Dynamic and Efficient Destination Switching during the Migration of Large VMs

Akito Ogata
Kyushu Institute of Technology
ogata@ksl.ci.kyutech.ac.jp

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

***Abstract*—The migration of a virtual machine (VM) needs the time proportional to the size of the states of the VM, especially, its memory size. Therefore, network congestion or the load increase of the destination host can occur during the migration of a VM with a large amount of memory. It is desirable to change the destination to a more suitable host, but traditional VM migration needs to be first canceled and then restarted from the beginning. Since the already transferred VM states are discarded by the cancellation, it is necessary to retransfer all VM states. This paper proposes *DCmigrate* for dynamically switching the destination of VM migration without cancellation. DCmigrate partially cancels VM migration and seamlessly transfers only not-yet-transferred VM states from the source to a new destination. In addition, it forwards already transferred VM states from the original destination to a new destination without discarding them. To minimize the migration time, it transfers VM states in parallel from the two hosts and optimizes the amount of VM states transferred from the two hosts using the work-stealing mechanism. We have implemented DCmigrate in libvirt and QEMU and confirmed that the migration time was successfully reduced in many cases.**

***Index Terms*—virtual machine, VM migration, destination switching, parallel transfer, work stealing**

1. Introduction

Most of the clouds provide services using virtual machines (VMs). Recently, the amount of memory used by VMs has become larger and larger. For example, Amazon EC2 provides VMs with up to 32 TB of memory [1]. One of the remarkable features of VMs is VM migration. When clouds perform host maintenance or load balancing, they can migrate VMs from one host to another. The migration time of a VM is basically proportional to its memory size because the memory is the largest state in the VM states. Therefore, the migration of a VM with a larger amount of memory takes much longer.

During a long VM migration, the load conditions could change from the starting time. For example, the network can be congested, or the load of the destination host can increase. Such conditional changes tend to occur more frequently as the migration time is longer. Consequently, the migration time can become further longer [2], or the performance after VM migration can become lower. To prevent such issues, it is desirable to change the destination host during VM migration [3]. However, traditional VM migration needs to be first canceled and then restarted using a new destination host [4]. Since the already transferred VM states are discarded, it is necessary to retransfer all VM states. This leads to the risk of changing the load conditions again during long VM migration.

This paper proposes *DCmigrate* for dynamically changing the destination of VM migration without cancellation. DCmigrate switches the destination to a new host and transfers only the not-yet-transferred VM states to it from the source. In addition, it forwards already transferred VM states from the original destination to the new one. This enables the source and the original destination to transfer VM states in parallel. Consequently, DCmigrate can achieve a shorter migration time after destination switching, compared with the traditional restart method. This can reduce the possibility of changing the load conditions and switching the destination again during VM migration after destination switching.

DCmigrate partially cancels the running VM migration when it switches the destination. This partial cancellation enables the source to seamlessly continue to transfer VM states to the new destination after destination switching. Also, it enables the original destination to preserve already received VM states without discarding them and forward them to the new destination. The new destination receives VM states from these two hosts in parallel and integrates both states. Since the VM is still running at the source host, the new destination prioritizes VM states received from the source so that the resulting VM states become the latest. To minimize the migration time, DCmigrate provides the work-stealing mechanism and adjusts the amounts of transferred VM states from the two hosts. Using this mechanism, the source can transfer VM states that have not yet been forwarded from the original destination when possible.

We have implemented DCmigrate by extending libvirt [5] and QEMU [6]. Using DCmigrate, we conducted several experiments in various load conditions. When the networks of both the source and the original destination were congested, DCmigrate could reduce the migration time, compared with the traditional restart method. When the network of only the original destination was congested,

DCmigrate could not reduce the migration time, but it could reduce the network bandwidth utilization.

The main contributions of this paper are as follows:

- We propose DCmigrate for dynamically and efficiently switching the destination during VM migration.
- We design and implement the protocol of dynamic destination switching using libvirt and QEMU.
- We evaluate DCmigrate using various load conditions and show its effectiveness.

The organization of the paper is as follows. Section 2 describes issues raised when the load conditions change during VM migration. Section 3 proposes DCmigrate for dynamically switching the destination of VM migration. Section 4 explains the implementation of DCmigrate, and Section 5 shows the experimental results using DCmigrate. Section 6 describes related work, and Section 7 concludes this paper.

2. Background

Recently, clouds provide VMs with a larger amount of memory. A VM can be migrated from one host to another without stopping it. This live VM migration enables administrators to maintain a host without disrupting services provided by the VMs running on it. When the load of a host increases, VMs on it can be migrated to underutilized hosts for load balancing. In general, the time needed for VM migration is proportional to the memory size of a VM because the memory is the largest state in the VM states such as virtual CPU states and device states. This means that VMs with a larger amount of memory need a much longer time to migrate them.

VM migration transfers the states of a VM from the source host to the destination host as follows. First, the destination creates a new empty VM and waits for VM states from the source. The source transfers memory data of the running VM to the destination, while the destination stores it in the memory of the new VM. The source retransfers memory data updated by the running VM during VM migration. When the amount of memory data to be transferred becomes less than the threshold, the source stops the VM and consistently transfers the remaining memory data and the states of virtual devices. The destination applies all received VM states to the new VM and resumes it.

In general, VM migration selects a host whose network is not congested and whose load is low as a destination. However, the load conditions can change from the start time of VM migration because VM migration takes time. For example, network congestion can occur between the source and destination hosts. In this case, the migration time can become much longer because of a low network bandwidth [2]. If the load of the destination host increases, the performance of the migrated VM can become lower due to a high host load. Such changes of load conditions tend to occur more frequently as VM migration takes longer. This means that VMs with a large amount of memory suffer from this issue at a higher probability.

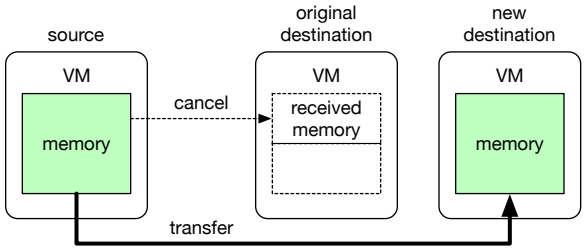


Figure 1. The restart method with migration cancellation.

When such conditional changes occur, it is desirable to change the destination to a more suitable host [3]. If there exists a host connected to a network with a higher bandwidth, VM migration could complete faster by switching the destination. In addition, the network performance of the VM can increase after the migration. If there exists an underutilized host, the migrated VM can run using sufficient resources, e.g., CPU time and memory. As a result, the computing performance of the VM can increase after the migration by destination switching.

However, VM migration needs to be canceled to switch the destination during VM migration [4], as illustrated in Fig. 1. The original destination destroys the new VM and discards all of the already received VM states. Therefore, VM migration has to be restarted from the beginning by specifying a new destination host. Consequently, all the VM states need to be transferred again, although the same VM is migrated. Since it takes time proportional to the memory size of the VM, the load conditions can change again during the new VM migration. In such a case, it is necessary to attempt destination switching again or to accept a long migration time without additional destination switching.

3. DCmigrate

This paper proposes DCmigrate for efficient VM migration by dynamically switching the destination without the full cancellation of the running VM migration. As illustrated in Fig. 2, DCmigrate switches the destination to a new host and enables the source to seamlessly transfer only not-yet-transferred VM states to the new destination. In addition, it allows the original destination to preserve already received VM states and forward them to the new destination. As such, both the source and the original destination can transfer VM states in parallel.

Compared with the traditional restart method, DCmigrate can reduce the migration time after switching the destination by the parallel transfers of VM states from the two hosts. This can reduce the risk of another destination switching by conditional changes during VM migration after one destination switching. When the source host needs maintenance, administrators can shut down that host earlier. When load balancing is necessary, the performance of VMs can be improved earlier. In addition, DCmigrate can reduce

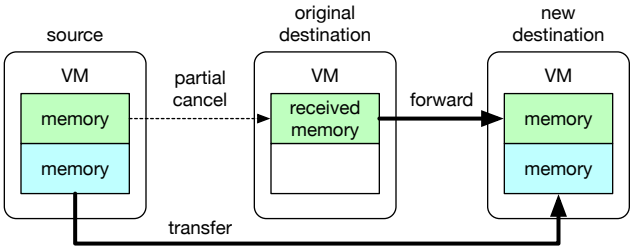


Figure 2. The migration flow in DCmigrate.

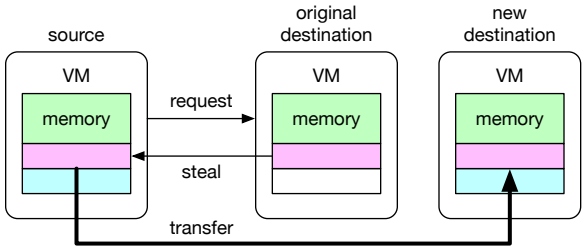


Figure 3. The work stealing of not-yet-forwarded VM states.

the load of the source host and distribute the network load because it can reduce the amount of VM states that the source needs to transfer.

DCmigrate partially cancels the running VM migration when it switches the destination. The source finishes the current job for migration to the original destination, but it continues to transfer VM states. After the new destination starts a new migration job, the source switches the host where it transfers VM states to the new one. Similarly, the original destination finishes the current job for a migration destination, but it does not discard the VM created for the migration. Consequently, it can preserve the already received VM states. After that, it starts a new job as a migration source and forwards the preserved VM states to the new destination.

The new destination integrates VM states received from both the source and the original destination and reconstructs a new VM with the same states as the source. In the case of live VM migration, the VM running in the source host can update its states during VM migration. Since the source retransfers updated VM states, the new destination could receive both old and new VM states from the source and the original destination. Since the source always has newer VM states than the original destination, the new destination prioritizes VM states received from the source. If it has VM states already received from the original destination, it replaces those states with newly received ones. If it has not yet received VM states, it discards VM states received from the original destination later.

To optimize VM migration after destination switching, DCmigrate adjusts the amount of VM states transferred from the source and the original destination using the *work-stealing* mechanism. Since VM states preserved in the original destination have been received from the source, the source always preserves the same or newer states. Using this characteristic, the source *steals* a part of not-yet-forwarded VM states, if any, from the original destination if it completes transferring VM states earlier, as illustrated in Fig. 3. Then, it transfers VM states that it preserves to the new destination instead of the original destination. Note that the original destination does not need to transfer those VM states to the source at this time. The source repeats work stealing and can achieve parallel transfers as much as possible, considering the transfer speeds of both the source and the original destination.

4. Implementation

We have implemented DCmigrate in libvirt [5] and QEMU [6]. Libvirt is an open-source management tool for virtualized systems. The libvirt daemon (libvirtd) runs on each host to manage VMs. When migrating a VM, the libvirt client such as virsh [5] and virt-manager [7] executes several remote procedure calls (RPCs) in the libvirt daemons. An RPC is a mechanism for enabling a client to execute a procedure provided by a server. The libvirt daemons send several commands to the QEMU that runs a VM using the QEMU Machine Protocol (QMP) [8] to manipulate the VM.

4.1. Protocol of Normal VM Migration

To migrate a VM normally, the libvirt client first connects to two libvirt daemons running on the source and destination hosts. Then, it executes VM migration using the protocol consisting of the following five steps.

In the first **Begin** step, the libvirt client sends an RPC request to the libvirt daemon in the source host. The libvirt daemon receives information on the destination and the migration options and starts a new job as a migration source. It generates XML data for the definition of the VM to be migrated and returns the XML data to the libvirt client.

In the second **Prepare** step, the libvirt client sends an RPC request to the libvirt daemon in the destination host. The libvirt daemon starts a new job as a migration destination and creates an empty VM using the XML data generated in the source host. Then, it sends the migrate-incoming command to the QEMU that runs the VM in the destination. This QEMU is denoted as QEMU$_d$. When receiving the command from the libvirt daemon, QEMU$_d$ waits for a connection from the QEMU running in the source host, which is denoted as QEMU$_s$.

In the next **Perform** step, the libvirt client sends an RPC request to the libvirt daemon in the source host again. The libvirt daemon sends the migrate command to QEMU$_s$ that runs the VM to be migrated. When receiving the command from the libvirt daemon, QEMU$_s$ connects to QEMU$_d$ and starts transferring the states of the VM. The libvirt daemon waits for QEMU$_s$ to complete transferring VM states to QEMU$_d$.

When QEMU$_s$ transfers all VM states to the destination, the libvirt client proceeds to the **Finish** step. It sends an RPC

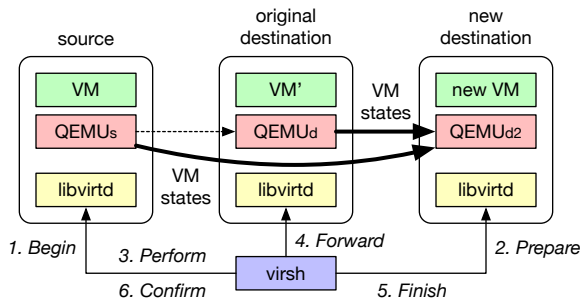


Figure 4. The six steps in DCmigrate.

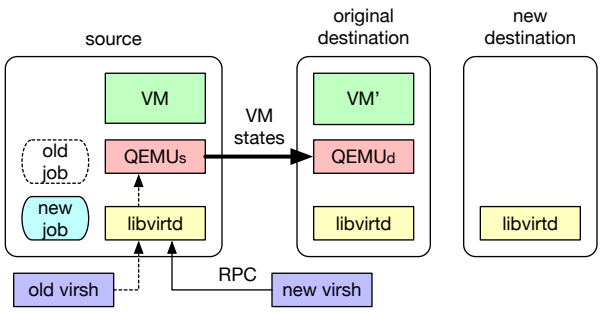


Figure 5. The Begin step.

request to the libvirt daemon in the destination host again. The libvirt daemon waits for QEMU$_d$ to complete receiving all VM states from QEMU$_s$. When QEMU$_d$ completes its migration task, the libvirt daemon finishes the migration job and resumes the VM.

In the final **Confirm** step, the libvirt client sends an RPC request to the libvirt daemon in the source host. The libvirt daemon terminates QEMU$_s$, destroys the VM, and finishes the migration job.

4.2. Protocol of Dynamic Destination Switching

To switch the destination during VM migration, DCmigrate uses six steps, as illustrated in Fig. 4. Five of the six steps are similar to those used in normal VM migration, but the protocol involves three hosts. The Forward step is newly added for the original destination. To switch the migration destination, the libvirt client connects to three libvirt daemons running in the source, original destination, and new destination hosts. Then, it switches the destination and migrates the VM to the new destination. The following is the protocol overview of dynamic destination switching.

In the **Begin** step, the libvirt client sends an RPC request to the libvirt daemon in the source host. The libvirt daemon partially cancels the migration job for the specified VM and starts a new job for migrating the same VM to a new destination.

In the **Prepare** step, the libvirt client sends an RPC request to the libvirt daemon in the new destination host. The libvirt daemon creates a new empty VM and sends a new QMP command to the QEMU that runs the VM, which is denoted as QEMU$_{d2}$, to receive VM states from both the original and new destinations.

In the **Perform** step, the libvirt client sends an RPC request to the libvirt daemon in the source host again. The libvirt daemon switches the destination and sends a new QMP command to QEMU$_s$ to transfer not-yet-transferred VM states to the new destination.

The newly added **Forward** step is executed in parallel to the Perform step. In this step, the libvirt client sends an RPC request to the libvirt daemon in the original destination host. The libvirt daemon cancels the job as a migration destination and starts a new job as one of the two migration sources. It sends a new QMP command to QEMU$_d$ to forward already received VM states to the new destination.

In the **Finish** step, the libvirt client sends an RPC request to the libvirt daemon in the new destination host again. The libvirt daemon waits for QEMU$_{d2}$ to complete receiving all VM states from both QEMU$_s$ and QEMU$_d$. When QEMU$_{d2}$ completes its migration task, the libvirt daemon finishes the migration job and resumes the VM.

In the **Confirm** step, the libvirt client sends an RPC request to the libvirt daemon in the source host. The libvirt daemon terminates QEMU$_s$, destroys the VM, and finishes the migration job. This step is exactly the same as in normal VM migration.

4.3. Migration Task in the Source

In the Begin step, the libvirt daemon partially cancels the specified migration job when receiving an RPC request, as illustrated in Fig. 5. It sends a cancel notification to the thread executing that migration job. That old migration thread waits for QEMU$_s$ to complete transferring VM states to QEMU$_d$ in the Perform step of normal VM migration. When receiving the cancel notification, it finishes waiting for QEMU$_s$ and sends back the cancel completion to the new migration thread.

When the libvirt daemon cancels VM migration, it usually sends the migrate_cancel command to QEMU$_s$ using QMP. In DCmigrate, however, it does not send that command so that the network connection is not torn down between QEMU$_s$ and QEMU$_d$. This prevents the termination of QEMU$_d$ and the destruction of the VM in the original destination host. QEMU$_d$ continues to keep the already received VM states in the VM. Finally, the libvirt daemon starts a new migration job and returns the XML data for the VM definition.

In the Perform step, the libvirt daemon first sends the query-migrate command to QEMU$_s$ when receiving an RPC request. This command is used to obtain the current migration status to update the internal state of the libvirt daemon. Then, it connects to QEMU$_{d2}$ using a new socket and passes the file descriptor for that socket to QEMU$_s$ by sending the getfd command. Finally, it sends the switch command to QEMU$_s$ to change the migration destination, as

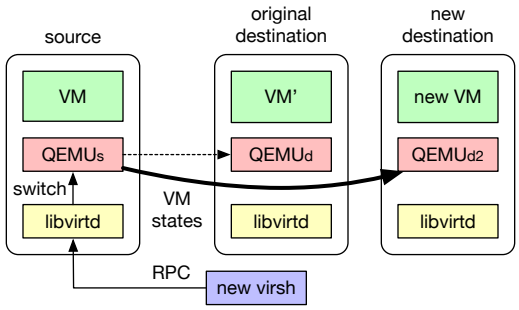


Figure 6. The Perform step.

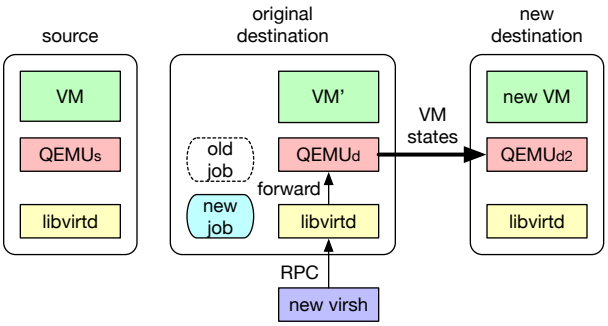


Figure 7. The Forward step.

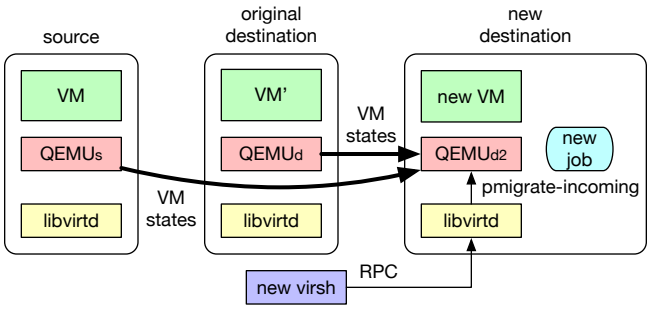


Figure 8. The Prepare step.

illustrated in Fig. 6. Then, it waits for QEMU$_s$ to complete transferring all VM states to QEMU$_{d2}$.

When QEMU$_s$ receives the switch command, it first creates an IO channel using the file descriptor passed by the getfd command. Then, it replaces the old IO channel for QEMU$_d$ with the new one for QEMU$_{d2}$. This transparently switches the connection with the original destination to that with the new destination. For normal VM migration, QEMU$_s$ transfers various information on the VM to be migrated to QEMU$_d$. When the destination is changed, it is necessary to retransfer the same information to the new destination, but it is time-consuming to collect the same information again. To optimize this task, DCmigrate saves the information transferred at the beginning of normal VM migration. Then, it just retransfers the saved information to the new destination.

After that, QEMU$_s$ seamlessly hands over the original migration task and restarts to transfer VM states to the new destination. Therefore, it transfers only the not-yet-transferred VM states naturally. Like normal VM migration, it retransfers the data of updated memory pages after transferring the data of all the memory pages. Finally, it stops the VM and transfers the remaining VM states. Before stopping the VM, the source establishes a new network connection with the original destination to steal not-yet-forwarded VM states, if any. QEMU$_s$ sends a work-stealing request to QEMU$_d$. If it receives the list of not-yet-forwarded memory pages, it transfers those pages to QEMU$_{d2}$ instead of QEMU$_d$. The details of work stealing are explained in the next subsection.

4.4. Migration Task in the Original Destination

In the Forward step, the libvirt daemon first cancels the current migration job when receiving an RPC request. Then, it starts a new migration job and sends the forward command to QEMU$_d$, as illustrated in Fig. 7. QEMU$_d$ establishes a new connection with QEMU$_{d2}$ and forwards already received VM states to QEMU$_{d2}$. To manage already received memory pages, it creates a reception bitmap and sets the bits corresponding to the pages received from QEMU$_s$. When forwarding memory data, it searches for the bitmap and forwards the data of the page corresponding to the bit whose value is one.

When QEMU$_d$ receives a work-stealing request from QEMU$_s$, it divides not-yet-forwarded memory pages into two groups, considering the network bandwidths of the source and the original destination. The pages are divided so that the number of pages is proportional to the transfer speed. QEMU$_s$ and QEMU$_d$ calculate the transfer speeds using the number of pages transferred per unit time. QEMU$_d$ receives the transfer speed of QEMU$_s$ with the request. Then, it returns the list of pages included in one group. After that, it forwards the data of only the pages included in the other group.

4.5. Migration Task in the New Destination

In the Prepare step, the libvirt daemon creates a new empty VM using the XML data generated in the source host when receiving an RPC request. Then, it sends the pmigrate-incoming command to QEMU$_{d2}$ instead of the migrate-incoming command, as illustrated in Fig. 8. When QEMU$_{d2}$ receives this command, it creates two threads to receive VM states from both QEMU$_s$ and QEMU$_d$ in parallel. One thread waits for a network connection from QEMU$_s$ and stores received memory data in the memory of the VM. The other thread waits for a network connection from QEMU$_d$ and stores received memory data in the VM.

To detect multiple receptions of the same memory page, QEMU$_{d2}$ creates a reception bitmap. When receiving the data of a page, it sets the corresponding bit to one and stores the data in the memory of the VM if the bit is still zero. If the bit is already one when QEMU$_{d2}$ receives

TABLE 1. HOST SPECIFICATION.

	source host	original/new destination hosts
CPU	Intel Xeon W-2245	Intel Core i7-10700
memory	64 GB	128 GB
NIC	10 Gigabit Ethernet	

memory data from QEMU_d, the page has already been received from QEMU_s. This means that QEMU_{d2} has received memory data updated by the VM running in the source host. Therefore, QEMU_{d2} prioritizes the data from QEMU_s and discards the data from QEMU_d. In contrast, if the bit is one when QEMU_{d2} receives memory data from QEMU_s, the page has already been received from QEMU_d. This means that QEMU_{d2} receives updated memory data retransferred from QEMU_s. Therefore, QEMU_{d2} prioritizes the data from QEMU_s and stores the data in the memory of the VM.

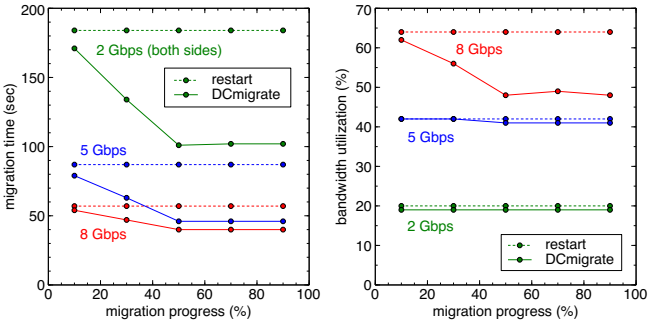
5. Experiments

To show the effectiveness of DCmigrate, we measured the migration time five times after dynamically switching the destination during VM migration. Also, we measured the average utilization of the network bandwidth in the source after destination switching. In the experiments, we assumed network congestion and limited the network bandwidth in the source and/or the original destination using Linux traffic control (tc command). To exactly specify the timing of switching the destination, we temporarily paused VM migration when transferring the specified amount of memory data and resumed it after switching the destination. To investigate the relationship between that timing and migration performance, we disabled the compression of migrated memory data. For comparison, we used (1) the *restart* method, which completely cancels VM migration to the original destination and restarts new migration to a new destination, and (2) the *static* method, which continues VM migration without dynamically switching the destination.

We used three hosts shown in Table 1. There hosts were connected using a 10 Gigabit Ethernet switch. We used Linux 5.15.0 as a host OS, KVM [9] as a hypervisor, QEMU 7.2.0 as a device emulator, and libvirt 8.9.0 as a VM management tool. We assigned 8 virtual CPUs and 40 GB of memory to a VM and ran Linux 5.15.0 as a guest OS. We did not run active applications in the VM to suppress data retransfers. We executed the commands of the virsh client in the source host.

5.1. Case 1: Similar Congestion on Both Sides

In this experiment, we assumed that network congestion occurred similarly in both the source and destination during VM migration. To simulate this condition, we limited the network bandwidth on both sides to 8, 5, or 2 Gbps. We did not limit the bandwidth in the new destination. Fig. 9(a) shows the average migration time needed after we switched the destination when the progress of VM migration reached



(a) Migration time (b) Bandwidth utilization

(c) Migration time (vs. static)

Figure 9. The impact of similar congestion on both sides.

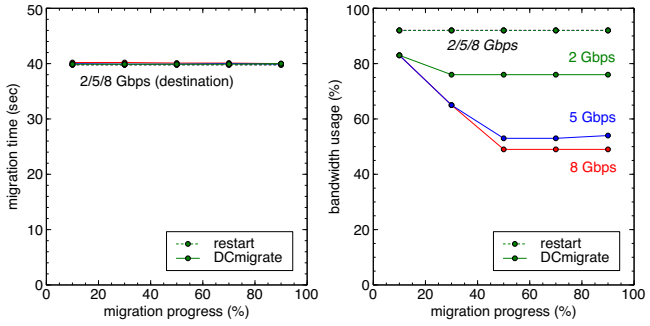
the specified point. We do not show the variance because the standard error is less than 1 second.

Compared with the restart method, DCmigrate could reduce the migration time by up to 36% when the network bandwidth was limited to 8 Gbps. This is because DCmigrate could transfer VM states from the original destination in parallel to the source. When the network bandwidth was limited to lower than or equal to 5 Gbps, DCmigrate could reduce the migration time by up to 47%. In this case, the total network bandwidth used by both the source and the original destination was lower than or equal to 10 Gbps. Therefore, both hosts could use up their network bandwidth.

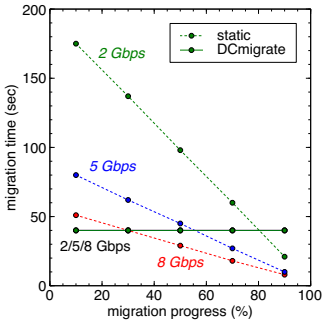
It should be noted that the work-stealing mechanism successfully prevented the increase in migration time even when the progress of VM migration exceeded 50%. In this case, the migration time should increase without work stealing because the transfer of VM states from the original destination becomes dominant.

Fig. 9(b) shows the average utilization of the network bandwidth in the source. The bandwidth utilization was lower than the restart method only when the network bandwidth was limited to 8 Gbps. This is because the network bandwidth of the new destination became a bottleneck. As a result, either the source or the original destination could transfer VM states only at 5 Gbps.

Fig. 9(c) shows the comparison between DCmigrate and the static method. When the network bandwidth was limited to lower than or equal to 5 Gbps, DCmigrate could achieve the same migration time if the progress of VM

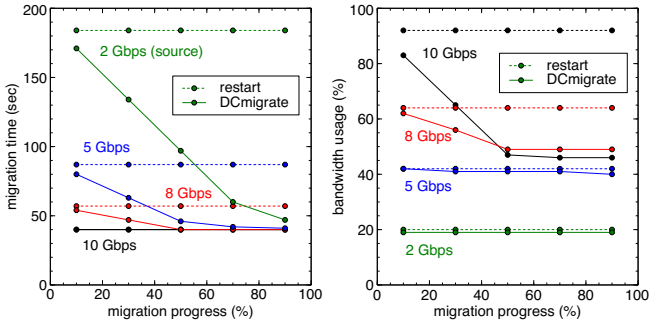


(a) Migration time (b) Bandwidth utilization

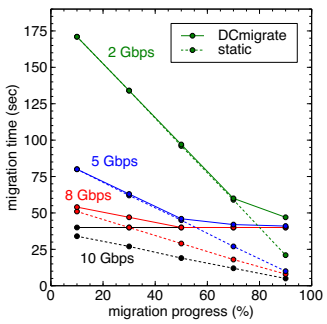


(c) Migration time (vs. static)

Figure 10. The impact of network congestion in the destination.



(a) Migration time (b) Bandwidth utilization



(c) Migration time (vs. static)

Figure 11. The impact of switching to a less loaded destination host.

migration was less than or equal to 50% when switching the destination. Since the source could transfer the remaining VM states at the same time after destination switching, the original destination could transfer all the received VM states during that period. In contrast, the static method was faster when the network bandwidth was limited to 8 Gbps. In DCmigrate, the source could transfer VM states only at 5 Gbps because both it and the original destination performed network transfers in parallel. Note that the network performance is lower after VM migration if the destination is not switched.

5.2. Case 2: Congestion in the Destination

In this experiment, we assumed that network congestion occurred only in the destination during VM migration. To simulate this condition, we limited the network bandwidth in the original destination to 8, 5, or 2 Gbps. We did not limit the bandwidth in the source or the new destination. As shown in Fig. 10(a), DCmigrate could not reduce the migration time, compared with the restart method. Since the network bandwidth of the new destination was saturated by network transfers from the source, parallel transfers from the original destination were not effective. Even in this case, the overhead of parallel transfers and work stealing was negligible in DCmigrate.

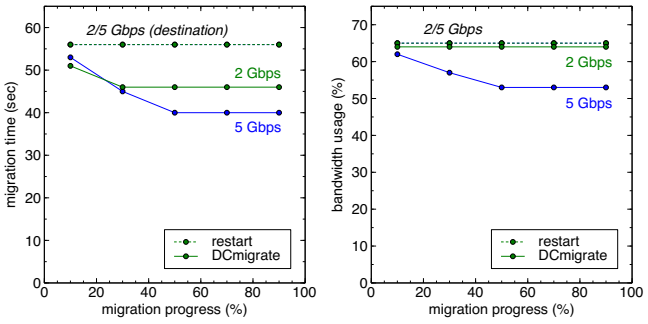
In contrast, DCmigrate could reduce the network bandwidth utilization, as shown in Fig. 10(b). This reduction became larger as the network bandwidth was higher in the original destination. This is because DCmigrate could reduce the amount of VM states to be transferred from the source more largely if the original destination could transfer more VM states.

Fig. 10(c) shows that DCmigrate could reduce the migration time if the destination was switched in the earlier phase of VM migration. When the network bandwidth of the original destination became lower, the migration time was reduced even if destination switching was done in the later phase. This is because VM migration took much longer in the static method.

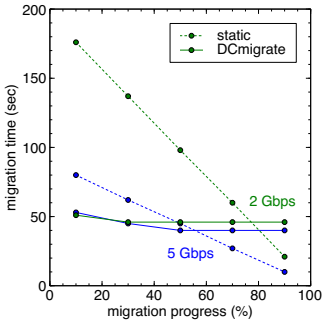
5.3. Case 3: Better Destination Host Found

In this experiment, we assumed that a less loaded host was found as a new destination, although the network of the original destination was not congested. In addition to the case of no network congestion in the source, we assumed that network congestion occurred in the source. In this case, we limited the network bandwidth to 8, 5, or 2 Gbps. We did not limit the bandwidth of the original or new destination.

As shown in Fig. 11(a), DCmigrate could not reduce the migration time when the network bandwidth of the source was 10 Gbps. Since the network bandwidth of the new destination became a bottleneck, the source and the original destination transferred VM states only at 10 Gbps in total. When the network bandwidth was lower, DCmigrate could reduce the migration time by up to 74%. As the network bandwidth of the source was lower, the migration time

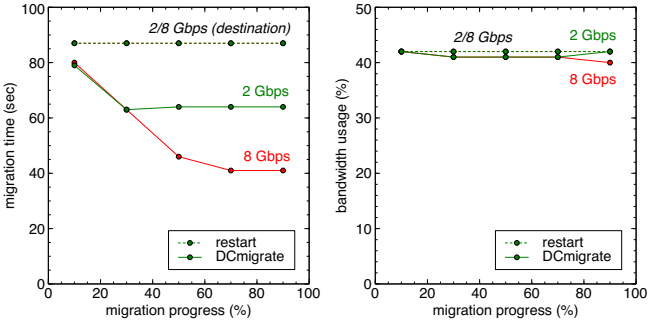


(a) Migration time (vs. restart) (b) Bandwidth utilization

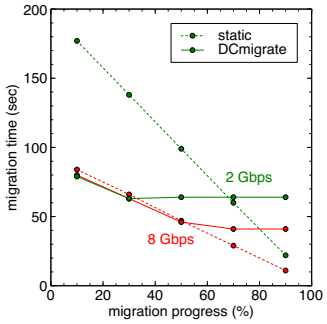


(c) Migration time (vs. static)

Figure 12. The impact of different network congestion (source: 8 Gbps).



(a) Migration time (vs. restart) (b) Bandwidth utilization



(c) Migration time (vs. static)

Figure 13. The impact of different network congestion (source: 5 Gbps).

was reduced more significantly. This is because the original destination could transfer VM states faster.

Fig. 11(b) shows that DCmigrate could reduce the network bandwidth utilization when the network bandwidth of the source was higher than or equal to 8 Gbps. Since the original destination could transfer VM states at 10 Gbps, the available network bandwidth was approximately 5 Gbps in the source.

DCmigrate could switch the destination without increasing the migration time when the network bandwidth of the source was lower than or equal to 5 Gbps, as shown in Fig. 11(c). This is because the original destination could complete forwarding VM states while the source transferred VM states. However, DCmigrate always increased the migration time when the network bandwidth of the source was higher than or equal to 8 Gbps. In this case, the source transferred VM states only at approximately 5 Gbps by conflicting with network transfers from the original destination. Note that the migrated VM must run on a heavily loaded host if the destination is not switched.

5.4. Case 4: Different Congestion on Both Sides

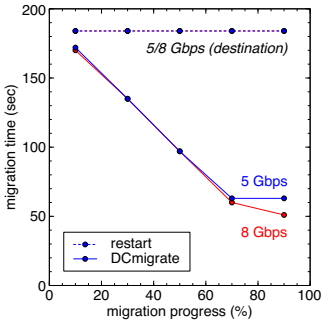
In this experiment, we assumed that network congestion occurred in both the source and the original destination differently. Fig. 12, Fig. 13, and Fig. 14 show the results when we limited the network bandwidth to 8, 5, and 2 Gbps in the source, respectively. When we limited the bandwidth on both sides, DCmigrate could always reduce the migration time, compared with the restart method. It was more effective as the network bandwidth of the source was lower and as that of the original destination was higher. DCmigrate forwards VM states from the original destination to the new destination, while the restart method retransfers all VM states from the source. Therefore, DCmigrate was faster than the restart method when the network bandwidth of the original destination was larger than that of the source.

In contrast, DCmigrate could reduce the network bandwidth utilization when we limited the network bandwidth of the source to 8 Gbps and that of the original destination to 5 Gbps. The used network bandwidth became lower in the source only when the total network bandwidth used by the source and the original destination was higher than or equal to 10 Gbps and when the network bandwidth of the source was higher than or equal to 5 Gbps.

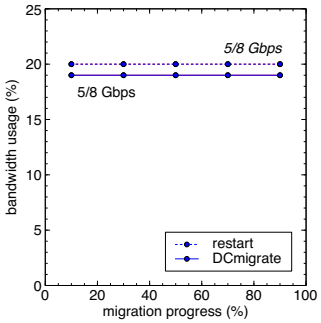
Compared with the static method, DCmigrate could reduce the migration time only when the network bandwidth of the original destination became lower than that of the source. In this case, the network bandwidth of the original destination became a bottleneck. In contrast, when the network bandwidth of the original destination was higher, DCmigrate could achieve the same migration time if it switched the destination earlier. This is because the transfer time from the original destination became shorter than that from the source.

TABLE 2. THE BEST MIGRATION METHODS FOR THE NETWORK BANDWIDTHS AND THE PROGRESS OF VM MIGRATION.

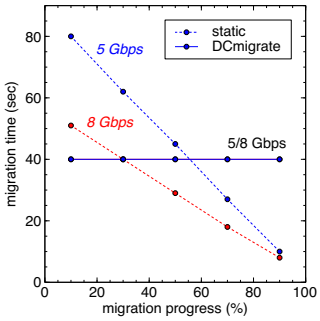
		original destination's bandwidth (Gbps)			
		2	5	8	10
source bandwidth (Gbps)	2	**DCmigrate (10-43%)** static (43-90%)	**DCmigrate (10-55%)** static (55-90%)	**DCmigrate (10-30%)** static (30-90%)	DCmigrate (10-70%) static (10-90%)
	5	**DCmigrate (10-68%)** static (68-90%)	DCmigrate (10-50%) static (10-90%)	**DCmigrate (10-50%)** static (50-90%)	DCmigrate (10-50%) static (10-90%)
	8	**DCmigrate (10-77%)** static (77-90%)	**DCmigrate (10-50%)** static (50-90%)	static (10-90%)	static (10-90%)
	10	**DCmigrate (10-80%)** restart (10-80%) static (80-90%)	**DCmigrate (10-55%)** restart (10-55%) static (55-90%)	**DCmigrate (10-30%)** restart (10-30%) static (30-90%)	static (10-90%)



(a) Migration time (vs. restart)



(b) Bandwidth utilization



(c) Migration time (vs. static)

Figure 14. The impact of different network congestion (source: 2 Gbps).

5.5. Summary

Table 2 summarizes the best migration methods for the network bandwidths of the source and the original destination. Since migration performance is affected by the progress of VM migration on destination switching, the table also shows the range of migration progress for which that migration method is best. DCmigrate is the best across most bandwidths and with less migration progress. When the source is not congested, the migration time of DCmigrate is the same as that of the restart method, but DCmigrate consumes a less network bandwidth in the source. When the original destination is not congested, the migration time of the static method is the same as or shorter than that of DCmigrate. However, the migrated VM must run on a heavily loaded host because the static method cannot switch the destination host to a less loaded one. When DCmigrate switches the destination in the later phase of VM migration, the migration time of the static method is shorter than that of DCmigrate. In exchange for this performance gain, the migrated VM is affected by the congested network to which the destination host is connected.

6. Related Work

Acinonyx [10] switches network routes during VM migration to reduce the migration time. It dynamically schedules a large amount of data flows between hosts using software-defined networking (SDN) in a datacenter with multiple network routes. However, it cannot change the destination of VM migration. When the source cannot detour congested networks, particularly the network of the destination, to reach the destination, Acinonyx cannot improve the migration performance. It is more effective to switch not only network routes but also the destination host by combining Acinonyx and DCmigrate.

Split migration [11] could be used to change the destination host when the load conditions change. It divides the memory of a VM into multiple pieces and transfers them to multiple destination hosts. After the migration, the VM runs across multiple hosts by exchanging memory data using remote paging. It might be possible to transfer VM states to one destination host at first and transfer the remaining states to the other destination host after the load conditions change. However, the performance of the migrated VM becomes lower because remote paging needs to involve the original destination host with performance issues.

To eliminate the overhead of remote paging, merge migration [12] can be used. It transfers VM states divided by split migration from multiple source hosts to one destination host. Thereby, it enables the VM to run without remote paging and restore the original performance. DCmigrate could be considered a migration method that combines split migration and merge migration. The important difference is that DCmigrate runs a VM at the source host until VM migration completes. The performance of the VM does not degrade due to remote paging during VM migration.

A parallel migration method to multiple destination hosts has been proposed [3]. It can select the best destination host during migration, but the amount of network transfers increases significantly. Similarly, Scatter-Gather live migration [13] transfers VM states to multiple intermediate

hosts and then transfers them to the destination host. Using intermediate hosts, it can start VM migration even when the network of the destination host is temporarily congested or when the memory pressure is high. It is similar to DCmigrate in that it transfers VM states from multiple hosts to the destination host in parallel. However, it cannot change the destination host after starting VM migration because it is based on post-copy migration [14].

Agile migration [15] swaps out the unused memory of a VM to the swap device in the network and then transfers only the used memory to the destination host. It can change the destination host until it completes the swap-outs to the swap device. To optimize the total migration time, it can adjust the amount of swapped-out memory, considering the network bandwidth between the source host and the swap device. However, the performance of the VM degrades if the VM needs memory stored in the swap device after the migration. DCmigrate can run the migrated VM without any performance degradation.

MiyakoDori [16] and VeCycle [17] preserve the memory data of a VM at the source host and reuse it in the next VM migration when possible. They assume that a VM is migrated back to the same host after it is migrated to another host. The forwarding mechanism from the original destination in DCmigrate is similar to them in that the already received VM states are not discarded but reused. However, DCmigrate can remove the VM states in the original destination after VM migration completes.

PMigrate [18] parallelizes VM migration from one source host to one destination host to reduce the migration time. It uses abundant resources such as CPUs and NICs and leverages data parallelism and pipeline parallelism. To improve the scalability of concurrent memory-mapping operations for transferring the memory of a VM, an abstraction called range lock is proposed. Since DCmigrate receives the memory of a VM in parallel at the destination host, the scalability could be improved by using range lock.

7. Conclusion

This paper proposes DCmigrate for dynamically switching the destination of VM migration without canceling it. DCmigrate enables the source to switch the destination to a new host and seamlessly transfer only the not-yet-transferred VM states to the new destination. In addition, it allows the original destination to forward already received VM states to the new destination. It can reduce the migration time using parallel transfers from the two hosts and the optimization of transferred data amounts with work stealing. We have implemented DCmigrate in libvirt and QEMU and showed that the migration time could be reduced.

One future direction is to apply various optimizations, e.g., compressing VM states, to the forwarding of VM states from the original destination. Also, we need to develop a more flexible work-stealing mechanism. For example, the original destination should steal back VM states from the source when the transfer speed of the source becomes slower after work stealing. Another direction is to automatically decide the best option among destination switching, the restart method, and the static method. Our results in Table 2 would help with this decision. In addition, we would like to apply DCmigrate to the migration of VMs with a larger amount of memory and improve its implementation.

Acknowledgment

This work was partially supported by JST, CREST Grant Number JPMJCR21M4, Japan.

References

[1] Amazon Web Services, Inc., "Amazon EC2 High Memory Instances," https://aws.amazon.com/ec2/instance-types/u7i/.

[2] U. Deshpande and K. Keahey, "Traffic-sensitive Live Migration of Virtual Machines," *Future Generation Computer Systems*, vol. 72, pp. 118–128, 2017.

[3] A. Hadas and E. Haas, "Virtual Machine Migration to Multiple Destination Nodes," US Patent 11,169,844, 2021.

[4] The QEMU Project Developers, "QEMU QMP Reference Manual (migrate_cancel command)," https://www.qemu.org/docs/master/interop/qemu-qmp-ref.html.

[5] Red Hat, Inc., "libvirt: The Virtualization API," https://libvirt.org/.

[6] The QEMU Project Developers, "QEMU," https://www.qemu.org/.

[7] Red Hat Inc., "Virtual Machine Manager," https://virt-manager.org/.

[8] The QEMU Project Developers, "QEMU Machine Protocol Specification," https://www.qemu.org/docs/master/interop/qmp-spec.html.

[9] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux Virtual Machine Monitor," in *Proc. Ottawa Linux Symp.*, 2007, pp. 225–230.

[10] A. Toosi and R. Buyya, "Acinonyx: Dynamic Flow Scheduling for Virtual Machine Migration in SDN Enabled Clouds," in *Proc. Int. Symp. Parallel and Distributed Processing with Applications*, 2018, pp. 886–894.

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

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

[13] U. Deshpande, Y. You, D. Chan, N. Bila, and K. Gopalan, "Fast Server Deprovisioning through Scatter-Gather Live Migration of Virtual Machines," in *Proc. Int. Conf. Cloud Computing*, 2014, pp. 376–383.

[14] M. R. Hines and K. Gopalan, "Post-Copy Based Live Virtual Machine Migration Using Adaptive Pre-Paging and Dynamic Self-Ballooning," in *Proc. Int. Conf. Virtual Execution Environments*, 2009, pp. 51–60.

[15] U. Deshpande, D. Chan, T. Guh, J. Edouard, K. Gopalan, and N. Bila, "Agile Live Migration of Virtual Machines," in *Proc. Int. Parallel and Distributed Processing Symp.*, 2016.

[16] S. Akiyama, T. Hirofuchi, R. Takano, and S. Honiden, "MiyakoDori: A Memory Reusing Mechanism for Dynamic VM Consolidation," in *Proc. Int. Conf. Cloud Computing*, 2012, pp. 606–613.

[17] T. Knauth and C. Fetzer, "VeCycle: Recycling VM Checkpoints for Faster Migrations," in *Proc. Annual Middleware Conf.*, 2015, pp. 210–221.

[18] X. Song, J. Shi, R. Liu, J. Yang, and H. Chen, "Parallelizing Live Migration of Virtual Machines," in *Proc. Int. Conf. Virtual Execution Environments*, 2013, pp. 85–96.