RICE UNIVERSITY

Design and Implementation of I/O Servers Using the Device File Boundary

by

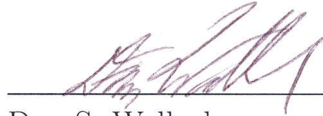
Ardalan Amiri Sani

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

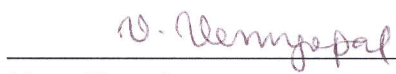
Approved, Thesis Committee:

Lin Zhong, Chair
Associate Professor of Electrical and Computer Engineering and Computer Science



Dan S. Wallach
Professor of Computer Science and Electrical and Computer Engineering

Peter J. Varman
Professor of Electrical and Computer Engineering and Computer Science



Venu Vasudevan
Adjunct Professor of Electrical and Computer Engineering

Houston, Texas

July, 2015

ABSTRACT

Design and Implementation of I/O Servers Using the Device File Boundary

by

Ardalan Amiri Sani

Due to historical reasons, today's computer systems treat I/O devices as second-class citizens, supporting them with ad hoc and poorly-developed system software. As I/O devices are getting more diverse and are taking a central role in modern systems from mobile systems to servers, such second-class system support hinders novel system services such as I/O virtualization and sharing. The goal of this thesis is to tackle these challenges by rethinking the system support for I/O devices.

For years, research for I/O devices is limited largely to network and storage devices. However, a diverse set of I/O devices are increasingly important for emerging computing paradigms. For modern mobile systems such as smartphones and tablets, I/O devices such as sensors and actuators are essential to the user experience. At the same time, high-performance computers in datacenters are embracing hardware specialization, or accelerators, such as GPU, DSP, crypto accelerator, etc., to improve the system performance and efficiency as the Dennard scaling has ended. Modern systems also treat such specialized hardware as I/O devices.

Since I/O devices are becoming the fundamental service provided by many computer systems, we suggest that they should be treated as *I/O servers* that are securely accessible to other computers, i.e., clients, as well. I/O servers will be the fundamental building blocks of future systems, enabling the novel system services mentioned

above. For example, they enable a video chat application running on a tablet to use the camera on the user's smart glasses and, for better consolidation, enable all applications running in a datacenter to share an accelerator cluster over the network.

We address two fundamental challenges of I/O servers: remote access and secure sharing. Remote access enables an application in one machine, either virtual or physical, to use an I/O device in a different machine. We use a novel boundary for remote access: Unix device files, which are used in Unix-like operating systems to abstract various I/O devices. Using the device file boundary for remote access requires low engineering effort as it is common to many classes of I/O devices. In addition, we show that this boundary achieves high performance, supports legacy applications and I/O devices, supports multiple clients, and makes all features of I/O devices available to unmodified applications.

An I/O server must provide security guarantees for untrusting clients. Using the device file boundary, a malicious client can exploit the – very common – security bugs in device drivers to compromise the I/O server and hence other clients. We propose two solutions for this problem. First, if available in the I/O server, we use a trusted hypervisor to enforce fault and device data isolation between clients. This solution assumes the driver is compromised and hence cannot guarantee functional correctness. Therefore, as a second solution, we present a novel device driver design, called library drivers, that minimizes the device driver Trusted Computing Base (TCB) size and attack surface and hence reduces the possibility of the driver-based exploits.

Using our solutions for remote access and secure sharing, we demonstrate that I/O servers enable novel system services: (*i*) I/O sharing between virtual machines, i.e., I/O virtualization, where virtual machines (VMs) share the I/O devices in the underlying physical machine, (*ii*) I/O sharing between mobile systems, where one mobile system uses the I/O devices of another system over a wireless connection, and (*iii*) I/O sharing between servers in a datacenter, where the VMs in one server use the I/O devices of other servers over the network.

Acknowledgments

First and foremost, I would like to thank my advisor, Lin Zhong, for his continuous support and guidance throughout the years. He helped me develop a better understanding of research methodologies. I am also thankful to my committee members, Dan S. Wallach, Peter J. Varman, and Venu Vasudevan. Their comments and feedback on this thesis have been of great value.

I am grateful to my group mates at RECG including Kevin Boos, Min Hong Yun, Shaopu Qin, and Felix Xiaozhu Lin for their help with this thesis research and for providing me with invaluable insights whenever I needed them.

Finally, I would like to thank my family and friends, especially Sitara Wijeratne, who have always been supportive of me.

Contents

Abstract ii

List of Illustrations ix

List of Tables xiii

1 Introduction 1

1.1 I/O Server Challenges 2

1.2 I/O Server System Services 3

1.2.1 I/O Sharing between Virtual Machines 4

1.2.2 I/O Sharing between Mobile Systems 4

1.2.3 I/O Sharing between Servers (in a Datacenter) 5

2 Related Work 7

2.1 I/O Sharing between Virtual Machines 7

2.2 I/O Sharing between Mobile Systems 9

2.3 I/O Sharing between Servers (in a Datacenters) 10

2.4 Device Driver and System Security 11

2.5 Other Related Work 15

2.5.1 Single System Image 15

2.5.2 Computation offloading 16

2.5.3 Use of Virtualization Hardware 16

3 Device File Boundary for Access to I/O Devices 17

3.1 Local Access 17

3.2 Remote Access 19

4 I/O Sharing between Virtual Machines 22

4.1 Paradice Design 25

4.1.1 Device Driver Sandboxing for Fault Isolation 26

4.1.2 Hypervisor-Enforced Access Permissions for Device Data Isolation 27

4.1.3 Key Benefits of Paradice Design 28

4.2 Isolation between Guest VMs 30

4.2.1 Fault Isolation 30

4.2.2 Device Data Isolation 33

4.3 Implementation 36

4.3.1 Paradice Architecture Details 36

4.3.2 Hypervisor-Assisted Memory Operations 40

4.3.3 Isolation between Guest VMs 42

4.4 Evaluation 44

4.4.1 Non-Performance Properties 45

4.4.2 Performance 45

4.5 Discussions 53

4.5.1 Performance Isolation & Correctness Guarantees 53

4.5.2 Other Devices, Hypervisors, and OSes 54

5 I/O Sharing between Mobile Systems 55

5.1 Use Cases 57

5.1.1 Use Cases Demonstrated with Rio 58

5.1.2 Future Use Cases of Rio 59

5.2 Design 60

5.2.1 Guarantees 61

5.3 Cross-System Memory Operations 62

5.3.1 Cross-System Memory Map 63

5.3.2 Cross-System Copy 66
5.4 Mitigating High Latency 67
5.4.1 Round Trips due to Copies 67
5.4.2 Round Trips due to File Operations 69
5.4.3 Round Trips due to DSM Coherence 70
5.4.4 Dealing with Poll Time-outs 70
5.5 Handling Disconnections 71
5.6 Android Implementation 72
5.6.1 Client & Server Stubs 73
5.6.2 DSM Module 75
5.6.3 Class-Specific Developments 77
5.6.4 Sharing between Heterogeneous Systems 79
5.7 Evaluation 80
5.7.1 Non-Performance Properties 81
5.7.2 Performance 82
5.7.3 Throughput 87
5.7.4 Power Consumption 90
5.7.5 Handling Disconnections 92
5.8 Discussions 92
5.8.1 Supporting more classes of I/O devices 92
5.8.2 Energy use by Rio 93
5.8.3 Supporting iOS 93

6 I/O Sharing between Servers (in a Datacenter) 94

6.1 Design 97
6.1.1 Cross-Machine Memory Map 98
6.1.2 Reducing Network Round Trips 100
6.2 Implementation 103

6.2.1 Linux Kernel Function Wrappers 104
6.3 Evaluation 106

7 Enhancing Device Driver Security 109

7.1 Library Driver: Design & Requirements 113
7.1.1 Design 113
7.1.2 Hardware Requirements 117
7.2 GPU Background 121
7.3 Glider: Library Driver for GPUs 124
7.3.1 Isolation of GPU Resources 125
7.3.2 The Device Kernel API 128
7.3.3 Reusing Legacy Driver Code for Glider 130
7.3.4 Other Implementation Challenges 131
7.4 Evaluation 132
7.4.1 Security 132
7.4.2 Performance 133
7.4.3 Library Driver Overheads 138
7.4.4 Engineering Effort 139
7.5 Discussions 140
7.5.1 Pros and Cons of Library Drivers 140
7.5.2 Devices with Multiple Hardware Execution Contexts 141
7.5.3 Radeon Instruction Set 141

8 Conclusions 143

Bibliography 144

Illustrations

3.1 I/O stack in Unix-like systems used for local access to I/O devices. . 18
3.2 Remote access to I/O devices at the device file boundary. 19

4.1 Devirtualization, our initial design for I/O paravirtualization 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. 25
4.2 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. 26
4.3 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. 28
4.4 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. 48
4.5 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. 49

4.6 3D HD games FPS at different resolutions. (DI) indicates the use device data isolation in Paradice. 50
4.7 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. 51
4.8 Guest VMs concurrently running the OpenCL matrix multiplication benchmark on a GPU shared through Paradice. The matrix order in this experiment is 500. 52

5.1 Rio uses the device file boundary for remote access. Hence, it splits the I/O stack between two mobile systems. 61
5.2 (a) Memory map for a local I/O device. (b) Cross-system memory map in Rio. 64
5.3 Typical execution of an `ioctl` file operation. Optimized Rio reduces the number of round trips. 68
5.4 Rio's architecture inside an Android system. Rio forwards to the server the file operations issued by the I/O service process through HAL. Rio supports unmodified applications but requires small changes to the class-specific I/O service process and/or HAL. 74
5.5 We use two different connections in our evaluation. In (a), the phones are connected to the same AP. This connection represents that used between mobile systems that are close to each other. In (b), the phones are connected over the Internet. This connection represents that used between mobile systems at different geographical locations. 80
5.6 Performance of (a) speaker and (b) microphone. The x-axis shows the buffering size in the HAL. The larger the buffer size, the smoother the playback/capture, but the larger the audio latency. The y-axis shows the achieved audio rate. 83

5.7 Performance of a real-time streaming camera preview (a) and photo capture (b) with a 21.9 Mbps wireless LAN connection between the client and server. Future wireless standards with higher throughput will improve performance without requiring changes to Rio. 85

5.8 Throughput for using (a) audio devices (speaker and microphone) and (b) the camera (for video streaming) with Rio. Note that the y-axis is in kbps and Mbps for (a) and (b), respectively. 89

5.9 Average system power consumption of when accelerometer, microphone, speaker, and camera (for video streaming) are used locally and remotely with Rio, respectively. For Rio, the power consumption for both the server and the client are shown. Scenarios 1 to 4 for audio devices correspond to audio buffering sizes of 3 ms, 10 ms, 30 ms, and 300 ms, respectively. Scenarios 1 to 6 for camera correspond to resolutions of 128×96, 176×144, 240×160, 320×240, 352×288, and 640×480, respectively. 91

6.1 Glance uses the device file boundary for remote access to GPUs. Using Glance, a guest VM in a physical machine without GPUs can remotely use a GPU in another physical machine, i.e., the target physical machine. The GPU is assigned to and controlled by a VM, called the driver VM. Compared to running the driver natively in the target physical machine, this provides better fault and device data isolation for guest VMs sharing the GPU. 98

6.2 OpenCL matrix multiplication. The x-axis shows the (square) matrix order. The y-axis shows the time it takes to execute the multiplication on the OpenCL-enabled platform, GPU or CPU. 107

7.1 (a) Legacy driver design. (b) Library driver design. Note that the solid-color interface in both figures refers to the set of APIs exposed by the device driver (or the device kernel in case of library drivers. These APIs are exposed through a device file, e.g., various `ioctls`, however in this figure, we do not show the device file itself for the sake of simplicity. 114

7.2 Simplified GPU hardware model. 122

7.3 OpenCL matrix multiplication. The x-axis shows the (square) matrix order. The y-axis is the time to execute the multiplication on the GPU, normalized to the average time by the legacy driver. 135

7.4 Graphics performance. The box plots show 90% percentile, 10% percentile, and median. The whiskerbars depicts maximum and minimum. The box plots are highly concentrated around the mean. . 136

7.5 A sample run of the Vertex Array benchmark. The framerate is measured every second. The figure shows the rare drops of framerate and the slow start of Glider. 137

Tables

2.1 Comparing I/O virtualization solutions. The "Paravirtualization" row refers to paravirtualization solutions other than Paradice. 8

4.1 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 4.2. (*) marks the devices that we have tested only with our previous system design, devirtualization (§4.1). We include them here to show that the device file boundary is applicable to various device makes and models. 37

4.2 Paradice code breakdown. 39

5.1 Rio code breakdown. 73

6.1 Glance code breakdown. 104

7.1 TCB size and attack surface for the legacy and Glider for the Radeon HD 6450 and Intel Ivy Bridge GPUs. The numbers for both TCB columns are in kLoC. The first TCB columns reports LoC in both source and header files. The second TCB column excludes the header files. 134

Chapter 1

Introduction

Due to historical reasons, today's computer systems treat I/O devices as second-class citizens, supporting them with ad hoc and poorly-developed system software. As I/O devices are getting more diverse and are taking a central role in modern systems from mobile systems to servers, such second-class system support hinders novel system services such as I/O virtualization and sharing. The goal of this thesis is to tackle these challenges by rethinking the system support for I/O devices.

For years, research for I/O devices is limited largely to network and storage devices. However, a diverse set of I/O devices are increasingly important for emerging computing paradigms. For modern mobile systems such as smartphones and tablets, I/O devices such as sensors and actuators are essential to the user experience. At the same time, high-performance computers in datacenters are embracing hardware specialization, or accelerators, such as GPU, DSP, crypto accelerator, etc., to improve the system performance and efficiency as the Dennard scaling has ended. Modern systems also treat such specialized hardware as I/O devices.

Since I/O devices are becoming the fundamental service provided by many computer systems, we suggest that they should be treated as *I/O servers* that are securely accessible to other computers, i.e., clients, as well. I/O servers will be the fundamental building blocks of future systems, enabling the novel system services mentioned above. For example, they enable a video chat application running on a tablet to use the camera on the user's smart glasses and, for better consolidation, enable all

applications running in a datacenter to share an accelerator cluster over the network.

1.1 I/O Server Challenges

We address two fundamental challenges of I/O servers: remote access to I/O devices and secure sharing of I/O devices between untrusting clients.

Remote Access: Remote access enables applications in one machine, virtual or physical, to use an I/O device in a different machine. The vast diversity among I/O devices has forced system software designers to employ I/O-specific and custom solutions for remote access to different devices. These solutions are often in the form of middleware services [1], custom applications [2–4], or custom virtualization solutions [5–7]. As a result, existing solutions suffer from important limitations: They require high engineering effort, do not support legacy I/O devices or applications, do not support more than one I/O client, do not support all the functionalities of I/O devices, and do not achieve high performance.

In order to solve these limitations, we propose a unified operating system boundary for remote access to various I/O devices. Our key idea is to leverage Unix device files for this purpose. In Unix-like operating systems, I/O devices are abstracted as device files, providing a common boundary for accessing many classes of I/O devices. To use this boundary for remote access, we create a virtual device file in the I/O client, intercept the file operations issued on the virtual device file by the applications in the client, and forward them to the to the I/O server to be executed by the device driver. In this thesis, we demonstrate that remote access using this boundary solves all the limitations of existing solutions mentioned earlier.

Secure Sharing: An I/O server might support multiple untrusting clients and hence it must guarantee secure shared access to the I/O device. However, by adopting

the device file boundary, an I/O server allows its clients to directly access the device driver. This allows malicious programs in the clients to exploit the – very common – security bugs in these drivers in order to compromise them. We propose two solutions for this problem. The first solution (§4.2) assumes that the device driver is already compromised. It uses a trusted hypervisor to enforce fault and device data isolation between the clients despite the compromised driver. It requires no or small changes to the device driver, and hence comes with low engineering cost. However, since the driver may be compromised, this solution cannot guarantee functional correctness. Moreover, this solution requires a trusted hypervisor in the I/O server, which is not always available.

The second solution, called library drivers (§7), takes a complementary approach. It reduces the size and attack surface of the device driver Trusted Computing Base (TCB), and hence reduces the possibility of the driver getting compromised in the first place. To do so, this solution refactors the resource management part of the driver into untrusted libraries loaded by applications. Library drivers require per-driver engineering effort. In addition to secure remote access, library drivers are useful for secure local access as well. That is, they can replace existing device drivers in the operating system to give untrusting applications secure local access to I/O devices.

1.2 I/O Server System Services

Using the aforementioned solutions for remote access and secure sharing, we demonstrate important system services enabled by I/O servers, as discussed below.

1.2.1 I/O Sharing between Virtual Machines

Virtualization has become an important technology for computers of various form factors, from servers to mobile systems, because it enables secure co-existence of multiple operating systems in one physical machine. I/O sharing between virtual machines (VMs), i.e., I/O virtualization, enables the VMs to share I/O devices in the underlying physical machine.

In §4, we present Paradice, an I/O virtualization solution at the device file boundary. In Paradice, the I/O server is a VM controlling the I/O device and running its device driver and the clients are the guest VMs that wish to use the I/O device. The clients communicate with the I/O server through the hypervisor, e.g., using hypervisor-based shared-memory pages and inter-VM interrupts. We solve two important challenge in Paradice. First, we support cross-VM memory operations needed for handling the forwarded (device) file operations. Second, as mentioned earlier, we use the hypervisor in Paradice to enforce fault and device data isolation between VMs sharing an I/O device.

1.2.2 I/O Sharing between Mobile Systems

I/O sharing between mobile systems enables applications in one mobile systems to use the I/O devices in another system over a wireless connection. I/O sharing between mobile systems has important use cases as users nowadays own a variety of mobile systems, including smartphones, tablets, smart glasses, and smart watches, each equipped with a plethora of I/O devices, such as cameras, speakers, microphones, sensors, and cellular modems. Example use cases are using a tablet to control the high-resolution rear camera of the smartphone to take a self-portrait, sharing music with a friend by playing the music on their smartphone speaker, and using the tablet

to make a phone call using the SIM card in the smartphone.

In §5, we present Rio, an I/O sharing solution at the device file boundary. In Rio, the I/O server and client are the mobile systems sharing an I/O device and communicating through a wireless connection. We solve three important challenges in Rio. First, compared to I/O virtualization, I/O sharing over the network poses a fundamental challenge for supporting the device driver memory operations since the client and server do not have access to physically shared memory. Our solution to this problem is a novel Distributed Shared Memory (DSM) design that supports access to shared pages by the process, the driver, and the I/O device (through DMA). Second, in order to combat the high wireless link latency, we minimize the number of communication round trips between the client and server as a result of copy operations, file operations, and DSM coherence messages. Third, we provide support for handling the disconnections in the wireless link while an I/O device is remotely accessed.

1.2.3 I/O Sharing between Servers (in a Datacenter)

I/O sharing between servers in a datacenter enables VMs in one server to use the I/O devices in another server over the network. New classes of I/O devices are being employed in datacenters. Accelerators, such as GPUs, are especially popular as they can execute certain workload faster and more efficiently compared to CPUs. For example, GPUs are suitable for scientific computing, visualization, video processing, encryption, and many other workloads. Unfortunately, accelerators are expensive to use in datacenters today since they are poorly consolidated.

In §6, we present Glance, a system for consolidating GPU resources in datacenters using the device file boundary. In Glance, the I/O server is a VM controlling the GPU and the I/O clients are other VMs in the datacenter running in different physical

machines communicating with the I/O server over the datacenter network. Compared to I/O sharing between mobile systems, we face a fundamental challenge in Glance. In order to achieve the best performance for devices, e.g., GPUs, datacenters often employ closed source drivers available from device vendors. However, closed source drivers create challenges for adopting a DSM solution similar to the one used in Rio and for reducing the number of round trips between the I/O server and client. In §6, we discuss these challenges and present our solutions.

Chapter 2

Related Work

2.1 I/O Sharing between Virtual Machines

Remote access to I/O devices at the device file boundary can be used for I/O sharing between virtual machines, i.e., I/O virtualization or more accurately I/O paravirtualization, as we will demonstrate with Paradice (§4). I/O paravirtualization achieves high performance by employing paravirtual drivers that minimize the overhead of virtualization, e.g., minimize the VM boundary crossings, compared to emulation. However, existing paravirtualization solutions [6, 8–11] only support one class of I/O devices and require significant development effort to support new device classes and features. This is because in existing solutions the boundary between the paravirtual drivers are class-specific. In contrast, Paradice requires low engineering effort to support various classes of I/O devices since it uses a common boundary, i.e., the device files.

In addition to paravirtualization, there are four main solutions for I/O virtualization in whole system virtualization. *Emulation* [5, 12], mentioned above, virtualizes the I/O device by emulating the hardware interface of the device. Emulation is known to have poor performance due to frequent VM boundary crossings incurred and due to limited capabilities of the emulated device compared to the physical one. *Direct device assignment* [13–17], also called *device pass-through*, provides high performance by allowing the VM to directly control and access the physical devices; however,

	High Performance	**Low Development Effort**	**Device Sharing**	**Legacy Device**
Emulation	*No*	*No*	Yes	Yes
Direct device assignment	Yes	Yes	*No*	Yes
Self-virtualization	Yes	Yes	Yes *(limited)*	*No*
Mediated pass-through	Yes	*No*	Yes	*No*
Paravirtualization	Yes	*No*	Yes	Yes
Paradice	Yes	Yes	Yes	Yes

Table 2.1 : Comparing I/O virtualization solutions. The "Paravirtualization" row refers to paravirtualization solutions other than Paradice.

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 (§4.1), but it allows multiple guest VMs to share the device, which is not possible with device assignment alone. *Self-virtualization* [18–21] requires virtualization support in the I/O device hardware and therefore does not apply to legacy devices. *Mediated pass-through*, e.g., gVirt [7] and NVIDIA GRID solutions [22], virtualizes an I/O device by giving the VMs direct access to performance-critical resources of the device while emulating their access to sensitive resources, allowing VMs to use the native device driver. Mediated pass-through is an I/O-specific solution and hence requires high engineering effort to support a wide range of devices. Moreover, a mediated pass-through solution requires some hardware features on the I/O device, e.g., memory protection primitives on the GPU to isolate accesses to the device memory, and hence does not apply to all legacy devices. Table 2.1 compares different I/O virtualization solutions.

Some solutions provide GPU framework virtualization by remoting OpenGL [23–25] or GPGPU (e.g., CUDA and OpenCL) [26–30] API. These solutions are more limited than Paradice because they are only applicable to those particular frameworks.

Moreover, these solution require to be updated frequently when the framework API is updated. LoGV [31] also remotes the GPGPU framework API. However, it gives VMs direct access to GPU resources, such as the command submission channel, for better performance and uses the GPU hardware protection mechanisms to enforce isolation between VMs, similar to mediated pass-through solutions.

GPUvm [32] emulates the hardware interface of existing GPUs in the hypervisor, or full virtualization of a GPU. This allows the guest VMs to use unmodified device drivers. Moreover, to reduce the overhead of emulation, it employs paravirtualization. Unlike Paradice, GPUvm is a GPU-specific solution.

Cells [33] 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.

2.2 I/O Sharing between Mobile Systems

Remote access to I/O devices at the device file boundary can be used for I/O sharing between mobile systems, as we will demonstrate with Rio (§5). The value of I/O sharing has been recognized by others both for mobile and non-mobile systems. Compared to Rio, existing solutions have three limitations: They do not support unmodified applications, do not expose all I/O device functions to the client, or are I/O class-specific.

Various I/O sharing solutions exist for mobile systems today, all of which suffer from the fundamental limitations described above. For example, IP Webcam [2] turns a mobile system's camera into an IP camera, which can then be viewed from another mobile system through a custom viewer application. The client system can-

not configure all camera parameters, such as resolution; These parameters must be manually configured on the server. Wi-Fi Speaker [3] allows music from a computer to be played on a mobile system's speaker. It does not, however, support sharing the microphone. MightyText [4] allows the user to send SMS and MMS messages from a PC or a mobile system using the SIM card and modem in another system. It does not support phone calls. Miracast [34] and Chromecast [35] allow one system to display its screen on another system's screen. These solutions are however graphics-specific. Morever, Chromecast requires specialized hardware. More recently, Apple Continuity enables sharing of the cellular modem between iOS machines as well [36].

Participatory and cooperative sensing systems collect sensor data from registered or nearby mobile systems [37,38]. These systems use custom applications installed on mobile systems and are therefore more limited than Rio, which supports a variety of I/O devices. Indeed, these systems can incorporate Rio to more easily collect sensor data from other systems.

Other I/O sharing solutions include thin client solutions (such as the X window system [1], THINC [39], Microsoft Remote Desktop [40], VNC [41], Citrix XenDesktop [42], and Sun Ray [43]), remote file systems [44–46], network USB devices [47–50], wireless displays [51], remote printers [52], IP cameras [53], remote visualization of 3D graphics [54], and multi-device platforms [55]. In contrast to Rio, these solutions are I/O class-specific.

2.3 I/O Sharing between Servers (in a Datacenters)

Remote access to I/O devices at the device file boundary can be used for I/O sharing between servers in a datacenter, as we will demonstrate with Glance (§6). Similarly, the Multi-Root I/O Virtualization (MR-IOV) standard allows multiple machines to

access and share the same set of PCI devices [56]. However, there are currently no PCI devices that implement this standard. Instead, Ladon [57] uses existing SR-IOV devices and the Non-Transparent Bridge (NTB) in a PCI network to achieve the same functionality as that provided by MR-IOV. Both of these techniques require hardware support on I/O devices and expensive PCI network. In contrast, Glance works with legacy I/O devices and low-cost datacenter network connections such as Ethernet.

An alternative solution is to intercept and forward the API calls of a framework from one machine to another, e.g., for OpenCL [58] and CUDA [30, 59, 60]. However, unlike Glance, these solutions are limited to a specific framework and require to be updated frequently as the framework API changes.

Asymmetric Distributed Shared Memory (ADSM) [61] is a programming model for heterogeneous computing. In ADSM, all the coherence and consistency protocols are implemented by the CPU (and not the GPU), conceptually similar to our DSM solution in Glance, where the coherence protocol is implemented by the client (and not the server). However, unlike Glance, ADSM targets applications using GPUs in the same machine. Moreover, ADSM is implemented on top of CUDA, unlike Glance that is agnostic to the GPGPU framework.

2.4 Device Driver and System Security

Library drivers (§7) and their realization for GPUs, called Glider, improve the system security by reducing the size of the device driver TCB and attack surface. They do so by refactoring the resource management part of the device driver into an untrusted library, which is loaded by each application's process. Used in an I/O server, library drivers improve the server's security in face of malicious clients as well. Many previous solutions have attempted to reduce the drivers' risk on the system security. Some

solutions move the driver to user space. For examples, in microkernels, drivers reside completely in user space in order to keep the kernel minimal [62–66]. SUD also encapsulates a Linux device driver in a user space process, and similar to Glider, uses the IOMMU and User-Mode Linux (UML) [67]. Microdriver is another solution, which splits the driver between the kernel and user space [68]. It keeps the critical path code of the driver in the kernel, but redirects the rest of the execution to the user space. Hunt [69] also presents a solution that moves the driver to the user space by employing a proxy in the kernel. All these solutions improve the security of the operating system kernel by removing the driver. However, in contrast to a library driver, they cannot improve the isolation between processes using the device, since the driver is fully shared between the processes.

Others move the device driver to a VM such as in LeVasseur et al. [70], VirtuOS [71], and iKernel [72]. They also improve the system security but do not improve the isolation between processes. Moreover, similar to the approach used in this thesis, iKernel [72] forwards the file operations from the host to a driver VM. It demonstrates support for a simple LED device, unlike our systems, e.g., Paradice, Rio, and Glance, that support more diverse and sophisticated I/O devices such as GPUs. Moreover, iKernel authors do not report support for the `mmap` file operation, which is used by many device drivers.

Other solutions try to protect against errors in the drivers, either using runtime and hardware-enforced techniques such as Nooks [73] or using language-based techniques such as SafeDrive [74]. In contrast, a library driver improves the system security by reducing the TCB size and attack surface through the principle of untrusted resource management.

Zhou et al. move the device driver to the user space in order to create a trusted

path between the application and the device [75, 76]. However, they assume a single application using an I/O device and therefore they assign the device to the application. Unlike them, library drivers aim at cases where the I/O device is shared between untrusting applications.

Schaelicke [77] and Pratt [78,79] propose hardware architectures to support secure user space access to devices. In this work, we show that adequate hardware primitives already exist on commodity platforms and accelerators, such as GPUs, to run the device management code in the user space.

Exokernel [80], U-Net [81], and sv3 [82] move part or all of the network device driver to the user space for better security or performance. MyCloud SEP detangles the resource management functions for disks and makes them untrusted in order to reduce the TCB of virtualization platforms [83]. We share the same goals with this line of work. However, with library drivers, we demonstrate that such an approach is applicable to a wider range of devices, such as GPUs.

Some existing device drivers incorporate a user space components in addition to the kernel driver. Such drivers differ from library drivers in one of the following two ways. First, the user space component is in the TCB since it is shared by applications using the device. By compromising this user space component, a malicious application can attack other applications. This component is either a process (e.g., the driver host process in Windows User-Mode Driver Framework (UMDF) [84]) or is a library loaded by a shared process (e.g., Mali GPU drivers for Android, where libraries are loaded by the surface_flinger process). Second, although the I/O stack includes untrusted libraries, the kernel driver implements high-level resource management with several API calls. For example, the in-kernel Display Miniport Driver of Windows Display Driver Model (WDDM) implements ~70 API calls including API

for command submission [85]. Mali kernel drivers for Linux exports ~40 API calls. In contrast, the kernel component (i.e., the device kernel) in a library driver exports a few low-level API calls.

Some devices provide per-process hardware support. Examples are some NVIDIA GPUs and Infiniband devices, which support per-process command queues. While such hardware support is added for performance, it improves the system security as well by reducing the TCB size, similar to library drivers. Our work shows that similar goals can be achieved for devices without such explicit hardware support as well.

Library drivers execute the resource management part of the device driver in an untrusted library. Similarly, library operating systems, such as Exokernel [80] and Drawbridge [86], improve the system security by executing the operating system management components as a library in the application's process address space. Library drivers are complementary to library operating systems; they can serve as a secure way for applications in a library operating system to use the devices.

Hardware sandboxing techniques, such as Embassies [87], Xax [88], and the Bromium micro-virtualization [89], improve the system security by reducing the TCB size. Library drivers can complement these sandboxes by providing them with secure access to devices.

CuriOS [90] improves the isolation between applications that use a microkernel service, such as the file system, by providing the service with access to client states only when the service is processing a request. This allows for recovery from most of the errors in the service. In contrast, library drivers improve the isolation between application by reducing the TCB size and attack surface.

Criswell et al. use compiler techniques to protect the applications from an untrusted operating system [91] or enhance the security of the operating systems by

providing complete control-flow integrity [92]. These solutions apply to device drivers as well and are complementary to the solutions presented in this thesis.

Heterogeneous System Architecture (HSA) [93] is a standard for future accelerators, targeting both the accelerator hardware and software. One hardware features of the HSA is support for user space dispatch, allowing applications to dispatch instructions to the accelerator without communicating with the driver in the kernel. This will hence reduce the size of the device driver in the kernel. In our work on library drivers, we demonstrated that such a feature is feasible even with commodity accelerators. Moreover, we anticipate that other resource management tasks, such as memory management, still remain in the trusted device drivers for HSA-compliant devices, whereas a library driver makes all the management code untrusted.

2.5 Other Related Work

2.5.1 Single System Image

Distributed and cluster operating systems and virtual machines such as Plan 9 [94], LOCUS [95], openMosix [96], OpenSSI [97], Kerrighed [98, 99], PVM [100], and vNUMA [101] provide a single system image abstraction for applications running on a cluster of machines. As a result, they often provide solutions for an application to use the I/O devices in other machines. Plan 9 uses files to achieve this. However, unlike our work, Plan 9 files do not support `mmap` and `ioctl` file operations, making it difficult to support modern I/O devices such as GPUs. OpenSSI also provides support for remote access to devices using the device files in Linux. However, at the time of this writing, OpenSSI does not support the `mmap` file operation needed for many modern I/O devices. Other solutions either do not provide a remote I/O mechanism

or provide it for a few classes of I/O devices, mainly storage and network devices, through class-specific solutions. In contrast, in this thesis, we provide remote access for many classes of I/O devices.

2.5.2 Computation offloading

There is a large body of literature regarding offloading computation from mobile systems [102]. Example solutions are [103], MAUI [104], CloneCloud [105], and COMET [106]. I/O sharing, as is concerned in this work, e.g., in Rio, invites a different set of research challenges and has a focus on system support rather than programming support. Nevertheless, both computation offloading and I/O sharing benefit from existing techniques for distributed systems. For example, both Rio and COMET employ DSM, albeit with different designs.

2.5.3 Use of Virtualization Hardware

Paradice and Glider use the new hardware virtualization extensions to provide isolation between VMs and applications, respectively. In addition to their primary use case, i.e., virtualization, hardware virtualization extensions have been used for other purposes as well. For example, the nonkernel [107] uses these extensions to give applications direct access to devices. Dune [108] uses the virtualization hardware to give applications direct but safe access to privileged hardware features and demonstrates important use case of such an access including sandboxing, privilege separation, and garbage collection. Willman et al. [109] propose different strategies to use the IOMMU to provide protection when VMs are given direct access to devices with DMA capability (although Paradice's use of the IOMMU to isolate the data of guest VMs shared with I/O devices is not addressed in [109]).

Chapter 3

Device File Boundary for Access to I/O Devices

In Unix-like operating systems, I/O devices are abstracted as device files. While device files have traditionally been used for local access to I/O devices, in this thesis, we show that device files can be used for remote access to I/O devices as well. In this chapter, we provide background information on device files and explain how in principle they can be used for remote access. §4, §5, and §6 will then provide details on how we leverage this boundary for remote access in different systems.

3.1 Local Access

Local access to I/O devices refers to an application using an I/O device in the same machine. In the case of local access, the application, I/O device, and the device driver all operate within the operating system inside that machine. Figure 3.1 shows a simplified I/O stack in a Unix-like operating system, which is used for local access to I/O devices. The kernel exports device files to the user space through a special filesystem, `devfs` (i.e., the `/dev` directory). An application 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 application process memory. There are two types of memory

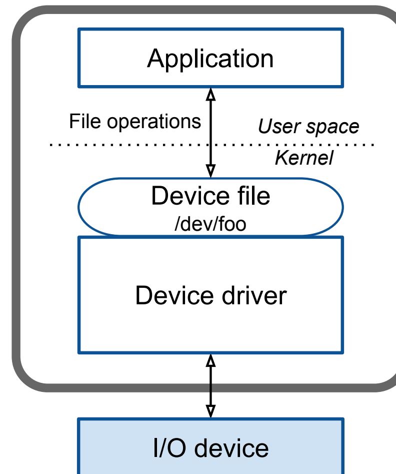


Figure 3.1 : I/O stack in Unix-like systems used for local access to I/O devices.

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.

It is often the application that initiates a communication with the device driver. However, if an I/O device needs to notify the application of events, e.g., a touch event on the touchscreen, the notification is done using the `poll` file operation, either blocking or non-blocking. In the blocking case, the application issues a `poll` file operation that blocks in the kernel until the event occurs. In the non-blocking case, the application periodically issues `polls` to check for the occurrence of an event. An alternative to the `poll` file operation is an asynchronous notification. In this case, the application requests to be asynchronously notified when events happen, e.g., when there is a mouse movement. In Linux, applications use the `fasync` file operation for setting up the asynchronous notification. When there is an event, the application is notified with a signal.

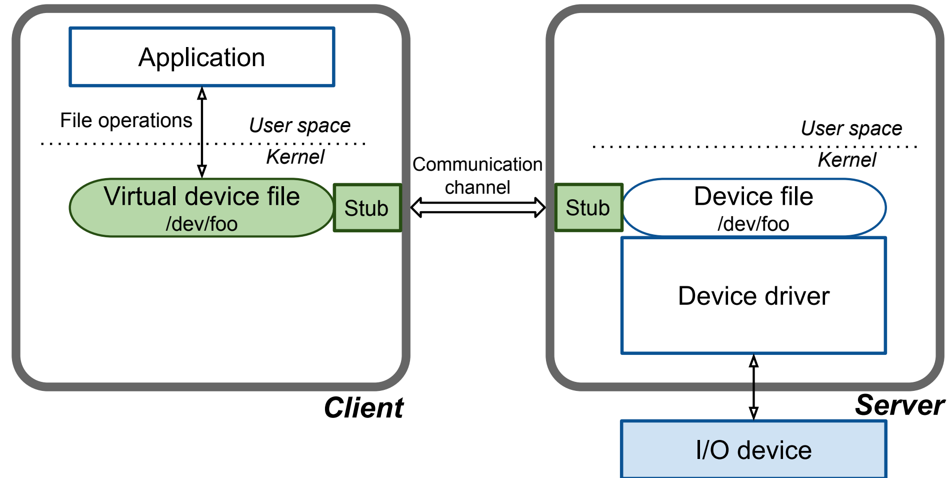


Figure 3.2 : Remote access to I/O devices at the device file boundary.

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 and the driver export this information to the user space, e.g., through the `/sys` and `/proc` 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 operating systems. Important exceptions are network and block devices, which have their own class-specific I/O stacks.

3.2 Remote Access

Remote access to I/O devices refers to an application in one machine, virtual or physical, to use an I/O device in another machine. Figure 3.2 depicts how we remotely access an I/O device at the device file boundary. It shows two machines: the *I/O server* (or server for short) and the *I/O client* (or client for short). The server has

an I/O device that the client wishes to use. To do so, we create a *virtual device file* in the client that corresponds to the actual device file in the server. The virtual device file creates the illusion to the client's application that the I/O device is present locally in the client. To use this I/O device, the application in the client executes file operations on the virtual device file. These file operations are then handled by the *client stub* module, which packs the arguments of each file operation into a packet and sends it to the *server stub* module over some communication channel. The server stub unpacks the arguments and executes the file operation on the actual device file. It then sends back the return values of the file operation to the client stub, which returns them to the application.

Note that Figure 3.2 only shows one client using a single I/O device from a single server. However, the device file boundary allows a client to use multiple I/O devices from multiple servers. It also allows multiple clients to use an I/O device from a single server. Moreover, this boundary allows a system to act as both a client and a server simultaneously for different devices or for different machines.

Figure 3.2 illustrates a generic view of remote access at the device file boundary. The server, the client, and the communication channel between them vary in different systems. As mentioned in §1, in this thesis, we demonstrate different system services of I/O servers. In I/O sharing between virtual machines, i.e., I/O virtualization, (§4), the server is the VM in control of the I/O device, the client is the VM that wishes to use the I/O device, and the communication channel is through the hypervisor, e.g., hypervisor-based shared memory pages and inter-VM interrupts. In I/O sharing between mobile systems (§5), the client and server are mobile systems and the communication channel between them is a wireless link. In I/O sharing between servers (in a datacenter) (§6), the I/O server is the virtual machine in control of the

I/O device, the clients are VMs in other physical machines that wish to use the I/O device, and the communication channel between them is the datacenter network.

Chapter 4

I/O Sharing between Virtual Machines

Virtualization has become an important technology for computers of various form factors, from servers to mobile systems, because it enables secure co-existence of multiple virtual machines (VMs), each with their operating systems, in one physical machine. I/O sharing between VMs, i.e., I/O virtualization, enables VMs to share the 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 [110].

An ideal I/O virtualization solution must satisfy four important requirements: the ability to share the device between multiple VMs, support for legacy devices without virtualization hardware, high performance, and low engineering effort. Unfortunately, none of the existing solutions, namely, emulation, paravirtualization, direct device assignment, self-virtualization, and mediated pass-through, meet all the four requirements, as was explained in §2.1. In this chapter, we demonstrate that I/O servers using the device file boundary provide us with an I/O virtualization solution that achieves all the four aforementioned requirements.

We present Paradice [111], our design and implementation of I/O virtualization using the device file boundary. To leverage this boundary, Paradice creates a virtual device file in the guest VM corresponding to the device file of the I/O device. Guest VM's applications issue file operations to this virtual device file as if it were the real

one. A stub in the guest OS kernel (i.e., the client stub) intercepts these file operations and forwards the file operations to be executed by the actual device driver. As will be discussed later, for better isolation, we run the device driver in a VM, called the driver VM, which has full control of the I/O device through direct device assignment. The driver VM acts as the I/O server for the guest VMs. A stub module in the OS kernel in the driver VM, i.e., the server stub, communicates with the guest VM in order to receive and pass on the file operations to the device driver.

Paradice, a short form *Para*virtual *D*ev*ice*, is indeed a form of I/O paravirtualzation, where the stub modules act as paravirtual drivers. Existing paravirtualization solutions require significant engineering effort since they are I/O class-specific. As a result of the required engineering effort, only network and block devices have enjoyed good paravirtualization solutions in the past [8, 10, 112, 113]. Limited attempts have been made to paravirtualize other device classes, such as GPUs [6], but such solutions provide low performance and do not support new class features, such as GPGPU. Paradice solves this limitation by using the common device file boundary, allowing a single implementation of Paradice to support various classes of I/O devices. Moreover, Paradice maintains three important properties of existing I/O paravirtualization solutions including the ability to share the device between multiple VMs, support for legacy devices without virtualization hardware, and high performance. Therefore, as mentioned earlier, Paradice manages to achieve all the four important requirements of an I/O virtualization solution.

We address two fundamental challenges in Paradice: (i) the guest VM application and the device driver reside in different virtualization domains, creating a memory barrier for executing the driver memory operations on the guest application process. Our solution to this problem is to redirect and execute the memory operations effi-

ciently in the hypervisor without any changes to the device drivers or applications. (*ii*) malicious applications can exploit device driver bugs [114,115] through the device file interface to compromise the machine that hosts the device driver [116,117]. 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 when using unmodified device drivers. However, it requires modest changes to the 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 [118]. Approximately 400 lines of this class-specific code are for device data isolation for GPU. We note that GPU has previously been challenging to virtualize due to its functional and implementation complexity. Yet, Paradice easily virtualizes GPUs of various makes and models with full functionality and close-to-local 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 as well, for example, for reusing Linux GPU drivers for FreeBSD, which typically does not have 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-local performance for various devices and applications, both for Linux and FreeBSD VMs; and

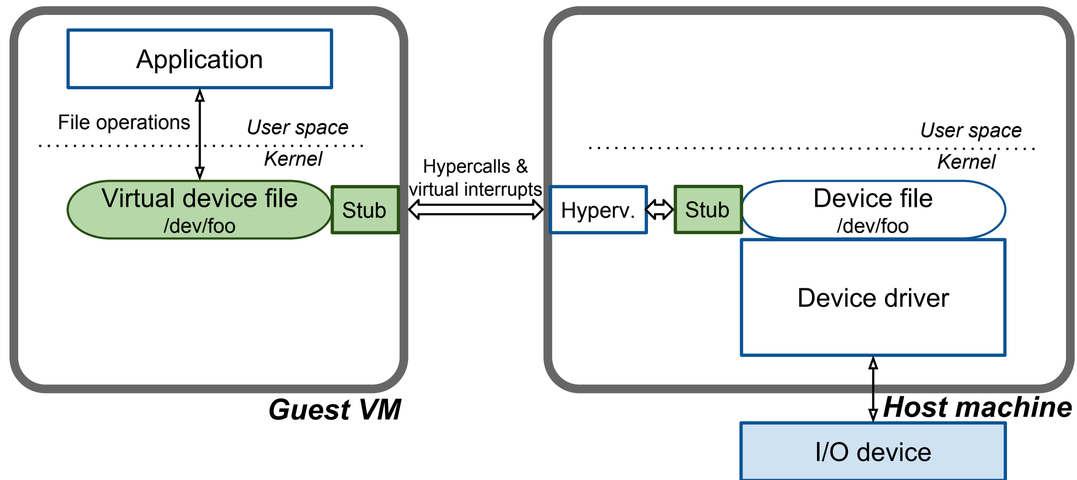


Figure 4.1 : Devirtualization, our initial design for I/O paravirtualization at the device file boundary. This design can be abused by a malicious guest VM to compromise the host machine, and hence the hypervisor and other guest VMs.

(*iii*) provides fault and device data isolation without incurring noticeable performance degradation.

4.1 Paradice Design

We use the device file as the paravirtualization boundary. Figure 4.1 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 client and server stubs are in the guest VM and in the host OS, respectively. We initially used this design, called devirtualization [119]. 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

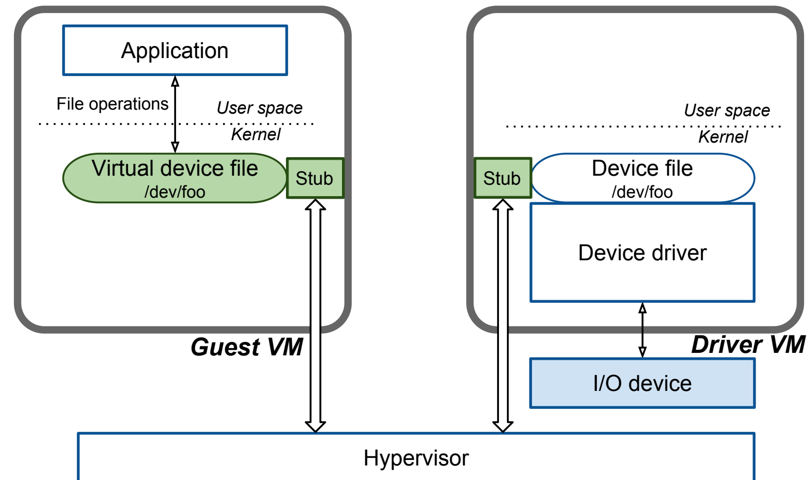


Figure 4.2 : Paradice's design, which sandboxes the device and its driver in a driver VM and performs strict runtime checks on requested memory operations for fault isolation.

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.

4.1.1 Device Driver Sandboxing 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 direct device assignment [13–16]. This design uses a Type I (i.e., bare-metal) hypervisor, as shown in Figure 4.2. 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 stub modules use shared memory pages and inter-VM interrupts to communicate, e.g., for forwarding the file operations.

Executing driver memory operations (§3.1) 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 (§3.1) efficiently in the hypervisor and provide an API for the driver VM to request them. Further, to support unmodified drivers, we provide function wrappers 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 client stub in the guest VM identifies and declares the legitimate memory operations to the hypervisor before forwarding a file operation to the server stub. §4.2.1 explains this issue in more detail.

4.1.2 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 in the driver VM memory and in the device memory for each guest VM's data and assigning appropriate access permissions to these regions (Figure 4.3). §4.2.2 elaborates on this technique.

Unmodified device drivers cannot normally function correctly in the presence of the hypervisor-enforced device data isolation and need modifications. In §4.3.3, we explain how we added only ~400 LoC to the complex Linux Radeon GPU driver

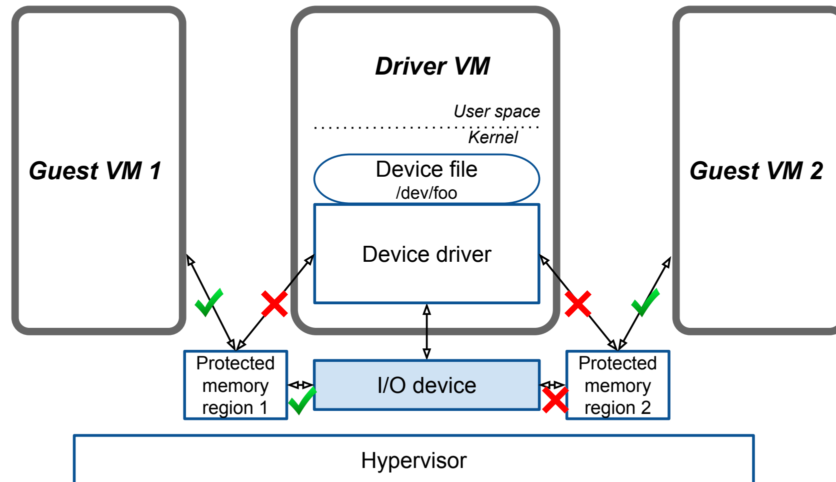


Figure 4.3 : Device data isolation, enforced by the hypervisor, which creates non-overlapping protected memory regions for each guest VM's data and enforces appropriate access permissions to these regions. Each protected region includes part of driver VM system memory and device memory.

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.

4.1.3 Key Benefits of Paradice Design

One Paravirtual Driver, Many I/O Devices

The key benefit of Paradice is that it requires a single pair of paravirtual drivers, i.e., client and server stubs, 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.

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 (§3.1) 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). §4.3.1 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 compatible.

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 stub module in the driver VM handles requests from the stub modules in 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. §4.3.1 discusses the issue of concurrency for different classes of devices.

4.2 Isolation between Guest VMs

Device drivers are buggy [114, 115], and these bugs can be used by malicious applications to compromise the machine, either virtual or physical, that hosts the driver [116, 117]. 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.

4.2.1 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 §4.1. 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 client stub 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 server stub. The legitimate memory operations can often be easily identified in the client stub 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 an object of 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 object in the process memory. Therefore in these cases, the client stub 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 an object, 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 hence can 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 client stub. Given an `ioctl` command, the client stub 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 client stub 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 in the client stub.

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.

4.2.2 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 assumed to be 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 4.3. The CPU code in the driver VM, including the device driver itself, does not have permission to read these memory regions. Each guest VM has access to its own memory region only (through the memory operation checks 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 transfer the buffer to 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 application 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 Extended Page Table (EPT) read permission for the protected memory regions. The hypervisor uses the EPTs to virtualizes the physical memory for the VMs. While memory virtualization was tradi-

tionally implemented entirely in software, i.e., the shadow page table technique [120], recent generations of micro-architecture provide hardware support for memory virtualization, e.g., the Intel 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. Enforcing access permissions for the driver VM by removing the EPT read permissions 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 have DMA access 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.

4.3 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 4.1 shows the list of devices that we have paravirtualized with Paradice. Table 4.2 breaks down the Paradice code structure. We use CLOC [121] (v1.56) for measuring the code size. We do not count comments or debugging code. The implementation is open source [122].

4.3.1 Paradice Architecture Details

Client and Server Stub Modules: The client and server stubs constitute a large portion of the implementation and consist of two parts each. The first part implements the interface to the hypervisor, e.g., invoking the hypervisor API in the server stub, or sharing a grant table with the hypervisor in the client stub. The second part interacts with the OS kernel only, e.g., by invoking the device driver's file operation handlers in the server stub, or by handling (and forwarding) the file operations in the client

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 4.1 : 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 4.2. (*) marks the devices that we have tested only with our previous system design, devirtualization (§4.1). We include them here to show that the device file boundary is applicable to various device makes and models.

stub.

The client and server stubs use shared memory pages and inter-VM interrupts to communicate. The client stub puts the file operation arguments in a shared page, and uses an interrupt to inform the server stub to read them. The server stub communicates the return values of the file operation in a similar way. Because interrupts have noticeable latency (§4.4.2), we support a polling mode for high-performance applications such as netmap. In this mode, the client and server stubs both poll the shared page for 200μs before they go to sleep to wait for interrupts. The polling period is chosen empirically and is not currently optimized. Moreover, for asynchronous

notifications (§3.1), the server stub uses similar techniques to send a message to the client stub, e.g., when the keyboard is pressed.

The client stub uses a grant table to declare the legitimate memory operations to the hypervisor (§4.2.1). The grant table is a single memory page shared between the guest VM and the hypervisor. After storing the operations in the table, the client stub generates a grant reference number and forwards it to the server stub along with the file operation. The server stub 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 server stub 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 §3.1, 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

Type	Total LoC	Platform	Component	LoC
Generic	6833	Linux	stub modules:	
			- client	1553
			- server	1950
			- shared	378
			Linux kernel function wrappers	198
			Virtual PCI module	285
			- Supporting kernel code	50
		FreeBSD	FreeBSD client stub:	
			- New code (approx.)	451
			- From Linux stub modules (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 4.2 : Paradice code breakdown.

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 stub modules 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 client stub with about 450 new LoC and about 760 LoC from the Linux client stub 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 client stub, since these addresses are needed by the Linux device driver and by the Paradice hypervisor API.

Concurrency Support: As mentioned in §4.1.3, 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 [118]).

4.3.2 Hypervisor-Assisted Memory Operations

The hypervisor supports the two types of memory operations needed by 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 an arbitrary 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 server stub, the client stub 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 §4.1, we employ function wrappers 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 server stub 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 function wrapper will invoke the appropriate hypervisor API when

executed in the context of a marked thread.

4.3.3 Isolation between Guest VMs

Fault Isolation: As described in §4.2.1, 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 [123] and its C language frontend, Clang [124]. 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 [125] 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 to the GPU (§4.2.2). 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 studied 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.

4.4 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-local 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.

4.4.1 Non-Performance Properties

Before we evaluate the performance of Paradice, we demonstrate that Paradice achieves the desired properties mentioned in §4. First, supporting new I/O devices with Paradice is easy and only requires developing small device info modules for new device classes (Table 4.2). These modules typically took us only a few person-hours each to implement. On the other hand, 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 (§4.1.3). 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 (§4.3.1). 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.

4.4.2 Performance

In this section, we quantify the overhead and performance of Paradice. We compare the performance of Paradice with the local performance (i.e., when an application is executed natively on the same hardware), as well as with the performance of direct device assignment (i.e., 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 in the driver VM.

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 inter-VM interrupts for communication, use Linux guest VM and Linux driver VM, and do not employ device data isolation. Other configurations will be explicitly mentioned.

Overhead Characterization

There are two potential sources of overhead that can degrade Paradice's performance compared to local: 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 server stub 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 local 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 local; hence, 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 large amounts of GPU memory.

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 [118]. 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 4.4 shows that Paradice can achieve a transmit rate close to that of local 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 local. However, with interrupts, a minimum batch size of around 30 is required. The figures also shows that both

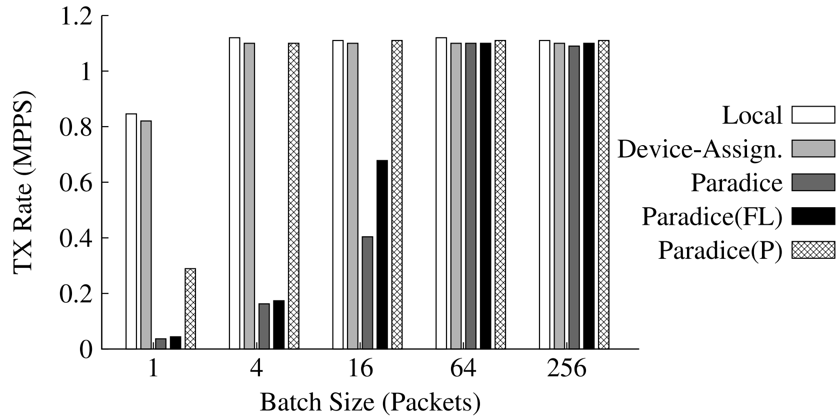


Figure 4.4 : netmap transmit rate with 64 byte packets. (FL) indicates FreeBSD guest VM using a Linux driver VM. (P) indicates the use of polling mode in Paradice.

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 local for different packet sizes.

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* [126], *OpenArena* [127], and *Nexuiz* [128], which are all widely used for GPU performance evaluation [129]. For all games, we use the Phoronix Test Suite engine [130] (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

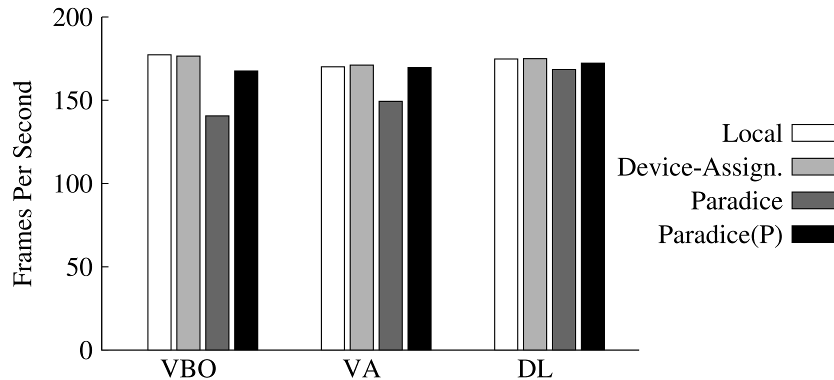
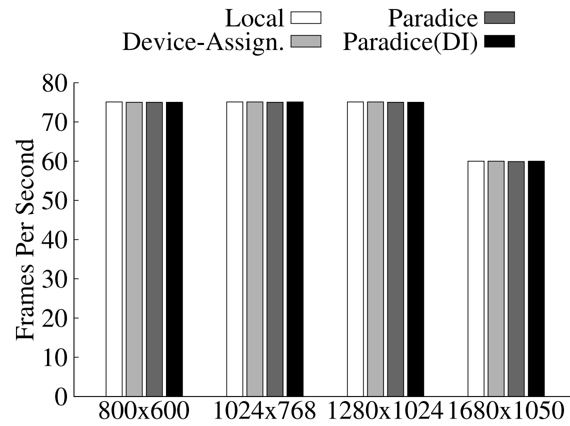


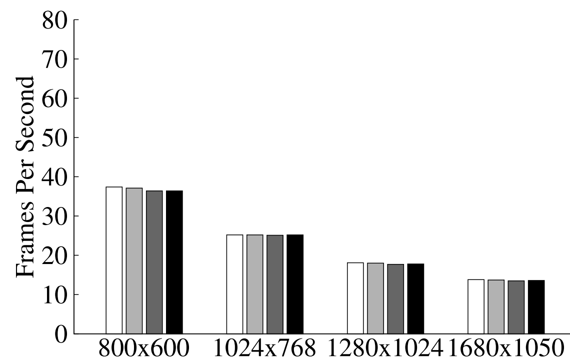
Figure 4.5 : OpenGL benchmarks FPS. VBO, VA, and DL stand for Vertex Buffer Objects, Vertex Arrays, and Display Lists. (P) indicates the use of polling mode in Paradice.

different OpenGL API including Vertex Buffer Objects, Vertex Arrays, and Display Lists [131, 132] 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.

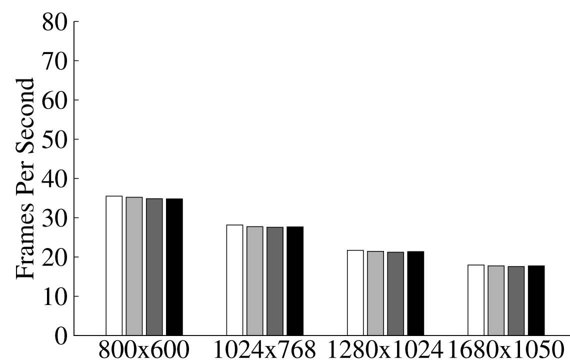
Figures 4.5 and 4.6 show the results. There are four important observations. First, Paradice achieves close performance to local and device assignment for various benchmarks, including OpenGL benchmarks and 3D HD games. Second, Paradice (with interrupts) achieves relatively better performance for more 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-local performance for all the benchmarks. Finally, data isolation has no noticeable impact on performance.



(a) Tremulous



(b) OpenArena



(c) Nexuiz

Figure 4.6 : 3D HD games FPS at different resolutions. (DI) indicates the use device data isolation in Paradice.

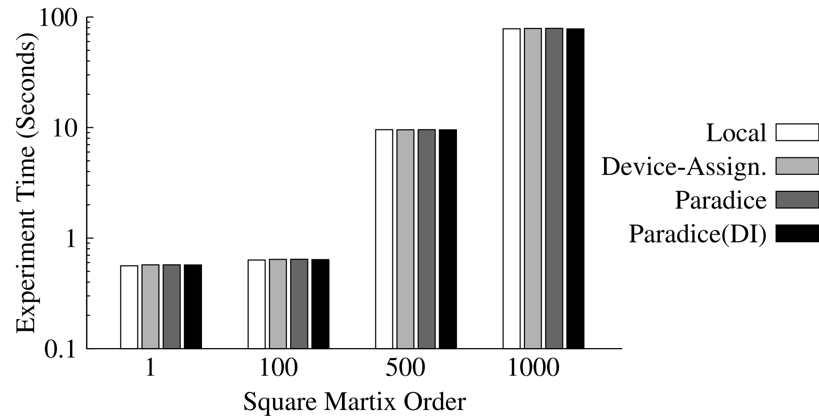


Figure 4.7 : OpenCL matrix multiplication benchmark results. The x-axis shows the order of the input square matrices. (DI) indicates the use device data isolation in Paradice.

GPU for Computation

We evaluate the performance of OpenCL applications. We use the Gallium Compute implementation of OpenCL [133] 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 4.7 shows the results. It shows that Paradice performance is almost identical to local and device assignment. Moreover, the figure shows that device data isolation has no 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,

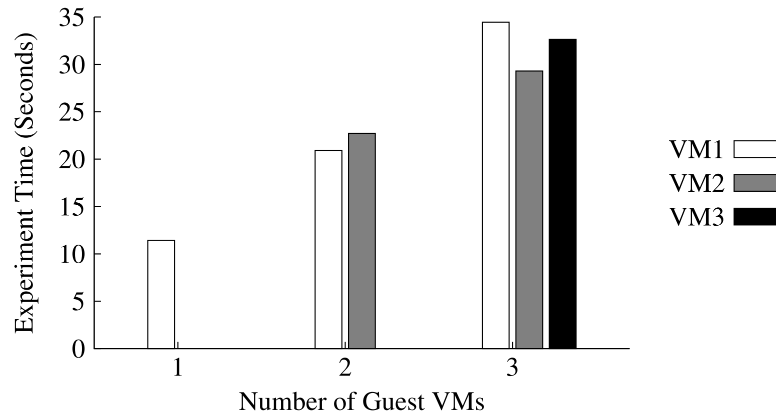


Figure 4.8 : Guest VMs concurrently running the OpenCL matrix multiplication benchmark on a GPU shared through Paradice. The matrix order in this experiment is 500.

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 4.8 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.

Mouse

We measure the latency of the mouse. Our results show that local, device 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 [134]. Much of the latency in Paradice comes from the communication between the client and server stubs, 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 a measurement is 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.

Camera & Speaker

We run the GUVCview [135] camera applications in the three highest video resolutions supported by our test camera for MJPG output: 1280×720, 1600×896, and 1920×1080. For all the resolutions, local, device assignment, and Paradice achieve about 29.5 FPS.

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

4.5 Discussions

4.5.1 Performance Isolation & Correctness Guarantees

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 [136]. 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 [137].

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. In §7, we present library drivers that will protect the device driver against such attacks by reducing the device driver TCB and attack surface. Another 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.

4.5.2 Other Devices, Hypervisors, and OSes

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.

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.

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

Chapter 5

I/O Sharing between Mobile Systems

A user nowadays owns a variety of mobile systems, including smartphones, tablets, smart glasses, and smart watches, each equipped with a plethora of I/O devices, such as cameras, speakers, microphones, sensors, and cellular modems. There are many interesting use cases for allowing an application running on one mobile system to access I/O devices on another system, for three fundamental reasons. (*i*) Mobile systems can be in different physical locations or orientations. For example, one can control a smartphone's high-resolution camera from a tablet to more easily capture a self-portrait. (*ii*) Mobile systems can serve different users. For example, one can a play music for another user if one's smartphone can access the other system's speaker. (*iii*) Certain mobile systems have unique I/O devices due to their distinct form factors and targeted use cases. For example, a user can make a phone call from her tablet using the modem and SIM card in her smartphone.

Unsurprisingly, solutions exist for sharing various I/O devices, e.g., camera [2], speaker [3], and modem (for messaging) [4]. However, these solutions have three fundamental limitations. First, they do not support unmodified applications. For example, IP Webcam [2] and MightyText [4] do not allow existing applications to use a camera or modem remotely; they only support their own custom applications. Second, they do not expose all the functionality of an I/O device for sharing. For example, IP Webcam does not support remote configuration of all camera parameters, such as resolution. MightyText supports SMS and MMS from another device, but

not phone calls. Finally, existing solutions are I/O class-specific, requiring significant engineering effort to support new I/O devices. For example, IP Webcam [2] can share the camera, but not the modem or sensors.

In this chapter, we introduce Rio (*R*emote *I/O*) [138], an I/O sharing solution for mobile systems based on the device file boundary and show that it overcomes all three aforementioned limitations. To do this, Rio creates a virtual device file inside the I/O client (the mobile systems wishing to use the I/O device remotely), intercepts the file operations and forwards them to the I/O server (the mobile system sharing its I/O device) to be executed. The device file boundary supports I/O sharing for unmodified applications, as it is transparent to the application layer. It also exposes the full functionality of each I/O device to other mobile systems by allowing processes in one system to directly communicate with the device drivers in another. Most importantly, it reduces the engineering effort to support various classes of I/O devices.

The design and implementation of Rio must address the following fundamental challenges. (*i*) A process may issue operations on a device file that require the driver to operate on the process memory. With I/O sharing, however, the process and the driver reside in two different mobile systems with separate physical memories. In Rio, we support cross-system memory mapping using a distributed shared memory (DSM) design that supports access to shared pages by the process, the driver, and the I/O device (through DMA). We also support cross-system memory copying with collaboration from both systems. (*ii*) Mobile systems typically communicate through a wireless connection that has a high round-trip latency compared to the latency between a process and driver within the same system. To address this challenge, we reduce the number of round trips between the systems due to file operations, memory operations, or DSM coherence messages. (*iii*) The connection between mobile systems

can break at any time due to mobility or reliability issues. This can cause undesirable side-effects in the OSes of all involved systems. We address this problem by properly cleaning up the residuals of a remote I/O connection upon disconnection, switching to a local I/O device of the same class, if present, or otherwise returning appropriate error messages to the applications.

We present a prototype implementation of Rio for Android systems. Our implementation supports four important I/O classes: camera, audio devices such as speaker and microphone, sensors such as accelerometer, and cellular modem (for phone calls and SMS). It consists of about 7100 Lines of Code (LoC), of which less than 500 are specific to I/O classes. It also supports I/O sharing between heterogeneous mobile systems, including tablets and smartphones. See [139] for a video demo of Rio.

We evaluate Rio on Galaxy Nexus smartphones and show that it supports existing applications, allows remote access to all I/O device functionality, requires low engineering effort to support different I/O devices, and achieves performance close to that of local I/O for audio devices, sensors, and modem, but suffers noticeable performance degradation for camera sharing due to Wi-Fi throughput limitation in our setup. With emerging wireless standards supporting much higher throughput, we posit that this degradation is likely to go away in the near future. In addition, we report the throughput and power consumption for using remote I/O devices with Rio and show that throughput highly depends on the I/O device class, and that power consumption is noticeably higher than that of local access.

5.1 Use Cases

We envision two categories of use cases for Rio. The first category, already tested with Rio, consists of those that simply combine Rio with existing application (§5.1.1).

This category is the focus of this paper. However, we envision applications developed specifically with I/O sharing in mind. Obviously, such applications do not exist today because I/O sharing is not commonly available. §5.1.2 presents some of such use cases.

5.1.1 Use Cases Demonstrated with Rio

Multi-system photography: With Rio, one can use a camera application on one mobile system to take a photo with the camera on another system. This capability can be handy in various scenarios, especially when taking self-portraits, as it decouples the camera hardware from the camera viewfinder, capture button, and settings. Several existing applications try to assist the user in taking self-portraits using voice recognition, audio guidance, or face detection [140]. However, Rio has the advantage in that the user can see the camera viewfinder up close, comfortably configure the camera settings, and press the capture button whenever ready, even if the physical camera is dozens of feet away. Alternatively, one can use the front camera, which typically has lower quality than the rear-facing one.

Multi-system gaming: Many mobile games require the user to physically maneuver the mobile system for control. Tablets provide a large screen for gaming but are bulky to physically maneuver. Moreover, maneuvers like tilting make it hard for the user to concentrate on the content of the display. With Rio, a second mobile system, e.g., a smartphone, can be used for physical maneuvers while the tablet running the game remains stationary.

One SIM card, many systems: Despite many efforts [141], users are still tied to a single SIM card for phone calls or SMS, mainly because the SIM card is associated with a unique number. With Rio, the user can make and receive phone calls and SMS from any of her mobile systems using the modem and SIM card in her smartphone.

For example, if a user forgets her smartphone at home, she can still receive phone calls on her tablet at work.

Music sharing: A user might want to allow a friend to listen to some music via a music subscription application on her smartphone. With Rio, the user can simply play the music on her friend's smartphone speaker.

Multi-system video conferencing: When a user is video conferencing on her tablet, she can use the speaker or microphone on her smartphone and move them closer to her mouth for better audio quality in a noisy environment. Or she can use the camera on her smart glasses as an external camera for the tablet to provide a different viewpoint.

5.1.2 Future Use Cases of Rio

With Rio, new applications can be developed to use the I/O devices available on another system.

Multi-user gaming: The multi-system gaming use case explained in the previous subsection combined with modifications to the application can enable novel forms of multi-user gaming across mobile systems. For example, two players can use their smartphones to wirelessly control a racing game on a single tablet in front of them. The smartphones' displays can even show in-game context menus or game controller keys, similar to those on game consoles, providing a familiar and traditional gaming experience for users.

Music sharing: If supported by the system software (e.g., audio service process in Android (§5.6)), a user can play the same music on her and her friend's smartphones simultaneously. With proper application support, the user can even play two different sound tracks on these two systems at the same time, much like multi-zone stereo receivers.

Multi-view video conferencing: A video conferencing application can be extended to show side-by-side video streams from the smart glasses and the tablet simultaneously. In this fashion, the user can not only share a video stream of her face with her friend, but she can also share a stream of the scene in front of her at the same time.

Multi-camera photography: Using the cameras on multiple mobile systems, one can realize various computational photography techniques [142]. For example, one can use the cameras of her smartphone and smart glasses simultaneously to capture photos with different exposure times in order to remove motion blur [143], or to increase the temporal/spatial resolution of video by interleaving/merging frames from both cameras [144, 145]. One can even use the smartphone as an external flash for the smart glasses camera.

5.2 Design

We describe the design of Rio, including its architecture and the guarantees it provides to mobile systems using it.

Rio is based on remote access to I/O devices at the device file boundary. That is, it intercepts communications at the device file boundary in one mobile system, i.e., the client, and forwards them to the other system, i.e., the server, to be executed by the rest of the I/O stack. Figure 5.1 shows the design of Rio, which is a specific case of Figure 3.1, where the communication channel is a wireless link.

As mentioned in §3.1, some file operations, such as `read`, `write`, `ioctl`, and `mmap`, require the driver to operate on the process memory. `mmap` requires the driver to map some memory pages into the process address space. For this, Rio uses a DSM design that supports access to shared pages by the client process as well as the server driver and the device (§5.3.1). The other three file operations often ask the driver to copy

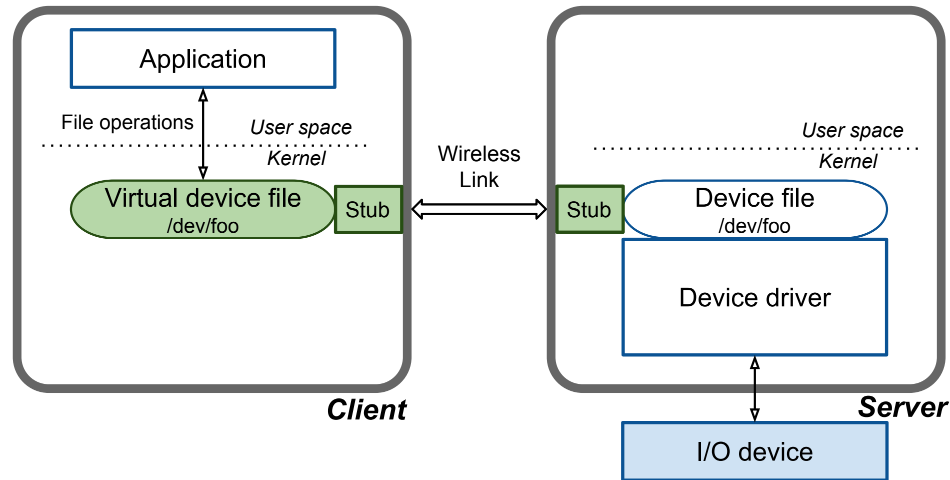


Figure 5.1 : Rio uses the device file boundary for remote access. Hence, it splits the I/O stack between two mobile systems.

data to or from the process memory. The server stub intercepts the driver's requests for these copies and services them in collaboration with the client stub (§5.3.2).

5.2.1 Guarantees

Using I/O remotely at the device file boundary impacts three expected behaviors of file operations: reliability of connection, latency, and trust model. That is, remote I/O introduces the possibility of disconnection between the process and the driver, adds significant latency to each file operation due to wireless round trips, and allows processes and drivers in different trust domains to communicate. Therefore, Rio provides the following guarantees for the client and server.

First, to avoid undesirable side-effects in the client and server resulting from an unexpected disconnection, Rio triggers a cleanup in both systems upon detecting a disconnection. Rio guarantees the server that a disconnection behaves similar to killing a local process that uses the I/O device. Rio also guarantees the client that

it will transparently switch the application to use a local I/O device of the same class after the disconnection, if possible; otherwise, Rio will return appropriate error messages to the application (§5.5).

Second, Rio reduces the number of round trips due to file or memory operations and DSM coherence messages (§5.4) in order to reduce latency and improve performance. Moreover, it guarantees that the additional latency of file operations only impacts the performance, *not the correctness*, of I/O devices. Rio can provide this guarantee because most file operations do not have a time-out threshold, but simply block until the device driver handles them. `poll` is the only file operations for which a time-out can be set by the process. In §5.4.4, we explain that `poll` operations used in Android for I/O devices we currently support do not use the `poll` time-out mechanism. We also explain how Rio can deal with the `poll` time-out, if used.

Finally, processes typically trust the device drivers with which they interact through the device file interface, and drivers are vulnerable to attacks by processes [116]. To maintain the same trust and security model in one mobile system, we intend the current design of Rio to be used among trusted mobile systems only. However, Rio can adopt the fault isolation technique In Paradice (§4.2.1 in order to protect the client from an untrusted server. Moreover, it can adopt library drivers (§7) to improve the security of the server in face of untrusted clients.

5.3 Cross-System Memory Operations

In order to handle file operations, the device driver often needs to operate on the process memory by executing memory operations. However, these operations pose a challenge for Rio because the process and the driver reside in different mobile systems with separate physical memories. In this section, we present our solutions.

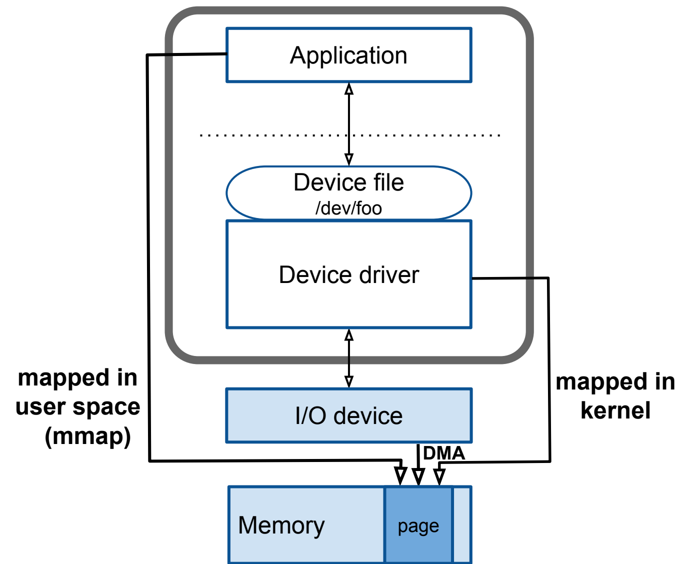
5.3.1 Cross-System Memory Map

Cross-system memory map in Rio supports the `map_page` memory operation across two mobile systems using Distributed Shared Memory (DSM) [146–150] between them. At the core of Rio's DSM is a simple write-invalidate protocol, similar to [149]. The novelty of the DSM in Rio is that it can support access to the distributed shared memory pages not only by a process, but also by kernel code, such as the driver, and also by the device (through DMA).

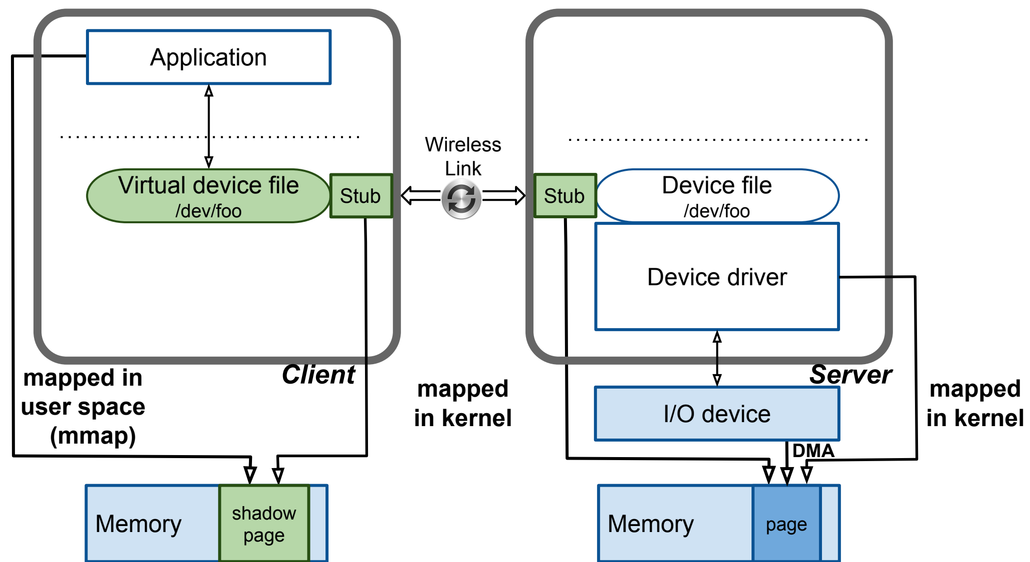
Figure 5.2 illustrates the cross-system memory map in Rio. When intercepting a `map_page` operation from the server driver, the server stub notifies the client stub, which then allocates a shadow memory page in the client (corresponding to the actual memory page in the server) and maps that shadow page into the client process address space. The DSM modules in these two stubs guarantee that the process, the driver, and the device have consistent views of these pages. That is, updates to both the actual and shadow pages are consistently available to the other mobile system.

We choose a write-invalidate protocol in Rio's DSM for efficiency. Compared to update protocols that proactively propagate the updates to other systems [148], invalidate protocols do so only when the updated data is needed on the other system. This minimizes the amount of data transmitted between the client and server, and therefore minimizes the resource consumption, e.g., energy, in both systems. With the invalidate protocol, each memory page can be in one of three possible states: read-write, read-only, or invalid. Although the invalidate protocol is the default in Rio, we can also use an update protocol if it offers performance benefits.

We use 4 KB pages (small pages) as the coherence unit because it is the unit of the `map_page` memory operation, meaning the driver can map memory as small as a single small page into the process address space. When many pages are updated



(a)



(b)

Figure 5.2 : (a) Memory map for a local I/O device. (b) Cross-system memory map in Rio.

together, we batch them altogether to improve performance (§5.4.3).

To manage a client process's access to the (shadow) page, we use the page table permission bits, similar to some existing DSM solutions [146]. When the shadow page is in the read-write state, the page table grants the process full access permissions to the page, and all of the process's read and write instructions execute natively with no extra overhead. In the read-only state, only write to these pages cause page faults, while both read and write cause page faults in the invalid state. Upon a page fault, the client stub triggers appropriate coherence messages. For a read fault, it fetches the page from the server. For a write fault, it first fetches the page if in invalid state, and then sends an invalidation message to the server.

To manage the server driver's access to the page, we use the page table permission bits for kernel memory since the driver operates in the kernel. However, unlike process memory that uses small 4 KB pages, certain regions of kernel memory, e.g., the identity-mapped region in Linux, use larger pages, e.g., 1 MB pages in ARM [151], for less contention in the Translation Lookaside Buffer (TLB). When the driver requests to map a portion of a large kernel page into the process address space, the server stub dynamically breaks the large kernel page into multiple small ones by destroying the old page table entries and creating new ones, similar to the technique used in K2 [152]. With this technique, the server stub can enforce different protection modes against kernel access at the granularity of small pages, rather than large pages. To minimize the side-effects of using small pages in the kernel, e.g., higher TLB contention, the server stub immediately stitches the small pages back into a single large page when the pages are unmapped by the process.

To manage the server I/O device's access to the page through DMA, the server stub maintains an explicit state variable for each page, intercepts the driver's DMA

request to the device, updates the state variable, and triggers appropriate coherence messages. Note that it is not possible to use page table permission bits for a device's access to the page since devices' DMA operations bypass the page tables.

Rio provides sequential consistency. This is possible for two reasons. First, the DSM module triggers coherence messages immediately upon page faults and DMA completion, and maintains the order of these messages in each system. Second, processes and drivers use file operations to coordinate their own access to mapped pages.

5.3.2 Cross-System Copy

Cross-system memory copy in Rio supports `copy_from_user` and `copy_to_user` memory operations between two mobile systems. We achieve this through collaboration between the server and client stubs. When intercepting a `copy_from_user` or `copy_to_user` operation from the driver, the server stub sends a request back to the client stub to perform the operation. In the case of `copy_from_user`, the client stub copies the data from the process memory and sends it back to the server stub, which copies it into the kernel buffer determined by the driver. In the case of `copy_to_user`, the server stub copies and sends the data from the kernel buffer to the client stub, which then copies it to the corresponding process memory buffer. Figure 5.3(b) illustrates the cross-system copy for a typical `ioctl` file operation.

When handling a file operation, the driver may execute several memory copy operations, causing as many round trips between the mobile systems because the server stub has to send a separate request per copy operation. Large numbers of round trips can degrade the I/O performance significantly. In §5.4.1, we explain how we reduce these round trips to only one per file operation by pre-copying the copy data in the client stub for `copy_from_user` operations and by batching the data of

`copy_to_user` operations in the server stub.

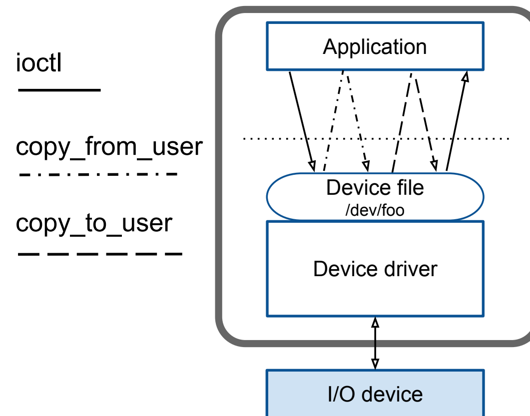
5.4 Mitigating High Latency

The connection between the client and the server typically has high latency. For example, Wi-Fi and Bluetooth have about 1-2 ms round-trip latency at best [153], which is significantly higher than the few microseconds of latency typical of native communications between a process and device driver (i.e., syscalls). In this section, we discuss the challenges resulting from such high latency and present our solutions to reduce its effect on I/O performance by reducing the number of round trips due to copy memory operations, file operations, and DSM coherence messages.

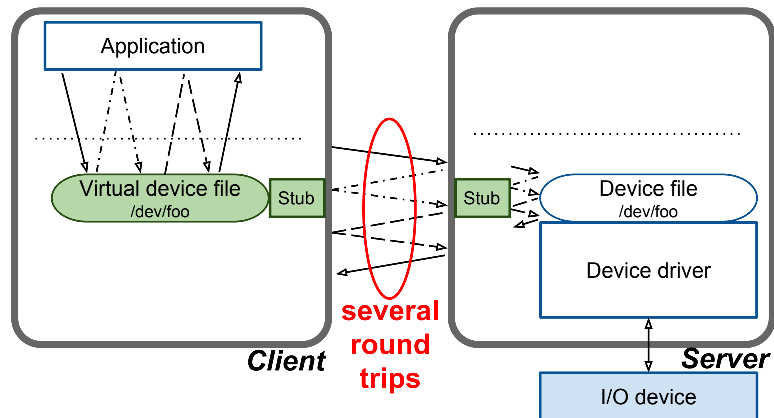
5.4.1 Round Trips due to Copies

Round trips due to `copy_from_user` and `copy_to_user` memory operations present a serious challenge to Rio's performance since a single file operation may execute several copy memory operations. For example, a single `ioctl` in Linux's PCM audio driver may execute four `copy_from_user` operations. To solve this problem, we use the following two techniques. (*i*) In the client stub, we determine and pre-copy all the data needed by the server driver and transmit it together with the file operation. With this technique, all `copy_from_user` requests from the driver are serviced locally inside the server. (*ii*) In the server stub, we buffer and batch all data that the driver intends to copy to the process memory and transmit it to the client along with the return value of the file operation. With this technique, all `copy_to_user` operations can be executed locally in the client. Figure 5.3(c) illustrates these techniques.

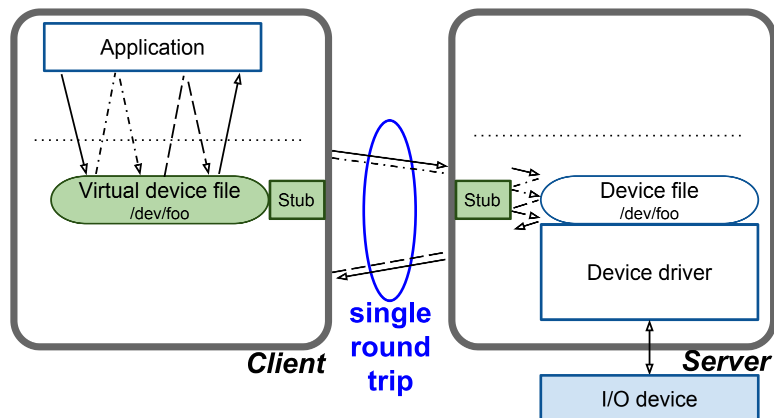
Pre-copying the data for driver `copy_from_user` requests requires the client stub module to determine *in advance* the addresses and sizes of the process memory data



(a) Local I/O device.



(b) Remote I/O device with unoptimized Rio (§5.3.2).



(c) Remote I/O device with optimized Rio (§5.4.1).

Figure 5.3 : Typical execution of an `ioctl` file operation. Optimized Rio reduces the number of round trips.

buffers needed by the driver. This is trivial for the `write` file operation, as this information is embedded in its input arguments. However, doing so for `ioctl` is non-trivial as the `ioctl` input arguments are not always descriptive enough. Many well-written drivers embed information about some simple driver memory operations in one of the `ioctl` input arguments, i.e., the `ioctl` command number. In such cases, we parse the command number in the client stub to infer the memory operations, similar to Paradice §4.2.1. There are cases, however, that the command number does not contain all necessary information. For these cases, we use the static analysis tool explained in §4.2.1 that analyses the driver's source code to extract a small part of the driver code, which can then be executed either offline or at runtime in the client stub to infer the parameters of driver memory operations. Finally, to maintain a consistent view of the process memory for the driver, Rio updates the pre-copied data in the server stub upon buffering the `copy_to_user` data if the memory locations overlap.

5.4.2 Round Trips due to File Operations

File operations are executed synchronously by each process thread, and therefore, each file operation needs one round trip. To optimize performance, the process should issue the minimum number of file operations possible. Changing the number of file operations is not always possible or may require substantial changes to the process source code, e.g., the I/O service code in Android (§5.6), which is against Rio's goal of reducing engineering effort. However, minimal changes to the process code can occasionally result in noticeable reductions in file operation issuance, justifying the engineering effort. §5.6.3 explains one example for Android audio devices.

5.4.3 Round Trips due to DSM Coherence

As mentioned in §5.3.1, we use 4 KB pages as the DSM coherence unit in Rio. However, when there are updates to several pages at once, such a relatively small coherence unit causes several round trips for all data to be transferred. In such cases, transmitting all updated pages together in a single round trip is much more efficient.

5.4.4 Dealing with Poll Time-outs

`poll` is the only file operations for which the issuing process can set a time-out. Since Rio adds noticeable latency to each file operation, it can break the semantics of `poll` if a relatively small time-out threshold is used. So far in our Android implementation, all I/O classes we support do not use `poll` time-out (i.e., the process either blocks indefinitely until the event is ready or uses non-blocking `poll`s). If `poll` is used with a time-out, the time-out value should to be adjusted for remote I/O devices. This can be done inside the kernel handler for `poll`-related syscalls, such as `select`, completely transparent to the user space. Using the heartbeat round-trip time (§5.5), the client stub can provide a best estimate of the additional latency that the syscall handler needs to add to the requested time-out value. Processes typically rely on the kernel to enforce the requested `poll` time-out; therefore, this approach guarantees that the process function will not break in the face of high latency. In the unlikely case that the process uses an external timer to validate its requested time-out, the process must be modified to accommodate additional latency for remote I/O devices.

5.5 Handling Disconnections

The connection between the client and the server may be lost at any time due to mobility. If not handled properly, the disconnection can cause the following problems: render the driver unusable, block the client process indefinitely, or leak resources, e.g., memory, in the client and server OSes. When faced with a disconnection, the server and client stubs take the appropriate actions described below.

We use a time-out mechanism to detect a disconnection. At regular intervals, the client stub transmits heartbeat messages to the server stub, which immediately transmits back an acknowledgement. If the client stub does not receive the acknowledgement before a certain threshold, or the server does not hear from the client, they both trigger a disconnection event. We do not use the in-flight file operations as a heartbeat because file operations can take unpredictable amounts of time to complete in the driver. Determining the best heartbeat interval and time-out thresholds to achieve an acceptable trade-off between overhead and detection accuracy is part of future work.

For the server, network disconnection is equivalent to killing a local process that is communicating with the driver. Therefore, just as the OS cleans up the residuals of a killed process, the server stub cleans up the residuals of the disconnected client process. For each `mmap`ed area and each file descriptor, the server stub invokes the driver's `close_map` handler and `release` file operation handler respectively, in order for the driver to release the allocated resources. Finally, it releases its own bookkeeping data structures.

We take two actions in the client upon disconnection. First, we clean up the residuals of the disconnected remote I/O in the client stub, similar to the cleanup process in the server. Next, we try to make the disconnection as transparent to the

application as possible. If the client has a local I/O device of the same class, we transparently switch to that local I/O device after the disconnection. If no comparable I/O device is present, we return appropriate error messages supported by the API. These actions require class-specific developments, and §5.6.3 explains how we achieve this for sensors. Switching to local I/O is possible for three of the I/O classes we currently support, including camera, audio, and sensors such as accelerometer. For the modem, the disconnection means that a phone call will be dropped or not initiated, or that an SMS will not be sent; all behaviors are understandable by existing applications.

5.6 Android Implementation

We have implemented Rio for Android OS and ARM architecture. The implementation currently supports four classes of I/O devices: sensors (e.g., accelerometer), audio devices (e.g., microphone and speaker), camera, and modem (for phone calls and SMS). It consists of about 7100 LoC, fewer than 500 of which are I/O class-specific as shown in Table 5.1. We have tested the implementation on Galaxy Nexus smartphones running CyanogenMod 10.1 (Android 4.2.2) atop Linux kernel 3.0, and on a Samsung Galaxy Tab 10.1 tablet running CyanogenMod 10.1 (Android 4.2.2) with Linux kernel 3.1. The implementation can share I/O between systems of different form factors: we have demonstrated this for sharing sensors between a smartphone and a tablet. The implementation is open source [154].

Figure 5.4 shows the architecture of Rio inside an Android system. In Android, the application processes do not directly use the device files to interact with the driver. Instead, they communicate to a class-specific *I/O service process* through class-specific APIs. The I/O service process loads a *Hardware Abstraction Layer (HAL)* library in order to use the device file to interact with the device driver. Rio's device file

Type	Total LoC	Component	LoC
Generic	6618	Server stub	2801
		Client stub	1651
		Shared between stubs	647
		DSM	1192
		Supporting Linux kernel code	327
Class-specific	498	Camera:	
		- HAL	36
		- DMA	134
		Audio device	64
		Sensor	128
		Cellular modem	136

Table 5.1 : Rio code breakdown.

boundary lies below the I/O service processes, forwarding its file operations to the server. As we will explain in the rest of this section, we need small modifications to the HAL or I/O service process, but no modifications to the applications are needed.

5.6.1 Client & Server Stubs

The client and server stubs are the two main components of Rio and constitute a large portion of Rio's implementation. Each stub has three modules. The first module supports interactions with applications and device drivers. In the client stub, this module intercepts the file operations and packs their arguments into a data structure; in the server stub, it unpacks the arguments from the data structure and invokes the file operations of the device driver. This module is about 3000 LoC and is shared with the implementation of Paradice (§4). The second module implements the communication with the other stub by serializing data structures into packets and transmitting them to the other end. Finally, the third module implements Rio's

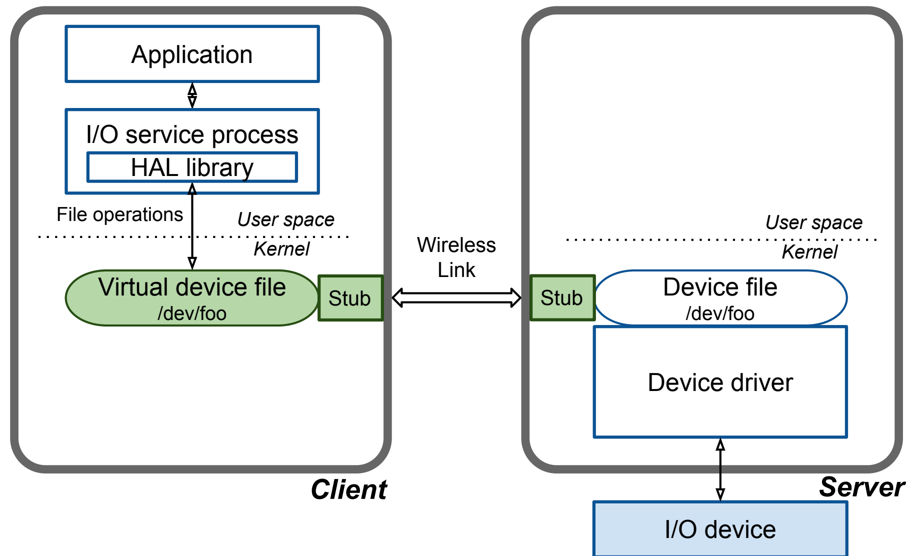


Figure 5.4 : Rio's architecture inside an Android system. Rio forwards to the server the file operations issued by the I/O service process through HAL. Rio supports unmodified applications but requires small changes to the class-specific I/O service process and/or HAL.

DSM, further explained in §5.6.2.

We use in-kernel TCP sockets for communication between the client and server stubs [155]. We use TCP to ensure that all the packets are successfully received, otherwise the device, driver, or the application might break.

To handle cross-system memory operations, the server stub intercepts the driver's kernel function calls for memory operations. This includes intercepting 7 kernel functions for `copy_to_user` and `copy_from_user` and 3 kernel functions for `map_page`. Using this technique allows us to support *unmodified drivers*.

5.6.2 DSM Module

Rio's DSM module is shared between the client and the server. It implements the logic of the DSM protocol, e.g., triggering and handling coherence messages. The DSM module is invoked in two cases: page faults and DMA. We instrument the kernel fault handler to invoke the DSM module when there is a page fault. Additionally, the DSM module must handle device DMA to DSM-protected pages. We monitor the driver's DMA requests to the device and invoke the DSM module upon DMA completion.

To monitor the driver's DMA requests to devices, we instrument the corresponding kernel functions. These functions are typically I/O bus-specific and will apply to all I/O devices using that I/O bus. Specialized instrumentation is needed if the driver uses non-standard interfaces. For example, the camera on the TI OMAP4 SoC inside Galaxy Nexus smartphones uses custom messages between the driver and the Imaging Subsystem (ISS) component, where the camera hardware resides [156]. We instrumented the driver responsible for communications with the ISS to monitor the DMA requests, only with 134 LoC.

When we receive a DMA completion notification for a memory buffer, we may use a DSM update protocol to immediately push the updated buffers to the client, an optional optimization. Moreover, we update the whole buffer in one round trip. These optimizations can improve performance as they minimize the number of round trips between mobile systems (§5.4.3); as such, we used them for camera frames.

As described in §5.3.1, certain regions of the kernel's address space, namely the identity-mapped region, use large 1 MB pages known as *Sections* in the ARM architecture. To split these 1 MB Sections into smaller 4 KB pages for use with our DSM module, we first walk the existing page tables to obtain a reference to the Section's first-level descriptor (a PGD entry). We then allocate a single new page that holds

512 second-level page table entries (PTEs), one for each page; altogether, these 512 PTEs reference two 1 MB Sections of virtual memory. We populate each second-level PTE with the correct page frame number and permission bits from the original Section. Finally, we change the first-level descriptor entry to point to our new table of second-level PTEs and flush the corresponding cache and TLB entries.

Support for Buffer Sharing using Android ION

Android uses the ION memory management framework to allocate and share memory buffers for multimedia applications, such as those using the GPU, camera, and audio [157]. The sharing of ION buffers creates unique challenges for Rio, as demonstrated in the following example.

The camera HAL allocates buffers using ION and passes the ION buffer handles to the kernel driver, which translates them to the physical addresses of these buffers and asks the camera to copy new frames to them. Once the frames are written, the HAL is notified and forwards the ION buffer handle to the graphics framework for rendering. Now, imagine using a remote camera in Rio. The same ION buffer handles used by the camera HAL in the client need to be used by both the server kernel driver and the client graphics framework, since the camera frames from the server are rendered on the client display.

To solve this problem, we provide support for global ION buffers that can be used both inside the client and the server. We achieve this by allocating an ION buffer in the server with similar properties (e.g., size) to the one allocated in the client; we use the DSM module to keep the two buffers coherent.

5.6.3 Class-Specific Developments

Most of Rio's implementation is I/O class-agnostic; our current implementation only requires under 500 class-specific LoC.

Resolving naming conflicts: In case the client has an I/O device of the same class that uses device files with similar names as those used in the server, the virtual device file must assume a different name (e.g., `/dev/foo_rio` vs. `/dev/foo` in Figure 5.1). However, the device file names are typically hard-coded in the HAL, necessitating small modifications to use a renamed virtual device file for remote I/O.

Optimizing performance: As discussed in §5.4.2, sometimes small changes to the I/O service code and HAL can boost the remote I/O performance significantly by reducing the number of file operations. For example, the audio HAL exchanges buffered audio segments with the driver using `ioctl`s. The HAL determines the size of the audio segment per `ioctl`. For local devices (with very low latency), these buffered segments can contain as low as 3 ms of audio, less than a single round-trip time in Rio. Therefore, we modify the HAL to use larger buffering segments for remote audio devices. Although this slightly increases the audio latency, it significantly improves the audio rate for remote devices. §5.7 provides measurements to quantify this trade-off. This modification only required about 30 LoC.

Support for hot-plugging and disconnection: Remote I/O devices can come and go at any time; in this sense, they behave similarly to hot-plugging/removal of local I/O devices. Small changes to the I/O service layer may be required to support hot-plugging and disconnection of remote I/O devices. For example, the Android sensor service layer opens the sensor device files (through the HAL) in the phone initialization process and only uses these file descriptors to read the sensor values. To support hot-plugging remote sensors, we modified the sensor service layer to open the

virtual device files and use their file descriptors too when remote sensors are present. Upon disconnection, we switch back to using local sensors to provide application transparency.

Avoiding duplicate I/O initialization: Some HAL libraries, including sensor and cellular modem's, perform initialization of the I/O device upon system boot. However, since the I/O device is already initialized in the server, the client HAL should not attempt to initialize the I/O device. Not only this can break the I/O device, it can also break the HAL because the server device driver rejects initialization-based file operations. Therefore, the HAL must be modified in order to avoid initializing a device twice. Achieving this was trivial for the open-source sensor HAL. However, since the modem's HAL is not open-source, we had to employ a workaround that uses a second SIM card in the client to initialize its modem's HAL. It is possible to develop a small extension to the modem open-source kernel driver in order to fake the presence of the SIM card and allow the client HAL to initialize itself without a second SIM card.

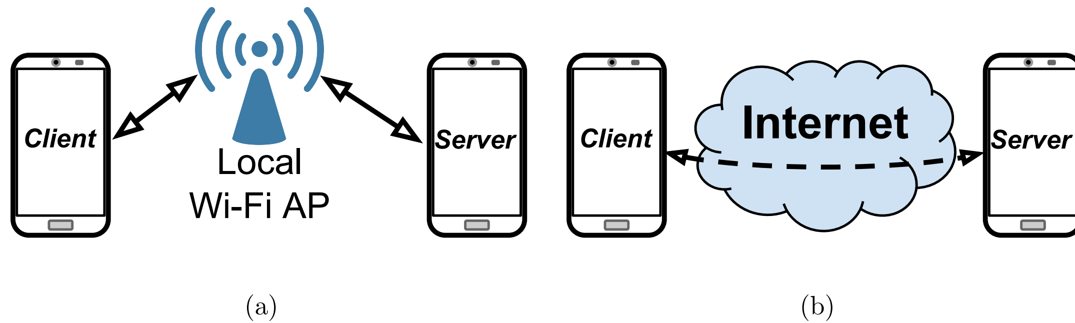
Sharing modem for phone calls: Through the device file interface, Rio can initiate and receive phone calls. However, it requires further development to relay the incoming and outgoing audio for the phone call. A phone call on Android works as follows. The modem HAL uses the modem device file to initiate a call, or to receive one. Once the call is connected, with instructions from the HAL, the modem uses the speaker and microphone for incoming and outgoing audio. The modem directly uses the speaker and microphone; therefore, the audio cannot be automatically supported by Rio's use of the modem device file. To overcome this, we leveraged Rio's ability to share audio devices in order to relay the audio. There are two audio streams to be relayed. The audio from the user (who is using the client device) to the target

phone (on the other side of the phone call) and the audio from the target phone to the client. For the former stream, we record the audio on the client using the client's own microphone and play it back on the server's speaker through Rio. The modem on the server then picks up the audio played on the speaker and transmits it to the target phone. For the latter stream, we record the incoming audio data on the server using server's microphone remotely from the client using Rio, and then play it back on the client's own speaker for the user. We used CyanogenMod 10.2 for relaying the audio for phone calls since CyanogenMod 10.1 does not support capturing audio during a phone call.

Currently, we can only support one audio stream at a time. The main reason for this is that supporting both streams will result in the user at the client hearing herself since the audio played on the server's speaker will be picked up by the microphone. While we currently have to manually arbitrate between the two streams, it is possible to support automatic arbitration by measuring the audio intensity on the two streams.

5.6.4 Sharing between Heterogeneous Systems

Because the device file boundary is common to all Android systems, Rio's design readily supports sharing between heterogeneous systems, e.g., between a smartphone and a tablet. However, the implementation has to properly deal with the HAL library of a shared I/O device because it may be specific to the I/O device or to the SoC used in the server. Our solution is to port the HAL library used in the server to the client. Such a port is easy for two reasons. First, Android's interface to the HAL for each I/O class is the same across Android systems of different form factors. Second, all Android systems use the Linux kernel and are mostly shipped with ARM processors. For example, we managed to port the Galaxy Nexus smartphone sensors HAL library



(a) (b)

Figure 5.5 : We use two different connections in our evaluation. In (a), the phones are connected to the same AP. This connection represents that used between mobile systems that are close to each other. In (b), the phones are connected over the Internet. This connection represents that used between mobile systems at different geographical locations.

to the Samsung Galaxy Tab 10.1 tablet by compiling the smartphone's HAL inside the tablet's source tree.

5.7 Evaluation

We evaluate Rio and show that it supports legacy applications, allows access to all I/O device functionality, requires low engineering effort to support different I/O devices, and achieves performance close to that of local I/O for audio devices, sensors, and modem, but exhibits performance drops for the camera due to network throughput limitations. We further discuss that future wireless standards will eliminate this performance issue. In addition, we report the throughput and power consumption for using remote I/O devices with Rio and show that throughput highly depends on the I/O device class, and that power consumption is noticeably higher than that of local devices. Finally, we demonstrate Rio's ability to handle disconnections.

All experiments are performed on two Galaxy Nexus smartphones. We use two connections of differing latencies for the experiments. The first connection (Figure 5.5(a)) is over a wireless LAN between mobile systems that are close to each other, e.g., both in the same room. We connect both phones to the same Wi-Fi access point. This connection has a latency with median, average, and standard deviation of 4 ms, 8.5 ms, and 16.3 ms, and a throughput of 21.9 Mbps. The second connection (Figure 5.5(b)) is between mobile systems at different geographical locations, one at home and one 20 miles away at work. We connect these two phones through the Internet using external IPs from commodity Internet Providers. This connection has a latency with median, average, and standard deviation of 55.2 ms 57 ms, and 20.9 ms, and a throughput of 1.2 Mbps. All reported results use the first LAN connection, unless otherwise stated.

5.7.1 Non-Performance Properties

First, Rio supports existing unmodified applications. We have tested Rio with both stock and third-party applications using different classes of I/O devices.

Second, unlike existing solutions, Rio exposes all functionality of remote I/O devices to the client. For example, the client system can configure every camera parameter, such as resolution, zoom, and white balance. Similarly, an application can configure the speaker with different equalizer effects.

Third, supporting new I/O devices in Rio requires low engineering effort. As shown in Table 5.1, we only needed 128, 64, 170, and 136 LoC to support sensors, audio devices (both speaker and microphone), camera, and the modem (for phone calls and SMS), respectively.

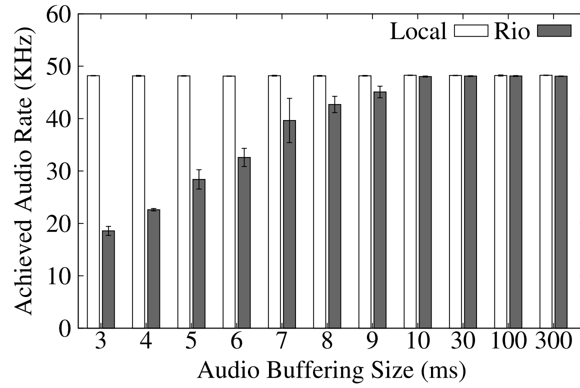
5.7.2 Performance

In this subsection, we measure the performance of different I/O classes in Rio and compare them to local performance. Unless otherwise stated, we repeat each experiment three times and report the average and standard deviation of measurements. The phones are rebooted after each experiment.

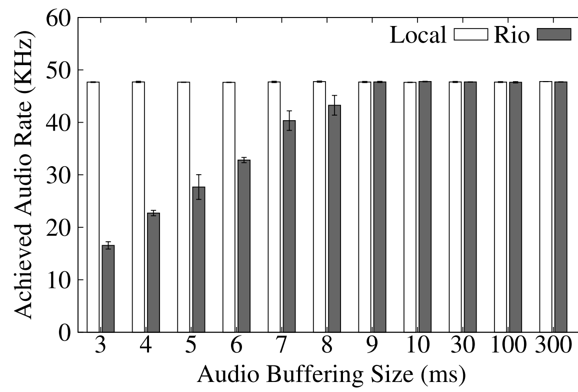
Audio devices: We evaluate the performance of the speaker and the microphone measuring the audio (sample) rate at different buffering sizes, and hence, different audio latency. Using larger buffering sizes reduces the interactions with the driver but increases the audio latency. Audio latency is the average time it takes a sample to reach the speaker from the process (and vice-versa for microphone), and is directly determined by the buffering size used in the HAL.

Figure 5.6 shows the achieved rate for different buffering sizes (and hence different latencies) for the speaker and the microphone when accessed locally or remotely through Rio. We use a minimum of 3 ms for the buffering size as it is the smallest size used for local speakers in Android in low latency mode. The figure shows that such a small buffering size degrades the audio rate in Rio. This is mainly because the HAL issues one `ioctl` for each 3 ms audio segment, but the `ioctl` takes longer than 3 ms to finish in Rio due to the network's long round-trip time. However, the figure shows that Rio is able to achieve the desired 48 kHz audio rate at slightly larger buffering sizes of 9 and 10 ms for microphone and speaker, respectively. We believe that Rio achieves acceptably low audio latency because Android uses a buffering size of 308 ms for non-low latency audio mode for speaker, and uses 22 ms for microphone (it uses 3 ms for low latency speaker).

We also measure the performance of audio devices with Rio when mobile systems are connected via the aforementioned high latency connection, e.g., for making a



(a)



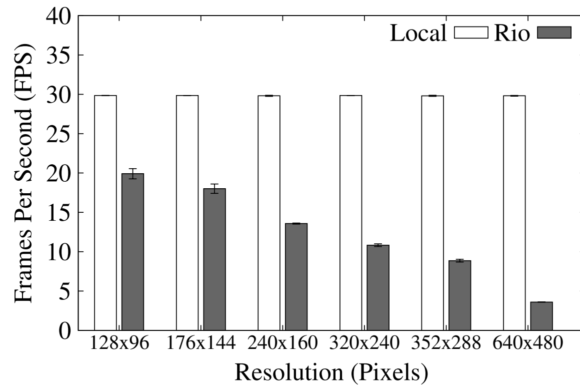
(b)

Figure 5.6 : Performance of (a) speaker and (b) microphone. The x-axis shows the buffering size in the HAL. The larger the buffer size, the smoother the playback/capture, but the larger the audio latency. The y-axis shows the achieved audio rate.

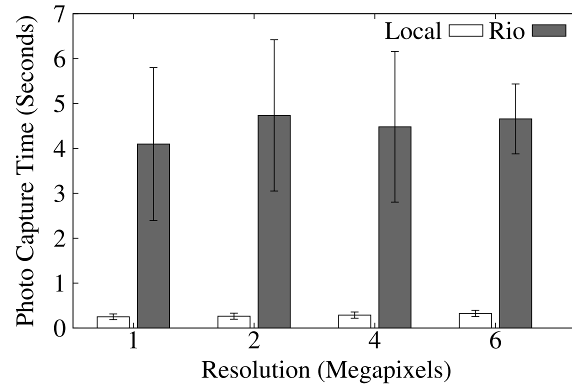
phone call remotely at work using a smartphone at home. Our measurements show that Rio achieves the desired 48 kHz for the microphone using buffering sizes as small as 85 ms. However, for the speaker, Rio can only achieve a maximum sampling rate of 25 kHz using a 300 ms buffer (other buffer sizes performed poorly). We believe this is because the speaker requires a higher link throughput, as demonstrated in §5.7.3.

Camera: We measure the performance of both a real-time streaming camera preview and that of capturing a photo. In the first case, we measure the frame rate (in frames per second) that the camera application can achieve, averaged over 1000 frames for each experiment. We ignore the first 50 frames to avoid the effects of camera initialization on performance. Figure 5.7(a) shows that Rio can achieve acceptable performance (i.e., >15 FPS) at low resolutions. The performance at higher resolutions is bottlenecked by network throughput between the client and server. Rio spends most of its time transmitting frames rather than file operations. However, streaming camera frames are uncompressed, requiring, for example, 612 KB of data *per frame* even for VGA (640×480) resolution, necessitating about 72 Mbps of throughput to maintain 15 FPS at this resolution.

We believe that the lower resolution camera preview supported by Rio is acceptable given that Rio supports capturing photos at maximum resolutions. Rio will support higher real-time camera resolutions using future wireless standards. For example, 802.11n, 802.11ac, and a 802.11ad can achieve around 200 Mpbs, 600 Mbps, and 7 Gbps of throughput respectively [158, 159]. Such throughput capabilities can support real-time camera streaming in Rio at 15 FPS for resolutions of 1280×720 and 1920×1080, which are the highest resolutions supported on Galaxy Nexus. Moreover, Rio can incorporate compression techniques, either in software or using the hardware-accelerated compression modules on mobile SoCs, to reduce the amount of data trans-



(a)



(b)

Figure 5.7 : Performance of a real-time streaming camera preview (a) and photo capture (b) with a 21.9 Mbps wireless LAN connection between the client and server. Future wireless standards with higher throughput will improve performance without requiring changes to Rio.

ferred for real-time camera, although we have not yet explored this optimization.

To evaluate photo capture, we measure the time from when the capture request is delivered to the camera HAL from the application until the HAL notifies the application that the photo is ready. We do not include the focus time since it is mainly dependent on the camera hardware and varies for different scenes. We take three photos in each experiment (totalling 9 photos in three experiments). Figure 5.7(b) shows the capture time for local and remote cameras using Rio. It shows that Rio adds noticeable latency to the capture time, mostly stemming from the time taken to transfer the raw images from the server to the client. However, the user only needs to point the camera at the targeted scene very briefly (similar to when using a local camera), as the image will be immediately captured in the server. This means that the *shutter lag is small*. It is important to note that the camera HAL in Galaxy Nexus uses the same buffer size regardless of resolution, hence the capture time is essentially resolution-independent. The buffer size is 8 MB, which takes about 3.1 seconds to transfer over our 21.9 Mbps connection. As with real-time camera streaming, future wireless standards will eliminate this overhead, providing latency on par with local camera capture.

Sensors: To evaluate remote sensor performance, we measure the average time it takes for the sensor HAL to obtain a new accelerometer reading. We measure the average time of 1000 samples for each experiment. Our results of 10 experiments show that the sensor HAL obtains a new local reading in 64.8 ms on average (with a standard deviation of 0.1 ms) and a new remote reading via Rio in 70.5 ms on average (with a standard deviation of 1.9 ms). The sensor HAL obtains a new reading by issuing a blocking `poll` operation that waits in the kernel until the data is ready, at which point the HAL issues a `read` file operation to read the new value. Rio causes

overhead in this situation because one and a half round trips are required for the blocking `poll` to return and for the `read` to complete. Fortunately, this overhead is negligible in practice and does not impact the user experience.

Modem: We measure the time it takes the dialer and messaging applications to start a phone call and to send an SMS, respectively. We measure the time from when the user presses the “dial” or “send SMS” button until the notification appears on the receiving phone. Our measurements show that local and remote modems achieve similar performance, as the majority of time is spent in carrier networks (from T-Mobile to AT&T). For local and remote modems, the phone call takes an average of 7.8 and 7.9 seconds (with standard deviations of 0.7 and 0.3 seconds) while SMS takes an average of 6.2 and 5.9 seconds (with standard deviations of 0.3 and 0.6 seconds), respectively.

5.7.3 Throughput

We measure the wireless throughput required for using different classes of I/O devices remotely with Rio. Our results show that using sensors, audio devices, and camera remotely requires small, moderate, and large throughput, respectively.

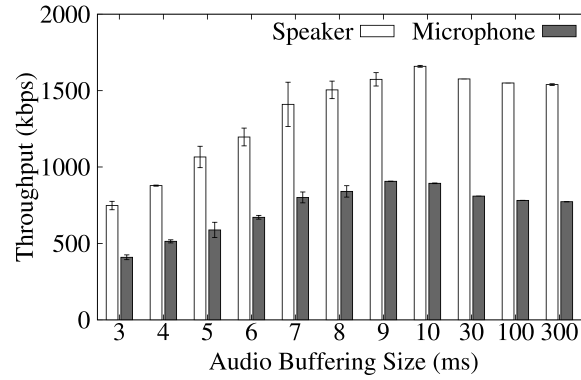
We quantify the throughput by measuring the amount of data transmitted between the client and the server. We measure the number of bytes transmitted over the TCP socket, therefore, our results does not include the overhead due to the TCP header and headers of lower layers. We run each experiment for one minute and measure the throughput. For microphone, we record a one minute audio segment. For speaker, we play a one minute audio segment. For camera, we stream frames for one minute, and for accelerometer, we receive samples for one minute. We report each experiment three times and report the average and standard deviation. We reboot the phone

before each experiment.

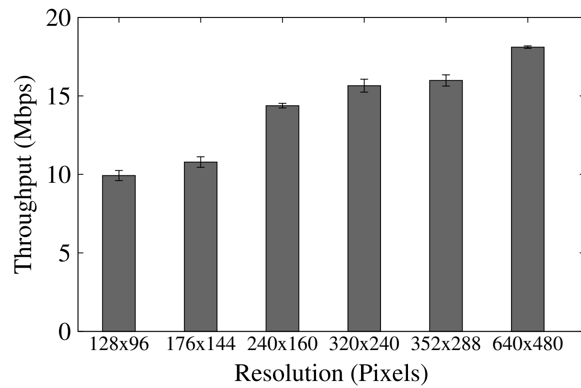
Figure 5.8(a) shows the results for the speaker and microphone. It shows that audio devices require a moderate throughput (hundreds of kbps) and therefore Rio's performance for these devices is not throughput bounded. The throughput increases as we increase the audio buffering size, caps when the buffering size is 9-10 ms, and then decreases for larger buffering sizes. This trend can be explained as follows: at low buffering sizes, the audio performance on Rio is bounded by the link latency due to the high number of round trips. As a result, fewer audio samples are exchanged, resulting in lower throughput. As the buffering size increases, more audio samples are exchanged, hence requiring higher throughput. The number of audio samples exchanged (i.e., the audio rate) is maximized when the buffering size is 9-10 ms. For larger buffering sizes, the number of audio samples are fixed but the Rio's communication overhead decreases, resulting in a lower overall throughput. Moreover, the results show that speakers uses twice the throughput used by the microphone. This is because audio samples for the speaker are twice the size of the audio samples used by the microphone in our experiments.

Figure 5.8(b) shows the throughput results for video streaming from a remote camera using Rio. We show that the camera requires high throughput, which is why the link throughput is a performance bottleneck in our setup. At high resolutions, the link is almost saturated with sending the frame content; Rio's overhead is small (since the number of frames is small). For lower resolutions, more frames are transmitted and therefore the effect of link latency becomes more noticeable. This is why Rio fails to saturate the link throughput at these resolutions.

We also measure the throughput when using the accelerometer remotely with Rio. Our measurements show the average throughput is 52.05 kbps with a standard



(a)



(b)

Figure 5.8 : Throughput for using (a) audio devices (speaker and microphone) and (b) the camera (for video streaming) with Rio. Note that the y-axis is in kbps and Mbps for (a) and (b), respectively.

deviation of 1.72. This shows that the throughput for sensors is small and therefore we can leverage low throughput, low energy links (such as Bluetooth) for these devices. Also, most of this throughput comes from Rio's overhead since accelerometer data are small.

5.7.4 Power Consumption

We evaluate the overhead of Rio in terms of both systems' power consumption. For each I/O device, we measure the average power consumption of the client and server in Rio and also the power consumption of the system when the I/O device is used locally. In each experiment, we use the device for one minute, similar to §5.7.3, and measure the average power consumption using the Monsoon Power Monitor [160]. We repeat each experiment three times and report the average and standard deviation of the experiment. The display consumes a large amount of power; therefore, we try to keep the display powered off whenever possible. More specifically, for the accelerometer and audio devices, we turn off the display on both the client and the server and also on the local system. For camera, we turn off the display only on the server but not on the client or the local system (because the camera frames are being displayed).

Figure 5.9 shows that Rio consumes noticeably more power than local devices. Considering the sum of the power consumption of client and server for Rio, Rio consume about 4× the power consumed by local accelerometer and audio devices, and 2× the power consumed by the local camera. These are expected results as Rio spans over two systems and uses the Wi-Fi interface. For camera, the source of power consumption of the local camera scenario is from the camera itself, the display, the CPU, and even the GPU, which is used for rendering the frames onto the screen.

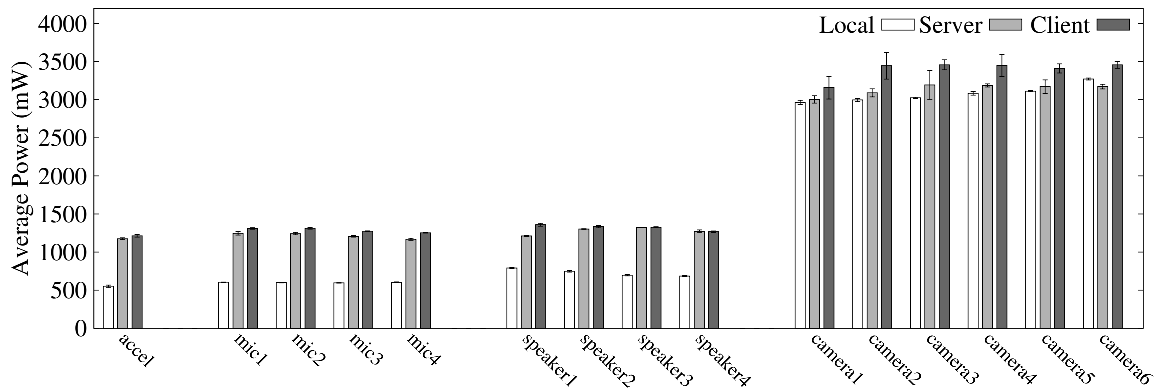


Figure 5.9 : Average system power consumption of when accelerometer, microphone, speaker, and camera (for video streaming) are used locally and remotely with Rio, respectively. For Rio, the power consumption for both the server and the client are shown. Scenarios 1 to 4 for audio devices correspond to audio buffering sizes of 3 ms, 10 ms, 30 ms, and 300 ms, respectively. Scenarios 1 to 6 for camera correspond to resolutions of 128×96, 176×144, 240×160, 320×240, 352×288, and 640×480, respectively.

For Rio, the source of power consumption on the client is from the Wi-Fi interface, the display, the CPU, and the GPU, and for the server, from the Wi-Fi interface, the CPU, and the camera. For other devices, the main source of power consumption for the local scenario is the device itself and the CPU. For Rio, the source of power consumption on the client is from the Wi-Fi interface and the CPU, and on the server is from the Wi-Fi interface, the CPU, and the device.

5.7.5 Handling Disconnections

We evaluate Rio's ability to react to disconnections for the accelerometer. We play a game on the client using the server's accelerometer. Without warning, we disconnect the server and the client, and then trigger a disconnection event after a customizable threshold. Rio then transparently switches to using the local accelerometer so that we can continue to play the game using the client's own accelerometer.

5.8 Discussions

5.8.1 Supporting more classes of I/O devices

Our current implementation supports four classes of I/O devices. It is possible to extend it to support graphics, touchscreen, and GPS, since they also use the device file interface. There are, however, two classes of I/O that Rio's design cannot support: network and block devices. This is because these I/O devices do not use the device file interface for communications between the process and the driver. Network devices use sockets along with the kernel networking stack and block devices use kernel file systems.

5.8.2 Energy use by Rio

Using an I/O device via a wireless link obviously incurs more energy consumption than using a local one, as demonstrated in §5.7.4. In this work, we did not address energy optimizations for Rio. Rather, we note that most of the performance optimizations in Rio, e.g., those described in §5.4, lead to more efficient use of the wireless link and therefore to reduced energy consumption. We also note that Rio's quest to reduce latency rules out the use of the standard 802.11 power-saving mode. On the other hand, many known techniques that trade a little latency for much more efficient use of the wireless link can benefit Rio, e.g., data compression [161] and μPM [162].

5.8.3 Supporting iOS

iOS also uses device files and hence can be supported in Rio. Sharing I/O devices between iOS systems should require similar engineering effort reported in this paper for sharing I/O devices between Android systems. However, sharing I/O devices between iOS and Android systems require potentially non-trivial engineering effort, mainly because these two systems have different I/O stack components and API.

Chapter 6

I/O Sharing between Servers (in a Datacenter)

New classes of I/O devices are being employed in datacenters. Accelerators, such as GPUs, are especially popular as they can execute some important workloads faster and more efficiently than CPUs. For example, GPUs excel at executing embarrassingly parallel workloads, such as encryption [163–165], graphics and video processing [166], visualization [54, 167], and scientific computation [168–170]. As another example, FPGAs are shown to be effective for accelerating a web search engine [171]. In this chapter, we propose a solution for consolidation of I/O devices in datacenters through I/O sharing between servers. While the solution is generic to all I/O devices, we focus on GPUs due to their increased importance and adoption in datacenters.

Unfortunately, in today's datacenters, GPUs are expensive to use. For example, at the time of this writing, the Amazon EC2 cloud offers two GPU VM instance types (i.e., VMs with access to GPUs). The smaller GPU instance incorporates one NVIDIA GPU (with 1536 cores and 4 GBs of device memory), 8 virtual CPUs (vCPUs), 15 GBs of memory, and is priced at $0.65 per hour. The larger instance incorporates 4 NVIDIA GPUs, 32 vCPUs, 60 GBs of memory, and is priced at $2.6 per hour. In contrast, EC2 offers non-GPU VM instances that are considerably less expensive. For example, it offers a VM with 1 vCPU and 1 GB of memory for as low as $0.013, a whopping 50× difference with the cheapest GPU instance.

There are two fundamental reasons for the high price of using GPUs. First, compared to more traditional resources, e.g., CPU and memory, GPU is a more

scarce resource. That is, not all the machines in the datacenter are equipped with GPUs. Indeed, high-end GPUs require specialized machines that meet the space and power requirements of GPUs, and the commonly used machines (e.g., 1U servers) cannot physically host such GPUs. Second, even in the machines with GPUs, this resource is poorly consolidated as VMs are given access to dedicated GPUs. GPU virtualization, as provided by Paradice (§4) and other solutions [6, 7, 22, 32], can alleviate the second problem by sharing the GPU between VMs running in the same machine. In this chapter, we present a solution to the first problem by enabling VMs running in different machines to remotely access and use the GPU resources over the datacenter network.

We propose Glance, our design and implementation of remote access to GPUs using the device file boundary. In Glance, a guest VM uses a virtual device file that corresponds to the actual device file of the GPU it wishes to use. A client stub modules intercepts the file operations issued by applications in the guest VM and forwards them to the virtual machine hosting the GPU, called the driver VM. The driver VM is in a different physical machine that the one hosting the guest VM.

We face a fundamental challenge in Glance: the use of closed source drivers. For best performance, datacenter vendors often use the closed source drivers available from GPU manufacturers rather than open source drivers, e.g., drivers provided by Linux. This is because not only closed source drivers achieve better performance than their open source counterparts, they also have more complete support for GPU frameworks, e.g, CUDA and OpenCL. Closed source drivers, however, create two important problems in Glance. First, the DSM solution with a write-invalidate coherence protocol, as used in Rio (§5), is not feasible in Glance since it requires small modifications to the device driver. We solve this problem using a DSM solution that implements

all the coherence logic in the guest VM, eliminating the need for modifications to the device driver source code. Second, static analysis of the device driver source code needed to reduce the round trips between the client and server, again as used in Rio, is not feasible with closed source drivers. We solve this problem by collecting and analyzing the execution traces of GPU workload.

We implement Glance for the x86 architecture and show its feasibility for a Radeon GPU, namely Radeon HD 6450, using the AMD closed source driver called `fglrx`.

We evaluate Glance and show that it provides comparable performance to local access to GPUs and far better performance compared to using CPUs for the same OpenCL workload. For example, the multiplication of two 5000×5000 square matrices take about 4.4, 9.3, 277.7, and 42.6 seconds for local GPU, Glance, one CPU core, and four CPU cores. This demonstrate that Glance is a viable solutions for providing high GPGPU performance for VMs running on machines without GPUs.

It is important to note that an alternative solution to remote access to GPUs is using VM migration. That is, one can migrate the VMs that require access to GPUs to a physical machine that contains GPUs and migrate them out of that machine once they are done with the GPU. However, migration has two important problems compared to remote GPU access. First, migration of the guest VM might not be possible due to scarcity of other resources, e.g., CPU and memory, in the target machine since those resources might be already committed to other VMs. Second, migration of VMs is costly, requiring pausing the VM and transferring all of its memory state and potentially its storage content.

6.1 Design

Glance enables remote access to GPUs at the device file boundary. That is, using this boundary, Glance enables a guest VM running in a physical machine without a GPU to remotely use the GPU in another physical machine, i.e., the target physical machine. To do this, we create a virtual device file for the target GPU inside the guest VM that wishes to use the device. The client stub in the guest VM then handles the file operations issued by applications in the VM on the virtual device file and forwards them to the server stub over the datacenter network. The server stub, running in a VM in the target physical machine, then passes these file operations to the GPU device driver to be executed. In Glance, the GPU is assigned to and controlled by a VM, called the driver VM. Compared to running the driver natively in the target physical machine, this design provides better isolation between guest VMs sharing the GPU. With this design, Glance can use the techniques discussed in §4.2 to provide fault and device data isolation between guest VMs sharing the GPU. Figure 6.1 illustrates this design.

As mentioned earlier, for better performance and for complete GPGPU API support, datacenter vendors often use the closed source GPU drivers available from GPU manufacturers. The use of these drivers creates challenges in Glance to support a DSM solution between the client and server and to analyze the device driver source code and use the analysis for better performance. In the following sections, we present our solutions to these problems. These solutions will make access and modifications to the device driver source code unnecessary.

It is important to note that the solutions that will be presented in this chapter can also be used in Rio (§5) to make access and modifications to the device driver source code unnecessary. However, the solutions presented here have overheads compared to

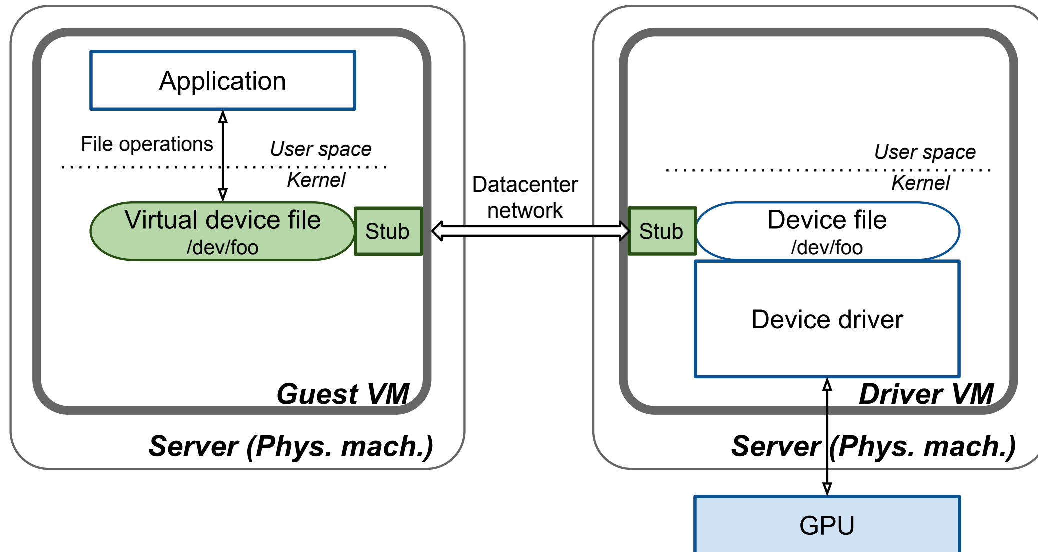


Figure 6.1 : Glance uses the device file boundary for remote access to GPUs. Using Glance, a guest VM in a physical machine without GPUs can remotely use a GPU in another physical machine, i.e., the target physical machine. The GPU is assigned to and controlled by a VM, called the driver VM. Compared to running the driver natively in the target physical machine, this provides better fault and device data isolation for guest VMs sharing the GPU.

the ones used in Rio. The specific overhead of each of the solutions will be discussed later in this section after the solution is presented. Therefore, Rio's solutions are preferred if the device driver source code is available.

6.1.1 Cross-Machine Memory Map

The device driver often maps physical pages into the application process when requested by the application through the `mmap` file operation. However, such mappings are not immediately possible in Glance since the guest VM and the driver VM are

in two different machines with separate physical memory. In Glance, similar to Rio, we achieve such a memory mapping using a Distributed Shared Memory (DSM) solution. That is, for every page that the driver needs to map into the process address space, we create a shadow page in the guest VM and map it to the process address space. The DSM solution then guarantees that the guest VM application always sees a coherent copy of the page.

However, using the GPU closed source driver poses challenges to adoption of a write-invalidate DSM solution used in Rio. This is because we cannot modify the device driver to inform us about device-side access to mapped pages. The device-side access is either through DMA for system memory pages or is internal to the GPU for GPU's own memory pages. Unlike reads/writes from the application in the client that can be forced to fault by removing the page table permission bits, device-side access is not affected by page table permission bits. Note that we cannot use IOMMU to force the DMAs to fault. This is because such IOMMU faults are not recoverable [172], and this will break the GPU functionality. Moreover, the IOMMU cannot be used for reads/writes to the GPU memory.

To solve the challenges mentioned above, we design and implement a DSM solution that implements the coherence logic fully in the guest VM. That is, the guest VM always fetches and updates the memory mapped pages from/to the driver VM assuming that the driver may access any mapped pages when handling a file operation. More specifically, this DSM solution forces every application's read and write from/to the mapped pages to fault, so that it can fetch the latest copy of the page from the driver VM. The application is then allowed to freely read and write from/to the page until the next file operation is issued at which point the page is made invalid again. Moreover, in the case of write, the updated page is transferred to the driver

VM before the next file operation is forwarded.

Such a DSM solution is possible due to the following important observation: the device and the application do not read/write to mapped pages concurrently. Their access to these pages is synchronized through the file operations. Therefore, as long as the pages are coherent upon the issue of file operations, memory coherency is satisfied.

However, this DSM solution may result in unnecessary data transfers compared to the write-invalidate DSM solution used in Rio. This is because the guest VM needs to always assume that all the mapped pages can be accessed by the device for each file operation. Therefore, if reads and writes to the same memory mapped page in the guest VM is interleaved with unrelated file operations, i.e., file operations that do not need the driver to access those mapped pages, unnecessary transfers will be incurred. In §6.1.2, we discuss how we reduce such overhead for buffers containing input/output data for the GPU.

Also, note that as in Rio, and for similar reasons explained in §5.3.1, Glance provides sequential consistency.

6.1.2 Reducing Network Round Trips

The second challenge that we faced is the large number of network round trips caused by communications between the guest and driver VMs, which negatively impacts the performance.

As also discussed in §5.4, there are three main reasons for such round trips: file operations, copy operations, and DSM messages. We next explain our solutions for reducing the round trips due to copy operations and DSM messages. Reducing the round trips due to file operations require modifications to the GPU runtime, which

is not possible in our prototype as the runtime is closed source.

Round trips due to copy operations: In handling some file operations, mainly `ioctl`, `read` and `write`, the device driver copies data to and from the application process memory. In a naive implementation, every such copy results in one round trip from the driver VM to the guest VM. To reduce this overhead, similar to Rio (§5), Glance precopies the input data of the driver in the guest VM and transmits them to the driver VM along with file operation request. Moreover, the driver VM buffers and batches all the output data of the device driver and sends them back to the guest VM with the return value of the file operation. While the latter is trivial to do, the former is difficult for the `ioctl` file operation since the guest VM stub module does not know which data the device driver requires for every `ioctl`. This is because, in contrast to `write`, the input arguments of the `ioctl` do not have any semantic meanings with respect to the operation of the device driver. In Rio, we solved this problem by performing static analysis on the device driver source code offline and extracting the required information. However, this solution is not possible with GPU's closed source drivers.

To solve this problem, we collect execution traces of GPU workload and extract the required information out of the traces. In the traces, we record the arguments of each `ioctl`, which includes a command number and a pointer. We also record the arguments of the copy requests issued by the driver when handling the `ioctl`. The copy request arguments include the source address, destination address, and the size of the copy. We also record the content that is copied for each request.

We recover the required information from the traces as follows. If the copy is from the pointer that is passed as an argument of the `ioctl`, then we just record the size of the copy. The first copy issued by the driver for handling each `ioctl` is always of

this type. If the copy is from a different pointer, the pointer value must have been passed to the driver as part of the data that were previously copied, i.e., a nested copy. Therefore, we look into the copied data and find the offset in the data that embeds the pointer value. We also record the size of the copy. However, the size can change for different instances of each copy since the size of the copy can also be passed to the driver in previous copies. We therefore record the largest size for each copy instance that we observed in our traces. At runtime, if the size of the copy is smaller than the one recovered from the traces, we copy less data from the process memory successfully. If it is larger, the server stub in the driver sends a request back to the client stub, resulting in an additional round trip, hence additional overhead.

Note that compared to the solution adopted in Rio that can always correctly precopy all the data that the driver needs (no more and no less), the proposed solution here is only a best-effort. That is, this approach only detects the copy operations observed in previous traces. Fortunately, our experience has shown that the copy operation of `ioctls` are mostly independent of the specific workload. For example, we recorded traces for the execution of an OpenCL FFT workload and used that to correctly predict the input data of the driver for an OpenCL matrix multiplication benchmark. Moreover, while we currently analyze the traces offline to extract the required information, it is possible to design a solution that learns such information at runtime.

Round trips due to DSM messages: As mentioned in §6.1.1, the guest VM keeps the mapped pages coherent by proactively moving their content from/to the driver VM. This results in one/two round trip(s) per page when the application reads/writes from/to a buffer, which can be prohibitively expensive for large data buffers, such as input data to a GPGPU kernel.

We reduce this overhead using the following observations: the memory buffers used to communicate the input and output data with the GPU are only written by the application or by the device and read by the other one (and not both). Moreover, only when one side is done writing to the buffer fully, the other side will attempt at reading the content.

Using these observations, we adopt the following optimizations. We do not use page faults to trigger the DSM messages for such buffers. We only transfer these buffers upon explicit request from the application. Moreover, for the input buffers to the GPUs, we avoid pulling the original content of the mapped pages from the driver VM since they will be immediately overwritten by the application.

Obviously, these optimizations require application support. That is, we require the developer to add some code in order to mark the allocation of data buffers and the transfer of these buffers to/from the GPU-accessible memory. However, adding such code is quite easy, e.g., requiring only two lines of code per GPGPU data buffer, resulting in four or six lines of code, overall, for many GPGPU benchmarks. Moreover, it is possible to move such additional code to the GPGPU runtime, eliminating the need for application developer effort. This is because buffer allocation and movement are typically done with explicit GPGPU API calls, which we can modify for our purpose. However, we cannot currently implement this alternative since the AMD OpenCL runtime is closed source.

6.2 Implementation

We implement Glance for the x86 architecture. Our implementation supports Radeon GPUs using the AMD closed source drivers, i.e., the AMD `fglrx` drivers.

Our implementation consists of four components: the client and server stubs, the

Component	LoC
Server stub	4063
Client stub	3070
- Shared between stubs	787
DSM	1092
Supporting Linux kernel code	216
GPU/PCI info modules	355

Table 6.1 : Glance code breakdown.

device info modules (§4.3.1), and the DSM module explained in §6.1.1. The client and server stubs are responsible for forwarding and handling the file operations issued in the guest VM. The device info modules virtualize the kernel-user interface for the applications, creating the illusion that the remote GPU is locally attached to the guest VM. In addition to what was explained in §4.3.1 for device info modules, we also noticed that the AMD closed source GPU runtime probes the kernel for the existence of the `fglrx` module (the AMD driver). Therefore, we rename our client stub to `fglrx` in the guest VM to successfully pass the runtime probe. Table 6.1 shows the breakdown of Glance's code base.

6.2.1 Linux Kernel Function Wrappers

Similar to Paradice and Rio, the server stub in Glance requires to intercept and take action on the device driver calls into some kernel functions while servicing a file operation from the guest VM. We achieve this by adding some function wrappers to these kernel functions and forwarding the function calls to the server stub when the function is called by the device driver. Such kernel functions include `copy_from_user`, `copy_to_user`, `insert_pfn`, etc. However, in addition to function wrappers used in

Paradice and Rio, we added wrapper support for two new kernel functions in Glance since they are used by the AMD closed source driver. Below, we explain these two functions.

`do_mmap()`: Memory mapping is often initiated by the application using the `mmap` file operation. However, Linux allows its modules, including drivers, to initiate the mapping as well, through a call to `do_mmap()`. The `fglrx` calls the `do_mmap()` function while handling an `ioctl`, effectively turning the `ioctl` into an alternative for the `mmap` file operation.

To support `do_mmap()`, we intercept the function call in the driver VM kernel and send a request to the guest VM. The client stub in the guest VM then executes the `do_mmap()`. This will then result into a callback into the `mmap` handler of the client stub, which is then forwarded to the driver VM to be executed. Once the `mmap` is handled in the driver VM, it returns to the guest VM, and the guest VM also returns from the `do_mmap()` request.

Adding support for `do_mmap()` requires supporting nested file operations in Glance. This is because the `mmap` is sent to the driver VM while the `ioctl` is being serviced. To do this, we modified several functions in the server stub in order to make them re-entrant, otherwise the nested file operation would corrupt the original in-flight file operation state.

`get_user_pages()`: The device driver can lock and get access to the underlying memory pages of a range of application's virtual addresses by calling the `get_user_pages()` kernel function. In Glance, we intercept the device driver's call into this kernel function in the driver VM and forward and execute it in the guest VM. However, the set of underlying pages returned by this function in the guest VM cannot be directly passed to the driver VM since the two VMs have different physical address space. We

solve this problem by leveraging our DSM solution explained in §6.1.1. That is, for every locked pages in the guest VM, we create a shadow page in the driver VM and pass that to the device driver. The DSM solution then guarantees that these pages stay coherent.

6.3 Evaluation

We evaluate Glance and show that remote access to GPUs achieves comparable OpenCL performance with local access to GPUs and superior performance than a CPU.

In the experiments reported below, we use the Xen hypervisor and configure each VM with one virtual CPU core and 2 GBs of memory, unless otherwise stated. We use the Radeon HD 6450 GPU in the experiments. For the network connection, we use two Intel X520-2 10 Gigabit Ethernet adapters, each of which is assigned to one of the VMs, i.e., the guest and driver VMs.

We use an OpenCL benchmark in the experiments. The benchmark is a matrix multiplication that multiplies two square matrices of the same order. A square matrix of order N is an N×N matrix. The figure reports the experiment time, which is the total time for the benchmark from start to finish, including the time for the application to probe the system to find OpenCL-enabled platforms, e.g., GPU or CPU, the time to program the platform, the time to transfer the data from/to the platform, and finally, the time for the platform to execute the OpenCL kernel. The figure shows the results for executing the matrix multiplication benchmark on a local GPU, on a remote GPU, and on a CPU (both with 1 core or 4 cores). We use Intel Core-i7 3770 for the CPU experiments.

Figure 6.2 shows the results. We observe the following from the figure. First,

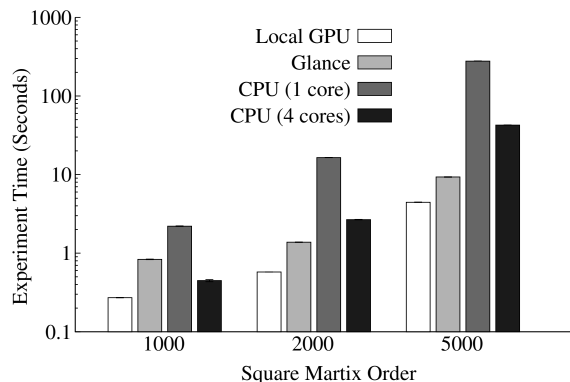


Figure 6.2 : OpenCL matrix multiplication. The x-axis shows the (square) matrix order. The y-axis shows the time it takes to execute the multiplication on the OpenCL-enabled platform, GPU or CPU.

remote GPU through Glance provides comparable performance to local access. Most of the overhead of Glance is for transferring the input data to driver VM (i.e., the two input matrices) and transferring the output data from the driver VM (i.e., the resulting matrix). A buffer to store a square matrix of order 1000, 2000, and 5000 is about 4 MB, 16 MB, and 96 MB in our benchmark, requiring about 12 MB, 50 MB, and 300 MB of data transfer for executing the benchmark, respectively. However, the results show that the additional overhead is larger than the time it takes to transfer this data at the line rate on a 10 Gigabit Ethernet connection. This is due to the overhead of the networking stack when using a TCP socket and the overhead of mapping and unmapping the pages from high memory, which is used for allocating large buffers in the kernel in the x86 architecture.

Second, network access to GPU provides better performance than executing the same OpenCL benchmark locally on CPU cores. This demonstrate that Glance is a viable solutions for providing high GPGPU performance for VMs running in physical

machines without GPUs.

Chapter 7

Enhancing Device Driver Security

Remote access to I/O devices at the device file boundary raises an important concern: the security of I/O servers in face of untrusted I/O clients. Through this boundary, applications in the clients are able to directly communicate with the device driver in the I/O server. Unfortunately, device drivers are the main sources of bugs in operating systems [115]. They are large, fast-changing, and developed by third parties. Since the drivers run in the kernel of modern monolithic operating systems and are fully shared by applications, their bugs are attractive targets for exploitation by malicious clients, and pose great risks to the security of I/O servers.

This longstanding problem is particularly troubling for hardware accelerators such as GPUs because they tend to have large and complex device drivers. For example, GPU device drivers have tens of thousands of lines of code and have seen quite a few attacks recently [116, 117, 173–175]. This problem is increasingly critical as GPUs are used even by untrusted web applications (through the WebGL framework). Indeed, browser vendors are aware of the this security risk and they disable the WebGL framework in the presence of GPU drivers that they cannot trust [176]. Obviously, this is a rough solution and provides no guarantees even in the presence of other drivers.

In this chapter, we present a solution to this problem. We present a novel device driver design, called *library drivers*, that reduces the size and attack surface of the driver Trusted Computing Base (TCB) and hence reduces the possibility of the driver

getting compromised.

The core of library drivers is the following insight: a large part of legacy drivers is devoted to device resource management. Therefore, library drivers, inspired by library operating systems, such as Exokernel [80] and Drawbridge [86], applies the fundamental principle of *untrusted resource management* to device drivers. The library driver design incorporates this principle in two steps. First, it separates device resource management code from resource isolation. In a library driver design, resource isolation is implemented in a trusted *device kernel* in the operating system kernel, and resource management is pushed out to the user space. Second, resource management is implemented as an untrusted library, i.e., a *device library*. That is, each client application that intends to use the device loads and uses its own device library. Based on some scheduling policy, the device kernel exports the device hardware resources securely to client applications, which manage and use the resources with their own device library.

Beyond securing I/O servers in face of remote access from untrusted clients, library drivers improve the security of monolithic operating systems for local access as well. Therefore, in the rest of this chapter, we mainly focus on using library drivers to enhance the security of local access in order to simplify the discussions. However, the security benefits of library drivers automatically extend to remote access in I/O servers as well since the I/O client uses the device file boundary to communicate with the device driver, very similar to how applications use this boundary for local access.

The library driver design improves overall system security by reducing the size and attack surface of the Trusted Computing Base (TCB). With a legacy driver, the whole driver is part of the TCB. However, with a library driver, only the device kernel, which is smaller than a legacy driver, is part of the TCB. Moreover, compared to a

legacy driver, the device kernel exposes a narrower lower-level interface to untrusted software, hence reducing the attack surface of the TCB. The security benefits of a library driver are two-fold: first, a library driver reduces the possibility of attacks on the operating system kernel through bugs in the driver. Second, it improves the isolation between applications using the device, as it reduces the amount of shared state between them. Importantly, a library driver improves the system security without hurting the performance. Indeed, a library driver can even outperform a legacy driver due to one fundamental reason: a library driver avoids the overhead of syscalls and user-kernel data copies since it is in the same address space and trust domain as the application. The performance improvement highly depends on the application; the more interactions there are between the application and the driver, the more significant the performance improvement will be.

Applying the principle of untrusted resource management to a device driver requires certain hardware properties on the device and platform. We articulate these properties into three requirements: *memory isolation primitives* such as an IOMMU, *innocuous device management interface*, and *attributable interrupts*. If a device and its platform meet these properties, then it is possible to retrofit the device resource management code in the form of an untrusted library. Every violation of these requirements, however, forces some of the resource management code to remain in the device kernel, resulting in weaker security guarantees.

We target library drivers mainly for accelerators, such as GPUs, for three reasons. First, they are an increasingly important subset of devices; we anticipate various accelerators to emerge in the near future. Second, they are sophisticated devices requiring large device drivers, in contrast to simpler devices such as a mouse. Third, they often meet the hardware requirements mentioned above, as we will discuss in

§7.1.2.

Based on the aforementioned principle, we implement Glider, a Linux library driver implementation for two GPUs of popular brands, namely the Radeon HD 6450 and Intel Ivy Bridge GPUs. We implement Glider based on the original legacy Linux drivers. The library driver design allows us to implement both the device kernel and the device library by retrofitting the legacy driver code, which significantly reduces the engineering effort compared to developing them from scratch. We present a full implementation for the Radeon GPU and a proof-of-concept implementation for the Intel GPU.

Our evaluation shows that Glider improves the security of the system by reducing the size and attack surface of the TCB by about 35% and 84% respectively for the Radeon GPU and by about 38% and 90% respectively for the Intel GPU. We also show that Glider provides at least competitive performance with a legacy driver, while slightly outperforming it for applications requiring intensive interactions with the driver, such as GPGPU applications with a small compute kernel and graphics applications using OpenGL's immediate mode.

In addition, library drivers can benefit other systems as well. For example, they can be integrated into a library operating system, such as Exokernel [80] or Drawbridge [86], or into sandboxing solutions such as Embassies [87], Xax [88], and Bromium micro-virtualization [89], to securely support shared access to I/O devices.

Researchers have long attempted to protect the system from device drivers. For example, user space device drivers are one of the principles of microkernels [62]. Even for monolithic operating systems, there exist solutions that move the driver to user space [67, 68], move it to a VM [70], or even sandbox it in-situ [73, 74]. All these solutions improve the security of the operating system kernel by removing the driver.

However, in contrast to a library driver, they cannot improve the isolation between processes using the device, since the driver is fully shared between the processes.

In §4.2, we presented solutions for enforcing fault and device data isolation between VMs in Paradice. However, as discussed in §4.5, those solutions fail to provide guarantees on the correctness of device functionality. This is because they assumed that the device driver is compromised and tried to provide isolation guarantees in face of a compromised device driver. Library drivers provided in this chapter are complementary. They try to reduce the possibility of the device driver getting compromised by reducing the size and attack surface of the driver TCB.

7.1 Library Driver: Design & Requirements

In this section, we discuss the library driver design, and elaborate on hardware properties necessary to implement it.

7.1.1 Design

The library driver design is based on an important principle from the library operating system research [80]: resource management shall be implemented as untrusted libraries in applications. Resource management refers to the code that is needed to program and use the device. In contrast, legacy device drivers implement resource management along with device resource isolation in the kernel, resulting in a large TCB with a wide high-level interface.

Figure 7.1 compares a library driver against a legacy driver. The library driver design applies the aforementioned principle in two steps. First, it separates resource management from resource isolation. It enforces resource isolation in a trusted *device kernel* in the operating system kernel and pushes resource management into user

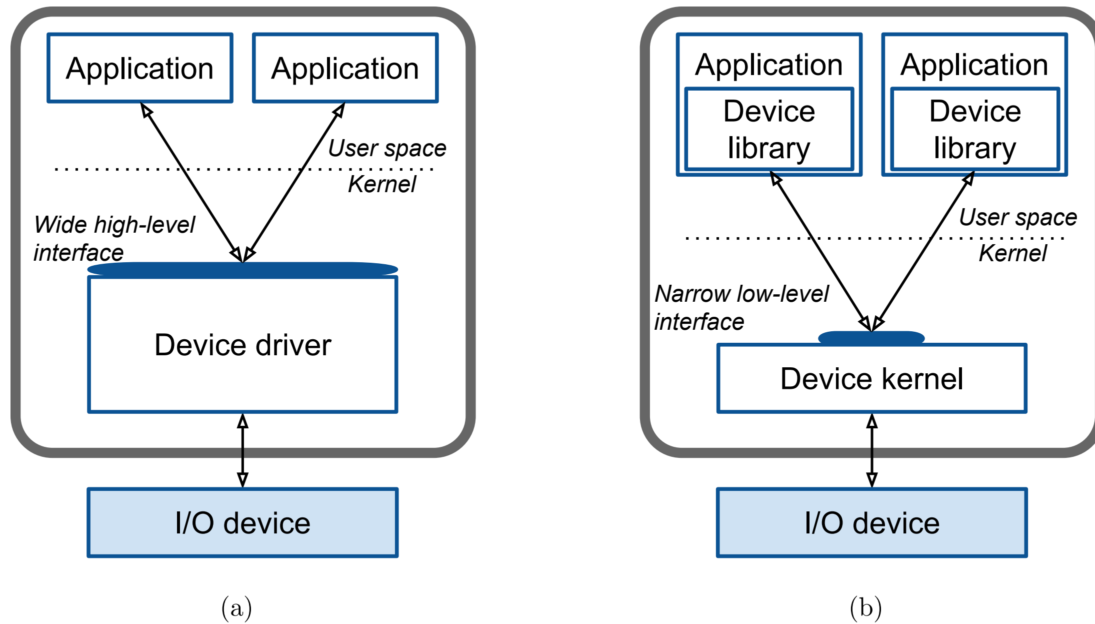


Figure 7.1 : (a) Legacy driver design. (b) Library driver design. Note that the solid-color interface in both figures refers to the set of APIs exposed by the device driver (or the device kernel in case of library drivers. These APIs are exposed through a device file, e.g., various `ioctls`, however in this figure, we do not show the device file itself for the sake of simplicity.

space. Second, the library driver design retrofits the resource management code into an untrusted library, i.e., a *device library*, which is loaded and used by each application that needs to use the device.

The device kernel is responsible for securely multiplexing the device resources between untrusting device libraries. Based on some scheduling policy, it *binds* the device resources to a device library at the beginning of a scheduling epoch and *revokes* them at the end of the epoch. If possible, the device kernel preempts the execution upon revoke. When hardware preemption is impossible or difficult to implement,

e.g., GPUs [177], the device kernel revokes access when allowed by the device*. Many devices, including the ones in our prototype, support only one hardware execution context. For them, the device kernel effectively time-multiplexes the device between device libraries. These devices are the main focus of this paper. Some devices, however, support multiple hardware execution contexts [18–22]. For those devices, the device kernel can dedicate a hardware execution context to a device library as will be discussed in §7.5.2.

The device library is responsible for managing the resources exported to it by the device kernel. It implements all the software abstractions and API calls needed by the application to use the device. For example, a GPU device library implements memory management API calls, such as `create_buffer`, `read_buffer`, `map_buffer`, and `move_buffer`, on top of the device memory exported to it by the device kernel. It is important to note that the device library implements the software abstraction and APIs in user space whilst legacy device drivers implement everything in the kernel. We also note that today's devices often employ user space libraries to create a unified interface to different drivers of the same device class, e.g., GPUs. Such user space libraries do not qualify as device libraries since they do not implement resource management.

Key Benefits

The library driver design improves system security because it reduces the size and attack surface of the TCB. In a library driver, only the device kernel is part of the TCB. Moreover, without any management responsibilities, the device kernel is able to

*It is up to the scheduling policy to ensure fairness, using techniques similar to the ones used in [29, 136, 178, 179]

expose a narrower lower-level interface to untrusted software. As a result, the library driver design improves two aspects of system security: it raises the bar for attacks that exploit device driver bugs, and it improves the isolation between applications by reducing the amount of shared state between them.

The library driver design improves the system security without hurting the performance. Indeed, a library driver can even outperform legacy drivers. This is because a library driver eliminates the overhead of the application interacting with the driver in the kernel, including costly syscalls [180] and user-kernel data copies. In a library driver, most of the interactions occur inside the process in the form of function calls rather than syscalls. Moreover, because the device library is in the same trust domain as the application, data can be passed by reference rather than being copied, which is commonly done in a legacy driver.

As we will demonstrate, the library driver design allows us to implement both the device kernel and the device library by retrofitting the legacy driver code. This significantly reduces the engineering effort to develop library drivers for existing devices compared to developing them from scratch.

In addition to security and performance, a library driver has other advantages and disadvantages. In short, the advantages include the possibility of driver customization for applications, easier user space development and debugging, and improved operating system memory usage accounting. The disadvantages include difficulty of multi-process programming, application launch time overhead, and coarser-grained device memory sharing for some devices. §7.5.1 discusses these issues in greater detail.

We find that library drivers are particularly useful for accelerators such as GPUs, which are increasingly important for a wide range of applications, from datacenters

to mobile systems. Accelerators are sophisticated hardware components and usually come with large, complex device drivers, particularly prone to the wide variety of driver-based security exploits. More importantly, we have found that accelerators like GPUs often have the necessary hardware properties to implement a library driver, elaborated next.

7.1.2 Hardware Requirements

Despite its benefits outlined above, library drivers cannot support all devices. The system must have three hardware properties for the device kernel to enforce resource isolation. The lack of any of these properties will leave certain resource management code in the trusted device kernel. (*i*) First, in order for each device library to securely use the memory exported to it by the device kernel, hardware primitives must be available to protect the part of the system and device memory allocated for a device library from access by other device libraries. (*ii*) Second, in order for a device library to safely program the device directly from user space, the registers and instruction set used to program and use the device must not be *sensitive*, i.e., they must not affect the isolation of resources. (*iii*) Finally, in order for the device kernel to properly forward the interrupts to the device libraries without complex software logic, the device interrupts must be easily attributable to different device libraries. Apparently not all devices have these properties; fortunately, accelerators of interest to us, e.g., GPUs, do have these properties, as elaborated below.

Memory Isolation Primitives

The library driver design requires hardware primitives to isolate the access to memory by different device libraries. There are two types of memory: system memory

and device memory, the latter being memory on the device itself. Isolation needs to be enforced for two types of memory access: *system-side* access via the CPU's load and store instructions and *device-side* access via direct device programming. Modern processors readily provide protection for system-side access with their memory management units (MMU). As a result, we only need to be concerned with protecting device-side accesses.

Device-side access to system memory: The device library can program the device to access system memory via direct memory access (DMA). Therefore, the device kernel must guarantee that the device only has DMA access permission to system memory pages allocated (by the operating system kernel) for the device library currently using the device. The I/O memory management unit (IOMMU) readily provides this protection by translating DMA target addresses. The device kernel can program the IOMMU so that any DMA requests from a device library are restricted to memory previously allocated for it. We observe that IOMMUs are available on most modern systems based on common architectures, such as x86 and ARM [13, 181]. Moreover, GPUs typically have a built-in IOMMU, which can be used by the device kernel even if the platform did not have an IOMMU.

Device-side access to device memory: The device kernel allocates the device memory for different device libraries, and it must protect them against unauthorized access. A device library can program a device to access the device memory, and such an access does not go through the IOMMU. Therefore, the device must provide hardware primitives to protect memory allocated for one device library from access by another. There are different forms of memory protection that a device can adopt, e.g., segmentation and paging. Each form of memory protection has its pros and cons. For example, segmentation is less flexible than paging since the allocations of

physical memory must be contiguous. On the other hand, paging is more expensive to implement on a device [182].

Isolating access to the device memory only applies to devices that come with their own memory. We note that accelerators packing their own memory, such as discrete GPUs and DSPs, often support some form of memory protection primitives. For example, NVIDIA GPUs (nv50 chipsets and later) support paging [183] and TI Keystone DSPs support segmentation [184].

A legacy driver does not require such memory protection primitives as it can implement software-based protection. A legacy driver implements the memory management code in the kernel and employs runtime software checks to ensure that untrusted applications never program the device to access parts of the memory that have not been allocated for them.

Innocuous Device Management Interface

The library driver design further requires the device management interface to be *innocuous* or not *sensitive*, as defined by the Popek-Goldberg theorem about virtualiziblity of an architecture [185]. According to Popek and Goldberg, sensitive instructions are those that can affect the isolation between virtual machines.

A device usually provides an interface for software to use it. This interface consists of registers and potentially an instruction set to be executed by the device. The device management interface is the part of the interface that is used for resource management. Examples include the GPU interface used to dispatch instructions for execution and the GPU interface used to set up DMA transfers. Other parts of the programming interface are used for initializing the device and also to enforce isolation between resources. Examples are the GPU interface used to load the firmware, the

interface used to initialize the display connectors on the board, the interface used to prepare the device memory and memory protection primitives.

For **registers**, this requirement means that management registers cannot be sensitive. That is, registers needed for resource management cannot affect the resource isolation. For example on a GPU, software dispatches instructions to the GPU for execution by writing to a register. This register is part of the management interface, and therefore must not affect the isolation, e.g., change the memory partition bounds.

For the **instruction set**, this requirement means that either the instruction set has no sensitive instructions, or the sensitive instructions fail when programmed on the device by untrusted code, such as a device library. The latter is more expensive to implement in the device as it requires support for different privilege levels, similar to x86 protection rings.

Commodity accelerators usually meet this requirement. This is because registers often have simple functionalities, hence management registers are not used for sensitive tasks. (§7.3.1 discusses one violation of this requirement). Also, the instruction set often does not contain sensitive instructions, and resource initialization and isolation is done through registers only.

A legacy driver does not need the device to meet the innocuous device management interface requirement. First, it does not allow untrusted software to directly access registers. Second, all the instructions generated by the application are first submitted to the driver, which can then perform software checks on them to guarantee that sensitive instructions, if any, are not dispatched to the device.

Attributable Interrupts

A device library using the device must receive the device interrupts in order to properly program and use the device. For example, certain GPU interrupts indicate that the GPU has finished executing a set of instructions. The interrupt is first delivered to the device kernel, and therefore, the device kernel must be able to redirect interrupts to the right device library without the need for any complex software logic.

This requirement is simply met for commodity accelerators with a single hardware execution context, since all interrupts belong to the device library using the device. On the other hand, we note that this requirement can be more difficult to meet for non-accelerator devices. One example is network interface cards with a single receive queue. Upon receiving a packet (and hence an interrupt), the device kernel cannot decide at a low level which device library the packet belongs to. This will force the device kernel to employ some management code, e.g., packet filters, to redirect the interrupts, similar to the solutions adopted by the Exokernel [80] and U-Net [81].

A legacy driver does not have this requirement because it incorporates all the management code that uses the interrupts. Device events are then delivered to the application using higher-level API.

7.2 GPU Background

Before presenting our library driver design for GPUs in §7.3, we provide background on the functions of a GPU device driver and GPU hardware as illustrated in Figure 7.2. A GPU driver has three functions: hardware initialization, resource management for applications, and resource isolation.

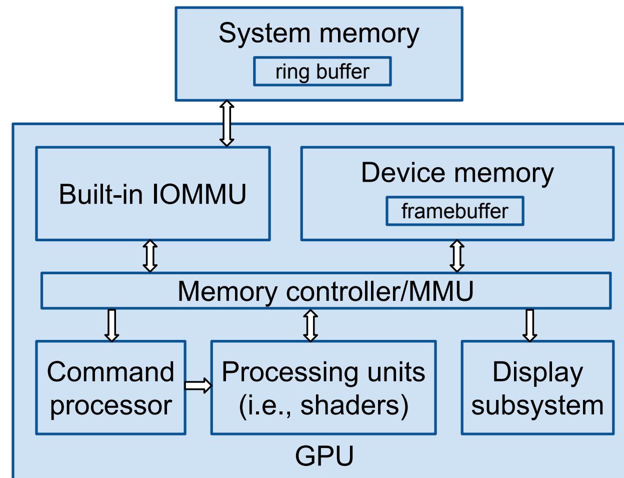


Figure 7.2 : Simplified GPU hardware model.

Initialization

GPU hardware initialization loads the firmware, sets up the GPU's memory controller or MMU in order to configure the address space used by the GPU, enables the interrupts, sets up the command processor (which is later used to dispatch instructions to the GPU), the device power management, and the display subsystem.

Management

Once the GPU hardware is initialized, the driver performs resource management for applications. It implements four important functionalities for this. First, it implements memory management API calls. An application can request three types of memory buffers: buffers on the device memory, buffers on the system memory, and buffers on the system memory accessible to the GPU for DMA. For the latter, the driver allocates a range of GPU addresses for the buffer and programs the GPU's built-in IOMMU to translate from these addresses to their actual physical addresses

on the system memory. Moreover, the driver pins these pages so that they do not get swapped out by the operating system kernel.

Second, the driver accepts GPU instructions from applications and dispatches them to the GPU for execution. Dispatching the instructions in done through a ring buffer. The driver writes the instructions onto the ring buffer, and the command processor unit on the GPU reads them and forwards them to the processing units, i.e., shaders, in the GPU. Note that similar to CPUs, GPUs cache memory accesses and the built-in IOMMU has a TLB for its translations. Therefore, the driver flushes the caches and the TLB as needed.

Third, the driver handles interrupts from the GPU. Most importantly, it processes the interrupts that indicate the end of the execution of the instructions by the processing units. The driver then informs the application so that the application can send in new instructions to be executed.

Finally, the driver implements some API calls for the application to allocate and use a framebuffer and to set the display mode. The framebuffer is a memory buffer holding the content that will be displayed. Through programming the GPU or by directly accessing the memory, the application can write to the framebuffer. The display mode includes the resolution, color depth, aspect ratio, etc. Different applications require to set different modes for the display. Traditionally, the display mode was set through the X server in Linux. However, it was recently moved to the legacy Linux open source drivers and is referred to as Kernel Mode Setting, or KMS. The advantage of KMS is that it allows for better graphics at boot time (before X is even initialized) and during Linux virtual console switching.

Isolation

The driver also enforces isolation between applications by performing appropriate hardware configurations, such as setting up the MMU page tables if supported, or through software checks. For instance, a legacy GPU driver enforces software memory isolation as follows. When allocating memory buffers for an application, the driver only returns an ID of the buffer to the application, which then uses that ID in the instructions that it submits to the driver. The driver then replaces all the IDs with the actual addresses of the buffers and then writes them to the ring buffer.

Glider implements these three functionalities as follows. Initialization is performed in the device kernel. Once the GPU is initialized, the device kernel exports the resources to device libraries so that they can perform resource management in the user space. For isolation, the device kernel leverages hardware properties introduced in §7.1.2. This design reduces the size and attack surface of the TCB and hence improves the system security. The GPU hardware often meets the required hardware properties. It provides either a memory controller or an MMU that can be used to enforce isolation for device-side accesses to its memory. The management registers are often not sensitive and the instruction set has no sensitive instructions. Moreover, with GPUs with a single hardware execution context, interrupts can be simply forwarded to the device library using the GPU.

7.3 Glider: Library Driver for GPUs

In this section, we present Glider, library drivers for the Radeon HD 6450 and Intel Ivy Bridge GPUs based on retrofitting their corresponding legacy drivers. We provide a fully-functional implementation for the Radeon GPU and a proof-of-concept

implementation for the Intel GPU.

7.3.1 Isolation of GPU Resources

We first identify the important GPU hardware resources and elaborate on how the device kernel securely exports them to device libraries.

Processing Units

The device kernel securely exports the GPU processing units, i.e., shaders, by allowing access to the GPU command processor. When binding the GPU to a device library, the device kernel allows the device library to update the command processor's registers that determine the ring buffer location. This enables the device library to allocate and use the ring buffer that it has allocated from its own memory. The device library then populates the ring buffer with instructions from the application and triggers the execution by writing to another register. At revoke time, the device kernel disallows further write to the trigger registers. It then waits for ongoing execution to finish on the GPU before binding the GPU to another device library. The device kernel also takes a snapshot of the registers updated by the device library so that it can write them back the next time it needs to bind the GPU to the same device library. Finally, it resets the command processor and flushes the GPU caches appropriately.

Memory

The operating system allocates system memory pages for a device library through standard syscalls, such as `mmap` and the device kernel allocates device memory for the device library through its API calls (§7.3.2). When using the device, the device kernel needs to guarantee that a device library can only access memory allocated for

it, as discussed in §7.1.2. Here, we provide the implementation details of isolating device-side access to memory.

For system memory, we use the system IOMMU for isolation. The device kernel provides an API call for the device library to map a page into the IOMMU. More specifically, the device library can ask the device kernel to insert a mapping into the IOMMU in order to translate a DMA target address to a given physical page address. To enforce isolation, the device kernel only maps pages that have been allocated for the device library. The device kernel also pins the page into memory so that it does not get swapped out or deallocated by the operating system kernel. Upon revoke, the device kernel stores all the mappings in the IOMMU and replaces them with the mappings for the next device library. The mappings of an IOMMU is in the form of page tables [13], very similar to the page tables used by the CPU MMU. Therefore, similar to a context switch on the CPU, changing the IOMMU mappings can be done very efficiently by only changing the root of the IOMMU page tables.

Alternatively, the GPU's built-in IOMMU can be used for isolation, which is useful for systems without a system IOMMU. To demonstrate this, we used the built-in IOMMU for our Intel GPU library driver. In this case, the device kernel takes full control of the GPU's built-in IOMMU and updates its page tables through the same API call mentioned above.

It is up to the device library when and how much memory it maps in the IOMMU. In our current implementation, the device library allocates about 20 MB of memory at launch time and maps all the pages in the IOMMU. We empirically determined this number to be adequate for our benchmarks. Alternatively, a device library can allocate and map the pages in the IOMMU as needed. This alternative option speeds up the device library's launch process but may result in degraded performance at

runtime.

For the Radeon device memory, we use the GPU memory controller for isolation. Isolating the device memory does not apply to the Intel GPU since it does not have its own memory. The memory controller on the Radeon GPU configures the physical address space seen by the GPU. It sets the range of physical addresses that are forwarded to the GPU memory and those that are forwarded to the system memory (after translation by the built-in IOMMU). By programming the memory controller, the device kernel can effectively segment the GPU memory, one segment per device library. Obviously, the memory segmentation is not flexible in that each device library can only use a single contiguous part of the GPU memory. Changing the segments can only be done using memory ballooning techniques used for memory management for VMs [186], although we have not implemented this yet.

Framebuffer and Displays

The hardware interface for the framebuffer can be securely exported to device libraries. The device kernel allows the device library to have access to the registers that determine the location of the framebuffer in memory. Therefore, the device library can allocate and use its own framebuffer. However, the hardware interface for display mode setting violates the requirement in §7.1.2, forcing us to keep the display mode setting code in the device kernel.

Every display supports a limited number of modes that it can support. However, instead of exposing the mode option with a simple interface, e.g., a single register, GPUs expect the software to configure voltages, frequencies, and clocks of different components on the GPU, e.g., display connectors and encoders, to set a mode, and it is not clear whether such a large hardware interface can be safely exported to un-

strusted software without risking damaging the device. Exacerbating the problem, newer Radeon GPUs have adopted mode setting through GPU BIOS calls. However, BIOS calls can also be used for other sensitive GPU configurations, such as for power management and operating thermal controllers, and we cannot export the BIOS register interface to device libraries securely.

As a result, we keep the display mode setting code in the device kernel. The device kernel exposes this mode setting functionality with a higher level API call to device libraries. This results in a larger TCB size as is reported in §7.4.1. Despite its disadvantage of increasing the TCB size, keeping the mode setting in the device kernel has the advantage that it supports Kernel Mode Setting (§7.2).

7.3.2 The Device Kernel API

The device kernel API include seven calls for device libraries and two calls for the system scheduler. Except for the `set_mode` call, which is GPU-specific (§7.3.1), the rest are generic. Therefore, we expect them to be used for device libraries of other devices as well. They constitute the minimal set of API calls to support device library's secure access to system memory, device memory, and registers. The seven calls for device libraries are as follows:

`void *init_device_lib(void)`: This call is used when a device library is first loaded. The device kernel prepares some read-only memory pages containing the information that the device library needs, such as the display setup, maps them into the device library's process address space, and returns the address of the mapped pages to the device library.

`iommu_map_page(vaddr, iaddr)`, `iommu_unmap_page(vaddr)`: With these two calls, the device library asks the device kernel to map and unmap a memory page to and

from the IOMMU. `vaddr` is the virtual address of the page in the process address space. The device kernel uses this address to find the physical address of the page. `iaddr` is the address that needs to to be translated by the IOMMU to the physical address. This will be the DMA target address issued by the device.

`alloc_device_memory(size)`, `release_device_memory(addr, size)`: With these two calls, the device library allocates and releases the device memory. These two calls are only implemented for GPUs with their own memory.

`int access_register(reg, value, is_write)`: With this call, a device library reads and writes from and to authorized registers. The implementation of this call in the device kernel is simple: it just checks whether the read or write is authorized or not, and if yes, it completes the request. The checking is done by maintaining a list of authorized registers. We implement read and write operations in one API call since their implementation is different only in a few lines of code.

Given that most registers on GPUs are memory-mapped (i.e., MMIO registers), One might wonder why the device kernel does not directly map these registers into the device library's process address space, further reducing the attack surface on the TCB. This is because with such an approach, protection of registers can be enforced at the granularity of a MMIO page, which contains hundreds of registers, not all of them are authorized for access by a device library.

`set_mode(display, mode)`: With this call, a device library asks the device kernel to set the mode on a given display.

We next present the two calls for the system scheduler. Schedulers for GPU resources, such as [29, 136, 178, 179], can be implemented on top of these two API calls.

`bind_device_lib(id)`, `revoke_device_lib(id)`: With these two calls, the sched-

uler asks the device kernel to bind and revoke the GPU resources to and from a device library with a given `id`. Since our GPUs do not support execution preemption (§7.1.1), the revoke call needs to block until the execution on the GPU terminates.

7.3.3 Reusing Legacy Driver Code for Glider

We reuse the Linux open source legacy driver code in Glider, both for the device kernel and the device library, rather than implementing them from the scratch, in order to reduce the engineering effort. Reusing the legacy driver code for device kernel is trivial since the device kernel runs in the kernel as well. However, reusing it for the device library is challenging since the device library is in the user space. We solve this problem by using the User-Mode Linux framework [187]. UML is originally designed to run a Linux operating system in the user space of another Linux operating system and on top of the Linux syscall interface. It therefore provides translations of kernel symbols to their equivalent syscalls, enabling us to compile the device library to run in the user space. We only use part of the UML code base that is needed to provide the translations for the kernel symbols used in our drivers.

Note that the UML normally links the compiled object files into an executable. We, however, link them into a shared library. Linking into a shared library may be challenging on some architectures. This is because assembly code, which is often used in the kernel, is not always position-independent, a requirement for a shared library. We did not experience any such problem for the x86 (32 bit) architecture. The solution to any such potential problem in other architectures is to either rewrite the assembly code to be position-independent or to replace it with equivalent non-assembly code.

It is interesting to understand how system memory allocation works in Glider's device library, which is compiled against the UML symbols. As mentioned in §7.3.1,

we allocate about 20 MB of system memory at the device library's launch. This memory is then managed by the slab allocator of UML, similar to how physical memory is managed by the slab allocator in the kernel. The retrofitted driver code in the device library then allocates system memory from the UML's slab allocator by calling the Linux kernel memory allocation functions, i.e., `kmalloc()` and its derivatives.

7.3.4 Other Implementation Challenges

We solved two other challenges in Glider. First, we replace syscalls for the legacy driver with function calls into equivalent entry points in the device library. Fortunately in the case of GPU, we managed to achieve this by only changing about 20 instances of such syscalls in Linux GPU libraries including the `libdrm`, `xf86-video-ati`, and GalliumCompute [133] libraries. An alternative solution with less engineering effort is to use the `ptrace` utility to intercept and forward the syscalls to the device library. This solution, however, will have noticeable overhead.

Second, we implement fast interrupt delivery to device libraries for good performance. We experimented with three options for interrupt delivery. The first two, i.e., using the OS signals and syscall-based polling, resulted in performance degradation as these primitives proved to be expensive in Linux. The third option that we currently use is polling through shared memory. For example, in the case of the Radeon GPU, in addition to the interrupt, the GPU updates a memory page with the information about the interrupt that was triggered, and the device library can poll on this page. This approach provided fast interrupt delivery so that the interrupts do not become a performance bottleneck. However, as we will show in §7.4.2, it has the disadvantage of increased CPU usage.

We are considering two other options that we believe will provide fast interrupt delivery without the extra CPU usage. The first approach is using upcalls, which allow the kernel to directly invoke a function in user space. The second approach is to use the interrupt remapping hardware available from virtualization hardware extensions. This hardware unit can deliver the interrupts directly to the device library's process, similar to Dune [108].

7.4 Evaluation

We evaluate Glider and show that it improves the system security by reducing the size and attack surface of the TCB. We also show that Glider provides at least competitive performance with a legacy kernel driver, while slightly outperforming it for applications that require extensive interactions with the driver.

7.4.1 Security

We measure the size and attack surface of the TCB, i.e., the whole driver in the case of a legacy driver and the device kernel in the case of Glider. Unlike the legacy driver that supports various GPUs of the same brand, Glider only supports one specific GPU. Therefore, for a fair comparison, we remove the driver code for other GPUs as best as we manually can. This includes the code for other GPU chipsets, other display connectors and encoders, the code for the legacy user mode setting framework not usable on newer GPUs (§7.2), the code for audio support on GPUs, and the code for kernel features not supported in Glider, such as with Linux `debugfs`.

Our results, presented in Table 7.1, show that Glider reduces the TCB size by about 35% and 38% for the Radeon and Intel GPUs, respectively. In the same table, we also show the size of the code in C source files (and not header files). These

numbers show that a large part of the code in Glider TCB are headers, which mainly include constants, such as register layouts. Not including the headers, Glider reduces the TCB size by about 47% and 43% for the Radeon and Intel GPUs.

As discussed in §7.3.1, the display subsystem hardware interface violates the hardware requirements for a library driver resulting in a larger TCB. To demonstrate this, we measure the code size in display-related files in the device kernel. For the Radeon and Intel GPUs, we measure this number to be 19 and 13 kLoC, which is 50% and 54% of the Glider TCB.

We also show the TCB attack surface in Table 7.1. Glider reduces the attack surface by 84% and 90% for the Radeon and Intel GPUs. Glider only exposes 9 and 7 API calls for these GPUs, as described in detail in §7.3.2 (Intel GPUs do not implement the two API calls for device memory). In contrast, the legacy driver exposes 56 and 68 API calls for the same GPUs. These large number of API calls are used for memory management, GPU execution, display mode setting, and inquiring information about the GPU. Glider supports the first two by securely giving a device library access to part of the GPU management interface. It supports mode setting with one API call and supports the information inquiring API either through the constants compiled into the device library or through the read-only information pages mapped into the device library (§7.3.1).

7.4.2 Performance

In this section, we evaluate the performance of Glider for the Radeon GPU using both compute and graphics benchmarks.

For the experiments, we run the drivers inside a 32-bit x86 (with Physical Address Extension) Ubuntu 12.04 operating system running Linux kernel 3.2.0. The machine

		TCB (all files)	TCB (source files)	API calls
Radeon	Legacy	55	34	56
	Glider	36	18	9
	Reduction	**35**%	**47**%	**84**%
Intel	Legacy	39	30	68
	Glider	24	17	7
	Reduction	**38**%	**43**%	**90**%

Table 7.1 : TCB size and attack surface for the legacy and Glider for the Radeon HD 6450 and Intel Ivy Bridge GPUs. The numbers for both TCB columns are in kLoC. The first TCB columns reports LoC in both source and header files. The second TCB column excludes the header files.

has a 3rd generation core i7-3770 (with hyper-threading disabled). We configure the machine with 2 GBs of memory and 2 cores. In order to minimize the effect of the operating system scheduler on our experiments, we isolate one of the cores at boot time using the Linux `isolcpus` command-line boot option. With this option, Linux only schedules kernel threads on the isolated core, but it does schedule user application threads on it unless explicitly asked for. We then pin our benchmarks to the isolated core and set the highest scheduling priority for them. In order to use the system IOMMU for the Radeon GPU, we run the benchmarks inside a Xen VM with the same configurations mentioned above (2 GBs of memory and 2 cores). The Radeon GPU is assigned to the VM using the direct device assignment [13–16].

Compute Benchmarks

We use a matrix multiplication benchmark running on top of the GalliumCompute [133] framework, an open source implementation of OpenCL. We evaluate the

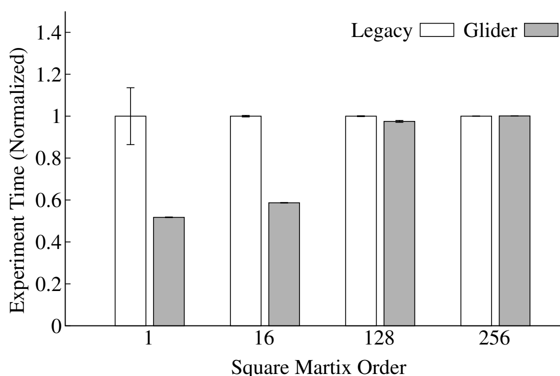


Figure 7.3 : OpenCL matrix multiplication. The x-axis shows the (square) matrix order. The y-axis is the time to execute the multiplication on the GPU, normalized to the average time by the legacy driver.

performance of multiplying square matrices of varying orders. For each experiment, we use a host program that populates input matrices and launches the compute kernel on the GPU. The programs repeats this for 1000 iterations in a loop and outputs the average time for a single iteration. We discard the first iteration to avoid the effect of Glider's launch time overhead (characterized in §7.4.3) and we do not include the time to compile the OpenCL kernel. We then repeat the experiment 5 times for each matrix size and report the average and standard deviation.

Figure 7.3 shows the results, normalized to the average performance by the legacy driver in each case. It shows that Glider outperforms the legacy driver for smaller matrix sizes, while providing competitive performance for all other sizes. This is because for large matrix sizes, the majority of the time is spent on transferring the data between the system and device memory and on the GPU executing the kernel. Consequently, the driver design does not impact the performance noticeably. For smaller sizes, on the other hand, the overhead of application's interaction with the

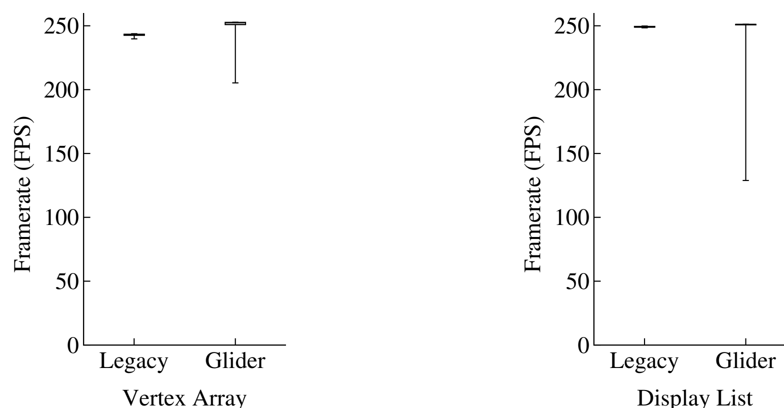


Figure 7.4 : Graphics performance. The box plots show 90% percentile, 10% percentile, and median. The whiskerbars depicts maximum and minimum. The box plots are highly concentrated around the mean.

driver becomes more significant.

Graphics Benchmarks

For graphics, we use two OpenGL benchmarks running on top of the MESA open source implementation of OpenGL [188]. Both benchmarks draw a teapot, and update the screen as fast as possible. One benchmark uses the Vertex Array API [189] (the immediate mode) and the other uses the Display List API [132]. In each experiment, we run the benchmark for 5 minutes, recording the framerate every second. We discard the first 5 frames to avoid the effect of Glider's launch time overhead (characterized in §7.4.3). We repeat each experiment three times.

The results, shown in Figure 7.4, demonstrates that Glider achieves similar performance as the legacy driver for the Display List benchmark but outperforms the legacy driver for the Vertex Array one. In order to explain these results, it is important to understand these two OpenGL API's. With Vertex Arrays, the program

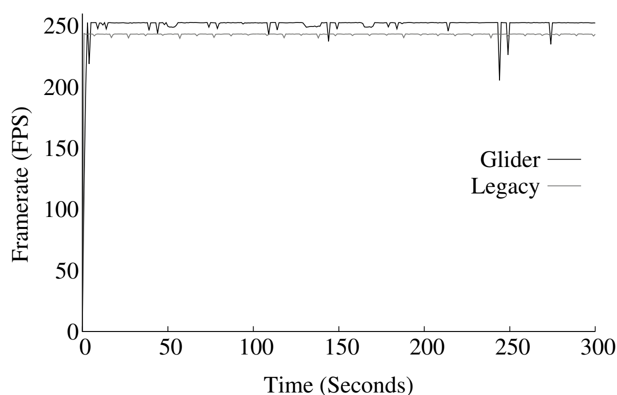


Figure 7.5 : A sample run of the Vertex Array benchmark. The framerate is measured every second. The figure shows the rare drops of framerate and the slow start of Glider.

needs to send the vertices information to the device for every frame, which requires interacting with the driver quite significantly. On the other hand, the Display List API allows this information to be cached by the device in order to avoid resending from the application. Intuitively, Glider improves the performance for the benchmark with more driver interactions, or the Vertex Array.

We also measure the CPU usage for both drivers when running these benchmarks. Our results show that the legacy driver consumes 53.7% and 41.7% of the CPU time for the Vertex Array and Display List benchmarks, respectively, whereas Glider consumes 75.3% and 71.6% of the CPU time for the same benchmarks. The extra CPU usage of Glider is due to polling the memory for interrupts (§7.3.4). One might wonder whether the extra CPU usage is the source of the performance improvement. To investigate this, we attempted to employ similar polling methods in the legacy driver, but failed to improve the performance. Therefore, we are convinced that Glider's performance improvement is due to eliminating the application and driver

interactions' overhead. We also report the performance when using syscall-based polling in the device library, which incurs delay in delivering the interrupt to the device library. With this method, for the Vertex Array and Display List benchmarks, respectively, Glider consumes 54.3% and 35.3% of the CPU time, and achieves 193.3 and 231.5 median frames per second, which are noticeably lower compared to when the device library polls the memory for interrupts.

Figure 7.4 also shows that Glider achieves a noticeably lower minimum framerate, although the minimum happens rarely. To further demonstrate this, we show a sample run of the Vertex Array benchmark in Figure 7.5. Our investigation shows that the performance drop is due to OS scheduling despite our attempts to minimize such effect.

7.4.3 Library Driver Overheads

We measure two important overheads of Glider: the device library's launch time overhead and the device kernel's switch time, i.e., the time it takes the core to revoke the GPU from one device library and bind it to another.

Figure 7.5 illustrates the effect of the device library's launch time overhead on the graphics benchmark reported before. It shows that the performance of Glider is inferior to that of the legacy driver in the first few seconds. There are two sources for this overhead: the device library's initialization overhead and the UML's slab allocator's initialization overhead. The former is the time it takes the device library to initialize itself so that it is ready to handle requests from the application. We measure this time to have an average/standard deviation of of 109 ms/7 ms and 66 ms/0.3 ms for the Radeon and Intel GPUs, respectively. The latter is because the slab allocator of UML (§7.3.3) takes some time to populate its own cache, very similar to

the suboptimal performance of the slab allocator in the kernel at system boot time.

We also measure the switch time in the device kernel, as defined above. We measure this time to have an average and standard deviation of 42 μs and 5 μs for the Radeon GPU (we have not yet implemented this feature for the Intel GPU). The switch time consists of the device kernel taking a snapshot of GPU registers updated by the current device library, writing the register values for the new device library, changing the IOMMU mappings, resetting the command processor, and flushing the GPU caches and the built-in IOMMU TLB. These measurements show that changing the GPU binding can be done fairly quickly.

7.4.4 Engineering Effort

As mentioned, we build Glider by retrofitting the Linux open source drivers as a baseline. We added about 4 kLoC and 2 kLoC for the Radeon and Intel library drivers, respectively. These changes were applied to 49 and 30 files, and we added two new files in each case. We have implemented both the device kernel and the device library on the same driver, and the two are differentiated at compile time. While we believe that reusing the legacy driver code significantly reduced our engineering effort compared to developing from scratch, we note that implementing the library drivers still required noticeable effort. The main source of difficulty was gaining a deep understanding of the GPU internals and its device driver with 10s thousands of lines of code. Fortunately, our experience with the Radeon GPU library driver made it easier for us to prototype the Intel GPU library driver. We therefore believe that experienced driver developers can develop library drivers without prohibitive engineering effort.

7.5 Discussions

7.5.1 Pros and Cons of Library Drivers

Other than improved security and performance, library drivers have three other advantages and three disadvantages. The advantages are as follows. First, library drivers allow each application to customize its own device library. For example, applications can trade-off the initial cost of allocating a pool of memory buffers with the runtime cost of allocating buffers as needed. Second, library drivers greatly simplify driver development because the developer can use user space debugging frameworks or high-level languages, none of them are available in the kernel. Moreover, developers can use a unified debugging framework for both the application and the device library, which can greatly help with timing bugs and data races. Third, library drivers improve memory usage accounting by the operating system. Legacy drivers for some devices, such as GPUs, implement their own memory management, which gives applications a side channel to allocate parts of the system memory, invisible to the operating system kernel for accounting. In contrast, with a library driver, all the system memory allocated by a device library is through the standard operating system memory management interface, e.g., the `mmap` and `brk` syscalls in Linux.

The library driver design has the following disadvantages. First, library drivers complicate multi-process programming. For example, sharing memory buffers between processes is easily done in a legacy driver, but requires peer-to-peer communication between the device libraries in a library driver. Second, library drivers incurs launch time overheads to applications. We evaluated this overhead in §7.4.3 and showed that it is not significant. Third, depending on the device, a library driver may achieve coarser-grained device memory sharing for applications. A legacy driver can

share the device memory at the granularity of buffers, whereas with devices without paging support for their own memory, such as the Radeon GPU in our setup, device memory can only be shared between device libraries using contiguous segments.

7.5.2 Devices with Multiple Hardware Execution Contexts

For devices with multiple execution contexts, the device kernel should bind and revoke the hardware contexts to device libraries independently. This puts new requirements on the device and platform hardware as we will explain next.

First, the hardware management interface for different contexts must be non-overlapping and isolated. That is, registers for each context should be separate and instructions should only affect the resources of the given context. Also, device DMA requests must be attributable to different contexts at a low level so that an IOMMU can be used to isolate them.

As an example, self-virtualized devices [18, 19, 21, 22] export multiple virtual devices, each of which can be separately assigned to a VM. As a result, they readily provide all the hardware primitives for the device kernel to bind different virtual devices to different device libraries.

7.5.3 Radeon Instruction Set

We allow a device library to directly dispatch instructions to our Radeon and Intel GPUs since the instructions are not sensitive. We noticed a curious case with the Radeon GPU's instruction set though. Using the instructions, an application can write to the registers of GPU processing units in order to program and use them. Fortunately, it is not possible to use the instructions to write to registers of components of the GPU that affect the isolation, such as the memory controller. However,

we noticed that the Linux open source Radeon driver returns error when inspecting the instructions submitted by an applications and detecting accesses to large register numbers, which surprisingly correspond to no actual registers according to the AMD reference guide [190] and the register layout in the driver itself. Therefore, we believe that this is a simple correctness check by the driver and not a security concern. Therefore, we currently do not employ such a check in device kernel, although that is a possibility. Adding the check to the device kernel will increase the TCB size by about a few kLoC, but should not degrade the performance compared to what was reported in this paper since we already performed these checks in the device library in our benchmarks (although it was not necessary).

Chapter 8

Conclusions

In this thesis, we presented our design and implementation of I/O servers using the device file boundary. We showed that remote access to I/O devices at this boundary provides important properties: low engineering effort, support for legacy applications and I/O devices, support for all functions of I/O devices, support for multiple clients, and high performance. Moreover, we demonstrated important system services of I/O servers using the aforementioned boundary: I/O sharing between virtual machines, i.e., I/O virtualization, I/O sharing between mobile systems, and I/O sharing between servers in a datacenter. We reported three systems, called Paradice, Rio, and Glance that implement these system services, and showed that they provide the important properties mentioned earlier. Finally, to enhance the security of I/O servers, we presented two solutions, one that provides fault and device data isolation between clients in face of a compromised device driver, and a complementary solution, i.e., library drivers, that enhances the security of I/O servers by reducing the size and attack surface of the device driver TCB.

Bibliography

[1] R. W. Scheifler and J. Gettys, "The X Window System," *ACM Transactions on Graphics (TOG)*, 1986.

[2] "Android IP Webcam application." https://play.google.com/store/apps/details?id=com.pas.webcam&hl=en.

[3] "Android Wi-Fi Speaker application." https://play.google.com/store/apps/details?id=pixelface.android.audio&hl=en.

[4] "MightyText application." http://mightytext.net.

[5] 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.

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

[7] K. Tian, Y. Dong, and D. Cowperthwaite, "A Full GPU Virtualization Solution with Mediated Pass-Through," in *Proc. USENIX ATC*, 2014.

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

[9] 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.

[10] 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.

[11] Z. Qi, J. Yao, C. Zhang, M. Yu, Z. Yang, and H. Guan, "VGRIS: Virtualized GPU Resource Isolation and Scheduling in Cloud Gaming," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.

[12] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *USENIX ATC, FREENIX Track*, 2005.

[13] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert, "Intel Virtualization Technology for Directed I/O," *Intel Technology Journal*, 2006.

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

[15] 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.

[16] 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.

[17] M. Liu, T. Li, N. Jia, A. Currid, and V. Troy, "Understanding the virtualization" Tax" of scale-out pass-through GPUs in GaaS clouds: An empirical study," in *Proc. IEEE High Performance Computer Architecture (HPCA)*, 2015.

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

[19] 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.

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

[21] 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.

[22] "NVIDIA GRID K1 and K2 Graphics-Accelerated Virtual Desktops and Applications. NVIDIA White Paper."

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

[24] 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.

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

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

[27] L. Shi, H. Chen, J. Sun, and K. Li, "vCUDA: GPU-accelerated High-performance Computing in Virtual Machines," *IEEE Transactions on Computers*, 2012.

[28] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, "GViM: GPU-Accelerated Virtual Machines," in *Proc. ACM Workshop on System-level Virtualization for High Performance Computing (HPCVirt)*, 2009.

[29] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan, "Pegasus: Coordinated Scheduling for Virtualized Accelerator-Based Systems," in *USENIX ATC*, 2011.

[30] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A GPGPU Transparent Virtualization Component for High Performance Computing Clouds," in *Proc. Springer Euro-Par Parallel Processing*, 2010.

[31] M. Gottschlag, M. Hillenbrand, J. Kehne, J. Stoess, and F. Bellosa, "LoGV: Low-Overhead GPGPU Virtualization," in *Proc. IEEE High Performance Computing and Communications & IEEE Embedded and Ubiquitous Computing (HPCC_EUC)*, 2013.

[32] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, "GPUvm: Why Not Virtualizing GPUs at the Hypervisor?," in *Proc. USENIX ATC*, 2014.

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

[34] "Miracast." https://www.wi-fi.org/sites/default/files/uploads/wp_Miracast_Industry_20120919.pdf.

[35] "Chromecast." http://www.google.com/intl/en/chrome/devices/chromecast/.

[36] "Apple Continuity." https://www.apple.com/ios/whats-new/continuity/.

[37] T. Das, P. Mohan, V. Padmanabhan, R. Ramjee, and A. Sharma, "PRISM: Platform for Remote Sensing Using Smartphones," in *Proc. ACM MobiSys*, 2010.

[38] Y. Lee, Y. Ju, C. Min, S. Kang, I. Hwang, and J. Song, "CoMon: Cooperative Ambience Monitoring Platform with Continuity and Benefit Awareness," in *Proc. ACM MobiSys*, 2012.

[39] R. A. Baratto, L. Kim, and J. Nieh, "THINC: A Remote Display Architecture for Thin-Client Computing," in *Proc. ACM SOSP*, 2005.

[40] B. C. Cumberland, G. Carius, and A. Muir, *Microsoft Windows NT Server 4.0, Terminal Server Edition: Technical Reference.* Microsoft Press, 1999.

[41] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper, "Virtual Network Computing," *IEEE Internet Computing*, 1998.

[42] "Citrix XenDesktop." http://www.citrix.com/products/xendesktop/go/overview.html.

[43] B. K. Schmidt, M. S. Lam, and J. D. Northcutt, “The Interactive Performance of SLIM: A Stateless, Thin-Client Architecture,” in *Proc. ACM SOSP*, 1999.

[44] “Network File System.” `http://etherpad.tools.ietf.org/html/rfc3530`.

[45] A. P. Rifkin, M. P. Forbes, R. L. Hamilton, M. Sabrio, S. Shah, and K. Yueh, “RFS Architectural Overview,” in *Proc. USENIX Conference*, 1986.

[46] P. J. Leach and D. Naik, “A Common Internet File System (CIFS/1.0) Protocol,” *IETF Network Working Group RFC Draft*, 1997.

[47] A. Hari, M. Jaitly, Y. J. Chang, and A. Francini, “The Switch Army Smartphone: Cloud-based Delivery of USB Services,” in *Proc. ACM MobiHeld*, 2011.

[48] “Digi International: AnywhereUSB.” `http://www.digi.com/products/usb/anywhereusb.jsp`.

[49] “USB Over IP.” `http://usbip.sourceforge.net/`.

[50] “Wireless USB.” `http://www.usb.org/wusb/home/`.

[51] “Intel WiDi.” `http://www.intel.com/content/www/us/en/architecture-and-technology/intel-wireless-display.html`.

[52] “Web Services on Devices: Devices That Are Controlled on the Network.” `http://msdn.microsoft.com/en-us/library/windows/desktop/aa826001(v=vs.85).aspx`.

[53] “Dropcam.” `https://www.dropcam.com/`.

[54] F. Lamberti and A. Sanna, "A Streaming-based Solution for Remote Visualization of 3D Graphics on Mobile Devices," *IEEE Transaction on Visualization and Computer Graphics*, 2007.

[55] J. Mysore, V. Vasudevan, J. Almaula, and A. Haneef, "The Liquid Media System – A Multi-device Streaming Media Orchestration Framework," in *UbiComp Workshop on Multi-Device Interfaces for Ubiquitous Peripheral Interaction*, 2003.

[56] "Multi-Root I/O Virtualization and Sharing 1.0 Specification, PCI-SIG," 2008.

[57] C. Tu, C. Lee, and T. Chiueh, "Secure I/O Device Sharing among Virtual Machines on Multiple Hosts," in *Proc. ACM ISCA*, 2013.

[58] S. Xiao, P. Balaji, Q. Zhu, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, and W. Feng, "VOCL: An Optimized Environment for Transparent Virtualization of Graphics Processing Units," in *Proc. IEEE Innovative Parallel Computing (InPar)*, 2012.

[59] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "rCUDA: Reducing the Number of GPU-Based Accelerators in High Performance Clusters," in *Proc. IEEE High Performance Computing and Simulation (HPCS)*, 2010.

[60] A. M. Merritt, V. Gupta, A. Verma, A. Gavrilovska, and K. Schwan, "Shadowfax: Scaling in Heterogeneous Cluster Systems via GPGPU Assemblies," in *Proc. ACM Int. Workshop on Virtualization Technologies in Distributed Computing (VTDC)*, 2011.

[61] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W. W. Hwu,

"An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems," in *Proc. ACM ASPLOS*, 2010.

[62] K. Elphinstone and G. Heiser, "From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels?," in *Proc. ACM SOSP*, 2013.

[63] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser, "User-Level Device Drivers: Achieved Performance," *Journal of Computer Science and Technology*, vol. 20, no. 5, 2005.

[64] A. Forin, D. Golub, and B. N. Bershad, "An I/O System for Mach 3.0," in *Proc. USENIX Mach Symposium*, 1991.

[65] D. B. Golub, G. G. Sotomayor, and F. L. Rawson, III, "An Architecture for Device Drivers Executing As User-Level Tasks," in *Proc. USENIX MACH III Symposium*, 1993.

[66] D. S. Ritchie and G. W. Neufeld, "User Level IPC and Device Management in the Raven Kernel," in *USENIX Microkernels and Other Kernel Architectures Symposium*, 1993.

[67] S. Boyd-Wickizer and N. Zeldovich, "Tolerating malicious device drivers in Linux," in *Proc. USENIX ATC*, 2010.

[68] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha, "The Design and Implementation of Microdrivers," in *Proc. ACM ASPLOS*, 2008.

[69] G. C. Hunt, "Creating User-Mode Device Drivers with a Proxy," in *Proc. USENIX Windows NT Workshop*, 1997.

[70] 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.

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

[72] 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.

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

[74] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer, "SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques," in *Proc. USENIX OSDI*, 2006.

[75] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune, "Building Verifiable Trusted Path on Commodity x86 Computers," in *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2012.

[76] Z. Zhou, M. Yu, and V. D. Gligor, "Dancing with Giants: Wimpy Kernels for On-demand Isolated I/O," in *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2014.

[77] L. Schaelicke, "Architectural Support for User-Level I/O," *Doctoral thesis, University of Utah*, 2001.

[78] I. A. Pratt, "The User-Safe Device I/O Architecture," *Doctoral thesis, University of Cambridge*, 1997.

[79] I. Pratt and K. Fraser, "Arsenic: A User-Accessible Gigabit Ethernet Interface," in *Proc. IEEE INFOCOM*, 2001.

[80] D. R. Engler, M. F. Kaashoek, and J. O. Jr., "Exokernel: an Operating System Architecture for Application-Level Resource Management," in *Proc. ACM SOSP*, 1995.

[81] T. Von Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," in *Proc. ACM SOSP*, 1995.

[82] J. Stecklina, "Shrinking the Hypervisor One Subsystem at a Time: A Userspace Packet Switch for Virtual Machines," in *Proc. ACM VEE*, 2014.

[83] M. Li, Z. Zha, W. Zang, M. Yu, P. Liu, and K. Bai, "Detangling Resource Management Functions from the TCB in Privacy-Preserving Virtualization," in *Proc. European Symposium on Research in Computer Security (ESORICS)*, 2014.

[84] "Windows User-Mode Driver Framework (UMDF)." `http://msdn.microsoft.com/en-us/library/windows/hardware/ff560442(v=vs.85).aspx`.

[85] "Functions Registered by DriverEntry of Display Miniport Driver in Windows Display Driver Model (WDDM)." `http://msdn.microsoft.com/en-us/`

`library/windows/hardware/ff566463(v=vs.85).aspx`.

[86] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the Library OS from the Top Down," in *Proc. ACM ASPLOS*, 2011.

[87] J. Howell, B. Parno, and J. Douceur, "Embassies: Radically Refactoring the Web," in *Proc. USENIX NSDI*, 2013.

[88] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch, "Leveraging Legacy Code to Deploy Desktop Applications on the Web.," in *Proc. USENIX OSDI*, 2008.

[89] "Understanding Bromium Micro-virtualization for Security Architects. Bromium White Paper,"

[90] F. M. David, E. Chan, J. C. Carlyle, and R. H. Campbell, "CuriOS: Improving Reliability through Operating System Structure," in *Proc. USENIX OSDI*, 2008.

[91] J. Criswell, N. Dautenhahn, and V. Adve, "Virtual Ghost: Protecting Applications from Hostile Operating Systems," in *Proc. ACM ASPLOS*, 2014.

[92] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels," in *Proc. IEEE Security and Privacy (S&P)*, 2014.

[93] G. Kyriazis, "Heterogeneous System Architecture: A Technical Review, AMD White Paper," 2012.

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

[95] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS Distributed Operating System," 1983.

[96] "openMosix: Single System Image for Linux." `http://openmosix.sourceforge.net`.

[97] "OpenSSI: Single System Image for Linux." `http://openssi.org/cgi-bin/view?page=openssi.html`.

[98] C. Morin, R. Lottiaux, G. Vallée, P. Gallard, G. Utard, R. Badrinath, and L. Rilling, "Kerrighed: a Single System Image Cluster Operating System for High Performance Computing," in *Proc. Springer Euro-Par Parallel Processing*, 2003.

[99] C. Morin, R. Lottiaux, G. Vallée, P. Gallard, D. Margery, J. Berthou, and I. D. Scherson, "Kerrighed and Data Parallelism: Cluster Computing on Single System Image Operating Systems," in *Proc. IEEE Cluster Computing (CLUSTER)*, 2004.

[100] V. S. Sunderam, "PVM: A Framework for Parallel Distributed Computing," *Concurrency: practice and experience*, 1990.

[101] M. Chapman and G. Heiser, "vNUMA: A Virtual Shared-Memory Multiprocessor," in *Proc. USENIX ATC*, 2009.

[102] J. Flinn, "Cyber Foraging: Bridging Mobile and Cloud Computing," *Synthesis Lectures on Mobile and Pervasive Computing*, 2012.

[103] R. K. Balan, D. Gergle, M. Satyanarayanan, and J. Herbsleb, "Simplifying Cyber Foraging for Mobile Devices," in *Proc. ACM MobiSys*, 2007.

[104] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, “MAUI: Making Smartphones Last Longer with Code Offload,” in *Proc. ACM MobiSys*, 2010.

[105] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, “CloneCloud: Elastic Execution between Mobile Device and Cloud,” in *Proc. ACM EuroSys*, 2011.

[106] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, “COMET: Code Offload by Migrating Execution Transparently,” in *Proc. USENIX OSDI*, 2012.

[107] M. Ben-Yehuda, O. Peleg, O. Agmon Ben-Yehuda, I. Smolyar, and D. Tsafrir, “The nonkernel: A Kernel Designed for the Cloud,” in *Proc. ACM APSYS*, 2013.

[108] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazieres, and C. Kozyrakis, “Dune: Safe User-level Access to Privileged CPU Features,” in *Proc. USENIX OSDI*, 2012.

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

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

[111] A. Amiri Sani, K. Boos, S. Qin, and L. Zhong, “I/O Paravirtualization at the Device File Boundary,” in *Proc. ACM ASPLOS*, 2014.

[112] A. Gordon, N. Har’El, A. Landau, M. Ben-Yehuda, and A. Traeger, “Towards Exitless and Efficient Paravirtual I/O,” in *Proc. SYSTOR*, 2012.

[113] 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.

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

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

[116] "Privilege escalation using NVIDIA GPU driver bug (CVE-2012-4225)." `http://www.securelist.com/en/advisories/50085`.

[117] "Integer overflow in Linux DRM driver (CVE-2012-0044)." `https://bugzilla.redhat.com/show_bug.cgi?id=772894`.

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

[119] 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.

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

[121] "CLOC." `http://cloc.sourceforge.net/`.

[122] "Paradice's source code." `http://paradice.recg.rice.edu`.

[123] 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.

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

[125] M. Weiser, "Program slicing," in *Proc. IEEE Int. Conf. on Software engineering*.

[126] "Tremulous." http://www.tremulous.net/.

[127] "OpenArena." http://openarena.ws/smfnews.php.

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

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

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

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

[132] "OpenGL Microbenchmark: Display List." http://www.songho.ca/opengl/gl_displaylist.html.

[133] "GalliumCompute." http://dri.freedesktop.org/wiki/GalliumCompute/.

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

[135] "Guvcview." http://guvcview.sourceforge.net/.

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

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

[138] A. Amiri Sani, K. Boos, M. Yun, and L. Zhong, "Rio: A System Solution for Sharing I/O between Mobile Systems," in *Proc. ACM MobiSys*, 2014.

[139] "Rio's video demo." `http://www.ruf.rice.edu/~mobile/rio.html`.

[140] "Applications for taking self photos." `http://giveawaytuesdays.wonderhowto.com/inspiration/10-iphone-and-android-apps-for-taking-self-portraits-0129658/`.

[141] `http://www.theverge.com/2011/06/09/google-voice-skype-imessage-and-the-death-of-the-phone-number/`.

[142] R. Raskar, J. Tumblin, A. Mohan, A. Agrawal, and Y. Li, "Computational Photography," in *Proc. STAR Eurographics*, 2006.

[143] L. Yuan, J. Sun, L. Quan, and H. Shum, "Image Deblurring with Blurred/Noisy Image Pairs," *ACM Transactions on Graphics (TOG)*, 2007.

[144] B. Wilburn, N. Joshi, V. Vaish, E. Talvala, E. Antunez, A. Barth, A. Adams, M. Horowitz, and M. Levoy, "High Performance Imaging Using Large Camera Arrays," *ACM Transactions on Graphics (TOG)*, 2005.

[145] S. C. Park, M. K. Park, and M. G. Kang, "Super-Resolution Image Reconstruction: a Technical Overview," *IEEE Signal Processing Magazine*, 2003.

[146] K. Li, "Ivy: A Shared Virtual Memory System for Parallel Computing," in *Proc. Int. Conf. Parallel Processing*, 1988.

[147] G. S. Delp, "The Architecture and Implementation of MEMNET: a High-Speed Shared Memory Computer Communication Network," *Doctoral thesis, University of Delaware*, 1988.

[148] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and Performance of Munin," in *Proc. ACM SOSP*, 1991.

[149] S. Zhou, M. Stumm, K. Li, and D. Wortman, "Heterogeneous Distributed Shared Memory," *IEEE Transactions on Parallel and Distributed Systems*, 1992.

[150] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Fine-Grain Access Control for Distributed Shared Memory," in *Proc. ACM ASPLOS*, 1994.

[151] ARM, "Architecture Reference Manual, ARMv7-A and ARMv7-R edition," *ARM DDI*, vol. 0406A, 2007.

[152] F. X. Lin, Z. Wang, and L. Zhong, "K2: A Mobile Operating System for Heterogeneous Coherence Domains," in *Proc. ACM ASPLOS*, 2014.

[153] R. Woodings and M. Pandey, "WirelessUSB: a Low Power, Low Latency and Interference Immune Wireless Standard," in *Proc. IEEE Wireless Communications and Networking Conference (WCNC)*, 2006.

[154] "Rio's source code." `http://rio.recg.rice.edu`.

[155] "Linux ksocket." `http://ksocket.sourceforge.net/`.

[156] Texas Instruments, "Architecture Reference Manual, OMAP4430 Multimedia Device Silicon Revision 2.x," vol. SWPU231N, 2010.

[157] "Android ION Memory Allocator." http://lwn.net/Articles/480055/.

[158] "Wireless LAN at 60 GHz - IEEE 802.11ad Explained," in *Agilent White Paper*.

[159] "802.11ac: The Fifth Generation of Wi-Fi," in *Cisco White Paper*, 2012.

[160] "Monsoon Power Monitor." http://www.msoon.com/LabEquipment/PowerMonitor/.

[161] K. C. Barr and K. Asanović, "Energy-Aware Lossless Data Compression," in *Proc. ACM MobiSys*, 2003.

[162] J. Liu and L. Zhong, "Micro Power Management of Active 802.11 Interfaces," in *Proc. ACM MobiSys*, 2008.

[163] J. Gilger, J. Barnickel, and U. Meyer, "GPU-Acceleration of Block Ciphers in the OpenSSL Cryptographic Library," in *Information Security*, Springer Berlin Heidelberg, 2012.

[164] G. Agosta, A. Barenghi, F. De Santis, A. Di Biagio, and G. Pelosi, "Fast Disk Encryption through GPGPU Acceleration," in *Proc. IEEE International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2009.

[165] G. Liu, H. An, W. Han, G. Xu, P. Yao, M. Xu, X. Hao, and Y. Wang, "A Program Behavior Study of Block Cryptography Algorithms on GPGPU," in *Proc. IEEE International Conference on Frontier of Computer Science and Technology (FCST)*, 2009.

[166] S. N. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc, "GPU-based Video Feature Tracking and Matching," in *Workshop on Edge Computing Using New*

Commodity Architectures (EDGE), 2006.

[167] F. Rößler, E. Tejada, T. Fangmeier, T. Ertl, and M. Knauff, "GPU-based Multi-Volume Rendering for the Visualization of Functional Brain Images," in *Proc. SimVis*, Publishing House, 2006.

[168] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens, "Efficient Computation of Sum-Products on GPUs Through Software-Managed Cache," in *Proc. ACM International Conference on Supercomputing (ICS)*, 2008.

[169] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, "Accelerating Molecular Modeling Applications with Graphics Processors," *Journal of Computational Chemistry*, 2007.

[170] S. S. Stone, J. P. Haldar, S. C. Tsao, W.-m. Hwu, B. P. Sutton, Z.-P. Liang, *et al.*, "Accelerating Advanced MRI Reconstructions on GPUs," *Journal of Parallel and Distributed Computing*, 2008.

[171] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," in *Proc. ACM/IEEE ISCA*, 2014.

[172] "Intel Virtualization Technology for Directed I/O, Architecture Specification," vol. Order Number: D51397-006, Rev. 2.2, September 2013.

[173] "Stack buffer overflow in NVIDIA display driver service in Windows 7."

https://www.securityweek.com/researcher-unwraps-dangerous-nvidia-driver-exploit- christmas-day.

[174] "Unprivileged GPU access vulnerability using NVIDIA driver bug (CVE-2013-5987)." http://nvidia.custhelp.com/app/answers/detail/a_id/3377/~/unprivileged-gpu-access-vulnerability---cve-2013-5987.

[175] "Privilege escalation using an exploit in Linux NVIDIA binary driver." http://seclists.org/fulldisclosure/2012/Aug/4.

[176] "WebGL Security." http://www.khronos.org/webgl/security/.

[177] J. Menon, M. De Kruijf, and K. Sankaralingam, "iGPU: Exception Support and Speculative Execution on GPUs," in *Proc. IEEE ISCA*, 2012.

[178] K. Menychtas, K. Shen, and M. L. Scott, "Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators," in *Proc. ACM ASPLOS*, 2014.

[179] K. Menychtas, K. Shen, and M. L. Scott, "Enabling OS Research by Inferring Interactions in the Black-Box GPU Stack," in *Proc. USENIX ATC*, 2013.

[180] L. Soares and M. Stumm, "FlexSC: Flexible System Call Scheduling with Exception-Less System Calls," in *Proc. USENIX OSDI*, 2010.

[181] A. Ltd, "ARM Cortex-A15 Technical Reference Manual," *ARM DDI*, vol. 0438C, 2011.

[182] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural Support for Address Translation on GPUs," in *Proc. ACM ASPLOS*, 2014.

[183] M. Kościelnicki, "NVIDIA Hardware Documentation." https://media.readthedocs.org/pdf/envytools/latest/envytools.pdf.

[184] T. Instruments, "KeyStone Architecture Multicore Shared Memory Controller (MSMC) - User Guide," vol. Literature Number: SPRUGW7A, 2011.

[185] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, 1974.

[186] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," *ACM SIGOPS Operating Systems Review*, 2002.

[187] J. Dike, "User-Mode Linux," in *Proc. Ottawa Linux Symposium*, 2001.

[188] "Mesa." http://www.mesa3d.org/.

[189] "OpenGL Microbenchmark: Vertex Array." http://www.songho.ca/opengl/gl_vertexarray.html.

[190] Advanced Micro Devices Inc., "Radeon Evergreen 3D Register Reference Guide, Revision 1.0." http://www.x.org/docs/AMD/old/evergreen_3D_registers_v2.pdf, 2011.