Fast Live Migration with Small IO Performance Penalty by Exploiting SAN in Parallel

Soramichi Akiyama*, Takahiro Hirofuchi†, Ryousei Takano† and Shinichi Honiden*‡
*Graduate School of Information Science and Technology, The University of Tokyo, Japan
Email: {akiyama, honiden}@nii.ac.jp
†National Institute of Advanced Industrial Science and Technology, Tsukuba, Ibaraki, Japan
Email: {t.hirofuchi, takano-ryousei}@aist.go.jp
‡National Institute of Informatics, Chiyoda, Tokyo, Japan

***Abstract*—Virtualization techniques greatly benefit cloud computing. Live migration enables a datacenter to dynamically replace virtual machines (VMs) without disrupting services running on them. Efficient live migration is the key to improve the energy efficiency and resource utilization of a datacenter through dynamic placement of VMs. Recent studies have achieved efficient live migration by deleting the page cache of the guest OS to shrink the memory size of it before a migration. However, these studies do not solve the problem of IO performance penalty after a migration due to the loss of page cache. We propose an advanced memory transfer mechanism for live migration, which skips transferring the page cache to shorten total migration time while restoring it transparently from the guest OS via the SAN to prevent IO performance penalty. To start a migration, our mechanism collects the mapping information between page cache and disk blocks. During a migration, the source host skips transferring the page cache but transfers other memory content, while the destination host transfers the same data as the page cache from the disk blocks via the SAN. Experiments with web server and database workloads showed that our mechanism reduced total migration time with significantly small IO performance penalty.**

***Keywords*-live migration; virtualization; cloud performance**

I. Introduction

Cloud computing has become a major computing paradigm used both for enterprise and personal purposes. Virtualization techniques are highly important as building blocks of the cloud computing paradigm. A cloud datacenter achieves easy maintenance, good isolation across users, and high degree of resource utilization thanks to the them. Live migration [1] enables a datacenter to dynamically replace virtual machines (VMs) without disrupting the services running on them. For example, high memory utilization [2], load balancing [3] and low energy consumption [4] are realized by dynamic VM placement using live migration. Efficient live migration is the key technique to apply these studies to real-world datacenters.

Page cache is a widely-adapted mechanism to improve performance of disk IO operations. It is equipped in many modern operating systems such as Linux, Windows and BSDs. A VM running a workload that treats large data has large amount of page cache. This prolongs total migration time of live migration thus it must be approached to achieve efficient live migration. Recent studies [5], [6] shortens total migration time by forcing the guest OS to delete the page cache before a migration to shrink the memory size of the VM. However, their techniques greatly reduces the IO performance of the VM after a migration due to the loss of the page cache.

We propose an advanced memory transfer mechanism for live migration, which skips transferring the page cache to shorten total migration time while restoring it transparently from the guest OS via the SAN to prevent IO performance penalty. To start a migration, our mechanism collects the mapping information between page cache and disk blocks. During a migration, the source host skips transferring the page cache but transfers other memory content, while the destination host transfers the same data as the page cache from the disk blocks via the SAN. Our technique shortens total migration time by simultaneously utilizing the SAN along with the general purpose network and achieves smaller IO performance penalty by transparently transferring the page cache rather than deleting it.

Our experiments using with web server and database workloads showed that our method reduced total migration time with significantly smaller IO performance penalty because the page cache is kept alive during a migration. We also give mathematical analysis of the method and validate the analysis using the experimental results.

This paper is structured as follows. Section II explains background. Section III refers related work. Section IV shows the overview of our method and how it reduces total migration time. Section V gives the details of our technical contributions. Section VI explains the implementation details. Section VII shows the evaluation results. Section VIII gives discussions and Section IX concludes the paper.

II. Background

A. Dynamic VM Placement

Dynamic optimization of VM placement, or simply dynamic VM placement, plays an important roles in the overall performance of a cloud datacenter, such as energy consumption, resource utilization, and load balancing. Usage pattern and load of the VMs on a cloud are highly dynamic and unpredictable, thus dynamic VM placement is mandatory.

This work is supported in part by KAKENHI (25330097) and Grant-in-Aid for JSPS Fellows.

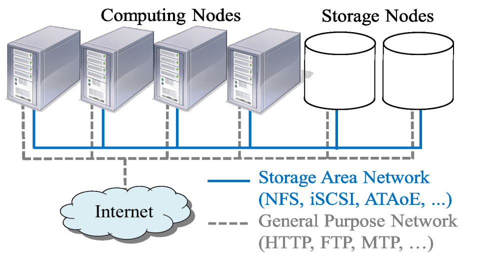

Fig. 1. Network Architecture of Cloud Datacenter: Storage nodes are connected with a designated storage area network (SAN).

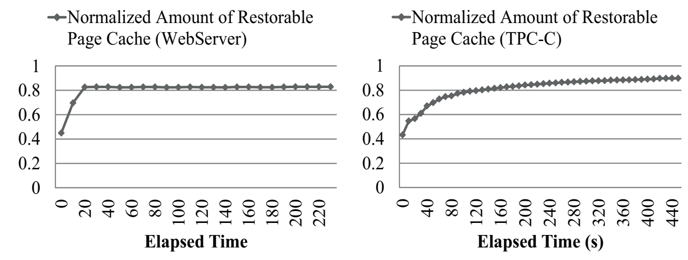

Fig. 2. Amount of Restorable Page Cache in the Memory with WebServer Workload (left) and TPC-C Workload (right), Normalized by the Total Memory Usage of the VM.

Examples of dynamic VM placement are as follows: VMs that have similar memory content are dynamically consolidated to share the identical memory pages and reduce the overall memory usage of the datacenter [2]; overloaded VMs are packed into under-utilized hosts while taking the network topology into account [3]; VMs under low load are live migrated into small number of physical hosts to turn off spare hosts and reduce the energy consumption [4].

Efficient live migration is a mandatory technique to realize dynamic VM placement, because it must have as little interference as possible to the services running on the VMs. We achieve efficient live migration when the target VM is running a workload with large data and has large amount of page cache. Our mechanism utilizes the storage area network (SAN) equipped in a datacenter to simultaneously transfer restorable page cache and other data inside the VM memory. The descriptions on the network links and restorable page cache are given in the following subsections.

B. Network Architecture of Cloud Datacenter

Figure 1 illustrates a simplified view of the network architecture of a typical cloud datacenter. The main point is that storage nodes are connected with a designated SAN along with a general purpose network. CISCO suggests a datacenter networking architecture that includes Storage Networking and Business Continuance Networking [7]. Nodes might have another link for management purposes. Descriptions of each node are as follows:

Storage Area Network (SAN): It is used to communicate with the storage nodes in the datacenter. A shared filesystem is built on top of this link and IO requests and data from/to the storage nodes go through this link. An important notice is that we do not assume IP-capability of this link. The shared filesystem can be built with any networking such as Infiniband or Ethernet without IP.

General Purpose Network: 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 datacenter go through this link.

C. Memory Redundancy Between Disk

A VM running a workload with large data can have many memory pages identical to disk blocks due to *restorable page cache*. An operating system uses as many free memory pages as possible for page cache. Page cache is an on-memory cache mechanism to hide the gap between the accessing speed of memory and storage and is implemented in many modern operating systems such as Linux, Windows and BSDs. When an IO operation is requested for a disk block, the read/written data is stored in the page cache to accelerate future requests for the same disk block. Restorable page cache refers memory pages whose data can be restored from the identical disk blocks even if it is deleted. Note that memory pages containing write-cache which has not yet been flushed are not restorable.

Figure 2 shows the amount of restorable page cache contained in the memory of a VM running a workload with large amount of data. 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. The VM has 3 GB of memory and the total size of the web contents is also 3 GB. TPC-C is a workload that simulates the typical database access pattern for an online shopping web site. The VM has 1 GB of memory and the total size of the database is 1.9 GB. The detailed descriptions of WebServer and TPC-C workloads are given in Section VII. The figure shows that both VMs have many memory pages (more than 80% of all pages) 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.

III. Related Work

It has been pointed out that the large amount of page cache slows down total migration time thus it must be approached. Koto *et al.* discuss computational and time costs of live migration (referred as *migration noise*) in [5]. The work reduces the migration noise by skipping the transfer of restorable pages including page cache, free pages, and kernel objects that can be regenerated 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 the page cache by using the balloon driver of Xen [6]. 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 because the VM must reload the deleted page cache from the disk.

Transferring page cache from storage to achieve fast live migration has been proposed [8], [9]. The main difference between this paper and these studies is that we simultaneously utilize the SAN and the general purpose network in a datacenter and it makes our proposal more widely applicable. Their weakness is that they assume transferring the restorable page cache from storage is much faster than transferring it via a normal migration network. They do not simultaneously use the SAN and the general purpose network thus they must use a faster channel to transfer restorable page cache to accelerate live migration. Authors of [9] also proposes using the SAN and the general purpose network in a datacenter in parallel [10]. The difference of this paper and [10] is that we transfer a portion of restorable page cache via the SAN to minimize the total migration time, while they transfer the all via the SAN.

Our mechanism and many existing studies optimize different phases of live migration thus they can be integrated and used together. Pre-copy live migration [1] has three phases: (1) copying all the memory pages at first, (2) iteratively copying the updated memory pages during the 1st and 2nd phases, and (3) copying small number of remaining memory pages while the VM is suspended. The 1st and 2nd phases are equally important as they transfer many memory pages, thus integrability of our proposal to existing research is an important characteristic. Our mechanism focuses on reducing the transferred memory of the 1st phase of live migration, while many existing studies [11]–[13] focus on the 2nd phase. Post-copy live migration [14], [15] completely eliminates the iterative copying thus it is also an optimization for the 2nd phase. Our method benefits the post-copy mechanisms as well because they do no optimization on the 1st phase.

IV. Proposal

A. Overview

We propose an advanced memory transfer mechanism that exploits restorable page cache and the network architecture explained in Section II. It has three features: (1) It accelerates live migration by exploiting the SAN of a datacenter to transfer restorable page cache. (2) It divide restorable page cache into two parts to fully utilize the general purpose network when almost all memory pages are restorable. (2) It achieves smaller IO performance penalty to the VM after a migration than existing research by keeping the page cache warmed up transparently from the guest OS.

Feature (1) is advanced compared to merely bonding two IP networks for faster migration. Our proposal is applicable without any changes to the underlying network settings even if the SAN uses Infiniband or Ethernet because transferring restorable page cache is handled by the shared filesystem (further description in Section VI-D).

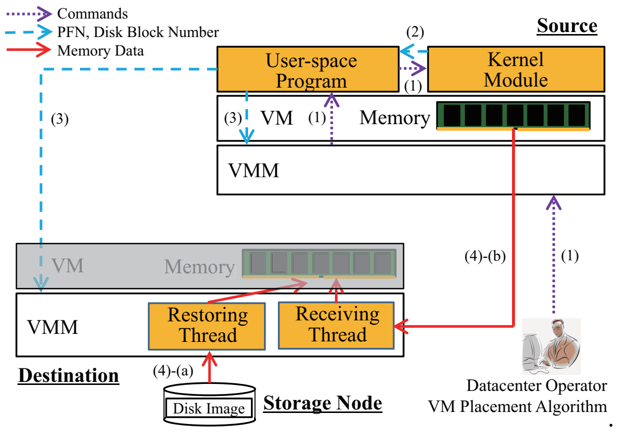

Fig. 3. Procedure of Live Migration with Our Mechanism

B. Migration Procedure with Our Mechanism

The procedure of a live migration with our mechanism is illustrated in Figure 3. A VM is being migrated from the source (top-right) to the destination (bottom-left). The VM on the destination is not running yet thus it is grayed out. The dotted arrows show commands, the dashed ones show metadata to realize our method, and the solid ones show transfers of memory data. Detailed descriptions are as follows:

1) The source VMM receives a command (from a datacenter operator or a dynamic VM placement algorithm) to execute a migration. It requests our user-space program inside the VM to get the page frame numbers (PFNs) of the restorable page cache and the block numbers of the identical disk blocks. Then the user-space program requests our kernel module to fetch them.
2) The kernel module detects the PFNs of the restorable page cache and the identical disk blocks.
3) The user-space program sends the PFNs and the block numbers to the source and destination VMMs.
4) The memory transfers are kicked-off. The SAN is exploited to accelerate live migration.
    a) Restoring Thread in the destination VMM transfers the restorable page cache from the disk image via the SAN.
    b) Receiving Thread in the destination VMM receives the other memory pages from the source VMM via the general purpose network.
5) Memory pages updated during above steps are transferred again via the general purpose network.
6) Once the amount of remaining memory becomes sufficiently small and all the restorable page cache is copied, the execution host of the VM is switched.

C. Dividing Page Cache

A portion of restorable page cache is transferred via the general purpose network, not via the SAN. This prevents underutilization of the general purpose network when almost all memory pages contain restorable page cache. A variable 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 general purpose network.

The optimal R that minimizes the total migration time depends on four factors: the amount of restorable page cache M_p, the amount of normal memory M_n, the throughput of transferring restorable page cache S_p, and the throughput of transferring normal memory S_n. The optimal R satisfies:

$$\frac{RM_p}{S_p} = \frac{M_n + (1-R)M_p}{S_n} \tag{1}$$

because both transfers must end at the same time to utilize the SAN and the general purpose network equally. Therefore, the optimal R is given by:

$$R \quad = \quad \frac{M_p + M_n}{M_p} \times \frac{S_p}{S_p + S_n} \tag{2}$$

We justify the equations above in Section VIII using the measured values in our experiments.

V. Simultaneous Transfers

A. Overview

The technical contribution of this paper is how to realize simultaneous transfers. It has two challenges and they are explained in the following subsections.

Memory Consistency: Transfers of normal memory and restorable page cache are done while the VM keeps running. This causes updates to restorable page during a migration. Updated restorable page cache must be detected and transferred as normal memory pages because there is not guarantee that the pages are flushed into the disk image.

Writing Algorithm: It is possible that one memory page is written by the two threads. The writing algorithm must be carefully designed to deal with this case.

B. Memory Consistency

The memory consistency of the VM during a migration must be carefully dealt with. The issue is that a memory page containing restorable page cache can be updated and can turn into non-restorable. Suppose a memory page is updated during a migration after it has been detected as restorable page cache. In this case, the updated and latest data must be transferred via the general purpose network because there is no guarantee that the updated data has been flushed into the disk. There are two cases in which a memory page used for the restorable page cache is updated: the cached data contained in the page is updated or the guest OS frees the page and use it for a purpose other than page cache because of 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.

C. Writing Algorithm

The simultaneous transfers are implemented by two threads and a buffer in the destination VMM. Receiving Thread receives and buffers normal memory pages transferred via the general purpose network. The buffer is flushed into the VM memory once it is filled. Restoring Thread fetches the restorable page cache from the storage node via the SAN and copies the data into the VM memory without buffering.

A lock mechanism is used to deal with two simultaneously transfers because the two threads can write to the same memory page. This happens when a memory page on the source host is updated after it has been detected to be restorable page cache as described in Section V-B. The destination VMM has a *received flag* and a mutex for each memory page. The memory footprint during a migration by the flags and mutex is enough small because a flag is 1 byte and a mutex is 40 bytes (in Linux pthread implementation) and the sum is 4 % of the size of a memory page. The flags and mutex are no longer required after a migration thus there is no extra memory required during non-migration time. The working algorithms of Receiving thread and Restoring thread are as follows:

Receiving Thread: It tries to acquire the lock for a memory page before writing to the page. If the lock cannot be acquired it processes the next memory page in the buffer (or the first memory page if it reaches to the end of the buffer). If the lock is acquired, it enables the received flag for the page and copies the transferred data into the VM memory, and then releases the lock. The buffer prevents the receiving thread from being blocked upon a lock conflict.

Restoring Thread: It also tries to acquire the lock for the page before writing to a memory page. If the lock cannot be acquired, it skips processing the page because a lock conflict means that the updated and latest data is being written to the memory page by the receiving thread. If the lock is acquired and the received flag is disabled, it copies the identical disk block to the memory page, and then releases the lock.

VI. Implementation

A. Components

Our system is implemented with three components. The details of each component are described in the following subsections. A **kernel module** detects the PFNs of the memory pages containing restorable page cache and the block numbers of the identical disk blocks to the memory pages. A **user-space program** sends the PFNs and the disk block numbers to the modified VMMs. **Modified VMMs** transfer the restorable page cache via the SAN, and at the same time transfers the other memory pages via the general purpose network.

B. Detecting Restorable Page Cache with Kernel Module

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 parameter and returns `struct page` kernel data of a memory page whose PFN is the 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`.

C. User-Space Program

Our user-space program works as a broker between the VMM and the kernel module. When a migration is invoked, the VMM in the source host sends a value R and a request to get PFNs of restorable page cache and the identical disk block numbers. The user-space program receives them and invokes an `ioctl` operation of the kernel module with the given R. Once it gets the PFNs and the identical disk block numbers from the module, it sends them to the VMMs in the source and destination hosts.

The mapping information sent from the user-space program to the VMMs is an array of disk block numbers indexed by memory page numbers, that is, the n^{th} disk block number in an array describes information about the n^{th} memory page of the VM. A non-zero number represents the disk block number containing the identical data to the memory page. A zero in an array means that the memory page is not restorable. For example, an array $\{1234, 0, 10, 0, ...\}$ means that the 1234^{th} disk block has the identical data to the 1^{st} memory page and the the 10^{th} disk block has the identical data to the 3^{rd} memory page, while the 2^{nd} and 4^{th} memory pages are not restorable, and so forth. The size of an array is given by $8 \times \mathrm{NumberOfMemoryPages}$ because a disk block number is represented with an 8-byte unsigned integer in modern Linux. If a VM has 4 GB of memory, the size of the mapping information of this VM is: $8 \times (4\ \mathrm{GB} \div 4\ \mathrm{KB/page}) = 8\ \mathrm{MB}$.

D. Modified VMM

We modify QEMU/KVM to add three functionalities. First, it has a new migration command that accepts the IP address of the VM and the value R along with the IP address of the destination hosts. Second, it communicates with the user-space program inside the target VM to fetch the PFNs of restorable page cache and the identical disk block numbers before it start transferring the VM memory. Third it invokes two threads to simultaneously receives the restorable page cache and the other memory pages at the destination.

TABLE I
EVALUATION ENVIRONMENTS

	HDD Cluster	*SSD Cluster*
CPU	Intel Xeon X5460	Intel Xeon 5160
Memory	8 GB	12 GB
Storage	256 GB **HDD** Read: 90 MB/s	512 GB **SSD** Read: 170 MB/s
Network	1 Gbps NIC $\times$ 3	1 Gbps NIC $\times$ 2
Host OS	Debian GNU/Linux 6.0.5	Cent OS 6.3
Guest OS	Debian GNU/Linux 6.0.5	
VMM	QEMU 0.13.0, KVM 2.6.32	

Retrieving the PFNs of restorable page cache and the identical disk block numbers uses our kernel module and our user-space program installed inside the VM. This requires users of VMs to install them, but we believe this requirement is lightweight because of two reasons. First, the kernel module and the user-space program are simple enough (both have approximately 200 lines of code in C) so that the users can verify our modules are innocent. Second, our method helps not only cloud providers but also datacenter users. Fast live migration achieves lower energy consumption (resulting in lower pricing) and faster load balancing.

Receiving thread and Restoring Thread are invoked to achieve simultaneous transfers. Receiving thread receives normal memory pages via the general purpose network, and Restoring thread transfers restorable page cache via the SAN. Restoring Thread uses normal read/write system calls to fetch the restorable page cache, therefore our implementation does not require any change to the underlying network settings. Normal read/write are automatically rerouted by the underlying filesystem because disk images are on a shared filesystem (such as NFS, ATAoE, and iSCSI) as normally done in cloud datacenters. This means that our mechanism can be applied even if the target datacenter uses Infiniband for the SAN while using IP for the general purpose. On the other hand, merely bonding the SAN and the general purpose network for faster migration requires IP-capability of the SAN and changes to existing datacenter network settings.

VII. EVALUATION

A. Total Migration Time: Methodology

This subsection describes the environments and workloads used for evaluations. Two different environments, *HDD Cluster* and *SSD Cluster*, are used in the evaluation. The detailed specifications of the environments are in Table I. Both clusters have at least two 1 Gbps network interface cards (NICs), used for the SAN and the general purpose network. The main difference between the two clusters is that HDD Cluster uses a hard disk drive (HDD) as the storage of each node, while SSD Cluster uses a solid state drive (SSD). The read throughput of the storages are measured using bonnie++ in the host OSes. The VMM is composed of QEMU 0.13.0 and KVM 2.6.32. KVM is the default version of the both host OSes.

The total migration time is measured across R ranging from 0 to 1 with 0.1 interval; i.e. $R \in \{0, 0.1, 0.2, ..., 0.9, 1\}$. All values shown in the evaluation are averaged over three runs. Each measurement uses three nodes from a cluster: 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 time consumed by the receiving thread and the restoring thread to transfer memory pages are also measured for detailed analysis.

The evaluation is conducted with two workloads: WebServer and TPC-C. **WebServer** is a workload that simulates a web server under high load. HDD Cluster is used for this workload. Apache web server has static files without database.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. The load generator runs on a designated host (not the same neither as source nor destination) and accesses the files via the third NIC of the HDD Cluster 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. The VM is configured to have 3 GB of memory, 1 vCPU and 20 GB of virtual disk.

TPC-C [16] is a workload that measures the performance of a database system. It generates database access patterns that simulates an online shopping web site. The total size of content of the database is 1.9 GB. TPC-C was executed using MySQL on SSD Cluster. SSD Cluster is selected because SSD is widely used to store or cache database contents to achieve high IO performance. The migration is executed 270 seconds after the the workload started, where the warming up phase of TPC-C has been finished. The VM is configured to have 1 GB of memory, 1 vCPU and 20 GB of virtual disk.

B. Total Migration Time: WebServer

The left figures of Figure 4 and Figure 5 show the total migration time of a VM running WebServer workload. The x-axis of each figure shows a value of R and the y-axis shows the total migration time for the R. Figure 4 shows the results when the network bandwidths are not limited, thus they have 1 Gbps bandwidths. Figure 5 shows the results when the network bandwidths are limited to 500 Mbps with `tc` command on the hosts. The 1 Gbps environment assumes that there is no interference on the network links between the source and the destination hosts. The 500 Mbps environment assumes that the network links cannot be fully occupied for the migration because the hosts and the networks are shared resource in a cloud datacenter.

The reduction ratio of total migration time achieved by our mechanism is calculated as follows. Let $TM(r)$ be the total migration time when $R = r$ and p be the value of R that achieves the shortest total migration time. The reduction ratio of total migration time by our method is calculated by:

$$1 - \frac{TM(p)}{TM(0) - Overhead} \tag{3}$$

The overhead refers the time consumed for the steps 1–3 described in Section IV-B. Our implementation works exactly the same as the original QEMU/KVM after the step 4 when $R = 0$. It is approximately 3 seconds in WebServer workload in both environments (regardless of R). Further discussion about the overhead is given in Section VIII-B.

The reduction ratio of the total migration time in the 1 Gbps and in the 500 Mbps environments are 13% ($p = 0.3$, $TM(p) = 28$) and 32% ($p = 0.5$, $TM(p) = 38$), respectively.

The middle figures of Figure 4 and Figure 5 show the elapsed time consumed by Receiving Thread and Restoring Thread. The x-axis shows a value of R and the y-axis shows the elapsed time of each thread for the R. Figure 4 shows the results when the network bandwidths are not limited. Figure 5 shows the results when the network bandwidths are limited to 500 Mbps. The blue (or light) bars are for Receiving Thread and the red (or dark) bars are for Restoring Thread. The shortest time is achieved when two bars have almost the same lengths because it means that the SAN and the general purpose network are fully utilized throughout the migration.

The right figures of Figure 4 and Figure 5 show the amount of data transferred by each thread. The blue (or light) bars are for Receiving Thread, the red (or dark) bars are for Restoring Thread, and the black line shows the sum of amounts of data transferred by both threads. Figure 4 shows the results when the network bandwidths are not limited. Figure 5 shows the results when the network bandwidths are limited to 500 Mbps. The sums are roughly the same across R, but they are not exactly the same because memory pages updated during a migration are transferred more than twice.

Summary: Our mechanism reduces total migration time of a VM running a web server under high load. The reduction ratios are 32% and 13% when a migration can use 500 Mbps and 1 Gbps bandwidth respectively.

C. Total Migration Time: TPC-C

The left figure of Figure 6 shows the total migration time of a VM running TPC-C workload. The bandwidth of each network link is limited to 500 Mbps to emulate the interference from other VMs on the datacenter. The x-axis shows a value of R and the y-axis shows the total migration time for the R.

The total migration time is minimized with $R = 0.1$. Even in this case, the reduction of the total migration time achieved by our method is: $1 - \frac{19}{21-1} = 5\%$. Note that in TPC-C workload overhead to retrieve PFNs and disk block numbers is approximately 1 second, which is shorter than in WebServer workload (details in Section VIII-B).

The small reduction ratio is because the read throughput of the restorable page cache from the SSD is much slower than the transferring throughput of normal memory. The throughput are calculated from the middle and right figures of Figure 6. For example, when $R = 0.6$ the read throughput via the SSD is around 17 MB/s while the transferring throughput of normal memory is around 58 MB/s. Table I shows the maximum read throughput of our SSD is 170 MB/s, but the 10X gap (17 MB/s vs 170 MB/s) is because the disk blocks containing

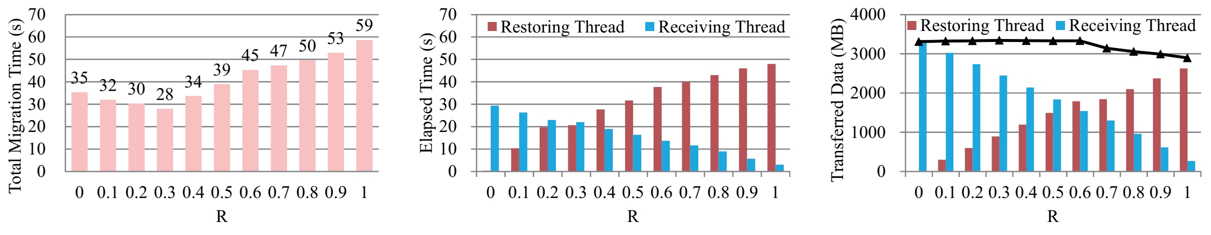

Fig. 4. Evaluation Results with WebServer Workload in 1 Gbps Environment. Left: Total Migration Time, Middle: Elapsed Time Consumed by Receiving Thread and Restoring Thread, Right:Amount of Data Transferred by Each Thread (red and blue bars) and in Total (black line).

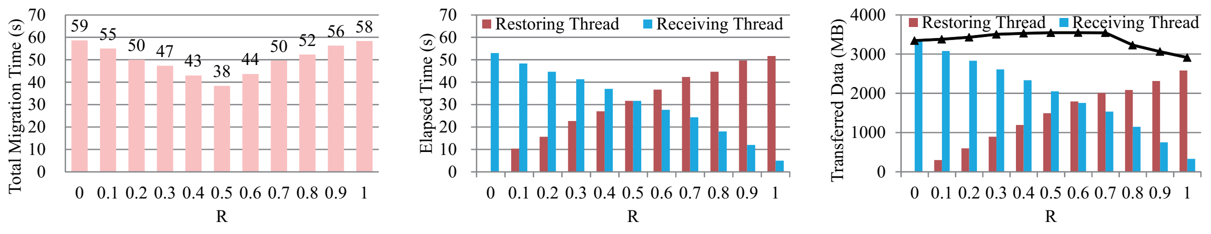

Fig. 5. Evaluation Results with WebServer Workload in 500 Mbps Environment. The Network Bandwidths are Limited to 500 Mbps to Emulate Interference from other VMs Running on the Datacenter. Left: Total Migration Time, Middle: Elapsed Time Consumed by Receiving Thread and Restoring Thread (Middle), Right: Amount of Data Transferred by Each Thread (red and blue bars) and in Total (black line).

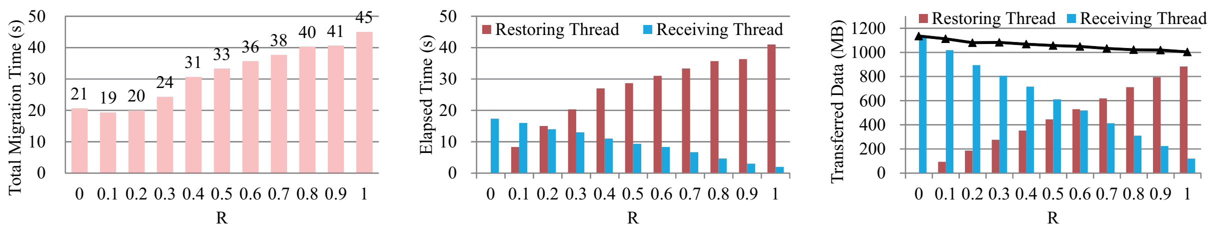

Fig. 6. Evaluation Results with TPC-C Workload in 500 Mbps Environment. The Network Bandwidths are Limited to 500 Mbps to Emulate Interference from other VMs Running on the Datacenter. Left: Total Migration Time, Middle: Elapsed Time Consumed by Receiving Thread and Restoring Thread, Right: Amount of Data Transferred by Each Thread (red and blue bars) and in Total (black line).

the restorable page cache are scattered across a wide address range on the SSD. It is not the case in WebServer workload because a file is not split when there are enough sequential blocks to store. Random accessing is slower than sequential accessing even for a SSD because the internal buffer does not work efficiently. We believe faster storage devices such as Fusion-io improve the efficiency of our mechanism, but we did not conduct further experiments due to equipment limitations.

Summary: Our mechanism reduces total migration time of a VM running TPC-C workload by 5% under 500 Mbps bandwidth. Faster storage devices should improve the ratio.

D. Small IO Performance Penalty to the VM

Our mechanism has small IO performance penalty to the VM after a migration because it keeps the page cache warmed up transparently from the VM. Figure 7 shows the file read throughput of a VM on the HDD cluster. The x-axis shows the elapsed time in second from the beginning and the y-axis shows the throughput in blocks/second. The VM with 3GB of memory is migrated using our method with $R = 0.5$ during $50 < x < 62$ to measure the IO performance penalty by our method. The file is 2 GB large and cached in the page cache

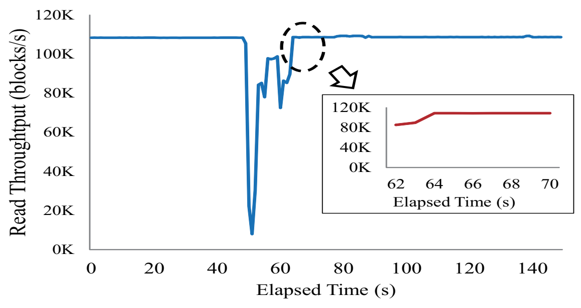

Fig. 7. File Read Throughput of a VM Reading a Large File. Blue line shows the result of the whole measurement and red line shows enlarged result for $62 \leq x \leq 70$. The VM is migrated during $50 < x < 62$ by our method. The throughput recovers to the maximum 2 seconds after the migration completes ($x = 64$). The degradation during the migration is out of our scope.

before the measurement. Once the end of the file is reached, the file is read from the head again. The blue line is the result of the whole measurement and the red line is the result during 9 seconds after the migration completes ($62 \leq x \leq 70$).

The read throughput fully recovers 2 seconds after the migration completes, because there is no loss of page cache after the migration in our method. We did not conduct the same measurement with existing methods because the implementations are not provided, but deleting page cache causes much larger IO performance penalty because reading a 2 GB file not on page cache takes 14 seconds in our HDD cluster. The degradation during the migration ($50 < x < 62$) is a general phenomenon of migration [17] and out of our scope.

Summary: Our mechanism has negligible IO performance penalty to a migrated VM. File read throughput of a migrated VM fully recovers 2 seconds after a migration completes.

VIII. Discussion

A. *Verification of Theory using Experiment Results*

The evaluation results match the theory described in Section IV-C. We describe validation steps using the measurement values with the WebServer workload in 1 Gbps environment. First, M_n and M_p are approximately 88 MB and 2930 MB, respectively. These are not estimated but the actual values. M_n is retrieved via `/proc/meminfo` inside the VM and M_p is retrieved by our kernel module. Second, S_n and S_p are estimated as 110 MB/s and 60 MB/s, respectively. They are estimated from the measured values in Section VII-B.

The equation (2) and M_n, M_p, S_n, S_p above give the theoretical optimal of R.

$$R = \frac{M_p + M_n}{M_p} \times \frac{S_p}{S_p + S_n} \approx 0.36$$

This almost matches the evaluation results where the minimum migration time is achieved with $R = 0.3$.

B. *Sending PFNs with TCP/IP*

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 performance disadvantage. In the WebServer benchmark, the transfer takes 3 seconds even though the size of data to transfer is 6 MB. This is calculated by multiplying the number of memory pages with 8 (in Linux a disk block number is 8-bytes long). This is because the web sever inside the VM arises many hardware interruptions to the network interface and our user-space 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%. Mechanisms that do not use network to for communications between a host and a VM can be alternatives. Examples are Symbiotic Virtualization [18] and the shared memory space used in [5], although they require much implementation cost.

IX. Conclusion and Future Work

Efficient live migration is a key to improve dynamic VM placement of cloud datacenters. Recent studies tackled a problem that a VM running workloads with large data has large amount of page cache in its memory, which makes live migration slow down. We propose an advanced memory transfer mechanism for live migration, which shortens total migration time of a VM having large amount of page cache with smaller IO performance penalty than existing studies. The experiments showed that our method shortened total migration time with significantly small IO performance penalty. Future work includes automatic tuning of R and integration of our method with dynamic VM placement algorithms.

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," in *USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, 2005, pp. 273–286.

[2] 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," in *International Conference on Virtual Execution Environments (VEE)*, 2009, pp. 31–40.

[3] N. Jain, I. Menache, J. S. Naor, and F. B. Shepherd, "Topology-aware vm migration in bandwidth oversubscribed datacenter networks," in *International Colloquium Conference on Automata, Languages, and Programming (ICALP)*, 2012, pp. 586–597.

[4] I. Goiri, F. Julia, R. Nou, J. Berral, J. Guitart, and J. Torres, "Energy-aware scheduling in virtualized datacenters," in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2010, pp. 58–67.

[5] A. Koto, H. Yamada, K. Ohmura, and K. Kono, "Towards unobtrusive vm live migration for cloud computing platforms," in *Asia-Pacific Workshop on Systems (APSys)*, 2012.

[6] M. R. Hines and K. Gopalan, "Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning," in *International Conference on Virtual Execution Environments (VEE)*, 2009, pp. 51–60.

[7] Cisco Systems, Inc., "Data center: Load balancing data center," Tech. Rep., 2004. [Online]. Available: https://learningnetwork.cisco.com/docs/DOC-3438

[8] S. Akiyama, T. Hirofuchi, R. Takano, and S. Honiden, "Fast wide area live migration with a low overhead through page cache teleportation," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2013, pp. 78–82.

[9] C. Jo, E. Gustafsson, J. Son, and B. Egger, "Efficient live migration of virtual machines using shared storage," in *International Conference on Virtual Execution Environments (VEE)*, 2013, pp. 41–50.

[10] C. Jo and B. Egger, "Optimizing live migration for virtual desktop clouds," in *IEEE International Conference on Cloud Computing Technology and Science (Cloudcom)*, 2013, pp. 1–8.

[11] P. Svärd, B. Hudzia, J. Tordsson, and E. Elmroth, "Evaluation of delta compression techniques for efficient live migration of large virtual machines," in *International Conference on Virtual Execution Environment (VEE)*, 2011, pp. 111–120.

[12] X. Zhang, Z. Huo, J. Ma, and D. Meng, "Exploiting data deduplication to accelerate live virtual machine migration," in *International Conference on Cluster Computing (CLUSTER)*, 2010, pp. 88–96.

[13] 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," in *International Conference on Virtual Execution Environments (VEE)*, 2011, pp. 121–132.

[14] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy live migration of virtual machines," *ACM SIGOPS Operrating Systems Review*, vol. 43, no. 3, pp. 14–26, Jul. 2009.

[15] T. Hirofuchi, H. Nakada, S. Itoh, and S. Sekiguchi, "Enabling instantaneous relocation of virtual machines with a lightweight vmm extension," in *IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, 2010, pp. 73–83.

[16] "TPC-C." [Online]. Available: http://www.tpc.org/tpcc/

[17] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya, "Cost of virtual machine live migration in clouds: A performance evaluation," in *Cloud Computing*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, vol. 5931, pp. 254–265.

[18] J. R. Lange and P. Dinda, "Symcall: Symbiotic virtualization through vmm-to-guest upcalls," in *International Conference on Virtual Execution Environments (VEE)*, 2011, pp. 193–204.