CS152/252 Spring 2019
Computer Architecture and Engineering
Graduate Computer Architecture

Lecture 21:
Virtual Machines

Lisa Wu, Krste Asanovic

http://inst.eecs.berkeley.edu/~cs152/sp19

OUTLINE

- Why do we need virtual machines?
- What are virtual machines?
- Types of virtual machines: User-level and System-level
- Techniques for implementing all or parts of a non-native ISA on a host machine:
  - Interpreter
  - Static binary translation
  - Dynamic binary translation
  - Hardware emulation

OUTLINE

- Why do we need virtual machines?
- What are virtual machines?
- Types of virtual machines: User-level and System-level
- Techniques for implementing all or parts of a non-native ISA on a host machine:
  - Interpreter
  - Static binary translation
  - Dynamic binary translation
  - Hardware emulation

A Computer System Abstraction

APPLICATIONS

OS

HARDWARE

APPLICATIONS

APPLICATIONS

APPLICATIONS

APPLICATIONS

OS

OS

HARDWARE

HARDWARE

IBM 360: Standardized Interface between Hardware and Software

APPLICATIONS

OS

HARDWARE

Instruction Set Architecture (ISA)

Advantages of Standardizing Interfaces

Major system design tasks are decoupled: hardware and software designers can work independently

Software can run on different hardware platforms implementing the same ISA across hardware generations

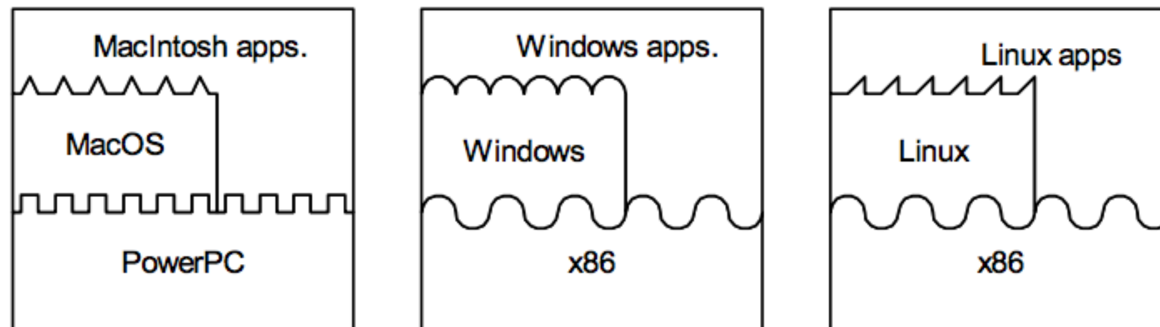
Disadvantages of Standardizing Interfaces

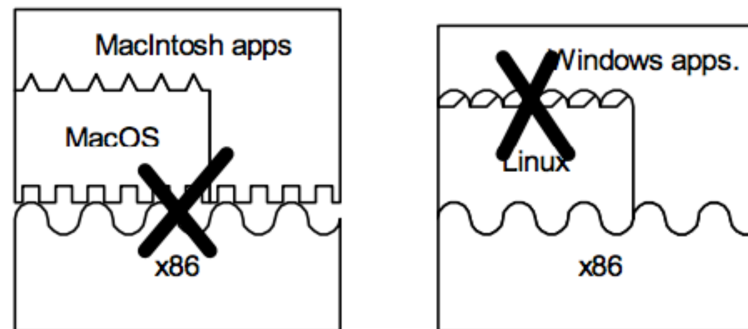
Major system components work together only in the proper combinations; problems worsen in networked environment

Innovation is sometimes limited by the need to support backward compatibility

Optimizations across major interfaces is difficult

Examples Illustrating Interoperability





"Virtual Machines: Architectures, Implementations and Applications", J. E. Smith and Ravi Nair, Morgan Kaufmann Publishers, 2004

Figure 0-2 Interoperability
a) Three popular computer systems composed of different ISAs, OSes, and Applications.
b) Although highly modular, the major system components are not always interoperable.

OUTLINE

- Why do we need virtual machines?
- What are virtual machines?
- Types of virtual machines: User-level and System-level
- Techniques for implementing all or parts of a non-native ISA on a host machine:
  - Interpreter
  - Static binary translation
  - Dynamic binary translation
  - Hardware emulation

What are Virtual Machines (VMs)?

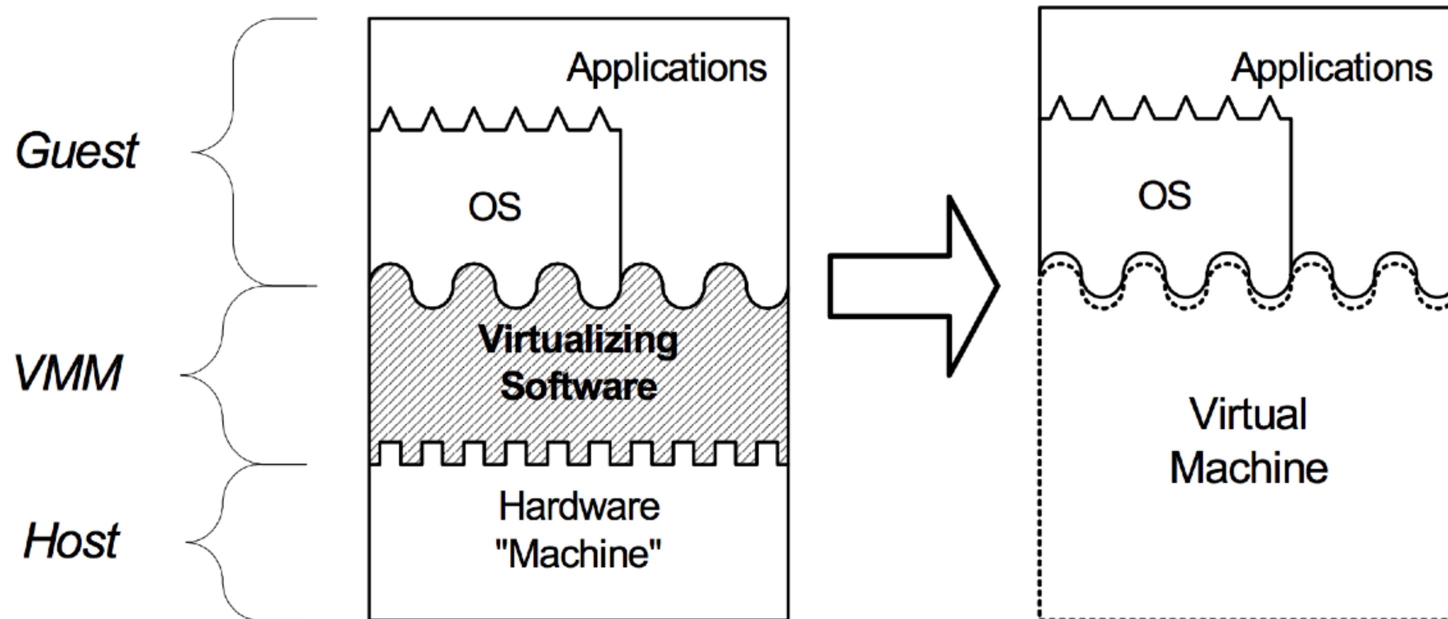
"Virtual Machines (VMs) eliminate real machine constraints and enable a much higher degree of **portability** and **flexibility**."

"A virtual machine is implemented by adding software to an execution platform to give it **the appearance of a different platform** or for that matter, to give the appearance of multiple platforms."

"A virtual machine may have an operating system, instruction set, or both, that **differ** from those implemented on the underlying real hardware."

"Virtual Machines: Architectures, Implementations and Applications", J. E. Smith and Ravi Nair, Morgan Kaufmann Publishers, 2004

Virtual Machine Basics



"Virtual Machines: Architectures, Implementations and Applications",
J. E. Smith and Ravi Nair, Morgan Kaufmann Publishers, 2004

Example Use Cases of VMs

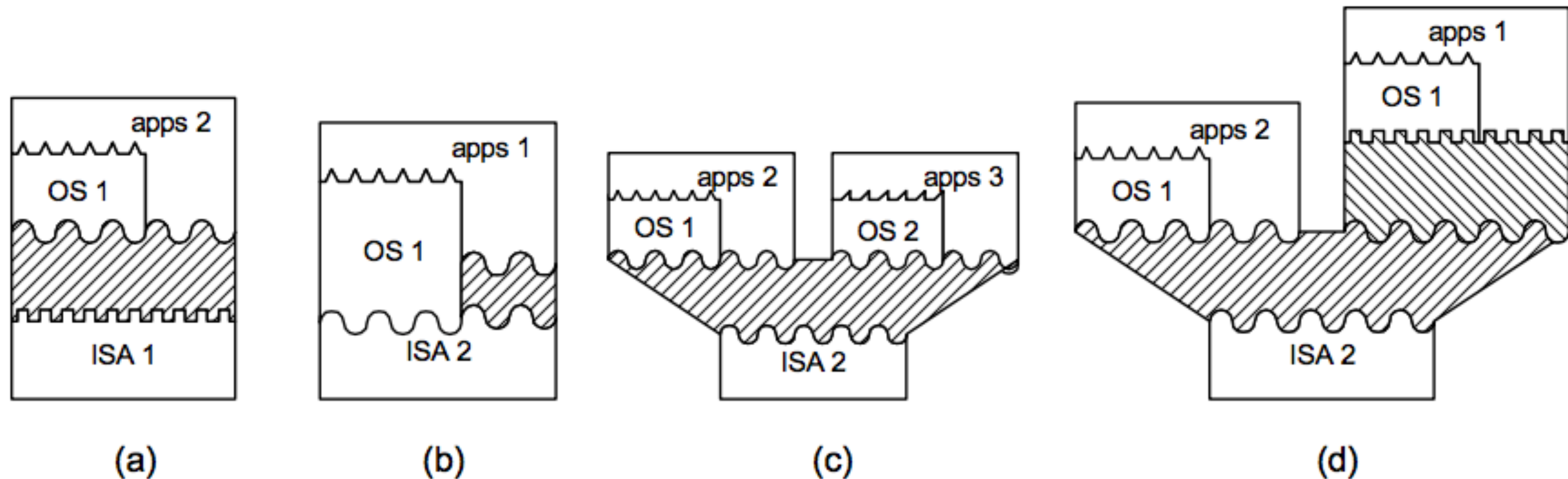


Figure 0-4. Examples of virtual machine applications.

“Virtual Machines: Architectures, Implementations and Applications”, J. E. Smith and Ravi Nair, Morgan Kaufmann Publishers, 2004

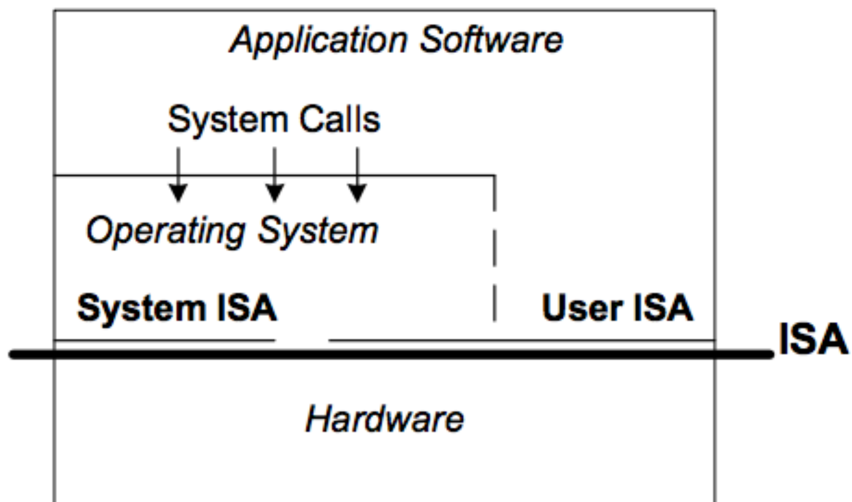
a) emulating one instruction set with another
b) optimizing an existing application binary for the same instruction set,
c) replicating a virtual machine so that multiple OSes can be supported simultaneously,
d) composing virtual machine software to form a more complex, flexible system.

OUTLINE

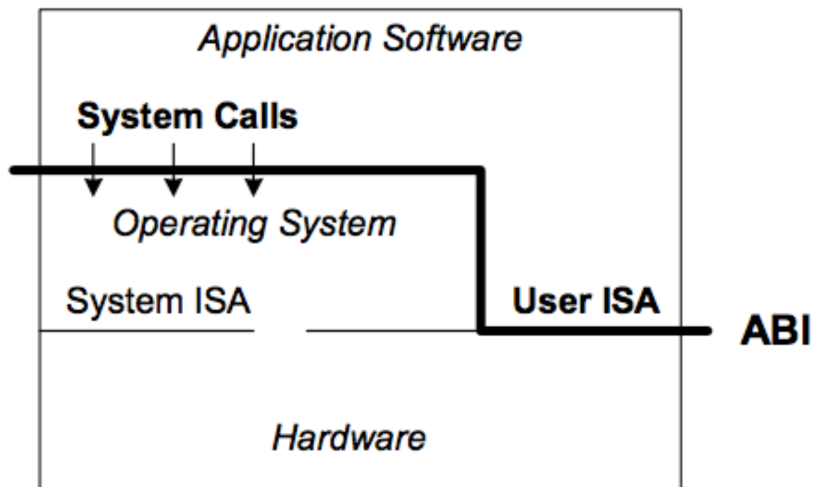
- Why do we need virtual machines?
- What are virtual machines?
- Types of virtual machines: User-level and System-level
- Techniques for implementing all or parts of a non-native ISA on a host machine:
  - Interpreter
  - Static binary translation
  - Dynamic binary translation
  - Hardware emulation

System Interfaces

“Virtual Machines: Architectures, Implementations and Applications”, J. E. Smith and Ravi Nair, Morgan Kaufmann Publishers, 2004



(a)



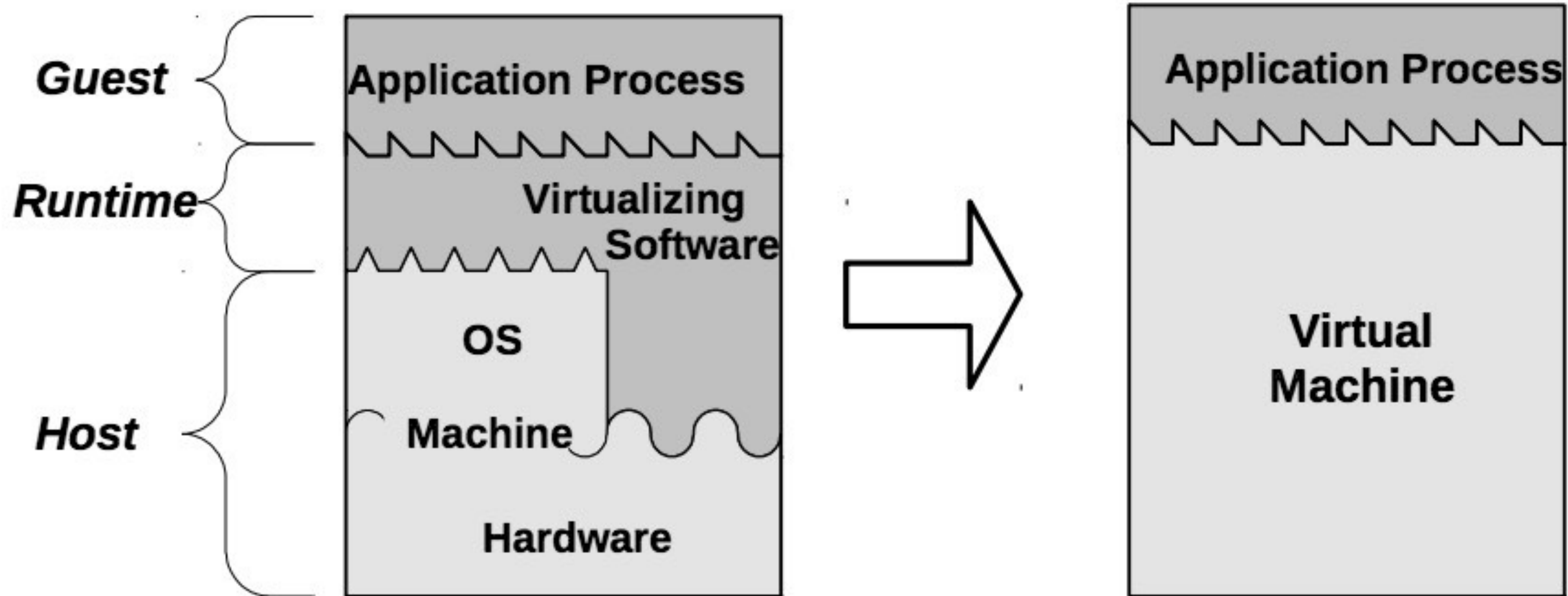
(b)

Types of Virtual Machine (VM)

- **User/Process Virtual Machines** run a single application according to some standard application binary interface (ABI).
  - Example user ABIs include Win32 for windows and Java Virtual Machine (JVM)
- **(Operating) System Virtual Machines** provide a complete system-level environment
  - E.g., IBM VM/370, VMware ESX Server, and Xen
  - Single computer runs multiple VMs, and can support a multiple, different OSes
    - On conventional platform, single OS "owns" all HW resources
    - With a VM, multiple OSes all share HW resources
- Underlying HW platform is called the host, where its resources used to run guest VMs (user and/or system)

© Krste Asanovic, 2015

User(Process) Virtual Machines



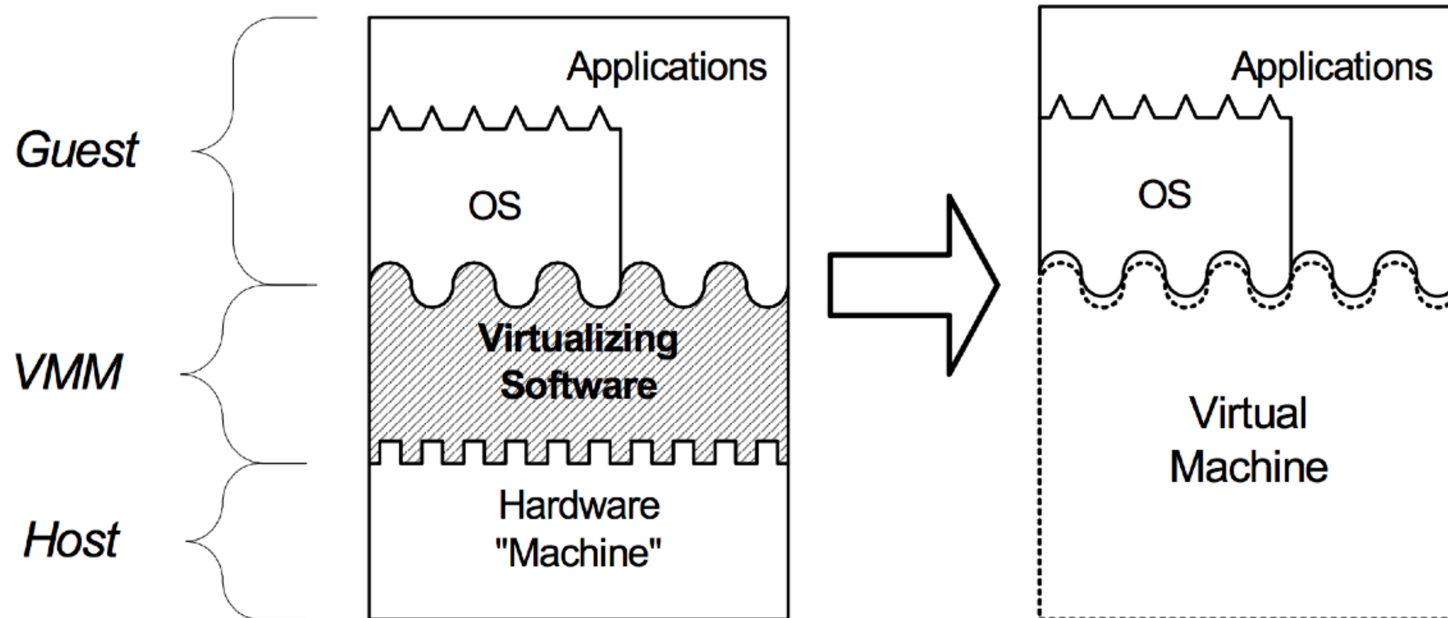
http://www.ittc.ku.edu/~kulkarni/teaching/EECS768/slides/chapter1.pdf
"Virtual Machines: Architectures, Implementations and Applications",
J. E. Smith and Ravi Nair, Morgan Kaufmann Publishers, 2004

Types of Virtual Machine (VM)

- **User/Process Virtual Machines** run a single application according to some standard application binary interface (ABI).
  - Example user ABIs include Win32 for windows and Java Virtual Machine (JVM)
- **(Operating) System Virtual Machines** provide a complete system-level environment
  - E.g., IBM VM/370, VMware ESX Server, and Xen
  - Single computer runs multiple VMs, and can support a multiple, different OSes
    - On conventional platform, single OS "owns" all HW resources
    - With a VM, multiple OSes all share HW resources
- Underlying HW platform is called the host, where its resources used to run guest VMs (user and/or system)

© Krste Asanovic, 2015

System Virtual Machines



"Virtual Machines: Architectures, Implementations and Applications", J. E. Smith and Ravi Nair, Morgan Kaufmann Publishers, 2004

Application Binary Interface (ABI)

- The ABI is a specification of the binary format used to encode programs for a virtual machine, encodes program text (instructions) and initial values of some data segments
- Virtual machine specifications include
  - what state is available at process creation
  - which instructions are available (the ISA)
  - what system calls are possible (I/O, or the environment)
- Operating system implements the virtual machine
  - at process startup, OS reads the binary program, creates an environment for it, then begins to execute the code, handling traps for I/O calls, emulation, etc.

© Krste Asanovic, 2015

User Virtual Machine = ISA + Environment

ISA alone not sufficient to write useful programs, need I/O too!

- Direct access to memory mapped I/O via load/store instructions problematic
  - time-shared systems
  - portability
- Operating system usually responsible for I/O
  - sharing devices and managing security
  - hiding different types of hardware (e.g., EIDE vs. SCSI disks)
- ISA communicates with operating system through some standard mechanism, i.e., **`syscall`** instructions
  - example convention to open file:

```
addi x1, x0, 27        # 27 is code for file open
addu x2, x0, rfname    # x2 points to filename
  string
syscall                # cause trap into OS
# On return from syscall, x1 holds file descriptor
```


© Krste Asanovic, 2015

OS Can Support Multiple User VMs

- Virtual machine features change over time with new versions of operating system
  - new ISA instructions added
  - new types of I/O are added (e.g., asynchronous file I/O)
- Common to provide backwards compatibility so old binaries run on new OS
  - SunOS 5 (System V Release 4 Unix, Solaris) can run binaries compiled for SunOS4 (BSD-style Unix)
  - Windows 98 runs MS-DOS programs
- If ABI needs instructions not supported by native hardware, OS can provide in software

© Krste Asanovic, 2015

ISA Implementations Partly in Software

- Often good idea to implement part of ISA in software:
- Expensive but rarely used instructions can cause trap to OS emulation routine:
  - e.g., decimal arithmetic instructions in MicroVax implementation of VAX ISA
- Infrequent but difficult operand values can cause trap
  - e.g., IEEE floating-point denormals cause traps in many floating-point unit implementations
- Old machine can trap unused opcodes, allows binaries for new ISA to run on old hardware
  - e.g., Sun SPARC v8 added integer multiply instructions, older v7 CPUs trap and emulate

© Krste Asanovic, 2015

OUTLINE

- Why do we need virtual machines?
- What are virtual machines?
- Types of virtual machines: User-level and System-level
- Techniques for implementing all or parts of a non-native ISA on a host machine:
  - Interpreter
  - Static binary translation
  - Dynamic binary translation
  - Hardware emulation

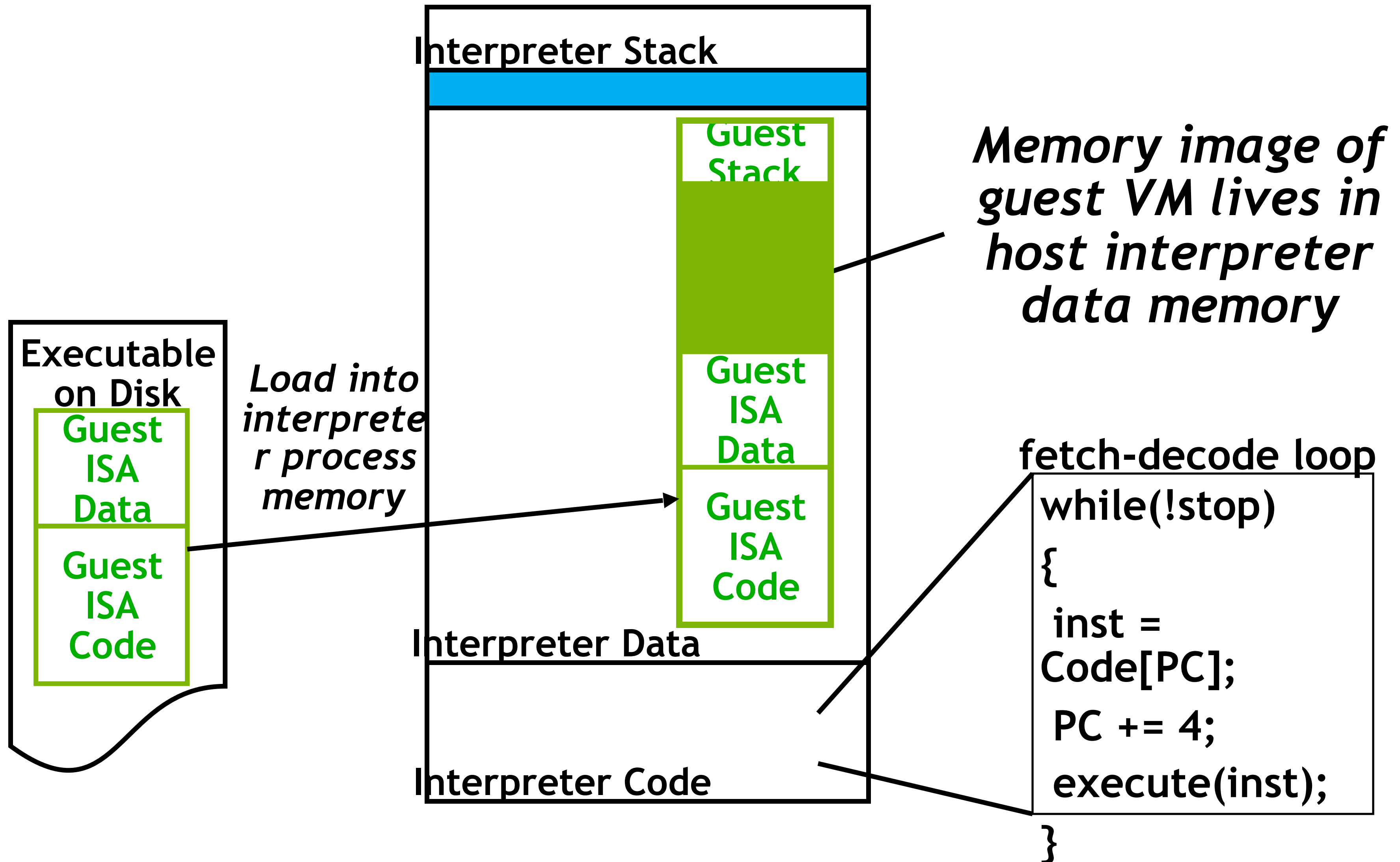
Supporting Non-Native ISAs

- Run programs for one ISA on hardware with different ISA
- Software Interpreter (OS software interprets instructions at run-time)
  - E.g., OS 9 for PowerPC Macs had interpreter for 68000 code
- Binary Translation (convert at install and/or load time)
  - IBM AS/400 to modified PowerPC cores
  - DEC tools for VAX->MIPS->Alpha
- Dynamic Translation (non-native ISA to native ISA at run time)
  - Sun's HotSpot Java JIT (just-in-time) compiler
  - Transmeta Crusoe, x86->VLIW code morphing
  - OS X for Intel Macs has dynamic binary translator for PowerPC (Rosetta)
- Run-time Hardware Emulation
  - IBM 360 had optional IBM 1401 emulator in microcode
  - Intel Itanium converts x86 to native VLIW (two software-visible ISAs)
  - ARM cores support 32-bit ARM, 16-bit Thumb, and JVM (three software-visible ISAs!)

© Krste Asanovic, 2015

Software Interpreter

- Fetch and decode one instruction at a time in software



© Krste Asanovic, 2015

Software Interpreter

- Easy to code, small code footprint
- *Slow*, approximately 100x slower than native execution for RISC ISA hosted on RISC ISA
- Problem is time taken to decode instructions
  - fetch instruction from memory
  - switch tables to decode opcodes
  - extract register specifiers using bit shifts
  - access register file data structure
  - execute operation
  - return to main fetch loop

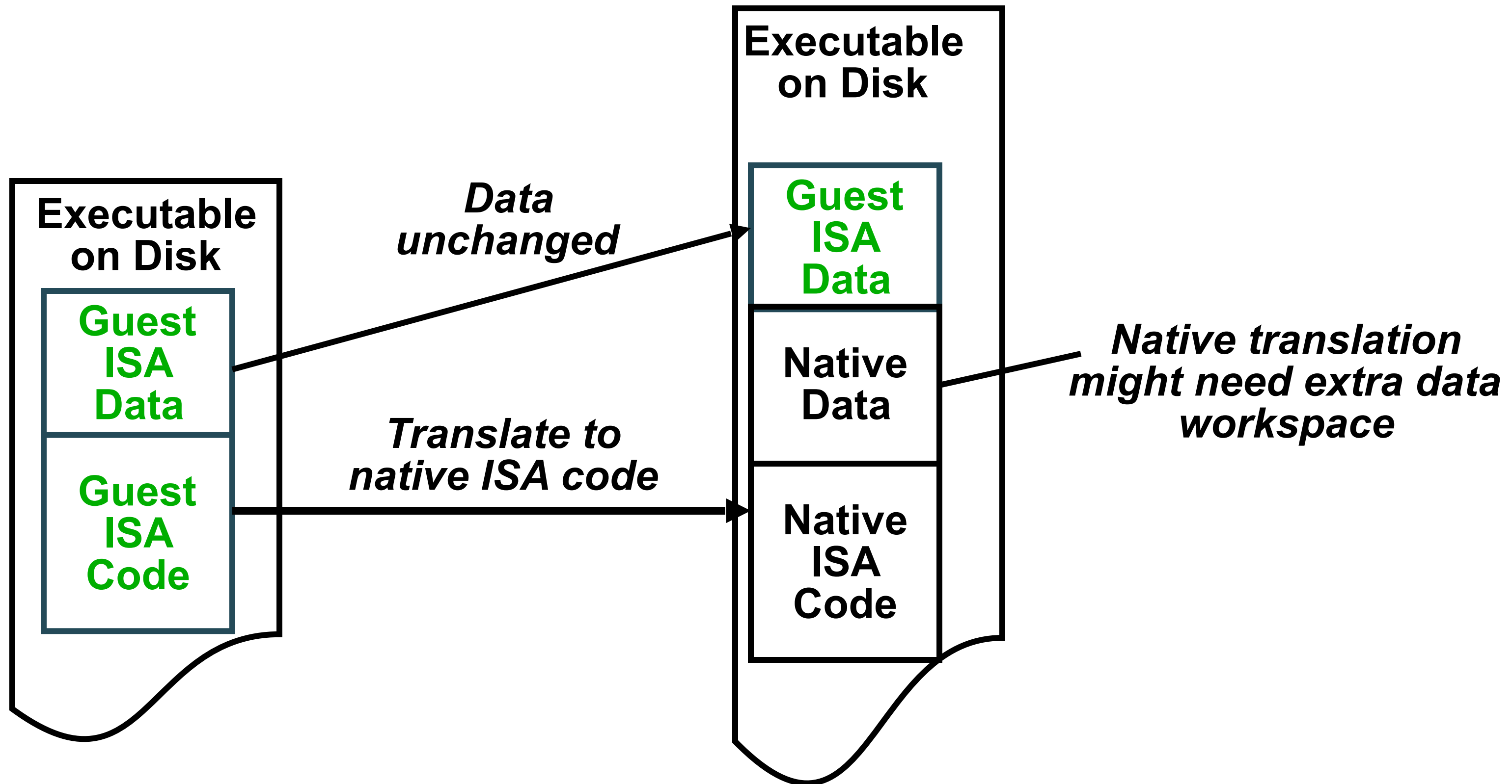
© Krste Asanovic, 2015

Binary Translation

- Each guest ISA instruction translates into some set of host (or *native*) ISA instructions
- Instead of dynamically fetching and decoding instructions at run-time, translate entire binary program and save result as new native ISA executable
- Removes interpretive fetch-decode overhead
- Can do compiler optimizations on translated code to improve performance
  - register allocation for values flowing between guest ISA instructions
  - native instruction scheduling to improve performance
  - remove unreachable code
  - inline assembly procedures

© Krste Asanovic, 2015

Binary Translation, Take 1



© Krste Asanovic, 2015

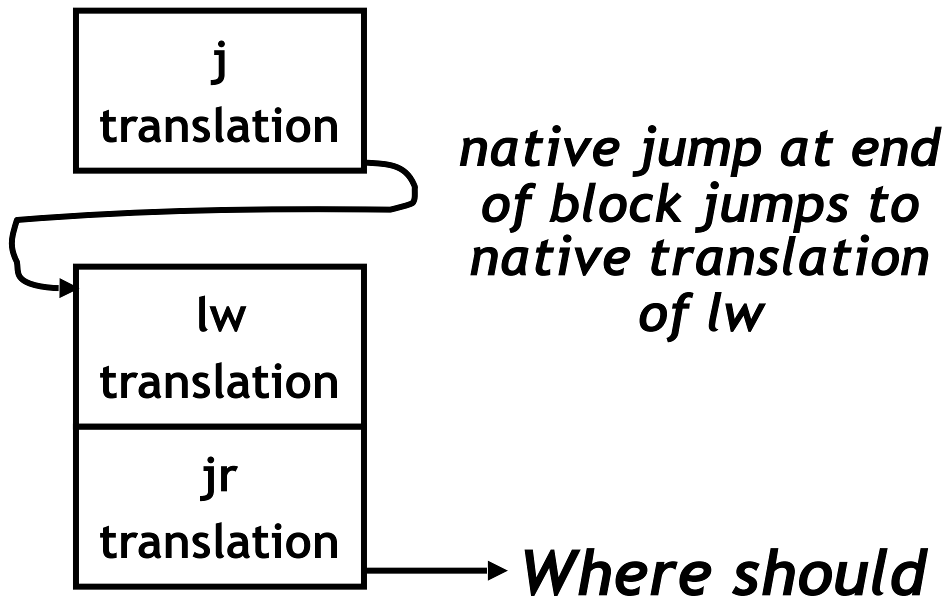
Binary Translation Problems

Branch and Jump targets

- guest code:

```
        j L1
     . . .
L1: lw r1, (r4)
    jr (r1)
```

- native code



native jump at end of block jumps to native translation of lw

Where should the jump register go?

© Krste Asanovic, 2015

PC Mapping Table

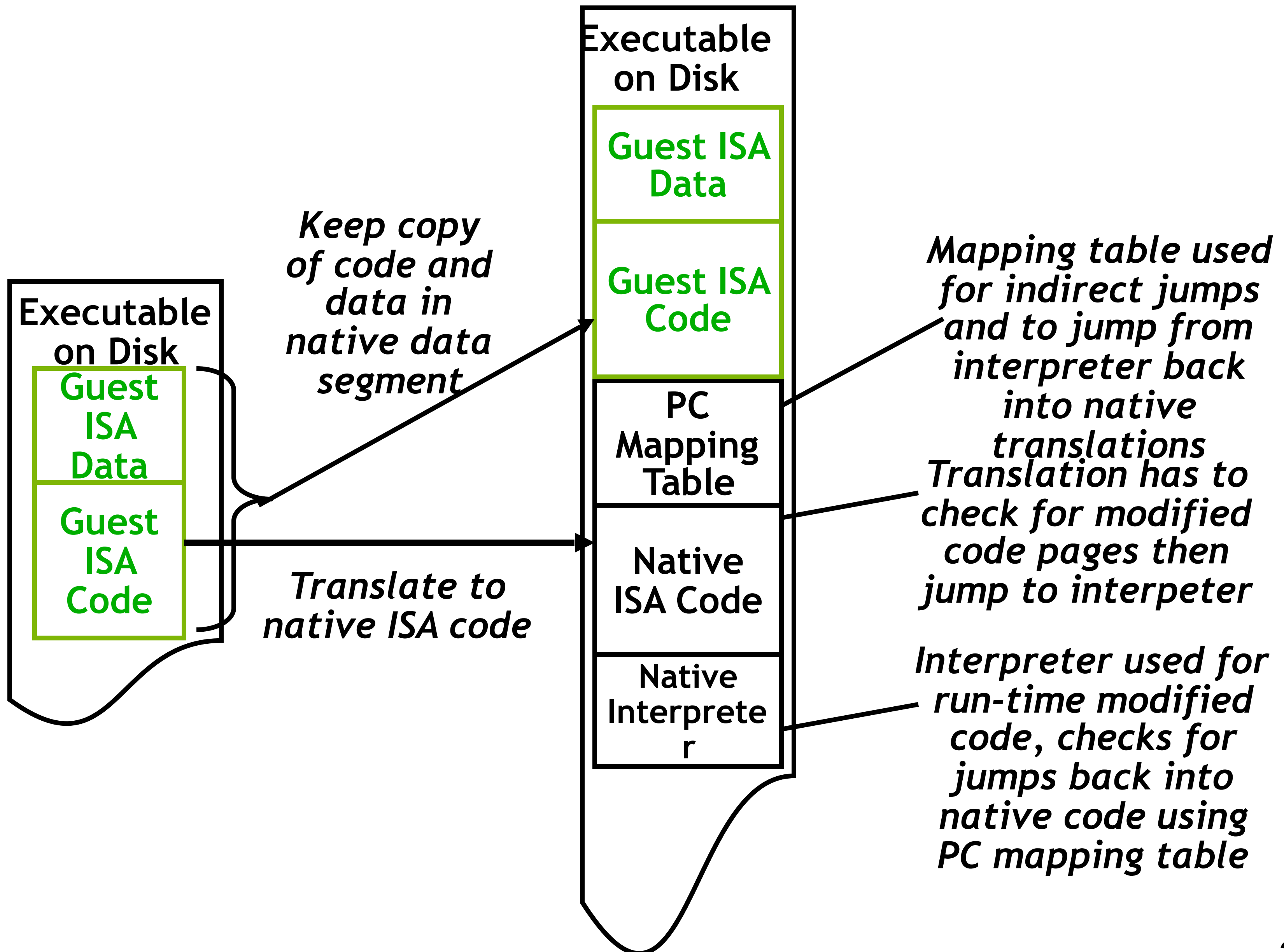
- Table gives translated PC for each guest PC
- Indirect jumps translated into code that looks in table to find where to jump to
  - can optimize well-behaved guest code for subroutine call/return by using native PC in return links
- If can branch to any guest PC, then need one table entry for every instruction in hosted program ➔ big table
- If can branch to any PC, then either
  - limit inter-instruction optimizations
  - large code explosion to hold optimizations for each possible entry into sequential code sequence
- Only minority of guest instructions are indirect jump targets, want to find these
  - design a highly structured VM design
  - use run-time feedback of target locations

© Krste Asanovic, 2015

Binary Translation Problems

- Self-modifying code!
  - `sw r1, (r2) # r2 points into code space`
- Rare in most code, but has to be handled if allowed by guest ISA
- Usually handled by including interpreter and marking modified code pages as “interpret only”
- Have to invalidate all native branches into modified code pages

Binary Translation, Take 2



© Krste Asanovic, 2015

Dynamic Translation

- Translate code sequences as needed at run-time, but cache results
- Can optimize code sequences based on dynamic information (e.g., branch targets encountered)
- Tradeoff between optimizer run-time and time saved by optimizations in translated code
- Technique used in Java JIT (Just-In-Time) compilers, and Virtual Machine Monitors (for system VMs)
- Also, Transmeta Crusoe for x86 emulation

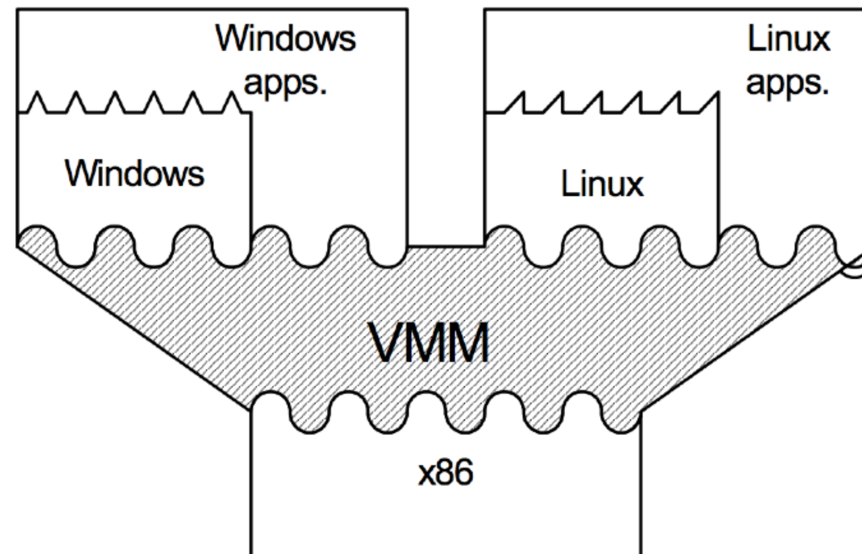
© Krste Asanovic, 2015

Introduction to System Virtual Machines

- VMs developed in late 1960s
  - Remained important in mainframe computing over the years
  - Largely ignored in single user computers of 1980s and 1990s
- Recently regained popularity due to
  - increasing importance of isolation and security in modern systems,
  - failures in security and reliability of standard operating systems,
  - sharing of a single computer among many unrelated users,
  - and the dramatic increases in raw speed of processors, which makes the overhead of VMs more acceptable

© Krste Asanovic, 2015

An Example System VM



- Supporting multiple OS environments on the same hardware

System VMs:
Supporting Multiple OSs on Same Hardware

- Can virtualize the environment that an operating system sees, an OS-level VM, or system VM
- Hypervisor layer implements sharing of real hardware resources by multiple OS VMs that each think they have a complete copy of the machine
  - Popular in early days to allow mainframe to be shared by multiple groups developing OS code
  - Used in modern mainframes to allow multiple versions of OS to be running simultaneously → OS upgrades with no downtime!
  - Example for PCs: VMware allows Windows OS to run on top of Linux (or vice-versa)

© Krste Asanovic, 2015

Virtual Machine Monitors (VMMs)

- Virtual machine monitor (VMM) or hypervisor is software that supports VMs
- VMM determines how to map virtual resources to physical resources
- Physical resource may be time-shared, partitioned, or emulated in software
- VMM is much smaller than a traditional OS;
  - isolation portion of a VMM is ≈ 10,000 lines of code

© Krste Asanovic, 2015

VMM Overhead?

- Depends on the workload
- User-level processor-bound programs (e.g., SPEC) have zero-virtualization overhead
  - Runs at native speeds since OS rarely invoked
- I/O-intensive workloads that are OS-intensive execute many system calls and privileged instructions, can result in high virtualization overhead
  - For System VMs, goal of architecture and VMM is to run almost all instructions directly on native hardware
- If I/O-intensive workload is also I/O-bound, low processor utilization since waiting for I/O
  - processor virtualization can be hidden, so low virtualization overhead

© Krste Asanovic, 2015

Requirements of a Virtual Machine Monitor

- A VM Monitor
  - Presents a SW interface to guest software,
  - Isolates state of guests from each other, and
  - Protects itself from guest software (including guest OSes)
- Guest software should behave on a VM exactly as if running on the native HW
  - Except for performance-related behavior or limitations of fixed resources shared by multiple VMs
- Guest software should not be able to change allocation of real system resources directly
- Hence, VMM must control ≈ everything even though guest VM and OS currently running is temporarily using them
  - Access to privileged state, Address translation, I/O, Exceptions and Interrupts, …

© Krste Asanovic, 2015

Requirements of a Virtual Machine Monitor

- VMM must be at higher privilege level than guest VM, which generally run in user mode
  $\Rightarrow$ Execution of privileged instructions handled by VMM
- E.g., Timer interrupt: VMM suspends currently running guest VM, saves its state, handles interrupt, determine which guest VM to run next, and then load its state
  - Guest VMs that rely on timer interrupt provided with virtual timer and an emulated timer interrupt by VMM
- Requirements of system virtual machines are same as paged-virtual memory:

1. At least 2 processor modes, system and user
2. Privileged subset of instructions available only in system mode, trap if executed in user mode
   - All system resources controllable only via these instructions

© Krste Asanovic, 2015

ISA Support for Virtual Machines

- If VMs are planned for during design of ISA, easy to reduce instructions that must be executed by a VMM and how long it takes to emulate them
  - Since VMs have been considered for desktop/PC server apps only recently, most ISAs were created without virtualization in mind, including 80x86 and most RISC architectures
- VMM must ensure that guest system only interacts with virtual resources ⇒ conventional guest OS runs as user mode program on top of VMM
  - If guest OS attempts to access or modify information related to HW resources via a privileged instruction--for example, reading or writing the page table pointer--it will trap to the VMM
- If not, VMM must intercept instruction and support a virtual version of the sensitive information as the guest OS expects

© Krste Asanovic, 2015

Impact of VMs on Virtual Memory

- Virtualization of virtual memory if each guest OS in every VM manages its own set of page tables?
- VMM separates real and physical memory
  - Makes real memory a separate, intermediate level between virtual memory and physical memory
  - Some use the terms virtual memory, physical memory, and machine memory to name the 3 levels
  - Guest OS maps virtual memory to real memory via its page tables, and VMM page tables map real memory to physical memory
  - In RISC-V, *user-virtual* addresses, plus *supervisor-physical*, *hypervisor-physical*, *machine-physical* addresses
- VMM maintains a shadow page table that maps directly from the guest virtual address space to the physical address space of HW
  - Rather than pay extra level of indirection on every memory access
  - VMM must trap any attempt by guest OS to change its page table or to access the page table pointer

© Krste Asanovic, 2015

ISA Support for VMs & Virtual Memory

- IBM 370 architecture added additional level of indirection that is managed by the VMM
  - Guest OS keeps its page tables as before, so the shadow pages are unnecessary
- To virtualize software TLB, VMM manages the real TLB and has a copy of the contents of the TLB of each guest VM
  - Any instruction that accesses the TLB must trap
  - TLBs with Process ID tags support a mix of entries from different VMs and the VMM, thereby avoiding flushing of the TLB on a VM switch
- Recent processor designs have added similar mechanisms to accelerate VMMs

© Krste Asanovic, 2015

Impact of Virtualization on I/O

- Most difficult part of virtualization
  - Increasing number of I/O devices attached to the computer
  - Increasing diversity of I/O device types
  - Sharing of a real device among multiple VMs,
  - Supporting the myriad of device drivers that are required, especially if different guest OSes are supported on the same VM system
- Give each VM generic versions of each type of I/O device driver, and let VMM handle real I/O

© Krste Asanovic, 2015

Impact of Virtualization on I/O

- Method for mapping virtual to physical I/O device depends on the type of device:
  - Disks partitioned by VMM to create virtual disks for guest VMs
  - Network interfaces shared between VMs in short time slices, and VMM tracks messages for virtual network addresses to ensure that guest VMs only receive their messages

© Krste Asanovic, 2015

Other Uses of VMs

1. Managing Software
   - VMs provide an abstraction that can run the complete SW stack, even including old OSes like DOS
   - Typical deployment: some VMs running legacy OSes, many running current stable OS release, few testing next OS release
2. Managing Hardware
   - VMs allow separate SW stacks to run independently yet share HW, thereby consolidating number of servers
     - Some run each application with compatible version of OS on separate computers, as separation helps dependability
   - Migrate running VM to a different computer
     - Either to balance load or to evacuate from failing HW

© Krste Asanovic, 2015

Extra Resource for Learning VM

- Recommended Reading:
  - “An Overview of Virtual Machine Architectures”, J. E. Smith and Ravi Nair, 21-page article
  - “Virtual Machines: Architectures, Implementations and Applications”, J. E. Smith and Ravi Nair, Textbook 2004

Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

© Krste Asanovic, 2015