



CS 152 Computer Architecture and Engineering

Lecture 22: Virtual Machines

Krste Asanovic

Electrical Engineering and Computer Sciences
University of California, Berkeley

<http://www.eecs.berkeley.edu/~krste>
<http://inst.eecs.berkeley.edu/~cs152>

April 22, 2010

CS152, Spring 2010



Outline

- Types of Virtual Machine
 - User-level
 - 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



Types of Virtual Machine (VM)

- User 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 at binary ISA
 - 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)



Software Applications

How is a software application encoded?

- What are you getting when you buy a software application?
- What machines will it work on?
- Who do you blame if it doesn't work, i.e., what contract(s) were violated?

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 r1, r0, 27          # 27 is code for file open
addu r2, r0, rfname     # r2 points to filename string
syscall                 # cause trap into OS
# On return from syscall, r1 holds file descriptor
```



Application Binary Interface (ABI)

- Programs are usually distributed in a binary format that encodes the 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*)
- The ABI is a specification of the binary format used to encode programs for a virtual machine
- 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.



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



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 almost all 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



Supporting Non-Native ISAs

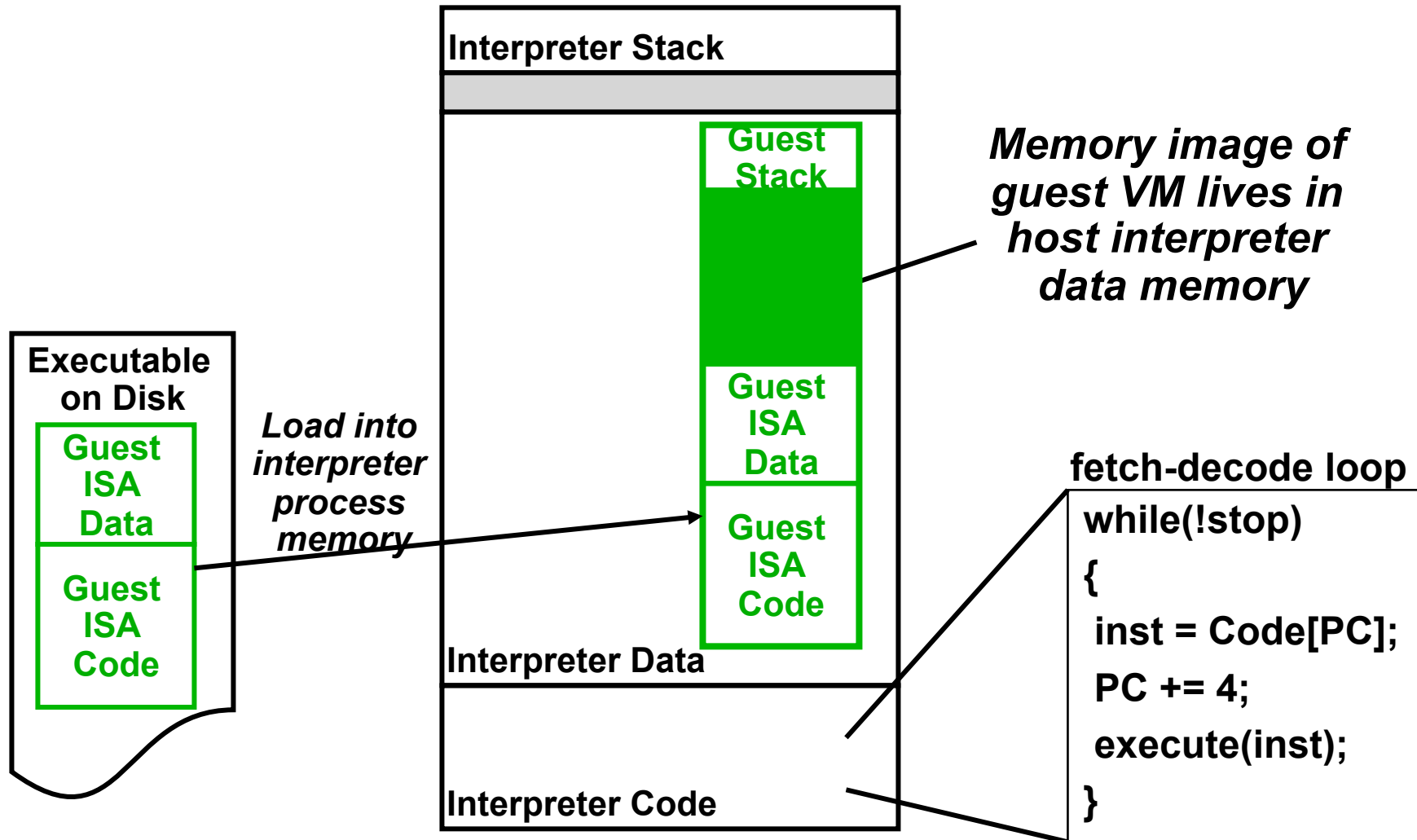
Run programs for one ISA on hardware with different ISA

- **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!)



Software Interpreter

- Fetch and decode one instruction at a time in software





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

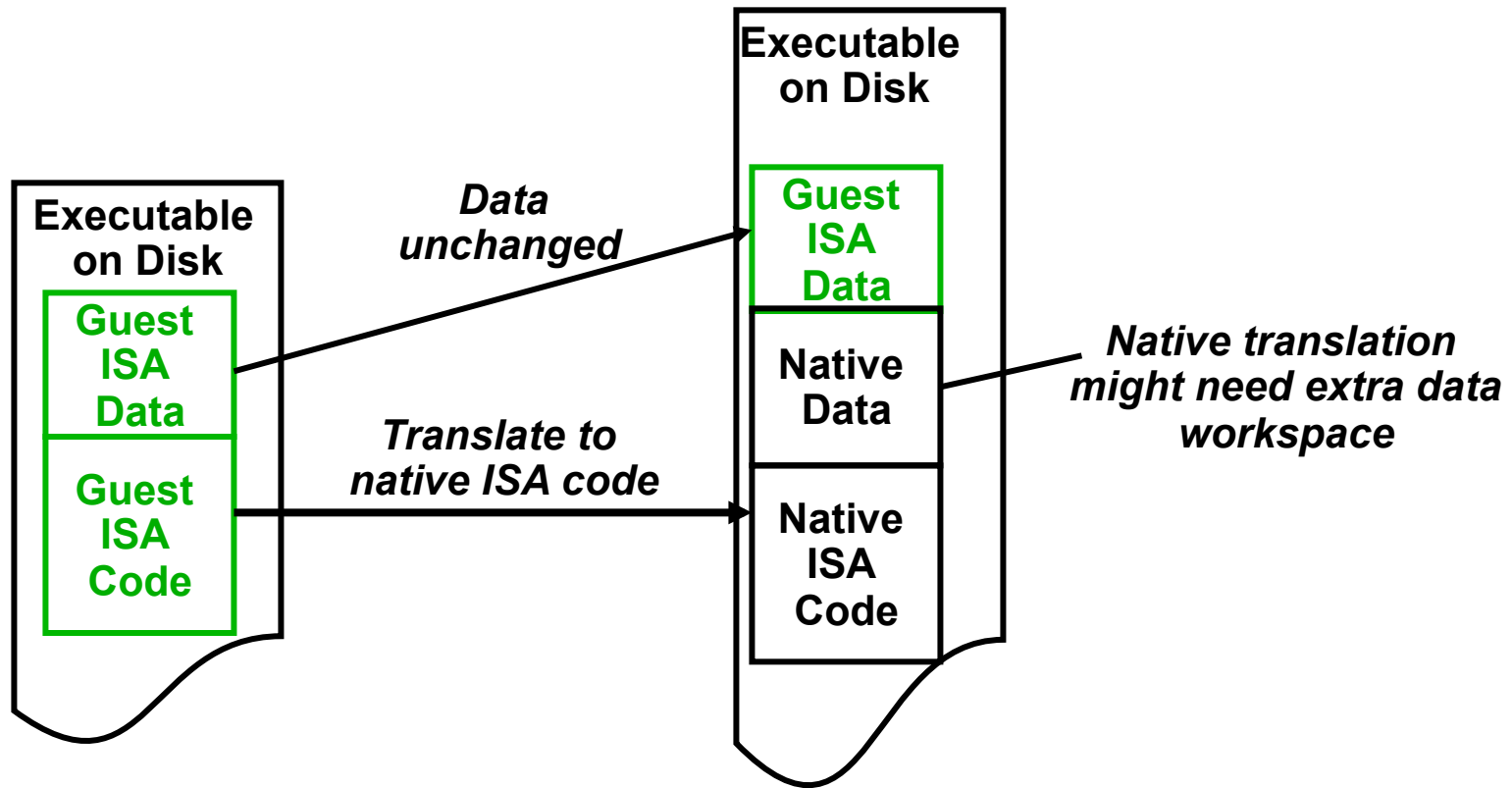


Binary Translation

- 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



Binary Translation, Take 1





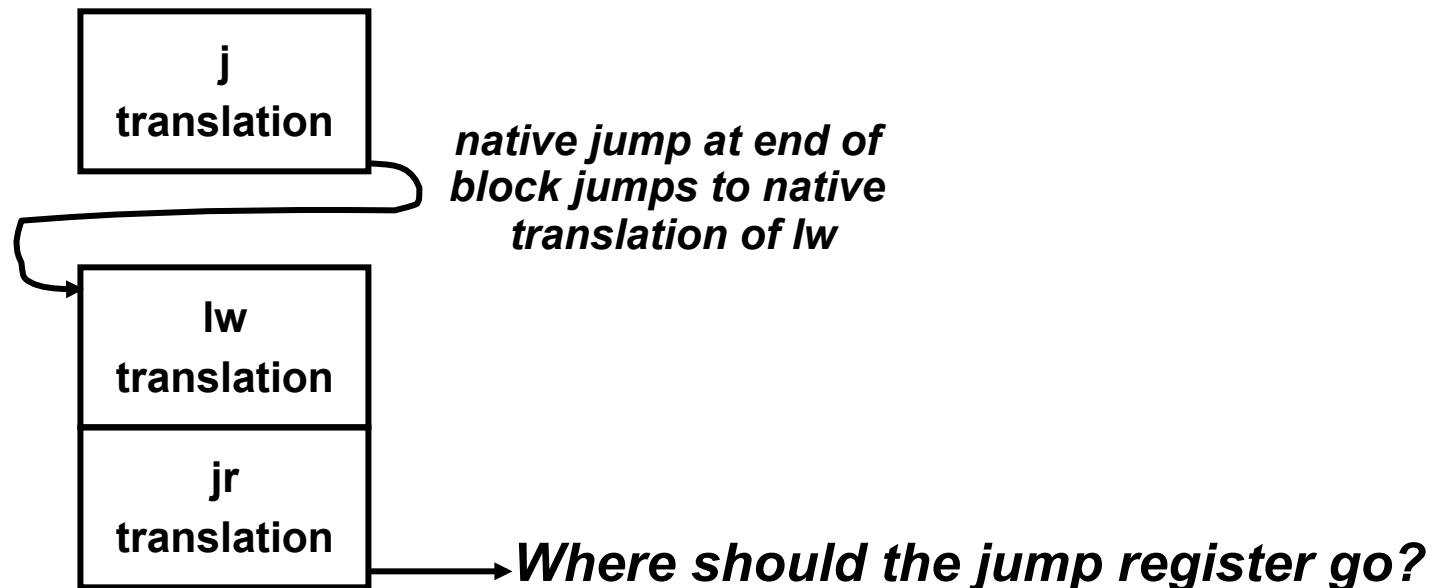
Binary Translation Problems

Branch and Jump targets

– guest code:

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

– native code





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

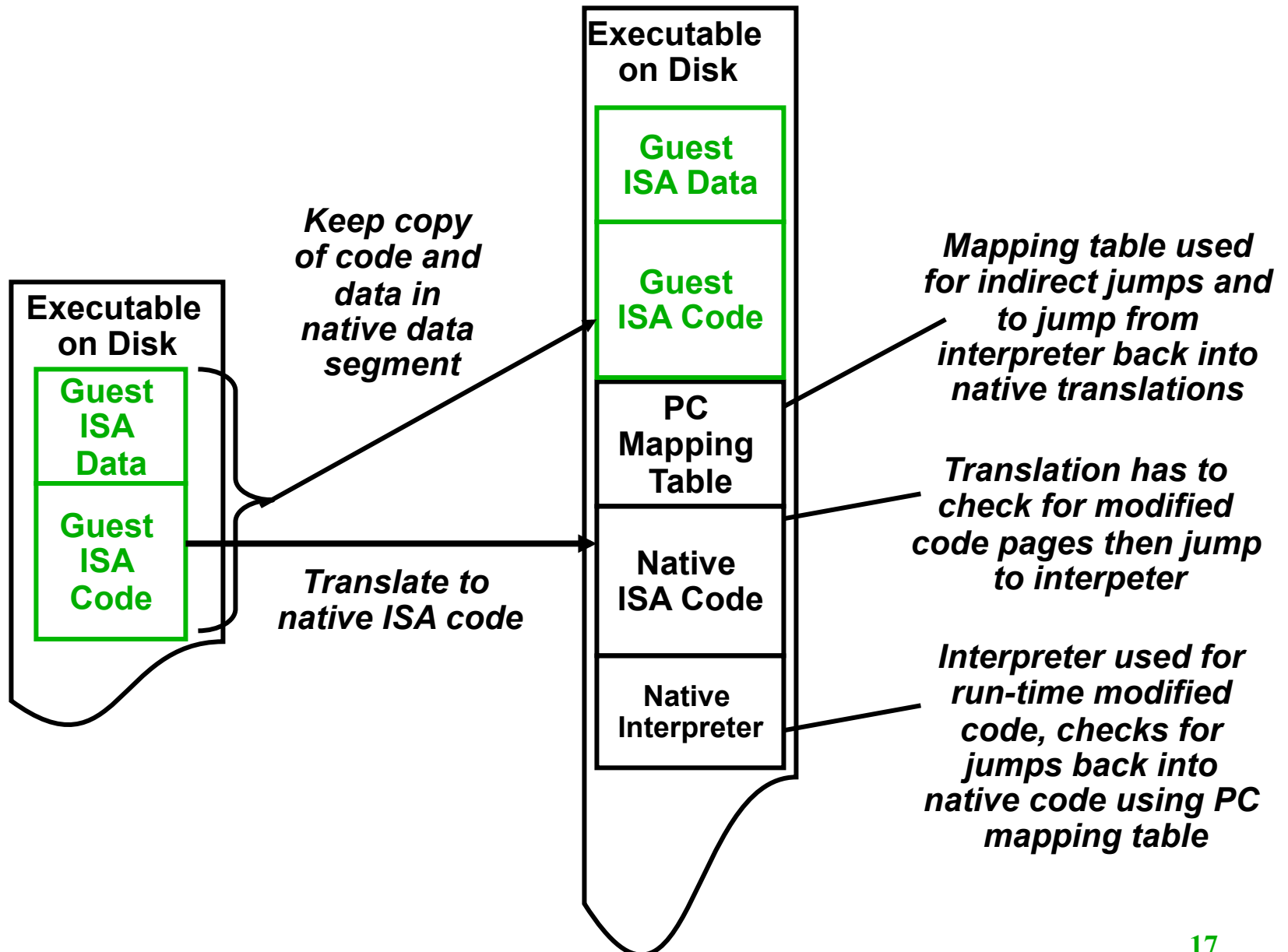


Binary Translation Problems

- 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



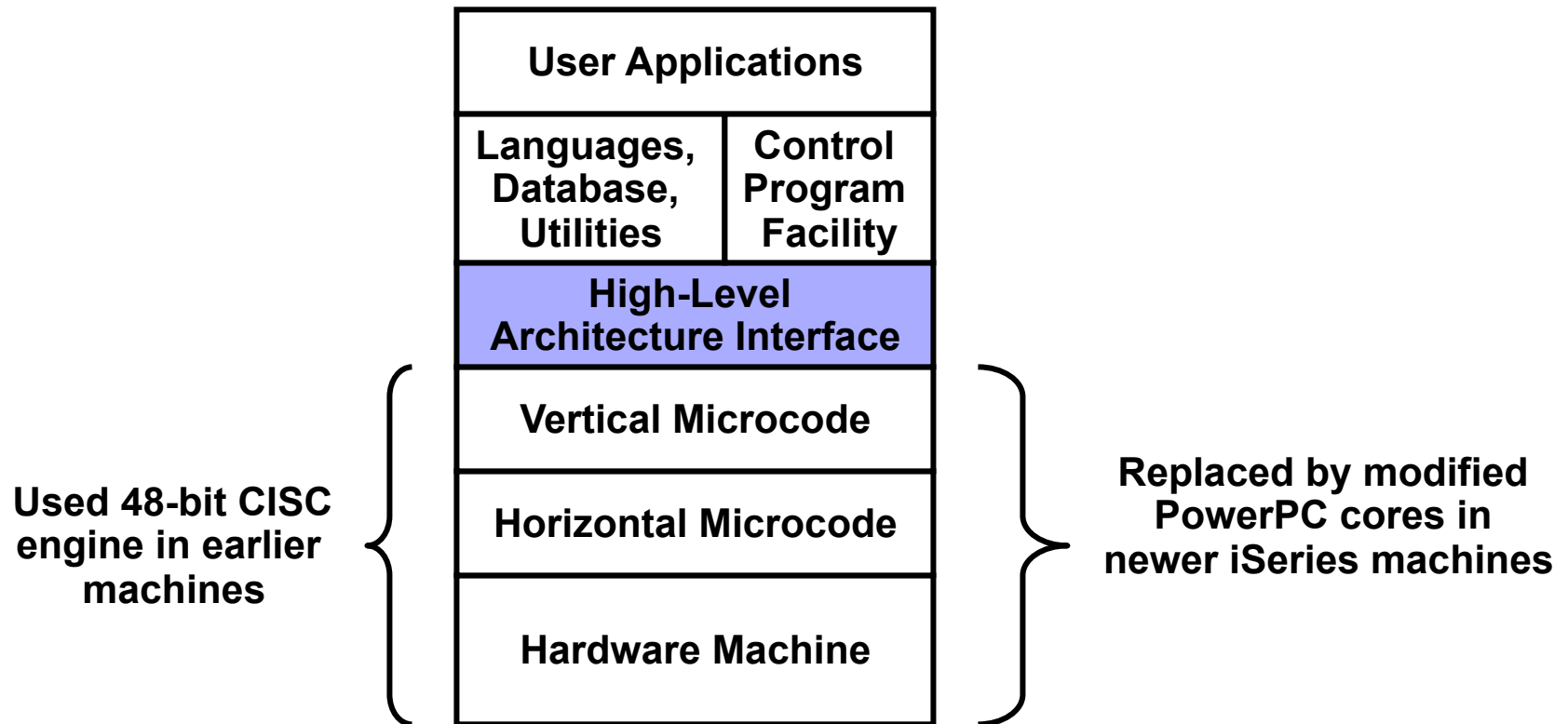
Binary Translation, Take 2





IBM System/38 and AS/400

- System/38 announced 1978
 - AS/400 is follow-on line, now called “System I” or “iSeries”
- High-level instruction set interface designed for binary translation
- Memory-memory instruction set, never directly executed by hardware



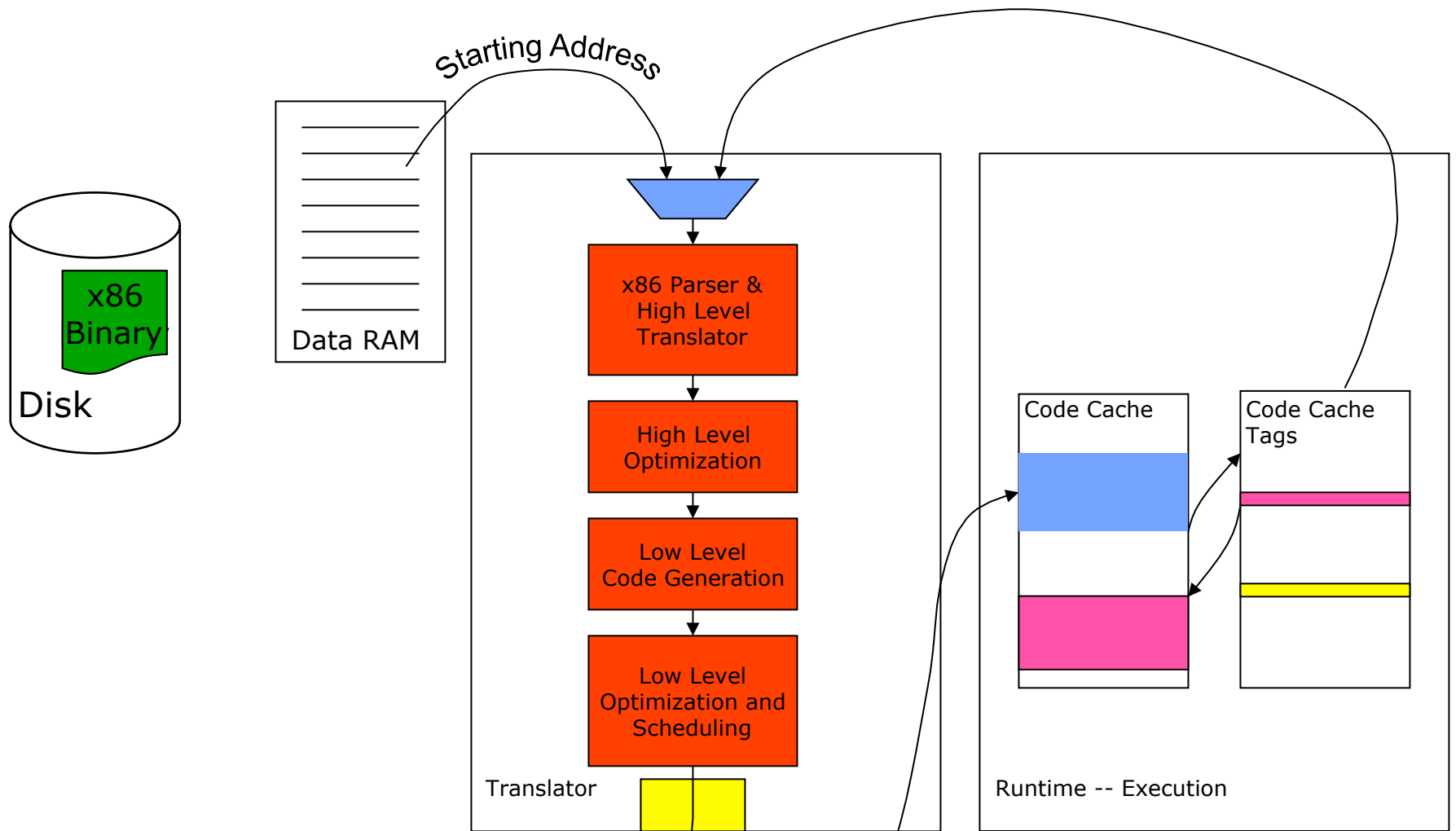


Dynamic Translation

- 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

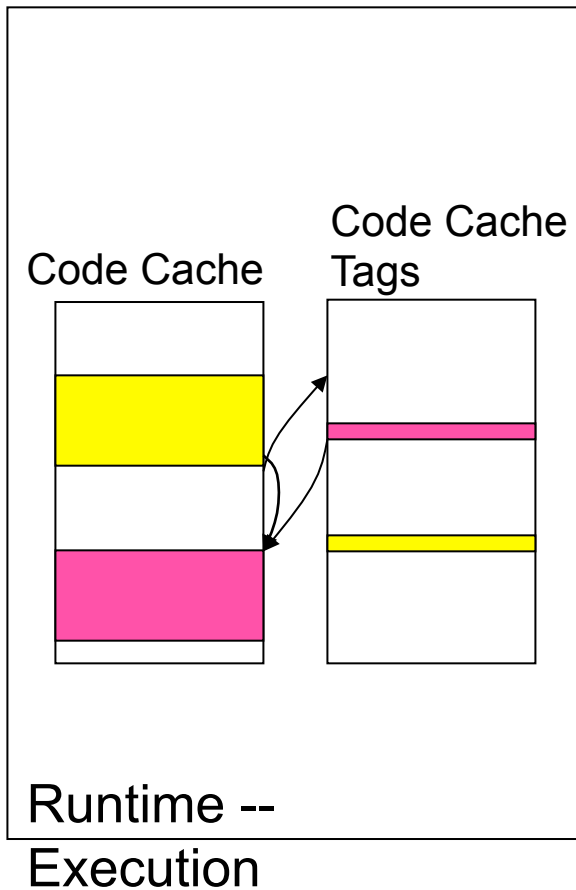


Dynamic Binary Translation Example:





Chaining



Pre Chained

```
add %r5, %r6, %r7
li %next_addr_reg, next_addr #load address
                                #of next block
j dispatch loop
```

Chained

```
add %r5, %r6, %r7
j physical location of translated
code for next_block
```



CS152 Administrivia

- Final quiz, Thursday April 29
 - Multiprocessors, Memory models, Cache coherence
 - Lectures 19-21, PS 5, Lab 5
- Last lecture, “Putting it all Together”, Tuesday April 27
 - Summary of the course
 - Case Study: Intel Nehalem
 - HKN Course Survey



Transmeta Crusoe (2000)

- Converts x86 ISA into internal native VLIW format using software at run-time → “Code Morphing”
- Optimizes across x86 instruction boundaries to improve performance
- Translations cached to avoid translator overhead on repeated execution
- Completely invisible to operating system – looks like x86 hardware processor

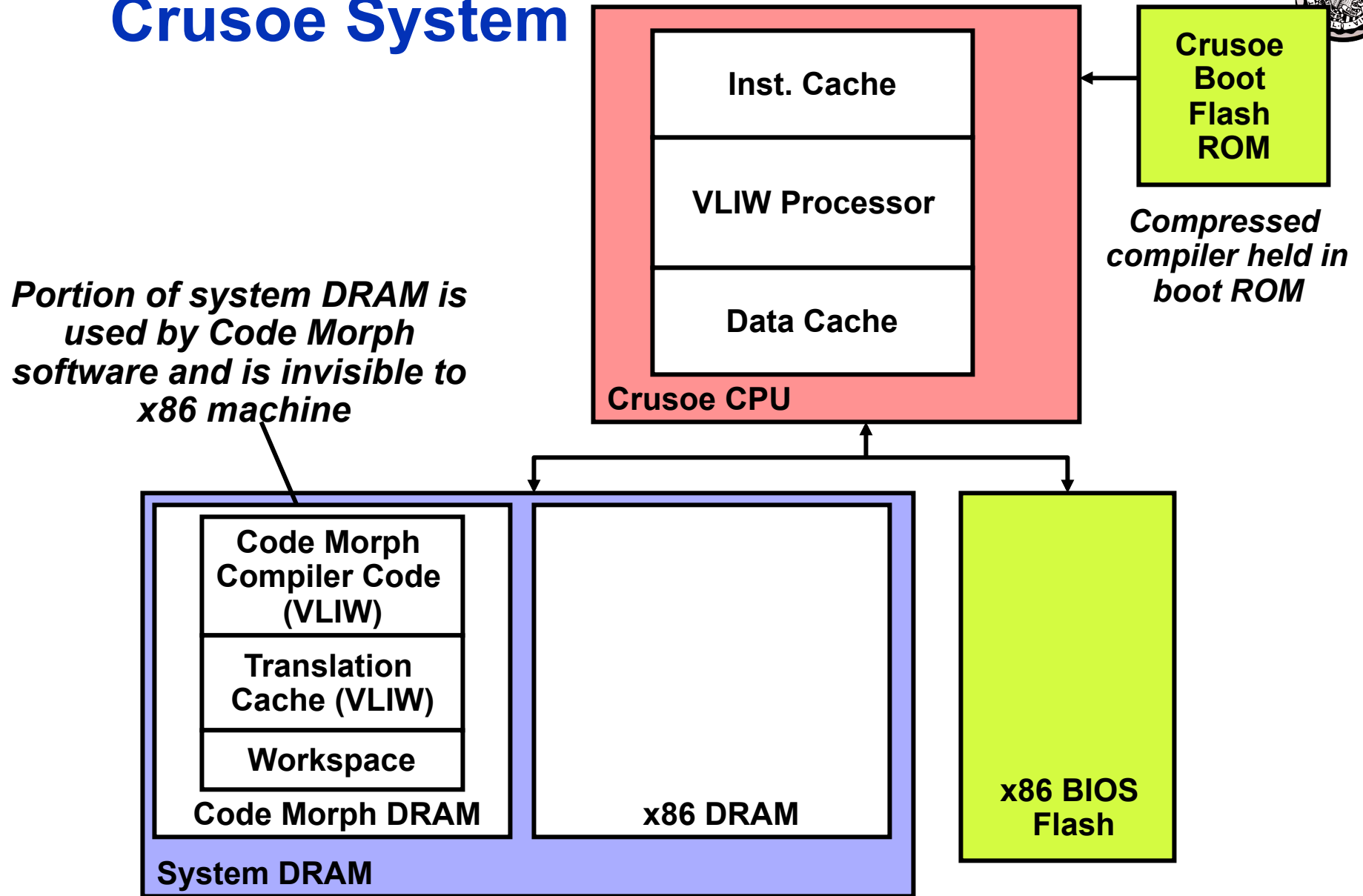
*[Following slides contain examples taken from
“The Technology Behind Crusoe Processors”,
Transmeta Corporation, 2000]*



Transmeta VLIW Engine

- Two VLIW formats, 64-bit and 128-bit, contains 2 or 4 RISC-like operations
- VLIW engine optimized for x86 code emulation
 - evaluates condition codes the same way as x86
 - has 80-bit floating-point unit
 - partial register writes (update 8 bits in 32 bit register)
- Support for fast instruction writes
 - run-time code generation important
- Initially, two different VLIW implementations, low-end TM3120, high-end TM5400
 - native ISA differences invisible to user, hidden by translation system
 - new eight-issue VLIW core planned (TM6000 series)

Crusoe System





Transmeta Translation

x86 code:

```
addl %eax, (%esp) # load data from stack, add to eax
addl %ebx, (%esp) # load data from stack, add to ebx
movl %esi, (%ebp) # load esi from memory
subl %ecx, 5      # sub 5 from ecx
```

first step, translate into RISC ops:

```
ld %r30, [%esp]      # load from stack into temp
add.c %eax, %eax, %r30 # add to %eax, set cond.codes
ld %r31, [%esp]
add.c %ebx, %ebx, %r31
ld %esi, [%ebp]
sub.c %ecx, %ecx, 5
```



Compiler Optimizations

RISC ops:

```
ld %r30, [%esp]           # load from stack into temp
add.c %eax, %eax, %r30    # add to %eax, set cond. codes
ld %r31, [%esp]
add.c %ebx, %ebx, %r31
ld %esi, [%ebp]
sub.c %ecx, %ecx, 5
```

Optimize:

```
ld %r30, [%esp]           # load from stack only once
add %eax, %eax, %r30
add %ebx, %ebx, %r30      # reuse data loaded earlier
ld %esi, [%ebp]
sub.c %ecx, %ecx, 5      # only this cond. code needed
```



Scheduling

Optimized RISC ops:

```
ld %r30, [%esp]           # load from stack only once
add %eax, %eax, %r30
add %ebx, %ebx, %r30      # reuse data loaded earlier
ld %esi, [%ebp]
sub.c %ecx, %ecx, 5       # only this cond. code needed
```

Schedule into VLIW code:

```
ld %r30, [%esp]; sub.c %ecx, %ecx, 5
ld %esi, [%ebp]; add %eax, %eax, %r30; add %ebx, %ebx, %r30
```



Translation Overhead

- Highly optimizing compiler takes considerable time to run, adds run-time overhead
- Only worth doing for frequently executed code
- Translation adds instrumentation into translations that counts how often code executed, and which way branches usually go
- As count for a block increases, higher optimization levels are invoked on that code



Exceptions

Original x86 code:

```
addl %eax, (%esp) # load data from stack, add to eax
addl %ebx, (%esp) # load data from stack, add to ebx
movl %esi, (%ebp) # load esi from memory
subl %ecx, 5      # sub 5 from ecx
```

Scheduled VLIW code:

```
ld %r30, [%esp]; sub.c %ecx, %ecx, 5
ld %esi, [%ebp]; add %eax, %eax, %r30; add %ebx, %ebx, %r30
```

- **x86 instructions executed out-of-order with respect to original program flow**
- **Need to restore state for precise traps**



Shadow Registers and Store Buffer

- All registers have working copy and shadow copy
- Stores held in software controlled store buffer, loads can snoop
- At end of translation block, commit changes by copying values from working regs to shadow regs, and by releasing stores in store buffer
- On exception, re-execute x86 code using interpreter



Handling Self-Modifying Code

- When a translation is made, mark the associated x86 code page as being translated in page table
- Store to translated code page causes trap, and associated translations are invalidated



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)
- Requires trap on access to privileged hardware state
 - easier if OS interface to hardware well defined



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



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 $\approx 10,000$ lines of code



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



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



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 \approx everything even though guest VM and OS currently running is temporarily using them
 - Access to privileged state, Address translation, I/O, Exceptions and Interrupts, ...



Requirements of a Virtual Machine Monitor

- VMM must be at higher privilege level than guest VM, which generally run in user mode
 - ⇒ 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



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 \Rightarrow 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 (examples soon)



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
- 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



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



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
- 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



Acknowledgements

- These slides contain material developed and copyright by:
 - Arvind (MIT)
 - Krste Asanovic (MIT/UCB)
 - Joel Emer (Intel/MIT)
 - James Hoe (CMU)
 - John Kubiatowicz (UCB)
 - David Patterson (UCB)
- MIT material derived from course 6.823
- UCB material derived from course CS252