A Virtual Graphics Card for Teaching Device Driver Design

Christopher Corsi
School of Computing
Clemson University
Clemson, South Carolina
ccorsi@clemson.edu

Robert Geist
School of Computing
Clemson University
Clemson, South Carolina
geist@clemson.edu

Dennis Lingerfelt
School of Computing
Clemson University
Clemson, South Carolina
dlinger@clemson.edu

ABSTRACT

Open source Linux has become increasingly popular as a vehicle for incorporating hands-on experience with a real system into both undergraduate and graduate operating systems courses. System virtualization tools, such as VMWare, Xen, VirtualBox, and KVM, allow students to freely experiment with kernel modifications without requiring dedicated hardware and without generating significant concern for the ill-effects of system crashes. Nevertheless, certain kernel projects that are highly desirable from an educational standpoint remain unavailable under standard approaches to virtualization. One such project that is known to carry substantial instructional value is the design and implementation of an SMP-safe driver for a high-performance graphics card. Standard virtualization tools export only a minimally-capable, SVGA graphics adapter, which is an inadequate architecture for such a project. This paper describes an extremely simple, kernel-independent, software tool for use by instructors of operating systems courses. The tool provides a virtual, high-performance graphics card that is suitable for Linux device driver design and implementation. The code for the virtual card, which is relatively short, is easily modified by instructors to present different interfaces each semester. The code for both the virtual card and a sample Linux 3.2.36 driver for it may be freely downloaded from *http://people.cs.clemson.edu/∼ccorsi/kyouko/*.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design;
D.4.0 [**Operation Systems**]: General—*Linux*;
K.3.2 [**Computers and Education**]: Computer and Information Science Education

Keywords

Operating Systems, Virtualization, Graduate Education

1. INTRODUCTION

The classical undergraduate course in operating systems, which focuses on operating systems principles, imparts some valuable information on operating systems theory, but it leaves students woefully unprepared for careers in modern systems software design. The complexity of any real operating system is comparatively enormous, and the simplicity suggested by a study of principles alone belies this complexity. As a result, many educators have begun to take advantage of open source Linux in order to include significant hands-on experience with a real system in their operating systems courses [5, 7, 9]. This has been the approach taken in the graduate-level course, CPSC 822, at Clemson University for many years. This course is a structured walk-through of the current Linux kernel, and it includes several student projects that require significant modifications to the kernel, most often to improve performance on targeted workloads.

Until recent years, in spite of high demand, the course has had severely limited enrollment. Students writing system-level code frequently crash their systems, often with disk-corrupting failures. Thus the course has required both dedicated hardware, one machine per student team, and the laboratory space in which to house it, which has collectively represented a large expense for a single course. The advent of x86 system virtualization, through *VMWare* [12], *Xen* [1], *VirtualBox* [4], or Linux *KVM* [8], coupled with the Intel's VT-x and AMD's AMD-V hardware extensions, opened the door to the possibility of using virtual machines for the course, which would reduce costs and increase course availability.

Nevertheless, a significant obstacle remained. The most important course project, both in terms of class time allocated and subsequent evaluation by course graduates and employers thereof, has always been the design and implementation of an SMP-safe device driver for a modern, high-performance graphics card. The importance of the project derives from the fact that an effective device driver for such a card is a microcosm of the operating system itself. To deliver high performance at the user level, the driver requires memory management, DMA transfers, resource allocation, security, process scheduling, process communication, and interrupt handling. The obstacle to complete course virtualization is then clear: available system virtualization tools (VMWare, Xen, VirtualBox, KVM) have traditionally exported only a minimally-capable, SVGA graphics adapter, which is an inadequate architecture for such a project. In recent years, there has been a push to provide 3D graphics acceleration to guests in both VMWare and VirtualBox. Nevertheless, their virtual devices are targeted solely at graphics acceleration. The VMWare device is proprietary, and the VirtualBox device does not provide an interface suitable for driver design.

Starting in 2009, we solved this problem using the technique proposed by Geist et al. [6]. They constructed a functional emulator of a 3DLabs Permedia II card [11] that was structured as a pair of communicating kernel modules together with a user-space background daemon. The background daemon handled screen up-

SIGCSE'14 March 05–08 2014, Atlanta, GA, USA

http://dx.doi.org/10.1145/2538862.2538895.

dates after reading a page of virtual device register values that had been memory-mapped from one of the kernel modules. In order to present the students who were writing drivers with a completely unmodified kernel source tree, numerous kernel functions were dynamically intercepted and replaced using a modified version of the Linux *kprobes* facility. The fact that the card was virtual remained essentially invisible to the students.

This approach was quite effective. The students became adept at driver design, and they particularly enjoyed the visual feedback afforded by a successful driver implementation. Nevertheless, the module-daemon communication code and the *kprobes*-based code were both elaborate and delicate. Linux kernel authors apparently have little regard for backward compatibility of internal structures. Each kernel version change broke the virtual graphics card, and this required extensive revision of the enabling code. Such changes occurred every semester. Thus we have sought a new solution, one that is, to the largest extent possible, independent of particular Linux kernel versions, is easily modified by course instructors to present a new architecture each semester, and is capable of emulating sophisticated card architectures with on-board memory, multiple modes of execution, DMA transfer, interrupt handling, and memory mapping of on-board registers. We present here the design of such a solution.

The remainder of the paper is organized as follows. In the next section we briefly review related work in using Linux as an instructional tool in operating systems courses. The design of our virtual graphics card, called *Kyouko*, follows, after which we describe the design of key components of a Linux driver for this card. A performance comparison of rendering speed using the new virtual card, the *kprobes*-based virtual card, and the underlying, real hardware is presented in Section 5. Student outcomes from the first use of the new virtual card in CPSC 822 are presented in Section 6, and conclusions follow in Section 7.

2. RELATED WORK

Clearly the idea for creating a virtual graphics card that is suitable for teaching device driver design was taken from the work of Geist et al. [6]. Our goal here is simply to provide an alternative virtual graphics card design (and sample driver) that is smaller, faster, portable across a broad range of Linux kernel versions, and whose code is easily modified by instructors to provide fresh card designs each semester.

We were also motivated by the work of Laadan et al. [9], whose course of instruction now includes a wide range of interesting Linux projects on system calls, synchronization, scheduling, virtual memory, and file systems. We would characterize their projects as kernel-inward facing and our own projects as kernel-outward facing. The two sets are quite complementary in nature.

In earlier work using Linux kernel projects in an undergraduate operating systems course, Hess and Paulson [7] observed that example problems from standard principles courses, "skirt the issue of operating systems practice," and "really only reinforce operating systems theory". They further argue that protecting senior undergraduates from the details of a production operating system is neither necessary nor beneficial. Although our own course is graduate level, it is frequently taken by our stronger undergraduates, and we completely concur with Hess and Paulson's conclusions.

Finally, Gaspar and Goodwin [5] argue for including hands-on Linux experience in an operating systems course by focusing on the use of loadable kernel modules. This mechanism allows the students to quickly develop new kernel code from scratch without first having to master an enormous code base. Our device driver design project fits nicely within this argument.

3. THE KYOUKO DEVICE

The key to building a virtual graphics card that is essentially independent of the Linux kernel, and so largely immune to the transient nature of its internal structures, was to build the virtual card outside the kernel and inside the emulator. QEMU (Quick Emulator) [2] is a dynamic translator that is often used in conjunction with KVM to run unmodified target operating systems inside virtual machines. QEMU itself is supported on several host operating systems, and, most importantly, QEMU has a relatively stable, slowly changing code base.

The available virtual graphics cards in QEMU include a Cirrus CLGD 5446 PCI VGA card, a standard VGA graphics card with Bochs VBE, a VMWare SVGA II compatible adapter, and a QXL paravirtual card designed for the SPICE protocol. Although none of these was suitable for our purposes, QEMU is open-source and distributed with an appropriate build tree that makes rebuilding from source very easy. Thus we could build a new virtual card of our own design. Our design was based on the VMWare SVGA adapter, found in ∼/hw/vmware_vga.c and ∼/hw/vmware_vga.h. Nevertheless, our virtual card provides very different capabilities, and it is much smaller, 556 lines of code rather than 1,070 lines.

We wanted the virtual card to provide two modes of operation, FIFO and DMA. Under FIFO, the driver can write values to our on-board device registers, but these writes are indirect. The bank of virtual device registers, which contains both state registers and command registers, is protected. Register writes by the driver are captured and stored as register-value pairs in an on-board FIFO queue. An internal *Kyouko* call, *vgpu_fifo_process( )*, removes entries from the queue and either updates state or executes the command. Since FIFO queue space is finite, the driver could attempt to overload it. In this case the commands are lost. So that the driver may avoid this, a readable register, *InfFIFO*, always contains the number of available slots in the queue.

Under DMA, the *Kyouko* virtual card will download a driver's DMA buffer to an on-board buffer of size 128KB and then execute the commands therein through the internal *Kyouko* call, *vgpu_dma_process( )*. The driver initiates this action by writing (under FIFO) the physical address of its current DMA buffer to the *CmdDMABuffer* register and a count of the number of commands in the buffer to the *CmdDMACount* register.

The internal operation of the virtual card is relatively simple. Initialization is encapsulated in a single function, *pci_kyouko_initfn( )*. The first important task of initialization is allocation of memory for the virtual device registers. This is done with a QEMU call, *memory_region_init_rom_device()*, which allocates the physical space for the virtual registers and then sets the write callback function that will load the FIFO queue on each attempted write to the register bank by the driver. We somewhat arbitrarily use one page (4KB) for the size of this register bank. The virtual framebuffer (8MB) is allocated by calling the QEMU function *vga_common_init( )* and is used for VESA device emulation. The second important task is to register both the virtual framebuffer and the memory region containing the virtual device registers with the PCI bus so that the driver may locate them during its bus probe. This is done with the QEMU command *pci_register_bar( )*. The framebuffer is assigned to PCI resource 0, and the register bank is assigned to PCI resource 1. The third essential task is initializing the MSI (Message Signaled Interrupt) system with a call to QEMU function *msi_init( )*. MSI interrupts are rapidly replacing the standard, PCIe INTx interrupts because the latter requires requires interrupt sharing and MSI does not. We want the *Kyouko* card to generate an interrupt on each DMA buffer completion, so that the driver may specify the next buffer of commands, if there is one waiting, and wake any user

process that may have suspended awaiting an empty driver buffer, that is, available driver DMA memory space. Finally, we register our basic update function, *kyouko_update_display( )*, with QEMU using the QEMU function *graphic_console_init( )*.

After initialization, control is completely in the hands of this basic update function, which is repeatedly called by QEMU. This function first examines virtual card state to determine whether a mode switch (VGA compatibility mode to graphics mode or vice versa) has been called for, and, if so, it effects the switch. If the switch was into VGA compatibility mode, *vgpu_fifo_process( )* is called, and then the update returns. If the switch was into graphics mode, *vgpu_fifo_process( )* is called, then *vgpu_dma_process( )* is called, and then the update returns.

The design of the *Kyouko* command set and supporting registers is completely flexible. The current layout (available sample) is designed to emulate the basic structure of modern devices such as the Intel HD 3000 [3], although it is greatly simplified to make it suitable for a class project. We anticipate that instructors will change it frequently to keep the device driver project fresh each semester. QEMU runs in user address space, and so *Kyouko* command execution can employ standard, user-level libraries. We use OpenGL. QEMU by default utilizes the Simple DirectMedia Layer (SDL) for graphical output, and this is supported by a display driver located in the QEMU source, in ∼/ui/sdl.c. We added an OpenGL display driver, whose source may be downloaded, along with that of the *Kyouko* card and the sample card driver, from the authors' website. This display driver creates the necessary OpenGL rendering context, and it must be paired with *Kyouko* card.

Having OpenGL available offers a practically limitless set of choices for the *Kyouko* primitive command set and state registers. Basic commands can range from very low level, in which only a handful of primitive graphic elements are available, to very high level, in which register state includes lighting, shading, texturing, and view transformation values. As an example, code for the *Kyouko* command *CmdVertex* is shown in Figure 1, where s→vtx contains vertex state upon entry.

```
case CmdVertex:
// Save time if there's been no state change.
if(s→vtx.vtx_dirty){
      glColor4fv(s→vtx.color);
      glNormal3fv(s→vtx.normal);
      glTexCoord2fv(s→vtx.texcoord);
      s->vtx.vtx_dirty = false;
      }
// Emit a vertex.
glVertex4fv(s→vtx.position);
break;
```

Figure 1: Sample Kyouko Command Implementation

The complete *Kyouko* command and state register space is shown in Table 1. This is, in effect, a 1-page hardware reference manual.

4. DRIVER DESIGN

We describe a Linux 3.2.36 driver for the *Kyouko* card. In principle, writing a device driver for a PCI card is entirely straightforward. After the card is inserted into an available motherboard slot, its command registers appear at fixed offsets from a physical base address. The driver then merely loads these registers with bit strings that comprise commands understood by the device hardware. In practice, it is more difficult. Linux, like most operating systems, uses paged memory management with both kernel mode and user mode address translations. Devices have no knowledge of these address translations, and yet they often operate independently of the CPU to transfer data under DMA control and then act upon it. Devices also issue interrupts, which are usually legitimate requests for service from the CPU, but they can be spurious. Even in a single-user environment, interrupt handlers may execute concurrently with service initiation requests, and they may attempt to access the same registers. Finally, ensuring good device performance demands effective management of on-board device memory, which is usually a scarce resource.

Our *Kyouko* device is treated as a standard Linux character device, and the driver is structured as a dynamically loadable kernel module. Linux character devices are created with the *mknod* command, and they support a collection of functions with fixed signatures, defined in the kernel structure *struct file_operations*. Our sample driver implements only four of these functions: *open, release, mmap,* and *ioctl*.

Linux kernel modules are collections of C functions that extend the kernel. They may be dynamically loaded with the command */sbin/insmod* and unloaded with */sbin/rmmod*. There is no "main", but two of the functions are special. An initialization function, identified by the macro *module_init*, is executed upon load, and an exit function, identified by *module_exit*, is executed upon removal.

Our initialization function loads a kernel structure of type *struct cdev* with major and minor *Kyouko* device numbers (# defines) and pointers to the four implemented file operations. It then calls *pci_register_device* which will launch a bus search for the matching (major, minor) pair. When the device is found, a driver-defined probe function is called to recover the physical base addresses of the frame buffer (resource 0) and the register bank (resource 1). The probe function will also enable bus-mastering on the device.

The driver's *open* function, which is invoked by a user-level call to open the file */dev/kyouko*, is nearly trivial. Its only real function is to call the kernel's *ioremap* to translate the physical base address of the register bank, recovered by the probe, into a kernel virtual address. We need the kernel virtual address to initialize the card (write to the card registers) from within the driver. All recovered and translated addresses are stored in an internal driver structure.

The driver's *release* function, which is invoked by a user-level call to close the device, is only slightly more involved. Its principal task is to release allocated resources (memory and interrupt lines) and ensure that the card has been left in VGA text mode.

The *mmap* function is not used in the final version of our driver, but, during early driver development, students are encouraged to use this function to memory map the card device registers back to user space for direct loading. This helps them quickly reach a point where they are comfortable with card settings under FIFO operation. Memory mapping the device registers is a 1-line call to the kernel function *io_remap_pfn_range*, where one argument is the physical base address of the device registers, right shifted by 12 bits (PAGE_SHIFT).

Most of our driver is devoted to the *ioctl* function, which is often termed the "Swiss Army Knife" of system calls, in that one of its arguments is a command string that may be used inside a driver switch statement to effectively implement many commands in one. We have three such command strings, *BIND_DMA*, to initialize driver DMA operation, *START_DMA*, to launch a filled buffer of *Kyouko* commands, and *VMODE* to switch video modes between VGA text and graphics modes of various resolutions.

The *BIND_DMA* command initializes the MSI system and uses *request_irq* to attach the driver's interrupt handler function to the interrupt that will be generated by the *Kyouko* card upon each buffer

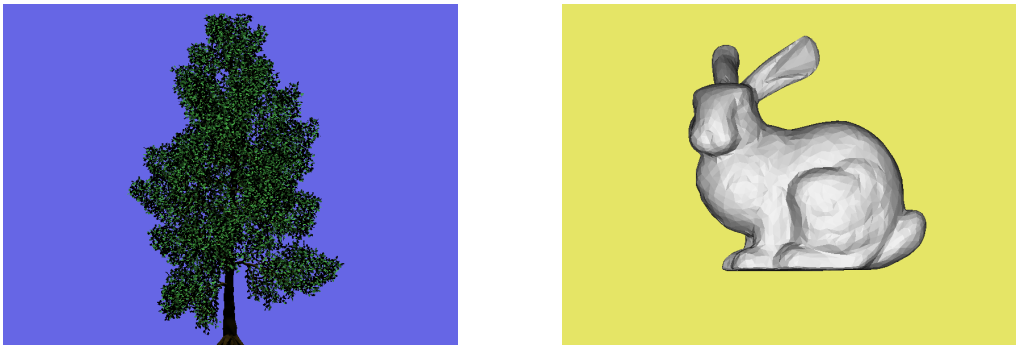

Figure 2: Sample Renders from the *Kyouko* device

completion. It allocates driver-resident DMA buffers and organizes them as a circular pool. It memory maps these buffers to the calling user's address space and returns, as a function value, the user virtual base address of the first available buffer. The user then simply loads the buffer with *Kyouko* commands and calls *START_DMA*, which will cause the driver to invoke the card's DMA operation by loading the *CmdDMABuffer* and *CmdDMACount* registers. In response to the call to *START_DMA*, the driver will return, as a function value, the user virtual base address of the next available buffer.

Of course, the user may fill buffers faster than the card can download and execute them. If the call to *START_DMA* fills the last available buffer in the circular pool, the driver must suspend the calling user process. Thus the driver side of the *START_DMA* call and the interrupt handler together function as a classical producer-consumer pair with respect to the driver's pool of DMA buffers.

We maintain two indices into the pool, *fill* and *drain*, indicating which buffer is in play, if any, by the producer and by the consumer. The condition *fill* = *drain* does not, by itself, determine whether the buffer pool is empty or full, but such can be determined from context. When the pool fills, the driver suspends the user on a kernel wait queue with a call to *wait_event_interruptible*.

Access to the buffer pool is guarded by a spinlock, and this introduces a complication that many students find challenging during their driver design. Upon discovering the need to suspend the caller, the driver's *START_DMA* code must first release the spinlock. If the condition supplied to *wait_event_interruptible* were simply *fill != drain*, a rapid fire of completion interrupts after the lock release and prior to the conditional suspension could move the buffer pool from full to empty before the condition is tested. At this point, the condition would be false, and the user process would suspend forever. Thus we suggest an additional boolean, *queuefull*, which is set prior to the lock release and reset only by the interrupt handler. The wait condition is then *!queuefull*. When the interrupt handler discovers *fill* = *drain* upon initiation, it does the reset and sends the wakeup to the wait queue.

5. PERFORMANCE

We ran benchmarks to compare the relative performance of the two virtualized graphics cards, the *kprobes*-based card and the *Kyouko* card. We included baseline tests with native OpenGL using an NVIDIA Quadro FX 5600 as a reference.

All tests were conducted using an Intel i7-980X CPU with 6 physical cores, 12 threads, and 12 GB of RAM. The hypervisor OS was Linux Fedora 16. The *kprobes*-based card was run using VMware's VMplayer 5.0, and the *Kyouko* card used QEMU 1.0.50. Three million smooth-shaded triangles were rendered at randomized positions on each system, and timing information was gathered. The slowdown factor, the ratio of virtual card performance to native OpenGL performance, is shown in Table 2.

	Execution Time (s)	**Slowdown Factor**
Native OpenGL	2.043	1.00
Kprobes-based	22.07	10.8
Kyouko	11.83	5.79

Table 2: Comparison of Device Performance

Although there is still overhead with a virtual hardware layer, the performance obtained by integrating directly with the hypervisor shows a significant improvement over the *kprobes*-based approach. This allows for more interesting scenes to be rendered or, equivalently, allows a greater number of virtual machines on shared hardware. Two sample renders using the *Kyouko* virtual card are shown in Figure 2. The first scene consists of a paper birch tree model with 307,336 faces. The second scene is of the well-known Stanford Bunny model and contains 4,804 faces. The two scenes, including initialization of their graphics context in the driver, took 3.31s and 1.00s to render, respectively.

6. STUDENT OUTCOMES

The *Kyouko* card was used for the first time as the target architecture for the device driver design project in CPSC 822 during the Spring semester of 2013. We compare here the drivers produced by the student teams with those produced in the same course during the Spring semester of 2012, when the *kprobes*-based card was used. The same instructor (author Geist) taught both courses.

The assignment, which was identical in both semesters, except for the virtual card name, is shown in Figure 3.

In each semester there were 6 student teams. All teams in both semesters were able to produce a driver and attendant user-level code that would put smooth-shaded triangles on the screen. Nevertheless, there were marked differences, particularly in SMP safety. There were no SMP safe drivers produced in Spring of 2012 for the *kprobes*-based card, although some of the drivers were quite robust and would require highly improbable sequences of events to violate safety. In contrast, 4 of the 6 teams in the Spring of 2013 produced SMP safe drivers for the *Kyouko* card.

We used the C and C++ Code Counter [10] to measure lines of code, number of modules, and McCabe's cyclomatic number for each group, and then we conducted a t-test for differences in means. The results are shown in Table 3.

We see that the *Kyouko* drivers were almost certainly smaller, probably more modular, and probably not different from the *kprobes*-based drivers in overall complexity.

CPSC 822 Project 1 A Device Driver January 24, 2013

Write a device driver for the Kyouko PCI graphics card that resides in each of the virtual Linux machines of the IBM "blue" bladeserver. The device driver should be built as a kernel module and should be structured so that the card appears as a character device (major 500, minor 127) accessible through system calls open(), close(), mmap() and ioctl().

The driver must deliver sufficient capability to the user level to enable the user to draw smooth-shaded triangles. Further, the driver must provide this capability in two ways:

1. directly, through the FIFO facility, using memory-mapped control registers, where the memory-mapping is invoked by a user-level system call to mmap()
2. indirectly, through the DMA facility, using driver-allocated DMA buffers that have been memory-mapped to user space

For the latter, DMA completion interrupt handling must be included (no polling), and the driver must be SMP safe. Complete card specifications are available on the class web page. Solutions, which must include both device driver code and user-level code that demonstrates driver capability, are due February 28.

Figure 3: Device Driver Assignment

(means)	*Kyouko*	*kprobes*-based	p-value
Lines of Code	230	413	0.0012
Module Count	3.2	2.2	0.1248
Cyclomatic Number	39.6	45.8	0.4687

Table 3: Comparison of Code Metrics

The difference in SMP safety was certainly the most important factor reflected in the course grade differences. The average team grade on this project in the Spring of 2012 was 77.72 (B), and the average team grade in the Spring of 2013 was 82.45 (B+/A-).

Although the empirical evidence is limited, we believe that all of these differences may be attributed to the differences in virtual card architectures. Integrating the *Kyouko* device directly into the emulated hardware allowed that card to have a smaller collection of virtual device registers and a more powerful set of primitive instructions than those available in the *kprobes*-based card. As a result, the students spent less time exploring the effects of the various register settings that were required to draw triangles and more time designing the driver's internal structures and operation.

7. CONCLUSIONS

We have described a new software tool for use by instructors of operating systems courses. The tool provides a fast, portable, kernel-independent, virtual graphics card of reasonably sophisticated design that is suitable for teaching device driver design to senior undergraduates and beginning graduate students. The virtual card architecture is easily modified by instructors to present alternative interfaces, so that a different functionality may be presented to students each semester. Since the virtual card executes in user space, debugging a new design requires little more than a few *printf* statements.

We used the *Kyouko* virtual card as the target architecture for the device driver design project in CPSC 822 at Clemson University for the first time in the Spring semester of 2013. We compared the results with data from the Spring semester of 2012, where a kernel-dependent virtual card based on the Linux *kprobes* facility [6] was used. A striking difference was in the SMP safety of the drivers. No drivers for the *kprobes*-based card were SMP safe, but two-thirds of the drivers for the *Kyouko* card were SMP safe.

We are investigating the construction of a meta-system that would automatically generate new virtual devices of this type from very high-level specifications. The meta-system would apply to devices of many classes including graphics cards, sound cards, and network interface cards. The meta-system would produce, as its output, source code suitable for input to the QEMU build tree. This would extend the range of our efforts beyond educational tools and into the realm of device prototyping. Driver design, development, and testing could then proceed in parallel with new hardware prototyping, which could reduce time to market for new devices.

8. 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] F. Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the USENIX Annual Technical Conference*, pages 41–46, Anaheim, CA, 2005.

[3] Intel Corp. 2010. Intel Hd Graphics Opensource PRM: Vol. 1 part 1: Graphics core. Retrieved Oct. 22, 2013 from http://www.x.org/docs/intel/HD/IHD_OS_Vol_1_Part1_BJS.pdf.

[4] Oracle Corp. 2013. Oracle VM VirtualBox User Manual. Retrieved Oct. 22, 2013 from https://www.virtualbox.org/manual/UserManual.html.

[5] A. Gaspar and C. Godwin. Root-kits & loadable kernel modules: exploiting the linux kernel for fun and (educational) profit. *J. Comput. Sci. Coll.*, 22(2):244–250, December 2006.

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

[7] R. Hess and P. Paulson. Linux kernel projects for an undergraduate operating systems course. In *Proceedings of the 41st ACM technical symposium on Computer science education*, SIGCSE '10, pages 485–489, New York, NY, USA, 2010. ACM.

[8] 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.

[9] O. Laadan, J. Nieh, and N. Viennot. Structured linux kernel projects for teaching operating systems concepts. In *SIGCSE '11*, pages 287–292, 2011.

[10] T. Littlefair. 2013. C and C++ Code Counter. Retrieved Oct. 22, 2013 from http://sourceforge.net/projects/cccc/.

[11] Texas Instruments. 1997. TVP4020 Permedia 2. Retrieved Oct. 22, 2013 from http://ftp.netbsd.org/pub/NetBSD/misc/cegger/hw_manuals/3dlabs.

[12] VMWare, Inc. 2007. Understanding Full Virtualization, Paravirtualization, and Hardware Assist. Retrieved Oct. 22, 2013 from http://www.vmware.com/files/pdf/VMware_-paravirtualization.pdf.

Offset	Register	Access	Type	Description
Immediate Register Space				
Configuration Registers				
0x0000	CfgSupported	RO	Boolean	1: Supported, 0: Not supported
0x0004	CfgMode	R/W	Bitfield	Bit 0: Graphics, Bit 1: Transform, Bit 2: Lighting, Bit 3: Texturing
0x0008	CfgAccel	R/W	Bitfield	Bit 0: 2d-Accel, Bit 1: 3d-Accel
0x000C	CfgWidth	R/W	Unsigned 10-bit	Width in pixels of requested mode
0x0010	CfgHeight	R/W	Unsigned 10-bit	Height in pixels of requested mode
0x0018	CfgFrame	R/W	Bitfield	Bits 0-3: Red bits, Bits 4-7: Green bits, Bits 8-11: Blue bits, Bits 12-15: Alpha bits, Bits 16-23: Depth bits, Bit 24: Double buffered
0x001C	CfgFlags	R/W	Bitfield	Flags indicating current state of card. Bit 0 indicates whether a DMA transfer is pending acknowledgment (write 0 back to this bit to indicate it has been acknowledged). Bit 1 indicates whether the card has encountered an error. If the error bit is set, the card will not respond to FIFO commands until the bit is reset. The card may be in an undefined state depending on the type of error.
0x0020	CfgFeatures	RO	Bitfield	Bits 0-7: Card revision number, Bits 8-15: Card Vendor, Bit 16: Texturing support, Bit 17: Lighting support, Bit 18: Extended DMA features
FIFO Register Space				
Command Registers				
0x0800	CmdReboot	WO	N/A	Any write reboots card. If & CfgSupported == 0& in current configuration, result is undefined
0x0804	CmdPrimitive	WO	Enum	0 -> None, 4 -> Triangle, 5 -> Triangle Strip, 6 -> Triangle Fan, 8 -> Quads, 9 -> Quad Strip
0x0808	CmdVertex	WO	N/A	Any write emits a vertex with state of current state registers
0x080C	CmdSync	WO	N/A	Any write will suspend FIFO processing until the next vertical sync period
0x0814	CmdActiveBuffer	WO	Bitfield	Bit 0: Active Screen Buffer, Bit 1: Active Draw Buffer
0x0818	CmdClear	WO	Bitfield	Bit 0: Clear color buffer to current VtxColor, Bit 1: Clear depth buffer
0x0820	CmdDMABuffer	WO	Address	Address from which to perform DMA operation. The lower 12 bits are assumed to be 0.
0x0824	CmdDMACount	WO	Bitfield	Writes to this initiate a DMA transfer. Bit 0: Type (Must be 0 for command buffers), Bits 1-16: Number of bytes to transfer
State Registers				
0x0900	VtxPosition	WO	Float4	XYZW of current vertex
0x0910	VtxColor	WO	Float4	RGBA of current vertex
0x0930	VtxTexCoord	WO	Float2	Pixel-Space UV coordinates
0x0A00	VtxTransform	WO	Float16	Column-Major transformation matrix
0x0F00	InfFIFO	RO	Integer	Current depth of the FIFO queue. The FIFO queue is guaranteed to be of length at least 32, but implementations may differ.

Notes: All writes to the configuration registers happen instantaneously, and the results may be read back without delay. Writes to any region in the FIFO space will be placed on the card's internal queue for processing. Care must be taken to ensure that the driver does not cause an overflow in the queue. The InfFIFO register may be used to synchronize access. It is expected that drivers will use the FIFO only for debugging and initialization. Most communication will occur indirectly via DMA buffers. Under DMA, the formatting of commands is as follows: All values are stored in 4 bytes, and they are read in sequential order from the buffer. For each command, the address of the command to execute is first, followed by any arguments. The same applies to the state registers, but these may only be accessed from their base register addresses. For example, to set the current vertex color to red, the value 0x910 in little endian would be written, followed by the proper RGBA values (1.0,0.0,0.0,0.0) in the binary32 IEEE-754 (float) format. It is not possible to issue changes to only a particular channel of a state register when issuing commands through DMA. When a DMA transfer is initiated by placing its physical address and buffer content size onto the FIFO queue, the card will begin execution of its commands as quickly as possible. Once the card has completed executing all commands from the buffer, or an error condition has occurred, an interrupt will be sent to the system via the PCI-Express' MSI system. The card will not continue execution until the interrupt (or error) has been acknowledged.

Table 1: The Current *Kyouko* Hardware Reference Manual