gMig: Efficient vGPU Live Migration with Overlapped Software-based Dirty Page Verification

Qiumin Lu, Xiao Zheng, Jiacheng Ma, Yaozu Dong, Zhengwei Qi, Jianguo Yao, Bingsheng He, and Haibing Guan

Abstract—This paper introduces *gMig*, an open-source and practical vGPU live migration solution for full virtualization. Taking the advantage of the dirty pattern of GPU workloads, gMig presents the One-Shot Pre-Copy mechanism combined with the hashing based Software Dirty Page technique to achieve efficient vGPU live migration. Particularly, we propose three core techniques for gMig: 1) Dynamic Graphics Address Remapping, which parses and manipulates GPU commands to adjust the address mapping and adapt to a different environment after migration, 2) Software Dirty Page, which utilizes a hashing based approach with sampling pre-filtering to detect page modification, overcomes the commodity GPU's hardware limitation, and speeds up the migration by only sending the dirtied pages, 3) Overlapped Migration Process, which significantly compresses the hanging overhead by overlapping the dirty page verification and transmission concurrently. Our evaluation shows that gMig achieves GPU live migration with an average downtime of 302 ms on Windows and 119 ms on Linux. With the help of Software Dirty Page, the number of GPU pages transferred during the downtime is effectively reduced by up to 80.0%. The design of sampling filter and overlapped processing can bring about further 30.0% and 10.0% improvements in page processing.

Index Terms—GPU, Virtualization, Migration

1 Introduction

GPUs have revolutionized cloud computing applications in domains such as 3D rendering and machine learning, leading to the rise of the demand of GPU resources on the cloud. In order to provide GPU instances on the cloud, several GPU virtualization solutions have been proposed. As a sophisticated and complex I/O device, GPU can be virtualized by mainly five methods: 1) device pass-through (PT), which is now widely used by vendors like Amazon [1] and Aliyun [2]; 2) API forwarding, such as vCUDA [3] and rCUDA [4]; 3) para-virtualization (PV), such as GPUvm [5], [6]; 4) SRIOV or similar hardware features, such as NVIDIA Grid [7] and AMD MxGPU [8]; 5) mediated pass-through (MPT), such as Intel GVT-g [9].

Among these methods and technologies, MPT (Mediated Pass-Through) based solutions are explored to provide a highly scalable IO virtualization architecture with near-native performance. It traps and emulates all privileged operations such as the MMIO register modification, and passes through performance-critical resources. As a result, by using MPT, a hypervisor is able to share a physical IO device among multiple VMs (virtual machines), and provides each VM a full featured device instance. Intel GVT-g, also known as gVirt, is an example of using MPT to virtualize IO devices. By successfully virtualizing GPU, GVT-g shows that MPT is an efficient method of IO virtualization.

VM migration is one of the most widely used key features in virtualization and cloud computing, and is of vital importance for IaaS (infrastructure as a service) providers. With VM migration, a cloud vendor can: 1) provide HA (high available) instances [10], 2) enable VM relocation and make the management of a data center more flexible, and 3) develop some power-saving techniques [11], [12]. We believe GPU live migration is important and it could bring a lot of benefits for IaaS vendors who provide virtualized GPU instances. By supporting GPU live migration, cloud vendors could provide all the mentioned features to GPU-enabled VMs. However, pass-through or SRIOV based GPU virtualization solutions are not migration-friendly, because the hypervisor can not be aware of some hardware context states [13], [14]. MPT is the only solution where the hypervisor can manipulate the GPU device states directly through the driver and control the vGPU execution flow [9], making it perfect for GPU live migration.

Still, even supporting GPU live migration on an MPT-based platform can be challenging. One major challenge is how to migrate a vGPU instance without stopping or pausing the virtual machine for a long time. This is due to both the complexity of GPU programming model and the lack of some special hardware features. For example, the absence of the dirty bit (a bit in the page table entry, which would be set to 1 if the corresponding page were modified) in the page table of Intel GPU[1], poses serious challenges

- *Q. Lu, Z. Qi, J. Yao and H. Guan are with the Shanghai Jiao Tong University.*
  E-mail: {luqiumin, qizhwei, jianguo.yao, hbguan}@sjtu.edu.cn
- *J. Ma is with University of Michigan.*
  E-mail: jcma@umich.edu
- *Q. Lu, J. Ma and Y. Dong are with Intel Corporation, and X. Zheng is with Alibaba Group.*
  E-mail: eddie.dong@intel.com, zhengxiao@gmail.com
- *B. He is with National University of Singapore.*
  E-mail: hebs@comp.nus.edu.sg

1. In this paper, Intel GPU refers to the Intel Processor Graphics.

when implementing live migration. Moreover, if a software-based solution is constructed to overcome the hardware limitation, how to reduce the overhead of the introduced software-based manipulation becomes another challenge.

This paper presents the system architecture called gMig[2], which is an efficient, practical, and open-source vGPU live migration solution for MPT based virtualization. First published in [15], gMig is also considered as a state-of-the-art GPU live migration solution. We demonstrate this gMig on the basis of Intel GVT-g, which is a sophisticated MPT-based full GPU virtualization solution for Intel GPU. In order to efficiently live-migrate virtual machines with vGPU instances, we overcome the hardware limitation and design a software-based migration mechanism including pre- and stop-and-copy phases with high efficiency in verification and transmission for graphics memory. Besides, compared with the gMig strategy with the improvement of hashing-based verification as the baseline, we inspect the shortages in current design for the further optimizations, including the optimizable page verification and the sequential migration process, and then reach a better performance in page verification and the whole migration process. The contributions of this paper about gMig can be summarized as follows:

- Provide a practical and efficient vGPU live migration solution in virtualization platform with mediated pass-through vGPU support, including the fundamental architecture and the optimized processing mechanisms. This solution is based on the Dynamic Graphics Address Remapping mechanism, solving the main problem of correcting shadowed vGPU memory address space mapping after migration. We denote it as gMig-Basic.
- Present the Software Dirty Page technique, which uses a hashing based approach to overcome hardware limitations of commodity GPU, and enables pre-copy for GPU live migration. This strategy, denoted as gMig-Hashing, is based on the conclusion from an analysis that vGPU has a better locality of memory writing.
- Present the sampling-based filtering stage implemented in dirty page verification, which faces the significant overhead of hashing process in the original gMig-Hashing. This fast bit-testing mechanism significantly reduces the unnecessary hashing frequency to improve the performance compared with this baseline.
- Present the overlapped migration process design to optimize the waiting overhead in the original gMig-Hashing by decoupling two sequential stages in page processing. According to this baseline, our strategy overlaps the dirty page verification and transmission concurrently in migration to optimize the computation resources and reduce the hanging overhead.
- Provide an evaluation to show that gMig achieves as fast as a sub-second downtime during live migration. Using a hashing based Software Dirty Page optimization technique, the average downtime can be reduced to 302 ms. Meanwhile, the GPU pages transferred during downtime are effectively reduced by 80.0%. As a baseline, this result can be further optimized with sampling pre-filtering by up to 37% and overlapped migration by about 13% compared with the original gMig configuration.

2. https://github.com/mjc0608/gMig-qemu

The rest of this paper is organized as follows. Section 2 describes the background of gMig system. In Section 3, we elaborate the design and implementation of gMig. In Section 4, we provide the evaluation of gMig. We discuss our work and present hardware proposal in Section 5. In Section 6, we present the related work, and finally in Section 7, we present the conclusions.

2 Background

2.1 Mediated Pass-through and Intel GVT-g

Mediated Pass-through (MPT) is a highly scalable methodology to virtualize IO devices, which can achieve both near-native performance and good security. By using MPT, a hypervisor can share a physical IO device among multiple VMs and provide each VM a full featured virtualized device. MPT traps and emulates privileged operations, and passes through the performance-critical resource directly. In the case of GPU, the MPT driver traps the modification of GPU page tables and MMIO registers, and executes the GPU commands, which are performance-critical, directly on the hardware. Intel GVT-g, also known as gVirt [9], is a practice of MPT-based virtualization solution for Intel GPU. With the help of GVT-g, a VM can freely access all the vGPU features through native graphics drivers.

Intel GPU reserves main memory partition as the graphics memory. In an Intel GPU, there are two page tables, called a) the *Global Graphics Translation Table* (GGTT) and b) the *Per-Process Graphics Translation Table* (PPGTT). The GGTT, presented in the MMIO space, is a single unique page table for all GPU components, e.g. the display engine and the render engine. The PPGTT is only visible to the corresponding process. Most batch buffers are presented in this address space, where the data for GPU access are stored.

GVT-g uses a software-based method to achieve GPU virtualization. However, the hardware limitations still raise serious challenges. Unlike the page tables of the main memory, neither the GGTT nor the PPGTT has support for dirty bits, because such support requires a DMA write, which is time-consuming. In a pre-copy style live migration, the dirty bits of the page table are used to trace the modifications in memory and reduce the unnecessary memory copy. Due to this lack of dirty-bit support, we may face challenge in employing this mechanism.

2.2 Live Migration and Pre-Copy

Live migration is a key feature of virtualization, where a virtual machine can be moved from one host to another without affecting or notifying the user of the VM. The effectiveness of live migration is measured by the required service downtime, which is the time that the VM must be stopped during the migration. Service downtime is mainly caused by the transmission of VM's memory.

The approach of pre-copy [16] is widely used to decrease the service downtime, where the key idea is to minimize

the pages transmitted during the downtime. This solution divides the migration procedure into two parts: 1) pre-copy phase, and 2) stop-and-copy phase. The first phase refers to the procedure copying data while the VM continues running at the source. In this phase dirtied pages are iteratively transmitted by rounds to minimize the transmission in the second phase. The second phase refers to the procedure transmitting data as the actual migration when the VM is restarted at the destination.

In the migration process, the hashing-based technique is introduced to verify and trace the modification of the memory space. In the page table, the dirty bit reflects if the related memory page is modified. However, traversing the whole memory page data and comparing it with the old record to verify the identity can be both time- and space-consuming. Alternatively, the hash fingerprint of a memory page is calculated and recorded, and the identity is verified through the comparison between the current fingerprint and the old one.

2.3 Dirty Pattern of GPU Workloads

Before migrating a vGPU, we should have an overview of the dirty pattern of GPU workloads, which has a strong influence on live migration. We trace and analyze the dirty pattern of different GPU intensive workloads in a virtualized environment. For this, we launch a new thread in QEMU to record the number of pages that are dirtied by the CPU and the GPU over a short period. From the analysis, we obtain the following statistics: 1) the ratio of pages dirtied by the CPU or the GPU at least once during execution, 2) the locality of the CPU and the GPU's memory writing operation in the whole process.

Firstly, we find that the dirty pattern of the GPU is similar to that of the CPU. In other words, only part of the graphics memory is actually dirtied. Secondly, we find that the memory writing of a GPU shows better locality than that of a CPU during GPU intensive workloads. Once a GPU page is dirtied, it is likely to be dirtied more frequently. Based on the above dirty pattern of GPU workloads, we obtain some intuitions that: 1) the pre-copy method can be applied to decrease the service downtime in GPU live migration, 2) since GPU dirty pattern shows better locality, we can only do few pre-copy rounds and skip a lot of rounds to decrease the total migration time. For the more detailed discussions and evaluation analysis about the GPU dirty pattern, the Appendix can be referenced.

3 Design and Implementation

In this section, we start with the overview of gMig, followed by the detailed design and implementation of each component, including the technique of detecting and verifying dirty pages without the help of hardware, along with performing graphics memory transmission and optimizing this process.

3.1 Top-level Architecture

The architecture of gMig is illustrated in Figure 1. gMig is based on KVMGT, which is the KVM version of Intel GVT-g solution. The gMig implementation includes modifications

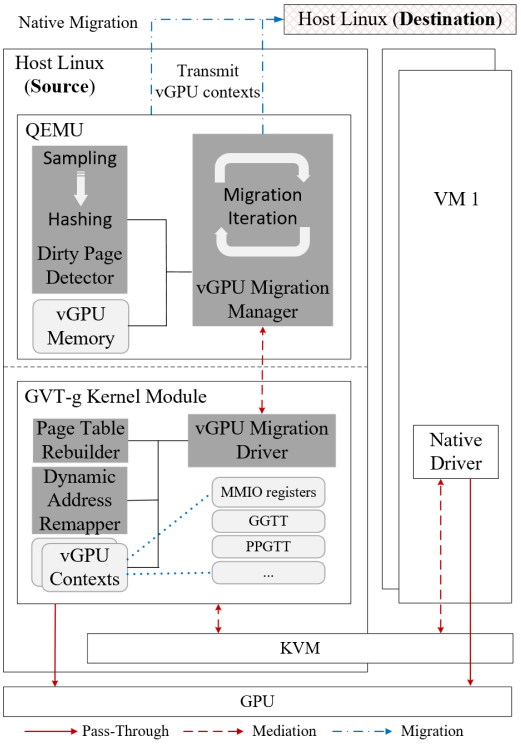


Fig. 1: gMig Architecture

on both Linux kernel and QEMU. In QEMU, the vGPU Migration Manager is responsible for controlling the vGPU migration procedure. It accesses the vGPU memory and then uses a hashing-based technique (i.e., Software Dirty Page) to identify the dirty pages of graphics memory in the Dirty Page Detector. After that, the Manager transmits these pages to the destination. Considering that the hashing traversal may usually be time-consuming, the additional sampling mechanism can be adopted to reduce the dirty verification overhead. Also, dirty identification and page transmission can be handled concurrently to avoid the hanging. vGPU Migration Manager also requires the kernel to provide some specific functionalities. gMig implements these functionalities and exposes them to QEMU via the vGPU Migration Kernel Driver. Through this interface, the Migration Manager can scan the memory access to determine the dirty region and then read the vGPU context status for the rebuilding after the migration.

Based on this architecture, the gMig migration flow illustrated in Figure 2 then becomes available in execution with the supporting functionalities it provides. In this migration iteration flow, first 1) the dirty record of vGPU memory pages constructed by the vGPU migration manager is traversed to search for dirty page address. Once a possibly dirtied page is found, 2) the system filters the dirty page through the sampling records and then 3) performs the hashing-based verification. After that, the searching and verifying stage is ended and then the system starts transmitting stage, where 4) the dirty page data is copied and transmitted for the 5) restoration at the destination. With the overlapped design, these two stages can be overlapped to process the verification and transmission concurrently.

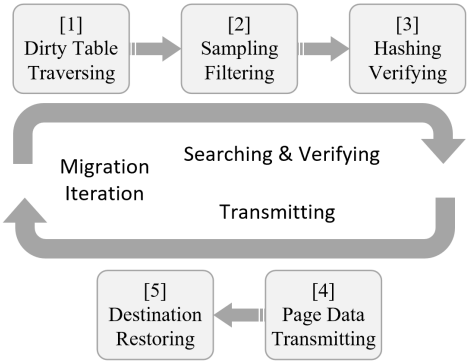


Fig. 2: gMig Migration Flow

3.2 Supporting Mechanisms for Migration

vGPU Context Clone: GVT-g maintains the states of each vGPU including GPU page tables (i.e. GGTT and PPGTT), MMIO register states and etc, which should be cloned and transmitted to the destination for guest reconstruction. The vGPU Migration Driver is responsible for packaging and exposing them to QEMU in the source, and then QEMU restores them in the destination.

1) Reconstruct Shadow Graphics Translation Tables: Due to the change of *Host Physical Address* (HPA) and host page tables of the VM after migration, both the shadow GGTT and PPGTT require reconstruction after migration to correct the memory access of the vGPU. In our implementation, the host driver possesses a copy of guest GGTT. While migrating a VM, gMig first transmits the guest GGTT to the destination, and then constructs a shadow page table where the GPA (guest physical address) in the PTE (page table entry) is replaced by HPA. The shadow PPGTT is rebuilt when the guest workload submits commands for the first time after migration. ***2) Reconstruct Shadow MMIO Registers:*** The CPU communicates with the GPU through MMIO registers. In order to share the GPU resource among the VMs, these registers are shadowed by GVT-g, and all visits to them will be trapped and emulated. During migration, these states are packaged by the vGPU Migration Driver and transmitted to the destination, where they are rebuilt.

Dynamic Graphics Address Remapping: As mentioned above, GVT-g shares the GGTT with a static partitioning, where each VM is assigned its disjoint partition. The guest driver will reference this information while generating GPU commands. Figure 3 shows an example, where the guest is assigned vGPU 1 partition. However, in the destination host, the guest has to use vGPU 2 after migration. If the guest driver is unaware of this partition change and still accesses the vGPU 1 partition, obviously this behavior will be considered illegal by GVT-g and crash the vGPU. gMig leverages GVT-g's command parser to fix the address mismatching with a tiny module to help the guest to adapt to the new environment. The graphics address space mismatching only happens in the GGTT address space. So we solve this mismatching problem by modifying all the addresses appearing in GPU commands located in the GGTT address space.

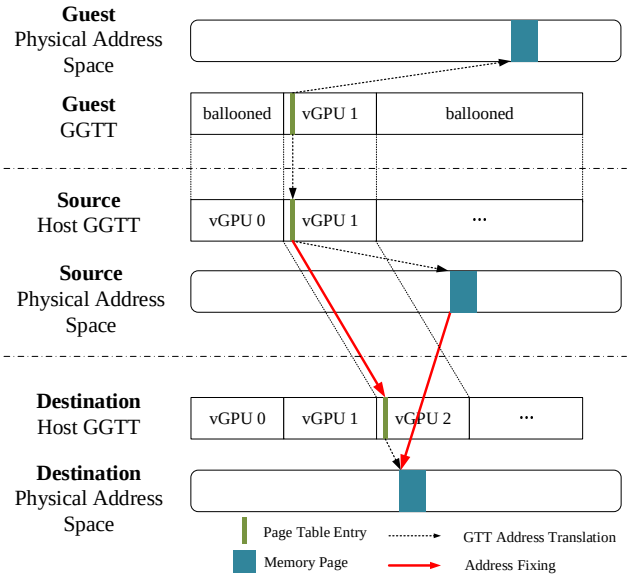


Fig. 3: Graphics Address after Migration

3.3 Mechanism for Dirty Page Verification

3.3.1 *Hashing Based Software Dirty Page*

Using the architecture described above, we can successfully migrate a vGPU instance by simply transferring and restoring the vGPU's context during the stop-and-copy phase, when both vCPU and vGPU are stopped. Unfortunately, the excessive number of pages accessed by vGPU may result in a relatively long service downtime, especially when the network is congested. In order to improve the performance of migration and reduce service downtime, we propose the idea of Software Dirty Page, which detects page modification by software and collects the set of dirtied vGPU memory pages. Then we are able to overcome the hardware limitation of commodity GPUs and implement a software based pre-copy mechanism for GPU live migration.

Considering the implementation difficulty and overhead, gMig implements Software Dirty Page by monitoring the hash value change of the pages. As shown in Figure 4, gMig ① generates and stores the hash value of all pages during the first round of memory page transmission, and ② transmits all pages. Then, during the rest transmission rounds, gMig ③ calculates the hash values again and ④ only transmits the pages whose hash value changes.

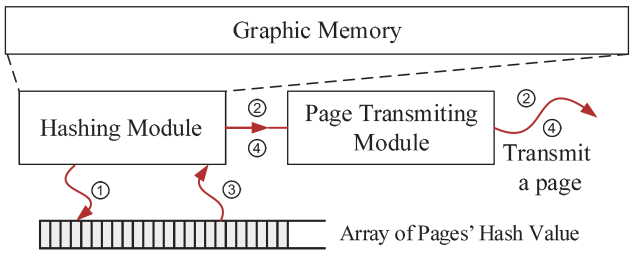


Fig. 4: Hashing Based Software Dirty Page Detector

When implementing the Software Dirty Page, various hashing algorithms were tested including XOR and

xxHash[3]. The hash value length is configured as 256 bits, and the XOR algorithm is optimized with AVX-2 instructions. We evaluate more hashing algorithms in Section 4.2. To ensure the reliability, the hash conflict is estimated. If assuming the pages are evenly distributed among all buckets, we can find that $p = 2^{-l}$, where p is the probability of two pages with the same hash value and l is the hash length. Since $l = 256$, p should be approximately 10^{-77}. Moreover, if denoting the probability of two pages with the same hash value in a k-page address space as P_k, there is:

$$P_k = 1 - \prod_{i=0}^{k-1} \frac{2^l - i}{2^l}. \tag{1}$$

The value of P_k should be approximately 10^{-62} in a 4GB address space (where k is 1M). This probability is far less than that of hardware failure, so it can be considered reliable. The downtime effect of utilizing this hashing optimization or not will be discussed in Section 4.2.

As mentioned before, gMig maintains the set of S_g in the host. While migrating a VM, QEMU reads S_g from the interface of the vGPU Migration Driver and obtains D_c from the kernel using KVM_IOCTL. After obtaining S_g and D_c, gMig categorizes all pages into two classes: 1) those should be transmitted directly, denoting as T_d, 2) those should be transmitted after hashing and comparing, denoting as T_h. It is obvious that at any time t_0:

$$T_d^{t_0} = D_c^{t_0}. \tag{2}$$

For stop-and-copy phase, we let:

$$T_h^{t_0} = \left(\bigcup^{t \le t_0} S_g^t \right) \setminus D_c^{t_0}. \tag{3}$$

For pre-copy phase, we let:

$$T_h^{t_0} = \bigcup^{t \le t_0} S_g^t. \tag{4}$$

All pages in D_c should be transmitted directly. However, since many GPU dirtied pages are also dirtied by the CPU, some pages may be transferred twice during a pre-copy or stop-and-copy round. Since we already know that these pages are dirtied and need to be transferred, to minimize service downtime, gMig removes the pages in D_c from the S_g bitmap during the stop-and-copy phase. We do not, however, remove these pages during the pre-copy phase. This is because during the pre-copy phase, the VM is still running and may still modify these pages. Besides, the guest driver may remove a modified GPU page from the GPU page tables. The removed GPU page may still be in the page table of the CPU, which means it can still be accessed by the CPU. If we do not hash and compare it, the CPU may read a wrong value or fail to read the value after the VM resumes in the destination. As a consequence, in Equation 3 and 4, we use the union of S_g since the beginning rather than $S_g^{t_0}$.

Moreover, apart from the pages in the GPU page tables, there is one extra case where the pages should be rehashed before transmission. We denote the set of these pages as H, and at any time t_0:

$$H^{t_0} = T_d^{t_0} \cap \left(\bigcup^{t \le t_0} S_g^t \right). \tag{5}$$

If a page dirtied by the CPU is also in the GPU page tables, the hash value of that page should also be updated, because the hash value kept by the QEMU should be exactly that of the page transmitted to the destination. If this value were not updated, some problem could occur in the next round of hashing and comparing.

3.3.2 Sampling-based Filtering in Dirty Verification

As we discussed above about the implementation of software-based memory tracking system on dirty page location with hash-like validating algorithms, it is obvious that the dirty verification process through the whole GPU memory space is time-consuming. The following evaluation results about the throughputs of various hash algorithms also confirm this fact, which becomes a significant factor of the overhead in migration process, causing throughput degradation and longer downtime. Facing this problem, we try to apply a specified strategy to optimize the whole verification process based on the speciality of GPU memory accessing pattern to relieve these extra expenses.

According to the dirty page pattern shown above, we can get a clear impression that the hit style of the GPU memory has better locality and higher frequency. It means that such a dirty page may be touched and modified repetitively in the whole migration procedure. So we can conclude that a majority of the pages tagged dirty in the pre-copy stage have been further modified and will be validated as dirty in the stop-and-copy phase. Considering this pattern, a strategy which is able to filter out most of the dirty page quickly can apparently optimize the overhead by reducing the hashing frequency while the migration.

So we introduce the sampling strategy, that the system stores a few sampling bytes of the whole memory page when first hashing this dirty one, and then tests these stored bytes for identity verification during the stop-and-copy phase. This strategy only requires much less time consumption, but can filter out most of dirty pages without the hashing process. It is obvious that the false negative rate is inversely proportional to the length of the sampling record. The left pages should be validated by the page-hashing process. With this sampling strategy, if the dirty rate of GPU memory page accessing is high enough, this sampling process can reduce much of the hashing work.

Also, considering the memory storage space expenses of this strategy, obviously it only needs a few bytes for every memory page as the page sampling record. Compared with the length of the whole memory page, the space overhead at this level is acceptable and negligible, since even the space overhead of hashing fingerprints is an order of magnitude more than this sampling records. So there is no block in our way to adopting sampling strategy if we need to consider the space issues.

Moreover, the sampling strategy can be flexibly adjusted to reach an optimized performance result in this specific applicative environment. One dimension of the adjustment is

3. https://github.com/Cyan4973/xxHash

the distribution of the sampling bits. The sampling strategy may have different false negative rate if we apply different distributions, such as uniform distribution, and assigning a higher density at the head or tail region of the whole address space. The other dimension of the adjustment is the number of the sampling bits. Apparently, more sampling bits can result in lower false negative rate but longer testing process, and less sampling bits can reduce the testing overhead but cause higher false negative rate. There should have a precise trade-off to get the optimized performance.

In our implementation, we mainly realize this sampling features appended to the hashing mechanism. First, we add a new field in the structure recording page-related logging data, where the dirty page table is another field word. We will record the sampling record per page at this position in the migration process. Also, we modify the functions responsible for driving the memory page traversing in the migration process. Before invoking the hashing methods, we insert the sampling features for record saving and identity verification. With these modifications, the migration process will record the configured sampling record of every page during pre-copy, and then filtering dirty page before hashing according to the recorded sampling record during stop-and-copy phase.

Although the simple speculative strategy described above requires only some incremental modification, it can intuitively reduce a majority of the computation expenses in validation process, and the overhead it introduces is obviously acceptable.

3.4 Memory Transfer Implementation and Optimization

3.4.1 Graphics Memory Transmission

Intel GPUs utilize part of the main memory as the graphics memory. As a result, the main memory can be categorized into two sets: S_g^t, which denotes for the set of memory used by the GPU at time t, and S_c^t, which denotes for the set of memory used by the CPU at time t. As a result, at any time t, we have:

$$S_c^t \neq \emptyset, S_g^t \neq \emptyset, S_g^t \not\subset S_c^t, S_c^t \cap S_g^t = S_{cg}^t \neq \emptyset. \quad (6)$$

Since only S_c and S_{cg} are CPU-accessible, we have to handle $S_g \setminus S_{cg}$ specifically during migration. We modify the procedure of allocation and deallocation of the graphics pages to keep the set of S_g in the vGPU Migration Driver. Then, as a basic approach, we simply add pages in S_g to the dirty bitmap of QEMU before the stop-and-copy phase and therefore all the requisite graphics pages can be successfully transferred to destination.

However, this basic approach suffers from a drawback. Since we have to send all pages in S_g in the stop-and-copy phase, the service downtime will be relatively longer. Actually, if denoting dirtied CPU pages at time t as D_c^t and dirtied GPU pages at time t as D_g^t, it is easy to find:

$$D_c^t \subset S_c^t, D_g^t \subset S_g^t, D_g^t \not\subset D_c^t, D_c^t \cap D_g^t = D_{cg}^t \neq \emptyset. \quad (7)$$

Only D_g, which is a subset of S_g, needs to be transmitted during the stop-and-copy phase. By applying a pre-copy alike method, we can reduce the number of pages that need to be transferred during the stop-and-copy phase, and minimize the service downtime. We have discussed this approach in Section 3.3.1.

3.4.2 Overlapped Migration Procedure

The gMig system discussed above has realizes the functionality of GPU memory migration, but it is still far from avoiding performance bottlenecks. The sequential design pattern of the whole migration procedure is just one obvious architectural bottleneck inherited from QEMU-based system prototype. This sequential design causes problems when the additional hash-like verification process is introduced into the gMig. Since the migration implementation in gMig requires the hash-like mechanism for tagging and verifying memory accessing pattern, the verification and transmission tasks have become two time-consuming workloads with no necessity to be executed in sequence and to be pending for each other. As a result, the hashing calculation delay becomes a new time overhead, where the overlapped design can utilize this time delay and accelerate the whole process.

In the pre-copy stage, the sequential design first searches for a dirty GPU memory page, and then generates the hash snapshot of the current page. After that, the migration process transfers this dirty page. Also, in the stop-and-copy phase, the sequential design searches for a dirty GPU memory page again, and then generates the hash snapshot of the current page to compare it with the previous snapshot, validating if this page is modified after being transferred. If there is no modification, this page will be skipped, and otherwise this page will be transferred again.

So it is obvious that the hashing step and the transferring step in the pre-copy phase have no dependent relationship. The migration process can easily hash a memory page and transfer it concurrently with no risk of data consistency. In the stop-and-copy phase, the situation becomes a bit more complicated. After all, there is a dependent relationship between the two steps. That is, only when the identity verification through hashing step is completed then the migration process can confirm if transferring this memory page is necessary. However, if we amortize the whole process, we can find out that there is no dependent relationship between the step of transferring the previous memory page and the step of validating the identity of the current memory page. We can also process the two steps in concurrency.

In our development, we overlap the hashing step and the transferring step in the migration process to improve the performance. In the pre-copy phase, the iteration procedure of the migration process still continues searching for dirty GPU memory page. Once such a page is found, the transferring task and the hashing task for this page will all be registered and respectively pushed into the pending queues for hashing and transferring. In this design, the hashing workloads and the transferring workloads are processed independently and asynchronously. In the stop-and-copy phase, the iteration procedure of the migration process continues searching for dirty GPU memory page and then pushes it into the pending queues for hash-based identity verification. When the verification is finished, if it is necessary for this page to be transferred again, the relevant transferring task will be pushed into the pending queues for transferring. Since the validating workloads and the transferring workloads are also processed independently

and asynchronously in this stage, the processing of the two types of workloads is also overlapped and in concurrency.

According to our implementation, we will introduce new worker thread to handle the hashing-based verification concurrently with the main migration thread responsible for memory page traversing and transmitting. In the original design, there is only one iteration loop executing on a single thread responsible for the whole migration process, including page traversing and verification, along with transmission to the destination. We upgrade this execution flow to a trivial producer-consumer model, where the main thread traverses the dirty page table to select dirty memory page. In the pre-copy phase, the selected page is submitted to the worker thread for calculating hashing fingerprint concurrently, while the main thread starts the transmission of the page data. In the stop-and-copy phase, the main thread submits the selected page to the worker thread and then fetches the completed hashing result to judge the necessity of transmission. As a result, then in the whole migration procedure, all the hashing workloads and transferring workloads are processed in overlapped pattern, without unnecessary hanging waiting for independent processing tasks and irrelevant system resources.

4 Evaluation

This section evaluates gMig from several aspects. Firstly, our experiments measure the service downtime of gMig under different configurations in various factors, considering that the service downtime is the main criterion when measuring the efficiency of live migration. Since the gMig is the only state-of-the-art solution for vGPU migration, we use the gMig-Basic configuration as the baseline, and then these experiments significantly illustrate the contribution of the proposed hashing-based memory verification strategy (gMig-Hashing). Secondly, we evaluate the performance impact of migration on workloads. The loss of efficiency shows that the GPU performance overhead due to migration is acceptable. Additionally, we evaluate the optimizing effect of the sampling mechanism for reducing hashing overhead in dirty page verification (gMig-Sampling) compared with the gMig-Hashing version as the baseline where only hashing-based verification is enabled in the migration. This strategy largely reduces the unnecessary hashing calculation to improve the performance of the memory page verification. Finally, we verify the effectiveness of overlapped migration design (gMig-Overlapped) in hanging time reduction, where the sequential migration design in the gMig-Hashing is the baseline. Obviously, with this strategy, the stalling idle can be omitted in the gMig migration process. All the experiment configurations for the evaluations are listed in the Table 1 as follows.

TABLE 1: **Experiment Configurations.**

Experiment	Configurations	
Algorithm Comparison	gMig-basic gMig-xor gMig-xxHash	
Bandwidth Comparison	5Gbps 7.5Gbps 10Gbps	
Baseline:gMig-Basic Improvement:gMig-Hashing		
Sampling Filtering	Workload	Tropic PerfTest Heaven 3DMark06
	Test Length	1-byte 2-byte 4-byte
	Test Location	Head Tail Uniform
Baseline:gMig-Hashing Improvement:gMig-Sampling		
Overlapped Migration	Sequential Idealized Overlapped	
Baseline:gMig-Hashing Improvement:gMig-Overlapped		

Source / Destination Host Configuration	
CPU	i5-6500
GPU	Intel Graphics HD 530
Memory	8GB
Storage	SAMSUNG 850EVO 256G
Network	Intel 82599ES 10-Gigabyte NIC
OS	Ubuntu 16.04
Kernel	4.3.0-rc6
Linux / Windows VM Configuration	
vCPU	2
Memory	2GB
OS	Ubuntu 16.04 / Windows 10 Redstone

TABLE 2: Experimental Configuration

4.1 Experimental Setup

The experiments are conducted on two physical machines configured as shown in Table 2. Each guest is configured with 2 vCPUs and 2 GB system memory, running Ubuntu 16.04 with the kernel version of 4.3, and Windows 10 Redstone with native graphics drivers. And there is only one pre-copy round in the following evaluations according to the pre-copy round impact analysis in Appendix. Since GPU workloads focus on jobs like 3D rendering and machine learning on cloud environment, our experiments mainly use 3D benchmarks and machine learning applications. For 3D rendering on Linux, we choose *Phoronix Test Suite 3D marks*[4]. It is a popular benchmarking suite with benchmarks including *Lightsmark*, *Nexuiz*, *OpenArena*, and *Warsow*. For Windows 3D rendering, we use *3DMark 06*[5], *3DMark 11*[6], *Heaven*[7], and *Tropics*[8]. Most machine learning benchmarks use CUDA, running on NVIDIA GPU. In order to include

4. https://www.phoronix-test-suite.com
5. https://www.futuremark.com/benchmarks/legacy
6. https://www.futuremark.com/benchmarks/3dmark
7. https://benchmark.unigine.com/heaven
8. https://benchmark.unigine.com/tropics

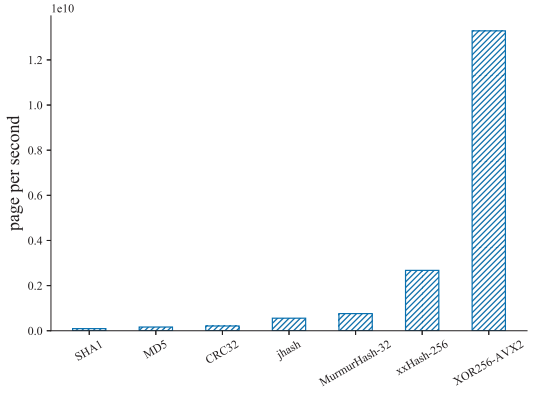


Fig. 5: Throughput of Different Hash Algorithms

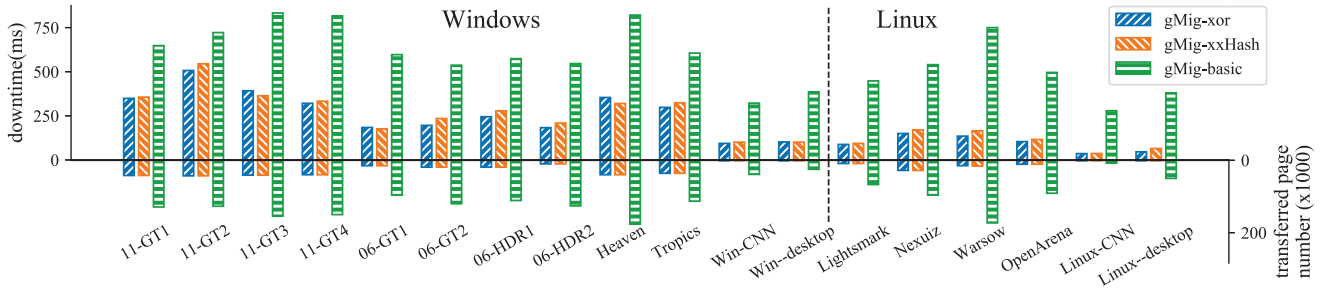


Fig. 6: Impact of Hash Algorithm on the Downtime

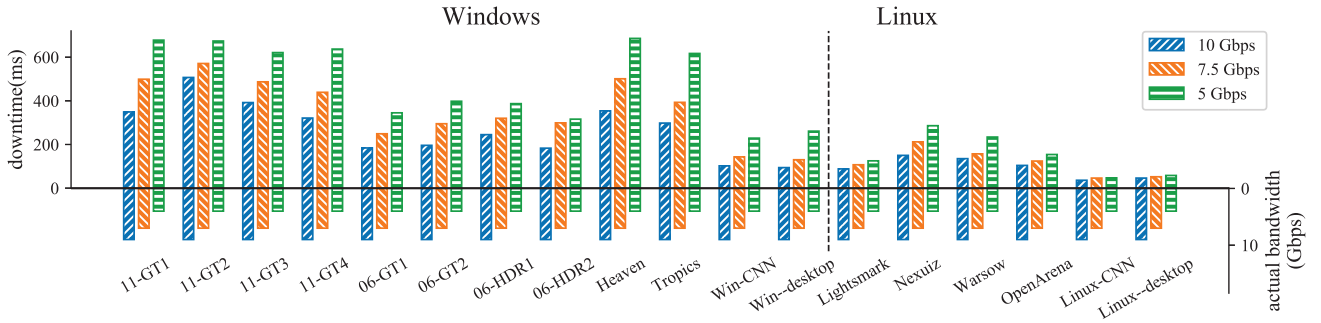


Fig. 7: Impact of the Bandwidth on the Downtime

machine learning benchmarks in our evaluation, we create an image recognition application using a classic convolutional neural network (CNN). The benchmark is based on clDNN[9], a compute library for neural networks on the Intel platform. We mainly use this model to evaluate the performance of gMig during migration.

4.2 Performance Analysis on Migration Downtime

Observation 1: The proposed hashing-based software dirty page verification techniques can significantly optimize the migration downtime by reducing page transmission. The optimization extent can be up to 80%. This influence is even much heavier than that of the bandwidth.

Algorithm: We experimented with several hashing algorithms while implementing the Software Dirty Page of gMig. The throughput of 7 different hashing algorithms is shown in Figure 5. Optimized by AVX2 instruction set extension, XOR256 is the fastest one and achieves more than 141x performance of SHA1 and 80x performance of MD5. xxHash is also fast, and achieves 28x performance of SHA1. We mainly use XOR256 and xxHash to evaluate gMig in the following experiments.

With hashing algorithms like XOR256 and xxHash, the Software Dirty Page can significantly decrease the service downtime by reducing the number of pages transferred during the stop-and-copy phase. However, executing a hashing function over all the GPU pages can be time-consuming. In this experiment, we individually apply the XOR256 or xxHash functions to evaluate the downtime of multiple benchmarks and record the number of GPU pages transferred during the stop-and-copy phase. We use gMig without Software Dirty Page as the baseline for evaluation. The results of the evaluation are shown in Figure 6. The *gMig-basic* transfers all the GPU pages while the *gMig-xor* and the *gMig-xxHash* adopt a Software Dirty Page Detector based on XOR256 and xxHash. *Win-desktop* and *Linux-desktop* respectively represent migrating Window and Linux without running any workloads other than its default desktop environment. *Win-CNN* and *Linux-CNN* represent a classical Convolutional Neural Network running on Window and Linux.

We find that when a hash function is adopted, there is an 80% average decrease in the number of GPU pages transferred during the stop-and-copy phase. This reduction in the number of pages can greatly decrease the service downtime and greatly improve the performance of the system.

Bandwidth: The network bandwidth between the source and the destination machines may have impact on the service downtime during migration. In this experiment, we evaluate the downtime of multiple benchmarks when the bandwidth is increased from 5 Gbps to 7.5 Gbps and to 10 Gbps. The XOR256 hash function is adopted in this experiment for the benchmarking.

As shown in Figure 7, the downtime of a service has a strong negative correlation with the network bandwidth. Even for some low-load benchmarks such as *Lightsmark* and *OpenArena*, the downtime increased by 42.1% and 48.1% respectively as the bandwidth was dropped from 10 Gbps to 5 Gbps. For benchmarks with heavy-load such as *Heaven*, the downtime can increase to over 90%. Besides, Figure 7 indicates the real bandwidth used in the migration. As we can find in the figure, this value is very close to the given band-

9. https://01.org/cldnn

width. Figure 7 also demonstrates that the network speed is approximately proportional to the speed of migration. This means that the number of pages transferred during the stop-and-copy phase approximately remains unchanged.

4.3 Workload Performance Analysis During Migration

Observation 2: The workload performance degradation during the gMig migration is not obvious, which indicates that the overhead of gMig solution is acceptable.

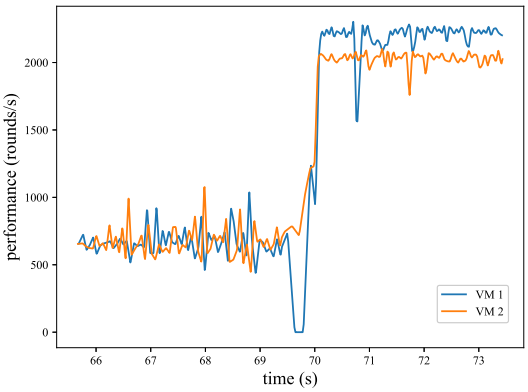


Fig. 8: The Speed of the CNN while Migrating

We create a classical CNN application to log the performance change. In this experiment, we launch two Windows VMs (VM 1 and VM 2) on the source host, with the above benchmark running inside them. Then we migrate VM 1 to the destination side and observe the performance of both VMs. Figure 8 shows the relationship of the performance of the CNN and the execution time during migration. The execution of the guest VM is inevitably stopped during the stop-and-copy stage. Therefore, a notable pause can be observed during the execution of the CNN in VM 1. The performance drop is not obvious in the pre-copy phase because the hashing procedure is done by the CPU while the benchmark is GPU-intensive. Additionally, the performance of both VMs boost up after migration. This shows that gMig effectively implements a load balancing policy in a GPU cluster. In the stop-and-copy phase, since the VM 1 is not scheduled, the performance of the machine drops to zero. However, considering the short service downtime, the performance loss is acceptable.

4.4 Performance Analysis of Sampling Pre-filtering

Observation 3: The sampling pre-filtering strategy can filter about 90% dirty page requiring hashing and provide up to 37.1% overhead reduction in dirty verification.

In this section, we will discuss the performance improvement effect of the sampling-based pre-filtering strategy. Since in the current system architecture, the minimum processing unit in the comparison operation is the byte-length word, there is no profit in reducing overhead to perform the sampling on a more fine-grained record length. So, we will perform the evaluation under the sampling record length of 1, 2, and 4 bytes on behalf of performance-first, mean and effectiveness-first settings respectively.

The Figure 9 refers to the percentage distribution of the types of processed dirty pages in stop-and-copy phase of the migration. During the migration, the workload Tropics was being executed and the first byte in every memory page is tested. According to the data, about 18.9% pages are new dirty pages which require no verification. In the remaining pages, about 38.4% pages have no modifications after being transferred, which have to be fully validated through hashing procedure. About 37.8% pages can be filtered dirty through sampling, saving the hashing overhead. Only about 4.8% pages will be false negative in sampling filter and require hashing procedure to confirm that they are dirty. According to this percentage distribution where the false negative rate is only about 11.4%, we can conclude that the sampling strategy is significantly effective. The Figure 10 further summarizes a horizontal comparison among different on-migration workloads including the Tropics, the PerfTest, the Heaven and the 3DMark06 Demo, along with different sampling record lengths (1, 2, and 4 bytes). The result still reflects the similar conclusion.

In the Figure 11 we illustrate the time expenses per page in the whole migration duration whether additional sampling is active. All the results are the integrations of the sampling overhead and the reduced hashing time optimized by sampling filter. The configuration is the same as above in Figure 9. According to the evaluation result, the data reflect that we can get much lower overhead in average after enabling sampling strategy. In the current situation, the average page processing overhead can be reduced from 468.6 ns to 353.5 ns, which is a 24.5% performance improvement. The Figure 12 expand the comparison among experiments under different configurations. We have listed the evaluation results of the sampling strategy with 1-, 2-, and 4-byte sampling records testing the beginning of the memory page, under the execution of the Tropic, PerformanceTest, Heaven and 3DMark06 workloads. We compare the performance under these configurations with that under non-sampling strategy. Then we can still get a similar conclusion as above. Quantitatively, the sampling strategy can provide up to 37.1% reduction on time overhead, which is 184.22ns per page. Also, reflected by the chart, we can conclude that the 1-byte sampling record is already an acceptable configuration in performance. According to the 3DMark06 example, the doubled or even quadrupled scanning overhead can be more than offset by the further reduced hashing time with longer sampling length.

Also, the Figure 13 and Figure 14 illustrates the status when we access the sampling data at different locations of the memory page, including the head, tail and spreading the whole space. According to the evaluation result, we can see that recording the head of a page is always best choice in performance. So it is obvious that it is unnecessary to sampling in uniform through the whole memory page, and the tail of a memory page is usually not sensitive to the modification on this page. For example, in the configuration that sampling 1 byte at page tail for the Tropic workload, the sampling overhead can not be offset by the reduced hashing time, and then enabling sampling results in even worse performance.

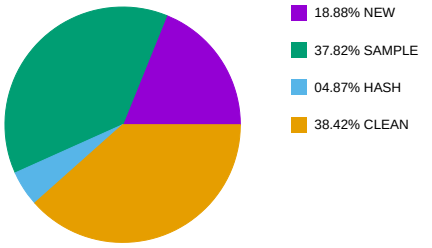


Fig. 9: Page processing type distribution.

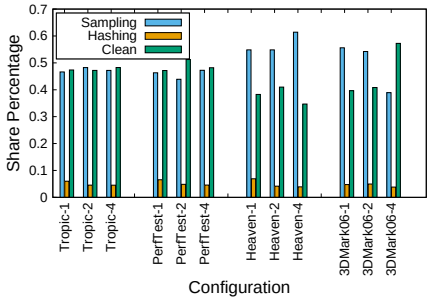


Fig. 10: Page processing type distribution in different configurations.

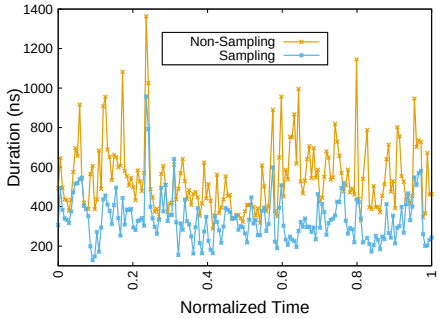


Fig. 11: Comparison of time expenses per page in migration duration whether sampling is active.

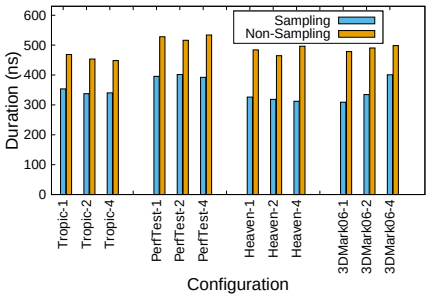


Fig. 12: Comparison of average time expenses per page whether sampling is active in different configurations.

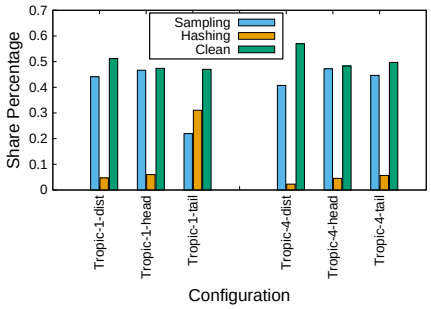


Fig. 13: Page processing type distribution in different record locations and different record lengths.

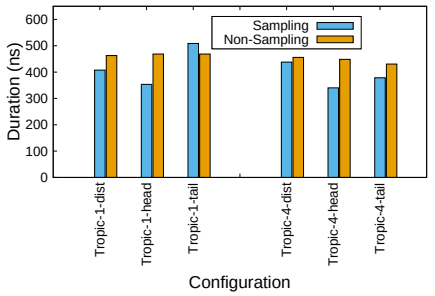


Fig. 14: Duration comparison in different record locations and different record length.

4.5 Performance Analysis of Overlapped Migration Process

Observation 4: The overlapped migration process design can reduce the hanging time overhead between the page verification and transmission with parallelized processing. This improvement can further reduce the time expense in migration by about 12%.

Now we will discuss the optimizable stalling overhead between the hashing step and the transferring step in the dirty GPU memory migration process and the improvable memory page processing throughput. In the following evaluations, we will record the precise timestamps representing the beginning and the end of every processing step, and then calculate out the time consumption of their duration and the stalling overhead in hashing and transferring every GPU memory page. Based on the comparison of these recorded data between the sequential situation and the improved overlapped one, we can apparently illustrate the improvement in performance with the overlapped design.

In the evaluations, we will change the configuration about the executing workloads during the whole migration process. Since different GPU workloads can result in various memory accessing pattern and different access distributions in the migration time period, we can get a more general summary about the optimization effect of the overlapped migration process design in various memory accessing pattern, which can confirm actual performance improvement.

In the following evaluations, we will record the timestamps of every dirty memory page processing iteration, and the durations of hashing stage and transferring stage respectively. In the non-overlapped original design, this iteration process is sequential, so we will simply record these runtime data at the end of every iteration. And in the improved overlapped design situation, the hashing stage and the transferring stage are running independently and asynchronously. So, we will record their timestamps and durations separately. Considering the fact that the transferring stage are processed in the main migration thread loop, we regard the time record of every beginning of dirty memory page transfer as the tag of the migration procedure.

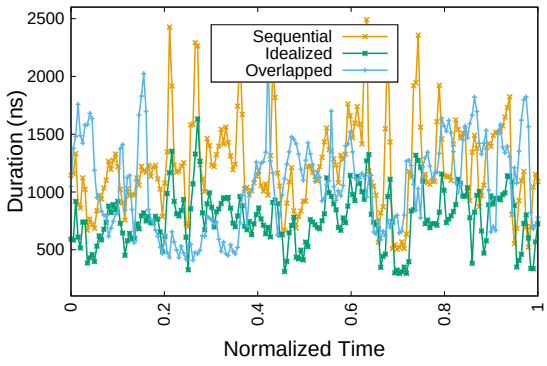


Fig. 15: Duration comparison for overlapped design in migration duration.

The Figure 15 illustrates the performance of our overlapped migration design through the comparison on the dirty page processing duration in the migration procedure.

In this figure, we mainly compare the performance under three different strategies. The first one is our proposed overlapped migration design. The second one is the original sequential design. And the last one reflects the idealized concurrent situation by recording the larger duration value between the hashing and transferring time expenses of processing every page. This figure records every per-page processing duration value during the whole stop-and-copy phase for more typical discussion.

According to this figure, we can see that in most time region, the proposed overlapped migration design can result in a lower per-page processing time compared with the original sequential design. Also, many processing time peaks which may caused by the network delay in the transferring step are relieved or avoided in our overlapped migration design. The idealized situation reflects the best performance level in this figure. Considering the new overhead in the proposed overlapped design introduced by the multi-thread mechanism, including the thread switching and data contention overheads, there is still a gap on performance between our overlapped design and the idealized situation.

Quantitatively, according to the evaluation result, we can see that the overlapped design has reduced the total time overhead compared with the sequential design from about 1218.9 ns to 1054.3 ns in average, where the idealized situation is 772.8 ns in average. There the overlapped design incurs an 13.5% (164.6 ns per page) performance improvement. However, there is still a gap between the evaluation result and the idealized situation at about 281.5 ns. This 26.7% performance gap can be a space for our further optimization.

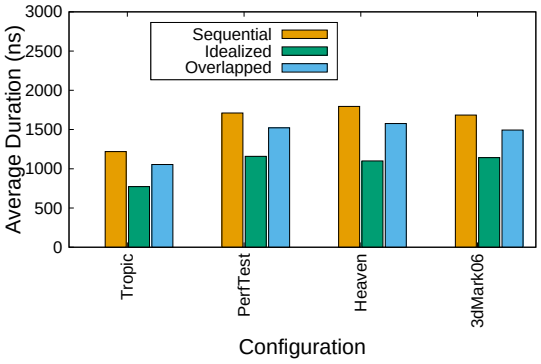


Fig. 16: Average duration comparison for overlapped design with different workload configurations.

This chart in Figure 16 is a comparison among evaluation results with different workload configurations during migration. We listed the average time expenses of page processing in sequential design, overlapped design and the idealized situation with overlapping. The workloads used include the Tropic, PerformanceTest, Heaven and 3dMark06. The evaluation results have confirmed the performance improvement reflected in the above discussion again. According to the figure, the time expense reductions are 13.5% (164.6 ns), 11.0% (189.0 ns), 12.1% (217.8 ns) and 11.3% (191.2 ns) respectively, which contributes to the performance improvement. Because of the introduced concurrent overhead, there is still a gap to the idealized performance level, which may require further optimizing.

5 Discussion

5.1 Architecture Independence

Although we implement our solution on the Intel platform, the overall idea and architecture are portable and architecture independent, and can be migrated to any other GPU virtualization solution based on mediated pass-through. In this paper, we analyze the dirty pattern of GPU workloads, which has similar characteristics on different platforms. We show the possibility of using pre-copy to reduce the downtime of GPU live migration, which is also architecture independent. Vendors such as AMD and Qualcomm, who have integrated CPU/GPU systems, can benefit from the model designed in this paper. For companies such as NVIDIA, which sell off-chip GPU systems, the idea is still useful and possible to be implemented because the hardware interfaces of on-chip GPU and off-chip GPU are similar. To implement Software Dirty Page on off-chip GPU platforms, we can simply offload the task of calculating the hash value of the graphics pages to the GPU. Furthermore, since gMig can achieve the live migration of GPU, which is one of the most complex I/O devices today, we believe the methodology could also be applied to other MPT based IO virtualization.

5.2 Hardware Proposal

Even though the vGPU can be migrated without the help of any specific hardware, the migration can definitely be better and easier with specific hardware support. If GPU vendors can add dirty bits or similar hardware features to the GPU hardware, the service downtime can be further shortened. This is because, by using dirty bits we can offload the task of detection of dirty pages to the hardware. This would ensure that the migration of vGPU does not suffer from the performance loss due to calculating and comparing hash value of pages. Once dirty bits or similar features are added to the GPU hardware, the migration time of the service can be decreased by approximately 15.3% compared with our implementation of hash method according to micro downtime analysis. To overcome the potential slow-down of the memory access of GPU, the hardware can let the driver determine whether to turn on this feature.

5.3 Power benefit

With the gMig solution introduced in the GPU-aware VM migration process, there can also be power benefit caused by the strategies proposed. First, the hashing-based memory page verification significantly reduces the memory accessing activities in the whole migration process, which largely degrades the power overhead. Second, the one-shot memory copying strategy also saves the power consumption by simplifying the data transmission. Moreover, the overlapped migration processing further reduces the power overhead by avoiding the stalling idles.

6 Related Work

6.1 Live Migration

Live migration has gained a lot of attention in both industrial and research communities [17]. Clark et al. in 2005, studied the live migration of operating systems and were the first group to propose the pre-copy based live migration approach [16]. M. R. Hines, in 2009, proposed a post-copy based live migration approach of a VM [18]. Nowadays, both the pre-copy and the post-copy techniques are quite popular in hypervisors such as Xen [16], [18] and KVM [19].

Some optimizations have been developed for transferring the VM's content during live migration. Jin et al. [20] studied the regularity of the pages in memory to divide pages into three categories based on their similarity and percentage of zero bytes inside them. Svard et al. [21] introduced a simple but fast compression algorithm which only migrates the delta values between the current and the previous pages. There are other methods which focus on only migrating some meta data and establishing the VM in the same-state at the destination. The CR/RT motion [22] is one such method, which only transfers the checkpoints and the trace-log files in pre-copy phase, and then recreates the memory page by reading these files. Koto et al. [23] conducted a comprehensive research on the usage of different VM pages by not transferring some soft pages and then recreating them on the destination VM.

Some researchers found that each time only small proportion of dirty pages are frequently updated. Hu et al. [24] proposed a time-sliced pre-copy approach, where the system would look through historical records, figure out the changeable pages and transmit these pages only at the last round to reduce repeated transmissions.

Similar to other areas, it is also an effective method to improve the performance of migration system through parallelized modification on the system architecture and the execution flow. Chanchio et al. [25] introduces multi-threads in migration pre-copy to fulfill the time-bounded requirement. Liu et al. [26] constructs the migration process with asynchronous mechanism for state consistence guarantee between the source and the destination. Gilesh et al. [27] schedules live migration placement among multiple virtual machines in parallel to optimize the completion time and relieve the inter-VM interference. Nathan et al. [28] has made a horizontal summary and related comparison.

Currently, the proposed gMig migration solution is the only state-of-the-art study in vGPU migration strategy, which is firstly discussed in [15]. There we denote the original version as gMig-Basic and the version with the proposed improvement of hashing-based verification as gMig-Hashing. In this paper extending this study for the better performance, we have further proposed the sampling pre-filtering strategy (gMig-Sampling) and the overlapped migration process design (gMig-Overlapped) to improve the original hashing-based page verification and the sequential migration process design.

6.2 GPU Virtualization

GPU virtualization has been widely researched in recent years [29], [30]. Traditionally, API forwarding qualifies vGPUs with functions such as graphics libraries for acceleration. For example, Xen3D [31] and Blink [32] use a new OpenGL library on Linux which forwards OpenGL API to help many VMs to concurrently perform their graphics tasks without interrupting the host machines. GViM [33] and vCUDA [3] have proposed a GPGPU runtime to VMs by forwarding CUDA commands. Furthermore, rCUDA [4], [34] uses RDMA to speed up the forwarding of CUDA commands. Intel's GVT-s [35] is used to accelerate the GPU computation in multiple VMs.

Several software based GPU virtualization solutions have also been proposed. Such solutions include gVirt (GVT-g) [9] for Intel GPU, and GPUvm [6] for NVIDIA GPU. gVirt is based on the idea of mediated pass-through (MPT), which traps and emulates privileged operations and passes through the most performance critical operations to the hardware. GPUvm is based on both MPT and PV (para-virtualization). gHyvi [36] uses a hybrid shadow page table to improve the performance of gVirt for memory-intensive workloads. In order to extend the scalability of GPU virtualization, gScale [37], [38] uses a private shadow page table to support up to 15 VMs on a single Intel GPU.

The GPU pass-through technology can achieve the best performance [13] for virtualization. Amazon [1] and Aliyun [2] applied this technique to achieve high computing efficiency on the GPU instances offered to their customers. Intel GVT-d [35] can pass-through Intel GPU to the VM. However, all device pass-through based GPU virtualization solutions can only support one VM per physical GPU.

SR-IOV [14] goes further to optimize I/O virtualization. Supported by hardware, SR-IOV or similar hardware based I/O virtualization can achieve both high performance and scalability. AMD recently puts forward its hardware-based GPU virtualization product: the AMD multiuser GPU [8], which is based on SR-IOV and can support up to 15 VMs per GPU. NVIDIA GRID [7] is now hardware supported and can handle up to 16 VMs per GPU. NVIDIA also recently announced a vGPU live migration solution based on XenServer for GRID [39]. However, all these products are close-sourced, with no publicly technical details.

7 Conclusion

gMig presents an open-source GPU live migration solution for MPT based full virtualization. By exploiting the dirty pattern of GPU workloads, gMig uses a one-shot pre-copy mechanism combined with the hashing-based Software Dirty Page technology to speed up vGPU migration by only sending the dirtied pages. gMig also implements Dynamic Graphics Address Remapping to adapt to a new environment after migration. Evaluations have shown that the average downtime can be reduced to 302 ms on Windows and 119 ms on Linux respectively. We also observed that number of GPU pages transferred during downtime is effectively reduced by 80.0%. In addition, we optimized the page verification with sampling-based filtering, which provides a further overhead improvement by up to 37.1% as 184.22 ns per page reduction. The overlapped migration design can also achieve the time reduction up to 13.5% (164.6 ns per page). We believe that gMig introduces good management flexibility for product-level full GPU virtualization platforms.

ACKNOWLEDGMENTS

This work was supported in part by National Key Research & Development Program of China (No.2016YFB1000502), National NSF of China (NO. 61672344, 61525204, 61572322, 61732010), and Shanghai Key Laboratory of Scalable Computing and Systems. The corresponding authors are Prof. Qi, Zhengwei and Prof. Yao, Jianguo.

REFERENCES

[1] (2016) Introducing amazon ec2 p2 instances, the largest gpu-powered virtual machine in the cloud. https://aws.amazon.com/about-aws/whats-new/2016/09/introducing-amazon-ec2-p2-instances-the-largest-gpu-powered-virtual-machine-in-the-cloud/.

[2] "Elastic gpu service," https://www.alibabacloud.com/product/gpu, 2017.

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

[4] C. Reaño and F. Silla, "Reducing the performance gap of remote gpu virtualization with infiniband connect-ib," in *2016 IEEE Symposium on Computers and Communication (ISCC)*, June 2016, pp. 920–925.

[5] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, "Gpuvm: why not virtualizing gpus at the hypervisor?" in *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference*. USENIX Association, 2014, pp. 109–120.

[6] ——, "Gpuvm: Gpu virtualization at the hypervisor," *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 2752–2766, 2016.

[7] "Grid virtual gpu user guide," http://images.nvidia.com/content/grid/pdf/GRID-vGPU-User-Guide.pdf, 2016.

[8] "Amd multiuser gpu: Hardware-enabled gpu virtualization for a true workstation experience," http://www.amd.com/Documents/Multiuser-GPU-White-Paper.pdf, 2016.

[9] K. Tian, Y. Dong, and D. Cowperthwaite, "A full gpu virtualization solution with mediated pass-through," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 121–132. [Online]. Available: http://dl.acm.org/citation.cfm?id=2643634.2643647

[10] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. San Francisco, 2008, pp. 161–174.

[11] A. Beloglazov and R. Buyya, "Energy efficient resource management in virtualized cloud data centers," in *Proceedings of the 2010 10th IEEE/ACM international conference on cluster, cloud and grid computing*. IEEE Computer Society, 2010, pp. 826–831.

[12] L. Liu, H. Wang, X. Liu, X. Jin, W. B. He, Q. B. Wang, and Y. Chen, "Greencloud: a new architecture for green data center," in *Proceedings of the 6th international conference industry session on Autonomic computing and communications industry session*. ACM, 2009, pp. 29–38.

[13] J. P. Walters, A. J. Younge, D. I. Kang, K. T. Yao, M. Kang, S. P. Crago, and G. C. Fox, "Gpu passthrough performance: A comparison of kvm, xen, vmware esxi, and lxc for cuda and opencl applications," in *2014 IEEE 7th International Conference on Cloud Computing*, June 2014, pp. 636–643.

[14] Y. Dong, X. Yang, X. Li, and H. Guan, "High performance network virtualization with sr-iov," in *IEEE, International Symposium on High Performance Computer Architecture*. IEEE, 2012, pp. 1471–1480.

[15] J. Ma, X. Zheng, Y. Dong, W. Li, Z. Qi, B. He, and H. Guan, "Gmig: efficient gpu live migration optimized by software dirty page for full virtualization," in *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, 2018, pp. 31–44.

[16] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," vol. 2. USENIX Association, 2005, pp. 273–286.

[17] K. Boos, A. A. Sani, and L. Zhong, "Eliminating state entanglement with checkpoint-based virtualization of mobile os services," in *Proceedings of the 6th Asia-Pacific Workshop on Systems*. ACM, 2015, p. 20.

[18] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy live migration of virtual machines," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 3, pp. 14–26, Jul. 2009. [Online]. Available: http://doi.acm.org/10.1145/1618525.1618528

[19] S. Sahni and V. Varma, "A hybrid approach to live migration of virtual machines," in *Cloud Computing in Emerging Markets (CCEM), 2012 IEEE International Conference on*. IEEE, 2012, pp. 1–5.

[20] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan, "Live virtual machine migration with adaptive memory compression," in *IEEE International Conference on Cluster Computing and Workshops*. IEEE, 2009, pp. 1–10.

[21] P. Svard, B. Hudzia, J. Tordsson, and E. Elmroth, "Evaluation of deltacompression techniques for efficient live migration of large virtualmachines," in *Proceedings of the 7th ACM International Comference on Virtual Execution Environments*. ACM, 2011, pp. 75–86.

[22] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu, "Live migration of virtualmachine based on full system trace and replay," in *Proceedings of the 18th International Symposium on High Performance Distributed Computing*. HPDC, 2009, pp. 101–110.

[23] A. Koto, H. Yamada, K. Ohmura, and K. Kono, "Towards unobtrusive vm live migration for cloud computing platforms," in *Proceedings of the Asia-Pacific Workshop on Systems*. ACM, 2012, p. 7.

[24] B. Hu, Z. Lei, and Y. Lei, "A time-series based precopy approach for live migration of virtual machines," in *IEEE International Conference on Parallel & Distributed Systems*. IEEE, 2012, pp. 947–952.

[25] K. Chanchio and P. Thaenkaew, "Time-bound, thread-based live migration of virtual machines," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*. IEEE, 2014, pp. 364–373.

[26] H. Liu, H. Jin, X. Liao, C. Yu, and C.-Z. Xu, "Live virtual machine migration via asynchronous replication and state synchronization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 12, pp. 1986–1999, 2011.

[27] M. Gilesh, S. Satheesh, A. Chandran, S. M. Kumar, and L. Jacob, "Parallel schedule of live migrations for virtual machine placements," in *2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*. IEEE, 2018, pp. 64–70.

[28] S. Nathan, U. Bellur, and P. Kulkarni, "On selecting the right optimizations for virtual machine migration," in *ACM SIGPLAN Notices*, vol. 51, no. 7. ACM, 2016, pp. 37–49.

[29] C.-H. Hong, I. Spence, and D. S. Nikolopoulos, "Gpu virtualization and scheduling methods: A comprehensive survey," *ACM Comput. Surv.*, vol. 50, no. 3, pp. 35:1–35:37, Jun. 2017. [Online]. Available: http://doi.acm.org/10.1145/3068281

[30] Z. Zhang, X. Xu, M. Xue, and Y. Dong, "gha: An efficient and iterative checkpointing mechanism for virtualized gpus," in *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*. ACM, 2016, pp. 1:1–1:8.

[31] C. Smowton, "Secure 3d graphics for virtual machines," in *Proceedings of the Second European Workshop on System Security*, ser. EUROSEC '09. New York, NY, USA: ACM, 2009, pp. 36–43. [Online]. Available: http://doi.acm.org/10.1145/1519144.1519150

[32] J. G. Hansen, "Blink: Advanced display multiplexing for virtualized applications," in *Proceedings of NOSSDAV*, 2007.

[33] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, "Gvim: Gpu-accelerated virtual machines," in *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*. ACM, 2009, pp. 17–24.

[34] F. Pérez, C. Reaño, and F. Silla, *Providing CUDA Acceleration to KVM Virtual Machines in InfiniBand Clusters with rCUDA*. Cham: Springer International Publishing, 2016, pp. 82–95. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-39577-7_7

[35] "Intel graphics virtualization technology (intel gvt)," https://01.org/igvt-g, 2015.

[36] Y. Dong, M. Xue, X. Zheng, J. Wang, Z. Qi, and H. Guan, "Boosting gpu virtualization performance with hybrid shadow page tables," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 517–528.

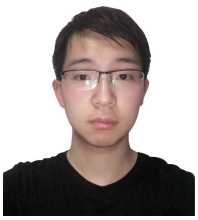
[37] M. Xue, K. Tian, Y. Dong, J. Ma, J. Wang, Z. Qi, B. He, and H. Guan, "gscale: Scaling up gpu virtualization with dynamic sharing of graphics memory space," in *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '16. Berkeley, CA, USA: USENIX Association, 2016, pp. 579–590. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026959.3027012

[38] M. Xue, J. Ma, W. Li, K. Tian, Y. Dong, J. Wu, Z. Qi, B. He, and H. Guan, "Scalable gpu virtualization with dynamic sharing of graphics memory space," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[39] "High availability: Nvidia grid showcases vgpu monitoring and migration," https://blogs.nvidia.com/blog/2017/06/22/high-availability-nvidia-grid-showcases-vgpu-monitoring-and-migration/, 2017.

Qiumin Lu received the M.E degree from Jiao Tong University, Shanghai, PR China in 2017. 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.

Xiao Zheng was senior staff engineer in Intel Open Source Technology Center, now work for Alibaba Cloud. His research focuses on parallel computing, virtualization. He received master degree from Zhejiang University in 2006.



Jiacheng Ma is a PhD student at University of Michigan. His research mainly focuses on operating systems and heterogeneous hardware.

Yaozu Dong received the PhD degree from the Computer Science & Engineering Department, Shanghai Jiaotong University. He is a principal engineer and software architect in the Intel Open Source Technology Center. His research focuses on architecture and systems, including virtualization, operating system, and distributed and parallel computing.

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 a 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 received the B.E, the M.E degree and the Ph.D. degree from the Northwestern Polytechnical University (NPU), Xian, Shaanxi, PR China, respectively in 2000, 2007 and 2010. Currently, he is an Associate Professor and Associate Dean of School of Software at the Shanghai Jiao Tong University. His research interests are cloud computing, virtualization and cyber-physical systems.

Bingsheng He ingsheng Heingsheng HeB is currently an Associate Professor at Department of Computer Science, National University of Singapore. Before joining NUS in May 2016, he held a research position in the System Research group of Microsoft Research Asia (2008-2010) and a faculty position in Nanyang Technological University, Singapore. He got the Bachelor degree in Shanghai Jiao Tong University (1999-2003), and the Ph.D. degree in Hong Kong University of Science & Technology (2003-2008). His current research interests include Big data management systems (with special interests in cloud computing and emerging hardware systems), Parallel and distributed systems and Cloud Computing.

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.