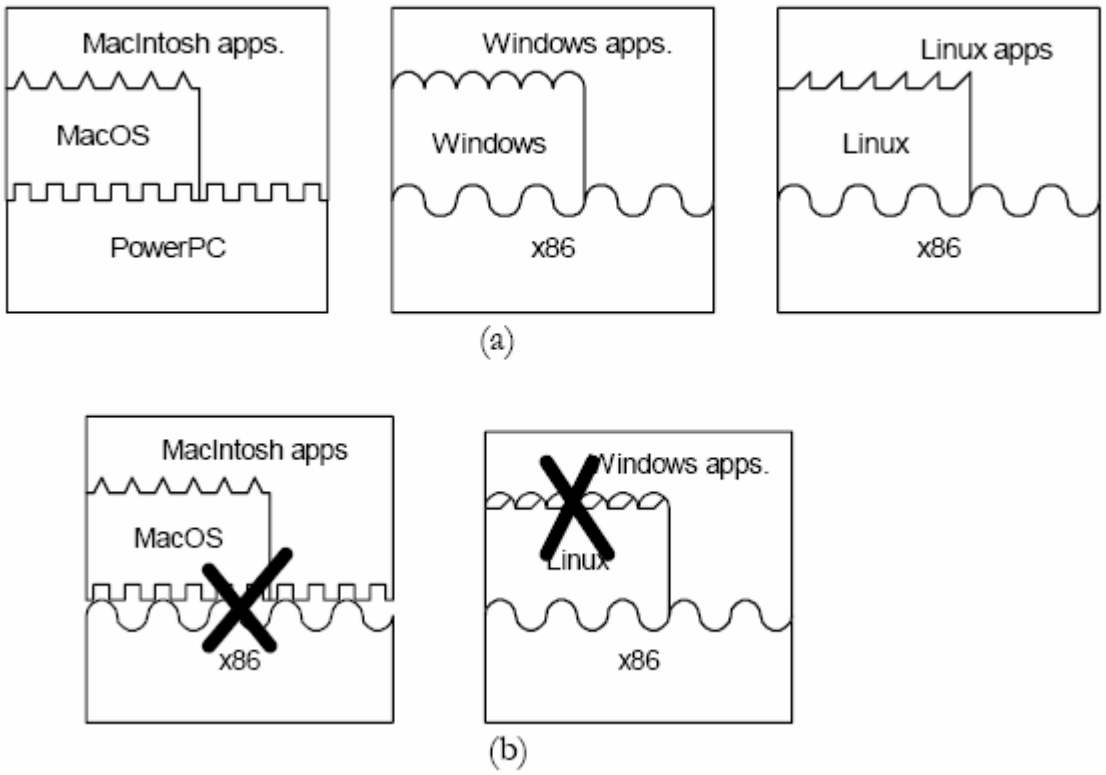
Overview of Virtual Machine Technology

Outline of My Talk Today

- What is a virtual machine (VM)?
- Why do we need a VM?
- VM solutions
- Classification of VMs
- Implementation issues of VM
- Other VM usages

What is a Virtual Machine (VM)?

- **Software** for cross-platform compatibility
  - Traditionally, an application program is bound to a specific platform (which is ISA + OS)



What is a Virtual Machine (VM)?

- VM eliminates this real-platform constraint for higher degree of portability and flexibility
  - How? Being added to a platform to give an appearance of a different platform or even multiple platforms
  - Result: a platform can run programs bounded to a different platform
  - VM may have an OS or an ISA or both, which can be different from the ones in the real platform
  - VM can also be useful for the same ISA and OS, but with different implementation of the ISA

Why do we need VM?

- Portability is essential in networked computing
  - In a network-is-a-computer environment
  - Especially useful for mobile, wireless-download platforms where we can achieve consistent execution environment on
    - diverse CPU/OS/hardware devices
    - e.g., Java VM, GVM, Brew, WIPI, …

Why do we need VM?

- Type-safe, OO, VM-implemented languages are mainstream
  - Java is everywhere
  - Dynamic languages such as Perl, Python, PHP, ..
  - VM can more easily provide a rich runtime environment for
    - Runtime-binding, automatic memory management, OO programming, security, strong typing

    which allows sophisticated software development

Why do we need VM?

- CPU Innovations are often limited by old interfaces
  - Supporting old ISA (mostly meaning x86) may make new high-performance, low-power H/W features difficult to implement
    - E.g., superscalar execution for x86
  - On the other hand, new CPUs that cannot run x86 binaries are not viable on the market
    - e.g., Alpha is dead although it was the fastest CPU
  - Solution: run x86 binaries on high-performance, low-power CPUs
    - Crusoe, FX!32, DAISY, ….

Why do we need VM?

- Optimizations across processor family are difficult
  - e.g., a binary is optimized for one processor model but it can also be executed on different processor model
    - While none wants to keep multiple binaries
  - One solution: dynamic, runtime optimization
    - HP Dynamo

Why do we need VM?

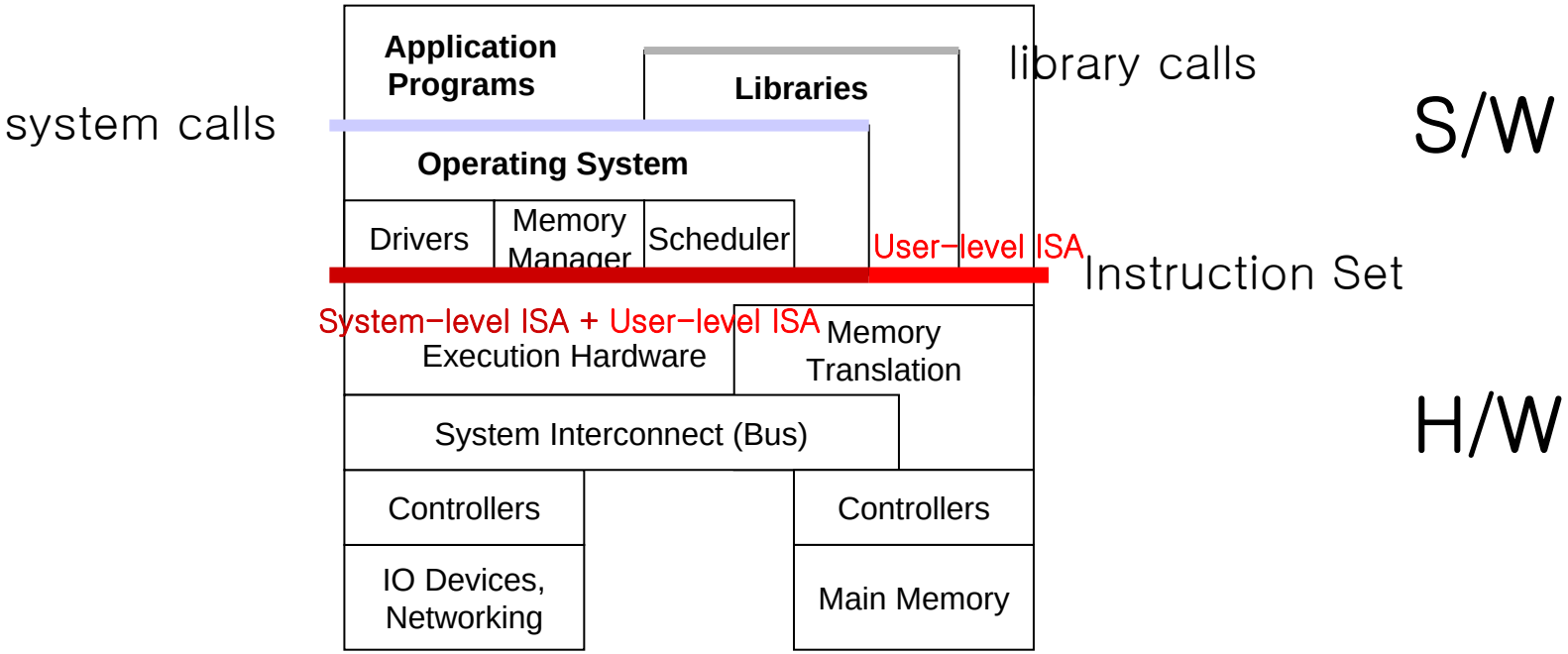
- Single OS on a H/W may open a security hole
  - e.g., a server shared by different groups of users who want to be assured of a secure environment
    - One security hole destroys the whole system
  - Solution: some logical decomposition
    - One VM for one group
    - e.g. IBM z/VM, user-mode Linux
  - We will see many more products soon

VM Solution for our Problems

- Implementing a S/W layer called VM for virtualization
  - Mapping a virtual guest system to a real host system
- Two types of VM, depending on where to put the layer
  - **Process VM**: at the ABI (application binary interface) level
  - **System VM**: at the ISA (instruction set architecture) level

Interfaces in Computer Systems

- A computer system architecture is composed of many interfaces which abstract implementations
  - ISA: interfacing H/W and S/W
  - System call: interfacing application and OS
  - Library call: interfacing application and standard libraries (API)



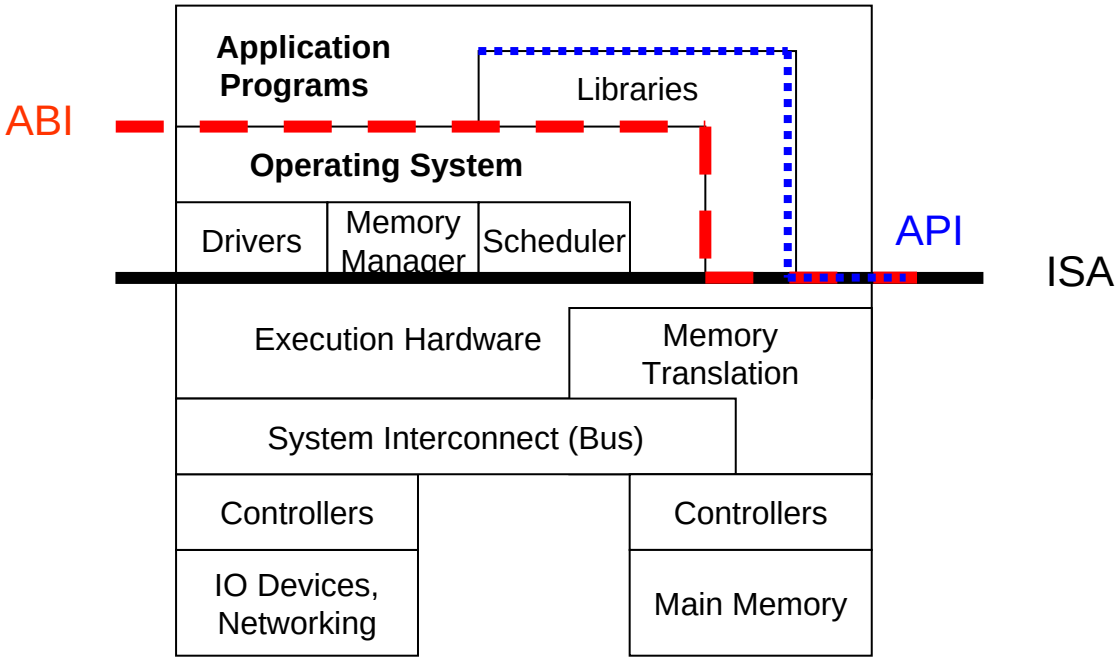
ABI and ISA

- Two interfaces are important for VM classification
  - ISA and ABI

- ISA is an interface bet'n S/W and H/W
  - User-level ISA: visible to application S/W and OS
  - System-level ISA: visible **only** to OS

- ABI is an interface bet'n application S/W and "system"
  - Composed of user-level ISA and OS system calls

VM and Interface Layers



- Process VM is placed at the ABI interface
- System VM is placed at the ISA interface

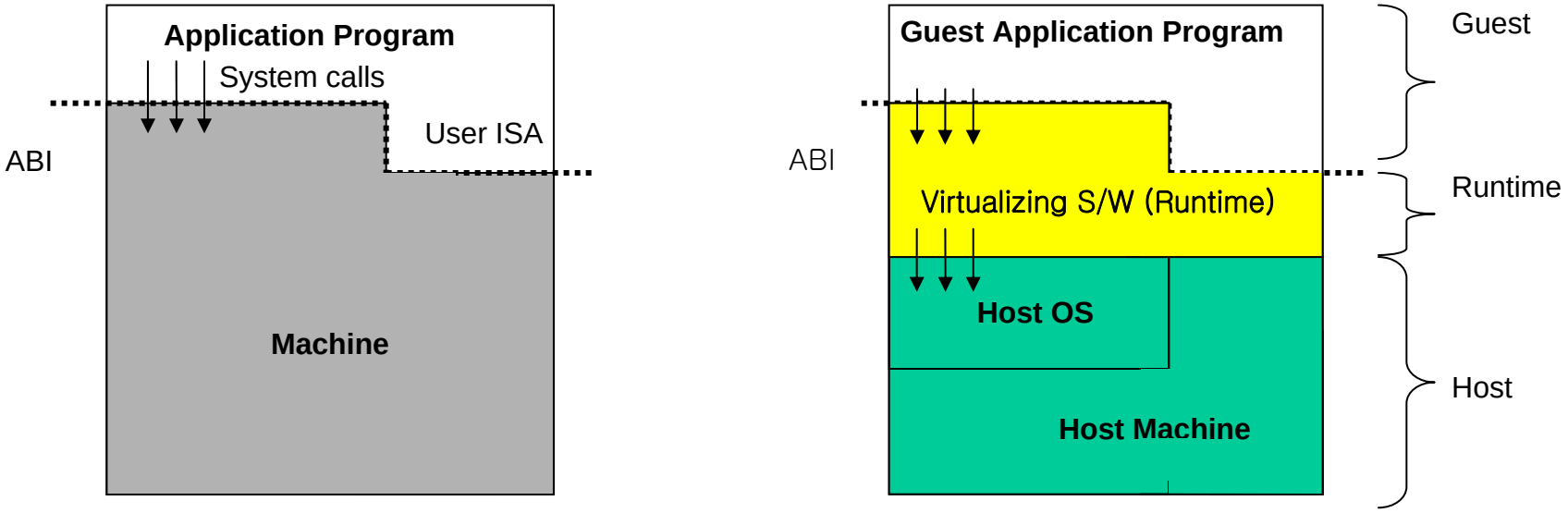
Process VM vs. System VM

- **Process VM** at the ABI level
  - Virtualization of individual processes
  - E.g., running x86 applications on Alpha CPU

- **System VM** at the ISA level
  - Virtualization of complete systems
  - E.g., running Linux (and its applications) on Windows

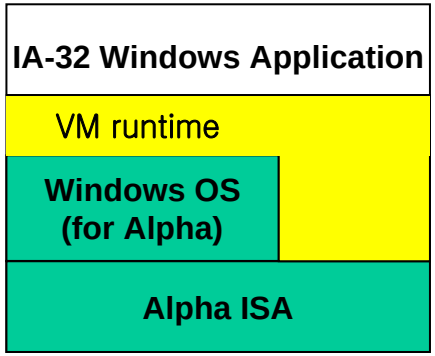
Process VM at the ABI Layer

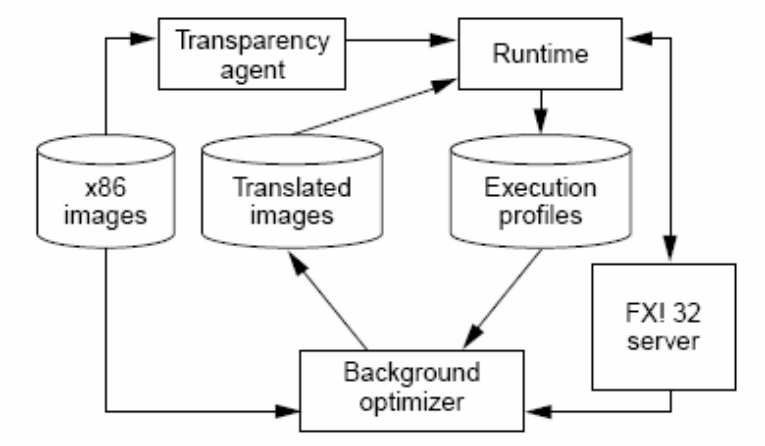
- Support an individual guest process
- Run executables of different ISA or OS
  - Virtualizes the ABI layer
  - VM (runtime) emulates user-level instructions & system calls
  - VM is a process from the view point of the host
  - Guest application program cannot see the host



Process VM Examples: Dynamic Translator

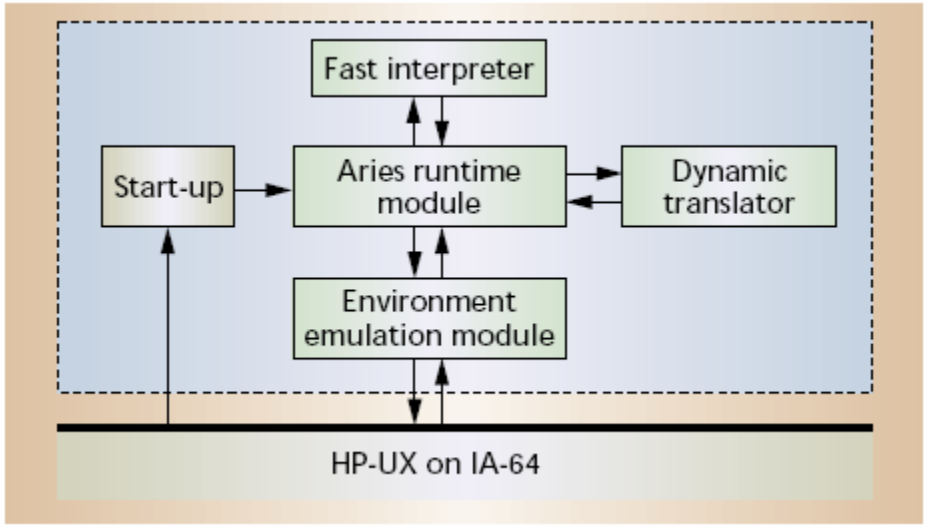
- Digital FX!32
  - Run x86 binaries on Alpha (both with Windows OS)
  - Interpretation at execution time which collects profiles
    - Esp. targets of indirect jumps
  - Static translation in-between program runs





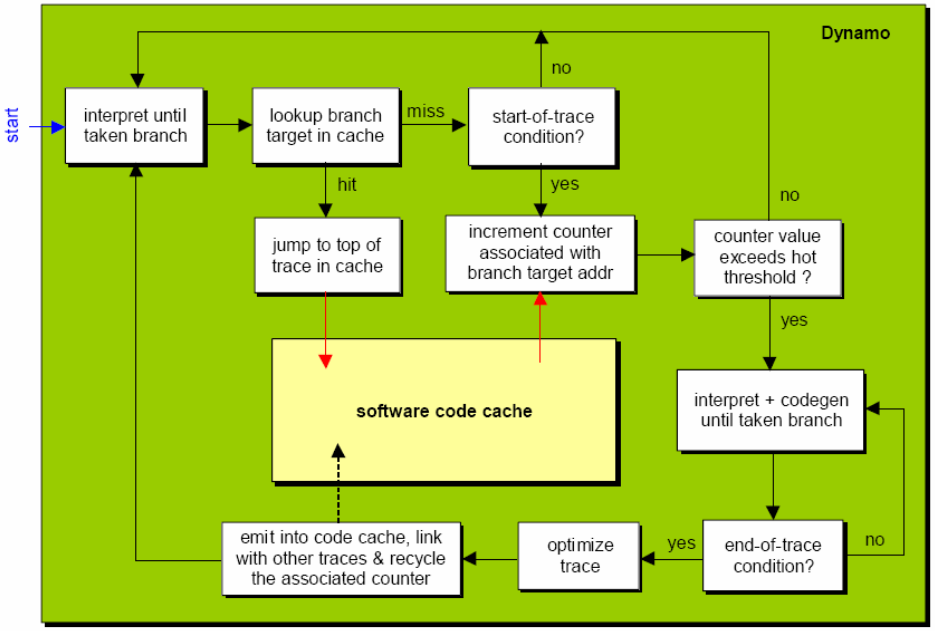
Process VM Examples: Dynamic Translator

- Aries
  - Run PA-RISC binaries on IA-64 IPF



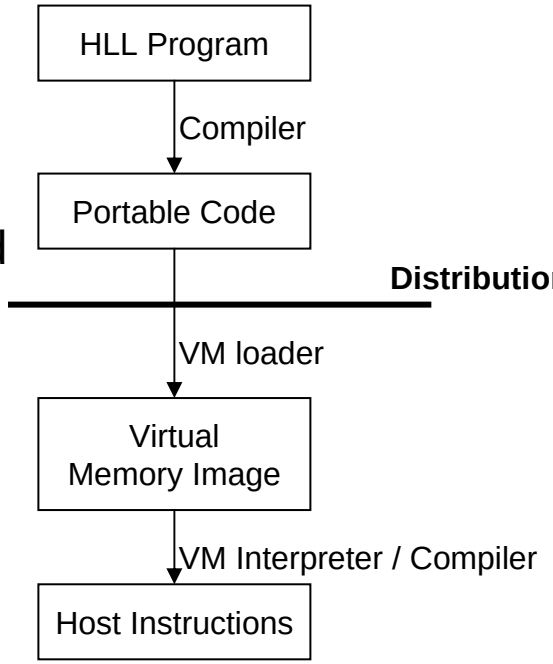
Process VM Examples: Dynamic Translator

- Dynamo
  - Optimize and Run PA-RISC binaries on PA-RISC
  - Start with interpretation to detect hot spot code
  - Trace-based optimized code is located in code cache



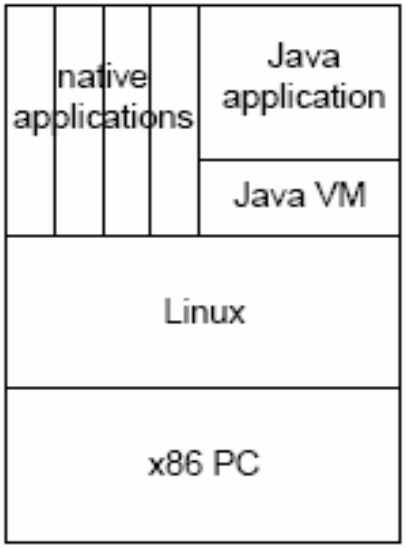
Process VM Example: HLL VMs

- Cross-platform portability for different ISAs is not easy and case-by-case
  - Can we have a full cross-platform portability?
    - An executable runnable on any platform
- High-level language (HLL) virtual machines
  - A VM environment designed for ease of portability and to match the features of a HLL
  - Minimizes H/W specifics and OS specifics
  - Portable intermediate code (virtual ISA) is distributed instead of binary object code
  - VM executes intermediate code
    - By interpretation or compilation
  - Pascal, Java, Python, .NET, ….



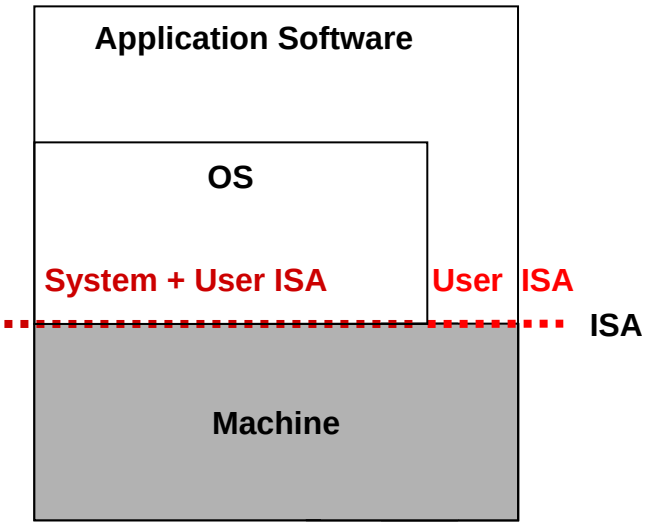
Process VM Example: Java VM

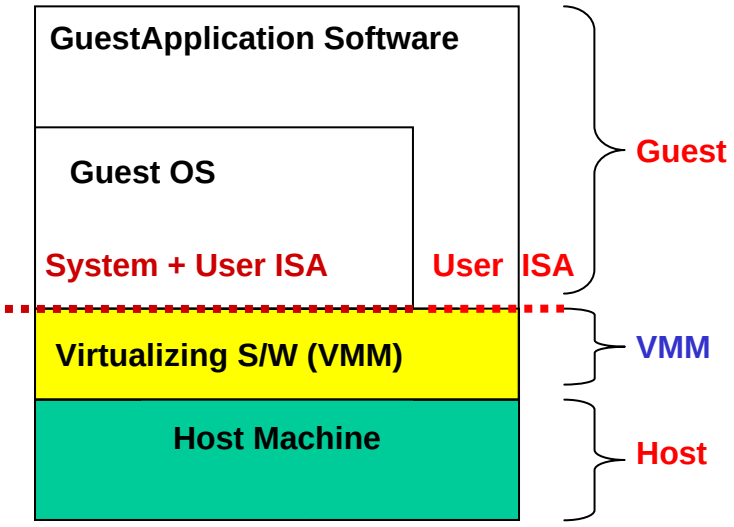
- Java VM
  - Platform independence and high security
  - Bytecode binary can be run in any platform where Java VM is installed



System VM at the ISA Layer

- Provides a complete guest system environment
- Run whole OS & executables
  - Virtualizes the ISA layer
  - VM, called a VMM, emulates both user-level & system-level ISA (i.e., both application code and OS code)



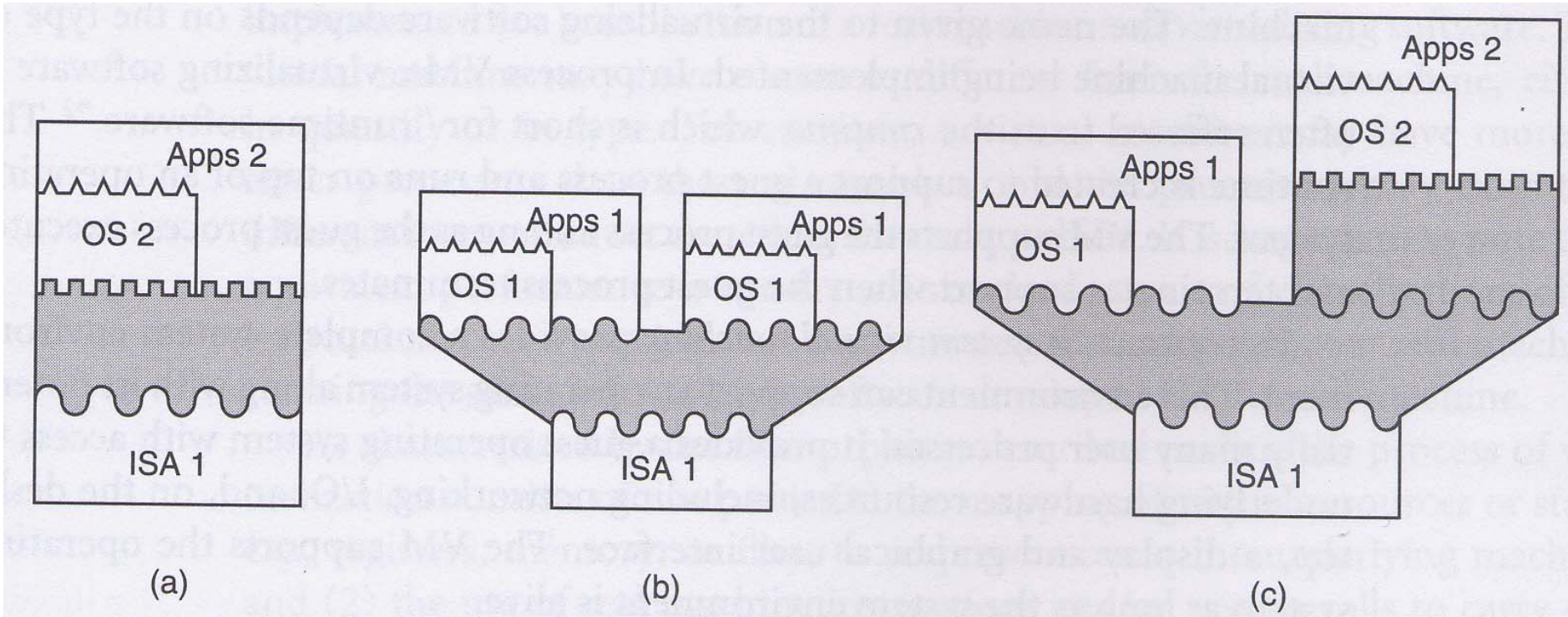


System VM Variations

(a) One ISA is emulated by the other ISA

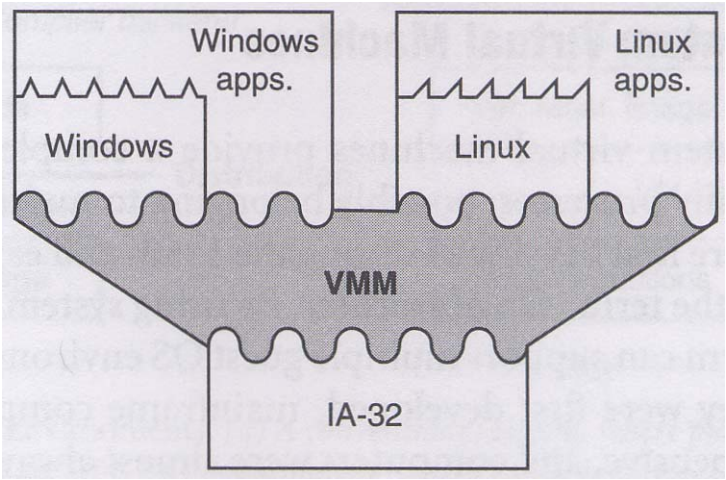
(b) Multiple platforms replicated on a single platform

(c) Multiple different platforms on a single platform



System VM Examples: Whole-System VM

- **Whole-system VM**
  - Run OS and applications on a platform with different OS and ISA
    - e.g., Virtual PC which allows Mac to run Windows system
  - Multiple platforms replicated on a single platform
    - Supports security among different user groups
  - Run multiple OS on a single platform
    - e.g., both Linux and Windows running on a PC
  - Examples: VMWare GSX, ESX, User-mode Linux, IBM z/VM, Xen

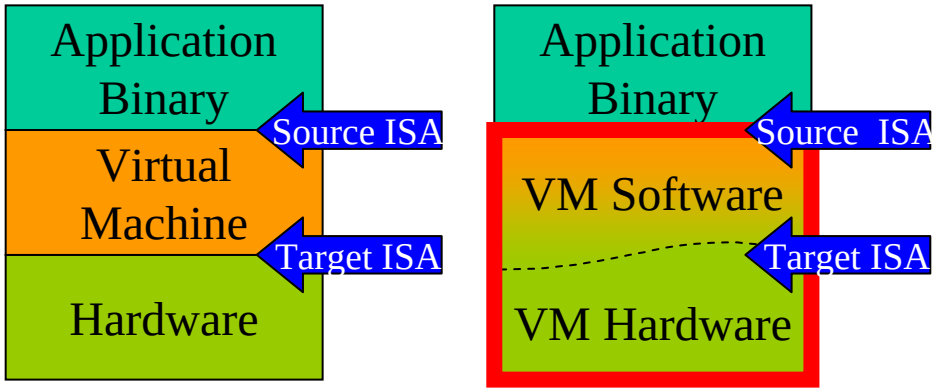


System VM Examples: Whole-System VM

- Main issue in implementing whole-system VM
  - Virtualization of hardware resources among (multiple, or even different) guest OS
  - Execution of privileged instructions by guest OS is intercepted and performed by the VMM
    - How? By running guest OS in user mode

System VM Examples: Codesigned VM

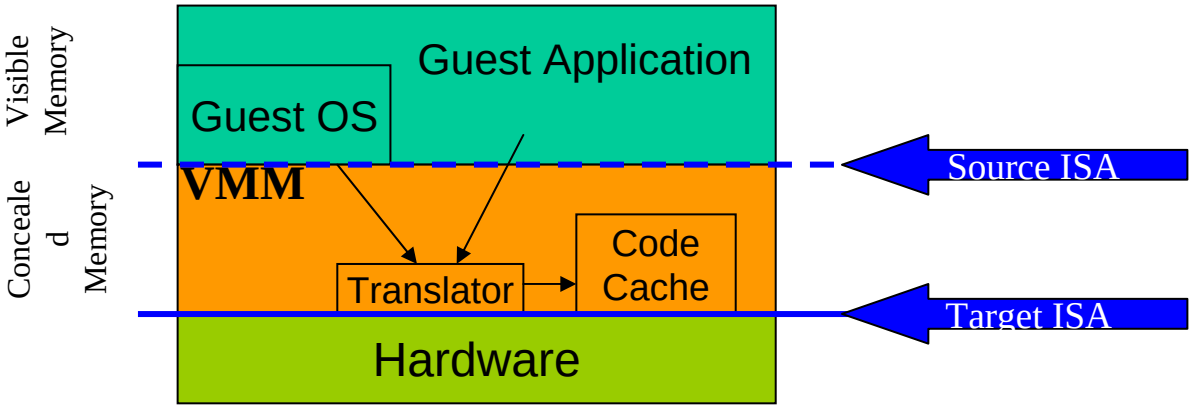
- **Codesigned VM**
  - Innovative ISA/implementation to run existing binary (e.g., x86) for higher performance with lower power
  - Unlike a conventional VM, codesigned VM S/W is a part of H/W, designed collaboratively with H/W
    - Divide the implementation of HW & SW in an optimal way
  - VMM is located in a concealed memory, not visible to app/OS
    - Controls H/W resources completely



System VM Examples: Co-Designed VM

How Codesigned VM works

- When executing guest application, VMM translates guest instructions to native instructions (with interpretation to detect hotspots), caches them in a concealed memory, and executes them directly on the H/W
- Guest application/OS can never be executed directly
- No native ISA applications exist; in fact, its ISA is hidden



System VM Examples: Co-Designed VM

- Codesigned VM vs. Process VM
  - Similarity: emulate the source ISA using interpretation, dynamic translation, and code cache
  - However, codesigned VMs require
    - Compatibility at the ISA level (not at the ABI level)
      - Both user-level & system-level ISA must be emulated
      - Do not stop emulation at system calls, unlike process VM
    - Improved performance, power efficiency, design simplicity
      - Compatibility is just a requirement, not a motivation.

System VM Examples: Codesigned VM

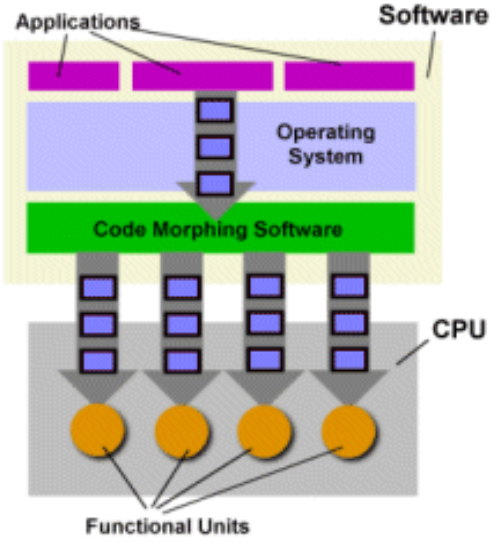
- Codesigned VM vs. whole-system VM
  - Similarity: support an entire system (OS + application)
    - Codesigned VM is a form of a system VM
  - However, codesigned VMs are not intended for
    - Virtualization of HW resources
    - Supporting multiple VM environments
  - Again, the goal is performance, power efficiency, and design simplicity which whole system VM do not pursue

Co-Designed VM Example: Crusoe

Transmeta Crusoe

• Low-power x86-compatible processor implemented with "code morphing" software

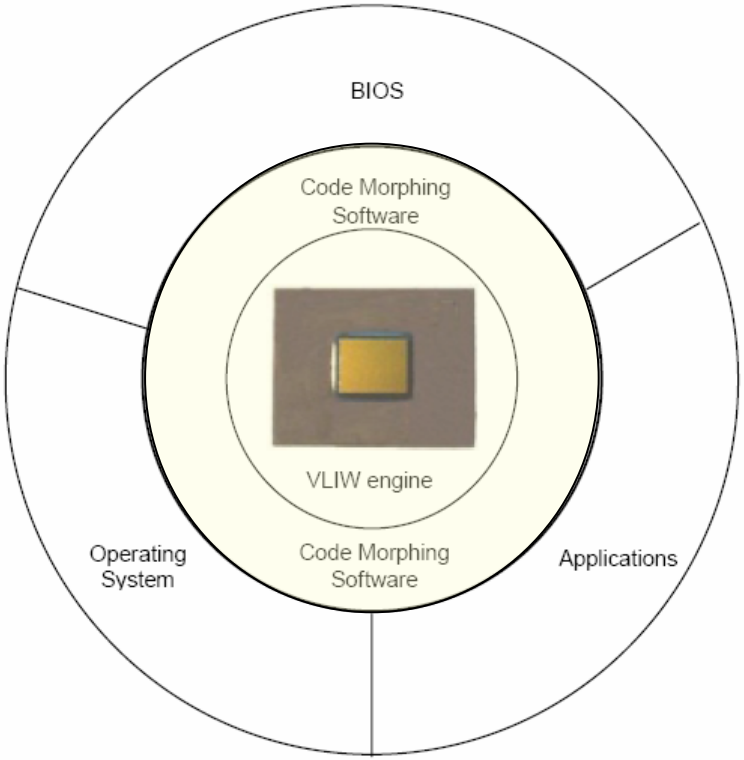
- Motivation and Goal
  - The x86 architecture is too complicated to implement, esp. decoding
    - 30% of logic gates in Pentium 3 is for instruction decoding
  - The goal is removing the overcomplicated instruction decoding from hardware, and design a simpler hardware
  - The Code Morphing software handles x86 decoding, and the morphed software runs on a simple VLIW processor



Crusoe's VLIW Instruction Scheduling

Co-Designed VM Example: Crusoe

- Code morphing example



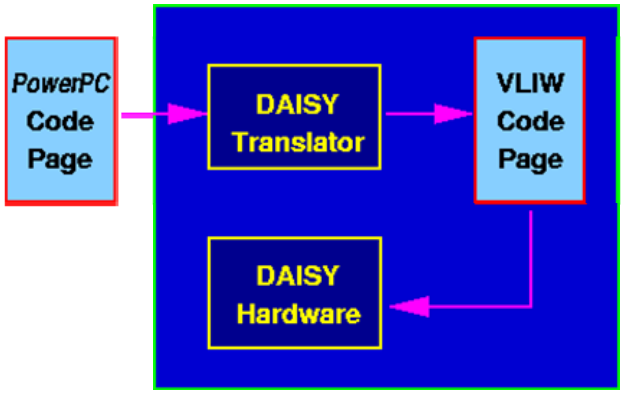

```
A. addl %eax,(%esp)          // load data from stack, add to %eax
B. addl %ebx,(%esp)          // ditto, for %ebx
C. movl %esi,(%ebp)          // load %esi from memory
D. subl %ecx,5               // subtract 5 from %ecx register
```

Translated from x86 ISA to Crusoe VLIW ISA
by code morphing software

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

Co-Designed VM Example: Daisy

- IBM Daisy
  - Translation of PowerPC code to VLIW code
  - Employing Tree-VLIW for massive parallel execution



DAISY Architecture

Original PowerPC Code

```
        add     r1,r2,r3
        bc      L1
        sli     r12,r1,3
        xor     r4,r5,r6
        and     r8,r4,r7
        bc      L2
        b       OFFPAGE
L1:     sub     r9,r10,r11
        b       OFFPAGE
L2:     cntlz   r11,r4
        b       OFFPAGE
```

VLIW1 : On Interrupt restart at add
VLIW2 : On Interrupt restart at sli

Translated VLIW Code

```
VLIW1:
                add     r1,r2,r3
                bc      L1
xor r63,r5,r6   sub     r9,r10,r11
b VLIW2   b OFFPAGE

VLIW2:
                sli     r12,r1,3
                r4  = r63
                and     r8,r63,r7
                bc      L2
                cntlz   r11,r63
b OFFPAGE  b OFFPAGE
```

Implementation Issues - Execution

How to execute guest instructions on a host

- Interpretation (emulation)
  - Slow execution speed, but easy implementation
- Translation
  - Dynamic translation (JIT): recompile/translation done while running
  - Static translation (AOT): multiple binary problem
  - Complicated but faster execution, especially with optimizations
- Run directly on the hardware
  - When the host and guest ISA is the same
  - e.g., VMWare handles system calls only, and run x86 instructions directly on the hardware

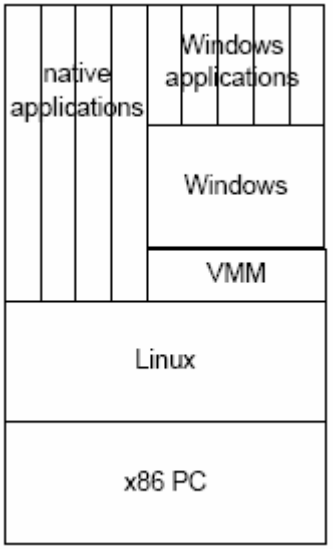
Implementation Issues - Optimization

How to optimize the code running on a VM

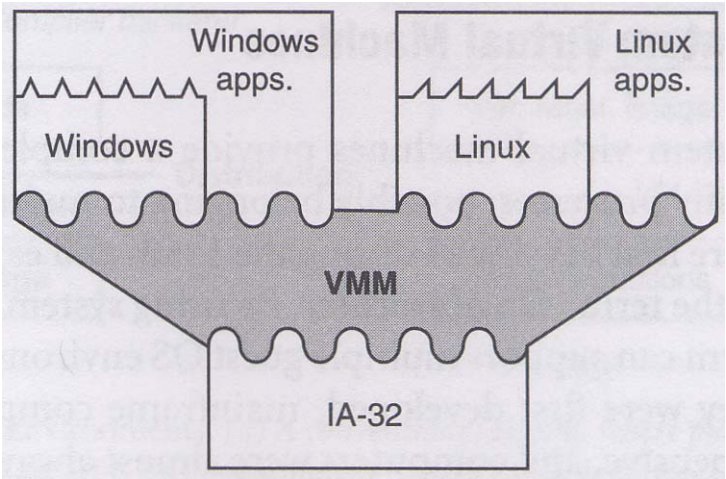
- Need more dynamic, runtime optimization
- Not exactly the same as static optimizations in static compilers
- Must utilize more runtime profiling information and more speculation [Hind, CGO 2005]

Implementation Issues – Platform

- Hosted
  - Runs as a process on an existing host OS
  - Rely on host OS for H/W interaction
  - VMWare GSX, user-mode Linux



- Stand-alone
  - VMM on top of bare hardware
  - All H/W interactions done by VMM itself
    - Highly efficient
  - VMWare ESX, IBM z/VM



Implementation Issues - Compatibility

Complicated compatibility issues for translation

- Self-modifying code
- Self-referencing code
- Precise traps

Other Virtual Machine Usages

- Legacy support
  - Vax emulator, HP calculator emulator
- Development environment
  - Can simulate a network of servers on a single H/W (e.g., via multiple VMware instances running on a single machine)
  - Migrate working environment
- Better workload distribution
  - Multiple VMs on multiprocessor systems
  - VM can move from processor to processor instantly

Summary

- VM is one of the hottest topics in architectures, compilers, and OS research community
- We will study various VMs and learn VM technology
- We will especially focus on translation and optimizations