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

---

A Dissertation Proposal by

Zachary Harrison Jones

26 February 2010

Submitted to the graduate faculty of the
School of Computing
in partial fulfillment of the requirements
for the Dissertation Proposal and
subsequent Ph.D. in Computer Science

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

Outline

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. The applications of the system include both instruction in device driver design and allowing device driver design to proceed in parallel with new hardware development. While this allows for arbitrary design, it is not able to model performance characteristics. We propose the a new kernel system that allows for arbitrarily changing the performance of a device. These virtual performance throttles (VPTs) use an model of the performance of a physical device with the framework provided by the virtual device architecture.

1 Introduction

In this dissertation, we will 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.

Nevertheless, the total impact of the course, in terms of the number of students served, has been limited by available resources. Students who have previously completed the CPSC 822 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. Many course components, e.g., new kernel builds and scheduler experiments, could be directly handled by any of the virtualization tools. Nevertheless, key course projects have remained out of reach of such tools. One such 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 thus 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. We also note that the virtual architecture allows concurrent development of new hardware designs and supporting system software.

Another 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 continues to be 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 that performance of the system virtual drive. We propose the introduction of Virtual Performance Throttles (VPTs) to effect this design. As we will see in Section 5.1, with careful study, 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.

1.1 Organization of this Document

In the next section 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 *kprobes*, the principal tools used in our implementation. In Section 3.1 we provide an overview of the virtual architecture of our system, which 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 Section 4, we provide an overview of the lab environment deployed for the current virtual Computer Science 822 class and discuss several challenges encountered and their solutions. In Section 5 we discuss preliminary and planned work on VPTs and virtual device generation tools. Conclusions follow in Section 6.

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 [4]. 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 [11]. 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 [12]. 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 [17]. In another approach, Xen [1] 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 has been 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, and a hardware state switch to/from guest mode that includes control registers, segment registers, and instruction pointer. Exit from guest mode generates a hardware report. 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) [7], 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 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. With Workstation 6.5, VMWare does support hardware-accelerated graphics in Windows XP guests, but this is virtualization at the graphics API level, not the card level. The VMGL system [8] 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 [6]. It is similar in spirit to Virtual GL [15], 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 [13].

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.

The Virtual Architecture, described in Section 3.1, makes 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 which structures the module as a collection of file operations that operate 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 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 card information, e.g. physical base addresses and memory sizes, 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 [10]. 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 that 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 were 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 we will describe in the next section and which accomplishes this task.

3 Virtual PCI Framework

3.1 Virtual Architecture.

The Virtual Architecture comprises three interacting code modules, shown in Figure 1, to replace the standard graphics cards: the Virtual Console Daemon, the Virtual GPU, and the Virtual PCI Bus.

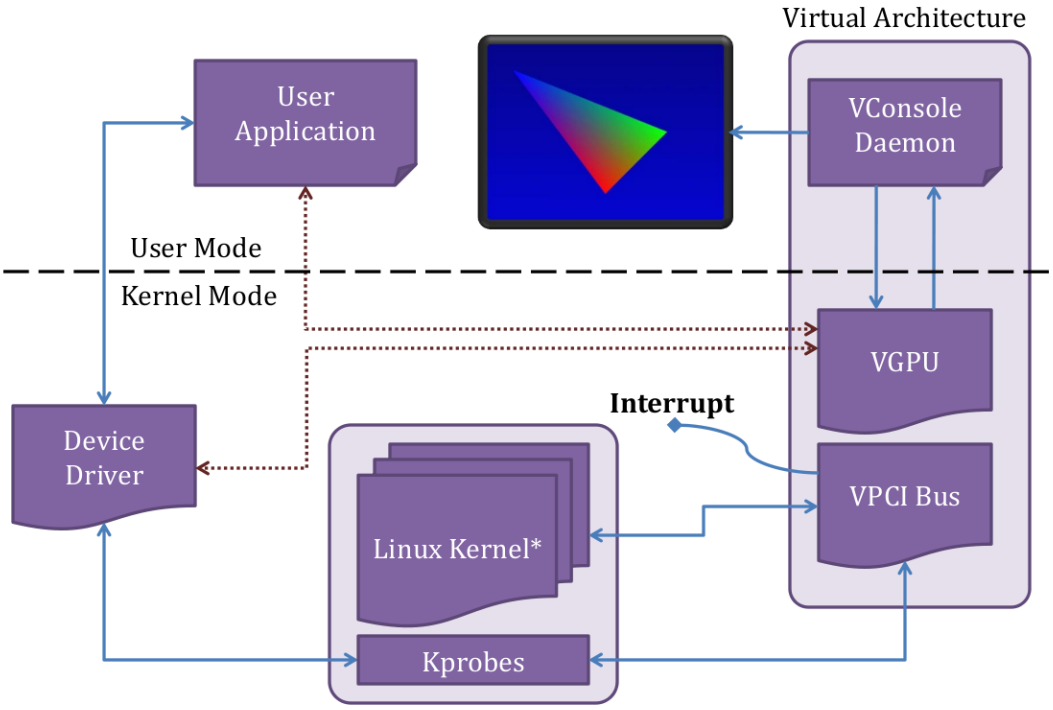


Figure 1: Virtual Architecture.

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 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 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 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 on 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 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*). 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 a *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 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 wakeup mechanism. As noted earlier, when the VCD detects no 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 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.

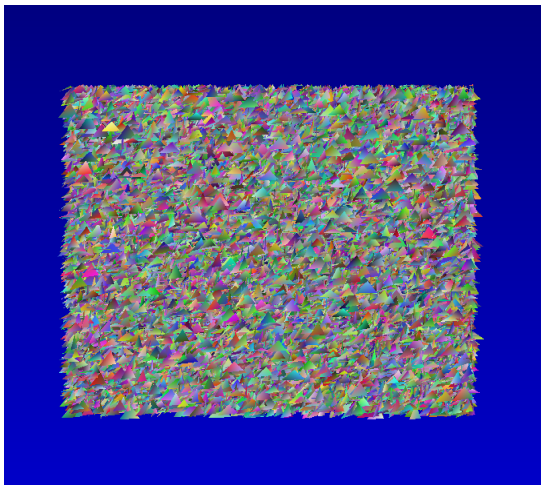
Generating an interrupt in the VGPU is relatively straightforward. On the Intel architecture, we can use the *int n* instruction with $n \geq 32$ or $n \geq 48$ on 32-bit and 64-bit systems respectively. The Linux kernel will use the Interrupt Descriptor Table (IDT) to invoke the handler registered for 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.

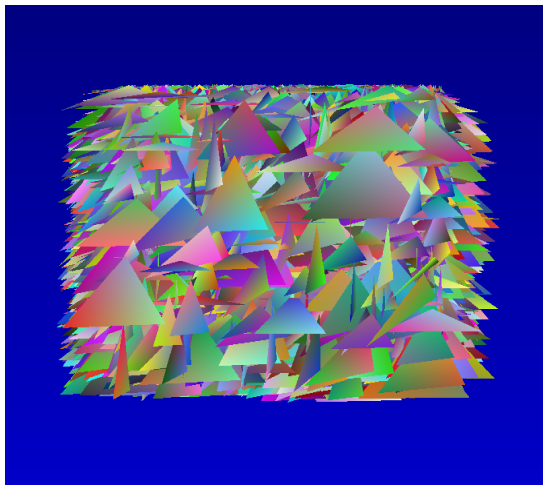
3.2 Rendering Performance.

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 [14], 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



triangles of height 20 pixels



triangles of height 100 pixels

Figure 2: Rendering Samples

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 2.

Run times were measured from the user application level using the standard Pentium cycle counter capture, asm("RDTSC"). Results are shown in Table 1. The triangle size is the height mea-

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 1: Rendering Performance Comparison

sured 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 [3]. 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 gives a distribution of the students answers.

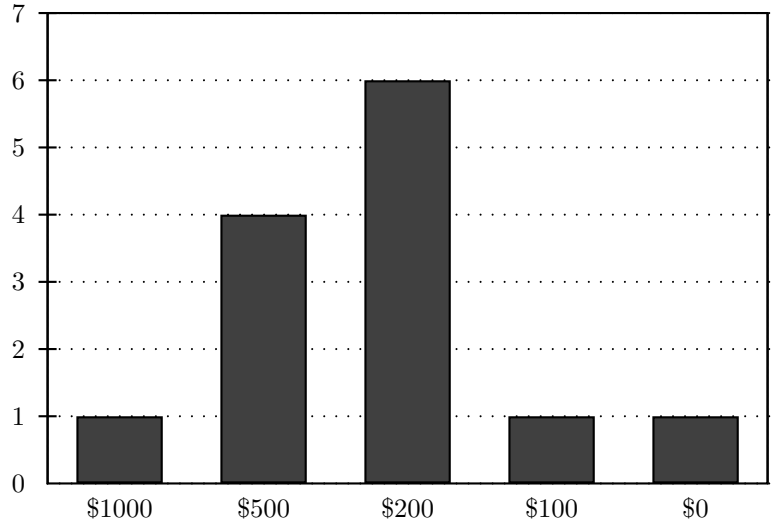


Figure 3: Student Evaluation of Price of the Zach1

4 Virtual Operating Systems Lab

4.1 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 this course. As a result, the second 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 [2]. Other tools such as libvirt [9] and the Virtual Machine Manager [16] 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.

4.1.1 Creating and Maintaining VMs

Creating a KVM virtual machine first starts with creating a virtual hard disk. Figure 4 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.

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

Figure 4: Creating a Disk Image

Next,the empty virtual disk must be loaded with a base operating system, in this case CentOS 5.3, 64-bit. The command is shown in Figure 5. Note that the drive type is specified as SCSI, not

```
qemu-system-x86_64 -drive file=822master.img,if=scsi,bus=0,unit=0
    -cdrom CentOS_64-5.3.iso
    -boot d -m 4096
```

Figure 5: Installing CentOS 5.3 on a Virtual SCSI Disk

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 Figure 6.

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

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

Figure 6: Launching a VM with SCSI Hard Disk

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 Figure 7. For 26 students in pairs we still require a minimum of 13

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

Figure 7: Creating Clone Images

virtual machines, each with a full CentOS 5.3 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.

However, 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 slow down performance. 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 standalone 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.

4.1.2 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 Figure 8) 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

```
#!/bin/bash
ssh -Y blue30
```

Figure 8: go_blue

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.

To simplify launching virtual machines and to protect the students from themselves, we provided the script *start_lab_vm* shown in Figure 9. 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."
```

Figure 9: start_lab_vm

A new addition to the virtual machines this semester is the shared second drive (*usr_local.img*) that every virtual lab machine mounts. 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 very 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 for the 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 Figure 10. Six virtual machines using X11 Tunneling will maximize CPU

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

Figure 10: Using X11 Forwarding to access VM monitor.

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.

4.2 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 5.3, the default clocksource is based on the jiffies counter and a programmable interrupt timer (PIT). Under heavy load, this clocksource 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 is certainly adequate.

4.3 Performance

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 Figure 11. 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
```

Figure 11: Launching a program pinned to the virtual CPUs 0 and 5.

Since a virtual machine is a process on the host, it can pinned to a group of CPUs in the same manner as a 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 the virtual machine to a pair of cores, as this prevents a migration that involves moving across processors or moving across L2

caches. Figure 12 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
```

Figure 12: 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 Figure 12. 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) [5]. Table 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 2: Runtime in seconds of CFD code for various configurations

5 Additional Proposed Work

5.1 Virtual Performance Timers

In addition to the graphics driver project, another 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 in a virtual machine. However, measuring the performance of virtual disks in a way that would allow prediction of the performance of 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, QEMU 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 the virtual CPSC 822 lab, the 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 disk(s) where the block resides.

While paravirtualized devices could provide increase performance, such would still not remove all the layers of abstraction. The NAS storage comprises multiple hard disks in a RAID configuration, and so individual accesses to the virtual disk can be distributed across multiple physical disks.

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 drives available. Otherwise, again, data corruption would occur when multiple virtual machines write to the same drive. Another solution is to implement a disk emulator within the OS or QEMU. Following the design goals from the introduction, we will design an emulator facility as a module extension to the Linux kernel, thereby allowing the solution to be portable and minimizing new code to study and modify. Further, with this approach we can leverage the tools, particularly the new intercept probes, in the existing framework for development.

This leads us to the concept of Virtual Performance Throttles. We start by specifying a seek model for the physical drive. The expected completion time (T) of a request is one-half the time to complete a revolution (R) plus the product of the seek time (S) and the seek distance x, expressed as a fraction of the maximum distance (D). Thus

$$T_r = \frac{R_r}{2} + S_r \cdot \frac{x_r}{D_r}$$

We can expect the linear model of the real disk to be a close fit to the observed seek times

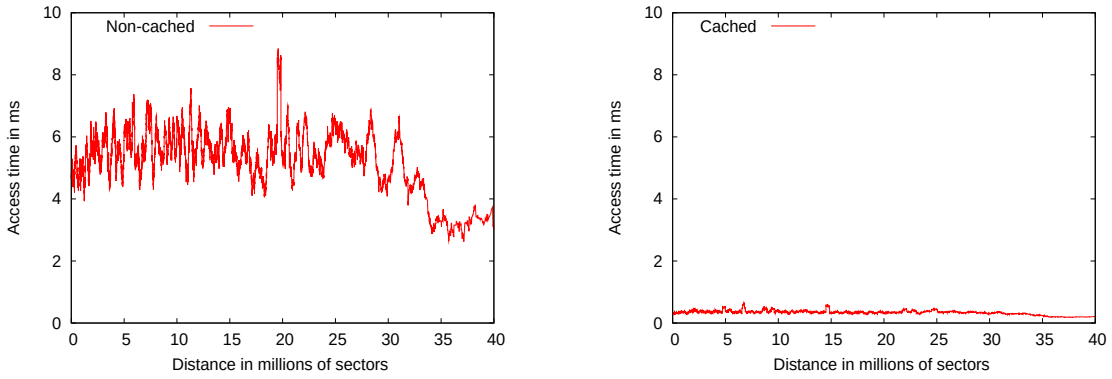


Figure 13: Seek Time of Virtual Machine Disk

from the targeted real disk. Nevertheless, the linear model of the virtual disk may exhibit wide disparity from the observed seek times on the virtual disk, since it is only a best-fit linear model. Seek times on the virtual disk may exhibit significant non-linearities due to non-linear mappings between virtual logical blocks and physical blocks. In addition to the mapping between blocks, RAM and caches in NAS devices can introduce further disparities. 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 near constant access time for all virtual logical blocks. Figure 13 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 hard disks on Solid State Drives will exhibit similar performance characteristics as the cached virtual machine.

Ideally, for a given virtual distance, x_v, and a target performance scale, k, we would like the virtual machine to see a completion time of $k\left(R_r/2 + S_r\left(x_r/D_r\right)\right) = k \cdot T_r$, but it will see a different time, T_v. To get the target completion time $k \cdot T_r$, we need to delay T_v by an amount, δ. To find δ, we need to calculate T_r for a given x_r. Therefore, we need to scale x_v to x_r using the ratio $x_v/D_v = x_r/D_r$, and calculate δ as follows

$$\begin{aligned}
\delta &= k \cdot T_r - T_v \\
&= k\left(\frac{R_r}{2} + S_r\left(\frac{x_r}{D_r}\right)\right) \\
&= k\left(\frac{R_r}{2} + S_r\left(\frac{x_v}{D_v}\right)\right)
\end{aligned}$$

We have a mapping between the virtual disk and the linear performance model of the the target physical disk. This leaves properly selecting a value for the scale factor k to reach the the targeted completion time of our targeted device. We can calculate a value for k from our model, but the model does not account for the changes in the virtual disk performance, which, as Figure 13 shows, can be drastic. Other factors that can also change the performance include server load and network congestion. A design goal of making the VPTs dynamically self-scaling is added to address this issue.

The VPT will constrain the flow of disk requests being served in the kernel based on the linear seek time model and the value of k. The VPT will use 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, will calculate 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 arriving, and the *iprobe* will then place 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 seek time on the virtual disk is subject to change, it is possible that a request will arrive to the *iprobe* after the target completion time. This informs the VPT that the scale factor, k, is too small for the observed performance and must be increased. If the VPT has a large queue of requests, we know that the target performance is slower (target times longer) than observed performance and the VPT needs to decrease k. When the VPT is dynamically scaling, it will be important to report the value of k in order to scale results properly to the physical disk being modeled. This gives us a feedback mechanism to compensate for variations in the virtual disk performance.

5.2 Virtual Device Generation Tools

Higher level software tools that assist in generating new virtual PCI devices from existing devices would be of significant value here, both in the new product development line and in the educational role, where a new device is required every semester. When considering the educational role, components of the virtual graphics card that should change are register names, register locations, register bit-field layouts, and bit-field values. Changing these do not affect the logic of the card, just how it is referenced. To simplify this, a template model of a base virtual graphics card could be used with a tool set to generate a custom card. Since the underlying functionality of the card is not being

changed, the template model could be extended to include a programmers reference manual (PRM), example device drivers, and example user space code.

Changing underlying functionality of a device, such as changing color palettes from RGB to CMYK or basic primitives, would require several underlying changes. These include adding/removing registers and changing the underlying logic. This would also require substantial changes in the section of the PRM dealing with how to render on the screen. It is possible that the template concept could be extended to create a super template that has all component choices, and the tool set would allow a user to select which components to enable. For successful creation of new devices, it is important that the template device be well organized and take advantage of code reuse when components need to be added or changed.

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.

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 believe that this system also has great application in academic instruction, and 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 in two directions. First, we are creating Virtual Performance Throttles. These will allow us to model the performance of I/O devices. We will be able to predict the performance of new algorithms and optimizations introduced into the kernel from model incorporated into a virtual machines and without the devices being studied. Second, we are creating a tool set to assist in the creation of new devices for the Virtual Architecture. These will allow for rapid

prototyping of new derivative devices with minimal code changes made by the developer. With the inclusion of these tools we believe that this will provide a complete tool set to use Linux as simulator for instruction, development, and performance research.

Bibliography

[1] 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.

[2] 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.

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

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

[12] 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.

[13] 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.

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

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

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

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