

VIRTUAL MACHINES

COMP9242
2006/S2 Week 5

VIRTUAL MACHINE

A virtual machine (VM) is an efficient, isolated duplicate of a real machine (PG74).

VIRTUAL MACHINE

A virtual machine (VM) is an efficient, isolated duplicate of a real machine (PG74).

Duplicate: VM should behave identical to the real machine

- programs cannot distinguish between execution on real or virtual hardware
- except for:
 - less resources available (and potentially different between executions)
 - some timing differences (when dealing with devices)

Isolated: several VMs execute without interfering with each other

Efficient: VM should execute at a speed close to that of real hardware

- requires that most instructions are executed directly by real hardware

VIRTUAL MACHINES, SIMULATORS AND EMULATORS

Simulator provides a *functionally accurate* software model of a machine

Emulator provides a *behavioural* model of a machine (and possibly S/W)

Virtual machine models a machine exactly and efficiently

VIRTUAL MACHINES, SIMULATORS AND EMULATORS

Simulator provides a *functionally accurate* software model of a machine

- ✓ may run on any hardware
- ✗ is typically slow (order of 1000 slowdown)

Emulator provides a *behavioural* model of a machine (and possibly S/W)

Virtual machine models a machine exactly and efficiently

VIRTUAL MACHINES, SIMULATORS AND EMULATORS

Simulator provides a *functionally accurate* software model of a machine

- ✓ may run on any hardware
- ✗ is typically slow (order of 1000 slowdown)

Emulator provides a *behavioural* model of a machine (and possibly S/W)

- ✗ not fully accurate
- ✓ reasonably fast (order of 10 slowdown)

Virtual machine models a machine exactly and efficiently

VIRTUAL MACHINES, SIMULATORS AND EMULATORS

Simulator provides a *functionally accurate* software model of a machine

- ✓ may run on any hardware
- ✗ is typically slow (order of 1000 slowdown)

Emulator provides a *behavioural* model of a machine (and possibly S/W)

- ✗ not fully accurate
- ✓ reasonably fast (order of 10 slowdown)

Virtual machine models a machine exactly and efficiently

- ✓ minimal slowdown
- ✗ needs to be run on the physical machine it virtualises

TYPES OF VIRTUAL MACHINES

- Contemporary use of the term VM is more general

TYPES OF VIRTUAL MACHINES

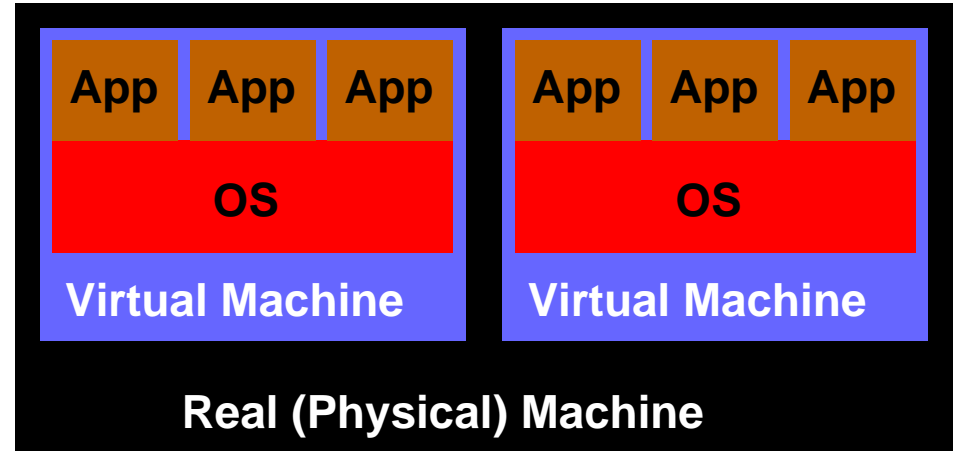
- Contemporary use of the term VM is more general
- Call virtual machines even if there is no correspondence to an existing real machine
 - e.g. *Java virtual machine*
 - can be viewed as virtualising at the ABI level
 - also called *process VM*

TYPES OF VIRTUAL MACHINES

- Contemporary use of the term VM is more general
- Call virtual machines even if there is no correspondence to an existing real machine
 - e.g. *Java virtual machine*
 - can be viewed as virtualising at the ABI level
 - also called *process VM*
- We only concern ourselves with virtualising at the ISA level
 - ISA = *instruction-set architecture* (hardware-software interface)
 - also called *system VM* (SN05)
 - will later see subclasses of this

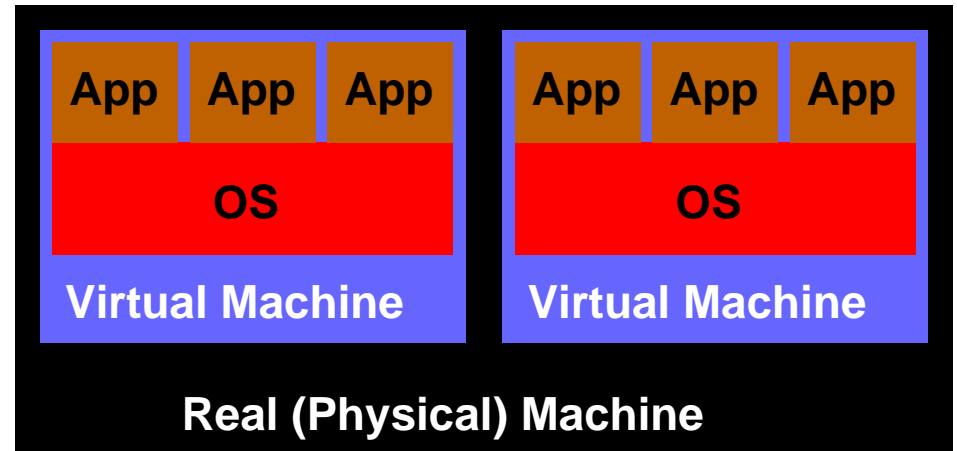
WHY VIRTUAL MACHINES?

- Historically used for easier sharing of expensive mainframes
 - run several (different) OSes on same machine
 - each on a subset of physical resources
 - “world switch” between VMs
- Gone out of fashion in 80’s



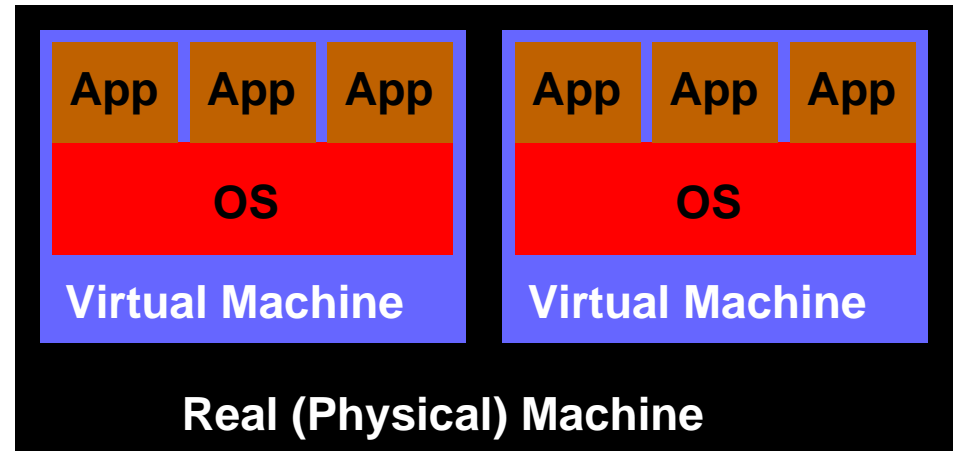
WHY VIRTUAL MACHINES?

- Historically used for easier sharing of expensive mainframes
 - run several (different) OSes on same machine
 - each on a subset of physical resources
 - “world switch” between VMs
- Gone out of fashion in 80’s
- Renaissance in recent years for improved isolation (RG05)
 - improved QoS and security
 - uniform view of hardware
 - complete encapsulation (replication, migration, checkpointing, debugging)
 - total mediation



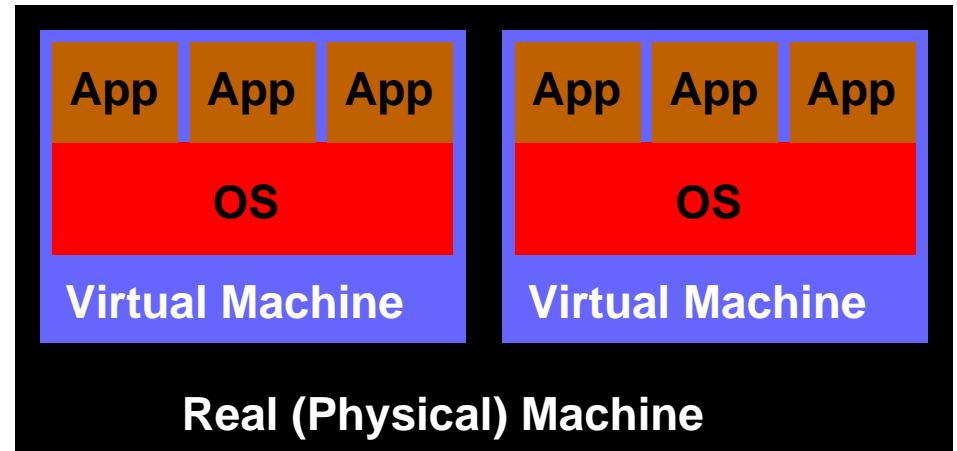
WHY VIRTUAL MACHINES?

- Historically used for easier sharing of expensive mainframes
 - run several (different) OSes on same machine
 - each on a subset of physical resources
 - “world switch” between VMs
- Gone out of fashion in 80’s
- Renaissance in recent years for improved isolation (RG05)
 - improved QoS and security
 - uniform view of hardware
 - complete encapsulation (replication, migration, checkpointing, debugging)
 - total mediation
- Isn’t that exactly the job description of an OS????



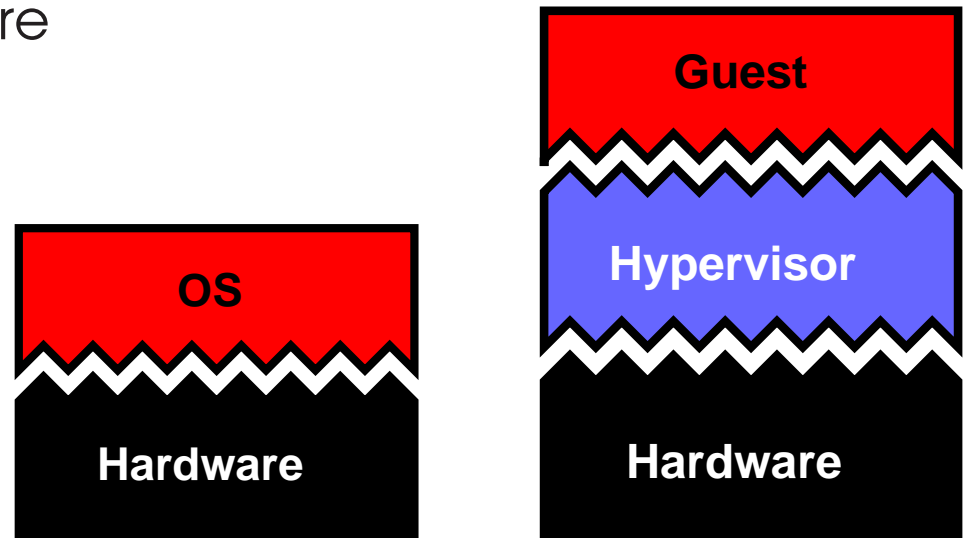
WHY VIRTUAL MACHINES?

- Historically used for easier sharing of expensive mainframes
 - run several (different) OSes on same machine
 - each on a subset of physical resources
 - “world switch” between VMs
- Gone out of fashion in 80’s
- Renaissance in recent years for improved isolation (RG05)
 - improved QoS and security
 - uniform view of hardware
 - complete encapsulation (replication, migration, checkpointing, debugging)
 - total mediation
- Isn’t that exactly the job description of an OS????
 - This really means that mainstream OSes suck!



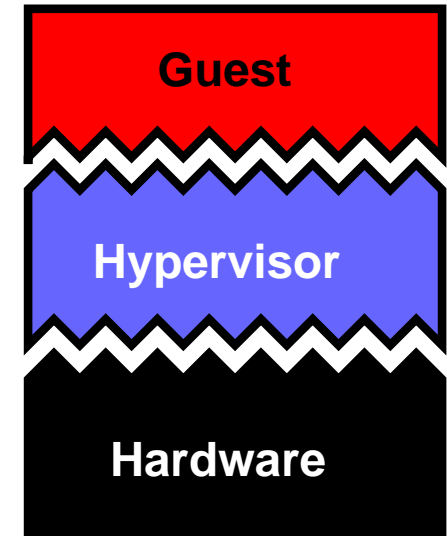
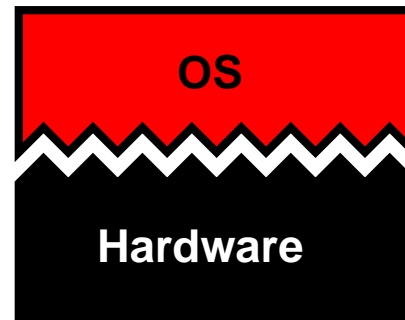
VIRTUAL MACHINE MONITOR (VMM), AKA HYPERVISOR

- Program that runs on real hardware to implement the virtual machine



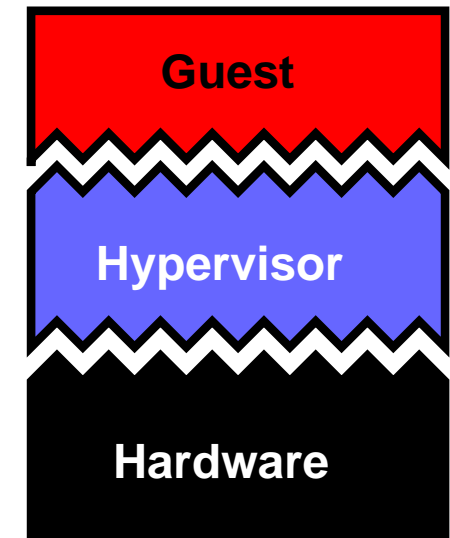
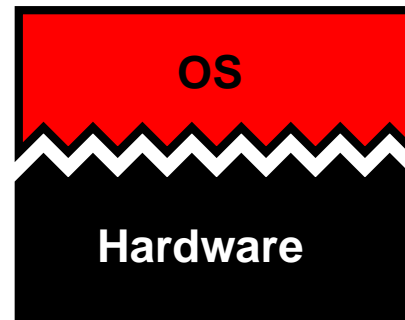
VIRTUAL MACHINE MONITOR (VMM), AKA HYPERVISOR

- Program that runs on real hardware to implement the virtual machine
- Controls resources
 - partitions hardware
 - schedules guests
 - mediates access to shared resources (eg. console)
 - performs world switch



VIRTUAL MACHINE MONITOR (VMM), AKA HYPERVISOR

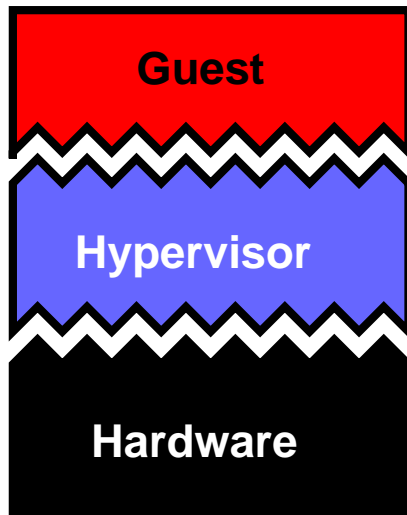
- Program that runs on real hardware to implement the virtual machine
- Controls resources
 - partitions hardware
 - schedules guests
 - mediates access to shared resources (eg. console)
 - performs world switch



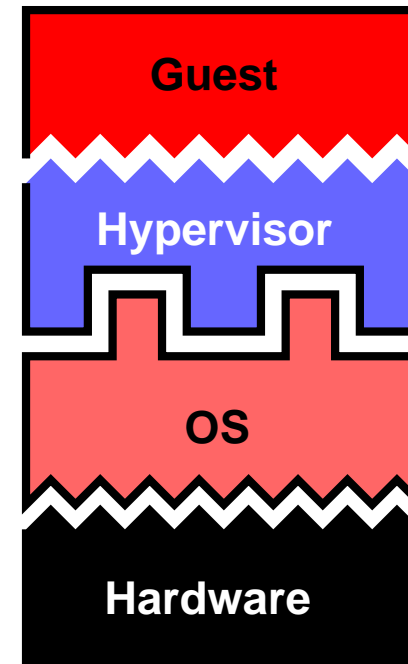
- Implications:
 - hypervisor executes in privileged mode
 - guest software executes in unprivileged mode
 - *privileged instructions* in guest cause a trap into the hypervisor
 - hypervisor interprets/emulates them
 - can have extra instructions for *hypercalls*

NATIVE VS. HOSTED VMM

Native/Classic/Bare-metal/Type-1

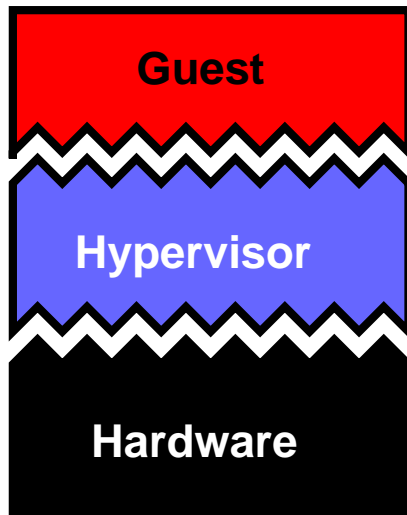


Hosted/Type-2

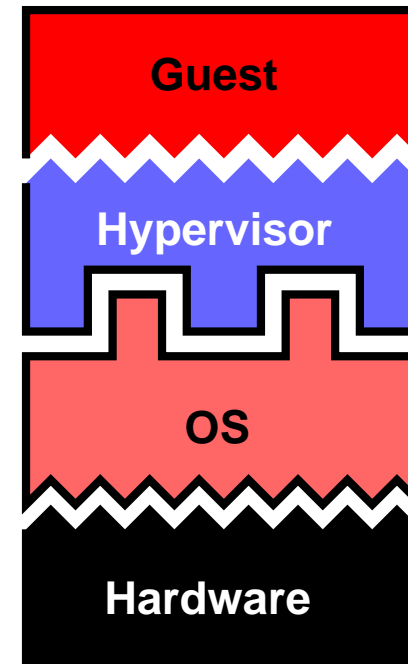


NATIVE VS. HOSTED VMM

Native/Classic/Bare-metal/Type-1



Hosted/Type-2



- Hosted VMM can run besides native apps
 - sandbox untrusted apps
 - run second OS
 - less efficient:
 - ⊗ guest privileged instruction traps into OS, forwarded to hypervisor
 - ⊗ return to guest requires a native OS system call

VMM TYPES

Classic as above

Hosted e.g. VMware GSX Server

Whole-system virtual hardware and operating system

→ really an emulation

→ e.g. Virtual PC (for Macintosh)

Physically partitioned allocate actual processors to each VM

Logically partitioned time-share processors between VMs

Co-designed hardware specifically designed for VMM

→ e.g. Transmeta Crusoe, IBM i-Series

REQUIREMENTS FOR VIRTUALISATION

Definitions:

Privileged instruction executes in privileged mode, traps in user mode

→ Note: trap is required, NOOP is insufficient!

Privileged state determines resource allocation

→ includes privilege mode, addressing context, exception vectors, ...

REQUIREMENTS FOR VIRTUALISATION

Definitions:

Privileged instruction executes in privileged mode, traps in user mode

→ Note: trap is required, NOOP is insufficient!

Privileged state determines resource allocation

→ includes privilege mode, addressing context, exception vectors, ...

Sensitive instruction control-sensitive or behaviour-sensitive

control sensitive *changes* privileged state

behaviour sensitive *exposes* privileged state

→ includes instructions which are NOOPs in user but not privileged mode

REQUIREMENTS FOR VIRTUALISATION

Definitions:

Privileged instruction executes in privileged mode, traps in user mode

→ Note: trap is required, NOOP is insufficient!

Privileged state determines resource allocation

→ includes privilege mode, addressing context, exception vectors, ...

Sensitive instruction control-sensitive or behaviour-sensitive

control sensitive *changes* privileged state

behaviour sensitive *exposes* privileged state

→ includes instructions which are NOOPs in user but not privileged mode

Innocuous instruction not sensitive

REQUIREMENTS FOR VIRTUALISATION

An architecture is *virtualizable* if all sensitive instructions are privileged (called *pure virtualisation*)

REQUIREMENTS FOR VIRTUALISATION

An architecture is *virtualizable* if all sensitive instructions are privileged (called *pure virtualisation*)

- Can then achieve accurate, efficient guest execution
 - guest's sensitive instructions trap and are emulated by VMM
 - guest's innocuous instructions are executed directly
 - VMM controls resources
- Virtualised execution is indistinguishable from native, except:
 - resources more limited (running on *smaller* machine)
 - timing is different (if there is an observable time source)

REQUIREMENTS FOR VIRTUALISATION

An architecture is *virtualizable* if all sensitive instructions are privileged (called *pure virtualisation*)

- Can then achieve accurate, efficient guest execution
 - guest's sensitive instructions trap and are emulated by VMM
 - guest's innocuous instructions are executed directly
 - VMM controls resources
- Virtualised execution is indistinguishable from native, except:
 - resources more limited (running on *smaller* machine)
 - timing is different (if there is an observable time source)
- Architecture is *recursively virtualizable* if VMM without timing dependency can be built

VIRTUALISATION OVERHEADS

- VMM needs to maintain virtualised privileged machine state
 - processor status
 - addressing context
- VMM needs to simulate privileged instructions
 - synchronise virtual and real privileged state as appropriate
 - e.g. *shadow page tables* to virtualize hardware

VIRTUALISATION OVERHEADS

- VMM needs to maintain virtualised privileged machine state
 - processor status
 - addressing context
- VMM needs to simulate privileged instructions
 - synchronise virtual and real privileged state as appropriate
 - e.g. *shadow page tables* to virtualize hardware
- Frequent virtualisation traps can be expensive
 - STI/CLI for mutual exclusion
 - frequent page table updates
 - MIPS KSEG addresses used for physical addressing in kernel

UNVIRTUALISABLE ARCHITECTURES

- x86: lots of unvirtualisable features
 - e.g. sensitive PUSH of PSW is not privileged
 - segment and interrupt descriptor tables in virtual memory
 - segment descriptors expose privileged level

UNVIRTUALISABLE ARCHITECTURES

- x86: lots of unvirtualisable features
 - e.g. sensitive PUSH of PSW is not privileged
 - segment and interrupt descriptor tables in virtual memory
 - segment descriptors expose privileged level
- Itanium: mostly virtualizable, but
 - interrupt vector table in virtual memory
 - THASH instruction exposes hardware page table address

UNVIRTUALISABLE ARCHITECTURES

- x86: lots of unvirtualisable features
 - e.g. sensitive PUSH of PSW is not privileged
 - segment and interrupt descriptor tables in virtual memory
 - segment descriptors expose privileged level
- Itanium: mostly virtualizable, but
 - interrupt vector table in virtual memory
 - THASH instruction exposes hardware page table address
- MIPS: mostly virtualizable, but
 - kernel registers $k0$, $k1$ (needed to save/restore state) user-accessible
 - performance issue with virtualising $KSEG$ addresses

UNVIRTUALISABLE ARCHITECTURES

- x86: lots of unvirtualisable features
 - e.g. sensitive PUSH of PSW is not privileged
 - segment and interrupt descriptor tables in virtual memory
 - segment descriptors expose privileged level
- Itanium: mostly virtualizable, but
 - interrupt vector table in virtual memory
 - THASH instruction exposes hardware page table address
- MIPS: mostly virtualizable, but
 - kernel registers $k0$, $k1$ (needed to save/restore state) user-accessible
 - performance issue with virtualising $KSEG$ addresses
- ARM: mostly virtualizable, but
 - some instructions undefined in user mode (banked regs, CPSR)
 - PC is a GPR, exception return is MOVS to PC, doesn't trap

UNVIRTUALISABLE ARCHITECTURES

- x86: lots of unvirtualisable features
 - e.g. sensitive PUSH of PSW is not privileged
 - segment and interrupt descriptor tables in virtual memory
 - segment descriptors expose privileged level
- Itanium: mostly virtualizable, but
 - interrupt vector table in virtual memory
 - THASH instruction exposes hardware page table address
- MIPS: mostly virtualizable, but
 - kernel registers $k0$, $k1$ (needed to save/restore state) user-accessible
 - performance issue with virtualising $KSEG$ addresses
- ARM: mostly virtualizable, but
 - some instructions undefined in user mode (banked regs, CPSR)
 - PC is a GPR, exception return is MOVS to PC, doesn't trap
- Most others have problems too

IMPURE VIRTUALISATION

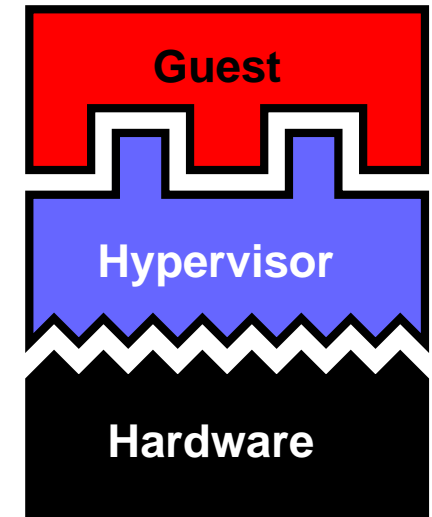
- Used for two reasons:
 - unvirtualisable architectures
 - performance problems of virtualisation

IMPURE VIRTUALISATION

- Used for two reasons:
 - unvirtualisable architectures
 - performance problems of virtualisation
- Two standard approaches:
 - ① para-virtualisation
 - ② binary translation

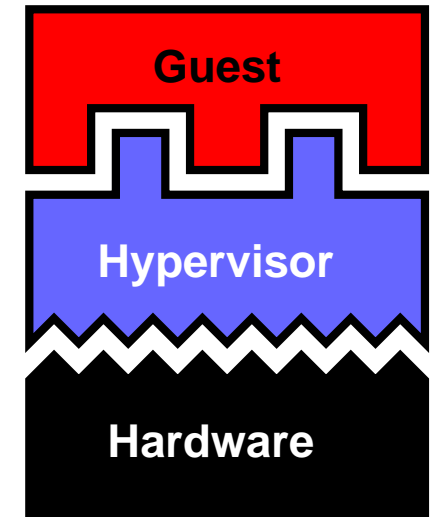
PARAVIRTUALISATION

- New name, old technique
 - Used in Mach Unix server, L⁴Linux (HHL⁺97), Disco (BDGR97)
 - Name coined by Denali (WSG02), popularised by Xen (BDF⁺03)



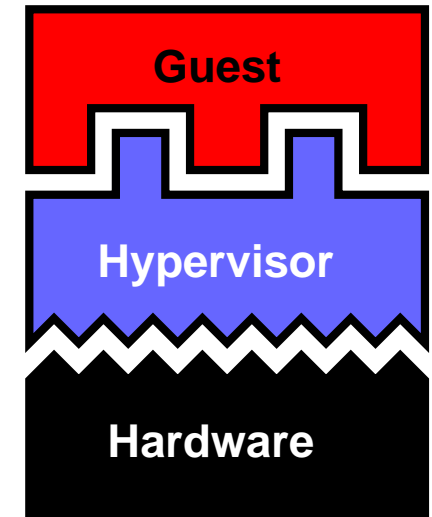
PARAVIRTUALISATION

- New name, old technique
 - Used in Mach Unix server, L⁴Linux (HHL⁺97), Disco (BDGR97)
 - Name coined by Denali (WSG02), popularised by Xen (BDF⁺03)
- Manually port the guest OS to modified ISA
 - augmented by explicit hypervisor calls (*hypercalls*)
 - ✓ idea is to provide more high-level API to reduce the number of traps
 - ✓ remove unvirtualisable instructions
 - ✓ remove “messy” ISA features which complicate virtualisation



PARAVIRTUALISATION

- New name, old technique
 - Used in Mach Unix server, L⁴Linux (HHL⁺97), Disco (BDGR97)
 - Name coined by Denali (WSG02), popularised by Xen (BDF⁺03)
- Manually port the guest OS to modified ISA
 - augmented by explicit hypervisor calls (*hypercalls*)
 - ✓ idea is to provide more high-level API to reduce the number of traps
 - ✓ remove unvirtualisable instructions
 - ✓ remove “messy” ISA features which complicate virtualisation
- Drawbacks:
 - ✗ significant engineering effort
 - ✗ needs to be repeated for each guest, ISA, hypervisor combination
 - ✗ paravirtualised guest needs to be kept in sync with native guest
 - ✗ requires source



BINARY TRANSLATION

- Locate unvirtualisable instructions in guest binary and replace on-the-fly by emulation code or hypercall
→ pioneered by VMware (RG05)

BINARY TRANSLATION

- Locate unvirtualisable instructions in guest binary and replace on-the-fly by emulation code or hypercall
 - pioneered by VMware (RG05)
 - ✓ can also detect combinations of sensitive instructions and replace by single emulation
 - ✓ doesn't require source
 - ✓ may (safely) do some emulation in user space for efficiency
 - ✗ very tricky to get right (especially on x86!)
 - ✗ needs to make some assumptions on sane behaviour of guest

VIRTUALISATION TECHNIQUES: MEMORY

- Shadow page tables
 - Guest accesses shadow PT
 - VMM detects changes (e.g. making them R/O) and syncs with real PT
 - can over-commit memory (similar to virtual-memory paging)

VIRTUALISATION TECHNIQUES: MEMORY

- Shadow page tables
 - Guest accesses shadow PT
 - VMM detects changes (e.g. making them R/O) and syncs with real PT
 - can over-commit memory (similar to virtual-memory paging)
 - Note: Xen exposes hardware page tables

VIRTUALISATION TECHNIQUES: MEMORY

- Shadow page tables
 - Guest accesses shadow PT
 - VMM detects changes (e.g. making them R/O) and syncs with real PT
 - can over-commit memory (similar to virtual-memory paging)
 - Note: Xen exposes hardware page tables
- Memory reclamation: *Ballooning* (VMware ESX Server)
 - load cooperating pseudo-device driver into guest
 - to reclaim, balloon driver requests physical memory from guest
 - VMM can then reuse that memory
 - guest determines which pages to release

VIRTUALISATION TECHNIQUES: MEMORY

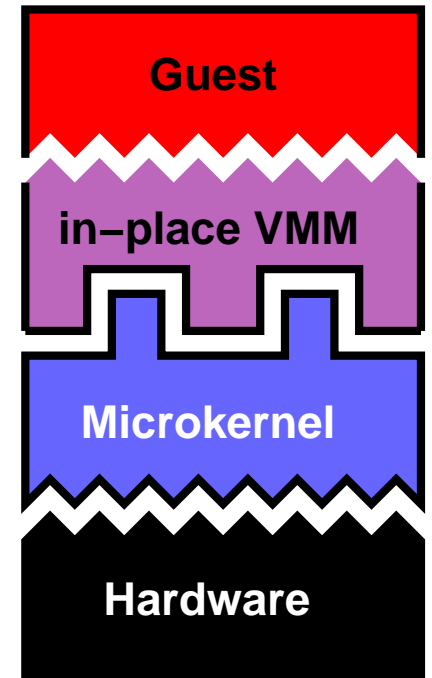
- Shadow page tables
 - Guest accesses shadow PT
 - VMM detects changes (e.g. making them R/O) and syncs with real PT
 - can over-commit memory (similar to virtual-memory paging)
 - Note: Xen exposes hardware page tables
- Memory reclamation: *Ballooning* (VMware ESX Server)
 - load cooperating pseudo-device driver into guest
 - to reclaim, balloon driver requests physical memory from guest
 - VMM can then reuse that memory
 - guest determines which pages to release
- Page sharing
 - VMM detects pages with identical content
 - establishes (copy-on-write) mappings to single page via shadow PT
 - significant savings when running many identical guest OSes

VIRTUALISATION TECHNIQUES: DEVICES

- Drivers in VMM
 - maybe ported legacy drivers
- Host drivers
 - for hosted VMMs
- Legacy driver in separate driver VM
 - e.g. separate Linux guest for each device (LUSG04)
- Virtualisation-friendly devices
 - IBM channel architecture
 - can safely be accessed by guest drivers if physical memory access is restricted (I/O-MMU)

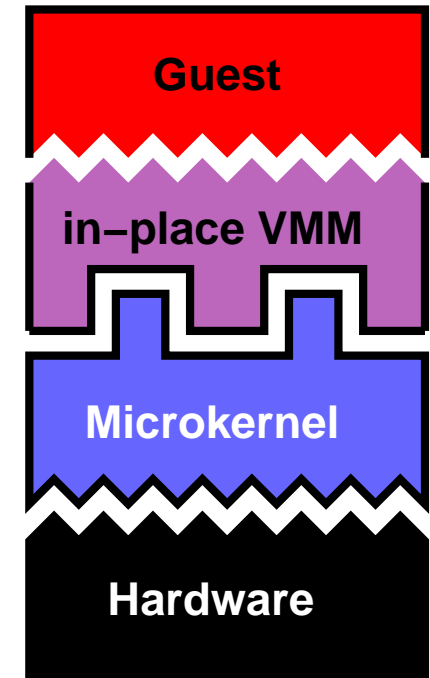
PREVIRTUALISATION

- Combines advantages of pure and para-virtualisation



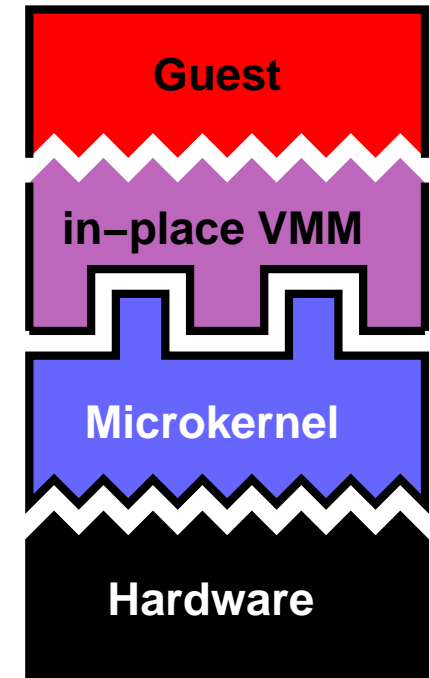
PREVIRTUALISATION

- Combines advantages of pure and para-virtualisation
- Multi-stage process
 - ① During build, pad sensitive instructions with NOPs and keep record
 - ② During profiling run, trap sensitive memory operations (e.g. PT accesses) and record
 - ③ Redo build, also padding sensitive memory operations
 - ④ Link emulation lib (*in-place VMM* or "*wedge*") to guest
 - ⑤ At load time, replace NOP-padded instructions by emulation code



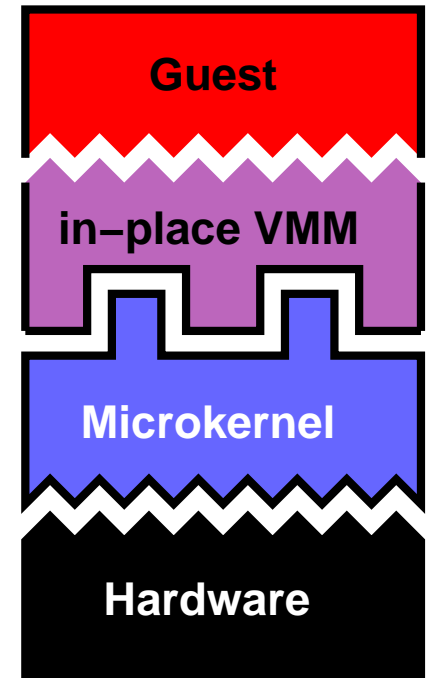
PREVIRTUALISATION

- Combines advantages of pure and para-virtualisation
- Multi-stage process
 - ① During build, pad sensitive instructions with NOPs and keep record
 - ② During profiling run, trap sensitive memory operations (e.g. PT accesses) and record
 - ③ Redo build, also padding sensitive memory operations
 - ④ Link emulation lib (*in-place VMM* or "*wedge*") to guest
 - ⑤ At load time, replace NOP-padded instructions by emulation code
- Features:
 - ✓ significantly reduced engineering effort
 - ✓ single binary runs on bare metal as well as *all* hypervisors
 - ✗ requires source (as does normal para-virtualisation)
 - ✗ performance may require some para-virtualisation

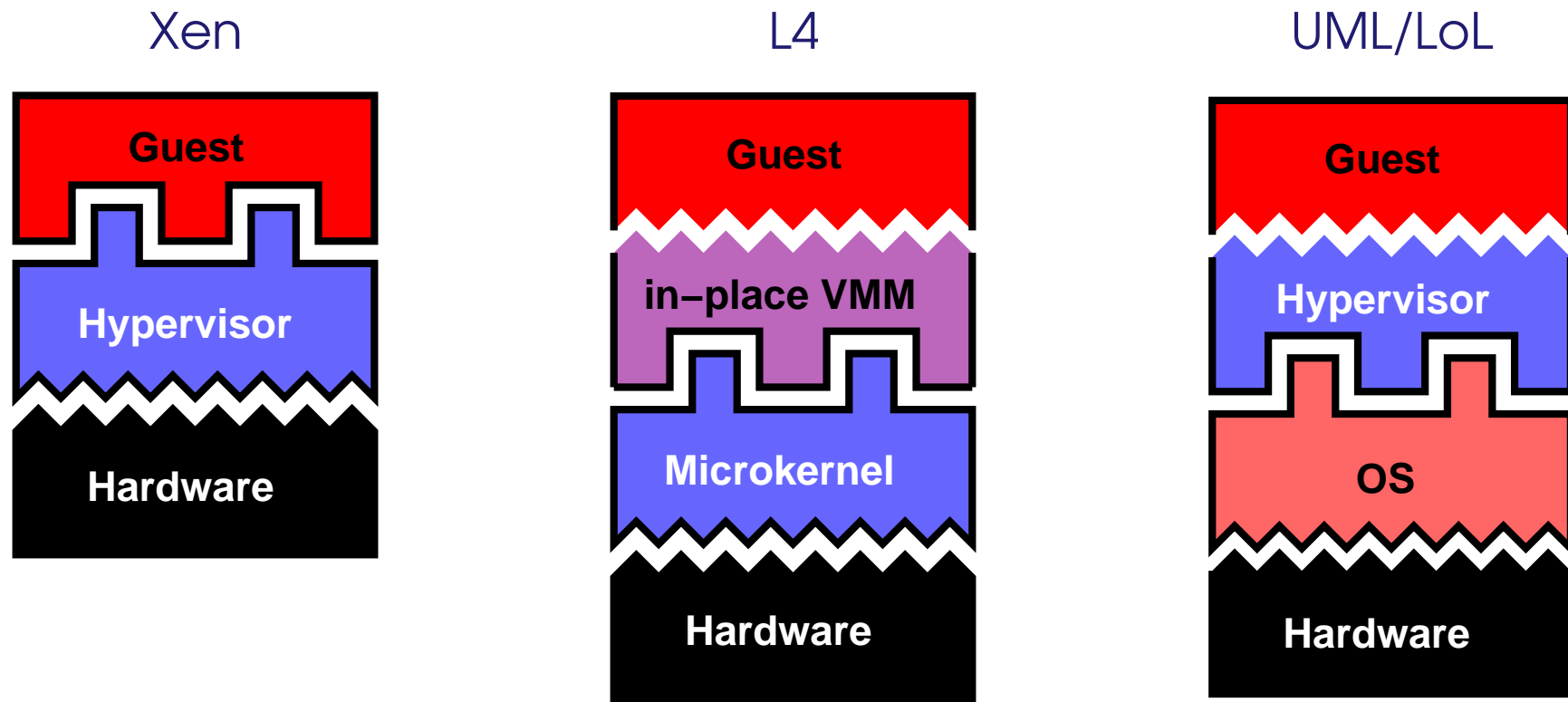


PREVIRTUALISATION

- Combines advantages of pure and para-virtualisation
- Multi-stage process
 - ① During build, pad sensitive instructions with NOPs and keep record
 - ② During profiling run, trap sensitive memory operations (e.g. PT accesses) and record
 - ③ Redo build, also padding sensitive memory operations
 - ④ Link emulation lib (*in-place VMM* or "*wedge*") to guest
 - ⑤ At load time, replace NOP-padded instructions by emulation code
- Features:
 - ✓ significantly reduced engineering effort
 - ✓ single binary runs on bare metal as well as *all* hypervisors
 - ✗ requires source (as does normal para-virtualisation)
 - ✗ performance may require some para-virtualisation
- See <http://14ka.org/projects/virtualization/afterburn/> (LUC⁺05)

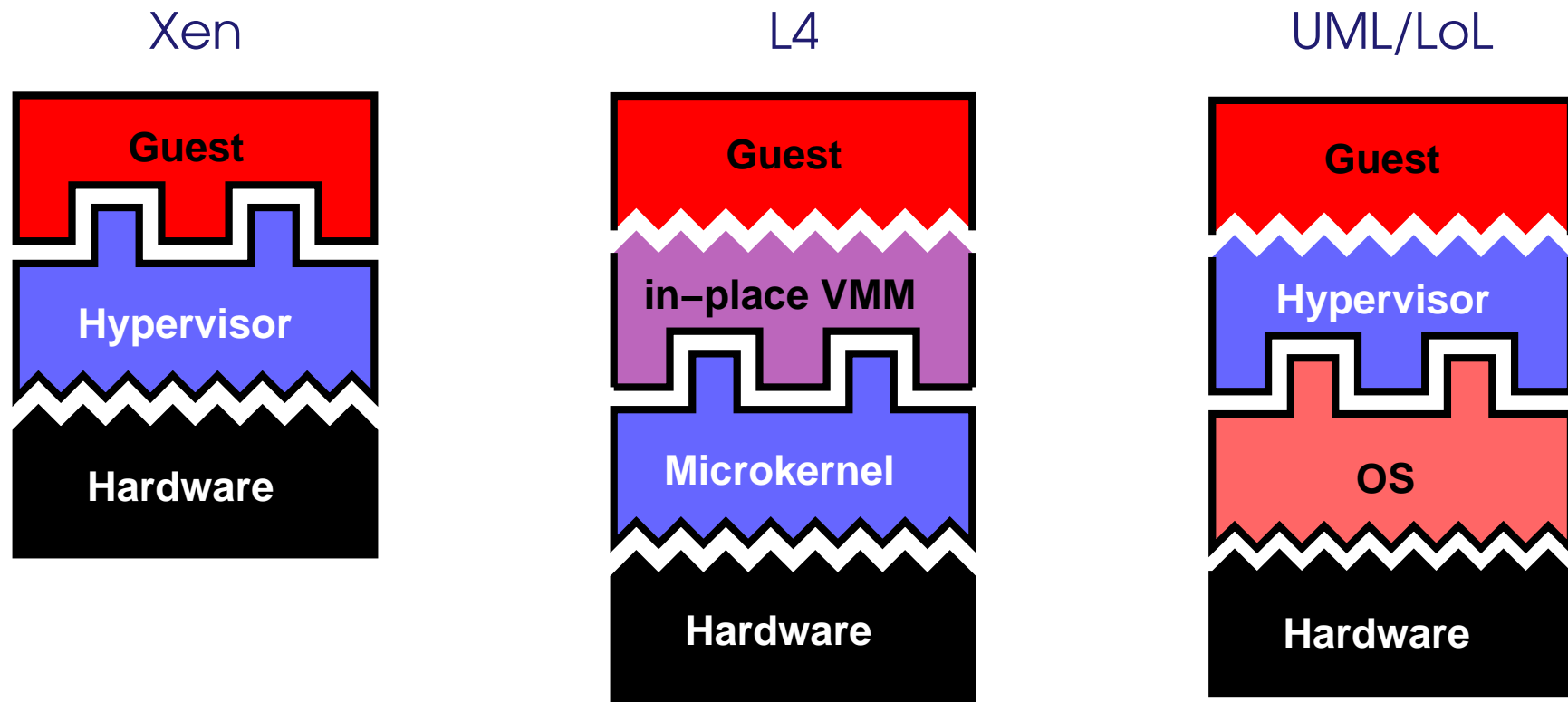


MICROKERNEL AS A HYPERVISOR



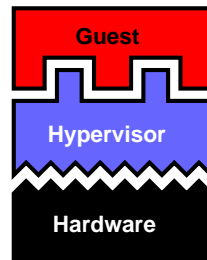
- Microkernel as hypervisor half-way between native and hosted VMM?

MICROKERNEL AS A HYPERVISOR

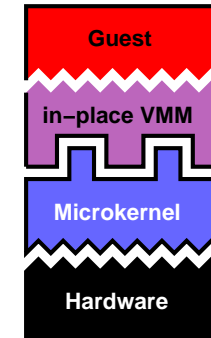


- Microkernel as hypervisor half-way between native and hosted VMM?
 - however, para-virtualisation may also benefit from in-place emulation
 - e.g. save mode switches by virtualising PSR inside guest address space

HYPERVISORS VS. MICROKERNELS

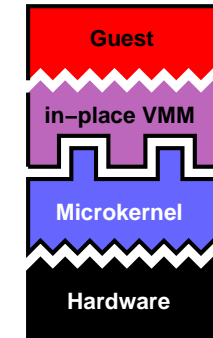
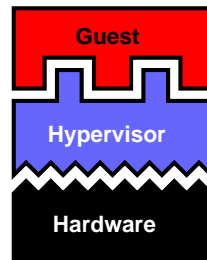


- no other code in kernel mode
- specialised, legacy guest OS only
- VMM completely in kernel(?)
- variety of mechanisms
- larger (Xen based on Linux)
- guests communicate via virtual NW
- strong subsystem partitioning



- no other code in kernel mode
- generic, guest OS & native apps
- VMM partially in guest AS
- minimal mechanisms
- small (L4 \approx 10kloc)
- guests communicate via IPC
- continuum:
partitioned — integrated

HYPERVERSORS VS. MICROKERNELS



- no other code in kernel mode
- specialised, legacy guest OS only
- VMM completely in kernel(?)
- variety of mechanisms
- larger (Xen based on Linux)
- guests communicate via virtual NW
- strong subsystem partitioning

- no other code in kernel mode
- generic, guest OS & native apps
- VMM partially in guest AS
- minimal mechanisms
- small (L4 \approx 10kloc)
- guests communicate via IPC
- continuum:
partitioned — integrated

- Microkernel can be seen as a generalisation of a hypervisor
 - Do we pay with performance?
 - See also (HWF⁺05, HUL06)

HYPERVERSORS VS. MICROKERNELS: PERFORMANCE

- Xen vs. L4 on Pentium 4 running Linux 2.6.9
- Device drivers in guest OS

System	Kernel Compile			Netperf send			Netperf receive		
	Time (s)	CPU (%)	O/H (%)	Xput (Mb/s)	CPU (%)	Cost (cyc/B)	Xput (Mb/s)	CPU (%)	Cost (cyc/B)
native	209	98.4	0	867.5	27	6.7	780.4	34	9.2
Linux on Xen	219	97.8	5	867.6	34	8.3	780.7	41	11.3
Linux on L4	236	97.9	13	866.5	30	7.5	780.1	36	9.8

HYPERVERSORS VS. MICROKERNELS: PERFORMANCE

- Xen vs. L4 on Pentium 4 running Linux 2.6.9
- Device drivers in guest OS

System	Kernel Compile			Netperf send			Netperf receive		
	Time (s)	CPU (%)	O/H (%)	Xput (Mb/s)	CPU (%)	Cost (cyc/B)	Xput (Mb/s)	CPU (%)	Cost (cyc/B)
native	209	98.4	0	867.5	27	6.7	780.4	34	9.2
Linux on Xen	219	97.8	5	867.6	34	8.3	780.7	41	11.3
Linux on L4	236	97.9	13	866.5	30	7.5	780.1	36	9.8

- Xen base performance is better
→ ... but much more intrusive changes to Linux
- Network performance shows that there is optimisation potential

HARDWARE SUPPORT

- Intel VT-x/VT-i: virtualisation support for x86/Itanium (UNR+05)
 - introduces new processor mode: *root mode* for hypervisor
 - if enabled, all sensitive instructions in non-root mode trap to root mode
 - VT-i (Itanium) also reduces virtual address-space size for non-root

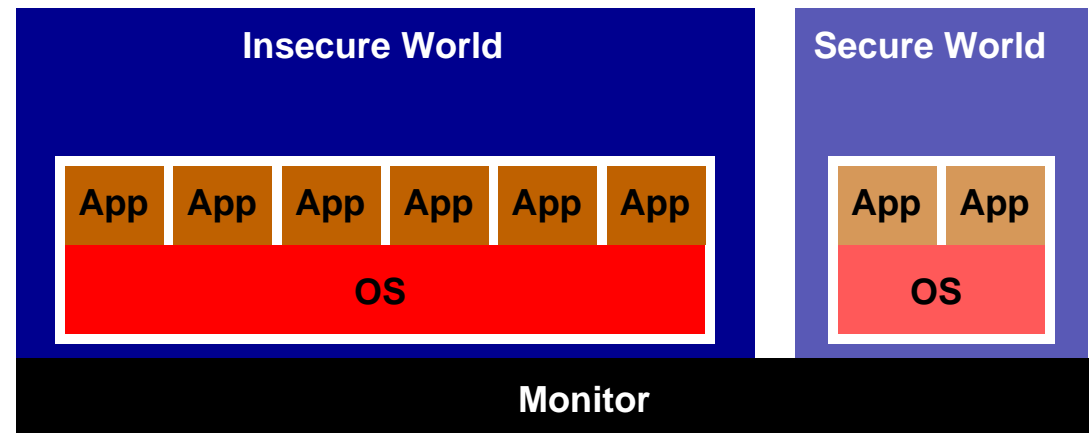
HARDWARE SUPPORT

- Intel VT-x/VT-i: virtualisation support for x86/Itanium (UNR+05)
 - introduces new processor mode: *root mode* for hypervisor
 - if enabled, all sensitive instructions in non-root mode trap to root mode
 - VT-i (Itanium) also reduces virtual address-space size for non-root
- Similar AMD (Pacifica), PowerPC, AMR (TrustZone)
- Aim is virtualisation of unmodified legacy OSes

CASE STUDY: TRUSTZONE — ARM VIRTUALISATION EXTENSIONS

ARM virtualisation extensions introduce:

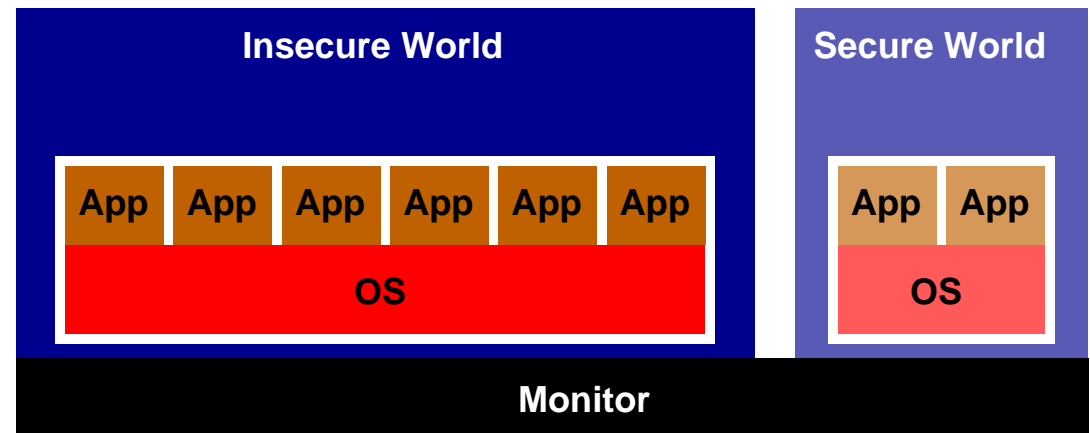
- new processor mode: *monitor*
 - banked registers (PC, LR)
 - guest runs in kernel mode
 - unvirtual. insts no problem
- new privileged instruction: *SMI*
 - enters monitor mode
- new processor status: *secure*
- partitioning of resources
 - memory and devices marked *secure* or *insecure*



CASE STUDY: TRUSTZONE — ARM VIRTUALISATION EXTENSIONS

ARM virtualisation extensions introduce:

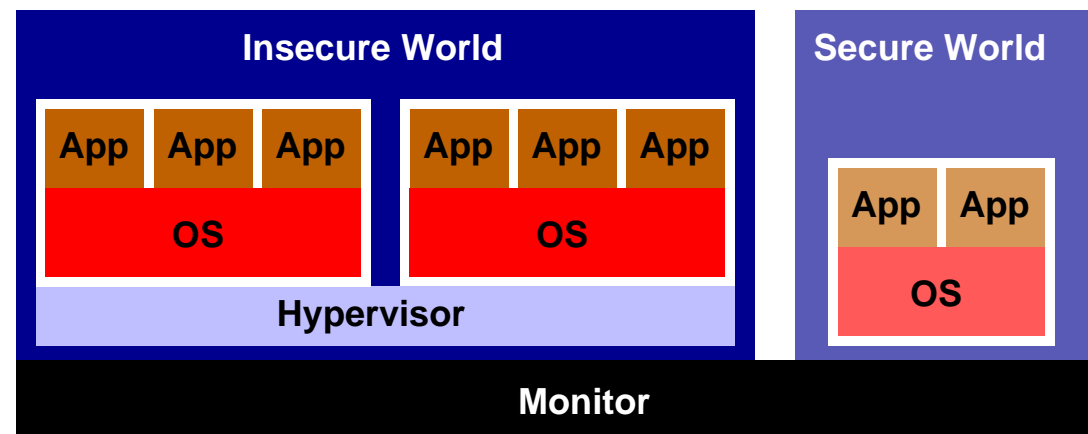
- new processor mode: *monitor*
 - banked registers (PC, LR)
 - guest runs in kernel mode
 - unvirtual. insts no problem
- new privileged instruction: *SMI*
 - enters monitor mode
- new processor status: *secure*
- partitioning of resources
 - memory and devices marked *secure* or *insecure*
- In *secure* mode, processor has access to all resources
- in *insecure* mode, processor has access to *insecure* resources only
- *Monitor* switches worlds (secure – insecure)



CASE STUDY: TRUSTZONE — ARM VIRTUALISATION EXTENSIONS

ARM virtualisation extensions introduce:

- new processor mode: *monitor*
 - banked registers (PC, LR)
 - guest runs in kernel mode
 - unvirtual. insts no problem
- new privileged instruction: *SMI*
 - enters monitor mode
- new processor status: *secure*
- partitioning of resources
 - memory and devices marked *secure* or *insecure*
- In *secure* mode, processor has access to all resources
- in *insecure* mode, processor has access to *insecure* resources only
- *Monitor* switches worlds (secure – insecure)
- Optional hypervisor switches insecure (para-virtualised) guests



REFERENCES

- (BDF⁺03) Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *19th SOSP*, pages 164–177, Bolton Landing, NY, USA, Oct 2003.
- (BDGR97) Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *Trans. Comp. Syst.*, 15:412–447, 1997.
- (HHL⁺97) Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ -kernel-based systems. In *16th SOSP*, pages 66–77, St. Malo, France, Oct 1997.
- (HUL06) Gernot Heiser, Volkmar Uhlig, and Joshua LeVasseur. Are virtual-machine monitors microkernels done right? *Operat. Syst. Rev.*, 40(1):95–99, Jan 2006.

REFERENCES

- (HWF⁺05) Steven Hand, Andrew Warfield, Keir Fraser, Evangelos Kottsovinos, and Dan Magenheimer. Are virtual machine monitors microkernels done right? In *10th HotOS*, Sante Fe, NM, USA, Jun 2005. USENIX.
- (LUC⁺05) Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: Slashing the cost of virtualization. Technical Report PA005520, National ICT Australia, Oct 2005.
- (LUSG04) Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *6th OSDI*, San Francisco, CA, USA, Dec 2004.
- (PG74) Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation

REFERENCES

- architectures. *CACM*, 17(7):413–421, 1974.
- (RG05) Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. *IEEE Comp.*, 38(5):39–47, 2005.
- (SN05) James E. Smith and Ravi Nair. The architecture of virtual machines. *IEEE Comp.*, 38(5):32–38, 2005.
- (UNR⁺05) Rich Uhlig, Gil Neiger, Dion Rodgers, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kägi, Felix H Leung, and Larry Smith. Intel virtualization technology. *IEEE Comp.*, 38(5):48–56, May 2005.
- (WSG02) Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. In *5th OSDI*, Boston, MA, USA, Dec 2002.