustain 80% of the default case. This negative effect on CPU performance can be eliminated by adding up cores (e.g. in a many-core platform, where less limitation of physical resources exists).

Our future agenda consists of exploring exotic scenarios using different types of devices shared across VMs (multi-queue and VM-enabled multi-Gbps NICs, hardware accelerators etc.), as well as implementing scheduler algorithms designed for specific cases. Specifically, we would like to experiment with schedulers featuring multi-hierarchical priority sets, as well as algorithms favoring low latency applications, random I/O, disk I/O etc.

Specifically we should answer to the following questions:

- Which kind of resources partitioning could be more efficient?
- What would be the results if we had 3 different kind of workloads: CPU,memory and I/O bound? (experiment with those kind of benchmarks)
- How could we characterize a VM? What will be the overhead? Who will decide?

- Whould it make sence to have a dynamic partitioning?
- How more efficient could be a tailored made scheduler?
- Are real world scenarios (overcommitment, idle VMs, different workloads per VM) applicable to our concept?
- What resources utilization we achieve with our proposal in case exotic hardware (10GbE) is used?
- Disk I/O or network/object file systems apply to our scenario?

Our findings suggest that community must dig and investigate further the scheduling effect in Virtual Machine Containers and try to come up with answers to previous questions in order to aim data center consolidation and better resourses utilization that could eventually result to great power savings, a more than desired outcome.

Bibliography

[1] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, pp. 412–421, July 1974.

[2] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, (Berkeley, CA, USA), pp. 41–41, USENIX Association, 2005.

[3] R. Russell, "virtio: towards a de-facto standard for virtual i/o devices," *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 95–103, July 2008.

[4] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, D. Tsafrir, and A. Schuster, "Eli: Bare-metal performance for i/o virtualization," in *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2012.

[5] D. Ongaro, A. L. Cox, and S. Rixner, "Scheduling i/o in virtual machine monitors," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, (New York, NY, USA), pp. 1–10, ACM, 2008.

[6] A. Nanos, G. Goumas, and N. Koziris, "Exploring I/O Virtualization Data paths for MPI Applications in a Cluster of VMs: A Networking Perspective," in *5th Workshop on Virtualization in High-Performance Cloud Computing (VHPC '10)*, (Naples-Ischia, Italy), 2010.

[7] L. Youseff, R. Wolski, B. Gorda, and C. Krintz, "Evaluating the Performance Impact of Xen on MPI and Process Execution For HPC Systems," in *1st Intern. Workshop on Virtualization Technology in Distributed Computing. VTDC 2006.*

[8] Y. Dong, J. Dai, Z. Huang, H. Guan, K. Tian, and Y. Jiang, "Towards high-quality I/O virtualization," in *SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, (New York, NY, USA), pp. 1–8, ACM, 2009.

[9] A. Landau, D. Hadas, and M. Ben-Yehuda, "Plugging the hypervisor abstraction leaks caused by virtual networking," in *SYSTOR '10: Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, (New York, NY, USA), pp. 1–9, ACM, 2010.

[10] A. Menon, A. L. Cox, and W. Zwaenepoel, "Optimizing network virtualization in Xen," in *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2006.

[11] K. K. Ram, J. R. Santos, and Y. Turner, "Redesigning xen's memory sharing mechanism for safe and efficient I/O virtualization," in *WIOV'10: Proceedings of the 2nd conference on I/O virtualization*, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 2010.

[12] F. Auernhammer and P. Sagmeister, "Architectural support for user-level network interfaces in heavily virtualized systems," in *WIOV'10: Proceedings of the 2nd conference on I/O virtualization*, (Berkeley, CA, USA), pp. 7–7, USENIX Association, 2010.

[13] Y. Dong, Z. Yu, and G. Rose, "SR-IOV networking in Xen: architecture, design and implementation," in *WIOV'08: Proceedings of the First conference on I/O virtualization*, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2008.

[14] K. Mansley, G. Law, D. Riddoch, G. Barzini, N. Turton, and S. Pope, "Getting 10 Gb/s from Xen: safe and fast device access from unprivileged domains," in *Euro-Par'07: Proceedings of the 2007 conference on Parallel processing*, (Berlin, Heidelberg), pp. 224–233, Springer-Verlag, 2008.

[15] H. Raj and K. Schwan, "High performance and scalable I/O virtualization via self-virtualized devices," in *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, (New York, NY, USA), pp. 179–188, ACM, 2007.

[16] L. Cherkasova, D. Gupta, and A. Vahdat, "Comparison of the three cpu schedulers in xen," *SIGMETRICS Perform. Eval. Rev.*, vol. 35, pp. 42–51, September 2007.

[17] Y. Hu, X. Long, J. Zhang, J. He, and L. Xia, "I/o scheduling model of virtual machine based on multi-core dynamic partitioning," in *IEEE International Symposium on High Performance Distributed Computing*, pp. 142–154, 2010.