CSCE 613: Virtualization

[ ] Overview

[13] Gerald J. Popek and Robert P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures". Communications of the ACM, Vol. 17, No. 7, July 1974, pp. 412 - 421.

[14] Keith Adams and Ole Agesen, "A Comparison of Software and Hardware Techniques for x86 Virtualization". Proceedings of the ASPLOS'06, October 2006, San Jose, CA.

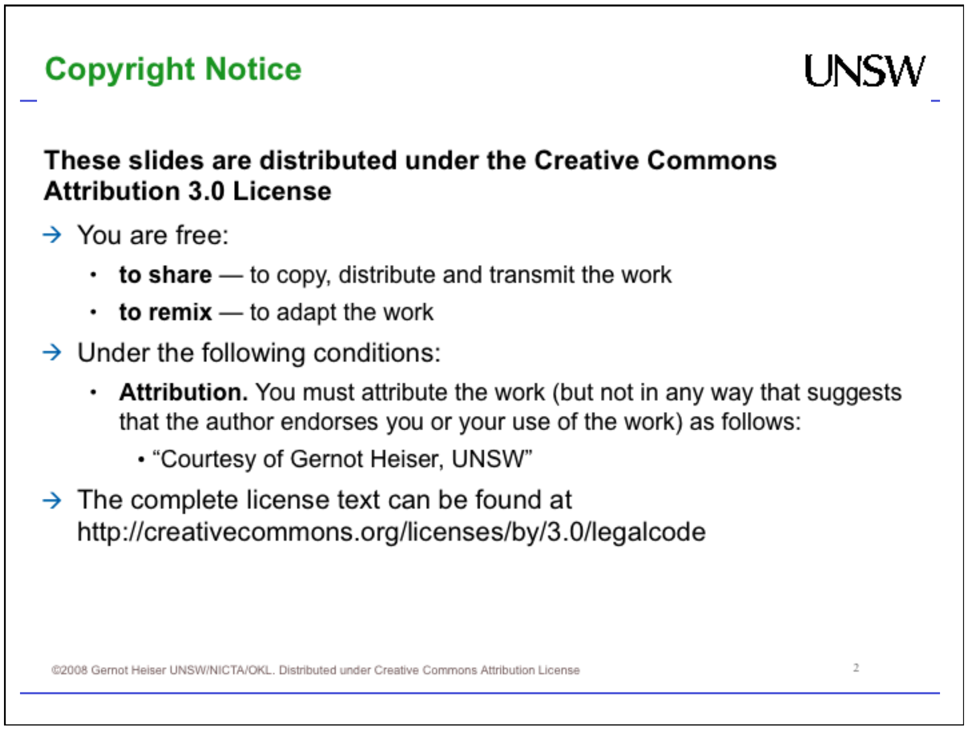
[15] Carl A. Waldspurger, "Memory Resource Management in VMWare ESX Server". Proceedings of OSDI'02.

[16] B. Yee, D. Sehr, G. Dardyk, J.B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code". Proceedings of the 2009 IEEE Symposium on Security and Privacy.

Virtual Machines: Overview/Recap

- Definitions, Terminology
- Why Virtual Machines?
- Mechanics of Virtualization

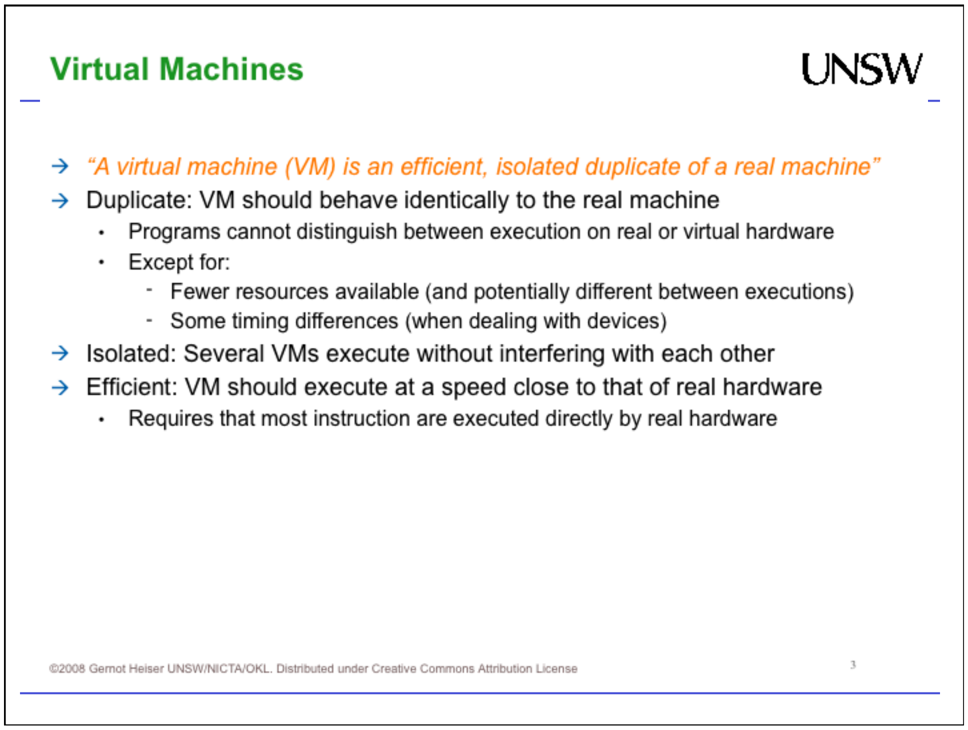
- Slides (for this part) made available Courtesy of Gernot Heiser, UNSW.



Copyright Notice

UNSW

2

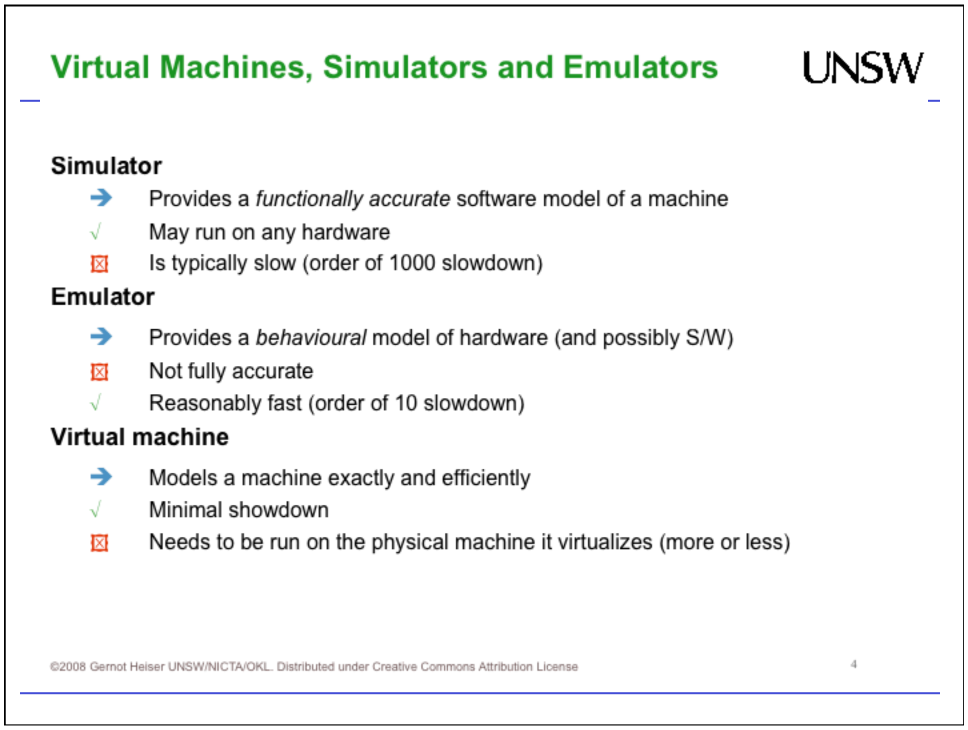


Virtual Machines

UNSW

- *"A virtual machine (VM) is an efficient, isolated duplicate of a real machine"*
- Duplicate: VM should behave identically to the real machine
  - Programs cannot distinguish between execution on real or virtual hardware
  - Except for:
    - Fewer 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 instruction are executed directly by real hardware

3



Virtual Machines, Simulators and Emulators

UNSW

Simulator

- → Provides a *functionally accurate* software model of a machine
- √ May run on any hardware
- ☒ Is typically slow (order of 1000 slowdown)

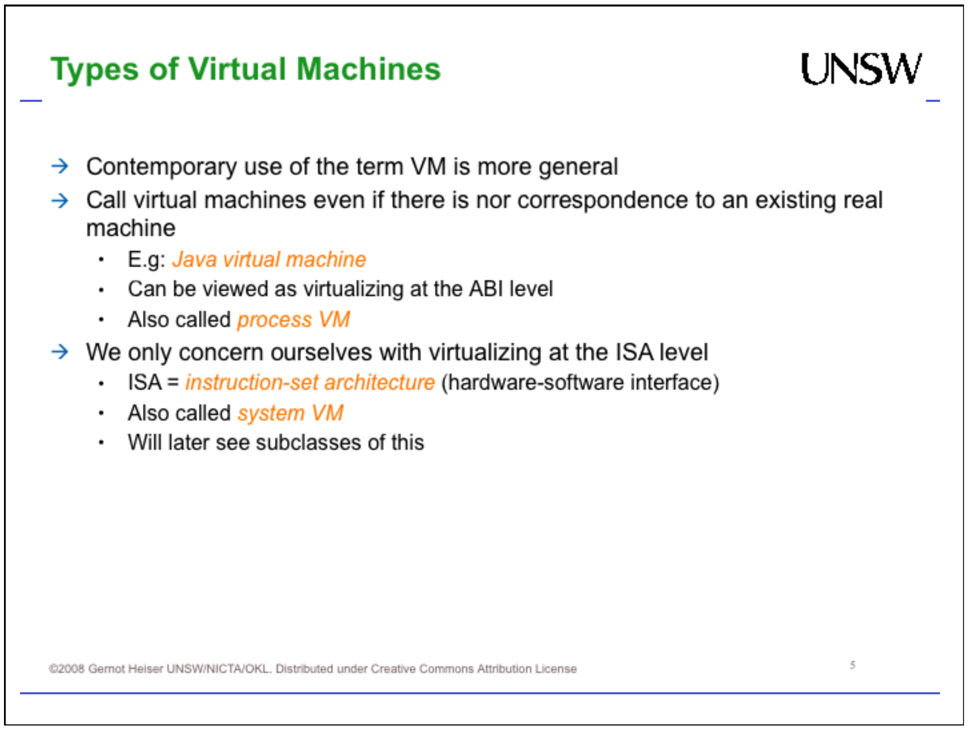
Emulator

- → Provides a *behavioural* model of hardware (and possibly S/W)
- ☒ Not fully accurate
- √ Reasonably fast (order of 10 slowdown)

Virtual machine

- → Models a machine exactly and efficiently
- √ Minimal showdown
- ☒ Needs to be run on the physical machine it virtualizes (more or less)

4

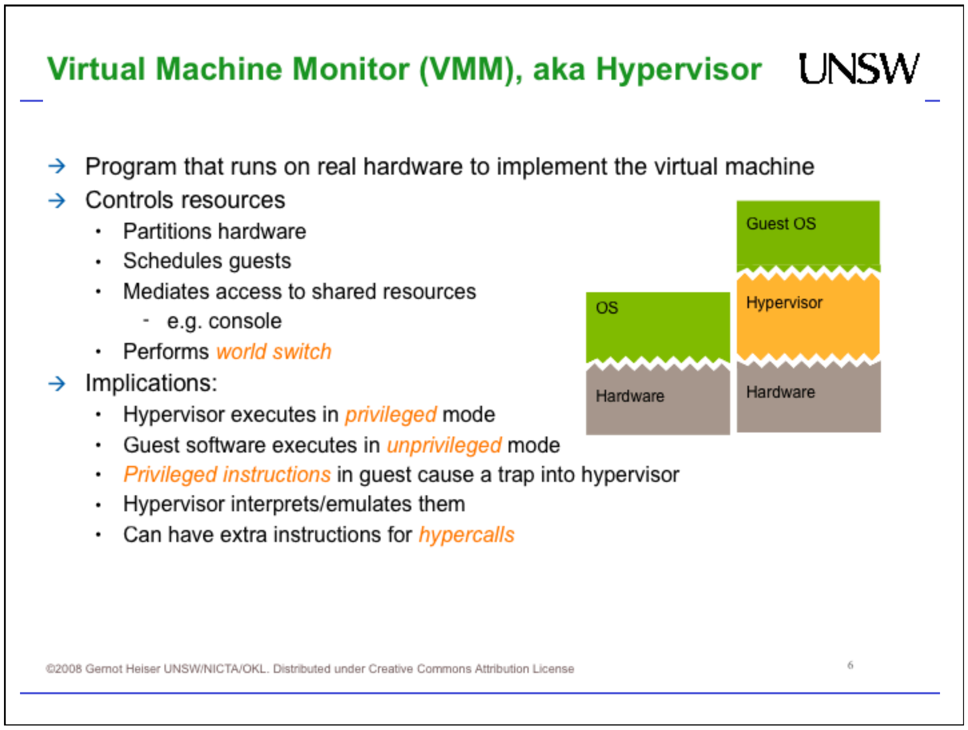


Types of Virtual Machines

UNSW

- → Contemporary use of the term VM is more general
- → Call virtual machines even if there is nor correspondence to an existing real machine
  - E.g: *Java virtual machine*
  - Can be viewed as virtualizing at the ABI level
  - Also called *process VM*
- → We only concern ourselves with virtualizing at the ISA level
  - ISA = *instruction-set architecture* (hardware-software interface)
  - Also called *system VM*
  - Will later see subclasses of this

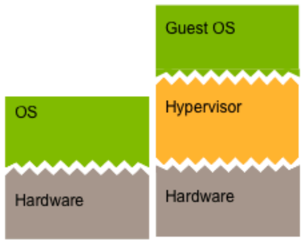
5



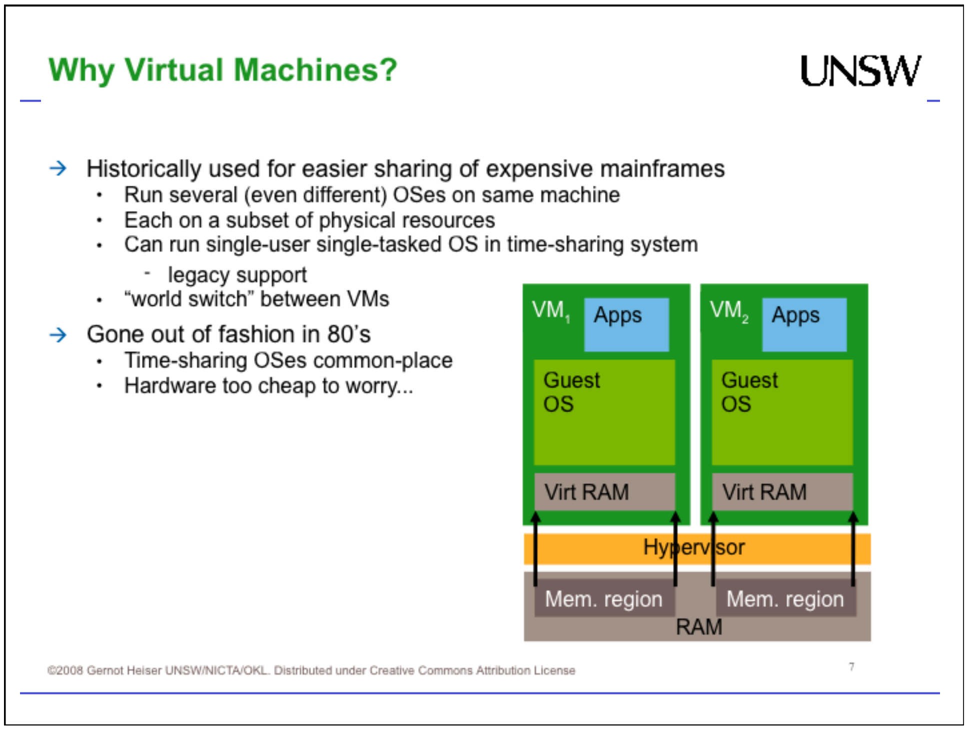
Virtual Machine Monitor (VMM), aka Hypervisor

UNSW

- Program that runs on real hardware to implement the virtual machine
- Controls resources
  - Partitions hardware
  - Schedules guests
  - Mediates access to shared resources
    - e.g. console
  - Performs *world switch*
- Implications:
  - Hypervisor executes in *privileged* mode
  - Guest software executes in *unprivileged* mode
  - *Privileged instructions* in guest cause a trap into hypervisor
  - Hypervisor interprets/emulates them
  - Can have extra instructions for *hypercalls*



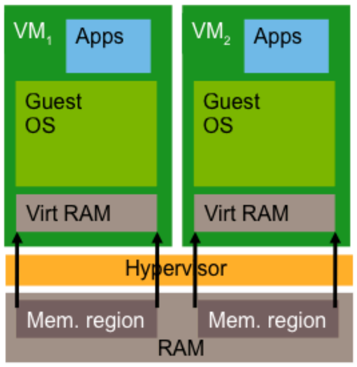
6



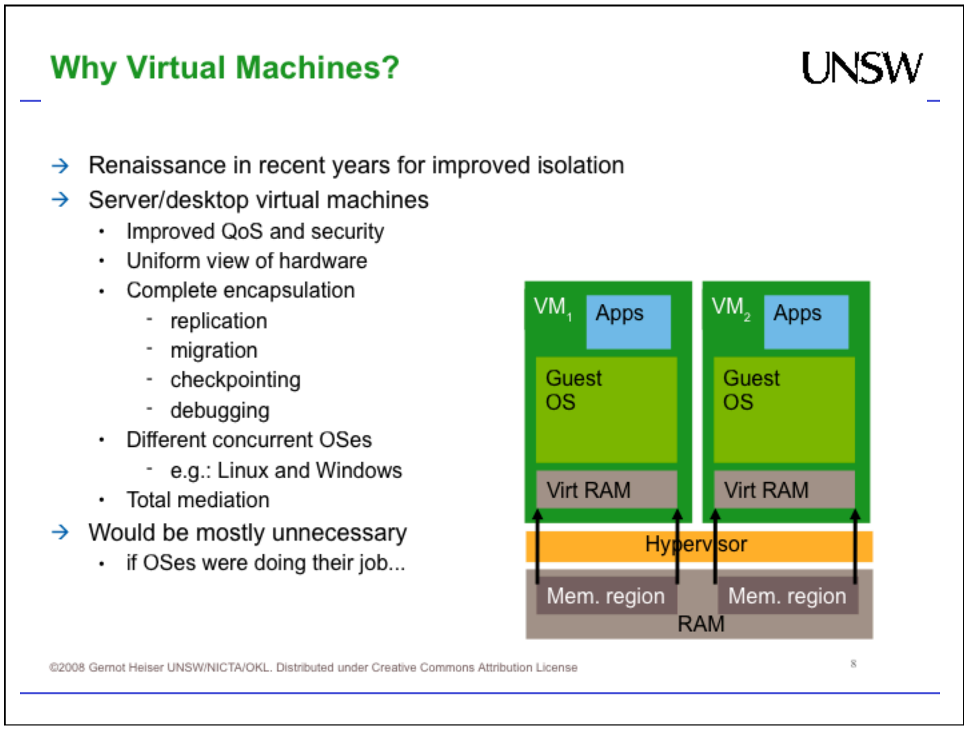
Why Virtual Machines?

UNSW

- Historically used for easier sharing of expensive mainframes
  - Run several (even different) OSes on same machine
  - Each on a subset of physical resources
  - Can run single-user single-tasked OS in time-sharing system
    - legacy support
  - "world switch" between VMs
- Gone out of fashion in 80's
  - Time-sharing OSes common-place
  - Hardware too cheap to worry...



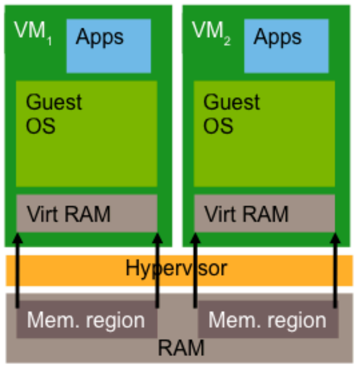
7



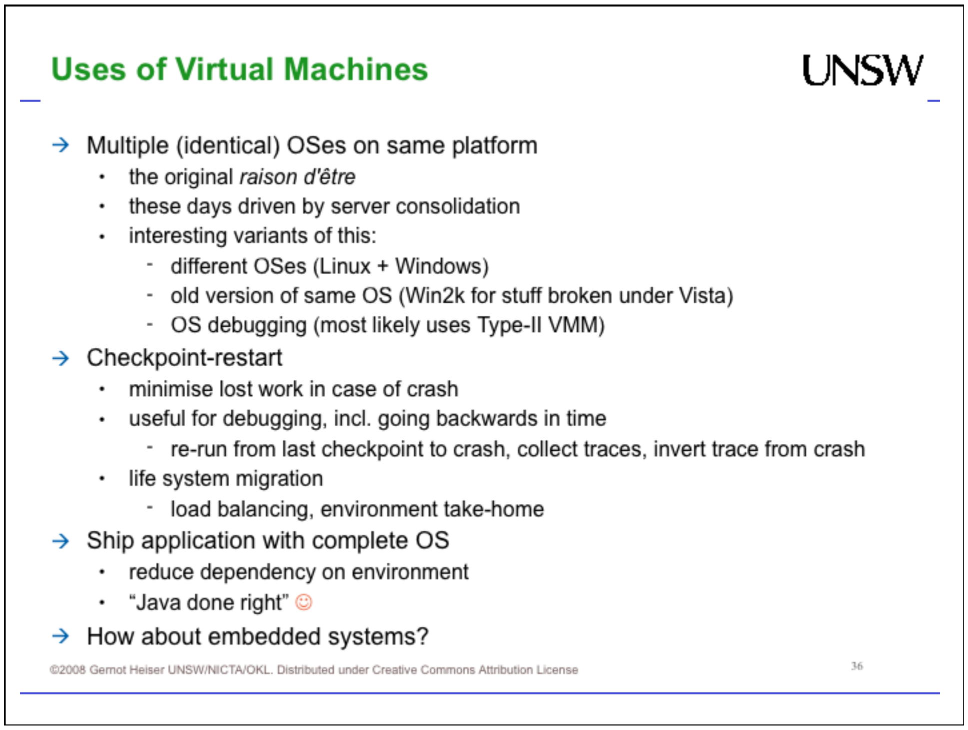
Why Virtual Machines?

UNSW

- Renaissance in recent years for improved isolation
- Server/desktop virtual machines
  - Improved QoS and security
  - Uniform view of hardware
  - Complete encapsulation
    - replication
    - migration
    - checkpointing
    - debugging
  - Different concurrent OSes
    - e.g.: Linux and Windows
  - Total mediation
- Would be mostly unnecessary
  - if OSes were doing their job...



8

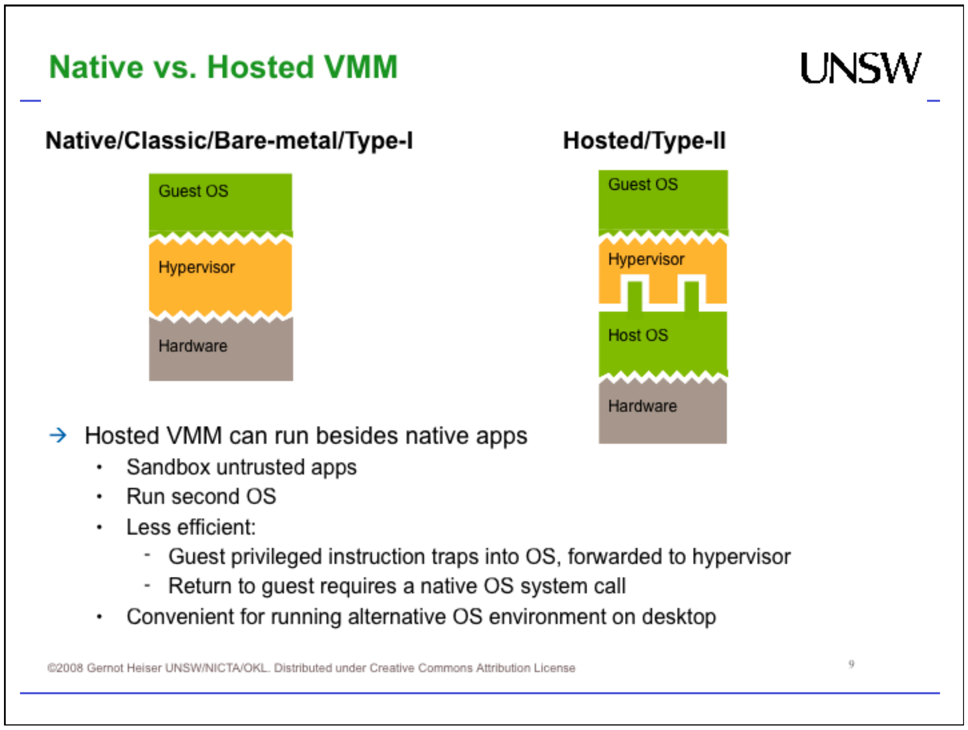


Uses of Virtual Machines

UNSW

- Multiple (identical) OSes on same platform
  - the original *raison d'être*
  - these days driven by server consolidation
  - interesting variants of this:
    - different OSes (Linux + Windows)
    - old version of same OS (Win2k for stuff broken under Vista)
    - OS debugging (most likely uses Type-II VMM)
- Checkpoint-restart
  - minimise lost work in case of crash
  - useful for debugging, incl. going backwards in time
    - re-run from last checkpoint to crash, collect traces, invert trace from crash
  - life system migration
    - load balancing, environment take-home
- Ship application with complete OS
  - reduce dependency on environment
  - "Java done right" ☺
- How about embedded systems?

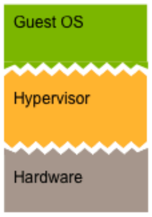
36



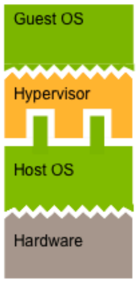
Native vs. Hosted VMM

UNSW

Native/Classic/Bare-metal/Type-I

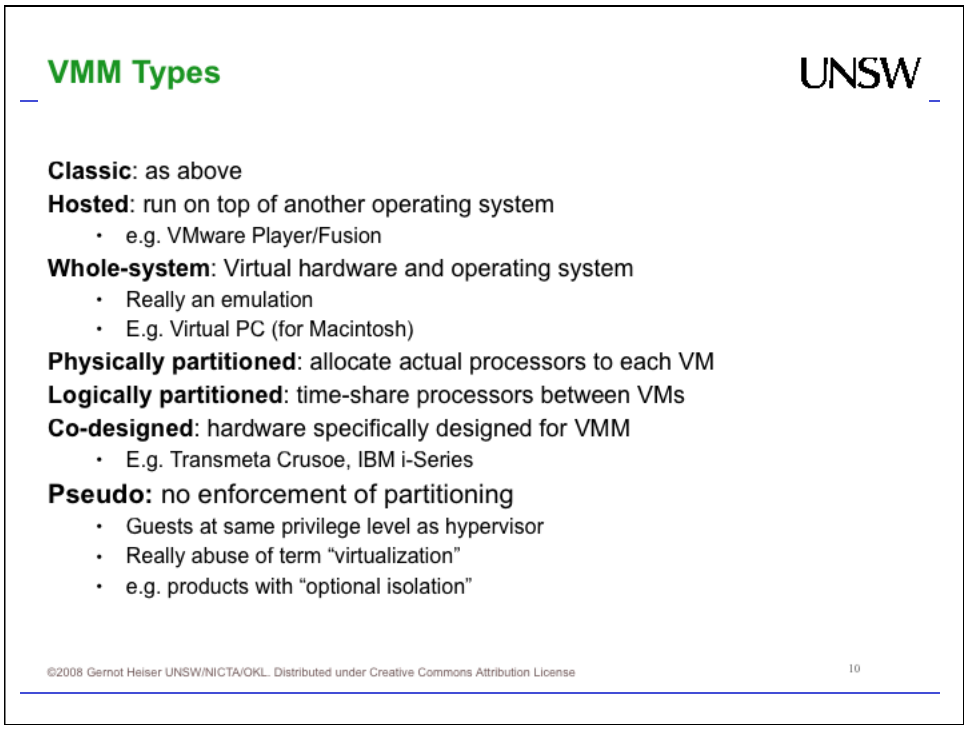


Hosted/Type-II



→ 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
- Convenient for running alternative OS environment on desktop

9



VMM Types

UNSW

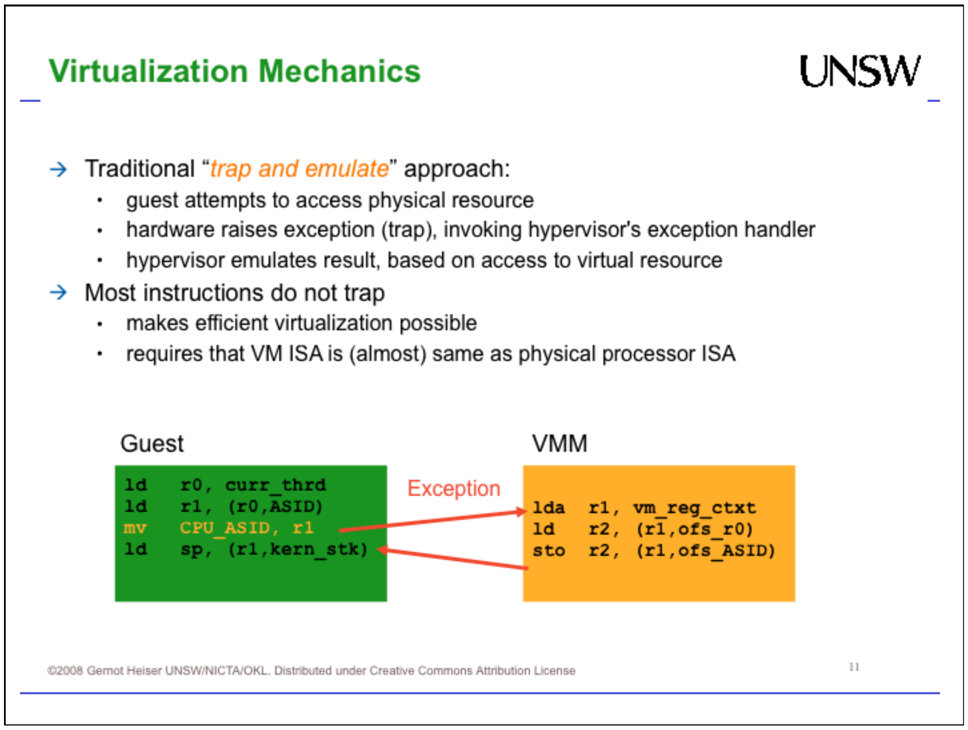
Classic: as above
Hosted: run on top of another operating system
- e.g. VMware Player/Fusion

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

Pseudo: no enforcement of partitioning
- Guests at same privilege level as hypervisor
- Really abuse of term “virtualization”
- e.g. products with “optional isolation”

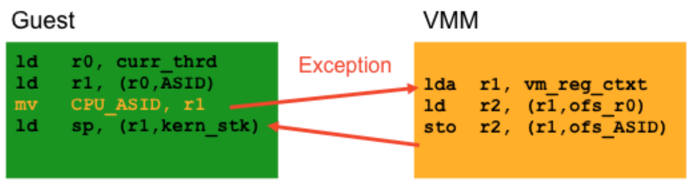
10



Virtualization Mechanics

UNSW

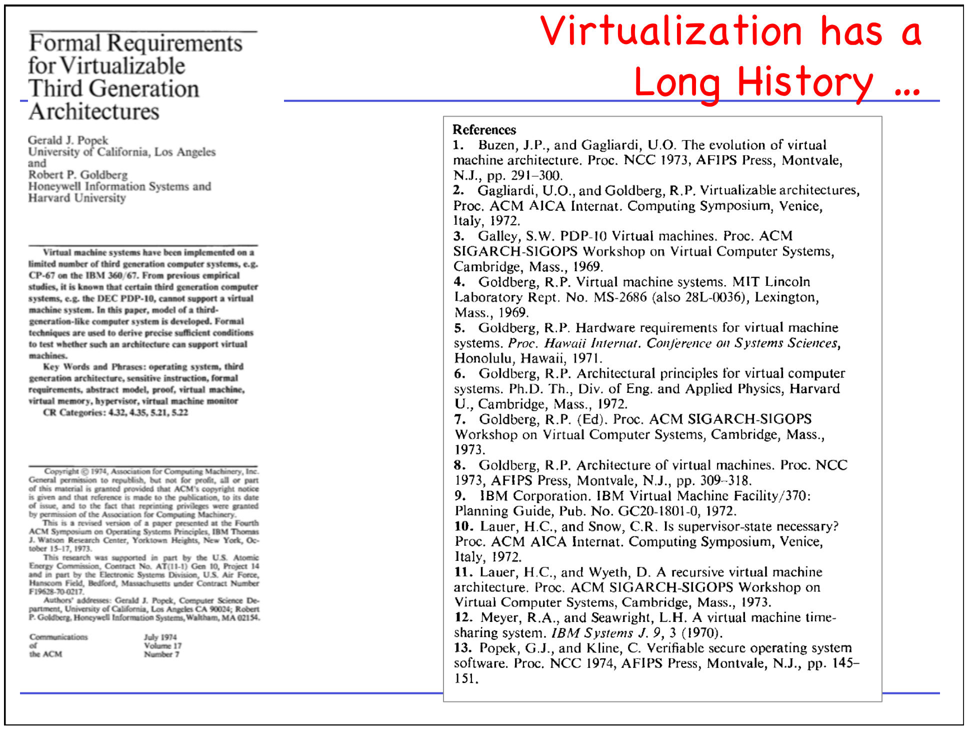
- Traditional "*trap and emulate*" approach:
  - guest attempts to access physical resource
  - hardware raises exception (trap), invoking hypervisor's exception handler
  - hypervisor emulates result, based on access to virtual resource
- Most instructions do not trap
  - makes efficient virtualization possible
  - requires that VM ISA is (almost) same as physical processor ISA



```
ld   r0, curr_thrd
ld   r1, (r0,ASID)
mv   CPU_ASID, r1
ld   sp, (r1,kern_stk)
```

```
lda  r1, vm_reg_ctxt
ld   r2, (r1,ofs_r0)
sto  r2, (r1,ofs_ASID)
```

11



Virtualization has a Long History …

Formal Requirements for Virtualizable Third Generation Architectures

Gerald J. Popek
University of California, Los Angeles
and
Robert P. Goldberg
Honeywell Information Systems and
Harvard University

Virtual machine systems have been implemented on a limited number of third generation computer systems, e.g. CP-67 on the IBM 360/67. From previous empirical studies, it is known that certain third generation computer systems, e.g. the DEC PDP-10, cannot support a virtual machine system. In this paper, model of a third-generation-like computer system is developed. Formal techniques are used to derive precise sufficient conditions to test whether such an architecture can support virtual machines.

Key Words and Phrases: operating system, third generation architecture, sensitive instruction, formal requirements, abstract model, proof, virtual machine, virtual memory, hypervisor, virtual machine monitor

CR Categories: 4.32, 4.35, 5.21, 5.22

This is a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15–17, 1973.

This research was supported in part by the U.S. Atomic Energy Commission, Contract No. AT(11-1) Gen 10, Project 14 and in part by the Electronic Systems Division, U.S. Air Force, Hanscom Field, Bedford, Massachusetts under Contract Number F19628-70-0217.

Authors' addresses: Gerald J. Popek, Computer Science Department, University of California, Los Angeles CA 90024; Robert P. Goldberg, Honeywell Information Systems, Waltham, MA 02154.

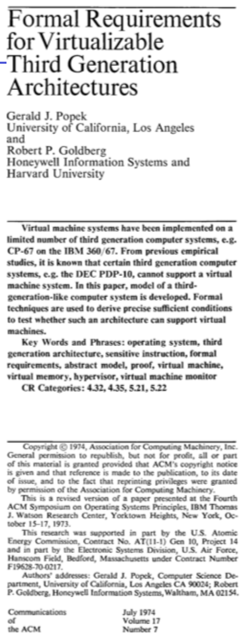
Communications of the ACM   July 1974 Volume 17 Number 7

References

1. Buzen, J.P., and Gagliardi, U.O. The evolution of virtual machine architecture. Proc. NCC 1973, AFIPS Press, Montvale, N.J., pp. 291–300.
2. Gagliardi, U.O., and Goldberg, R.P. Virtualizable architectures, Proc. ACM AICA Internat. Computing Symposium, Venice, Italy, 1972.
3. Galley, S.W. PDP-10 Virtual machines. Proc. ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems, Cambridge, Mass., 1969.
4. Goldberg, R.P. Virtual machine systems. MIT Lincoln Laboratory Rept. No. MS-2686 (also 28L-0036), Lexington, Mass., 1969.
5. Goldberg, R.P. Hardware requirements for virtual machine systems. *Proc. Hawaii Internat. Conference on Systems Sciences*, Honolulu, Hawaii, 1971.
6. Goldberg, R.P. Architectural principles for virtual computer systems. Ph.D. Th., Div. of Eng. and Applied Physics, Harvard U., Cambridge, Mass., 1972.
7. Goldberg, R.P. (Ed). Proc. ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems, Cambridge, Mass., 1973.
8. Goldberg, R.P. Architecture of virtual machines. Proc. NCC 1973, AFIPS Press, Montvale, N.J., pp. 309–318.
9. IBM Corporation. IBM Virtual Machine Facility/370: Planning Guide, Pub. No. GC20-1801-0, 1972.
10. Lauer, H.C., and Snow, C.R. Is supervisor-state necessary? Proc. ACM AICA Internat. Computing Symposium, Venice, Italy, 1972.
11. Lauer, H.C., and Wyeth, D. A recursive virtual machine architecture. Proc. ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems, Cambridge, Mass., 1973.
12. Meyer, R.A., and Seawright, L.H. A virtual machine time-sharing system. *IBM Systems J. 9*, 3 (1970).
13. Popek, G.J., and Kline, C. Verifiable secure operating system software. Proc. NCC 1974, AFIPS Press, Montvale, N.J., pp. 145–151.

[13] Formal Virtualization Reqs.

- Def: Machine State: $S = \langle E, M, P, R \rangle$
  - E executable storage
  - M processor mode
  - P program counter
  - R relocation-bounds register
- Def: Instruction i is **privileged** iff for any pair of states $S_1 = \langle e, super, p, r \rangle$ and $S_2 = \langle e, user, p, r \rangle$ in which $i(S_1)$ and $i(S_2)$ do not memory trap: $i(S_2)$ traps and $i(S_1)$ does not.
- Example: … many
- Def: Instruction i is **control sensitive** if there exists a state $S_1 = \langle e_1, m_1, p_1, r_1 \rangle$, and $i(S_1) = S_2 = \langle e_2, m_2, p_2, r_2 \rangle$ such that $i(S_1)$ does not memory trap, and either $r_1 \neq r_2$, or $m_1 \neq m_2$, or both.
- Example: manipulate PSW


Formal Requirements for Virtualizable Third Generation Architectures

Gerald J. Popek
University of California, Los Angeles
and
Robert P. Goldberg
Honeywell Information Systems and
Harvard University

Virtual machine systems have been implemented on a limited number of third generation computer systems, e.g. CP-67 on the IBM 360/67. From previous empirical studies, it is known that certain third generation computer systems, e.g. the DEC PDP-10, cannot support a virtual machine system. In this paper, model of a third-generation-like computer system is developed. Formal techniques are used to derive precise sufficient conditions to test whether such an architecture can support virtual machines.

Key Words and Phrases: operating system, third generation architecture, sensitive instruction, formal requirements, abstract model, proof, virtual machine, virtual memory, hypervisor, virtual machine monitor

CR Categories: 4.32, 4.35, 5.21, 5.22

Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This is a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15–17, 1973.

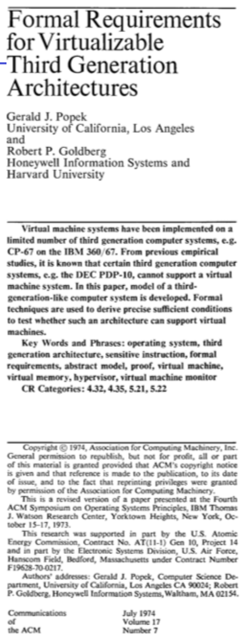
This research was supported in part by the U.S. Atomic Energy Commission, Contract No. AT(11-1) Gen 10, Project 14 and in part by the Electronic Systems Division, U.S. Air Force, Hanscom Field, Bedford, Massachusetts under Contract Number F19628-70-0217.

Authors' addresses: Gerald J. Popek, Computer Science Department, University of California, Los Angeles CA 90024; Robert P. Goldberg, Honeywell Information Systems, Waltham, MA 02154.

Communications of the ACM — July 1974, Volume 17, Number 7

Formal Virtualization Reqs. (2)

- Def: Machine State: $S = \langle E, M, P, R \rangle$
  - E executable storage
  - M processor mode
  - P program counter
  - R relocation-bounds register
- Def: Instruction i is **behavior sensitive** if there exists an integer x and states:
  (a) $S_1 = \langle e \mid r, m_1, p, r \rangle$, and
  (b) $S_2 = \langle e \mid r * x, m_2, p, r * x \rangle$,
  where …
- Intuitively, and instruction is behavior sensitive if the effect of its execution depends on the value of the relocation-bounds register, i.e. upon its location in real memory, or on the mode.
- Example: load physical address!


Formal Requirements for Virtualizable Third Generation Architectures

Gerald J. Popek
University of California, Los Angeles
and
Robert P. Goldberg
Honeywell Information Systems and
Harvard University

Virtual machine systems have been implemented on a limited number of third generation computer systems, e.g. CP-67 on the IBM 360/67. From previous empirical studies, it is known that certain third generation computer systems, e.g. the DEC PDP-10, cannot support a virtual machine system. In this paper, model of a third-generation-like computer system is developed. Formal techniques are used to derive precise sufficient conditions to test whether such an architecture can support virtual machines.

Key Words and Phrases: operating system, third generation architecture, sensitive instruction, formal requirements, abstract model, proof, virtual machine, virtual memory, hypervisor, virtual machine monitor

CR Categories: 4.32, 4.35, 5.21, 5.22

Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This is a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15–17, 1973.

This research was supported in part by the U.S. Atomic Energy Commission, Contract No. AT(11-1) Gen 10, Project 14 and in part by the Electronic Systems Division, U.S. Air Force, Hanscom Field, Bedford, Massachusetts under Contract Number F19628-70-0217.

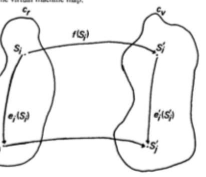
Authors' addresses: Gerald J. Popek, Computer Science Department, University of California, Los Angeles CA 90024; Robert P. Goldberg, Honeywell Information Systems, Waltham, MA 02154.

Communications of the ACM — July 1974, Volume 17, Number 7

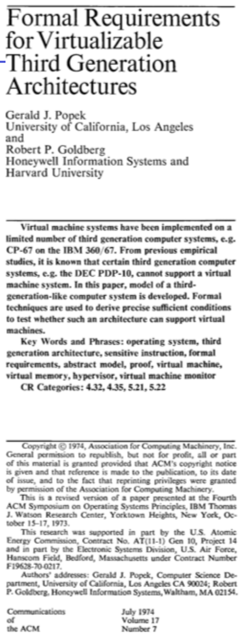
Formal Virtualization Reqs. (3)

- Theorem: "For any conventional third generation [1974] computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions."
- Virtual Machine Map:

Fig. 2. The virtual machine map.



- Recursive Virtualization: "A conventional third generation computer is recursively virtualizable if it is (a) virtualizable, and (b) a VMM without any timing dependencies can be constructed for it."


Formal Requirements for Virtualizable Third Generation Architectures

Gerald J. Popek
University of California, Los Angeles
and
Robert P. Goldberg
Honeywell Information Systems and
Harvard University

Virtual machine systems have been implemented on a limited number of third generation computer systems, e.g. CP-67 on the IBM 360/67. From previous empirical studies, it is known that certain third generation computer systems, e.g. the DEC PDP-10, cannot support a virtual machine system. In this paper, model of a third-generation-like computer system is developed. Formal techniques are used to derive precise sufficient conditions to test whether such an architecture can support virtual machines.

Key Words and Phrases: operating system, third generation architecture, sensitive instruction, formal requirements, abstract model, proof, virtual machine, virtual memory, hypervisor, virtual machine monitor

CR Categories: 4.32, 4.35, 5.21, 5.22

This is a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15–17, 1973.

This research was supported in part by the U.S. Atomic Energy Commission, Contract No. AT(11-1) Gen 10, Project 14 and in part by the Electronic Systems Division, U.S. Air Force, Hanscom Field, Bedford, Massachusetts under Contract Number F19628-70-0217.

Authors' addresses: Gerald J. Popek, Computer Science Department, University of California, Los Angeles CA 90024; Robert P. Goldberg, Honeywell Information Systems, Waltham, MA 02154.

Communications of the ACM
July 1974
Volume 17
Number 7

Formal Virtualization Reqs. (4)

- "Hybrid" Virtualization (with interpreted instr's):
- Def: Machine State: S = <E, M, P, R>
  - E executable storage
  - M processor mode
  - P program counter
  - R relocation-bounds register
- Def: Instruction i is user sensitive if there exists a state S = <E, user, P, R> for which i is control sensitive or behavior sensitive.
- Theorem: A hybrid virtual machine (HVMM) monitor may be constructed for any conventional third generation machine in which the set of user sensitive instructions are a subset of the set of privileged instructions.
- Example: PDP-10 JRST 1 (return to user mode) is non-privileged, but supervisor control sensitive. Therefore, PDP-10 cannot host VMM, but can host HVMM.

Formal Requirements for Virtualizable Third Generation Architectures

Gerald J. Popek
University of California, Los Angeles
and
Robert P. Goldberg
Honeywell Information Systems and
Harvard University

Virtual machine systems have been implemented on a limited number of third generation computer systems, e.g. CP-67 on the IBM 360/67. From previous empirical studies, it is known that certain third generation computer systems, e.g. the DEC PDP-10, cannot support a virtual machine system. In this paper, model of a third-generation-like computer system is developed. Formal techniques are used to derive precise sufficient conditions to test whether such an architecture can support virtual machines.

Key Words and Phrases: operating system, third generation architecture, sensitive instruction, formal requirements, abstract model, proof, virtual machine, virtual memory, hypervisor, virtual machine monitor

CR Categories: 4.32, 4.35, 5.21, 5.22

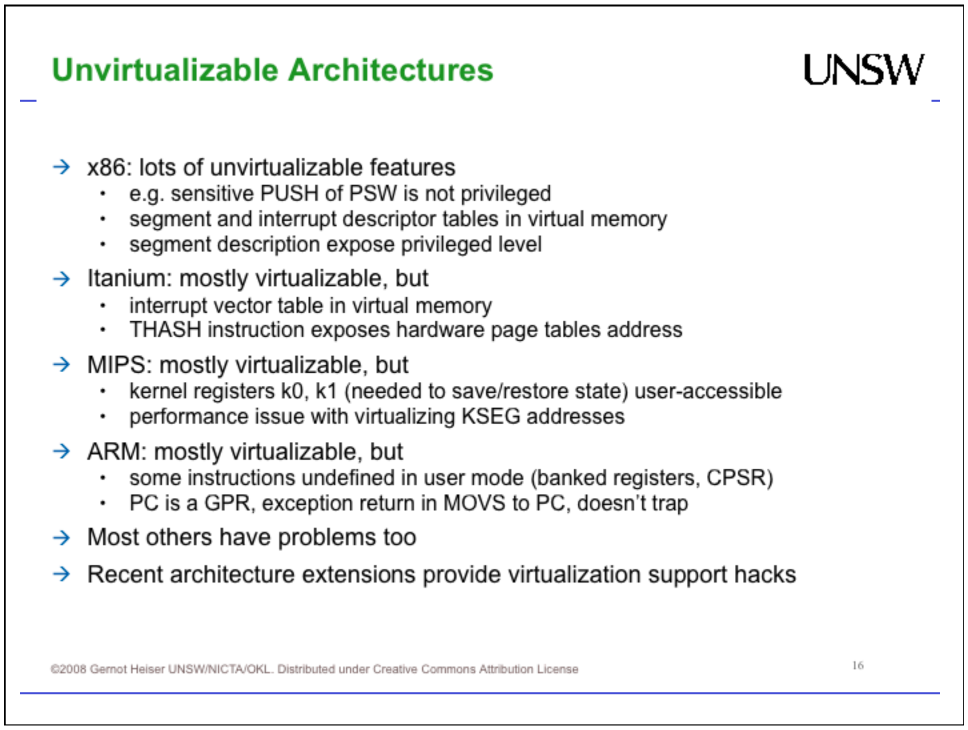
Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This is a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15–17, 1973.

This research was supported in part by the U.S. Atomic Energy Commission, Contract No. AT(11-1) Gen 10, Project 14 and in part by the Electronic Systems Division, U.S. Air Force, Hanscom Field, Bedford, Massachusetts under Contract Number F19628-70-0217.

Authors' addresses: Gerald J. Popek, Computer Science Department, University of California, Los Angeles CA 90024; Robert P. Goldberg, Honeywell Information Systems, Waltham, MA 02154.

Communications of the ACM
July 1974
Volume 17
Number 7

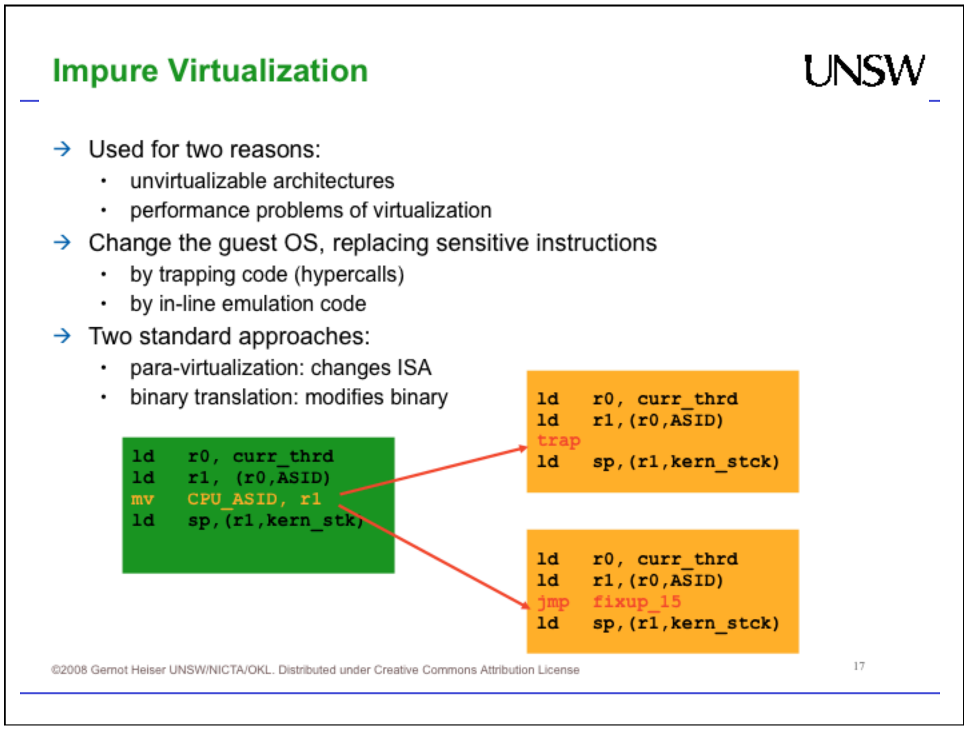


Unvirtualizable Architectures

UNSW

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

16



Impure Virtualization

UNSW

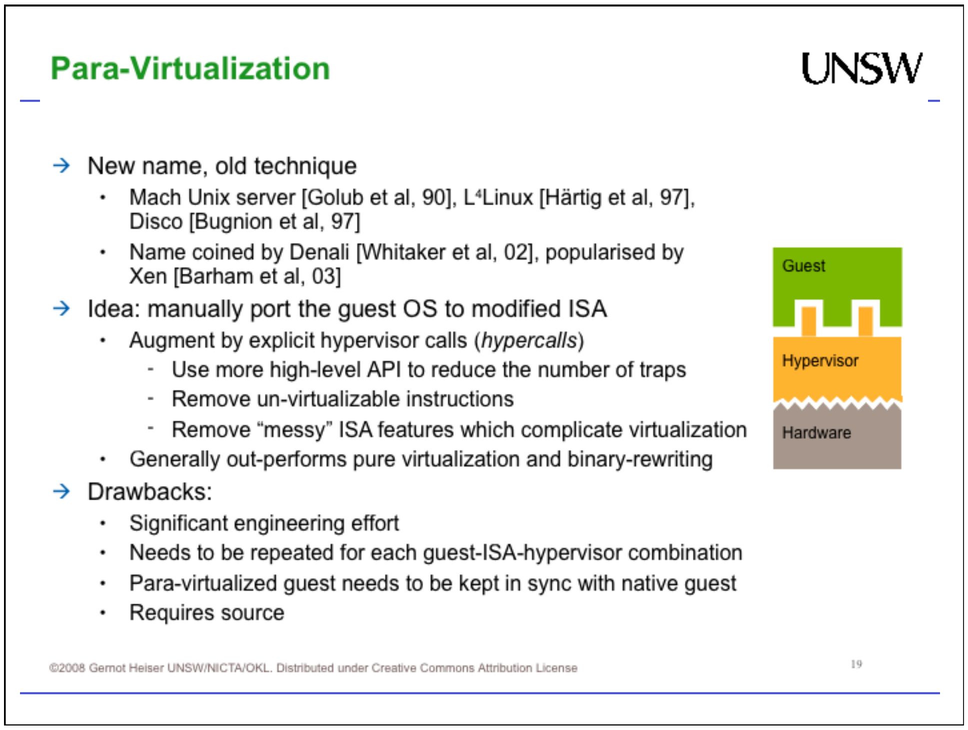
- Used for two reasons:
  - unvirtualizable architectures
  - performance problems of virtualization
- Change the guest OS, replacing sensitive instructions
  - by trapping code (hypercalls)
  - by in-line emulation code
- Two standard approaches:
  - para-virtualization: changes ISA
  - binary translation: modifies binary

```
ld   r0, curr_thrd
ld   r1, (r0,ASID)
mv   CPU_ASID, r1
ld   sp,(r1,kern_stk)
```

```
ld   r0, curr_thrd
ld   r1,(r0,ASID)
trap
ld   sp,(r1,kern_stck)
```

```
ld   r0, curr_thrd
ld   r1,(r0,ASID)
jmp  fixup_15
ld   sp,(r1,kern_stck)
```

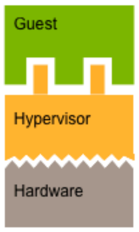
17

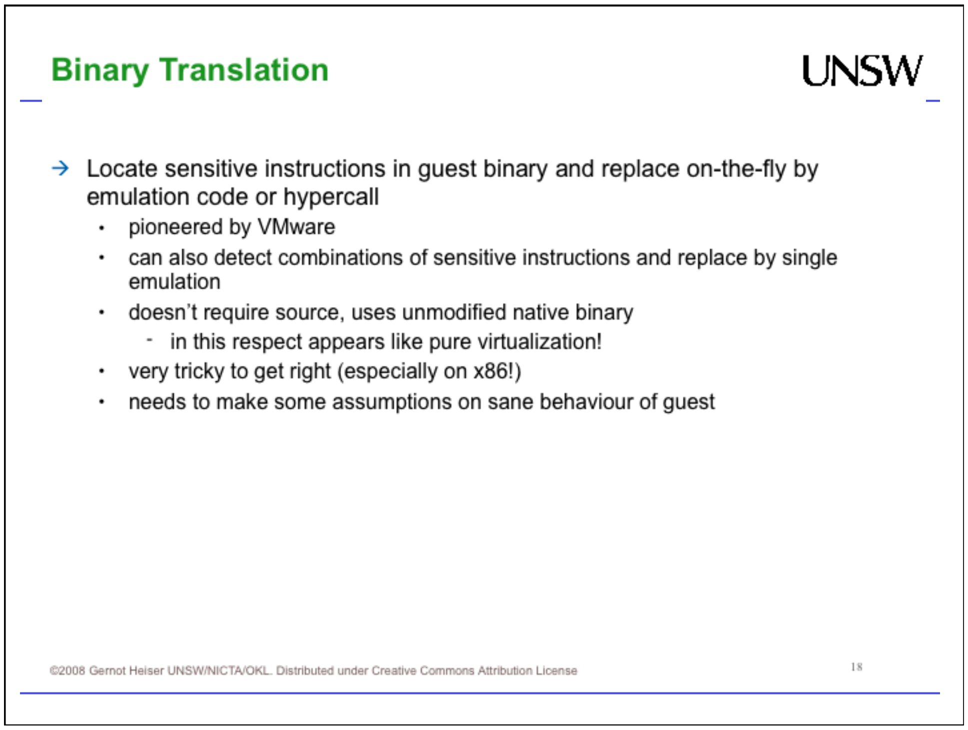


Para-Virtualization

UNSW

- New name, old technique
  - Mach Unix server [Golub et al, 90], L4Linux [Härtig et al, 97], Disco [Bugnion et al, 97]
  - Name coined by Denali [Whitaker et al, 02], popularised by Xen [Barham et al, 03]
- Idea: manually port the guest OS to modified ISA
  - Augment by explicit hypervisor calls (*hypercalls*)
    - Use more high-level API to reduce the number of traps
    - Remove un-virtualizable instructions
    - Remove “messy” ISA features which complicate virtualization
  - Generally out-performs pure virtualization and binary-rewriting
- Drawbacks:
  - Significant engineering effort
  - Needs to be repeated for each guest-ISA-hypervisor combination
  - Para-virtualized guest needs to be kept in sync with native guest
  - Requires source





Binary Translation

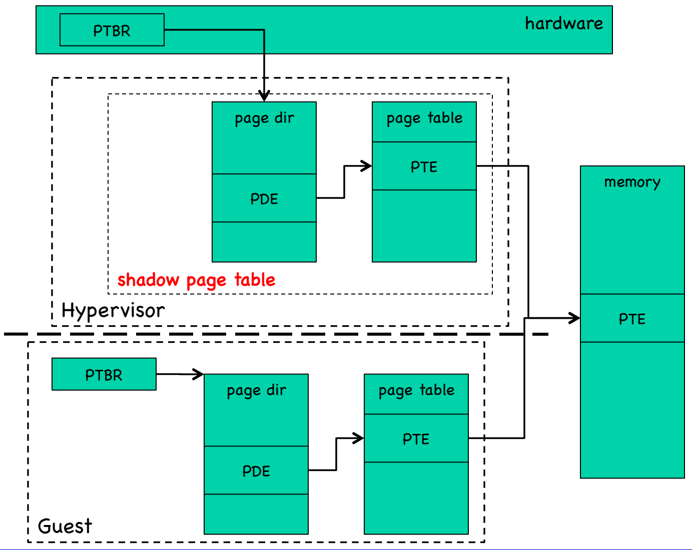
UNSW

- Locate sensitive instructions in guest binary and replace on-the-fly by emulation code or hypercall
  - pioneered by VMware
  - can also detect combinations of sensitive instructions and replace by single emulation
  - doesn’t require source, uses unmodified native binary
    - in this respect appears like pure virtualization!
  - very tricky to get right (especially on x86!)
  - needs to make some assumptions on sane behaviour of guest

Memory Virtualization

- Note: Guest OS expects zero-based physical address space.
- In traditional system:
  virtual address -> physical address
- In VMM system:
  virtual address -> physical address -> machine address
- Each VM maintains pmap to translate physical pages to machine pages.
- Operations on TLB are intercepted by VMM, which prevents manipulation of the MMU by the guest.
- Mapping from virtual pages to machine pages is maintained in **shadow page table**.
  - This table is used by the CPU!
  - Is maintained consistent with physical -> machine mapping.

Shadow Page Table

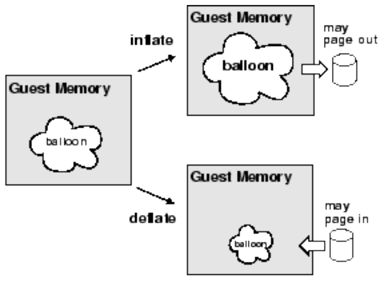


Every time the guest modifies its page mapping, either by changing the content of a translation, creating a new translation, or removing an existing translation, the virtual MMU module will capture the modification and adjust the **shadow page tables** accordingly.

Issues in Page Replacement

- Memory Over-Commitment: What if memory requirements exceed available resources?
  - Move some "physical" memory to disk.
- Issue 1: How does this affect page replacement?
  - A page replacement algorithm now needs to pick
    - victim virtual machine (ok)
    - victim page (huh?! what is a good page to replace?!)
- Issue 2: Double-Paging Problem:
  - What can happen when we page out a "physical" page that is on disk?
    1. Guest picks "physical" on disk as victim.
    2. In order to page it out by guest, it needs to be paged-in by VMM beforehand.
  - This causes two page faults per fault.

Avoiding paged-out "physical" pages



Ballooning. "ESX Server controls a *balloon module running within the guest, directing it to allocate guest pages and pin them in "physical" memory. The machine pages backing this memory can then be reclaimed by ESX Server. Inflating the balloon increases memory pressure, forcing the guest OS to invoke its own memory management algorithms. The guest OS may page out to its virtual disk when memory is scarce. Deflating the balloon decreases pressure, freeing guest memory." (Waldspurger, OSDI'02)*

Potential Problems with Ballooning

- Ballooning works fine as long as it works.
- Ballooning drivers may be uninstalled, disabled explicitly, unavailable during booting.
- Upper levels on balloon sizes may be imposed by guest OSs.

- Solution: Fall back on basic paging mechanisms…
  - Problems?

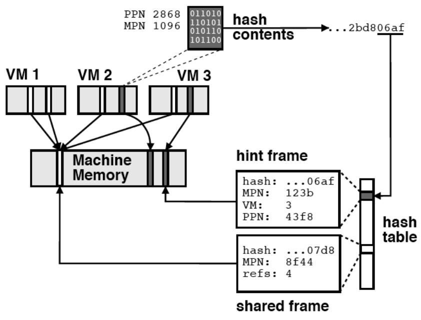
Memory Sharing across Virtual Machines

- Why memory sharing?
  - Eliminate redundant copies of pages.
  - This allows for more over-commitment of memory.

- Example: Transparent page sharing in Disco
  - Map multiple "physical" pages onto machine page, and mark it as copy-on-write.
  - Q: How do we know when a redundant copy has been created?
  - A: Need hooks into guest OS!

- Content-Based Page Sharing
  - Identify shareable pages by their content.
  - Agnostic about origin of generation of identical pages.
  - Use hashing to identify potentially shareable pages.

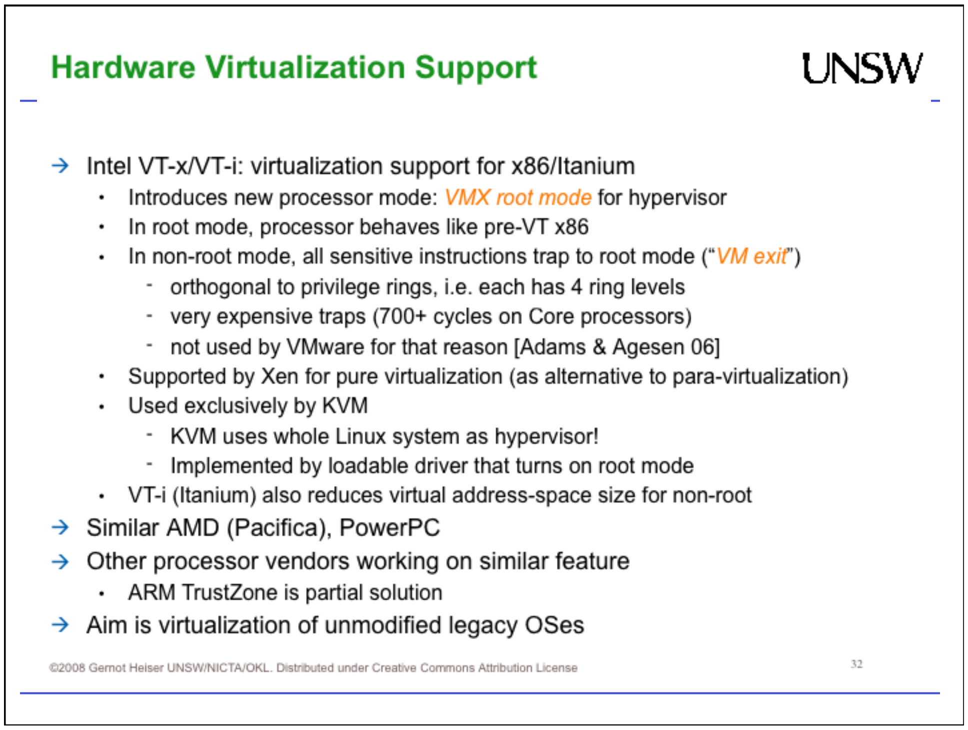
Content-Based Page Sharing in ESX Server



Content-Based Page Sharing. *ESX Server scans for sharing opportunities, hashing the contents of candidate PPN 0x2868 in VM 2. The hash is used to index into a table containing other scanned pages, where a match is found with a hint frame associated with PPN 0x43f8 in VM 3. If a full comparison confirms the pages are identical, the PPN-to-MPN mapping for PPN 0x2868 in VM2 is changed from MPN 0x1096 to MPN 0x123b, both PPNs are marked COW, and the redundant MPN is reclaimed.*

How to Adjust Memory Allocation

- Memory allocation with unequal requirements across VMs?
- Fair allocation: e.g. Proportional Share algorithms.
- Reclaiming idle memory: idle memory tax.
- How to measure idle memory: sampling.

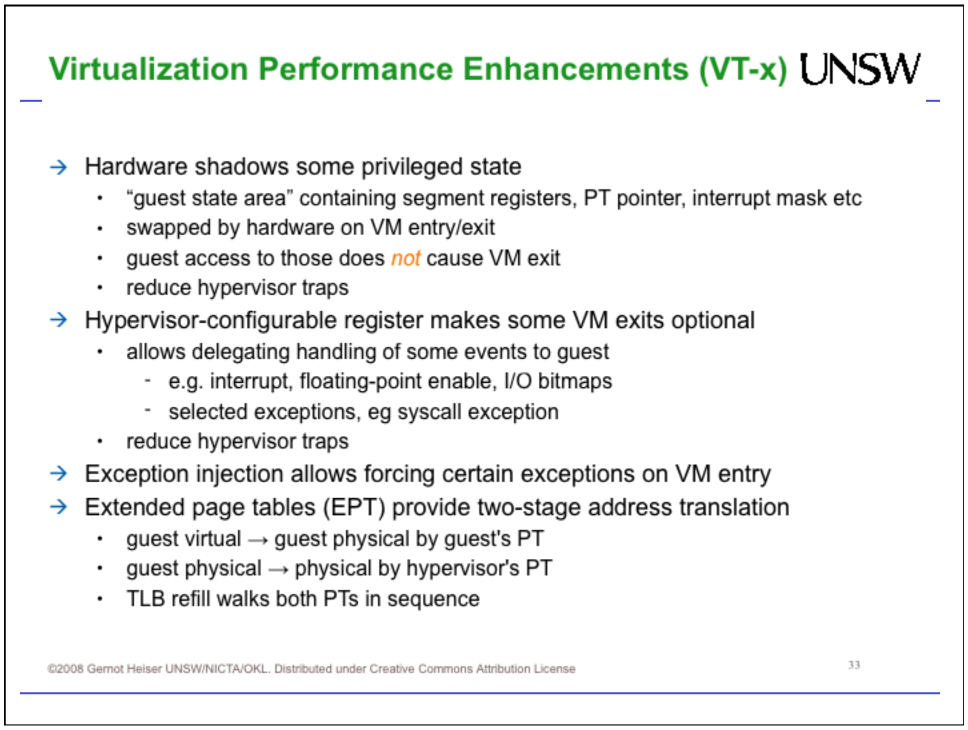


Hardware Virtualization Support

UNSW

- Intel VT-x/VT-i: virtualization support for x86/Itanium
  - Introduces new processor mode: *VMX root mode* for hypervisor
  - In root mode, processor behaves like pre-VT x86
  - In non-root mode, all sensitive instructions trap to root mode ("*VM exit*")
    - orthogonal to privilege rings, i.e. each has 4 ring levels
    - very expensive traps (700+ cycles on Core processors)
    - not used by VMware for that reason [Adams & Agesen 06]
  - Supported by Xen for pure virtualization (as alternative to para-virtualization)
  - Used exclusively by KVM
    - KVM uses whole Linux system as hypervisor!
    - Implemented by loadable driver that turns on root mode
  - VT-i (Itanium) also reduces virtual address-space size for non-root
- Similar AMD (Pacifica), PowerPC
- Other processor vendors working on similar feature
  - ARM TrustZone is partial solution
- Aim is virtualization of unmodified legacy OSes

32



Virtualization Performance Enhancements (VT-x)

UNSW

- Hardware shadows some privileged state
  - "guest state area" containing segment registers, PT pointer, interrupt mask etc
  - swapped by hardware on VM entry/exit
  - guest access to those does *not* cause VM exit
  - reduce hypervisor traps
- Hypervisor-configurable register makes some VM exits optional
  - allows delegating handling of some events to guest
    - e.g. interrupt, floating-point enable, I/O bitmaps
    - selected exceptions, eg syscall exception
  - reduce hypervisor traps
- Exception injection allows forcing certain exceptions on VM entry
- Extended page tables (EPT) provide two-stage address translation
  - guest virtual → guest physical by guest's PT
  - guest physical → physical by hypervisor's PT
  - TLB refill walks both PTs in sequence

33

I/O Virtualization Enhancements (VT-d)

UNSW

- Introduce separate *I/O address space*
- Mapped to physical address space by I/O MMU
  - under hypervisor control
- Makes DMA safely virtualizable
  - device can only read/write RAM that is mapped into its I/O space
- Useful not only for virtualization
  - safely encapsulated user-level drivers for DMA-capable devices
  - ideal for microkernels ☺
- AMD IOMMU is essentially same
- Similar features existed on high-end Alpha and HP boxes
- ... and, of course, IBM channels since the '70s...

34

Binary Translation

[14] Keith Adams and Ole Agesen, "A Comparison of Software and Hardware Techniques for x86 Virtualization". Proceedings of the ASPLOS'06, October 2006, San Jose, CA.

Recall: Characteristics of Virtualization

1. Fidelity: VMM is transparent, except for performance.
2. Performance: Most instructions executed on HW directly.
3. Safety: VMM manages all HW resources.

Techniques in Classical Virtualization

- De-privileging
  - All instructions that read/write privileged state trap when executed in unprivileged level.
  - Execute guest OS directly, but at unprivileged level.
- Primary and Shadow Structures
  - On-CPU privileged state: easy! maintained in context descriptor. Associated with traps.
  - Off-CPU privileged state: Not associated with traps.
- Memory Traces
  - Use memory protection mechanisms to enforce coherency of shadow and primary structures.
  - e.g. primary and shadow Page Table Entries
  - e.g. primary and shadow memory-mappings for devices

Extensions/Refinements to Classical Virt.

- Para-Virtualization
  - "Modify quest operating system to provide higher-level information to VMM."
- Interpretive Execution
  - Add dedicated HW execution mode for running the guest OS.
  - e.g. IBM 370 SIE ("start interpretive execution") instruction.
  - Allows for access of shadow fields in interpretive execution
  - Reduces number of required traps.

Obstacles to Virtualization

- "Visibility of Privileged State"
  - e.g. Current Privilege Level is stored in code segment register.
  - Guest therefore can know that it runs in deprivileged mode.
- "Lack of Traps when Privileged Instructions run at User-Level"
  - Some privileged instructions generate NOOP in user mode rather than generating a trap.
  - e.g. "pop flags", which modifies ALU and system flags, must generate trap for VMM to intervene.

VMware Software VMM: Binary Translation

- Traditionally, software VMMs run very slow due to interpretation.
- Binary Translation:
  - Binaries as input, not source code.
  - Dynamic translation at run-time.
  - On-demand (lazy) translation -> no need to explicitly separate data from code.
  - Instruction-level translation, not at higher ABI level.
  - Input is full x86 instruction set. Output is safe subset.
  - Adaptive. Adjust translated code as guest behavior changes.

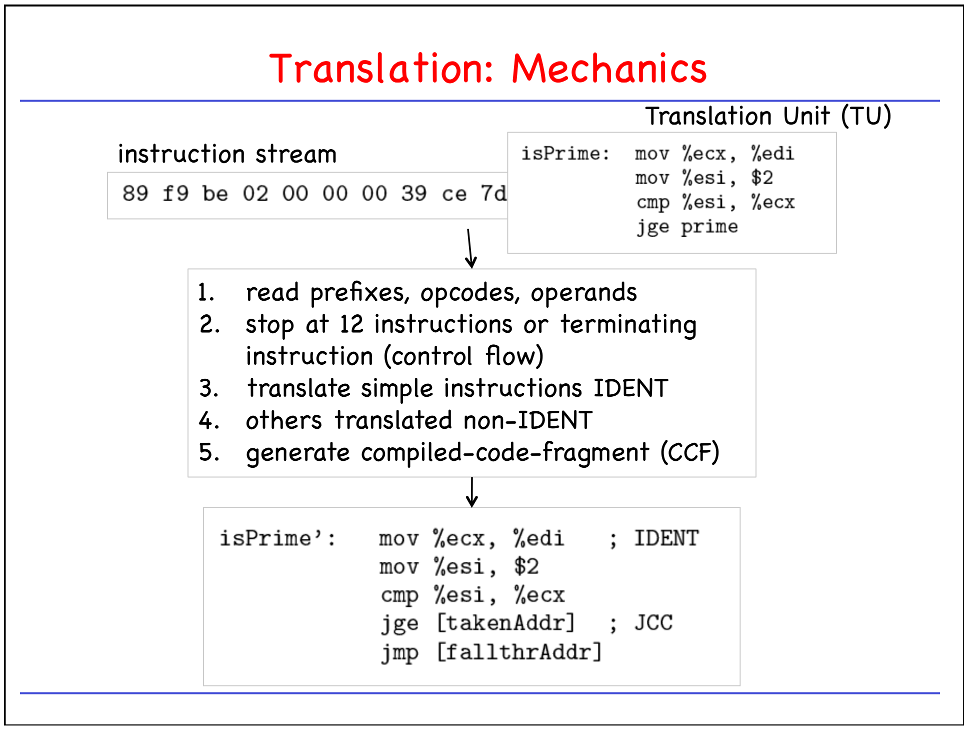
Binary Translation: Simple Example

```
int isPrime(int a) {
  for (int i = 2; i < a; i++) {
    if (a % i == 0) return 0;
  }
  return 1;
}
```

<- small example, C code

same code, compiled ->

```
isPrime:  mov    %ecx, %edi ; %ecx = %edi (a)
          mov    %esi, $2   ; i = 2
          cmp    %esi, %ecx ; is i >= a?
          jge    prime      ; jump if yes
nexti:    mov    %eax, %ecx ; set %eax = a
          cdq               ; sign-extend
          idiv   %esi       ; a % i
          test   %edx, %edx ; is remainder zero?
          jz     notPrime   ; jump if yes
          inc    %esi       ; i++
          cmp    %esi, %ecx ; is i >= a?
          jl     nexti      ; jump if no
prime:    mov    %eax, $1   ; return value in %eax
          ret
notPrime: xor    %eax, %eax ; %eax = 0
          ret
```



Translation: Mechanics

instruction stream

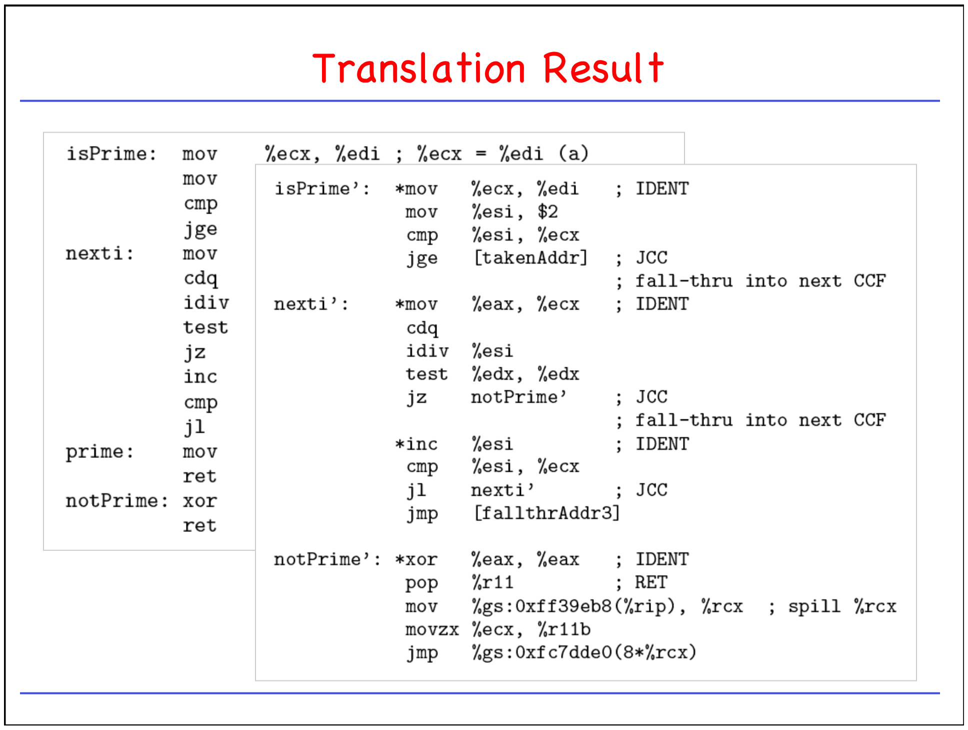
```
89 f9 be 02 00 00 00 39 ce 7d
```

Translation Unit (TU)

```
isPrime:  mov %ecx, %edi
          mov %esi, $2
          cmp %esi, %ecx
          jge prime
```

1. read prefixes, opcodes, operands
2. stop at 12 instructions or terminating instruction (control flow)
3. translate simple instructions IDENT
4. others translated non-IDENT
5. generate compiled-code-fragment (CCF)

```
isPrime':   mov %ecx, %edi    ; IDENT
            mov %esi, $2
            cmp %esi, %ecx
            jge [takenAddr]   ; JCC
            jmp [fallthrAddr]
```



Translation Result

```
isPrime:   mov    %ecx, %edi ; %ecx = %edi (a)
           mov
           cmp
           jge
nexti:     mov
           cdq
           idiv
           test
           jz
           inc
           cmp
           jl
prime:     mov
           ret
notPrime:  xor
           ret
```

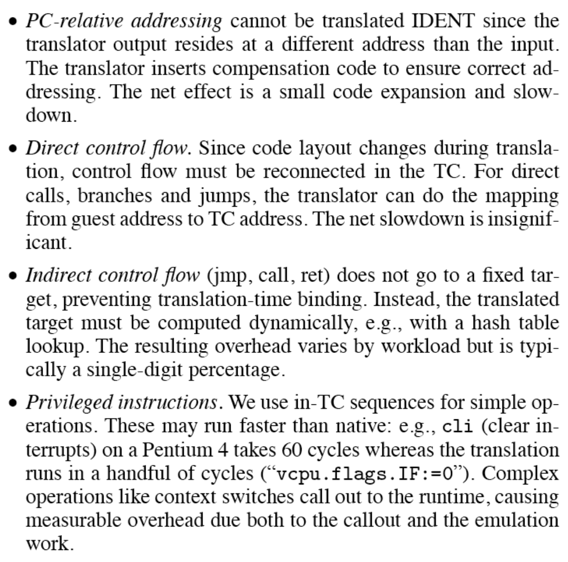
```
isPrime':  *mov   %ecx, %edi    ; IDENT
            mov   %esi, $2
            cmp   %esi, %ecx
            jge   [takenAddr]   ; JCC
                                ; fall-thru into next CCF
nexti':    *mov   %eax, %ecx    ; IDENT
            cdq
            idiv  %esi
            test  %edx, %edx
            jz    notPrime'     ; JCC
                                ; fall-thru into next CCF
           *inc   %esi          ; IDENT
            cmp   %esi, %ecx
            jl    nexti'        ; JCC
            jmp   [fallthrAddr3]

notPrime': *xor   %eax, %eax    ; IDENT
            pop   %r11          ; RET
            mov   %gs:0xff39eb8(%rip), %rcx   ; spill %rcx
            movzx %ecx, %r11b
            jmp   %gs:0xfc7dde0(8*%rcx)
```

Translation: Observations

- This approach scales well:
  - e.g., Windows XP boot/halt translates
    - 229,347 64-bit TUs
    - 23,909 32-bit TUs
    - 6,680 16-bit TUs
- Translator captures execution trace of guest code.
  - This is good for instruction-cache locality
  - Rarely-executed code (e.g. error handling) is placed off the "hot" execution path.

Most instructions are translated IDENT, except



- *PC-relative addressing* cannot be translated IDENT since the translator output resides at a different address than the input. The translator inserts compensation code to ensure correct addressing. The net effect is a small code expansion and slowdown.
- *Direct control flow.* Since code layout changes during translation, control flow must be reconnected in the TC. For direct calls, branches and jumps, the translator can do the mapping from guest address to TC address. The net slowdown is insignificant.
- *Indirect control flow* (jmp, call, ret) does not go to a fixed target, preventing translation-time binding. Instead, the translated target must be computed dynamically, e.g., with a hash table lookup. The resulting overhead varies by workload but is typically a single-digit percentage.
- *Privileged instructions.* We use in-TC sequences for simple operations. These may run faster than native: e.g., `cli` (clear interrupts) on a Pentium 4 takes 60 cycles whereas the translation runs in a handful of cycles ("`vcpu.flags.IF:=0`"). Complex operations like context switches call out to the runtime, causing measurable overhead due both to the callout and the emulation work.

Binary Translation of User-Level Code?

- "BT is not required for safe execution of most user code on most guest operating systems."
- Switch between BT and direct execution:
  - Use direct execution of guest in user-mode
  - Use BT for guest in kernel-mode
- This permits application to run at native speed.

Adaptive Binary Translation

- Q: How to deal with traps generated by non-privileged instructions accessing sensitive data (e.g. page table)?
- A: Monitor traps, and adapt translation:
  - retranslate non-IDENT to avoid trap (e.g. call interpreter)
  - patch original IDENT with jump to new translation

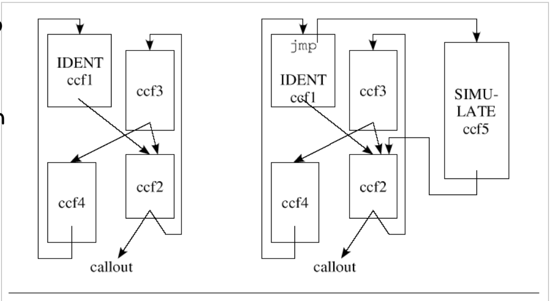


Figure 1. Adaptation from IDENT to SIMULATE.

Compare BT to Hardware Virtualization

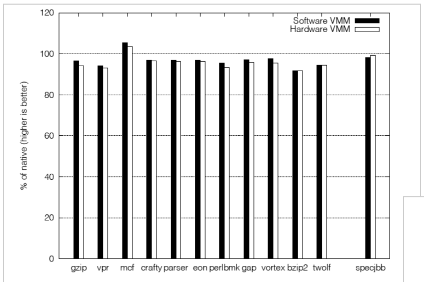


Figure 2. SPECint 2000 and SPECjbb 2005.

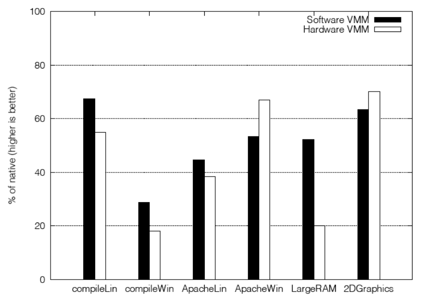


Figure 3. Macrobenchmarks.