A Framework for Virtual Device Drive Development & Virtual Device-Based Performance Modeling

Robert Geist ● Zachary H. Jones ● Mike Westall

Acknowledgements

- This research has been supported by
  - IBM Faculty Award
  - IBM Ph.D. Fellowship
  - NSF Grant Award 0722313

Thesis Statement

We will provide solutions to problems in operating system virtualization that have been motivated by major projects in CPSC 822.

Motivation

- CPSC 822, Operating System Design: A Case Study
- Projects:
  - modify schedulers for performance
  - new kernels
  - write/build drivers for real devices
- Limitations:
  - dedicated hardware required (usually crashed)
  - limited enrollment
  - waiting list, every semester

Virtualization

- Since mid-1960's
- Generally has been a difficult (expensive) problem
- Intel VT-x/AMD-V: Virtualization for the masses
- Large part of the course can be virtualized
  - VMWare, KVM, VirtualBox

Real Challenges

- **Device Driver Development**
  - Write real device drivers on virtual devices
- Virtual Lab for CPSC 822
  - Real OS and performance work on virtual hardware
- Disk Scheduling
  - Model real hardware in a virtual machine

Device Driver Development

- Device driver for non-trivial graphics card
  - Formerly 3Dlabs Permedia2V
- Virtualization provides only a simple VGA controller
  - Hardware accelerated graphics solutions virtualized at API level rather than architecture level
  - Experimental support for importing devices to virtual machines starting to emerge
- Still leaves one more problem…

One Problem

- Virtualization of one architecture is inadequate
  - Graduate students have been known to communicate with one another
  - Kernel changes (each semester) rendering last semester’s driver inoperable
  - *But* too many students waste time patching last semester’s solution

- Solution? Replace the graphics card.
  - Companies believe that giving specifications away would increase the risk of competitors stealing trade secrets

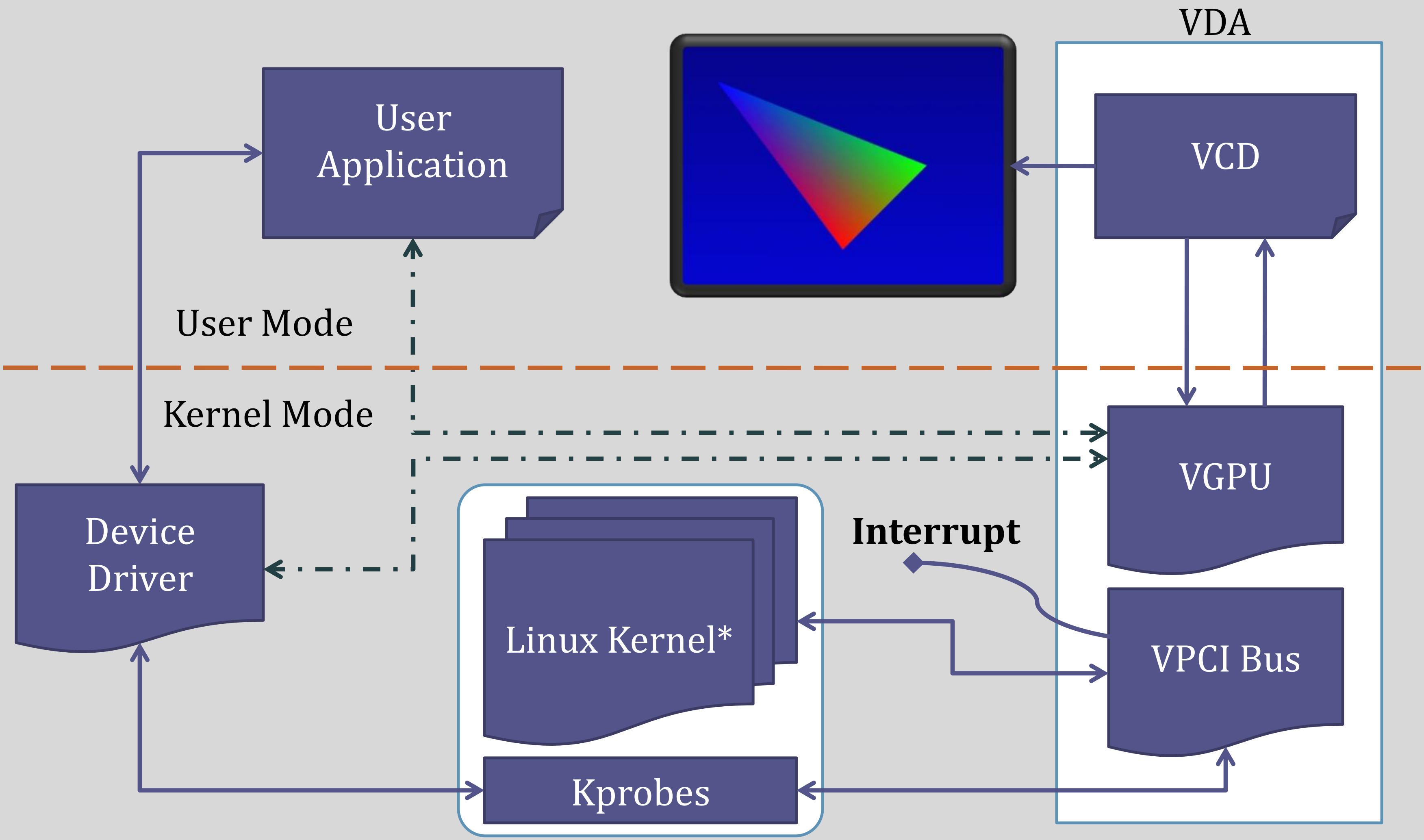
Design Goals for Virtual Architecture

- Support sophisticated components:
  - Requires scheduling
  - Requires memory mapping
  - Generates Interrupts
  - Provides DMA
- Require no special function calls.
- No modifications to existing Linux kernel
- Virtual architecture that can be easily reconfigured each semester

Device Driver

- Scan PCI bus for GPU
  - Finds base address and IRQ
- Call ioremap()
  - Map base address to kernel virtual memory
- Initialize registers
- Call request_irq()
  - Register DMA interrupt handler
- In response to ioctl() from the user app:
  - Memory map registers to user virtual memory
  - Handle DMA buffers, queuing, and initiating
- In response to interrupt
  - Interrupt handler is called and processes next DMA buffer queued (if buffer queue is not empty).

Virtual Device Architecture



Kernel Probes

- Debug mechanism to monitor events in a system.
  - Probe almost any instruction in the kernel
  - Pre-handler runs, execute probed instruction, post-handler runs
- Three flavors
  - Kprobe
  - Jprobe
  - Kretprobe
- Kprobe utility lacks ability to dynamically replace any kernel function.
  - Introducing Iprobes – Hybrid of Kprobe and Jprobe

VPCI Bus: Iprobes

- Use Jprobe to check function parameters
- If caller flagged as accessing VPCI device, intercept
  - Temporarily replace probed instruction with *nop*
  - Kprobe post-handler modifies the IP to address of replacement function.
- Most VPCIB probes are Iprobes, a few are Jprobes

- Caveat: Probes cannot be attached to inline functions or functions declared with __*kprobes*

VPCI Bus: Interrupts

- When a DMA buffer is finished, an IRQ must be raised so the DD interrupt handler is notified

- How do you throw a hardware IRQ from software?
  - Intel x86 instruction *int* is for generating software interrupts

  - x86 interrupt mapping starts at 32
  - x86_64 interrupt mapping starts at 48

Detecting Writes to VGPU Registers

- Added VPCI Page Fault to the page fault handler (*do_page_fault()*)
- When device driver requests the memory to be mapped with write permissions
  - VPCIB intercepts
  - sets the memory to read-only; flush TLB
  - activates the fault handler
- When the device driver/user code writes to the memory
  - Page fault exception
  - *do_page_fault()* hands control to the VPCI fault handler.
  - VPCI fault handler restores write permission, flush TLB, wakes up the VCD, and returns execution to spot of the fault

VGPU

- Allocates a kernel page
- Registers the device with VPCI Bus
  - Gives memory address, IRQ number, device IDs for use with Iprobes
- In response to fault handler
  - Wake up daemon
- In response to VCD
  - When done processing registers, write protect the register page, flush TLB, sleep
  - Generate interrupt if DMA is running, sleep until device driver interrupt handler finishes

VCD

Map register page from VGPU

Initialize registers

Use ioctl() to sleep

Forever {

 if (register values changed) take action

  e.g. start graphics mode, draw to the screen, initiate DMA

 Use ioctl() to write protect the page and sleep

}

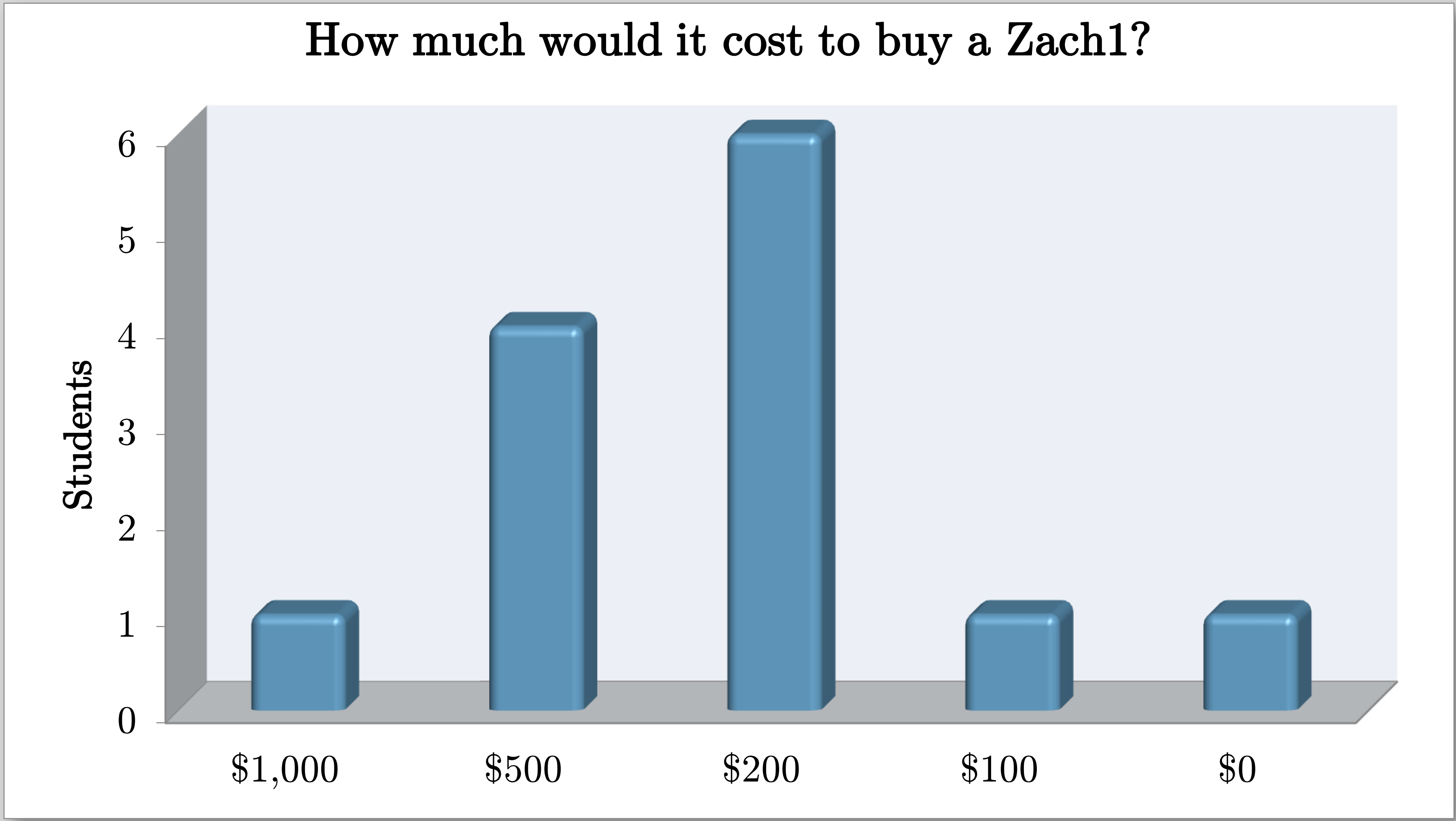
Status

- Successfully deployed in CPSC 822 for 3 semesters
  - Spring 2009 – physical lab machines
  - Fall 2009, Spring 2010 – virtual lab machines
- Kernel Modifications:
  - Most are avoided by kernel probes
  - Remove *__kprobes* from *do_page_fault()*
- Devices:
  - Can write binary compatible device drivers for use with real hardware
- Allow for parallel development of DD and HW
- TODO: Create device generation tools.

Performance

- Rendering
- Students
  - All teams have been able to complete FIFO drivers
  - Teams still have difficulty with SMP-safe drivers

Transparency to Students



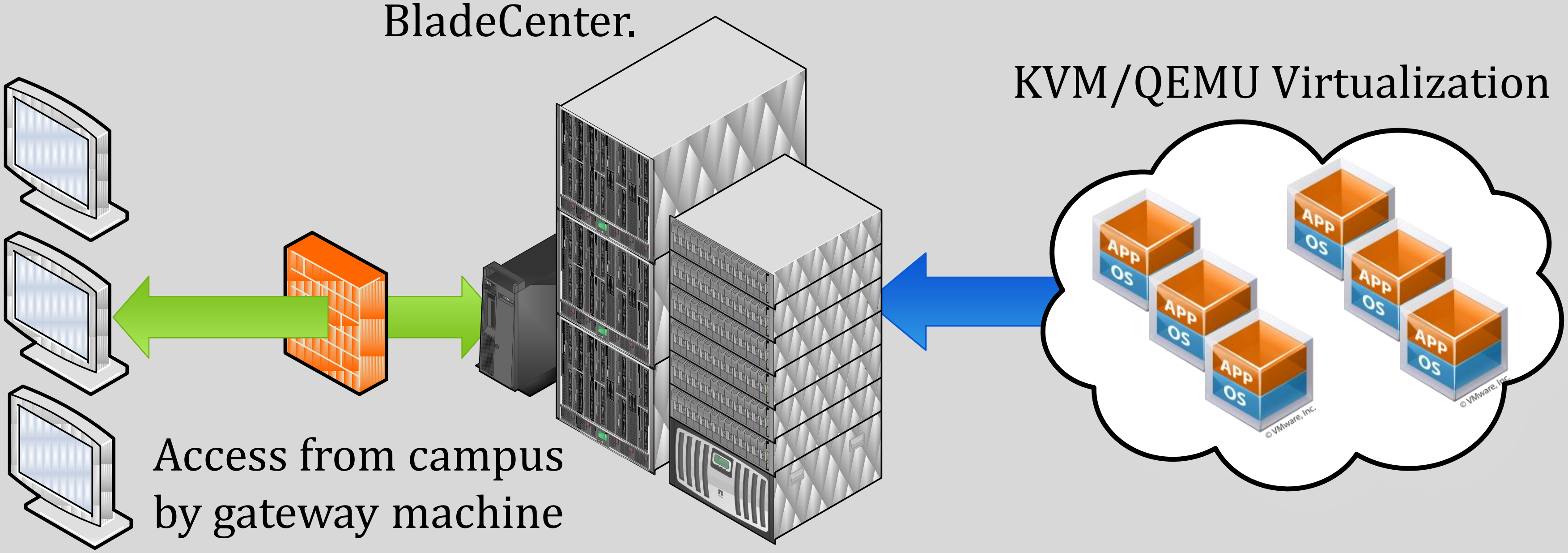
Real Challenges

- Device Driver Development
  - Write real device drivers on virtual devices
- **Virtual Lab for CPSC 822**
  - Real OS and performance work on virtual hardware
- Disk Scheduling
  - Model real hardware in a virtual machine

The Virtual Lab

VMs run on 6 dual-Xeon blades.
Part of 42-blade eServer BladeCenter.

KVM/QEMU Virtualization



Access from campus by gateway machine via SSH.

VMs stored on NAS, accessed by NFS.

Custom scripts used for managing and accessing lab.

Virtual Machines

- Create a virtual disk
  - *qemu-img create –f qcow2 822master.img*
- Install OS into virtual machine
  - *qemu-system-x86_64 –drive file=822master.img,if=scsi,bus=0,unit=0 –cdrom CentOS_64-5.3.iso –boot d –m 1024*
- Regular boot sequence
  - *qemu-system-x86_64 –drive file=822master.img,if=scsi,bus=0,unit=0 –m 1024*

Accessing the Lab

- BladeCenter on private network behind gateway machine
  - Students SSH into gateway machine
  - Accounts provided for each team
- *go_blue*
  - Static load-balancing across blades
- *start_lab_vm*
  - Adds a lock file mechanism to keep students from starting two VMs with one hard drive
  - Invokes the standard VM boot command
- X11 Forwarding
  - SSH Tunneling “easier" to set up but uses more CPU resources on gateway than Forwarding

Virtual Disks

- Independent virtual disks
  - Each lab machine would use ~6GB disk space
  - ~78GB of storage space for13VMs
- Clone images
  - Use one master image; use clone images for each lab machine
  - Each clone image reads files from master image
  - A file write copies the file to the clone
  - Access from clone after that point
- Performance considerations
  - 1 master for 13 clones saves ~72GB
  - Two reads to access blocks when present only in master. (check for block in clone, then read from master)

Timing

- Linux kernel in CentOS 5.3 uses the *jiffies* counter and a PIT to attempt to keep time
- Under heavy load, guest VM clock can skew
  - Full Linux kernel build results in +/- 60 seconds skew from host time
- Students cannot accurately measure performance
- *make* system becomes confused

Paravirtualized Clock

- Paravirtualized clock introduced into mainline Linux kernel 2.6.26
- When compiled into the kernel (*CONFIG_PARAVIRT_CLOCK=Y*), the guest kernel will receive system time updates from the hypervisor
- Accuracy (~1 millisecond) is more than adequate for class needs

Performance

- Virtualization is not free
- KVM uses Intel VT or AMD-V, but performance penalty may be incurred through use of QEMU
- Host system may migrate VMs among different cores on the system
- Linux utility *taskset* allows for setting which virtual CPUs will execute a process
  - *taskset 0x3 cfdlight8 ii9.ex.perked > out.lit*
- A virtual machine is another process
  - *taskset 0x3 qemu-system-x86_64*
    –drive file=822master.img,if=scsi,bus=0,unit=0 –m 1024

Performance Results

Time in seconds	Real	Virtual
Unpinned	1634.56	1678.88
Pinned	1651.86	1652.68

- Without pinning, penalty of 2.71%
- With pinning, penalty of 0.05%

Status

- Virtual lab for CPSC 822:
  - KVM-based, hosted on 6 dual-Xeon blades
  - Peaked at 18 VMs (26 graduate students)
- Simple, customs scripts for management and allocation
- Still maintain relatively good performance of lab machines

Real Challenges

- Device Driver Development
  - Write real device drivers on virtual devices
- Virtual Lab for CPSC 822
  - Real OS and performance work on virtual hardware
- **Disk Scheduling**
  - Model real hardware in a virtual machine

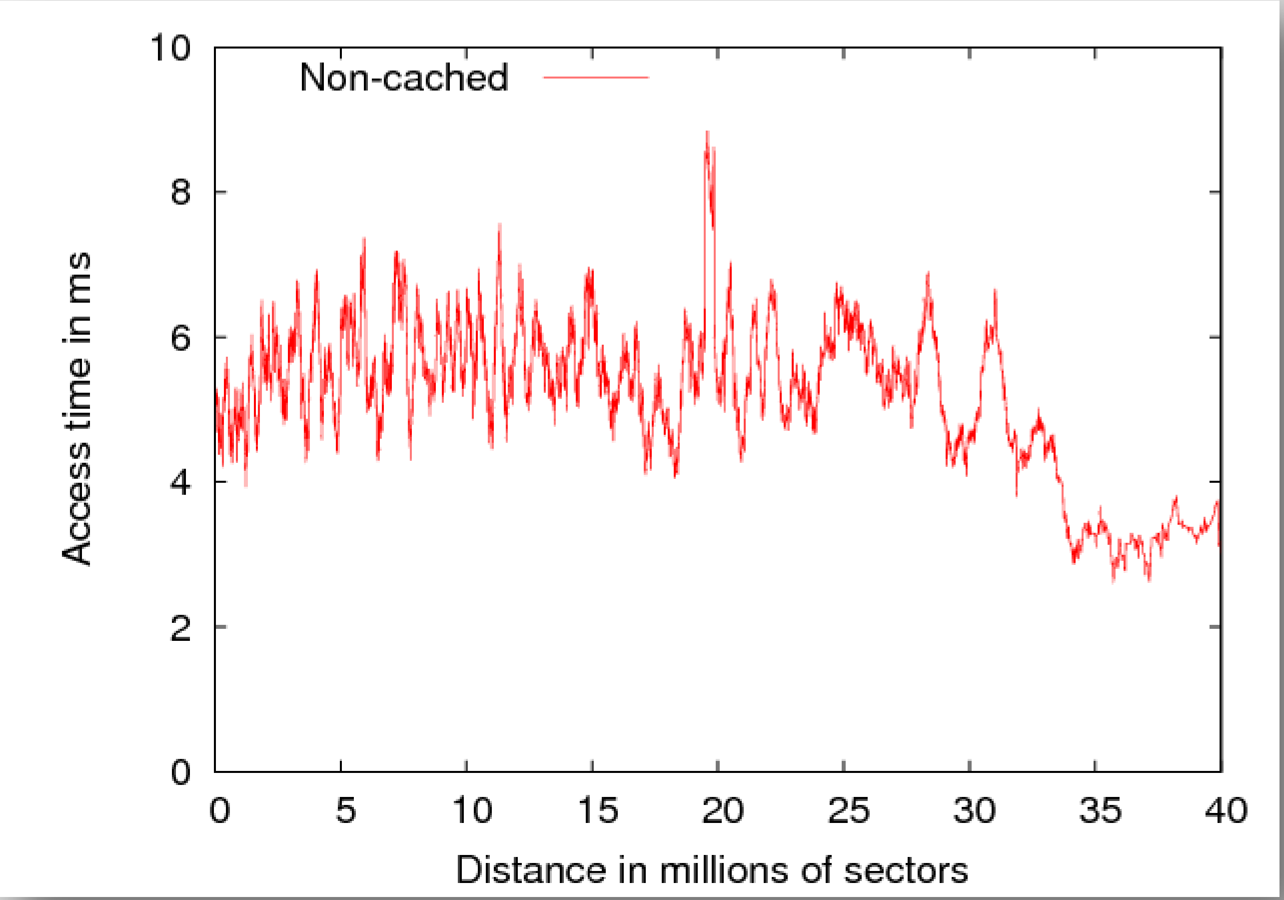
Disk Scheduling

- Design a new scheduler with increased performance under targeted class of workloads
- Development can be carried out easily on virtual machines.
- Measuring performance of virtual disks does not translate to performance of physical disks.
- Problem is in abstraction.

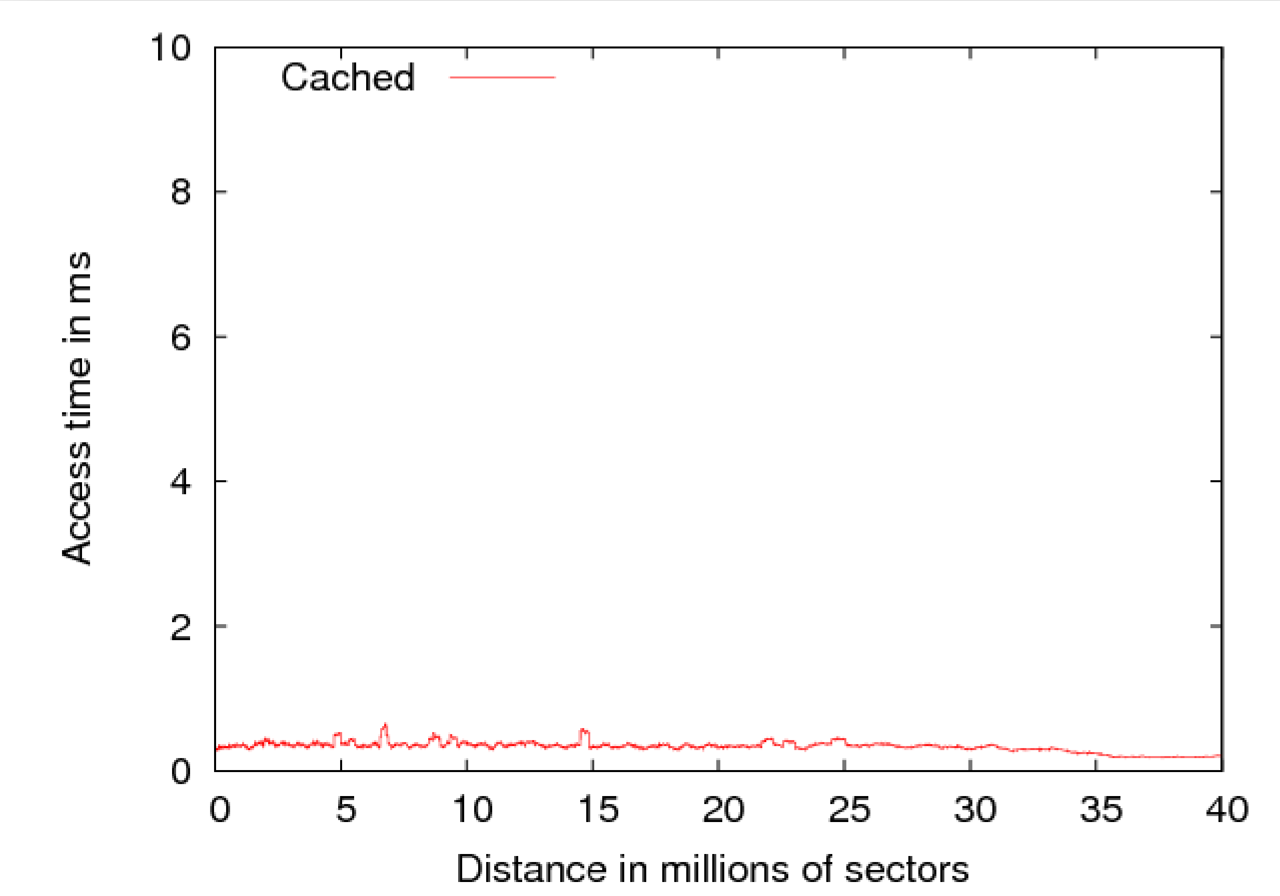
Virtual-to-Physical Abstraction

- No mapping between virtual and physical blocks.
  - VM translates a block request to a location in the virtual disk image (a file) and requests the block from that file.
  - Request travels across the network to NAS device
  - Request travels the request path on NAS device to the particular disk(s) where the blocks of file reside.
- Access speeds to virtual disk will never match a real device
- Solution? Paravirtualized Disk
  - Removes some but not all abstraction
  - Does not resolve multiple VMs on one disk
  - Does not address problems of SAN and NAS

NAS Consistency Problem



NAS Consistency Problem



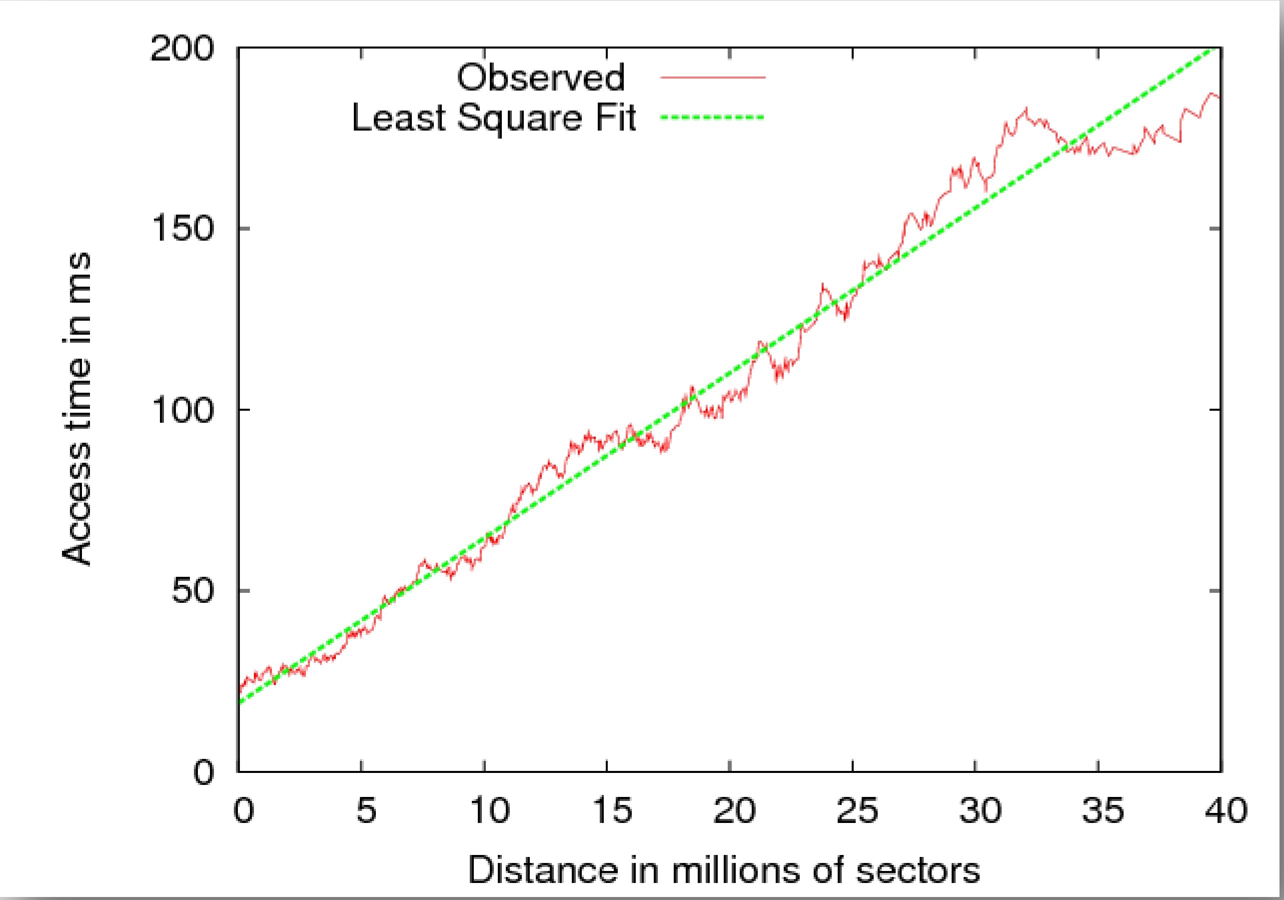
Virtual Performance Throttles

- We introduce VPTs to overcome these limitations.
- Given a linear seek time performance model of a physical disk, we can find the time to complete a request, T.
- We scale the performance model by a constant factor to overcome abstraction, k
- We know the time it takes for a request to complete on the virtual disk, V.
- We delay the request by $k*T - V$ time.
- Dynamically scale k to adjust for virtual disk performance.

Proposed Implementation

- Attach Jprobe to down path of the disk request
  - Calculate the distance traveled from last sector to this one
  - Calculate target completion time, $k*T$.
- Attach Iprobe to up path of the disk request.
  - If current time < $k*T$ place request on a queue
  - Else missed request, k needs to be increased.
- Timer
  - Responsible for draining queue of requests once the requests reach their target completion times
  - If requests in queue for "too long", then k needs to be lowered

Proof of Concept VPT



Status

- Proof of concept VPT exists
- CPSC 822 Disk Scheduling project is only a few weeks away from being assigned
- TODO:
  - Target a VPT to a linear performance model
  - Add dynamic scaling
  - Incorporate a target disk cache model into VPT

Conclusion

- Solutions to *real* challenges
  - Framework for device driver development on virtual hardware
    - Binary compatible device drivers
    - Almost no changes to Linux Kernel
  - Virtual Lab for CPSC 822
    - Resource constraints for class removed
    - Timing and performance are acceptable for our need
  - Framework for performance modeling on virtual hardware
    - Extends Virtual Device Architecture
    - Proof of concept exists
- Framework for “Linux as a Simulator”

Questions