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 World

- Why VMMs now? Are there new reasons for using VMMs?

- 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

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
push dword flags
push dword path
mov eax, 5
push eax
int 80h
```

- 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

- Application remains the same
- 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 the call and records the information)
  - so jump into OS; which executes the actual handler, performs another privileged instruction (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

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 and must do so transparently
  - 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 resulting physical address to machine memory via its per-OS page tables

Address Translation Primer

- Assume a system with software-managed Translation Lookaside Buffer (TLB)
  - TLB maps virtual address to physical address for each instruction
  - TLB miss: trap into the OS which looks up page tables and installs translation and retries instruction
- Consider 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

- What are the design goals in building a virtualization solution?

Design Space

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

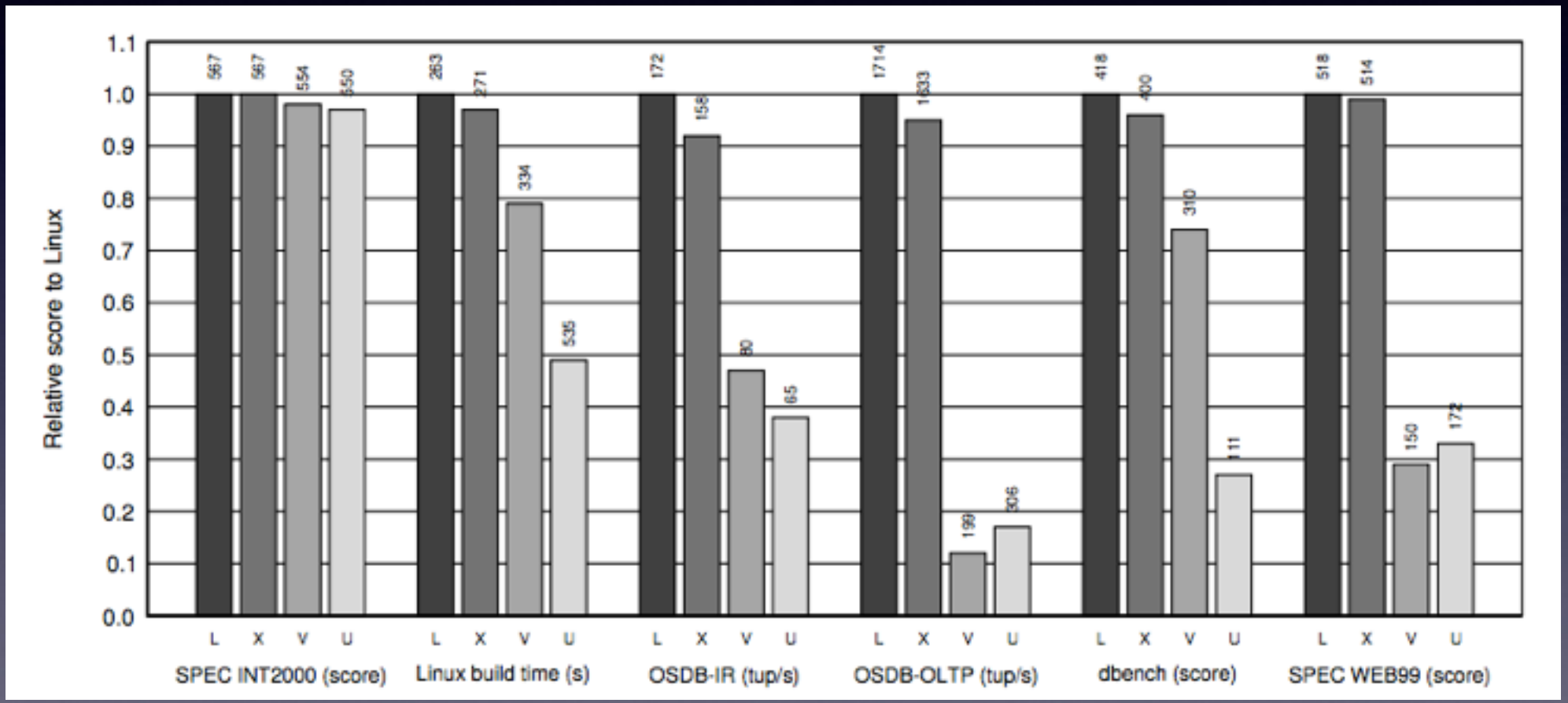
Xen

- 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
  - Aims for performance isolation

Xen & Paravirtualization

- VM-style virtualization on an uncooperative architecture
- Support full-featured multi-user multi-application OSes
  - contrast with Denali: thin OSes for lightweight services
- 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

Performance



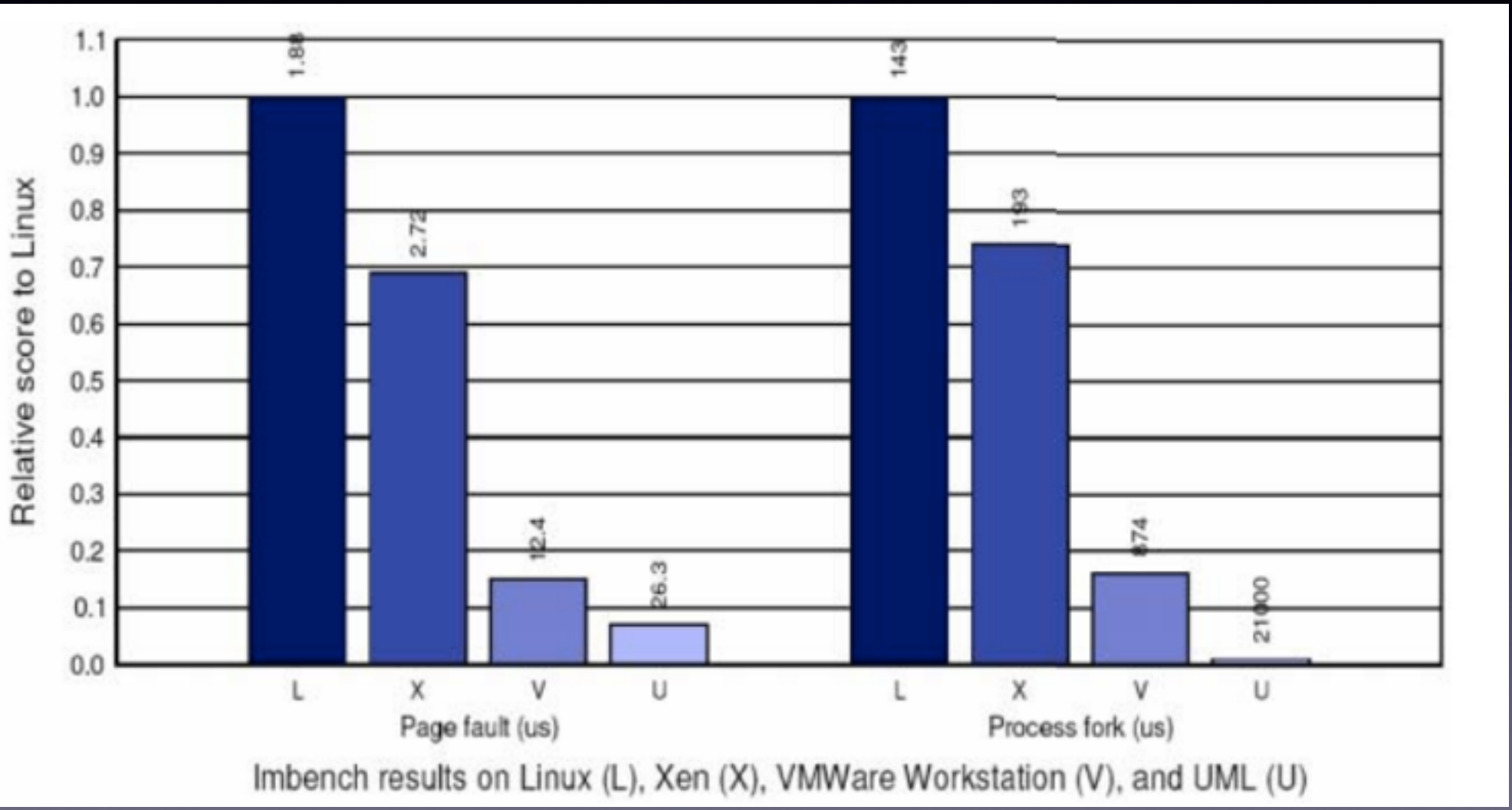
Fully virtualizing the MMU

- Constraints:
  - Hardware-based TLB
  - 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 the MMU

- 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
- Updates to page tables must be 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



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

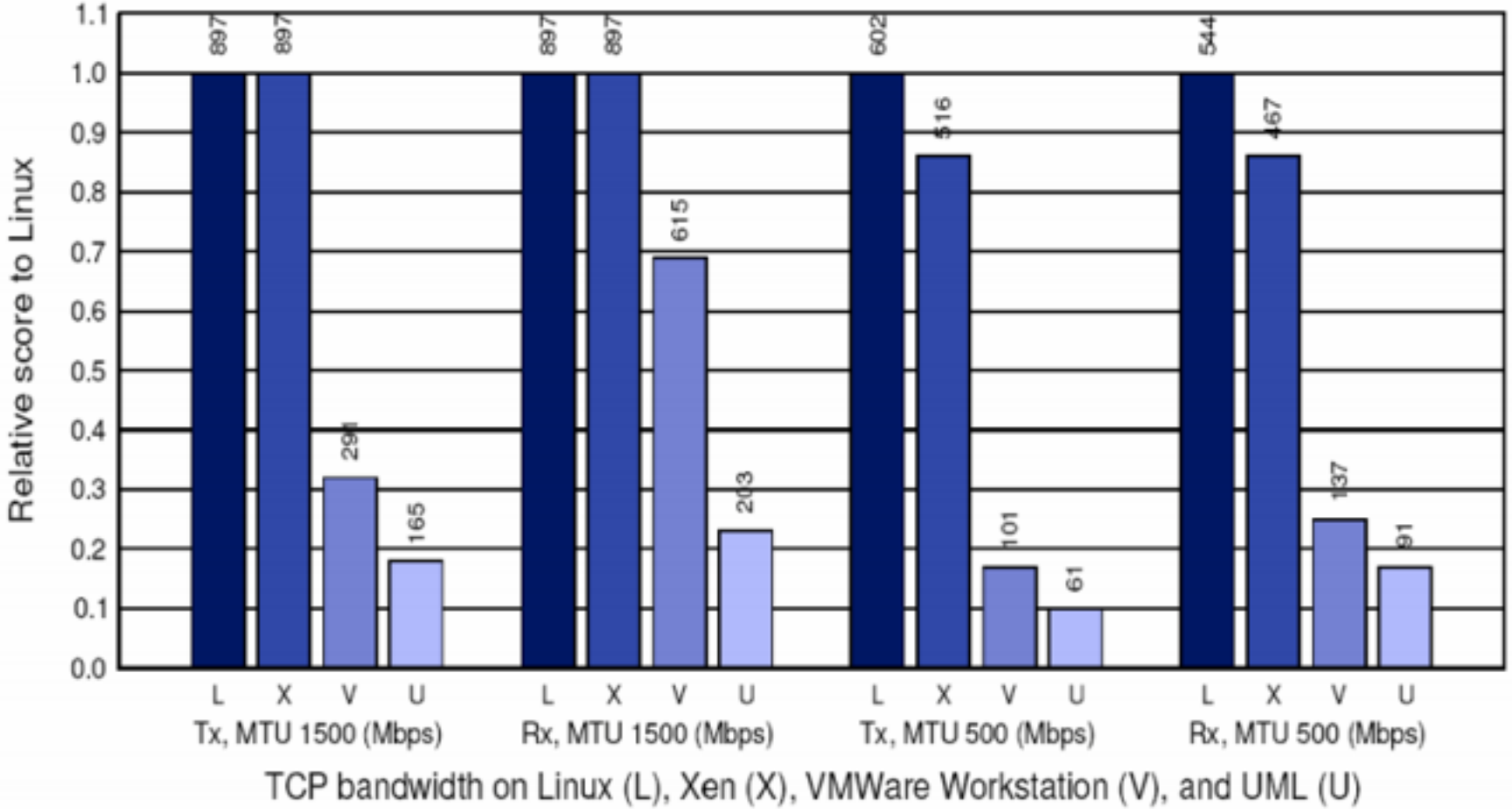
I/O Virtualization

- Need to minimize cost of transferring bulk data via Xen
  - copying costs time
  - copying pollutes caches
  - copying requires intermediate memory
- Device classes
  - network
  - disk
  - graphics

I/O Virtualization

- Xen uses rings of buffer descriptors
  - descriptors are small, cheap to copy and validate
  - descriptors refer to bulk data
  - no need to map or copy the data into Xen's address space
  - exception: checking network packet headers prior to TX
- Use zero-copy DMA to transfer bulk data between hardware and guest OS
  - net TX: DMA packet payload separately from header
  - net RX: page-flip receive buffers into guest address space

TCP Results


Relative score to Linux
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
897
897
291
165
897
897
615
203
602
516
101
61
544
467
137
91
L X V U
Tx, MTU 1500 (Mbps)
L X V U
Rx, MTU 1500 (Mbps)
L X V U
Tx, MTU 500 (Mbps)
L X V U
Rx, MTU 500 (Mbps)
TCP bandwidth 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