A Framework for Virtual Device Driver Development and Virtual Device-Based Performance Modeling

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

by
Zachary Harrison Jones
December 2010

Accepted by:
Robert M. Geist, III, Committee Chair
James M. Westall
Brian C. Dean
Harold C. Grossman

Abstract

Operating system virtualization tools such as VMWare, XEN, and Linux KVM export only minimally capable SVGA graphics adapters. This paper describes the design and implementation of system that virtualizes high-performance graphics cards of arbitrary design to support the construction of authentic device drivers. Drivers written for the virtual cards can be used verbatim, without special function calls or kernel modifications, as drivers for real cards, should real cards of the same design exist. While this allows for arbitrary design, it is not able to model performance characteristics. We describe a new kernel system that allows for arbitrarily changing the performance of a device. These virtual performance throttles (VPTs) use the framework provided by the virtual device architecture and a simple linear service model a physical drive to simulate the relative performance characteristics of the physical disk. The applications of the system include instruction in device driver and disk scheduler design, allowing device driver design to proceed in parallel with new hardware development, and for relative performance measurements without needing access to the physical device being modeled.

Dedication

This work is dedicated to my family: my parents, my brother, my grandmothers, and many others. Their love, encouragement, belief, and patience inspired and enabled me to see this work through to completion.

Acknowledgments

This work was supported in part by two IBM Faculty Awards, by an IBM Ph.D. Fellowship, and by the U.S. National Science Foundation under Award 0722313. I would like to acknowledge the people listed below; this research effort would not be possible without their efforts:

1. Robert Geist, my advisor and James Westall. They are tremendously brilliant individuals who I am deeply indebted to for providing challenging and rewarding research.

2. My committee, for their guidance and insight and for providing such a rewarding time in graduate school.

3. The technical and administrative staff at the School of Computing, for all of the work they did on my behalf.

4. William Pressly and Jay Steele, for all of the assistance and fellowship they provided.

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

In this dissertation, we provide solutions to problems in operating system virtualization that have been motivated by major projects in Computer Science 822 (CPSC 822) at Clemson University. Computer Science 822, Operating System Design: A Case Study, has been offered as an advanced, graduate course in operating systems at Clemson University since 1985. The hardware platform and the operating system have changed through the years (currently Linux 2.6.30 on Intel hardware), but the structure and the principal thrust have remained the same. It is a walk-through of the source of a UNIX derivative in which the students modify schedulers to improve performance, build new kernels with additional system call capabilities, and write device drivers for real devices. Students have found the course extremely valuable in advancing their understanding of system software design, and as a result, their opportunities for employment in the systems industry.

Nevertheless, the total impact of the course, in terms of the number of students served, has been limited by available resources. Students who have completed the course have always been in high demand by the systems industry, but total student production (course throughput) at Clemson University has been severely limited by the available budget. Students writing system-level code frequently crash their systems, often with disk-corrupting failures, and thus the course has always required both dedicated hardware and the laboratory space in which to house it, which collectively represents a relatively large expense for a single course.

The recent, hardware-enabled move to system virtualization, typified by VMWare, XEN, and Linux KVM, offers great potential to expand course impact. A large number of virtual machines can be hosted on a relatively small number of physical machines, and repairing a crashed virtual machine requires only a simple file copy, rather than a complete re-install of the operating system. Many course components, e.g., new kernel builds and scheduler experiments, could be directly handled by any of the virtualization tools. Nevertheless, two key course projects have remained out of reach of such tools. One project is building a device driver for a high-performance graphics card. Most graphics cards have proprietary interfaces, and their manufacturers supply only binary drivers. Unlike disks with standard IDE, SATA, or SCSI interfaces, or CPUs with standard instruction sets (e.g. Intel x86), there is no industry standard interface for graphics cards beyond that of the minimally capable SVGA, which is exactly what the virtualization tools export. Even if specific, high-performance graphics cards were recognized and exported by the virtualization tools, implementation would be limited to platforms with those cards. Effective platform expansion requires the availability of a virtual, high-performance card architecture that can be exported from a heterogeneous collection of generic PCs or server blades.

In designing such an architecture, we have identified four design goals. First, the virtual architecture must support drivers that require sophisticated, system-level components, in particular, scheduling, memory mapping, DMA, and interrupt handling. Second, driver design for this virtual architecture should require no specialized function calls to ensure that driver design is authentic. The same Linux kernel functions used to access the real hardware, e.g., *pci_register_driver()*, should be used, verbatim, to access the virtual hardware. Third, no modifications to the standard Linux kernel are allowed. Functionality must be encapsulated in drop-in kernel modules, the standard tool for dynamic kernel extensions and most device drivers. Finally, the system must be easily reconfigurable, so that different (virtual) card architectures can be quickly designed and implemented.

Design and implementation are complete, and details are described herein. With the virtual device architecture, we have removed the principal roadblock to including the device driver development project in a course supported entirely by virtual systems. We also note that the virtual architecture allows concurrent development of new hardware designs and supporting system software. Device drivers can be ready when the first device arrives from fabrication.

The second key, motivating project from CPSC 822 is writing a disk scheduler of a new design. CPU speeds have increased by orders of magnitude over last 10 years, but disk speeds are essentially unchanged. Thus, disks have become common performance bottlenecks and any improvement in access times, say through scheduling, can offer substantial benefits. Disk scheduler development can be carried out easily on a virtual machine, but the goal of any new scheduler is improved performance. Measuring the performance of virtual disks in a way that would allow prediction of the performance of real disks is a difficult problem due to the layers of abstraction between the virtual disk and physical disk. In a virtual machine, the virtual disk may actually reside on a Network-Attached Storage (NAS) system across multiple disks. Thus, the access speed to the virtual disk will never match that of a targeted real device. Our goal is to provide a kernel module that will accept a minimal description of a targeted, real disk drive and then intercept and modulate the performance of the system virtual drive to match. We introduce Virtual Performance Throttles (VPTs), a method to use kernel modules to achieve this design without kernel modification. As we will see in Chapter 4, a simple model of any physical disk can be created and used in configuring the VPT. We extend the virtual device architecture to alter the performance of the disk request path in the kernel. This design, in effect, incorporates an entire operating system as a simulation tool.

Finally, we describe the deployment of virtual CPSC 822, where lab machines are no longer physical machines in a dedicated lab but are virtual machines residing in an IBM BladeCenter cluster and the hard drives are stored on a NetApp FAS960c Network Attached Server accessed over NFS.

1.1 Organization of this Document

In the next chapter we briefly describe related work in virtualization of both operating systems and access to hardware-accelerated graphics. We also include background on Linux kernel modules, character devices and kernel probes, the principal tools used in our implementation. We end the chapter with an introduction to disk scheduling. In Section 3.1, we provide an overview of the virtual architecture of our system that allows driver development for virtual graphics cards. It is composed of three interacting code modules: one at the user level and two at the kernel level. Sections 3.2 and 3.3 describe two very different performance evaluations of our system: the former evaluates the rendering performance of a virtual graphics card, and the latter evaluates the performance of a class of graduate students who were given the task of writing drivers for a virtual graphics card. In Sections 4.1, 4.2, and 4.3, we discuss the modeling of a physical disk and implementation of the model in a VPT. In Section 4.4 we present a case study and discuss results that compares the performance of a VPT modeled drive to a targeted physical drive under a variety of scheduling algorithms. In Chapter 5, we provide an overview of the lab environment deployed for the current virtual Computer Science 822 class, discuss several challenges encountered and their solutions, and introduce a utility to aid in creating new virtual PCI devices. Conclusions follow in Chapter 6.

Chapter 2

Background

2.1 Virtualization

System virtualization has been of interest to the computing community since at least the mid-1960s, when IBM developed the CP/CMS (Control Program/Conversational Monitor System or Cambridge Monitor System) for the IBM 360/67 [8]. In this original design, a low-level software system called a *hypervisor* or *virtual machine monitor* sits between the hardware and multiple guest operating systems, each of which runs unmodified. The hypervisor handles scheduling and memory management. *Privileged* instructions, those that trap if executed in user mode, are simulated by the hypervisor's trap handlers when executed by a guest OS.

Aspects of the architecture of the host machine affect the difficulty of constructing a secure and efficient hypervisor. These elements are described from a somewhat formal perspective by Popek and Goldberg [23]. They characterize as *sensitive* those instructions that may modify or read resource configuration data. They show that an architecture is most readily virtualized if the sensitive instructions are a subset of the privileged instructions.

In the x86 architecture, a relatively large collection of instructions are sensitive but not privileged. Therefore, a guest OS running at privilege level 3 may execute one of them without generating a trap that would allow the hypervisor to virtualize the effect of the instruction. A detailed analysis of the challenges presented by these instructions is presented by Robin and Irvine [25]. For completeness, we include two such examples here:

1. Because reading x86 system configuration registers is not privileged, a guest OS may read and store the contents of the CS (code segment) register, which contains the privilege level. Upon inspection of the saved value, the guest OS could see that the kernel is actually executing at privilege level 3, instead of the expected level 0, and incorrectly infer that a catastrophic failure has occurred. Similarly, the Linux kernel function *do_signal()* tests the saved CS register of the caller and takes different paths based on its value. An unmodified guest Linux would always see the same value and then sometimes take the incorrect path.

2. When the processor is executing at privilege level 0, POPF (pop flags) can modify both the I/O privilege level and the interrupt enable flag. However, when executed by a guest OS at privilege level 3, changes to the I/O privilege level and the interrupt enable flag are simply suppressed. When this occurs, the guest OS and hypervisor may have inconsistent views of whether or not interrupts can be delivered to the virtual machine.

The designers of VMWare provided the first solution to this trapping problem by using a binary translation of guest OS code [31]. In another approach, Xen [2] provided an open-source virtualization of the x86 using *paravirtualization*, in which the hypervisor provides a virtual machine interface that is similar to the hardware interface but avoids the instructions whose virtualization would be problematic. Because those instructions are avoided, each guest OS must then be modified to run on the virtual machine interface.

Much of the difficulty of virtualizing the x86 architecture was removed with the 2005 and 2006 extensions to the architecture, the Intel VT-x and AMD-V. The extensions include a "guest" operating mode, which carries all the privilege levels of the normal operating mode, except that system software can request that certain instructions be trapped. The hardware state switch to/from guest mode includes control registers, segment registers, and instruction pointer. Exit from guest mode includes the cause of the exit, which informs the hypervisor how to proceed. These extensions have allowed the development of a full virtualization Xen, in which the guest operating systems can run unmodified, and the Kernel-based Virtual Machine (KVM) [18], which uses a standard Linux kernel as hypervisor. The KVM-supported kernel includes a character device, (*/dev/kvm*) whose *ioctl()* calls can create new virtual machines, allocate virtual machine memory, read and write virtual CPU registers, and inject interrupts to and run virtual CPUs.

Nevertheless, VMWare, Xen, and KVM are inadequate for a project in graphics card driver design because they export only basic, SVGA graphics cards to the guest systems. With Workstation 6.5 and newer, VMWare does support hardware-accelerated graphics in Windows XP guests, but this is virtualization at the graphics API level, not the card level. With the lack of standardization and proprietary interfaces for GPUs, virtualization at the graphics API level, rather than the architecture level, has become the focus area of rapid development. The VMGL system [20] allows hardware accelerated OpenGL applications to run inside virtual machines provided by any of VMWare, Xen, or KVM, and it works with ATI, Intel, or NVIDIA cards. VMGL uses the machine's loopback interface and a transport based on WireGL [17]. It is similar in spirit to Virtual GL [29], which allows low-cost, remote visualization by rendering on highly accelerated servers and then, through a suitable transport, pushing pixels to less capable clients. Both systems probably trace their origins to Stegmaier et al [27].

Thus, although we can access the performance of a high-speed graphics card within virtual machines, we cannot, through available tools, access the architecture of such a card, which is the goal of device driver development.

2.2 Kernel Modules and Character Devices.

Both the Virtual Architecture, described in Chapter 3, and the Virtual Performance Throttle described in Chapter 4, make extensive use of the Linux *kernel module* facility. Kernel modules are collections of functions that can be dynamically loaded to extend the capabilities of a running base kernel. Two of the functions in the collection are identified as special: the *module_init()* function is executed when the module is dynamically loaded and the *module_exit()* function is executed when it is removed. Modules can export their functionality to the running base kernel (or other modules) via an *EXPORT_SYMBOL()* macro.

The structure of the collection of functions that comprise the module is otherwise arbitrary, but in practice the most common design is probably one that structures the module as a collection of file operations on a special type of file, called a *character device*. Character devices are created by the *mknod* command, which takes a target name and a target device number as arguments. The device number usually corresponds to the device identifier that is on-board a physical controller or card, but it need not, as character devices can be entirely logical constructs. The file operations that operate on character devices have fixed signatures (specified in the kernel include file, fs.h) and are invoked by corresponding system calls from the user level, but their implementation is at the discretion of the module designer. The most commonly implemented file operations are *open*, *release*, *mmap*, and *ioctl*. The *ioctl()* call is particularly useful, in that one of its arguments is a command identifier, which can be used in a module *switch()* statement to provide a wide variety of capabilities.

The *module_init()* function typically connects the module's file operations to the character device structure (*struct cdev*) via the kernel's *cdev_init()* function and connects the character device number to this same structure with *cdev_add()*. It can then invoke a scan of the PCI bus in search of a physical card with the target device number by a call to *pci_register_driver()*. On success, the scan will provide an address from which key controller or card information, e.g. physical base addresses, memory sizes, and IRQ lines can be read and stored in the module's structures.

2.3 Kernel Probes

The kernel probe or *kprobe* utility was designed to facilitate kernel debugging [22]. It first appeared in Linux kernel 2.6.9 and is fully supported in i386, x86_64, IA64, and Power architectures. This utility has evolved over time and now allows for multiple probes to be attached to the same point in the kernel, multiple probes per CPU, and multiple instances of the of the same probe running on different CPUs.

All *kprobes* have the same basic operation. A *kprobe* structure is initialized, usually by a kernel module, to identify a target (kernel) instruction and specify pre-handler and post-handler functions. When the *kprobe* is registered, it copies the target instruction and replaces it with a breakpoint. When the breakpoint is hit, the pre-handler is executed, then the copied instruction is executed in single step mode, then the post-handler is executed. Finally, a return resumes execution after the breakpoint. There are two variations of the *kprobe* supplied with Linux: the *jprobe* and the *kretprobe*.

The *jprobe* or jump probe is intended for probing function calls, rather than arbitrary kernel instructions. Conceptually, it is a *kprobe* with a two-stage pre-handler and an empty post-handler. On registration, it copies the first instruction of the registered function and replaces that with the breakpoint. When this breakpoint is hit, the first-stage pre-handler, which is fixed, is invoked. It copies both registers and stack, in addition to loading the saved instruction pointer with the address of the supplied, second-stage pre-handler. The second-stage pre-handler then sees the same register values and stack as the original function.

The kernel return probe or *kretprobe* is intended for probing the return value of a function call. A *kretprobe* is attached to a function at its entry point and when this function is called, the probe point is immediately hit. A special pre-handler saves the return address of the caller and replaces it with the address of *kretprobe_trampoline()*. When the function reaches the return, *kretprobe_trampoline()* hands control over to the *kretprobe* handler supplied by the user and sets its return address to the original caller.

While most functions can be probed in the kernel, there are two exceptions: inline functions and functions declared with the *__kprobes* qualifier. Inline functions are inserted into the body of caller function at compile time. Thus, they are not present in the compiled version of the kernel and the *kprobe* utility is unable to find them. The *__kprobes* qualifier instructs the compiler to place a function in a location in memory where the *kprobe* utility is forbidden from attaching probes. This forbidden area is where most of the *kprobe* utility itself and certain other functions, which the kernel developers have decided would be hazardous to probe, all reside.

Although the *kprobe*, *jprobe*, and *kretprobe* utilities are quite useful and flexible, none is adequate for the fundamental task required by the design of our virtual architecture, namely, dynamic replacement of an entire kernel function with a custom version. Thus we have designed a new type of probe, the intercept probe, which accomplishes this task. It we will be described in Section 3.1.2.

2.4 Disk Scheduling

Scheduling algorithms that re-order pending requests for disks have been studied for at least four decades. Such scheduling algorithms represent a particularly attractive area for investigation in that the algorithms are not constrained to be *work-conserving*. A scheduling algorithm is work-conserving if: 1) individual customer service demands are no affected by scheduling order and 2) the server is not idle when there is work pending.

For all such scheduling algorithms, which are most common withing operating systems, it can be shown that the mean of the product of service time and wait time, $\mathbf{E}[SW]$, is constant [19].

algorithm	mean service	mean response
greedy 82, 120, 200, 20	79.0	131.5
optimal 120, 82, 20, 200	75.0	124.5

Table 2.1: Sub-optimal performance of the greedy algorithm.

This is a severe limitation on the performance (service time, wait time, response time, throughput) is improved for one class of customers, another class will suffer. Since disk scheduling algorithms are not constrained to be work-conserving, the severe limitation does not apply, and performance improvement through scheduling can be wide-spread and dramatic.

Further, it is easy to dismiss naive (but commonly held) beliefs about such scheduling, in particular, that a greedy or shortest-access-time-first algorithm will deliver performance that is optimal with respect to any common performance measure, such as mean service time or mean response time. Consider a hypothetical system in which requests are identified by their starting blocks and service time between blocks is equal to distance. Suppose the read/write head is on block 100 and requests in queue are for blocks 20, 82, 120, and 200. The greedy schedule and the differing, optimal schedule are shown in Table 2.1.

Disk scheduling algorithms are well-known to be analytically intractable with respect to estimating response time moments. Early successes in this area, due to Coffman and Hofri [6] and Coffman and Gilbert [5] were restricted to highly idealized, *polling* servers, in which the read/write head sweeps back and forth across all the cylinders, without regard to the extent of the requests that are actually queued.

Almost all knowledge of the performance of real schedulers is derived from simulation and measurement studies. Geist and Daniel described UNIX system measurements of the performance of a collection of "mixture" algorithms that blended scanning and greedy behavior [12]. Geist, Reynolds, and Pittard [13] measured the performance of such mixture algorithms under UNIX System V and observed that the choice of scheduler dramatically affected the arrival process as seen at the drive. Worthington, Ganger, and Patt [32] showed, in simulation, that scheduling with full knowledge of disk subsystem timing delays, including rotational delays and on-board cache operations, could offer major performance improvements. They also concluded that knowledge of the on-board cache operation was far more important than an accurate mapping of logical block to physical sector. We will contend, in the case study of Chapter 4, that the on-board cache has relatively little effect on scheduler performance for most workloads.

More recently, Pratt and Heger [24] provided a comparison of the four schedulers distributed with Linux 2.6 kernels. Prior to 2.6, Linux used a uni-directional or circular scan (CSCAN), in which requests are served in ascending order of logical block number until none remains, whereupon the read/write head sweeps back down to the lowest-numbered pending request. With recognition that the best scheduler is likely workload-dependent, Linux authors changed the 2.6 kernel to allow single-file, modular, drop-in schedulers that could be dynamically switched. Four schedulers were provided. The default is the completely fair queuing (*cfq*) algorithm, which has origins in network scheduling. Each process has its own logical queue, and requests at the front of each queue are batched, sorted and served. The *deadline* scheduler was designed to limit response time variance. Each request sits in two queues, one sorted by CSCAN order, one FIFO, and each has a deadline. The CSCAN order is used, unless a deadline would be violated, and then FIFO is used. As with many algorithms, reads are separated from and given priority over writes because the requesting read process has usually suspended to await I/O completion, and so there are actually four (logical) queues. The *anticipatory* scheduler is no longer supported, and the *noop* scheduler is essentially FIFO, which delivers poor performance on almost all workloads, and thus these two will not be discussed.

A fundamental departure from greedy algorithms, scanning algorithms, and $O(N)$ mixtures thereof was offered by Geist and Ross [11]. They observed that, over the preceding decades, CPU speeds had increased by several orders of magnitude while disk speeds remained essentially unchanged. They suggested that $O(N^2)$ algorithms might be competitive and offered a statically optimal solution that was based on Bellman's *Dynamic Programming* [4], in which a table of size $O(N^2)$ containing optimal completion sequences was constructed. Although their algorithm was shown to deliver excellent performance in tests on a real system, there were two easily-identifiable problems. It ignored the dynamics of the arrival process, and it ignored the effects of any on-board disk cache.

Motivated by the (now-defunct) anticipatory scheduler, Geist, Steele, and Westall [15] partially addressed the issue of arrival dynamics. Although at first glance counter-intuitive, it is often beneficial for disk schedulers with a non-empty queue of pending requests to do nothing at all. The process that issued the most recently serviced request is often likely to issue another request for a nearby sector, and that request could be served with almost no additional effort. They added a so-called *busdriver* delay, to mimic the actions of a bus driver who would wait at a stop for additional riders, to the table-based, dynamic programming algorithm of Geist and Ross, and showed that it delivered excellent performance, superior to any of the four schedulers distributed with Linux 2.6, for a fairly generic, web file-server workload designed by Barford and Crovella [1].

Chapter 3

Virtual PCI Framework

3.1 Virtual Architecture.

We now describe the architecture which supports a sophisticated virtual graphics card that is available withing the Linux kernel for driver development. The virtual card delivers real graphics output, and yet its deign can be changed quickly and easily. The Virtual Architecture comprises three interacting code modules, shown in Figure 3.1, to replace the standard graphics cards: the Virtual Console Daemon, the Virtual GPU, and the Virtual PCI Bus.

3.1.1 Virtual Console Daemon

The Virtual Console Daemon (VCD) is a user-level process that simply reads the virtual device registers, which are located in kernel memory that is part of the Virtual Graphical Processing Unit (VGPU), and updates the display accordingly. The read operation could be executed via a standard system call (*ioctl()* on the VGPU device), but it is faster to memory map the virtual device registers of the VGPU back to the user space of the VCD and read them directly in user space. To avoid busy-waiting on virtual register updates, the VCD will suspend on a kernel wait queue if it detects no register changes since its last read. It simulates entry to and exit from graphics mode by starting and stopping an XWindows server. An X server serves this purpose well as it can easily be configured to run without borders or icons, which gives the appearance of an “empty” underlying framebuffer. Graphics primitives are generated by the VCD using a combination of OpenGL and

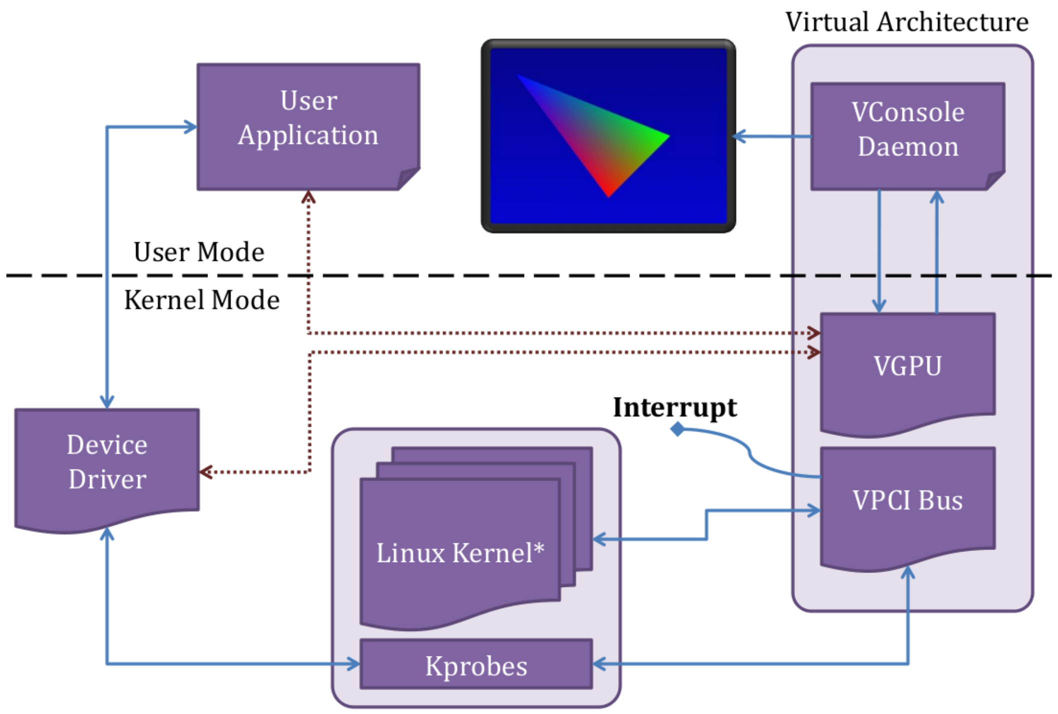


Figure 3.1: Virtual Architecture.

XDraw commands. Thus, although the VCD must incorporate a simulator of the target virtual architecture, it is a functional-level simulator, not a command-level interpreter.

If the VCD detects changes to the DMA registers, located in the kernel memory component of the VGPU, it is responsible for simulating the DMA transfer by reading buffers of (graphics) commands from the device driver and executing them. When a buffer has drained, the VCD (through *ioctl()* must initiate the sequence for the VGPU to generate an interrupt to the device driver, if the driver has enabled DMA-completion interrupts on the VGPU. With the exception of the direct reads of the memory-mapped virtual registers, the VCD communicates with the VGPU through *ioctl()* calls.

3.1.2 Virtual PCI Bus

A long term goal for the project is to allow multiple, simultaneously enabled, virtual PCI devices, and so we have elected to gather common functionality into a single module, the Virtual PCI Bus (VPCIB), with which lightweight, device-specific kernel modules may then register and share in its exported functions. The VPCIB is a Linux kernel module which actually contains most of the functionality, including the intercept probes. Functions exported from VPCIB and executed by VGPU include suspending the VCD on a kernel wait queue, write protecting the page of virtual registers, and generating an interrupt when the VCD makes a buffer completion *ioctl()* call.

With careful use of the pre-handler and post-handler, an entire kernel function can be replaced dynamically with an alternative version. For this task, we modified the *jprobe* utility to create an intercept probe (*iprobe*). The first-stage pre-handler is identical to that of the *jprobe*. It copies the state of the registers and stack of the process context, which was interrupted by the breakpoint and then loads the saved instruction pointer register with the address of the second-stage handler. Our *iprobe* second-stage pre-handler decides whether or not to replace the original function. If it decides to do so, it makes a backup copy of the saved (function entry) instruction and then overwrites the saved instruction with a no-op. As is standard with a *jprobe*, the second-stage pre-handler then executes *jprobe_return()*, which traps again to restore the original register values and stack. The saved instruction (which now could be a no-op) is then executed in single step mode. Next the post-handler runs. On a conventional *jprobe*, this is empty, but on the *iprobe*, the post-handler checks to see if replacement was called for by the second-stage pre-handler. If this is the case, the single-stepped instruction was a no-op. The registers and stack necessarily match those of the original function call. We simply load the instruction pointer with the address of the replacement function, restore the saved instruction from the backup copy (overwrite the no-op), and return. With this method, we can intercept and dynamically replace any kernel function of our choice.

It is possible to have two calls to the same probed function, one that we should intercept and the other that we should ignore. This can lead to an interesting race condition on multiprocessor (SMP) systems which, in worst case, could result in a kernel panic. Recall, the second-stage pre-handler might or might not replace the saved instruction with a no-op instruction. The swap of instructions introduces a small time window in which the first call could affect the second. For instance, suppose the first call is one that installs a replacement of the original kernel function and the second does not. The second call could run through the probe with the no-op instruction still in place from the first call. It would then miss that first instruction of the target function, which it should execute. This can be avoided by acquiring a spinlock in the second-stage pre-handler and releasing it in the post-handler.

We use the *iprobe* to intercept several functions. We intercept *pci_register_driver()*, which device drivers use to scan the PCI bus. Another function we intercept is *dma_alloc_consistent()*, which a driver would call to allocate its own DMA buffers. We intercept this only to capture the buffer addresses, which the VCD will ultimately need to read and execute buffer contents. We also intercept *remap_pfn_range()*. A driver may choose to memory map some or all of its device register space back to the user application's address space. We need to detect writes to the page of virtual device registers, and if this page has been memory mapped to user address space, writes can come from both user virtual addresses and kernel virtual addresses.

The reason we need to detect writes is simply for the VCD wake up mechanism. As noted earlier, when the VCD detects no virtual device register change, it suspends itself through an *ioctl()* call to the VGPU that places it on a kernel wait queue. Within the call, prior to the suspension, it write-protects the page of virtual device registers. The next direct write to the page, either by the driver or by the user application (under memory mapping) generates a page fault. We intercept *do_page_fault()* to test whether the faulting address is, via user page table or kernel page table, within the page of virtual registers. If so, we wake the VCD, make the page writable again, and return, which allows the write to complete.

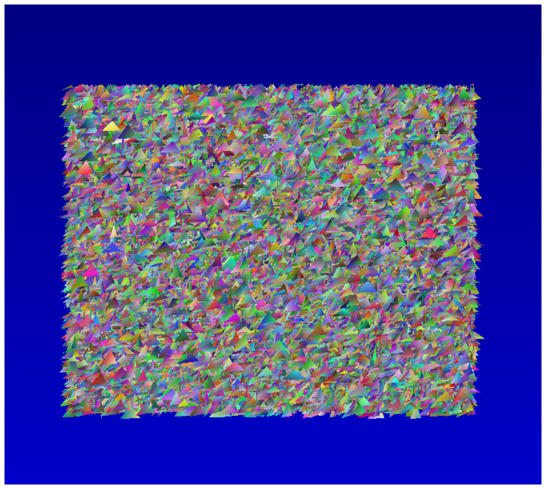
3.1.3 Virtual GPU

The VGPU is another Linux kernel module. On initialization, it allocates a kernel page to hold the virtual device registers. On a real PCI device, the device registers would normally appear at some high physical address found during a driver scan of the PCI bus. The driver would then use *ioremap()* to map this register bank to kernel virtual space for driver use. The VGPU must also register itself with the VPCIB. It passes along information, such as the device IRQ, device IDs, and register locations, needed by the VPCIB to communicate with device drivers. The VGPU is not an active device until it receives communication from the VCD via the *ioctl()* call.

Another race condition can occur in the time when the VCD initiated call to VGPU writes protects the virtual device register space and goes to sleep. After the register space becomes write-protected, a page fault can occur, which hands control over to the *do_page_fault() iprobe*. When this situation occurs, the *iprobe* will always unprotect the virtual device register space, issue a command to wake-up the VGPU and exit, before the VPGU resumes execution. In this instance, the *iprobe* runs as *do_page_fault()* and therefore cannot be interrupted by normal kernel execution. Once the VGPU resumes on the CPU, it will proceed to go to sleep and as a result be in a state where it will never be awakened. The potential system failure can be avoided with two changes. First, we place the critical sections of code, the protect/unprotect, inside of spinlocks. Since we cannot go to sleep while holding a spinlock, our second addition is to use a conditional sleep call, *wait_event_interruptible()*. With this call, the VGPU will only go to sleep when the flag is cleared and only wake up when the flag is set. The wake up flag is cleared in the VGPU critical section, executed by a call from the VCD, before releasing the spinlock. The wake up flag is set in the *do_page_fault()* handler critical section. This prevents the potential system failure associated with this race condition from occurring.

Generating an interrupt in the VGPU is relatively straightforward. On the Intel architecture, the operating system communicates with the Advanced Programmable Interrupt Controller to map hardware interrupt requests (IRQs) into the Interrupt Descriptor Table (IDT). Therefore, we can use the *int n* instruction with $n \geq 32$ or $n \geq 48$ on 32-bit and 64-bit systems respectively to invoke the handler registered for IRQ $n - 32$ or $n - 48$. Thus we can supply the driver with an IRQ of our choice during the intercepted *pci_register_driver()* command and then have the VGPU simulate that interrupt with the *int* instruction whenever the VCD detects an end of buffer.

Again, an interesting race condition arises on a multiprocessor (SMP) system. The interrupt handler in the driver may be updating the DMA registers in the page of virtual registers at the same time the VCD is scanning the virtual registers looking for changes. When the VCD is processing DMA commands with interrupts enabled, it will notify the VGPU to invoke a sleep after a buffer has finished processing. If the VGPU invokes a sleep before the device driver interrupt handler completes, it may not be awakened. There is a two-fold solution for this. We attach a *jprobe* to *request_irq()* that is called when a device driver registers an IRQ handler. In the second-stage pre-handler we register a *kretprobe* on the driver's interrupt handler that was passed in to the *request_irq()* call. This probe will then execute when the handler is finished and will wake up the VGPU.


triangles of height 20 pixels

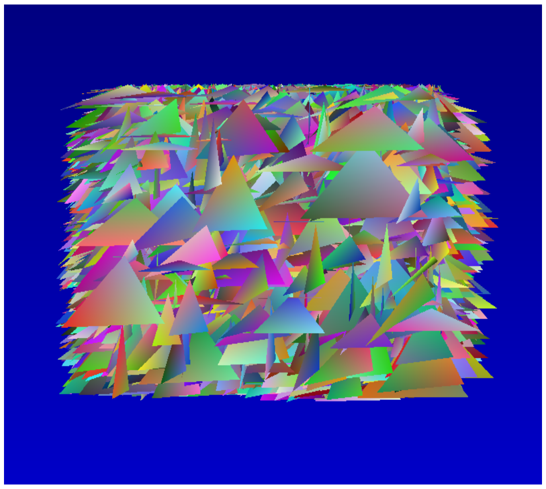

triangles of height 100 pixels

Figure 3.2: Rendering Samples

3.2 Rendering Performance.

The rendering performance of the virtual architecture cannot possibly match that commonly seen from executing directly on hardware GPUs. As noted earlier, the goal of the virtual graphics card project is not to achieve high-speed rendering but rather to provide a completely portable platform for driver design and development. The only issues are whether the penalty is so great that it precludes effective system use and, if not, whether the virtual architecture's rendering performance scales properly with task difficulty.

We conducted a series of tests comparing the rendering performance of a somewhat dated, but 3D hardware-accelerated graphics card, the 3DLabs Permedia 2v, for which the hardware reference manual and programmer's reference manual are available online [28], with a virtual version of the same card, both installed on a Dell Optiplex GX520 with a 2.8GHz Intel Pentium D CPU and 1GB main memory. running a Linux 2.6.26 kernel.

At the user application level, the tests used the card's DMA capability to render 1 million smooth-shaded triangles as quickly as possible where the only variable was triangle size. This rendering test did not make use of all of the registers on the real Permedia 2v card, and so the virtual version could use a reduced register set. The driver for the real card and the driver for the virtual card were thus identical, except for register count, register names, and static values used in register initialization. Screen captures during rendering are shown in Figure 3.2.

Run times were measured from the user application level using the standard Pentium cycle counter capture, *asm("RDTSC")*. Results are shown in Table 3.1. The triangle size is the height

Triangle Size	Real sec.	Virtual sec.	Slowdown Multiplier
20	3.876	136.0	35.17
40	10.71	227.7	21.26
60	20.76	340.4	16.39
80	24.23	476.7	13.93
100	51.03	632.5	12.39

Table 3.1: Rendering Performance Comparison

measured in pixels from the so-called dominant edge, that with maximum y range, to the opposite vertex. We see that the relative performance of the virtual card improves rather rapidly as a greater share of the task effort is shifted toward actual rendering and away from DMA buffer handling, page-fault interception, instruction decoding/interpreting, and interrupt injection. Even the longest rendering time for the virtual architecture, 632.5 seconds, or 1,581 triangles/sec., was judged adequate for driver design purposes.

3.3 Student Performance.

We tested the feasibility of using the virtual architecture for driver design and implementation in CPSC 822 during the first four weeks of Spring semester of 2009. Class lectures during the period focused on Linux kernel modules and principles of driver design. Much of the information can be found in Corbet et al [7]. Student teams composed of 4 graduate students were given hardware reference manuals and programmer reference manuals for the virtual card described in the previous section. The manuals detailed both the capabilities that the driver was to deliver and the interface it was to provide to the application layer. Teams were assigned to specific machines on which we had installed the virtual architecture. They were not told that the card was virtual.

All of the teams delivered an operational driver on time. This was somewhat unusual, compared to the collective performance of teams working on real hardware in previous semesters. In most previous semesters, at least one team had serious driver faults. We tentatively ascribe this to the fact that the virtual hardware is more tolerant of timing errors caused by less than careful saving and restoring of VGA text mode registers. Circumventing these errors is often a time-consuming challenge for students.

Nevertheless, none of the students' drivers was SMP-safe, and this was disappointing. The most common problem was a race condition between the interrupt handler and the driver *ioctl()* code that handled command buffer queuing. Failure to adjust the use of spinlocks to account for the possibility of a rapid succession of multiple buffer completion interrupts could cause a graphics subsystem deadlock. Although this problem is somewhat subtle and rarely occurs during normal operation, in previous semesters at least one team was able to recognize it and handle it.

As a final experiment, we wanted to determine, indirectly, whether the students realized that the graphics card was virtual. We added an extra credit question to the in-class exam that was given in the week following the project deadline. We asked them to estimate the best online price for that model of graphics card for which they had just built a driver. One student clearly realized the card was virtual and answered, "$0". The others gave estimates ranging from $100 to $1,000, with an average above $200. Figure 3.3 gives a distribution of the students answers.

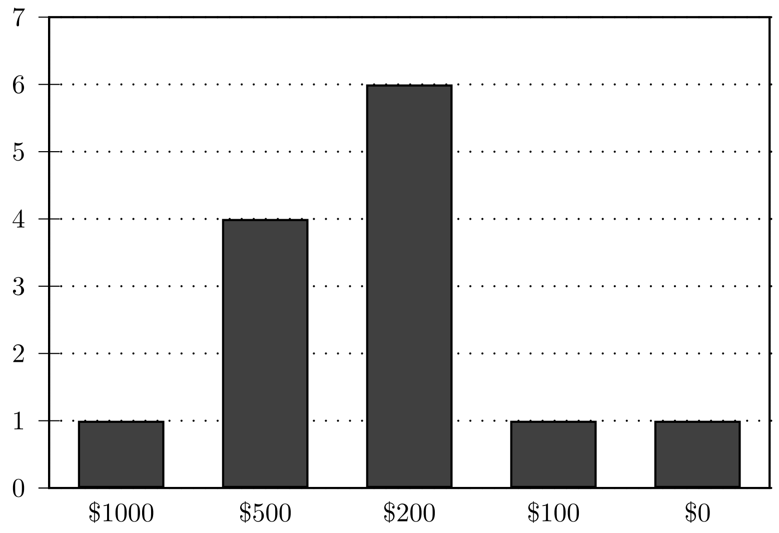


Figure 3.3: Student Evaluation of Price of the Zach1

Chapter 4

Virtual Performance Throttles

4.1 Modeling a Disk Drive

The second key project for CPSC 822 is writing a disk scheduler of a new design. The goal of any new disk scheduler design is increased performance under a targeted class of workloads. Scheduler design and implementation can be carried out easily on a virtual machine. However, measuring the performance of scheduling algorithms on virtual disks in a way that would allow prediction of their performance on real disks continues to be a difficult and important problem. The cause for this lies in abstraction: when a virtual machine requests a block from the virtual disk, the emulator running the virtual machine translates the block request to a location in the virtual disk image (a file) and requests the block from that file. In the case of CPSC 822, the QEMU [3] emulator, which utilizes KVM for accelerated performance, is used. Additionally fro CPSC 822, the virtual disk image files are located on an NFS exported NAS device, and so the virtual machine incurs additional overhead for the request to travel across the network to the NAS device and through the request path on that device to the particular physical disk(s) where the block resides. Of course the disk blocks and even the entire virtual disk may be cached in the main memory of the NAS device.

A possible solution is to use the physical disk option in QEMU. The physical disk option limits the number of virtual machines per server supported, based on the number of physical drives available. Otherwise, again, data corruption would occur when multiple virtual machines write to the same drive. Another solution is to implement a complete disk emulator within the OS or QEMU. Following the design goals from the introduction, we now provide an extremely light-weight emulator facility as a module extension to the Linux kernel, thereby allowing the solution to be portable and minimizing requirements. In this approach we can leverage the tools, particularly the new intercept probes, in the existing framework for development.

4.2 Virtual Performance Throttles

We start by specifying a linear service time model for the physical drive. Linear models are an approximation, but as seen in our case study, the approximation is quite accurate. The expected completion time (T) of a request is one-half the time to complete a revolution (R) plus the product of the maximum seek time (S) and the seek distance x, expressed as a fraction of the maximum seek distance (D). Thus

$$T_r = \frac{R}{2} + S \cdot \frac{x}{D}$$

Although we expect the linear model of service times for a real disk to be a close fit to the observed service times for the targeted real disk, any linear model of service times for a virtual disk may exhibit wide disparity from the observed service times on the virtual disk. Service times on the virtual disk may exhibit significant non-linearities due to non-linear mappings between virtual logical blocks and physical blocks. Further, as noted earlier, with a large, enterprise-class NAS device that contains gigabytes of RAM, it is possible for an entire virtual machine disk to be cached into memory, resulting in constant access time for all virtual logical blocks. Figure 4.1 shows sample average seek times on a virtual disk when it is not cached and the same virtual disk when it is cached on our NAS device. It should be noted that hosting virtual machine disks on Solid State Drives will exhibit similar performance characteristics as the cached virtual machine.

The idea of the virtual performance throttle (VPT) is to insert an *iprobe* into the SCSI path of the virtual system to force virtual service times to be proportional to the real ones, with an identifiable constant of proportionality. For a given virtual seek distance, x_v, with a maximum D_v and a target performance scale, k, we would like the virtual machine to see a service time of $k\,(R/2 + S\,(x_v/D_v)) = k \cdot T_r$, but it will instead see a different time, T_v. To get the target completion time $k \cdot T_r$, we use a *jprobe* to delay the request by an amount, $k \cdot T_r - T_v$.

Selecting an appropriate value of k is difficult. If k is too small $k \cdot T_r - T_v$ is negative, which means we have missed the targeted completion time and this can affect the accuracy of our performance estimates. If k is too large, the total time required to estimate disk performance may

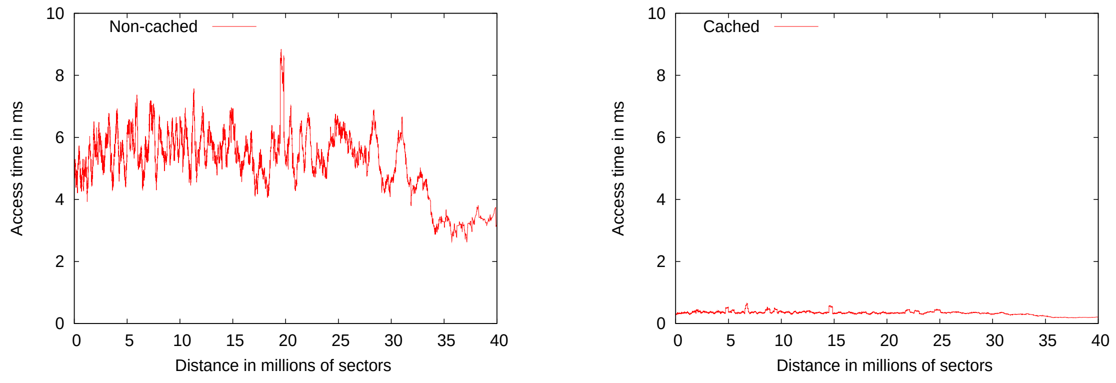


Figure 4.1: Seek Time of Virtual Machine Disk

become excessive. Further, as showing in Figure 4.1, dynamic changes in virtual disk performance, which can be caused by caching, server load, or network congestion, can be dramatic, which can require dynamic changes to k. Thus, we need a self-scaling system.

The VPT constrains the flow of disk requests being served in the kernel based on the target linear model and the current value of k. The VPT uses two probes in the generic SCSI driver: a *jprobe* in the down path to record when requests leave for the virtual disk and an *iprobe* in the up path to intercept and delay completed requests on return from the virtual disk. The *jprobe*, in addition, calculates the target completion time of the request utilizing the seek distance between the current and last request. This information is the used by the *iprobe* to determine how long the request should be delayed after completion but before returning to the requesting process. The *iprobe* places the request on a queue. The queue is checked periodically with a timer. Once the target completion time has past, the request is injected back into the SCSI Generic path.

Since the service time of the virtual disk is subject to change, it is possible that a request will arrive to the *iprobe* queue after its target completion time. This informs the VPT that the scale factor, k, is too small for the observed performance and must be increased. Similarly, if the VPT has a large queue of requests awaiting re-insertion into the SCSI path, we know that the target virtual performance is less (target service times longer) than the observed virtual performance, and so the VPT needs to decrease k. This gives us a feedback mechanism to compensate for variations in the virtual disk performance. While the VPT is dynamically adjusting the scaling factor, it is important to report the value of k in order to scale results properly to the physical disk being modeled.

To this point, the only input required for a VPT is a linear model of service time. Nevertheless, most disks have on-board caches. Although we contend that the effect of such caches is minimal for most workloads, for some workloads the effect can be dramatic. If we wish to capture on-board cache effects in our performance predictions, we must augment the service time model to include a shadow cache. We model the cache with just three parameters: the number of segments, sectors per segment, and pre-fetch size. We assume the cache is fully associative with FIFO replacement.

4.3 VPT Implementation

In order to emulate the target drive, we have to delay disk requests completing on the (virtual) drive attached to our test virtual machine. In order to calculate the expected completion time, we must track the distance the disk head will travel between sectors, the time of service initiation, and if we are modeling cache effects, predict if the requests would be in the cache. The virtual SCSI drive exported by KVM/QEMU has a Symbios Logic 53C8XX controller. We attach a *jprobe* on *sym53c8xx_queue_command()* to capture information about requests headed down the disk path and we attach an *iprobe* on *scsi_done()*. The *jprobe* location was chosen because it is after the request leaves the disk scheduler but before the general scsi interface sends the request down to the disk. Note the virtual machine devices are different from the target devices. Therefore, we want to abstract away any characteristics of the virtual machine device or device drivers that might interfere with the model. The *iprobe* location was chosen because it is the last call before the block layer receives the completed request. Further, the *iprobe*, by design, must replace an existing function and *scsi_done()* is a one line function that calls *blk_complete_request()*. This makes it an ideal target for interception. We also use two queues in our implementation: the “gone-to-disk” and “wait” queues. Their uses will be described below.

Our *jprobe* examines each request to calculate the expected completion time, based on the linear model of the target disk, a specified scale factor (k), the seek distance between the previous and present requests, and the cache model, if such is in use. To implement VPTs in a kernel modules, without modifying the kernel source, we are not able to modify the *struct request* to hold any VPT data. Therefore, we use the “gone-to-disk” queue to place the information about the request, including the expected completion time, the system time when the request should return to the application.

The expected completion time, is calculated using the linear seek model multiplied by the current scaling factor, k. The seek distance in the model is the number of sectors between the current request and the previous request, d_v, divided by the total number of sectors on the virtual disk, D_v. If both queues are currently empty (meaning the virtual disk is not processing requests), the expected completion time is added to the current system time to form the expected return time for this request. Otherwise, at least one queue contains elements, and we add the expected completion time to expected return time of the newest element in the queues to form the expected return time for this request. Once the expected return time is calculated and the entry added to the "gone-to-disk" queue, the *jprobe* returns and normal system execution resumes.

After the disk services the request, the completion interrupt handler will send the request back to the application. In the call path of the interrupt handler, the request will be passed to *scsi_done()* and the *iprobe* will assume control. Our *iprobe* attaches to a function that is inside the interrupt handler, and therefore we want to minimize the computational time this step takes. We remove the request entry from the "gone-to-disk" queue and check the expected return time. If this time is in the future, $\delta = k \cdot T_r - T_v > 0$, we add the disk request to the "wait" queue. If the expected arrival time is in the past, $\delta < 0$, we send the request to *blk_complete_request().* Afterwards, the *iprobe* returns back to the caller. It is important to highlight, that in the usual case, when we place a request on the "wait" queue, we have removed this request from the normal call path.

In order to eventually remove the request for the "wait" queue, we use a high-resolution kernel timer, that periodically checks for requests with expected delivery times that have recently been passed by the system time. We removes these requests from the queue and send them to *blk_complete_request().* At this point, the requests have been inserted back into the normal call path.

The standard Linux kernel timer accuracy is constrained to the kernel *jiffies* counter resolution. (In most systems today this is 1000Hz.) A kernel timer does not provide enough resolution for accurate modeling of a disk unless the scale factor is unreasonably large. The *hrtimer* infrastructure [16] was introduced into the Linux kernel to take advantage of High Precision Event Timers, which first appeared in PC chipsets in 2005 and offers counter resolution in excess of 10 MHz.

Note that with the removal and delayed insertion of the requests, we delay the requests to match a linear model specified for a target physical disk. This effectively slows down the disk performance without any changes to the hardware, drivers, or stock kernel.

4.3.1 Concurrency Issues

Two concurrency issues must be addressed to ensure proper delivery of all requests in order: concurrent access to the queues and handling of multiple, stacked disk interrupts. The former, concurrent access to the queues, is addressed by using interrupt disabling spinlocks in the code segments not inside an interrupt handler.

The more interesting challenge is handling multiple interrupts that may be triggered while the disk interrupt handler is running. On an SMP system with the Intel APIC, the APIC may trigger multiple interrupts on multiple processors. Further, an interrupt handler running on one processor may be interrupted by another interrupt handler, resulting in stacked interrupts. The kernel interrupt processing allows only one handler per interrupt line IRQ to be running on the entire system at any time.

Because we attach an *iprobe*, the interrupt handler path for the disk request is interrupted with an exception (the normal triggering of a *kprobe*), thus temporarily changing the context. This is something that by design does not occur in a normal Linux kernel. Fortunately, the kernel relies on set flags per interrupt to determine if a handler is active. Therefore, when the handler path transitions from the hardware IRQ context to the *kprobe* context, the kernel is still able to identify that the handler for that particular interrupt line is running. Thus, we are ensured all requests are delivered correctly.

4.3.2 Dynamic Scaling

The *iprobe* attached to *scsi_done()* provides the ideal place to add dynamic scaling infrastructures. Whenever we send the request to *blk_complete_request()* and bypass the "wait" queue, the request arrived after the expected return time ($\delta > 0$). This indicates that the scaling factor is too low, and we should increase it. Nevertheless, we should also consider the possibility that the extra delay was due to a temporary change in the underlying host infrastructure. Thus, we keep track of the number of late requests in the last 1000 requests. When more than 1% of the requests arrive late, we increase the scaling factor. This also provides the ability to decrease the scaling factor, if the underlying host infrastructure has allowed for faster service times. If less than 0.1% of the requests miss the expected delivery time then we decrease the scaling factor.

In addition to *ioctl()*, the Linux kernel provides the *sysfs /sys* directory for communication with the kernel through file I/O. Our VPT kernel module registers with the *sysfs* infrastructure to provide user level access to tune the scaling parameters. We provide the ability to read and set the current scaling factor, k. Due to the changes in virtual disk performance that can occur, it is possible that the scaling factor could change during performance measurements. We provide two facilities to deal with this: the ability to disable dynamic scaling and the ability to report the percentage of requests with missed delivery times. Before doing any performance measurements, we run warm-up workloads to allow for the VPT to stabilize the scaling factor. After the scaling factor is stabilized, we can disable the dynamic scaling. By having the VPT report the percentage of missed requests, we can determine the accuracy of the performance measurements.

4.3.3 Cache Model

The final piece to the VPT implementation is the shadow cache model. High performance disks typically utilize an on-bard cache with pre-fetching to increase performance. Cache reads have no seek time, just the time needed to retrieve the data from the cache. The implementation of the shadow cache model resides in the *jprobe*. Recall, that we calculate the expected delivery time on the down path of the request; the code on the up path is solely responsible for delivery of the requests. To the jprobe, we add the simplified cache model, describe in Section 4.4.2.

We keep an array of cache segments, equal in length to the number of cache segments in the target real disk. Each index of the array represents a cache line and stores the range of the sectors on the disk that should reside in the target disk's cache. Before we calculate the expected completion time, we check to see if this request should be in the cache of the target disk model. If the request resides entirely in the disk cache, we set the expected completion time to 250 microseconds, instead of using the linear model calculation. The cache model is not changed. If the read request was not in the cache, we use linear model calculation and update the cache model. We add the request to array of cache segments, overwrite the cache segment with the oldest inserted (not accessed) data. For requests where the data is larger than a single cache line, we followed the cache model described in Section 4.4.2 and assume wrap-around with over-write within the segment.

4.4 Case Study

To test the viability of our Virtual Performance Throttle, we use it to predict the performance of a collection of disk scheduling algorithms, one of which is new. We compare the performance of each algorithm as measured on a virtual machine employing a VPT with its performance as measured on a real machine.

4.4.1 CATS

The cache-aware table scheduler (CATS) is a new disk scheduling algorithm first reported in [10]. It is a modified version the table-building bus driver (TBBD) algorithm suggest by Geist, Steele, and Westall [15]. The essential features are these:

1. It uses an $O(n^2)$ dynamic programming algorithm to compute a statically optimal completion sequence for the current list of n requests and then server the first segment in the list.

2. It uses a *busdriver* delay. That is upon request completion, it will delay for a few milliseconds to see if the requesting process will send another request for a nearby location.

3. It includes the simplified shadow cache, which is modeled by the number of segments, sectors per segment, and pre-fetch size. If any pending request is predicted to be a cache hit, it preempts the optimal order and is scheduled immediately.

Details may be found in [10].

4.4.2 Platform

The real test platform used in our study was a Linux (2.6.30) system with two, Intel Xeon 2.80GHz processors, 1 GB main memory, a Western Digital IDE system drive and two external Seagate Cheetah 15K.4 SCSI drives, each with its own Adaptec 39320A Ultra320 SCSI controller. Tests were restricted to a single Cheetah drive. The disk is a model ST373454 with 4 recording surfaces and a formatted capacity of 73.4 GBytes. It rotates at 15,000 rpm yielding a rotation time of 4 ms. The disk has 50,864 tracks per recording surface and was formatted at 512 bytes per sector.

To approximate the linear service time at the macroscopic level, we disabled the on-board cache, opened */dev/sda* using the *O_DIRECT* mode, which forces a by-pass of the main memory page buffer cache, and read 100,000 randomly selected pages across the entire disk. The time required to read each page and the distance in sectors from the previously read page were captured. We sorted this data in order of increasing distance and plotted distance versus time. The result was a reasonably linear band of noise approximately 4 ms in width. The data was then smoothed using a filter that replaced each point with the average of the (up to) 1001 points centered at the point in question. The filtered data and the least squares approximation to it are plotted in Figure 4.2. The

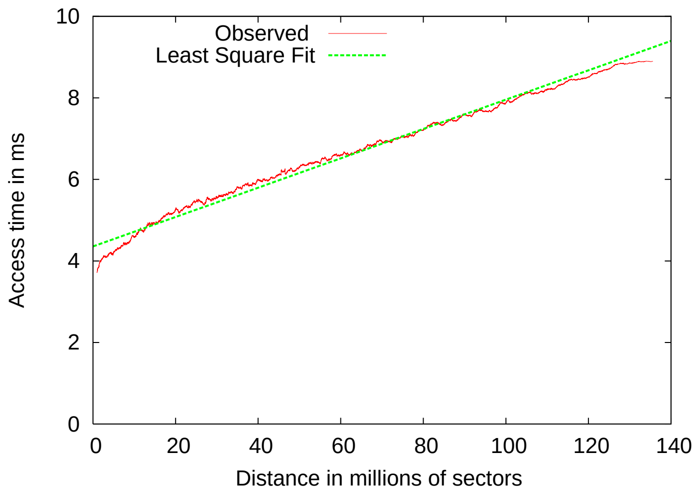


Figure 4.2: Stochastic sector to sector costs

linear model, obtained from the sampling, is $T_r = 4.25 + 5.25(d_r/D_R)$.

The Cheetah manual [26] indicates that 7,077KB is available for caching, which yields 221 512-byte sectors per segment. We assumed that a single span of requested sectors, plus the prefetch, would be cached in each segment. Single request spans larger than 157 sectors (221-64) were rare in our workloads, but for those cases we assumed a wrap-around with over-write within the segment. To determine the access time when a request is in the cache, we examined the service time distribution of a large collection of single-sector requests, independent of any trace, and found a prominent initial spike at 250 microseconds.

The KVM-based virtual machine was hosted on an IBM 8853AC1 dual-Xeon blade. It was configured with a 73GB virtual SCSI disk, for which the available emulator was an LSI Logic / Symbios Logic 53c895. The virtual disk image was stored on a NetApp FAS960c and accessed via NFS. We used the sampling process used on the target physical disk on the VPT-enabled virtual disk to obtain a sector to sector cost. Figure 4.3 shows the comparison of the physical and virtual sampling. The two samplings are close to one another, and the virtual sampling is nearly identical to the least square fit of the physical sampling. The linear model for the VPT disk sampled, with k set to 10, is $T_v \approx 42.5 + 52.5(d_v/D_v) = 10 \cdot T_r$.

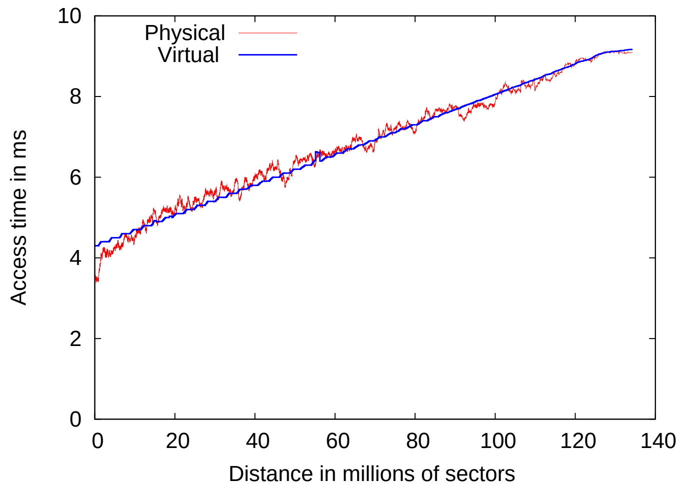


Figure 4.3: Comparison of Sector to Sector Costs of Physical and Virtual Disks

4.4.3 Workload

We compared the performance of our cache-aware, table-scheduling (CATS) algorithm with Linux schedulers *cfq* and *deadline* on both real and virtual platforms. For the virtual machines, once the scale factor was established, by self-scaling on a preliminary workload, it was fixed for each of the CATS, *cfq*, and *deadline* schedulers. The delay timing parameters for the schedulers are tunable through the *sysfs* interface with the exception of one parameter in *cfq*. We could avoid recompiling the kernel for CATS and *deadline*. We choose to recompile the kernel for *cfq*, since ignoring any tuning parameter would reduce the accuracy of the measurements.

We used two workloads in this study. Both were similar, at the process level, to that used by Geist, Steele, and Westall [15], which was based on the approach used by Barford and Crovella [1] in building their Scalable URL Reference Generator (SURGE) tool. Each of 50 processes executed an ON/OFF infinite request loop, shown in pseudo-code in Figure 4.1.

```
**forever** {
    generate a file count, $n$, from $Pareto(\alpha_1, k_1)$;
    **repeat**($n$ times) {
        select filename using Zipf($N$);
        **while**(file not read) {
            read page from file ;
            generate $t$ from $Pareto(\alpha_2, k_2)$;
            sleep $t$ milliseconds;
        }
    }
}
```

Listing 4.1: ON/OFF execution by each of 50 concurrent processes

The Pareto(α,k) distribution is a heavy-tailed distribution often encountered in network modeling. The distribution function is

$$F_X(x) = \begin{cases} 1 - (k/x)^{\alpha} & x \geq k \\ 0 & elsewhere \end{cases} \tag{4.1}$$

The discrete Zipf distribution is given by

$$p(i) = k/(i+1), \qquad i = 0, 1, ..., N \tag{4.2}$$

where k is a normalizing factor, specifically, the reciprocal of the $N + 1^{st}$ harmonic number. A continuous approximation,

$$F_X(x) = \frac{log(x+1)}{log(N+2)} \qquad 0 \leq x \leq N+1 \tag{4.3}$$

suffices for our study.

File count parameters were taken directly from the Barford and Crovella study, $(\alpha_1, k_1) =$ (2.43,1.00). The shape parameter of the sleep interval, $\alpha_2 = 1.50$, was also taken from this study, but we used a different scale parameter, $k_2 = 2.0$, because our sleep interval was milliseconds per block rather than seconds per file.

Both the virtual SCSI drive and the Cheetah drive were loaded with 1 million files in a two-level directory hierarchy where file sizes were randomly selected from a mixture distribution also suggested by Barford and Crovella. This mixture distribution is lognormal(9.357,1.318) below

133 KB and Pareto(1.1,133K) above, where the lognormal(μ,σ) distribution function is given by:

$$F_Y(y) = \int_0^y e^{-\frac{(log_e t-\mu)^2}{2\sigma^2}} / (t\sigma\sqrt{2\pi})dt \quad y > 0 \tag{4.4}$$

To induce reasonable fragmentation, we erased a half million files, selected at random, and then added back a half million files with different, randomly selected sizes. Due to the relatively small capacity of these drives, we chose to truncate at 100 MB any files that would have exceeded that size.

For each algorithm, for each test run, we captured the arrival times, service initiation times, and service completion times of 50,000 requests. We captured these time stamps by directly instrumenting the kernel outside of the schedulers, and, during each test run, we stored the time stamps to a static kernel array. The time stamp data was extracted from the kernel array after the test run by using a custom system call.

The two workloads were identical at the process level but decidedly different at the drive level. For the first, each file was opened with mode flag *O_DIRECT*. This forced the associated I/O to by-pass the main memory page buffer cache. The second workload differed from the first only in that the *O_DIRECT* flag was not used. The page buffer cache, a standard feature of UNIX-derivative operating systems, is dynamic and can grow to become quite large. For tests described here, 65MB was often observed.

Finally, although the Cheetah drive supports tagged command queuing (TCQ), we disabled it for all tests. We found that, for all schedulers, allowing re-scheduling by the drive hardware <u>decreased</u> performance. We would have thought this to be an anomaly, but we have observed the same result for other SCSI drives on other Linux systems.

4.4.4 Results

The results for the first workload, using *O_DIRECT*, are shown in Table 4.1. We see that the virtual system uniformly predicted higher mean service, higher mean response, and lower throughput than was found from measurements of the real system. Nevertheless, on all three measures, the predicted performance rank of the three algorithms was correct: CATS performs better than *deadline*, which performs better than the Linux default, *cfq*. Thus algorithm selection could be made solely on the basis of the virtual system predictions.

	real			virtual (k=8)		
algorithm	*cats*	*deadline*	*cfq*	*cats*	*deadline*	*cfq*
mean service (ms)	1.96	2.71	1.39	2.58	3.24	2.36
variance service	8.51	9.76	5.85	9.03	8.23	7.78
mean response (ms)	37.35	59.87	124.70	53.79	78.27	117.13
variance response	6961.50	561.15	839270.49	16641.07	633.28	28651.71
throughput (sectors/ms)	8.19	6.08	2.19	6.15	5.06	3.38

Table 4.1: Performance on *O_DIRECT* workload.

Also, although response time moments show considerable differences between real and virtual, overall response time distributions for real and virtual systems are reasonably close to one another as seen in Figure 4.4.

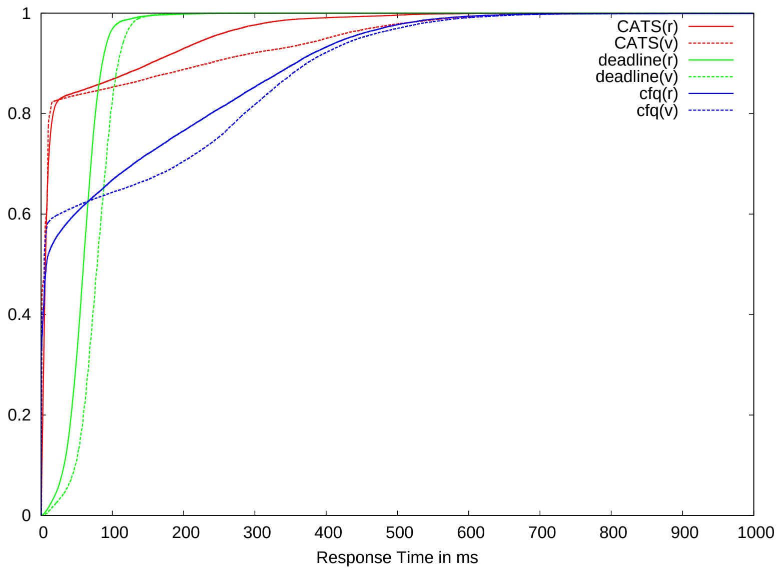


Figure 4.4: Distribution of Response Times of *O_DIRECT* Workload

We also gauged the effectiveness of the shadow cache in predicting real cache hits by placing record markers within the captured time stamp trace of the CATS scheduler on all records that were predicted by the shadow cache to be hits. We then processed the CATS trace and marked any record with service time below 250 microseconds as an actual hit. We found that the shadow cache correctly predicted 97% of the 31,949 actual hits observed.

When I/O is staged through the main memory page buffer cache (the second workload), the results are decidedly different, as shown in Table 4.2. Again the virtual system uniformly overestimated mean service time and mean response time and underestimated throughput, but again

	real			virtual (k=8)		
algorithm	*cats*	*deadline*	*cfq*	*cats*	*deadline*	*cfq*
mean service (ms)	6.53	7.41	7.80	7.15	7.60	8.57
variance service	11.13	8.80	17.62	6.22	6.05	10.30
mean response (ms)	114.91	121.87	179.17	189.45	198.33	258.75
variance response	8080.16	3296.87	35349.39	19292.52	6839.66	65796.33
throughput (sectors/ms)	12.00	12.04	9.08	11.44	11.68	8.82

Table 4.2: Performance on non-*O_DIRECT* workload.

it correctly predicted the performance rank of all three algorithms on all three measures. Again, overall response time distributions for real and virtual systems are reasonably close to one another, though not as close as the distribution of the *O_DIRECT* workload, as seen in Figure 4.5.

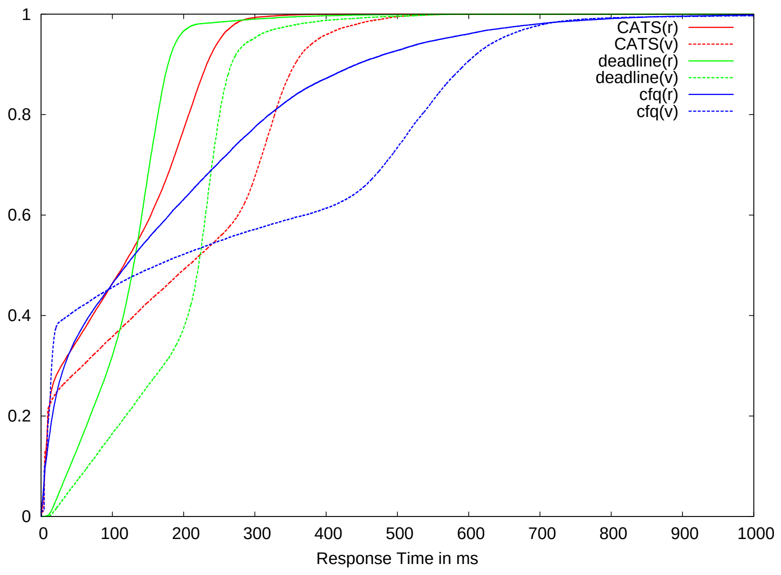


Figure 4.5: Distribution of Response Times of *non-O_DIRECT* Workload

The error in estimation obtained from the virtual system can be attributed to two factors. First, recall that for requests that are serviced from the cache model, we adjust the expected completion time to 250 microseconds, as this is the maximum access time we observed for cache hits from the sampling. However, the mean access time all of the cache hit requests in the sampling is half of the maximum access time, 125 microseconds, with a variance of 0.002. The error in the results could be reduced by lowering cache hit access time. Reducing the expected completion time for cache hits, unfortunately, requires the scale factor, k to be increased to bring the expected completion times up to the service times of the virtual disk.

The VPT also incurs overhead in the virtual system and the expected completion time is only measured up to when the request arrives at *scsi_done()*, not when it is returned to the application. On an idle system, this extra work incurs negligible extra service time. On a heavily used system (such as our virtual machine running the workload), this overhead can impact service times as much as $2ms$ (unscaled). This overhead also impacts cache hit requests, since their access times are much smaller than cache miss requests. As the scaling factor is increased, the impact of the overhead diminishes and the error is reduced. Accounting for the overhead in the VPT module is possible, but it also requires the scale factor to be increased. To minimize the error (improve absolute accuracy in performance predictions) requires a larger scale factor and consequentially, slower performance runs.

Chapter 5

Virtual Operating Systems Lab

5.1 The Virtual Lab

With the virtual architecture in place, we are now able to deploy a KVM-based, virtual PC lab in which the virtual PCs export the entire systems-level interface required for CPSC 822. As a result, the third offering of Virtual CPSC 822 is underway as of this writing. The virtual PC lab is hosted on only 6 IBM 8853AC1 dual-Xeon blades in a 42-blade eServer BladeCenter.

Of course, KVM alone does not allow for emulation. It must be used in conjunction with a user-space program, QEMU [3]. Other tools such as libvirt [21] and the Virtual Machine Manager [30] allow for easy creation, deployment and management of virtual machines, but we chose not to use these. We have found that custom scripts give us the control that we need while still providing the students with a simple interface through which to access their virtual machines.

5.1.1 Creating and Maintaining VMs

Creating a KVM virtual machine first starts with creating a virtual hard disk. Listing 5.1 shows the QEMU command for creating a virtual, 30GB hard disk. The *-f* option specifies the file format. For our virtual lab, we chose to use the qcow2 format. One benefit of this format is the minimal footprint it leaves on the physical drive. It will only use space on the physical drive for those bytes actually allocated in the virtual disk.

Next, the empty virtual disk must be loaded with a base operating system, in this case

CentOS 5, 64-bit. The command is shown in Listing 5.2.

```
qemu-img create -f qcow2 822master.img 30G
```

Listing 5.1: Creating a Disk Image

```
qemu-system-x86_64 -drive file=822master.img,if=scsi,bus=0,unit=0
    -cdrom CentOS_64-5.iso
    -boot d -m 4096
```

Listing 5.2: Installing CentOS 5 on a Virtual SCSI Disk

Note that the drive type is specified as SCSI, not the default IDE, because CPSC 822 includes a full traversal and detailed examination of the SCSI read path. A minor drawback to this particular installation command is that the automatic re-boot afterward fails, but the system is easily re-booted thereafter as shown in Listing 5.3.

```
qemu-system-x86_64 -drive file=822master.img,if=scsi,bus=0,unit=0,boot=on
```

Listing 5.3: Launching a VM with a SCSI Hard Disk

The BladeCenter, on which the virtual machines are located, is isolated from the main campus network via a gateway machine accessible through SSH. Each two-person team of students is given a lab account that is unique to the BladeCenter. Accounts are synchronized among blades with NIS (Network Information Service). Home directories are shared across all blades by NFS (Network File System) from a NAS (network-attached storage) device.

A virtual machine image is placed in each team's home directory. Rather than a full image, we use a clone image. The qcow2 image format allows for multiple clone images to use a common base master image. When a virtual machine using a clone image reads an original file, it reads from the master image. When a virtual machine writes a file, the copy-on-write is executed and the new contents are written to clone image. After the copy-on-write, only the clone image is accessed when either a read or write is executed on the modified file. The command to create a clone image from a base master image is shown in Listing 5.4.

For 26 students in pairs we still require a minimum of 13 virtual machines, each with a full CentOS 5 installed. The full installation occupies approximately 6 GB, much of which remains read-only throughout the semester, and thus the use of the clones saves approximately 72 GB per set of virtual machines.

```
qemu−img create −b 822master.img −f qcow2 822lab.img
```

Listing 5.4: Creating Clone Images

Nevertheless, in our lab environment, saving space comes at the cost of performance. When all 13 virtual machines are actively accessing the virtual disks, the master/clone images become a point of contention and hence performance degradation. We measured the time required to build a full Linux 2.6.30 kernel on a clean source tree in two environments: 6 virtual machines with stand-alone hard drives and 6 virtual machines sharing one master hard drive. The build time on the cloned hard drives took approximately 45 minutes, but the build on the stand alone hard drives took approximately 35 minutes, a 22% reduction. To balance performance and storage, we decided to use only 2 clones per master image. This configuration still saves 36 GB and yields acceptable performance.

5.1.2 Using the Virtual Machines

All students must first access the gateway before proceeding to a KVM-enabled blade. Teams access a KVM-enabled blade by the *go_blue* script (example in Listing 5.5) that is placed in their home directories. Each script will SSH to a predetermined blade to provide some (static) load-balancing across available blades. Each team has a password-less SSH key, and so they do not need to re-enter their passwords when executing *go_blue.* Teams could manually execute an SSH command to access other blades in the BladeCenter, and we do not explicitly prevent this, but it could be prevented by using the netgroups feature of NIS.

```
#!/bin/bash
ssh −Y blue30
```

Listing 5.5: go_blue Script

To simplify launching virtual machines and to protect the students from themselves, we provided the script *start_lab_vm* shown in Listing 5.6. The script checks for a lock file to prevent multiple virtual machines from launching with the same primary virtual drive. Having multiple virtual machines writing to a single virtual drive (a file) will result in data corruption and a system crash.

```
#!/bin/bash

if test -f $HOME/kvm/.hd_lock; then
   echo "Oops...HD already in use!"
   exit 1
fi

touch $HOME/kvm/.hd_lock
/usr/local/kvm/bin/qemu-system-x86_64 -m 512 \\
   -drive file=$HOME/kvm/822lab.img,if=scsi,bus=0,unit=0,boot=on \\
   -drive file=/home3/822lt00/kvm/usr_local.img,if=scsi,bus=0,unit=1

rm $HOME/kvm/.hd_lock
echo "VM Powered Off."
```

Listing 5.6: start_lab_vm Script

All virtual lab machines mount a second drive (*usr_local.img*). This drive is shared among all virtual lab machines. Throughout the course, the virtual lab machines need to be upgraded to enable new lab exercises and major course projects. Launching each individual virtual machine and making the changes would be a slow, tedious, and error-prone process. With a common shared drive, changes only need to be made once. Unlike the primary drive, this image file has read-only permissions. There will never be any writes to the virtual drive, and therefore it is safe for all lab machines to mount this drive concurrently. However, the virtual lab machines cannot be running when *usr_local.img* is updated. To prevent teams from launching their virtual machines during these maintenance periods, we use additional lock files in the team directories.

With this configuration, the virtual monitor is displayed on the student's local machine after being forwarded through an SSH X11 Tunnel. An alternative to this approach is to launch the virtual machine as a daemon and direct the monitor through a VNC server. While this is the default method used by Virtual Machine Manager, it is cumbersome to use in the virtual CPSC 822 lab. Since there is a gateway separating our BladeCenter from the network, the VNC ports are not easily accessible. As with any operating system development, the virtual operating system will crash often, and running the virtual machine as a daemon complicates the process of restarting and power cycling.

While the SSH X11 Tunnel option is easy for students to use, it is CPU intensive, and this can create a significant performance bottleneck on our 3.2 GHz, Pentium 4 (with Hyper-Threading) gateway machine. Each tunnel requires packets to be copied back and forth between kernel and user space and requires packets to be decrypted and encrypted. An alternative to SSH X11 Tunnels is to use X11 Forwarding. X11 Forwarding does not use encryption, and the packets traveling through the gateway machine are handled solely in kernel space. An example command sequence to use X11 Forwarding is given in Listing 5.7.

```
$ xhost +
$ ssh <gateway machine>
$ ./go_blue
$ export DISPLAY=<ip of local machine>:0
$ ./start_lab_vm
```

Listing 5.7: Using X11 Forwarding to access VM monitor.

Six virtual machines using X11 Tunneling will maximize CPU usage (64.5% user, 31.4% system, 4.1% idle) on the gateway machine. Six virtual machines using X11 Forwarding will use only 32% of the CPU (0.1% user, 29.7% system, 70.2% idle) on the gateway machine.

5.2 Timing

A critical component of the course is accurately measuring subsystem response time and throughput, such as measuring the performance of disk I/O under various scheduling algorithms. With the 2.6.18 kernel provided in CentOS 5, the default clock source is based on the *jiffies* counter and a programmable interrupt timer (PIT). Under heavy load, this clock source is inaccurate. When performing a full kernel build (*make bzImage, make modules, make modules_install, make install*) on a clean kernel source tree, one can easily find the system time in the virtual machine to be skewed by over 60 seconds forward or backward from the host system time. Clock skews into the future confuse the make system, and this can yield incomplete builds. Thus, in order for a kernel build to work properly, the build directory must be cleaned before the next build attempt. This can be an expensive operation, and it negates the benefits of incremental compilation and linking. With both forward and backward clock skewing, timing measurements will be inaccurate, resulting in all time-related performance measurements being unreliable.

The cause of the skew is that when the system is under heavy load, some PIT interrupts will be missed, and the hypervisor will attempt to re-inject the interrupts. The re-injections can cause the clock to skew either forward or backward. QEMU supports the option *-no-kvm-pit-reinjection*, which will disable reinjection of the interrupts into the virtual machine. Launching QEMU with this option will guarantee that the clock will not skew forward, and this prevents incomplete builds resulting from a confused make system, but it still allows the clock to skew backward.

A paravirtualized clock was introduced into the mainline 2.6.26 Linux kernel for use with KVM-based virtual machines to solve clock skewing in both directions. When this option is compiled into the kernel (*CONFIG_PARAVIRT_CLOCK=y*) and is running inside a KVM virtual machine, the OS will receive the TSC information from the hypervisor when updating the system time. We find that with the paravirtualized clock, the system time in the virtual machine is always within milliseconds of the system time on the host machine. This may not be accurate enough for networking performance measurements. Nevertheless, for disk scheduling performance measurements and using the make system, it suffices.

5.3 Performance

A standard issue in virtualization of any system is the resulting performance penalty. While KVM utilizes the native virtualization features of Intel VT and AMD-V, it still relies on QEMU for providing the interface to the virtual system, and this may introduce a significant performance penalty. In estimating the penalty, it is important to realize that a host system may migrate virtual machines among different cores on the system. The Linux utility *taskset* allows a user to set (or change) a process's CPU affinity or pin that process to one or multiple virtual CPUs. An example invocation of *taskset* is shown in Listing 5.8. The command arguments are a bitfield to specify target CPUs and the process to launch, along with its parameters.

```
taskset 0x11 ./light8 ii9.ex.perked > out.lit
```

Listing 5.8: Launching a program pinned to the virtual CPUs 0 and 5

Since a virtual machine is actually a process on the host, it can pinned to a group of CPUs in the same manner as any other process. The dual-Xeon blades on which we are deploying the virtual machines are based on the Core 2 Quad architecture. In this processor architecture, a pair of cores shares an L2 cache, and there is no L3 cache. In our testing, we chose to pin each virtual machine to a pair of cores, as this prevents a migration that involves moving across physical processors or moving across L2 caches. Listing 5.9 shows a virtual machine configuration with 1 CPU pinned to the first pair of cores on the blade.

```
taskset 0x3 qemu−system−x86_64 −drive file=822kvm.img,if=scsi,bus=0,unit=0,boot=on
```

Listing 5.9: Launching a VM pinned to 2 cores with shared L2 cache

For tests on the real machine, we pinned the workload process to the first pair of cores on the blade. For tests on the virtual machine, we pinned the VM process as in Listing 5.9. Since our principal interest was the computational penalty from KVM, rather than the I/O penalty from the combined effects of KVM and QEMU, our test workload was a compute-intensive application taken from three-dimensional computational fluid dynamics (CFD) [14]. Table 5.1 shows the results of running the CFD code on the real and virtual machines. We see a performance penalty of 2.71% (without CPU affinity) and 0.05% (with CPU affinity) for running the CFD code inside a virtual machine.

	On the Metal	In the VM
Unpinned	1634.56	1678.88
Pinned	1651.86	1652.68

Table 5.1: Runtime in seconds of CFD code for various configurations

5.4 Virtual Device Generation Utility

A time consuming and tedious part of the maintenance of the Virtual CPSC 822 lab is the creation of new graphic cards and documentation every semester. The instructor who uses problems from a previous semester is guaranteed to receive a solution from a previous semester! Design a new (virtual) graphics card is a tedious process. Default register values must be recalculated and register values, names and locations must be changed in several locations.

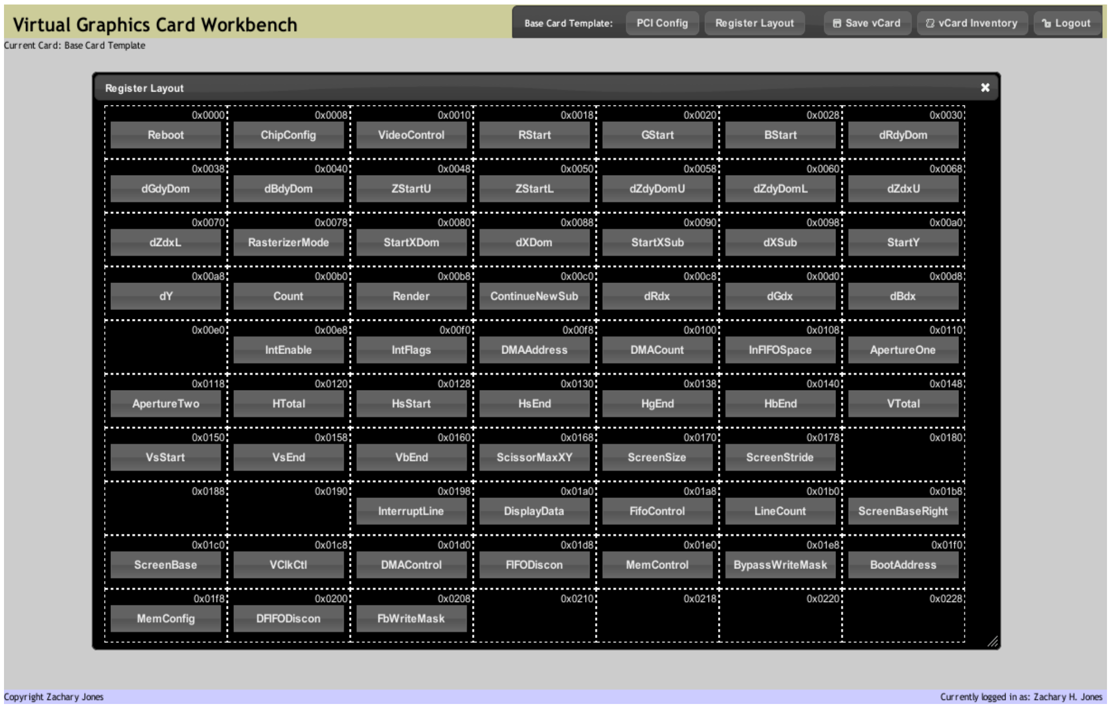
To reduce these difficulties, we provide a browser-based graphical user interface for configuring a Virtual PCI graphics card. Figures 5.1 and 5.2 show two dialogs of the user interface. The GUI provides the user (course instructor) the capabilities to:

- Changed general device properties. (Card Name, Author, PCI IDs, IRQ Number)
- Change register locations (layout in memory) with a drag 'n' drop interface.
- Change register bitfield layouts with a drag 'n' drop interface.

- Edit register names.
- Create, save, and load card specifications.
- Generate documentation and source code for the loaded card specification.

There are two motivating factors that led us to build the utility as a web application: deployment and rapid prototyping. Since the application is web based, it allows us to work on card specifications on any Internet-connected machine without having to install the utility and its dependencies. Further, card specifications are stored on the remote server, removing the need to keep track of where the card specification files are located. It integrates version control at no cost. The user interface was created with HTML, CSS, Javascript and the JQuery and JQuery UI javascript libraries. The libraries allow for simplified, yet flexible, browser side scripting and UI creation. For instance, the libraries provide utilities to easily create the interfaces, interactively move and swap registers, and interactively rearrange the bitfields of a particular register. Therefore, it required minimal coding to achieve the results.

The back-end generation of source code and documentation relies on two components: the card specification and a template of virtual card and documentation. After designing several virtual graphics cards, we were able to identify common areas of the source code and documentation that remained the same, and also areas that always changed. From this knowledge were able to create a template version of the virtual card with the known variables marked for replacement. With the template in hand, we simply use the card specification as a way to replace the marked variables with the specified values. Each offering of Virtual CPSC 822 has used a different virtual graphics card.


Virtual Graphics Card Workbench

Base Card Template: PCI Config | Register Layout | Save vCard | vCard Inventory | Logout

Current Card: Base Card Template

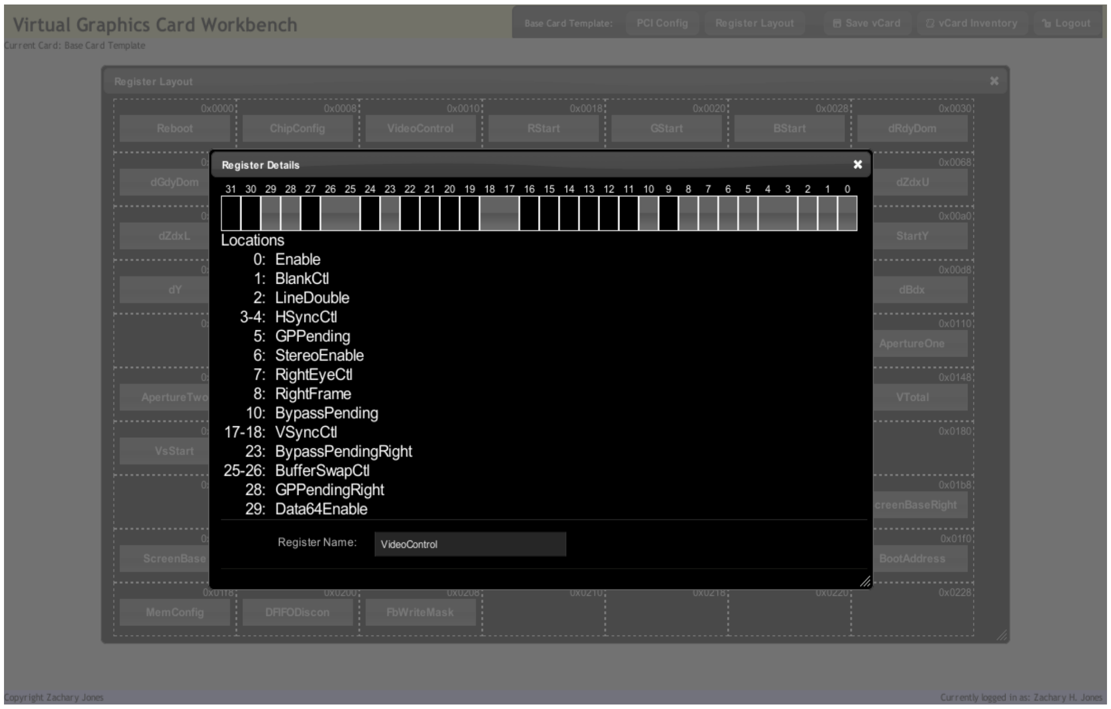
Register Layout

0x0000 Reboot	0x0008 ChipConfig	0x0010 VideoControl	0x0018 RStart	0x0020 GStart	0x0028 BStart	0x0030 dRdyDom
0x0038 dGdyDom	0x0040 dBdyDom	0x0048 ZStartU	0x0050 ZStartL	0x0058 dZdyDomU	0x0060 dZdyDomL	0x0068 dZdxU
0x0070 dZdxL	0x0078 RasterizerMode	0x0080 StartXDom	0x0088 dXDom	0x0090 StartXSub	0x0098 dXSub	0x00a0 StartY
0x00a8 dY	0x00b0 Count	0x00b8 Render	0x00c0 ContinueNewSub	0x00c8 dRdx	0x00d0 dGdx	0x00d8 dBdx
0x00e0	0x00e8 IntEnable	0x00f0 IntFlags	0x00f8 DMAAddress	0x0100 DMACount	0x0108 InFIFOSpace	0x0110 ApertureOne
0x0118 ApertureTwo	0x0120 HTotal	0x0128 HsStart	0x0130 HsEnd	0x0138 HgEnd	0x0140 HbEnd	0x0148 VTotal
0x0150 VsStart	0x0158 VsEnd	0x0160 VbEnd	0x0168 ScissorMaxXY	0x0170 ScreenSize	0x0178 ScreenStride	0x0180
0x0188	0x0190	0x0198 InterruptLine	0x01a0 DisplayData	0x01a8 FifoControl	0x01b0 LineCount	0x01b8 ScreenBaseRight
0x01c0 ScreenBase	0x01c8 VClkCtl	0x01d0 DMAControl	0x01d8 FIFODiscon	0x01e0 MemControl	0x01e8 BypassWriteMask	0x01f0 BootAddress
0x01f8 MemConfig	0x0200 DFIFODiscon	0x0208 FbWriteMask	0x0210	0x0218	0x0220	0x0228

Copyright Zachary Jones

Currently logged in as: Zachary H. Jones

Figure 5.1: Layout of device registers in memory.


Register Details

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Locations

0:	Enable
1:	BlankCtl
2:	LineDouble
3-4:	HSyncCtl
5:	GPPending
6:	StereoEnable
7:	RightEyeCtl
8:	RightFrame
10:	BypassPending
17-18:	VSyncCtl
23:	BypassPendingRight
25-26:	BufferSwapCtl
28:	GPPendingRight
29:	Data64Enable

Register Name: VideoControl

Figure 5.2: Bitfield layout of a register.

Chapter 6

Conclusions

We have provided the design and implementation of a virtual architecture that allows system-level, functional emulation of high-performance graphics cards for the purposes of driver design and development. We have tested this architecture on a class of graduate students who were given the task of writing a driver and most did not even realize that the card was virtual. Rendering performance through the virtual card was not strong, but acceptable for the purposes of instruction and development.

We conclude that we have met design goals for the virtual card project, with one exception: the goal of implementing this architecture with zero changes to the standard Linux kernel was missed, by a single word. In the 2.6.26 and newer kernels, *do_page_fault()* is declared with the *__kprobes* qualifier, which precludes the use of *kprobes* to intercept this function. The concern is an infinite recursion, should the *kprobe* handler page fault. Our handler writes only to a page table, which will not fault, and so we simply remove the declaration and intercept as planned. Nevertheless, this (removal) is a one-word kernel modification. It is worth noting, that a carefully crafted kernel module could replicate the *kprobe* and *jprobe* functions responsible for installing new *kprobes* into the system minus the check for the *__kprobes* qualifier. While this would achieve our goal of a zero modifications to the kernel, this code is difficult to maintain and not worth pursuing for a virtual lab environment.

We believe that this system can have significant impact on the process of driver design. Driver design, development and testing could proceed in parallel with new hardware development, thus reducing time to market for new PCI products.

We have also developed Virtual Performance Throttles (VPTs) to perform performance measurements on virtual hardware with relative performance to physical hardware. Our VPTs provide a method for predicting the performance of algorithms on real machines using a model based only their measured performance on virtual machines. By using a dynamically loaded kernel *iprobe* and *jprobe*, our VPT can adjust low-level virtual device timings to match that of a simple model derived from the real device. As a case study, we predicted the performance of three disk scheduling algorithms, one of which is new.

Although the virtual system was seen to uniformly underestimate performance, it correctly predicted the relative performance of the three algorithms as measured on a real system under two workloads. It it fair to charge that we are simply using a virtual machine running Linux as an elaborate simulator. This is a fair claim, but this simulator provides all the nuances of a real operating system and yet requires only minimal coding effort. A low-level model of device performance and the drop-in *iprobe* and *jprobe* are all that is required.

We believe that VPTs have potentially broad application to measuring of real systems using only virtual systems and simple models. VPTs can be used to obtain relative performance measurements of devices while designing algorithms. Only when testing required absolute performance measurements, would the real systems be used. Thus, saving resources and allowing better resource allocation of the real systems.

Further, we believe that both tools also have great application in academic instruction. The Virtual PCI Framework allows students to design device drivers for high performance graphics card and the Virtual Performance Throttles allow students to design and test new disk schedulers on a high-end SCSI drives without the need for dedicated resources or the physical devices. Therefore, we are underway in using a virtual lab for CPSC 822. The virtual environment we have built allows for larger enrollment, better resource utilization, less administration, and discourages students from seeking alternative methods to complete assignments.

The virtual lab is KVM-based, hosted on only 6 dual-Xeon blades, and provides the experimental platforms for many graduate students. Some relatively simple, custom shell scripts manage resource allocation and provide good performance, even under relatively heavy loads. Most of the difficulties that arise in having students work at the systems level on virtual machines have been overcome.

Current development is proceeding with the goal to extend the reach of virtualization. We are exploring virtualizing CUDA-capable NVIDIA graphics cards. The goal is to present to a user of a physical or virtual machine a single CUDA/OpenCL capable graphics card that, hidden to the user, is either a portion of a physical card or a combination of multiple graphics cards. This will allow us to extend the benefit of the virtual lab to other courses and potentially as a way to increase the availability of CUDA devices by virtually partitioning devices, for research.

References

[1] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. *SIGMETRICS Perform. Eval. Rev.*, 26(1):151–160, 1998.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. 19th ACM Symp. on Operating System Principles*, pages 164–177, Bolton Landing, New York, October 2003.

[3] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[4] R.E. Bellman. *Dynamic Programming.* Dover Publications, Incorporated, 2003.

[5] E. G. Coffman and E. N. Gilbert. Polling and greedy servers on a line. *Queueing Systems*, 2:115–145, 1987. 10.1007/BF01158396.

[6] E G Coffman and M Hofri. On the expected performance of scanning disks. *SIAM Journal on Computing*, (11), 1982.

[7] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers.* O'Reilly Media, Inc., Sebastopol, CA, 3rd edition, February 2005.

[8] R. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research & Development*, 25(5):483–490, September 1981.

[9] R. Geist, Z. H. Jones, and J. Westall. Virtualizing high-performance graphics cards for driver design and development. In *Proc.* 19^{th} *Annual Int. Conf. of the IBM Centers for Advanced Studies (CASCON 2009)*, Toronto, Ontario, Canada, November 2009.

[10] R. Geist, Z. H. Jones, and J. Westall. Predicting performance with virtual machines. In *Performance Evaluation of Computer and Communication Systems: Milestones and Future Challenges (PERFORM 2010)*, Vienna, Austria, October 2010.

[11] R. Geist and R. Ross. Disk scheduling revisited: Can $o(n^2)$ algorithms compete? In *ACM-SE 35: Proceedings of the 35th annual Southeast regional conference*, pages 51–56, Murfreesboro, TN, USA, 1997. ACM.

[12] Robert Geist and Stephen Daniel. A continuum of disk scheduling algorithms. *ACM Transactions on Computer Systems*, (5), 1987.

[13] Robert Geist, Robert Reynolds, and Eve Pittard. Disk scheduling in system v. In *SIGMETRICS '87: Proceedings of the 1987 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 59–68, New York, NY, USA, 1987. ACM.

[14] Robert Geist, Jay Steele, and James Westall. Convective clouds. In *Natural Phenomena 2007 (Proc. of the Eurographics Workshop on Natural Phenomena)*, pages 23–30, 83, back cover, Prague, Czech Repubilc, 2007.

[15] Robert Geist, Jay E. Steele, and James Westall. Enhancing web server performance through the use of a drop-in, statically optimal disk scheduler. In *Int. CMG Conference*, pages 697–706. Computer Measurement Group, 2005.

[16] Thomas Gleixner and Douglas Niehaus. Hrtimers and Beyond: Transforming the Linux Time Subsystems. In *Proceedings of the Linux Symposium, Ottawa, Canada*, volume 1, pages 333–346, 2006.

[17] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. Wiregl: A scalable graphics system for clusters. In *Proc. ACM SIGGRAPH 2001*, pages 129–140, August 2001.

[18] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. *kvm*: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, pages 225–230, Ottawa, Ontario, Canada, July 2007.

[19] H. Kobayashi. *Modeling and Analysis. An Introdution to System Performance Evaluation Methodology.* Addison-Wesley, 1978.

[20] H. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara. VMM-independent graphics acceleration. In *Proc. ACM Conf. on Virtual Execution Environments*, pages 33–43, San Diego, California, June 1315 2007.

[21] libvirt. libvirt: The Virtualization API. `http://www.libvirt.org/`, 2009.

[22] A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A Keshavamurthy, and M. Hiramatsu. Probing the guts of Kprobes. In *Proceedings of the Linux Symposium Volume Two*, pages 101–116, Ottawa, Ontario, Canada, July 2006.

[23] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[24] Stephen Pratt and Dominique Heger. Workload dependent performance evaluation of the linux 2.6 i/o schedulers. In *In Proceedings of the Linux Symposium, volume 2. Ottawa Linux Symposium*, 2004.

[25] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel®Pentium's^{TM}ability to support a secure virtual machine monitor. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*, pages 10–10, Berkeley, CA, USA, 2000. USENIX Association.

[26] Seagate Technology LLC, Scotts Valley, CA, USA. *Product Manual Cheetah 15K.4 SCSI*, rev. d edition, 2005.

[27] S. Stegmaier, M. Magalln, and T. Ertl. A generic solution for hardware-accelerated remote visualization. In *Proc. of the Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization*, pages 87–94, Barcelona, Spain, 2002.

[28] Texas Instruments. TVP4020 Permedia 2. http://ftp.netbsd.org/pub/NetBSD/misc/cegger/hw_manuals/3dlabs/, Aug. 1997.

[29] The virtual GL project. http://www.virtualgl.org/.

[30] VMM. Virtual Machine Manager. `http://virt-manager.et.redhat.com/`, 2009.

[31] VMWare, Inc. Understanding full virtualization, paravirtualization, and hardware assist. http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf, 2007.

[32] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. Scheduling algorithms for modern disk drives. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 241–251, New York, NY, USA, 1994. ACM.