PAPER

Virtualizing Graphics Architecture of Android Mobile Platforms in KVM/ARM Environment

Sejin PARK[†a)], **Byungsu PARK**[†b)], **Unsung LEE**[†c)], ***Nonmembers, and*** **Chanik PARK**[†d)], ***Member***

SUMMARY With the availability of virtualization extension in mobile processors, e.g. ARM Cortex A-15, multiple virtual execution domains are efficiently supported in a mobile platform. Each execution domain requires high-performance graphics services for full-featured user interfaces such as smooth scrolling, background image blurring, and 3D images. However, graphics service is hard to be virtualized because multiple service components (e.g. ION and Fence) are involved. Moreover, the complexity of Graphical Processing Unit (GPU) device driver also makes harder virtualizing graphics service. In this paper, we propose a technique to virtualize the graphics architecture of Android mobile platform in KVM/ARM environment. The Android graphics architecture relies on underlying Linux kernel services such as the frame buffer memory allocator ION, the buffer synchronization service Fence, GPU device driver, and the display synchronization service VSync. These kernel services are provided as device files in Linux kernel. Our approach is to para-virtualize these device files based on a split device driver model. A major challenge is to translate guest-view of information into host-view of information, e.g. memory address translation, file descriptor management, and GPU Memory Management Unit (MMU) manipulation. The experimental results show that the proposed graphics virtualization technique achieved almost 84%-100% performance of native applications.

key words: *GPU virtualization, mobile virtualization, KVM, para-virtualization*

1. Introduction

Traditionally, virtualization technique was server-side technology to maximize resource utilization but this technology has been begun to be applied to mobile environments, e.g., Xen on ARM [1] and KVM/ARM [2]. Mobile virtualization is getting attention in various areas such as enterprise environment, financial apps, private environment and so on. In case of enterprise environment, Gartner predicts half of employers will require employees to supply their own device for work purposes by 2017 [3]. This "bring your own device (BYOD)" environment faces various challenges. For instance, a compromised app can be installed in a user's mobile device. If a user accesses corporate data using the device, the data can be leaked or crashed by the compromised app. In case of financial app or mobile wallet, they cannot work normally without security concern [4]. Mobile virtualization technologies such as [1], [2] effectively handle these critical issues owing to their isolation functionality. There are further cases for mobile virtualization. A user may want to have his or her own multiple environments for various purposes. In this case, mobile virtualization technique is also the most powerful solution. However, there is a significant problem in current mobile virtualization technique: high-performance graphics support. In contrast to server environment that uses text-based terminal or low performance basic GUI, high performance graphics support is fundamental in a mobile environment. Basically, a mobile device is user-interactive and it should support fast user responsiveness. This is achieved by high performance graphics services including GPU device and relevant components. Note that, most mobile applications require GPU-accelerated graphics operations since their base SDK is based on OpenGL ES [5]. Even for basic smooth scrolling operation, hardware accelerated 2D rendering pipeline is required. However, such graphics services are poorly supported in current mobile virtualization environment due to their limited controllability and information.

In this paper, we propose a virtualization technique for the Android graphics architecture, which relies on underlying Linux kernel services such as the ION memory allocator, the Fence synchronization service, GPU device drivers, and display device drivers. Such kernel services are provided in Linux as device files. Therefore, we para-virtualize device files at VFS layer using a split device driver model [6]. All guest graphics requests are captured at VFS layer and sent to the host. Then, the host Linux device driver directly handles those graphics requests on behalf of the guest. In order to connect the request between guest and host, we need guest-host memory address translation to memory to enable the host device driver to directly access the guest address space. Note that GPU devices can access user memory directly via their own Memory Management Unit (MMU) hardware; therefore, we need to set up proper guest-host address mappings in the GPU MMU page table. We also need a guest-host file descriptor mapping because a file descriptor for the same object could be different between the guest and host.

The remainder of the paper is organized as follows: in Sect. 2, motivation and contribution of this paper are described; next, detailed analysis of Android's graphic stack is presented in Sect. 3; Sect. 4 describes the proposed system design, and Sect. 5 briefly explains its optimization; Sect. 6 illustrates various experimental results; Sect. 7 discusses related work and finally, Sect. 8 provides concluding remarks

Manuscript received October 25, 2016.
Manuscript revised March 14, 2017.
Manuscript publicized April 18, 2017.

[†]The authors are with POSTECH, Hyoja-dong, Pohang, 790–784, Republic of Korea.
a) E-mail: baksejin@postech.ac.kr
b) E-mail: bspark@postech.ac.kr
c) E-mail: dnsrjd1212@postech.ac.kr
d) E-mail: cipark@postech.ac.kr

DOI: 10.1587/transinf.2016EDP7435

and discusses future works regarding this research.

2. Motivation and Contribution

In mobile environment, GPU acceleration based high performance graphics support is an essential functionality to support from fast user responsiveness to full-featured user interfaces. Modern mobile OSes such as Android or iOS use GPU accelerated User Interface SDK based on OpenGL ES [5]: Android SDK for Android and Quartz Core Framework [7] for iOS. Without GPU acceleration, a user experiences significantly low quality user interfaces such as non-smooth scrolling or low quality image blurring. Even worse, without GPU, an app's performance is significantly low since the software-based renderer such as [8] shows extremely low performance. However, existing mobile virtualization techniques [1], [2] do not consider GPU virtualization. This is because, these mobile virtualization technologies are ported from existing server virtualization techniques such as Xen [6] or KVM [9]. Since most server workloads are processed in terminal environment, they do not require high speed graphics processing. Thus existing server virtualization techniques do not require high performance graphics service including GPU device.

In order to see the performance degradation under existing mobile virtualization environment, we run various applications on Android JB-MR1 guest on KVM/ARM [2]. The experiment was conducted on an Exynos 5250 Arndale board [10]. It includes an ARM Cortex-A15 dual core CPUs (1.7 GHz), 2 GB of system memory, and an ARM Mali-T604 GPU [11]. The guest OS has one VCPU and 512MB of memory. We launched a simple PDF reader application [12] and scrolled it as a normal user ordinarily does but it shows poor scrolling responsiveness. This result is originated from the lack of OpenGL ES support in the environment. Actually, the smooth scrolling is achieved by GPU accelerated 2D rendering pipeline. We can see further interesting phenomenon when we launch Angry Birds [13]. Its launching time was more than 3 minutes and average frame per second (fps) was less than 1 fps. This performance reduction is also originated from no GPU acceleration. Although the board has GPU, the guest OS cannot recognize it. When we launch them again on a native environment that can use GPU on the same experimental board, the PDF reader application shows smooth scrolling and the launching time of Angry Birds is about 6 seconds and it shows native frame per second (about 60 fps). This performance gap is significantly high. In order to virtualize GPU, one of the possible approaches is to configure a physical GPU device to be exclusively dedicated to a single virtualized domain for high-performance graphics operations; however, this does not allow GPU devices to be shared among multiple domains.

As a matter of fact, graphics virtualization in x86 environment is quite familar and relatively well-explored area. Many previous works such as VMGL [14], Mediated Passthrough [15], SVGA Architecture [16] and etc. have well analyzed x86 graphics architecture and shown various methods to virtualize graphics architecture on x86 platform using software-based virtualization. In addition, graphic card manufacturers introduced their SR-IOV-enabled graphics card [17] that supports hardware-based graphics virtualization. Recently, x86 processor manufacturers also introduced hardware-based GPU virtualization technology such as Intel GVT-s, GVT-d and GVT-g [18]. That is, x86 environment supports well-organized way to virtualize device drivers.

In contrast, only several recent researches [19], [20] proposed graphics virtualization on mobile environment. One approach is to virtualize every graphics operation like API remoting [14]. This approach incurs high overhead, resulting in low graphics performance. As a trade-off between graphics performance and GPU sharing, an OS-level virtualization approach [20] was introduced. Since all graphics operations are processed in a single OS, it shows near-native performance; however, it depends on a single OS, resulting in potentially low reliability.

This is because graphics architecture virtualization on mobile environment is relatively new area and has not been well studied yet. The complexity of GPU devices worsens the situation. Furthermore, Android has various components and devices for graphics architecture that involve GPU, Fence, Ion integrated memory allocator, Vsync, SurfaceFlinger, display device and so on. Some of the components such as Ion allocator, SurfaceFlinger and etc. are unique features of Android that are not existed in x86. This not-yet-studied unique features make harder to virtualize it compared to x86. Note that graphics architecture for ARM based mobile environment such as Android has been complicated and optimized to achieve high graphics performance even for low-end mobile devices.

In this paper, we propose Android graphics virtualization technique based on KVM/ARM environment, which is fundamental for virtualization in mobile platform. The contributions of this paper are as follows:

1. First KVM/ARM based GPU virtualization technique.
2. Analysis on Android graphics architecture in terms of mobile virtualization.
3. Design and implementation of para-virtualization based high performance virtualization technique for Android graphics architecture.

3. Android Graphics Internal

In this section, we describe the Android graphics architecture based on version Jelly Bean MR1. Figure 1 depicts the Android graphics architecture stack, in which the process called SurfaceFlinger plays a central role. SurfaceFlinger issues allocation requests of memory buffers to the ION device driver in the underlying Linux kernel via Gralloc HAL (Hardware Adaptation Layer). After a memory buffer is allocated, SurfaceFlinger issues OpenGL commands on the memory buffer to build a graphics image, and then com-

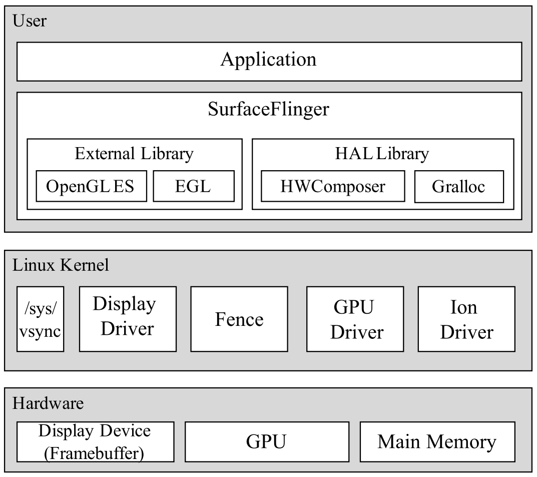


Fig. 1 The architecture of the Android graphics. Android consists of a user-level native layer and Linux kernel. The native layer includes HAL and the SurfaceFlinger service.

poses multiple buffers into a single buffer with the help of HWComposer HAL. Then, SurfaceFlinger sends the composed buffer to the display device. Finally, the contents in the composed buffer appear on the physical display device, such as an LCD.

3.1 ION

ION is an integrated memory allocator for graphics buffer memory used by I/O devices such as the GPU, display controller, and camera. Before Android 4.0, each vendor provided its own specific memory allocators and interfaces, so compatibility was an open issue across different vendors' Android devices. In order to solve this compatibility issue, ION was introduced with Android 4.0. Since then, SurfaceFlinger has been able to allocate buffers using ION.

3.2 Fence

Fence is a synchronization framework that facilitates the management of graphics buffers accessed by multiple processes. Memory buffers are concurrently accessed to fill up with display contents by multiple processes called producers and consumers.

3.3 GPU

The embedded graphics library (EGL) and OpenGL ES [5] have been adopted to support 2D and 3D graphics acceleration in Android mobile devices. EGL is an interface worked with OpenGL ES that is required to manage graphics contexts, window system, surface and buffer binding, and rendering synchronization. Also, the OpenGL ES API is used to render and compose surfaces by applications and SurfaceFlinger. Vendor-provided OpenGL libraries send several file operations to the GPU device driver.

3.4 VSync and Display

A VSync is a synchronization signal generated by a display device as a notification that the contents of a buffer are completely displayed and the buffer is now free. So VSync is very important for smooth animation. VSync first appeared in Android Jelly Bean. Typically, a display device sends VSync interrupts to the display device driver periodically at the display refresh rate interval.

4. Design

Figure 2 shows the proposed graphics virtualization architecture on ARM/KVM for Android mobile platforms. Note that the proposed graphics architecture virtualization techniques can be applied to platforms other than Android because the virtualization techniques are mostly applied to Linux kernel.

4.1 Architecture

The proposed graphics virtualization model is based on para-virtualization, which splits the device driver into two parts: frontend driver in the guest OS and backend driver in the host OS. The frontend and the backend drivers communicate with each other to forward device file requests of a guest user thread to the associated host QEMU thread. The split device driver model is a well-known technique to virtualize an I/O device in a virtualized environment. Most of the existing split device drivers are divided at the common interface layer, such as a block device bio or network device socket buffer [6], [21]. These interfaces are generic and vendor independent, so they are relatively easy to split.

However, GPU devices are not simple to split in the middle of the device driver because there is no standard or generic interface. Furthermore, a GPU device driver is provided as a proprietary driver of a GPU vendor, and most of the source code is closed. For our research, we chose the VFS layer as the common interface between the process and device drivers. VFS requests for a device in a guest process are sent to a host and handled directly by the host device driver. In order for the host device driver to handle guest VFS requests directly, we need two special modules: Address Translator and FD Manager. Address Translator translates the guest virtual address (GVA) to a host virtual address (HVA). A GVA in the request from a guest is translated to an HVA so that a host driver can access the guest address space directly. The other component is FD Manager. The request of a guest process is delivered and issued in the host backend driver, but the file descriptor in the host and guest may be different. FD Manager maps file descriptors between the host and guest. A detailed explanation is provided in the Device File Virtualization section. Four device files are virtualized the Android graphics architecture: GPU device, ION device, display device, and Fence. Each device driver is implemented as a loadable kernel module.

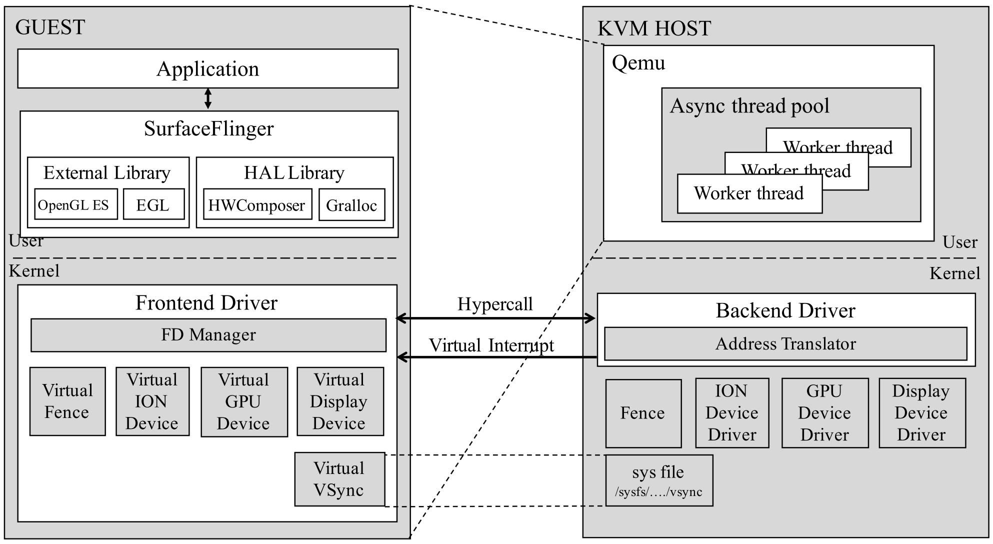


Fig. 2 Architecture of the proposed virtualization model.

In our prototype, */dev/mali0*, */dev/ion*, and */dev/graphics/fb0* are implemented and represent the GPU device, ION device, and display device, respectively. For Fence, it is not a device driver, but it is generated in a GPU and display device driver and has its own file operation table. Thus, a process can access Fence using its file descriptor.

4.2 Guest-Host Communication

Frontend/Backend Driver: The frontend driver is served as a device file in the guest. It delivers device file requests from a guest process to the backend driver using a hypercall. Then, the backend driver delivers the request to the physical device driver using the VFS interface to generate all of the kernel structures for the request in the host. The backend driver is running in the context of the host QEMU thread of the hypercall issued VM since the VM is running in the user space of the QEMU thread. Thus, the delivered request is issued to the physical device driver by the host QEMU thread. After the backend driver finishes the physical device driver service, it returns to the frontend driver with the result from the physical device driver.

Synchronous/Asynchronous Communication support between Backend and Frontend Driver: In order to communicate between backend (host) and frontend (guest VM) driver, synchronous and asynchronous methods are supported according to the file operation. Most of the file operations, such as open(), close(), ioctl(), mmap(), and munmap(), are synchronously handled. These operations use a hypercall to deliver a request from the guest to the host. Note that while the request is being processed in the host, the entire guest VM that issued the hypercall is blocked until the hypercall returns in the synchronous method.

However, read() operation is asynchronously processed between frontend and backend. In general, when read() operation issues, the operation is blocked and waits for the completion (e.g. interrupt) from corresponding device.

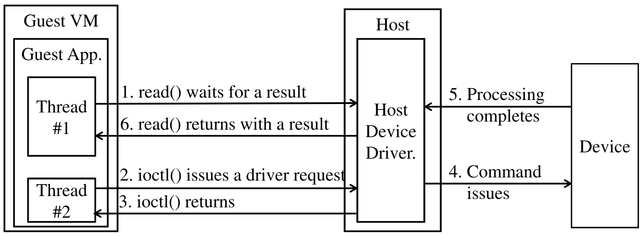


Fig. 3 read() processing sequence. The purpose of read() is waiting and reading a result from an ioctl(). The guest read() waits (1) for the ioctl() processing completion (5). If the communication is in synchronous mode, thread #1's read() will wait for a result forever, and it never returns to the guest OS because thread #2 cannot issue the ioctl() command with the guest OS being blocked owing to the unreturned read() from thread #1.

Thus when the frontend driver delivers read() to the backend driver, the read() waits in the backend driver. Therefore, if read() operation is synchronously processed, the whole guest VM is blocked forever. Figure 3 illustrates an example sequence of read() processing in current communication model. In Fig. 3, the guest application has two threads. Thread #1 calls read() to wait for a result (1). Then thread #2 calls ioctl() to deliver a device (e.g., GPU) command (2) and returns (3). The command is delivered to the device (4) and completed (5). Then the waited read() call reads the result and returns to the guest (6). In this sequence, if the (1) read() call is synchronous, then (2) thread #2's ioctl() call cannot be issued because the whole guest VM is blocked by the unreturned read() hypercall from thread #1. Specifically, in Mali-GPU T-604 [11], read() call waits for the completion of an ioctl command IOCTL_KBASE_FUNC_JOB_SUBMIT to read the result of the command.

To address this issue, asynchronous communication methods are supported. The communication method from a guest to a host is the same with the synchronous type (i.e., hypercall interface), but the returning method is different. When the request arrives at the host, it wakes up the worker

Table 1 Virtualized device file operations

Virtual Device	Virtualized File Operations
Virtual GPU Device	open(), close(), ioctl(), read(), mmap(), *munmap()
Virtual Display Device (Framebuffer)	open(), close(), ioctl()
Virtual ION Device	open(), close(), ioctl()
Virtual Fence	close(), ioctl()

** Actually, munmap() is not categorized as a file operation but works with mmap().*

threads for this request and immediately returns to the guest. In the guest, the process that issued the request is sleeping, but the guest itself is not blocked because the hypercall is returned. This allows the guest to run the other processes while the I/O request is running on the host. In the host, a worker thread runs the I/O request. After the worker thread is completed, it sends a virtual interrupt to the guest. The interrupt handler in the guest catches this interrupt and wakes up the sleeping process that issued the request. We manage these threads using thread pools to avoid thread creation/release overhead because they are frequently created and released.

4.3 Device File Virtualization

Android graphics architecture requires GPU device, display device, ION device, and Fence. All of these, except Fence, are provided as device files in Linux kernel. Fence is not provided as a device file, but it is backed with a file and file operations. Because file operations are to be virtualized in our proposed approach, Fence can also be considered in the same way as device files. Our approach is to para-virtualize file operations of device files. All requests are captured at VFS layer of device files in the guest. These requests are sent to the host and are directly handled by the host's Linux kernel. That is, graphics requests issued by the guest VM are directly handled by the host VM. In order to enable this handling, several address translations and file descriptor management are applied to graphics requests by the virtualized device driver in the guest VM. Table 1 summarizes the device file operations required to virtualize the Android graphics architecture.

Address Translation: Guest graphics I/O requests include memory addresses specified in GVAs. Our proposed technique allows the host to directly handle guest graphics requests, i.e., it allows the host device and its device driver to directly access guest memory. In order to enable this direct handling of guest requests by the host, it is essential to translate memory addresses represented by the GVA into the corresponding memory addresses represented by the HVA. Note that guest I/O requests sent to the host are processed by the QEMU thread. (In KVM, one QEMU thread is associated with one guest VM.) In summary, a guest application issues graphics I/O requests with memory addresses in a GVA. The frontend driver in the guest forwards these I/O requests to a QEMU thread in the host domain. The QEMU thread handles these I/O requests by invoking the backend

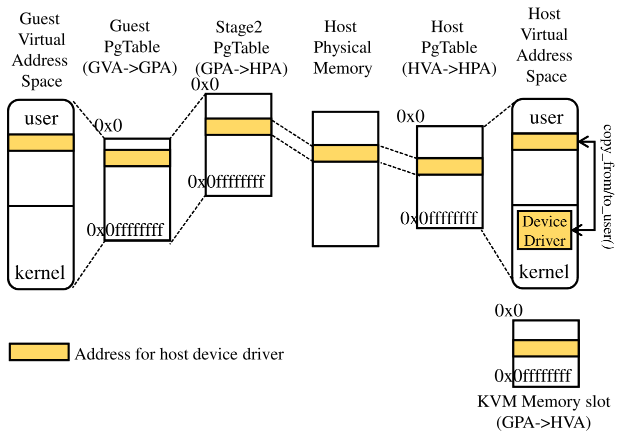

Guest Virtual Address Space
Guest PgTable (GVA->GPA)
Stage2 PgTable (GPA->HPA)
Host Physical Memory
Host PgTable (HVA->HPA)
Host Virtual Address Space
user
kernel
0x0
0x0ffffffff
Device Driver
copy_from/to_user()
Address for host device driver
KVM Memory slot (GPA->HVA)

Fig. 4 Memory addresses for address translation. The GVA is mapped to the HVA. The host device driver can access the GVA using its corresponding HVA.

device driver. In order for the QEMU thread to handle these I/O requests with memory addresses in a GVA, we need to translate memory addresses in the GVA into addresses in the HVA.

Figure 4 shows the address relationships between the guest user space and a host device driver. In the guest, a GVA is mapped to a GPA by the guest page table, and the GPA is mapped to an HPA by the Stage-2 page table. In KVM, an HVA is also mapped to an HPA by the host page table. That is, an HPA can be accessed by an HVA or a GVA.

The module called Address Translator in the backend device driver is responsible for the GVA→HVA address translation. The details of the translation operation are described as follows. Address Translator parses the memory addresses (GVAs) specified in the I/O requests and translates them to HVAs. Since the host device driver uses copy_from/to_user() to access user space addresses (GVAs), tracking GVAs is simple. When a GVA is tracked, it is translated to a GPA. To translate a GVA to a GPA, a software page table walking method can be used. In the ARM architecture, there is an address translation operation named ATS1CPR that translates a VA to a PA [22]. We used this operation to translate GVAs to GPAs. In KVM, there is a special mapping structure called a KVM memory slot. In this structure, GPA→HVA mapping information is stored. We used this structure to translate GPAs to HVAs. This translation allows the host device driver to directly access the GVA in the request using the HVA.

In addition, we have to consider mmap(), which user processes can issue I/O requests directly to a device. Linux kernel sets up a mapping entry in the page table of a user process according to mmap(). Moreover, if mmap() is issued on a device's memory and the device has its own MMU, Linux kernel also sets up a corresponding mapping entry in the device MMU. For example, the ARM Mali GPU device [11] has its own internal MMU. The ioctl() command of KBASE_FUNC_JOB_SUBMIT has the arguments to specify the job chain list that includes several guest memory addresses to represent external graphics resources such as texture images. So, before issuing this ioctl() command, a user process has to set up Mali GPU MMU mappings in advance by mmap() operations.

Now, we describe how guest and host Linux kernels handle mmap() and then how to set up mapping entries in the GPU MMU page table. When a guest user process calls mmap() on a guest virtual GPU device, the guest Linux kernel can set up a mapping of GVA→GPA only after the destination GPA and its associated HVA are dynamically allocated by the host Linux kernel. The guest mmap() call is sent to the host QEMU thread, which finally issues mmap() to a physical GPU device. Then, the host Linux kernel sets up a mapping of HVA→HPA. Note that the host virtual memory region in the HVA is arbitrarily allocated by the host Linux kernel. In order to complete the mmap() operation, the guest Linux kernel needs to know the dynamically allocated HVA and its associated GPA in order to set up a GVA→GPA mapping in the guest.

The problem here is that a guest OS does not recognize whether a GPA is mapped or not since the allocated GPA by mmap() is established by the backend driver in the host OS. Thus, the GPA can be re-allocated or overwritten by the guest OS because the GPA is still un-allocated state for the Guest OS' point of view. In order to solve this problem, two design approaches can be available. One approach is modifying guest OS to recognize the mapping state of the mmapped GPA and another approach is reserving a GPA area for mmap() support. The former requires guest OS modification for mmap() and munmap(). However, if the guest OS has unmanaged-physical address space, the latter approach can be implemented without guest OS modification. For example, if the guest OS has 4GB of main memory, above 4GB of GPA is guest OS unmanaged. So we can use the area as reserved for mmap() support. In our prototype, we use 32-bit guest and 32-bit host OSes since Cortex A15 does not support 64-bit architecture but it supports LPAE that extends physically addressable space up to 40 bits.

Thus, in our design, the guest physical address space above the 4 GB physical address space is reserved for mmap() on virtual devices and is statically mapped to the host virtual address space. That is, a HVA has a one-to-one mapping to a GPA. Since this reserved space is above the 4 GB linear address space, a 32-bit guest Linux kernel can be used without any modification (see Fig. 5). In order to maintain a one-to-one mapping between a GPA and an HVA, the memory slots in the KVM hypervisor are pre-installed for this reserved GPA space (above the 4 GB linear address space) to be mapped into HVA space. Therefore, the guest Linux kernel can set up a GVA→GPA mapping even if the address space of the HVA is dynamically created by mmap(). This space is called the “Guest-OS-unmanaged address area” in Fig. 5 and is possible because of ARM's LPAE extending the address space to 40 bits.

Next, we describe how to set up mapping entries in the GPU MMU page table. When mmap() is issued for a GPU device, the host Linux kernel sets up an HVA→HPA mapping, and the corresponding mapping entry is set up in the GPU MMU page table. We need to change the mapping entry of HVA→HPA to the mapping entry of GVA→HPA in the GPU MMU so that the physical GPU device can directly access the guest memory specified in I/O requests (in the GVA). In order to update the mapping entry, the host Linux kernel needs to know the corresponding GVA allocated by the guest Linux kernel. (Recall that when a guest user process calls mmap(), the VFS layer allocates a free GVA that will be mapped to the target GPA.) This is simply done by sending the GVA from the guest frontend driver to the host backend driver. Then, the host Linux kernel can set a GVA→HPA mapping entry in the GPU MMU page table. A new GPU context is created when a process calls open() on the GPU device. GPU MMU entries are managed independently according to GPU contexts. Therefore, each process can freely set up mappings in the GPU MMU without any interference from other processes.

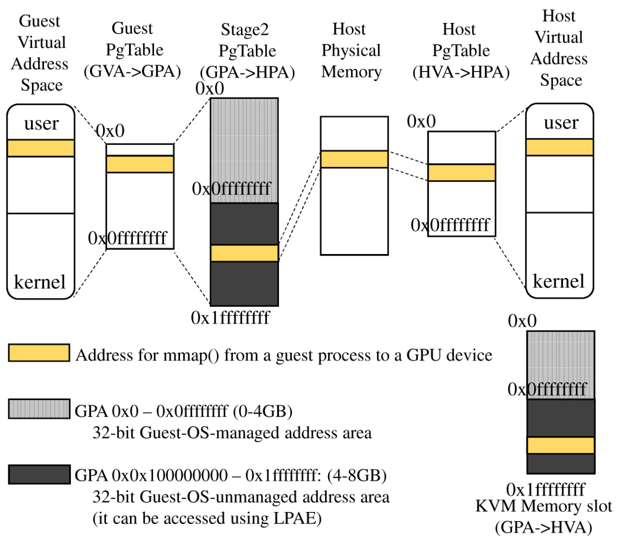

Guest Virtual Address Space
Guest PgTable (GVA->GPA)
Stage2 PgTable (GPA->HPA)
Host Physical Memory
Host PgTable (HVA->HPA)
Host Virtual Address Space
user
kernel
0x0
0x0ffffffff
0x1ffffffff
0x0
0x0ffffffff
0x0
0x0ffffffff
user
kernel
0x0
0x0ffffffff
0x1ffffffff
KVM Memory slot (GPA->HVA)
Address for mmap() from a guest process to a GPU device
GPA 0x0 – 0x0ffffffff (0-4GB) 32-bit Guest-OS-managed address area
GPA 0x0x100000000 – 0x1ffffffff: (4-8GB) 32-bit Guest-OS-unmanaged address area (it can be accessed using LPAE)

Fig. 5 Memory addresses for mmap() support. The GPA area for mmap() is located in the Guest-OS-unmanaged area because the guest OS does not recognize the new GPA→HPA mapping that is created by the backend driver.

File Descriptor Management: FD Manager is responsible for file descriptor mapping between a guest and a host. File descriptors are created in two ways: most are created by the open() system call from a user process, and some are created implicitly by a device driver itself without an explicit open() system call from a user process. For example, in Android graphics architecture, file descriptors of Fence objects and buffer memory objects are created implicitly by the Fence and ION device driver.

When a process opens a device, a file descriptor is returned to the process and will be used as a handle to access the device. In our graphics virtualization architecture, if a guest process calls open(), a new file descriptor is created by the guest OS, and the open() request is sent to the associated QEMU thread on the host. Then, the QEMU thread also calls open() to create a new file descriptor in the host OS.

Thus, we have two distinct file descriptors, one in the guest OS and the other in the host OS, for a single open() system call issued by a guest user process. FD Manager keeps this pair of file descriptors as a file descriptor mapping. Later, when a guest file operation is to be sent to the host, FD Manager translates the guest file descriptor into the host file descriptor based on the mapping. In some cases, FD Manager should analyze the arguments of a device file operation because there are device file operations with file descriptors encapsulated in arguments. For example, the IOCTL command S3CFB_WIN_CONFIG specifies a file descriptor of buffer memory in the command argument. In this case, FD Manager should parse the arguments of IOCTL commands for the file descriptor mapping.

Now, we discuss the second case, in which file descriptors are created by a device driver without an explicit open() system call from a user process. In Android graphics architecture, file descriptors of Fence objects and buffer memory objects are created implicitly by the Fence and ION device drivers. In our graphics virtualization architecture, this means that file descriptors can be created by host device drivers without the guest OS being aware of the creation of these file descriptors. In order to resolve this issue, FD Manager on the guest OS has an additional function to create a new file descriptor whenever a new file descriptor is created implicitly by host device drivers.

Device File Operations: Basically, when a file operation is issued from a guest process, FD Manager finds the corresponding file descriptor for the host and sends it to the host.

1. open(): open() creates a new file descriptor for the guest and host. FD Manager stores the file descriptors of the guest and host for later access. The parameter of open() contains a path to the device file opened. This path is allocated in the GVA and accessed by copy_from_user(). This path should be delivered to the host so that the host can open the correct device file. Address Translator translates the GVA that represents a path to the HVA.
2. close(): When a guest issues a close() call on a device file, both the guest and the host release the file descriptor for the device file. FD Manager removes the file descriptor mapping information between the guest and host.
3. ioctl(): ioctl() contains many commands and arguments for the GPU. Address Translator translates the GVA to an HVA for some of the arguments that are accessed by copy_from/to_user().
4. read(): Address Translator translates the GVA to an HVA for arguments such as the user buffer address accessed by copy_from/to_user().
5. mmap(): As explained in the Address Translation section, an mmap() operation creates a mapping between the guest user address (GVA) and host physical address (HPA). If mmap() is issued with virtual GPU device memory, a mapping entry of GVA→HPA will be set up additionally in the GPU MMU page table.
6. munmap(): When a guest process calls munmap(), the associated mapping is destroyed both in the guest Linux kernel and host Linux kernel. In case of GPU device memory, the corresponding mapping entry in the GPU MMU should also be deleted. In order to handle the mapping removal by munmap(), we register a callback function in *vm_operations.close()* when mmap() is executed in the guest kernel (e.g., frontend driver). Later, the callback function is invoked when munmap() is issued by a guest process.

4.4 VSync Support

Since VSync is the signal to draw all surfaces to the display device, virtualizing VSync is required on Android platform. The VSync event is received by a poll() system call on device file */sys/.../vsync*. The guest event thread waits for the VSync signal until the event has arrived. The host display device generates the VSync signal at 60 FPS in most cases to the host device driver. Thus, we need to deliver this event to the guest OS where the physical display device does not exist. In the proposed model, a VSync event in the host is relayed to the frontend driver. When the host kernel interrupt handler receives the VSync interrupt, it sends this interrupt to the guest kernel via virtual interrupt. After the guest event thread receives the VSync event, it checks device file */sys/.../vsync* to read the sync value. This device file on the guest is mapped to the same device file (*/sys/.../vsync*) on the host side. This mapping is established using Stage-2 page table remapping.

4.5 Device Sharing

In order for a single device to support multiple guest OSes, device sharing should be enabled. Our virtualization model is based on the host device driver. Fortunately, GPU, Fence, and ION drivers allow multiple accesses from multiple processes. This feature is inherited by the multiple threads in multiple VMs because when a process in a VM opens the device driver, a new file descriptor is returned to the process in the VM and a new context is created for the process. The display device is special because it is the final output device. If multiple VMs send their frame buffers simultaneously, the display output can be abnormally mixed. In the proposed model, we support a foreground / background VM model for the display device. In this model, only foreground VM can access the display device. Therefore, this model gives performance benefit for foreground VM that a user directly uses. Please note that, in Android, overlaid images such as status bar on top of the screen are also applications. Thus this kind of overlaid images are located inside of a screen in a VM by the VM's SurfaceFlinger. In evaluation section, we show the foreground VM's GPU running performance in multi-VM case.

4.6 An Illustrative Example

In this section, we describe how this works based on the access path diagram in Fig. 6. The GVA is mapped to the HPA via the GPA that is skipped in Fig. 6, and the HVA is also mapped to the HPA.

In order to virtualize the graphics architecture, we need to support four access paths. When a guest user thread issues a device driver request to the frontend driver, the request is delivered to the backend driver using the hypercall interface, and the backend driver delivers the request to the device driver. Thus, the guest request is sent to the host device driver. The solid line in Fig. 6 shows this path. Then the device driver may access a user virtual address via copy_from/to_user(), which can access the host user virtual address. In the proposed mechanism, the GVAs in the request are translated to corresponding HVAs by the Address Translator in the backend driver. As a result, the host device driver can successfully access the guest user address space using an HVA because the host user QEMU has the full address space for its guest VM. The dotted line in Fig. 6 shows this path. With this path in place, the request can be successfully processed in the device driver.

Some devices, such as a GPU, require access to user virtual addresses to read/write data directly. As explained in mmap() support of the section Address Translation, the problem is that the command sets issued to the device are not open to a developer due to the vendor's policy. That is, the GVA in the command cannot be translated to the HVA. We solved this problem by altering the GPU page table entry for GVA→HPA. Thus, the device can directly access the correct memory address using the GVA. The dash-dotted line

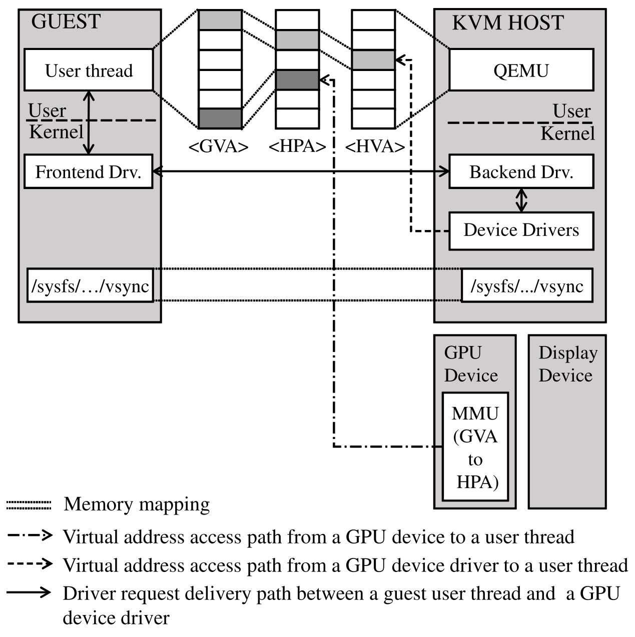

GUEST
User thread
User
Kernel
Frontend Drv.
/sysfs/…/vsync
<GVA>
<HPA>
<HVA>
KVM HOST
QEMU
User
Kernel
Backend Drv.
Device Drivers
/sysfs/…/vsync
GPU Device
MMU (GVA to HPA)
Display Device
Memory mapping
Virtual address access path from a GPU device to a user thread
Virtual address access path from a GPU device driver to a user thread
Driver request delivery path between a guest user thread and a GPU device driver

Fig. 6 An illustrative accessing path diagram for the proposed system. Each line shows the accessing path between various components.

in Fig. 6 shows this path.

The VSync event is generated from the display device. When the display device generates this event, the user application can read the VSync event value via the sysfs interface. In the proposed system, the */sysfs/. . . /vsync* file on the guest OS is mapped to the host OS's file. When the display device generates a VSync event, a virtual interrupt is generated from the backend device driver to the frontend device driver, and the guest user thread can read data from the mapped */sysfs/. . . /vsync*. Thus, all graphics-architecture-relevant components can communicate with each other without any problems.

5. Optimization

5.1 Fence Operations: File Structure Reusing and Close() File Operation Coalescing

In Android, synchronization entities, Fences, are frequently created and released to synchronize graphics buffers between consumers and producers. Figure 7 shows the number of ioctl() calls issued when the 3D benchmark is running on an Arndale board [10]. Note that a new Fence is created by the backend driver in the host for those ioctl() calls marked with an asterisk (*) (i.e., SYNC_IOC_MERGE, S3CFB_WIN_CONFIG, and KBASE_FUNC_JOB_SUBMIT). A new file structure must be created in the frontend driver of the guest kernel after the backend driver in the host creates a new Fence. The newly created file structure in the frontend driver is associated with the Fence just created in the backend driver in the host.

The overhead of creating a file structure in the frontend driver is not negligible. Thus, an optimization technique called file reuse is applied. The idea is to create several file structures in advance for this purpose. When a new file is needed in the frontend driver, one of these pre-created file structures will be used. In the prototype, we pre-allocated

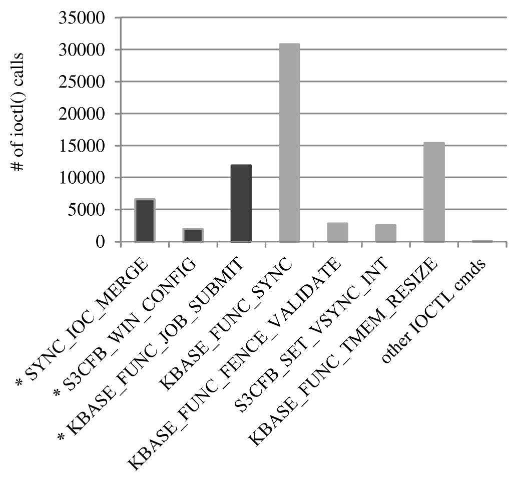


Fig. 7 Number of ioctl() commands called during 3D benchmark Nenamark2 execution. A new Fence is created when a command marked with an asterisk (*) is issued.

60 file structures based on our empirical estimation. We also apply close() file operation coalescing. In order to release each Fence, a close() file operation is called. This frequently called close() operation does not need to be handled synchronously. So, when a guest application calls close(), the frontend driver does not forward the close() operation to the host. Instead, this close() operation is stacked in a temporal buffer. When the buffer is full, the frontend driver forwards the buffered close() operations to the backend driver by issuing HVC (hypervisor call) commands.

5.2 Virtual Interrupt Optimization

In KVM virtualization environment, one QEMU process is associated with one guest. A QEMU process is responsible for generating virtual interrupts to the guest OS since (virtual) I/O devices for the guest OS are emulated by the QEMU process. When a virtual I/O device generates an interrupt, the QEMU process issues a KVM ioctl() to generate a corresponding virtual interrupt. This ioctl() operation causes user-kernel context-mode switching overhead on the host side. As an optimization, we bypass the QEMU process to generate virtual interrupts. That is, the backend driver directly issues KVM ioctl() commands to generate virtual interrupts without relying on the QEMU process.

6. Evaluation

6.1 Prototype and Workload Description

We implemented the proposed virtualization technique on an Exynos 5250 Arndale board [10]. It included an ARM Cortex-A15 CPU with dual cores (1.7 GHz), 2 GB of system memory, and an ARM Mali-T604 GPU [11]. Total LOC of the proposed method is about 2K. Specifically, we implemented frontend and backend driver (1.5K LOC). We do not modified core KVM/ARM architecture and Android platform. We added a new hypercall on KVM/ARM to communicate between guest and host (0.5K LOC for hypercall handler). We simply modified GPU device driver on the host side. The GPU device driver establishes HVA-HPA map using mmap(). We modified the map to GVA-HPA map so that the guest OS can directly access GPU. To enable this, only 3 LOC are modified in the GPU device driver.

The guest VM was running Android JB-MR1 with Linux kernel 3.9 and was configured to run with one VCPU and 512 MB of memory. The host environment of KVM/ARM was Android JB-MR1 with Linux kernel 3.4.5. Note that KVM/ARM is available in Linux kernel 3.9 or later. To deal with this version mismatch, we decided to port KVM/ARM from Linux kernel version 3.9 to Linux kernel 3.4.5. The reason we chose Linux kernel 3.4.5 for the KVM/ARM host was the limited availability of GPU drivers in the Arndale board. It is known that the Arndale board officially supports Linux 3.4.5 with device drivers and device tree binary that is incompatible with Linux kernel version 3.9 or later. Table 2 shows a detailed description of 3D benchmarks that we used to evaluate the proposed system. All of the benchmarks were collected from the Google Play store in Android [28].

Table 2 3D benchmark descriptions

Benchmark	Description
Nenamark2 [23]	OpenGL ES2.0 benchmark using realistic scene
RD3D Benchmark [24]	OpenGL ES2.0 benchmark with CPU-based physics calculation
Bonsai Benchmark [25]	3D effect based benchmark such as static lighting, motion blur, bloom, and Sepia effect
XBenchmark [26]	A benchmark to test CPU and GPU using various technologies
KFS Benchmark (Galactic core) [27]	A simple benchmark to test Android device's OpenGL performance

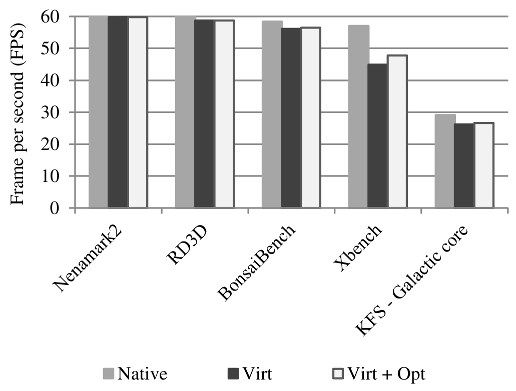


Fig. 8 Performance results of 3D benchmarks. "Native" means the performance of a native Linux without any virtualization. "Virt" and "Virt + Opt" represents the performance results of our proposed technique without optimization and with optimization, respectively.

We have uploaded a demo video of the proposed method on Youtube [29]. The video includes performance comparison (S/W Renderer v.s. Virtual GPU, Native GPU v.s. Virtual GPU) and Virtual GPU Sharing using the proposed method.

6.2 Performance Analysis

Figure 8 shows the performance results of the 3D benchmarks. "Virt + Opt" is the result of our proposed technique with all of the optimization techniques described in Sect. 5 (i.e., Fence and virtual interrupt optimization). Figure 9 shows the normalized performance results with "Native" as a base. VCPU utilizations were also measured to evaluate how intensively each benchmark consumed CPU cycles. For a fair comparison between native Linux and our model, the guest VM was configured to have a single VCPU even though the board provides two physical CPUs.

It is shown in Fig. 9 that our proposed virtualization model provides over 96% of native performance when VCPU utilization is not saturated (i.e., 99.6%, 97.9%, and 96.1% in Nenamark2, RD3D, and Bonsai benchmarks, respectively). The optimization techniques improved the performance of Bonsai and KFS benchmarks from 96.1% to

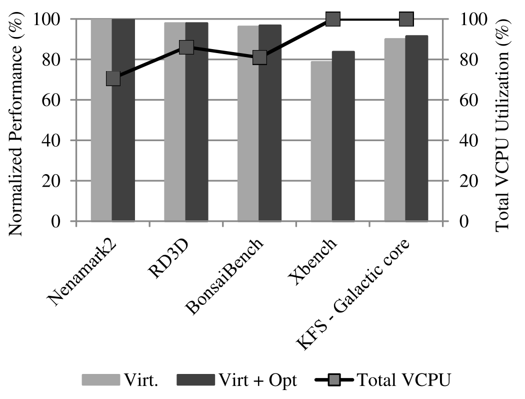


Fig. 9 Normalized performance to native (bar) and VCPU utilization (line). 100% of normalized performance is the same with native performance.

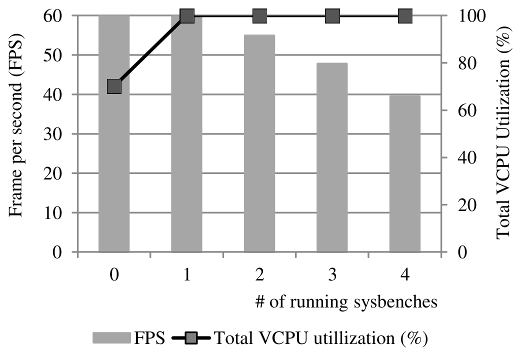


Fig. 10 Performance interference by SysBench workload, which was configured to generate CPU-intensive workloads. FPS results of Nenamark2 benchmark.

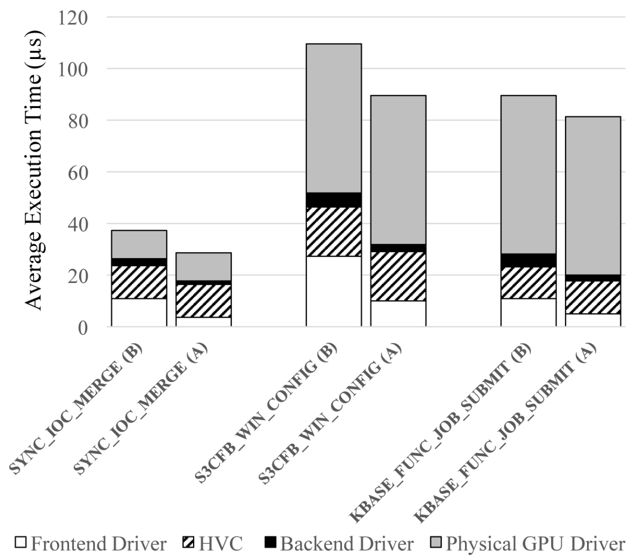


Fig. 11 Average execution time profiling result for IOCTL commands related with Fence operation under workloads in Table 2. (B) indicates "Before Optimization" and (A) indicates "After Optimization".

96.7% and from 90.1% to 91.4%, respectively. In cases of XBenchmark and KFS benchmarks, the performance was near 80% and 90% of native performance, respectively. This performance degradation was due to their high CPU-intensive operations. As depicted in Fig. 9, the total VCPU utilization of these benchmarks was 100%. A detailed explanation is presented in the next section. As shown in Fig. 9, CPU utilization may affect the performance of graphics applications. We further evaluated the performance interference by CPU-intensive workloads. In the experiment, the SysBench benchmark tool [30] was used to generate CPU-intensive workloads.

In Fig. 10, we show how the performance of Nenamark2 varied with increasing CPU-intensive workloads, that is, increasing the number of concurrent SysBench processes. When the number of SysBench processes is zero or one, it still supports 60 FPS, which was originated from the VSync frequency of the Android (60 Hz). We believe that the native performance of the Nenamark2 benchmark was much higher than 60 FPS, but the average FPS would be cut by 60 FPS. When we further increased the number of SysBench processes, the performance of the Nenamark2 benchmark degraded, which means that the performance of our proposed virtualization technique is dependent on total VCPU utilization.

As depicted in Fig. 8 and 9, the optimization results in 2-3 FPS increased performance at best case. However, even this small FPS gain gets important when a user needs real-time interactivity and accuracy in interactive simulation such as 3D game or virtual reality contents. In order to see the effectiveness of the proposed optimization techniques, we profiled IOCTL commands related with Fence operation because the two optimization techniques handle Fence operation and virtual interrupt. They improve frontend driver and backend driver execution time, respectively. Figure 11 shows detailed average profiling result on major IOCTL commands for the optimization technique. As depicted, they optimize about 60% of the frontend driver execution time and about 50% of the backend driver execution time.

6.3 Overhead Analysis

Hypervisor interventions are required in KVM/ARM virtualization operations such as coprocessor accesses, instruction/data aborts, secure monitor calls (SMCs), and hypervisor calls (HVCs). Our proposed virtualization technique for Android graphics generates additional HVCs. In order to analyze the overhead of our proposed method, we categorized hypervisor interventions into two groups: HVCs generated for file operations by our proposed method (denoted as HVC_FEDrv) and the remaining hypervisor interventions required for KVM/ARM operations (denoted as Others).

Figure 12 shows the frequency of HVCs when running 3D benchmarks. It is observed that the hypervisor interventions required by our model, HVC_FEDrv, took more than 50% of hypervisor traps in all 3D benchmarks. In particular, XBenchmark generated a much higher number of hypervisor traps than other 3D benchmarks, and the KFS benchmark generated a relatively small number of hypervi-

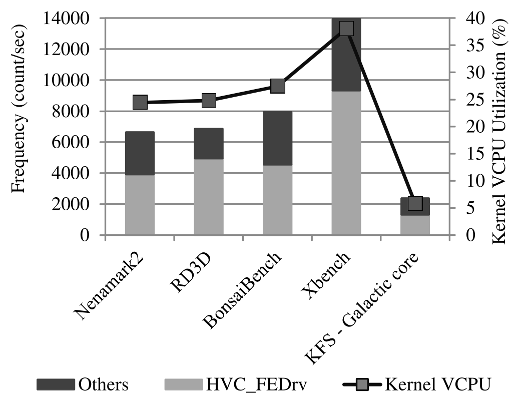


Fig. 12 HVC frequency and kernel VCPU utilization for various 3D benchmarks. "HVC_FEDrv" means HVC called by the graphics architecture virtualization model and "Others" means HVC called by default KVM/ARM.

sor traps. This means that the virtualization performance is dependent on the total VCPU utilization rather than the number of hypervisor traps. The XBenchmark and KFS benchmarks caused 100% VCPU utilization (Fig. 9), which is why they show relatively higher performance degradation than other 3D benchmarks.

Figure 12 also shows kernel VCPU utilization. The KFS benchmark shows almost 5% of kernel VCPU utilization (i.e., 95% of user VCPU utilization). In other words, the performance deduction for the KFS benchmark is mostly due to the native virtualization overhead of KVM/ARM. In contrast, XBenchmark shows almost 40% of kernel VCPU utilization (i.e., 60% of user VCPU utilization). It also has a high HVC_FEDrv frequency. Thus, the performance could be degraded from our virtualization model.

If a workload consumes much time in the kernel, a larger performance improvement by the optimization techniques can be expected. The higher frequency of hypervisor traps in XBenchmark shows that the performance improvement by the optimization techniques in XBenchmark is much larger than in other 3D benchmarks. The optimization improves about 6.4% for XBenchmark (0 - 1.5% improvement for other 3D benchmarks). In the cases of Nenamark2, RD3D, and Bonsai Benchmark, 6.6K, 7K, and 8K hypervisor trap events were generated, respectively. The number of trap events for these benchmarks was not so small, but virtualization overhead was very low (their virtualization overhead is less than 2.1% in Fig. 9). There are two reasons for low virtualization overhead in these three benchmarks. First, VCPU utilization is not 100%. Nenamark2, RD3D, and Bonsai Benchmark cause 70%, 86%, and 81% VCPU utilization, respectively, as depicted in Fig. 9. This means that VCPU still had sufficient computational power to serve graphics architecture virtualization. Second, their benchmark results for native performance were almost 60 FPS, i.e., full frame rate in Android. In Android, a display update rate is based on a VSync event, and its frequency is about 60 Hz, so even though SurfaceFlinger can display more than 60 FPS, the frame rate will be cut by an average of 60 FPS; thus, virtualization overhead can be hidden if the native performance is 60 FPS.

Table 3 Multiple VM benchmark results

Case	VM	Workload	FPS
Multi-VM Case #1	VM1	Nenamark2	59.7
	VM2	RD3D	58.6
Multi-VM Case #2	VM1	Nenamark2	59.7
	VM2	BonsaiBench	56.1
Multi-VM Case #3	VM1	Nenamark2	59.7
	VM2	XBench	44.9
Multi-VM Case #4	VM1	Nenamark2	59.7
	VM2	KFS-Galactic core	26.0
Single-VM Case		Nenamark2	59.9
		RD3D	58.7
		BonsaiBench	56.1
		XBench	44.9
		KFS-Galactic core	26.2

6.4 Multiple-VM Case

In order to show the effectiveness of the proposed method, we launch two android virtual machines (VMs) that use GPU. Each VM has 512MB of system memory and one VCPU. We measured the foreground-background VM switching latency since a foreground VM can access display device. To measure it, we run Nenamark2 workload in VM1 with a foreground state and RD3D benchmark in VM2 with background state at the same time. When we conduct foreground/background switch from VM1 to VM2, the switching latency is about 8.5 microseconds.

We also run various GPU workloads on two VMs. Table 3 shows the result of various GPU workloads. In this experiment, VM1 runs Nenamark2 benchmark and VM2 runs various workloads. The result shows that the performance is almost the same compared to the result of single VM case. This result is originated from our sharing model (Foreground/Background model). In this model, only a foreground VM directly uses display device so that a user can experience high performance GPU execution for currently displaying VM.

7. Related Work

There are several methods to support GPU virtualization. They can be classified into device emulation [31], API remoting [14], [19], device pass-through [32], para-virtualization [6], [21], and OS-level virtualization [20].

The device emulation method supports virtual GPU devices using software, so it is too slow, and it cannot support full GPU functionality such as hardware-accelerated graphics. This kind of method is mainly used in the server virtualization environment.

The API remoting method forwards the graphics library API to the corresponding remote-side GPU device driver, so this method can support many different device drivers from several vendors in the host environment. Despite this advantage, this method has problems such as low performance due to the bulk data transfer and a flexibility issue due to requiring graphics library stack compatibility between the guest OS and the host OS. Dowty et. al. introduced hosted GPU acceleration [16] that is a mixed version of the device emulation and API remoting. They emulated SVGA Device and API remoting (only support for Direct3D on Windows XP). Therefore, it still contains the limitations of the API remoting that are low performance and dependency of specific version of user-level graphics librararies such as OpenGL. In contrast, the proposed method is not affected by user-level graphics libraries because it virtualizes graphics device driver that is in the kernel and this design achieves high performance as well. In addition, this driver-level design gives much more compatibility than user-level design because device drivers are not easily updated compared with user-level libraries.

The device pass-through allows a guest VM to use physical GPU devices directly without hypervisor intervention. However, this approach cannot share the GPU devices because the pass-through enabled guest VM monopolizes the GPU device. To solve this sharing problem, NVIDIA has recently announced a hardware-supported GPU pass-through virtualization solution called NVIDIA GRID [17], which is similar to the network SR-IOV techniques [33]. Although this technique guarantees high performance, it requires expensive hardware devices.

Mediated pass-through technique is recently introduced [15]. They can run native GPU device driver in the guest OS. However, this approach has additional resource management overheads since GPU device drivers can simultaneously run on each guest. This makes difficulty to port it to other platform since the resource management mechanism is driver-dependent. Please note that the native GPU device driver already supports multiple accesses and our proposed method leverages this feature and it gives high portability. Moreover, applying to Android platform is much more difficult because of many sophisticated components of the Android Graphics stack such as Fence or ION.

Para-virtualization can improve I/O virtualization performance by providing a split device driver model between a guest OS and a host OS. In addition, this approach makes it easy to share devices among multiple VMs. However, existing split driver models mainly focus on x86 architecture instead of the ARM architecture. Also, I/O para-virtualization is hard to be applied in the Android OS because of Android-specific components such as ION, Fence, and VSync. In this paper, we suggest the FD manager and VSync virtualization to solve these problems.

Cells [20] provides a high-performance GPU virtualization using OS-level virtualization methods on ARM-based Android platforms. However, fundamentally, OS-level virtualization has significantly more problems than system-level virtualization. It cannot support isolation among guest VMs. That is, if a guest has a system fault, it can be delegated to the host. This is an inevitable problem of OS-level virtualization. However, the proposed technique does not have this problem because it is a system-level virtualization technique.

8. Conclusion and Future Work

In this paper, we proposed an efficient technique to virtualize the Android graphics architecture in a KVM/ARM environment. The graphics architecture of the Android platform relies on device files of the underlying Linux kernel: GPU devices, the ION integrated memory allocator, display devices, and the Fence synchronization framework. Our technique is to para-virtualize these device files using a split device driver model in the VFS layer. That is, the proposed technique enables the host to directly handle device file requests issued by the guest. The main idea include address translation and file descriptor mapping. Further optimizations, such as Fence operations and virtual interrupt generation, were applied for high-performance graphics.

The experimental results show that the proposed virtualization technique achieves almost 84%-100% performance compared with the native application. In addition, the foreground/background sharing model gives high-performance graphics execution for the user interactive foreground VM when multiple VMs are running. Our future work includes enhancing the security of the proposed technique. Because our model is based on para-virtualization using a split device driver model, security concerns may arise due to shared host device drivers. We are currently attempting to use the concept of Linux containers to isolate host device drivers from each guest domain with the concept of Linux containers [34].

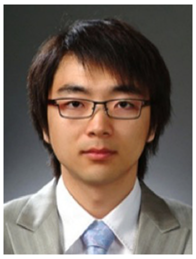
Acknowledgments

This study was supported by Brain Korea 21 PLUS project for POSTECH Computer Science & Engineering Institute.

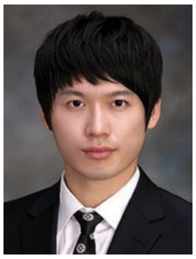
References

[1] J.-Y. Hwang, S.-B. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim, "Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones," Consumer Communications and Networking Conference, CCNC 2008. 5th IEEE, pp.257–261, IEEE, 2008.
[2] C. Dall and J. Nieh, "Kvm/arm: the design and implementation of the linux arm hypervisor," ACM SIGPLAN Notices, pp.333–348, ACM, 2014.
[3] D.A. Willis, "Bring your own device: the facts and the future," Gartner Inc, 2013.
[4] Briefcase, "Your mobile wallet isn't safe," 2015.
[5] KHORONOS, Opengl es, available from: http://www.khoronos.org/opengles.
[6] 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," ACM SIGOPS Operating Systems Review, vol.37, no.5, pp.164–177, 2003.
[7] Apple, iOS 9, quartz core framework, available from: http://developer.apple.com, 2015.
[8] Tungsten, The MESA 3D graphics library, available from: http://www.mesa3d.org.
[9] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM:

the Linux virtual machine monitor," Proceedings of the Linux symposium, pp.225–230, 2007.
[10] Insignal, Arndale board exynos 5250 cortex-a15 dual core, available from: http://www.arndaleboard.org/wiki/index.php/wiki.
[11] ARM, Mali-t604, available from: http://www.arm.com/products/multimedia/mali-performance-efficient-graphics/mali-t604.php.
[12] Adobe, Adobe Acrobat Reader, available from: https://play.google.com/store/apps/details?id=com.adobe.reader, 2015.
[13] R. Ltd., Angry birds, available from: https://www.angrybirds.com, 2015.
[14] H.A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. De Lara, "Vmm-independent graphics acceleration," Proceedings of the 3rd international conference on Virtual execution environments, pp.33–43, ACM, 2007.
[15] K. Tian, Y. Dong, and D. Cowperthwaite, "A full gpu virtualization solution with mediated pass-through," USENIX Annual Technical Conference, pp.121–132, 2014.
[16] M. Dowty and J. Sugerman, "Gpu virtualization on vmware's hosted i/o architecture," ACM SIGOPS Operating Systems Review, vol.43, no.3, pp.73–82, 2009.
[17] NVIDIA, Nvidia grid enterprise graphics virtualization, available from: http://www.nvidia.com/object/enterprise-virtualization.html.
[18] J. Song, Z. Lv, and K. Tian, "Kvmgt: a full GPU virtualization solution," 2014.
[19] L. Bhatia, "Dual-android on nexus 10," available from: http://www.slideshare.net/xen_com_mgr/sruk-xen-presentation2013v7dualandroid, 2013.
[20] J. Andrus, C. Dall, A.V. Hof, O. Laadan, and J. Nieh, "Cells: a virtual mobile smartphone architecture," Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pp.173–187, ACM, 2011.
[21] R. Russell, "virtio: towards a de-facto standard for virtual i/o devices," ACM SIGOPS Operating Systems Review, vol.42, no.5, pp.95–103, 2008.
[22] A. ARM, "Architecture reference manual (armv7-a and armv7-r edition)," ARM DDI C, vol.406, 2008.
[23] NenaInnovationAB, Nenamark2, available from: https://play.google.com/store/apps/details?id=se.nena.nenamark2&hl=en, 2012.
[24] RDMobile, RD 3D benchmark, available from: https://play.google.com/store/apps/details?id=ru.rdmobile.bench&hl=en.
[25] RejectedGames, Bonsai benchmark, available from: https://play.google.com/store/apps/details?id= com.rejectedgames.bonsaibenchmark.
[26] ManiacSoftware, Xbenchmark - next mark 2.0, available from: https://play.google.com/store/apps/details?id=com.maniac.xbench.
[27] KittehfaceSoftware, Kfs opengl benchmark, available from: https://play.google.com/store/apps/details?id= fishnoodle.benchmark.
[28] Google, Play store, available from: https://play.google.com/store.
[29] U. Lee, S. Park, and B. Park, "GPU virtualization for android on KVM/ARM," https://youtu.be/HXL0gMm_L1E, 2014.
[30] A. Kopytov, "Sysbench: a system performance benchmark," URL: http://sysbench.sourceforge.net, 2004.
[31] F. Bellard, "Qemu, a fast and portable dynamic translator," USENIX Annual Technical Conference, FREENIX Track, pp.41–46, 2005.
[32] Y. Dong, J. Dai, Z. Huang, H. Guan, K. Tian, and Y. Jiang, "Towards high-quality i/o virtualization," Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference, p.12, ACM, 2009.
[33] P. Primer, "An introduction to sr-iov technology," Intel LAN Access Division, Revision, vol.2, 2008.
[34] M. Helsley, "Lxc: Linux container tools," IBM devloperWorks Technical Library, 2009.



Sejin Park received a B.S. degree in software engineering from Kumoh National University of Technology, Korea, in 2007. He is currently a Ph.D. candidate in the Department of Computer Science and Engineering, POSTECH, Korea. His research interests include virtualization technology, storage systems, and embedded systems.



Byungsu Park received a B.S. degree in computer science engineering from Hongik University, Korea, in 2010. He is currently a Master's candidate in the Department of Computer Science and Engineering, POSTECH, Korea. His research interests include virtualization technology, storage systems, and embedded systems.

Unsung Lee received a B.S. degree in computer science engineering from POSTECH, Korea, in 2012. He is currently a Ph.D. candidate in the Department of Computer Science and Engineering, POSTECH, Korea. His research interests include virtualization and embedded systems.

Chanik Park received a B.S. degree in 1983 from Seoul National University, Seoul, Korea, and an M.S. degree and Ph.D. in 1985 and 1988, respectively, from Korea Advanced Institute of Science and Technology. Since 1989, he has been working for POSTECH, where he is currently a professor in the Department of Computer Science and Engineering. He was a visiting scholar with the Parallel Systems group in the IBM Thomas J. Watson Research Center in 1991, and was a visiting professor with the Storage Systems group in the IBM Almaden Research Center in 1999. He has served at a number of international conferences as a program committee member. His research interests include storage systems, operating systems, and virtualization technology.