Memory-virtualizing and -devirtualizing VM Migration with Private Virtual Memory

Yuji Muraoka
Kyushu Institute of Technology
murayu@ksl.ci.kyutech.ac.jp

Kenichi Kourai
Kyushu Institute of Technology
kourai@csn.kyutech.ac.jp

***Abstract*—Recently, Infrastructure-as-a-Service clouds provide virtual machines (VMs) with a large amount of memory. Such large-memory VMs can be migrated to other hosts, but it is costly to always preserve hosts with sufficient memory as the destinations of VM migration. Instead, *memory-virtualizing VM migration* using virtual memory is possible, but the performance of VM migration and migrated VMs largely degrades because traditional virtual memory causes excessive paging during and after VM migration. This paper proposes *VMemDirect*, which achieves efficient memory-virtualizing VM migration. VMemDirect provides *private virtual memory* with private swap space on an NVMe SSD for each VM. Then, it directly transfers the memory data of a VM to either physical memory or private swap space to completely avoid paging during VM migration. In addition, VMemDirect provides efficient *memory-devirtualizing VM migration* for VMs running on private virtual memory. To optimize the performance of migrated VMs, it uses a more accurate and efficient LRU approximation using *chunk queues* and performs asynchronous paging. We showed that VMemDirect dramatically improved the performance of VM migration and migrated VMs using large VMs.**

1. Introduction

Recently, Infrastructure-as-a-Service (IaaS) clouds provide VMs with a large amount of memory. For example, VMs provided by Amazon EC2 have up to 24 TB of memory. When a host running a VM is maintained, the execution of the VM can be continued by migrating the VM to another host in advance. VM migration transfers the state of a VM, e.g., virtual CPUs, memory, and virtual devices, and resumes the VM at the destination host. Therefore, VM migration requires a large host with sufficient memory that can accommodate the entire memory of a VM. However, it is not cost-efficient to always preserve such large hosts as the destinations of occasional VM migration. In particular, private clouds may not afford to prepare a sufficient number of large hosts.

Even if there is no such destination host, *memory-virtualizing VM migration* is possible by using virtual memory. In this method, part of the memory of a migrated VM is stored in swap space. When a VM requires memory data in swap space, that data is paged in, while unnecessary memory data is paged out. However, traditional virtual memory is not suitable for VM migration. Since frequent paging occurs during VM migration, the performance of VM migration degrades largely. After the migration, the execution performance of the VM is also largely affected by paging because necessary memory data is often paged out. Split migration [1] using multiple hosts has been proposed to address these issues, but it is costly and is subject to host and network failures.

This paper proposes *VMemDirect* for efficient memory-virtualizing VM migration. VMemDirect provides *private virtual memory*, which consists of a fixed amount of physical memory and *private swap space* created on a fast and inexpensive NVMe SSD, for each VM. Then, it directly transfers the memory data of a VM to either physical memory or private swap space. Since this *direct memory transfer* does not cause any paging, performance degradation can be avoided. Using memory access prediction of a VM, VMemDirect transfers likely accessed memory data to physical memory and reduces paging after VM migration. Also, VMemDirect supports efficient *memory-devirtualizing VM migration*, which migrates a VM running on private virtual memory to the destination host with sufficient memory.

We have implemented VMemDirect in KVM. Private swap space is created using a *sparse file* to enable direct memory transfer. Upon VM migration, VMemDirect divides the memory of a VM into two by the likelihood of future memory access. For memory access prediction, VMemDirect performs least-recently-used (LRU) management accurately and efficiently using the *chunk queues*. After VM migration, it performs *asynchronous paging* to handle page-ins as fast as possible. Our experimental results show that VMemDirect could dramatically improve the performance of VM migration and migrated VMs. It was also shown that VMemDirect often outperformed split migration.

This paper is an extension of our workshop paper [2]. First, we have implemented our memory-virtualizing VM migration with various optimizations. Second, we have designed and implemented efficient memory-devirtualizing VM migration. Third, we conducted thorough experiments using large VMs with up to 240 GB of memory and compared VMemDirect with split migration.

The organization of this paper is as follows. Section 2

describes issues on traditional memory-virtualizing VM migration. Section 3 proposes VMemDirect, and Section 4 describes its implementation. Section 5 shows experimental results. Section 6 discusses related work, and Section 7 concludes this paper.

2. Migrating Large-memory VMs

If a destination host does not have sufficient memory, *memory-virtualizing VM migration* is possible. It leverages virtual memory at the destination host. It can transparently store part of the memory of a VM in swap space on secondary storage and use a necessary amount of memory. When the VM requires memory data in swap space, a page-in is performed and the data is moved to physical memory. In exchange for this, a page-out is performed and unlikely accessed data in physical memory is moved to swap space.

However, it is pointed out that traditional virtual memory is not suitable for VM migration [1]. The migration time increases due to excessive paging. VM migration first transfers the entire memory data of a VM to physical memory at the destination host. After physical memory becomes full, the following transfers always cause page-outs to store the received data in physical memory. Also, retransfers of memory data updated during VM migration often cause paging. If memory data to be updated exists in swap space, VM migration has to page in that data and then update it.

The downtime also increases due to frequent paging. In the final phase, VM migration stops the VM and transfers the rest of the state consistently. If the device emulator for a VM uses the same virtual memory as the VM at the destination host, its memory is usually paged out during VM migration. When the device emulator restores the state of virtual devices, many page-ins occur. After VM migration, read-only memory data can be stored in swap space even if it is frequently accessed. Since such data is transferred only once, it is often paged out by following memory transfers. It can cause page-ins later.

To counteract these problems, split migration [1] has been proposed. It divides the memory of a VM and transfers the memory fragments to smaller hosts. It transfers likely accessed memory data to the main host and the rest of the memory data to sub-hosts. If the VM requires memory data in sub-hosts after VM migration, the data is transferred to the main host by remote paging. Since no paging occurs during VM migration, split migration can improve the migration performance. Also, it can increase the execution performance of the migrated VM because necessary memory data is transferred to the main host in advance.

However, split migration is more costly than memory-virtualizing VM migration. It requires one or more sub-hosts in addition to the main host. For efficient remote paging, a dedicated high-speed network such as InfiniBand is necessary [3]. NICs and switches for such a network are expensive. In addition, migrated VMs are subject to host and network failures because they run across multiple hosts. If a failure occurs in only one of the hosts or the network, a VM cannot continue to run.

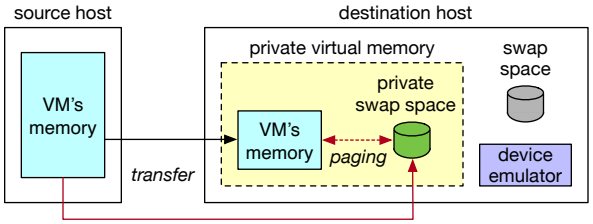


Figure 1. Memory-virtualizing VM migration using private virtual memory.

Therefore, we *revisit* memory-virtualizing VM migration in this paper. Recently, NVMe SSDs are widely used and become less expensive. Using NVMe SSDs as swap space, the performance of memory-virtualizing VM migration could be improved. Although too many writes to swap space shrink the lifespan of NVMe SSDs, migrated VMs usually do not cause excessive paging thanks to memory access locality. However, only using NVMe SSDs cannot eliminate the above-mentioned overhead completely. To achieve the performance comparable to split migration, we need special-purpose memory management for VM migration and migrated VMs.

3. VMemDirect

This paper proposes *VMemDirect* for efficient memory-virtualizing VM migration. VMemDirect uses *private virtual memory* instead of system-wide virtual memory at the destination host, as illustrated in Fig. 1. VMemDirect assigns a fixed amount of memory to private virtual memory when starting VM migration. It also creates *private swap space* on a fast and inexpensive NVMe SSD for private virtual memory. Thanks to virtual private memory, VMemDirect can prevent performance degradation due to the paging of the memory of the device emulator at the destination host. Since the device emulator runs outside private virtual memory, the memory of the device emulator is not paged out by memory transfers during VM migration.

Upon VM migration, VMemDirect directly transfers the memory data of a VM to either physical memory or private swap space instead of relying on paging. At the source host, VMemDirect divides the memory of a VM into two by determining the locations where memory data is stored. At the destination host, it writes memory data to the specified locations. This *direct memory transfer* can prevent paging from occurring during VM migration. No data in physical memory is paged out, while no memory data in private swap space is paged in. When VMemDirect retransfers memory data, it directly updates data in either physical memory or private swap space. Since the blocks in private swap space are mapped onto the memory chunks in the VM one-to-one, it is easy to find the corresponding blocks.

In addition to memory-virtualizing VM migration, VMemDirect supports efficient *memory-devirtualizing VM migration*. This migration method migrates a VM running on private virtual memory to the destination host with sufficient memory. VMemDirect directly transfers memory data from

both physical memory and private swap space to physical memory in the destination host. It does not perform paging to transfer memory data in private swap space. Even if the running VM causes paging during VM migration, VMemDirect transfers that memory data correctly. VMemDirect can perform both memory-devirtualization and memory-virtualization at the same time. It directly reads memory data from both physical memory and private swap space at the source host. Then, it transfers the data to either physical memory or private swap space at the destination host.

To improve the performance of migrated VMs, VMemDirect predicts the future memory access of a VM using the LRU algorithm. Like the previous work [1], it uses the aging algorithm as an LRU approximation. Since the previous work uses only an 8-bit memory access history for each memory chunk, the accuracy of LRU decreases if a large amount of memory access is done in a short period. For more accuracy, VMemDirect uses *chunk queues*, which manage the access recency of memory chunks more rigidly. The chunk queues also allow VMemDirect to select a victim memory chunk more rapidly. In addition, VMemDirect performs page-ins and page-outs asynchronously. This enables VMemDirect to handle page-in requests as fast as possible.

4. Implementation

We have implemented VMemDirect in QEMU-KVM 2.11.2 and Linux 4.18.

4.1. Private Swap Space

For private swap space, VMemDirect creates a swap file on an NVMe SSD. This swap file has the same size as private virtual memory and has one-to-one mapping onto the memory of a VM. Memory data of a VM is stored in either physical memory or the swap file. To save space on storage, the swap file is created as a *sparse file*, which contains memory data only in necessary file blocks. The other blocks have no memory data and are called *holes*. When memory data is moved to physical memory, VMemDirect changes the corresponding blocks of the swap file to holes.

VMemDirect accesses the swap file using direct I/O so that the page cache is not allocated for its file blocks. Direct I/O enables data to be directly read from and written to storage without storing it in the page cache managed by system-wide virtual memory. To use the bandwidth of NVMe SSDs as much as possible, VMemDirect accesses the swap file by the 256-page memory chunk.

4.2. Chunk Queues for LRU Management

VMemDirect uses 2^m queues called *chunk queues* to achieve an accurate and efficient aging algorithm. As shown in Fig. 2, the first queue manages least recently used memory chunks, while the last queue manages most recently used ones. VMemDirect uses 256 queues ($m = 8$) in the current implementation. At first, all memory chunks are added to

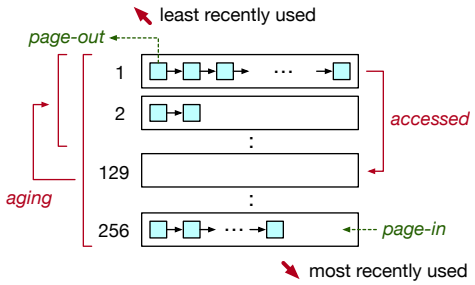


Figure 2. The chunk queues for efficient LRU management.

the first queue. The head of each queue is a less recently used memory chunk, while the tail is a more recently used one.

VMemDirect periodically updates the chunk queues according to the access bits in the extended page tables (EPT). If the access bit for a memory chunk is set, VMemDirect moves that memory chunk in the i-th queue to the tail of the $(i + 2^{m-1})$-th queue. After a certain period, VMemDirect compresses 2^m queues into the former 2^{m-1} queues for aging. That is, it creates a new i-th queue from the $(2i-1)$-th and $2i$-th queues. Then, it makes the latter 2^{m-1} queues empty.

Upon a page-out, VMemDirect can find a victim memory chunk in $O(2^m)$. It searches for a non-empty queue from the first queue and just dequeues a memory chunk from the head of that queue. Upon a page-in, VMemDirect appends a memory chunk to the tail of the last queue in $O(1)$. For memory division on VM migration, VMemDirect scans the chunk queues from the head of the first queue. Then, it marks the specified number of memory chunks to send to private swap space and the rest of the memory chunks to be sent to physical memory.

4.3. Asynchronous Paging

VMemDirect performs asynchronous paging to handle page-ins as fast as possible. When a VM accesses a non-existent memory page, the page-in thread reads memory data from the swap file and writes it to the memory pages allocated to the VM. After that, it adds a page-out request to the *paging queue* and can handle the subsequent page-in request immediately. Unlike synchronous paging, the page-in thread does not wait for the completion of page-outs. It does not need to delete the corresponding blocks from the swap file.

When the page-out thread asynchronously receives that request via the paging queue, it selects a victim memory chunk. To prevent the conflict with page-ins performed in parallel, it acquires a lock for that memory chunk and then performs page-outs. In addition, it deletes the blocks corresponding to the paged-in memory chunk from the swap file and makes holes asynchronously.

As a further optimization, VMemDirect can perform not only page-outs but also most of the page-ins in each memory

TABLE 1. EXPERIMENTAL SETUP.

	source host	destination (main) host	destination sub-host
CPU	Intel Xeon Silver 4110 ×2	AMD EPYC 7262 ×1	Intel Xeon Silver 4110 ×2
Memory	2666 MT/s RDIMM 256 GB	2666 MT/s RDIMM 128 GB	2666 MT/s RDIMM 128 GB
NIC	Intel X550-T2 (10 GbE)	Broadcom 57416 (10 GbE)	Intel X550-T2 (10 GbE)
SSD	–	Samsung 970 PRO NVMe SSD 1 TB	–

chunk asynchronously. The page-in thread handles only a faulting page. Then, it adds a request to the paging queue. As a result, it can handle the subsequent page-in without waiting for page-ins of the rest of the pages in the same memory chunk. Later, the page-out thread performs the rest of the page-ins and page-outs asynchronously.

5. Experiments

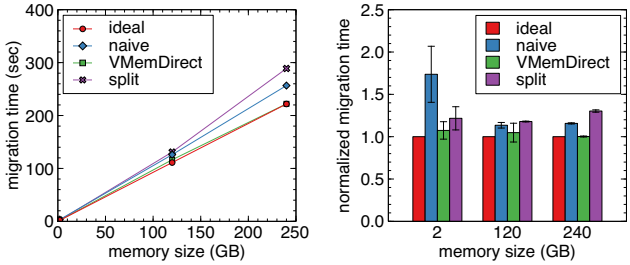
We conducted experiments to examine the performance of VM migration and migrated VMs in VMemDirect. For comparison, we examined the performance of *ideal* migration, *naive* migration, and split migration [1]. The ideal migration migrated a VM to the destination host with sufficient memory. The naive migration performed memory-virtualizing and -devirtualizing VM migration using traditional virtual memory. We applied the efficient LRU management in VMemDirect to split migration as well.

We used three hosts shown in Table 1. When we performed the ideal migration, we attached an extra 128 GB of RDIMMs to the destination host to accommodate a VM with 240 GB of memory. These hosts were connected using a 10 Gigabit Ethernet (GbE) switch. We ran a VM with four virtual CPUs and 2-240 GB of memory. We used Linux 4.18.17 as the host and guest operating systems and QEMU-KVM 2.11.2 as the device emulator. For VMemDirect, we configured the size of physical memory assigned to private virtual memory to half of the memory size of the VM. For the naive migration, we adjusted the size of free memory in the host to half of the memory size of the VM. For split migration, we equally divided the memory of the VM into two.

5.1. Performance of Migrating an Idle VM

To examine migration performance, we first measured the time needed to migrate an idle VM. As shown in Fig. 3(a), VMemDirect was as fast as the ideal migration, faster than the naive migration, and much faster than split migration. To compare the migration time in detail, we show the normalized migration time in Fig. 3(b). VMemDirect was 8.3-62% faster than the naive migration. It could transfer memory data directly to private swap space and suppress paging. Compared with split migration, the performance improvement was 13-30%. This is probably because the CPU performance of the sub-host was lower than that of the destination host used for VMemDirect. The performance degradation from the ideal migration was only 0.2-6.9%.

Next, we measured the downtime during VM migration. As shown in Fig. 4, VMemDirect could reduce the downtime



(a) Migration time (b) Migration time (ratio)

Figure 3. The migration time of an idle VM.

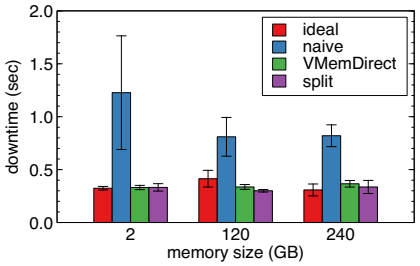


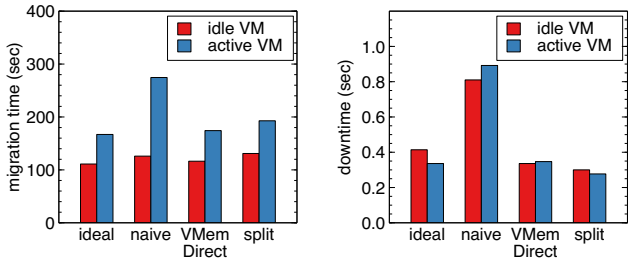
Figure 4. The downtime of an idle VM.

by 454-896 ms, compared with the naive migration. In the naive migration, many page-ins occurred when virtual devices were resumed in the final phase. In VMemDirect, in contrast, paging was suppressed because the memory of the device emulator was not managed by private virtual memory. The downtime in VMemDirect was comparable to those in split migration and the ideal migration.

5.2. Performance of Migrating an Active VM

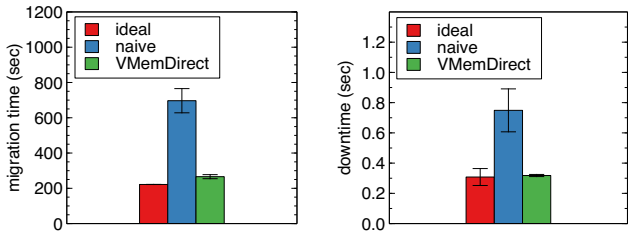
To examine how memory updates affect migration performance, we measured the time needed to migrate an active VM. We assigned 120 GB of memory to a VM and ran the benchmark that modified 60 GB of memory in it. As shown in Fig. 5(a), the migration time of the active VM was longer than that of the idle VM in any type of migration. However, the migration time was 2.2x longer in the naive migration, while that was 1.5x longer in VMemDirect. As a result, VMemDirect was 58% faster than the naive migration. This means that VMemDirect could successfully transfer updated memory to physical memory and reduce paging.

For split migration and the ideal migration, the migration time of the active VM was also 1.5x longer than that of the idle VM. The increase in migration time was almost the same between VMemDirect and split migration, but VMem-



(a) Migration time (b) Downtime

Figure 5. The migration performance of an active VM.



(a) Migration time (b) Downtime

Figure 6. The performance of memory-devirtualizing VM migration.

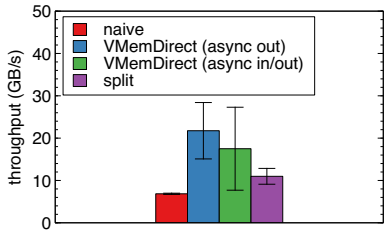


Figure 7. The execution time of the memory benchmark.

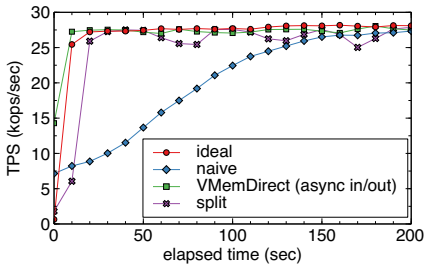


Figure 8. The memcached performance with YCSB.

Direct was 11% faster than split migration. Compared with the ideal migration, VMemDirect was only 4.1% slower.

Next, we measured the downtime during the migration. As shown in Fig. 5(b), the downtime increased by 82 ms in the naive migration because retransferred memory data caused paging in the final phase. In VMemDirect, the downtime increased only by 11 ms. As a result, the downtime difference was 545 ms. In contrast, split migration and the ideal migration decreased the downtime by 23 ms and 78 ms, respectively.

5.3. Performance of Devirtualizing Migration

We measured the time needed for memory-devirtualizing VM migration when using an idle VM with 120 GB of memory. As shown in Fig. 6(a), the naive migration took much longer because paging occurred to transfer memory data in swap space. VMemDirect was 2.6x faster than the naive migration thanks to direct memory transfer from private swap space. It was only 17% slower than the ideal migration. Fig. 6(b) shows the downtime during memory-devirtualizing VM migration. The downtime in the naive migration was much longer because paging occurred at the source host in the final phase. VMemDirect could reduce the downtime by 431 ms. This downtime was only 10 ms longer than that in the ideal migration.

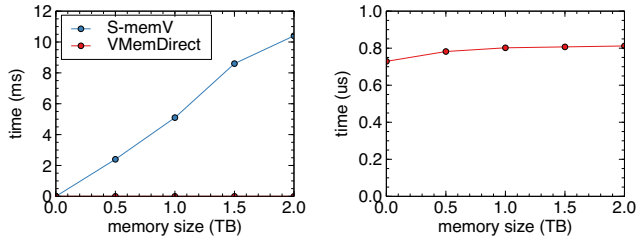
5.4. Performance of Private Virtual Memory

We examined the performance of private virtual memory after VM migration. First, we migrated a VM with 120 GB of memory and ran a memory benchmark in the VM. This benchmark caused excessive paging because the total working-set size exceeded the size of the physical memory assigned to private virtual memory. As shown in Fig. 7, VMemDirect was 3.2x faster than the naive migration. When VMemDirect performed not only page-outs but also page-ins asynchronously, the performance was degraded by 20%. This is because the benchmark accessed memory sequentially and asynchronous page-ins resulted in delay. The performance after split migration was worse than VMemDirect due to the network overhead in remote paging.

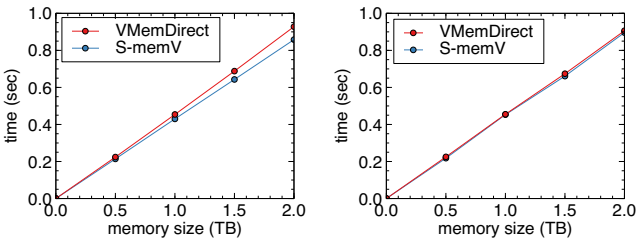
Next, we ran memcached in a VM and migrated the VM. We assigned 12 GB of memory to the VM and allocated 5 GB of memory to memcached. The total working-set size was less than the size of the physical memory assigned to private virtual memory. We measured the performance of memcached with the YCSB benchmark. As shown in Fig. 8, the performance after the naive migration was restored gradually. This is because memory data used by memcached was stored in swap space. In contrast, VMemDirect could restore as high performance as after the ideal migration only in 10 seconds. This is thanks to memory access prediction. When we used split migration, it took longer than VMemDirect, and the performance degradation was 7% on average.

5.5. Performance of LRU Management

We compared the performance of the LRU management in VMemDirect with that in S-memV [1] developed for split migration. We measured the time for a page-out decision, a VM's memory division, and the update of LRU management data. To examine the scalability over the size of physical memory, we emulated these three functions. Fig. 9(a) shows the time for a page-out decision, and Fig. 9(b) magnifies only the results for VMemDirect. The decision time in VMemDirect was much shorter than in S-memV. Also, the scalability in VMemDirect was higher because the decision



(a) Page-out decision (b) Page-out (VMemDirect)



(c) Memory division (d) LRU update

Figure 9. The performance of LRU management.

time was only slightly proportional to the memory size of a VM.

Fig. 9(c) shows the time for dividing the memory of a VM. The division time in VMemDirect was up to 8.1% longer than in S-memV. This is because the chunk queues consist of linked lists and the hit ratio of the CPU cache decreased. However, this extra overhead is negligible compared with a long migration time. Fig. 9(d) shows the time for updating LRU management data, i.e., the chunk queues in VMemDirect and the memory access history used in S-memV. The update time in VMemDirect was almost the same as in S-memV. This overhead came from the reconstruction of the linked lists in the chunk queues.

6. Related Work

vMotion provides two different migration methods in terms of swap space [4]: unshared- and shared-swap vMotion. Shared-swap vMotion stores a swap file in shared storage. Upon VM migration, the destination host uses temporary swap space and integrates it into a shared one later. This method always needs slow network paging. Like shared-swap vMotion, Agile live migration [5] locates swap space for each VM in the network. It aggressively pages out memory data to swap space on VM migration. This method can further improve migration performance, but the paging overhead is much larger.

FlashVM [6] is virtual memory using paging based on SSDs. It pages out more memory pages at once than when using HDDs. Since random reads of SSDs are fast, FlashVM prefetches more useful pages to reduce page faults. In addition, it adjusts the rate of writeback to SSDs to reduce the latency of page faults. This prefetching technique can improve the performance of private virtual memory in VMemDirect.

VSwapper [7] improves the performance of VMs using virtual memory. It monitors storage I/O and prevents unmodified pages from being written to swap space on page-outs. Also, it stores data written to paged-out pages in a temporal buffer and prevents data from being read from swap space if the entire page is written. These optimizations achieve 10x performance improvement. They can also be applied to private virtual memory in VMemDirect.

Multi-generational LRU (MGLRU) [8] uses multiple lists called generations. An accessed page is added to the youngest generation. An idle page is moved to the next older generation. An evicted page is selected from the oldest generation. This is similar to our chunk queues, but VMemDirect moves an accessed memory chunk to an intermediate queue, considering the previous access history.

7. Conclusion

This paper proposed VMemDirect for efficient memory-virtualizing VM migration. VMemDirect provides private virtual memory with private swap space on an NVMe SSD for each VM. It directly transfers memory data to either physical memory or private swap space. Our experimental results show that VMemDirect could improve the performance of VM migration and migrated VMs dramatically. Our future work is to compare the performance of VMemDirect using various types of SSDs. We are also planning to create private swap space on Intel Optane DC persistent memory.

Acknowledgment

This work was partially supported by JST, CREST Grant Number JPMJCR21M4, Japan. These research results were partially obtained from the commissioned research (No.05501) by National Institute of Information and Communications Technology (NICT), Japan.

References

[1] M. Suetake, T. Kashiwagi, H. Kizu, and K. Kourai, "S-memV: Split Migration of Large-memory Virtual Machines in IaaS Clouds," in *Proc. Int. Conf. Cloud Computing*, 2018, pp. 285–293.

[2] Y. Muraoka and K. Kourai, "Efficient Migration of Large-memory VMs Using Private Virtual Memory," in *Proc. Int. Workshop on Information Network Design*, 2019, pp. 380–389.

[3] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker, "Can Far Memory Improve Job Throughput?" in *Proc. European Conf. Computer Systems*, 2020.

[4] I. Banerjee, P. Moltmann, K. Tati, and R. Venkatasubramanian, "VMware ESX Memory Resource Management: Swap," *VMware Technical Journal*, vol. 3, no. 1, pp. 48–56, 2014.

[5] U. Deshpande, D. Chan, T. Guh, J. Edouard, K. Gopalan, and N. Bila, "Agile Live Migration of Virtual Machines," in *Proc. Int. Parallel and Distributed Processing Symp.*, 2016.

[6] M. Saxena and M. Swift, "FlashVM: Virtual Memory Management on Flash," in *Proc. Annual Technical Conf.*, 2010.

[7] N. Amit, D. Tsafrir, and A. Schuster, "VSwapper: A Memory Swapper for Virtualized Environments," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 2014, pp. 349–366.

[8] J. Corbet, "The Multi-generational LRU," https://lwn.net/Articles/851184/, 2021.