vGASA: Adaptive Scheduling Algorithm of Virtualized GPU Resource in Cloud Gaming

Chao Zhang, Zhengwei Qi, Jianguo Yao, Miao Yu and Haibing Guan

Abstract—As the virtualization technology for GPUs matures, cloud gaming has become an emerging application among cloud services. In addition to the poor default mechanisms of GPU resource sharing, the performance of cloud games is inevitably undermined by various runtime uncertainties such as rendering complex game scenarios. The question of how to handle the runtime uncertainties for GPU resource sharing remains unanswered. To address this challenge, we propose vGASA, a virtualized GPU resource adaptive scheduling algorithm in cloud gaming. vGASA interposes scheduling algorithms in the graphics API of the operating system, and hence the host graphic driver or the guest operating system remains unmodified. In order to fulfill the Service Level Agreement as well as maximize GPU usage, we propose three adaptive scheduling algorithms featuring feedback control that mitigates the impact of the runtime uncertainties on the system performance. The experimental results demonstrate that vGASA is able to maintain frames per second of various workloads at the desired level with the performance overhead limited to 5-12%.

Index Terms—GPU, resource management, scheduling, control theory, cloud gaming

1 Introduction

Virtualization technology has significantly influenced how resources are used and managed in cloud data centers. Several virtualization solutions are widely used to construct cloud computing systems. For instance, VMware has become the industry standard in the field of commercial virtual machines (VMs). Xen [1] is the pioneer of paravirtualization technology in the field of open-source software. However, research on graphics processing unit (GPU) virtualization is still at an initial stage. It is difficult to virtualize a GPU in hypervisor mainly due to its sophisticated infrastructure design as well as the device driver.

Despite these difficulties, GPU virtualization technology has developed dramatically in the past few years. We have conducted experiments on some hypervisors to evaluate the performance of their 3D rendering and find that 3D acceleration of VMs is reaching its maturity. In the experiments, Windows 7 was used as both the guest and the host operating system (OS), and 3DMark06, a GPU performance benchmark, was used as the workload. The results for VMware player demonstrate that VMware player 4.0 is able to achieve 95.6% of the native performance while VMware Player 3.0, which was released three years ago, achieves only 52.4%. Due to the advantage of high native performance, there is an increasing number of data centers dedicated to GPU computing tasks such as cloud gaming, video rendering, and general purpose GPU (GPGPU) computing [2], [3], [4], [5], [6]. Taking cloud gaming for instance, the platform renders games remotely and streams the running result over the network so that clients can play high-end games without the support of the latest hardware.

GPU virtualization technology promotes the increasing applications of cloud games. However, GPU resource scheduling for cloud gaming in the virtualized environment is not well studied. The default GPU resource sharing in existing virtualization solutions is pretty poor as shown in Fig. 1. The frames per second (FPS) and frame latency results are given with respect to three popular games running concurrently on separate VMs. In Fig. 1a, Starcraft 2 has an average 50 FPS, while DiRT 3 has only around 25. Usually, a game with FPS above 30 can provide a smooth user experience, and, hence, DiRT 3 is a little unplayable under such circumstances. Also, as illustrated in Fig. 1b, Starcraft 2 suffers high frame latency due to the heavy resource contention among the three games: 12.78% frame latency exceeds 34 ms and 1.26% exceeds 60 ms.

One likely reason that GPU virtualization is not extensively applied in data centers, is the poor performance of the default resource scheduling mechanisms. For instance, the default GPU sharing in VMware player tends to allocate resources in a first-come, first-served manner. As a result, some VMs may not satisfy their service level agreement (SLA) requirements. Another reason is that multiple VMs are required to run on a single server; hence, they often suffer performance variations, especially in cloud gaming

- *Chao Zhang, Zhengwei Qi, Jianguo Yao and Haibing Guan are with Shanghai Jiao Tong University, Shanghai, China. E-mail: {kevin_zhang, qizhwei, jianguo.yao, hbguan}@sjtu.edu.cn*
- *Miao Yu is with Carnegie Mellon University. E-mail: superymk@cmu.edu*

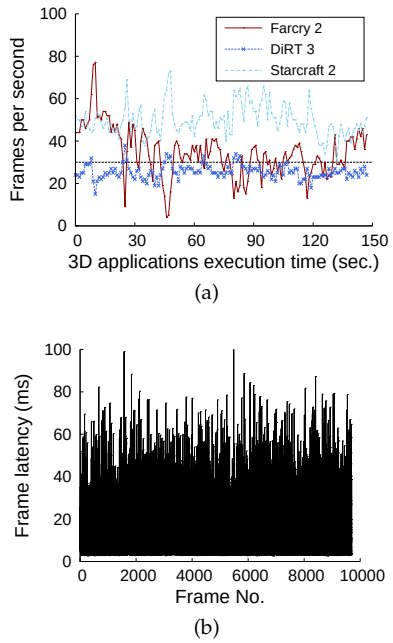


Fig. 1: Default scheduling results in poor performance under heavy contention:(a) FPS of three popular games; (b) corresponding frame latency of Starcraft 2.

platforms. In addition, the FPS rate of a game running in the virtualized environment is significantly affected by the central processing unit (CPU) time of executing the game logic, the GPU time of asynchronous rendering, unpredictable game scenario changes and interference factors from other VMs. In order to achieve desirable system performance, the current FPS of the game should be fed back and the hypervisor should react and schedule GPU resources in response to these uncertainties.

Contributions This paper proposes vGASA, an adaptive scheduling algorithm for virtualized GPU resources in cloud gaming. We discuss the choice of designing and implementing vGASA in a virtualized environment in the online supplemental file. By leveraging GPU paravirtualization technology and library interception, none of the guest applications, the guest OS, or the host graphic drivers need to be modified. To address the runtime uncertainties for GPU resource sharing, vGASA features a feedback control loop using the proportional-integral (PI) controller [7]. It is worth noting that vGASA is a lightweight scheduler in the host supporting three adaptive scheduling algorithms to achieve different goals. The three algorithms mitigate the impact of the runtime uncertainties on the system performance to ensure high resource utilization of the system. More specifically, `SLA-Aware (SA)` scheduling strives to achieve SLA requirements for each VM, which can allow plenty of users to play games on a single server. However, some users may not satisfy the basic SLA requirements when playing games. `Fair SLA-Aware (FSA)` scheduling allows gaming servers to provide a smoother user experience by maximizing the usage of GPU resources. It reallocates GPU resources from VMs with higher FPS rates to those who do not meet SLA requirements in a fair way. However, this algorithm may reduce the number of users a single gaming machine is able to serve. `Enhanced SLA-Aware (ESA)` scheduling balances the gaming performance and the number of users on a single machine. It calculates the desired runtime FPS rates for all the games while fully utilizing the GPU resources. In addition, compared with our previous work [8], vGASA applies the control theory to the scheduling algorithms, which makes itself adaptive to various runtime uncertainties.

Our experimental results show that the three scheduling algorithms have significant effect on various workloads. For example, applying `SA` scheduling to the same workload in Fig. 1, all of the games satisfy their SLA requirement of 30 FPS. The percentage of frames with excessive latency drops to 0.20%. The experiments also show that the three algorithms are adaptive in response to uncertainties of running time. Meanwhile, the GPU performance overhead incurred by vGASA is limited to 5-12%.

The rest of the paper is organized as follows. Section 2 formulates the adaptive scheduling problem and the overall architecture of vGASA. Section 3 discusses the design and implementation of the three adaptive scheduling algorithms. Section 4 presents the experimental results of the algorithms with real games and benchmark programs. Section 5 reviews related work, and Section 6 concludes the paper.

2 vGASA Architecture

2.1 Closed-loop Scheduling Problem and Approach Overview

Most games are designed in an infinite loop, which is described in detail in the supplemental file. Basically, both the game scene rendering and the CPU computation determine the FPS rate of a game. One of our objectives is to control the FPS rates of games, and hence the GPU resources can be scheduled. However, real-world games such as Starcraft 2, in fact, seem not always to run at the same FPS rate during the process of gaming. The FPS rate may continuously vary with the change of the game scenes.

In this paper, we exploit library interposition to insert a `Sleep` function in the loop of a GPU computation model. Thus, we can control the FPS rates of games by setting a proper sleep time for each VM. Given a desired FPS F, the sleep time can be calculated by:

$$T_{sleep} = \frac{1000}{F} - T_{cpu} - T_{gpu}, \quad (1)$$

where $\frac{1000}{F}$ is the time, in milliseconds, of each iteration when FPS F is wanted, and T_{cpu} and T_{gpu} is the

time of game logic and GPU processing, respectively. After the sleep time, T_{sleep} is calculated, the `Sleep` function is invoked to make the VM sleep for T_{sleep}. The actual output FPS is generated and can be detected accurately.

However, the output FPS may not always be the same as what is desired because there are several uncertain factors that affect the time of a frame, or, in other words, the frame latency. Hence, the FPS rate is not easily maintained in practice. First, CPU computation time changes irregularly at run time. For instance, switching to a game scene that contains more objects (e.g., enemies, trees, buildings, etc.) than the last scene will suddenly result in a great increase of CPU computing. In addition, CPU-intensive processes running in the background in the guest OS will have an unpredictable influence on the game's CPU computing. Second, the graphics API calls are invoked in an asynchronous manner. Thus, the time of game rendering cannot be simply estimated. Fortunately, we can predict the time, but the accuracy must be taken into consideration. Moreover, the time of rendering is greatly affected by the complexity of the game scene. If there is a sudden change of a game scene, the time of GPU computation will vary dramatically. These uncertainties may cause large deviation and lead to poor scheduling performance, as discussed in prior work [8].

Therefore, the detection results must be fed back to change the sleep time of that VM to a proper value whenever the output FPS has changed undesirably. The relationship between the sleep time (T_{sleep}) and the output FPS (F_{out}) can be given by definition. Specifically, FPS is the number of frames per second, which is the inverse of the time of a frame. Besides, the time of each frame is calculated by (1). We can then derive that the FPS rate has an inverse relation with the sleep time. For generality, we write the equation in the form of power function with the exponent set to a negative value:

$$F_{out} = mT_{sleep}^{-n}, \tag{2}$$

where m and n are two positive constants. The correlation result of the supplemental evaluation corresponds with this equation, which is described in more detail in the supplemental evaluation.

2.2 System Architecture

vGASA is designed and implemented within the GPU virtualization framework [9], as shown in Fig. 2. Designing the framework in the virtualized environment has several advantages over designing it in the non-virtualized environment. Please refer to the supplemental file for more about the GPU virtualization framework and the advantages. Modules introduced by vGASA are highlighted in grey. Based on library interposition (see the supplemental file), vGASA is

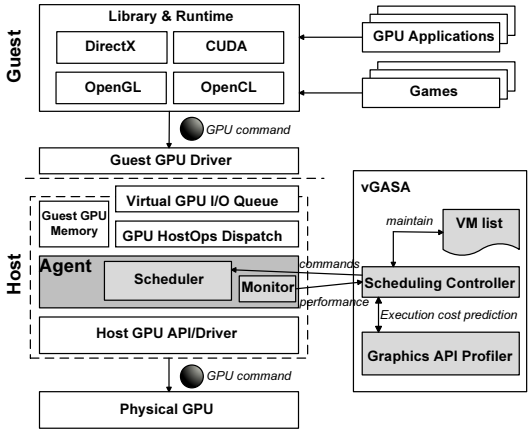


Fig. 2: vGASA architecture based on the technology of paravirtualization.

a lightweight scheduler between GPU HostOps Dispatch and Host GPU API.

vGASA is composed of a *scheduling controller*, *monitor*, *scheduler*, *graphics API profiler,* and *VM list*. The scheduling controller receives performance feedback from all the running VMs and sends commands to trigger the control system to work. There is also one *agent* consisting of the scheduler and monitor module for each VM. The monitor sends information about the real-time performance of the VM on which it resides to the scheduling controller. The scheduler receives commands from the controller and schedules GPU computation tasks according to different scheduling algorithms. The graphics API profiler predicts the execution costs of GPU commands issued by the graphics API. The VM list contains the current running VMs. Each VM is indexed by vGASA. When a new VM is launched, vGASA can automatically add it to the VM list and reschedule GPU resources.

How the scheduler chooses the scheduling algorithms depends on the system's goals as mentioned in Section 1. To address the closed-loop scheduling problem discussed in Section 2.1, the control theory is involved in the algorithms. The detailed design and implementation of the these algorithms are introduced in Section 3.2.

3 Adaptive Scheduling Algorithms

In this section, we first discuss the control law for the problem mentioned in Section 2.1 based on feedback control theory. Then we design and implement three algorithms incorporated in vGASA. These algorithms address different requirements for different GPU computing applications.

3.1 Control Law

The feedback control system is shown in Fig. 3. $G_c(z)$, $G_p(z)$, and $H(z)$ represent the transfer function of the controller, vGASA, and the feedback of the present

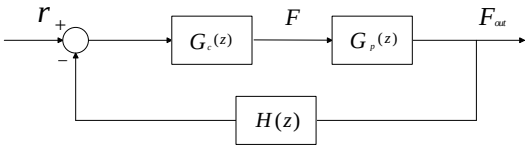


Fig. 3: The closed-loop system for vGASA, where $G_c(z), G_p(z), andH(z)$ represent the transfer functions of the PI controller, the vGASA framework and the feedback of the FPS rate, respectively.

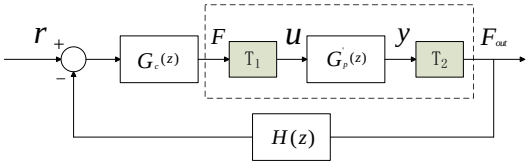


Fig. 4: The transformed closed-loop system, where $G_c(z), G'_p(z), andH(z)$ represent the transfer function of the PI controller, the transformed system and the feedback of the FPS rate, respectively. T_1 and T_2 stand for the transformation functions.

FPS rate, respectively. The objective of this paper is to ensure the output FPS, F_{out}, of vGASA is equal to r, as shown in Fig. 3. Therefore, we can define the problem in Section 2.1 as a typical discrete-time control problem, where r is the *reference*, F is the *control input*, and F_{out} is the *controlled variable*.

At the end of each GPU computation loop, the system measurement is just the system output F_{out}. Hence, the measurement transfer function is $H(z) = 1$. We adopt a PI controller which is sufficient to achieve the stability and convergence of the closed-loop system [7]. Hence, we can get the transfer function of the PI controller for the discrete system as follows:

$$G_c(z) = k_p + k_i \frac{z}{z-1}, \tag{3}$$

where k_p is the proportional gain and k_i is the integral gain. They should be chosen properly to ensure the stability and convergence of the system. The next step is to get the transfer function ($G_p(z)$) of vGASA. According to (1) and (2), F_{out} is actually an inverse function of the detection threshold F. By replacing T_{sleep} with (1), we can derive:

$$F_{out} = m(\frac{1000}{F} - T_{cpu} - T_{gpu})^{-n}. \tag{4}$$

Because it is difficult to directly take z-transform [7] to (4) to get $G_p(z)$, we transform 4 by adopting natural logarithms to both sides of the equation. Then we derive:

$$\ln\left(\left(\frac{F_{out}}{m}\right)^{-\frac{1}{n}} + T_{cpu} + T_{gpu}\right) = \ln 1000 - \ln F. \tag{5}$$

After letting $y = \ln((\frac{F_{out}}{m})^{-\frac{1}{n}} + T_{cpu} + T_{gpu})$ and $u = -\ln F$, we get $y = \ln 1000 + u$, of which we can get the transfer function $G'_p(z)$ as $G'_p(z) = 1$. Until

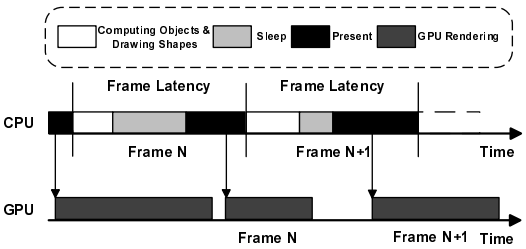


Fig. 5: Frame Latency controlled by the three algorithms.

now, we get a variant of $G_P(z)$ which is $G'_p(z)$ as the transfer function of vGASA because it is hard to directly take z-transform to (4) to get $G_p(z)$. After the transformation, a new closed-loop system is achieved, where u is the new *control input* and y is the new *controlled variable*. In order to analyze the stability and convergence of the origin system, we first should discuss the stability and convergence of the transformed system. The block diagram shown in Fig. 4 is derived from Fig. 3 by transforming F and F_{out}.

Then we analyze the stability of the system. The closed-loop transfer function of Fig. 4, denoted by $T_c(z)$, is given by

$$\begin{aligned} T_c(z) =& \frac{G_c(z)G'_p(z)}{1 + G_c(z)G'_p(z)H(z)} \\ =& \frac{(k_p + k_i)z - k_p}{(k_p + k_i + 1)z - (k_p + 1)}. \end{aligned} \tag{6}$$

From the formula, we can get the pole of the closed-loop discrete system at

$$z = \frac{k_p + 1}{k_p + k_i + 1}. \tag{7}$$

From control theory [7], if the pole is within the unit circle centered at the origin, i.e.,

$$\left|\frac{k_p + 1}{k_p + k_i + 1}\right| < 1, \tag{8}$$

then, the closed-loop discrete system is stable.

Note that when the integral parameter k_i is 0, the closed-loop transfer function (6) can be rewritten as $T_c(z) = \frac{k_p}{k_p+1}$. Hence, the closed-loop system with proportional control is always stable without any poles.

vGASA supports three scheduling algorithms, the `SA`, `FSA` and `ESA`. The control theory is applied to all the three scheduling algorithms to make them adaptive to runtime uncertainties.

3.2 Scheduling Algorithms

Since the main objective of vGASA is to control the FPS rate, the FPS must be converged to the reference of the control system. As illustrated in Fig. 5, vGASA controls each frame by delaying its last call, `Present`. This is achieved via inserting a `Sleep` call before

`Present`. The amount of delay is given by (1) in which the FPS rate, CPU, and GPU computation time are given by vGASA. The CPU computation time can be precisely measured by vGASA. However, as for the GPU computation time, it is predicted instead of measured because of the asynchronous manner of GPU processing. We can predict the GPU time because the change between the two adjacent frames occurs across a gradient. The next frame is rendered on the basis of the last frame except that the whole game scenario changes. Therefore, we can use the historical execution time of `Present` to predict the GPU time of the upcoming frame. Besides, we also observe that the prediction can be more accurate with a `Flush` call just before `Present`, which is discussed in the supplemental material.

Below is the detailed design and implementation of the three adaptive scheduling algorithms. In summary, `SA` calculates proper sleep time to make all the games just meet their SLA requirements so that one server can accommodate as many games as possible. `FSA` and `ESA` are designed based on `SA` and allow games to provide a much smoother user experience. Compared with `FSA`, `EA` makes a tradeoff between accommodating more games and providing a smoother gaming experience. In common, the three algorithms all insert a `Sleep` call and employ the PI controller to control the FPS rates of games. What is different is the reference computation of the control system. `SA` sets the reference to the criteria, which is 30 in most cases, while the other two set the value according to the runtime status.

Algorithm 1 The SA scheduling algorithm.

```
1: set r to criteria
2: while TRUE do
3:    ComputeObjectsInFrame()
4:    DrawShapes(VGA_Buffer)
5:    get feedback, F_out, from last iteration
6:    d = r − F_out
7:    F = PICalAndControl(d)
8:    Flush()
9:    sleep_time = CalcSleepTime(F)
10:   Sleep(sleep_time)
11:   Present()
12: end while
```

SA Scheduling: The `SA` scheduling is designed to make all the running games satisfy their SLA requirements. In most cases, a rate of 30 FPS is enough for a game to supply a smooth user experience. Algorithm 1 shows the pseudocode of the algorithm. In the algorithm, r is the reference of the control system. F_{out} is the FPS feedback from the last iteration of the game loop, and d is the deviation of r and F_{out}. The algorithm first statically sets the reference of the control system to the criteria. Then, in each iteration, it uses the PI controller to control the FPS to the reference. Finally, the sleep time is calculated according to both the reference and equation 1, and the game is made to sleep before it invokes the frame rendering call.

Algorithm 2 The FSA scheduling algorithm.

```
1: for each VM_i in VM_list do
2:    if FPS[VM_i] below criteria then
3:       gpu_res_to_get = CalcGPUResource()
4:       vm_above_30_list = scanVMList()
5:       n = Len(vm_above_30_list)
6:       gpu_reduce = gpu_res_to_get / n
7:       for each VM_j in vm_above_30_list do
8:          desired_FPS = calcFPS(gpu_reduce)
9:          set r_j to desired_FPS
10:      end for
11:   end if
12: end for
13: Schedule()
```

FSA Scheduling: The `FSA` scheduling policy releases GPU resources of those VMs with high FPS rates and reallocates them to those with rates lower than the criteria FPS. In the scheduling, the VMs with high FPS rates share the equal part of the GPU resources to be reallocated to the VMs with low FPS rates. As a result, the VM with the most GPU resources still keeps the most after scheduling. Thus, under the `FSA` scheduling policy, the way of reallocating GPU resources is fair, and the total amount of GPU usage suffers virtually no loss, which can achieve higher GPU usage and supply higher gaming performance than `SA` can.

Algorithm 2 demonstrates the pseudocode of the algorithm. In our current implementation, all the information kept by a VM is stored as a global value so that VMs can simply share status. As a consequence, any operations on these global value should be mutex-exclusive. We just omit the lock operation to make the pseudocode simple and intuitive. In the scheduling, the scheduling controller introduced in Section 2.2 collects FPS and GPU usage from all the VMs. Once it detects a VM denoted as VM_a that is running below the criteria FPS, it will find those VMs with FPS above the criteria and calculate the average GPU resource they will release so as to increase the FPS of VM_a to the criteria. The amount of the GPU resources that each VM will release is calculated by the proportional relation of GPU usage and FPS in `CalcGPUResource`. Additionally, the proportional coefficient of each game is calculated by the history FPS and GPU usage. Chances are that more than one VM is detected with FPS rates below the criteria. If such case occurs, the amount of the GPU resources to be released is accumulated. However, `FSA` scheduling will never make a VM release too many resources to

run at a FPS rate below the criteria.

After the amount is achieved, the desired FPS is then calculated in `calcFPS` where the procedure is just the reverse order of the GPU usage calculation in `CalcGPUResource`. Finally, the scheduling controller sets the reference of each VM to the calculated FPS rate and invokes `Schedule` to activate the control system on all the VMs. Then all the games go through an iteration, which is the same as they do in Algorithm 1.

Algorithm 3 The ESA scheduling algorithm.

```
1: sum = 0
2: for each VM_i in VM_list do
3:     k_i = CalcCoef()
4:     sum += k_i
5: end for
6: desired_FPS = 1 / sum
7: for each VM_i in VM_list do
8:     set r to desired_FPS
9: end for
10: Schedule()
```

ESA Scheduling: The `ESA` scheduling policy allows all the VMs to run at the same FPS rate while maximizing the GPU usage. Instead of the criteria FPS being statically set in `SA`, the FPS rate at which all the VMs run in `ESA` is dynamically determined at runtime. The value is also calculated by the proportional relation of the GPU usage and FPS rate. As shown in Algorithm 3, the scheduling controller scans the VM list and calculates the proportional coefficient for each VM as well as the sum of the coefficients. Then, the desired FPS rate at which the VMs should run is derived simply by $\frac{1}{\sum_{i=1}^{n} k_i}$ where k_i is the coefficient for VM_i. This is because the sum of all the VMs' GPU usage is 100% theoretically. More particularly, for each VM, suppose we have $g_i = k_i * f_i$ where g_i denotes the GPU usage and f_i denotes the FPS. Since `ESA` makes all the VMs run at the same FPS, we suppose the value is F. Hence we get $g_i' = k_i * F$, in which g_i' is the GPU usage that is adjusted to achieve the scheduling goal while the coefficients remain the same. Because $\sum_{i=1}^{n} g_i' = 100\%$, we can get $F = \frac{1}{\sum_{i=1}^{n} k_i}$. Finally, once the desired FPS is calculated, the scheduling controller sets the reference of each VM and invokes `Schedule`, just as it does in `FSA`.

4 Evaluation

We now provide a detailed quantitative evaluation of vGASA on the testbed configured with an i7-2600k 3.4GHz CPU and 16GB RAM. AMD HD6750 with the default frequency and 2GB of video memory are used as the graphics card on the testbed. Both the host OS and the guest OS are Windows 7 x64. The version of VMware player is 4.0. Each hosted VM is configured with dual cores and 2GB RAM. The screen resolution is set to 1280x720, a high graphic quality, for all workloads and benchmarks. To simplify performance comparison, swap space and GPU-accelerated windowing system are disabled on the host side.

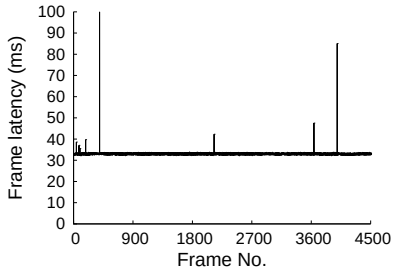


Fig. 6: Frame latency of Starcraft 2 after SLA-Aware scheduling.

Two different types of workload are used in the following experiments. The first workload group, named *Ideal Model Games* can maintain a stable FPS because it has a stable game scene with almost fixed objects and views. PostProcess, ShadowVolume and LocalDeformablePRT from DirectX 9.0 SDK samples are chosen as the representatives of this group. The other group of the workload is the *Reality Model Games* in which the FPS keeps constant for a short period of time but varies during gaming. We pick DiRT 3, Starcraft 2, and Farcry 2 as the workloads of this group used in the experiments.

First, we evaluate the performance of the three scheduling policies in case of under-provision of GPU resources to show that all the algorithms can achieve design goals. Next, we evaluate the effectiveness of the control system, demonstrating the feature as well as the control process of the system. PostProcess is used to produce various profiling with different control parameters. Finally, we conduct micro- and macro-analysis to evaluate the performance impact of vGASA onto the guest legacy softwares. Other evaluation results are included in the online supplement.

4.1 Effective Scheduling of vGASA

In this experiment, the workloads used are DiRT 3, Farcry 2 and Starcraft 2. The SLA requirement is set to 30 FPS, which means all workloads are supposed to run at an FPS rate larger than 30 after being scheduled by the three algorithms.

Basic SLA-Aware Fig. 6 shows the frame latency of Starcraft 2 after using the basic `SA` scheduling. Compared with Fig. 1b, the game runs much faster when scheduled. The percentage of excessive frame latency drops to 0.20% and the latencies of only two frames are greater than 60 ms. Fig. 7a shows the performance of the `SA` scheduling when three workloads are concurrently sharing the GPU. The criterion for FPS is set at 30. Compared with Fig 1a, all the

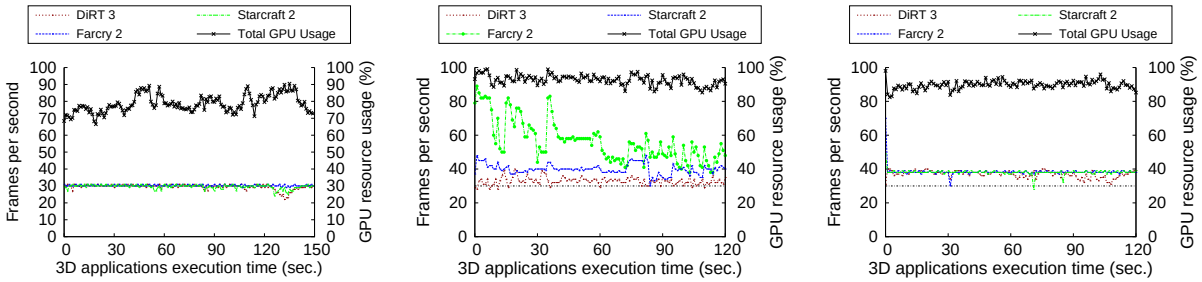


Fig. 7: Results of the three scheduling: (a) Basic SLA-Aware; (b) Fair SLA-Aware; (c) Enhanced SLA-Aware

workloads meet their SLA requirements. The average FPS rate of DiRT 3, Starcraft 2 and Farcry 2 is 30.1, 30.5, and 30.3, respectively, under the SA scheduling policy. Besides, GPU usage typically ranges from 70-80% and seldom exceeds 90%. The average GPU usage is 77.8%. An extra game can concurrently run with these three workloads to improve GPU utilization.

Fair SLA-Aware Fig. 7b presents the results from the same setup as that used in the SA policy experiments. At the very beginning, Farcry 2 has the highest FPS rate while DiRT 3 has a FPS rate lower than 30 because of GPU contention. Then the FSA scheduling policy starts to work. At 2 sec, vGASA detects that the FPS rate of DiRT 3 is about 28 and those of the other two workloads are both above 30. It releases the GPU resources from Starcraft 2 and Farcry 2 and reallocates them to DiRT 3 so that at 3 sec, the FPS rate of DiRT 3 increases to 33 and those of the other two workloads decrease slightly. During scheduling, Farcry 2 still has the top FPS rate and the FPS rates of both Farcry 2 and Starcraft do not decrease lower than the criteria. From the figure, we can also see that the FPS rate varies during gaming. However, when the FPS rate of a workload drops below the criteria (e.g., at 13, 18, and 32 sec), vGASA can allocate more GPU resources to it in the next loop so that the SLA requirement is satisfied. As for GPU usage, the maximal is 99.1% while the minimal is 85.2%. The average is 92.7%. Although there is still a little waste of GPU resources, FSA can provide a better gaming performance than SA when the number of games is same.

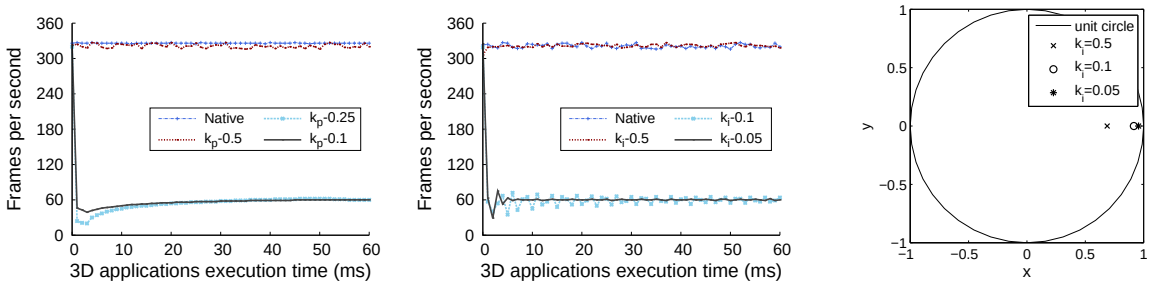
Enhanced SLA-Aware Next we evaluate the performance of the ESA scheduling policy. Also, 30 is set as the FPS criterion. Fig. 7c shows the ESA scheduling results. Compared with Fig. 7a, the policy schedules all the workloads so that they run at around 38 FPS instead of 30, creating a 26.7% improvement in performance. During gaming, when there are uncertain factors influencing the FPS rate such as at 33 and 73 sec, the FPS drops dramatically. But in the next loop, the FPS rate is able to increase to its desired level, benefiting from the PI controller. Similar to the FSA experiment, DiRT 3 gets extra GPU resources to increase its FPS rate with a decrease in the FPS rates of the other two workloads. From the figure, we can see that the FPS rate of Starcraft 2 and Farcry 2 are stably controlled while the FPS rate of DiRT 3 sometimes is not maintained at the desired level. For most of the time, DiRT 3 only has a close FPS, especially after 90 sec. The average GPU usage is about 90.0%, representing a 15.7% improvement compared with the 77.8% GPU usage of the basic SA scheduling policy.

4.2 Impact of Control Parameters

In this experiment, the effects of the adaptive algorithms are evaluated with different control parameters and the reference being set to 60 FPS. Since we only use the PI controller, we just evaluate the k_p and k_i control parameters for proportional gain and integral gain respectively. Actually, there is many sets of k_p and k_i values that can make the control system stable. In the experiment, we choose some of the values to show their effect on the scheduling framework and then determine the proper value of k_p and k_i used in the experiments throughout this section. As for the scheduling policy, SA is involved in the evaluation. The other two policies give similar performance.

Fig. 8a shows the control effect with different values of k_p while $k_i = 0$. PostProcess is used and the native FPS rate without scheduling keeps at 320 stably. When k_p equals 0.5, the scheduling has no effect. K_p with values of 0.25 and 0.1 can make the control variable converge to the reference. The scheduling can get a faster convergence when $k_p = 0.1$. Besides, the system gets steady after the 37th ms when k_p is 0.1. After that, the average FPS of the workload is 61.3. Compared with the reference, there is a small error by 1.3 FPS, which is defined as the steady-state error. Such an error may occur when there is only a proportional controller in the system.

Fig. 8b shows the system performance when giving various k_i with $k_p = 0.1$. When k_i equals 0.5, the scheduling also has no effect. The system with $k_i = 0.05$ gives a better stability and convergence than does $k_i = 0.1$. From the figure, we can see that the system gets steady after the 9th ms when k_i is 0.05. The average FPS rate after that is 60.2. The steady-state error is decreased or even eliminated when



(a) (b) (c)

Fig. 8: Impacts of control parameters: (a) Impact with different k_p and $k_i = 0$; (b) Impact with different k_i and $k_p = 0.1$; (c) The poles of discrete closed-loop control system with various k_i and $k_p = 0.1$.

TABLE 1: Microbenchmark result of SLA-Aware scheduling.

	Ideal Model Games	Reality Model Games
GPU Command Flush (ms)	4.842	2.265
Present() Execution (ms)	0.117	0.707
Scheduling Tasks (ms)	0.124	0.178
GPU Usage Measurement (ms)	0.014	0.177

an `integral` controller is involved in the control system. The poles of the discrete closed-loop control system with various k_i and $k_p = 0.1$ are 0.6875, 0.9167, and 0.9565, shown in Fig. 8c, in which the poles within the unit circle indicate that the closed-loop systems are stable.

Fig. 8a and 8b also demonstrate the control process within dozens of milliseconds, during which the FPS gradually converges to the reference. Therefore, though there are frames with sudden changes (e.g., the frame of DiRT 3 at 13 sec in Fig. 7b), which cause the overall FPS rate of DiRT 3 to decrease, vGASA is able to immediately control the FPS rate in the next second reacting to such runtime uncertainties because the control procedure can be accomplished within only one second.

4.3 Overhead Discussion

In order to quantify the performance overhead imposed by vGASA, we only measure the cost time incurred by the framework itself. This cost time constitutes time of GPU command flush, `Present` execution, scheduling tasks, and GPU usage measurement. Table 1 shows the microbenchmark results of the `SA` scheduling. Based on the design of `SA`, `FSA` and `ESA` have a similar performance, except that they spend a little more time on scheduling tasks because the latter two algorithms need to compute the reference or desired FPS rate, in other words. The GPU command flush operations contribute the main performance overhead. This is due to the design of the current Direct3D library and the implemented flush strategy in our vGASA prototype, as discussed in the online supplemental file. It is possible to achieve a better performance by adopting a different flush strategy in the future. As for the library interception, it virtually incurs no overhead. In our analysis, benefiting from leveraging `hook` to intercept Direct3D APIs in the implementation, once the hooked function is invoked, the Windows kernel directly redirects the invocation to vGASA, which has the same time as directly invoking the corresponding Direct3D APIs.

Fig. 9 gives the average FPS rate of three workloads running concurrently under different scheduling policies. We remove the `Sleep` function so as to just evaluate the overhead caused by our scheduling framework itself. Assigning `SA` incurs about 5% performance overhead for the workloads due to the extra calculation time of T_{sleep}. `FSA` and `ESA` have a little more overhead. `FSA` incurs the most overhead because it needs to manipulate some global variables, which causes extra time in lock operation. But overall, the three scheduling provided by vGASA incur slight performance overhead, with the worst about 12%. Moreover, the overhead is warranted when multiple games run concurrently because their SLA requirements are all satisfied under vGASA, compared with those under OnLive, a commercial cloud games provider that can only run a single game per graphics card.

5 Related Work

This section mainly discusses the related work from the perspective of GPU resource sharing, GPU virtualization for cloud gaming and general GPU computing, and feedback control. Some of them are introduced in the supplemental file.

The scheduling problem of GPU resources is a hot topic. Previous GPU resource scheduling approaches mainly target native systems. For example, TimeGraph [10] 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

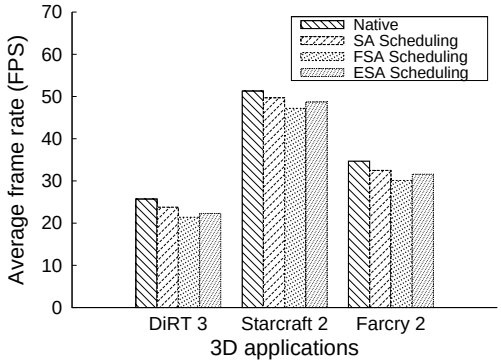


Fig. 9: Performance overhead with `Sleep` function disabled.

them according to the predefined settings as well as GPU hardware measurements. TimeGraph cannot guarantee SLA for all the VMs, especially for less important workloads. Instead, our `FSA` and `ESA` algorithms are used to effectively provide both SLA and maximize the GPU resource usage. Fumihiko *et al.* [11] propose a Fine Grained Cycle Sharing (FGCS) system on GPUs for accelerating sequence homology search in local area network environments. Compared with it, our framework balances the GPU resources, instead of improving performance of the VMs on which a game is running.

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 [12], [13], [14]. OnLive, one of the most popular cloud gaming service providers, supplies at least 60 FPS at 1280*720 resolution for high-end games. With the same SLA as OnLive, our approach is able to run multiple game VMs that efficiently share GPU resources.

In general-purpose GPU computing, hiCUDA [6] proposes high level GPGPU programming. vCUDA [3] introduces GPU computing into a virtualization execution environment. rCUDA [4] and Duato's work [15] try to decrease the power-consuming GPUs from high-performance clusters while preserving their 3D-acceleration capability to remote nodes. Gupta *et al.* [16] propose Pegasus using NVIDIA GPGPUs coupled with x86-based general purpose host cores to manage combined platform resources. Based on Pegasus, Merritt *et al.* [17] propose Shadowfax, a prototype of GPGPU Assemblies that improves GPGPU application scalability as well as increases application throughput. Zhang *et al.* [5] propose a framework that can automatically generate 3D stencil code with optimal parameters for heterogeneous GPUs. Lawrence [18] makes use of the commodity GPU for common tasks of numerically integrating ODEs. In comparison, our approach tries to improve the SLA of GPU computation on a cloud gaming platform and maximize the overall resource usage. Additionally, vGASA provides three representative scheduling algorithms to meet multiple optimization goals in the case of under- and over-provisioned GPU resources.

Feedback control has been widely adopted to improve the adaptability of systems. It is employed to improve application adaption of VMs [19], to scale web applications in cloud services [20], and to dynamically allocate resources in virtualized servers [21]. Park *et al.* [22] use feedback control to predicate high-performance computing. An adaptive resource control system [23] is developed to dynamically adjust resource shares to meet application-level QoS goals in virtual data centers. Wang *et al.* [24] propose using feedback control to adaptively control power for chip multiprocessors. Different from these works, we develop a control-theoretical scheduling framework for cloud gaming, which fulfills the SLA requirements of games as well as maximize GPU resource usage.

6 Conclusion

We presented vGASA, an adaptive virtualized GPU resource scheduling algorithm for cloud gaming. By introducing an agent per VM and a centralized scheduling controller to the paravirtualization framework, vGASA achieves in-VM GPU resource measurements and regulates the GPU resource usage. Moreover, we propose three representative scheduling algorithms: `SA` scheduling allocates just enough GPU resources to fulfill the SLA requirement; `FSA` scheduling allocates maximum GPU resources to all running VMs in a fair way while still guaranteeing their SLA requirements; and under `ESA` scheduling, all the VMs run at the same or a close FPS rate 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 with performance overhead limited to 5-10%. We plan to extend vGASA to multiple physical GPUs and multiple physical machine systems for data center resource scheduling in our future work.

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] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, "GViM: GPU-accelerated virtual machines," in *Proceedings of the Workshop on System-level Virtualization for High Performance Computing (HPCVirt)*, 2009.

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

[4] 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.

[5] Y. Zhang and F. Mueller, "Autogeneration and autotuning of 3D stencil codes on homogeneous and heterogeneous GPU clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 3, pp. 417–427, 2013.

[6] T. D. Han and T. S. Abdelrahman, "hiCUDA: High-level GPGPU programming," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 78–90, 2011.

[7] K. Ogata, *Discrete-time control systems*. Prentice-Hall, 1995.

[8] M. Yu, C. Zhang, Z. Qi, J. Yao, and H. Guan, "VGRIS: Virtualized GPU resource isolation and scheduling in cloud gaming," in *Proceedings of the ACM International Symposium on High Performance Distributed Computing (HPDC)*, 2013.

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

[10] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "Timegraph: GPU scheduling for real-time multi-tasking environments," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2011.

[11] F. Ino, Y. Munekawa, and K. Hagihara, "Sequence homology search using fine grained cycle sharing of idle GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 4, pp. 751–759, 2012.

[12] 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 Annual International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 2006.

[13] L. Deboosere, J. D. Wachter, P. Simoens, F. D. Turck, B. Dhoedt, and P. Demeester, "Thin client computing solutions in low- and high-motion scenarios," in *Proceedings of International Conference on Networking and Services (ICNS)*, 2007.

[14] 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.

[15] 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*, ser. Euro-Par Workshops, 2009.

[16] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan, "Pegasus: Coordinated scheduling for virtualized accelerator-based systems," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2011.

[17] 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 International Workshop on Virtualization Technologies in Distributed Computing (VTDC)*, 2011.

[18] L. Murray, "GPU acceleration of runge-kutta integrators," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 1, pp. 94–101, 2012.

[19] Y. Zhang, A. Bestavros, M. Guirguis, I. Matta, and R. West, "Friendly virtual machines: Leveraging a feedback-control model for application adaptation," in *Proceedings of the USENIX International Conference on Virtual Execution Environments (VEE)*, 2004.

[20] A. Ashraf, B. Byholm, J. Lehtinen, and I. Porres, "Feedback control algorithms to deploy and scale multiple web applications per virtual machine," in *Proceedings of the EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, 2012.

[21] W. Zhang, J. Liu, Y. Song, M. Zhu, L. Xiao, Y. Sun, and L. Ruan, "Dynamic resource allocation based on user experience in virtualized servers," *Procedia Engineering*, vol. 15, no. 0, pp. 3780 – 3784, 2011.

[22] S.-M. Park and M. A. Humphrey, "Predictable high-performance computing using feedback control and admission control," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 3, pp. 396–411, 2011.

[23] P. Padala, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, K. Salem, and K. G. Shin, "Adaptive control of virtualized resources in utility computing environments," in *Proceedings of the EuroSys*, 2007.

[24] X. Wang, K. Ma, and Y. Wang, "Adaptive power control with online model estimation for chip multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 10, pp. 1681–1696, 2011.

Chao Zhang received the B.E degree from Fudan University, Shanghai, PR China in 2007. Currently, he is a graduate student at Shanghai Key Laboratory of Scalable Computing and Systems, School of Software, Shanghai Jiao Tong University. His research interests mainly include GPU virtualization, feedback control applications and concurrent programming.

Zhengwei Qi received his B.Eng. and M.Eng degrees from Northwestern Polytechnical University, in 1999 and 2002, and Ph.D. degree from Shanghai Jiao Tong University in 2005. He is an Associate Professor at the School of Software, Shanghai Jiao Tong University. His research interests include program analysis, model checking, virtual machines, and distributed systems.

Jianguo Yao (M'12) received the B.E and the M.E degree from the Northwestern Polytechnical University (NPU), Xian, Shaanxi, PR China, respectively in 2000 and 2007. He received the Ph.D. degree from the NPU in 2010. He was a Joint Postdoc Fellow at the Ecole Polytechnique de Montreal and the McGill University in 2011. Currently, he is an Assistant Professor at the Shanghai Jiao Tong University. His research interests are feedback control applications, real-time and embedded computing, power management of data centres and cyber-physical systems.

Miao Yu graduated from Shanghai Jiao Tong University in 2009 and 2012, for his B.Eng degree and M.Eng degree. Currently he is a PhD student advised by Dr. Virgil Gligor in CyLab, Carnegie Mellon University. His primary research interest lies in the system security area and virtualization area.

Haibing Guan received Ph.D. degree from Tongji University in 1999. He is a Professor of School of Electronic, Information and Electronic Engineering, Shanghai Jiao Tong University, and the director of the Shanghai Key Laboratory of Scalable Computing and Systems. His research interests include distributed computing, network security, network storage, green IT and cloud computing.