Virtualizing High-Performance Graphics Cards for Driver Design and Development

Robert Geist, Zachary H. Jones, James Westall

School of Computing
Clemson University

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 a 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.

1 Introduction.

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 2.6.26 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. The machines, under control of experimenting students, frequently crash, sometimes with disk-corrupting failures, and thus dedicated hardware has been required. Course enrollment has been restricted by platform cost and available lab space. There is always a waiting list for the 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, the most important of the course projects, building a device driver for a high-performance graphics card, remains out of reach of these tools. Most such 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 in-

struction 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 a specific, high-performance graphics card 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 across a heterogeneous collection of generic PCs or server blades.

We describe here the design and implementation of such an architecture for Linux systems. Design goals are several. 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. 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. This ensures that driver design is authentic. 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.

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 and character devices, the principal tools used in our implementation. In section 3 we provide an overview of the virtual architecture of our system. It comprises three interacting code modules, one at the user level and two at the kernel level. Sections 4 and 5 describe two very different performance evaluations of our system. The former evaluates the rendering performance of our virtual graphics card, and the latter evaluates the performance of a class of graduate students who were given the task of writing drivers for the virtual card. Conclusions follow in section 6.

2 Background.

2.1 Related Work.

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 [3]. 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 [8]. 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 [9]. For completeness, we include two examples here.

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.

The POPF (pop flags) instruction poses another type of challenge. When the processor is executing at privilege level 0, POPF 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 problem by using a binary translation of guest OS code [13]. Xen [1] provided an open-source virtualization of the x86 using *para-virtualization*, 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. 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) [5], 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 still export only basic, SVGA graphics cards to the guest systems. 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. With the lack of standardization and proprietary interfaces for GPUs, virtualization at the graphics API level has become the focus area of rapid development. The VMGL system [6] 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. It is similar in spirit to Virtual GL [12], 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 [10]. VMGL uses the machine's loopback interface and a transport based on WireGL [4].

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 here.

2.2 Kernel Modules and Character Devices.

The Virtual Architecture, described in the next section, 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 that 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. Character devices can be entirely logical constructs. The file operations that op-

erate 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. Here we implement only file operations *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.

3 Virtual Architecture.

The Virtual Architecture comprises three interacting code modules, shown in Figure 1. 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 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 can easily be configured to run without borders or icons, which gives the appearance of an "empty" underlying framebuffer. Graphics primitives are generated 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 to generate an interrupt to the 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.

The VGPU is a 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. Obviously, the standard Linux command used by drivers to scan the bus, *pci_register_driver()*, must be intercepted. We intercept this, as well as several other commands to be described, using Linux *kprobes* [7].

The *kprobe* utility was designed to facilitate kernel debugging. There are several varieties, but the basic operation is the same. 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.

Note that through careful use of the pre-handler and post-handler, an entire kernel function can be replaced with an alternative version. The *jprobe* variation 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,

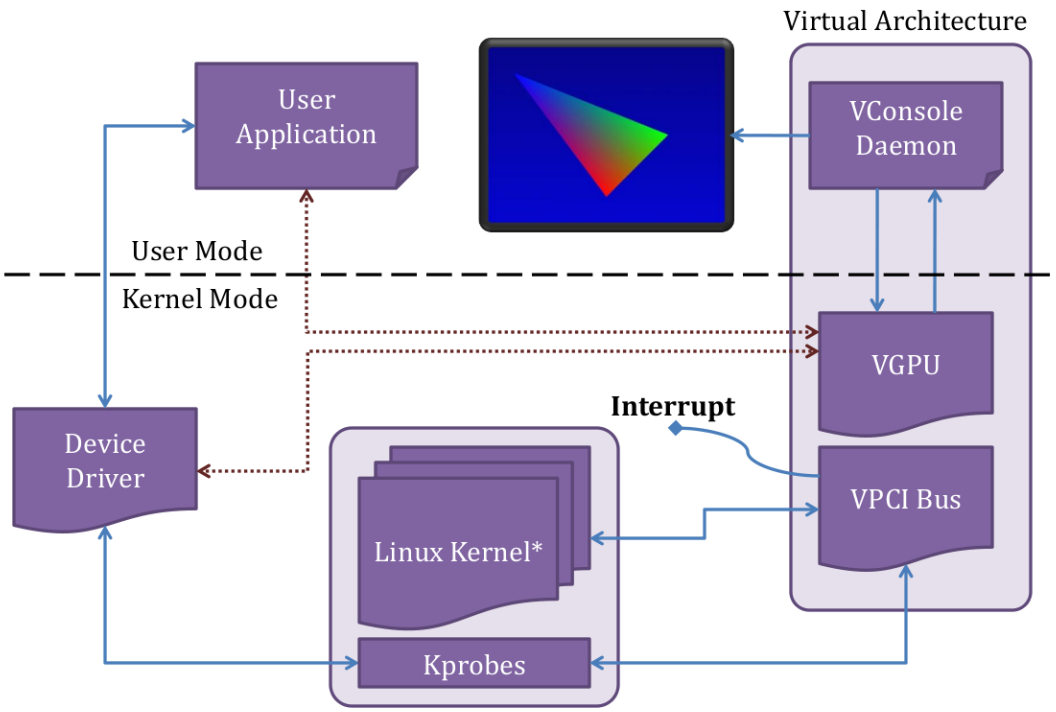


Figure 1: Virtual Architecture.

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. Our second-stage pre-handler then decides whether or not to replace the original function. If it decides to do so, it makes a backup copy of the saved 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. Our post-handler checks to see if replacement was called for by the second-stage pre-handler. If so, the single-stepped instruction was a no-op, and so 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. Thus we can intercept and 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 in-

struction. 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.

In addition to the functions already mentioned, we intercept *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 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.

The Virtual PCI Bus (VPCIB) is a second Linux kernel module which actually contains most of the functionality we have described, including the intercepting *kprobes*. 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 VPCIB, with which lightweight, device-specific kernel modules may then register and share in its exported functions. 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.

Generating an interrupt in the VGPU is relatively straightforward. On the Intel architecture, we can use the *int n* instruction with $n \geq 32$. The Linux kernel will use the Interrupt Descriptor Table (IDT) to invoke the handler registered for IRQ $n - 32$. 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. If the VCD decides to sleep before the interrupt handler completes, it may not be awakened. In this case, we have to attach another special type of *kprobe*, called a *kretprobe*, which executes on kernel function completion, to the driver's interrupt handler.

4 Rendering Performance.

As noted earlier, the goal of the project is not to achieve high-speed rendering but rather to provide a completely portable platform for driver design and development. The rendering performance of the virtual architecture cannot possibly match that commonly seen from executing directly on hardware GPUs. 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 [11], 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. The operating system was Linux 2.6.26.

At the user application level, the tests used the card's DMA capability to render 1 million smooth-shaded triangles as quickly as possible. 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

Triangle Size	Real sec.	Virtual sec.	Slowdown %
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

height 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.

5 Student Performance.

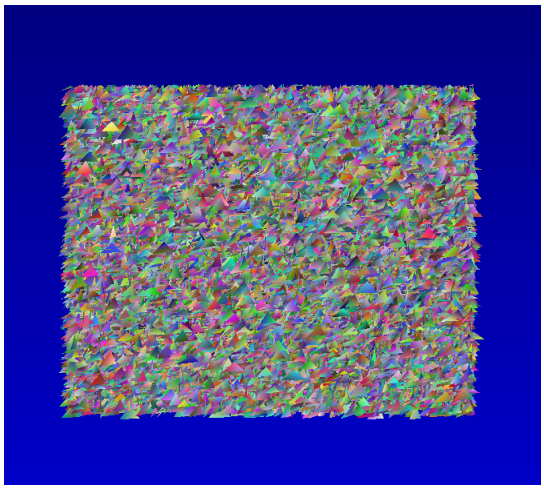
We tested the feasibility of using the virtual architecture for driver design and implementation in CPSC 822 during the Spring semester of 2009. Student teams with 4 graduate students per team were given hardware reference manuals and programmer reference manuals for the virtual card described in the previous section. Teams were assigned to specific machines on which we had installed the virtual architecture. They were given specifications for a graphics card driver design that detailed both the capabilities that the driver was to deliver and the interface it was to provide to the application layer. Students were not told that the card was virtual.

They were given four weeks to complete the project. 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 [2].

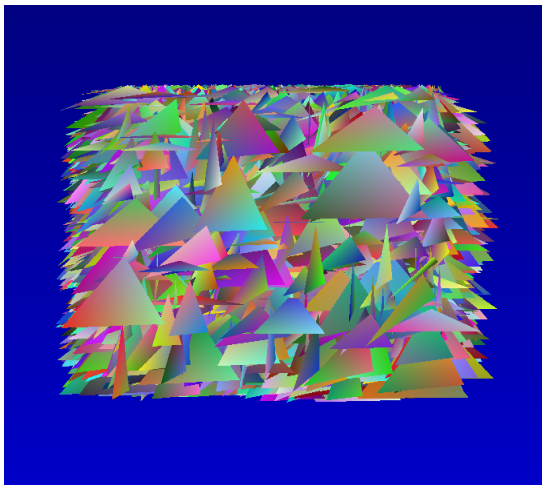
All of the teams delivered an operational driver on time. This was somewhat unusual, compared to the collective performance of teams working on real hardware in previous semesters. In most previous semesters, at least one team had serious driver faults. We tentatively ascribe this to the fact that the 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 queueing. 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 an-



triangles of height 20 pixels



triangles of height 100 pixels

Figure 2: Rendering Samples

swered, "$0". The others gave estimates ranging from $100 to $1,000, with an average above $200.

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. Most did not even realize that the card was virtual. Rendering performance through the virtual card was not strong, but acceptable. We conclude that we have met design goals, 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 kernel, the *do_page_fault()* signature carries a declaration, *__kprobes*, 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.

Current development is proceeding in two directions. First, a substantially different virtual PCI device that "plugs in" to the Virtual PCI Bus would significantly expand system flexibility. The design of a virtual gigabit Ethernet interface card, having operational characteristics similar to the Intel®Pro/1000 class of devices, is underway but still in its infancy. It is expected that this task will essentially drive the full development of the Virtual PCI Bus specification.

Second, 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.

Acknowledgments

This work was supported in part by an IBM Faculty Award and by the U.S. National Science Foundation under Award 0722313. Au-

thor Jones is supported by an IBM Ph.D. Fellowship for the 2009-2010 academic year.

About the Authors

Robert Geist is a Professor in the School of Computing at Clemson University. His research interests include operating systems performance, computer graphics, and visualization. He received a Ph.D. in mathematics from the University of Notre Dame.

Zachary H. Jones is a Ph.D. candidate in the School of Computing at Clemson University. His research interests include operating systems, high performance computing, and algorithmic optimization.

James Westall is a Professor in the School of Computing at Clemson University. His research interests include measurement and modeling of computer systems and networks, computing education, and computer graphics. He received a Ph.D. in mathematics from the University of North Carolina at Chapel Hill.

References

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

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

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

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