University of Glasgow

School of Computing Science

60 YEARS OF COMPUTING AT GLASGOW

Virtualisation

Advanced Operating Systems

Lecture 17

Lecture Outline

- What is virtualisation?
- Full system virtualisation
  - Hypervisors – type 1 and type 2
  - Virtualising CPUs, memory, device drivers
  - Live migration
- Example: Xen

Virtualisation Concepts

- Enable resource sharing by decoupling execution environments from physical hardware
  - *Full system virtualisation* – run multiple operating systems on a single physical host; virtual machines (VMs)
  - Desirable for data centres and cloud hosting environments – computing as a service
  - Benefits for flexible management and security
- Use isolation to protect services
  - *Containers* – virtualise resources of interest
  - Lightweight, but imperfect, virtualisation; constrains ability of a group of processes to access system resources
  - Enhances security within an operating system

Full System Virtualisation

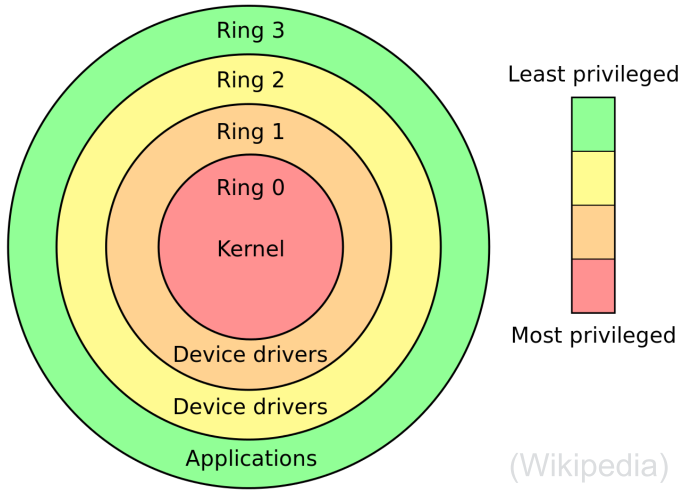
- Allow more than one operating system to run on the same physical hardware
  - First implemented: IBM System/360 mainframe – mid-1960s
  - Popular current implementations: Xen, VMWare, QEMU, VirtualBox

- Introduces *hypervisor* abstraction:
  - The hypervisor is a process that manages the virtualisation, and emulates the hardware
    - Processors
    - Memory
    - Device drivers
  - The guest operating systems run as processes on the hypervisor
  - The hypervisor API presents a virtual machine abstraction – each operating system thinks it's running on real hardware

(source: W

Hardware Virtualisation: CPU

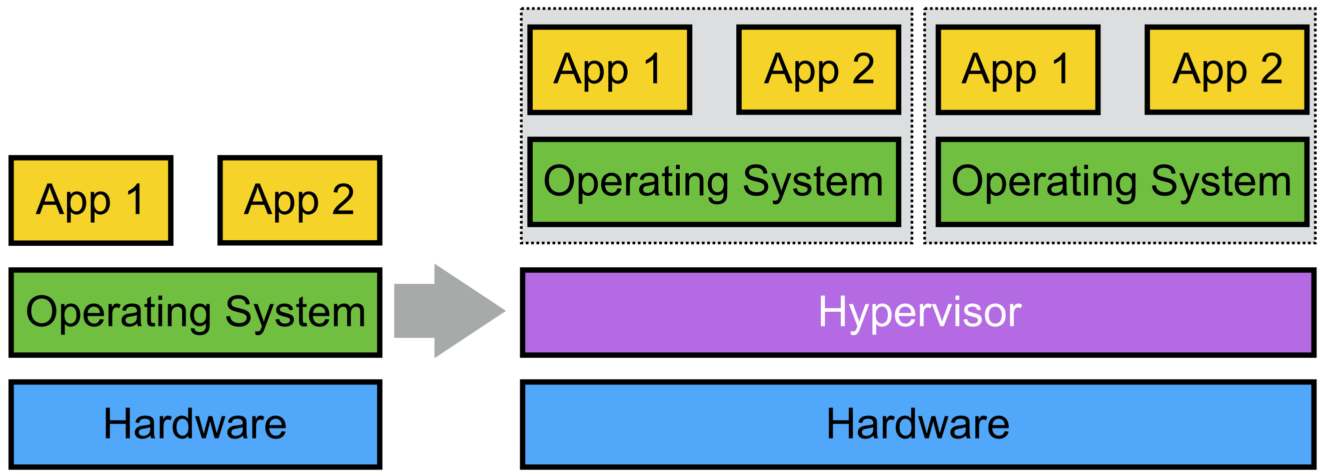
- Processors distinguish privileged and unprivileged operations
  - Known as *protection rings* or *privilege levels*
    - ARM7 CPUs have “application”, “operating system”, and “hypervisor” privileges
    - x86 CPUs have four levels of numbered protection rings
  - Instructions that control I/O, interrupt handlers, virtual memory, memory protection, etc., tend to be privileged
  - Attempts to execute privileged instructions from unprivileged code trap and invoke a handler at next higher privilege level
    - Handler can check permissions, and either allow the operation, terminate lower privilege process, or otherwise arbitrate access
- Full virtualisation requires either:
  - Hypervisor running at higher privilege than the guest operating systems – allowing it to arbitrate access between those guests, without support from the guests
  - Or, rewriting operating system code to be aware of virtualisation, and to call into the hypervisor to perform privileged operations – known as paravirtualisation, where the operating systems agree to cooperate with the hypervisor



(Wikipedia)

Hardware Virtualisation: Hypervisor Mode

- Unmodified guest operating systems running on a hypervisor
  - The hypervisor is aware of the guest operating systems it's running
  - The guest operating systems don't know that they're being virtualised



- Traps normally privileged operations to the hypervisor
  - Cache control
  - Page tables and virtual memory
  - Interrupt handlers
  - …
- Performance reduced – depends on degree of hardware support

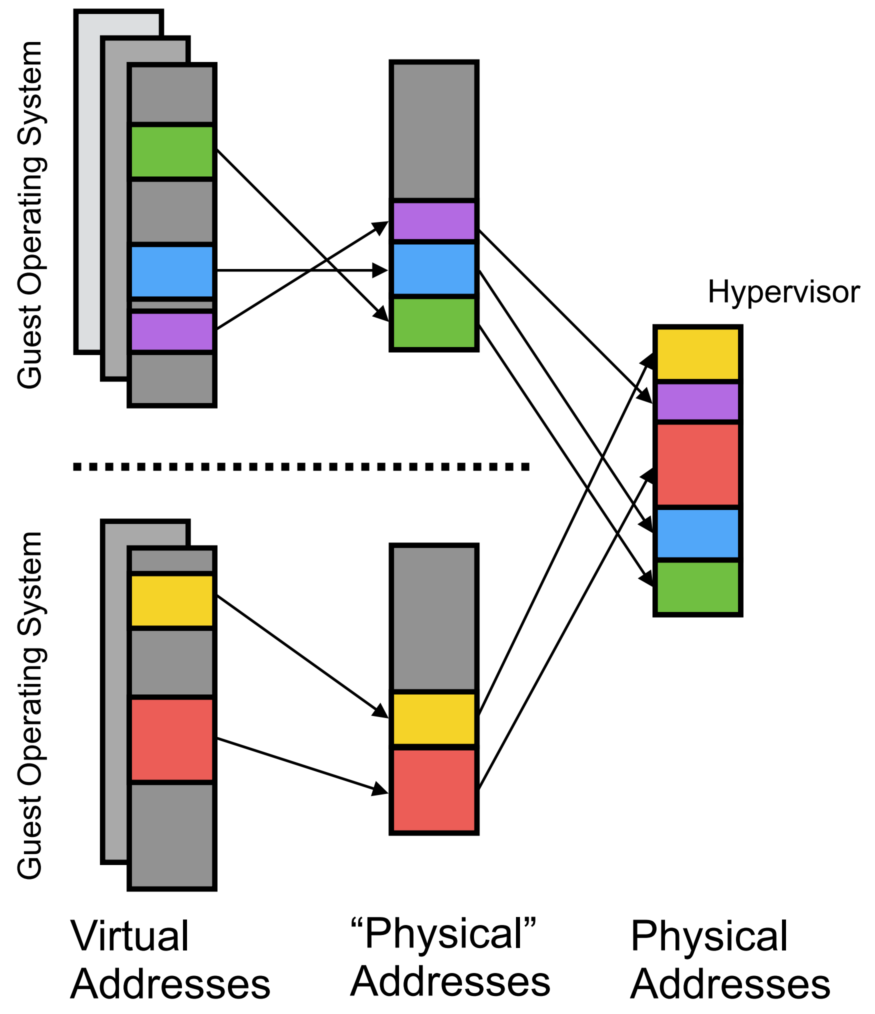
Hardware Virtualisation: Paravirtualisation

- Paravirtualisation – provide a virtual machine that is similar, but not identical, to underlying hardware
- Guest operating systems aware of virtualisation
  - Guests are "ported" to run on the virtualised environment – modified to never execute privileged instructions
  - Hypervisor provides an API:
    - Cache control
    - Pages tables and virtual memory
    - Interrupt handlers
    - …
  - Guest operating systems call that API – much as a user process calls into an operating system kernel
  - Relies on cooperation between hypervisor and guest operating systems
    - Needed if hardware doesn't provide a hypervisor privilege level – but require trust in guests (trust in the guest kernels, not in the applications that run on them)

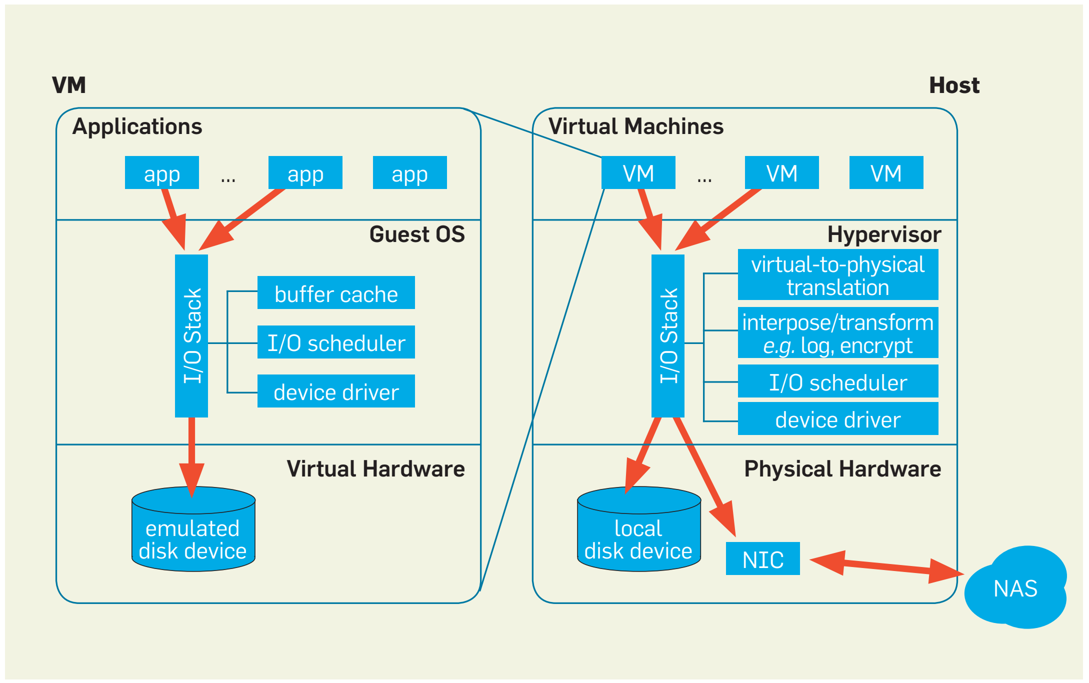
Hardware Virtualisation: Memory

- Operating systems provide virtual addressing
  - Each process believes it has access to the entire address space
  - Kernel configures hardware address translation tables – mapping virtual to physical addresses, and isolating the different processes
- Hypervisor requires additional translation
  - Guest operating systems believe they own the entire address space
  - Hypervisor maps this onto physical addresses, isolating the guest operating systems – needs hardware support
  - "Page table virtualisation"
- Must support all devices that access memory
  - Direct memory access by storage/networking devices
  - Access to memory mapped hardware registers

Two-stage address translation



Hardware Virtualisation: Device Drivers

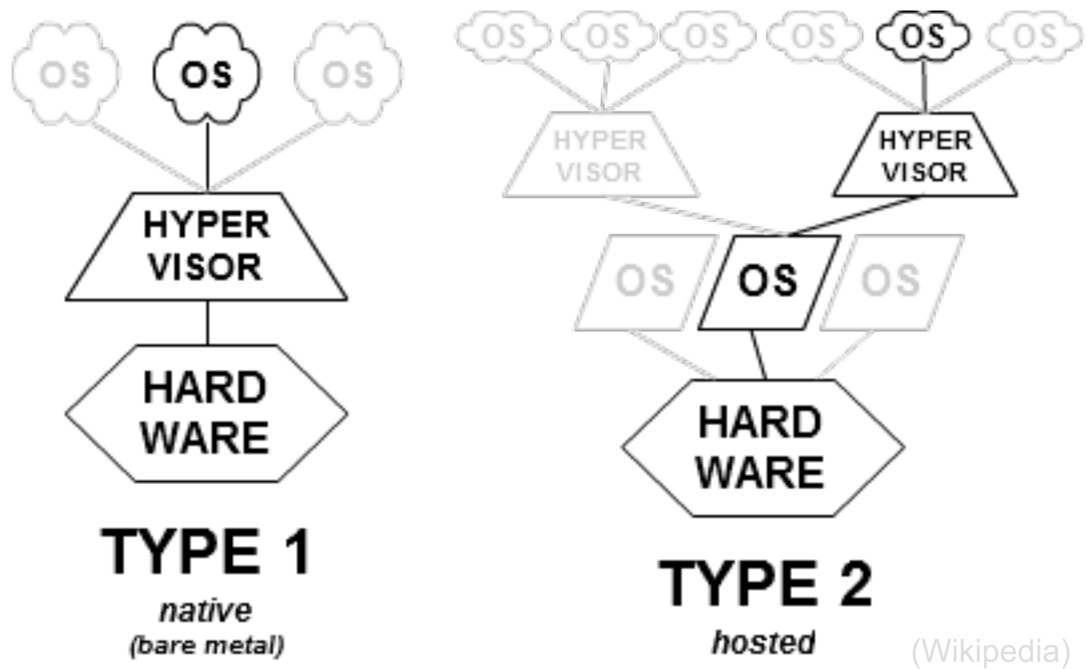


Source: C. Waldspurger and M. Rosenblum, "I/O Virtualisation", CACM, Vol 55, Num 1, pp 66-73, Jan. 2012. DOI: 10.1145/2063176.2063194

- Similar to two-stage address translation, interrupts and other hardware accesses are virtualised
  - Hardware support from CPU, PCI bus controllers, BIOS, etc.
  - Example: PCI single root I/O virtualisation
  - Software support for hypervisor – arbitrate access to real hardware (e.g., only one guest operating system can actually read the keyboard at once)
- Some devices can be shared between guest operating systems
  - Storage devices might appear as an entire device, but only give access to only a single partition – by translating block addresses
  - Network devices can have multiple MAC addresses that are used to deliver packets to particular guests

Types of Hypervisor

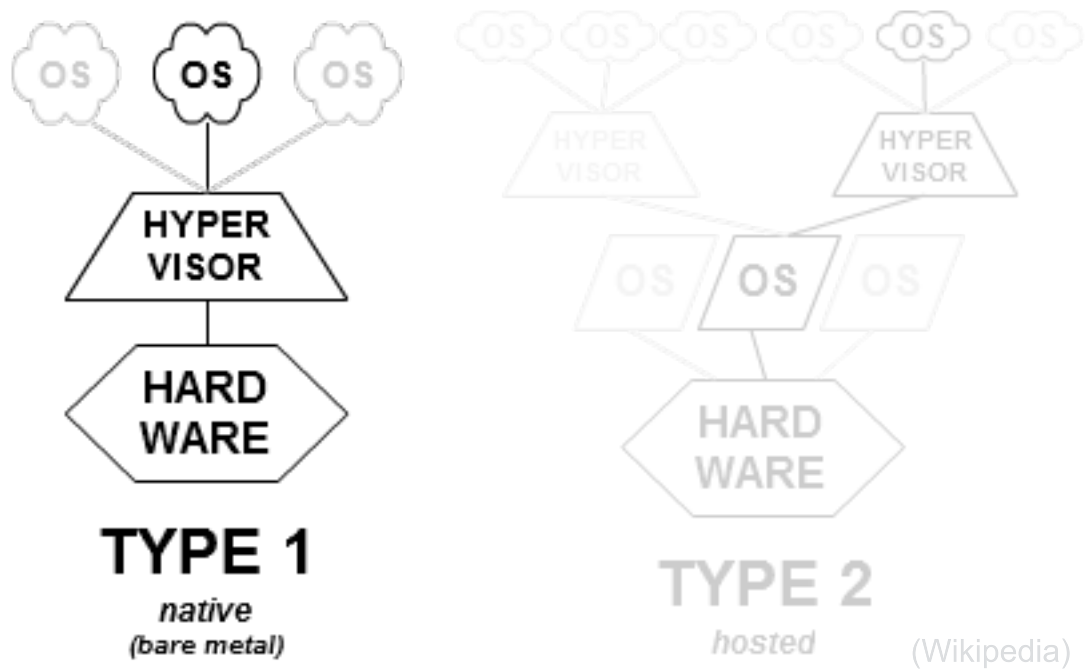
- Type 1 ("native") hypervisor:
  - Hypervisor is the operating system for the underlying hardware
  - Requires its own device drivers, and has to be ported to new hardware platforms
  - Example: Xen
- Type 2 ("hosted") hypervisor:
  - The hypervisor is an application running on an existing operating system
  - Example: VMWare, VirtualBox



(Wikipedia)

Type 1 Hypervisors

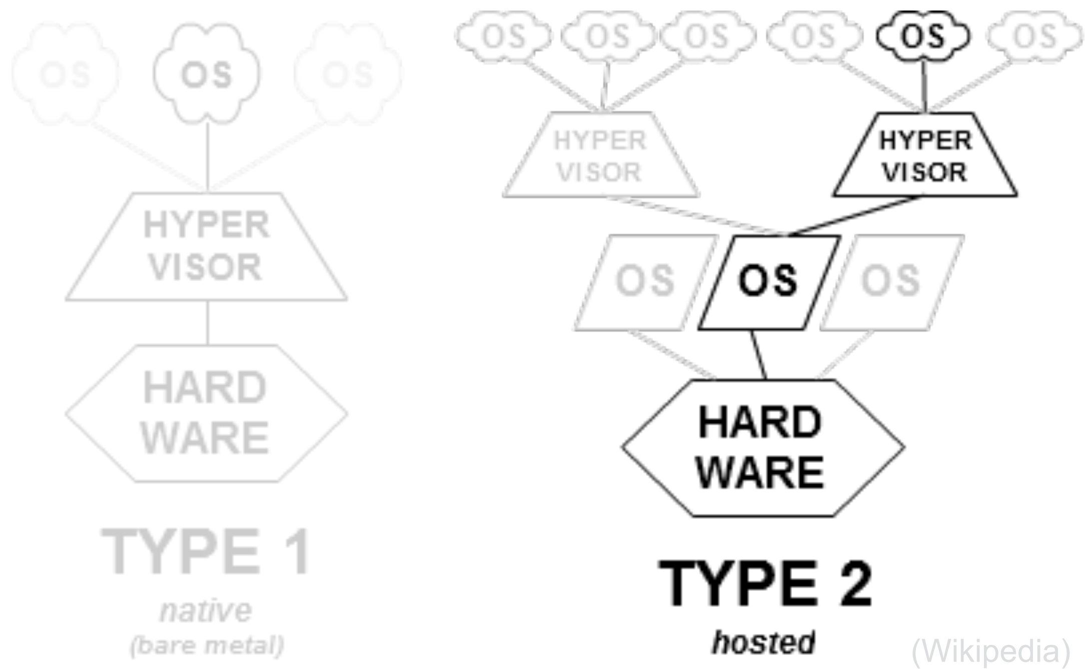
- The hypervisor is an operating system
  - Controls/arbitrates access to the hardware
  - Schedules the execution of guest operating systems
- Needs device drivers for the underlying hardware
  - Some devices are driven by the hypervisor – the hypervisor executes operations on behalf of the guests
  - For other devices, configure resources to give an individual guest direct access to resources
- Exposes a control API
  - Allows one of the guests to act as controller for the hypervisor
  - Example: “domain 0” in Xen
- Extremely efficient – basis for most cloud hosting platforms



(Wikipedia)

Type 2 Hypervisors

- The hypervisor is a process that runs within an existing operating system
    - Doesn't hard privileged access to the underlying hardware, CPU, etc.
    - Cannot run unmodified guest operating systems – since it doesn't have privilege to virtualise their operation
- Requires paravirtualisation
    - Guest operating systems must be modified to call hypervisor for privileged operations
    - Hypervisor itself depends on access given by the underlying operating system
    - Often requires software emulation of the hardware – e.g., interrupt handlers, I/O, page tables – can be very slow
- Good for development – not useful as high performance or as a cloud hosting platform



(Wikipedia)

Management of Virtual Machines

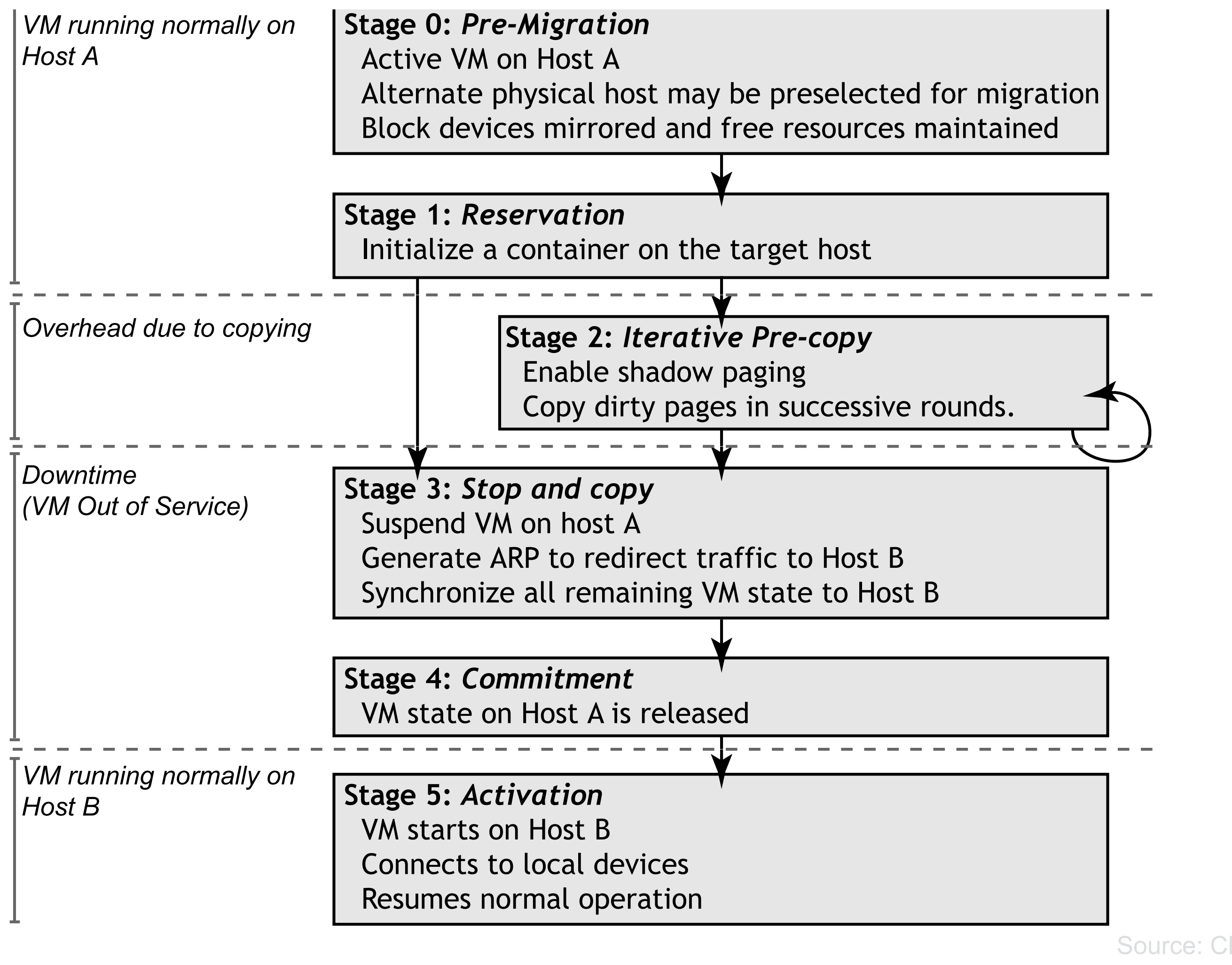
- Type 1 hypervisors need a management interface
  - Features:
    - Configuring, starting, stopping, and migrating VMs
    - Managing underlying hardware
    - Managing network configuration
  - Hyper-call API – works like system call API (INT 0x82 vs. INT 0x80) – that can be called by management software
  - Designed for large-scale, automated, administration – virtual machines as a service
- Type 2 hypervisors can be configured as any other application on the host operating system
  - Designed for small-scale personal use

Management of Virtual Machines

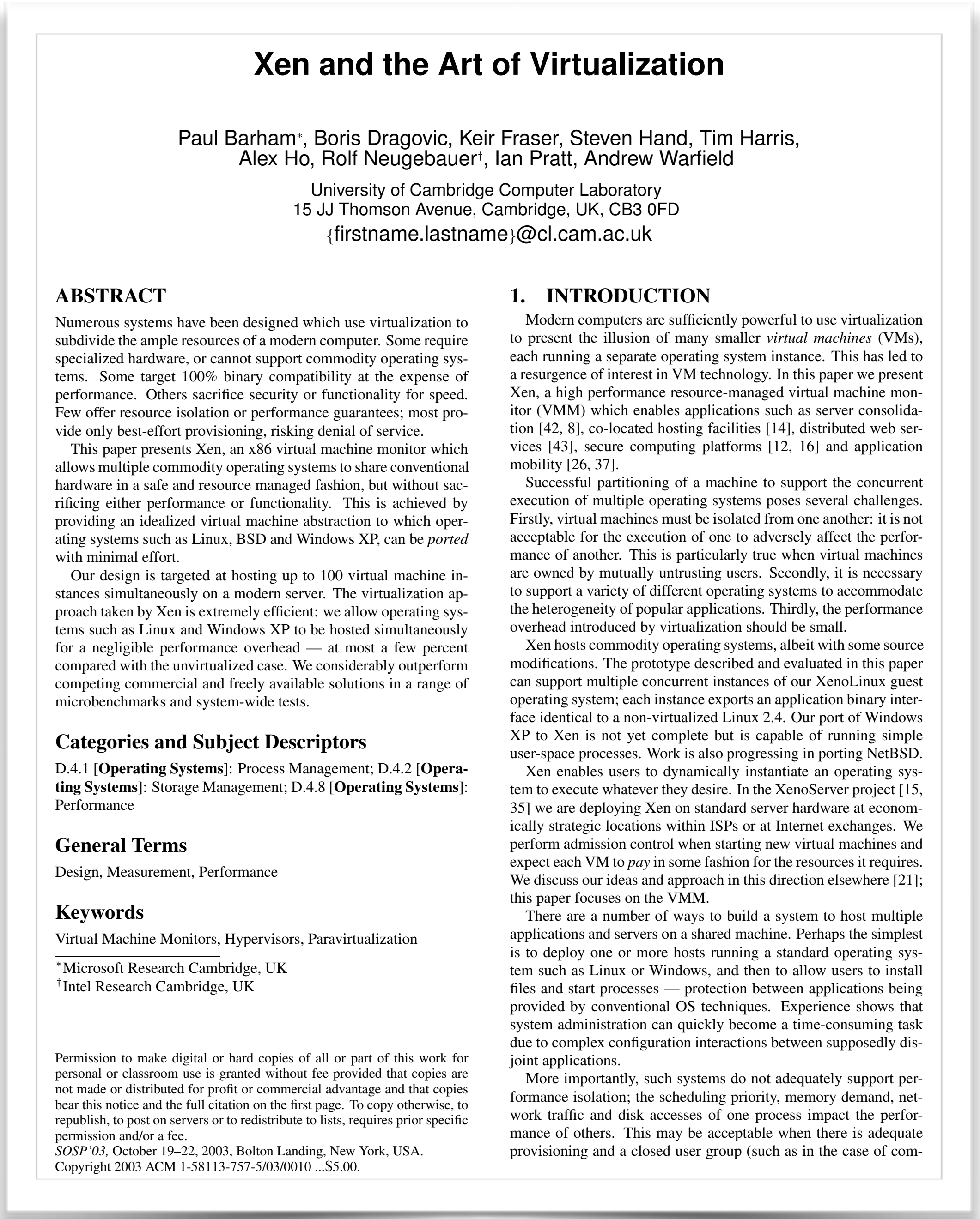
- Hypervisors allow creation of *virtual machines*
  - Hypervisor + guest operating system + local state → virtual machine
  - Allows on-demand instantiation of servers within a virtualised system
  - Can be (largely) independent of underlying hardware
    - Configure VM with generic device drivers, reasonable amount of memory, etc.
    - A subset of the real hardware available in the hosting environment
    - VM can be instantiated on any physical system that meets requirements
    - Many VMs can be instantiated on a single machine – performance constraints?
    - User of the VM typically unaware which physical hardware used; physical hardware can change over time
    - Virtual machines can be migrated between physical servers when stopped, but also *while in use* if care is taken
- Enables cloud computing platforms and computing as a service

Live Migration

- Possible to migrate running guest operating systems to new system
  - Move VM between physical servers, *while running*, without users noticing
- Live migration procedure:
  - Activate new VM, with matching hardware resources
  - Copy contents of memory and storage – keep track of writes after copy
  - Stop VM, copy any outstanding memory, storage, and other state
  - Reassign network addresses to new host
  - Restart VM on new host
- Performance heavily dependent on amount of active I/O on VM; speed of network
  - Sub-second downtime, with performance impacts for several seconds preceding is possible, with care


VM running normally on Host A
Stage 0: Pre-Migration
Active VM on Host A
Alternate physical host may be preselected for migration
Block devices mirrored and free resources maintained
Stage 1: Reservation
Initialize a container on the target host
Overhead due to copying
Stage 2: Iterative Pre-copy
Enable shadow paging
Copy dirty pages in successive rounds.
Downtime (VM Out of Service)
Stage 3: Stop and copy
Suspend VM on host A
Generate ARP to redirect traffic to Host B
Synchronize all remaining VM state to Host B
Stage 4: Commitment
VM state on Host A is released
VM running normally on Host B
Stage 5: Activation
VM starts on Host B
Connects to local devices
Resumes normal operation
Source: Cl

Example: Xen



Xen and the Art of Virtualization

Paul Barham[*], Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer[†], Ian Pratt, Andrew Warfield

University of Cambridge Computer Laboratory
15 JJ Thomson Avenue, Cambridge, UK, CB3 0FD
{firstname.lastname}@cl.cam.ac.uk

ABSTRACT

Numerous systems have been designed which use virtualization to subdivide the ample resources of a modern computer. Some require specialized hardware, or cannot support commodity operating systems. Some target 100% binary compatibility at the expense of performance. Others sacrifice security or functionality for speed. Few offer resource isolation or performance guarantees; most provide only best-effort provisioning, risking denial of service.

This paper presents Xen, an x86 virtual machine monitor which allows multiple commodity operating systems to share conventional hardware in a safe and resource managed fashion, but without sacrificing either performance or functionality. This is achieved by providing an idealized virtual machine abstraction to which operating systems such as Linux, BSD and Windows XP, can be *ported* with minimal effort.

Our design is targeted at hosting up to 100 virtual machine instances simultaneously on a modern server. The virtualization approach taken by Xen is extremely efficient: we allow operating systems such as Linux and Windows XP to be hosted simultaneously for a negligible performance overhead — at most a few percent compared with the unvirtualized case. We considerably outperform competing commercial and freely available solutions in a range of microbenchmarks and system-wide tests.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management; D.4.2 [**Operating Systems**]: Storage Management; D.4.8 [**Operating Systems**]: Performance

General Terms

Design, Measurement, Performance

Keywords

Virtual Machine Monitors, Hypervisors, Paravirtualization

[*]Microsoft Research Cambridge, UK
[†]Intel Research Cambridge, UK

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.
Copyright 2003 ACM 1-58113-757-5/03/0010 ...$5.00.

1. INTRODUCTION

Modern computers are sufficiently powerful to use virtualization to present the illusion of many smaller *virtual machines* (VMs), each running a separate operating system instance. This has led to a resurgence of interest in VM technology. In this paper we present Xen, a high performance resource-managed virtual machine monitor (VMM) which enables applications such as server consolidation [42, 8], co-located hosting facilities [14], distributed web services [43], secure computing platforms [12, 16] and application mobility [26, 37].

Successful partitioning of a machine to support the concurrent execution of multiple operating systems poses several challenges. Firstly, virtual machines must be isolated from one another: it is not acceptable for the execution of one to adversely affect the performance of another. This is particularly true when virtual machines are owned by mutually untrusting users. Secondly, it is necessary to support a variety of different operating systems to accommodate the heterogeneity of popular applications. Thirdly, the performance overhead introduced by virtualization should be small.

Xen hosts commodity operating systems, albeit with some source modifications. The prototype described and evaluated in this paper can support multiple concurrent instances of our XenoLinux guest operating system; each instance exports an application binary interface identical to a non-virtualized Linux 2.4. Our port of Windows XP to Xen is not yet complete but is capable of running simple user-space processes. Work is also progressing in porting NetBSD.

Xen enables users to dynamically instantiate an operating system to execute whatever they desire. In the XenoServer project [15, 35] we are deploying Xen on standard server hardware at economically strategic locations within ISPs or at Internet exchanges. We perform admission control when starting new virtual machines and expect each VM to *pay* in some fashion for the resources it requires. We discuss our ideas and approach in this direction elsewhere [21]; this paper focuses on the VMM.

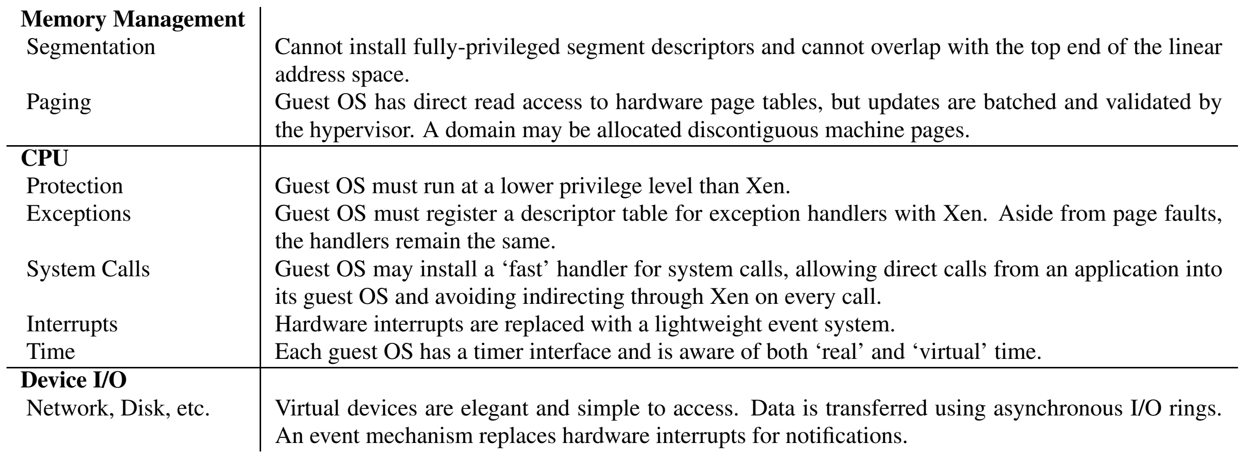
There are a number of ways to build a system to host multiple applications and servers on a shared machine. Perhaps the simplest is to deploy one or more hosts running a standard operating system such as Linux or Windows, and then to allow users to install files and start processes — protection between applications being provided by conventional OS techniques. Experience shows that system administration can quickly become a time-consuming task due to complex configuration interactions between supposedly disjoint applications.

More importantly, such systems do not adequately support performance isolation; the scheduling priority, memory demand, network traffic and disk accesses of one process impact the performance of others. This may be acceptable when there is adequate provisioning and a closed user group (such as in the case of com-

- A modern type 1 hypervisor for x86
- Supports two types of virtualisation:
  - Full virtualisation – needs hardware support
    - Older x86 CPUs had no way of trapping supervisor mode instruction calls, to allow efficient emulation by a hypervisor; modern x86 CPUs do provide this
  - Paravirtualisation
    - Guest operating system is ported to a new system architecture, which looks like x86 with problematic features adapted to ease virtualisation
    - Operating system knows it is running in a VM
    - Processes running within the OS cannot tell that virtualisation is in use
- Control operations delegated to a privileged guest operating system
  - Known as Domain0
  - Provides policy and controls the hypervisor

Xen

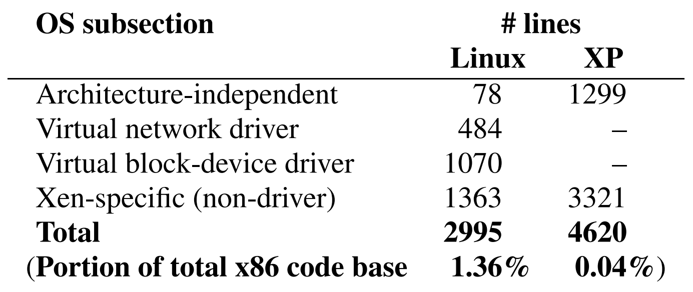
- Guest operating systems must be ported to Xen
- The Xen hypervisor provides an API to mediate hardware access by guest operating systems:
    - Privileged CPU instructions
    - Memory management
    - Interrupt handling
    - Access to network, disk, etc., hardware



Memory Management	
Segmentation	Cannot install fully-privileged segment descriptors and cannot overlap with the top end of the linear address space.
Paging	Guest OS has direct read access to hardware page tables, but updates are batched and validated by the hypervisor. A domain may be allocated discontiguous machine pages.
CPU	
Protection	Guest OS must run at a lower privilege level than Xen.
Exceptions	Guest OS must register a descriptor table for exception handlers with Xen. Aside from page faults, the handlers remain the same.
System Calls	Guest OS may install a 'fast' handler for system calls, allowing direct calls from an application into its guest OS and avoiding indirecting through Xen on every call.
Interrupts	Hardware interrupts are replaced with a lightweight event system.
Time	Each guest OS has a timer interface and is aware of both 'real' and 'virtual' time.
Device I/O	
Network, Disk, etc.	Virtual devices are elegant and simple to access. Data is transferred using asynchronous I/O rings. An event mechanism replaces hardware interrupts for notifications.

Table 1: The paravirtualized x86 interface.

- Required changes to guest systems are small
    - Porting guests to an API that allows efficient virtualisation allows significant performance benefits
    - Simplifies hypervisor design for x86, where the processor architecture makes full virtualisation expensive
- The user-space API of guest operating systems is unchanged



OS subsection	**# lines**	
	Linux	**XP**
Architecture-independent	78	1299
Virtual network driver	484	–
Virtual block-device driver	1070	–
Xen-specific (non-driver)	1363	3321
Total	**2995**	**4620**
(**Portion of total x86 code base**	**1.36%**	**0.04%**)

Table 2: The simplicity of porting commodity OSes to Xen. The cost metric is the number of lines of reasonably commented and formatted code which are modified or added compared with the original x86 code base (excluding device drivers).

Xen

- Xen provides mechanisms, but does not define policy
  - Virtualisation primitives are well understood and stable
  - Control mechanisms and policies evolve to match business models
  - Separation of mechanism and policy allows these to develop at separate rates
- Interactions via a mix of hyper-calls and message passing
  - Hyper-calls allow a guest operating system to call into the hypervisor
    - Implement the Domain0 control interface
    - Implement the paravirtualisation interface
  - Message passing allows the hypervisor to queue events to be processed by a guest
    - Callbacks to a guest, to indicate completion of request or receipt of asynchronous data

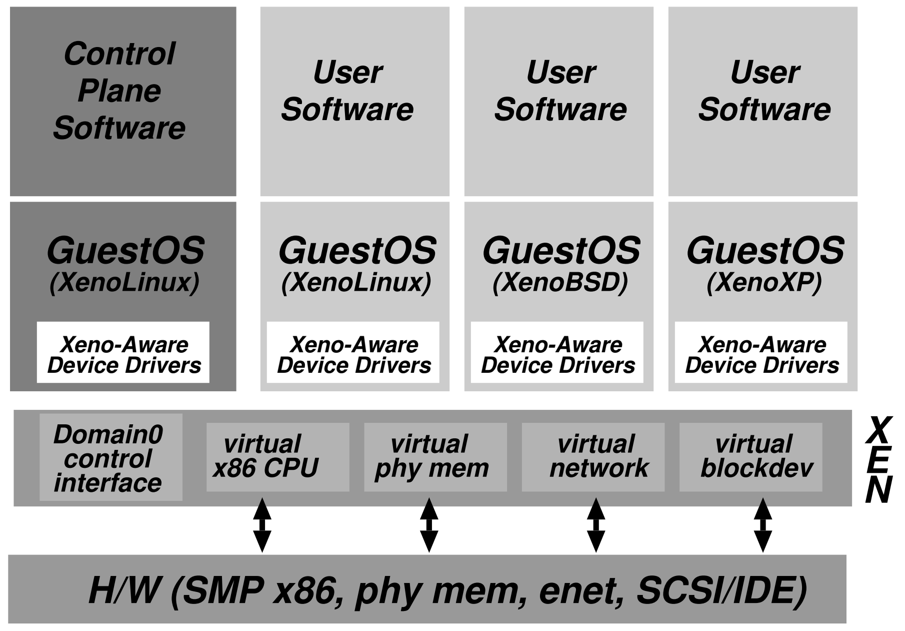


Figure 1: The structure of a machine running the Xen hypervisor, hosting a number of different guest operating systems, including *Domain0* running control software in a XenoLinux environment.

Further Reading

- P. Barham *et al,* "Xen and the art of virtualization", Proc. ACM Symposium on Operating Systems Principles, October 2003. DOI:10.1145/945445.945462
  - Trade-offs of paravirtualisation vs. full virtualisation?
  - What needs to be done to port an OS to Xen?
  - Is paravirtualisation worthwhile, when compared to full system virtualisation?
  - How do Dom0 and device drivers work?

Xen and the Art of Virtualization

Paul Barham[*], Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris,
Alex Ho, Rolf Neugebauer[†], Ian Pratt, Andrew Warfield
University of Cambridge Computer Laboratory
15 JJ Thomson Avenue, Cambridge, UK, CB3 0FD
{firstname.lastname}@cl.cam.ac.uk

ABSTRACT

Numerous systems have been designed which use virtualization to subdivide the ample resources of a modern computer. Some require specialized hardware, or cannot support commodity operating systems. Some target 100% binary compatibility at the expense of performance. Others sacrifice security or functionality for speed. Few offer resource isolation or performance guarantees; most provide only best-effort provisioning, risking denial of service.

This paper presents Xen, an x86 virtual machine monitor which allows multiple commodity operating systems to share conventional hardware in a safe and resource managed fashion, but without sacrificing either performance or functionality. This is achieved by providing an idealized virtual machine abstraction to which operating systems such as Linux, BSD and Windows XP, can be *ported* with minimal effort.

Our design is targeted at hosting up to 100 virtual machine instances simultaneously on a modern server. The virtualization approach taken by Xen is extremely efficient: we allow operating systems such as Linux and Windows XP to be hosted simultaneously for a negligible performance overhead — at most a few percent compared with the unvirtualized case. We considerably outperform competing commercial and freely available solutions in a range of microbenchmarks and system-wide tests.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management; D.4.2 [**Operating Systems**]: Storage Management; D.4.8 [**Operating Systems**]: Performance

General Terms

Design, Measurement, Performance

Keywords

Virtual Machine Monitors, Hypervisors, Paravirtualization

[*]Microsoft Research Cambridge, UK
[†]Intel Research Cambridge, UK

1. INTRODUCTION

Modern computers are sufficiently powerful to use virtualization to present the illusion of many smaller *virtual machines* (VMs), each running a separate operating system instance. This has led to a resurgence of interest in VM technology. In this paper we present Xen, a high performance resource-managed virtual machine monitor (VMM) which enables applications such as server consolidation [42, 8], co-located hosting facilities [14], distributed web services [43], secure computing platforms [12, 16] and application mobility [26, 37].

Successful partitioning of a machine to support the concurrent execution of multiple operating systems poses several challenges. Firstly, virtual machines must be isolated from one another: it is not acceptable for the execution of one to adversely affect the performance of another. This is particularly true when virtual machines are owned by mutually untrusting users. Secondly, it is necessary to support a variety of different operating systems to accommodate the heterogeneity of popular applications. Thirdly, the performance overhead introduced by virtualization should be small.

Xen hosts commodity operating systems, albeit with some source modifications. The prototype described and evaluated in this paper can support multiple concurrent instances of our XenoLinux guest operating system; each instance exports an application binary interface identical to a non-virtualized Linux 2.4. Our port of Windows XP to Xen is not yet complete but is capable of running simple user-space processes. Work is also progressing in porting NetBSD.

Xen enables users to dynamically instantiate an operating system to execute whatever they desire. In the XenoServer project [15, 35] we are deploying Xen on standard server hardware at economically strategic locations within ISPs or at Internet exchanges. We perform admission control when starting new virtual machines and expect each VM to *pay* in some fashion for the resources it requires. We discuss our ideas and approach in this direction elsewhere [21]; this paper focuses on the VMM.

There are a number of ways to build a system to host multiple applications and servers on a shared machine. Perhaps the simplest is to deploy one or more hosts running a standard operating system such as Linux or Windows, and then to allow users to install files and start processes — protection between applications being provided by conventional OS techniques. Experience shows that system administration can quickly become a time-consuming task due to complex configuration interactions between supposedly disjoint applications.

More importantly, such systems do not adequately support performance isolation; the scheduling priority, memory demand, network traffic and disk accesses of one process impact the performance of others. This may be acceptable when there is adequate provisioning and a closed user group (such as in the case of com-