Efficient Container Restoration outside Heavily Loaded VMs

Shota Kimoto
Kyushu Institute of Technology
shota@ksl.ci.kyutech.ac.jp

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

***Abstract*—Containers are widely used in clouds to provide various services. Users can periodically save the state of a container using the checkpoint mechanism and rapidly restore it using the restoration mechanism when they detect an anomaly in the container. Since a container often runs in a VM, the performance of container checkpointing and restoration is affected by virtualization overhead and the load of the VM. Previous work addresses this issue by saving the state of a container using VM introspection, but it is not easy to restore a container outside a VM. This paper proposes OVrestorer for enabling efficient container restoration outside a VM by cooperating with a helper inside the VM. In OVrestorer, the OVR core outside a VM primarily restores the state of a container without the impacts of virtualization overhead and the VM load. It analyzes the memory of the VM and overwrites the state of a container with the saved state. In addition, the OVR helper inside a VM offers minimal support to handle virtual hardware and memory management. Our experiments showed that OVrestorer could restore a container more rapidly than the traditional in-VM method.**

***Index Terms*—Container, checkpointing, restoration, virtual machine, virtualization**

1. Introduction

Recently, containers are provided by various container orchestration services in clouds, such as Amazon Elastic Container Service (ECS) [1], Google Kubernetes Engine (GKE) [2], and Microsoft Azure Kubernetes Service (AKS) [3]. A container is a lightweight virtualized environment that consists of several processes of the operating system (OS) and their execution environment, such as a filesystem and a network. Most existing orchestration services restart a container when they detect an anomaly, such as a failure or an attack, in the container [4]–[6]. This method is reasonable if the container has less state, but it takes time to restore the container if the container has more state. For example, containers used for in-memory databases and big data processing store a large amount of data in memory.

For such containers, container orchestration services can periodically save the state of a container using the container checkpointing mechanism [7]. When they detect an anomaly in a container, they can rapidly restore the container from a checkpoint using the container restoration mechanism. However, containers often run inside virtual machines (VMs) in clouds. As a result, container management systems running in VMs are affected by virtualization overhead when they perform container checkpointing and restoration. When the loads of the VMs are high, the performance of container checkpointing and restoration is affected more significantly. Previous work [8] addresses these issues by saving the state of a container using VM introspection [9]. However, it is not easy to restore a container outside a VM using the same technique because container restoration needs to not only observe the state in memory but also change the system and hardware state.

This paper proposes *OVrestorer* for enabling efficient container restoration outside a VM by cooperating with an in-VM helper. In OVrestorer, the *OVR core* running outside a VM primarily restores the state of a container by directly rewriting OS data in the memory of the VM. It reads a checkpoint file to obtain the state saved at checkpointing, but it does not suffer from virtualization overhead caused by access to the virtual disk. The performance of the OVR core is basically not degraded by the load of the VM, but it can be affected by access to shared resources, such as disks, between the host and the VM. To suppress performance degradation by accessing shared resources, OVrestorer limits the resources of the VM, such as disk bandwidth, only during container restoration.

Additionally, the *OVR helper* runs inside the VM and offers minimal support for container restoration. It efficiently communicates with the OVR core outside the VM using a VM socket (Vsock) [10]. For example, it creates a new empty container in the VM so that the OVR core can rewrite its state. It also configures timer hardware for CPU limits and interval timers. Since the communication overhead is not small, OVrestorer asynchronously allocates and deallocates kernel objects without communication. The OVR helper stores pre-allocated kernel objects in the object pool, while the OVR core uses them when necessary. The OVR core stores unnecessary kernel objects in the object pool, while the OVR helper deallocates them later. The OVR helper suffers from virtualization overhead and the VM load, but it can suppress that impact by minimizing I/O and running in the OS kernel.

We implemented OVrestorer for a VM that ran Linux

5.15 as a guest OS and ran on QEMU-KVM 4.2.0 [11]. To observe and change OS data in the memory of the VM outside the VM, we used KVMonitor [12] and LLView [13]. According to our experiments, OVrestorer could correctly restore a container from a checkpoint file. Compared with the in-VM container management system, it was up to 2.9x faster under a low load and up to 5.2x faster under a high load.

The organization of this paper is as follows. Section 2 describes performance degradation of container checkpointing and restoration. Section 3 proposes OVrestorer for efficient container restoration outside VMs, and Section 4 explains its implementation. Section 5 shows experimental results for container restoration using OVrestorer. Section 6 describes related work, and Section 7 concludes this paper.

2. Container Checkpointing and Restoration

Since containers run complex services, they often suffer from anomalies, such as failures and attacks. Most container orchestration services restart a container when they detect an anomaly in the container [4]–[6]. This method can recover a container from a transient fault, but it is useless for a persistent fault. In addition, restarting a container is reasonable if the container has less state, but it takes a long time to restore a container with a large amount of state, especially in-memory data. When a container is compromised, it is desirable to abandon the entire state of the container because the entire system in the container can be affected. In this case, the user must re-create a new container from scratch.

For such containers, container orchestration services can prepare for anomalies by periodically saving the state of containers [7]. Using the checkpointing mechanism, the container management system saves the state of the processes running in a container and the state of the filesystem used for the container to a checkpoint file. When container orchestration services detect an anomaly in the container, they can rapidly recover the container using one of the checkpoint files. Using the restoration mechanism, the container management system creates a new container and restores the state of the processes and the filesystem. This method can recover a container even from a persistent fault and an attack if it uses the checkpoint file saved before an anomaly.

In cloud environments, containers often run in VMs because cloud providers can manage VMs more flexibly than physical machines. For example, they can easily create and destroy VMs for autoscaling and migrate VMs for load balancing or power saving. In this case, the performance of checkpointing and restoration of containers could degrade because the container management systems running in VMs suffer from virtualization overhead. In preparation for host failures, they often transfer checkpoint files to remote servers and restore containers by obtaining the files from the servers. This is similar to container migration, which moves a container from the source VM to the destination VM. It is reported that the primary overhead of container migration was network virtualization and that CPU utilization reached

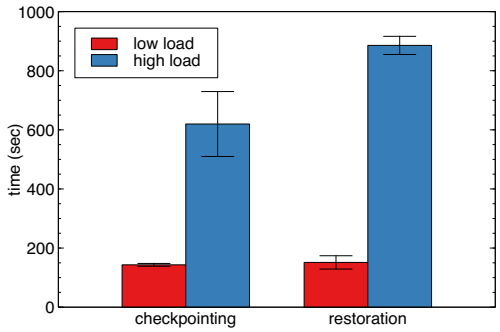


Figure 1: Impact of the VM load on container checkpointing and restoration.

70% in the source VM and 118% in the destination VM [14].

When the load of the VM is high, the performance of container checkpointing and restoration can also degrade because the container management system in the VM is affected by the VM load. To perform periodic checkpointing, it needs to save the state of a container even when the load of the target container or other containers is high in the VM. While it restores a container, the load of the other containers running in the VM can become high. Upon container restoration, container orchestration services can choose a VM whose load is low, but they must choose a VM whose load is high when the restored container needs data locally stored in the VM or needs to collaborate with containers running in the same VM. According to our experiment, the checkpointing and restoration times increased by 4.3x and 5.8x, respectively, under a high load, as shown in Fig. 1. Conversely, container checkpointing and restoration could increase the VM load. This results in the performance degradation of containers running in the VM.

Previous work [8] enables efficient container checkpointing by saving the state of a container outside the VM. Since the state of a container running in the VM is stored in the memory of the VM, the external checkpointing mechanism can analyze the memory and obtain the state using VM introspection [9]. As a result, it is not affected by virtualization overhead and the VM load. However, it is not easy to apply this technique to container restoration. Container restoration is much more complex than container checkpointing because it needs to not only observe the state in memory but also change the system and hardware state.

3. OVrestorer

This paper proposes *OVrestorer* for enabling efficient container restoration outside the VM that runs containers by cooperating with a helper running inside the VM. Fig. 2 shows the system architecture of OVrestorer. Traditionally, the container management system running inside a VM restores containers in the same VM. In contrast, OVrestorer restores a container using an external component called the *OVR core* running outside the VM and an internal

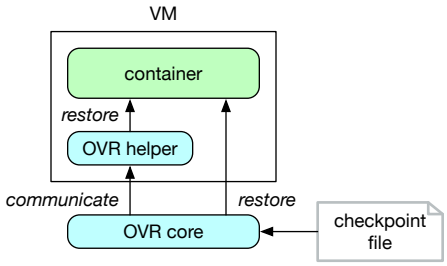


Figure 2: System architecture of OVrestorer.

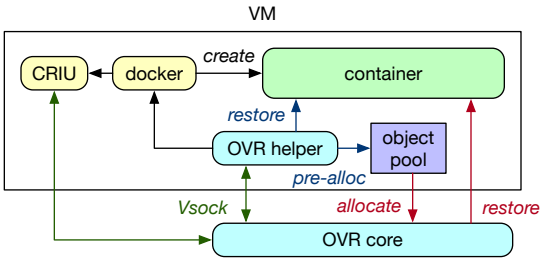


Figure 3: Cooperative container restoration in OVrestorer.

component called the *OVR helper* running inside the VM. When it restores a container, the OVR core overwrites the state of the container in the memory of the VM with the state saved at checkpointing.

Since the OVR core does not run inside the VM, it is not affected by virtualization overhead. It can restore the state of a container without accessing virtual disks or networks. Instead, it can read checkpoint files stored in physical disks or receive them via physical networks. In addition, the OVR core basically does not suffer from the VM load because the VM can use only the resources assigned to it. However, the OVR core and the VM could compete with each other for shared resources, such as disks and networks. Therefore, the OVR core configures resource limits to the VM only during container restoration. For example, the performance of container restoration can degrade when containers in the VM access virtual disks intensively because the OVR core needs to read the saved state of a container from the same disk. To reduce performance degradation, the OVR core limits disk bandwidth so that it can keep the necessary disk bandwidth.

The state of a container mainly consists of the state of the processes running in the container. Therefore, the OVR core analyzes OS data in the memory of the VM and rewrites the state of the processes. Specifically, it searches for various kernel objects and overwrites them with the values saved at checkpointing. To restore the process memory, it identifies the memory areas assigned to the process in the memory of the VM and overwrites the area with the saved memory data. To make it easier to analyze and rewrite OS data, OVrestorer enables the OVR core to use the source code of the guest OS and transparently access the memory of the VM.

It is not easy for only the OVR core to restore the entire state of a container outside the VM. Therefore, the OVR helper running inside the VM offers minimal support for container restoration. For example, it creates a new empty container and empty processes in the container when OVrestorer starts container restoration. Then, the OVR core rewrites the state of the processes. Also, the OVR helper configures hardware timers when OVrestorer restores the interval timers and CPU limits of processes. This enables consistent access to virtual hardware. In exchange, the OVR helper suffers from virtualization overhead and the VM load, but it can suppress that impact by minimizing I/O, such as disk access, and running in the OS kernel. To reduce communication overhead between the OVR core and the OVR helper, OVrestorer uses Vsock [10], which is an efficient communication channel between the host and a VM.

Furthermore, OVrestorer asynchronously allocates and deallocates kernel objects. The OVR core needs kernel memory management to allocate new kernel objects and deallocate unnecessary kernel objects, but it is difficult to perform such memory management outside the VM. If the OVR core communicates with the OVR helper in a fine-grained manner whenever it needs to allocate or deallocate a kernel object, communication overhead degrades restoration performance. To avoid this communication overhead, the OVR helper pre-allocates new kernel objects, while the OVR core uses them when necessary. When the OVR core does not need kernel objects, the OVR helper deallocates them later.

4. Implementation

We implemented OVrestorer for a VM that ran on QEMU-KVM 4.2.0 [11] and ran Linux 5.15 as a guest OS. The OVR helper leverages Docker 27.4.1 [15] and CRIU 3.17.1 [16].

4.1. Cooperative Restoration

Fig. 3 illustrates cooperative container restoration using the OVR core and the OVR helper in OVrestorer.

4.1.1. Flow of Container Restoration. To restore the state of a container, the OVR core first sends a request to the OVR helper using Vsock. When the OVR helper receives the request, it executes the Docker command for container restoration, specifically `docker start --checkpoint`. This command creates a new container and then creates new processes inside it by executing modified CRIU. CRIU is a userspace tool for checkpointing and restoring a process. After CRIU creates a process used for restoration, it communicates with the OVR core using Vsock. Next, the OVR core starts restoring the state of the process and sends requests to the OVR helper when necessary. When the OVR core completes restoring the state of all processes in the container, the `docker` command waiting for CRIU to complete the execution restores the rest of the state of the container. Finally, the OVR helper returns to the OVR core.

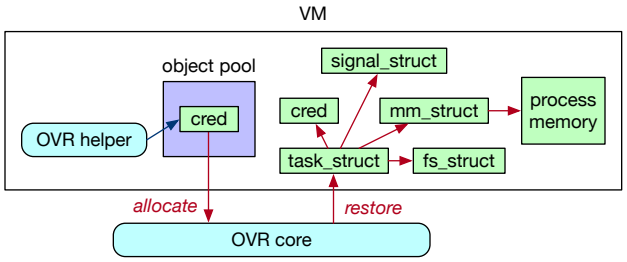


Figure 4: Restoration by the OVR core.

To avoid resource conflicts between the VM and OVrestorer, the OVR core limits the resources of the VM only during container restoration. In the current implementation, it limits the bandwidth of virtual disks using the `virsh` command, specifically `virsh blkdeviotune`. `Virsh` is a command-line tool for VM management. This bandwidth limit can suppress the impact of accessing virtual disks inside the VM on reading a checkpoint file outside the VM. When the OVR core completes restoring the state of the container, it removes the resource limit.

4.1.2. Asynchronous Allocation and Deallocation. To enable OVrestorer to allocate and deallocate kernel objects asynchronously, the OVR helper allocates various types of kernel objects that may be needed during restoration when starting container restoration. It stores the pre-allocated kernel objects in the object pool created in the kernel memory. When the OVR core needs one of the kernel objects, it accesses the object pool by analyzing the memory of the VM and obtains a necessary kernel object. When it does not need a used kernel object, it stores the kernel object in the object pool. When finishing container restoration, the OVR helper deallocates the kernel objects in the object pool. This can eliminate communication overhead between the OVR core and the OVR helper. If the OVR core uses up the kernel objects in the object pool, it communicates with the OVR helper to obtain a necessary kernel object.

4.2. OVR Core

OVrestorer restores a process using a checkpoint file saved by CRIU. Like CRIU, the OVR core reads the specified checkpoint file using protocol buffers [17]. Since its format is defined in the proto files of CRIU, OVrestorer generates code for analyzing a checkpoint file from the same proto files using protocol buffers. Using the generated code, the OVR core obtains saved state from the checkpoint file and restores the state of a process by rewriting kernel objects in the memory of the VM. It searches for the `task_struct` object corresponding to the target process ID and manipulates various kernel objects linked from it, as depicted in Fig. 4.

4.2.1. File Mode Creation Mask. The file mode creation mask of a process can be restored only by the OVR core outside the VM. It is used to determine the default permission of files and directories. Traditionally, a new process created by CRIU in the VM restores its own file mode creation mask by issuing the `umask` system call. In OVrestorer, the OVR core analyzes the checkpoint file and obtains the saved value of the file mode creation mask, such as `022`. Next, it finds the `fs_struct` object from the `task_struct` object. Then, it overwrites the value of its `umask` member with the obtained value.

4.2.2. Process Memory. The contents of the process memory can also be restored only by the OVR core. Since the process memory is usually the biggest among the state of a process, the impacts of virtualization overhead and the VM load are relatively large. Therefore, it is important to restore process memory outside the VM. Traditionally, a new process created in the VM reads the saved memory data from a checkpoint file and overwrites its own memory with the data.

In OVrestorer, the OVR core first finds the `mm_struct` object, which stores information for managing the process memory, from the `task_struct` object. Then, it obtains the virtual address of the page tables used by the process and translates the address into a physical address using the page tables for the kernel. This is because it needs to access the memory of the VM using physical addresses outside the VM. For each memory area in the process, the OVR core translates its virtual address into a physical address and directly writes the memory data stored in the checkpoint file to the memory of the VM.

4.2.3. Group ID. OVrestorer restores the group ID of a process using the asynchronous allocation and deallocation of kernel objects. Traditionally, a new process created in the VM restores the group ID by issuing the `setresgid` system call. In the system call, the kernel stores the specified group ID in the `gid` member of the `cred` object, which is found in the `task_struct` object. However, directly rewriting the member may break kernel integrity because the `cred` object can be referenced by various kernel objects. To rewrite the object atomically, the kernel allocates a new `cred` object, copies the old `cred` object to it, rewrites the `gid` member of the new object, and replaces the old object in the `task_struct` object with the new one.

Therefore, the OVR core rewrites the `gid` member of the `cred` object outside the VM in a similar manner. It first analyzes the checkpoint file and obtains the saved value of the group ID. Next, it obtains a new `cred` object, which is pre-allocated by the OVR helper, from the object pool in the memory of the VM, copies the old `cred` object to it, and rewrites its `gid` member. Then, it replaces the old `cred` object with the new one. Finally, it stores the old `cred` object in the object pool. This object is deallocated by the OVR helper later.

4.2.4. Resource Limits. Restoring the resource limits of a process needs cooperation with the OVR helper, depending on the saved values. Traditionally, a new process created in the VM restores its own resource limits by issuing

the `setrlimit` system call. In OVrestorer, the OVR core analyzes the checkpoint file and obtains the saved values of resource limits. Next, it finds the `signal_struct` object from the `task_struct` object. Then, it overwrites the values of the `rlim` member with the obtained values. This member is an array that maintains the limits for various resources, such as CPU time, the maximum size of physical memory, the number of open files, and so on.

For CPU time, OVrestorer needs to start a timer so that the CPU limit is applied if the value is not infinite. Since it is difficult for the OVR core to access timer hardware outside the VM, the OVR core invokes the OVR helper to configure timer hardware inside the kernel. It should be noted that the CPU limit is often infinite. In this case, only the OVR core can restore resource limits.

4.3. OVR Helper

To minimize virtualization overhead and the VM load, the OVR helper runs in the kernel and restores the state of a process using the kernel functions, instead of system calls.

4.3.1. Interval Timers. OVrestorer restores the interval timers used by a process inside the VM because restoring interval timers needs to access timer hardware. Interval timers are configured by a process and deliver a signal when the specified time has elapsed. Traditionally, a new process in the VM restores its own interval timers by issuing the `setitimer` system call. In OVrestorer, the OVR core analyzes the checkpoint file and obtains the values of interval timers. Next, it sends the values to the OVR helper using Vsock. The OVR helper finds the `signal_struct` object from the `task_struct` object. Then, it overwrites its `real_timer` and `it` members with the received values. At the same time, it configures timer hardware to start a timer. This is similar to the `setitimer` system call, but the difference is that OVrestorer sets the interval timers of the specified process, instead of those of the current process.

4.3.2. Memory Mapping. OVrestorer restores the memory mapping of a process inside the VM because it is difficult to change memory mapping outside the VM. Traditionally, a new process in the VM restores its own memory mapping by issuing the `mmap` system call. Then, the kernel allocates physical memory on demand when the process accesses the memory areas to restore the memory contents. In OVrestorer, the OVR core analyzes the checkpoint file, obtains the virtual addresses and the sizes of the memory areas in the process, and sends that information to the OVR helper using Vsock. Then, the OVR helper allocates the specified virtual memory areas to the process. At this time, it pre-allocates physical memory to the memory areas in advance because it is also difficult to allocate physical memory outside the VM.

4.4. Access to Kernel Objects outside VMs

The OVR core reads and writes kernel objects in the memory of a VM using KVMonitor [12]. As illustrated in Fig. 5, OVrestorer creates a memory file and maps it to the VM as its physical memory. Also, the OVR core maps the file to its address space in a writable manner. When it accesses a kernel object, it communicates with QEMU running the VM and obtains the value of the CR3 register of its virtual CPU, where the physical address of the page tables is stored. Using this address, it walks the page tables in the memory of the VM and translates the virtual address of a kernel object into a physical address. Then, it accesses the target kernel object in the mapped memory file using the obtained physical address.

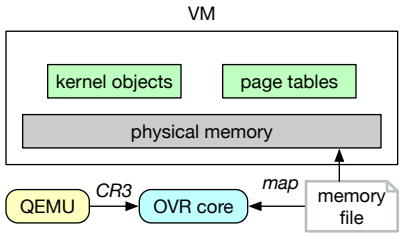


Figure 5: Access to kernel objects using KVMonitor.

TABLE 1: Experimental setup

	host	VM
CPU	Intel Xeon W-2245	4
memory	64 GB	30 GB
disk	2 TB SATA HDD	100 GB
OS	Linux 5.15	Linux 5.15
hypervisor	QEMU-KVM 4.2.0	–

To make it easier to access complex kernel objects in the memory of a VM, we developed the OVR core using the LLView framework [13]. LLView enables developers to write analysis programs using data structures, global variables, inline functions, and macros defined in the header files of the Linux kernel. It compiles programs and converts generated LLVM intermediate representation. Specifically, it embeds code just before the `load` and `store` instructions used for memory access. The code performs address translation so that the compiled programs access the memory of a VM, instead of their local memory.

5. Experiments

We conducted several experiments to confirm that OVrestorer can restore a container outside a VM and to examine restoration performance. For comparison, we measured the performance when we restored a container using the restoration mechanism of Docker 27.4.1 and CRIU 3.17.1 inside the VM. We used a host PC and a VM in Table 1.

5.1. Checking the Restored State

We examined whether OVrestorer could restore the state of a container correctly. We created a container running a cache server, `memcached` [18], inside a VM and stored 1 GB of data in `memcached`. Then, we saved the state of the container using the checkpoint mechanism of Docker inside

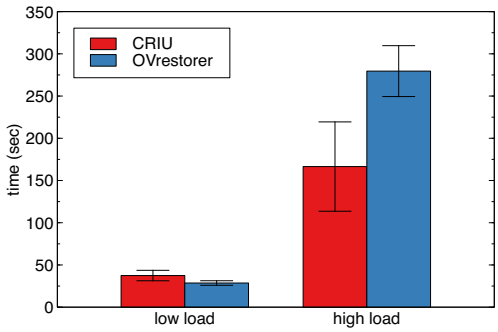
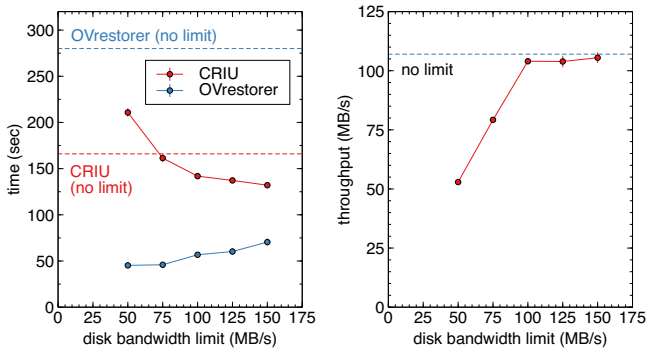


Figure 6: Restoration time of a process.



(a) Restoration time (b) Disk throughput

Figure 7: Impact of the disk bandwidth limit of the VM.

the VM and stopped the container. Next, we transferred the checkpoint file to the outside of the VM and restored the container outside the VM using OVrestorer. After container restoration, we confirmed that we could obtain the data stored before checkpointing from `memcached` in the container.

Since `memcached` did not use interval timers or resource limits, we examined whether OVrestorer could restore their state correctly using simple programs. First, we ran a program that incremented a counter using an interval timer in a container and restored the container from the checkpoint file. After container restoration, we confirmed that the process continued to increment the counter from the value at checkpointing. Next, we ran a program that limited the stack size of a process in a container and restored the container from the checkpoint file. We also confirmed that the stack size of the restored process was limited to the value at checkpointing.

5.2. Performance of Process Restoration

First, we measured only the time needed to restore a process using OVrestorer. We ran a process using 5 GB of memory and saved the state of the process using CRIU. Then, we restored the process using the saved checkpoint file outside the VM. For comparison, we restored it using CRIU inside the VM.

Fig. 6 shows the restoration time when we changed the VM load. To increase the VM load, we ran the `stress-ng` command [19] in the VM and accessed the virtual disk intensively. In this experiment, we did not limit the disk bandwidth of the VM. When the VM load was low, OVrestorer was 25% faster than CRIU. This means that OVrestorer could suppress virtualization overhead. When the VM load was high, OVrestorer was 69% slower than CRIU. This is because the OVR core shared the disk with the VM and was affected by the high load inside the VM.

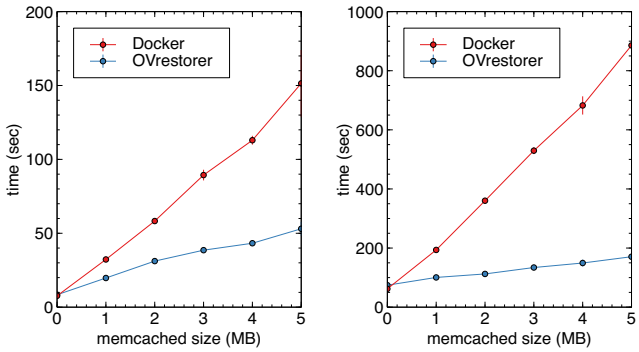
Fig. 7(a) shows the restoration time when we changed the limit of the disk bandwidth of the VM. As a result, when we limited the bandwidth only a bit, the restoration performance was significantly improved in OVrestorer. For example, when we limited the bandwidth to 150 MB/s, OVrestorer was 1.8x faster than CRIU. As we limited the bandwidth more, the performance improvement became more significant. In exchange for this improvement, limiting disk bandwidth can decrease disk access performance in the VM.

Therefore, we measured disk access performance using the `fio` benchmark [20] in the VM. Fig. 7(b) shows the performance when we changed the limit of disk bandwidth. When we limited the bandwidth, disk access performance decreased to the limit as expected until the limit was less than 100 MB/s. Since the maximal throughput was 107 MB/s without the limit, the performance did not decrease when we set the limit to the larger value. Since restoration performance did not change significantly when the limit was larger than 100 MB/s, we set the limit to 150 MB/s in the following experiments.

5.3. Performance of Container Restoration

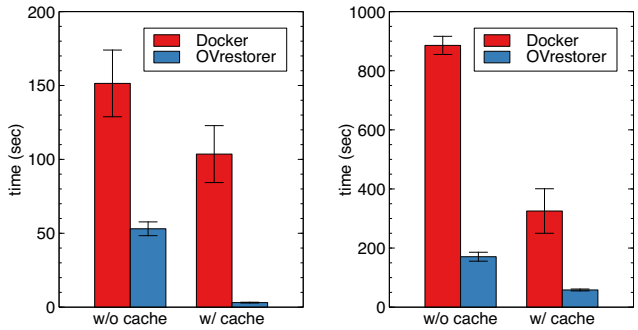
Next, we examined the restoration performance of the entire container running `memcached`. We measured the restoration time of the container. We created a container and ran `memcached` using various sizes of memory in the container. Then, we saved the state of the container using Docker inside the VM. After that, we restored the container outside the VM using OVrestorer. For comparison, we restored the container using Docker inside the VM.

Fig. 8(a) shows the restoration time when the VM load was low. The restoration time was proportional to the memory size of `memcached` in both OVrestorer and Docker. OVrestorer was up to 2.9x faster than Docker. This is because it could reduce virtualization overhead. Since Docker accessed the checkpoint file to restore the container, access to the virtual disk was largely affected by virtualization. In contrast, OVrestorer could directly access the physical disk without virtualization. As the size of the memory used by `memcached` increased, performance improvement became larger. Note that OVrestorer was 9% slower when the memory size of `memcached` was almost zero. This indicates that the restoration performance of the process memory was improved by OVrestorer. When the memory size of



(a) Low load (b) High load

Figure 8: Restoration time of a container.



(a) Low load (b) High load

Figure 9: Restoration time of a container with the page cache.

`memcached` was 5 GB, the OVR core restored 80% of the state outside the VM.

Fig. 8(b) shows the restoration time when the VM load was high. We ran the `stress-ng` command in the VM to increase the load of the virtual disk. As under a low load, the restoration time was proportional to the memory size of `memcached`, but OVrestorer was up to 5.2x faster than Docker. This is because it could suppress the impacts of not only virtualization overhead but also the VM load. However, restoration performance degraded by at least 3.2x even in OVrestorer, compared with under a low load. Although disk bandwidth was limited, the VM load affected both the OVR core and the OVR helper.

5.4. Impact of the File Cache

To exclude the impact of disk access, we examined the restoration performance when the checkpoint file saved by Docker was preserved in the file cache. This situation could occur when the container orchestration service detects an anomaly just after the periodic checkpointing of a container. Fig. 9 shows the restoration time when the memory size of `memcached` was 5 GB. Under a low load, the restoration time decreased by approximately 50 seconds in both OVrestorer and Docker. This is the overhead of disk access. As a result, Docker became 1.5x faster but still took more than 100 seconds. In contrast, OVrestorer became 17x faster and completed container restoration in only 3.1 seconds. In this case, OVrestorer was 33x faster than Docker.

Under a high load, OVrestorer and Docker became 2.9x and 2.7x faster, respectively, by using the file cache. The restoration time decreased by 560 seconds in Docker and 110 seconds in OVrestorer. This large difference came from the significant reduction of the overhead of disk virtualization in Docker. Even in this case, OVrestorer was 5.6x faster than Docker.

5.5. Various VM Loads

We measured the restoration time when we increased various loads using the `stress-ng` command in the VM.

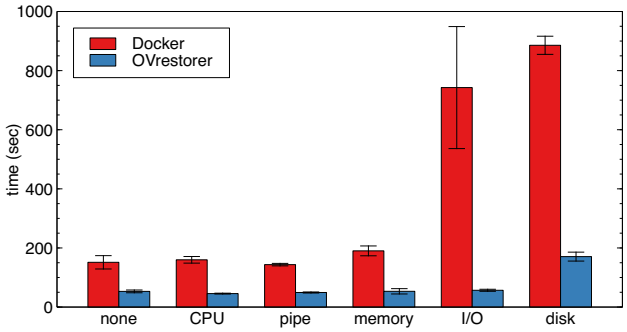


Figure 10: Impact of various VM loads.

Fig. 10 shows the restoration time of a container when the memory size of `memcached` was 5 GB. As a result, OVrestorer was affected only by the disk load because access to a large checkpoint file was a bottleneck in container restoration. In contrast, Docker was largely affected by disk and I/O loads. Since I/O tends to be affected by virtualization overhead, it degraded restoration performance inside the VM.

6. Related Work

Container migration combines the checkpointing and restoration of a container and the transfer of the saved state. Several systems have been proposed to reduce the load of migrating a container running in a VM. OVmigrate [8] can save the state of a process outside the source VM. It obtains the state stored in the memory of the VM using VM introspection [9] and transfers it to the destination VM using the physical network. Therefore, it can save and transfer the state of a process without the impacts of virtualization overhead or the VM load. However, it still needs to save and transfer the state of a container, except for that of the processes, inside the source VM. In addition, it is necessary to restore the state of a container inside the destination VM as traditional.

Portkey [14] optimizes network transfers in container migration to reduce the overhead of network virtualization. After CRIU saves the state of a container inside the source VM, it invokes the hypervisor via the kernel module to bypass the network stack of the guest OS. The hypervisor transfers the state of the container to the destination VM without virtualization overhead. In the destination VM, CRIU receives the state from the hypervisor via the kernel module and restores the container. This can suppress CPU utilization during container migration. Since OVrestorer can restore a container outside the VM, it is not necessary to receive the state inside the VM.

mWarp [21] relocates the process memory between source and destination VMs to avoid memory copies and network transfers. In the source VM, CRIU notifies the hypervisor of the memory information of a process. In the destination VM, CRIU invokes the hypervisor to map the memory of the source VM to its own, which completes transferring the memory of the process. However, mWarp enables container migration between VMs only on the same host. This optimization could be leveraged when OVrestorer is applied to container migration.

To recover the systems from attacks outside VMs, several techniques for eliminating the impact have been proposed. Exterior [22] prepares a secure VM that runs the same OS as a target VM and synchronizes memory updates in the secure VM with the target VM. Using this mechanism, it can recover the system in the target VM by executing commands in the secure VM. For example, it can terminate malicious processes by executing the `kill` command and unload malicious kernel modules by executing the `rmmod` command. However, it needs extra resources to run the secure VM for recovery.

VMMfas [23] recovers the system from attacks by rewriting the memory of a VM and modifying OS data. For example, it can terminate malicious processes by rewriting signal-related OS data and mimicking the delivery of the KILL signals. At this time, it also changes process scheduling by rewriting OS data for the scheduler so that the OS can deliver signals to paused processes. In the current implementation of OVrestorer, the OVR helper continues the restored processes. Using the mechanism of VMMfas, the OVR core could continue the restored processes by delivering the CONT signals.

7. Conclusion

This paper proposed OVrestorer for enabling efficient container restoration outside a VM. OVrestorer runs the OVR core outside the VM and restores a container by rewriting its state in the memory of the VM. This minimizes the impacts of virtualization overhead and the VM load on container restoration. Since it is not easy for the OVR core to restore the entire state, the OVR core cooperates with the OVR helper running inside the VM. According to our experiments, OVrestorer could restore a container more efficiently than the traditional in-VM method even when the VM load was high.

One future direction is to enable more state to be restored outside the VM. In the current implementation, the OVR helper restores some of the state to avoid complexity. Another direction is to examine the performance of restoring multiple containers in one or multiple VMs in parallel.

Acknowledgments

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

References

[1] Amazon Web Services, Inc., "Amazon Elastic Container Service (Amazon ECS)," https://aws.amazon.com/ecs/.
[2] Google, "Google Kubernetes Engine (GKE)," https://cloud.google.com/kubernetes-engine.
[3] Microsoft Corporation, "Azure Kubernetes Service," https://azure.microsoft.com/en-us/products/kubernetes-service.
[4] Amazon Web Services, Inc., "Restart Individual Containers in Amazon ECS Tasks with Container Restart Policies," https://docs.aws.amazon.com/AmazonECS/latest/developerguide/container-restart-policy.html.
[5] Microsoft Corporation, "Run Containerized Tasks with Restart Policies," https://learn.microsoft.com/lb-lu/Azure/container-instances/container-instances-restart-policy.
[6] The Kubernetes Authors, "Liveness, Readiness, and Startup Probes," https://kubernetes.io/docs/concepts/workloads/pods/probes/.
[7] ——, "Kubelet Checkpoint API," https://kubernetes.io/docs/reference/node/kubelet-checkpoint-api/.
[8] Y. Asakura and K. Kourai, "An Effient State-Saving Mechanism for Out-of-band Container Migration," in *Proc. IEEE Int. Conf. Cloud Computing Technology and Science*, 2024, pp. 63–70.
[9] 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.
[10] S. Hajnoczi, "virtio-vsock: Zero-configuration Host/guest Communication," KVM Forum 2015, 2015.
[11] F. Bellard, "QEMU," https://www.qemu.org/.
[12] K. Nakamura and K. Kourai, "Effcient VM Introspection in KVM and Performance Comparison with Xen," in *Proc. IEEE Pacific Rim Int. Symp. Dependable Computing*, 2014, pp. 192–202.
[13] Y. Ozaki, S. Kanamoto, H. Yamamoto, and K. Kourai, "Detecting System Failures with GPUs and LLVM," in *Proc. ACM SIGOPS Asia-Pacific Workshop on Systems*, 2019, pp. 47–53.
[14] C. Prakash, D. Mishra, P. Kulkarni, and U. Bellur, "Portkey: Hypervisor-Assisted Container Migration in Nested Cloud Environments," in *Proc. ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments*, 2022, pp. 3–17.
[15] Docker Inc., "Docker: Accelerated Container Application Development," https://www.docker.com/.
[16] The CRIU Team, "CRIU," https://criu.org/Main_Page.
[17] Google LLC, "Protocol buffers," https://protobuf.dev/.
[18] Dormando, "memcached – a Distributed Memory Object Caching System," https://memcached.org/.
[19] C. I. King, "Stress-ng," https://github.com/ColinIanKing/stress-ng.
[20] J. Axboe, "fio: Flexible I/O Tester," https://github.com/axboe/fio.
[21] P. Sinha, S. Doddamani, H. Lu, and K. Gopalan, "mWarp: Accelerating Intra-Host Live Container Migration via Memory Warping," in *Proc. IEEE INFOCOM*, 2019.
[22] Y. Fu and Z. Lin, "Exterior: Using a Dual-VM Based External Shell for Guest-OS Introspection, Configuration, and Recovery," in *Proc. ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments*, 2013, pp. 97–110.
[23] K. Kimura and K. Kourai, "Xfas: Fault Recovery by Externally Controlling OS Behavior," in *Proc. IEEE/ACM Int. Conf. Utility and Cloud Computing*, 2023.