A Full GPU Virtualization Solution with Mediated Pass-Through

Kun Tian, Yaozu Dong, and David Cowperthwaite, *Intel Corporation*

https://www.usenix.org/conference/atc14/technical-sessions/presentation/tian

This paper is included in the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.

June 19–20, 2014 • Philadelphia, PA

978-1-931971-10-2

A Full GPU Virtualization Solution with Mediated Pass-Through

Kun Tian, Yaozu Dong, David Cowperthwaite
Intel Corporation

Abstract

Graphics Processing Unit (GPU) virtualization is an enabling technology in emerging virtualization scenarios. Unfortunately, existing GPU virtualization approaches are still suboptimal in performance and full feature support.

This paper introduces gVirt, a product level GPU virtualization implementation with: 1) *full GPU virtualization* running native graphics driver in guest, and 2) *mediated pass-through* that achieves both good performance and scalability, and also secure isolation among guests. gVirt presents a virtual full-fledged GPU to each VM. VMs can directly access performance-critical resources, without intervention from the hypervisor in most cases, while privileged operations from guest are trap-and-emulated at minimal cost. Experiments demonstrate that gVirt can achieve up to 95% native performance for GPU intensive workloads, and scale well up to 7 VMs.

1. Introduction

The Graphics Processing Unit (GPU) was originally invented to accelerate graphics computing, such as gaming and video playback. Later on, GPUs were used in high performance computing, as well, such as image processing, weather broadcast, and computer aided design. Currently, GPUs are also commonly used in many general purpose applications, with the evolution of modern windowing systems, middleware, and web technologies.

As a result, rich GPU applications present rising demand for *full GPU virtualization* with good performance, full features, and sharing capability. Modern desktop virtualization, either locally on clients such as XenClient [35] or remotely on servers such as VMware Horizon [34], requires GPU virtualization to support uncompromised native graphical user experience in a VM. In the meantime, cloud service providers start to build GPU-accelerated virtual instances, and sell GPU computing resources as a service [2]. Only *full GPU virtualization* can meet the diverse requirements in those usages.

However, there remains the challenge to implement *full GPU virtualization*, with a good balance among performance, features and sharing capability. Figure 1

shows the spectrum of GPU virtualization solutions (with hardware acceleration increasing from left to right). *Device emulation* [7] has great complexity and extremely low performance, so it does not meet today's needs. *API forwarding* [3][9][22][31] employs a frontend driver, to forward the high level API calls inside a VM, to the host for acceleration. However, API forwarding faces the challenge of supporting full features, due to the complexity of intrusive modification in the guest graphics software stack, and incompatibility between the guest and host graphics software stacks. *Direct pass-through* [5][37] dedicates the GPU to a single VM, providing full features and the best performance, but at the cost of device sharing capability among VMs. *Mediated pass-through* [19], passes through performance-critical resources, while mediating privileged operations on the device, with good performance, full features, and sharing capability.
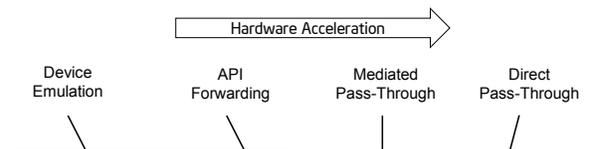

Figure 1: The spectrum of I/O virtualization

This paper introduces gVirt, the first product level GPU virtualization implementation, to our knowledge, with: 1) *full GPU virtualization* running a native graphics driver in guest, and 2) *mediated pass-through* that achieves good performance, scalability, and also secure isolation among guests. A virtual GPU (vGPU), with full GPU features, is presented to each VM. VMs can directly access performance-critical resources, without intervention from the hypervisor in most cases, while privileged operations from guest are trap-and-emulated to provide secure isolation among VMs. The vGPU context is switched per quantum, to share the physical GPU among multiple VMs without user notice. As such, gVirt achieves *full GPU virtualization*, with a great balance among performance, features, and sharing capability. We implement gVirt in Xen, with integrated Intel® Processor Graphics [13] in the 4th generation Intel® Core™ processor. The principles and architecture of gVirt, however, is also applicable to different GPUs and hypervisors. gVirt was initially presented at the Xen Summit [10], and all the gVirt source code is now available to the open source community [8].

This paper overcomes a variety of technical challenges and makes these contributions:

- Introduces a *full GPU virtualization* solution with mediated pass-through that runs the native graphics driver in guest

- Passes through performance-critical resource accesses with graphics memory resource partitioning, address space ballooning, and direct execution of guest command buffer

- Isolates guests by auditing and protecting the command buffer at the time of command submission, with smart shadowing

- Further improves performance with virtualization extension to the hardware specification and the graphics driver (less than 100 LOC changes to the Linux kernel mode graphics driver)

- Provides a product level open source code base for follow-up research on GPU virtualization, and a comprehensive evaluation for both Linux and Windows guests

- Demonstrates that gVirt can achieve up to 95% of native performance for GPU-intensive workloads, and up to 83% for workloads that stress both the CPU and GPU

The rest of the paper is organized as follows. An overview of the GPU is provided in section 2. In section 3, we present the design and implementation of gVirt. gVirt is evaluated with a combination of graphics workloads, in section 4. Related work is discussed in section 5, and future work and conclusion are in section 6.

2. GPU Programming Model

In general, Intel Processor Graphics works as shown in Figure 2. The *render engine* fetches GPU commands from the command buffer, to accelerate rendering graphics in many different features. The *display engine* fetches pixel data from the frame buffer and then sends them to external monitors for display.

This architecture abstraction applies to most modern GPUs but may differ in how graphics memory is implemented. Intel Processor Graphics uses system memory as graphics memory, while other GPUs may use on-die memory. System memory can be mapped into multiple virtual address spaces by GPU page tables. A 2GB global virtual address space, called *global graphics memory*, accessible from both the GPU and CPU, is mapped through *global page table. Local graphics memory* spaces are supported in the form of multiple 2GB local virtual address spaces, but are only

limited to access from the render engine, through *local page tables. Global graphics memory* is mostly the frame buffer, but also serves as the command buffer. Massive data accesses are made to *local graphics memory* when hardware acceleration is in progress. Other GPUs have some similar page table mechanism accompanying the on-die memory.
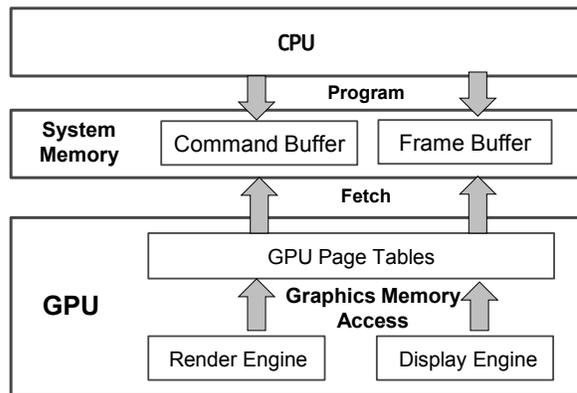

Figure 2: The architecture of the Intel Processor Graphics

The CPU programs the GPU through GPU-specific commands, shown in Figure 2, in a producer-consumer model. The graphics driver programs GPU commands into the *command buffer*, including primary buffer and batch buffer, according to high level programming APIs like OpenGL and DirectX. Then the GPU fetches and executes the commands. The primary buffer, a ring buffer (*ring buffer*), may chain other batch buffers (*batch buffer*) together. We use the terms: primary buffer and ring buffer, interchangeably hereafter. The batch buffer is used to convey the majority of the commands (up to ~98%) per programming model. A register tuple (*head, tail*) is used to control the ring buffer. The CPU submits the commands to the GPU by updating *tail*, while the GPU fetches commands from *head,* and then notifies the CPU by updating *head*, after the commands have finished execution.

Having introduced the GPU architecture abstraction, it is important for us to understand how real-world graphics applications use the GPU hardware so that we can virtualize it in VMs efficiently. To do so, we characterized, for some representative GPU-intensive 3D workloads (Phoronix Test Suite [28]), the usages of the four critical interfaces: the *frame buffer*, the *command buffer*, the GPU *Page Table Entries* (PTEs) which carry the GPU page tables, and the *I/O registers* including Memory-Mapped I/O (MMIO) registers, Port I/O (PIO) registers, and PCI configuration space registers for internal state. Figure 3 shows the average access frequency of running Phoronix 3D workloads on four interfaces.

The *frame buffer* and *command buffer* exhibit the most performance-critical resources, as shown in Figure 3. The detail test configuration is shown in section 4. When the applications are being loaded, lots of source vertexes and pixels are written by the CPU, so the frame buffer accesses dominate, in the 100s of thousands per second. Then at run-time, the CPU programs the GPU, through the commands, to render the frame buffer, so the command buffer accesses become the largest group, also in the 100s of thousands per second. PTE and I/O accesses are minor, in tens of thousands per second, in both load and run-time phases.
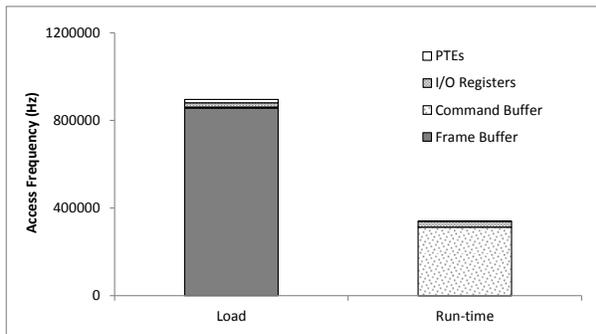

Figure 3: Access patterns of running 3D workloads

3. Design and Implementation

gVirt is a *full GPU virtualization* solution with mediated pass-through. As such, gVirt presents every VM a full-fledged GPU, to run native graphics driver inside a VM. The challenge, however, is significant in three ways: 1) complexity in virtualizing an entire sophisticated modern GPU, 2) performance due to multiple VMs sharing the GPU, and 3) secure isolation among the VMs without any compromise. gVirt reduces the complexity and achieves good performance, through the mediated pass-through technique, in subsection 3.1, 3.2, 3.3, and 3.4, and enforces the secure isolation, with the smart shadowing scheme in subsection 3.5.

3.1. Architecture

Figure 4 shows the overall architecture of gVirt, based on Xen hypervisor, with Dom0 as the privileged VM, and multiple user VMs. A gVirt stub module, in Xen hypervisor, extends the memory virtualization module (vMMU), including EPT for user VMs and PVMMU for Dom0, to implement the policies of trap and pass-through. Each VM runs the native graphics driver, and can directly access the performance-critical resources: the frame buffer and command buffer, with resource partitioning as presented in subsection 3.3 & 3.4. To protect privileged resources, that is, the I/O registers and PTEs, corresponding accesses, from the graphics driver in user VMs and Dom0, are trapped and

forwarded to the mediator driver in Dom0 for emulation. The mediator uses hypercall to access the physical GPU. In addition, the mediator implements a GPU scheduler, which runs concurrently with the CPU scheduler in Xen, to share the physical GPU amongst VMs.
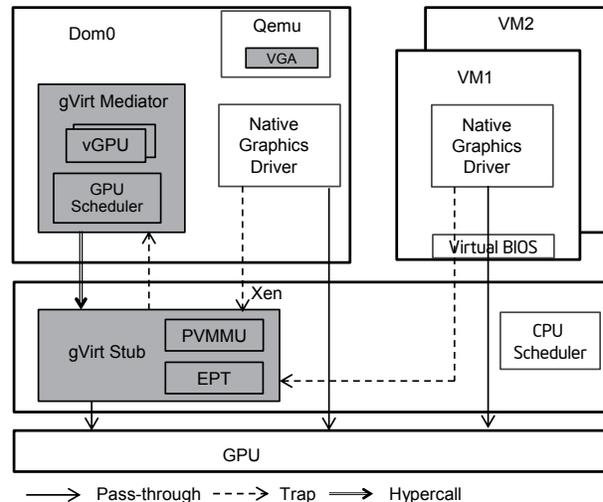

Figure 4: The gVirt Architecture

gVirt uses the physical GPU to directly execute all the commands submitted from a VM, so it avoids the complexity of emulating the render engine, which is the most complex part within the GPU. In the meantime, the resource pass-through, of both the frame buffer and command buffer, minimizes the hypervisor's intervention on the CPU accesses, while the GPU scheduler guarantees every VM a quantum for direct GPU execution. So gVirt achieves good performance when sharing the GPU amongst multiple VMs.

gVirt stub: We extend the Xen vMMU module, to selectively trap or pass-through guest access of certain GPU resources. Traditional Xen supports only pass-through or trap of the entire I/O resource of a device, for either device emulation or device pass-through. gVirt manipulates the EPT entries to selectively present or hide a specific address range to user VMs, while uses a reserved bit of PTEs in PVMMU for Dom0, to selectively trap or pass-through guest accesses to a specific address range. In both cases, the PIO accesses are trapped. All the trapped accesses are forwarded to the mediator for emulation, and the mediator uses hypercalls to access the physical GPU.

Mediator: gVirt mediator driver emulates virtual GPUs (vGPUs) for privileged resource accesses, and conducts context switches amongst the vGPUs. In the meantime, gVirt relies on the Dom0 graphics driver to initialize the physical device and to manage power. gVirt takes a flexible release model, by implementing the mediator as

a kernel module in Dom0, to ease the binding between the mediator and hypervisor.

A split CPU/GPU scheduling mechanism is implemented in gVirt, for two reasons. First, the cost of the GPU context switch is over 1000X the cost of the CPU context switch (~700us vs. ~300ns, per our experiments). Second, the number of the CPU cores likely differs from the number of the GPU cores, in a computer system. So, gVirt implements a separate GPU scheduler from the existing CPU scheduler. The split scheduling mechanism leads to the requirement of concurrent accesses to the resources from both the CPU and the GPU. For example, while the CPU is accessing the graphics memory of VM1, the GPU may be accessing the graphics memory of VM2, concurrently.

Native driver: gVirt runs the native graphics driver inside a VM, which directly accesses a portion of the performance-critical resources, with privileged operations emulated by the mediator. The split scheduling mechanism leads to the resource partitioning design in subsection 3.3. To support resource partitioning better, gVirt reserves a Memory-Mapped I/O (MMIO) register window, called gVirt_info, to convey the resource partitioning information to the VM. Note that the location and definition of gVirt_info has been pushed to the hardware specification as a virtualization extension, so the graphics driver must handle the extension natively, and future GPU generations must follow the specification for backward compatibility. The modification is very limited, with less than 100 LOC changes to Linux kernel mode graphics driver.

Qemu: We reuse Qemu [7] to emulate the legacy VGA mode, with the virtual BIOS to boot user VMs. This design simplifies the mediator logic, because the modern graphics driver doesn't rely on the BIOS boot state. It re-initializes the GPU from scratch. The gVirt extension module decides whether an emulation request should be routed to the mediator or to Qemu.

3.2. GPU Sharing

The mediator manages vGPUs of all VMs, by trap-and-emulating the privileged operations. The mediator handles the physical GPU interrupt, and may generate virtual interrupt to the designated VMs. For example, a physical completion interrupt of command execution may trigger a virtual completion interrupt, delivered to the rendering owner. The idea of emulating a vGPU instance per semantics is simple; however, the implementation involves a large engineering effort and a deep understanding of the GPU. For example, ~700 I/O registers are accessed by the Linux graphics driver.

Render engine scheduling: gVirt scheduler implements a coarse-grain quality of service (QoS) policy. A time quantum of 16ms is selected as the scheduling time slice, because it is less human perceptible to image change. Such a relatively large quantum also comes from that, the cost of the GPU context switch is over 1000X that of the CPU context switch, so it can't be as small as the time slice in CPU scheduler. The commands from a VM are submitted to the GPU continuously, until the guest runs out of its time-slice. gVirt needs to wait for the guest ring buffer to become idle before switching, because most GPUs today are non-preemptive, which may impact the fairness. To minimize the wait overhead, gVirt implements a coarse-grain flow control mechanism, by tracking the command submission to guarantee the piled commands, at any time, are within a certain limit. Therefore, the time drift between the allocated time slice and the executed time is relatively small, compared to the large quantum, so a coarse-grain QoS policy is achieved.

Render context switch: gVirt saves and restores internal pipeline state and I/O register states, plus cache/TLB flush, when switching the render engine among vGPUs. The internal pipeline state is invisible to the CPU, but can be saved and restored through GPU commands. Saving/restoring I/O register states can be achieved through reads/writes to a list of the registers in the render context. Internal cache and Translation Lookaside Table (TLB), included in modern GPUs to accelerate data accesses and address translations, must be flushed using commands at render context switch, to guarantee isolation and correctness. The steps used to switch a context in gVirt are: 1) save current I/O states, 2) flush the current context, 3) use the additional commands to save the current context, 4) use the additional commands to restore the new context, and 5) restore I/O state of the new context.

gVirt uses a *dedicated ring buffer* to carry the additional GPU commands. gVirt may reuse the (audited) guest ring buffer for performance, but it is not safe to directly insert the commands into the guest ring buffer, because the CPU may continue to queue more commands as well, leading to overwritten content. To avoid the race condition, gVirt switches from the guest ring buffer to its own *dedicated ring buffer*. At the end of the context switch, gVirt switches from the *dedicated ring buffer* to the guest ring buffer of the new VM.

Display management: gVirt reuses the Dom0 graphics driver to initialize the display engine, and then manages the display engine to show different VM frame buffers. When two vGPUs have the same resolution, only the

frame buffer locations are switched. For different resolutions, gVirt uses the hardware scalar, a common feature in modern GPUs, to scale the resolution up and down automatically. Both methods take mere milliseconds. In many cases, gVirt may not require the display management, if the VM is not shown on the physical display, for example, when it is hosted on the remote servers [34].

3.3. Pass-Through

gVirt passes through the accesses to the *frame buffer* and *command buffer* to accelerate performance-critical operations from a VM. For the *global graphics memory* space, 2GB in size, we propose graphics memory resource partitioning and address space ballooning mechanism. For the *local graphics memory* spaces, each with a size of 2GB too, we implement per-VM *local graphics memory*, through render context switch, due to *local graphics memory* only accessible by GPU.

Graphics memory resource partitioning: gVirt partitions the *global graphics memory* among VMs. As explained in subsection 3.1, split CPU/GPU scheduling mechanism requires that the *global graphics memory* of different VMs can be accessed simultaneously by the CPU and the GPU, so gVirt must, at any time, present each VM with its own resource, leading to the resource partitioning approaching, for *global graphics memory*, as shown in Figure 5.
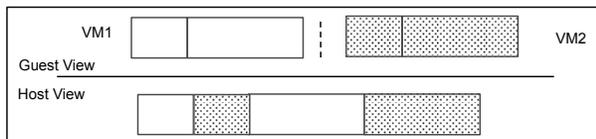

Figure 5: Graphics memory with resource partitioning

The performance impact of the reduced *global graphics memory* resource, due to the partitioning, is very limited according to our experiments. Results are shown in Figure 6, with performance normalized to the score of the default 2GB case. We did experiments in the native environment, and then scaled the 2GB *global graphics memory* down to 1/2, 1/4, and 1/8, with negligible performance impact observed. This is because the driver uses the *local graphics memory* to hold the massive rendering data, while the *global graphics memory* mostly serves only for the frame buffer, and the ring buffer, which are limited in size.

The resource partitioning also reveals an interesting problem: the guest and host now have an inconsistent view of the *global graphics memory*. The guest graphics driver is unaware of the partitioning, assuming with exclusive ownership: the *global graphics memory* is contiguous, starting from address zero. gVirt has to translate between the host view and the guest view, for

any graphics address, before being accessed by the CPU and GPU. It therefore incurs more complexity and additional overhead, such as additional accesses to the command buffer (usually mapped as un-cacheable and thus slow on access).
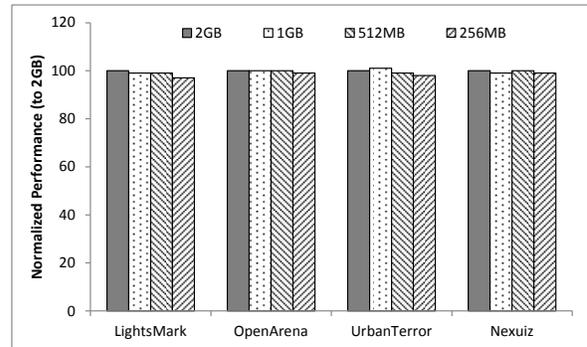

Figure 6: The performance with different size of the *global graphics memory*

Address space ballooning: We introduce the *address space ballooning* technique, to eliminate the address translation overhead, illustrated in Figure 7. gVirt exposes the partitioning information to the VM graphics driver, through the gVirt_info MMIO window. The graphics driver marks other VMs' regions as 'ballooned', and reserves them from its graphics memory allocator. With such design, the guest view of global graphics memory space is exactly the same as the host view, and the driver programmed addresses, using guest physical address, can be directly used by the hardware. Address space ballooning is different from traditional memory ballooning techniques. Memory ballooning is for memory usage control, concerning the number of ballooned pages, while address space ballooning is to balloon special memory address ranges.
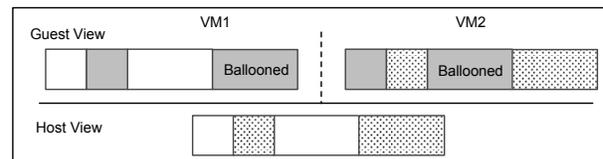

Figure 7: Graphics memory with address space ballooning

Another benefit of address space ballooning is to directly use the guest command buffer, without any address translation overhead, for direct GPU execution. It simplifies the implementation a lot, by eliminating the need of the shadow command buffer, in addition to performance guarantee. However, such scheme may be susceptible to security violation. We address this issue with smart shadowing, by auditing and protecting the command buffer from malicious attack, which is discussed in subsection 3.5.

Per-VM *local graphics memory*: gVirt allows each VM to use the full *local graphics memory* spaces, of its own, similar to the virtual address spaces on CPU. The *local graphics* memory spaces are only visible to the render engine in the GPU. So, any valid *local graphics memory* address programmed by a VM can be used directly by the GPU. The mediator switches the *local graphics memory* spaces, between VMs, when switching the render ownership.

3.4. GPU Page Table Virtualization

gVirt virtualizes the GPU page tables with shared shadow *global page table* and per-VM shadow *local page table*.

Shared shadow *global page table*: To achieve resource partitioning and address space ballooning, gVirt implements shared shadow *global page table* for all VMs. Each VM has its own guest *global page table*, translated from the graphics memory page number to the Guest memory Page Number (GPN). Shadow *global page table* is then translated from the graphics memory page number to the Host memory Page Number (HPN). The shared shadow *global page table* maintains the translations for all VMs, to support concurrent accesses from the CPU and GPU concurrently. Therefore, gVirt implements a single, shared shadow *global page table,* by trapping guest PTE updates, as shown in Figure 8. The *global page table*, in MMIO space, has 512K PTE entries, each pointing to a 4KB system memory page, so in overall creates a 2GB *global graphics memory* space. gVirt audits the guest PTE values, according to the address space ballooning information, before updating the shadow PTE entries.
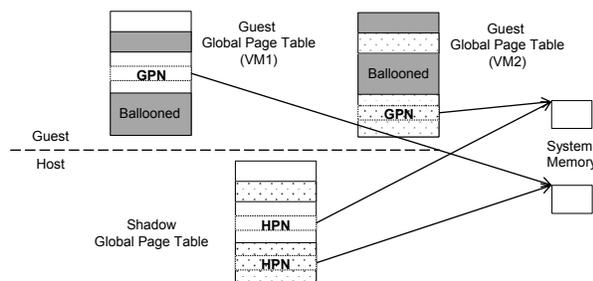

Figure 8: Shared shadow *global page table*

Per-VM Shadow *local page tables*: To support pass-through of *local graphics memory* access, gVirt implements per-VM shadow *local page tables*. The local graphics memory is only accessible from the render engine. The *local page tables are* two-level paging structures, as shown in Figure 9. The first level Page Directory Entries (PDEs), located in the *global page table*, points to the second level Page Table Entries (PTEs), in the system memory. So, guest access

to the PDE is trapped and emulated, through the implementation of shared shadow *global page table*. gVirt also write-protects a list of guest PTE pages, for each VM, as the traditional shadow page table approach does [15][25] . The mediator synchronizes the shadow page with the guest page, at the time of write-protection page fault, and switches the shadow *local page tables* at render context switches.
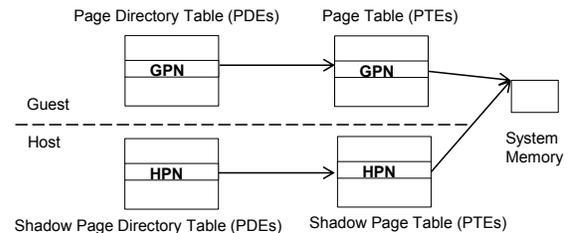

Figure 9: Per-VM shadow *local page table*

3.5. Security

Pass-through is great for performance, but it must meet the following criteria for secure isolation. First, a VM must be prohibited from mapping unauthorized graphics memory pages. Second, all the GPU registers and commands, programmed by a VM, must be validated to only contain authorized graphics memory addresses. Last, gVirt needs to address denial-of-service attacks, for example, a VM may deliberately trigger lots of GPU hangs.

3.5.1. Inter-VM Isolation

Isolation of CPU accesses: CPU accesses to privileged I/O registers and PTEs are trap-and-emulated, under the control of the mediator Therefore a malicious VM can neither directly change the physical GPU context, nor map unauthorized graphics memory. CPU access to frame buffer and command buffer is also protected, by the EPT.

On the other hand, gVirt reuses the guest command buffer, for the GPU to execute directly for performance, as mentioned in subsection 3.3, but, it may violate isolation, for example, a malicious command may contain an unauthorized graphics memory address. gVirt solves the problem with smart shadowing as detailed in subsection 3.5.2.

Isolation of GPU accesses: gVirt audits graphics memory addresses, in registers and commands, before the addresses are used by the GPU. It is implemented at the time of trap-and-emulating the register access, and at the time of command submission.

Denial-of-service attack: gVirt uses the *device reset* feature, widely supported in modern GPUs, to mitigate the deny-of-service attacks. The GPU is so complex,

that an application may cause the GPU to hang for many reasons. So, modern GPUs support *device reset* to dynamically recover the GPU, without the need to reboot the whole system. gVirt uses this capability to recover from a variety of GPU hangs, caused by problematic commands from VMs. In the meantime, upon the detection of a physical GPU hang, gVirt also emulates a GPU hang event, by removing all the VMs from the run queue, allowing each VM to detect and recover accordingly. A threshold is maintained for every VM, and a VM is destroyed if the number of GPU hangs exceeds the threshold.

3.5.2. Command Protection

Balancing performance and security is challenging for full GPU virtualization. To guarantee no unauthorized address reference from the GPU, gVirt audits the guest command buffer at the time of command submission. However there exists a window, between the time when the commands are submitted and when they are actually executed, so a malicious VM may break the isolation by modifying the commands within that window. General shadowing mechanism, such as the shadow page table [15][25], may be applied. However, it is originally designed for the case where the guest content is frequently modified. It may bring large performance overhead and additional complexity in gVirt.

The programming models of the command buffers actually differ from that of the page tables. First, the primary buffer, structured as a ring buffer, is statically allocated with limited page number (32 pages in Linux and 16 pages in Windows), and modification to submitted ring commands (from *head* to *tail*) is not allowed, per the hardware specification. It may be efficient enough to copy only the submitted commands to the shadow buffer. Second, the batch buffer pages are allocated on demand, and chained into the ring buffer. Once the batch buffer page is submitted, it will unlikely be accessed until the page is retired. Shadow buffer can be avoided for such one-time usage.

gVirt implements a smart shadowing mechanism, with different protecting schemes for different buffers, by taking advantage of their specific programming models. That is: *Write-Protection* to the batch buffer, which is unlikely modified (so, the write emulation cost is very limited), and *Lazy-Shadowing* for the ring buffer, which is small in size and can be copied from the guest buffer to the shadow buffer with trivial cost.

Lazy-shadowing **to the ring buffer:** gVirt uses a lazy shadowing scheme to close the attack window on the ring buffer. gVirt creates a separate ring buffer, that is, the shadow ring buffer, to convey the actual commands submitted to the GPU. Guest submitted commands are copied from the guest ring buffer to the shadow ring buffer on demand, after the commands are audited. Note that only the commands submitted to the GPU, are shadowed here. Guest access remains passed through to the guest ring buffer, without the hypervisor intervention. The shadow buffer lazily synchronizes with the guest buffer, when the guest submits new commands. The shadow buffer is invisible to a VM, so there is no chance for a malicious VM to attack.

Write-Protection **to the batch buffer:** The batch buffer pages are write-protected, and the commands are audited before submitting to the GPU for execution, to close the attack window. The write-protection is applied per page on demand, and is removed after the execution of commands in this page is completed by the GPU, which is detected by tracking the advance of ring head. Modification to the submitted commands is a violation of the graphics programming model per specification, so any guest modification to the submitted commands is viewed as an attack leading to the termination of the VM. In the meantime, the command buffer usage may not be page aligned, and the guest may use the free sub-page space for new commands. gVirt tracks the used and unused space of each batch buffer page, and emulates the guest writes to the unused space of the protected page for correctness.

Lazy-shadowing works well for the ring buffer. It incurs an average number of 9K command copies per second, which is a minor cost to a modern multi-GHz CPU. In the meantime, *Write-Protection* works well for the batch buffer, which protects ~1700 pages with only ~560 trap-and-emulations per second, on average.

3.6. Optimization

An additional optimization is introduced to reduce the trap frequency, with minor modifications to the native graphics driver. According to the hardware specification, the graphics driver has to use a special programming pattern at the time of accessing certain MMIO, with up to 7 additional MMIO register accesses [12][13], to prevent the GPU from entering power saving mode. It doesn't incur an obvious cost in the native world, but it may become a big performance challenge, in gVirt, due to the induced mediation overhead. Our GPU power management design gives us a chance to optimize: gVirt relies on Dom0 to manage the physical GPU power, while the guest power management is disabled. Based on this, we optimize the native graphics driver, with a few lines (10 LOC change in Linux) of changes, to skip the additional MMIO register accesses, when it runs in the virtualized

environment. This optimization reduces the trap frequency by 60%, on average.

The graphics driver identifies whether it is in a native environment or a virtualization environment, by the information in gVirt_info MMIO window (refer to subsection 3.1). The definition of gVirt_info has been pushed into the GPU hardware specification, so backward compatibility can be followed by future native graphics driver and future GPU generations.

3.7. Discussion

Architecture independency: Although gVirt is currently implemented on Intel Processor Graphics, the principles and architecture can also be applied to different GPUs. The notion of *frame buffer, command buffer*, *I/O registers,* and *page tables*, are all abstracted very well in modern GPUs. Some GPUs may use on-die graphics memory, however, the graphics memory resource partitioning and address space ballooning mechanism, used in gVirt, are also amendable to those GPUs. In addition, the shadowing mechanism, for both the page table and command buffer, is generalized for different GPUs as well. The GPU scheduler is generic, too, while the specific context switch sequence may be different.

Hypervisor portability: It is easy to port gVirt to other hypervisors. The core component of gVirt is hypervisor agnostic. Although the current implementation is on a type-1 hypervisor, we can easily extend gVirt to the type-2 hypervisor, such as KVM [17], with hooks to host MMIO access (Linux graphics driver). For example, one can register callbacks on the I/O access interfaces, in the host graphics driver, so the mediator can intercept and emulate the host driver accesses to the privileged GPU resources.

VM scalability: Although partitioning graphics memory resource may limit scalability, we argue it can be solved in two orthogonal ways. The first way is to make better use of the existing graphics memory, by implementing a dynamic resource ballooning mechanism, with additional driver cooperation, to share the graphics memory among vGPUs. The other way is to increase available graphics memory resource, by adding more graphics memory in future generation GPUs.

Scheduling dependency: An additional challenge, of *full GPU virtualization*, is the dependency of engines, such as 3D, blitter, and media. The graphics driver may use semaphore commands, to synchronize shared data structures among the engines, while the semaphore commands may not be preempted. It then brings the issue of inter-engine dependency, and leads to the gang scheduling policy in gVirt, to always schedule all engines together; however, it impacts the sharing efficiency. We argue this limitation can be addressed, with a hybrid scheme combining both per-engine scheduling and gang scheduling, through constructing an inter-engine dependency graph, when the command buffers are audited. Then, GPU scheduler can choose per-engine scheduling and gang scheduling policies dynamically, according to the dependency graph.

4. Evaluation

We run 3D and 2D workloads in both Linux and Windows VMs. For Linux 3D workloads, gVirt achieves 89%, 95%, 91%, and 60% of native performance in LightsMark, OpenArena, Nexuiz, and UrbanTerror, respectively. For Linux 2D workloads, gVirt achieves 81%, 35%, 28%, and 83% of native performance, in firefox-asteroids, firefox-scrolling, midori-zoomed, and gnome-system-monitor, respectively. For Windows workloads, gVirt achieves 83%, 80%, and 76% of native performance, running 3DMark06, Heaven3D, and PassMark2D, respectively. In the meantime, gVirt scales well without a visible performance drop, up to 7 VMs.

4.1. Configuration

The hardware platform includes the 4th generation Intel Core processor with 4 CPU cores (2.4Ghz), 8GB system memory, and a 256GB Intel 520 series SSD disk. The Intel Processor Graphics, integrated in the CPU socket, supports a 2GB *global graphics memory* space and multiple 2GB *local graphics memory* spaces.

We run 64bit Ubuntu 12.04 with a 3.8 kernel in both Dom0 and Linux guest, and 64-bit Windows 7 in Windows guest, on top of Xen version 4.3. Both Linux and Windows runs native graphics driver with virtualization extension (refer to subsection 3.1). Each VM is allocated with 4 VCPUs and 2GB system memory. The *global graphics memory* resources are evenly partitioned among VMs, including Dom0. For example, the guest is partitioned with 1GB *global graphics memory* in the 1-VM case, and 512MB in the 3-VM case, respectively.

We use the Phoronix Test Suite [28] 3D benchmark including LighsMark, OpenArena, UrbanTerror, and Nexuiz, and Cairo-perf-trace [4] 2D benchmark including firefox-asteroids (firefox-ast), firefox-scrolling (firefox-scr), midori-zoomed (midori), and gnome-system-monitor (gnome), as the Linux benchmarks. In subsection 4.5, we run Windows 3DMark06 [1], Heaven3D [11] and PassMark2D [26] workloads. All benchmarks are run in full screen resolution (1920x1080). We compare gVirt to the

native, the direct pass-through (based on Intel VT-d), and also to an API forwarding solution (VMGL [9]). We didn't collect the software emulation approach, since it has already been proved infeasible for modern GPU virtualization [9][22].

Three gVirt configurations are examined to show the merits of individual technologies incrementally.

- *gVirt_base:* baseline gVirt without smart shadowing and trap optimization
- *gVirt_sec*: gVirt_base with smart shadowing
- *gVirt_opt*: gVirt_sec with trap optimization

4.2. Performance

Figure 10 shows the performance of both Linux 3D and 2D workloads normalized to native. 3D workloads are GPU intensive except UrbanTerror. gVirt_base achieves 90%, 94%, 89%, and 47% of native performance for LightsMark, OpenArena, Nexuiz, and UrbanTerror, respectively. UrbanTerror is both CPU and GPU intensive, so it suffers from mediation cost more than the others.

For Linux 2D workloads, gVirt_base achieves 63% and 75% of native performance, for firefox-asteroids (firefox-ast) and gnome-system-monitor (gnome), respectively. However, it reaches only 12% and 15% of native performance, for firefox-scrolling (firefox-scr) and midori-zoomed (Midori) workloads, respectively. This is because they are both CPU and GPU intensive, incurring an up to 61K/s trap frequency, resulting in a very high mediation cost, explained in subsection 4.3.
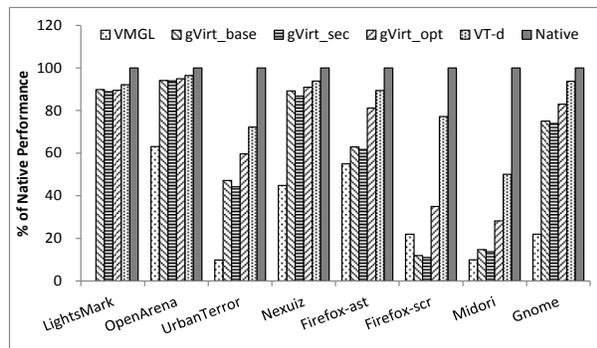

Figure 10: Performance running 3D and 2D workloads

gVirt_sec incurs an average 2.6% and 4.3% performance overhead in 3D and 2D workloads, respectively, much more efficient than a traditional shadowing approach [15][25]. It demonstrates that the smart shadowing scheme can protect the command buffer very effectively, taking advantage of the GPU programming model.

gVirt_opt further improves the performance, up to 214% and 35%, in 2D and 3D workloads, respectively, by

optimizing the native graphics driver to reduce the trap frequency. Firefox-scrolling and midori-zoomed achieves the most obvious increase in 2D workloads, by 214% and 104%, respectively. This is because they trigger very high access frequency of I/O registers (54k/s and 40k/s), so they benefit more from trap optimization. In 3D workloads, gVirt with optimization achieves 89%, 95%, 91%, and 60% of native performance, in LightsMark, OpenArena, Nexuiz, and UrbanTerror, respectively. The performance of gVirt is very close to VT-d with direct GPU pass-through. VMGL performs much worse than gVirt, with only 13% of native performance (vs. 60% in gVirt) in UrbanTerror, average 29% of native performance (vs. 57% in gVirt) in 2D workloads, and it fails to run LightsMark.
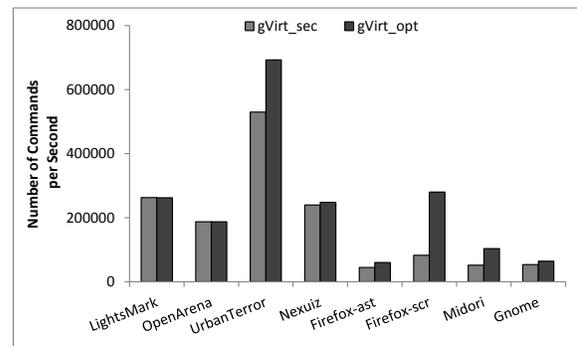

Figure 11: gVirt handles up to 238% more commands, per second, with trap optimization

Furthermore, Figure 11 shows the number of submitted commands per second, with and without trap optimization. UrbanTerror submits 31% more commands per second, with optimization, matching the 35% performance improvement in Figure 10. In firefox-scrolling and midori-zoomed, gVirt handles 238% and 99% more commands per second, with optimization, matching the 214% and 104% performance increase in Figure 10, as well.
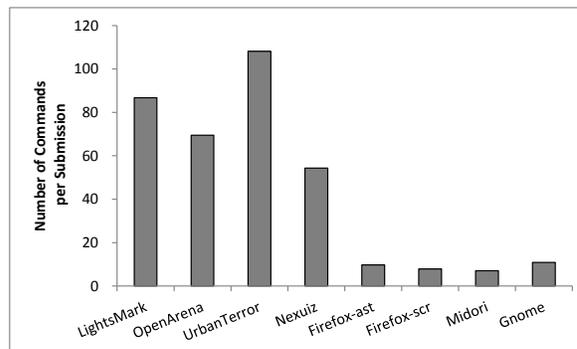

Figure 12: gVirt handles average 8X more commands, per submission, in 3D workloads

We also compare the number of commands per submission, between 3D and 2D workloads, as shown in Figure 12. On average, 3D workloads submit 8X more commands, in every submission, compared to 2D workloads. As a result, 3D workloads induce less mediation overhead per command and achieve better performance.

4.3. Overhead Analysis

We categorize the trap events of gVirt into 4 groups: power management registers (PM) accesses, tail register accesses of ring buffer (Tail), PTE accesses (PTE), and other accesses (Others).

Figure 13 illustrates the break-down of the trap events in gVirt_sec. For 3D workloads, there are around 23K, 22K, 27K, and 33K trap events per second, when running LightsMark, OpenArena, UrbanTerror and Nexuiz, respectively. Among them, 'PM' register access dominates, accounting for up to 67%, 65%, 72%, and 61% of the total trap events, because Linux graphics driver accesses additional PM registers (up to 7) to protect the hardware from entering power saving mode, per hardware specification, when accessing certain registers [12][13]. Tail register access counts for 13%, 12%, 17%, and 13% of the total trap events, respectively. Similarly, 'PM' register access in 2D workloads dominates the trap events as well, accounting for 76% of the total trap rate, on average. 2D workloads has an average 37K/s trap events, 42% higher than that in the 3D workload (26K/s).
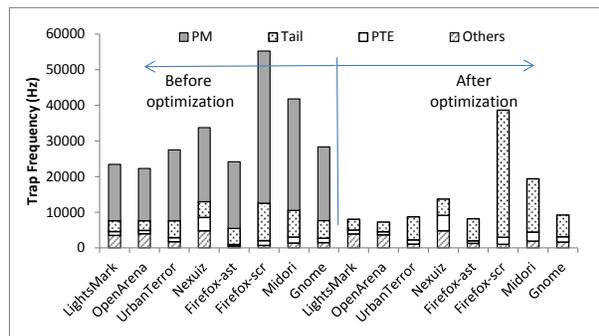

Figure 13: Break-down of the trap frequency, before and after optimization

gVirt_opt reduces the trap events dramatically, as shown in Figure 13. The trap event reduction comes from the removal of all the PM register accesses, which is unnecessary to vGPUs (The real power is managed by Dom0). After the optimization, gVirt reduces the trap rate by an average 65% and 54% for 3D and 2D workloads, respectively. Firefox-scrolling and midori-zoomed have more tail updates, from 19% and 18%, respectively, to 92% and 76% of total traps,

which matches the much improved performance (214% and 104% higher), as seen in Figure 10.

The overhead of the smart shadowing scheme is very limited. gVirt_sec copies average 5K and 12.8K ring buffer commands (typically 1-5 double-words per command), per second, for 3D and 2D workloads, respectively. It write-protects an average of 2000 and 1300 batch buffer pages, along with ~870 and ~150 write emulations due to unaligned batch buffer usages, per second, in 3D and 2D workload, respectively. The CPU cycles spent for smart shadowing are trivial for a modern multi-GHz processor. gVirt_sec incurs very limited virtualization overhead, matching the performance shown in subsection 4.2.

4.4. Scalability

Figure 14 presents the scalability of gVirt (gVirt_opt), with all features and optimizations, from 1 VM to 7 VMs, running the same workloads in all VMs, with performance normalized to 1 VM case. For LightsMark, OpenArena and Nexuiz, the performance remains almost flat, demonstrating that the GPU computing power can be efficiently shared among multiple VMs. In UrbanTerror, we see an 8% performance increase, from 1vm to 7vm, because CPU parallelism helps UrbanTerror, which is both GPU and CPU intensive. For 2D workloads, firefox-asteroids and gnome-system-monitor doubles performance from 1vm to 3vm, because they are more CPU intensive (relatively low access rate to GPU resources), so adding more VMs improves performance. The physical CPU cores saturate eventually, so the performance remains flat, from 3vm to 7vm. In all cases, the performance of gVirt doesn't drop obviously with more VMs, demonstrating very good scalability.
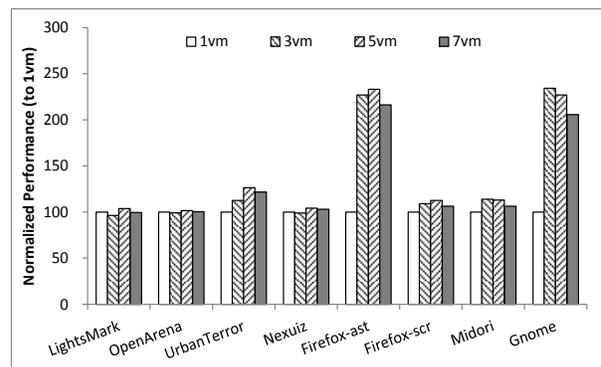

Figure 14: Scalability of gVirt

4.5. Windows

Figure 15 shows the performance of Windows graphics workloads, with smart shadowing and trap optimization (gVirt_opt). We didn't run the baseline gVirt

performance, because the Windows driver we received from the production group has already implemented the virtualization extension, without an option to turn off the trap optimization. For 3DMark06 and Heaven3D, gVirt achieves 83% and 81% of native performance, respectively, which are very close to the VT-d performance (85% and 87% of native performance). In PassMark2D, gVirt achieves 76% of native performance, better than that of the Linux 2D workloads (average 57% of native performance), because Windows 2D workload incurs only an average 6k traps per second, 57% less than that of Linux 2D workloads, and therefore less mediation cost. VMGL doesn't support Windows guest.
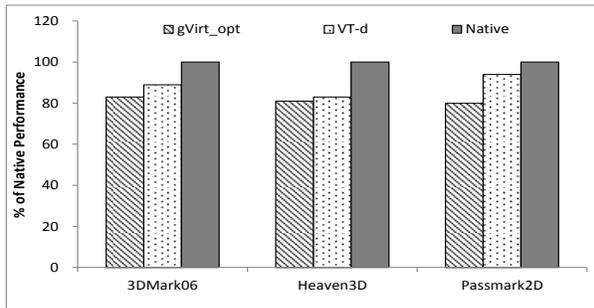

Figure 15: Performance running Windows 3D/2D workloads

Further experiments show that smart shadowing brings only 1.1% and 4.8% performance overhead for Windows 3D and 2D workloads, respectively. It write-protects an average 3600 batch buffer pages, and copies about 10k ring buffer commands, per second, demonstrating that the smart shadowing scheme can protect the command buffer very efficiently, taking advantage of the GPU programming model, in Windows as well.

5. Related Work

Emulating a full-fledged GPU, purely through software, is impractical due to complexity and extremely low performance. Qemu [7] emulates only the legacy VGA cards, with a para-virtualized frame buffer [20] to accelerate 2D specific frame buffer accesses.

API forwarding is the most widely studied technique for GPU virtualization, so far. VMGL [9], Xen3D [3] and Blink [14] install a new OpenGL library in Linux VM, forwarding OpenGL API calls to the host graphics stack for acceleration.. GViM [31], vCUDA [18] and LoGV [23] implement similar API forwarding techniques, focusing on GPGPU computing. VMware's Virtual GPU [22] emulates a virtual SVGA device, implementing a private SVGA3D protocol to forward the DirectX API calls. However, API forwarding faces the challenge of supporting full features, due to the complexity of intrusive modification in the guest

graphics stack, and incompatibility between the guest and host graphics stack.

Device Pass-through achieves high performance in I/O virtualization. VT-d [5][37] translates memory addresses of DMA requests, allowing the GPU to be assigned to a single VM. SR-IOV [27] extends the VT-d technology with a device hardware extension. It has been widely used in the network device [36], by creating multiple virtual functions, which can be individually assigned to VMs. VPIO [19] introduces a "virtual pass-through I/O" concept, where the guest can access the hardware resource directly, mostly of the time, for legacy network cards (NE2000 and RTL8139). They either sacrifice the sharing capability, or are not yet available to modern GPUs.

GPU scheduler is well explored. Kato [29] *et al.* implements a priority-based scheduling policy for multi-tasking environment, based on monitoring GPU commands issued from user space. Kato [30] *et al.* further extends that policy with a context-queuing scheme and virtual GPU support. Gupta [32] *et al.* proposes CPU and GPU coordinated scheduling, with a uniform resource usage model to describe the heterogeneous computing cores. Ravi [33] *et al.* implements a scheduling policy, based on affinity score between kernels, when consolidating kernels among multiple VMs. Becchi [21] *et al.* proposes a virtual memory based runtime, supporting flexible scheduling policies, to dynamically bind applications to a cluster of GPUs. Menychtas [16] *et al.* proposes a disengaged scheduling policy, having the kernel grant application access to the GPU, based on infrequent monitoring of the application's GPU cycle use. They were not applied to *full GPU virtualization*, yet.

NVIDIA GRID [24] allows each VM's GPU commands to be passed directly to the GPU for acceleration. A vGPU manager shares the GPU based on time slices. It looks similar to gVirt in some ways; however there is no public information on technical details, or open access to the project.

6. Conclusion and Future Work

gVirt is a *full GPU virtualization* solution with mediated pass-through, running a native graphics driver in the VM, with a good balance among performance, features, and secure sharing capability. We introduce the overall architecture, with the policies of mediation and pass-through base on the access patterns to the GPU interfaces. To ensure efficient and secure graphics memory virtualization, we propose graphics memory resource partitioning, address space ballooning, shared shadow global page table, per-VM shadow local page

table, smart shadowing mechanism, and additional optimization to remove the unnecessary trap events. gVirt presents a vGPU instance to each VM, with full features, based on trap-and-emulating privileged operations. Such full GPU virtualization solution allows the native graphics driver to be run inside a VM. We also reveal that different programming model of applications might introduce different trap frequency and therefore different virtualization overhead. Lastly, gVirt is an open source implementation, so it provides a solid base for follow-up GPU virtualization research.

As for future work, we will focus on the areas of portability, scalability, and scheduling areas, as discussed in subsection 3.7, in addition to fined-grained QoS scheduling policy. In the meantime, we will evaluate hardware assistance to further reduce the mediation cost. Hypervisor interposition features are also interesting to us, for example, supporting VM suspend/resume and live migration [6]. With gVirt as the vehicle, we will extend *full GPU virtualization* to more usages, in desktop, server, and mobile devices, to exploit specific challenges in different use cases.

References

[1] 3DMark06. http://www.futuremark.com

[2] Amazon GPU instances. http://aws.amazon.com/ec2/instance-types/

[3] C. Smowton. Secure 3D graphics for virtual machines. In EuroSEC'09: Proceedings of the Second European Workshop on System Security. ACM, 2009, pp. 36-43.

[4] Cairo-perf-trace. http://www.cairographics.org

[5] 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, 10, August, 2006.

[6] E. Zhai, G. D. Cummings, and Y. Dong. Live migration with pass-through device for linux vm. In *Proc.* OLS (2008)

[7] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proc.* USENIX ATC (2005)

[8] gVirt. https://github.com/01org/XenGT-Preview-kernel, https://github.com/01org/XenGT-Preview-xen, https://github.com/01org/XenGT-Preview-qemu

[9] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. D. Lara. VMM-independent graphics acceleration. In *Proc.* VEE (2007), pp. 33-43

[10] H. Shan, K. Tian, Y. Dong, and D. Cowperthwaite. XenGT: a Software Based Intel Graphics Virtualization Solution. Xen Project Developer Summit (2013)

[11] Heaven3D. http://unigine.com/products/heaven

[12] Intel Graphics Driver. http://www.x.org/wiki/IntelGraphicsDriver/

[13] Intel Processor Graphics PRM. https://01.org/linuxgraphics/documentation/2013-intel-core-processor-family

[14] J. G. Hansen. Blink: Advanced display multiplexing for virtualized applications. In *Proc.* NOSSDAV (2007)

[15] K. Adams and O.Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proc.* ASPLOS (2006)

[16] K. Menychtas, K. Shen, and M. L. Scott. Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators. In *Proc.* ASPLOS (2014)

[17] KVM. www.linux-kvm.org/

[18] L. Shi, H. Chen, and J. Sun. vCUDA: GPU Accelerated High Performance Computing in Virtual Machines. In Proc. IEEE IPDPS (2009)

[19] L. Xia, J. Lange, P. Dinda, and C. Bae. Investigating virtual passthrough I/O on commodity devices. In *Proc.* ACM SIGOPS (2009), pp. 83-94

[20] M. Armbruster. The Xen Para-virtualized Frame Buffer. Xen Summit (2007).

[21] M. Becchi, K. Sajjapongse, I. Graves, A. Procter, V. Ravi, and S. Chakradhar. A virtual memory based runtime to support multi-tenancy in clusters with GPUs. In *Proc.* HPDC (2012), pp. 97-108

[22] M. Dowty and J. Sugerman. GPU virtualization on VMware's hosted I/O architecture. In *Proc.* ACM SIGOPS (2009), pp. 73-82

[23] M. Gottschlag, M. Hillenbrand, J. Kehne, J. Stoess, and F. Bellosa. LoGV: Low-overhead GPGPU virtualization. In *Proc.* IEEE Workshop on Frontiers of Heterogeneous Computing (2013).

[24] NVIDIA GRID. http://on-demand.gputechconf.com/gtc/2013/presentations/S3501-NVIDIA-GRID-Virtualization.pdf

[25] 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), pp. 164-177

[26] PassMark2D. http://www.passmark.com

[27] PCI SIG. I/O virtualization. http://www.pcisig.com/specifications/iov

[28] Phoronix Test Suites. http://phoronix-test-suite.com

[29] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments. In *Proc.* USENIX ATC (2011)

[30] S. Kato, M. McThrow, C. Maltzahn, and S. BrandtGdev. Gdev: First-Class GPU Resource Management in the Operating System. In *Proc.* USENIX ATC (2012)

[31] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. GViM: GPU-accelerated virtual machines. In *Proc.* ACM HPCVirt (2009), pp. 17-24

[32] V. Gupta, K. Schwan, N. Tolia, V.Talwar, and P. Ranganathan. Pegasus: Coordinated scheduling for virtualized accelerator-based system. In *Proc.* USENIX ATC (2011)

[33] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar. Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework, In *Proc.* HPDC (2011), pp. 217-288

[34] VMware Horizon View. http://www.vmware.com/products/horizon-view/

[35] XenClient. http://www.citrix.com/products/xenclient/overview.html

[36] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H, Guan. High performance network virtualization with SR-IOV. In *Proc.* IEEE HPCA (2010), pp. 1-10

[37] Y. Dong, J. Dai, Z. Huang, H. Guan, K. Tian, Y. Jiang, Towards high-quality I/O virtualization. SYSTOR 2009