DBMS-Assisted Live Migration of Virtual Machines

Kota Asanuma and Hiroshi Yamada, *Member, IEEE*

***Abstract*—Live migration of virtual machines (VMs) is a technique that moves active VMs between different physical hosts without losing any running states. Although it is desirable for administrators that the live migration is completed as quickly as possible, the pre-copy-based live migration, widely used in modern hypervisors, does not satisfy this demand on the current trend that VMs on which running applications are performance-critical such as database management systems (DBMSes) have quite large memory. *DMigrate*, presented in this paper, shortens the time for live-migrating VMs with even large memory DBMSes. To quickly produce the running state of the migrating VMs on the destination, DMigrate performs regular memory transfers while simultaneously constructing the DBMS's buffer-pool by fetching the data items from the shared storage. We prototyped DMigrate on MySQL 5.7.30, QEMU 5.1.0, and Linux 4.18.20. The experimental results show that the migration time of the prototype is up to 1.71× and 1.71× shorter under workloads, including sysbench and TPC-C, than the default pre-copy and post-copy schemes, respectively.**

***Index Terms*—Live migration, database management systems, system virtualization, cloud computing.**

I. Introduction

LIVE migration of virtual machines (VMs) is a technique that moves active VMs between different physical hosts without losing any running states. Within a local area network (LAN), the main task of the live migration is to transfer VM memory pages to the destination while the VM disks are assumed to be stored in the shared storage. Live migration is helpful to intra-datacenter administrations including load balancing [1], [2], power saving [3], [4], and effective software upgrades [5], [6]. It is used in real-world data centers [7].

Although it is desirable for administrators that the live migration is completed as quickly as possible, the pre-copy-based live migration [8], widely used in modern hypervisors [9], [10], [11], does not satisfy this demand on the current trend that VMs on which memory-intensive applications such as database management systems (DBMSes) are running have quite large memory. For example, Amazon RDS offers VMs whose memory sizes are up to 3904 GiB [12]. Pre-copy transfers updated pages of a VM from the source to destination iteratively until the number of dirty pages is below a threshold, suspends the VM, and then passes the control to the destination after sending all dirty pages and CPU states such as register values. The migration of such VMs requires a long time due to the large number of memory page transfers. In addition, a larger VM memory size makes each iteration longer and could result in the VM spawning dirty pages that have to be transferred in the next iteration.

To address this issue, numerous studies have been conducted. However, it is still challenging to migrate a VM where a DBMS is running. Existing solutions include post-copy-based migration [13], [14], [15], application-level memory knowledge [16], parallelizing live migration execution [17], OS-level memory knowledge [18], [19], memory compression/decompression [20], [21], [22], [23], [24], memory deduplication [25], the use of a high-speed network [26], cooperation of a software-defined network [27], [28], and execution throttling [8], [29]. These approaches incur high resource costs and are difficult to apply to live migration of DBMS-running VMs. Although the application-assisted approach [16] enables to exploit DBMS memory knowledge to skip transfers of selective VM memory, like live migration without transfers of the OS buffer cache [18], the benefit is limited; when we skip the DBMS's buffer-pool to reduce memory page transfers, the DBMS at the destination causes buffer-pool misses, and thus its performance is poor until the warming up of the buffer-pool completes.

DMigrate, presented in this paper, shortens the time for live-migrating VMs with even large memory DBMSes. DMigrate exploits the significantly improved speed of modern storage devices and networks. To efficiently migrate such VMs, DMigrate constructs the VM memory image at the destination by leveraging not only the source host but also shared storage; the DBMS memory contains data objects reproducible from the shared storage, namely the buffer-pool that is dominant in the VM's memory space. DMigrate efficiently produces the running state of the target VMs by performing the regular memory transfers and the DB block fetches from the shared storage in parallel.

This paper makes the following contributions:

- We propose DMigrate that performs live migration of DBMS-running VMs in an efficient manner. DMigrate has several unique characteristics. DMigrate shortens the time for migrating VMs where large-memory DBMSes are running. Also, DMigrate mitigates resource contention with the co-located running VMs on the source during the migration. It does not cause significant performance degradation in the DBMS-running VMs during and after the live migration (Sec. III).

Manuscript received 22 October 2022; revised 22 September 2023; accepted 29 October 2023. Date of publication 7 November 2023; date of current version 15 January 2024. This work was supported in part by the JST, PRESTO, Japan under Grant JPMJPR21P9 and in part by the JSPS KAKENHI under Grant 23H03367. Recommended for acceptance by M. Kaâniche. *(Corresponding author: Hiroshi Yamada.)*

The authors are with the Department of Computer and Information Sciences, TUAT, Koganei-shi, Tokyo 184-8588, Japan (e-mail: hiroshiy@cc.tuat.ac.jp).

Digital Object Identifier 10.1109/TC.2023.3329943

- We introduce software mechanisms for efficient live migration of DBMS-running VMs. The main challenge here is how we bridge the semantic gap between the DBMS and hypervisor. To address this issue, DMigrate uses *DBMS-Enlightenment* where the hypervisor exploits the full knowledge of the DBMS internals. DMigrate cooperates with the DBMS and hypervisor to skip transfers of the buffer-pool's memory pages and build the buffer-pool by fetching its DB blocks from its DB files. Also, DMigrate dynamically controls the balance between memory transfers and data block fetches to complete the live migration as fast as possible (Sec. IV).
- We prototyped DMigrate on MySQL 5.7.30, QEMU 5.1.0, and Linux 4.18.20. The experimental results show that the migration time of the prototype is up to 1.71× and 1.71× shorter under workloads, including sysbench and TPC-C, than the default pre-copy and post-copy schemes, respectively. The results also show that the prototype is helpful for three synthetic migration scenarios: VM eviction, batch migration, and migration in congestion (Secs. V and VI).

II. Background

A. Live Migration

To move running VMs from their current host to another, the live migration mechanisms transfer the running states of the target VM to the destination host and start the VM whose states are the same on the source host. The typical live migration scheme, named *pre-copy* [8], sends memory pages repeatedly and starts the VM after transferring CPU cores' states. Pre-copy consists of two phases: iteration and stop-and-copy. In the iteration phase, pre-copy first sends all the memory pages of the target VM and then repeatedly sends memory pages that get dirty during the previous memory transfer. The phase is moved to the stop-and-copy phase when the dirty pages are under the threshold [30] or the memory transfer iterations become upper-bound. In the stop-and-copy phase, the pre-copy mechanism suspends the VM on the source and sends the remaining dirty pages and virtual CPU states. Finally, it starts the VM at the destination and releases the VM resources on the source.

The live migration facilitates VM replacements inside datacenters without significant service downtime. The VM replacements are adequate for effective administration in datacenters. Live-migrating several VMs from an overloaded host mitigates hotspots in datacenters. The live migration of VMs from underutilized hosts to a single host to turn them off, namely consolidation, reduces energy consumption. Also, all the VMs on a host are live-migrated to other hosts for physical host maintenance and hypervisor updates. In doing so, administrators can move the target VMs to appropriate hosts by performing one operation instead of complicated tasks such as replicating the target VM images and launching new VMs from the images. Such an easy VM replacement significantly reduces the burden on the administrators.

In this paper, we attempt to determine how to live-migrate VMS with modern DBMSes quickly. The total migration time depends mainly on the memory consumption of the target VM. Although live migration is as short as possible since the ideal VM placement is defined by the current situation, VMs with DBMSes make the VM replacement quite difficult. Relational DBMSes, such as MySQL and PostgreSQL, allocate a huge amount of memory in their address spaces to enhance high throughput and low latency by reducing storage accesses. For example, Amazon RDS [12], a relational database service offered by Amazon, serves a VM with 3904 GiB of memory. The live migration of such a VM takes a long time and becomes a hurdle to achieving the appropriate VM replacement.

B. Previous Approaches

Numerous studies for live migration schemes to overcome the weak points of pre-copy have been conducted. The representative scheme is post-copy [13]. Post-copy first executes the stop-and-copy phase and then transfers the memory pages from the source to the destination. Post-copy specifically sends the virtual CPU states, starts the VM on the destination, and transfers the memory pages from the source to the destination while urgently sending the pages that the VM accesses but do not exist at the destination. The total migration of post-copy is much less than pre-copy, especially under write-intensive workloads, since post-copy sends the VM memory pages once. Due to its effectiveness, a hybrid scheme of pre- and post-copy [31], and the improvement schemes of post-copy [14], [15] have been studied. However, the memory transfer size is at least the same as the VM memory size, and thus migration time becomes longer as the VM's memory size is bigger. In addition, post-copy obtains its efficiency at the expense of the reliability of the live migration; the migrating VM can be broken when network failures occur during the post-copy migration since not-yet-arrived pages at the destination cannot be fetched.

Several approaches aim at shortening the total migration time. Compression-based live migration [21], [22], [23], [24], [25] performs compression and decompression of VM's memory pages at the source and destination, respectively. The approaches accelerate the memory transfer phases by reducing the size of memory page transmission. The delta compression schemes [21] transfer only the dirty parts of the updated pages, instead of the whole pages. The schemes memorize the hot page contents, calculate the differences when hot pages are dirty during the previous iteration phase, and send the differences. Similar to these, the deduplication, which deduplicates the same contents in pages, is used in the live migration [22], [23], [24], [25], [32]. Memory Buddies [33] detects the same content pages of the VM on the destination and reuses them to skip their transfers. These approaches cause high resource utilization on the host, leading to the performance degradation of the co-existing VMs on the source and destination, referred to as *migration noises* [18].

PMigrate [17] parallelizes the live migration execution to leverage the modern wide bandwidth of LAN interface cards.

TABLE I
COMPARISON OF DMIGRATE AND OTHER STUDIES IN THE LIVE MIGRATION OF DBMS-RUNNING VMS

	Migration Time	Migration Noise	Perf. Degradation of the Migrating VM
Pre-copy [8]	Long	Small	Small
Post-copy [13], [14], [15], [31]	Long	Small	Small
Compression-based [21], [22], [23], [24], [25]	Long	Large	Small
Deduplication-based [22], [23], [24], [25], [32]	Long	Large	Small
Delta-Compression [21]	Long	Large	Small
Parallel Migration [17]	Short	Large	Small
Java/OS-aware Migration [16], [18], [19]	Long	Small	Small
vCPU throttling [29], [35], [36]	Long	Small	Large
DMigrate	Short	Small	Small

PMigrate leverages both data and pipeline parallelism to parallelize live migration. For example, VM page fetch and page transfer are executed in a pipelined manner. This approach also causes severe migration noises where its high utilization of the CPU cores affects the performance of the co-located VMs.

Several approaches skip a part of memory transmission to shorten the memory transfer phases by exploiting running software knowledge. JAVMM [16] skips the young memory regions of Java applications in the iteration phase and transfers their pages in the stop-and-copy phase. In JAVMM, the Java runtime actively notifies the memory addresses of young memory regions of the hypervisor since the hypervisor does not know which memory pages are used as the young memory region, widely known as the semantic gap [34]. Jo et al. [19] and SonicMigration [18] skips the transfers of the buffer cache in the OS kernel since the buffer cache can be restored by fetching the same blocks from the storage. Although these approaches inspired our approach, they cannot shorten the live migration of DBMS-running VMs. To do so, we need to overcome unique design challenges, as described in the following sections.

A number of approaches control the dirty page rate of the target VMs by throttling virtual CPU core speeds. XvMotion [29] monitors the dirty page rate in each iteration and reduces virtual CPU allocation to make the number of dirty pages smaller than the page transfer sizes. Autoconverge, used in QEMU [35], gradually slows virtual CPU core execution time down to the threshold specified in advance. Le et al. [36] uses virtual CPU throttling to efficiently live-migrate the VMs under the limited network bandwidth. Although the live migration using the CPU throttling technique shortens the total migration time under write-heavy workloads, CPU throttling degrades the performance of the applications running on target VMs. Furthermore, the technique is not effective to large-memory VMs whose dirty page rates are low.

DBMS-level live migration moves running DBMS instances to another host [37], [38], [39], [40], [41]. These approaches enables us to live-migrate the running DBMS from the current VM to another. However, performing administration tasks is potentially difficult since the IP address of the DBMS-running VM changes before/after the migration. This involves the reconfiguration of applications using the DBMS. Our focus is on the live migration of the DBMS-running VMs, which means the DBMS's IP address does not change.

C. System Model

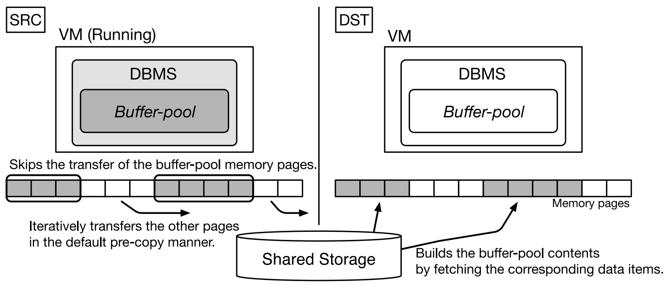
DMigrate, designed to be used in intra datacenters, inherits the same assumption of previous LAN-based live migration approaches. The source and destination hosts connect to a shared storage whose contents can be accessed by both hosts. The DBMS running inside a VM stores its DB files in the shared storage so that the DBMS can access the files after migration. The live migration traffics go through the backend network connections that are different from the connections for traffics from/to Internet users and other VMs.

III. DMIGRATE

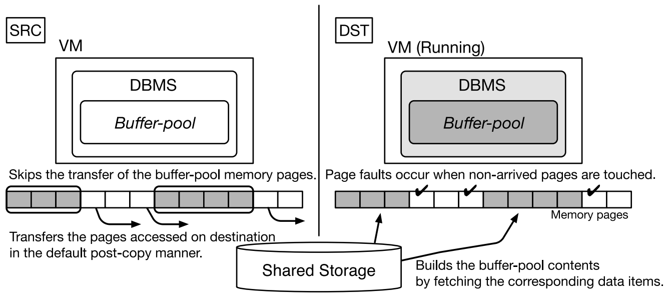
This paper presents *DMigrate*, a live migration scheme for VMs with large-memory DBMSes. Table I briefly compares DMigrate with existing approaches from the viewpoint of the live migration of DBMS-running VMs. DMigrate is driven by the following design goals.

- **Shortens the total time for migrating DBMS-running VMs:** DMigrate migrates VMs even with large-memory DBMSes as quick as possible.
- **Mitigates migration noises:** Different from conventional techniques to accelerate live migration such as parallelization and memory compression, DMigrate avoids the high consumption of computational resources on the source and destination.
- **Avoids degrading the performance of the target VMs:** Our approach migrates the target VMs without significant performance degradation. DMigrate does not use resource throttling techniques, unlike vCPU speed controls for adjusting dirty page rates.
- **Applicable to the pre- and post-copy schemes:** Both schemes need to transfer all the memory pages of the DBMS-running VM, whose total size is hundreds to thousands of gigabytes, at once, requiring a long time. DMigrate is designed to be applicable to both the pre- and post-copy schemes.

To satisfy the design goals, DMigrate exploits the knowledge of DBMS memory management. An overview of DMigrate is shown in Fig. 1. The key observation behind DMigrate is that the memory space of the VM with a modern DBMS is dominated by its buffer-pool that caches DB blocks in memory to improve the throughput and latency. Unlike conventional


SRC
VM (Running)
DBMS
Buffer-pool
DST
VM
DBMS
Buffer-pool
Skips the transfer of the buffer-pool memory pages.
Memory pages
Iteratively transfers the other pages in the default pre-copy manner.
Shared Storage
Builds the buffer-pool contents by fetching the corresponding data items.

(a) Pre-copy-based DMigrate


SRC
VM
DBMS
Buffer-pool
DST
VM (Running)
DBMS
Buffer-pool
Skips the transfer of the buffer-pool memory pages.
Page faults occur when non-arrived pages are touched.
Memory pages
Transfers the pages accessed on destination in the default post-copy manner.
Shared Storage
Builds the buffer-pool contents by fetching the corresponding data items.

(b) Post-copy-based DMigrate

Fig. 1. DMigrate overview.

live migration that utilizes the only resources of the source, DMigrate performs the regular memory page transfers and fetches data blocks from the shared storage to restore the buffer-pool region on the destination in parallel. This is based on the fact that modern storage servers are fast and reliable enough for numerous accesses. In fact, Amazon EC2 UserGuide [42] mentions that the network and random read performance of Amazon EBS, whose storage is SSD-based, is up to 100 Gbps and 8.18 gigabytes per second.

Our design of DMigrate is based on two observation. First, the buffer-pool is typically dominant of the memory usage in DBMS-running VMs. Amazon EC2 conservatively sets the buffer-pool size of MySQL-running instances to 75% of VM memory by default [12]. The DBMS can cache more data items to enhance its performance as the buffer-pool is bigger. Since each DBMS-running VM typically executes one DBMS only in cloud environments, the bigger buffer-pool leads to effective memory utilization. Second, the dirty memory rate of DBMS workloads does not increase due to the storage access involvement. Specifically, the dirty rates in our experiments are 5.9 MB/s of read-only sysbench and 111.3 MB/s of TPC-C, which are much lower than network bandwidth, i.e., 10 Gbps (1,280 MB/s). Based on these facts, even in the pre-copy migration, the memory page transfer before the iteration phases is the main factor in the migration.

DMigrate's buffer-pool construction is effective for the pre- and post-copy schemes. DMigrate shortens all the iteration phases in pre-copy since the storage fetch reduces the number of memory pages to transfer from the source (Fig. 1(a)). The pre-copy-based DMigrate delays the stop-and-copy phase until the buffer-pool construction completes. The source does not need to send all the buffer-pool memory pages as that is basically up to the storage fetch mechanism. For post-copy, DMigrate also reduces the memory transfer phase where the buffer-pool region is constructed additionally by the storage fetch (Fig. 1(b)). We note that the storage fetch does not construct all the buffer-pool memory region. DMigrate forces the source to transfer its dirty pages in pre-copy and accessed pages not yet fetched from the storage in post-copy, as described in detail in Sec. IV.

DMigrate helps administrators perform the VM replacement using live migration even for VMs with large-memory DBMSes. The examples of usage scenarios for DMigrate are as follows.

- **Quickly mitigating hotspots:** VM eviction that live-migrates the target VM to another host is useful for load balancing when overutilized VMs appear on the same host. When the loads of the VMs become significantly high on the same host as a DBMS-running VM, DMigrate quickly releases physical resources for the overutilized VMs by migrating the VM.
- **Migrating all running VMs at once for maintenance:** Migrating all running VMs quickly is preferable for urgent maintenance such as emergent hypervisor updates and partial hardware failures [7]. DMigrate facilitates the urgent live migration of DBMS-running VMs.
- **Smooth live migration under congested migration connections:** Congested network connections for the live migration affect memory page transfers to the destination. Compared with the conventional live migration, DMigrate smoothly migrates DBMS-running VMs since the network traffic for building the buffer-pool mainly goes through the storage connection.

Designing DMigrate poses technical challenges for the hypervisor, where migration mechanisms run: 1) detect memory pages for the buffer-pool to skip their transfers, 2) identify DB blocks used for the buffer-pool and fetch them from storage to restore the buffer-pool, and 3) balance memory transfers between the source host and storage server. We describe our solutions for these challenges in the next section.

IV. Design Details

DMigrate offers software mechanisms for the technical challenges. To skip the transfers of the buffer-pool's memory pages and fetch its DB blocks from storage, DMigrate uses the *DBMS enlightenment* where the hypervisor is aware of the DBMS's memory and storage management. To maximize the use of the source and storage hosts' resources during live migration, DMigrate uses the *adaptive DB block transfer* that builds the buffer-pool region on the destination by not only fetching the corresponding DB blocks from storage but also memory transfers from the source host.

A. DBMS-Memory Enlightenment

To skip the transfers of the buffer-pool's memory pages, the hypervisor needs to identify them at the machine physical page frame (mPFN) level. Migration mechanisms, running inside the hypervisor, transfer memory pages of the target VMs to the destination. The migration mechanisms can use only low-level information such as machine page frame numbers and their dirtiness. They lack the knowledge of applications and

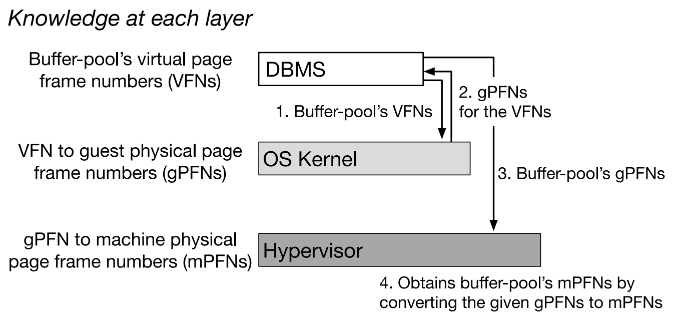

Knowledge at each layer
Buffer-pool's virtual page frame numbers (VFNs)
DBMS
1. Buffer-pool's VFNs
2. gPFNs for the VFNs
VFN to guest physical page frame numbers (gPFNs)
OS Kernel
3. Buffer-pool's gPFNs
gPFN to machine physical page frame numbers (mPFNs)
Hypervisor
4. Obtains buffer-pool's mPFNs by converting the given gPFNs to mPFNs

Fig. 2. Buffer-pool enlightenment.

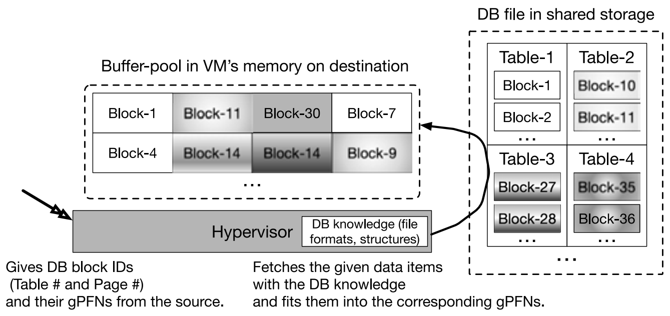

DB file in shared storage
Buffer-pool in VM's memory on destination
Block-1
Block-11
Block-30
Block-7
Block-4
Block-14
Block-14
Block-9
Table-1
Block-1
Block-2
Table-2
Block-10
Block-11
Table-3
Block-27
Block-28
Table-4
Block-35
Block-36
Hypervisor
DB knowledge (file formats, structures)
Gives DB block IDs (Table # and Page #) and their gPFNs from the source.
Fetches the given data items with the DB knowledge and fits them into the corresponding gPFNs.

Fig. 3. Pages on buffer-pool and storage.

even OS's memory management, widely known as a semantic gap [34]; the hypervisor does not know which application- and OS-level memory objects are in the memory pages. Due to this, the migration mechanisms cannot identify which memory pages are used as the buffer-pool, failing to skip the transfers of the buffer-pool's memory pages.

The DBMS-memory enlightenment, used in DMigrate, enables the hypervisor to identify the memory pages of the DBMS's buffer-pool. The main idea of the DBMS-memory enlightenment is that the DBMS actively informs the memory page addresses of its buffer-pool to the underlying hypervisor. Its overview is illustrated in Fig. 2. It orchestrates three software layers; DBMS, OS kernel, and hypervisor. On triggering the live migration execution, our mechanism inside the DBMS tracks the buffer-pool and notifies the OS kernel's virtual page frame numbers (VFNs) via a system call. The OS kernel then translates the VFNs to the guest physical page frame numbers (gPFN), and sends them to the hypervisor via a hypercall. The hypervisor converts the given gPFNs to mPFNs and avoids sending the mPFNs' pages, which means transferring the buffer-pool pages is skipped.

B. DBMS-Storage Enlightenment

In designing mechanisms for building the buffer-pool in the migrating VM memory image, we also pay attention to the DB block location in the shared storage. To restore the current buffer-pool, the mechanism needs to identify the DB blocks used in the buffer-pool and fetch them from the storage. Due to the semantic gap described in the previous section, the hypervisor lacks the semantics of the DBMS's storage management; the hypervisor cannot know where the DB blocks are in storage-level data blocks. Even if the migration mechanism knows the IDs of the DB blocks in the buffer-pool by receiving their metadata or glimpsing the VM's memory with the DBMS-memory enlightenment, it cannot fetch the target DB blocks correctly since the mechanism does not know which data blocks correspond to the IDs.

To fetch the buffer-pool's DB blocks from the storage, DMigrate enlightens the DBMS-storage management at the hypervisor level. Like the DBMS-memory enlightenment, the DBMS-storage enlightenment enables DMigrate to be aware of the DB file's semantics, including its formats, the data types of DB blocks and their metadata, and fetch the target DB blocks from storage. An overview of DMigrate's buffer-pool construction is illustrated in Fig. 3. DMigrate fetches the DB blocks at the destination in accordance with the information from the source and fits the DB block contents into the migrating VM memory image. When the migration is triggered, the DMigrate's mechanism at the source traces the buffer-pool's contents to memorize the DB blocks' IDs and their virtual addresses. The mechanism then sends the IDs and each block's gPFNs to one running at the destination and starts performing the regular live migration without transferring the buffer-pool. The destination mechanism fetches the DB blocks from storage in accordance with their IDs, and fits their contents into the corresponding gPFNs of the target VM memory region.

Regarding dirty memory pages in the buffer-pool, DMigrate relies on the pre-copy scheme. During DMigrate's live migration, the buffer-pool contents can be modified by DB update queries, and their pages become dirty. Since DBMSes do not always write the newer contents back to the DB file, our mechanism at the destination can use the older contents of the DB blocks for the buffer-pool construction. To address this issue, DMigrate uses the pre-copy scheme to send the dirty buffer-pool memory pages from the source to the destination. Although an alternative approach is to notify the DB block updates of the mechanism and force it to fetch the updated blocks, additional communication between the running DBMS and the hypervisor's migration mechanism is required, and the total migration time can be longer since the mechanism needs to wait for the write-back of the updated DB blocks. Similar to this, DMigrate also relies on the default post-copy in creating the buffer-pool's pages that have not yet been fetched from the storage. The pages are sent from the source and DMigrate does not fetch the corresponding DB blocks.

To maximize the storage throughput, DMigrate fetches DB blocks in a multi-threaded manner. The hardware resources of modern storage-conscious instances, such as networks and media, are typically designed to be high-powered to avoid performance bottlenecks under high workloads. DMigrate's mechanism at the destination spawns several threads to fetch the target DB blocks in parallel after receiving DB block numbers in the buffer-pool from the source. On the basis of our experience, our prototype spawns three threads for the DB block fetches.

C. Adaptive DB Block Transfer

We also handle the balance in the buffer-pool construction between the source's memory transfers and the destination's storage fetches. Our goal is to complete the live migration of the DBMS-running VM as quick as possible. Thus, it is better for the source's memory transfer to support the buffer-pool construction. We assume the buffer-pool construction is done only by the destination's storage fetch. In this case, the source's migration mechanism becomes idle after its transfer of the other pages completes faster than the buffer-pool construction, even if the source has the buffer-pool's memory pages. This happens frequently since the buffer-pool dominates the memory consumption of DBMS-running VMs.

The *adaptive DB block transfer* of DMigrate constructs the buffer-pool at the destination by using both the source's memory transfer and the destination's storage fetch. DMigrate enforces the source's migration mechanism to send non-dirty memory pages of the buffer-pool when the transfer of other pages is complete and the storage fetches are still constructing the buffer-pool. In the adaptive DB block transfer, the source mechanism sends the pages of the DB blocks not yet fetched from the storage. When the transfer of the memory pages except for the buffer-pool's ones completes, the source mechanism checks whether the buffer-pool has been constructed. If not, the destination sends a part of the unconstructed buffer-pool's memory addresses to the source so that it can transfer the pages. This source's buffer-pool transfer is repeated until the buffer-pool construction completes. In accordance with our experience, the current prototype requests the source to transfer 20% of the memory pages of the unconstructed buffer-pool region.

In summary, the processing workflow of DMigrate, whose mechanisms run at the three layers: DBMS, OS kernel, and hypervisor, is as follows. On the source, when DMigrate-based migration starts, the DBMS module exclusively tracks DB blocks cached in the buffer-pool, memorizes their IDs and VFNs, and passes the VFNs to the underlying kernel module to convert them to gPFNs. We note that the DBMS module skips dirty DB blocks for the consistency of DB files in storage and new dirty DB blocks are sent to the destination during the pre-copy. The DBMS module also sends the IDs and gPFNs to the destination for the buffer-pool reconstruction. The hypervisor module sets the dirty bitmap entries of the gPFNs passed from the DBMS module to 0 to skip the page transfers, then starts pre-copy or post-copy. On the destination, the hypervisor builds the VM memory with the transferred pages. The hypervisor module, running inside the QEMU process in the prototype, fetches DB blocks and fits them into the VM memory according to given IDs and gPFNs. For the adaptive DB block transfer, the hypervisor module pulls from the source some buffer-pool pages that have not been fetched yet.

V. Implementation

We prototyped DMigrate in MySQL 5.7.30, Linux 4.18.20, and QEMU 5.1.0. The prototype orchestrates its software mechanisms on each software layer at the source and destination. Our prototype does not use particular functionalities of MySQL and QEMU-KVM; DMigrate can be implemented by integrating its mechanisms into functionalities supported by typical DBMSes and hypervisors. Although the implementation itself is not portable since DMigrate leverages knowledge of the buffer-pool management and migration processing, we believe that the concept of DMigrate is applicable to other DBMSes and hypervisors. Note that DMigrate does not work well when the target DBMS relies on the OS-level buffer cache rather than the buffer-pool built in its user-space. DMigrate manages buffer-pool by exploiting the semantics of the application-level memory layout and thus cannot deal with DB blocks in the OS-level buffer cache. DMigrate does not work effectively with such DBMSes like PostgreSQL [43], whose buffer-pool parameter, effective_cache_size parameter is set to use the buffer cache. One of DMigrate's future directions is to handle DB blocks in the buffer cache by controlling DB blocks at the OS-level.

The prototype's mechanism running inside MySQL tracks the buffer-pool region to extract the DB block identifier and its memory addresses to fetch the DB Blocks from the DB file and skip the buffer-pool transfer. InnoDB, a storage engine in MySQL, packs accessed DB blocks into the buffer-pool in an least recently used (LRU) manner. The buffer-pool is divided into chunks, each of which is page-aligned 16 KiB. InnoDB identifies DB blocks with their table numbers and offsets. The DMigrate's mechanism extracts the table number and offset of buffer-pooled DB blocks by tracking its LRU list, and sends them to the destination's storage fetch mechanism. In doing so, it also memorizes the VFNs at the chunk unit. Since the chunk size is four regular pages, each of which is 4 KiB, one memorized address covers four memory pages. In the current implementation, our mechanism converts the VFNs to gPFNs via a special system call and sends them to QEMU using its character device.

When receiving the gPFNs of the buffer-pool, the migration mechanism inside QEMU skips the transfer of the corresponding memory pages. QEMU manages a dirty bitmap, where each bit represents the status of a page, to check which memory pages must be sent. QEMU tracks the dirty pages of the migrating VM with the KVM and sets the bits corresponding to the dirty ones to 1. QEMU initializes all bits in the initialized dirty bitmap to 1 and sets them to 0 one by one after sending the pages. To avoid sending the buffer-pool's memory pages, our mechanism sets the buffer-pool memory pages' bits to 0 in accordance with the gPFNs from MySQL. Our prototype relies on the original migration mechanism to manage the dirtiness of the buffer-pool's memory pages. When a page in the buffer-pool is updated, the QEMU's original migration mechanism naturally sends the page since its entry in the dirty bitmap is 1.

The destination mechanism fetches the DB blocks cached in the buffer-pool and fits their contents into the corresponding gPFN's pages of the migrating VM. The migration mechanism first receives the buffer-pool information from the source and opens the MySQL's DB file in the shared storage. In accordance with the table numbers and offsets, the three threads spawned by our mechanism first check whether DB blocks to read are sent from the source. If not, the threads read the DB blocks in parallel, and then copy the contents to their location in the VM memory image. When the source mechanism requests to transfer the buffer-pool pages, the destination one sends 20% of the gPFNs of the buffer-pool that have yet to be constructed.

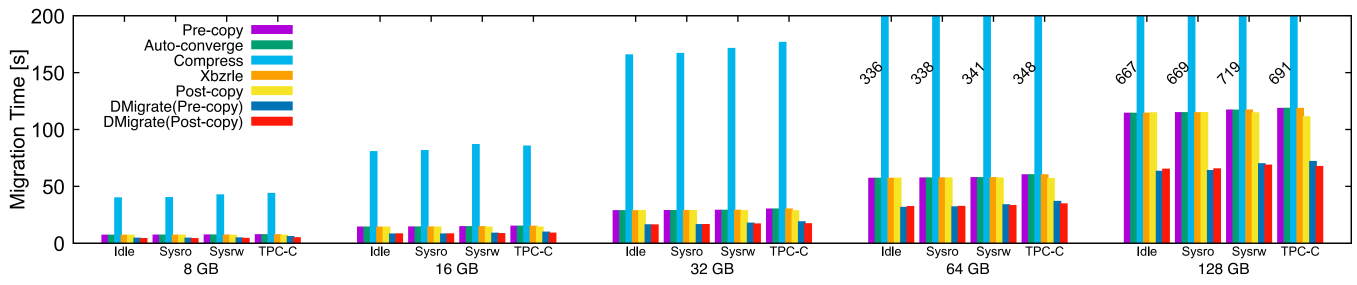


Fig. 4. Migration time.

VI. Experiments

A. Experimental Setup

To confirm the effectiveness of DMigrate, we conduct several experiments with our prototype. We prepare three machines that are connected to each other. Two machines are used as the source and destination hosts and the other is a shared storage. The source and destination machines, each of which has a 2.2 GHz 10-core Intel Xeon Silver 4210 CPU and DDR4 320GB memory, are connected with 10G Ethernet. The storage machine has two 2.5 GHz 10-core Intel Xeon Gold 5215 CPUs, DDR4 128GB memory, and is connected to the source and destination machines with 10G Ethernet. We also prepare a VM where MySQL and Linux is running. It has two vCPUs and 8 to 128 GB of memory configured in the experiments. We run Linux 4.18.20 on these machines. The objective of the experiments is to determine the following: 1) how DMigrate migrates the target VM, 2) how the optimization mechanisms, multi-threaded fetching, and adaptive DB block transfer work, and 3) how effective DMigrate is on migration scenarios.

We used the following workloads as benchmarks in the experiments and measured their scores. The benchmarks were executed three times and the average scores are reported. We also use values measured in the first trial for the graphs. The database size is set to 70% or more of the VM memory size. The Sysbench and TPC-C benchmarks spawn two threads.

- **Idle:** No workload runs after the MySQL is warmed-up by the Sysbench database.
- **Sysro:** The Sysbench read-only workload runs.
- **Sysrw:** The Sysbench workload whose read/write ratio is 7:3 runs.
- **TPC-C:** TPC-C workloads run.

B. Live Migration

To demonstrate how DMigrate migrates the target VM, we migrate a VM with MySQL using DMigrate and QEMU-supported migration schemes, including pre-copy, post-copy, compression, delta-compression, and vCPU throttling, under the workloads. QEMU's compression-based migration, named compress, compresses/decompresses the memory pages at the source and destination in a multi-threaded manner while its delta-compression, named xbzrle, extracts changes of the memory pages and transfers only the changes. The vCPU throttling, named auto-converge, slows vCPU speeds down to the threshold given in advance when the dirty page rate is higher than the memory transfer rate. We configure the compress and auto-converge as default and xbzrle's compression buffer size to be 1 GiB. To compare these live migration schemes, we vary the VM memory and MySQL's buffer-pool size from 8 GiB and 5.8 GiB to 128 GiB and 111 GiB respectively, and measure total migration time, resource consumption, and benchmark scores using the live migration schemes.

1) Total Migration Time and Downtime: The total migration times are shown in Fig. 4. These figures show that DMigrate successfully shortens the migration times in all cases. DMigrate's total migration times are short on all memory size cases compared with the other live migration schemes. In average, the times in the 8 GB cases are 1.50× and 1.66× shorter in the pre-copy- and post-copy-based DMigrates while the times in the 128 GB cases are 1.73× and 1.71× shorter than the defaults. The three optimization techniques, compress, auto-converge, and xbzrle, are not effective even if the memory sizes are changed. Auto-converge and xbzrle take almost the same migration times as pre-copy because the network connection for the live migration is faster than the dirty page rates of all workloads. The total migration times in compress are longest since the memory contents under our workloads are different from each other, and thus we cannot benefit from the compression. Its migration time is up to 10.7× longer than DMigrate's one.

Also, DMigrate achieves the shortest live migration times for all workloads. In the pre-copy cases, DMigrate shortens migration times by 1.51× and 1.67× in the 8GB Idle and TPC-C cases, while the times are 1.73× and 1.71× shorter in the 128 GB cases, respectively. The improvement of DMigrate is almost stable regardless of read-only and read-write workloads due to their low dirty rate. Specifically, the times in the 128 GB cases are 1.80× and 1.65× shorter in Sysro and Sysrw, but the effect of the reduction was diminished by the fact that the data was written to memory. Compared with the pre-copy-based version, the post-copy-based DMigrate takes a stable 1.66× live migration time under all workloads because of no iterative retransfers of the memory pages.

The downtimes in the stop-and-copy phases are shown in Fig. 6. The figures reveal that the downtimes of DMigrate are shortest in all live migration schemes; less than 500 ms since the dirty page rate of the database workloads is low due to the involvement of the storage accesses. From the viewpoint of the downtime, DMigrate is more effective for pre-copy. Since DMigrate reduces the source's memory transfers by the storage

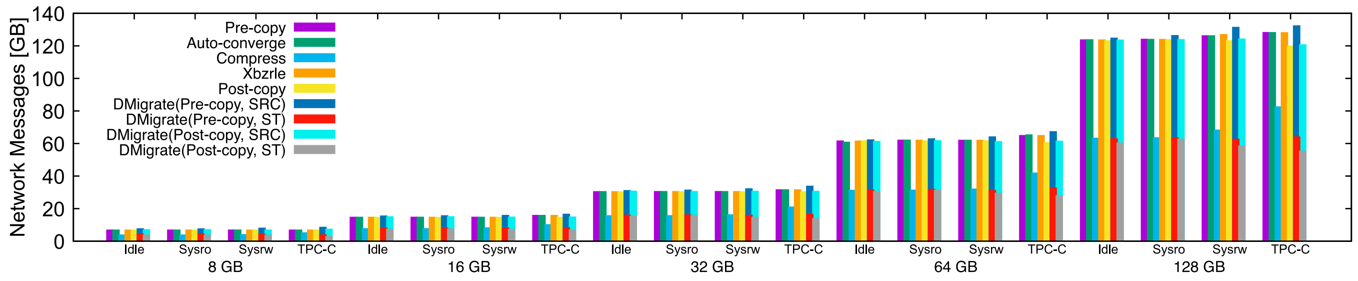


Fig. 5. Migration traffics.

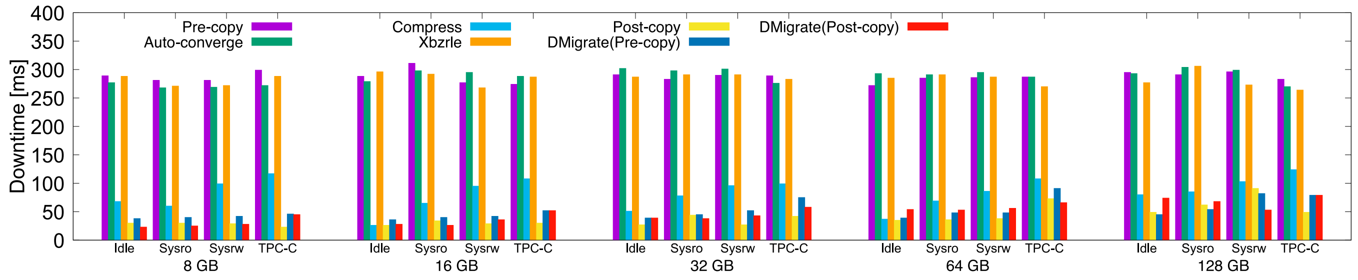


Fig. 6. Live migration downtime.

fetch, the number of transferred pages in its stop-and-copy phase becomes small as well. DMigrate's downtimes in the 8 GB cases are 6.93× and 0.93× shorter than the default pre-copy and post-copy while the downtimes in the 128 GB cases are 4.48× and 0.92× shorter. The downtimes of DMigrate in all post-copy cases are less than 100 ms. Since the behavior of post-copy's stop-and-copy phase is the same between the default and DMigrate, the downtime is similar to each other.

2) Resource Consumption: **Network Traffics:** We measured the two basic aspects of the migration noise, network traffics, and CPU usage during the live migration. Fig. 5 shows total network traffics during live-migrating the MySQL-running VM. In pre-copy, the total number of DMigrate's network messages, which consists of source's memory transfer and destination's storage fetch, is similar to the vanilla one in all cases. In the 8 GB and 128 GB cases, the number of DMigrate's network messages is at least 1.05× and 1.00× larger than the vanilla pre-copy. Compress has the fewest network messages since its compression successfully makes memory pages small. Compress in the TPC-C case has slightly more network messages than those in the other workloads because the memory contents under the TPC-C are more different and are hard to compress. The post-copy-based DMigrate has a similar number of network messages compare with the default since both of them transfer the total memory size of the target VM.

CPU consumption: Figs. 7 to 13 illustrate QEMU's CPU utilization of the live migration schemes at the source and destination. The results only show Sysrw in the 128 GB cases due to limited space. The figures show that DMigrate restricts the excessive CPU core consumption. The CPU usage during migration on the source in DMigrate is 1.8% higher than that of the default pre-copy (Figs. 7 and 8). This comes from MySQL's mechanism, including buffer-pool tracking, its address translation, and interaction with QEMU. On the other hand, the

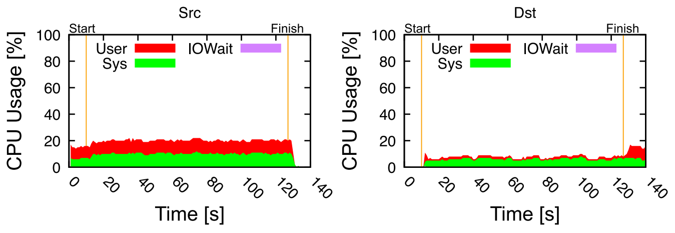


Fig. 7. CPU usage (pre-copy, 128GB, Sysrw).

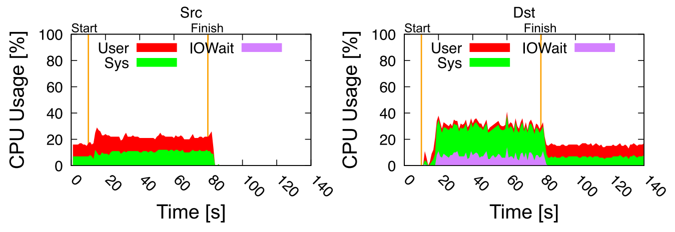


Fig. 8. CPU usage (DMigrate(pre-copy), 128GB, Sysrw).

DMigrate's CPU usage on the destination is 21.1% higher than the default one. This is caused by the storage fetch. The same situation happens in the post-copy case (Figs. 12 and 13). Compress significantly consumes CPU cores at the source and destination due to page compression and decompression. Its CPU utilization is 51.7% and 12.4% higher than the default, and the total CPU consumption is the largest in all schemes. The CPU utilization of auto-converge and xbzrle is similar to that of the default pre-copy.

3) Application Throughput: The benchmark scores of Sysro, Sysrw, and TPC-C during the live migration are shown in Figs. 14, 15, and 16, respectively. We can see that DMigrate does not cause signification performance degradation of the running MySQL in all benchmarks. The benchmark scores of the pre-copy-based DMigrate are almost similar in all benchmarks

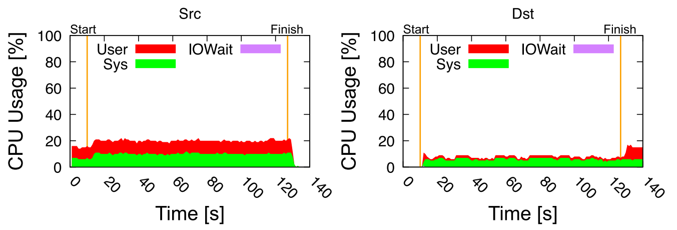


Fig. 9. CPU usage (auto-converge, 128GB, Sysrw).

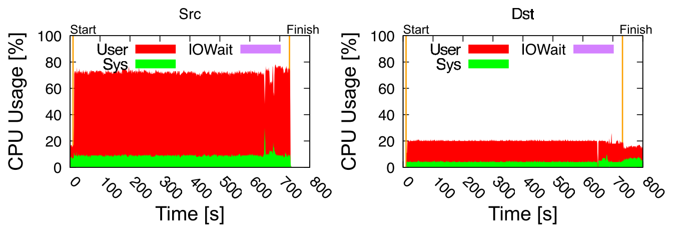


Fig. 10. CPU usage (compress, 128GB, Sysrw).

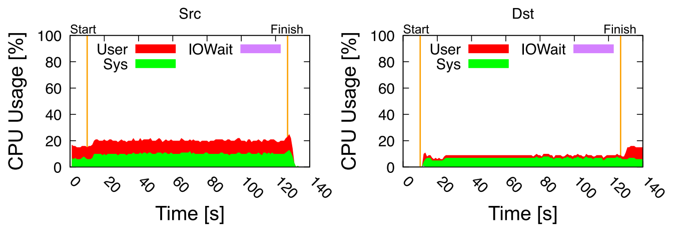


Fig. 11. CPU usage (Xbzrle, 128GB, Sysrw).

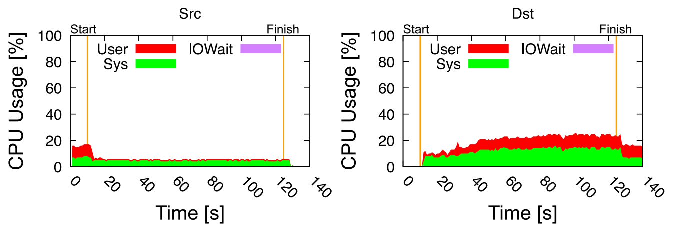


Fig. 12. CPU usage (post-copy, 128GB, Sysrw).

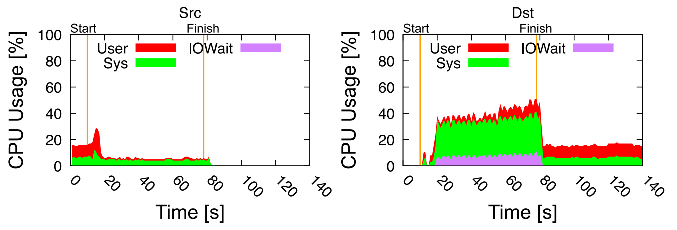


Fig. 13. CPU usage (DMigrate(post-copy), 128GB, Sysrw).

to those of the default. The slight performance degradation of DMigrate at the migration start point comes from the transfer of the buffer-pool information for the storage fetch. Compress's scores are the lowest during the live migration due to CPU contention between MySQL and multi-threaded compression. In post-copy, DMigrate is worse in the two Sysbench-based workloads than the default. This is because the workload accesses the buffer-pool contents sequentially and QEMU's optimization that sends neighbor memory pages around the transferred page works effectively. In the TPC-C case, DMigrate outperforms the

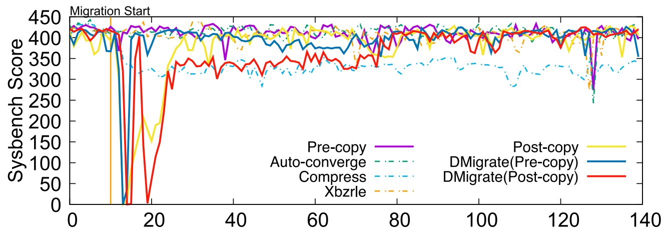


Fig. 14. Benchmark score transition during live migration. (Sysro)

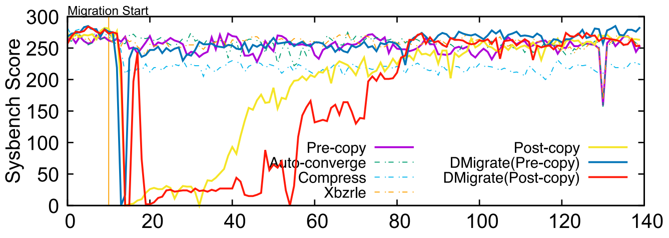


Fig. 15. Benchmark score transition during live migration. (Sysrw)

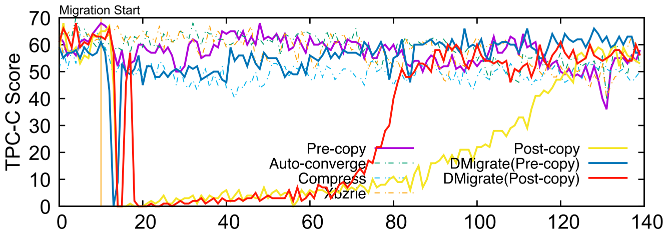


Fig. 16. Benchmark score transition during live migration. (TPC-C)

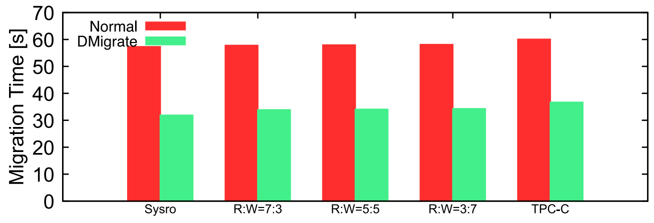


Fig. 17. Migration time under read/write workloads.

default since the DMigrated VM can achieve full performance more quickly than the default.

C. Behavior Analysis

To take a closer look at the effectiveness of DMigrate, we prepare a MySQL-running VM with the 64 GB configuration and run sysbench whose read/write transaction ratio is changed to 3:7, 5:5, and 7:3. The VM is migrated using the default and pre-copy-based DMigrate. We also compare the behaviors with those under sysrw and TPC-C.

The result shown in Fig. 17 reveals that the migration time of DMigrate is shorter than that of pre-copy on all of the benchmarks. The pre-copy-based DMigrate builds the VM image on the destination using both the source (page transfer) and shared storage (buffer-pool reconstruction). The result also shows that the migration times of both migration schemes are similar among different ratio sysbenches. This is because the dirty

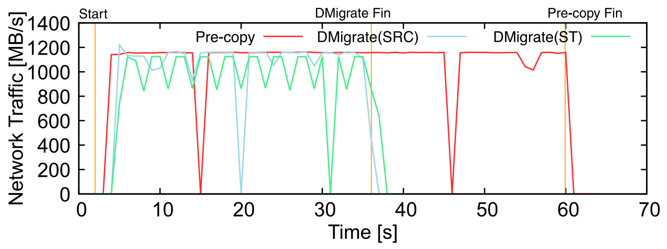


Fig. 18. Network utilization under read/write (5:5) workload.

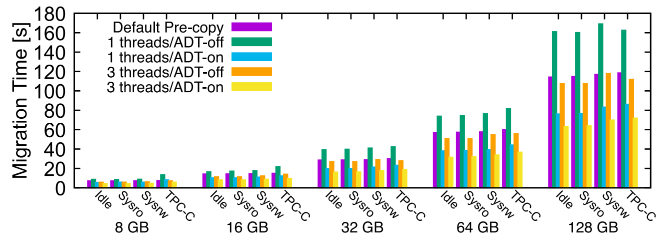


Fig. 19. Migration time (with and without optimization, pre-copy).

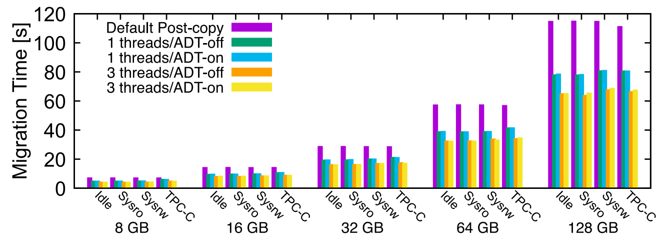


Fig. 20. Migration time (with and without optimization, post-copy).

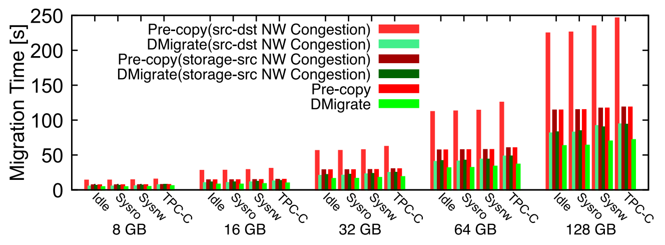


Fig. 21. Migration time (pre-copy schemes under network throttling).

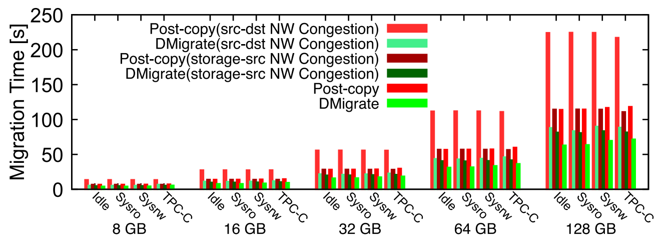


Fig. 22. Migration time (post-copy schemes under network throttling).

rates of sysbenches are not dramatically different; the transactions involve slow storage accesses, and thus, the benchmarks' dirty rates are not high (up to 111.3 MB/s) enough to affect live migration behavior. Fig. 18 shows the network utilization of both migrations under 5:5 sysbench. The default pre-copy iterates three memory page transfers that fully utilize network bandwidth. On the other hand, the pre-copy-based DMigrate first transfers memory pages without non-dirty DB block pages to the destination while fetching the buffer-pool pages from the shared storage. And then, the source mechanism sends the dirty pages. At the same time, the destination mechanism keeps fetching the buffer-pool pages and pulls them from the source using the adaptive DB block transfer. These behaviors are similar among the benchmarks.

D. *Effectiveness of Optimizations*

To show the effectiveness of our two optimizations, the multi-threaded storage fetch and adaptive DB block transfer, we compare the total migration time and benchmark scores of DMigrate with and without these optimizations. We set one and three threads for the storage fetch, and we measure the total migration times of DMigrate with and without the adaptive DB block transfer (ADT) by varying the network bandwidth. We throttle the network bandwidth of the destination to the source and storage using Linux's token bucket filter. We set the connections' throughput to be 10 to 5 Gbps. In doing so, we also measure the benchmark scores in each configuration.

The total migration times are shown in Figs. 19 and 20 without network throttling. The figures reveal that our optimizations for DMigrate contribute to shortening the total migration times. In pre-copy, the parallel storage fetch outperforms the sequential one while the migration time with the adaptive DB block transfer is shorter than that without in all cases (Fig. 19). The migration times with the parallel storage fetch and adaptive DB block transfer are up to 1.21× and 1.60× slower than without them, respectively. The all-off DMigrate is the worst because almost all buffer-pool memory regions are constructed by the sequential storage fetch while the memory transfer of the source completes and is idle. Both optimizations are orthogonal, and thus the migration times of the full-fledged DMigrate are the shortest in all configurations in all cases. Specifically, the full-fledged DMigrate's migration times are 1.71× shorter on average than those of the default. Also, the full-fledged DMigrate in post-copy achieves the shortest migration times in all cases (Fig. 20). This is because the source in the post-copy-based DMigrate does not become idle compared with the pre-copy-based one. The post-copy-based DMigrate inherently fetches the buffer-pool's pages from the source when they have yet to be fetched from the storage.

Figs. 21 and 22 show the total migration times under network throttling. From the figures, we can see that DMigrate outperforms the default pre-copy and post-copy even under network congestion in the source and storage connections. In pre-copy, the total migration times of DMigrate are up to 2.65× and 1.30× shorter than the default in the source and storage network congestion, respectively. On the other hand, the post-copy-based DMigrate takes up to 2.52× and 1.39× shorter migration times than the default. The adaptive DB block transfer constructs the buffer-pool memory region by controlling data transfers of the source and storage in accordance with network congestion. Both default schemes are severely degraded under congestion in the source to destination connection since it is the only migration connection for the VM's memory image construction, unlike DMigrate, which uses connections of the

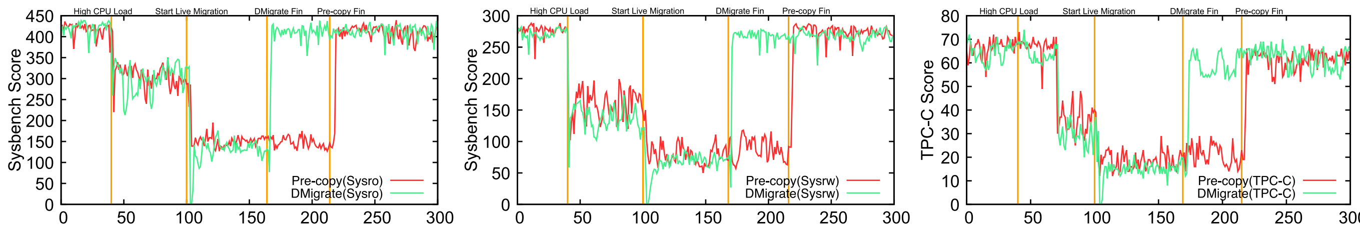


Fig. 23. Benchmark score transition (VM eviction).

source and storage hosts. DMigrate's improvements are therefore relatively better as the memory size is bigger.

E. Migration Scenarios

To demonstrate the effectiveness of DMigrate under migration scenarios, we use it on the following three typical migration scenarios: evicting a DBMS-running VM collocated with an over-utilized VM (*VM Eviction*), parallelly migrating DBMS-running VMs for physical machine faults (*Batch Migration*), and migrating a DBMS-running VM with congestion in the migration connection (*Migration in congestion*). We artificially generate the three situations. We compare the default pre-copy and pre-copy-based DMigrate in the experiment since the pre-copy scheme is widely accepted in practice.

VM Eviction: One of the typical scenarios for live migration is to balance the loads of the physical host. Live migration specifically allocates more resources to an over-utilized VM by evicting other VMs. DMigrate can contribute to quickly create free resources by migrating DBMS-running VMs and thus leading to the swift mitigation of the hotspots. To show DMigrate's effectiveness on such a scenario, we prepare a DBMS-running VM and a CPU load benchmark-running VM. After a specific amount of time, we increase the intensity of CPU load on one VM to saturate the underlying CPU cores, and then migrate the other VM using the default pre-copy and DMigrate. In so doing, we measure the benchmark scores and migration time.

Fig. 23 shows the result. The x- and y-axes represent the elapsed time and benchmark score, respectively. The result shows that DMigrate mitigates over-utilization more quickly than the default pre-copy. The benchmark scores of the two VMs get low when the benchmark intensity is increased due to CPU contention. The performance degradation during the live migration is almost the same between DMigrate and the default pre-copy. The score of DMigrate becomes high more quickly than that of the default because of DMigrate's faster completion. Specifically, DMigrate's migration time is 66.9 sec. while the default's one is 115.2 sec.

Batch Migration: It is better to migrate all running VMs as soon as possible when the host faces partial hardware failures or emerging hypervisor updates are needed. To show that DMigrate effectively migrates multiple DBMS-running VMs, we migrate four MySQL-running VMs with the pre-copy-based DMigrate and the pre-copy scheme. We migrate the VMs at once, running Sysbench-RO, -RW, and TPC-C as the workloads and total the migration times.

The total migration times are shown in Fig. 24. These figures show that DMigrate successfully shortens total migration times

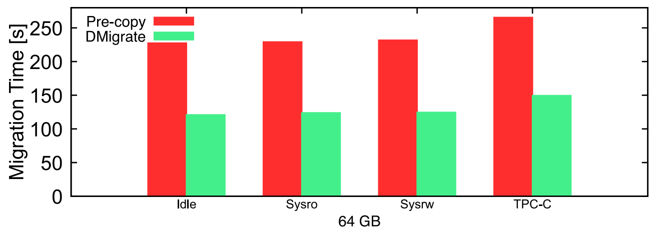


Fig. 24. Migration time (batch migration).

in all cases. DMigrate's migration times are 1.84× shorter on average than the default. Fig. 25 shows the result. The x- and y-axes represent the elapsed time and benchmark score, respectively. DMigrate's total benchmark scores are 1.01× higher than the default since its migration times are short, and thus the intervals of migration noise are also short.

Migration in congestion: Network connections for live migration are sometimes congested due to the traffics of migrating multiple VMs and client traffics if the connections are shared with them. To confirm the effectiveness of DMigrate under network congestion, we intentionally cause congestion in the migration connection by using iPerf [44]. Specifically, we run an iPerf instance that generates 3-Gbps traffic in the migration connection, and perform a MySQL-running VM using the pre-copy-based DMigrate and the default pre-copy. We run the four benchmarks on the VM. In so doing, we measure the network activities in the migration connections and the total migration times.

Fig. 26 shows the network traffics. The x- and y-axes represent the elapsed time and amount of network traffics, respectively. The result reveals that the total migration times of DMigrate are shorter than those of the default by effectively utilizing both migration and storage connections. Theoretically, the default pre-copy is 2.4× slower than DMigrate in the network congestion since the migration connection is the only one to transfer memory pages for the VM construction at the destination. The total migration times are shown in Fig. 27. Specifically, the default pre-copy with iPerf is 1.51× slower than without iPerf. On the other hand, DMigrate additionally uses the storage connection to fetch the DB blocks in the bufferpool region, and thus its total migration times are up to 2.18× shorter than the default.

VII. Discussions and Limitations

Live migration of VMs with a huge amount of memory is challenging in modern cloud platforms where such VMs are common. DMigrate achieves the quick live migration of

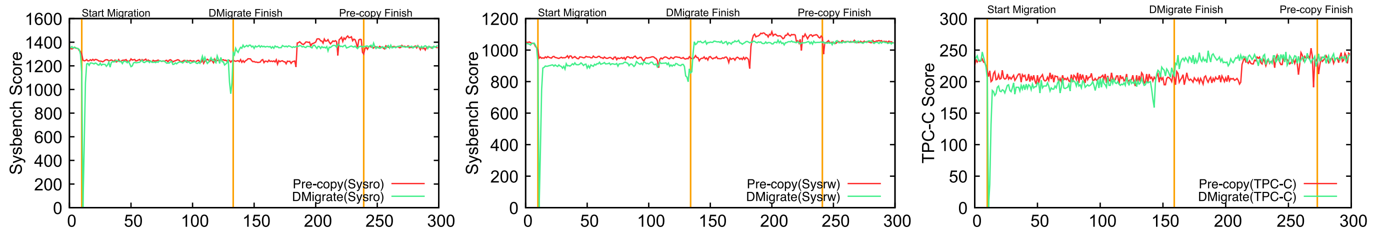


Fig. 25. Benchmark score transition during live migration (batch migration).

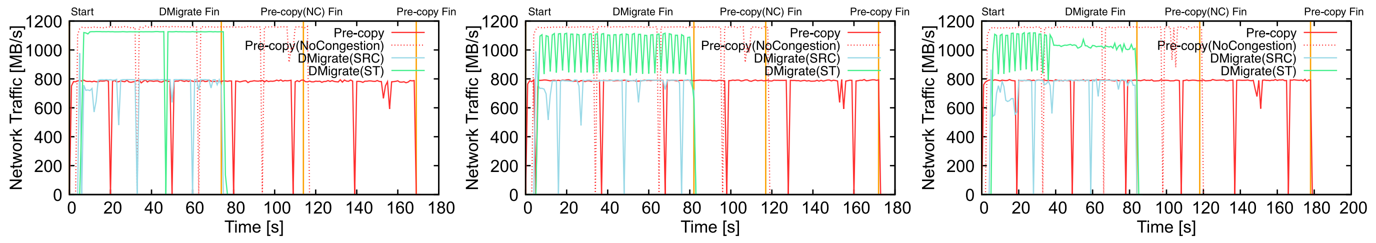


Fig. 26. Changes in data traffic amount (128GB).

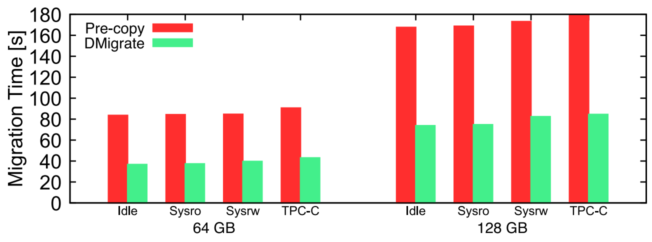


Fig. 27. Migration time (migration in congestion).

DBMS-running VMs. It has several discussions and limitations to be addressed in future work.

Exploiting application-specific knowledge is effective: Modern in-memory applications such as in-memory key-value stores (KVSes) and in-memory processing frameworks allocate a large amount of memory at their address spaces. Like DMigrate, enlightening in-memory applications' memory management at the hypervisor is a helpful approach to quickly live-migrating the VMs. For example, in migrating an in-memory KVS-running VM, we can reduce memory page transfers from the source by only sending hot items or restoring KV regions using other replica. In migrating a VM where tasks for in-memory processing framework are running, we can construct the VM at the destination using intermediate on-storage data for recovery like the Resilient Distributed Datasets [45] in Spark. Exploring these mechanisms is an interesting research topic.

An extension is needed to use DMigrate in non-shared storage environments: An assumption of the current DMigrate, similar to other live migration schemes designed for intra datacenters, is that the source and destination share the storage and can access the DB files on it. This assumption is not acceptable for live migration in inter datacenters where the target VM is moved over WAN. We believe that the idea of DMigrate using DB blocks of a buffer-pool in the storage for the VM construction is applicable to live migration on such non-shared storage environments. Live migration mechanisms for such environments typically transfer the storage contents. At this point, storing DB blocks in the buffer-pool on not only the storage but also memory at the destination can reduce total migration time since we can construct the buffer-pool in the migrating VM memory image independently of memory page transfer, similar to the current DMigrate.

DBMS read-replicas are an alternative source for building buffer-pools: DMigrate could quickly construct the buffer-pool region on the destination by using not only the original DB files but also the target DBMS's read-replicas. Amazon RDS supports quick read-replica generation for load balance. The buffer-pools of the read-replicas are attractive for DMigrate since they can contain data items for the buffer-pool of the migrating DBMS and the accesses to them are faster than the DB files due to modern high bandwidth networks. Integrating into DMigrate a mechanism that fetching these data items from the replicas for the buffer-pool construction can contribute to accelerate DMigrate-based migration.

Memory page alignments of the DB block chunks are preferable: The live migration handles memory events like the dirtiness detection in a memory page unit and transfers the VM's memory at the page granularity. If the buffer-pool's chunks are smaller than the memory page size and are scattered over the memory region, DMigrate fails to skip its memory transfers. The current prototype can cancel the transfer of the buffer-pool region since MySQL's chunks, each of which is 16 KiB, are basically page aligned and the buffer-pool consists of them. Modern DBMSes such as PostgreSQL and memcached manage data items in the page aligned chunks in memory. To apply DMigrate to a DBMS whose buffer-pool is not page aligned, we need to modify its structure to gather the data item caches into memory pages.

VIII. Conclusion

The live migration of DBMS-running VMs is non-trivial due to their tremendously large memory footprint, disturbing live migration-based administration in datacenters. This paper presented DMigrate that shortens the time for migrating

DBMS-running VMs. To produce the running state of the migrating VMs on the destination, DMigrate performs regular memory transfers while simultaneously constructing the DBMS's buffer-pool by fetching the data items from the shared storage. We prototyped DMigrate and conducted several experiments. The experimental results show that our prototype successfully shortens migration times of the pre-copy and post-copy schemes, and the prototype is effective under our synthetic migration scenarios, namely, VM eviction, batch migration, and migration in congestion.

References

[1] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall, "Entropy: A consolidation manager for clusters," in *Proc. 5th ACM Int. Conf. Virtual Execution Environ. (VEE)*, 2009, pp. 41–50.

[2] H. Wang, C. Isci, L. Subramanian, J. Choi, D. Qian, and O. Mutlu, "A-DRM: Architecture-aware distributed resource management of virtualized clusters," in *Proc. 11th ACM Int. Conf. Virtual Execution Environ. (VEE)*, 2015, pp. 93–106.

[3] S. Govindan, D. Wang, A. Sivasubramaniam, and B. Urgaonkar, "Leveraging stored energy for handling power emergencies in aggressively provisioned datacenters," in *Proc. 17th ACM Int. Conf. Archit. Support Program. Lang. Operating Syst. (ASPLOS)*, 2012, pp. 75–86.

[4] M. Wang, X. Meng, and L. Zhang, "Consolidating virtual machines with dynamic bandwidth demand in data centers," in *Proc. 30th IEEE Int. Conf. Comput. Commun. (INFOCOM)*, 2011, pp. 71–75.

[5] X. Zhang et al., "Fast and scalable VMM live upgrade in large cloud infrastructure," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, New York, NY, USA: ACM, Apr. 2019, pp. 93–105, doi: 10.1145/3297858.3304034.

[6] S. Doddamani, T. H. K. Cheng, P. Sinha, H. H. Bagdi, H. Lu, and K. Gopalan, "Fast and live hypervisor replacement," in *Proc. 15th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, 2019, pp. 45–58.

[7] A. Ruprecht et al., "VM live migration at scale," in *Proc. 14th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ. (VEE)*, 2018, pp. 45–56.

[8] C. Clark et al., "Live migration of virtual machines," in *Proc. 2nd USENIX Symp. Networked Syst. Des. Implementation (NSDI)*, 2005, pp. 273–286.

[9] P. Barham et al., "Xen and art of virtualization," in *Proc. 19th ACM Symp. Operating. Syst. Princ. (SOSP)*, Oct. 2003, pp. 164–177.

[10] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: The Linux virtual machine monitor," in *Proc. Linux Symp.*, 2007, pp. 225–230.

[11] "Windows server Hyper-V." Microsoft. Accessed: Oct. 1, 2022. [Online]. Available: https://www.microsoft.com/en-us/server-cloud/solutions/virtualization.aspx

[12] "Amazon Relational Database Service (Amazon RDS)." Amazon Web Services. Accessed: Oct. 1, 2022. [Online]. Available: https://aws.amazon.com/rds/

[13] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy live migration of virtual machines," *J. ACM SIGOPS Operating Syst. Rev.*, vol. 43, no. 3, pp. 14–26, Jul. 2009. doi: 10.1145/1618525.1618528.

[14] Y. Abe, R. Geambasu, K. Joshi, and M. Satyanarayanan, "Urgent virtual machine eviction with enlightened post-copy," in *Proc. 12th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, 2016, pp. 51–64.

[15] D. Fernando, J. Terner, K. Gopalan, and P. Yang, "Live migration ate my VM: Recovering a virtual machine after failure of post-copy live migration," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Jun. 2019, pp. 343–351.

[16] K.-Y. Hou, K. Shin, and J.-L. Sung, "Application-assisted live migration of virtual machines with Java applications," in *Proc. 10th ACM SIGOPS Eur. Conf. Comput. Syst. (EuroSys)*, 2015, pp. 15:1–15:15.

[17] X. Song, J. Shi, R. Liu, J. Yang, and H. Chen, "Parallelizing live migration of virtual machines," in *Proc. 9th ACM Int. Conf. Virtual Execution Environ. (VEE)*, 2013, pp. 85–96.

[18] A. Koto, H. Yamada, K. Ohmura, and K. Kono, "Towards unobtrusive VM live migration for cloud computing platforms," in *Proc. 3rd ACM Asia-Pacific Conf. Syst. (APSys)*, 2012, pp. 7:1–7:6.

[19] C. Jo, E. Gustafsson, J. Son, and B. Egger, "Efficient live migration of virtual machines using shared storage," in *Proc. 9th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ. (VEE)*, 2013, pp. 41–50.

[20] C. Li et al., "BAC: Bandwidth-aware compression for efficient live migration of virtual machines," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, 2017, pp. 1–9.

[21] P. Sväard, B. Hudzia, J. Tordsson, and E. Elmroth, "Evaluation of delta compression techniques for efficient live migration of large virtual machines," in *Proc. 7th ACM Int. Conf. Virtual Execution Environ. (VEE)*, 2011, pp. 111–120.

[22] L. Cui et al., "VMScatter: Migrate virtual machines to many hosts," in *Proc. 9th ACM Int. Conf. Virtual Execution Environ. (VEE)*, 2013, pp. 63–72.

[23] A. Rai, R. Ramjee, A. Anand, V. N. Padmanabhan, and G. Varghese, "MiG: Efficient migration of desktop VMs using semantic compression," in *Proc. USENIX Conf. Annu. Tech. Conf. (USENIX ATC)*, 2013, pp. 25–36.

[24] J.-H. Chiang, H.-L. Li, and T. Chiueh, "Introspection-based memory de-duplication and migration," in *Proc. 9th ACM Int. Conf. Virtual Execution Environ. (VEE)*, 2013, pp. 51–62.

[25] 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 *Proc. 7th ACM Int. Conf. Virtual Execution Environ. (VEE)*, 2011, pp. 121–132.

[26] W. Huang, Q. Gao, J. Liu, and D. K. Panda, "High performance virtual machine migration with RDMA over modern interconnects," in *Proc. IEEE Int. Conf. Cluster Comput. (Cluster)*, 2007, pp. 11–20.

[27] D. Fernando, P. Yang, and H. Lu, "SDN-based order-aware live migration of virtual machines," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, 2020, pp. 1818–1827.

[28] S. Ghorbani et al., "Transparent, live migration of a software-defined network," in *Proc. ACM Symp. Cloud Comput. (SoCC)*, 2014, pp. 3:1–3:14.

[29] A. J. Mashtizadeh, M. Cai, G. Tarasuk-Levin, R. Koller, T. Garfinkel, and S. Setty, "XvMotion: Unified virtual machine migration over long distance," in *Proc. USENIX Conf. Annu. Tech. Conf. (USENIX ATC)*, 2014, pp. 97–108.

[30] R. H. O. Online, "Linux KVM." Accessed: Oct. 21, 2021. [Online]. Available: https://linux-kvm.org/page/Main_Page

[31] P. Lu, A. Barbalace, and B. Ravindran, "HSG-LM: Hybrid-copy speculative guest OS live migration without hypervisor," in *Proc. 6th Int. Syst. Storage Conf. (SYSTOR)*, 2013, pp. 2:1–2:11.

[32] J. Zheng, T. S. E. Ng, and K. Sripanidkulchai, "Workload-aware live storage migration for clouds," in *Proc. 7th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ. (VEE)*, 2011, pp. 133–144.

[33] T. Wood et al., "Memory buddies: Exploiting page sharing for smart colocation in virtualized data centers," in *Proc. ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ. (VEE*), 2009, pp. 31–40.

[34] P. M. Chen and B. D. Noble, "When virtual is better than real," in *Proc. 8th Workshop Hot Topics Operating Syst. (HotOS)*, Jun. 2001, pp. 133–138.

[35] J. J. Herne, "Qemu features/autoconvergelivemigration," 2016. Accessed: Oct. 1, 2022. [Online]. Available: https://wiki.qemu.org/Features/AutoconvergeLiveMigration

[36] H. Li, G. Xiao, Y. Zhang, P. Gao, Q. Lu, and J. Yao, "Adaptive live migration of virtual machines under limited network bandwidth," in *Proc. 17th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, 2021, pp. 98–110.

[37] J. Kang et al., "Remus: Efficient live migration for distributed databases with snapshot isolation," in *Proc. Int. Conf. Manage. Data*, 2022, pp. 2232–2245.

[38] T. Mishima and Y. Fujiwara, "Madeus: Database live migration middleware under heavy workloads for cloud environment," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2015, pp. 315–329.

[39] Y.-S. Lin, S.-K. Pi, M.-K. Liao, C. Tsai, A. Elmore, and S.-H. Wu, "MgCrab: Transaction crabbing for live migration in deterministic database systems," *Proc. VLDB Endowment*, vol. 12, no. 5, pp. 597–610, Jan. 2019.

[40] A. J. Elmore, S. Das, D. Agrawal, and A. E. Abbadi, "Zephyr: Live migration in shared nothing databases for elastic cloud platforms," in *Proc. ACM Int. Conf. Manage. Data (SIGMOD)*, 2011, pp. 301–312.

[41] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi, "Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration," *Proc. VLDB Endowment*, vol. 4, no. 8, pp. 494–505, 2011.

[42] “User guide for Linux instances.” Amazon Web Services, 2022. Accessed: Oct. 1, 2022. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/

[43] “PostgreSQL: The world’s most advanced open source relational database.” The PostgreSQL Global Development Group. Accessed: Oct. 1, 2022. [Online]. Available: https://www.postgresql.org/

[44] “iPerf: The TCP/UDP bandwidth measurement tool,” distributed by CiNii. Accessed: Oct. 1, 2022. [Online.] Available: http://dast.nlanr.net/Projects/Iperf/

[45] M. Zaharia et al., “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proc. 9th USENIX Symp. Networked Syst. Des. Implementation (NSDI)*, 2012, pp. 15–28.

Kota Asanuma received the B.E. degree from Tokyo University of Agriculture and Technology, in 2020. He is currently working toward the master’s degree. His interests include operating systems, virtualization, and cloud computing.

Hiroshi Yamada (Member, IEEE) received the B.E. and M.E. degrees from the University of Electro-Communications, in 2004 and 2006, respectively, and the Ph.D. degree from Keio University, in 2009. He is currently an Associate Professor with the Division of Advanced Information Technology and Computer Science at Tokyo University of Agriculture and Technology. His research interests include operating systems, virtualization, dependable systems, and cloud computing. He is a member of ACM and USENIX.