九州工業大学学術機関リポジトリ
Kyutacar
Kyushu Institute of Technology Academic Repository

Optimizing VMs across Multiple Hosts with Transparent and Consistent Tracking of Unused Memory

著者	Tauchi Soichiro, Kourai Kenichi, Lukman Ab. Rahim
journal or publication title	2021 IEEE 14th International Conference on Cloud Computing (CLOUD)
page range	467-477
year	2021-11-13
URL	http://hdl.handle.net/10228/00008652

doi: https://doi.org/10.1109/CLOUD53861.2021.00061

Optimizing VMs across Multiple Hosts with Transparent and Consistent Tracking of Unused Memory

Soichiro Tauchi
Kyushu Institute of Technology
tauchi@ksl.ci.kyutech.ac.jp

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

Lukman Ab. Rahim
Universiti Teknologi Petronas
lukmanrahim@utp.edu.my

Abstract*—Recently, Infrastructure-as-a-Service (IaaS) clouds provide virtual machines (VMs) with a large amount of memory. To make the migration of such large-memory VMs flexible, split migration has been proposed. It divides the memory of a VM into smaller pieces and transfers them to multiple destination hosts. After the migration, the VM runs across multiple hosts and its memory data is exchanged between hosts by remote paging. There is often unused memory in a large-memory VM, but data of even unused memory is transferred via the network. This paper proposes *FCtrans* to achieve efficient split migration and remote paging by considering unused memory. FCtrans avoids transferring data of unused memory to destination hosts on split migration. Similarly, it does not perform remote paging for unused memory and immediately continues the execution of the VM. To enable this, FCtrans keeps track of the memory usage of a VM after starting split migration. In addition, it transparently and consistently reclaims memory released after used by the guest operating system using VM introspection and deals with it as unused. We have implemented FCtrans in KVM and conducted experiments using a VM with 352 GB of memory on the StarBED testbed. It is shown that split migration became up to 29x faster and the memory access performance of a VM across multiple hosts improved by up to 85%.

1. Introduction

Recently, Infrastructure-as-a-Service (IaaS) clouds provide virtual machines (VMs) with a large amount of memory, e.g., instances with 24 TB of memory in Amazon EC2 [1]. Such large-memory VMs are used for in-memory databases [2], [3] and big data analysis [4], [5]. Like regular-sized VMs, they should be migrated without stopping the execution on host maintenance. VM migration transfers the state of a VM such as memory to a destination host and continues its execution. As such, the destination host has to have sufficient memory to accommodate the entire memory of a VM. For large-memory VMs, it is not cost-efficient or flexible even for cloud providers to always preserve hosts with a large amount of available memory.

Split migration [6], [7] enables migrating a large-memory VM to the main host and sub-hosts by dividing its memory. After split migration, the migrated VM runs across multiple hosts and is called a *split-memory VM*. It exchanges memory data between two hosts by performing *remote paging*. A *remote page-in* transfers data from a sub-host to the main host, while a *remote page-out* transfers it from the main host to the sub-host. When a VM has a large amount of memory, there are often unused memory regions in the VM. However, split migration transfers data even in unused memory regions to the destination hosts. This results in a long migration time. Similarly, remote paging transfers data between hosts even for unused memory. This degrades the performance of a split-memory VM. Various techniques have been proposed to avoid transferring data for unused memory [8]–[11], but they focus only on the traditional one-to-one migration. In addition, they are inefficient or not transparent to VMs.

This paper proposes *FCtrans* to archive efficient split migration and remote paging by considering unused memory. Upon split migration, FCtrans avoids transferring data in unused memory regions to reduce the migration time. It transfers used memory to the main host as much as possible to suppress remote paging after the migration. When a split-memory VM starts to access unused memory existing in sub-hosts, FCtrans does not transfer the data by a remote page-in. Instead, it performs a *local page-in* and allocates memory reserved in the main host to the VM. This enables the execution of the split-memory VM to be immediately continued. In addition, FCtrans does not perform remote page-outs as long as the reserved memory remains in the main host. As a result, the performance of the split-memory VM is improved.

To identify unused memory regions in a VM, FCtrans keeps track of the memory usage of a VM by detecting the first access to unused memory with traps. To reduce this overhead, it starts this tracking at the beginning of split migration. In addition, FCtrans deals with memory released after used by the guest operating system (OS) as unused again. To transparently enable this without modifying the guest OS, it obtains information on the memory usage of the guest OS using *VM introspection (VMI)* [12]. VMI allows FCtrans to analyze kernel data structures in the memory of a VM from the outside of the VM. FCtrans consistently reclaims free memory in the guest OS without stopping the VM and deallocates that memory from the VM. Then, it merges information on the memory usage of both the VM and the guest OS.

We have implemented FCtrans in QEMU-KVM supporting split migration and remote paging. To show the performance improvement by FCtrans, we compared FCtrans with the original split migration and remote paging. We used a VM with up to 352 GB of memory in the NICT testbed named StarBED [13]. Consequently, FCtrans could reduce the migration time by up to 97% and increase the memory access performance of a split-memory VM by up to 85%. The reclamation of free memory in the guest OS was up to 62% faster than the traditional method using memory ballooning [14].

The organization of this paper is as follows. Section 2 describes split migration and remote paging and their issues. Section 3 proposes FCtrans for avoiding the transfers of data for unused memory. Section 4 explains the implementation of FCtrans and Section 5 shows our experimental results. Section 6 describes related work and Section 7 concludes this paper.

2. VMs across Multiple Hosts

2.1. Split Migration

VM migration moves a VM running in one host to another host. Using this technique, any hosts running VMs can be maintained without service disruption. VM migration is also used to consolidate VMs into a small number of hosts for power saving and deconsolidate VMs from overloaded hosts for load balancing. It transfers the memory of a target VM to a new VM created at the destination host. Then, it retransfers the memory modified during the transfers and stops the VM if the amount of memory to be retransferred becomes small enough. In the final phase, it transfers the rest of the modified memory and the state of the VM core such as virtual CPUs and virtual devices. Finally, it resumes the execution of the VM at the destination host.

VM migration requires a larger amount of available memory than the memory size of a migrated VM at the destination host. Recently, the memory size of a VM is increasing for in-memory databases and big data analysis [1]. As a result, it is becoming difficult to find an appropriate destination host for such a large-memory VM. Even in public clouds, it would not be cost-effective or flexible to always preserve many hosts with a large amount of memory in preparation for large-scale maintenance. Private clouds may not be able to prepare large hosts as the destination of occasional VM migration. Such hosts are expensive for small or medium-sized private clouds and require a measurable amount of power. If there is no appropriate host, a VM cannot be migrated and the users cannot use services provided by the VM during host maintenance.

For flexible migration of large-memory VMs without large hosts, *split migration* has been proposed [6], [7]. As illustrated in Fig. 1, it divides the memory of a VM into small pieces and transfers them to multiple smaller hosts. One of the destination hosts is called the main host and

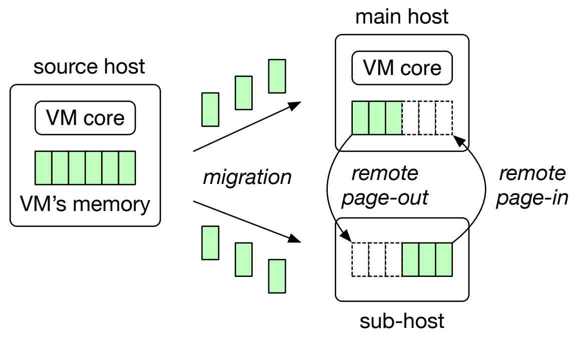


Figure 1: Split migration.

runs the VM core, while the others are called sub-hosts. Split migration transfers the state of the VM core and likely accessed memory to the main host and the rest of the memory to the sub-hosts. Such memory access prediction is performed using the least recently used (LRU) algorithm on the basis of the memory access history of the VM.

After split migration, the migrated VM runs across multiple hosts. This VM is called a *split-memory VM*. Since the memory of the VM is distributed, a split-memory VM runs by performing remote paging between the main host and one of the sub-hosts. When the VM core at the main host requires the memory existing in a sub-host, that memory is moved from the sub-host to the main host, which is called a *remote page-in*. In exchange, the most unlikely accessed memory is moved from the main host to that sub-host, which is called a *remote page-out*. Since likely accessed memory has been transferred to the main host in advance at the migration time, the frequency of remote paging is suppressed after split migration.

2.2. Unnecessary Network Transfers

As such, the memory of a VM is transferred via the network when necessary, but there are often unused regions in the memory. For example, most of the memory is unused just after the guest OS boots in a VM. Even used memory regions become unused again if the guest OS releases them after the termination of applications. In fact, it is reported that only 10% of the memory is used on average for VMs running web applications in clouds [15]. It is also revealed that VMs used for scientific computing have a large amount of unused memory [16]. Similarly, only about 50% of the memory is used in the clusters of Google and Alibaba [17].

Split migration transfers even unnecessary data contained in the unused memory of a VM to the destination hosts. Therefore, it takes time to migrate a VM in proportion to its memory size. After split migration, remote paging transfers the memory data from a sub-host for a remote page-in even when a split-memory VM starts to access unused memory at that sub-host. Then, it selects unused memory at the main host if any on the basis of LRU and transfers it to the sub-host for a remote page-out. Such unnecessary remote paging degrades the performance of the

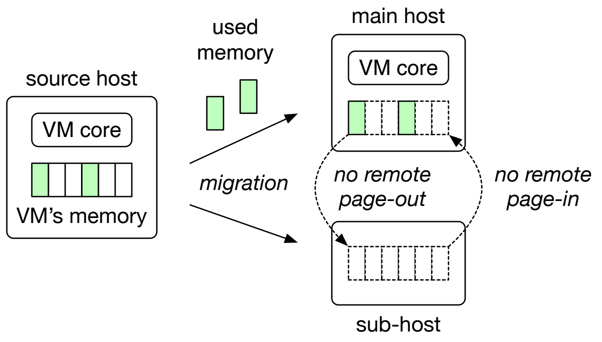


Figure 2: The optimization of network transfers in FCtrans.

split-memory VM. When the VM accesses the memory at a sub-host, its virtual CPU is suspended until a remote page-in completes. Although remote page-outs can be done asynchronously after the virtual CPU is resumed, they largely affect the performance of the VM and the network.

For the traditional one-to-one migration, various techniques have been proposed to avoid transferring data of unused memory [8]–[11]. However, they are inefficient or not transparent to VMs. One category of the techniques obtains information on unused memory at the VM level. QEMU [18] scans the entire memory of a VM to identify unused memory, which is filled by zero, but this overhead offsets the improvement of migration performance by fast networks such as 10 GbE. Another technique [8] suffers from the overhead of always detecting modified memory since a VM boot. In addition, it cannot detect that memory regions are no longer used. The other category obtains information on unused memory at the guest-OS level [9]–[11]. Using information of the guest OS is more efficient, but it needs to modify the guest OS. This should be avoided in clouds.

3. FCtrans

This paper proposes FCtrans, which optimizes network transfers in split migration and remote paging.

3.1. Optimization of Network Transfers

FCtrans achieves efficient split migration and remote paging by considering unused memory in VMs. It transparently keeps track of the memory usage of a VM without modifying its guest OS. Then, it avoids transferring the data of unused memory to destination hosts during split migration, as illustrated in Fig. 2. Unlike the original split migration, it transfers used memory to the destination main host as much as possible. As a result, it can suppress remote paging after the migration. In addition, no destination hosts allocate physical memory for unused memory to flexibly handle it later.

Even when remote paging is required after split migration, FCtrans does not perform unnecessary network transfers for unused memory. When a VM starts to access unused memory at a sub-host, FCtrans performs a *local page-in* without a network transfer, instead of a remote page-in. In other words, it just allocates memory reserved in the main host to the VM. As a result, the VM does not need to wait for the completion of a remote page-in and then is resumed immediately. FCtrans performs a remote page-in only for used memory. In addition, as long as there is a sufficient amount of available memory at the main host, FCtrans does not perform remote page-outs. It performs remote page-outs as traditional only if this condition is not met. This can reduce the overhead caused by remote page-outs.

3.2. Transparent and Consistent Memory Tracking

To keep track of the memory usage of a VM, FCtrans detects both changes from unused to used memory and from used to unused memory. When a VM starts to use an unused memory region, that access has to be detected immediately. If this detection is delayed, FCtrans would apply the optimization of avoiding network transfers to already used memory in split migration and remote paging. This results in losing the data of used memory. Therefore, FCtrans immediately detects the change from unused to used memory by using traps. Upon VM creation, it configures all the memory regions as unused. When the VM accesses a memory region for the first time, FCtrans handles a caused trap and updates that region as used in the memory usage of the VM.

Since FCtrans needs information on unused memory only during and after split migration, it starts to track unused memory just after starting split migration. This can suppress performance degradation until split migration is applied to a VM. As a result, FCtrans cannot know the memory usage until split migration. To address this dilemma, it obtains the memory usage at once just before starting split migration using the method described later. After split migration, it continues this tracking of unused memory. If a split-memory VM runs at one host again after merge migration [19], FCtrans stops tracking the memory usage.

On the other hand, it is difficult to detect that a memory region is changed from used to unused in a VM. This is because FCtrans cannot know that the memory region is no longer used. For example, even if a VM does not access a memory region for a long time, it is not guaranteed that the VM does not use that region in the future. This means that FCtrans cannot change any memory regions back to unused ones once memory regions become used. In contrast, the guest OS in the VM knows whether the memory region is in use or not. It manages memory regions that are used once but released as free memory. From the viewpoint of a VM, such free memory is still in use.

Therefore, FCtrans periodically merges information on the memory usage of both a VM and its guest OS, as illustrated in Fig. 3. This cannot immediately detect the change from used to unused memory, but it is tolerable to detect that with some delay. Even if FCtrans transfers already unused memory without correctly applying the optimization of network transfers, that does not cause a consistency issue

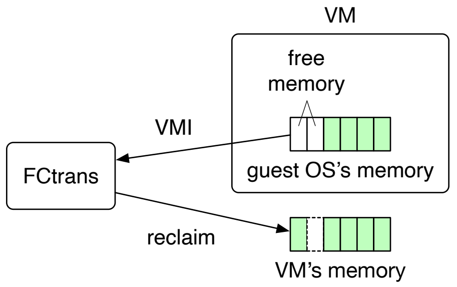


Figure 3: The integration of the memory usage of a VM and the guest OS.

although it can degrade performance. To obtain information on the memory usage of the guest OS without modifying the guest OS, FCtrans uses VM introspection (VMI) [12]. VMI is a technique for analyzing the data structure of the guest OS in the memory from the outside of a VM. If a memory region becomes free, FCtrans reclaims that region. Specifically, it deallocates physical memory from the VM and changes that memory region back to unused.

FCtrans consistently performs this reclamation of free memory without stopping the guest OS. Since VMI is asynchronously applied to a running VM, a memory region of the guest OS might become in use at the time of reclamation even if it is free at the time of check. To deal with this race condition, FCtrans speculatively deallocates physical memory in a memory region from the VM when the region is free. At the same time, it atomically saves the contents of that region in preparation for the failure of this speculation. Next, it checks the region again using VMI. If the region is still free, FCtrans completes the reclamation of that free memory. Otherwise, it rolls back the speculatively performed memory deallocation because the memory might be reused and modified during the reclamation process. FCtrans allocates a new memory to the VM and restores the contents from the saved data. Since FCtrans defers access after the memory deallocation if any, such access is correctly reflected in the memory region after the rollback.

FCtrans also uses this reclamation mechanism to obtain the latest memory usage at once when it starts split migration. Since it does not know the memory usage until then, it first assumes that all the memory regions of a VM are used. Then, it reclaims free memory in the guest OS and knows unused memory regions.

4. Implementation

We have implemented FCtrans in QEMU-KVM 2.11.2 supporting split migration and remote paging. Our target of the guest OS is Linux 4.14, but FCtrans can easily support other versions of Linux.

4.1. Detection of Used Memory

To detect access to unused memory by using page faults, FCtrans uses the userfaultfd mechanism provided by the host Linux OS, as illustrated in Fig. 4. It first registers all the memory regions assigned to a VM in QEMU-KVM to userfaultfd. Since FCtrans does not allocate physical memory pages to the regions at the boot time of a VM, a page fault occurs when the VM accesses an unused page for the first time. At this time, that fault is notified to QEMU-KVM by userfaultfd. Before split migration, FCtrans allocates a physical page filled by zero to that faulting page using the system call for userfaultfd. For a split-memory VM, it performs a local page-in if the accessed page is unused and allocates the memory reserved in the main host to that region. To reduce the overhead of this detection of used memory, FCtrans allocates 256 contiguous physical pages, called a memory chunk, including the faulting page at once in the current implementation. After that, no page fault occurs for those pages.

In addition, FCtrans records these allocated pages as used. To manage whether each memory page of a VM is used or unused, FCtrans uses a bitmap called a *usage bitmap*. This bitmap consists of as many bits as the number of pages assigned to a VM. FCtrans initializes all the bits to zero, which means unused. When it detects first access to a page, it sets the corresponding bit to one.

4.2. Consistent Reclamation of Free Memory

To deal with free memory in the guest OS as unused memory in a VM, FCtrans periodically finds free memory using VMI and reclaims it. Linux manages physical memory using the buddy system and allocates physical pages in a power of two. FCtrans obtains information on the memory usage of the guest OS from the page structure. The page structure manages each physical page and contains information on whether the page is free or not. If the page is the head of a free memory region, the page structure also contains the number of pages included in that region. Linux manages an array of page structures, whose length is equal to the number of physical pages. FCtrans traverses this array from page frame number 0 and finds the head page of a free memory region. Then, it changes the pages included in that region back to unused.

To transparently obtain such information from a VM, we have developed a VMI framework embedded in QEMU-KVM. The framework enables writing code for VMI using the Linux header files. For example, FCtrans can check

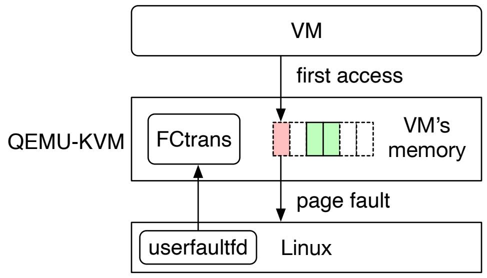


Figure 4: The detection of used memory with userfaultfd.

Algorithm 1 Consistent reclamation of free memory.

```
Input: page
 1: if (page is allocated but free) then
 2:     acquire a lock
 3:     if (page exists in the main host) then
 4:         save and deallocate page
 5:         if (page is free) then
 6:             clear the usage bitmap
 7:         else
 8:             roll back page deallocation
 9:         end if
10:     else if (page exists in a sub-host) then
11:         if (page is free) then
12:             clear the usage bitmap
13:             send a release request
14:         end if
15:     end if
16:     release a lock
17: end if
```

whether a page is free or not by applying the PageBuddy macro to the page structure. It can obtain the number of pages included in a free memory region using the page_order macro. Then, this framework generates intermediate representation from the code and transforms that using LLVM so that the code accesses the memory of a VM when necessary. Specifically, it inserts code that translates virtual addresses of OS data into physical ones before all the load instructions.

FCtrans consistently reclaims free memory in the running guest OS using VMI. Algorithm 1 is the pseudo code for consistent reclamation of free memory. If FCtrans detects that a page is allocated to a VM but free in the guest OS, it acquires a lock for the memory chunk including that page (lines 1–2). Since remote paging and memory allocation are done in a chunk granularity, this lock guarantees that the target page is not paged in from a sub-host, not paged out to a sub-host, or not newly allocated. This lock is fine-grained, so that remote paging and memory allocation can be done for the other memory chunks in parallel to memory reclamation of the target page.

If the target page exists in the main host, FCtrans speculatively deallocates that page from the VM (lines 3–4). At the same time, it saves the data of the page in preparation for the conflict of memory reclamation. These two operations are atomically done by using the userfaultfd feature extended for remote page-outs. After the speculative memory deallocation, FCtrans can detect any access to that page because a page fault occurs. Since a page fault is handled in QEMU-KVM using userfaultfd, FCtrans defers that fault handling until memory reclamation completes by the chunk lock acquired at the beginning. This guarantees that the being reclaimed page is not modified after the speculative memory deallocation. Fig. 5 shows how a running VM is synchronized with VMI by this access protection.

After the memory deallocation, FCtrans re-checks

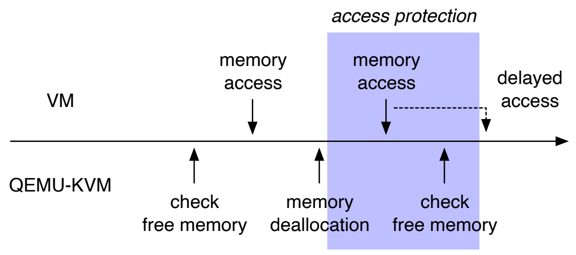


Figure 5: Synchronization of a running VM with VMI.

whether the page is still free or not. Since the guest OS is running, free memory can be reused during memory reclamation. If the page is still free, FCtrans sets the corresponding bit to zero in the usage bitmap and completes the reclamation of that page (lines 5–6). Otherwise, it aborts the reclamation of that page and rolls back the speculative memory deallocation (lines 7–8). Specifically, it allocates a physical page again and writes the saved data to that page using userfaultfd. The memory data is consistently restored because the saved data contains all the modifications to the page before the memory deallocation. Pending modification after the memory deallocation is applied after the rollback because the page fault caused by that is handled after the chunk lock is released (line 16).

If the target page exists in a sub-host, FCtrans re-checks whether the page is still free without speculative memory deallocation (line 10–11). Since remote page-ins are disabled by the chunk lock, it is not necessary to detect access to the page. When the page is still free, FCtrans sets the corresponding bit to zero in the usage bitmap (lines 12–13). In addition, it sends a request for memory release to the sub-host but does not wait for the completion. When the sub-host receives that request, it removes the corresponding entry from the table for memory management and releases the memory allocated for that page. When the page becomes in use at the second check, FCtrans does nothing.

4.3. Optimization of Split Migration

When FCtrans starts split migration of a VM, it first registers all the memory regions of the VM to userfaultfd. After that, it can detect the change from unused to used memory pages. Next, it sets all the bits in the usage bitmap to one, which means a used page. Then, it reclaims free memory in the guest OS using VMI. As a result, it can obtain the usage bitmap including the latest memory usage.

FCtrans examines the usage bitmap whenever it attempts to transfer the data of each page. It transfers the data only if the corresponding bit is one. If the destination is the main host, FCtrans transfers memory data with its metadata. If the destination is a sub-host, FCtrans transfers memory data to the sub-host and its metadata to the main host. On the other hand, it transfers neither memory data nor its metadata if the corresponding bit of the usage bitmap is zero, which means an unused page. In other words, there are unused pages nowhere after our optimized split migration. FCtrans

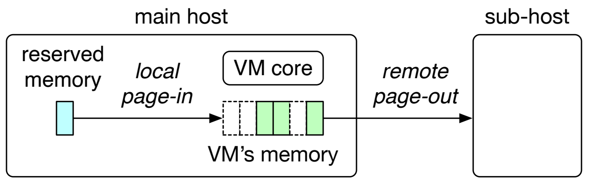


Figure 6: The optimization of remote paging using local page-ins.

determines a destination host per memory chunk on the basis of memory access history, which is obtained by periodically checking access bits in the extended page tables (EPT) of a VM. It transfers recently accessed and currently used memory to the main host as much as possible.

The destination main host re-constructs the usage bitmap on the basis of received metadata of used pages. This can be done without directly transferring the usage bitmap from the source host. When FCtrans starts split migration, it allocates a new usage bitmap at the destination main host and initializes all the bits to zero. Whenever the destination main host receives page metadata, FCtrans sets the corresponding bit to one. As a result, FCtrans can deal with the pages that are unused at the source host as unused at the destination hosts as well.

4.4. Optimization of Remote Paging

After split migration, there are three types of pages: used pages existing in the main host, used pages existing in sub-hosts, and unused pages existing nowhere. When a VM accesses the last two types of pages, a page fault occurs. At this time, FCtrans examines the usage bitmap to distinguish the two types. If the corresponding bit is one, which means a used page existing in a sub-host, FCtrans performs a remote page-in as usual. Otherwise, FCtrans does not perform it because the page does not exist at any sub-hosts. Instead, it performs a local page-in and just allocates the memory reserved in the main host to the VM using userfaultfd, as illustrated in Fig. 6. Then, it sets the corresponding bit to one in the usage bitmap. Since FCtrans performs remote page-ins for a chunk of 256 pages at once, local page-ins are also performed for the same chunk.

To perform such local page-ins, FCtrans manages the memory reserved for a VM in the main host. The original remote paging keeps a fixed amount of physical memory used for a VM by always performing both remote page-in and page-out. In contrast, FCtrans often uses a smaller amount of physical memory than assigned to a VM. If there is no memory available in the main host, FCtrans repeats remote page-outs by a memory chunk until it can preserve a sufficient amount of reserved memory. Since a chunk can include unused pages, FCtrans cannot always page out a sufficient number of pages for one chunk.

5. Experiments

We conducted several experiments to examine the performance improvement of split migration and remote paging by FCtrans. For comparison, we used a system that supported the original split migration and remote paging. In these experiments, we used three hosts provided by the NICT integrated testbed named StarBED [13]. These hosts consisted of one source host, one destination main host, and one destination sub-host. Each host was equipped with two Intel Xeon E5-2683 v4 processors, 384 GB of memory, and 10 Gigabit Ethernet. We ran Linux 4.18 modified for remote page-outs and QEMU-KVM 2.11.2 modified for FCtrans. We created a VM with 64 virtual CPUs and up to 352 GB of memory. We ran Linux 4.14 in this VM. Upon split migration, we equally divided the memory of the VM into two.

5.1. Performance of Memory Reclamation

We first measured the time needed for reclaiming free memory in the guest OS when a VM ran in one host. This type of memory reclamation is done at the beginning of split migration. In this experiment, we used a VM with 352 GB of memory. We ran a memory benchmark that accessed the specified amount of memory in this VM and then terminated it to change the used memory to free memory. We changed the amount of reclaimed free memory from 8 to 192 GB.

For comparison, we applied memory ballooning [14] to reclaim the free memory of a VM. Originally, memory ballooning is used to change the memory size of a VM with the help of the balloon driver installed in the guest OS. The balloon driver allocates the specified amount of memory from the pool of free memory and returns it to the hypervisor. The returned memory is deallocated from the VM. In this experiment, we used the `virsh setmem` command and first decreased the memory size of the VM by the size of reclaimed free memory. Then, we increased that to the original size so that the guest OS could reuse free memory when necessary.

Fig. 7 shows the time needed for reclaiming a specified amount of free memory. The reclamation time was basically proportional to the amount of reclaimed free memory. Compared with memory ballooning, FCtrans could reduce the reclamation time by 53–62%. It was still faster than memory ballooning only for decreasing the memory size of the VM. It should be noted that the increase in reclamation time was smaller for more than 128 GB of reclaimed free memory. This reason is currently under investigation, but it is promising for large-memory VMs. For memory ballooning, the reclamation time becomes a bit longer because we need additional time for recording the reclaimed free memory as unused.

In addition, we examined the impact of the memory reclamation on a target VM. We ran an in-memory database called memcached [20] in a VM and measured the throughput using the memaslap benchmark [21]. We assigned 30 GB of memory to memcached and accessed 4-KB data. Fig. 8(a)

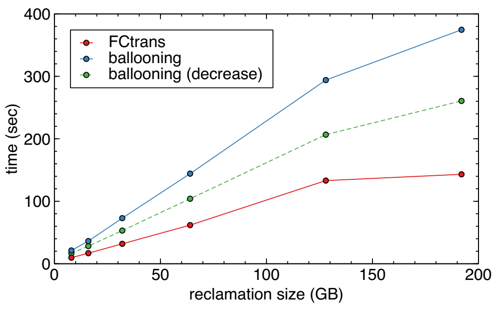
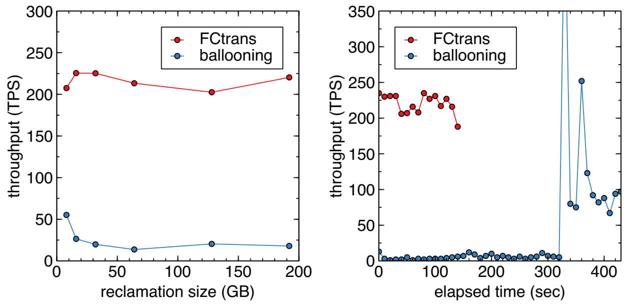


Figure 7: The reclamation time of free memory.

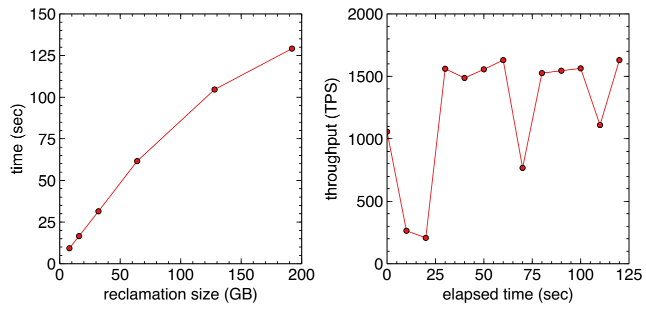


(a) VM performance (b) Changes in throughput

Figure 8: The VM performance during memory reclamation.

shows the average throughput during the memory reclamation. FCtrans could improve the throughput by 3.8–12x, compared with memory ballooning. Specifically, Fig. 8(b) shows the changes in throughput when we reclaimed 192 GB of free memory. For memory ballooning, the average throughput was only 5.2 transactions per second (TPS) during decreasing the memory size of the VM although it was improved to 105 TPS during increasing the memory size. In contrast, FCtrans always kept 220 TPS on average during memory reclamation. This throughput is still lower than during no memory reclamation, but it could be improved if FCtrans reclaims free memory more slowly.

Next, we measured the reclamation time of free memory for a VM across two hosts. This type of memory reclamation is done after split migration. Since memory ballooning was not currently supported for split-memory VMs, we examined the performance only for FCtrans. As in the above experiment, we used a VM with 352 GB of memory and ran the memory benchmark for accessing the specified amount of memory. As shown in Fig. 9(a), the reclamation time was proportional to the amount of reclaimed free memory when the size was less than or equal to 64 GB. In contrast, it became shorter for the reclamation of a larger amount of free memory. This is because FCtrans reclaimed free memory existing not only in the main host but also in the sub-host. For the sub-host, FCtrans just sent requests for memory deallocation asynchronously. Therefore, it was more lightweight to reclaim free memory existing in the sub-host.



(a) Reclamation time (b) Changes in throughput

Figure 9: Reclamation performance for a VM across two hosts.

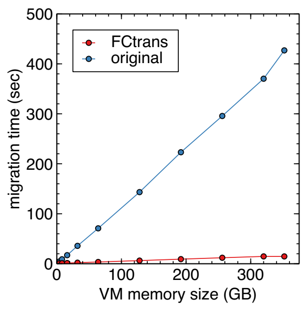
Also, we examined the throughput of memcached in the VM across two hosts during memory reclamation. Fig. 9(b) shows the changes in throughput when we reclaimed 192 GB of free memory. The throughput largely fluctuated because FCtrans reclaimed free memory existing in the sub-host, which was more lightweight. In this experiment, FCtrans reclaimed 169 GB of free memory in the main host, while it reclaimed 23 GB of free memory in the sub-host. As a result, the average throughput was 1223 TPS and much higher than memory reclamation for a VM running in a single host.

5.2. Performance of Split Migration

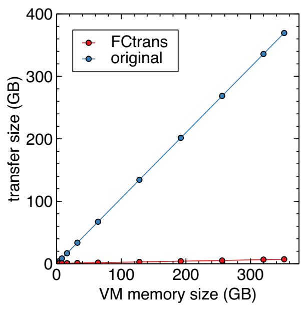
We first performed split migration just after a VM was booted. In this case, most of the memory in the VM was unused. We changed the amount of the memory assigned to the VM from 2 to 352 GB. As shown in Fig. 10(a), the migration time was proportional to the memory size of the VM in both FCtrans and the original split migration. Compared with the original split migration, FCtrans could reduce the migration time by 75–97%. This is because the number of network transfers was largely reduced, as shown in Fig. 10(b). Even for a VM with 352 GB of memory, the size of the transferred data was only 7.2 GB and the migration time was only 14.6 seconds.

It should be noted that the optimization for zero pages in QEMU-KVM can be used to avoid network transfers of unused memory. A zero page is a page filled by zero and an unused page becomes a zero page when it is first accessed for VM migration. Since it is not currently applicable to split migration, we measured the time for one-to-one migration of a VM with 256 GB of memory. With this optimization, however, the migration time was reduced only by 5.9%. This is because the detection of zero pages is heavyweight.

Next, we performed split migration after we ran the memory benchmark that accessed the specified amount of memory in a VM. We used a VM with 352 GB of memory and changed the amount of used memory from 2 to 320 GB. As shown in Fig. 11, the migration time was almost constant



(a) Migration time



(b) Network transfers

Figure 10: The migration time of a VM just after the boot.

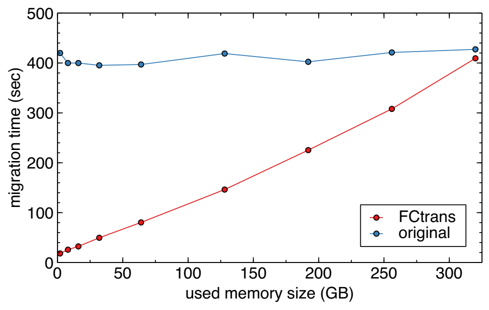


Figure 11: The migration time of a VM with used memory.

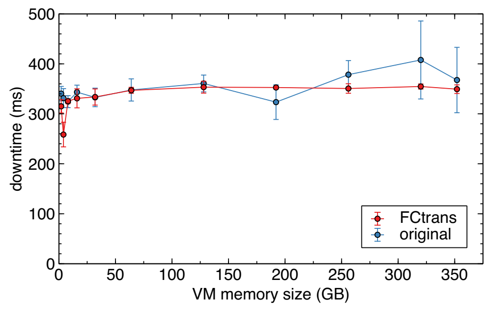


Figure 12: The downtime during split migration.

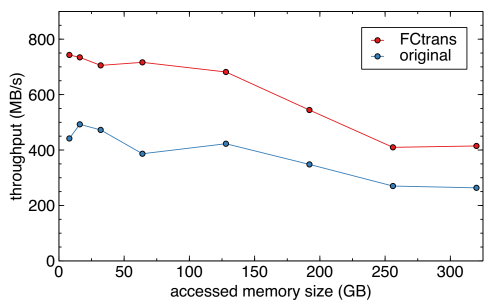


Figure 13: The performance of the memory benchmark.

in the original split migration. However, it was proportional to the amount of used memory in FCtrans because FCtrans transferred only used memory. FCtrans could reduce the migration time by 4.2–96%.

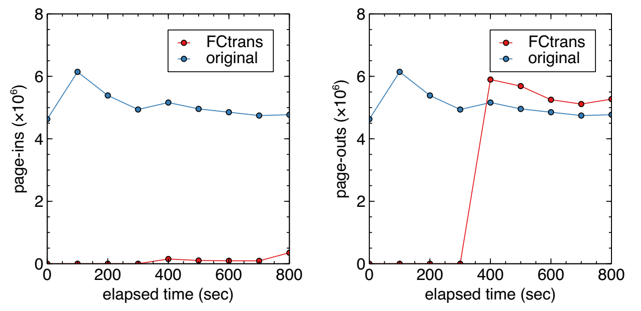
Finally, we measured the downtime during split migration. The downtime is the time between stopping a VM at the source host and resuming it at the destination hosts. We changed the amount of memory assigned to the VM. As shown in Fig. 12, the downtime in FCtrans was 334 ms on average and was almost the same as that in the original split migration. Note that QEMU-KVM stops a VM when it estimates that the rest of the memory can be transferred in 300 ms. After that, it transfers that memory data and the state of virtual CPUs and virtual devices. The optimization of avoiding data transfer of unused memory in FCtrans almost did not affect this estimation by QEMU-KVM.

5.3. Performance of a Split-memory VM

We examined the performance of accessing unused memory in a split-memory VM across two hosts. We first performed split migration of a VM just after the boot. Then, we ran the memory benchmark and accessed the specified amount of unused memory in the split-memory VM. We used a VM with 352 GB of memory and changed the amount of accessed memory from 8 to 320 GB. Fig. 13 shows the throughput of this benchmark in FCtrans and the original split-memory VM. FCtrans could improve the memory access performance by 49–85% thanks to the optimization of remote paging. As the amount of accessed memory increased, the performance degraded more largely in FCtrans. This is because the working set size exceeded the amount of memory existing in the main host and remote page-outs occurred even in FCtrans.

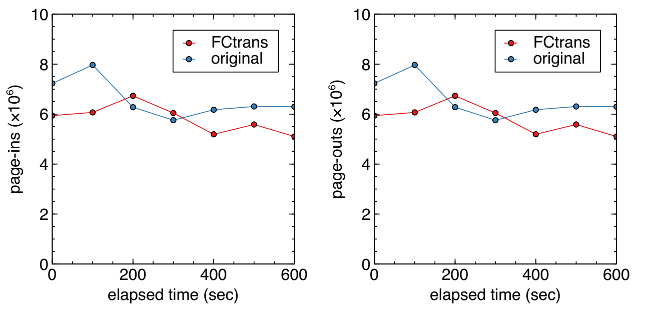
To examine the occurrence of remote paging, we measured the numbers of remote page-ins and page-outs caused by the memory benchmark. Fig. 14 shows the changes while the benchmark accessed 256 GB of unused memory. FCtrans could successfully reduce remote page-ins by 98%. It performed only local page-ins whenever the benchmark accessed unused memory. In contrast, the original split-memory VM needed remote page-ins whenever accessed memory existed in the sub-host. Also, FCtrans could reduce remote page-outs by 40% on average. The number of remote page-outs was zero for the first 300 seconds, while it suddenly increased after that. Even in FCtrans, remote page-outs were needed to preserve memory used for local page-ins after reserved memory ran out at the main host. The reason why the number was larger than the original split-memory VM is that the VM could access memory faster in FCtrans.

Next, we measured the time needed for accessing used memory, which was first accessed by the above memory benchmark. This second-time access was also 52% faster in FCtrans when the benchmark accessed 256 GB of used memory. To inspect the reason, we measured the numbers of remote page-ins and page-outs. As shown in Fig. 15, FCtrans could reduce the numbers of remote page-ins and



(a) Remote page-in (b) Remote page-out

Figure 14: Remote paging during the memory benchmark for 256 GB of unused memory.



(a) Remote page-in (b) Remote page-out

Figure 15: Remote paging during the second-time access to 256 GB of used memory.

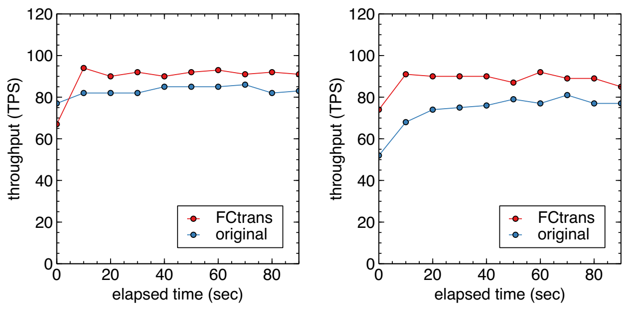
page-outs by 10%, respectively.

5.4. Performance of a Real Application

To examine the performance improvement using a real application, we ran memcached in a split-memory VM across two hosts. Then, we measured the throughput using the memaslap benchmark. This benchmark sent requests to 1-MB data using 32 threads. We used a VM with 352 GB of memory and set 100 GB of data to memcached in advance. After we performed split migration of the VM, the main host had about 70 GB of memory reserved for the VM.

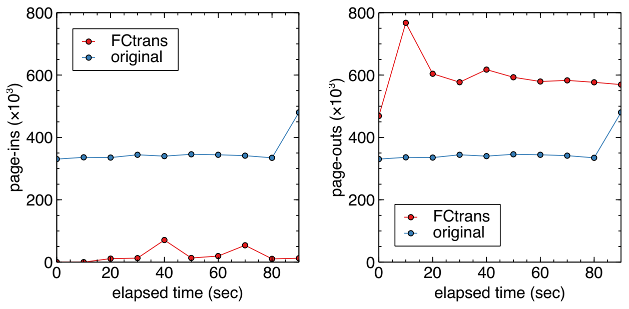
Fig. 16(a) shows the changes in throughput of memcached. FCtrans could improve the throughput by 7.5% on average. In FCtrans, remote paging did not occur because memcached did not access more than 70 GB of memory in 90 seconds. FCtrans performed only local page-ins. In contrast, the original split-memory VM caused many remote page-ins and page-outs because a larger amount of unused memory existed in the sub-host. About 175 GB of the unused memory existed in the sub-host, while only about 70 GB of that existed in the main host.

Next, we ran memcached together with the memory benchmark accessing 256 GB of unused memory. As shown in Fig. 16(b), FCtrans could keep the throughput similar



(a) memcached only (b) memcached + benchmark

Figure 16: The throughput of memcached.



(a) Remote page-in (b) Remote page-out

Figure 17: Remote paging by memcached with the memory benchmark.

to the above experiment without the memory benchmark. The performance degradation was only 1.7%, while that was 11% in the original split-memory VM. As a result, FCtrans could improve the throughput by 19% on average. Since the amount of used memory exceeded that of memory available in the main host, remote paging occurred even in FCtrans. However, FCtrans performed only a small number of remote page-ins, as shown in Fig. 17(a). Instead, it could perform local page-ins. In contrast, FCtrans increased the number of remote page-outs by 68%, as shown in Fig. 17(b). This is because the memory benchmark was less affected by remote page-ins and could access unused memory faster than in the original split-memory VM.

6. Related Work

For the traditional one-to-one migration of a VM, various techniques have been proposed for avoiding the network transfers of unused memory. QEMU-KVM scans the entire data contained in each page before the network transfer. If a page is unused on this memory scan, physical memory is allocated to the page and the page is filled by zero. For such a zero page, QEMU-KVM sends only one-byte data standing for a zero page. At the destination host, QEMU-KVM does not allocate physical memory for the page. However, this technique causes a large overhead for scanning all the page contents. In addition, physical memory is allocated to unused memory at the source host only for VM migration.

Li et al. have proposed an optimization for avoiding transferring unused memory using dirty page logging in KVM [8]. QEMU-KVM uses a dirty bitmap to control page transfers and sends a page only if the corresponding bit is set. The proposed technique keeps track of memory writes since a VM boot. Since the bit for an unused page is not set in the dirty bitmap, QEMU-KVM does not transfer unused pages. However, this technique needs to always enable dirty page logging and the overhead is not small. In addition, it transfers even pages released after used by the guest OS.

Several systems extend the guest OS to provide information on unused memory to the hypervisor [9], [10]. ME2 [9] scans virtual memory in a VM and then finds pages to which physical memory is not allocated. It sends only one-byte data for such an unused page on VM migration. SonicMigration [10] always stores information on free memory pages of the guest OS in the memory shared with the hypervisor. It does not transfer these pages on VM migration. These systems can deal with free memory in the guest OS as unused. However, the applicability is limited in clouds because it is necessary to modify the guest OS.

The optimization of VM migration using VMI has been also proposed [22], [23]. Like FCtrans, these systems analyze the memory of the guest OS from the outside of a VM and identify whether each page is free or not. However, they do not keep track of the memory usage of a VM, unlike FCtrans. Instead, one previous system [22] obtains the entire memory usage only at the beginning of the migration. Since this information becomes obsolete during a long migration time, performance improvement can be limited for the migration of large-memory VMs. Obsolete information is critical for the optimization of remote page-ins because local page-ins can be wrongly applied to the pages reused after that information is obtained.

The other previous system [23] checks the memory usage on demand. Since split migration needs to divide the memory of a VM by considering unused memory for optimization, the memory usage of all the pages is necessary additionally at the beginning of split migration. Extra VMI for this purpose would increase the overhead for the optimization. To optimize remote paging, this previous system has to check whether a page is free or not on a page fault. VMI for this check causes an extra page fault when a necessary page exists in a sub-host. If such a double fault occurs, the VM could get stuck. Also, it is difficult to find unused memory for remote page-outs because that always needs information on the entire memory usage.

Memory ballooning [14] can be used to reclaim the free memory of the guest OS. The balloon driver installed in the guest OS inflates a balloon by allocating memory from the pool of free memory. Then, it returns the allocated memory to the hypervisor, while the hypervisor deallocates the returned memory from the VM. To enable the guest OS to reuse that free memory, the balloon driver deflates the balloon and returns the released memory to the pool of free memory. This mechanism is consistent because the guest OS is involved, but the overhead of the inflation and deflation is not small, especially for a large-memory VM.

Generalized memory de-duplication [22] identifies free memory pages in the guest OS using VMI and reallocates only one page to all of them. To avoid a race condition, it first write-protects each free page and then shares that page with one specific page if that page is still free and not modified. This technique is similar to the consistent memory reclamation of FCtrans in that one free page is checked twice. However, it basically needs two page-table manipulations, while FCtrans needs only one. In addition, it is difficult to apply this technique to split-memory VMs because the memory to be shared is distributed across hosts.

VSwapper [24] proposes various optimizations to improve the performance of VMs using virtual memory. It is shown that the performance of virtual memory degrades when a VM reads data from a disk to a page and the page is paged out without modification. To prevent this performance degradation, VSwapper monitors disk IO and avoids writing back an unmodified page to a disk by a page-out. Also, it is shown that performance degradation occurs when the entire page is modified after it is paged in. To prevent this, VSwapper temporarily saves writes to a paged-out page in a buffer and avoids reading data from a disk by a page-in. These optimizations can be applied to remote paging.

7. Conclusion

This paper proposes FCtrans to achieve efficient split migration and remote paging by considering unused memory. FCtrans avoids transferring data of unused memory to the destination hosts on split migration and between hosts on remote paging. To efficiently and transparently identify unused memory, it keeps track of used memory since the start of split migration. To reclaim free memory in the guest OS and change it back to unused memory, FCtrans consistently merges information on the memory usage of a VM and its guest OS without stopping the VM. We conducted experiments using a large-memory VM in the StarBED testbed and confirmed that FCtrans could significantly improve the performance of split migration and a split-memory VM.

One of our future work is to further reduce the overhead of the reclamation of free memory in the guest OS using VMI. We need to first reveal the reason for that performance degradation. In addition, it is necessary to support various guest OSes because memory reclamation in FCtrans depends on the guest OS. Another direction is to apply FCtrans to other migration methods such as partial migration of split-memory VMs [19].

Acknowledgements

The research results have been achieved by the "Resilient Edge Cloud Designed Network (19304)," the Commissioned Research of National Institute of Information and Communications Technology (NICT), Japan.

References

[1] Amazon Web Services, Inc., "Amazon EC2 High Memory Instances," https://aws.amazon.com/ec2/instance-types/high-memory/, 2019, accessed 7/20/2021.

[2] SAP SE, "What is SAP HANA? An Unrivaled Data Platform for the Digital Age," https://www.sap.com/products/hana.html, accessed 7/20/2021.

[3] Microsoft Corporation, "SQL Server 2017 on Windows and Linux," https://www.microsoft.com/en-us/sql-server/sql-server-2017, accessed 7/20/2021.

[4] Apache Software Foundation, "Apache Spark – Lightning-Fast Cluster Computing," http://spark.apache.org/, accessed 7/20/2021.

[5] Facebook, Inc., "Presto: Distributed SQL Query Engine for Big Data," https://prestodb.io/, accessed 7/20/2021.

[6] M. Suetake, H. Kizu, and K. Kourai, "Split Migration of Large Memory Virtual Machines," in *Proc. ACM SIGOPS Asia-Pacific Workshop of Systems*, 2016.

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

[8] L. Li and Y. Zhang, "KVM Live Migration Optimization," KVM Forum 2015, 2015.

[9] Y. Ma, H. Wang, J. Dong, Y. Li, and S. Cheng, "ME2: Efficient Live Migration of Virtual Machine with Memory Exploration and Encoding," in *Proc. IEEE Int. Conf. Cluster Computing*, 2012, pp. 610–613.

[10] A. Koto, H. Yamada, K. Ohmura, and K. Kono, "Towards Unobtrusive VM Live Migration for Cloud Computing Platforms," in *Proc. Asia-Pacific Workshop on Systems*, 2012.

[11] L. Li and J. Yunhong, "Real Time & Fast Live Migration Update for NFV," KVM Forum 2016, 2016.

[12] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proc. Network and Distributed Systems Security Symp.*, 2003, pp. 191–206.

[13] National Institute of Information and Communications Technology, "StartBED4 Project," https://starbed.nict.go.jp/en/, accessed 7/20/2021.

[14] C. Waldspurger, "Memory Resource Management in VMware ESX Server," in *Proc. Symp. Operating Systems Design and Implementation*, 2002.

[15] S. Shen, V. Beek, and A. Iosup, "Statistical Characterization of Business-Critical Workloads Hosted in Cloud Datacenters," in *Proc. IEEE/ACM Int. Symp. Cluster, Cloud and Grid Computing*, 2015, pp. 465–474.

[16] D. Klusáček and B. Parák, "Analysis of Mixed Workloads from Shared Cloud Infrastructure," in *Proc. Workshop on Job Scheduling Strategies for Parallel Processing*, 2017, pp. 25–42.

[17] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation," in *Proc. USENIX Symp. Operating Systems Design and Implementation*, 2018, pp. 69–87.

[18] F. Bellard, "QEMU," https://www.qemu.org/, accessed 7/20/2021.

[19] T. Kashiwagi and K. Kourai, "Flexible and Efficient Partial Migration of Split-memory VMs," in *Proc. IEEE Int. Conf. Cloud Computing*, 2020, pp. 248–257.

[20] B. Fitzpatrick, "memcached – A Distributed Memory Object Caching System," http://memcached.org/, accessed 7/20/2021.

[21] B. Aker, "memaslap – Load Testing and Benchmarking a Server," http://docs.libmemcached.org/bin/memaslap.html, accessed 7/20/2021.

[22] J. Chiang, H. Li, and T. Chiueh, "Introspection-based Memory Deduplication and Migration," in *Proc. ACM Int. Conf. Virtual Execution Environments*, 2013, pp. 51–62.

[23] C. Wang, Z. Hao, L. Cui, X. Zhang, and X. Yun, "Introspection-based Memory Pruning for Live VM Migration," *Int. J. Parallel Program*, vol. 45, no. 6, pp. 1298–1309, 2017.

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