IEICE TRANSACTIONS

on Information and Systems

VOL. E98-D NO. 12
DECEMBER 2015

A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

PAPER Special Section on Parallel and Distributed Computing and Networking

Postcopy Live Migration with Guest-Cooperative Page Faults

Takahiro HIROFUCHI[†a)], ***Member***, **Isaku YAMAHATA**[††,†††], ***and*** **Satoshi ITOH**[†], ***Nonmembers***

SUMMARY Postcopy live migration is a promising alternative of virtual machine (VM) migration, which transfers memory pages after switching the execution host of a VM. It allows a shorter and more deterministic migration time than precopy migration. There is, however, a possibility that postcopy migration would degrade VM performance just after switching the execution host. In this paper, we propose a performance improvement technique of postcopy migration, extending the para-virtualized page fault mechanism of a virtual machine monitor. When the guest operating system accesses a not-yet-transferred memory page, our proposed mechanism allows the guest kernel to defer the execution of the current process until the page data is transferred. In parallel with the page transfer, the guest kernel can yield VCPU to other active processes. We implemented the proposed technique in our postcopy migration mechanism for Qemu/KVM. Through experiments, we confirmed that our technique successfully alleviated performance degradation of postcopy migration for web server and database benchmarks.

key words: *virtual machine, live migration, page fault, postcopy migration*

1. Introduction

Live migration of a virtual machine is a key technology in cloud computing infrastructure. It allows relocating a VM to another physical machine without stopping the VM. In data centers, live migration enables dynamic load balancing and flexible server maintenance.

As far as we know, widely-used virtual machine monitors implement so-called precopy migration [1]. All the memory pages of the VM are transferred to the destination PM, before the execution of the VM is suspended at the source PM and resumed at the destination. In this memory transfer phase, the guest operating system of the VM is still running at the source PM, updating memory pages of the VM. The VMM needs to transfer updated memory pages repeatedly, until the size of remaining memory pages becomes sufficiently small thereby minimizing downtime upon the switch of the execution host. It is well known that this behavior sometimes results in a long and non-deterministic migration time, especially in actively-running VMs being updating memory intensively

Thus, postcopy migration is considered promising for remedying the drawback of the precopy mechanism. Memory pages are transferred after the execution host is switched to destination. The guest operating system keeps running even before the memory transfer completes. A VMM promptly resolves the page faults caused by the VM accessing not-yet-transferred memory pages. Because postcopy migration does not involve iterative memory transfer, its migration time is deterministic and mostly shorter than that of precopy migration.

We developed a postcopy live migration mechanism for Qemu/KVM, and applied it to VM packing systems dynamically adjusting the locations of VMs in response to ever-changing resource requirement. Experiments showed that postcopy migration can contribute to achieving a higher degree of performance assurance and energy save than precopy migration [2]. The production-level code of the postcopy migration (Yabusame [3]) is publicly available under an open source license, which is intended to be merged to the mainline of Qemu/KVM.

The challenge of the development is to reduce the possibility of performance degradation that is sometimes observed just after the execution host of a VM is switched. Before a postcopy migration completes, there is a possibility that the VM may temporally pause in a very short period of time due to miss hit to memory pages. When the VM accesses a not-yet-transferred memory page, a hypervisor temporally stops the VM, transfers the content of the page as soon as possible, and then restart the VM. In order to reduce miss hit, the prototype of our postcopy migration parallelizes migration data transfer; an on-demand transfer stream copies missed memory pages in a real-time basis, and a background one copies the rest of memory pages in bulk. The hypervisor analyzes memory offsets of page faults and begins the background transfer with hot memory pages being frequently accessed now. Although this technique contributes to alleviating performance degradation, we consider that there is room for further improvement minimizing performance overhead of postcopy migration.

In this paper, we propose an advanced technique of postcopy migration, which ameliorates performance overhead by means of pseudo-virtualization of page faults. Even when a migrating VM miss-hits a memory page, the proposed technique allows the guest operating system to keep running without temporary pause. It suspends only the process or the thread on the guest operating system, which accessed the not-yet-transferred memory page, and keeps assigning CPU time among other native threads on it. This

Manuscript received January 5, 2015.
Manuscript revised May 14, 2015.
Manuscript publicized September 15, 2015.

[†]The authors are with National Institute of Advanced Industrial Science and Technology (AIST), Tsukuba-shi, 305–8568 Japan.
[††]The author was with VA Linux Systems Japan K.K., Tokyo, 135–0061 Japan.
[†††]The author is with Intel Corporation, San Jose, CA 95054-1549 USA.
a) E-mail: t.hirofuchi@aist.go.jp

DOI: 10.1587/transinf.2015PAP0011

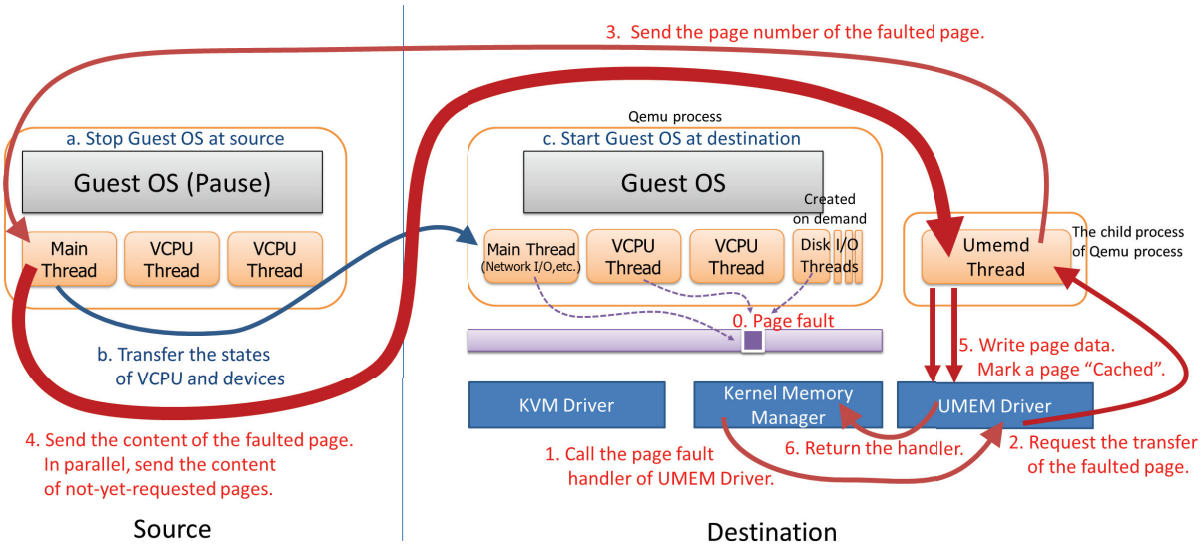


Fig. 1 Overview of postcopy live migration for Qemu/KVM (Yabusame)

mechanism is intended to improve the throughput of multi-threaded or multi-process applications during a postcopy migration. When a process or a thread on the guest operating system tries to access a not-yet-transferred memory page, a page fault happens in the hypervisor level, and triggers VM Exit. The physical CPU executing the VM quits the VM context and returns to the context of the hypervisor. Without the proposed mechanism, the guest operating system never notices this page fault, being frozen until the hypervisor makes the data of the page ready. On the other hand, with the proposed mechanism, the hypervisor injects a special interrupt into the guest operating system just after the page fault, and promptly returns to the guest operating system. The special interrupt requests the guest kernel to remove the current thread from the RUN queue and schedule CPU time among other threads. It should be noted that Linux Kernel 2.6.38 or later natively supports the proposed mechanism. These kernels work as a guest operating system of the proposed mechanism without any modification to them.

Section 2 explains the overview of postcopy migration, and Sect. 3 presents the proposed mechanism. Section 4 shows results of evaluation experiments. Section 5 describes related work. Finally, Sect. 6 concludes this paper. †

2. Overview of Postcopy Live Migration

Figure 1 illustrates the overview of our postcopy live migration mechanism for Qemu/KVM (Yabusame). This figure does not include the component of the proposed mechanism in this paper. Since the first prototype was presented in [5], it has been carefully redesigned to be fully compatible with other features of Qemu/KVM. The recent design supports not only the KVM-accelerated mode but also the TCG (Tiny Code Generator) mode of Qemu. The most parts of the postcopy mechanism are implemented by using the existing functions of the precopy mechanism. The extension to the existing code is minimized and reliable enough to be used in production-level environments.

In Qemu/KVM, a VM corresponds to a Qemu process on the host operating system, and the Qemu process is composed of several native threads. A VCPU thread corresponds to a virtual CPU of a VM. The number of VCPU threads of a VM is the same as that of virtual CPUs of the VM. A VCPU thread executes the code of the guest operating system. The main thread provides emulation of various devices such as a network interface. Because the disk I/O of Qemu is performed through a mechanism similar to POSIX Asynchronous I/O, a Qemu process sporadically invokes worker threads dedicated to this mechanism. At the inside of the kernel of the host operating system, the KVM driver works for controlling virtualization mechanisms of physical CPUs such as shadow page tables of VMs.

In the case of postcopy migration, a device driver trapping memory access, UMEM, is loaded into the host operating system at destination. The Qemu process creates the RAM of the VM with anonymous memory pages provided by UMEM. The Qemu process invokes a child process (UMEM daemon) transferring the data of memory pages.

The postcopy live migration of a VM works as illustrated in Fig. 1: (a) The VMM stops the VM at its source PM. (b) The VMM transfers the state of VCPU and devices to the destination PM. (c) The VMM restarts the VM at the destination PM. Now, the VM is running on the destination. When the VM accesses a not-yet-transferred memory page, the VMM transfers the data of the memory page from the source PM. In addition, the VMM also transfers the rest of

†This paper substantially extends our preliminary work presented at a symposium (ComSys2012) [4]. Especially, we have improved the evaluation section with the latest, performance-optimized implementation of the proposed mechanism. Bugs that affected performance have been fixed. Experiments with various applications have been conducted.

remaining memory pages in the background. After transferring all the memory pages, the VMM releases the data of memory pages on the source PM.

Steps 0-6 in Fig. 1 explain the detail of the transfer mechanism. Just after the VM restarts at the destination PM, memory pages are mapped to the destination Qemu process, but are not yet actually allocated. When any one of the native threads comprising the Qemu process accesses each memory page at the first time, a page fault happens at the destination PM. If this page fault is triggered by a VCPU thread being executing the context of the guest operating system, the VCPU thread promptly exits from the guest OS (i.e., VM Exit). In the case of a VCPU thread or any other threads, the execution of the thread causing the page fault is blocked on the host operating system, and the memory management system of the host operating system kernel calls the page fault handler of the UMEM driver.

The page fault handler of the UMEM driver checks whether the content of the memory page is already transferred. If the content is already transferred, the page fault handler returns promptly. The host operating system unblocks the execution of the thread. In the case that a VCPU thread being executing the guest context has caused the page fault, the VCPU thread restarts the execution of the guest context (i.e., VM Entry). In any case, the thread causing the page fault is temporally stopped in a very short period of time, which will not affect VM performance noticeably. This paper refers such page fault as minor fault.

On the other hand, if the content of the page is not yet transferred, the page fault handler requests the UMEM daemon in the userland to transfer it. The UMEM daemon sends the page number of the page to the source Qemu process. The source QEMU process returns the content of the page to the UMEM daemon. The UMEM driver saves the received page data and marks the page as transferred. Then, the UMEM driver finishes the page fault handler, unblocking the execution of the Qemu thread. This paper refers such page fault as major fault.

Major faults potentially cause noticeable performance degradation of a migrating VM. In comparison to a minor fault, a major fault involves data transfer between the source and destination, which means that the Qemu thread causing the page fault needs to be blocked in a long period of time. For example, while a minor fault costs only several microseconds in the environment of our experiments, a major fault costs several hundred microseconds beyond a GbE link.

3. Postcopy Live Migration with Guest-Cooperative Page Fault

We propose an advanced postcopy migration technique that reduces performance degradation during the postcopy phase. Instead of fully hiding miss hits from the guest operating system, the proposed mechanism implements a guest-cooperative postcopy mechanism. Upon the miss hit of a memory page, the hypervisor injects a special interrupt to the VM, which notifies the guest operating system that the accessed page is not ready. The guest operating system reschedules the process that has accessed the page, and continues other processes. The proposed mechanism is intended to effectively work for multi-threaded/process applications (i.e., web servers), which can been seen everywhere in cloud data centers.

The proposed mechanism is implemented by extending the Asynchronous Page Fault (APF) feature of KVM. APF is originally designed to alleviate performance drop when the guest operating system accesses a memory page that has been swapped out to a storage device on the host operating system level. Without APF, the guest operating system needs to be completely stopped until the host operating system swaps in the data of the memory page. On other hand, APF allows the guest operating system to keep running without disruption. In parallel, the host operating system is swapping in the data. In the proposed mechanism, our UMEM driver implements a special page fault handler to support APF. This allows continuing the execution of the guest operating system while retrieving the data of a fault page from the source Qemu process. The recent KVM drivers and Linux kernels with the APF feature support the proposed mechanism without any modification to them.

Figure 2 illustrates the overview of the proposed mechanism. It works only at the destination side. In the first phase, the guest operating system is notified of a miss hit.

1. The guest operating system accesses a memory page for the first time.
2. A page fault happens at the hardware level of the PM, resulting in an exit from the VM context.
3. The KVM driver tries to map the fault page to the address space of the Qemu process, which is to be performed in a non-blocking basis. The memory management system of the host operating system kernel calls the page fault handler of the UMEM driver with the non-blocking flag.
4. The UMEM driver scans the bitmap that records transferred memory pages. If the fault page is found on it, the page fault handler promptly returns with success, and then the execution of the guest context is resumed. Otherwise, this case is a major fault. The UMEM driver requests the UMEM daemon to transfer the page from the source, and follows the below steps.
5. The UMEM driver finishes the page fault handler with the ”Page Not Present” flag.
6. The memory management system of the host kernel invokes a kernel thread (i.e., workqueue) dedicated for deferred page fault handling.
7. The KVM driver injects a special interrupt to the VM, notifying the guest operating system that the page is not ready.
8. The interrupt handler of the guest kernel suspends the execution of the current process (that has triggered the major fault), and remove it from the run queue of the

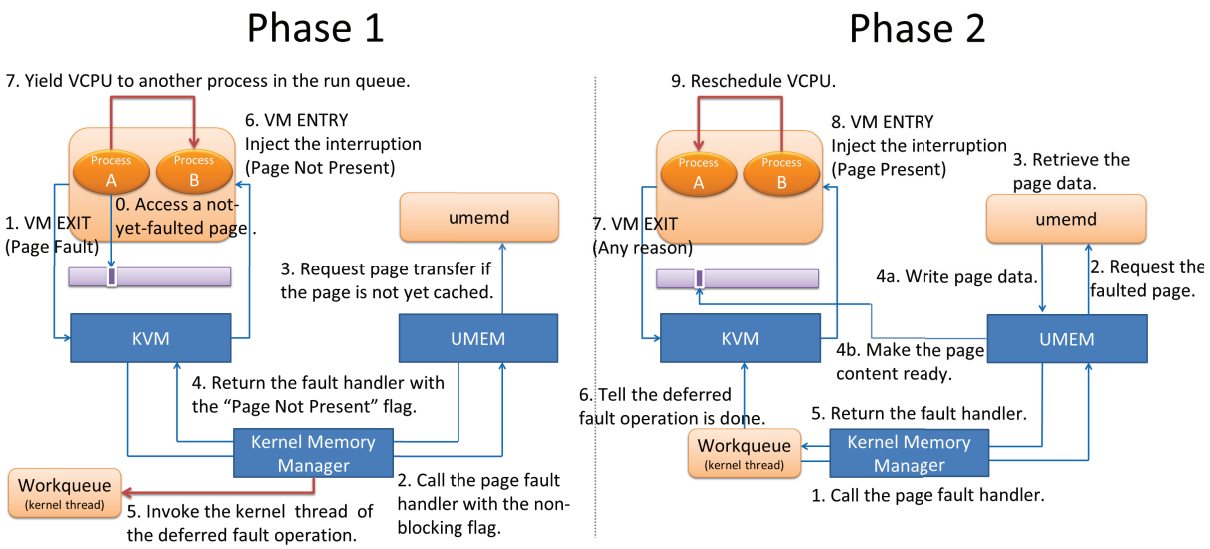


Fig. 2 Overview of the proposed mechanism.

process scheduler. Then, the guest kernel executes other processes in the run queue if available.

These steps are performed in a non-blocking basis. They are completed in several microseconds in our test environments.

In the second phase, the proposed mechanism waits for the data transfer of the fault page, and then notifies the guest kernel of the completion.

1. The workqueue of deferred page fault tries to map the fault page to the address space of the Qemu process in a blocking basis this time. The memory management system calls the page fault handler of the UMEM driver without the non-blocking flag. This is the normal behavior of a page fault handler in the Linux kernel.
2. The UMEM driver scans the bitmap again. There is the possibility that the data of the memory page has been transferred since the previous scan. If the faulted page is found, the mechanism jumps to Step 5. Otherwise, the UMEM driver requests the UMEM daemon to transfer the page from the source.
3. The UMEM daemon receives the content of the memory page from the source.
4. The UMEM daemon writes the page data to the VM memory, and then marks the page as transferred in the bitmap of the UMEM driver.
5. The UMEM handler finishes the page fault handler.
6. The workqueue notifies the KVM driver of the completion of the page fault.
7. (Case A) If the VCPU thread is running in the guest context, the notification to the guest is performed asynchronously. After a VM Exit event happens for any reason, goes to the next step. (Case B) If the VCPU thread already exited from the VM (e.g., the VCPU executed HLT), the KVM driver unblocks the execution of the VCPU.
8. The KVM driver injects an interrupt to the VM upon VM Entry, notifying it that the page is ready.
9. The interrupt handler of the guest kernel resumes the execution of the fault process. It moves the fault process to the run queue.

In the second phase, the call of the page fault handler is performed in a blocking basis, because the handler likely waits for the data transfer of the fault page. Thus, the workqueue thread is used here instead of the VCPU thread, so that the execution of the VCPU is not prevented.

The proposed mechanism is targeted on the case that a major fault happens when a VCPU thread is executing the guest context (i.e., during the VM Entry state). It is not targeted on the case that a major fault happens when a VCPU thread is executing the host context (i.e., during the VM Exit state). For example, a VCPU thread accesses the page table of a guest operating system for the emulation of APIC. The page used for the page table is not handled by the proposed mechanism, because the emulation is done in the host context. Major faults caused by the other threads than VCPU threads are not also handled by the proposed mechanism. In these cases, the execution of the thread is blocked until the content of the page is transferred from the source.

A VM with multiple VCPUs is composed of the same number of VCPU threads. The proposed mechanism works for each VCPU thread, respectively. There is the case that a VCPU thread triggers a major fault for the same memory page again, which is awaiting the completion of data transfer in the workqueue. This paper refers this case as double fault. Upon a double fault, the page fault is solved by the normal way; the CPU thread is blocked until the fault page is transferred. Supposedly, there is less possibility that the guest operating system continues to run without the fault page.

We have implemented the above mechanism for the latest Qemu/KVM and Linux Kernel. We are improving it according to feedback from users. It has been publicly available under the open source license since June 2012.

4. Evaluation

We conducted experiments to evaluate the proposed mechanism. First, we used a simple multi-thread program to carefully observe how the mechanism works, and then set up a web server system and a database server system to see its performance in the reality. Figure 3 illustrates the overview of our system setup. Host A is a source PM where a VM is firstly launched, and Host B is a destination PM which the VM is migrated to. A 10GbE link is used for migration traffic. Network latency between the PMs is approximately 30us by the `ping` program. Host C serves a storage server sharing the virtual disk of the VM with the PMs. Host C also runs a web server benchmark program or a database benchmark program. Both the PMs are equipped with 2 Intel Xeon E5620 processors, which are 8 CPU cores in total per PM, and 24 GBytes memory. Intel EPT (Extended Page Table), i.e., the hardware feature assisting address translation for VMs, is enabled. We disabled the C-state and P-state modes of the PMs in their BIOS and kernel configurations. This ensures that results of experiments are not affected by the dynamic power control mechanisms.

Since experiments were focused on performance degradation due to major faults, our postcopy migration mechanism was configured to use only the on-demand memory transfer of faulted memory pages. The background transfer of remaining memory pages was disabled during measurement periods. Because there is a possibility that the bursty background transfer suppresses the on-demand transfer and degrades system performance, we intend that in real use-cases the background transfer will be invoked after the network traffic of the on-demand memory transfer becomes sufficiently small.

Experiments were performed with 4KB pages. In the network of experiments, the minimum latency of a page transfer transaction was approximately 100us, which was measured on the UMEM daemon during a live migration of an idle VM. It is the period of time between Step 3 of Fig. 1 (i.e., the time when the UMEM daemon sends the 64bit offset number of a faulted page) and Step 4 (i.e., the time when the UMEM daemon receives the 4KB data of the page).

4.1 System Trace

We developed a simple multi-thread program running on the guest operating system of a migrating VM. In this experiment, a VM has one VCPU and 1 GBytes RAM. It creates 4 native threads, and each thread allocates a 200 MB memory space.

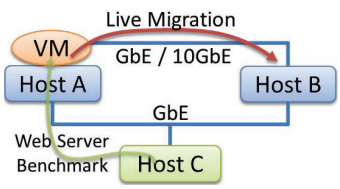


Fig. 3 System setup for experiments

At the same moment that a postcopy live migration starts, each thread starts reading its memory space sequentially from the first byte to the last, respectively. The elapsed time until the last byte is measured in each thread, respectively. With the proposed mechanism enabled, the elapsed time was approximately 9.8 seconds in each thread. Otherwise, it was approximately 17.6 seconds. The proposed mechanism contributed to reduce the elapsed time approximately by 44 %. In the situation where serious major faults are inevitable, the proposed mechanism allowed the VM to overlap the sequential memory reads with memory data transfer of major faults. The performance impact by postcopy major faults was successfully alleviated.

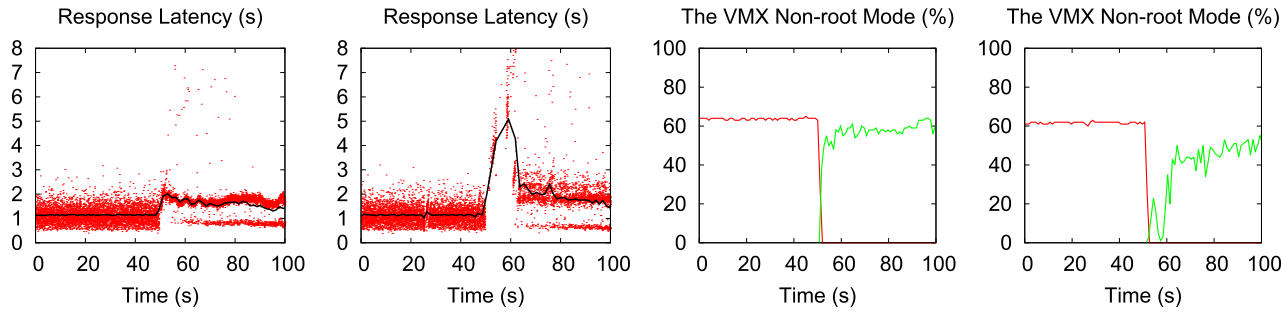
To examine how the proposed mechanism works in detail, the same experiment was performed again with SystemTap [6] enabled. We wrote a system tap script that monitors guest/host in-kernel events related to the proposed mechanism. It records VM Entry/Exit and the start and end of the page fault handler of the UMEM driver on the host operating system, and the call of the APF interrupt handler and the schedule of processes on the guest operating system. Figure 4 and 5 show the time sequence of these events regarding the major fault of a memory page (page frame 0x180d in the guest). In the figures, the last column points to corresponding steps in Fig. 2. In Fig. 4, at the host-level clock time of 198 microseconds, a VM Exit event happened due to page fault. Then, the page fault handler of the UMEM driver was called by the host operating system. At 209 microseconds, the page fault handler returned with the need-retry flag, which means that the page was not found in the already-transferred data. In Fig. 5, at the guest-level clock time 634 microseconds the guest operating system was notified of the

```
198: [vm exit]                      => (Step 1: 1)
203: [umem enter]  gfn=0x180d       => (Step 1: 2)
209: [umem return] gfn=0x180d
                          need_retry => (Step 1: 4)
226: [umem enter]  gfn=0x180d       => (Step 2: 1)
242: [vm entry]                     => (Step 1: 6)
     (skip events in 163usec)
405: [umem return] gfn=0x180d done => (Step 2: 5)
     (skip events in 99usec)
504: [vm entry]                     => (Step 2: 8)
```

Fig. 4 The host-level behavior of the proposed mechanism when handling the major fault of a memory page (page frame 0x180d in the guest view). The first column is the time clock in micro seconds. `gfn` is a page number in the guest. The last column points to the corresponding step in Fig. 2.

```
634: [apf wait] tid=4884 gfn=0x180d  => (Step 1: 6)
674: [schedule] tid=4884 to tid=4885 => (Step 1: 7)
900: [apf wake]          gfn=0x180d  => (Step 2: 8)
948: [schedule] tid=4885 to tid=4884 => (Step 2: 9)
```

Fig. 5 The guest-level behavior of the proposed mechanism when handling the major fault of Fig. 4. Note that the time clock of the guest operating system is not accurately synchronized with that of the host operating system. `tid` is a kernel thread id.



(a) Enabled (b) Disabled (c) Enabled (d) Disabled

Fig. 6 The web server benchmark with HTTP connections. 128 server processes. (a) and (b) show the response latency time of each request with/without the proposed mechanism, respectively. The black Bezier line shows the trend of Y-axis values. (c) and (d) show the percentage of the VMX non-root mode during the experiments.

major fault of the memory page. It promptly rescheduled the current thread (thread id 4884) that had caused the fault, and started executing another thread (thread id 4885). In this way, the guest operating system kept scheduling other threads that did not access the memory page. At the host-level clock time of 405 microseconds, the memory page was transferred from the source and the page fault handler returned with success. This information was injected to the VM at 504 microseconds. At the guest-level clock time of 900 microseconds, the guest operating system was notified of it, and put the fault thread 4855 to the run queue. After 48 microseconds, the process scheduler actually assigned CPU to the thread.

Through the above experiment, we have confirmed that the proposed mechanism can continue executing the guest operating system in parallel with the retrieval of fault pages. By analyzing the output of the SystemTap script, we found that when the proposed mechanism was enabled, the VCPU thread ran in the context of the guest operating system in 55 % of actual time. Without the proposed mechanism, the VCPU thread ran the guest only in 38 %. The proposed mechanism alleviated performance degradation caused by major faults of postcopy migration.

4.2 Application Benchmarks

We evaluated system performance through application benchmarks. The proposed system will contribute to reducing performance degradation of multi-threaded (or multi-process) web server programs.

4.2.1 Apache Web Server

We set up an Apache web server program on the guest operating system, and measured its performance before/during/after a live migration. In this experiment, a VM has one VCPU and 16 GBytes RAM. The web server program served 10 GBytes web contents to clients. The file size of each content was 1 MBytes, and there were 10000 files in total. The Apache server was configured to the pre-fork mode, in which an HTTP/HTTPS session is handled by each server process. On Host C, we launched a web server benchmark program, Siege [7]. It was configured to create 128 client threads. Each client thread continuously accessed random web contents. The Apache server was also configured to launch up to 128 server processes. After the network throughput of the web server benchmark became stable, we started measurement. It suggests that the guest operating system had cached most web contents in the page cache of the kernel.

Figure 6 shows the results of the web server benchmark with HTTP connections. At 50 seconds, the VM was migrated from the source PM to the destination. As shown in Fig. 6(a) and 6(b), the response latency time of a request increased due to major faults after the execution host of the VM was switched. In the case with the proposed mechanism, the response latency time increased approximately from 1 second to 2 seconds. Otherwise, it increased to 5 seconds or larger. It is clear that the proposed mechanism successfully reduced performance loss in the web server system. Figure 6(c) and 6(d) show the percentage of the VMX non-root mode, i.e., the period of time in which the VCPU thread is executing the guest-OS context. Just after the live migration, the proposed mechanism allowed the VCPU thread to execute the guest-OS context approximately at the percentages of 55-60%, which are (slightly smaller but) very close to that of the normal state. Without the proposed mechanism, a large performance drop was observed just after the migration, and then the percentage did not promptly recover. In comparison to the case of the proposed mechanism, the VCPU thread could not efficiently run the guest-OS context.

We also conducted the same experiment with HTTPS connections. As shown in Fig. 9, the proposed mechanism reduced performance degradation after the execution host of the VM was switched. It should be noted that, because the guest operating system consumed CPU cycles for data encryption and decryption, the percentage of the VMX non-root mode was higher than that of the HTTP benchmark.

As shown in the above experiments, the proposed mechanism will efficiently work for a server program cre-

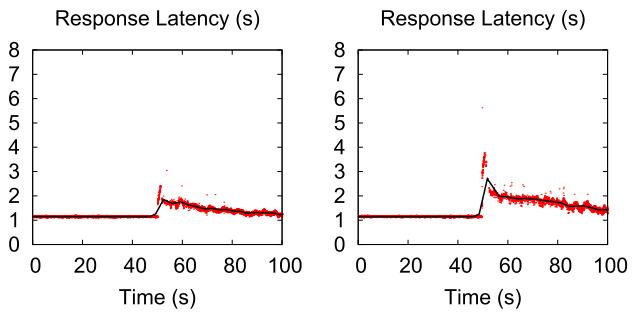
ating many concurrent processes. If the number of concurrent processes is small, there will be less possibility that the proposed mechanism hides network latencies. To confirm this point, we changed the configuration of the Apache server, reducing the number of server processes to 16 or 4. Figure 7 and 8 show the results of the HTTP benchmark. In the case of 16 server processes, we can observe that the proposed mechanism clearly contributed to reducing performance loss. The difference between the results with/without the proposed mechanism was smaller than that of the case of 128 server processes. In the case of 4 server processes, the difference between the results was still observable but very subtle. It should be noted that in the default setting of the pre-fork mode of Apache-2.2, the maximum number of server processes being concurrently created is 256. This value will be sufficient for the proposed mechanism to efficiently work.

4.2.2 PostgreSQL Database Server

Next, we set up a PostgreSQL database server on the guest operating system and created a database file of 2 GBytes. On Host C, we launched a database benchmark program, pgbench. It was configured to create 8 parallel client sessions. Each client session continuously generated query requests for a randomly-selected row in the database. During the benchmark, the database server on the guest operating system kept executing 8 server processes. After the throughput of the benchmark became stable, we started measurement. It suggests that the guest operating system had cached most blocks of the database file in the page cache of the kernel.

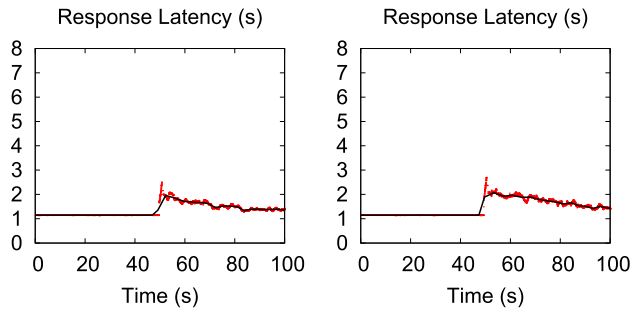
As shown in Fig. 10, the proposed mechanism greatly contributed to reducing performance penalty of major faults. Although there was a temporal increase of the request response latency just after the execution host of the VM was switched, the latency promptly became back small values. Most requests finished in less than 2 seconds. On the other hand, without the proposed mechanism, a substantial number of requests finished in 10-100 seconds. This large performance degradation remained unsolved until the end of the benchmark. Figure 10(c) and 10(d) explain that without the proposed mechanism the VCPU thread executed the guest-OS context only in 10-20% of the elapsed time. The execution efficiency of the VM became drastically worse than the case with the proposed mechanism (i.e., 80-90%).

We observed that after the execution host was switched, some requests finished in a shorter period of time than that of the normal state. In Fig. 10(a) and 10(b), after 50 sec-



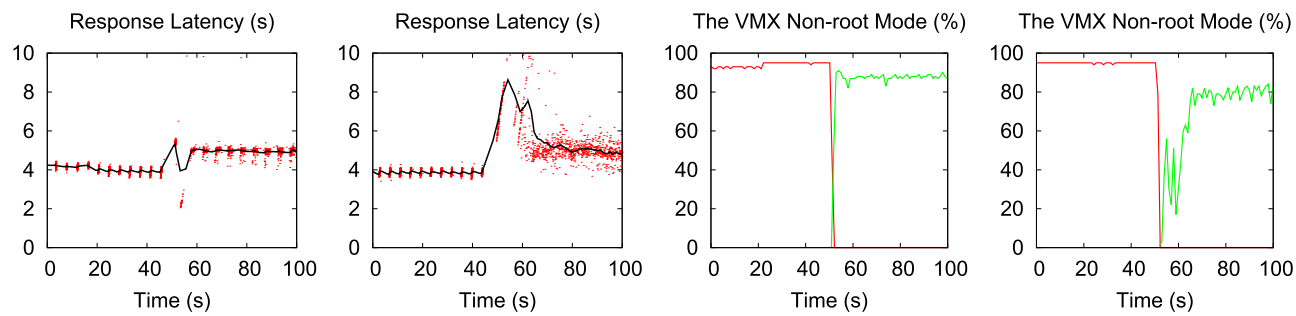
(a) Enabled (b) Disabled

Fig. 7 The web server benchmark with HTTP connections. 16 server processes. (a) and (b) show the response latency time of each request with/without the proposed mechanism, respectively. The black Bezier line shows the trend of Y-axis values.



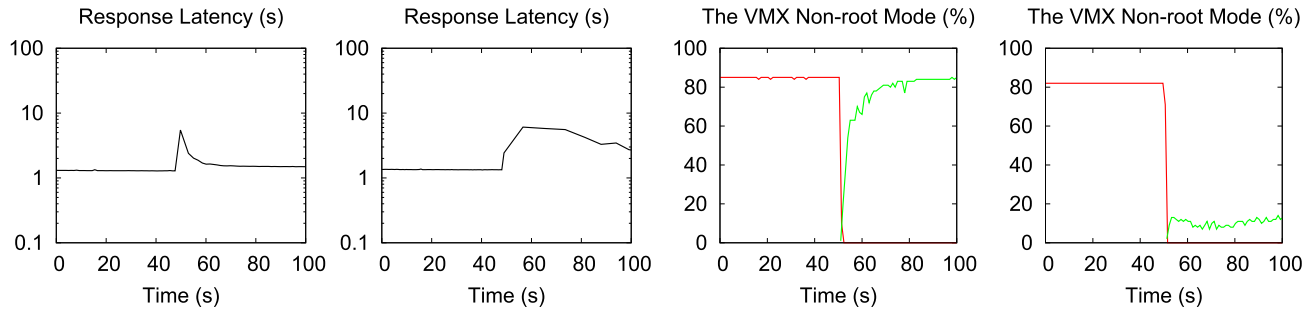
(a) Enabled (b) Disabled

Fig. 8 The web server benchmark with HTTP connections. 4 server processes. (a) and (b) show the response latency time of each request with/without the proposed mechanism, respectively. The black Bezier line shows the trend of Y-axis values.



(a) Enabled (b) Disabled (c) Enabled (d) Disabled

Fig. 9 The web server benchmark with HTTPS connections. 128 server processes. (a) and (b) show the response latency time of each request with/without the proposed mechanism, respectively. The black Bezier line shows the trend of Y-axis values. (c) and (d) show the percentage of the VMX non-root mode during the experiments.



(a) Enabled (b) Disabled (c) Enabled (d) Disabled

Fig. 10 The database server benchmark. (a) and (b) show the response latency time of each request with/without the proposed mechanism, respectively. The black Bezier line shows the trend of Y-axis values. (c) and (d) show the percentage of the VMX non-root mode during the experiments. The Y axis in (a) and (b) uses logarithmic scale.

onds, there are noticeable scattered dots near 0.5 seconds of the response latency time. In the normal state, the task scheduler of the guest kernel fairly assigns CPU time to each active process. However, when postcopy live migration is invoked, the task scheduler cannot guarantee fair CPU time. An active process that has caused a major fault needs to temporarily pause until the target page is transferred. Another active process that does not cause major faults can obtain more CPU time than usual.

In the experiments with the web server benchmark, even if the proposed mechanism was not used, serious performance degradation continued only during 10-15 seconds after the execution host of the VM was switched. For example, in Fig. 6(d), the period of time in the VMX non-root mode reached only 20% or less between 50 and 60 seconds but then increased approximately to 40%. It is considered that, since (not all but) the major part of necessary memory pages was transferred during this period of time, system performance got recovered to a degree. In the experiments with the database server benchmark, however, if the proposed mechanism was not used, serious performance degradation continued. The time in the VMX non-root mode kept being only at 10% approximately. In this case, the major part of necessary memory pages was not yet transferred. For a situation where major faults continuously happen, postcopy live migration without the proposed mechanism will suffer large performance penalty. Our proposed mechanism can virtually hide major faults, so that postcopy live migration will efficiently work with small performance overhead.

5. Related Work

It is well known that there is a possibility that memory pages of a VM are swapped out to external storage at the host operating system level, which likely results in intense performance degradation of the VM. This problem has been called double paging [8]. IBM z/VM has the feature that injects a special interrupt (Pseudo-Page-Fault Interruption) to a VM. The interrupt notifies the VM that a memory page is swapped out at the hypervisor level. It enables the guest operating system to continue running even when accessing a swapped-out memory page. KVM recently implemented a similar mechanism called Asynchronous Page Fault. The proposed mechanism in this paper has applied this technique to the page fault handling of postcopy live migration. As far as we know, this is the first attempt to use the technique for performance improvement of live migration. This paper has proved that postcopy migration can also take great advantage of asynchronized page faults.

There are other techniques improving performance of postcopy migration. SnowFlock [9] is a mechanism to rapidly clone a VM to other PMs, which is based on the same technique as postcopy live migration. In order to reduce performance impact, the VM clone mechanism cooperates with the memory management mechanism of the guest operating system. It does not transfer memory pages newly allocated after the execution of the VM is switched to destination. The existing content of these memory pages are not necessary and will be overwritten soon by the guest operating system. A following study [10] proposed a mechanism that analyzes the memory structure of the guest operating system, such as page tables, and transfer relevant memory pages at once, so as to reduce major faults. A study [11] presented a postcopy migration mechanism using a special swap device in the guest operating system. It reduces major faults by optimizing the order of memory pages to be transferred. Because a region of memory pages around the most recent fault page is regarded as a hot spot for the current workload, it transfers memory pages in order with the proximity to the hot spot.

6. Conclusion

In this paper, we propose a performance improvement technique of postcopy live migration, which is based on pseudo-virtualization of page faults. When the guest operating system accesses a not-yet-transferred memory page, the proposed technique allows the guest kernel to defer the execution of the current process until the page data is transferred. In parallel with the page transfer, the guest kernel can yield

VCPU to other active processes. We implemented the proposed technique in our postcopy migration mechanism for Qemu/KVM. Through experiments, we confirmed that our technique successfully alleviated performance degradation of postcopy migration for web server and database benchmarks.

Acknowledgments

This research project is partially supported by JST/CREST ULP, KAKENHI 23700048, and KAKENHI 25330097. The development of Yabusame was partly funded by METI (Minister of Economy, Trade and Industry) and supported by NTT Communications Corporation.

References

[1] C. Clark, K. Fraser, S. Hand, J.G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," Proceedings of the 2nd Symposium on Networked Systems Design and Implementation, pp.273–286, USENIX Association, 2005.

[2] T. Hirofuchi, H. Nakada, S. Itoh, and S. Sekiguchi, "Reactive cloud: Consolidating virtual machines with postcopy live migration," IPSJ Transactions on Advanced Computing Systems, vol.ACS37, pp.86–98, March 2012.

[3] Postcopy Live Migration for Qemu/KVM (Yabusame). http://grivon.apgrid.org/quick-kvm-migration

[4] T. Hirofuchi, I. Yamahata, and S. Itoh, "Improving performance of postcopy live migration with para-virtualized page fault," Proceedings of Computer System Symposium 2012, Information Processing Society of Japan, Nov. 2012.

[5] T. Hirofuchi, H. Nakada, S. Itoh, and S. Sekiguchi, "Enabling instantaneous relocation of virtual machines with a lightweight VMM extension," Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, pp.73–83, IEEE Computer Society, May 2010.

[6] V. Prasad, W. Cohen, F.C. Eigler, M. Hunt, J. Keniston, and B. Chen, "Locating system problems using dynamic instrumentation," Proceedings of Linux Symposium 2005, pp.49–64, IEEE Computer Society, 2005.

[7] J. Fulmer. http://www.joedog.org/siege-home

[8] R.P. Goldberg and R. Hassinger, "The double paging anomaly," Proceedings of the national computer conference and exposition, pp.195–199, ACM Press, 1974.

[9] H.A. Lagar-Cavilla, J.A. Whitney, A. Scannell, P. Patchin, S.M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan, "SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing," Proceedings of the fourth ACM european conference on Computer systems, pp.1–12, ACM Press, 2009.

[10] R. Bryant, A. Tumanov, O. Irzak, A. Scannell, K. Joshi, M. Hiltunen, H.A. Lagar-Cavilla, and E. de Lara, "Kaleidoscope: cloud micro-elasticity via vm state coloring," Proceedings of the sixth conference on computer systems, pp.273–286, ACM Press, 2011.

[11] M.R. Hines and K. Gopalan, "Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning," Proceedings of the 5th International Conference on Virtual Execution Environments, pp.51–60, ACM Press, 2009.

Takahiro Hirofuchi is a senior researcher of National Institute of Advanced Industrial Science and Technology (AIST) in Japan. He is working on virtualization technologies for advanced cloud computing and Green IT. He obtained a Ph.D. of engineering in March 2007 at the Graduate School of Information Science of Nara Institute of Science and Technology (NAIST). He obtained the BS of Geophysics at Faculty of Science in Kyoto University in March 2002. He is an expert of operating system, virtual machine, and network technologies.

Isaku Yamahata is a software architect in the Open Source Technology Center, Intel. His main focus is network virtualization as Software Defined Networking and Network Function Virtualization. Isaku is an active OpenStack Neutron (networking) developer and has in the past contributed significantly to Qemu, Kvm, Xen, and the Ryu SDN framework.

Satoshi Itoh obtained a Ph.D. in physics from University of Tsukuba, Japan, in 1987. From 1987 to 2002 he worked for high-performance and parallel computing in the both area of material science and business application at Central Research Laboratory, Hitachi, Ltd. In 2002, he moved to National Institute of Advanced Industrial Science and Technology (AIST), Japan and has researched on Grid computing, Cloud computing, and Green IT. He is currently the Director of Research Institute for Secure Systems, AIST.