IEICE TRANSACTIONS

on Information and Systems

VOL. E99-D NO. 12
DECEMBER 2016

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

IEICE TRANS. INF. & SYST., VOL.E99–D, NO.12 DECEMBER 2016

PAPER Special Section on Parallel and Distributed Computing and Networking

Fast Live Migration for IO-Intensive VMs with Parallel and Adaptive Transfer of Page Cache via SAN*

Soramichi AKIYAMA[†a)], ***Nonmember***, **Takahiro HIROFUCHI**[†], ***Member***, **Ryousei TAKANO**[†], ***and*** **Shinichi HONIDEN**[††,†††], ***Nonmembers***

SUMMARY Live migration plays an important role on improving efficiency of cloud data centers by enabling dynamically replacing virtual machines (VMs) without disrupting services running on them. Although many studies have proposed acceleration mechanisms of live migration, IO-intensive VMs still suffer from long total migration time due to a large amount of page cache. Existing studies for this problem either force the guest OS to delete the page cache before a migration, or they do not consider dynamic characteristics of cloud data centers. We propose a parallel and adaptive transfer of page cache for migrating IO-intensive VMs which (1) does not delete the page cache and is still fast by utilizing the storage area network of a data center, and (2) achieves the shortest total migration time without tuning hand-crafted parameters. Experiments showed that our method reduces total migration time of IO-intensive VMs up to 33.9%.

key words: *live migration, virtualization, cloud performance*

1. Introduction

Virtualization techniques are highly important for cloud computing. Cloud data centers enjoy easy maintenance, isolation across users, and high resource utilization thanks to virtualization. Live migration [2] is one of the virtualization techniques utilized in cloud data centers. It allows dynamical relocation of virtual machines (VMs) without disrupting the services running on them, which achieves high memory utilization [3], load balancing [4] and low energy consumption [5] of cloud data centers. Because live migration is the fundamental building block of these studies, efficient live migration is the key to apply them to real-world data centers.

Migrating a VM requires to transfer the memory of the target VM, which can be a few gigabytes to tens of gigabytes. Among the large amount of memory usage, page cache (a.k.a. file cache or buffer cache) dominates a large portion when the VM runs IO-intensive workloads. Page cache is a widely-adapted mechanism to improve performance of disk IO operations and is equipped in most modern operating systems. Large amount of page cache prolongs live migration thus it must be approached to achieve efficient live migration. Existing studies tackle this problem by either (1) force the guest OS to delete the page cache before a migration to shrink the memory size [6], [7], or (2) use the storage area network (SAN) for transforming page cache [1], [8], [9]. However, the former studies greatly penalize the IO performance of the VM due to the loss of the page cache, and the latter studies cannot adapt to the dynamic characteristics of cloud data centers (described in detain in Sect. 3.3).

In this paper, we propose a *parallel and adaptive* mechanism to efficiently transfer page cache during a live migration. The technique utilizes the storage area network (SAN) and the general purpose network (GPN) of a data center to transfer page cache *in parallel* via both of them, while *adaptively* determining which network to use for each memory page. Our experiments using WebServer, Postmark, and TPC-C workloads showed that our method greatly reduced total migration for various IO-intensive workloads (thanks to the parallelism), without manually tuning how much portion of page cache must be transferred via SAN (thanks to the adaptiveness).

This paper is structured as follows. Section 2 explains the background and motivates the challenge. Section 3 describes our core ideas. Section 4 shows the overview of our method and how it shortens total migration time. Section 5 illustrates technical contributions. Section 6 explains the implementation details. Section 7 shows the evaluation results. Section 8 gives further discussions. Section 9 refers related work and Sect. 10 concludes the paper.

2. Background

2.1 Live Migration and Its Applications

Live migration of VMs (or simply live migration) is one of the virtualization techniques that makes today's cloud computing paradigm different from traditional grids/clusters. It enables a VM to move from one host to another without interrupting services running on the VM. Live migration is used to improve efficiency (of both resource and energy), to achieve fault tolerance, and to ease data center maintenance.

Dynamic optimization of VM placement, or simply dynamic VM placement, is an important application of live migration. It improves the overall performance of a data center,

Manuscript received January 8, 2016.
Manuscript revised May 26, 2016.
Manuscript publicized August 24, 2016.

[†]The authors are with National Institute of Advanced Industrial Science and Technology (AIST), Tsukuba-shi, 305–8560 Japan.
[††]The author is with The University of Tokyo, Tokyo, 113–8654 Japan.
[†††]The author is also with National Institute of Informatics (NII), Tokyo, 101–8430 Japan.

*This is a journal version of our previous paper [1], and a part of the texts in Sect. 1, 2.3, 3.1, 4.2, 5.1, 6.1, 7.1 and 8.2 appear in the previous work.

a) E-mail: s.akiyama@aist.go.jp

DOI: 10.1587/transinf.2016PAP0021

such as energy consumption, resource utilization, and load balancing. Changes of usage pattern and load of the VMs on a cloud are highly dynamic and unpredictable, thus dynamic VM placement is an essential technique to adapt to the changes.

Examples of dynamic VM placement are: VMs with similar memory contents are dynamically consolidated to share the identical memory pages and reduce the overall memory usage of the data center [3]; Overloaded VMs are packed into under-utilized hosts while taking the network topology into account [4]; VMs under low load are live migrated into small number of physical hosts to turn off spare hosts and reduce the energy consumption [5].

2.2 Technical Details of Live Migration

Live migration of a VM requires to transfer (a) memory pages (b) disk image (c) device states (e.g. CPU registers) of the target VM. The primary focus of live migration researches is how to efficiently transfer the memory pages. The disk image is too large to transfer at a migration time thus to locate it on a shared storage such as NFS is common. The device states are in contrast very small and easy to transfer.

Pre-copy live migration is the most well-known mechanism of live migration. It was proposed by Clark *et al.* [2] and Nelson *et al.* [10], and widely implemented in hypervisors such as KVM [11] and Xen [12]. Transferring the memory pages of a target VM has three phases in pre-copy live migration:

1. All the memory pages of the migrated VM are transferred at first. For example, if the VM's memory footprint is 1 GB, the amount of transferred memory in this phase is also 1 GB.
2. Some memory pages are updated while other pages are transferred because the VM keeps running during a migration. The updated memory pages are transferred again until the number of remaining memory pages become sufficiently small.
3. The VM is suspended for a short period to transfer the remaining memory pages and device states.

Although many researches have proposed novel techniques for the second phase [13]–[17], the first phase is not yet well optimized for VMs that treat large data. This problem must be solved because the first and the second phases are equally important since they both transfer large number of memory pages. The details of the problem we tackle are discussed in Sect. 2.3.

2.3 Large Memory Consumption for Page Cache

Among a large amount of memory transferred in the first phase of a migration, *restorable page cache* dominates it when the VM runs IO-intensive workloads. Page cache is an on-memory cache mechanism to hide the gap between the accessing speed of memory and storage, and memory usage for page cache can be very large because modern operating systems use as many free memory pages as possible for page cache. *Restorable* page cache refers memory pages whose data can be restored from the identical disk blocks even if the pages are deleted, that is, all memory pages cached for read operations and memory pages cached for write operations and then flushed back to the disk.

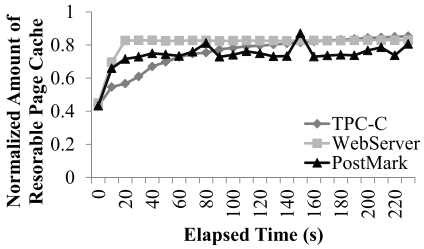

Fig. 1 Amount of restorable page cache in the memory under WebServer, Postmark, and TPC-C, workloads, normalized by the total memory usage of the VM.

Figure 1 shows the amount of restorable page cache contained in the memory of VMs running IO-intensive workloads. The x-axis shows the elapsed time from the beginning of the workload, and the y-axis shows the normalized amount of restorable page cache. The values are normalized by being divided by the total memory usage of the VM. WebServer is a workload that simulates a web server under high load. A load generator outside of the VM accesses the web contents with high access rate. Postmark is a workload to measure IO performance of small files to estimate server performance for web and mail services. TPC-C is a workload that simulates the typical database access pattern for an online shopping web site. The detailed descriptions of WebServer, Postmark, and TPC-C workloads are given in Sect. 7. The figure shows that in all three workloads many memory pages (more than 70% of all pages) are identical to disk blocks due to the restorable page cache most of the time during the workload execution. The values are small in the beginning but this does not weaken our claim because the periods are warming-up phases of the workloads.

3. Core Ideas

3.1 Network Architecture of Cloud Data Centers

An important characteristic of data center networks that we exploit is explained here. Figure 2 illustrates a simplified view of the network architecture of a typical data center. The main point is that storage nodes are connected with a designated SAN along with a general purpose network. For example, CISCO suggests a data center networking architecture that includes Storage Networking and Business Continuance Networking [18]. Nodes might have another link for management purposes. Descriptions of each network are as follows:

Storage Area Network (SAN): It is used to communicate with the storage nodes in the data center. A shared file

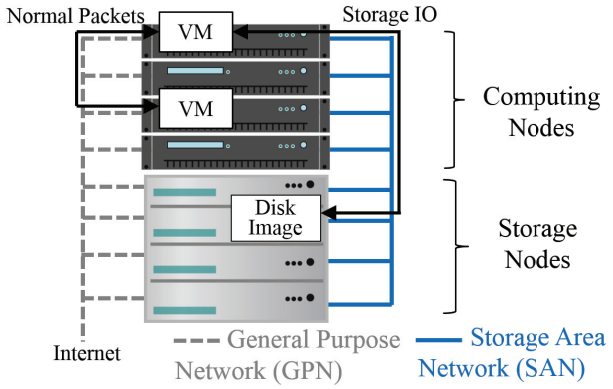

Fig. 2 Network architecture of a data center. Storage nodes and computing nodes have designated networks: SAN and GPN.

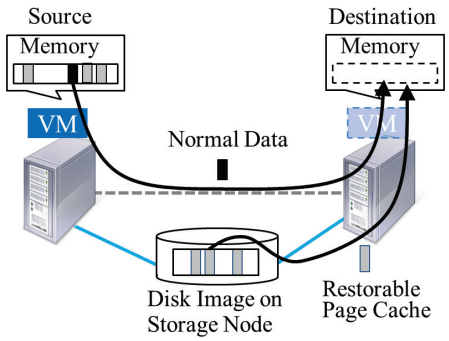

Fig. 3 How Live Migration is Accelerated by Our Proposal. The SAN and the GPN can be used in parallel because restorable page cache can be transferred via both links.

system is built on top of this link and IO requests and data from/to the storage nodes go through this link.

General Purpose Network (GPN): It is used to deal with any network packets other than storage-related ones. For example, HTTP requests sent from the Internet or sent between services running in the data center go through this link.

3.2 Parallel Transfer of Page Cache

The first core idea of this paper is that restorable page cache can be fetched both via the GPN and the SAN because the same data resides both in the VM memory and the disk image. Therefore, transferring a memory page containing restorable page cache via the SAN can be parallelized with transferring another memory page containing restorable page cache or any other data via the GPN. This greatly reduces the total migration time of a migration since the available network bandwidth for the migration doubles in the best case.

Figure 3 shows how live migration is accelerated by our proposal. The source and the destination hosts are connected by the GPN (general purpose network), and the storage node is connected with them by the SAN. The rectangles in the VM memory and the disk image indicate memory pages and disk blocks. The black one contained only in the VM memory is a normal data page. The gray ones contained both in the VM memory and the disk image are restorable page cache pages. In the figure, transferring the normal data page and transferring a restorable page cache page is parallelized as the gray rectangles can be fetched both via the GPN and the SAN.

3.3 Adaptive Transfer of Page Cache

The second core idea of this paper is adaptive transfer of page cache to adapt to the highly dynamic characteristics of cloud data centers. For each memory page containing restorable page cache, which network link (SAN or GPN) to use to transfer it must be determined *adaptively*. This means that, let R ($0 \leq R \leq 1$) be a ratio of restorable page cache transferred via the SAN against the total size of the restorable page cache, finding the best R a-priori to a migration is infeasible because of two issues:

1. Available network bandwidths for migration are unpredictable and dynamically changing. This means that a-priori-determined R can be no longer the best when the available network bandwidths change. For example, the SAN can suddenly be congested after 50% of memory transfer of a migration has finished.
2. Reading page cache from the disks of the storage nodes, rather than transferring it via the networks, can be the bottleneck in some cases. In this case predicting disk read throughput is required to find the best R, which is infeasible for real workloads. We will show a real example where disk read throughput is the bottleneck in the evaluation.

In order to mitigate this issue, our adaptive page cache transfer technique achieves the shortest total migration time without determining R a-priori to a migration by automatically transfers a portion of restorable page cache via the SAN. The technical details of the method is described in detail in Sect. 5.3.

Adaptive page cache transfer allows cloud providers to adjust the interference to the SAN from migration traffic, for example in order not to violate their SLAs. More concretely, cloud providers can throttle the throughput of page cache transfer via the SAN to an arbitary value to keep the interference within a desired amount, because our mechanism adapts to the available network bandwidth at runtime. However the reduction ratio of total migration time by our mechanims is also limited when the available SAN bandwidth for migration is adjusted.

4. Proposed System

4.1 Design Overview

We propose an advanced memory transfer mechanism that exploits our core ideas. The design criteria of the system are:

1. Performance interference of the system must be small

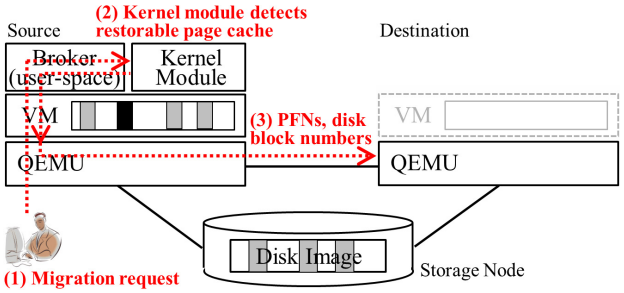

Fig. 4 Meta data to realize our method

because dynamic VM placement is a background operation from which users of the VMs do not want to be affected.
2. Implementation must be easy with little dependency to the guest OS because types of guest OS in a cloud has a large variety.
3. The system must have no special assumption/ modification to the storage nodes, because in a data center a storage node is not necessarily a server but can be an integrated solution (e.g. NetApp).

The design criteria lead to our system components. Small performance interference and implementation easiness are achieved by using a *kernel module inside the guest OS* to locate the restorable page cache within the memory. No special assumption to the storage nodes requires the system to be transparent from the underlying shared file system and the network protocol of the SAN.

4.2 Migration Procedure with Our Mechanism

The procedure of a live migration with parallel and adaptive page cache transfer is described in this section. The meta data other than the actual memory content to implement the procedure is illustrated in Fig. 4.

1. The source VMM receives a request to execute a migration. The requests is passed to our kernel module installed inside the guest OS with the help of a small user-space broker program ((1) in Fig. 4).
2. The kernel module detects the page frame numbers (PFNs) of restorable page cache and the identical disk blocks ((2) in Fig. 4). They are sent to the source and the destination VMMs with the help of the user-space program ((3) in Fig. 4).
3. The memory transfers are kicked-off. Memory pages are transferred using the SAN and the GPN in parallel.
    a. The destination VMM fetches memory pages containing restorable page cache from the storage node via the SAN.
    b. The source VMM sends memory pages containing normal data via the GPN. Once the number of normal data pages to be transferred becomes sufficiently small, the source VMM starts sending restorable page cache via the GPN.
4. Memory pages updated during above steps are transferred again via the GPN to guarantee the memory consistency.
5. Once the amount of remaining memory becomes sufficiently small and all the restorable page cache is transferred, the execution host of the VM is switched.

Note that the procedure 4 is done after the procedure 3 finishes but not in parallel with it. Therefore, the utilization of the two network links is not perfectly balanced if the VM aggressively updates its memory pages. This choice stems from a well-known fact that updated memory pages are highly possible to be re-updated in near future, thus immediately retransmitting updated memory pages can incur repeated retransimission [19]. However, parallelizing the procedure 3 and procedure 4 is straight-forward, because sending restorable page cache via the GPN (procedure 3-b) can be preempted anytime to give priority to retransmitting updated memory pages (procedure 4). Our algorithm automatically adopts to the change of transfer throughpout of restorable page cache via each network link (details in Sect. 5.3).

5. Technical Challenges and Approaches

The technical details of this paper is described in this section. Realizing our system is involved in three technical challenges and each of them is described in detail.

5.1 Memory Consistency

Memory pages containing restorable page cache can be updated during a migration and turn into non-restorable. In this case, the latest data of the page must be transferred via the GPN (general purpose network) because there is no guarantee that the updated data is flushed into the disk and accessible via the SAN. The are two scenarios that incur this situation: (1) a write operation to the cache occurs, or (2) the guest OS re-allocates the memory page for non-page cache purpose due to a memory pressure.

We solve this issue with the dirty page tracking functionality of the VMM. The x86 architecture has a dirty bit for each memory page that is set when the page is updated. The VMM provides a functionality to read the dirty bits from the software level. Dirty page tracking is enabled at the source host when a migration starts and all the memory writes after that are tracked. A memory page updated during the tracking is transferred via the general purpose network, even if the memory page was restorable when the kernel module detected restorable page cache.

5.2 Memory Writing Algorithm

The memory writing algorithm on the destination side must be carefully designed. This is because a memory page at the destination host can be written by two threads, one holding the latest data transferred from the GPN and another holding

an obsolete data transferred from the SAN (see Sect. 5.1 for the reason why this happens).

The VMM in the destination host has a *received flag* and a mutex for each memory page. Both the two threads try to acquire the lock before writing to a memory page to achieve an atomic write. However, a lock confliction does not mean the same to the two threads: When a lock confliction occurs, the thread holding the latest data always have to wait for the lock to be released and then overwrite the latest data to the memory page. On the other hand, the thread holding an obsolete data just discards the data because it is obsolete. Note that we introduce a small buffer in order not to block the thread executions on a lock confliction. The buffer includes the latest data for a small number of memory pages. When a lock cannot be acquired for a memory page, the thread can try acquiring the lock for the next page in the buffer, instead of being blocked.

The memory- and performance- overhead due to the locking is enough small. A flag is 1 byte and a mutex is 40 bytes (in Linux pthread implementation) thus the sum of them is 4% of the size of a memory page. The important notice is that the flags and mutex are no longer required after a migration thus there is no extra memory required during non-migration time. We confirmed that the number of lock confliction can be negligibly small by setting the size of the buffer 32, which consumes merely 128 KB (4 KB/page $\times$ 32 pages) of memory.

5.3 Adaptive Page Cache Transfer

The adaptive page cache transfer mechanism automatically determines which network link (SAN or GPN) to use for each memory page containing restorable page cache. This mechanism is highly important because determining the amount of transferred page cache via each link prior to a migration results in network load imbalance, and thus non-optimal total migration time, due to the dynamic characteristics of cloud data centers (the details are described in Sect. 3.3).

The fundamental issue is that memory pages are "pushed" from the source host to the destination host because only the source host knows which memory pages are updated during live migration, while the disk blocks are "pulled" by the destination host from the storage nodes because the design criteria (3) allows no modification to the storage nodes. This means that the source and the destination hosts have to **independently** decide which link to use for a given memory page.

We solve this issue by our *reverse ordered transferring algorithm*:

1. Preparation: The VMMs in the source and the destination hosts have pairs of PFNs and disk block numbers $\{P_i, D_i\}$ $(1 \leq i \leq n)$. Memory page P_i contains the same data as the disk block D_i does, and n is the number of memory pages that contain restorable page cache. The pairs are sorted in the order of D_i.
2. Parallel transfer phase: The two network links are assigned the pairs to transfer in reversed orders. Memory transfer via the GPN is done in the **descending** order of D_i, while disk block transfer via the SAN is done in the **ascending** order of D_i. Because the two transfers are done in reversed orders of D_i, they do not overlap until all restorable page cache has been transferred.
3. End of transfer phase: The parallel transfers reach to the same pair $\{P_s, D_s\}$, meaning that all restorable page cache has been transferred. The destination host can detect this point by the received flag (described in Sect. 5.2). It skips pulling the disk blocks D_i $(i > s)$, and notifies the source host of the end the transfers.

6. Implementation

6.1 Detecting Restorable Page Cache

Our kernel module detects the PFNs of the memory pages containing restorable page cache and the block numbers of the identical disk blocks to the restorable page cache. It utilizes OS dependent kernel functions and data structures to easily detect them. Our current implementation requires Linux guest, but we believe it is easy to implement it for other guest OS (Windows provides similar kernel functions to the ones we use in Linux). The size of the module is 155 KB only and it takes less than a second to detect restorable page cache from 1 GB of memory and 20 GB of disk.

The use of kernel functions and data structures greatly reduces the implementation cost to detect restorable page cache. In Linux, `pfn_to_page` kernel function takes an integer as the input and returns `struct page` kernel data of a memory page whose PFN is the input integer. If the page contains page cache, the `struct page` includes a flag indicating whether the page is flushed back to the disk, which means this page is restorable. The disk block number that has the identical data to the page is retrieved by passing the `struct page` to another function `bmap`.

Installing an extra kernel module and a broker program might not be acceptable for users with strict security policies. In this case, our system can simply fall back to normal live migration without any special mechanism. The VMMs can detect that there is no broker program inside the VM from a failure of connection establishment in the migration procedure (1) described in Sect. 4.2. Once a connection failure is detected, the VMMs can regard the amount of restorable page cache to be zero and this automatically makes our mechanism transfer all the memory pages via the GPN, which is exactly the same as normal live migration.

6.2 Modified VMM

Our modified version of QEMU/KVM has two unique steps compared the vanilla one. First, they retrieve the PFNs (page frame numbers) of restorable page cache and the identical

disk block numbers from the migrated VM. As described in Sect. 6.1, locating the restorable page cache uses our kernel module installed inside the VM. Because a kernel module cannot send data itself, we use a user-space broker program to pass the data, retrieved by an `ioctl` call to the kernel module, to the VMMs. We simply use TCP/IP for the data transfer between the user-space program and the VMMs. The size of the transferred data is 8 MB when the VM memory size is 4 GB (4 byte for PFN + 4 byte for disk block number) $\times$(4 GB $\div$ 4 KB/page). While using TCP/IP largely simplifies the implementation, it has a performance disadvantage compared to using more sophisticated method (described in Sect. 8.2).

Second, the VMM in the destination host invokes two threads, receiving thread and restoring thread, for the parallel transfers. The receiving thread receives normal memory pages via the general purpose network, and the restoring thread copies restorable page cache via the SAN. The implementation of the restoring thread is straight forward: it uses normal read/write system calls to fetch the restorable page cache and does not require any change to the underlying network settings/storage nodes. Normal read/write calls are automatically dispatched to the SAN by the underlying file system because disk images are on a shared file system (such as NFS, ATAoE, and iSCSI) as normally done in cloud data centers.

7. Evaluation

7.1 Methodology

Our evaluation consists of two parts and the metric each part measures is as follows:

1. Total migration time under various workloads, with all our proposals enabled.
2. Total migration time under various workloads, without the adaptive page cache transfer but with a pre-defined parameter R that specifies how much portion of restorable page cache is transferred via the SAN.

The first part shows that our proposal successfully shortens total migration time. The second part shows that the adaptive page cache transfer automatically yields the optimal results by comparing with manually tuned cases.

The evaluations environments are shown in Table 1. The physical hosts have three 1 Gbps network interface cards (NICs), two of which are used for the SAN and the GPN. The read throughput of the storage devices are measured using bonnie++ in the host OSes. The VMM is composed of QEMU 0.13.0 and KVM 2.6.32. The KVM is the default version of the host OS. Each measurement uses three servers: two computing nodes and a storage node shared across the cluster via Network File System (NFS). A VM running a workload is migrated from a computing node to another computing node. The disk image of the VM is stored in the storage node. The read/write block size of NFS is tuned to 8 KB because it achieved the best result[†].

Table 1 Evaluation environments

	Host	Guest
CPU	Intel Xeon X5460	1 core vCPU
Memory	8 GB	3 GB
Storage	256 GB HDD (read: 90 MB/s)	20 GB (raw disk image)
Network	1 Gbps NIC $\times$ 3	1 Gbps vNIC (bridged)
OS	Debian GNU/Linux 6.0.5 (Linux 2.6.32)	
VMM	QEMU 0.13.0, KVM 2.6.32	

The evaluation is conducted with three workloads: WebServer, TPC-C and Postmark.

WebServer simulates a web server under high load. Apache web server has static files. The number of files is 10,000 and the size of each file is 300 KB. A load generator, httperf, fetches the files with the speed of 50 files/s. Therefore the page cache is read with 15MB/s and no write occurs to it (since the files are static). Read from the storage node occurs only once at the beginning. The load generator runs on a designated host (not the same neither as source nor destination) and accesses the files via the third NIC to avoid interference to the migration process. The migration is executed 250 seconds after the workload started, where all the files has been cached in the page cache.

Postmark [20] is a benchmark that measures IO performance of small and short-lived files to simulate load of mail, net news, or web servers. A survey on file system benchmarking tools [21] reports that Postmark is the third most popular in research during 2009–2010. We set up the parameters of Postmark as follows: repeat 80,000 transactions in the speed of at most 500 per second with 1MB–5MB files and 4KB of read/write buffers. In our VM the workload achieved reading data with 154.56 MB/s and writing with 155.31 MB/s during the execution, not including the warming up time. Please note that this does not mean the amount of unique bytes read or written, and actual percentage of restorable page cache within the memory at each time point is shown in Fig. 1.

TPC-C [22] is a benchmark that measures the performance of a database system. It generates database access patterns that simulates an online shopping web site. It has 5 types of transcations (new-order, payment, delivery, order-status, and stock-level) and the detailed database schema is in page 9 of an official document [23]. We used the default mixture ratio of the 5 types (45%, 43%, 4%, 4%, and 4% respectively) and the *warehouse* parameter is set to 20, which gives the total data size of 1.9 GB. In our VM the benchmark achieved 129.7 tpmc (transcations per minute) in average, not including the warming up time. The official document [23] shows in page 16 that 1 tpmc typically generats 0.5 physical IOs/sec to the disks. Therefore 129.7 TpmC generates 64.85 IOs/sec, which is a moderately high IO load for an HDD. The migration is executed 270 seconds after the the workload started, where the warming up phase

[†]In our previous paper [1], the block size was 4KB.

of TPC-C has been finished.

7.2 Total Migration Time with Adaptive Page Cache Transfer

Figure 5 (a), 5 (b) and 5 (c) show total migration time under WebServer, Postmark, and TPC-C workloads, respectively. The light-colored bars indicated "original" are the results with un-modified live migration implemented in QEMU 0.13.0, and the dark-colored bars indicated "proposed" are the results with our proposal. Each value is calculated by averaging over 10 runs.

The reduction ratios of the total migration time against the original QEMU are 33.9% under Postmark workload, 20.3% under WebServer workload, and 13.3% under TPC-C workload. These results show that our proposal efficiently reduces the total migration time under various IO-intensive workloads.

There are large difference between the reduction ratios depending on each workload. The difference stems from characteristics of IO-operations in each workload. The details are discussed in Sect. 8.4.

7.3 Experiments w/o Adaptive Page Cache Transfer

To confirm that our adaptive page cache transfer mechanism achieves the optimal result, the total migration time in Sect. 7.2 are compared with manually tuned results without using the adaptive page cache transfer. The adaptive page cache mechanism is turned off and a pre-defined parameter R is given. R represents the ratio of restorable page cache transferred via the SAN to all the restorable page cache. That is, $R = 0.6$ means that 60% of the restorable page cache is transferred via the SAN and 40% is transferred via the GPN.

The left-side graphs of Fig. 6, Fig. 7, and Fig. 8 show the total migration time without the adaptive page cache transfer under WebServer, Postmark, and TPC-C workloads, respectively. The x-axis of each figure shows a value of R and the y-axis shows the total migration time for the R. Note that total migration time with $R = 0$ are not the same as the "original" in Sect. 7.2, because the values with $R = 0$ include overhead of retrieving the locations of restorable page cache (further discussion in Sect. 8.2). For all workloads, the shortest total migration time achieved in this experiment matches the one in Sect. 7.2. This means our adaptive page cache transfer mechanism achieves the optimal result without manually tuning R.

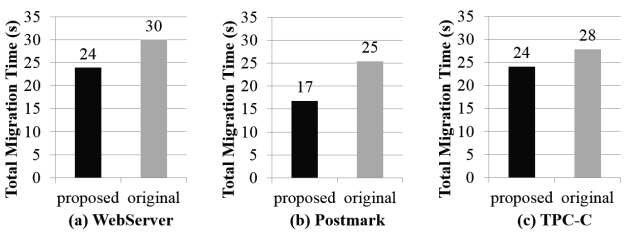

Fig. 5 Total migration time with and without our proposal under (a) WebServer (b) Postmark (c) TPC-C workloads.

The graphs in the middle of Fig. 6, Fig. 7, and Fig. 8 show the elapsed time consumed by *Receiving Thread* and *Restoring Thread*. Receiving thread is a thread in the destination host that receives memory pages sent from the source host, and Restoring thread is another thread in the destination host that fetches restorable page cache from the storage node. The x-axis shows a value of R and the y-axis shows the elapsed time of each thread for the R. For all workloads, the shortest total migration time is achieved when two bars have almost the same lengths. This means that the total migration time is the shortest when the SAN and the GPN are both utilized throughout the migration.

The right-side graphs of Fig. 6, Fig. 7, and Fig. 8 show the amount of data transferred by each thread. An interesting point is that the trend is not necessarily the same as the trend in the elapsed time of each thread. In Fig. 8, the elapsed time by the two threads are almost the same at $R = 0.4$ (where the shortest total migration time is achieved), but the amount of data transferred by the two threads are largely different at $R = 0.4$. This is because the bottleneck is the throughput of the HDD in the storage node, but not the networks. Thanks to the adaptive page cache transfer mechanism, our proposal can automatically achieve the shortest total migration time even though the bottleneck is located in different places depending on each case.

8. Discussion

8.1 IO Performance Improvement

To estimate the IO performance improvement by our mechanism compared to the existing mechanisms focusing on page cache (that is, dropping page cache before a migration for acceleration), we measured IO performance of a migrating VM with *File Read* workload. The workload repeatedly reads a large file and reports the throughput every 1 second. The size of the file is 2 GB and the memory size of the VM is 3 GB (the same as Table 1), thus the whole file is cached in the page cache after it is read once. We show that actual IO performance overhead with our mechanism is smaller than a lower bound estimation with the existing mechanisms, because no exsiting work provides the implementation.

Figure 9 shows the IO performance of File Read workload before, during, and after a migration with our mechanism. The x-asix shows the elapsed time, the y-asix on the left shows the actual throughput in blocks/s (solid line), and the y-asix on the right shows the accumulated lost throughput due to the migration overhead in blocks (dashed line). A block is 4 K bytes. The accumulated lost throughput at each x shows the total number of blocks that cannot be read by time x due to the migration overhead. Before the migration, the VM can read the file with its maximum throughput, which is around 108K blocks/s and we refer this value as M. During the migration, the throughput degrades due to the CPU and memory overhead incurred by the migration itself.

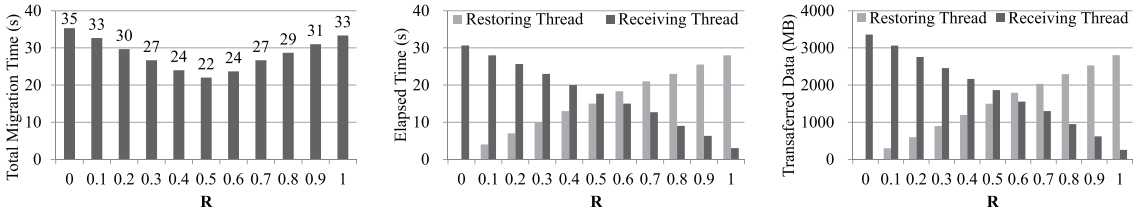

Fig. 6 Total migartion time, elapsed time and transferred data by the two communication threads in **WebServer** Workload.

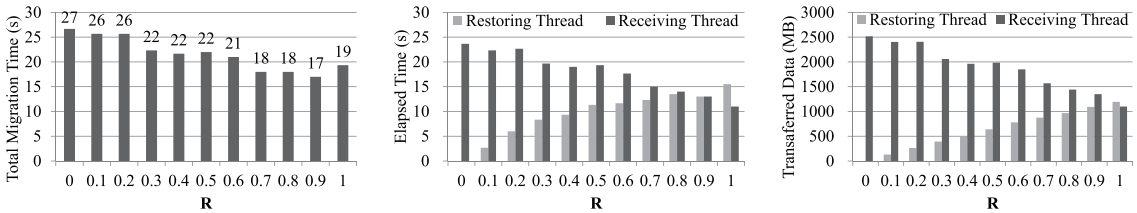

Fig. 7 Total migartion time, elapsed time and transferred data by the two communication threads in **Postmark** Workload.

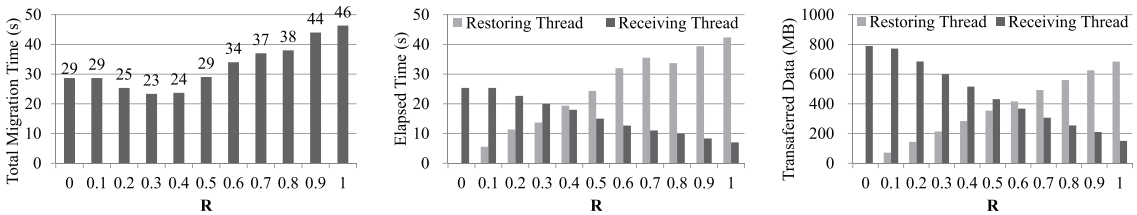

Fig. 8 Total migartion time, elapsed time and transferred data by the two communication threads in **TPC-C** Workload.

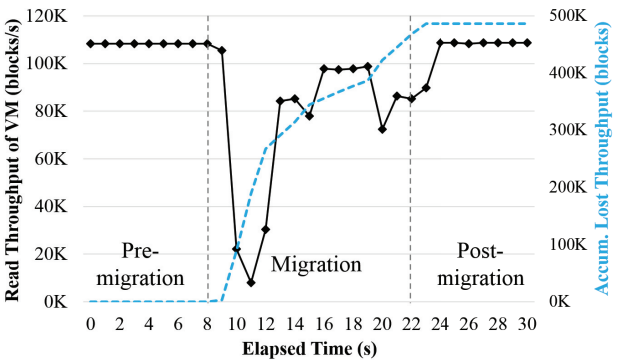

Fig. 9 Read throughput and accumulated lost throughpout of a file inside a VM before, during, and after a migration with our machanism. The throughput recovers to the max in 2 seconds after the migration. The accumulated lost throughput due to the migration overhead is 490 K blocks.

The migration finishes at $x = 22$ and the read throughput recovers to its maximum quickly (in 2 seconds at $x = 24$), thanks to the page cache transferred by our mechanism.

The accumulated lost throughput shows the total IO performance degradation through a migration. The value is around 490 K blocks in our mechanism. A lower bound of the accumulated lost throughput in the existing mechanisms is estimated as in Fig. 10. We first assume that dropping page cache makes total migration time to be zero, thus accumulated lost throughput during a migration is also zero. Note that this assumption does not break the lower-bound-ness of the estimation. We also assume that the file can be re-loaded from a disk without any bottleneck in the existing work. Then, the accumulated lost throughput is $T \times (M - m)$, where T and m are the time took and the throughput achieved to re-load the file from a disk. Table 1 shows the sequential read throughput of our HDD is 90MB/s, thus $m \approx 23$ K blocks/s and $T \approx 22$ sec. From these value we can get a lower bound: $22 \times (108 - 23) = 1870$ K blocks. This is 3.8 $\times$ larger than 490 K blocks and dividing the difference by M shows extra time the workload takes when migrated with the existing mechanisms: $(1870 - 490) \div 108 \approx 12.8$ seconds. Therefore we conclude that our mechanism improves the performance of File Read workload under a migration compared to the existing work. Because File Read is the most page-cache-access-intensive workload, the same applies to other workloads used in Sect. 7 as well.

8.2 Using TCP/IP for VMM/Broker Communication

Transferring PFNs of restorable page cache and disk block numbers with TCP/IP is the easiest method and applicable to any practical guest OS, but has overhead on total migration time. In the WebServer benchmark, the transfer takes 3

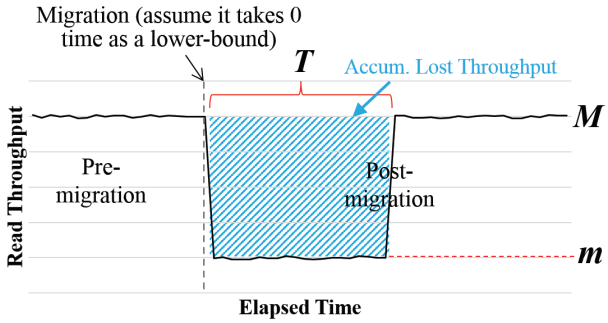

Fig. 10 Lower bound estimation of accumulated lost throughpout of a file inside a VM with existing mechanisms (dropping page cache before a migration). The read throughput degrades from M to m for T seconds after a migration, due to the loss of whole page cache. We assume the migration takes zero time, as no real implementation is available and this assumption does not break lower-bound-ness of the estimation.

Table 2 Comparison of methods to detect restorable page cache

	Kernel Module	VM Introspection	IO Monitoring
Implementation	*Easy*	Hard	Middle
Runtime Overhead	*None*	*None*	Small
Guest OS Limitation	Module Installed	Specific Version	*None*

seconds even though the size of data to transfer is just 6 MB. This is because the web sever inside the VM arises many interruptions to the network interface and the broker program cannot use the network functionalities efficiently. Without the web server, it takes less than 1 second to send the PFNs and disk block numbers even if the vCPU usage is 100%. The overhead was 1.6 seconds for Postmark workload and 1.2 seconds for TPC-C workload.

Mechanisms that do not use network to for communications between a host and a VM can be alternatives. Examples are Symbiotic Virtualization [24] and the shared memory space used in [6], although they require much implementation cost.

8.3 Page Cache Detection w/o Kernel Module

Installing a kernel module into the guest OS is the most feasible method to detect restorable page cache, although it is not the only one. The alternatives are VM introspection techniques and monitoring IO operations to/from the storage. VM introspection (e.g. [25]) allows the host OS to understand the memory content of a VM running on it. The host OS requires no help of the guest OS, but the guest OS kernel must be a specific version that the host OS expects.

IO monitoring in [8] captures all IO operations between the guest OS and external storage. The method is implemented in the layer of VMM and emulated hardware thus it required no modification to the guest OS. However, it incurs small overhead while the VM is executed because of the IO capturing.

Table 2 shows comparison between our method (kernel module), VM introspection, and IO monitoring. The best characteristics among each row are shown *italic*. Kernel module is the easiest to implement among three methods because the guest kernel knows everything about the guest OS thus the module has only 200 lines of code in C (error handling excluded). Runtime overhead means the overhead incurred by each method to the VM performance during non-migration time. The kernel module incurs literary no runtime overhead because the module is invoked just before a migration and it even does not need to be loaded into the kernel during non-migration time. This characteristic is highly important as described in the design criteria (Sect. 4.1).

The dependency of our kernel module on kernel functionalities is sufficiently small to support wide range of OSes/kernel versions. The module assumes that the in-kernel data for memory page management can be retrieved from a page frame number (with pfn_to_page in our implementation), and the disk block containing the same data as a memory page can be found directly from the kernel data (with bmap in our implementation). These functionalities are fundamental for the page cache mechanism itself, because flushing data cached in a memory page requires the location of the disk block into which the page is written. Therefore they are highly possible to be supported in future versions of Linux and other OSes as long as the page cache mechanism exists. Interface changes (e.g. function name, numbers/types of arguments) can occur more easily, but adopting the module to this type of changes is not difficult because the module has only 200 lines of code in C.

Table 3 Average cluster size of restorable page cache.

Workload	Average Cluster Size
WebServer	57.1
Postmark	124.3
TPC-C	18.9

8.4 Block Sequentiality

The total migration time in our method is largely affected by the sequentiality of disk blocks that contain restorable page cache. This is because random accessing to an HDD is much slower than sequential accessing. Therefore, when disk blocks to transfer are scattered across wide address range of the HDD, read throughput of the storage node becomes the bottleneck instead of the networks.

Table 3 shows sequentiality of disk blocks to transfer in the workloads used in the evaluation. For each workload, we calculated average cluster size of restorable page cache for the workload. A cluster means a set of sequential disk blocks within the disk blocks to transfer. For example, if the disk blocks to transfer are {1, 2, 55, 56, 100,101,102}, the average cluster size is $\frac{2+2+3}{3} \approx 2.3$. Larger cluster size results in better read throughput of the blocks. The results show that the average cluster size is the smallest in TPC-C workload and relatively the large in WebServer and Postmark workloads. These values obviously show why the re-

duction ratio of total migration time is not large under TPC-C workload. It can be expected that database-related workloads have small average cluster size in general and our proposal does not work efficiently for those workloads.

9. Related Work

Some studies force the guest OS to delete the page cache to accelerate live migration. Koto *et al.* discuss computational and time costs of live migration (referred as *migration noise*) in [6]. They reduce the migration noise by skipping the transfer of restorable pages including page cache, free pages, and kernel objects reconstrucable from other data. This method degrades IO performance of the VM due to the loss of page cache after a migration. Hines *et al.* also skips the transfer of page cache by using the balloon driver of Xen [7]. In a paravirtualization environment with Xen, a guest OS returns unused memory pages to the Xen using the balloon driver. Hence, Xen can skip the transfer of the deleted page cache in a live migration. This method also degrades the IO performance of the VM after a migration.

Transferring page cache from storage to achieve fast live migration without deleting the cache has been proposed [8], [9]. The main advantage of our work is that we proposed the adaptive page cache transfer. Existing studies do transfer restorable page cache via the SAN in parallel with normal data transfer via the GPN, but they use the GPN exclusively for normal data. To mitigate the load imbalance between the SAN and the GPN, [9] introduces *lazy fetch* mechanism, whose core idea is similar to well-known post-copy live migration. The lazy fetch mechanism switches the execution host of the VM as soon as transferring normal data is finished. After the execution host of the VM is switched, the remaining restorable page cache is fetched on demand in response to the memory accesses at the destination host. This mechanism has the same problem as post-copy live migration has: accesses to the page cache which is not transferred yet takes long time and degrades the overall performance of the workload running in the VM. Our adaptive page cache transfer mechanism mitigates the load unbalance without penalizing the IO performance thus more suitable for IO-intensive VMs than existing work.

There are many existing studies on live migration, but our technique is complementary with most of them. This is because our technique and many live migration researches optimize different phases of live migration as described in Sect. 2.2. For example, Svärd *et al.* decrease the amount of transferred data by transferring only the difference of a memory page and its previous version when the memory page is updated. This technique does nothing on the first phase of live migration, thus it can be combined easily with our system.

10. Conclusion

VMs running IO-intensive workloads suffer from long total migration time due to a large amount of page cache in its memory. Existing studies either force the guest OS to drop page cache (with great penalty to the application performance), or do not consider the dynamic characteristics of cloud data centers. We propose a parallel and adaptive page cache transfer mechanism to mitigate this problem, and our system shortens total migration time by up to 33.9% fast without forcing the guest OS to delete page cache nor manually tuning any parameters.

Acknowledgments

This work is supported in part by Japan Society for the Promotion of Science (JSPS) Grant in Aid for JSPS Fellows and KAKENHI (25330097).

References

[1] S. Akiyama, T. Hirofuchi, R. Takano, and S. Honiden, "Fast live migration with small io performance penalty by exploiting san in parallel," IEEE 7th International Conference on Cloud Computing (IEEE CLOUD), pp.40–47, 2014.
[2] C. Clark, K. Fraser, S. Hand, J.G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," Proc. USENIX Symposium on Networked Systems Design & Implementation (NSDI), pp.273–286, 2005.
[3] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M.D. Corner, "Memory buddies: exploiting page sharing for smart colocation in virtualized data centers," International Conference on Virtual Execution Environments (VEE), pp.31–40, 2009.
[4] N. Jain, I. Menache, J. Naor, and F.B. Shepherd, "Topology-aware vm migration in bandwidth oversubscribed datacenter networks," Proc. International Colloquium on Automata, Languages, and Programming (ICALP), vol.7392, pp.586–597, 2012.
[5] I. Goiri, F. Julia, R. Nou, J. Berral, J. Guitart, and J. Torres, "Energy-aware scheduling in virtualized datacenters," IEEE International Conference on Cluster Computing (CLUSTER), pp.58–67, 2010.
[6] A. Koto, H. Yamada, K. Ohmura, and K. Kono, "Towards unobtrusive vm live migration for cloud computing platforms," Asia-Pacific Workshop on Systems (APSys), 2012.
[7] M.R. Hines and K. Gopalan, "Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning," International Conference on Virtual Execution Environments (VEE), pp.51–60, 2009.
[8] C. Jo, E. Gustafsson, J. Son, and B. Egger, "Efficient live migration of virtual machines using shared storage," International Conference on Virtual Execution Environments (VEE), pp.41–50, 2013.
[9] C. Jo and B. Egger, "Optimizing live migration for virtual desktop clouds," IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp.1–8, 2013.
[10] M. Nelson, B.H. Lim, and G. Hutchins, "Fast transparent migration for virtual machines," Proc. Annual Conference on USENIX Annual Technical Conference (USENIX ATC), pp.391–394, 2005.
[11] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: the Linux Virtual Machine Monitor," Proc. Linux Symposium, pp.225–230, 2007.
[12] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," Proc. ACM Symposium on Operating Systems Principles (SOSP), pp.164–177, 2003.
[13] P. Svärd, B. Hudzia, J. Tordsson, and E. Elmroth, "Evaluation of delta compression techniques for efficient live migration of large virtual machines," International Conference on Virtual Execution Environment (VEE), pp.111–120, 2011.
[14] X. Zhang, Z. Huo, J. Ma, and D. Meng, "Exploiting data deduplication to accelerate live virtual machine migration," International

Conference on Cluster Computing (CLUSTER), pp.88–96, 2010.
[15] T. Wood, K.K. Ramakrishnan, P. Shenoy, and J. Van der Merwe, "Cloudnet: dynamic pooling of cloud resources by live wan migration of virtual machines," International Conference on Virtual Execution Environments (VEE), pp.121–132, 2011.
[16] M.R. Hines, U. Deshpande, and K. Gopalan, "Post-copy live migration of virtual machines," ACM SIGOPS Operating Systems Review, vol.43, no.3, pp.14–26, July 2009.
[17] T. Hirofuchi, H. Nakada, S. Itoh, and S. Sekiguchi, "Enabling instantaneous relocation of virtual machines with a lightweight vmm extension," IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid), pp.73–83, 2010.
[18] Cisco Systems, Inc., "Data center: Load balancing data center," Tech. Rep., 2004.
[19] P. Svard, J. Tordsson, B. Hudzia, and E. Elmroth, "High performance live migration through dynamic page transfer reordering and compression," International Conference on Cloud Computing Technology and Science (CloudCom), pp.542–548, 2011.
[20] J. Katcher, "Postmark: A new file system benchmark," Tech. Rep. TR3022, 1997.
[21] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer, "Benchmarking file system benchmarking: It *is* rocket science," USENIX conference on Hot topics in Operating System (HotOS), pp.1–5, 2011.
[22] "TPC-C." http://www.tpc.org/tpcc/
[23] "Sigmod'97 industrial session 5 standard benchmarks for database systems." http://www.tpc.org/information/ sessions/sigmod/ sigmod97.ppt
[24] J.R. Lange and P. Dinda, "Symcall: Symbiotic virtualization through vmm-to-guest upcalls," International Conference on Virtual Execution Environments (VEE), pp.193–204, 2011.
[25] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction," Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS), pp.128–138, 2007.

Soramichi Akiyama is a researcher at Artificial Intelligence Research Center, National Institute of Advanced Industrial Science and Technology (AIST), Japan. He received a Ph.D. degree from Graduate School of Information Science and Technology, The University of Tokyo in 2015. His research interests include operating systems, virtualization technologies, and high performance computing targetting artificial intelligance applications.

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.

Ryousei Takano received his Ph.D. degree from Tokyo University of Agriculture and Technology in 2008. He joined AXE, Inc. in 2003. He joined Institute of Advanced Industrial Science and Technology (AIST) in 2008. He is currently a research group leader of AIST. His research interests include operating systems, high performance networking, and distributed parallel computing. He is a member of ACM, IEEE, and IPSJ.

Shinichi Honiden received the PhD degree in electrical engineering from Waseda University, Tokyo, Japan, in 1986. From 1978 to 2000, he was working at Toshiba Corporation. Since April 2000, he has been a Professor and a Director of the Information Systems Architecture Research Division at the National Institute of Informatics (NII), Tokyo, Japan. Since 2012, he has been a Deputy Director General at NII. He is also a Professor in the Graduate School of Information Science and Technology at The University of Tokyo, Tokyo, Japan. His research interests include agent technology, pervasive computing, and software engineering. He is a member of the IEEE.