An Efficient State-Saving Mechanism for Out-of-band Container Migration

Yuki Asakura
Kyushu Institute of Technology
asakura@ksl.ci.kyutech.ac.jp

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

***Abstract*—Many clouds provide containers as lightweight virtualized environments inside virtual machines (VMs). Containers can be migrated between source and destination VMs for various reasons such as load balancing. However, the performance of container migration is largely degraded by the load and virtualization overhead of VMs because the migration mechanism runs inside VMs. Conversely, the performance of containers is largely affected by the load of container migration. This paper proposes OVmigrate to enable out-of-band container migration, which migrates a container running inside a VM from the outside of the VM. OVmigrate analyzes and obtains the states of a container in the memory of the source VM using a technique called VM introspection. It enables the migration mechanism outside a VM to independently save the states of a container running inside the VM. We have implemented OVmigrate for Linux and KVM and compared the performance of state saving with the existing tool called CRIU running inside the VM.**

***Index Terms*—Container, migration, virtual machine, VM introspection, high load**

1. Introduction

Recently, clouds that provide containers are widely used, e.g., Amazon Elastic Container Service (ECS) [1] and Elastic Kubernetes Service (EKS) [2], Google Kubernetes Engine (GKE) [3], and Microsoft Azure Kubernetes Service (AKS) [4]. A container is a lightweight virtualization environment provided by the operating system (OS) and consists of several processes and its execution environment. In clouds, containers often run in virtual machines (VMs) to make node management easier [1]–[4]. A container can be moved between VMs using a technique called *container migration*. For example, container migration can be used for load balancing, which moves several containers in an overutilized VM to underutilized VMs at the same or different hosts.

Container migration often has to be performed when the load of the source VM is high. As mentioned above, load balancing is necessary to decrease the load of overutilized VMs. In such a case, the performance of container migration can be largely degraded because the migration mechanism runs inside the overutilized source VM. In addition, the performance of container migration is largely affected by the virtualization overhead introduced by VMs. For example, the performance of I/O, such as virtual disks and networks, is lower in a VM. Since the load of container migration itself is high, the migration mechanism inside the source VM can also affect the performance of containers largely.

This paper proposes OVmigrate to enable *out-of-band container migration*, which migrates a container running inside a VM from the *outside* of the VM. OVmigrate obtains the states of a container stored in the memory of the source VM using a technique called *VM introspection (VMI)* [5]. Then, it transfers the saved states to the destination host and restores a container in the destination VM. OVmigrate can achieve rapid container migration without the impact of the load or virtualization overhead of VMs. In addition, it prevents container migration from affecting the performance of containers in the source VM.

We have implemented OVmigrate using KVMonitor [6] and LLView [7]. OVmigrate shares the memory of a VM with the migration mechanism running outside the VM and obtains OS data from the shared memory. Then, it analyzes the OS data using the source code of the OS. Since the states of a container mainly consist of the states of processes running in the container, OVmigrate saves information on process memory, files, threads, clocks, resource control, etc. Our experiments showed that OVmigrate could save the states of a process from the outside of a VM and that the existing tool called CRIU [8] could successfully restore that process using the saved states inside the VM. Compared with CRIU, the performance of state saving was up to 7.3x faster.

The organization of this paper is as follows. Section 2 describes issues in container migration. Section 3 proposes OVmigrate for saving the states of a container outside a VM, and Section 4 explains its implementation. Section 5 shows experiments to examine the performance of state saving. Section 6 describes related work, and Section 7 concludes this paper.

2. Background

Recently, clouds such as Amazon ECS and EKS, Google GKE, and Microsoft AKS provide containers. A container is a lightweight virtualized environment provided by the OS. Unlike traditional virtualization using VMs, containers share

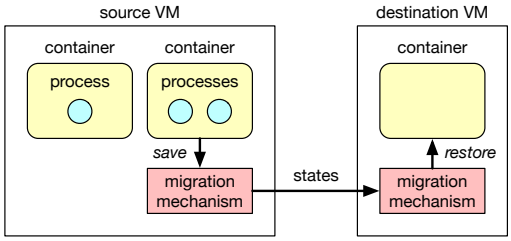
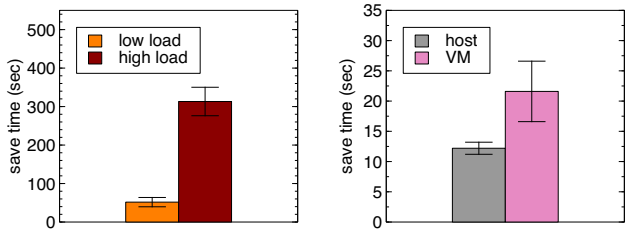


Figure 1: Traditional container migration inside VMs.



(a) Load (b) Virtualization

Figure 2: The impact of the load and virtualization overhead of a VM on state saving.

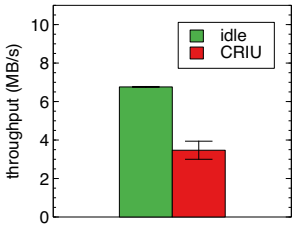


Figure 3: The impact of state saving on an in-memory database inside a VM.

the underlying OS kernel. A container runs several processes on top of the OS and provides an execution environment such as filesystems and network interfaces. Since a container consists of a smaller amount of resources, it can boot and run faster than a VM. In clouds, containers often run in VMs [1]–[4]. This is because it is more flexible to manage VMs than physical hosts.

Like VM migration, a container running in a VM can be moved to another VM using a similar technique. Fig. 1 illustrates traditional container migration. First, the migration mechanism in the source VM saves the states of the running processes and the execution environment of a container. Then, it transfers the saved states to the destination VM via the virtual network. The migration mechanism in the destination VM restores the container from the received states. Using container migration, load balancing can be achieved between VMs by migrating containers from overutilized VMs to underutilized VMs at the same or different hosts. When administrators update the OS in a VM and reboot the VM, they can continue to run containers by migrating them to other VMs in advance.

Since many containers are consolidated into one VM, containers often have to be migrated when the load of the VM running them is high. For example, load balancing with container migration is performed when a high VM load is detected. The load of the VM running a container can further increase while that container is being migrated. In such cases, container migration can be largely affected by the load of the VM. This leads to large performance degradation of container migration. Conversely, the load of container migration can affect the performance of containers running in the same VM.

In terms of VMs, load balancing with VM migration has been explored extensively. For example, Sandpiper [9] recommends performing VM migration when the utilization of CPUs or networks exceeds 75%. This threshold is determined to accommodate the overhead of VM migration because VM migration degrades server and network performance by 50% and 20%, respectively. However, low resource utilization results in an increase in the cost of clouds. Also, it is reported that Google Compute Engine (GCE) adjusts the speed of VM migration when the load of the source host is high [10]. Using this method, load balancing cannot be completed rapidly.

In addition, container migration is affected by the virtualization overhead of VMs because the migration mechanism also runs inside VMs. In fact, it is reported that network virtualization is the root cause of performance degradation and increases CPU utilization by 70% and 118% in the source and destination VMs, respectively [11]. Therefore, Portkey [11] bypasses the virtual network to transfer the states in container migration. However, it needs to modify the migration mechanism and the guest OS in a VM.

According to our experiments, the performance of container migration is also degraded due to inter-process communication and disk I/O by the migration mechanism in the source VM. Fig. 2(a) shows the impact of a high load inside a VM on saving the states of a process. Compared with under a low load, the save time is 6.1x longer and more unstable. Fig. 2(b) shows how virtualization affects the performance of state saving. The save time in a VM is 20-77% longer and much more unstable than without a VM. In addition, Fig. 3 shows that state saving degrades the performance of an in-memory database inside a VM by 2x.

3. OVmigrate

This paper proposes OVmigrate to enable *out-of-band container migration*, which migrates containers outside VMs for lightweight container migration. Fig. 4 illustrates out-of-band container migration with OVmigrate. As described in Section 2, traditional container migration runs the migration mechanism inside VMs with the target container. In contrast, OVmigrate runs the migration mechanism in the same host but outside VMs. At the source host, the migration mechanism outside the VM saves the states of a

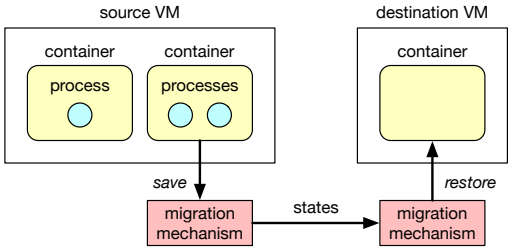


Figure 4: Out-of-band container migration with OVmigrate.

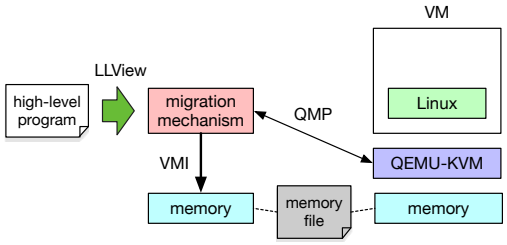


Figure 5: State saving with VMI.

container running inside the VM. Then, it transfers the saved states to the destination host using the physical network, not the virtual one. At the destination host, the migration mechanism outside the VM restores that container inside the VM using the received states. Since the source VM tends to be overutilized, we focus on saving the states of a container outside the source VM in this paper.

Since OVmigrate performs the state saving of a container outside a VM, it can easily protect the migration mechanism from the load of the VM. It can assign a fixed amount of resources such as CPUs and memory to the VM and prevent the VM from using extra resources. It can limit the I/O bandwidth to the VM and suppress the load of the VM even when the VM submits a large amount of I/O. In addition, the migration mechanism can avoid virtualization overhead introduced by VMs because it does not run inside the VM. Conversely, the migration mechanism does not affect the performance of containers running inside the VM because the host can usually assign surplus resources to the migration mechanism.

OVmigrate mainly saves the states of processes as the states of a container. A container consists of running processes and their execution environment, but most of the states are those of the processes. In addition, most of the information on the execution environment of a container is included in the states of processes. The states of a process consist of information on the memory assigned to the process, files used by the process, threads created in the process, clocks used by the process, etc. Also, the states of an execution environment consist of the resource assignment and limitation to a process group.

OVmigrate obtains the states of a process in a VM using *VMI* [5]. VMI is a technique for obtaining information on the system inside a VM from the outside of the VM. It analyzes the data of the OS and processes in the memory of the VM. OVmigrate finds the process structure from the process identifier (ID) and obtains various states of the process by traversing pointers stored in the structure. VMI enables the migration mechanism to independently run outside a VM without running a helper process or modifying the guest OS inside the VM.

To control the execution of a process to be saved from the outside of a VM, OVmigrate sends signals to the process using *pseudo signal sending* [12]. It pauses a process by sending the STOP signal when it saves the states. This enables the migration mechanism to consistently save the states of the process without any updates from the process itself. When the migration mechanism completes saving the states, OVmigrate terminates the process in the source VM by sending the KILL signal. Pseudo signal sending is achieved by rewriting kernel data structures in memory using extended VMI. It changes the states of a process as if a signal is sent to the process. In addition, it performs *pseudo process scheduling* [12] to deliver the signal to the target process immediately.

4. Implementation

We have implemented OVmigrate for Linux 5.4 as a guest OS in VMs running on top of QEMU-KVM 4.2.0 [13]. The states that OVmigrate currently saves are information on memory, files, threads, clocks, resource control, and so on. All of them are saved by CRIU [8], which is the traditional tool for saving and restoring the states of processes. This enables CRIU to restore the process inside the destination VM.

4.1. State Saving with VMI

OVmigrate performs VMI using KVMonitor [6], as illustrated in Fig. 5. First, OVmigrate prepares the memory assigned to a VM as a special file called a memory file. It maps the memory file to the VM and provides physical memory. It also maps that file to the migration mechanism so that the migration mechanism can access the memory of the VM. Next, OVmigrate communicates with QEMU-KVM using the QEMU machine protocol (QMP) and obtains the address of the page tables, which is stored in the CR3 register of the virtual CPUs in the VM. Using the obtained page tables, it translates the virtual addresses of OS data into physical ones and accesses the memory of the VM using those addresses.

To develop the migration mechanism that obtains the states of processes using VMI, we used the LLView framework [7]. It is not an easy task to analyze OS data in the memory of a VM because developers need to find necessary data using only low-level information. LLView enables users to develop systems that analyze OS data with VMI as kernel modules. Developers can write high-level programs using the kernel data structures and global variables defined in the header files of the Linux kernel. LLView compiles developed programs and converts generated LLVM intermediate code to embed code for accessing the memory of a VM.

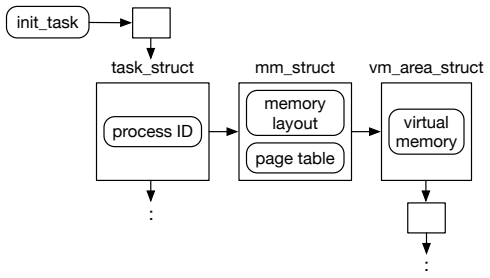


Figure 6: Analyzing memory information.

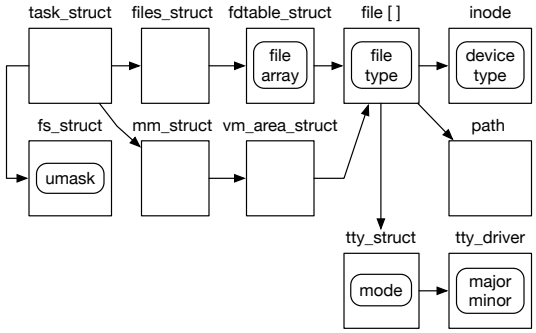


Figure 7: Analyzing file information.

4.2. Saving Memory Information

As information on memory used by a process, OVmigrate saves the layout of process memory, as illustrated in Fig. 6. First, OVmigrate traverses the process list from the init_task kernel variable in the memory of a VM and finds the task_struct structure corresponding to the specified process ID. Next, it follows the pointer in that structure and finds the mm_struct structure, which contains information on the entire process memory. Then, it obtains the start and end addresses of the code, data, and heap areas, and the start address of the stack. In addition, it obtains the start and end addresses of the areas storing the arguments passed to the process and the environment variables.

As information on virtual memory, OVmigrate saves information on all the virtual memory areas allocated to the process. For this purpose, it finds the vm_area_struct structure from the mm_struct structure. Then, it obtains the start and end addresses of virtual memory areas, access rights, flags, and status. For a virtual memory area that maps a file, it also obtains the file offset.

As information on page mapping, OVmigrate saves the start addresses of virtual memory areas and the number of physical pages actually assigned. The former information is included in the information on virtual memory areas saved above, but it is saved independently so that it corresponds to the saved data of memory pages. Since physical pages are not assigned to all the pages in the virtual memory areas, OVmigrate checks whether physical pages are assigned or not from the start address of each virtual memory area obtained from the vm_area_struct structure. To do this, it examines the page tables of the process obtained from the mm_struct structure. If there is no page table entry or if the present bit in the corresponding page table entry is zero, OVmigrate does not save information on that page. In addition, it does not save information on a page if the page is not anonymous. Finally, it divides the memory region where physical pages are assigned and contiguous into 1024 pages at maximum and saves information on these chunks.

As actual memory data, OVmigrate saves the contents of the pages that are assigned to the process and where files are not mapped. For an existent page, it obtains the 4-KB data of the page. Then, it saves the data in the order of the virtual memory areas saved in the page mapping.

4.3. Saving File Information

For the files that the process opens, OVmigrate saves information on their file descriptors, as illustrated in Fig. 7. First, it finds the files_struct structure from the task_struct structure. Then, it finds the fdtable structure from that structure. These structures manage information on all the files opened by the process. The fdtable structure has an array of the file structure, which stores information on a file. OVmigrate obtains the file descriptor number from the index in that array. It also obtains the file type, the flags associated with the file descriptor, and the value of the file pointer from the file structure. In addition, it calculates an ID using that file pointer and the inode number stored in the inode structure, which OVmigrate finds from the file structure.

Furthermore, OVmigrate saves detailed information for each file type. In the case of an executable file and shared libraries loaded into the process and normal files mapped onto the process address space, it finds the file structure from the vm_area_struct structure. Then, it finds the inode structure from that structure and obtains the file size and the detailed type. Also, it recursively traverses the directories from the path structure contained in the file structure and obtains the path name of the file.

In the case of terminal devices (tty) such as standard I/O, OVmigrate saves detailed information on the devices. It first finds the tty_struct structure from the file structure. From this structure, it obtains the I/O modes, control characters, the I/O speed, etc., which are defined in the POSIX termios. It also obtains the state of the terminal device and the IDs of the session and the process group that controls the terminal device. Furthermore, it finds the tty_driver structure from the tty_struct structure and calculates the ID of the terminal device using the major and minor device numbers obtained from that structure. In addition, it obtains the user and group IDs of the device from the inode structure.

As information on filesystems, OVmigrate saves the access permission (umask) used for creating new files and directories. It finds the fs_struct structure from the task_struct structure and obtains the value of umask.

4.4. Saving Thread Information

As the execution state of the process, OVmigrate saves the states of threads. Since register information depends on processors, OVmigrate currently supports x86-64 processor families. First, it obtains the address of the kernel stack assigned to the process. Since the CPU registers of a stopped process are stored in the kernel stack, OVmigrate obtains the values of CPU registers from the kernel stack. For the FSBASE and GSBASE registers, OVmigrate finds the thread_struct structure from the task_struct structure and obtains the values stored in the structure or calculates the values from the information stored in the structure. For the XMM registers and the segment descriptors, it obtains the values from this structure.

4.5. Saving Clock Information

As clock information, OVmigrate saves the monotonic time and the boot time. To calculate the monotonic time, it obtains the wall clock and the offset from the wall clock stored in the timekeeper structure, which is pointed by the tk_core kernel variable. Then, it adds the offset to the wall clock. To calculate the boot time, it obtains the offset from the monotonic time and adds the offset to the monotonic time. For these times, it is necessary to add time differences from the last update on the wall clock. This needs to obtain the value of the timestamp counter by executing the RDTSC instruction inside the VM. However, OVmigrate cannot obtain this value outside the VM. In the current implementation, it assumes that this time difference is zero, although the saved times contain slight errors.

4.6. Saving Cgroup Information

As the information of the control group (cgroup), OVmigrate saves the names and paths of its subsystems, the parameters, and their values. Cgroup is a mechanism for assigning and limiting resources to a process group. It is used to isolate a process from the other containers. In cgroup, subsystems exist for each resource such as memory and CPUs and have several parameters. For example, the maximum size of available memory is set to the limit_in_bytes parameter in the memory subsystem.

As illustrated in Fig. 8, OVmigrate first finds the css_set structure from the task_struct structure and then finds the cgroup structure from that structure. The cgroup structure contains the node information of the kernel filesystem (kernfs). OVmigrate obtains the path by traversing kernfs from that node to the root node. Next, it obtains the names of the parameters from the cgroup_subsys structure provided for each subsystem. Also, it finds the cgroup_subsys_state structure from css_set structure and then finds a subsystem-specific structure from that structure. For example, the cpuset structure contains parameters in the CPU subsystem. The mem_cgroup structure contains parameters in the memory subsystem. OVmigrate obtains the values of the parameters from these structures.

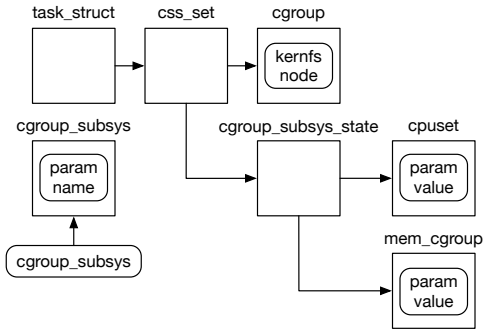


Figure 8: Analyzing cgroup information.

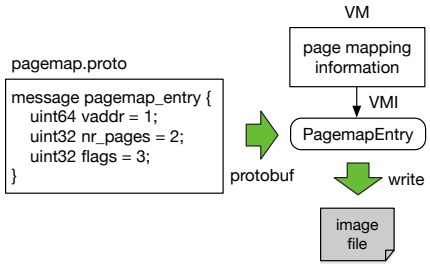


Figure 9: Saving process information with protocol buffer.

4.7. Serialization with Protocol Buffer

OVmigrate saves the states of a process in the same format as used in CRIU 3.16. CRIU defines the states in the proto files and serializes the states using the protocol buffer [14]. A proto file defines a message for each type of the states and defines each state in each field of the message. OVmigrate uses the same proto files, saves the states using the protocol buffer for C [15], and writes them to image files, as illustrated in Fig. 9.

4.8. Controlling Saved Processes

OVmigrate sends signals to processes in a VM using pseudo signal sending provided by VMMfas [12] to pause and terminate the processes. It finds the sigpending structure from the task_struct structure and sets the bit corresponding to the signal bitmap in the structure. Then, it finds the thread_info structure from the task_struct structure and sets the pending flag. When the kernel schedules this process, it checks this flag and delivers signals to the process if necessary.

Since only this cannot deliver a signal to a paused process, OVmigrate resumes the process using pseudo process scheduling. It finds the sched_entity structure from the task_struct structure and adds it to the red-black tree contained in the cfs_rq structure, which is used by the CFS scheduler in Linux. Finally, it changes the state of the process contained in the task_struct to be runnable. When the kernel performs process scheduling, it schedules this process soon.

5. Experiments

We conducted several experiments to examine the effectiveness of OVmigrate. First, we confirmed that OVmigrate could save the states of a process outside a VM. Then, we measured the time needed to save the states under various loads 10 times. For comparison, we measured the time to save the states inside a VM using CRIU. In addition, we examined the performance impact on another process inside the VM by state saving. We used a PC with an Intel Core i7-10700 processor, 64 GB of memory, and 2 TB of SATA HDD. We ran Linux 5.4 and QEMU-KVM 4.2.0 as virtualization software on this PC. We assigned two virtual CPUs and 30 GB of memory to a VM by default and ran Linux 5.4 in the VM.

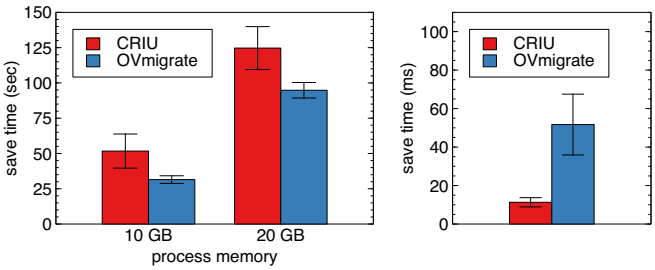
5.1. Correctness of State Saving

To confirm the capabilities of OVmigrate, we saved the states of a process outside a VM. As a process, we executed a program that increased a counter value every second and showed the value. This program used an interval timer, the ALRM signal, and the standard output. When OVmigrate completed saving the states, the process was terminated. After that, we transferred the saved states to the inside of the VM and restored the process using CRIU. As a result, the process was restored correctly and continued to show the counter value. This means that OVmigrate could save the states of the process correctly.

5.2. State Saving under a Low Load

We measured the time needed to save the states of a process under a low load. In this experiment, we ran a process using memory dynamically allocated between 0 and 20 GB in the VM with 30 GB of memory. Fig. 10 shows the save time in OVmigrate and CRIU. When the process allocated 10 and 20 GB of memory, OVmigrate was 1.6x and 1.3x faster than CRIU, respectively, because it was not affected by the virtualization overhead of the VM. The variance was also smaller in OVmigrate, which means that OVmigrate could save the states of the process more stably. In contrast, OVmigrate was 4.5x slower than CRIU when the process did not dynamically allocate memory at all. This is due to the overhead of analyzing the complex data structures of the OS using VMI. In this case, the variance was larger in OVmigrate.

Next, we examined the performance when the amount of free memory was smaller at the host level. We assigned 50 GB of memory to the VM, so that the host-level free memory was only 11 GB. In this experiment, we ran a process using memory dynamically allocated between 10 and 40 GB in the VM. Fig. 11 shows the save time. When the process allocated 10 GB of memory, OVmigrate was 2.9x slower than CRIU. This is because the host-level page cache overflowed when OVmigrate saved more than 10 GB of process memory to the image file. As a result, the host OS wrote dirty data on the disk during state saving. In contrast, OVmigrate became faster than CRIU when the process allocated more than 20 GB of memory. For 40 GB, the performance was improved by 1.3x. This is because the guest-level page cache also overflowed due to state saving by CRIU inside the VM.



(a) Large processes (b) Small process

Figure 10: The save time under a low load.

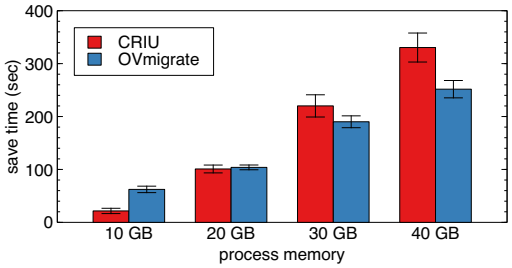


Figure 11: The save time with less host-level memory under a low load.

5.3. State Saving under a High CPU Load

We measured the time needed to save the states of a process under a high CPU load. We ran stress-ng [16] inside the VM to generate a high CPU load. First, we ran a process using 10 GB of memory in the VM with 30 GB of memory and compared the save time between low and high loads. As shown in Fig. 12, CRIU became 1.8x slower under a high CPU load. This means that CRIU was largely affected by the CPU load inside the VM. In contrast, OVmigrate was only 11% slower because the CPU load inside the VM almost did not affect the migration mechanism using CPUs outside the VM.

Next, we compared the save time between OVmigrate and CRIU. We ran a process using memory dynamically allocated between 10 and 20 GB in the VM with 30 GB of memory. As shown in Fig. 13, OVmigrate was 2.6x and 2.0x faster than CRIU, respectively. This performance improvement is much larger than in a low load. This means that OVmigrate could reduce the impact of the load inside a VM more largely than that of virtualization overhead. The variance of the save time was also smaller.

Similarly, we measured the save time when the host-level free memory was smaller, i.e., 11 GB. We ran a process

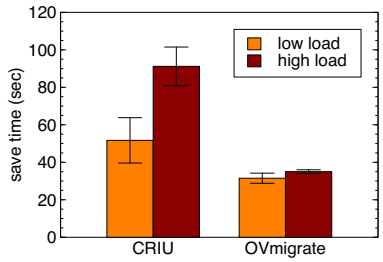


Figure 12: Comparison between low and high loads.

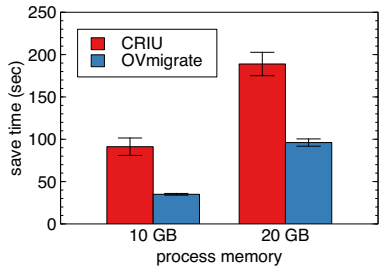


Figure 13: The save time under a high CPU load.

using memory between 10 and 40 GB inside the VM with 50 GB of memory. Unlike under a low load, OVmigrate was always faster than CRIU, as shown in Fig. 14. As the amount of memory used by the process increased, the performance improvement became larger. OVmigrate was 1.5x faster for 10 GB of memory, whereas it was 1.9x faster for 40 GB of memory. This is because the guest-level page cache overflowed more largely due to state saving by CRIU inside a VM when the size of process memory became larger.

5.4. State Saving under Various Loads

We measured the time for saving the states of a process under various loads. We generated a high load on CPUs, pipe, memory, I/O, the filesystem, and the device by running stress-ng inside the VM. We ran a process using 10 GB of memory in the VM with 30 GB of memory. Fig. 15 shows the save time in OVmigrate and CRIU. CRIU was affected by all but the device load. Compared with a low load, the save time was 4.4x and 6.1x longer under the pipe and filesystem loads, respectively. Especially, it failed to save the states for the loads on memory and I/O.

In contrast, OVmigrate always succeeded in saving the states of the process. For the pipe load, the save time was the same as in a low load. Since CRIU was largely affected by the pipe load, OVmigrate was 7.4x faster than CRIU. However, it was largely affected by the loads on I/O and the filesystem. These save times were 13x longer than in a low load. Especially, OVmigrate was 23% slower than CRIU under the filesystem load. This is because we did not explicitly limit the amount of resources assigned to the VM.

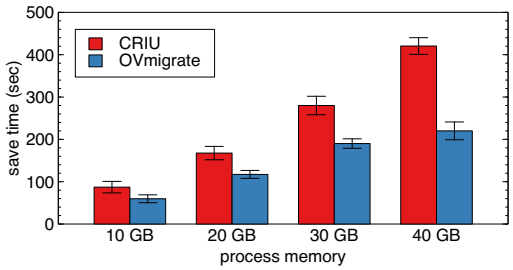


Figure 14: The save time with less host-level memory under a high CPU load.

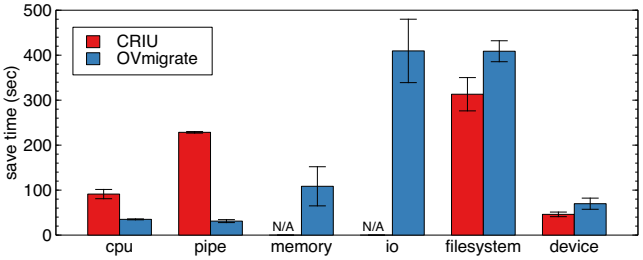


Figure 15: The save time under various loads.

If we limit the usage of such shared I/O, the performance of OVmigrate would be improved.

5.5. Performance Impact of State Saving

To examine the impact on other processes inside a VM by state saving, we measured the time needed to store data in an in-memory database. We ran a process using 30 GB of memory in the VM with 50 GB of memory. We also ran memcached [17] in the VM and stored data of 1 GB during state saving. As a baseline, we measured the time while the VM was idle. As shown in Fig. 16, OVmigrate did not affect the performance of memcached at all. In contrast, CRIU slowed down memcached by 2x due to resource contention in the VM.

6. Related Work

Portkey [11] enables containers in VMs to be efficiently migrated by optimizing network transfers. When CRIU transfers the states of a container, it invokes the kernel module provided by Portkey. The kernel module invokes the hypervisor by bypassing network processing in the guest OS. The hypervisor transfers the states to the destination host. In the destination VM, CRIU receives the states from the hypervisor via the kernel module. Portkey can suppress CPU utilization during container migration by network transfers at the hypervisor layer, but it cannot reduce the migration time. In contrast, OVmigrate can transfer the states of a container completely outside the source VM. This could

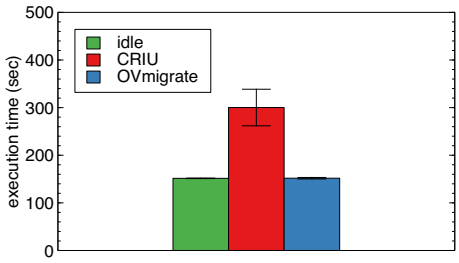


Figure 16: The performance of memcached during state saving.

eliminate CPU utilization more largely and lead to the reduction of the migration time.

mWarp [18] relocates process memory between VMs for container migration and does not perform the time-consuming copy or transfer of memory data. In the source VM, CRIU invokes the hypervisor via the system call and notifies information on process memory. In the destination VM, CRIU invokes the hypervisor and re-maps the memory of the source VM to the destination VM to complete transferring process memory. However, mWarp is applicable only to container migration between VMs in the same host. OVmigrate can perform the same optimization when the source and destination VMs exist in the same host.

Similarly, VMBeam [19] enables the zero-copy migration of a guest VM running in a host VM. The migration mechanism invokes the hypervisor in the source and destination host VMs. The hypervisor swaps the memory of a guest VM between the two host VMs to complete memory transfers efficiently. VMBeam is also an optimization that is applicable only to VM migration in the same host.

Sledge [20] enables efficient live migration of Docker containers. It does not transfer redundant layers in the hierarchical image used by a container at the source host. It repeatedly transfers only the differences of process memory using incremental checkpointing provided by CRIU. It does not perform the time-consuming reload of the Docker daemon at the destination host but loads only the management context. OVmigrate can use some of these optimization techniques.

7. Conclusion

This paper proposed OVmigrate to enable out-of-band container migration. OVmigrate analyzes the memory of a VM using VMI and saves the states of a container running inside the VM from the outside of the VM. It can minimize the impact of the load and virtualization overhead of VMs on the performance of container migration. Also, it can minimize the impact of the load of container migration on the performance of containers. Our experiments showed that OVmigrate could save the states of a process up to 7.3x faster.

Our future work is to support state saving for various containers. For example, OVmigrate needs to save the states of processes that use unsupported OS functions and those managed by a container engine like Docker [21]. Currently, we are developing a mechanism for state restoring outside VMs. Using this mechanism, OVmigrate can completely achieve out-of-band container migration.

Acknowledgments

This work was partially supported by JST, CREST Grant Number JPMJCR21M4, Japan. These research results were partly obtained from the commissioned research (JPJ012368C05501) by National Institute of Information and Communications Technology (NICT), Japan.

References

[1] Amazon Web Services, Inc., "Amazon Elastic Container Service," https://aws.amazon.com/ecs/.
[2] ——, "Amazon Elastic Kubernetes Service," https://aws.amazon.com/eks/.
[3] Google, Inc., "Google Kubernetes Engine," https://cloud.google.com/kubernetes-engine.
[4] Microsoft Corporation, "Azure Kubernetes Service," https://azure.microsoft.com/en-us/products/kubernetes-service/.
[5] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proc. Network and Distributed Systems Security Symp.*, 2003, pp. 191–206.
[6] K. Nakamura and K. Kourai, "Efficient VM Introspection in KVM and Performance Comparison with Xen," in *Proc. Pacific Rim Int. Symp. Dependable Computing*, 2014, pp. 192–202.
[7] Y. Ozaki, S. Kanamoto, H. Yamamoto, and K. Kourai, "Reliable and Accurate Fault Detection with GPGPUs and LLVM," in *Proc. Int. Conf. Cloud Computing*, 2023, pp. 540–546.
[8] OpenVZ Team, "CRIU," https://criu.org/Main_Page.
[9] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Black-box and Gray-box Strategies for Virtual Machine Migration," in *Proc. Symp. Networked Systems Design & Implementation*, 2007.
[10] A. Ruprecht, D. Jones, D. Shiraev, G. Harmon, M. Spivak, M. Krebs, M. Baker-Harvey, and T. Sanderson, "VM Live Migration at Scale," in *Proc. Int. Conf. Virtual Execution Environments*, 2018, pp. 45–56.
[11] C. Prakash, D. Mishra, P. Kulkarni, and U. Bellur, "Portkey: Hypervisor-Assisted Container Migration in Nested Cloud Environments," in *Proc. Int. Conf. Virtual Execution Environments*, 2022, pp. 3–17.
[12] K. Kimura and K. Kourai, "Xfas: Fault Recovery by Externally Controlling OS Behavior," in *Proc. Int. Conf. Utility and Cloud Computing*, 2023.
[13] F. Bellard, "QEMU," https://www.qemu.org/.
[14] Google Inc., "Protocol Buffers," https://developers.google.com/protocol-buffers.
[15] D. Benson, "Protocol Buffers Implementation in C," https://github.com/protobuf-c/protobuf-c.
[16] C. I. King, "Stress-ng," https://github.com/ColinIanKing/stress-ng.
[17] B. Fitzpatrick, "memcached – A Distributed Memory Object Caching System," http://memcached.org/.
[18] P. Sinha, S. Doddamani, H. Lu, and K. Gopalan, "mWarp: Accelerating Intra-Host Live Container Migration via Memory Warping," in *Proc. Conference on Computer Communications Workshops*, 2019.
[19] H. Ooba and K. Kourai, "Zero-copy Migration for Lightweight Software Rejuvenation of Virtualized Systems," in *Proc. Asia-Pacific Workshop on Systems*, 2015.
[20] B. Xu, S. Wu, J. Xiao, H. Jin, G. Shi, J. Rao, L. Yi, and J. Jiang, "Sledge: Towards Efficient Live Migration of Docker Containers," in *Proc. Int. Conf. Cloud Computing*, 2020, pp. 321–328.
[21] Docker Inc., "Docker: Accelerated Container Application Development," https://www.docker.com/.