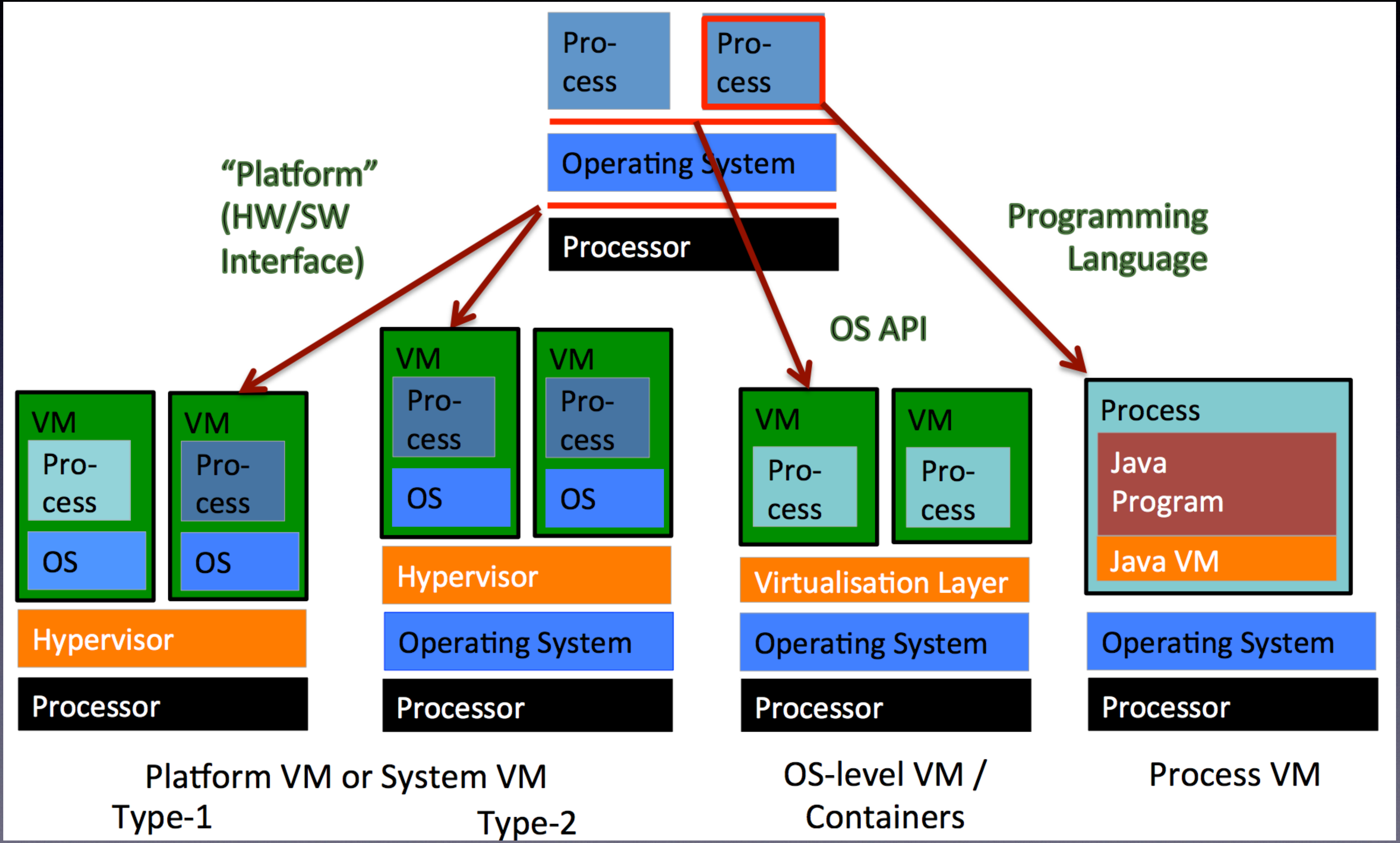
Virtual Machines

Background

- IBM sold expensive mainframes to large organizations
  - Some wanted to run different OSes at the same time (because applications were developed on old OSes)
  - Solution: IBM developed virtual machine monitor (VMM) or hypervisor (circa 1974)
- Monitor sits between one or more OSes and HW
  - Gives the illusion that each OS controls the HW
  - Monitor multiplexes running OSes
  - A level of indirection: apps assume separate CPU, unlimited memory; now another layer to provide similar illusion to OS

Today’s “VMs"



Pro-
cess
Pro-
cess
Operating System
Processor
“Platform”
(HW/SW
Interface)
OS API
Programming
Language
VM
Pro-
cess
OS
VM
Pro-
cess
OS
Hypervisor
Processor
VM
Pro-
cess
OS
VM
Pro-
cess
OS
Hypervisor
Operating System
Processor
VM
Pro-
cess
VM
Pro-
cess
Virtualisation Layer
Operating System
Processor
Process
Java
Program
Java VM
Operating System
Processor
Platform VM or System VM
Type-1
Type-2
OS-level VM /
Containers
Process VM

Today’s VMs

- Why VMMs now? Are there new reasons for using VMMs?
- Which deployment models are appropriate for which uses?
- What are the key challenges/issues in building VMMs?

Resurgence in VMs

- Sparked by work on Disco (system from Stanford/ Rosenblum)
- Resulted in VMware -- now a market leader in virtualization

VM Observations

- Instruction-set architectures is one of the few well-documented complex interfaces
  - interface includes meaning of interrupt numbers, etc.
- Anything that implements the interface can execute the software for the platform
- Virtual machine is a software implementation of this interface

Outline

- Disco project
- Design space for virtualization
- Xen project

What is a machine?

- The hardware architecture defines the following:
  - instructions
  - special instructions (system calls, setting trap handlers, etc.)
  - memory management (page tables, TLB)
  - device interface (I/O - memory mapped loads/stores)
- How do we build a “virtual machine” that conforms to an architecture specification?

Traditional Process-OS Model

- Processes run in user mode
- Processes “trap” into OS
  - when they want services from the OS
  - or when they have “faults”
- OS runs in privileged (kernel) mode
  - can execute instructions to setup/update TLB or paging
  - can execute instructions to install trap handlers

Virtualizing CPU

- Basic technique: limited direct execution
- Ideal case:
  - VMM jumps to first instruction of the OS and lets the OS run
  - Generalize a context switch on processes to machine switch
    - save the entire machine state of one OS including registers, PC, and privileged hardware state
    - restore the target OS state
  - Guest OS cannot run privileged instructions (like TLB ops); VMM must intercept these ops and emulate them

System Call Primer

- Consider: *open(char*path, int flags, mode_t mode)*

```
open:
	push dword mode		// args
	push dword flags
	push dword path
	mov eax, 5			// system call number
	push eax
	int 80h				// trap
```

- Process code, hardware, and OS cooperate to implement the interface
- Trap: switches to kernel mode, jumps to OS trap handler; trap handlers registered by OS at startup

Virtualized Platform

- Trap handler is inside the VMM; executed in kernel mode
- What should the VMM do?
  - does not know the details of the guest OSes
  - but knows where the OS's trap handler is
    - when the guest OS attempted to install trap handlers, VMM intercepts and records the information
  - so jump into OS; which executes the actual handler, performs another privileged instr (iret on x86), bounces back into VMM
  - VMM performs a real return from trap and returns to app

Execution Privileges

- OS cannot be in kernel mode
- Disco project: MIPS hardware had a supervisor mode
  - kernel > supervisor > user
  - supervisor can access little more memory than user, but cannot execute privileged instructions
- No extra mode:
  - run OS in user mode and use memory protection (page tables and TLBs) to protect OS data structures appropriately
- x86 has 4 protection rings, so extra mode is available

Virtual Memory Primer

- TLB: fast cache used in every instruction
  - TLB miss handled by OS in some processors (software TLB)
  - In other cases, hardware fills TLB using a page table
    - OS manages the page table
    - Hardware is a consumer of the page table
- Question: what issues arise with virtual machines?
  - How do we tackle such issues?

Virtualizing Memory

- Normally:
  - each program has a private address space, OS virtualizes memory for its processes
- Now:
  - multiple OSes can share the actual physical memory
  - So we have virtual memory (VM), physical memory (PM), and machine memory (MM)
  - OS maps virtual to physical addresses via its per-process page tables
  - VMM maps the physical address to *machine memory* via its per-OS page tables

2-Level Translation

- Let us consider software managed TLB
- In a virtualized system:
  - Application traps into VMM; VMM jumps to OS trap handler
  - OS tries to install (VM, PM) in TLB, but this traps
  - VMM installs (VM, MM), returns to OS and then App
  - VMM maintains (PM, MM) mappings and even does paging

Information Gap

- VMM often doesn’t know what the OS is doing
- For example, if OS has nothing else to run:
  - go into an idle loop and spin waiting for the next interrupt
- Another example:
  - most OSes zero pages before giving to processes for security
  - VMM also has to the do the same, resulting in double work!
- One option is inference of OS behavior, another is paravirtualization

Design Space

	App is not modified	App is modified
OS is not modified	Disco (VMWare)	---
OS is modified	Xen	Denali

Xen

- Key idea: change the machine-OS interface to make VMs simpler and higher performance
  - Pros:
    - better performance on x86
    - some simplifications in VM implementation
    - OS might want to know that it is virtualized
  - Cons: must modify the guest OS

Xen & Paravirtualization

- VM-style virtualization on an *uncooperative architecture*
- OSes are ported to a new “x86-xeno” architecture
  - call to Xen for privileged operations
  - porting requires source code
- Retain compatibility with OS API
  - Must virtualize application visible architecture features

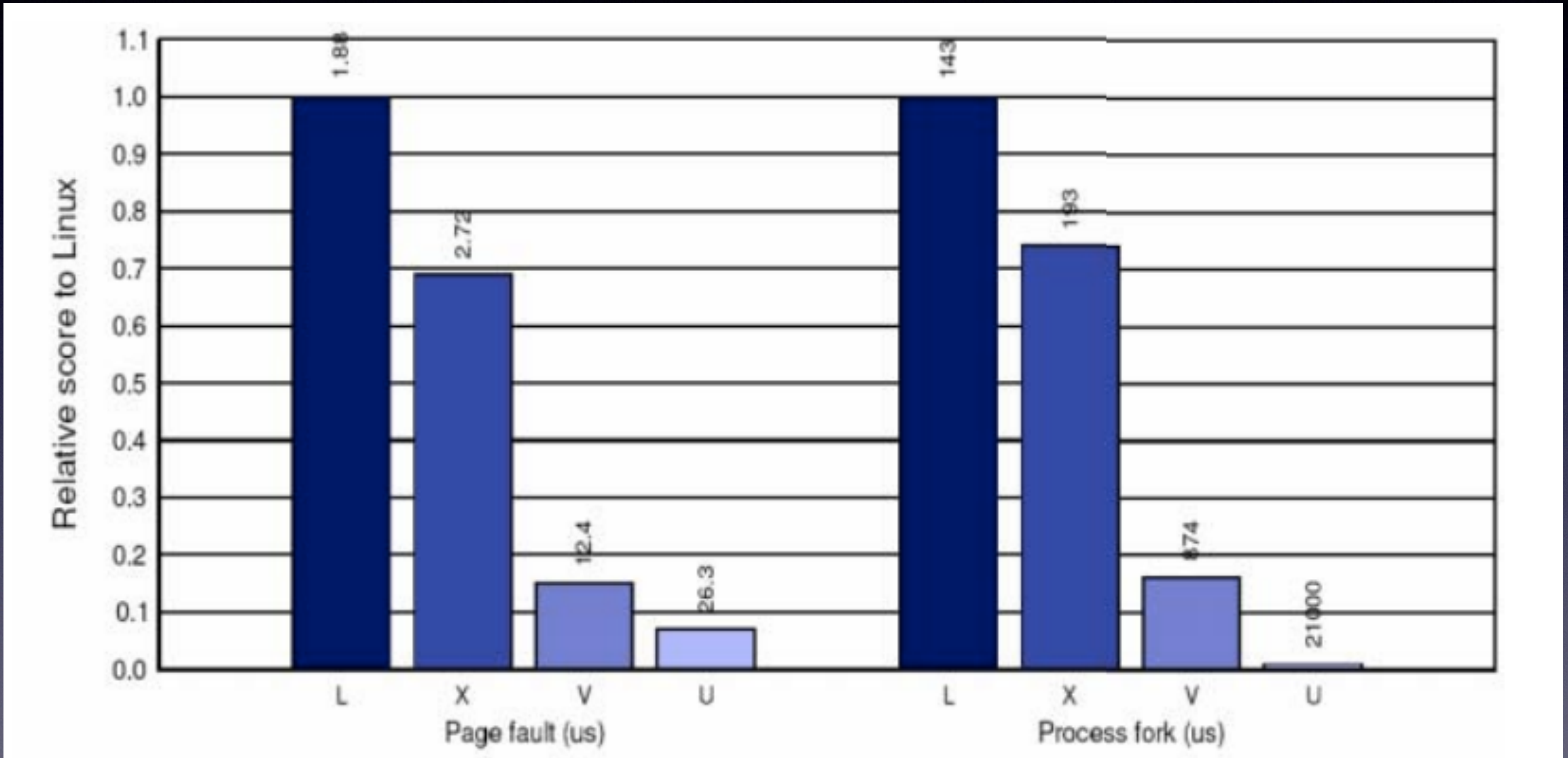
Fully virtualizing Hardware TLB

- Constraints:
    - No tags on TLB
- Use shadow page tables
    - Guest OS maintains “virtual to physical mem” map
    - VMM maintains “virtual to machine mem” map
- Guest reads of page table is free
- Guest writes need switching to VMM
- Accessed/dirty bits require upcalls into OS

Paravirtualizing Hardware TLB

- Paravirtualization obviates the need for shadows
  - modify the guest OS to handle sparse memory maps
- Guest OSes allocate and manage their own PTs
  - map Xen into top 64 MB in all address spaces
- Page table updates passed to Xen for validation (use batching)
- Validation rules:
  - only map a page if owned by the requesting guest OS
  - only map a page containing PTEs for read-only access
- Xen tracks page ownership and current use

Memory Benchmarks



lmbench results on Linux (L), Xen (X), VMWare Workstation (V), and UML (U)

Other Nice Ideas

- Domain 0:
  - run the VMM management at user level
  - easier to debug
- Network and disk are virtual devices
  - virtual block devices: similar to SCSI disks
  - model each guest OS has a virtual network interface connected to a virtual firewall router