I/O Paravirtualization at the Device File Boundary

Ardalan Amiri Sani   Kevin Boos   Shaopu Qin   Lin Zhong
Rice University

Abstract

Paravirtualization is an important I/O virtualization technology since it uniquely provides all of the following benefits: the ability to share the device between multiple VMs, support for legacy devices without virtualization hardware, and high performance. However, existing paravirtualization solutions have one main limitation: they only support one I/O device class, and would require significant engineering effort to support new device classes and features. In this paper, we present Paradice, a solution that vastly simplifies I/O paravirtualization by using a common paravirtualization boundary for various I/O device classes: Unix device files. Using this boundary, the paravirtual drivers simply act as a class-agnostic indirection layer between the application and the actual device driver.

We address two fundamental challenges: supporting cross-VM driver memory operations without changes to applications or device drivers and providing fault and device data isolation between guest VMs despite device driver bugs. We implement Paradice for x86, the Xen hypervisor, and the Linux and FreeBSD OSes. Our implementation paravirtualizes various GPUs, input devices, cameras, an audio device, and an Ethernet card for the netmap framework with ~7700 LoC, of which only ~900 are device class-specific. Our measurements show that Paradice achieves performance close to native for different devices and applications including netmap, 3D HD games, and OpenCL applications.

Categories and Subject Descriptors C.0 [*Computer Systems Organization*]: General—System architectures; D.4.6 [*Operating Systems*]: Security and Protection; D.4.8 [*Operating Systems*]: Performance; I.3.4 [*Computer Graphics*]: Graphics Utilities—Virtual device interfaces

Keywords I/O; Virtualization; Paravirtualization; Isolation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '14, March 1–5, 2014, Salt Lake City, Utah, USA.
Copyright © 2014 ACM 978-1-4503-2305-5/14/03. . . $15.00.
http://dx.doi.org/10.1145/2541940.2541943

1. Introduction

Virtualization has become an important technology for computers of various form factors, from servers to mobile devices, because it enables hardware consolidation and secure co-existence of multiple operating systems in one physical machine. I/O virtualization enables virtual machines (VMs) to use I/O devices in the physical machine, and is increasingly important because modern computers have embraced a diverse set of I/O devices, including GPU, DSP, sensors, GPS, touchscreen, camera, video encoders and decoders, and face detection accelerator [4].

Paravirtualization is a popular I/O virtualization technology because it uniquely provides three important benefits: the ability to share the device between multiple VMs, support for legacy devices without virtualization hardware, and high performance. Existing paravirtualization solutions have one main drawback: they only support one I/O device class, and would require significant engineering effort to support new device classes and features. As a result of the required engineering effort, only network and block devices have enjoyed good paravirtualization solutions so far [28, 31, 33, 44]. Limited attempts have been made to paravirtualize other device classes, such as GPUs [27], but they provide low performance and do not support new class features, such as GPGPU. Significant engineering effort is an increasingly important problem to paravirtualization in light of the ever-increasing diversity in I/O devices.

We present Paradice (*Para*virtual *De*vice), a solution that greatly simplifies I/O paravirtualization by using a common, class-agnostic I/O paravirtualization boundary, *device files*. Unix-like OSes employ device files to abstract most I/O devices [9], including GPU, camera, input devices, and audio devices. To leverage this boundary, Paradice creates a virtual device file in the guest OS corresponding to the device file of the I/O device. Guest applications issue file operations to this virtual device file as if it were the real one. The paravirtual drivers act as a thin indirection layer and forward the file operations to be executed by the actual device driver. Since device files are common to many I/O devices, Paradice reduces the engineering effort to support various I/O devices. Moreover, as we will demonstrate, Paradice maintains the three aforementioned advantages of I/O paravirtualization.

We address two fundamental challenges: (*i*) the guest process and the device drivers reside in different virtualization domains, creating a memory barrier for executing the driver memory operations on the guest process. Our solution to this problem is to redirect and execute the memory operations efficiently in the hypervisor without any changes to the device drivers or applications. (*ii*) malicious applications can exploit device driver bugs [25, 29] through the device file interface to compromise the machine that hosts the device driver [1, 2]. Our solution to this problem is a suite of techniques that provide fault and device data isolation between the VMs that share a device. Paradice guarantees fault isolation even with unmodified device drivers. However, it requires modest changes to the device driver for device data isolation (i.e., ~400 LoC for the Linux Radeon GPU driver).

We present a prototype implementation of Paradice for the x86 architecture, the Xen hypervisor, and the Linux and FreeBSD OSes. Our prototype currently supports five important classes of I/O devices with about 7700 LoC, of which about 900 are specific to device classes: GPU, input device, camera, audio device, and Ethernet for the netmap framework [43]. Approximately 400 lines of this class-specific code are for device data isolation for GPU. We note that GPU has not previously been amenable to virtualization due to its functional and implementation complexity. Yet, Paradice easily virtualizes GPUs of various makes and models with full functionality and close-to-native performance.

Paradice supports cross-OS I/O paravirtualization. For example, our current implementation virtualizes I/O devices for a FreeBSD guest VM using Linux device drivers. Hence, Paradice is useful for driver reuse between these OSes too, for example, to reuse Linux GPU drivers on FreeBSD, which typically does not support the latest GPU drivers.

We report a comprehensive evaluation of Paradice and show that it: (*i*) requires low development effort to support various I/O devices, shares the device between multiple guest VMs, and supports legacy devices; (*ii*) achieves close-to-native performance for various devices and applications, both for Linux and FreeBSD VMs; and (*iii*) provides fault and device data isolation without incurring noticeable performance degradation.

2. Background

2.1 I/O Stack

Figure 1(a) shows a simplified I/O stack in a Unix-like OS. The kernel exports device files to the user space through a special filesystem, `devfs` (the `/dev` directory). A process thread issues file operations by calling the right system calls on the device file. These system calls are handled by the kernel, which invokes the *file operation handlers* implemented by the device driver. The commonly used file operations are `read`, `write`, `poll`, `ioctl`, and `mmap`.

When servicing a file operation, the device driver often needs to perform memory operations on the process memory. There are two types of memory operations: copying a kernel buffer to/from the process memory, which are mainly used by the `read`, `write`, and `ioctl` file operations, and mapping a system or device memory page into the process address space, which is mainly used by the `mmap` file operation and its supporting `page_fault` handler.

Instead of using the `poll` file operation, a process can request to be notified when events happen, e.g., where there is a mouse movement. Linux employs the `fasync` file operation for setting up the asynchronous notification. When there is an event, the process is notified with a signal.

To correctly access an I/O device, an application may need to know the exact make, model or functional capabilities of the device. For example, the X Server needs to know the GPU make in order to load the correct libraries. As such, the kernel collects this information and exports it to the user space, e.g., through the `/sys` directory in Linux, and through the `/dev/pci` file in FreeBSD.

The I/O stack explained here is used for most I/O devices in Unix-like OSes. Famous exceptions are network and block devices, which have their own class-specific I/O stacks. See §8 for more details.

2.2 I/O Paravirtualization

An I/O paravirtualization solution uses a pair of drivers: a frontend driver in the guest VM and a backend driver in the same domain as the main device driver, e.g., the Xen driver domain [28]. The frontend and backend drivers cooperate to enable guest VM applications to use the I/O device. For example, the paravirtual network drivers exchanges packets that the applications need to send or receive [28, 44]. We refer to the boundary in the I/O stack that the paravirtual drivers are located at as the *paravirtualization boundary*.

2.3 Memory Virtualization

The hypervisor virtualizes the physical memory for the VMs. While memory virtualization was traditionally implemented entirely in software, i.e., the shadow page table technique [35], recent generations of micro-architecture provide hardware support for memory virtualization, e.g., the Intel Extended Page Table (EPT). In this case, the hardware Memory Management Unit (MMU) performs two levels of address translation from guest virtual addresses to guest physical addresses and then to system physical addresses. The guest page tables store the first translation and are maintained by the guest OS. The EPTs store the second translation and are maintained by the hypervisor.

3. Paradice Design

3.1 Architectural Design Choices

Design choice 1: device files as the paravirtualization boundary. Our key observation in this work is that Unix-like OSes abstract most I/O devices with device files, which can be used as a common paravirtulization boundary. With this

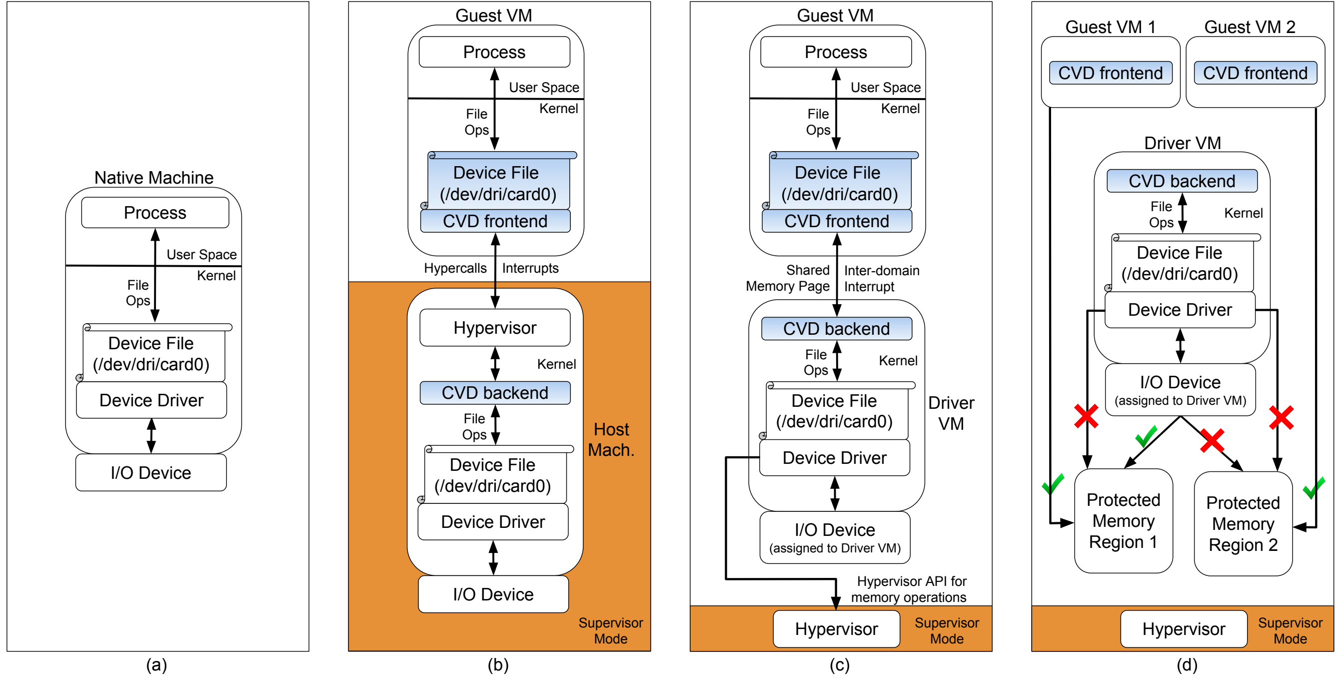

Figure 1. (a) The simplified I/O stack in a Unix-like OS. (b) Devirtualization's design, our previous work that also paravirtualizes I/O devices at the device file boundary. This design can be abused by a malicious guest VM to compromise the host machine, and hence the hypervisor and other guest VMs. (c) Paradice's design, which sandboxes the device and its driver in a driver VM and performs strict runtime checks on requested memory operations for fault isolation. (d) Device data isolation, enforced by the hypervisor, which creates non-overlapping protected memory regions for each guest VM's data and enforces appropriate access permissions to these regions. Each protected region includes part of driver VM system memory and device memory.

boundary, the paravirtual drivers simply act as an indirection layer between the application and the actual device driver.

In order to paravirtualize I/O devices at the device file boundary, we create a virtual device file inside the guest VM that mirrors the actual device file. Applications in the guest VM issue file operations to this virtual device file as if it were the real one. The paravirtual drivers, i.e., the *Common Virtual Driver* (*CVD*) frontend and backend, deliver these operations to the actual device file to be executed by the device driver.

Figure 1(b) shows a simple design for using the device file boundary for paravirtualization with a Type II (i.e., hosted) hypervisor. In this design, the device driver, device file, and also the hypervisor reside in supervisor mode. The CVD frontend and backend are in the guest VM and in the host OS, respectively. We used this design in our previous work, called devirtualization [20]. However, devirtualization's design does not provide isolation between guest VMs. As device drivers run in the host OS, which executes in supervisor mode, a malicious guest VM application can use the driver bugs to compromise (i.e., crash, gain root access in, or execute malicious code in) the whole system including the hypervisor and other guest VMs. This important flaw led us to the design of Paradice, described below.

Design choice 2: device and driver sandboxing and strict runtime checks of driver memory operations for fault isolation. Paradice solves the flaw in devirtualization's design by sandboxing the device and its driver in a separate VM, called the *driver VM*, using device assignment [19, 24, 30, 39]. This design uses a Type I (i.e., bare-metal) hypervisor, as shown in Figure 1(c). For device assignment, the hypervisor maps the device registers and memory to the driver VM and restricts the device DMA addresses to the driver VM memory using the hardware I/O Memory Management Unit (IOMMU). The CVD frontend and backend use shared memory pages and inter-VM interrupts to communicate, e.g., for forwarding the file operations.

Executing driver memory operations (§2.1) in Paradice introduces new challenges since the device driver and the guest process reside in different VMs with isolated memory. As a solution, we implement the two types of memory operations efficiently in the hypervisor and provide an API for the driver VM to request them. Further, to support unmodified drivers, we provide wrapper stubs in the driver VM kernel that intercept the driver's kernel function invocations for memory operations and redirect them to the hypervisor through the aforementioned API.

In Paradice's design, a malicious guest VM can still compromise the driver VM through the device file interface. Therefore, we perform strict runtime checks on the memory operations requested by the driver VM in order to guarantee that the compromised driver VM cannot be abused to pollute other guest VMs. For the checks, the CVD frontend in the guest VM identifies and declares the legitimate memory operations to the hypervisor before forwarding a file operation to the backend. §4.1 explains this issue in more detail.

Design choice 3: hypervisor-enforced access permissions for device data isolation. Applications exchange data with I/O devices. Device data isolation requires such data to be accessible only to the guest VM that owns the data but not to any other guest VMs. We enforce device data isolation in the hypervisor by allocating non-overlapping protected memory regions on the driver VM memory and on the device memory for each guest VM's data and assigning appropriate access permissions to these regions (Figure 1(d)). §4.2 elaborates on this technique.

Unmodified device drivers cannot normally function correctly in the presence of the hypervisor-enforced device data isolation. In §5.3, we explain how we added only ~400 LoC to the complex Linux Radeon GPU driver for this purpose. Unlike all other Paradice components, device data isolation is not generic. However, many of the techniques we developed for the Radeon GPU driver apply to other device drivers as well.

3.2 Key Benefits of Paradice Design

3.2.1 One Paravirtual Driver, Many I/O Devices

The key benefit of Paradice is that it requires a single pair of paravirtual drivers, i.e., the CVD frontend and backend, and very small class-specific code to support many different device classes. In contrast, prior solutions employ class-specific paravirtual drivers. Moreover, Paradice supports all features of an I/O device class since it simply adds an indirection layer between applications and device drivers. In contrast, prior solutions only support a limited set of features and require more engineering effort to support new ones.

3.2.2 Compatibility between Different OSes

The device file interface is compatible across various Unix-like OSes; therefore Paradice can support guest VMs running different versions of Unix-like OSes in one physical machine, all sharing the same driver VM. We investigated the file operations interface in FreeBSD and many versions of Linux and observed the following: (*i*) the file operations that are mainly used by device drivers (§2.1) exist in both Linux and FreeBSD and have similar semantics; (*ii*) these file operations have been part of Linux since the early days and have seen almost no changes in the past couple of years, i.e., from Linux 2.6.35 (2010) to 3.2.0 (2012). §5.1 discusses our deployment of a Linux driver VM, a FreeBSD guest VM, and a Linux guest VM running a different major version of Linux. In order to use a device driver, applications might require appropriate libraries, e.g., the Direct Rendering Manager (DRM) libraries for graphics. These libraries are usually available for different Unix-like OSes. If not, porting the library from another OS is possible since Unix-like OSes are mostly source code compatible.

3.2.3 Concurrent Device Access by Multiple Guests

If supported by the device driver, Paradice allows for multiple guest VMs to concurrently use the device because the device file interface allows multiple processes to issue file operations simultaneously. In this case, the same CVD backend supports requests from CVD frontends of all guest VMs. Some device drivers, e.g., the Linux GPU drivers, can handle concurrency, but some others, e.g., the Linux camera drivers, only allow one process at a time. §5.1 discusses the issue of concurrency for different classes of devices.

4. Isolation between Guest VMs

Device drivers are buggy [25, 29], and these bugs can be used by malicious applications to compromise the machine, either virtual or physical, that hosts the driver [1, 2]. Compromising a machine refers to crashing, gaining root access in, or executing malicious code inside the machine. This section elaborates on how we provide fault and device data isolation between guest VMs despite such driver bugs.

4.1 Fault Isolation

Fault isolation requires that a malicious guest VM cannot compromise other guest VMs. To provide fault isolation, we must prevent guest VMs from access to unauthorized and sensitive parts of the system memory, e.g., the hypervisor's memory or other guest VMs' memory. We employ two techniques to achieve this. First, to protect the hypervisor, we sandbox the device driver and the device inside the driver VM, resulting in the design of Paradice as was explained in §3.1. With this design, a malicious guest VM can compromise the driver VM, but not the hypervisor. Therefore, in the rest of the discussion, we assume that the driver VM is controlled by a malicious guest VM and cannot be trusted. This leads us to the second technique. To protect other guest VMs, we employ strict runtime checks in the hypervisor to validate the memory operations requested by the driver VM, making sure that they cannot be abused by the compromised driver VM to compromise other guest VMs, e.g., by asking the hypervisor to copy data to some sensitive memory location inside a guest VM kernel.

For validation, the CVD frontend identifies the legitimate memory operations of each file operation and declares them to the hypervisor using a grant table before forwarding the file operation to the backend. The legitimate memory operations can be easily identified in the CVD frontend by using the file operation's input arguments. For example, the `read` file operation requires the driver to copy some data to the

process memory, and the `read` input arguments include the start address and size of the user space buffer that the driver needs to write to. However, the input arguments are not always enough for `ioctl`.

To identify the memory operations of an `ioctl`, we use two techniques. First, we use the `ioctl` input arguments, if possible. The arguments of an `ioctl` include a command number and an untyped pointer. The device driver may execute different memory operations based on the value of these two arguments. The most common `ioctl` memory operations are to copy a command-specific data structure from the process memory to the kernel memory and vice-versa. Fortunately, device drivers often use OS-provided macros to generate `ioctl` command numbers, which embed the size of these data structures and the direction of the copy into it. Moreover, the `ioctl` untyped pointer holds the address of this data structure in the process memory. Therefore in these cases, the CVD frontend simply parses the command number and uses it along with the value of the untyped pointer to determine the arguments of the memory operations. We have successfully tested this approach with the UVC camera driver and the majority of `ioctl` commands in Radeon GPU driver.

However, this approach is not applicable if the driver performs memory operations other than simple copying of a data structure, or if the driver does not use the OS-provided macros. For example, for some Radeon driver `ioctl` commands, the driver performs nested copies, in which the data from one copy operation is used as the input arguments for the next one. Therefore, we provide a second solution that can identify such memory operations. For this solution, we develop a static analysis tool that analyzes the unmodified driver and extracts a simplified part of its `ioctl` handler. This code extract has no external dependencies and can even be executed without the presence of the actual device. We then execute this code offline and generate the arguments of legitimate memory operations in the form of static entries in a source file that is included in the CVD frontend. Given an `ioctl` command, the CVD frontend can look up these entries to find the legitimate operations. However, offline execution is impossible for some memory operations, such as the nested copies mentioned above. In this case, the CVD frontend identifies the memory operation arguments just-in-time by executing the extracted code at runtime. Our tool correctly analyzes the Radeon GPU driver. It detects instances of nested copies in 14 `ioctl` commands in the Radeon driver. For these commands, it automatically generates about 760 lines of extracted code from the driver that are added to the source file used by the CVD frontend.

The memory operations executed by the driver for each `ioctl` command rarely change across driver updates because any such changes can break application compatibility. Therefore, the existing code and entries in the source file generated by our tool do not need to be updated with every driver update. However, newer `ioctl` commands might be added to a driver, which then need to be analyzed by our tool. Our investigation of Radeon drivers of Linux kernel 2.6.35 and 3.2.0 confirms this arguments as the memory operations of common `ioctl` commands are identical in both drivers, while the latter has four new `ioctl` commands.

4.2 Device Data Isolation

Guest VM processes exchange data with the I/O device. Device data isolation requires such data to be isolated and not accessible to other guest VMs. Processes' data may either be on the device memory, or on the driver VM system memory, which can be accessed by the device through DMA. Isolating these data is challenging since the driver VM is compromised, and hence the malicious VM has full access to driver VM system memory and to the device memory. We enforce device data isolation in the hypervisor by protecting the device memory and part of the driver VM system memory from the driver VM for hosting the guest VMs' data. We then split the protected memory into non-overlapping memory regions for each guest VM's data, and then assign appropriate access permissions to these regions, as illustrated in Figure 1(d). The CPU code in the driver VM, including the device driver, does not have permission to read these memory regions. Each guest VM has access to its own memory region only (through the memory operations executed by the hypervisor), and the device has access permission to one memory region at a time.

This set of access permissions prevents a malicious guest VM from stealing the data of another guest VM because it stops the following attacks: first, the malicious VM cannot use the hypervisor API to access the data buffers allocated for other VMs because the hypervisor prohibits that. Second, the malicious VM cannot use the compromised driver VM to read the data buffer, because the driver VM does not have read permission to the memory regions. Finally, the malicious VM cannot program the device to copy the buffer outside a memory region because the device can only access one memory regions at a time.

When adding device data isolation support to a driver, we need to determine which guest VM's data buffers are sensitive and need to be protected. For example for GPU, we protected all the raw data that guest VMs share with the GPU including the graphics textures and GPGPU input data. Some data that are moved from the guest applications to driver VM are not sensitive and do not need to be protected. For example, most `ioctl` data structures are mainly instructions for the driver and do not contain raw application data. After determining the sensitive data, we use the hypervisor to protect them. It is then important to make sure that the driver does not try to read the data, otherwise the driver VM crashes. Indeed, we observed that all the sensitive data that we determined for the GPU were never read by the driver. They were only accessed by the device itself or by the process, making it easy to add device data isolation. In case a driver

normally reads the sensitive data, it needs to be modified to avoid that access in order to be compatible with device data isolation in Paradice. Finally, we need a policy to tell the hypervisor which data buffers to protect. We observed that most applications typically use `mmap` to move sensitive data buffers to/from the device mainly because it provides better performance compared to copying. For example with the Radeon driver, applications only use `mmap` to move graphics textures and GPGPU input data to the device. As a result, we currently use a simple policy in the hypervisor to enforce isolation for mapped buffers only. More sophisticated policies that enforce isolation for other sensitive data, that are for example copied from the guest VM to the driver VM, are also possible.

Next, we explain how the hypervisor enforces the access permissions. Access permissions for the driver VM are enforced by removing the EPT read permission for the protected memory regions. This technique is adequate since both the system and the device memory need to be mapped by EPTs in order to be accessible to the driver VM.

System memory access permissions for the device are enforced using the IOMMU. Normally with device assignment, the hypervisor programs the IOMMU to allow the device to DMA to all physical addresses in the driver VM. For device data isolation, the hypervisor does not initially create any mappings in the IOMMU. The IOMMU mappings are added per request from the device driver. When asking to add a page to the IOMMU, the driver needs to attach the corresponding memory region ID. The hypervisor maintains a list of which pages have been mapped in IOMMU for each region. Whenever the device needs to work with the data of one guest VM, the driver asks the hypervisor to switch to the corresponding memory region. When switching the region, the hypervisor unmaps all the pages of the previous region from the IOMMU and maps the pages of the new region.

Enforcing device memory access permissions for the device requires device-specific solutions. For the Radeon Evergreen series (including the HD 6450), we leveraged the GPU memory controller, which has two registers that set the lower and upper bounds of device memory accessible to the GPU cores. The hypervisor takes full control of the GPU memory controller registers and does not map them into the driver VM, so that it can enforce the accessible device memory for the GPU at any time. If the GPU tries to access memory outside these bounds, it will not succeed. Note that this solution partitions and shares the GPU memory between guest VMs and can affect the performance of guest applications that require more memory than their share.

Class	Class-spec. code (LoC)	Device Name	Driver Name
GPU	92	Disc. ATI Radeon HD 6450	DRM/Radeon
		Disc. ATI Radeon HD 4650	DRM/Radeon
		Int. ATI Mobility Radeon X1300(*)	DRM/Radeon
		Int. Intel Mobile GM965/GL960(*)	DRM/i915
Input	58	Dell USB Mouse	evdev/usbmouse
		Dell USB Keyboard	evdev/usbkbd
Camera	43	Logitech HD Pro Webcam C920	V4L2/UVC
		Logitech QuickCam Pro 9000	V4L2/UVC
Audio	37	Intel Panther Point HD Audio Cont.	PCM/snd-hda-intel
Ethernet	21	Intel Gigabit Adapter	netmap/e1000e

Table 1. I/O devices paravirtualized by our Paradice prototype with very small class-specific code. For correct comparison, we do not include the code for device data isolation and graphics sharing. Those can be found in Table 2. (*) marks the devices that we have tested only with our previous system design, devirtualization (§3.1). We include them here to show that the device file boundary is applicable to various device makes and models.

5. Implementation

We implement Paradice for the Xen hypervisor, 32-bit x86 architecture with Physical Address Extension (PAE), and Linux and FreeBSD OSes. The implementation is modular and can be revised to support other hypervisors, architectures, and Unix-like OSes. It virtualizes various GPUs, input devices (such as keyboard and mouse), cameras, a speaker, and an Ethernet card for netmap, all with about 7700 LoC, of which only about 900 LoC are specific to device classes. Moreover, around 400 lines of this class-specific code are for device data isolation for GPU. Table 1 shows the list of devices that we have paravirtualized with Paradice. Table 2 breaks down the Paradice code structure. We use CLOC [3] (v1.56) for counting code. We do not count comments or debugging code.

5.1 Paradice Architecture Details

Common Virtual Driver (CVD): The CVD frontend and backend constitute a large portion of the implementation consist of two parts each. The first part implements the interface to the hypervisor, e.g., invoking the hypervisor API in the backend, or sharing a grant table with the hypervisor in the frontend. The second part interacts with the OS kernel only, e.g., by invoking the device driver's file operation handlers in the backend, or by handling (and forwarding) the file operations in the frontend.

The CVD frontend and backend use shared memory pages and inter-VM interrupts to communicate. The frontend puts the file operation arguments in a shared page, and uses an interrupt to inform the backend to read them. The backend communicates the return values of the file operation in a similar way. Because interrupts have noticeable latency (§6.1.1), CVD supports a polling mode for high-performance applications such as netmap. In this mode, the frontend and backend both poll the shared page for $200\mu s$ before they go to sleep to wait for interrupts. The polling period is chosen empirically and is not currently optimized. Moreover, for asynchronous notifications (§2.1), the CVD backend uses similar techniques to send a message to the frontend, e.g., when the keyboard is pressed.

Type	Total LoC	Platform	Component	LoC
Generic	6833	Linux	CVD:	
			- frontend	1553
			- backend	1950
			- shared	378
			Linux kernel wrapper stubs	198
			Virtual PCI module	285
			- Supporting kernel code	50
		FreeBSD	FreeBSD CVD frontend:	
			- New code (approx.)	451
			- From Linux CVD (approx.) (not calculated in the total)	758
			- Supporting kernel code	15
			Virtual PCI module	74
			- Supporting kernel code	29
		Xen	Paradice API	1349
		Clang	Driver `ioctl` analyzer	501
Class-specific	825	Linux	Device info modules:	
			- GPU	92
			- input device	58
			- camera	43
			- audio device	37
			- Ethernet (netmap)	21
			Graphics sharing code	145
			- Supporting DRM driver code	15
			Data isol. for Radeon driver	382
		FreeBSD	Device info modules:	
			- Ethernet (netmap)	32

Table 2. Paradice code breakdown.

The CVD frontend uses a grant table to declare the legitimate memory operations to the hypervisor (§4.1). The grant table is a single memory page shared between the frontend VM and the hypervisor. After storing the operations in the table, the CVD frontend generates a grant reference number and forwards it to the backend along with the file operation. The backend then needs to attach the reference to every request for the memory operations of that file operation. The reference number acts as an index and helps the hypervisor validate the operation with minimal overhead.

The CVD backend puts new file operations on a wait-queue to be executed. We use separate wait-queues for each guest VM. We also set the maximum number of queued operations for each wait-queue to 100 to prevent malicious guest VMs from causing denial-of-service problems by issuing too many file operations. We can modify this cap for different queues for better load balancing or enforcing priorities between guest VMs.

Device Info Modules: As mentioned in §2.1, applications may need some information about the device before they can use it. In Paradice, we extract device information and export it to the guest VM by providing a small kernel module for the guest OS to load. Developing these modules is easy because they are small, simple, and not performance-sensitive. For example, the device info module for GPU has about 100 LoC, and mainly provides the device PCI configuration information, such as the manufacturer and device ID. We also developed modules to create or reuse a virtual PCI bus in the guest for Paradice devices.

Device File Interface Compatibility: Paradice supports guest VMs using different versions of Unix-like OSes in one physical machine. For example, we have successfully deployed Paradice with a Linux driver VM, a FreeBSD guest VM and a Linux guest VM running a different major version of Linux (versions 2.6.35 and 3.2.0). To support a different version of Linux, we added only 14 LoC to the CVD to update the list of all possible file operations based on the new kernel (although none of the new file operations are used by the device drivers we tested). To support FreeBSD, we redeveloped the CVD frontend with about 450 new LoC and about 760 LoC from the Linux CVD frontend implementation. To support `mmap` and its `page_fault` handler, we added about 12 LoC to the FreeBSD kernel to pass the virtual address range to the CVD frontend, since these addresses are needed by the Linux device driver and by the Paradice hypervisor API.

Concurrency Support: As mentioned in §3.2.3, the device file interface allows for concurrent access from multiple processes if the driver supports it. We define the policies for how each device is shared. For GPU for graphics, we adopt a foreground-background model. That is, only the foreground guest VM renders to the GPU, while others pause. We assign each guest VM to one of the virtual terminals of the driver VM, and the user can easily navigate between them using simple key combinations. For input devices, we only send notifications to the foreground guest VM. For GPU for computation (GPGPU), we allow concurrent access from multiple guest VMs. For camera and Ethernet card for netmap, we only allow access from one guest VM at a time because their drivers do not support concurrent access. Note that Paradice will automatically support sharing of these devices too if concurrency support is added to their drivers, as is the plan for netmap (see [43]).

5.2 Hypervisor-Assisted Memory Operations

Hypervisor supports the two types of memory operations needed by the device drivers. For copying to/from the process memory, the hypervisor first translates the virtual addresses of the source and destination buffers (which belong to different VMs) into system physical addresses and then performs the copy. If the source or destination buffers span more than one page, the address translation needs to be performed per page since contiguous pages in the VM address spaces are not necessarily contiguous in the system physical address space. To perform the translation, the hypervisor first translates the VM virtual address to the VM physical address by walking the VM's own page tables in software. It then translates the VM physical address to the system physical address by walking the EPTs.

For mapping a page into the process address space, the hypervisor fixes the EPTs to map the page to an (arbitrary) physical page in the guest physical address space, and then fixes the guest page tables to map the guest physical page to the requested virtual address in the guest process address space. We can use any guest physical page address in the mappings as long as it is not used by the guest OS. The hypervisor finds unused page addresses in the guest and uses them for this purpose. Moreover, before forwarding the `mmap`

operation to the backend, the CVD frontend checks the guest page tables for the mapping address range, and creates all missing levels except for the last one, which is later fixed by the hypervisor. This approach provides better compatibility with the guest kernel than fixing all the levels of the guest page tables in the hypervisor. Upon unmapping a previously mapped page, the hypervisor only needs to destroy the mappings in the EPTs since the guest kernel destroys the mappings in its own page tables before informing the device driver of the unmap.

As mentioned in §3.1, we employ wrapper stubs in the driver VM kernel to support unmodified device drivers. For this purpose, we modified 13 Linux kernel functions, e.g., the `insert_pfn` function, which maps a page to a process address space. When the CVD backend invokes a thread to execute the file operation of a guest VM, it marks the thread by setting a flag in the thread-specific `task_struct` data structure. The wrapper stubs will invoke the appropriate hypervisor API when executed in the context of a marked thread.

5.3 Isolation between Guest VMs

Fault Isolation: As described in §4.1, we have developed a static analysis tool to identify the legitimate memory operations of driver `ioctl` commands. We implemented this tool as a standalone C++ application built upon the LLVM compiler infrastructure [37] and its C language frontend, Clang [12]. Using Clang, our tool parses the driver source code into an Abstract Syntax Tree (AST) and then analyzes the AST to extract memory operations of interest. Our Clang tool uses classic program slicing techniques [51] to shrink the source code by selecting the subset of functions and statements that affect the input arguments of a given memory operation. This analysis is almost fully automated, requiring manual annotation only when human knowledge of the code is necessary to resolve function pointers or other external dependencies.

Device Data Isolation: As mentioned earlier, we added ~400 LoC to the Radeon driver to support the device data isolation enforced by the hypervisor. Currently, our changes only support the Radeon Evergreen series (including the Radeon HD 6450 in our setup), but the code can be easily refactored to support other Radeon series as well. Moreover, our current implementation has minimal support for switching between memory regions; improving the switching is part of our future work.

We made four sets of changes to the driver. (*i*) In the driver, we explicitly ask the hypervisor to map pages in or unmap pages from IOMMU for different memory regions. For better efficiency, we allocate a pool of pages for each memory region and map them in IOMMU in the initialization phase. The hypervisor zeros out the pages before unmapping. (*ii*) The driver normally creates some data buffers on the device memory that are used by the GPU, such as the GPU address translations buffer. We create these buffers on all memory regions so that the GPU has access to them regardless of the active memory region. (*iii*) We unmap from the driver VM the MMIO page that contains the GPU memory controller registers used for determining the device memory accessible by the GPU (§4.2). If the driver needs to read/write to other registers in the same MMIO page, it issues a hypercall. (*iv*) While removing the read permissions from EPTs of protected memory regions are enough for device data isolation, we had to remove both read and write permissions since x86 does not support write-only permissions. In rare cases, the driver needs write permissions to some memory buffers such as the GPU address translation buffer. If the buffer is on the system memory, we emulate write-only permissions by making the buffer read-only to the device through the IOMMU and giving read/write permissions to the driver VM. If the buffer is on the device memory, we require the driver VM to issue a hypercall.

For device data isolation, we need to make sure that the driver VM cannot access the GPU memory content through any other channels. For this purpose, we studies the register programming interface of the Evergreen series, and confirmed that the driver cannot program the device to copy the content of memory buffers to registers that are readable by the device driver.

We faced one problem with interrupts. Some Radeon series, including the Evergreen series, use system memory instead of registers to convey the interrupt reason from the device to the driver. That is, the device writes the reason for the interrupt to this pre-allocated system buffer and then interrupts the driver. However, to enforce device data isolation, we cannot give the driver read permission to any system memory buffer that can be written by the device. Therefore, we currently disable all interrupts, except for the fence interrupt needed to monitor the GPU execution, and then interpret all interrupts as fences. The main drawback of this approach is that we cannot support the VSync interrupts, which can be used to cap the GPU graphics performance to a fixed number of frames per second. As a possible solution, we are thinking of emulating the VSync interrupts in software. We do not expect high overhead since VSync happens relatively rarely, e.g., every 16ms for rendering 60 frames per second.

As a guideline for our implementation, we did not add device-specific code to the hypervisor and implemented the hypervisor functions in the form of a generic API. In the few cases that device-specific information was needed, we leveraged the *driver initialization phase* to call generic hypervisor API to achieve the required function, such as unmapping the memory controller MMIO page from the driver VM. Since no guest VM can communicate with the driver before the initialization phase is over, we assume that the driver is not malicious in this phase. We believe this approach is superior to implementing such device-specific functions in the hypervisor because it minimizes the new code in the hypervisor and improves the system reliability as a whole.

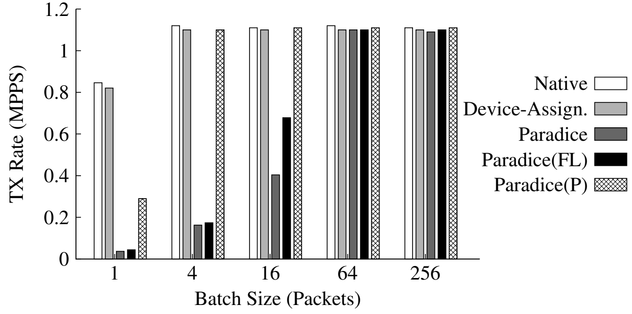

Figure 2. netmap transmit rate with 64 byte packets. (FL) indicates FreeBSD guest VM using a Linux driver VM. (P) indicates the use of polling mode in Paradice.

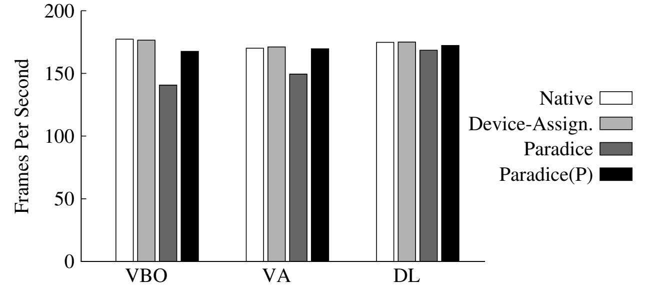

Figure 3. OpenGL benchmarks FPS. VBO, VA, and DL stand for Vertex Buffer Objects, Vertex Arrays, and Display Lists. (P) indicates the use of polling mode in Paradice.

6. Evaluation

Using the implementation described above, we evaluate Paradice and show that it: (*i*) requires low development effort to support various I/O devices, shares the device between multiple guest VMs, and supports legacy devices; (*ii*) achieves close-to-native performance for various devices and applications, both for Linux and FreeBSD guest VMs; and (*iii*) provides fault and device data isolation without incurring noticeable performance degradation.

Before we evaluate the performance of Paradice, we demonstrate that Paradice achieves the desired properties mentioned in §1. First, supporting new I/O devices with Paradice is easy and only requires developing small device info modules for new device classes (Table 2). These modules typically took us only a few person-hours each to implement. It took us a few person-weeks to add device data isolation support to the Radeon driver, a very complex drivers with ~111000 LoC in Linux 3.2. A big portion of this time was spent on implementing the supporting hypervisor code, therefore, we anticipate less development effort to add device data isolation to other drivers.

Second, Paradice effectively shares I/O devices between guest VMs (§3.2.3). For example in one experiment, we ran two guest VMs, one executing a 3D HD game and the other one running an OpenGL application, both sharing the GPU based on our foreground-background model (§5.1). In the next section, we show the performance of GPU when used by more than one VM for GPGPU computations.

Third, Paradice supports legacy devices. None of the devices that we have successfully virtualized so far have hardware support for virtualization.

6.1 Performance

In this section, we quantify the overhead and performance of Paradice. We compare the performance of Paradice with the native performance (when an application is executed natively on the same hardware), as well as with the performance of direct device assignment (when an application is executed in a VM with direct access to the I/O device). The performance of direct device assignment is the upper-bound on the performance of Paradice due to our use of device assignment to sandbox the device and its driver.

For our experiment setup, we use a desktop with a quad-core Intel Core i7-3770 CPU, 8GB of DDR3 memory, and an ASRock Z77 Extreme6 motherboard. For I/O devices, we use a Radeon HD 6450 GPU connected to a 24” Dell LCD for both graphics and GPGPU benchmarks, Intel Gigabit Network Adapter for netmap, Dell mouse, Logitech HD Pro Webcam C920, and the on-board Intel Panther Point HD Audio Controller for speaker. We configure the VMs with one virtual CPU and 1GB of memory. For device data isolation between two guest VMs, we split the 1GB GPU memory between two memory regions, therefore, all the benchmarks with data isolation can use a maximum of 512MB of GPU memory. As the default configuration for Paradice, we use the interrupts for communication, Linux guest VM and Linux driver VM, and do not employ device data isolation. Other configurations will be explicitly mentioned.

6.1.1 Overhead Characterization

There are two potential sources of overhead that can degrade Paradice’s performance compared to native: the added latency to forward a file operation from the application to the driver and isolation.

We first measure the added latency using a simple no-op file operation, where the CVD backend immediately returns to the frontend upon receiving the operation. The average of 1 million consecutive no-op operations shows that the added latency is around 35μs, most of which comes from two inter-VM interrupts. With the polling mode, this latency is reduced to 2μs.

The overhead of isolation comes from both fault and device data isolation. The main overhead of fault isolation is sandboxing the device and the driver. Therefore, this overhead is equal to the performance difference between native and direct device assignment. In all the benchmarks below, we report the performance of direct device assignment as well and show that it is almost identical to native; hence,

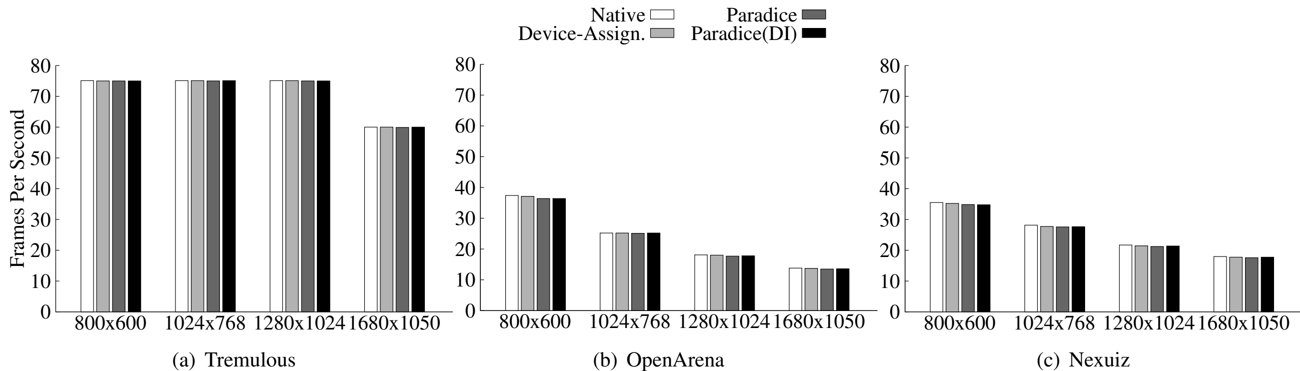

Figure 4. 3D HD games FPS at different resolutions. (DI) indicates the use device data isolation in Paradice.

fault isolation has no noticeable overhead on our benchmarks. The overhead of data isolation is mainly due to additional hypercalls issued by the driver. However, we show that this overhead does not noticeably impact the GPU's performance either, both for graphics and computation. We have not, however, quantified the impact of partitioning the GPU memory for device data isolation on applications requiring high amounts of GPU memory.

6.1.2 Ethernet Card for netmap

We evaluate the performance of netmap, a framework that can send and receive packets at the line rate on 1 and 10 gigabit Ethernet cards [43]. We use the netmap packet generator application that transmits fixed-size packets as fast as possible. In each experiment, we transmit 10 million packets and report the transmit rate. We run two sets of experiments for Paradice: one with Linux guest VM and Linux driver VM, and another with FreeBSD guest VM and Linux driver VM.

Figure 2 shows that Paradice can achieve a transmit rate close to that of native and device assignment. The figure shows the netmap transmit rate for 64-byte packets and for different packet batch sizes. The packet generator issues one `poll` file operation per batch, therefore, increasing the batch size improves the transmit rate as it amortizes the cost of the `poll` system call and, more importantly, the cost of forwarding the `poll` file operation in Paradice. When using the polling mode, a minimum batch size of 4 allows Paradice to achieve similar performance to native. However, with interrupts, a minimum batch size of around 30 is required. The figures also shows that both Linux and FreeBSD guest VMs achieve similarly high performance. Moreover, our experiments, not shown here, show that Paradice can maintain a transmit rate close to native for different packet sizes.

6.1.3 GPU for Graphics

We evaluate the performance of 3D HD games and OpenGL applications. In all evaluations, we report the standard Frames Per Second (FPS) metric. We also disable the GPU VSync feature, which would otherwise cap the GPU FPS to 60 (display refresh rate).

We use three 3D first-person shooter games: *Tremulous* [8], *OpenArena* [15], and *Nexuiz* [7], which are all widely used for GPU performance evaluation [14]. For all games, we use the Phoronix Test Suite engine [16] (v3.6.1), a famous test engine that automatically runs a demo of the game for a few minutes, while stressing the GPU as much as possible. We test the games at all possible resolutions.

We also use some OpenGL benchmarks in our experiments. The benchmarks use different OpenGL API including Vertex Buffer Objects, Vertex Arrays, and Display Lists [13, 18] to draw a full-screen teapot that consists of about 6000 polygons. In each experiment, we run the benchmark for 3 minutes and measure the average FPS.

Figures 3 and 4 show the results. There are four important observations. First, Paradice achieves close performance to native and device assignment for various benchmarks, including OpenGL benchmarks and 3D HD games. Second, Paradice (with interrupts) achieves relatively better performance for more demanding 3D games that it does for simpler OpenGL benchmarks. This is because Paradice adds a constant overhead to the file operations regardless of the benchmark load on the GPU. Therefore, for 3D games that require more GPU time to render each frame, it incurs a lower percentage drop in performance. Third, Paradice with polling can close this gap and achieve close-to-native performance for all the benchmarks. Finally, data isolation has no noticeable impact on performance.

6.1.4 GPU for Computation

We evaluate the performance of OpenCL applications. We use the Gallium Compute implementation of OpenCL [5] and use an OpenCL program that multiplies two square matrices of varying orders. We run the benchmark for different matrix orders and measure the experiment time, i.e., the time from when the OpenCL host code sets up the GPU to execute the program until when it receives the resulting matrix.

Figure 5 shows the results. It shows that Paradice performance is almost identical to native and device assignment. Moreover, the figure shows that device data isolation has no

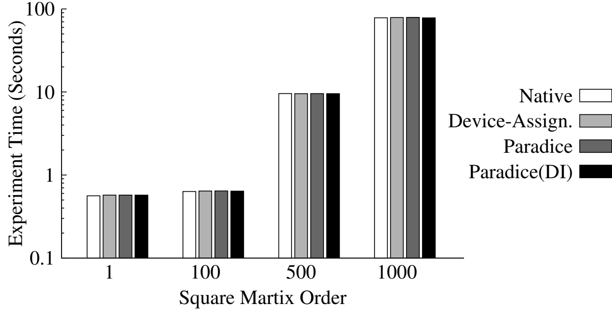

Figure 5. OpenCL matrix multiplication benchmark results. The x-axis shows the order of the input square matrices. (DI) indicates the use device data isolation in Paradice.

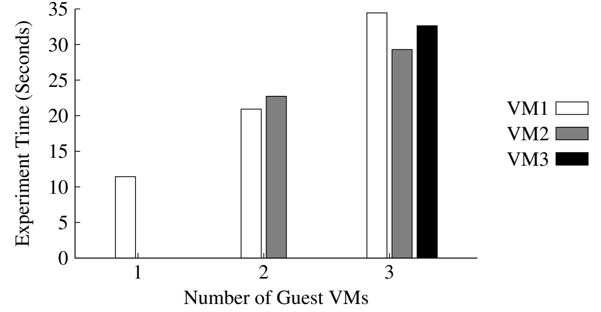

Figure 6. Guest VMs concurrently running the OpenCL matrix multiplication benchmark on a GPU shared through Paradice. The matrix order in this experiment is 500.

noticeable impact on performance. The reason for the high Paradice performance is that OpenCL benchmarks issue few file operations, and most of the experiment time is spent by the GPU itself.

We also measure the performance of the same OpenCL benchmark when executed from more than one guest VM concurrently on the same GPU. For this experiment, we use a matrix order of 500 and execute the benchmark 5 times in a row from each guest VM simultaneously and report the average experiment time for each guest VM. Figure 6 shows that the experiment time increases almost linearly with the number of guest VMs. This is because the GPU processing time is shared between the guest VMs.

6.1.5 Mouse

We measure the latency of the mouse. Our results show that native, direct assignment, Paradice using interrupts, and Paradice using polling achieve about 39μs, 55μs, 296μs, and 179μs of latency, respectively, no matter how fast the mouse moves. The extra latency of Paradice does not result in a human-perceivable difference since the latency is well below the 1ms latency required for input devices [17]. Much of the latency in Paradice comes from the communication between the CVD frontend and backend, and therefore, the polling mode reduces the latency significantly. We note that the most accurate way of measuring the latency of input devices is to measure the elapsed time between when the user interacts with the device, e.g., moves the mouse, and when this event shows up on the screen. However, such measurement is very difficult, especially for the mouse, which generates many events in a short time period. Instead, we measure the time from when the mouse event is reported to the device driver to when the `read` operation issued by the application reaches the driver.

6.1.6 Camera & Speaker

We run the GUVCview [6] camera applications in the three highest video resolutions supported by our test camera for MJPG output: 1280$\times$720, 1600$\times$896, and 1920$\times$1080. For all the resolutions, native, device assignment, and Paradice achieve about 29.5 FPS.

We play the same audio file on our test speaker. Native, device assignment, and Paradice all take the same amount of time to finish playing the file, showing that they all achieve similar audio rates.

7. Related Work

7.1 I/O Virtualization

Paradice is an I/O paravirtualization solution. Existing paravirtualization solutions [22, 27, 28, 44] only support one device class, and require significant development effort to support new device classes and features. In addition to paravirtualization, there are three main solutions for I/O virtualization in whole system virtualization. *Emulation* [23, 47] is known to have poor performance. *Direct Device Assignment* [19, 24, 30, 39] provides high performance by allowing the VM to directly own and access the physical devices; however, it can only support a single VM. Paradice, in fact, leverages device assignment in its design by assigning the I/O device to the driver VM (§3.1), but it allows multiple guest VMs to share the device, which is not possible with device assignment alone. *Self-virtualization* [26, 42, 52] requires virtualization support in the I/O device hardware, e.g., [10, 11], and therefore does not apply to legacy devices. Table 3 compares different I/O virtualization solutions.

Cells [21] employs user space virtualization and virtualizes I/O devices in the Android OS. A virtual phone in Cells has its own user space, but shares the kernel with other virtual phones hence resulting in weaker isolation compared to whole system virtualization targeted by Paradice.

Some solutions provide GPU framework virtualization by remoting OpenGL [32, 36, 46] or CUDA [45] APIs. These solutions are more limited than Paradice because they are only applicable to those particular frameworks.

7.2 Other Related Work

Paradice allows the guest to reuse the device drivers with a common indirection layer. It therefore provides a useful

	High Performance	Low Develop. Effort	Device Sharing	Legacy Device
Emulation	*No*	*No*	Yes	Yes
Direct I/O	Yes	Yes	*No*	Yes
Self Virt.	Yes	Yes	Yes *(limited)*	*No*
Paravirt.	Yes	*No*	Yes	Yes
Paradice	Yes	Yes	Yes	Yes

Table 3. Comparing I/O virtualization solutions. "Paravirt." indicates paravirtualization solutions other than Paradice.

way to leverage device drivers since driver development is complicated and bug-prone [25, 48]. There have been related efforts in reusing device drivers. LeVasseur et al. [38] execute the device driver in a separate virtual machine and allow other guests to communicate with this VM for driver support. They demonstrate support for network and block devices and use device class-specific interfaces, i.e., the translation modules, for the guest to communicate with the driver. In contrast, Paradice supports a different set of I/O devices, while building the virtualization boundary on device files, a common interface for many I/O devices, thus significantly reducing the development effort.

iKernel [50] and VirtuOS [40] run device drivers in VMs to improve system reliability. They are therefore different from Paradice that is designed for I/O paravirtualization. iKernel forwards the file operations from the host to a driver VM. It demonstrates support for a simple LED device, unlike Paradice that supports sophisticated I/O devices such as GPU. Moreover, iKernel authors do not report support for the `mmap` file operation, which is used by many device drivers.

Willman et al. [53] propose different strategies to use the IOMMU to provide protection when VMs are given direct access to devices with DMA capability. However, Paradice's use of the IOMMU to isolate the data of guest VMs shared with I/O devices is not addressed in [53].

Plan 9 distributed system [41] uses files to share resources, including I/O devices, between machines. In contrast, Paradice uses device files for I/O paravirtualization for Unix-like OSes. Moreover, Plan 9 files do not support `mmap` and `ioctl` file operations, making it difficult to support modern I/O devices such as GPUs.

8. Limitations of Current Design

Virtualization at the device file boundary is not possible if the driver does not employ the device file interface to interact with applications. Important examples include network device drivers that employ sockets and sit below the kernel network stack, block device drivers that sit below the kernel file systems, and the device drivers that are implemented completely in the user space. However, as we demonstrated with netmap, Paradice can still be useful for other frameworks using these devices.

Paradice does not currently support performance isolation between the guest VMs that share a device. This has two implications: first, Paradice does not guarantee fair and efficient scheduling of the device between guest VMs. The solution is to add better scheduling support to the device driver, such as in [34]. Second, a malicious guest VM can break the device by corrupting the device driver and writing unexpected values into the device registers. One possible solution to this problem is to detect the broken device and restart it by simply restarting the driver VM or by using techniques such as shadow drivers [49].

Paradice cannot currently guarantee the correct behavior of the I/O device. Malicious guest VMs can use carefully-designed attacks to cause incorrect performance by the device, e.g., rendering unwanted content on the screen. One possible solution is to protect certain parts of the device programming interface that allows us to achieve either *correct performance or no performance at all*. Taking GPU as an example, we can consider protecting the command streamer interface to ensure that an application's GPU commands either execute as expected or do not execute at all.

While we demonstrated Paradice for a bare-metal hypervisor, it is also applicable to hosted hypervisors as long as the driver and device can be sandboxed in a VM using the device assignment technique.

Paradice currently requires the guest VM and the I/O device to reside in the same physical machine as it uses the physical shared memory to execute the driver memory operations. We are currently working on a DSM-based solution that allows the guest and driver VM to reside in separate physical machines, which will then support the migration of the guest VM while maintaining its access to the I/O device.

Finally, Paradice uses the Unix device file as the paravirtualization boundary and hence cannot support non-Unix-like OSes, most notably, the Windows OS.

9. Conclusions

We have presented Paradice, an I/O paravirtualization solution that uses a novel boundary, device files, to support many I/O devices with low engineering effort. We are able to virtualize various GPUs, input devices, cameras, an audio device, and an Ethernet card (for netmap). Our measurements show that Paradice achieves close-to-native performance for various benchmarks such as interactive 3D HD games. We believe that Paradice opens a new door for I/O virtualization across computing platforms of various form factors.

Acknowledgments

The work was supported in part by NSF Awards #0923479, #1054693, and #1218041. The authors would like to thank Jon Howell from Microsoft Research and Dan Wallach from Rice University for their feedback on isolation. The authors would also like to thank Sreekumar Nair, then at Nokia Research, who first suggested the use of the device file boundary and contributed significantly to the devirtualization project [20].

References

[1] Privilege escalation using NVIDIA GPU driver bug. http://www.securelist.com/en/advisories/50085, .

[2] Privilege escalation using DRM/Radeon GPU driver bug. https://lkml.org/lkml/2010/1/18/106, .

[3] CLOC. http://cloc.sourceforge.net/.

[4] OMAP4 Face Detection Module, Chapter 9 of the TRM. http://focus.ti.com/pdfs/wtbu/OMAP4460_ES1.0_PUBLIC_TRM_vF.zip.

[5] GalliumCompute. http://dri.freedesktop.org/wiki/GalliumCompute/.

[6] Guvcview. http://guvcview.sourceforge.net/.

[7] Nexuiz. http://www.alientrap.org/games/nexuiz.

[8] Tremulous. http://www.tremulous.net/.

[9] Everything is a file in Unix. http://ph7spot.com/musings/in-unix-everything-is-a-file.

[10] VGX. http://www.nvidia.com/object/vgx-hypervisor.html.

[11] VMDq. http://www.intel.com/content/www/us/en/network-adapters/gigabit-network-adapters/io-acceleration-technology-vmdq.html.

[12] Clang: a C Language Family Frontend for LLVM. http://clang.llvm.org/.

[13] OpenGL Microbenchmarks: Display List. http://www.songho.ca/opengl/gl_displaylist.html.

[14] GPU Benchmarking. http://www.phoronix.com/scan.php?page=article&item=virtualbox_4_opengl&num=2.

[15] OpenArena. http://openarena.ws/smfnews.php.

[16] Phoronix Test Suite. http://www.phoronix-test-suite.com/.

[17] Touchscreen Latency. http://www.engadget.com/2012/03/10/microsoft-cuts-touchscreen-lag-to-1ms/.

[18] OpenGL Microbenchmarks: Vertex Buffer Object and Vertex Array. http://www.songho.ca/opengl/gl_vbo.html.

[19] D. Abramson. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal*, 10(3):179–192, 2006.

[20] A. Amiri Sani, S. Nair, L. Zhong, and Q. Jacobson. Making I/O Virtualization Easy with Device Files. *Technical Report 2013-04-13, Rice University*, 2013.

[21] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: a Virtual Mobile Smartphone Architecture. In *Proc. ACM SOSP*, 2011.

[22] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proc. ACM SOSP*, 2003.

[23] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX ATC, FREENIX Track*, 2005.

[24] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B. A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proc. USENIX OSDI*, 2010.

[25] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating Systems Errors. In *Proc. ACM SOSP*, 2001.

[26] Y. Dong, Z. Yu, and G. Rose. SR-IOV Networking in Xen: Architecture, Design and Implementation. In *Proc. USENIX Workshop on I/O Virtualization (WIOV)*, 2008.

[27] M. Dowty and J. Sugerman. GPU Virtualization on VMware's Hosted I/O Architecture. *ACM SIGOPS Operating Systems Review*, 2009.

[28] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *Proc. Wrkshp. Operating System and Architectural Support for the On demand IT InfraStructure (OASIS)*, 2004.

[29] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP Kernel Crash Analysis. In *Proc. USENIX LISA*, 2006.

[30] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, D. Tsafrir, and A. Schuster. ELI: Bare-Metal Performance for I/O Virtualization. In *Proc. ACM ASPLOS*, 2012.

[31] A. Gordon, N. Har'El, A. Landau, M. Ben-Yehuda, and A. Traeger. Towards Exitless and Efficient Paravirtual I/O. In *Proc. SYSTOR*, 2012.

[32] J. G. Hansen. Blink: Advanced Display Multiplexing for Virtualized Applications. In *Proc. ACM Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, 2007.

[33] N. HarEl, A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger, and R. Ladelsky. Efficient and Scalable Paravirtual I/O System. In *Proc. USENIX ATC*, 2013.

[34] S. Kato, K. Lakshmanan, R. R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU Scheduling for Real-time Multi-tasking Environments. In *Proc. USENIX ATC*, 2011.

[35] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux Virtual Machine Monitor. In *Proc. Linux Symposium*, 2007.

[36] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. D. Lara. VMM-Independent Graphics Acceleration. In *Proc. ACM VEE*, 2007.

[37] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. IEEE Int. Conf. on Code Generation and Optimization*, 2004.

[38] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proc. USENIX OSDI*, 2004.

[39] J. Liu, W. Huang, B. Abali, and D. K. Panda. High Performance VMM-Bypass I/O in Virtual Machines. In *Proc. USENIX ATC*, 2006.

[40] R. Nikolaev and G. Back. VirtuOS: An Operating System with Kernel Virtualization. In *Proc. ACM SOSP*, 2013.

[41] D. Presotto, R. Pike, K. Thompson, and H. Trickey. Plan 9, a Distributed System. In *Proc. of the Spring 1991 EurOpen Conf.*, 1991.

[42] H. Raj and K. Schwan. High Performance and Scalable I/O Virtualization via Self-Virtualized Devices. In *Proc. ACM HPDC*, 2007.

[43] L. Rizzo. netmap: a Novel Framework for Fast Packet I/O. In *Proc. USENIX ATC*, 2012.

[44] R. Russel. virtio: Towards a De-Facto Standard for Virtual I/O Devices. *ACM SIGOPS Operating Systems Review*, 2008.

[45] L. Shi, H. Chen, and J. Sun. vCUDA: GPU Accelerated High Performance Computing in Virtual Machines. In *IEEE Int. Symp. Parallel & Distributed Processing*, 2009.

[46] C. Smowton. Secure 3D Graphics for Virtual Machines. In *Proc. ACM European Wrkshp. System Security*, 2009.

[47] J. Sugerman, G. Venkitachalam, and B. H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proc. USENIX ATC*, 2001.

[48] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proc. ACM SOSP*, 2003.

[49] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering Device Drivers. In *Proc. USENIX OSDI*, 2004.

[50] L. Tan, E. M. Chan, R. Farivar, N. Mallick, J. C. Carlyle, F. M. David, and R. H. Campbell. iKernel: Isolating buggy and malicious device drivers using hardware virtualization support. In *Proc. IEEE Int. Symp. Dependable, Autonomic and Secure Computing (DASC)*, 2007.

[51] M. Weiser. Program slicing. In *Proc. IEEE Int. Conf. on Software engineering*.

[52] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel. Concurrent Direct Network Access for Virtual Machine Monitors. In *Proc. IEEE High Performance Computer Architecture (HPCA)*, 2007.

[53] P. Willmann, S. Rixner, and A. L. Cox. Protection Strategies for Direct Access to Virtualized I/O Devices. In *Proc. USENIX ATC*, 2008.