University of Glasgow

School of Computing Science

Virtualisation

Advanced Operating Systems (M)

Lecture 8

Colin Perkins | https://csperkins.org/ | Copyright © 2018 | This work is licensed under the Creative Commons Attribution-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Lecture Outline

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

CC BY ND

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

CC BY ND

Full System Virtualisation

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



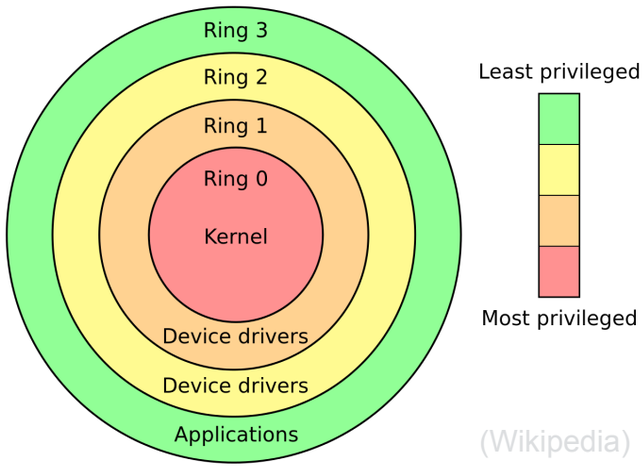
(source: Wikipedia) IBM System/360

- 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

CC BY ND

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



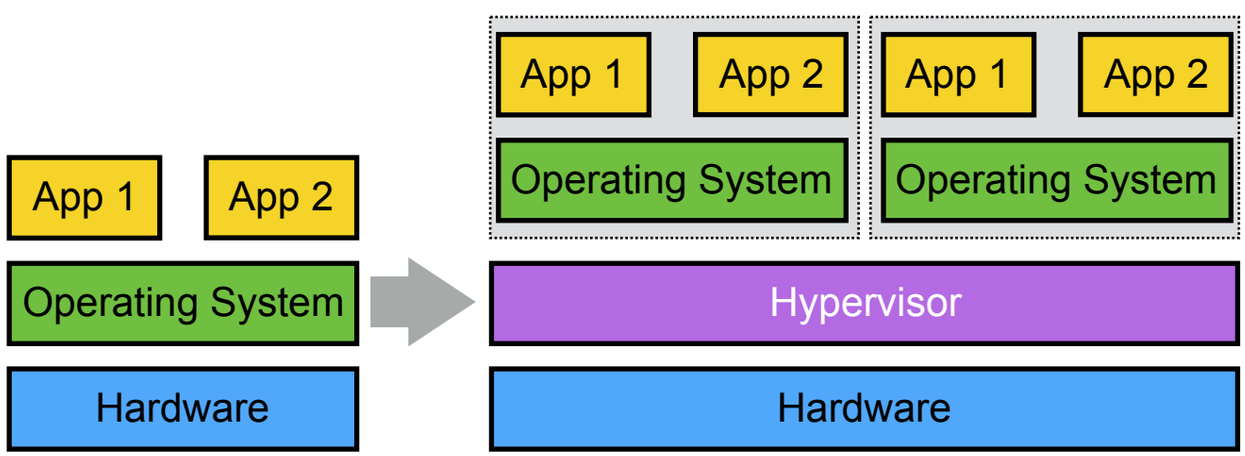
(Wikipedia)

- 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

CC BY ND

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

CC BY ND

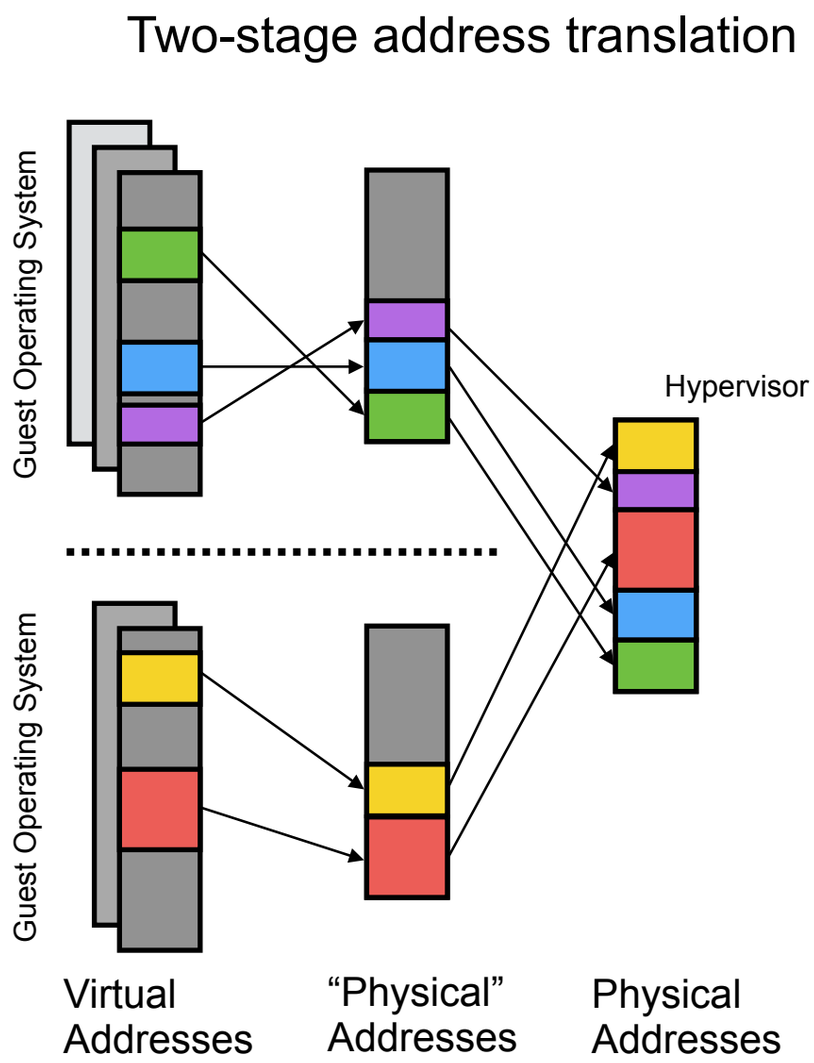
Hardware Virtualisation: Paravirtualisation

- 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)

CC BY ND

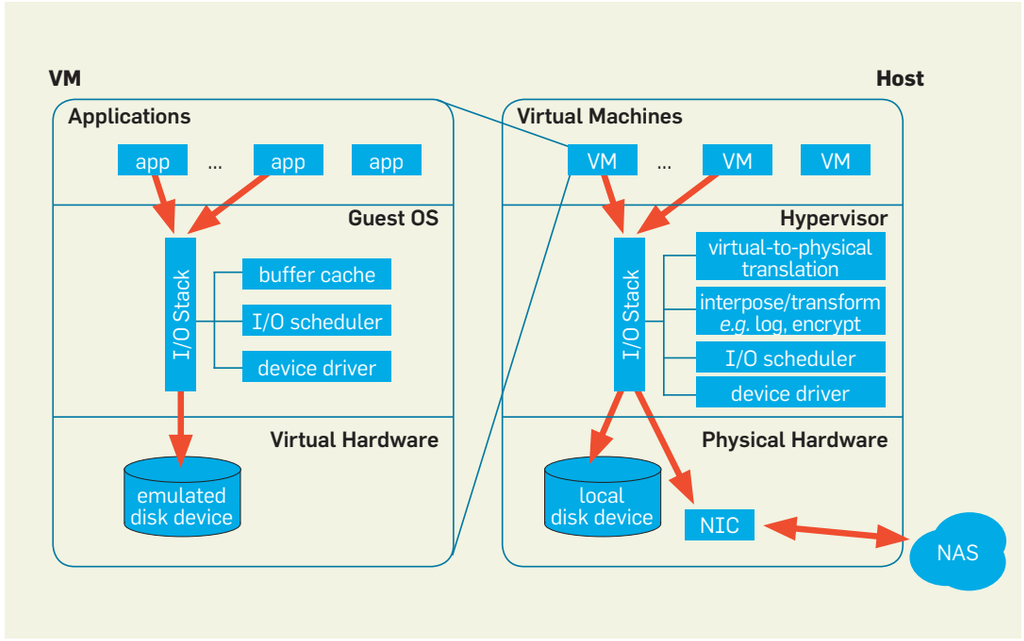
Hardware Virtualisation: Memory

- 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 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



Colin Perkins | https://csperkins.org/ | Copyright © 2018

Hardware Virtualisation: Device Drivers



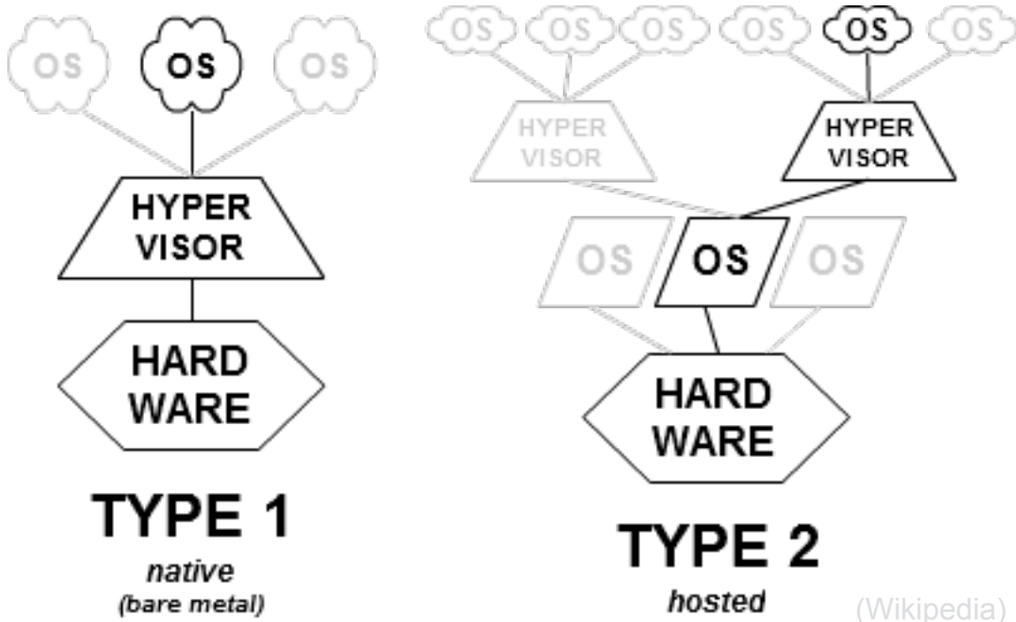
Source: C. Waldspurger and M. Rosenblum, "I/O Virtualisation", CACM, Vol 55, Num 1, pp 66-73, Jan. 2012. DOI: 10.1145/2063176.2063194

- 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

CC BY ND

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

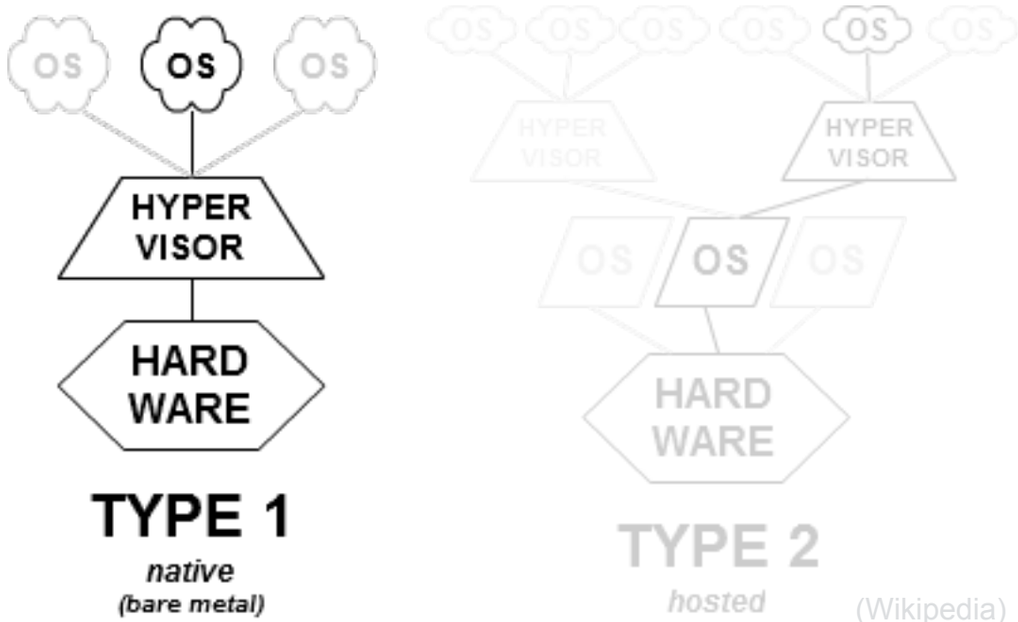


(Wikipedia)

Colin Perkins | https://csperkins.org/ | Copyright © 2018

Type 1 Hypervisors

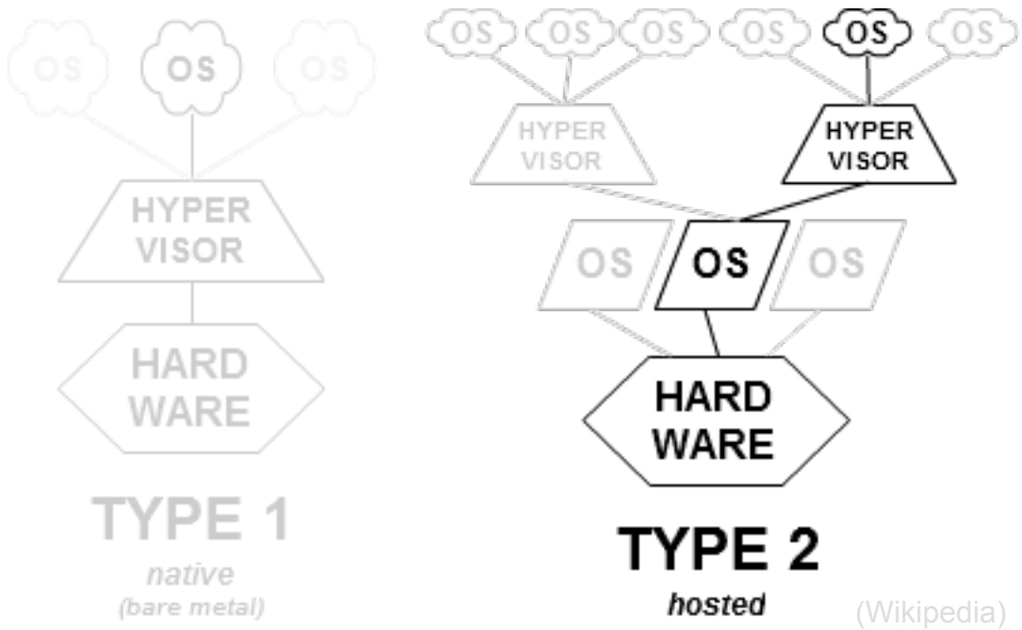
- The hypervisor is an operating system
  - Controls/arbitrates access to the hardware
  - Schedules the execution of guest operating systems
- Needs device drivers for the underlying hardware
  - Some devices are driven by the hypervisor – the hypervisor executes operations on behalf of the guests
  - 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



Colin Perkins | https://csperkins.org/ | Copyright © 2018

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)

Colin Perkins | https://csperkins.org/ | Copyright © 2018

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

CC BY ND

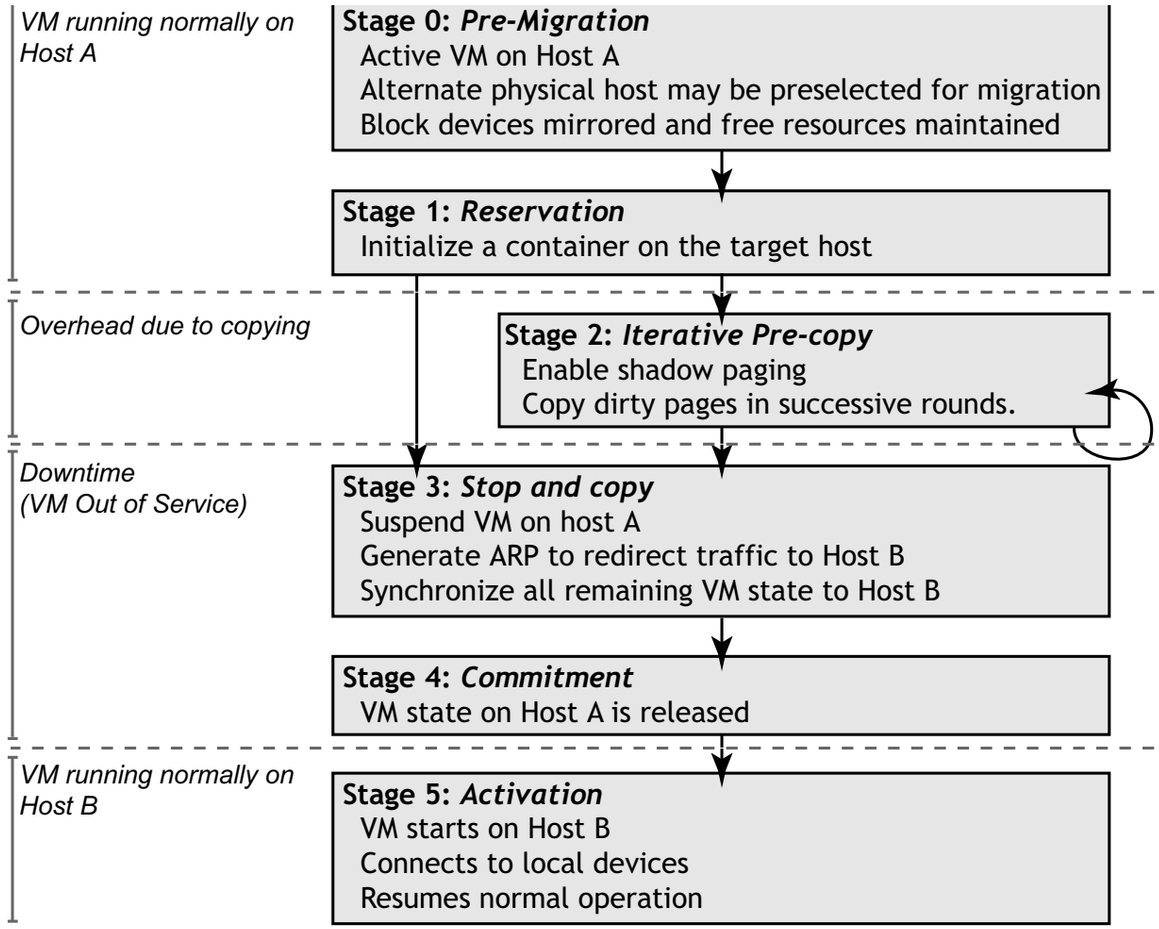
Management of Virtual Machines

- 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

CC BY ND

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 impact for several seconds preceding is possible, if care taken



Source: Cl

CC BY ND

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

CC BY ND

Xen Paravirtualisation

- Guest operating systems is 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).

CC BY ND

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

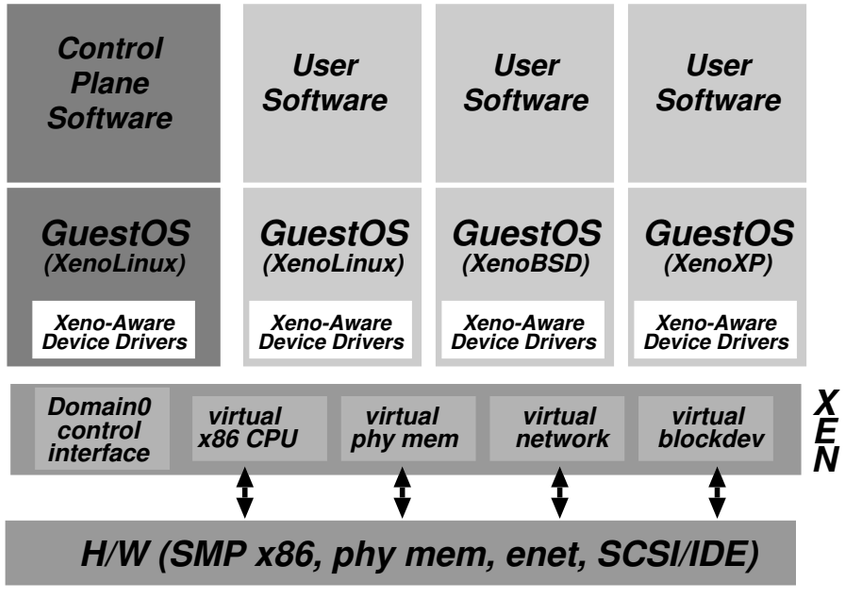


Figure 1: The structure of a machine running the Xen hypervisor, hosting a number of different guest operating systems, including *Domain0* running control software in a XenoLinux environment.

CC BY ND

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

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-

CC BY ND

Virtualisation: Jails and Unikernels

Jails and Unikernels

- Alternatives to full virtualisation
- Sandboxing processes:
  - FreeBSD jails and Linux containers
  - Container management
- Unikernels and library operating systems

Colin Perkins | https://csperkins.org/ | Copyright © 2018

Alternatives to Full Virtualisation

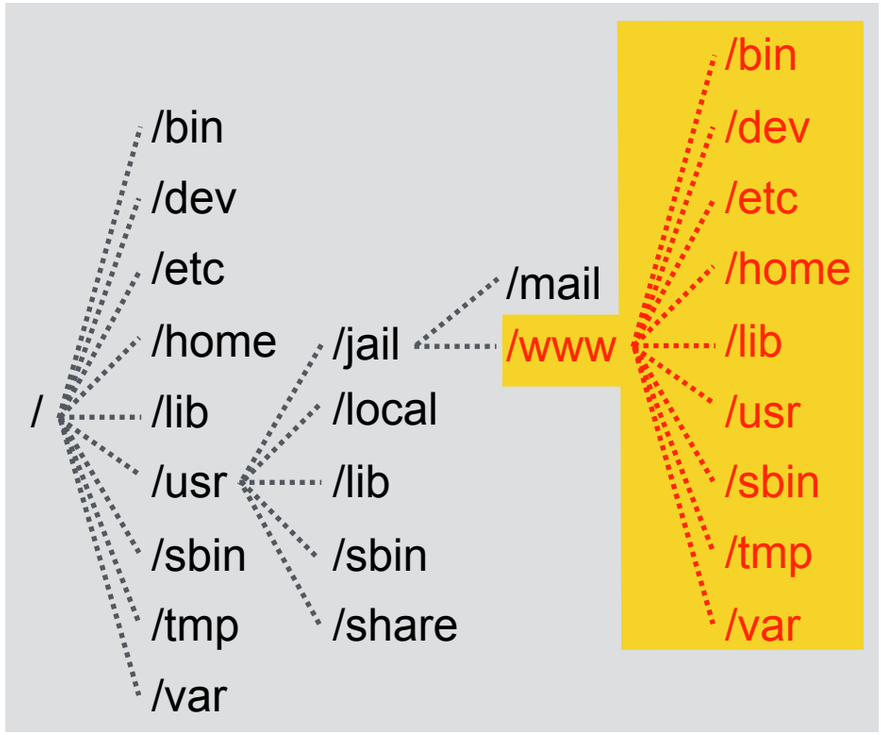
- Full operating system virtualisation is expensive
  - Need to run a hypervisor
  - Need to run a complete guest operating system instance for each VM
  - High memory and storage overhead, high CPU load from running multiple OS instances on a single machine – but excellent isolation between VMs
- Running multiple services on a single OS instance can offer insufficient isolation
  - All must share the same OS
  - A bug in a privileged service can easily compromise entire system – all services running on the host, and any unrelated data
- A *sandbox* is desirable – to isolate multiple services running on a single operating system

CC BY ND

Sandboxing: Jails and Containers

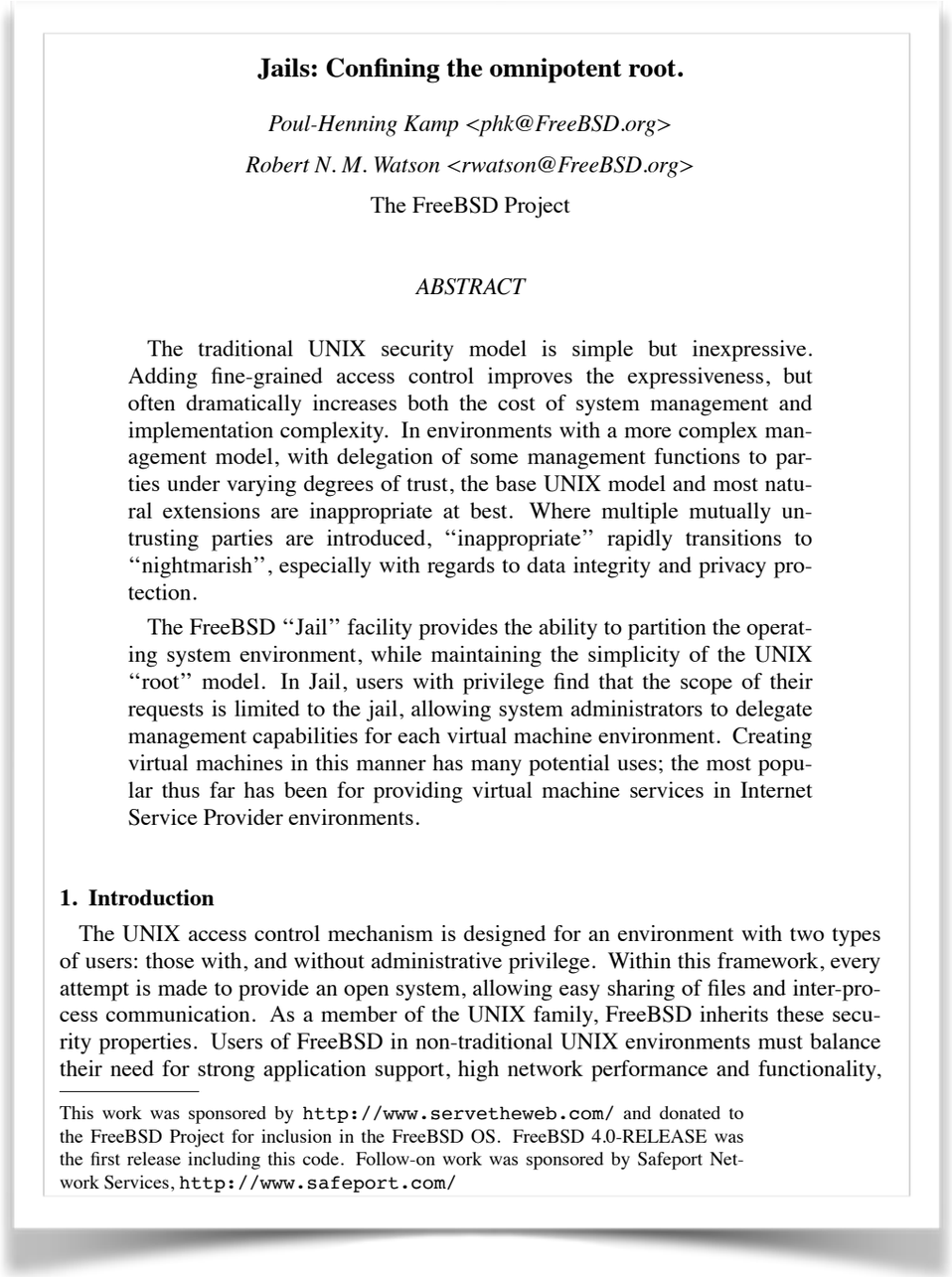
- Concept – lightweight virtualisation
  - A single operating system kernel
  - Multiple services
  - Each service runs within a jail, a service specific container, running just what's needed for that application

- Jail restricted to see only a subset of the filesystem and processes of the host
  - A sandbox within the operating system
  - Partially virtualised



Colin Perkins | https://csperkins.org/ | Copyright © 2018

Example: FreeBSD Jail subsystem



Jails: Confining the omnipotent root.

Poul-Henning Kamp <phk@FreeBSD.org>

Robert N. M. Watson <rwatson@FreeBSD.org>

The FreeBSD Project

ABSTRACT

The traditional UNIX security model is simple but inexpressive. Adding fine-grained access control improves the expressiveness, but often dramatically increases both the cost of system management and implementation complexity. In environments with a more complex management model, with delegation of some management functions to parties under varying degrees of trust, the base UNIX model and most natural extensions are inappropriate at best. Where multiple mutually untrusting parties are introduced, ''inappropriate'' rapidly transitions to ''nightmarish'', especially with regards to data integrity and privacy protection.

The FreeBSD ''Jail'' facility provides the ability to partition the operating system environment, while maintaining the simplicity of the UNIX ''root'' model. In Jail, users with privilege find that the scope of their requests is limited to the jail, allowing system administrators to delegate management capabilities for each virtual machine environment. Creating virtual machines in this manner has many potential uses; the most popular thus far has been for providing virtual machine services in Internet Service Provider environments.

1. Introduction

The UNIX access control mechanism is designed for an environment with two types of users: those with, and without administrative privilege. Within this framework, every attempt is made to provide an open system, allowing easy sharing of files and inter-process communication. As a member of the UNIX family, FreeBSD inherits these security properties. Users of FreeBSD in non-traditional UNIX environments must balance their need for strong application support, high network performance and functionality,

This work was sponsored by `http://www.servetheweb.com/` and donated to the FreeBSD Project for inclusion in the FreeBSD OS. FreeBSD 4.0-RELEASE was the first release including this code. Follow-on work was sponsored by Safeport Network Services, `http://www.safeport.com/`

- Unix has long had `chroot()`
  - Limits process to see a subset of the filesystem
  - Used in, e.g., web servers, to prevent buggy code from serving files outside chosen directory
- Jails extend this to also restrict access to other resources for jailed processes:
  - Limits process to see subset of the processes (the children of the initially jailed process)
  - Limits process to see specific network interface
  - Prevents a process from mounting/un-mounting file systems, creating device nodes, loading kernel modules, changing kernel parameters, configuring network interfaces, etc.
  - Even if running with `root` privilege
- Processes outside the jail can see into the jail – from outside, jailed processes look like regular processes

CC BY ND

Benefits of Jails

- Extremely lightweight implementation
  - Processes in the jail are regular Unix processes, with some additional privilege checks
  - FreeBSD implementation added/modified ~1,000 lines of code in total
  - Contents of the jail are limited: it's not a full operating system – just the set of binaries/libraries needed for the application
- Straightforward system administration
  - Few new concepts: no need to learn how to operate the hypervisor
  - Visibility into the jail eases debugging and administration; no new tools
- Portable
  - No separate hypervisor, with its own device drivers, needed
  - If the base system runs, so do the jails

CC BY ND

Disadvantages of Jails

- Virtualisation is imperfect
  - Full virtual machines can provide stronger isolation and performance guarantees – processes in jail can compete for resources with other processes in the system, and observe the effects of that competition
  - A process can discover that it's running in a jail – a process in a VM cannot determine that it's in a VM
- Jails are tied to the underlying operating system
  - A virtual machine can be paused, migrated to a different physical system, and resumed, without processes in the VM being aware – a jail cannot

CC BY ND

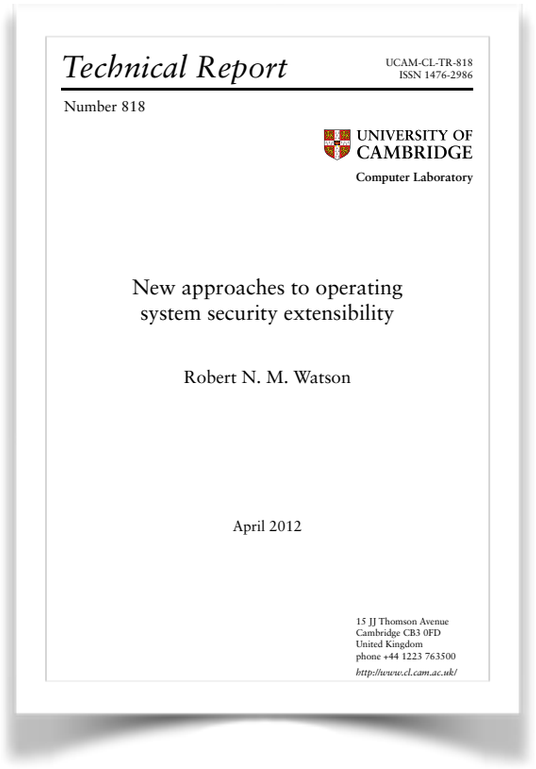
Example: Linux Containers

- Two inter-related kernel features:
  - Namespace isolation – separation of processes
  - Control groups “cgroups” – accounting for process CPU, memory, disk and network I/O usage
  - Both part of the mainline Linux kernel
- Combined to provide container-based virtualisation
  - Implemented via tools such as “lxc”, “OpenVZ”, “Linux-VServer”
  - Very similar functionality to FreeBSD Jails, but using the Linux kernel as the underlying OS
  - Run a subset of processes within a containers – can be a full OS-like environment, or something much for limited

CC BY ND

Example: iOS/macOS Sandbox

- All apps on iOS, and all apps from the Mac App Store on macOS, are sandboxed
  - Sandbox functionality varies with macOS/iOS version – gradually getting more sophisticated and more secure
  - Kernel heavily based on FreeBSD, but sandboxing doesn't use Jails
    - Mandatory access control – http://www.trustedbsd.org/mac.html
    - Apps are constrained to see a particular directory hierarchy – but given flexibility to see other directories outside this based on user input
    - Policy language allows flexible control over what OS functions are restricted – restricts access to files, processes, and other namespaces like Jails, but more flexibility in what parts of the system are exposed
    - At most restrictive, looks like a FreeBSD Jail containing a single-process and it's data directories
    - Can allow access to other OS services/directories as needed – more flexible, but more complex and hence harder to sandbox correctly
  - Heavily based on Robert Watson's PhD and work on FreeBSD MAC and Capsicum security framework
    - http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-818.pdf
- Trade-off: flexibility, complexity, security


Technical Report
UCAM-CL-TR-818
ISSN 1476-2986
Number 818
UNIVERSITY OF CAMBRIDGE
Computer Laboratory
New approaches to operating system security extensibility
Robert N. M. Watson
April 2012
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500
http://www.cl.cam.ac.uk/

BY ND

Container Management

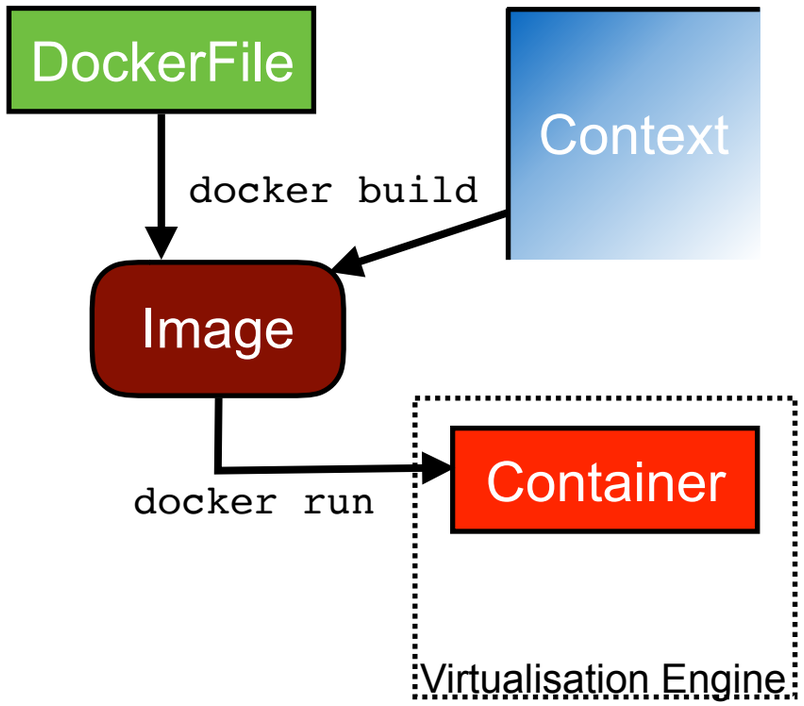
- Full virtualisation – VMs run complete OS, and can use standard OS update/management procedures
- Jails run a container with a subset of the full OS
  - Want minimal stack needed to support services running in the container
  - Less software installed in container → security vulnerability less likely

- How to ensure all images are up-to-date?
  - No longer sufficient to patch the OS, must also patch each container
  - Need to minimally subset the OS for the services in the container – can this be automated?
  - How to share service/container configuration?

CC BY ND

Example: Docker

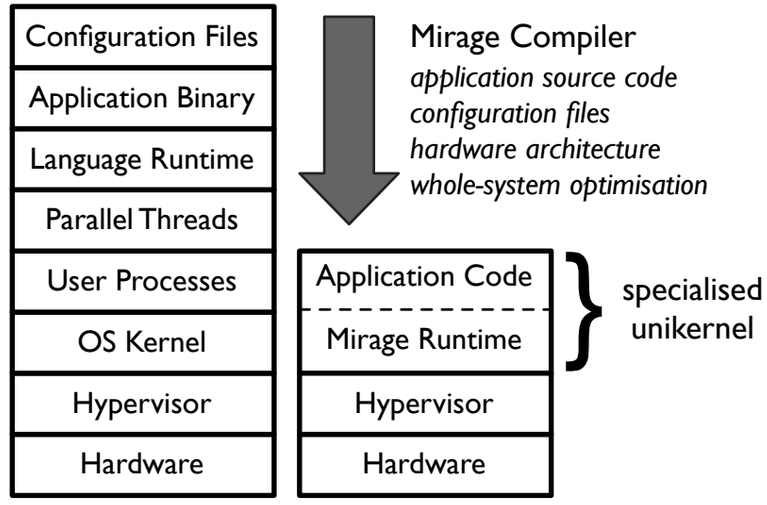
- DockerFile and context specify an image
  - Base system image
    - Docker maintains official, managed and security updated, images for popular operating systems
    - Images can build on/extend other images
  - Install additional binaries and libraries
  - Install additional data
  - Commands to execute when image is instantiated
  - Metadata about resulting image
- Images are immutable → instantiated in container
  - Virtualisation uses Linux containers, the macOS sandbox, or FreeBSD jails to instantiate images, depending on underlying operating system running the container
    - macOS and Windows run Linux in a hypervisor, and use that to execute containers
    - FreeBSD relies on native Linux system call emulation, and jails
  - Image can specify external filesystems to mount, to hold persistent data accessible by running image
  - A standardised way of packaging a software image to run in a container – plus easy-to-manage tools for instantiating a container



CC BY ND

Unikernels: Library Operating Systems

- Taking the idea of service-specific containers to its logical extreme – a library operating system
- Well-implemented kernels have clear interfaces between components
  - Device drivers, filesystems, network protocols, processes, …
- Implement as a set of libraries, that can link with an application
  - Only compile in the subset of functions needed to run the desired application (e.g., you don't use UDP, it isn't linked in to your kernel, etc…)
  - Link entire operating system kernel and application as a single binary
  - Run on bare hardware, or within a hypervisor
- Typically written in high-level, type-safe, languages, with expressive module systems – not backwards compatible with Unix
  - E.g., MirageOS largely written in OCaml
  - Type-safe languages give clear interface boundaries → automatic minimisation of images
- Single-purpose virtual machines for the cloud

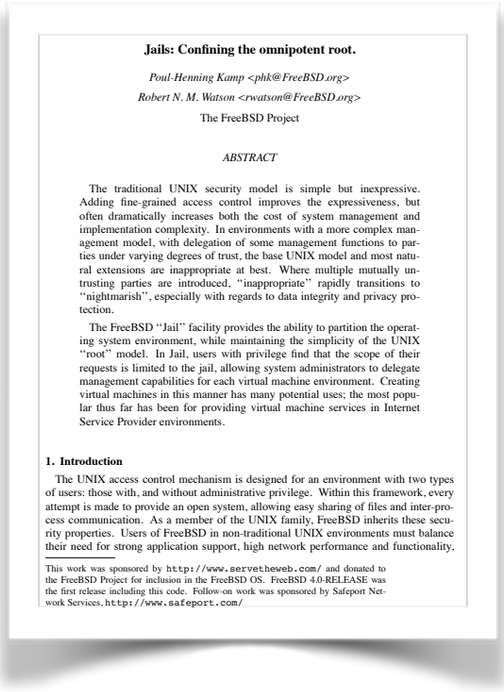


Source: A. Madhavapeddy *et al.*, "Unikernels: Library Operating Systems for the Cloud", Proc. ACM ASPLOS, Houston, TX, USA, March 2013. DOI:10.1145/2451116.2451167

CC BY ND

Further Reading for Tutorial

- P.-H. Kamp and R. Watson, “Jails: Confining the omnipotent root”, System Administration and Network Engineering Conference, May 2000. http://www.sane.nl/events/sane2000/papers/kamp.pdf
  - Trade-offs vs. complete system virtualisation?
  - Overheads vs. flexibility vs. ease of management?
  - Benefits of Docker-style configuration of images



Jails: Confining the omnipotent root.

Poul-Henning Kamp <phk@FreeBSD.org>
Robert N. M. Watson <rwatson@FreeBSD.org>
The FreeBSD Project

ABSTRACT

The traditional UNIX security model is simple but inexpressive. Adding fine-grained access control improves the expressiveness, but often dramatically increases both the cost of system management and implementation complexity. In environments with a more complex management model, with delegation of some management functions to parties under varying degrees of trust, the base UNIX model and most natural extensions are inappropriate at best. Where multiple mutually untrusting parties are introduced, ‘‘inappropriate’’ rapidly transitions to ‘‘nightmarish’’, especially with regards to data integrity and privacy protection.

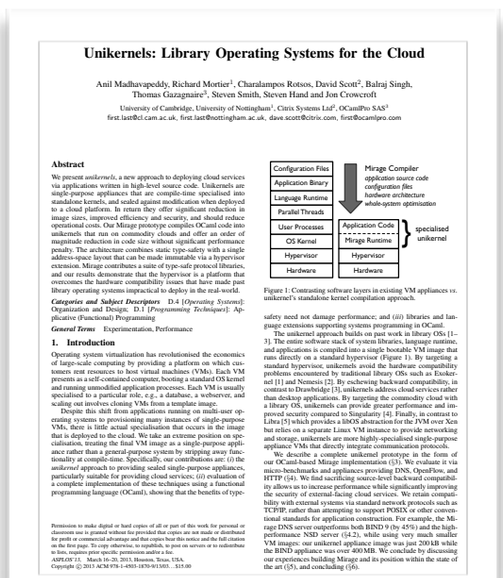
The FreeBSD ‘‘Jail’’ facility provides the ability to partition the operating system environment, while maintaining the simplicity of the UNIX ‘‘root’’ model. In Jail, users with privilege find that the scope of their requests is limited to the jail, allowing system administrators to delegate management capabilities for each virtual machine environment. Creating virtual machines in this manner has many potential uses; the most popular thus far has been for providing virtual machine services in Internet Service Provider environments.

1. Introduction

The UNIX access control mechanism is designed for an environment with two types of users: those with, and without administrative privilege. Within this framework, every attempt is made to provide an open system, allowing easy sharing of files and inter-process communication. As a member of the UNIX family, FreeBSD inherits these security properties. Users of FreeBSD in non-traditional UNIX environments must balance their need for strong application support, high network performance and functionality,

This work was sponsored by http://www.servetheweb.com/ and donated to the FreeBSD Project for inclusion in the FreeBSD OS. FreeBSD 4.0-RELEASE was the first release including this code. Follow-on work was sponsored by Safeport Network Services, http://www.safeport.com/

- A. Madhavapeddy et al., “Unikernels: Library Operating Systems for the Cloud”, ACM ASPLOS, Houston, TX, USA, March 2013. DOI:10.1145/2451116.2451167
  - Is optimising operating system for single application going too far?
  - Are unikernels maintainable?
  - Relation to containers and hypervisors?



Unikernels: Library Operating Systems for the Cloud

Anil Madhavapeddy, Richard Mortier[1], Charalampos Rotsos, David Scott[2], Balraj Singh, Thomas Gazagnaire[3], Steven Smith, Steven Hand and Jon Crowcroft
University of Cambridge, University of Nottingham[1], Citrix Systems Ltd[2], OCamlPro SAS[3]
first.last@cl.cam.ac.uk, first.last@nottingham.ac.uk, dave.scott@citrix.com, first@ocamlpro.com

Abstract

We present *unikernels*, a new approach to deploying cloud services via applications written in high-level source code. Unikernels are single-purpose appliances that are compile-time specialised into standalone kernels, and sealed against modification when deployed to a cloud platform. In return they offer significant reduction in image sizes, improved efficiency and security, and should reduce operational costs. Our Mirage prototype compiles OCaml code into unikernels that run on commodity clouds and offer an order of magnitude reduction in code size without significant performance penalty. The architecture combines static type-safety with a single address-space layout that can be made immutable via a hypervisor extension. Mirage contributes a suite of type-safe protocol libraries, and our results demonstrate that the hypervisor is a platform that overcomes the hardware compatibility issues that have made past library operating systems impractical to deploy in the real-world.

Categories and Subject Descriptors D.4 [*Operating Systems*]: Organization and Design. D.1 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Experimentation, Performance

1. Introduction

Operating system virtualization has revolutionised the economics of large-scale computing by providing a platform on which customers rent resources to host virtual machines (VMs). Each VM presents as a self-contained computer, booting a standard OS kernel and running unmodified application processes. Each VM is usually specialised to a particular role, e.g., a database, a webserver, and scaling out involves cloning VMs from a template image.

Despite this shift from applications running on multi-user operating systems to provisioning many instances of single-purpose VMs, there is little actual specialisation that occurs in the image that is deployed to the cloud. We take an extreme position on specialisation, treating the final VM image as a single-purpose appliance rather than a general-purpose system by stripping away functionality at compile-time. Specifically, our contributions are: (*i*) the *unikernel* approach to providing sealed single-purpose appliances, particularly suitable for providing cloud services; (*ii*) evaluation of a complete implementation of these techniques using a functional programming language (OCaml), showing that the benefits of type-

Figure 1: Contrasting software layers in existing VM appliances *vs.* unikernel's standalone kernel compilation approach.

safety need not damage performance; and (*iii*) libraries and language extensions supporting systems programming in OCaml.

The unikernel approach builds on past work in library OSs [1–3]. The entire software stack of system libraries, language runtime, and applications is compiled into a single bootable VM image that runs directly on a standard hypervisor (Figure 1). By targeting a standard hypervisor, unikernels avoid the hardware compatibility problems encountered by traditional library OSs such as Exokernel [1] and Nemesis [2]. By eschewing backward compatibility, in contrast to Drawbridge [3], unikernels address cloud services rather than desktop applications. By targeting the commodity cloud with a library OS, unikernels can provide greater performance and improved security compared to Singularity [4]. Finally, in contrast to Libra [5] which provides a libOS abstraction for the JVM over Xen but relies on a separate Linux VM instance to provide networking and storage, unikernels are more highly-specialised single-purpose appliance VMs that directly integrate communication protocols.

We describe a complete unikernel prototype in the form of our OCaml-based Mirage implementation (§3). We evaluate it via micro-benchmarks and appliances providing DNS, OpenFlow, and HTTP (§4). We find sacrificing source-level backward compatibility allows us to increase performance while significantly improving the security of external-facing cloud services. We retain compatibility with external systems via standard network protocols such as TCP/IP, rather than attempting to support POSIX or other conventional standards for application construction. For example, the Mirage DNS server outperforms both BIND 9 (by 45%) and the high-performance NSD server (§4.2), while using very much smaller VM images: our unikernel appliance image was just 200 kB while the BIND appliance was over 400 MB. We conclude by discussing our experiences building Mirage and its position within the state of the art (§5), and concluding (§6).

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.
Copyright © 2013 ACM 978-1-4503-1870-9/13/03…$15.00

CC BY ND