Hindawi
Scientific Programming
Volume 2019, Article ID 5240956, 11 pages
https://doi.org/10.1155/2019/5240956

Research Article

Ballooning Graphics Memory Space in Full GPU Virtualization Environments

Younghun Park, Minwoo Gu, and Sungyong Park

Department of Computer Science and Engineering, Sogang University, Seoul 04107, Republic of Korea

Correspondence should be addressed to Sungyong Park; parksy@sogang.ac.kr

Received 1 February 2019; Accepted 28 March 2019; Published 23 April 2019

Guest Editor: Tarek Abdelrahman

Advances in virtualization technology have enabled multiple virtual machines (VMs) to share resources in a physical machine (PM). With the widespread use of graphics-intensive applications, such as two-dimensional (2D) or 3D rendering, many graphics processing unit (GPU) virtualization solutions have been proposed to provide high-performance GPU services in a virtualized environment. Although elasticity is one of the major benefits in this environment, the allocation of GPU memory is still static in the sense that after the GPU memory is allocated to a VM, it is not possible to change the memory size at runtime. This causes underutilization of GPU memory or performance degradation of a GPU application due to the lack of GPU memory when an application requires a large amount of GPU memory. In this paper, we propose a GPU memory ballooning solution called *gBalloon* that dynamically adjusts the GPU memory size at runtime according to the GPU memory requirement of each VM and the GPU memory sharing overhead. The *gBalloon* extends the GPU memory size of a VM by detecting performance degradation due to the lack of GPU memory. The *gBalloon* also reduces the GPU memory size when the overcommitted or underutilized GPU memory of a VM creates additional overhead for the GPU context switch or the CPU load due to GPU memory sharing among the VMs. We implemented the *gBalloon* by modifying the *gVirt*, a full GPU virtualization solution for Intel's integrated GPUs. Benchmarking results show that the *gBalloon* dynamically adjusts the GPU memory size at runtime, which improves the performance by up to 8% against the *gVirt* with 384 MB of high global graphics memory and 32% against the *gVirt* with 1024 MB of high global graphics memory.

1. Introduction

Running graphics-intensive applications that include three-dimensional (3D) visualization and rendering in a virtualized environment creates a new challenge for high-performance graphics processing unit (GPU) virtualization solutions. GPU virtualization is a technique that allows multiple virtual machines (VMs) to share a physical GPU and run high-performance graphics applications with performance guarantees.

Among a wide range of GPU virtualization techniques, application programming interface (API) remoting [1–11] is a method of intercepting API calls and passing them to the host. This method is easy to implement, but requires modification every time the version of the API library or GPU driver changes. This method also cannot provide all GPU functions. Direct pass-through [12, 13] allocates a GPU exclusively to a single VM and allows it to directly use the GPU with no intervention by the hypervisor. This method provides a high performance that is similar to the native environment, but GPU sharing among VMs is impossible. Currently, AWS [14] and Azure [15] provide GPU services to VMs through the direct pass-through method or through the NVIDIA GRID [16] which is a GPU virtualization solution at the hardware level.

To solve the problems of the approaches mentioned above, GPU virtualization solutions at the hypervisor level, such as *gVirt* [17], *GPUvm* [18], and VMCG [19], have been proposed. The *gVirt* is a full GPU virtualization technology for Intel's integrated GPUs. Unlike dedicated or discrete GPUs in which dedicated graphic cards have independent graphics memory, integrated GPUs share a portion of the

system RAM for graphics memory (or GPU memory). The original *gVirt* divides the graphics memory into chunks and allocates them exclusively to each VM. As a result, a single host could create up to only a maximum of three VMs. The *gScale* [20, 21] solved this scalability problem by dividing the graphics memory into multiple small slots and letting VMs share the slots. Private and physical graphics translation tables (GTTs) are used to translate the virtual address generated by each VM into the physical address for the graphics memory. When a VM is scheduled to run, the private GTT entries of the corresponding VM are copied to the physical GTT. Every time entries are updated in the physical GTT, the modified contents are synchronized with the private GTT that each VM has.

However, the GPU memory size, which is set during the initial creation of the VM, cannot be changed dynamically. This causes the following problems. First, a VM must be restarted to change the allocated GPU memory size. The user must restart the VM to execute the GPU application that requires GPU memory larger than the current setting. As a result, the VM stops, and the service interruption is expectable. Second, the VM that occupies GPU memory larger than necessary can degrade the performance of other CPU and GPU applications running on other VMs. Whenever a GPU context switch occurs, the CPU copies the private GTT entries to a physical GTT, which also takes up the CPU time of other VMs. If the memory utilization is low, this unnecessary copying overhead may degrade the performance of other VMs. In addition, the GPU context switch time increases as the GPU memory size of each VM gets larger [22]. Third, as we reported in a previous study [22], small GPU memory size affects the performance of GPU workload, especially when VMs run with graphics operations for rendering or high-resolution display environments.

Although several studies [23–29] dynamically adjusted the memory allocation of existing VMs, in these studies, memory was taken from a specific VM and allocated to another VM when the physical memory was insufficient. As these approaches assume an environment in which each VM has an independent virtual address space, it is difficult to apply those techniques directly to the full GPU virtualization environment in which the same virtual GPU memory space is shared.

In this paper, we propose a dynamic GPU memory ballooning scheme called *gBalloon* which dynamically increases or decreases the GPU memory size allocated to each VM at runtime. The *gBalloon* detects performance degradation due to the lack of GPU memory and increases the GPU memory size allocated to each VM. In addition, the GPU memory size of each VM can be reduced when the overcommitted or underutilized GPU memory of a VM creates additional overhead for the GPU context switch or the CPU load due to GPU memory sharing among the VMs. We implement the *gBalloon* using the 2016Q4 version of *gVirt*. As the *gScale*'s GPU memory-sharing technique is also implemented, the *gBalloon* can scale up to 15 Linux VMs. Using various CPU and GPU benchmarks, we also show that the *gBalloon* dynamically adjusts the GPU memory size at runtime and outperforms the *gVirt* (modified *gVirt* in which the *gScale*'s features are added) by up to 8% against the *gVirt* with 384 MB of high global graphics memory and 32% against the *gVirt* with 1024 MB of high global graphics memory.

Although current *gBalloon* is mainly targeted at Intel's integrated GPU, its design principle can be applied to other architectures as well. For example, the proposed idea can be easily applied to other integrated GPUs from AMD and Samsung, where the system memory is used as GPU memory. In addition, we believe that other discrete GPUs with dedicated memory such as NVIDIA can benefit from the *gBalloon* since they also use graphics translation table for address translation. However, special care has to be taken to reduce the memory copy overhead across the system bus if the *gBalloon* is implemented over discrete GPUs. As the *gVirt* is open source and the access to the source codes for the NVIDIA driver and runtime is limited, we decided to use the *gVirt* as a software platform to verify our proposed idea.

The rest of the paper is organized as follows. In Section 2, we outline the structure of *gVirt* and present the motivations behind the design of *gBalloon*. In Section 3, we explain the design and implementation issues of *gBalloon*. In Section 4, we evaluate the performance of *gBalloon* and compare it with that of *gVirt* using various configurations. In Section 5, we discuss related works, and in Section 6, we conclude with suggestions for future work.

2. Background and Motivation

In this section, we provide an overview of the *gVirt* and discuss the motivations for the proposed approach.

2.1. Overview of gVirt. The *gVirt* is a high-performance GPU virtualization solution that provides mediated pass-through capability [17]. The *gVirt* allows VMs to directly access resources that have a large effect on performance and make other privileged operations be intervened through a hypervisor. Due to the restriction on the number of simultaneous VMs in the *gVirt*, we modified the original *gVirt* over Xen hypervisor (XenGT) and added the *gScale*'s scalability features [20]. Throughout this paper, we consider the modified *gVirt* as *gVirt*.

In the *gVirt*, the mediator located in Dom0 emulates the virtual GPU (vGPU) for each VM. The mediator schedules the vGPU in a round-robin fashion for fair scheduling among the VMs. Considering the high cost of the GPU context switch, each vGPU is switched at approximately 16 ms interval, which is a speed at which people cannot recognize an image change [17]. When the time quantum allocated to each vGPU is fully used, the mediator saves the registers and memory information of the corresponding vGPU and restores the GPU state for the next vGPU. Because the *gVirt* does not support preemption (although NVIDIA starts to provide preemption capability at the hardware level from Pascal architecture, the feature is not exposed to user's control at the time of this writing) at the time of this writing, it traces the submitted GPU commands and waits for the results until each vGPU can finish its jobs.

For example, if a vGPU executes GPU kernels longer than 16 ms, it runs without preemption. To prevent each vGPU from overusing the time quantum, the *gVirt* places a limit on the number of GPU kernels a vGPU is allowed to run within the time quantum.

Intel's global graphics memory is divided into two parts as shown in Figure 1: low global graphics memory and high global graphics memory. Only the GPU can access high global graphics memory, but the CPU and the GPU can access low global graphics memory. The CPU can access low global graphics memory through the *aperture* mapped to the GPU memory. Thus, the amount of low global graphics memory that can be accessed depends on the *aperture* size. The maximum *aperture* size currently supported by the motherboard is 512 kB, which is mapped up to 512 MB of the low global graphics memory.

Figure 1 shows the memory mapping and management structure between global graphics memory and system memory. In the *gVirt* (modified *gVirt* in which the *gScale*'s features are added), part of the low global graphics memory is shared by all vGPUs, and the high global graphics memory is divided into 64 MB slots that can also be shared among the vGPUs. The virtual address of the global graphics memory is converted into a physical address through the physical GTT. Each vGPU has a private GTT, which is a copy of the physical GTT corresponding to the allocated low global graphics memory and high global graphics memory. The private GTT entries of the vGPU are synchronized every time each vGPU modifies the physical GTT entries. To activate the vGPU in a GPU context switch, if the entry does not exist in the physical GTT, the state is restored by copying the private GTT to the physical GTT. However, if the vGPU is scheduled out, the CPU cannot access the vGPU through *aperture*. To solve this problem, the *gScale* allows the CPU to access the vGPU space at all times through the *fence memory space pool*, which ensures the proper operation of *ladder mapping* for mapping the guest physical address to the host physical address [20].

The *gVirt* framework focuses on the acceleration of graphics-intensive applications such as 2D or 3D rendering over a virtualized environment, rather than general-purpose computing on graphics processing units (GPGPU) computing over clouds.

2.2. Motivation. In current *gVirt*, 64 MB low global graphics memory and 384 MB high global graphics memory are recommended for Linux VM [20] because those memory sizes are enough to support most GPU workloads without performance degradation and crash from the experiments. However, we showed in a previous study [22] that large high global graphics memory can sometimes increase the performance of GPU workloads. We also observed in the study that large high global graphics memory can increase the possibility of overlapping address spaces, which may incur large overhead in a GPU context switch and thus degrade the performance of other VMs. Furthermore, in an environment where the VMs require large global graphics memory, it is highly likely that their GTT entries do not exist in the

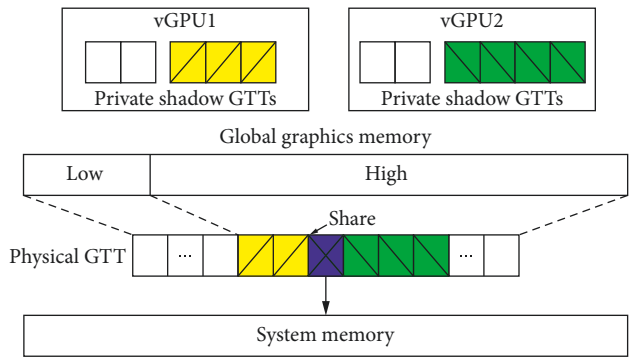

Figure 1: Global graphics memory structure of *gVirt*.

physical GTT in a GPU context switch because the GPU memory space is shared among the VMs. If a large amount of GPU memory is allocated although it is not fully utilized, unnecessary copies of the GTT entries can occur in a GPU context switch. This increases the time for the GPU context switch, which also decreases the time for each vGPU to occupy the GPU per unit time. As a result, not only the performance of the GPU applications running on all VMs is degraded but also the time for the CPU to copy the GTT entries increases. Therefore, the performance of the CPU applications running on the VMs can be degraded as well.

To confirm this, we conducted two experiments to investigate the effects of copying GTT entries on the performance of GPU application due to excessive occupation of high global graphics memory. For the two experiments, 384 MB and 1024 MB are used for the high global graphics memory size. It is possible to use other configurations as long as it is bigger than 384 MB and smaller than the size of physical GPU memory. Furthermore, the size should be multiple of slot size. Figure 2 shows the sum of the frames per second (FPS) for each VM by executing *Nexuiz* 3D benchmarks from Phoronix Test Suite [30] as we increase the number of VMs from 3 to 15. As shown in Figure 2, when the size of the high global graphics memory is small (384 MB), the VMs start to share the GPU memory from the point when the number of VMs reaches around 10. Then, the performance of the VMs degrades as we increase the number of VMs to 12 or 15. However, when the size of the high global graphics memory is large (1024 MB), the VMs start to share GPU memory from the relatively small number of VMs. This means that the copying of the GTT entries in the GPU context switch causes a very large performance degradation. When the number of VMs is 6 or 9, the performance at 1024 MB degraded by approximately 3.5 times compared with the performance at 384 MB.

Overall, the performance of the VMs is highly affected by the size of the GPU memory allocated to each VM, and the memory size must be adjusted at runtime to optimize the performance.

3. Design of *gBalloon*

In this section, we describe the design and implementation of the *gBalloon* that adjusts the GPU memory size of VMs at

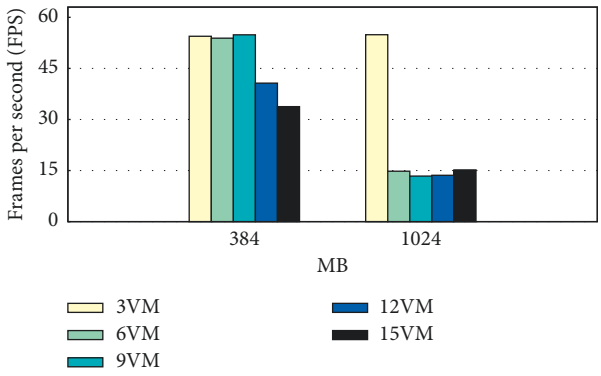

Figure 2: Performance degradation due to GPU memory sharing.

runtime. As we identified in the previous section, the performance of a GPU application can be degraded due to the static allocation of GPU memory. The *gBalloon* monitors the lack of GPU memory in VMs and then allocates the required amount of GPU memory to the corresponding VM. Furthermore, the *gBalloon* also checks the performance of CPU and GPU applications and decreases the GPU memory size when the performance of each VM degrades due to the GPU memory sharing among the VMs. The *gBalloon* is implemented by modifying the *gVirt* 2016Q4 release [31]. In the following, we present the GPU memory expansion and reduction strategies implemented in the *gBalloon* in detail.

3.1. GPU Memory Expansion Strategy. When a VM is created, the GPU driver of the VM obtains the available range of GPU memory from the GPU driver in Dom0 and balloons the requested memory area excluding the space that has been allocated to the VM. Then, the GPU driver of the VM searches for memory space excluding the ballooned area when allocating a new memory object. If the space for allocating an object is insufficient, the GPU driver creates an empty space by returning the existing memory objects. As a result, the performance of a GPU application that requires graphics-intensive operations, such as rendering operations, is degraded because the same objects are frequently returned. To reduce this overhead, the *gBalloon* detects the VMs' lack of GPU memory by tracing the number of memory object returns at runtime and reduces the ballooned area of other VMs for the required amount of memory space so that the VM with the lack of memory can use additional GPU memory.

Figure 3 shows the process in which the *gBalloon* allocates additional GPU memory to the guest. When a guest GPU driver must return an existing object due to the lack of GPU memory, the following four steps take place. Step 1: the guest requests additional GPU memory space from the host. Step 2: to expand the GPU memory space with the requested size, the host chooses the optimal strategy that can minimize the GPU memory-sharing overhead based on the GPU memory adjustment algorithm which will be explained later. Step 3: based on the results of the GPU memory adjustment algorithm, the size of the guest's private GTT is increased. Step 4: finally, the guest receives information about the GPU memory expansion from the host and shrinks the existing ballooned area so that it can be used to allocate objects. For example, as shown in Figure 3, the high global graphics memory area of vGPU1 is expanded to the right, and the shared memory areas of vGPU1 and vGPU3 are increased accordingly.

3.2. GPU Memory Reduction Strategy. As the GPU memory is expanded by the GPU memory expansion request of a VM, the size of the GPU memory shared among the VMs can also be increased. Consequently, the probability that the entry will not exist in the physical GTT during the GPU context switch is increased, resulting in more GTT entry copies. This degrades the performance of the GPU application. In addition, as the number of entries to be copied is also increased, the CPU consumes more time copying the GTT entries, thus degrading the performance of the CPU application.

The *gBalloon* monitors the CPU cycles consumed for copying GTT entries during the GPU context switch to check the performance degradation of the GPU application. Profiling is performed at every t cycles. When N is the number of CPUs in the host and C is the CPU cycles consumed for copying GTT entries, the rate of time R_{copy} consumed by the CPU to copy the GTT entries for unit time t can be expressed as follows:

$$R_{\text{copy}} = \frac{C}{t \times N} \times 100\,(\%). \tag{1}$$

Because one CPU processes the copying of the GTT entries, a total of N CPUs consume the cycles for the unit time. Thus, the number of CPUs must be reflected in R_{copy}.

To check the performance degradation of the CPU application due to the competition among the vCPUs, the *gBalloon* uses the steal time. The steal time is the time when the vCPU of a VM exists in the ready queue. Assume that w_{ij} is the steal time of the vCPU_j of VM_i and s_{ij} is the time when the vCPU_j of VM_i exists in other queues. Then, the rate W of the steal times in all VMs can be expressed as follows:

$$W = \sum_i \sum_j \frac{w_{ij}}{w_{ij} + s_{ij}}\,(\%). \tag{2}$$

A large value of W means that there is severe competition among the VMs. Therefore, the state of a physical machine (PM) can be defined by the values of W and R_{copy}. If both W and R_{copy} are large, then the performance of the CPU application is being degraded by the copying of the GTT entries. In this case, the host must prevent the performance degradation of the CPU and GPU applications by rejecting the GPU memory expansion requests of the VMs and reducing the GPU memory sharing among the VMs. Whereas, if W is large, but R_{copy} is small, although there is severe competition among the VMs, it is not caused by the copying of the GTT entries. In this case, the performance of the CPU application could be degraded due to the increase in R_{copy}. Thus, the GTT size should be reduced if possible. However, if W is small, but R_{copy} is large, there is no competition among

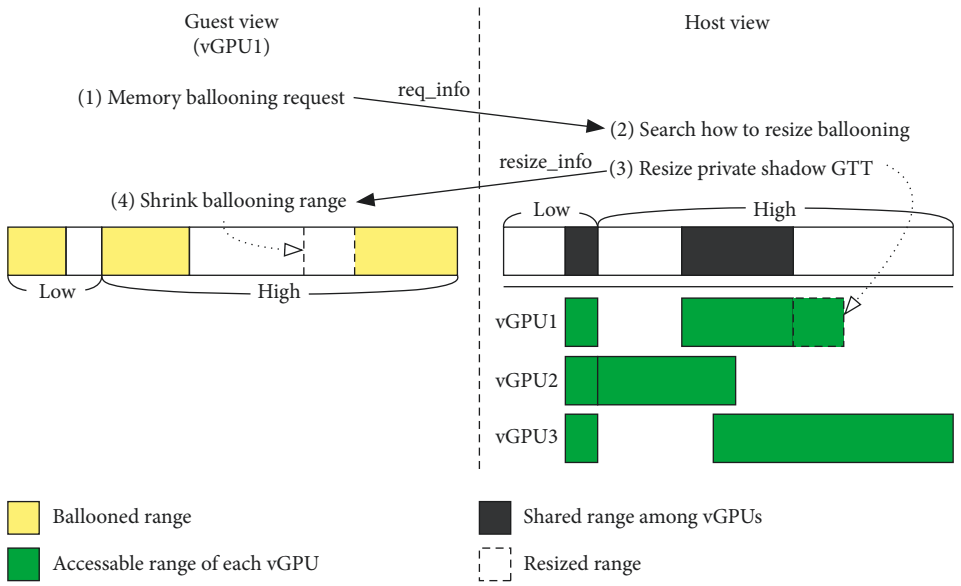

Figure 3: GPU memory expansion process.

the VMs, but many copies of the GTT entries are occurring. In this case, if the VMs perform CPU applications, then the competition among the vCPUs can become more severe due to the copies of the GTT entries, and thus, the performance of the CPU application can degrade. Therefore, the host should try to reduce the sharing of GPU memory as much as possible. Finally, if both W and R_{copy} are small, there is no overhead in the current PM, and the host does not need to take any action.

Based on the implications described above, the *gBalloon* identifies the degree of overhead in the host using the values of W and R_{copy}. $f(W, R_{copy})$, which represents the degree of overhead incurred due to the copying of the GTT entries, can be expressed as follows:

$$f\left(W, R_{copy}\right) = \alpha \times W + \beta \times R_{copy}, \quad (3)$$

where α and β are the reciprocals of maximum W and R_{copy}. Those parameters normalize the overall value by giving the same weight to W and R_{copy}. The maximum W and R_{copy} are dependent upon a particular hardware platform and are generally determined through experiments.

The *gBalloon* calculates $f(W, R_{copy})$ every 0.5 seconds and classifies the value with two thresholds Threshold_{low} and Threshold_{high}. Based on the classification, the *gBalloon* applies different GPU memory policies. The threshold values are experimentally determined from 0 to 2 as the value of $f(W, R_{copy})$ ranges from 0 to 2. For the experiments, we use 0.5 (25% of the maximum value) and 1 (50% of the maximum value) for Threshold_{low} and Threshold_{high}, respectively. If $f(W, R_{copy})$ is smaller than Threshold_{low}, the *gBalloon* approves all requests for GPU memory expansion and does not reduce the GPU memory of the VMs. If $f(W, R_{copy})$ is larger than Threshold_{low}, the *gBalloon* starts to decrease the size of the GPU memory allocated to each VM to reduce the overhead of the GPU memory sharing among the VMs. Instead of reducing the GPU memory of all VMs, the *gBalloon* reduces the GPU memory of the VM that occupies a large amount of GPU memory and has the lowest GPU memory utilization. If $f(W, R_{copy})$ is larger than Threshold_{high}, the *gBalloon* rejects all requests for GPU memory expansion from the VMs and reduces a space corresponding to two 64 MB slots regardless of the GPU memory usage of each VM. The method for reducing GPU memory is to minimize the GPU memory sharing by using Algorithm 1 described in the next subsection.

3.3. GPU Memory Adjustment Algorithm. The GPU memory is adjusted to minimize the GPU memory sharing with the existing VMs. When the *gBalloon* decides the number of GPU memory slots and the target vGPU to adjust based on the GPU memory expansion and reduction strategies discussed earlier, the spaces at both sides of the GPU memory space allocated to VMs are increased or decreased.

For example, let us assume that vGPU4 initiated a GPU memory expansion request for two slots when there are four vGPUs from vGPU1 to vGPU4 that require two, two, two, and one slots, respectively. Also assume that there are five slots, and the GPU memory size of the host is 320 MB as shown in Figure 4. In this case, there are three possible methods for expanding the two slots as requested by vGPU4: (1) expanding two slots to the left side, (2) expanding one slot each to the left and right sides, and (3) expanding two slots to the right side. In the first case, vGPU4 shares slots 1 and 2 with vGPU1, resulting in two shared slots in total. In the second case, vGPU4 shares slot 2 with vGPU1 and slot 4 with vGPU2 and vGPU3, resulting in three shared slots in total. In the third case, vGPU4 shares slot 4 with vGPU2 and vGPU3 and slot 5 with vGPU3, resulting in three shared slots in total. Therefore, a strategy of expanding two slots to the left minimizes the number of shared slots. The *gBalloon* measures the number of shared slots and expands the GPU memory allocated to the VM to minimize the sharing overhead among the VMs.

Require:
M: the total number of slots in high global graphics memory
N: the number of vGPUs
$V = \{V_0, V_1, \ldots, V_{N-1}\}$: a set of vGPUs
$S = \{S_0, S_1, \ldots, S_{N-1}\}$: a set of the number of occupied slots by vGPUs
$P = \{P_0, P_1, \ldots, P_{N-1}\}$: a set of start slot index of vGPUs
Q: the number of requested slots to expand/reduce
K: vGPU number that initiated memory expansion request
Ensure:
L, R: the number of slots to expand/reduce to the left side and the right side

procedure *Expansion*
(1) **if** $S_K + Q > M$ **then return** ERROR
(2) $minimum \longleftarrow \infty$
(3) **for** $i \longleftarrow Q\text{-min}(Q, M - P_K - S_K)$ **to** $\min(Q, P_K)$ **step** 1 **do**
(4) $count \longleftarrow 0$
(5) **for** $j \longleftarrow 0$ **to** $N - 1$ **step** 1 **do**
(6) $Lcount \longleftarrow$ the number of occupied slots from $P_K - i$ to $P_K - 1$ by V_j
(7) $Rcount \longleftarrow$ the number of occupied slots from $P_K + S_K$ to $P_K + S_K + Q - i - 1$ by V_j
(8) $count \longleftarrow count + Lcount + Rcount$
(9) **end for**
(10) **if** $minimum > count$ **then**
(11) $minimum \longleftarrow count;$
(12) $L \longleftarrow i$
(13) **end if**
(14) **end for**
(15) $R \longleftarrow Q - L$
(16) **return** L, R
end procedure

procedure *Reduction*
(1) **if** $S_K < Q$ **then return** ERROR
(2) $maximum \longleftarrow -1$
(3) **for** $i \longleftarrow 0$ **to** Q **step** 1 **do**
(4) $count \longleftarrow 0$
(5) **for** $j \longleftarrow 0$ **to** $N - 1$ **step** 1 **do**
(6) **if** $j != K$ **then**
(7) $Lcount \longleftarrow$ the number of occupied slots from P_K to $P_K + i - 1$ by V_j
(8) $Rcount \longleftarrow$ the number of occupied slots from $P_K + S_K + i - Q$ to $P_K + S_K - 1$ by V_j
(9) $count \longleftarrow count + Lcount + Rcount$
(10) **end if**
(11) **end for**
(12) **if** $maximum < count$ **then**
(13) $maximum \longleftarrow count$
(14) $L \longleftarrow i$
(15) **end if**
(16) **end for**
(17) $R \longleftarrow Q - L$
(18) **return** L, R
end procedure

Algorithm 1: GPU memory adjustment algorithm.

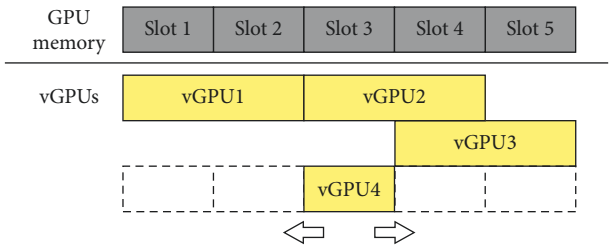

Figure 4: Example of GPU memory expansion.

The method for reducing GPU memory is the same as that for expanding it. In Figure 4, when the GPU memory of vGPU3 should be reduced by one slot, reducing slot 4 rather than slot 5 can minimize the GPU memory sharing among the VMs. Thus, the policy for minimizing the GPU memory sharing can maximize the effect of the predictive-copy technique [21] that copies the GTT entries in advance by predicting the next scheduled vGPU before the GPU context switch. The detailed algorithms for memory expansion and reduction are presented below.

4. Performance Evaluation

In this section, we compare the performance of the *gBalloon* with that of the *gVirt* using various workloads. Table 1 shows the experimental environment for the performance evaluation. The global graphics memory size of the host is set to 4 GB, which consists of 256 MB of low global graphics memory and 3840 MB of high global graphics memory. Dom0 does not share the global graphics memory with other domains, but other guest VMs share 64 MB of the low global graphics memory and 3456 MB of the high global graphics memory excluding the Dom0 area. The low global graphics memory size of every guest VM is set to 64 MB as recommended in [20], whereas the high global graphics memory size is set differently depending on the experiments.

The experiments use four 3D benchmarks and four 2D benchmarks. To measure the 3D performance, *Lightsmark*, *Openarena*, *Nexuiz*, and *Urbanterror* of Phoronix Test Suite [30] and Unigine Valley (*valley*) [32] that requires many rendering operations are used. To measure the 2D performance, Firefox-asteroids (*Firefox-ast*), Firefox-scrolling (*Firefox-scr*), gnome-system-monitor (*gnome*), and *Midori* of Cairo-perf-trace [33] are used. The performance of the 3D benchmarks is measured by the average number of FPS, and the performance of the 2D benchmarks is measured by the execution time. Furthermore, the NAS Parallel Benchmark (NPB) [34] is used to measure the overhead for the CPU due to the GPU context switch.

4.1. Performance Comparison Using a Single GPU Application. In this subsection, the performance of the *gVirt* and the *gBalloon* is compared when *valley* that requires many rendering operations is executed by multiple VMs. For the *gVirt*, the *gVirt*-384 (a *gVirt* version with the high global graphics memory set to 384 MB) and the *gVirt*-1024 (a *gVirt* version with the high global graphics memory set to 1024 MB) are used for the performance comparison. For the 2D and 3D benchmarks that do not demand many rendering operations, additional GPU memory is not allocated because they require a small amount of high global graphics memory. Therefore, *valley* is used to compare the performance of the dynamic GPU memory expansion policy of the *gBalloon* with that of the *gVirt*. To observe the performance variations due to the increase in the number of VMs and the change in the degree of GPU memory sharing, experiments were performed in which the number of VMs was increased by three.

Figure 5 depicts the performance of the *gVirt*-384, the *gVirt*-1024, and the *gBalloon* normalized to the performance of the *gVirt*-384 by the sum of the FPS values of all VMs when there are 3, 6, 9, 12, or 15 VMs. When the number of VMs is six or fewer, the *gVirt*-1024 shows better performance than the *gVirt*-384 because the overhead from the GPU memory sharing is small. The *gBalloon* also shows a similar performance as the *gVirt*-1024 because the *gBalloon* allocates the required amount of GPU memory to VMs. In particular, as the number of VMs is increased, the performance of the *gVirt*-1024 and the *gBalloon* becomes more similar because the effect of overhead on the performance degradation becomes small, and the effect of performance degradation due to the GPU memory sharing becomes large. For this reason, all implementations show similar performance when the number of VMs is nine or higher.

4.2. Performance Comparison Using Multiple GPU Applications. In this subsection, the performance of *gVirt*-384, *gVirt*-1024, and *gBalloon* is compared by running various types of GPU applications on 15 VMs. As shown in Table 2, 15 VMs run randomly selected 2D and 3D benchmarks. Because various benchmarks are mixed, it is possible to compare the degree of the performance degradation of the GPU applications due to the GPU memory-sharing overhead that may occur as the requirements for the GPU memory change.

Figure 6 shows the performance comparison when the randomly selected 2D and 3D benchmarks shown in Table 2 are executed by 15 VMs. All performance values are normalized to that of the *gVirt*-384. Strangely, *valley* in the *gVirt*-1024 shows a better performance than the *gVirt*-384 although the overhead is large due to the GPU memory sharing. However, the performance of the other 3D benchmarks is decreased by 50% or higher and the performance of the 2D benchmarks by 25%. Thus, the performance of the total VMs drops by 24%, on average, compared with that of the *gVirt*-384. In the case of the *gBalloon*, the performance of *valley* is guaranteed because the GPU memory size of each VM expands as the required amount of GPU memory increases. Moreover, the *gBalloon* minimizes the overhead due to the GPU memory sharing by dynamically adjusting the GPU memory size according to the GPU memory usage. As a result, the GPU context switch time is decreased, and the performance of *valley* is increased by up to 28%. The performance of the other benchmarks is similar to that of the *gVirt*-384. Figure 7 shows a summary of the performance in all benchmarks. The performance of the *gBalloon* is higher by 8% than that of the *gVirt*-384 and higher by 32% than that of the *gVirt*-1024.

4.3. Performance Comparison Using CPU and GPU Applications. In this subsection, the effect of performance degradation in CPU applications caused by the copying of the private GTT entries is analyzed. Among the 15 VMs, 7 VMs run with CPU workloads, whereas the remaining 8 VMs run with GPU workloads. The CPU workload uses *cg*

Table 1: Evaluation environment.

Host physical machine	
Processor	Intel core i7-6700 3.40 GHz (8 cores)/ Intel HD Graphics 530
Memory	48 GB
Disk	Samsung SSD 850 PRO 256 GB ∗ 3
Host virtual machine (Dom0)	
vCPU	8
Memory	4096 MB
Hypervisor	Xen version 4.6.0
OS	Ubuntu 16.04.1 (kernel version 4.3.0)
Low/high global graphics memory	64 MB/384 MB
Guest virtual machine	
vCPU	2
Memory	2560 MB
OS	Ubuntu 16.04 (kernel version 4.3.0)
Low global graphics memory	64 MB

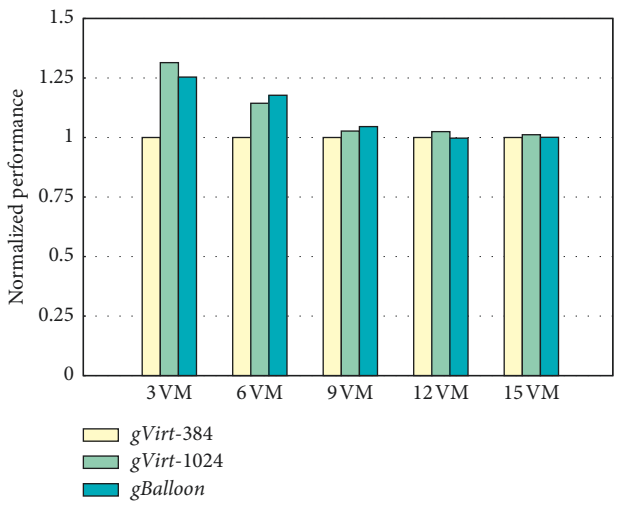

Figure 5: Performance of a single GPU benchmark.

Table 2: The benchmark sequence that each VM performs.

	Benchmark 1	Benchmark 2	Benchmark 3
VM1	*Gnome*	*Openarena*	*Valley*
VM2	*Valley*	*Firefox-ast*	*Urbanterror*
VM3	*Midori*	*Valley*	*Nexuiz*
VM4	*Valley*	*Firefox-scr*	*Urbanterror*
VM5	*Valley*	*Firefox-scr*	*Lightsmark*
VM6	*Lightsmark*	*Gnome*	*Valley*
VM7	*Gnome*	*Valley*	*Openarena*
VM8	*Urbanterror*	*Valley*	*Firefox-scr*
VM9	*Midori*	*Nexuiz*	*Valley*
VM10	*Valley*	*Lightsmark*	*Firefox-scr*
VM11	*Gnome*	*Openarena*	*Valley*
VM12	*Nexuiz*	*Firefox-ast*	*Valley*
VM13	*Valley*	*Openarena*	*Firefox-ast*
VM14	*Midori*	*Valley*	*Lightsmark*
VM15	*Valley*	*Nexuiz*	*Firefox-ast*

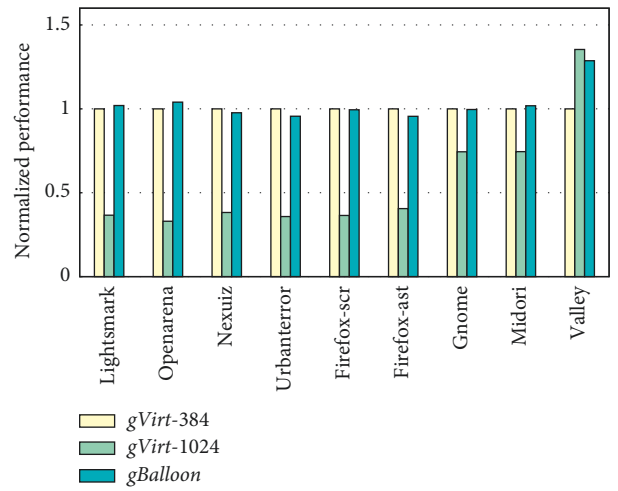

Figure 6: Performance of each GPU benchmark.

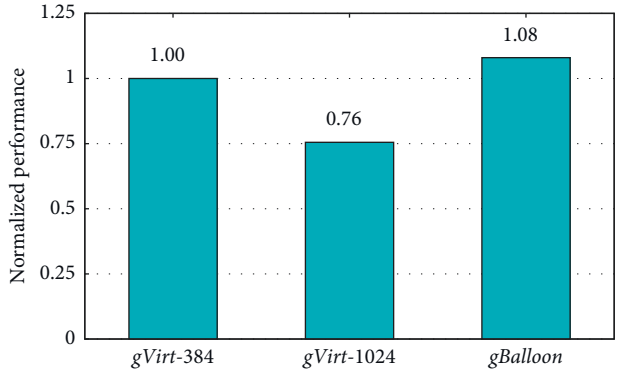

Figure 7: Performance summary of all benchmarks.

of the NPB benchmark, which is a workload to find the smallest eigenvalue of the matrix using the conjugate gradient method.

Figure 8 shows the performance of the CPU and GPU benchmarks normalized to that of the *gVirt*-384. In the case of the *gVirt*-1024, the performance of *cg* is decreased by 19% compared with that of the *gVirt*-384. This is because the CPU consumes a great deal of time copying the private GTT entries due to the large amount of GPU memory shared among the VMs. Furthermore, the performance of *valley* is increased slightly by approximately 4% due to this overhead. In contrast, the *gBalloon* limits the increase in the GPU memory size of the VMs by detecting the overhead of the CPU and the overhead due to the copying of the GTT entries. As a result, the performance of *cg* is decreased by approximately 1.5%, and the performance of *valley* is increased by approximately 2% compared with that of *gVirt*-384.

4.4. Overhead and Sensitivity Analysis. In this subsection, the performance of the *gBalloon* and the *gVirt* in a single VM environment is compared. High global graphics memory sizes of 384 MB and 1024 MB are set for the VMs of the *gBalloon* and the *gVirt*, respectively. Figure 9 shows the

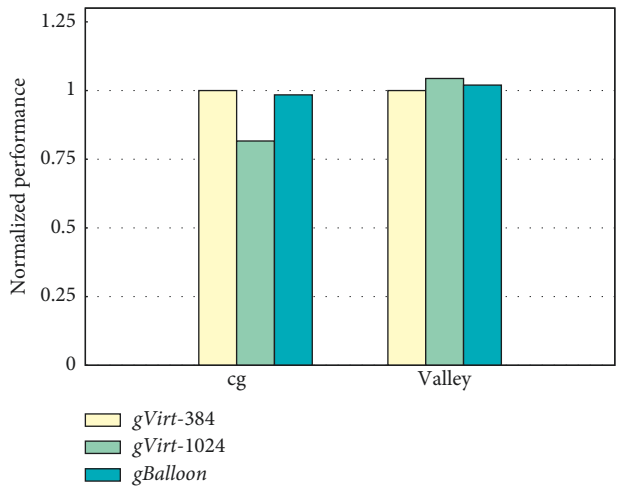

Figure 8: Performance when CPU and GPU benchmarks are performed at the same time.

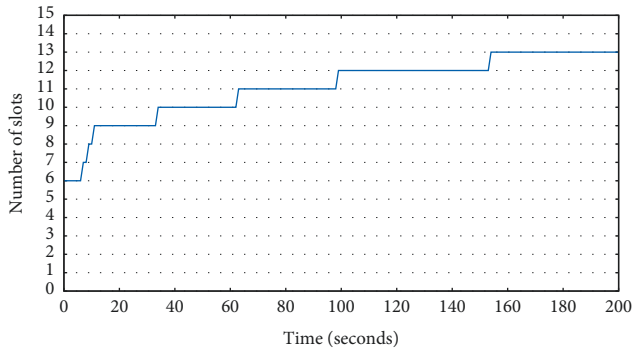

Figure 9: Changes in GPU memory size allocated to VMs when GPU benchmarking requiring a lot of GPU memory is executed.

adaptive behavior in the GPU memory slots of the VMs over time when *valley* is performed with the dynamic GPU memory expansion policy of the *gBalloon*. *Valley* is composed of 18 scenes in total, and the amount of GPU memory required is different for each scene. When *valley* is executed first, the required amount of GPU memory is increased sharply, and the *gBalloon* expands three slots in 2 second intervals. Then, one slot is expanded for several scenes. The number of slots is increased up to 13, and the size of the high global graphics memory of the VMs is increased to 832 MB.

Figure 10 shows the performance of *gBalloon* for each scene and the overall performance, which is normalized to that of the *gVirt*. As shown in Figure 10, the overall performance of the *gBalloon* is lower by approximately 1.9% than that of the *gVirt*. This is because the slots are increased one by one, resulting in performance degradation due to the temporary lack of GPU memory despite the sharp increase in the amount of GPU memory required in the first scene. However, this performance degradation is negligible. From the 10th scene when the number of slots becomes 12, the performance degradation disappears due to the frequent expansion requests and the lack of GPU memory. Thus, the FPS values of the *gVirt* and the *gBalloon* are similar.

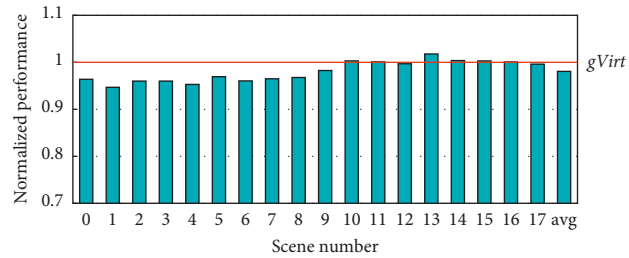

Figure 10: Performance comparison of VMs when GPU benchmarking requiring a lot of GPU memory is executed.

5. Related Works

Kato et al. [35], Wang et al. [36], Ji et al. [37], and Becchi et al. [38] proposed technologies for solving the problem of insufficient GPU memory when compute unified device architecture (CUDA) is performed in the NVIDIA GPU environment. When the amount of GPU memory is insufficient, the data in the GPU memory are moved to the system memory to secure space in the GPU memory, which is allocated to the applications. However, this copy operation has large overhead when performed at runtime, and the user must use a modified API.

Kehne et al. [39] and Kehne et al. [40] proposed a swap policy for reducing the overhead at runtime and improving the resource fairness among GPU applications and the utilization of GPU memory. GPUswap divides the buffer into chunks of a fixed size, randomly selects the chunk of an application that occupies the largest amount of GPU memory, and moves the chunk to the system memory when the GPU memory is insufficient. GPUswap randomly selects chunks from applications that occupy the largest amount of memory. However, because the chunk to be removed from the system memory is randomly selected, the performance of the corresponding applications may be degraded if highly reusable data are removed from the GPU memory. To reduce this overhead, GPrioSwap determines the priorities of the chunks by profiling the GPU memory access counts of the GPU applications and moves the chunk with the lowest priority when the GPU memory is insufficient.

Studies have also been conducted to prevent program crashes when the GPU is shared between containers. Kang et al. [41] proposed a solution that proposes the amount of GPU memory that can be allocated to each container. When a container asks to use more than the limited GPU memory size, ConVGPU rejects the request. In contrast, when the GPU memory is insufficient, ConVGPU lets the container wait until GPU memory becomes available even if the requested amount of memory is less than the limited memory size. However, these studies were targeted at discrete GPUs whose data are transferred through PCIe bus and cannot be directly applied to the heterogeneous system architecture. In this architecture, the system memory is used as the GPU memory, and the data copying between the CPU and the GPU is carried out through a zero-copy buffer.

To improve the memory efficiency in the hypervisor environment, the memory overcommitment technique that decreases or increases the memory allocated to VMs is used. Waldspurger [23], Zhou et al. [24], Zhao et al. [25], Guo [26], Kim et al. [27], and Lu and Shen [28] periodically profiled the access frequencies of pages by nullifying the translation look-aside buffer (TLB) entries of randomly selected pages. Based on this, the least accessed pages are returned when the amount of memory is insufficient. However, this method has a problem because performance degradation may occur due to the nullification of the TLB entries. Furthermore, in [26], [27], [28], and [29], the problem of an inability to respond to sudden changes in VMs' memory demands due to the cyclic overcommitment exists. To solve this problem, the memory pressure aware (MPA) ballooning [27] applies different memory return policies by distinguishing the degree of memory pressure between an anonymous page and a file page. The MPA reduces the performance degradation caused by page return by setting the page with a high probability of becoming the least accessed as the object of return using Linux active and inactive lists. Furthermore, the MPA responds to the VMs' unexpected memory requests by immediately reallocating the memory to sudden memory requests and returning the memory slowly.

Recently, Park et al. [22] proposed a dynamic memory management technique for Intel's integrated GPU called DymGPU that provides two memory allocation policies: size- and utilization-based algorithms. Although DymGPU improves the performance of VMs by minimizing the overlap of the graphics memory space among VMs and thus reduces the context switch overhead, DymGPU's allocation is still static, and the memory size cannot be changed at runtime.

6. Conclusion and Future Works

In GPU virtualization, due to the static allocation of GPU memory, the performance of VMs that require more GPU memory can be degraded or the GPU application can crash. The *gBalloon*, proposed in this paper, improves the performance of VMs due to the lack of GPU memory by dynamically adjusting the GPU memory size allocated to each VM. Moreover, the *gBalloon* detects the increase in overhead due to GPU memory sharing and reduces the GPU memory size of the VMs that unnecessarily occupy a large amount of GPU memory. Consequently, the GPU context switch time is decreased, and the performance of the GPU applications is increased. Furthermore, the performance of the CPU applications is also guaranteed because the CPU load is reduced. The study demonstrated through experiments that the performance of the *gBalloon* improved by up to 32% when compared with the performance of *gVirt* with 1024 MB of high global graphics memory.

Currently, the *gBalloon* increases or decreases only the spaces at both sides to adjust the GPU memory space allocated to VMs. This problem can be solved by allocating non-consecutive spaces of small slot units rather than consecutive GPU memory spaces to VMs. We are currently investigating this issue.

Data Availability

The data used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This research was supported by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (2017M3C4A7080245).

References

[1] C. Reaño, F. Silla, G. Shainer, and S. Schultz, "Local and remote GPUs perform similar with EDR 100G InfiniBand," in *Proceedings of the Industrial Track of the 16th International Middleware Conference*, Vancouver, BC, Canada, December 2015.

[2] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Orti, "rCUDA: reducing the number of GPU-based accelerators in high performance clusters," in *Proceedings of International Conference on High Performance Computing & Simulation*, Caen, France, June 2010.

[3] L. Shi, H. Chen, J. Sun, and K. Li, "vCUDA: GPU-accelerated high-performance computing in virtual machines," *IEEE Transactions on Computers*, vol. 61, no. 6, pp. 804–816, 2012.

[4] Z. Qi, J. Yao, C. Zhang, M. Yu, Z. Yang, and H. Guan, "VGRIS: virtualized GPU resource isolation and scheduling in cloud gaming," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 2, pp. 1–25, 2014.

[5] S. M. Jang, W. Choi, and W. Y. Kim, "Client rendering method for desktop virtualization services," *ETRI Journal*, vol. 35, no. 2, pp. 348–351, 2013.

[6] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A GPGPU transparent virtualization component for high performance computing clouds," in *European Conference on Parallel Processing*, Springer, Berlin, Heidelberg, 2010.

[7] R. Montella, G. Giunta, and G. Laccetti, "Virtualizing high-end GPGPUs on ARM clusters for the next generation of high performance cloud computing," *Cluster Computing*, vol. 17, no. 1, pp. 139–152, 2014.

[8] R. Montella, G. Giunta, G. Laccetti et al., "On the virtualization of CUDA based GPU remoting on ARM and X86 machines in the GVirtuS framework," *International Journal of Parallel Programming*, vol. 45, no. 5, pp. 1142–1163, 2017.

[9] S. Xiao, P. Balaji, Q. Zhu et al., "VOCL: an optimized environment for transparent virtualization of graphics processing units," in *Proceedings of Innovative Parallel Computing (InPar)*, San Jose, CA, USA, May 2012.

[10] C. Zhang, J. Yao, Z. Qi, M. Yu, and H. Guan, "vGASA: adaptive scheduling algorithm of virtualized GPU resource in cloud gaming," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 11, pp. 3036–3045, 2014.

[11] C. Lee, S.-W. Kim, and C. Yoo, "VADI: GPU virtualization for an automotive platform," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 1, pp. 277–290, 2016.

[12] D. Abramson, "Intel virtualization technology for directed I/O," *Intel Technology Journal*, vol. 10, no. 3, 2006.
[13] C.-T. Yang, J.-C. Liu, H.-Y. Wang, and C.-H. Hsu, "Implementation of GPU virtualization using PCI pass-through mechanism," *Journal of Supercomputing*, vol. 68, no. 1, pp. 183–213, 2014.
[14] Amazon high performance computing cloud using GPU, http://aws.amazon.com/hpc/.
[15] R. Jennings, *Cloud Computing with the Windows Azure Platform*, John Wiley & Sons, Hoboken, NJ, USA, 2010.
[16] A. Herrera, *NVIDIA GRID: Graphics Accelerated VDI with the Visual Performance of a Workstation*, Nvidia Corp, Santa Clara, CA, USA, 2014.
[17] K. Tian, Y. Dong, and D. Cowperthwaite, "A full GPU virtualization solution with mediated pass-through," in *Proceedings of USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA, USA, June 2014.
[18] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, "GPUvm: why not virtualizing GPUs at the hypervisor?," in *Proceedings of 2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA, USA, June 2014.
[19] H. Tan, Y. Tan, X. He, K. Li, and K. Li, "A virtual multi-channel GPU fair scheduling method for virtual machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 2, pp. 257–270, 2019.
[20] M. Xue, "gScale: scaling up GPU virtualization with dynamic sharing of graphics memory space," in *Proceedings of 2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, USA, June 2016.
[21] M. Xue, J. Ma, W. Li et al., "Scalable GPU virtualization with dynamic sharing of graphics memory space," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 8, pp. 1823–1836, 2018.
[22] Y. Park, M. Gu, S. Yoo, Y. Kim, and S. Park, "DymGPU: dynamic memory management for sharing GPUs in virtualized clouds," in *Proceedings of 2018 IEEE 3rd International Workshops on Foundations and Applications of Self∗ Systems (FAS∗W)*, Trento, Italy, September 2018.
[23] C. A Waldspurger, "Memory resource management in VMware ESX server," *ACM SIGOPS Operating Systems Review*, vol. 36, pp. 181–194, 2002.
[24] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic tracking of page miss ratio curve for memory management," *ACM SIGOPS Operating Systems Review*, vol. 38, no. 5, p. 177, 2004.
[25] W. Zhao, Z. Wang, and Y. Luo, "Dynamic memory balancing for virtual machines," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 3, pp. 37–47, 2009.
[26] F. Guo, *Understanding Memory Resource Management in VMware vSphere 5.0*, VMware, Inc., Palo Alto, California, USA, 2011.
[27] J. Kim, V. Fedorov, P. V. Gratz, and A. L. Narasimha Reddy, "Dynamic memory pressure aware ballooning," in *Proceedings of the 2015 International Symposium on Memory Systems*, Washington DC, USA, October 2015.
[28] P. Lu and K. Shen, "Virtual machine memory access tracing with hypervisor exclusive cache," in *Proceedings of Usenix Annual Technical Conference*, Santa Clara, CA, USA, June 2007.
[29] D. Magenheimer, C. Mason, D. McCracken et al., "Transcendent memory and linux," in *Proceedings of the Linux Symposium*, Montreal, QC, Canada, July 2009.
[30] Phoronix Test Suite, http://phoronix-test-suite.com.
[31] Intel GVT-G (XENGT) public release-Q4'2016, https://01.org/igvt-g/blogs/wangbo85/2017/intel-gvt-g-xengt-public-release-q42016.
[32] UNIGINE valley, https://benchmark.unigine.com/valley.
[33] Cairo-perf-trace, http://www.cairographics.org.
[34] D. H. Bailey, E. Barszcz, J. T. Barton et al., "The nas parallel benchmarks," *International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991.
[35] S. Kato, M. McThrow, C. Maltzahn, and S. A. Brandt, "Gdev: first-class GPU resource management in the operating system," in *Proceedings of 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, Boston, MA, USA, June 2012.
[36] K. Wang, X. Ding, R. Lee, S. Kato, and X. Zhang, "GDM," *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, no. 1, pp. 533–545, 2014.
[37] F. Ji, H. Lin, and X. Ma, "RSVM: a region-based software virtual memory for GPU," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, Edinburgh, UK, 2013.
[38] 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 the 21st International Symposium on High-Performance Parallel and Distributed Computing*, Delft, Netherlands, June 2012.
[39] J. Kehne, J. Metter, and B. Frank, "GPUswap: enabling oversubscription of GPU memory through transparent swapping," *ACM SIGPLAN Notices*, ACM, vol. 50, no. 7, pp. 65–77, 2015.
[40] J. Kehne, M. Hillenbrand, J. Metter, M. Gottschlag, M. Merkel, and F. Bellosa, "GPrioSwap: towards a swapping policy for GPUs," in *Proceedings of the 10th ACM International Systems and Storage Conference*, Haifa, Israel, May 2017.
[41] D. Kang, T. J. Jun, D. Kim, J. Kim, and D. Kim, "ConVGPU: GPU management middleware in container based virtualized environment," in *Proceedings of 2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Honolulu, HI, USA, September 2017.