VGRIS: Virtualized GPU Resource Isolation and Scheduling in Cloud Gaming

Miao Yu *, Chao Zhang, Zhengwei Qi, Jianguo Yao
School of Software, Shanghai Jiao Tong University, Shanghai, China
{superymk, kevin_zhang, qizhwei, jianguo.yao}@sjtu.edu.cn

Yin Wang
HP Labs, Palo Alto, USA
yin.wang@hp.com

Haibing Guan
Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, Shanghai, China
hbguan@sjtu.edu.cn

ABSTRACT

Fueled by the maturity of virtualization technology for Graphics Processing Unit (GPU), there is an increasing number of data centers dedicated to GPU-related computation tasks in cloud gaming. However, GPU resource sharing in these applications is usually poor. This stems from the fact that the typical cloud gaming service providers often allocate one GPU exclusively for one game. To achieve the efficiency of computational resource management, there is a demand for cloud computing to employ the multi-task scheduling technologies to improve the utilization of GPU.

In this paper, we propose VGRIS, a resource management framework for **V**irtualized **G**PU **R**esource **I**solation and **S**cheduling in cloud gaming. By leveraging the mature GPU paravirtualization architecture, VGRIS resides in the host through library API interception, while the guest OS and the GPU computing applications remain unmodified. In the proposed framework, we implemented three scheduling algorithms in VGRIS for different objectives, i.e., Service Level Agreement (SLA)-aware scheduling, proportional-share scheduling, and hybrid scheduling that mixes the former two. By designing such a scheduling framework, it is possible to handle different kinds of GPU computation tasks for different purposes in cloud gaming. Our experimental results show that each scheduling algorithm can achieve its goals under various workloads.

Categories and Subject Descriptors

C.4 [**PERFORMANCE OF SYSTEMS**]: Modeling techniques, measuring techniques

*The first author's current affiliation is Cylab, Carnegie Mellon University. The contact email is superymk@cmu.edu

HPDC'13, June 17–21, 2013, New York, NY, USA.

Keywords

GPU, Resource management, Scheduling, Cloud gaming

1. INTRODUCTION

The cloud computing significantly reduces cost of capital and equipment maintenance by allowing users to host their softwares on the cloud under a simple pay-as-you-go. As a cloud service, cloud gaming is a game service that executes the game programs and renders the graphics on the server side, while players stream the video through broadband connection using thin clients. This gaming model has several advantages. It allows easy access to games without owning a console or high-end graphics cards or Graphics Processing Unit (GPU)s. Game distribution and maintenance become much easier.

Concurrently, virtualization technology is making a significant impact on how resources are used and managed in a cloud data center. Several virtualization solutions (VMware products, Xen [1], VirtualBox) are getting more and more mature in constructing a huge cloud computing center. As virtualization technology has been successfully applied to a variety of devices, GPU virtualization technology has developed dramatically in the past a few years. Due to the powerful performance on floating-pointing arithmetic as well as cost-efficiency, GPU virtualization has been widely studied, especially in the High Performance Computing (HPC) domain. Several research work [29, 28, 13, 30, 9] leverages GPU virtualization for general purpose computing on GPU (GPGPU). Based on the interception of vendor specific library such as Nvidia CUDA, AMD Accelerated Parallel Processing and OpenCL, the GPU resources are efficiently shared in the virtualization environment. Besides, HPC applications running on systems including GViM [13], vCUDA [30], rCUDA [9], etc. has a competitive performance with those running in a native, non-virtualized environment.

In addition to GPGPU, the other main application scenario of the GPU is for graphics processing including gaming, 3D rendering and so on. Techniques of GPU virtualization for graphics processing such as VGA-passthrough [25] and GPU paravirtualization [20, 7] are reaching their maturity. For example, VMware player 4.0 achieves 95.6% of the native performance using paravirtualization (3DMark06

with Windows 7 as both the guest and host), while VMware Player 3.0 released four years ago achieved only 52.4%. Due to these technological advances, there is an increasing number of data centers dedicated to GPU computing tasks such as cloud gaming and video rendering. Taking cloud gaming for instance, the platform renders games remotely and streams the result over the network so that clients can play high-end games without owning the latest hardware. Many cloud gaming service providers such as OnLive [1] became publicly known in the past four years. OnLive is currently partnering with more than 90 publishers and servicing close to 300 games online.

However, how a graphics card is shared among games running on top of the Virtual Machines (VM)s is not well studied. Resource sharing in existing virtualization solutions is often poor. For example, while OnLive runs multiple instances of a game that requires very little or no GPU computation, it allocates one GPU per instance for other games [15]. Proprietary motherboards are also used to host more GPU adapters in one machine. On the other hand, game developers heavily optimize their products to meet the capacity of mid-range hardware. Hence allocating a whole graphics card to some game causes the waste of hardware resources.

This paper proposes VGRIS, a scheduling framework for **V**irtualized **G**PU **R**esource **I**solation and **S**cheduling. VGRIS transparently enables different VMs in the cloud to share a single GPU efficiently. Leveraging GPU paravirtualization technology, VGRIS is a lightweight resource scheduler in the host. A challenge of resources management on GPU is that graphics processing such as frame rendering is executed in an asynchronous and non-preemptive manner. Specifically, VGRIS adopts *library API interception* so as not to care about the underlying scheduling. Different from GViM, vCUDA and rCUDA, VGRIS intercepts the library for graphics processing such as DirectX and OpenGL, instead of the one for GPU programming. One major benefit of the library API interception is that we only need to modify a few binary within the intercepted library. No other part of the software stack on top of the physical machine needs to be changed to embrace VGRIS. Moreover, VGRIS does not need any source code or design information of the library in order to perform such modification.

Similar as our previous storage scheduling system does [32], we implement two scheduling policies which address the trade-off between the Service Level Agreement (SLA) and the throughput, based on VGRIS framework. More specifically, SLA-aware scheduling strives to achieve SLA requirements for each VM, which can benefit cloud gaming platforms. However, the GPU may not be fully utilized under this scheduling policy. Another policy, Proportional-share scheduling, allocates GPU resources to each VM in proportion to its given weight, which can benefit job prioritization in rendering farms and the total throughput at the cost of SLA. Furthermore, VGRIS introduces the hybrid scheduling that guarantees minimum resources for SLA while proportionally shares surplus resources among all VMs. This third policy has better resource utilization than SLA-aware scheduling and it prevents starvation that may occur with proportional sharing.

Our experimental results show that all the three scheduling policies satisfy their design goals under various workloads. For example, applying SLA-aware scheduling, the average Frames Per Second (FPS) of workloads increases by 65%. The percentage of frames with excessive latency drops to 3.19%. In the meanwhile, the GPU performance overhead incurred by VGRIS is limited to 3.66%.

The contributions of this paper are summarized as follows.

- We propose a GPU scheduling framework based on the GPU paravirtualization architecture, which can be applied to servers for various GPU computing tasks for efficient resource management. Benefited from library API interception, VGRIS is lightweight and requires no source code level changes in the guest OS, the guest game and the host graphic drivers.
- We implement three scheduling policies in the proposed framework for different typical performance needs: high performance of SLA, proportional resource sharing and performance and fairness trade-offs.
- We implement the VGRIS through real games and benchmark programs to demonstrate the effectiveness of our framework and scheduling policies.
- We conduct several experiments with various types of workloads. The overhead of our framework is limited to 3.66%.

The rest of the paper is organized as follows. Section 2 describes motivating experiments to show the poor performance of the default scheduling mechanism for GPU. Section 3 introduces the framework of VGRIS as well as the design and implementation of the three scheduling policies we integrate in VGRIS. Section 4 presents the experimental results of the proposed VGRIS with real games and benchmark programs. Section 5 is the related work, and Section 6 concludes the paper with a discussion.

2. MOTIVATION

This section mainly describes some motivating experiments to show the poor performance and low utilization of the default GPU resource scheduling as well as the analysis of the problem. We conducted the experiments on the machine with mid-range CPU and an ATI HD6750 graphics card. Before analyzing the poor performance and low utilization of running multiple VMs on a single graphics card, we first briefly describe the standard 3D rendering and programming model and then discuss how the original graphics library schedules GPU resource.

2.1 GPU Computation Model

One of our objectives is to control the FPS of workloads, and hence the GPU resources can be scheduled. However, the real world games such as DiRT3, in fact, seem not run at the same or a close FPS during the process of gaming. The FPS may continuously vary with the change of game scenes. Basically, the GPU processing as well as the CPU computation determines the FPS. As shown in Figure 1, GPU computation for various applications, e.g., gaming, rendering, stream processing, is usually processed in an infinite loop [27]. Each loop determines exactly one frame.

First `UploadComputeKernel` uploads the computation program to the GPU, and `DeclareThreadGrid` specifies the

[1] OnLive, Inc. OnLive. http://www.onlive.com/.

```
UploadComputeKernel();
DeclareThreadGrid(&Threads);

While(1) {
  CPUComputation();
  // copy data from memory to GPU buffer
  UploadData(&VGA_Buf, &Input_Buf);
  // GPU computation
  DispatchComputation(&Threads);
  // send results back to memory
  DownloadData(&VGA_Buf, &Output_Buf);
}
```

Figure 1: GPU Computation model.

number of threads for the computation. After the initial setup, each iteration of the loop performs some tasks, e.g., drawing a frame for gaming and rendering arithmetic calculations for general-purpose computation. There are four stages. First some CPU computation prepares the data for GPU, e.g., calculating objects in the upcoming frame according to the game logic. The data is uploaded to the GPU buffer next, and then the GPU performs the computation, e.g., rendering, using its buffer contents. Finally, the calculation result is sent back to the main memory for the next iteration or output to the screen. The GPU computation library depends on the application, e.g., Direct3D or OpenGL for gaming and rendering, DirectCompute, OpenCL, or CUDA for general-purpose GPU computation. The detailed API calls vary too, e.g., `glutSwapBuffers()` for OpenGL and `IDirect3DSwapChain9::Present()` for Direct3D 9. Since we mainly focus on the graphics processing of a specified GPU, we evaluate VGRIS framework using Direct3D library which is the most popular graphics processing library among game vendors on the planet. The design principle applies to other libraries and platforms as well.

Under the Direct3D architecture, graphics API calls are asynchronous. Each application has its own Direct3D command queue, and a command is non-blocking unless the queue is full. Direct3D runtime decides when to submit the queue to the device driver. Using library API interception, such Direct3D API invocation will firstly notify VGRIS framework before executing the API. Under the framework, we implement various policies for GPU resource isolation and scheduling among multiple VMs. Since all Direct3D APIs are processed at the host, Paravirtualization greatly facilitates our design and implementation. The paravirtualization technology will be further discussed in Section 3.1. Therefore we implement VGRIS in the host and in the meanwhile, neither the guest application, the guest OS nor the host graphics drivers need to be changed.

2.2 Inefficiency of Default GPU Sharing

We now present the experiments to show the potential improvements of throughput in a shared GPU environment while guaranteeing SLA of each workload. To illustrate the potential improvements of throughput, we first evaluate the performance of the individual workloads on the platform with windows 7 as the host OS. We choose five popular games listed on GameSpot [2] on November the 10th, 2011. The version of the graphics library is Direct3D 9. Table 1 shows the performance results, in which the GPU usage is calculated based on hardware counters. Usually, cloud gaming requires the FPS rate in the range of 30 to 60 for smooth user experience. The lower rate will make the game unplayable while higher rate does not make a difference for human eye. As we can see in the figure, all the workloads are able to provide a smooth user experience. But running these workloads individually results in waste of GPU and CPU resources though the corresponding FPS is fast enough to provide a smooth user experience. For instance, the workload of DiRT3 only occupies about half of the GPU utilization and 39.61% CPU utilization when providing a smooth FPS. The rest of the GPU and CPU resources are sufficient enough to play another game, even on our mid-range ATI HD6750 graphics card. Since cloud gaming service providers like OnLive upgrade their CPUs and GPUs to the latest every six months [15], running these games with dedicated GPUs will inevitably cause unnecessary low GPU utilization.

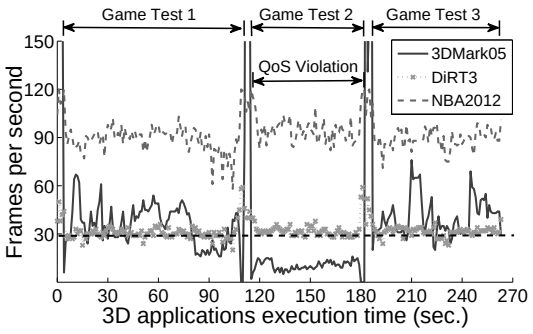
Table 1: Game performance on iCore7 2600K + HD6750.

Game	FPS	GPU Usage	CPU Usage
DiRT 3	67.14	56.14%	39.61%
Portal 2	212.70	94.77%	85.42%
Shogun 2	64.76	84.33%	29.48%
Call Of Duty 7	68.97	73.48%	69.09%
NBA 2012	104.57	69.50%	86.45%

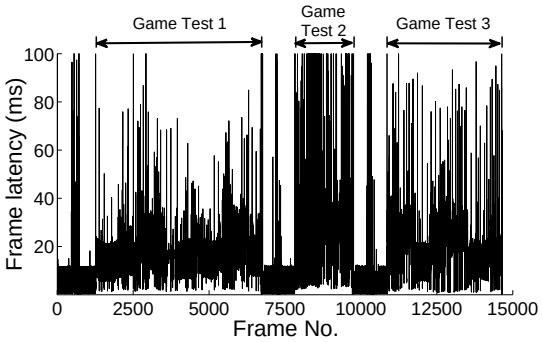
Based on this observation, we then show the performance results of running multiple VMs on the single ATI HD6750 graphics card, as shown in Figure 2. The experiment involves in three workloads: DiRT3, 3DMarks05 and NBA2012. DiRT3 and NBA2012 are two popular games while 3DMarks05 is a 3D benchmark that renders several game scenes and measures the FPS. Each workload concurrently runs in a separate VM which is configured with Windows 7 as the guest OS supporting the Direct3D graphics library. In Figure 2a, DiRT3 has an average FPS of 31 while NBA2012 runs at around 90 FPS. Compared to their original performance with the same game configuration, their FPS reduce a little due to the GPU resources contention. However, from 115th sec. to 180th sec., the second game test of 3DMarks05 runs at a FPS below 30 FPS, which offers a rough user experience. Except for FPS, the user experience also depends on the frame latency which defines the cost time of one frame. Figure 2b illustrates the corresponding frame latency of the second game test scene in 3DMark05 in Figure 2a. As we can see, the latencies of more than 6.22% frames are beyond 33 ms. The maximum latency is 388.82 ms. The larger frame latency is, the more difficult the user can play the game.

One likely reason of the default poor resource scheduling mechanism is the asynchronous and non-preemptive nature of GPU process. For instance, the default GPU scheduling mechanism in Direct3D runtime library tends to allocate resources on a first-come first-serve manner, which results in excessive FPS for low-end games and unplayable FPS for GPU demanding games when they are running concurrently on separate VMs. Graphics APIs also typically work in an asynchronous way to maximize hardware performance. APIs such as `Present` in Direct3D immediately return when they issue a GPU command and submit to the GPU. The GPU maintains a command buffer for the coming request

[2]GameSpot. http://www.gamespot.com/.



(a) FPS of Three Workloads



(b) Frame Latency of 3DMark05

Figure 2: Default scheduling results in poor performance under heavy contention.

from the user space. Therefore, if the underlying command buffer is full, the 3D application has to be blocked for some time. Take Direct3D applications for example. In a typical 3D application development, every 3D application creates a unique Direct3D device to represent its own graphics context. The Direct3D calls issued by an application is usually converted into device-independent commands, batched in a command queue within the application's context. When the command queue is full or at any appropriate time, the Direct3D runtime submits the current device's command queue to the underlying GPU driver. The driver stores the coming queue into its local command buffer for the GPU cores to process asynchronously. There are commands still kept by Direct3D runtime for a period of time until available room is found in the command buffer at the driver side. Thus, if two or more 3D applications run concurrently on a single graphics card, the resources contention inevitably occurs. If one 3D application runs a little fast and submits its command queue frequently to the underlying layer, it probably gets more GPU resources. Meanwhile, another 3D application hence suffers from severe starvation, causing its FPS low as it is running. Besides, it is noteworthy that a 3D application needs to recreate resources after its windows has been updated. Hence, it is common that only one GPU-accelerated 3D application occupies the whole GPU for a period of time regardless of how many cores or threads the GPU has.

Based on the aforementioned analysis, we focus on the graphics runtime library. If we can intercept DirectX APIs, especially the ones related to GPU rendering, we are able to do some scheduling for all the running 3D applications. It

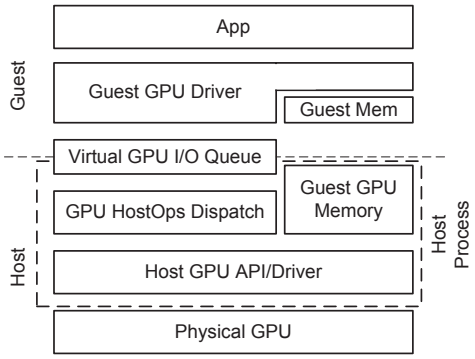


Figure 3: GPU Paravirtualization Architecture.

also brings an additional benefit that no modification is required for the 3D application, the underlying driver and the hypervisor. For instance, if a 3D application runs a little fast, we can make it slower so that other 3D applications get more change to access the GPU and will not starve any more. We are also able to assign each 3D application a priority and hence a 3D application with high priority is capable of getting more resources and responsive on the GPU. Currently, we integrate three scheduling policies into our VGRIS framework. One is for guaranteeing SLA, one is for high throughput and the other mixes the former two to balance the trade-off between SLA and throughput. Other scheduling policies are applicable to the VGRIS framework as well.

3. VGRIS ARCHITECTURE

This section mainly discusses the design and the implementation of the VGRIS as well as the three algorithms we have incorporated in it. Before introducing VGRIS architecture, we first present the necessary background of GPU paravirtualization since VGRIS leverages the technology. The three algorithms address different requirements for different GPU computing applications.

3.1 GPU Paravirtualization and VGRIS Framework

Paravirtualization provides virtual machines a software interface different from the underlying hardware. This interface significantly reduces the overhead of operations which are substantially more difficult to run in a virtual environment. The guest operating system must be explicitly ported to exploit the new interfaces for better performance. For commercial operating systems that cannot be modified, this is often achieved by paravirtualization-aware device drivers. Due to the complexity of GPU device drivers, hypervisors that support GPU paravirtualization achieve near-native efficiency only recently.

Figure 3 shows the typical GPU paravirtualization architecture for type 2 (hosted) hypervisor [7]. Typically a GPU rendering task issued by a guest Operating Systems (OS) application is executed as follows. After the guest application invokes a standard GPU rendering API, the guest GPU computation library, e.g., OpenGL, Direct3D, DirectCompute, CUDA, prepares the corresponding GPU buffer contents in main memory and issues the GPU command packets. These packets are pushed into the virtual GPU I/O

queue, which are subsequently processed by the HostOps Dispatch in the host. Finally, this dispatch layer sends the commands to the device driver in an asynchronous manner. Buffer contents in guest OS memory are transferred to the GPU buffer using Direct Memory Access (DMA) through this process. We choose Windows 7 x64 as the guest OS running on VMware player 4.0 since it is most compatible with commercial games, especially high-end ones. Running 3DMark06 on guest OS with both Windows 7 x64 and Ubuntu 11.04 x64 hosts, the FPS are 95.5% and 62.9% of the native performance, respectively. Therefore we use Windows 7 x64 as the host for all our experiments.

Figure 4 is the architecture of VGRIS within the paravirtualization framework shown in Figure 3, where modules introduced by VGRIS are highlighted in grey. These modules are all inside the host. There is one *agent* for each VM, which schedules GPU computation tasks and monitors the performance. In addition, there is a centralized scheduling controller that serves two purposes. First, it receives commands from the administrator to decide which scheduling algorithm to use. Second, under the hybrid scheduling policy, it automatically selects between the SLA-aware and proportional-share policy based on the performance feedback received from all agents. The content and frequency of the performance report from each agent are specified by the central controller too. Some scheduling algorithm does not require any feedback at all. In our prototype implementation, each agent simply intercepts Direct3D API invocations from GPU HostOps Dispatch for rescheduling. Its performance monitoring function utilizes GPU performance instrumentation methods. The centralized scheduling server is implemented as an independent process.

Similar to our previous design [32], we implement three representative scheduling policies for different optimization goals. The brief introductions of the three policies are as follows.

- **SLA-aware scheduling** allocates just enough GPU resource to each VM to fulfill its SLA requirement. However, the GPU resources may be not fully used under this policy.
- **Proportional-share scheduling** allocates all GPU resources to all running VMs in proportion to their weights assigned by the administrator. Due to the mistake or thoughtlessness of the administrator, some VM may not fulfill the SLA requirement.
- **Hybrid scheduling with a compromise** mixes the above two schemes. It first allocates minimal amount of resource to each VM so its SLA is satisfied, surplus resource is then proportionally allocated to all VMs to maximize GPU utilization.

3.2 Scheduling Policies

Currently, VGRIS mainly integrate three scheduling policies. Other scheduling algorithms are applicable to VGRIS architecture as well.

SLA-aware Scheduling SLA requirements in cloud gaming service providers try to guarantee a minimum FPS and a maximum latency for smooth user experience. As Figure 2a illustrates, the default GPU scheduling algorithm allocates resources fairly under contention. As a result, even if the SLA requirement is the same for all VMs, less GPU demanding ones may get more resources than necessary while GPU demanding ones cannot meet the requirement. SLA-aware scheduling is designed to address this issue. It allocates just enough resource for each VM to guarantee its SLA. To achieve this goal, we slow down less demanding applications to free extra resources for more demanding applications. We use the application of cloud gaming to illustrate this idea. The solution can be extended to other applications.

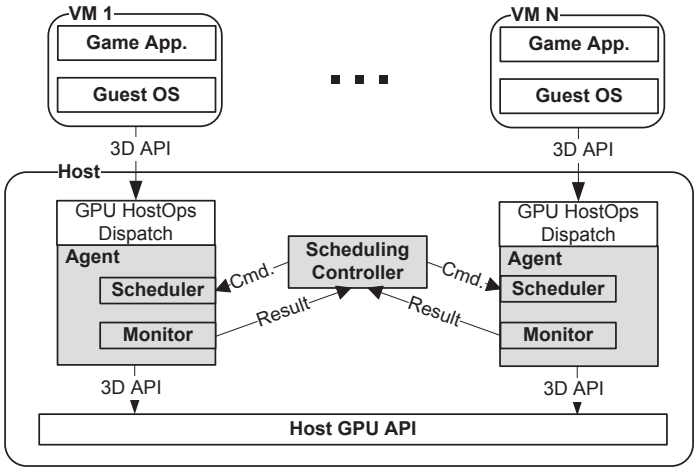


Figure 4: VGRIS Architecture.

For smooth and responsive gaming experience, the latency of each frame must be in the range. Maximum latency is always implied by cloud providers' SLA. Therefore we consider the latency requirement as our SLA objective.

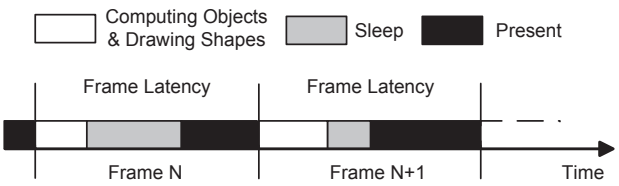
Computer games follow the same GPU computation model in Figure 1, where each iteration calculates and displays exactly one frame. For example, Figure 5a is the pseudocode using Direct3D. Methods `ComputeObjectsInFrame`, `DrawPrimitive`, and `Present` correspond to `CPUComputation`, `UploadData`, and `DispatchComputation` in Figure 1, respectively. In computer gaming, there is no need to send the result back to main memory. Instead, the GPU outputs the calculated frame through its external interfaces either to a screen or the network (after hardware compression). After the `Present` call returns, we have no direct control over when the frame becomes visible. However, the extra delay is negligible in the case with a local display. If the frame is displayed remotely, we assume a fixed amount of network delay. Therefore, we consider a frame latency as the time duration in-between the returns of two consecutive `Present` calls, illustrated in Figure 5b.

To stabilize the frame latency according to a given SLA, we extend each frame by delaying its last call, `Present`. This is achieved via inserting a `Sleep` call before `Present`. The amount of delay should be equal to the desirable latency subtracted by the computation time of `ComputeObjectsInFrame`, `DrawPrimitive`, and `Present` altogether. While VGRIS measures the computation time of the former two operations, the computation time of `Present` can only be predicted.

Fortunately we observe that the computation time of `Present` is very stable for each game application running in a VM, because it is mostly affected by the complexity of the scene, which changes only gradually. Furthermore, since each agent predicts the computation time based on its own historical

```
While(1) {
  ComputeObjectsInFrame();
  DrawPrimitive(&VGA_Buffer);
   Sleep(calculated_sleep_time);
  Present(&VGA_Buffer);
}
```

(a) Pseudocode under Direct3D



(b) Frame Latency

Figure 5: SLA-aware Scheduling Approach.

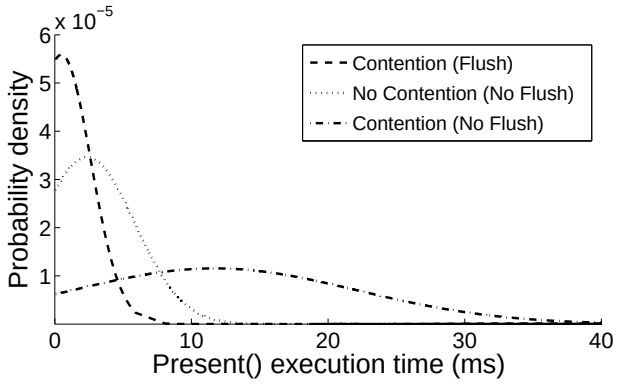


Figure 6: Probability distribution of `present` time cost.

information only, GPU context switch [18] has little impact on prediction accuracy of an individual agent. Therefore our prototype implementation simply uses the average time of the past twenty `Present` calls as the prediction for the upcoming one.

We also observe that the computation time of `Present` varies. When there is heavy contention, the average execution time of `Present` raises from 2.37 ms to 11.70 ms, as shown in Figure 6. This is because the DirectX runtime batches Direct3D commands for better efficiency. Hence, heavy contention increases the possibility of full command buffer, resulting in the execution time of `Present` less predictable. The `Flush` command can mitigate the problem significantly. Figure 6 shows that the average computation time of `Present` is reduced from 11.70 ms to 0.48 ms under heavy contention. The `Flush` command induces extra CPU computation cost. Since we mainly consider GPU bound VMs, it is reasonable to spend a little extra CPU time for more accurate prediction, and therefore more stable latency of each frame. We insert `Flush` in each iteration immediately before `Sleep` so we can measure, instead of estimate, its computation time.

Proportional-share Scheduling The SLA-aware scheduling strives to meet SLA requirements, which may result in low resource utilization when there are insufficient numbers of VMs. For applications such as offline rendering and general-purpose computation on the GPU, we may want to fully utilize the resources while ensuring each VM gets a fair amount of shares. Proportional sharing is a scheduling mechanism that is very well suited for these application scenarios.

Our proportional-share scheduling algorithm adopts the *Posterior Enforcement Reservation* policy used in TimeGraph [18], which queues and dispatches GPU commands based on task priorities. First each VM i is assigned a share s_i that represents the percentage of GPU resource it can use in each period t. The shares of all VMs add up to one. Budget e_i is the amount of GPU time that VM i is entitled for execution. This budget is decreased by the amount of time consumed on the GPU, and is replenished by at most ts_i once every period t as follows

$$e_i = \min(ts_i, e_i + ts_i). \quad (1)$$

The proportional-share scheduling dispatches `Present` API invocation if the budget for the corresponding VM is greater than zero, otherwise it postpones the dispatch. We set $t = 1$ ms in our implementation, which is sufficiently small to prevent long lags.

Hybrid Scheduling SLA-aware scheduling may result in low GPU utilization with an insufficient number of VMs. On the other hand, proportional-share scheduling can maximize utilization but inappropriate weights can lead to starvation of some VM. Our hybrid scheduling mechanism combines the benefits of the two by automatically choosing the appropriate algorithm with calculated parameters. In order to achieve this, we introduce a centralized scheduling controller that monitors the performance of each VM and coordinates all agents.

Algorithm 1 Hybrid scheduling algorithm. $FPSthres$ is the minimal acceptable FPS; $GPUthres$ is the preferred minimal overall GPU usage; $Time$ is the maximum bearable duration for unsatisfied feedbacks.

```
1: while each second do
2:    if CurrentAlgo = PropShare and
      FPS < FPSthres for Time sec then
3:       CurrentAlgo ← SLAAware
4:    else if CurrentAlgo = SLAAware and
      GPUTotalUsage < GPUthres for Time sec then
5:       CurrentAlgo ← PropShare
6:       CalcShareForAllVMs()
7:    end if
8: end while
```

The scheduling controller collects the performance information from each VM every second. It determines the appropriate scheduling algorithm for all VMs based on user pre-defined criteria settings. When initialized, hybrid scheduling algorithm retrieves the threshold values from user settings and employs proportional-share scheduling with a fair share as the default algorithm. During runtime, any reported status below the criteria for the wait duration will lead to changing the scheduling algorithm among all agents. For example, the administrator may indicate the wait duration is 5 seconds. If proportional-share scheduling is leveraged as the current scheduling algorithm for all VMs, hybrid scheduling uses SLA-aware scheduling algorithm if and only if some VM has a low FPS for five seconds. On the contrary, the proportional-share scheduling algorithm is selected if the

current scheduling method is SLA-aware scheduling and the physical GPU usage is below a certain criteria for 5 seconds.

The hybrid scheduling algorithm needs to determine the proper share for each VM when switching to proportional-share scheduling algorithm, as illustrated in Line 6 in Algorithm 1. The proportional share for the i-th VM (s_i) is achieved as follows:

$$s_i = u_i + \frac{(1 - \sum_{i=1}^{n} u_i)}{n}. \quad (2)$$

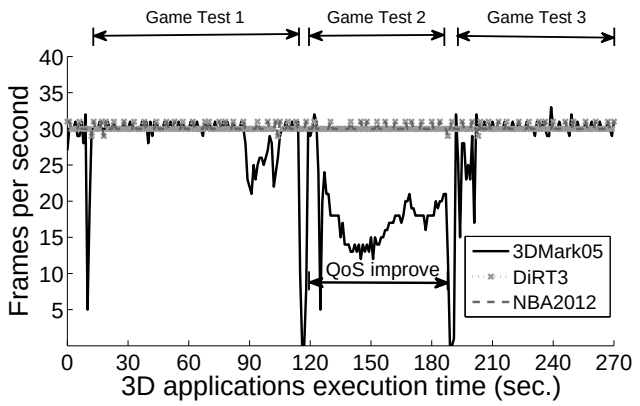
This formula approaches proportional sharing while guaranteeing SLA for each VM. u_i means the GPU usage of the i-th VM. It represents the minimum share of GPU resource needed when switching to proportional-share scheduling. Meanwhile, $(1 - \sum_{i=1}^{n} u_i)/n$ represents the fairness division of the abundant GPU resource to each VM. This fairly division permits that every VM owns more GPU resource than required to fulfill the SLA requirement in the current situation.

4. EXPERIMENTAL EVALUATIONS

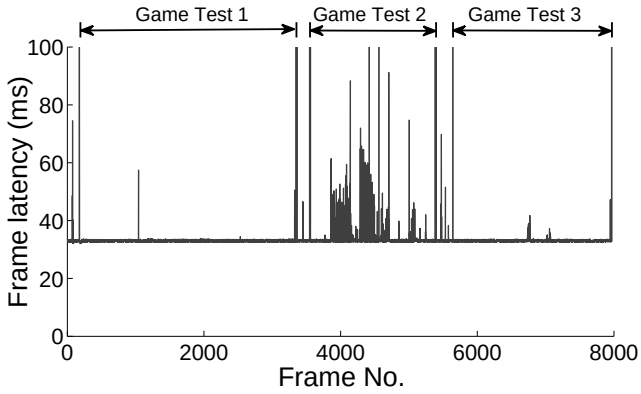
We now provide a detailed quantitative evaluation of V-GRIS. All the experiments are conducted with the same workloads for the three scheduling policies. First, we evaluate SLA-aware scheduling in case of under-provision GPU resource. Then, we evaluate proportional-share scheduling's ability in maximizing GPU resource usage. Thirdly, we evaluate the effectiveness of hybrid scheduling. At last, we provide the micro- and macro-analysis to evaluate VGRIS's performance impact to guest legacy software.

The configurations of the testbed and VMs are derived from the top 5 most popular games listed in Table 1. The testbed is configured with i7-2600k 3.4GHz CPU, 16GB RAM, and an ATI HD6750 graphics card. Each hosted VM owns dual Cores and 2GB RAM. Windows 7 x64 is used as both the host OS and guest OSes. All the games are running under high graphic quality with 1280×720 resolution. To simplify performance comparison, swap space and GPU-accelerated windowing system are disabled on the host side.

We use two different types of workload and one benchmark in the following experiments. The first workload group, named *Ideal Model Games*, has almost fixed objects and views and hence a stable FPS is maintained. Many strategy games belong to this type. We choose PostProcess, ShadowVolume, Parallax and LocalDeformablePRT from DirectX 9.0 SDK samples as the representations of this kind of workload. The other workload group is the *Reality Model Games*, whose FPS keeps constant for a short time but varies from minute to minute. Games of First Person Shooter genre and Sports genre mainly constitute this group of games. We pick DiRT 3, Portal 2, and NBA2012 as the representative games. The 3DMark benchmarks (including 3DMark05 and 3DMark06) are also employed as the Reality Model Games because they satisfy the features mentioned above. 3DMark05 doesn't fully employ GPU resource. It sequentially runs three Game Tests (GT), in which GT1 doesn't consume all the GPU resources to produce a high FPS. GT2 and GT3 consume all the GPU resources, but the GT2 produces below 55 FPS for more than 30% of running time while the FPS of GT3 is below 70 FPS only within 5% of running time. 3DMark06 maximizes the GPU resource usage in all two game tests and two High Dynamic Range Tests (HDRT).



(a) FPS of Three Workloads



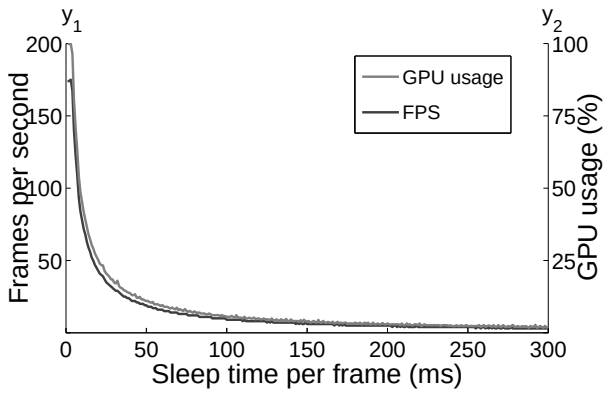
(b) Latency of 3DMark05

Figure 7: SLA-aware scheduling improves performance.

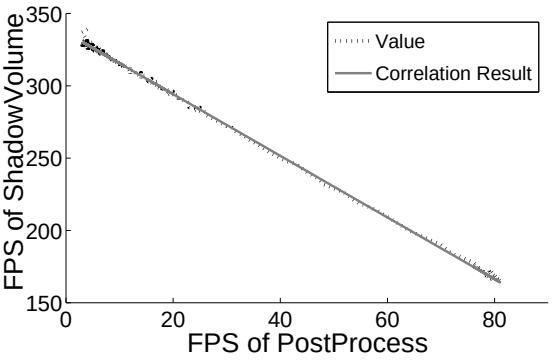
4.1 SLA-aware Scheduling Evaluation

We first evaluate SLA-aware scheduling provided by V-GRIS. We evaluate the policy with three workloads concurrently running in separate VMs and sharing one single graphics card. Using the same configurations with the experiments in Section 2, Figure 7 shows result improvements, compared with Figure 2. In Figure 7a, the average FPS of the GT2 rises 65.05% after SLA-aware scheduling. Also, the percentage of frames of excessive latency drops to 3.19% in Figure 7b, with the maximum value decreasing to 131.27 ms. Insufficient GPU hardware capability indicates that it cannot meet the FPS criteria (≥30 FPS) for all three games at the same time in Figure 7a.

Next, we evaluate SLA-aware scheduling's effectiveness in controlling the FPS and GPU resource usage of the only VM. PostProcess application is used in this experiment and consumes 100% of GPU resource without control. The initially complete GPU resource usage is achieved by setting the resolution at 1920 × 1200 as well as enabling the Bloom effect specifically in the PostProcess application. An initial sleep time per frame of 300 ms is set by VGRIS which then decreases the sleep time by 1 ms in each second. Figure 8a depicts that sleep time (x) is approximately reciprocal with FPS (y_1) and GPU usage (y_2), similar to frame latency. The correlation result of the sleep time (x) and FPS (y_1) is $y_1 = 646.11x^{-0.927}$ with the correlation coefficient (R^2)



(a) FPS and GPU usage result for one VM



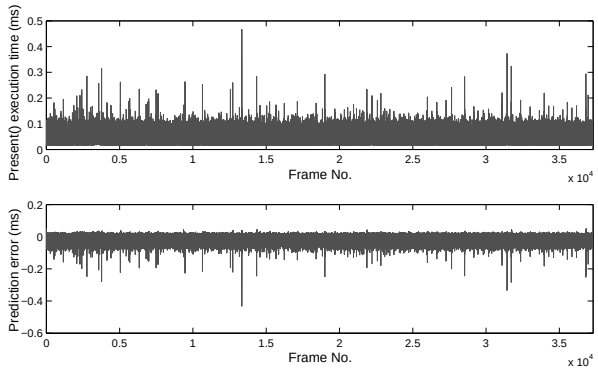
(b) FPS result for two VMs

Figure 8: Scheduling effectiveness on Ideal Model Games.

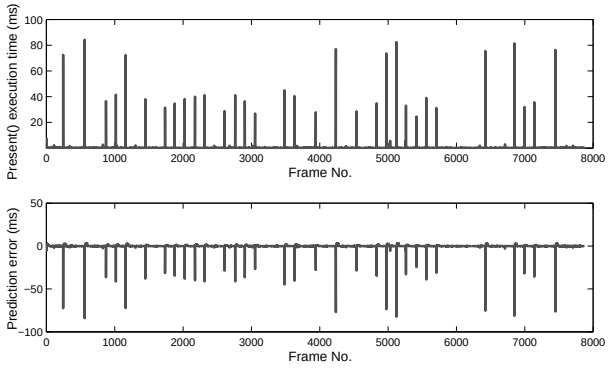
to be 0.9828. Meanwhile, the correlation result of the sleep time (x) and GPU usage (y_2) is $y_2 = 332.33x^{-0.887}$ with R^2 to be 0.9838. The reason is straightforward according to the definition of FPS. Also, the loop-based GPU rendering model results that GPU resource usage shares the same trend with FPS.

In order to evaluate the impact to other VMs' FPS when controlling one VM only, we execute multiple VMs concurrently. Based on the same configuration in the last experiment, a new VM running ShadowVolume is introduced in. Its resolution is also set at 1920×1200 to fully consume GPU resource individually. As shown in Figure 8b, when the FPS of PostProcess increases, the FPS of ShadowVolume decreases in an approximately linear way. The correlation result of FPS value satisfies $y = -2.13x + 336.8$ and R^2 equals 0.9981. This experiment proves that the amount of GPU and CPU resource stripped from one VM can be acquired by other VMs and hence VGRIS can effectively control the GPU resource on multiple VMs.

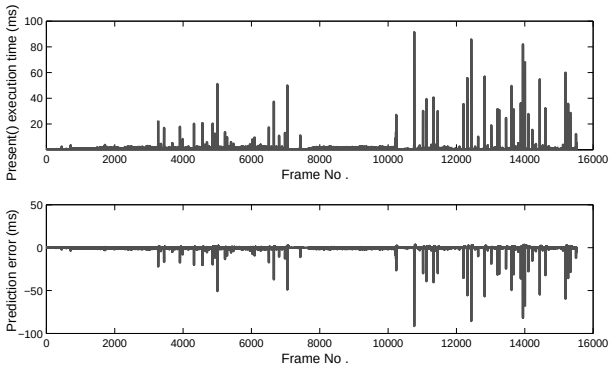
Figure 9 depicts both actual and prediction of `Present` API execution cost in two game models: Ideal Model Games and Reality Model Games. The GPU resource is fully consumed and we record the result for a 60-second period. Figure 9a shows that the proposed prediction approach achieves 0.4 ms error margin. When GPU resource competition occurs, though the prediction error increases to -84.17 ms at most, only 4.12% of the frames have the predicted `Present`



(a) LocalDeformablePRT - No GPU Contention



(b) LocalDeformablePRT - With GPU Contention



(c) 3DMark05 - With GPU Contention

Figure 9: Errors of `Present` API execution cost prediction.

API execution costs more than 2 ms error margin (Prediction Failures), as shown in Figure 9b. Meanwhile, even for the Reality Model Games in contention case, the percentage of prediction failures is 1.95% and the maximum prediction error is -91.32 ms, as presented in Figure 9c. It is noteworthy that a 2 ms prediction error only results in an instant decrease from 30 to 28.30 FPS or from 60 to 53.57 FPS. This is acceptable in the frequent long-time gaming experience.

4.2 Proportional-share Scheduling Evaluation

We next demonstrate the effectiveness of proportional-share scheduling in regulating GPU resource usage according to user settings. Figure 10 shows the GPU resource usage of Reality Model Games using different initial GPU shares: (1) NBA2012 is set to use 30% GPU resource while requiring 44.48% GPU resource individually. (2) Looped

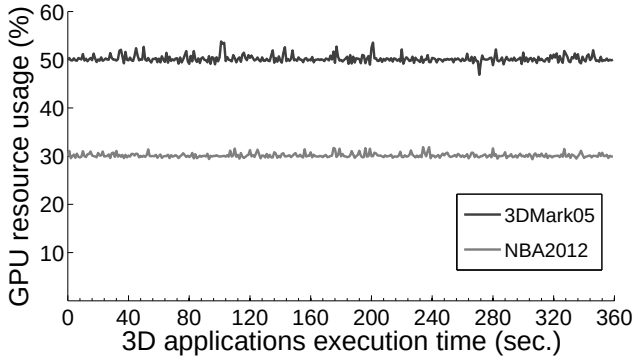


Figure 10: Proportional-share scheduling evaluation result.

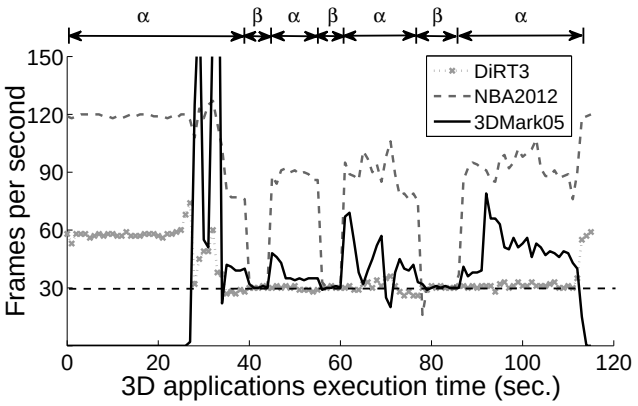


Figure 11: Hybrid Scheduling Results.

Table 2: Performance comparison of proportional-share scheduling and SLA-aware scheduling.

	SLA-Aware Scheduling		Proportional-Share Scheduling	
	FPS	GPU Usage	FPS	GPU Usage
NBA2012	30.12	12.41%	79.16	30.12%
3DMark05	30.30	26.95%	61.76	50.30%

GT 1 in 3DMark05 is set to use 50% GPU resource while requiring 70.82% GPU resource individually. Especially, we measure the GPU usage discrepancy within each instance in Figure 10 to evaluate the control accuracy. The results show that the range is 13.75% to the average GPU usage value for 3DMark05 and 8.31% for NBA2012. This result proves that proportional-share scheduling successfully provides user specified GPU resource share. It even works for Reality Model Games which owns the inherently dynamic nature of complex scene switches and abruptly changing in visible objects.

Furthermore, we compare the GPU resource usage of the proportional-share and SLA-aware scheduling to evaluate proportional-share policy's ability in maximizing hardware performance. Proportional-share scheduling uses the same resource allocation with the former experiment while the FPS criteria of SLA-aware scheduling is set to be 30 FPS. Table 2 depicts that proportional-share scheduling employs more available GPU resource than SLA-aware scheduling does. This is because SLA-aware scheduling limits FPS according to the user indicated FPS criteria. Meanwhile, GPU resource usage is direct proportion to the FPS for the same game. Hence, SLA-aware scheduling doesn't use GPU resource effectively due to its criteria based FPS limitation.

4.3 Hybrid Scheduling Evaluation

We now evaluate hybrid scheduling's automatic determination of scheduling algorithms with proper parameters. Three Reality Model Games are used to evaluate hybrid scheduling's effectiveness, including NBA2012, DiRT3, and 3DMark05. First, we run the NBA2012 and DiRT3 games concurrently. After 28 seconds, we start 3DMark05's GT3 which will finish its execution in 87 seconds. Figure 11 illustrates the selection of algorithms and the impacts to the FPS of running VMs. In this figure, α represents proportional-share scheduling and β stands for SLA-aware scheduling. Firstly, hybrid scheduling employs proportional-share scheduling algorithm and assigns full GPU resource for each VM since both of their FPS satisfy the FPS criteria. At the time of 40 second, the VM running DiRT3 has not got sufficient GPU resource to maintain its SLA for the most recent time which is 5 seconds according to the administrator's setting. Hence, hybrid scheduling employs SLA-aware scheduling to release the excessive GPU resources in other VMs. However, this results in a low overall GPU usage and hybrid scheduling switches back to proportional-share scheduling after duration. Because hybrid scheduling always fairly divides the abundant GPU resource and assigns them to each VM, it can be observed that VM's FPS increases when switching back to the proportional-share scheduling algorithm. In the rest of Figure 11, the algorithm selection always follows the above mechanism.

4.4 Performance Discussions

In order to evaluate VGRIS's performance impact to legacy applications and OSes, we first perform micro analysis to illustrate the potential hot spot. PostProcess and 3DMark06 GT1 are leveraged to fully utilize available GPU resource. We only evaluate the execution cost of each part in SLA-aware scheduling and proportional-share scheduling. The hybrid scheduling is not included because there are only trivial changes based on the other two scheduling methods and the performance impact can be ignored. Figure 12 shows the microbenchmark results. The execution time of SLA-aware scheduling constitutes four parts, in which the GPU command flush operations contribute the main performance overhead. This is due to the design of current Direct3D library and the implemented flush strategy in VGRIS prototype. It's possible to achieve a better result by adopting different flush strategies in the future.

Having no GPU Command `Flush` operation, proportional-share scheduling contains three parts in its execution time. For the same reason, the `Present` API execution time unsurprisingly becomes the most expensive operation. It is noteworthy that no aggressive flush of Direct3D command buffer is added in proportional-share scheduling, because proportional-share scheduling always assumes the existence of over-provision GPU resource. In total, SLA-aware scheduling algorithm incurs 6.74% overhead for PostProcess while

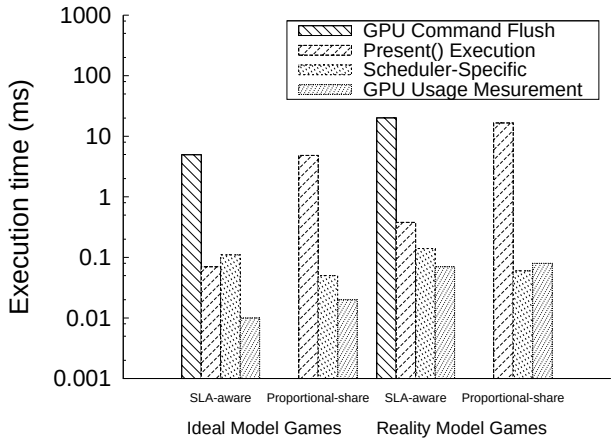


Figure 12: Microbenchmark Results.

Table 3: Macrobenchmark Results.

	Native	SLA-aware Scheduling		Proportional-share Scheduling	
	FPS	FPS	Overhead	FPS	Overhead
GT1	43.023	42.044	2.28%	42.221	1.86%
GT2	48.686	45.996	5.53%	48.284	0.83%
HDRT1	59.062	57.923	1.93%	57.700	2.31%
HDRT2	65.808	62.854	4.90%	65.984	-0.27%

24.01% for 3DMark06. The results for proportional-share scheduling are 1.56% and 0.11%.

When considering the infrequent invocation of `Present` API, the performance overhead is significantly decreased. 3DMark06 with default settings is used to evaluate VGRIS's performance overhead in application level. Table 3 shows the evaluation results, proving VGRIS brings in 3.66% performance overhead in average for SLA-aware scheduling while 1.18% for proportional-share scheduling. Thus, the scheduling methods provided by VGRIS are demonstrated to incur slight performance overhead. Moreover, VGRIS is even able to provide the same SLA with that provided by commercial cloud gaming services (e.g., OnLive). Our evaluation result shows that VGRIS is able to run one DiRT3 and two Portal 2 instances concurrently with the FPS criteria set to be 60. As a result, VGRIS is able to execute multiple game VMs concurrently while ensuring acceptable SLA individually.

5. RELATED WORK

Virtualized resource management is an active area of research over the past decade. Based on the general trend of all related research works, we can broadly classify them into three groups: 1)scheduling in virtualization, 2) GPU scheduling, and 3) applications of GPU virtualization.

Scheduling in Virtualization: Previous works focus on CPU and I/O scheduling including disk and network resources. Credit, Simple Earliest Deadline First (SEDF), Borrowed Virtual Time (BVT) [10] and vSlicer [34] are available CPU schedulers [6] for general purpose hypervisor like Xen [1]. Achieving the ability of scheduling processor resource according to the indicated proportions, these methods can also be employed in the proportional-share scheduling in VGRIS. BVT is optimized for latency sensitive applications by decreasing the corresponding job's next schedule time and borrowing time slices from its future processor usage. Credit scheduling achieves the same optimization by boosting corresponding virtual CPU in the block state when an external event arrives [6]. Besides, CPU schedulers for real time guest OS control the expected latency by arranging virtual CPU run queue in certain order [21, 35]. However, these scheduling methods cannot be applied to manage GPU resources to fulfill the SLA requirement. The reason is that all of these scheduling approaches treat VM to be black box and hence ignore guest applications' SLA-related measurements. In contrast, by effective library API interception on host side, VGRIS can perform SLA-aware scheduling algorithm without modifying guest software.

For the I/O resource scheduling in virtualization, prior works mainly analyzed the scheduling methods of disk and network resources. Similar to our approach, AVATAR [36] is implemented to ensure the proportional-share scheduling of storage resources and fulfill the service level objectives. However, the dynamic change in GPU resource usage in certain kinds of GPU applications results that AVATAR possibly dissatisfies the SLA of GPU computation tasks. This issue is identified to be one important problem and solved by hybrid scheduling in VGRIS.

DVT [19] is primarily designed for network resource scheduling. It provides differential resource scheduling and gradual latency variation in case of workload capacity's change to support performance isolation for guest OS's resource management mechanisms. Stillwell *et al.* [31] focus on scheduling algorithms on distributed platforms. The algorithms can allocate resources to competing services. Based on workload data supplied by Google, the algorithms provide good performance. Compared with them, VGRIS provides both proportional-share and SLA-aware scheduling by obtaining guest application's SLA measurements.

GPU Scheduling: Previous GPU resource scheduling approaches mainly target native systems. For example, Phull *et al.* [28] present a framework to predict and handle interference and schedule GPU resources in a time-share model. Kato *et al.* [17] address the priority inversion problems of user GPU tasks in GPU-accelerated windowing systems. Elliott *et al.* [12] have presented two methods for integrating GPUs into soft real-time multiprocessor systems to improve total system performance. Maeda *et al.* [23] develop an automatic resource scheduling to accelerate stencil applications on GPGPU Clusters. A task-based dynamic load-balancing scheduling [4] is proposed for single- and multi-GPU systems. Ravi *et al.* [29] propose a framework that enable applications running within VMs to transparently share one or more GPUs. Compared with them, VGRIS mainly focuses on graphics processing including 3D rendering and gaming. Both SLA of the 3D applications in the VMs and the overall throughput are taken into account. TimeGraph [18] implements a real-time GPU scheduler to isolate performance for important GPU workloads. To achieve its design goal, TimeGraph queues GPU command groups in the driver layer and submits them according to user predefined settings as well as GPU hardware measurements. TimeGraph cannot guarantee SLA for all the VMs, especially for less important workloads. Instead, our hybrid scheduling algorithm is used to effectively provide both SLA and maximized the GPU

resource usage. Becchi *et al.* [3] add two features to improve the sharing of GPUs: dynamic application-to-GPU binding and virtual memory for GPUs. Aimed at different goals, VGRIS can further employ this work to support load balancing and solve GPU memory constraint for applications.

GERM [2, 11] aims at providing fair GPU resource allocation. Besides, fixed frame rate approaches like Vertical Synchronization (V-Sync) [3] are designed for games to avoid excessively use of hardware resource. Unfortunately, GERM fails to consider SLA requirements while fixed frame rate approaches fail to consider using hardware resource effectively. Due to fixed frame rate, both approaches are inflexible to adjust resource utility on-the-fly.

Applications of GPU Virtualization: The rapid development of GPU virtualization accelerates many new applications, especially in cloud gaming and general-purpose GPU computing.

In cloud gaming, previous studies on cloud gaming platform focus on streaming graphical content and decreasing the required network bandwidth [5, 33, 26, 16]. Li *et al.* [22] take cryo-electron microscopy 3D reconstruction as an example to present how to exploit parallelism on both CPU and GPU in a heterogeneous system. Different from them, our approach is able to run multiple game VMs sharing with GPU resource based on GPU Para-Virtualization (PV) technique.

In general-purpose GPU computing, vCUDA [30] introduces GPU computing into virtualization execution environment. It motivates our research in scheduling resources for GPU computing. rCUDA [9] and Duato's work [8] try to decrease the power-consuming GPUs from high performance clusters while preserving their 3D-acceleration capability to remote nodes. Gupta *et al.* [14] propose Pegasus that uses NVIDIA GPGPUs coupled with x86-based general purpose host cores to manage combined platform resources. Based on Pegasus, Merritt *et al.* [24] propose Shadowfax, a prototype of GPGPU Assemblies, improves GPGPU application scalability as well as increases application throughput. However, none of these approaches has studied the management of virtualized GPU resource isolation and scheduling to achieve the computational efficiency in cloud gaming which is the main focus of this paper. Comparing with them, our approach tries to improve the SLA of GPU computation on cloud platform and maximize the overall resource usage. Additionally, VGRIS provides three representative scheduling algorithms to meet multiple optimization goals in case of under- and over-provisioned GPU resource.

6. CONCLUSION

We presented VGRIS, a Virtualized GPU Resource Isolation and Scheduling framework for GPU-related computation tasks. By introducing an agent per VM and a centralized scheduling controller to the paravirtualization framework, VGRIS achieves in-VM GPU resource measurements and regulates the GPU resource usage. Moreover, we propose three representative scheduling algorithms: SLA-aware scheduling allocates just enough GPU resources to fulfill the SLA requirement; Proportional-share scheduling allocates all GPU resources to all running VMs in proportion to their weights; Hybrid scheduling provides a mixed solution to meeting the SLA requirement while maximizing the overall GPU resource usage. Using the cloud gaming scenario as a case study, our evaluation demonstrates that each scheduling algorithm enforces its goals under various workloads. We plan to extend VGRIS to multiple physical GPUs and multiple physical machine systems for data center resource scheduling as our future work.

[3]V-Sync. http://en.wikipedia.org/wiki/Vertical_synchronization/.

7. ACKNOWLEDGMENTS

Thanks for Jiewei Wu and Xi Chen's contribution to this project. We also thank for Yueqiang Cheng's and Zheng Zhang's suggestions. Also, we appreciate the valuable comments come from the reviewers. They help us in revising our work one step further. This work is supported by the Program for PCSIRT and NCET of MOE, NSFC (No. 61073151, 61272101), 863 Program (No. 2011AA01A202, 2012AA010905), 973 Program (No. 2012CB723401), the key program (No. 313035) of MOE, and International Cooperation Program (No. 11530700500, 2011DFA10850), and Shanghai Natural Science Foundation (No.12ZR1445700).

8. REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of ACM Symposium on Operating Systems Principles*, SOSP, 2003.

[2] M. Bautin, A. Dwarakinath, and T. cker Chiueh. Graphic engine resource management. In *Proceedings of Multimedia Computing and Networking*, MMCN, 2008.

[3] 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 *Proceedings of international symposium on High-Performance Parallel and Distributed Computing*, HPDC, 2012.

[4] L. Chen, O. Villa, S. Krishnamoorthy, and G. R. Gao. Dynamic load balancing on single- and multi-gpu systems. In *Proceedinigs of IEEE International Symposium on Parallel Distributed Processing*, IPDPS, 2010.

[5] L. Cheng, A. Bhushan, R. Pajarola, and M. E. Zarki. Realtime 3D graphics streaming using MPEG-4. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, BroadWise, 2004.

[6] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three CPU schedulers in Xen. *SIGMETRICS Performance Evaluation Review*, 35(2):42–51, 2007.

[7] M. Dowty and J. Sugerman. GPU virtualization on VMware's hosted I/O architecture. *SIGOPS Operating Systems Review*, 43:73–82, 2009.

[8] J. Duato, F. D. Igual, R. Mayo, A. J. Peña, E. S. Quintana-Ortí, and F. Silla. An efficient implementation of GPU virtualization in high performance clusters. In *Proceedings of European Conference on Parallel Processing*, Euro-Par Workshops, 2009.

[9] J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Ortí. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *Proceedings of the International Conference on High Performance Computing and Simulation*, HPCS, 2010.

[10] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP, 1999.

[11] A. Dwarakinath. A fair-share scheduler for the graphics processing unit. *Master Thesis*, 2008.

[12] G. A. Elliott and J. H. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems*, 48(1):34–74, 2012.

[13] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. GViM: Gpu-accelerated virtual machines. In *Proceedings of the ACM Workshop on System-level Virtualization for High Performance Computing*, HPCVirt, 2009.

[14] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan. Pegasus: Coordinated scheduling for virtualized accelerator-based systems. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, ATC, 2011.

[15] Joystiq. GDC09 interview: OnLive founder Steve Perlman, continued. `http://www.joystiq.com/2009/04/02/gdc09-interview-onlive-founder-steve-perlman-continued/`.

[16] A. Jurgelionis, P. Fechteler, P. Eisert, F. Bellotti, H. David, J.-P. Laulajainen, R. Carmichael, V. Poulopoulos, A. Laikari, P. H. J. Perälä, A. D. Gloria, and C. Bouras. Platform for distributed 3D gaming. *Int. J. Computer Games Technology*, 2009.

[17] S. Kato, K. Lakshmanan, Y. Ishikawa, and R. R. Rajkumar. Resource sharing in GPU-accelerated windowing systems. In *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS, 2011.

[18] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, ATC, 2011.

[19] M. Kesavan, A. Gavrilovska, and K. Schwan. Differential virtual time (DVT): rethinking I/O service differentiation for virtual machines. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC, 2010.

[20] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara. VMM-independent graphics acceleration. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE, 2007.

[21] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik. Supporting soft real-time tasks in the Xen hypervisor. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE, 2010.

[22] L. Li, X. Li, G. Tan, M. Chen, and P. Zhang. Experience of parallelizing cryo-em 3D reconstruction on a CPU-GPU heterogeneous system. In *Proceedings of the ACM International Symposium on High Performance Distributed Computing*, HPDC, 2011.

[23] K. Maeda, M. Murase, M. Doi, H. Komatsu, S. Noda, and R. Himeno. Automatic resource scheduling with latency hiding for parallel stencil applications on GPGPU clusters. In *Proceedinigs of IEEE International Symposium on Parallel Distributed Processing*, IPDPS, 2012.

[24] A. M. Merritt, V. Gupta, A. Verma, A. Gavrilovska, and K. Schwan. Shadowfax: scaling in heterogeneous cluster systems via GPGPU assemblies. In *Proceedings of the 5th international workshop on Virtualization technologies in distributed computing*, VTDC, 2011.

[25] B. H. Ng, B. Lau, and A. Parkash. Direct access to graphics card leveraging VT-d. Technical report, University of Michigan, 2009.

[26] Y. Noimark and D. Cohen-Or. Streaming scenes to MPEG-4 video-enabled devices. *IEEE Computer Graphics and Applications*, 23(1):58–64, 2003.

[27] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879 –899, 2008.

[28] R. Phull, C.-H. Li, K. Rao, S. Cadambi, and S. T. Chakradhar. Interference-driven resource management for GPU-based heterogeneous clusters. In *Proceedings of the ACM International Symposium on High Performance Distributed Computing*, HPDC, 2012.

[29] V. T. Ravi, M. Becchi, G. Agrawal, and S. T. Chakradhar. Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In *Proceedings of the ACM International Symposium on High Performance Distributed Computing*, HPDC, 2011.

[30] L. Shi, H. Chen, and J. Sun. vCUDA: GPU accelerated high performance computing in virtual machines. In *Proceedinigs of IEEE International Symposium on Parallel Distributed Processing*, IPDPS, 2009.

[31] M. Stillwell, F. Vivien, and H. Casanova. Virtual machine resource allocation for service hosting on heterogeneous distributed platforms. In *Proceedinigs of IEEE International Symposium on Parallel Distributed Processing*, IPDPS, 2012.

[32] Y. Wang and A. Merchant. Proportional-share scheduling for distributed storage systems. In *Proccedings of the 5th conference on File and storage technologies*, FAST, 2007.

[33] D. D. Winter, P. Simoens, L. Deboosere, F. D. Turck, J. Moreau, B. Dhoedt, and P. Demeester. A hybrid thin-client protocol for multimedia streaming and interactive gaming applications. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV, 2006.

[34] C. Xu, S. Gamage, P. N. Rao, A. Kangarlou, R. R. Kompella, and D. Xu. vSlicer: latency-aware virtual machine scheduling via differentiated-frequency CPU slicing. In *Proceedings of the ACM International Symposium on High Performance Distributed Computing*, HPDC, 2012.

[35] P. Yu, M. Xia, Q. Lin, M. Zhu, S. Gao, Z. Qi, K. Chen, and H. Guan. Real-time enhancement for Xen hypervisor. In *Proceedings of Embedded and Ubiquitous Computing*, EUC, 2010.

[36] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel. Storage performance virtualization via throughput and latency control. *Trans. Storage*, 2:283–308, 2006.