Low-cost and Fast Failure Recovery Using In-VM Containers in Clouds

Tomonori Morikawa
Department of Creative Informatics
Kyushu Institute of Technology
Fukuoka, Japan
tomonori@ksl.ci.kyutech.ac.jp

Kenichi Kourai
Department of Computer Science and Networks
Kyushu Institute of Technology
Fukuoka, Japan
kourai@ksl.ci.kyutech.ac.jp

***Abstract*—Recently, various services are provided using virtual machines (VMs) in clouds. Therefore, it is necessary to prepare for system failures of VMs, hosts running VMs, and even data centers, e.g., using active/standby clustering. However, a trade-off exists between the maintenance cost for additional VMs and the recovery time in traditional techniques. For example, hot standby can rapidly fail over to the secondary system on a system failure, but the secondary system has to always run the same number of VMs as the primary system. In contrast, cold standby does not need to run VMs until a system failure, but it has to boot VMs on failure recovery. In this paper, we propose *VCRecovery*, which is the system for achieving both low-cost and fast failure recovery. VCRecovery consolidates services using containers inside VMs (*in-VM containers*) in the secondary system. For hot standby, it can reduce the maintenance cost by using only a smaller number of VMs in the secondary system. For cold standby, it can reduce the recovery time by quickly booting in-VM containers. If a VM is overloaded after the recovery, VCRecovery can migrate several in-VM containers to other VMs. To synchronize storage between VMs in the primary system and in-VM containers in the secondary system, it efficiently performs minimum file-based synchronization based on software packages. We have implemented VCRecovery using LXD and Zabbix and examined the performance.**

***Keywords*-failure detection, failure recovery, active/standby, virtual machines, containers**

I. Introduction

Recently, various services are provided using virtual machines (VMs) in clouds. Clouds are well maintained, but they still suffer from system failures. For example, large service disruption has occurred in Amazon S3 and AWS [1], [2]. Therefore, it is necessary to prepare for system failures of VMs, hosts running VMs, and even data centers. To counteract system failures and achieve high availability in clouds, redundancy is often introduced in systems by using clustering. Active/standby clustering uses two systems and fails over from the primary system to the secondary system on a system failure.

However, a trade-off exists between the maintenance cost for additional VMs and the recovery time in traditional techniques. For example, hot standby always runs services in VMs of the secondary system as well as the primary system. Therefore, it can rapidly fail over to the secondary system on a system failure. In exchange for this advantage, the additional maintenance cost for VMs in the secondary system is high. In contrast, cold standby does not run VMs until a system failure but has to boot VMs in the secondary system on failure recovery. The maintenance cost for VMs in the secondary system is zero, whereas the recovery time is long because it takes a long time to boot multiple VMs. Warm standby can take this trade-off, but a lower cost results in a longer recovery time, and vice versa.

This paper proposes *VCRecovery*, which is the system for achieving both low-cost and fast failure recovery. VCRecovery runs a smaller number of VMs in the secondary system and consolidates services in the VMs using the same number of containers as that of VMs used in the primary system. Using such *in-VM containers*, VCRecovery can improve the trade-off between the maintenance cost and the recovery time. For hot standby, the additional maintenance cost is reduced only to that for a smaller number of VMs, while the recovery time is almost the same as when using VMs in the secondary system. For cold standby, the recovery time is reduced by more quickly booting in-VM containers than VMs on failure recovery although the maintenance cost can increase a bit for faster recovery. If VMs are overloaded after recovery, VCRecovery can migrate several containers in the VMs to other VMs and reduce the system load.

We have implemented VCRecovery using LXD [3] inside VMs provided by KVM. LXD is called a container-type hypervisor and supports container migration. To synchronize storage between VMs in the primary system and in-VM containers in the secondary system, VCRecovery performs minimum file-based synchronization based on software packages. It optimizes a list of files excluded from the synchronization and achieves efficient synchronization. For failure detection, VCRecovery obtains the state of VMs without installing monitoring agents in VMs using libvirt-snmp [4] and notifies the Zabbix server [5] of failure occurrence via SNMP. Then, the Zabbix server executes the script for recovery in the secondary system. To switch users from VMs in the primary system to in-VM containers in the secondary system, VCRecovery uses dynamic DNS update [6].

To show the effectiveness of VCRecovery, we conducted several experiments, assuming abnormal termination of VMs, and measured the recovery time. For hot standby, the

recovery time was almost the same as the traditional method using only VMs although in-VM containers suffer from extra virtualization overhead. For cold standby, the recovery time was reduced by 50% because booting in-VM containers was much faster than booting the same number of VMs. In addition, we examined the performance overhead of running containers inside VMs. As a result, it was shown that the performance was largely affected by the storage drivers of LXD, but the overhead was less than 10% for the highest-performance storage driver.

The organization of this paper is as follows. Section II describes countermeasures against system failures in clouds. Section III proposes VCRecovery for achieving both low-cost and fast failure recovery and Section IV explains its implementation. Section V reports the results of our experiments. Section VI describes related work and Section VII concludes this paper.

II. Countermeasures against System Failures

To counteract system failures and achieve high availability in clouds, redundancy is often introduced in systems by using clustering. Active/active clustering runs the same number of VMs in two systems in parallel and both systems always provide the same services to users. Even if a system failure occurs in one system, the other system can seamlessly continue to provide all the services. However, the total system performance is reduced by half of that before a system failure. To prevent this performance degradation, each system has to have sufficient computing capabilities by itself. This raises the cost twice.

On the other hand, active/standby clustering also uses two systems, but only one system provides services in VMs at the same time. A primary system provides services until a system failure occurs. Upon a system failure, the primary system fails over to a secondary system. Then, failure recovery is completed by switching users to VMs in the secondary system. The recovery time is defined as the time after a system failure occurs until the recovery is completed. For the secondary system, the maintenance cost for additional VMs is needed and depends on a system configuration. The system configurations include *hot standby*, *cold standby*, and *warm standby*, and a trade-off exists between the recovery time and the maintenance cost.

Hot standby runs the same number of VMs with services running in the secondary system as in the primary system. This is illustrated in Fig. 1. As a result, it can rapidly fail over to VMs in the secondary system on a system failure in the primary system and reduce the recovery time. However, the maintenance cost increases twice as in active/active clustering because unused VMs for preparing system failures always run in the secondary system and needs to synchronize storage between VMs. To reduce this cost, it is possible to run smaller VMs in the secondary system because the cost is basically proportional to the size of VMs in clouds. This

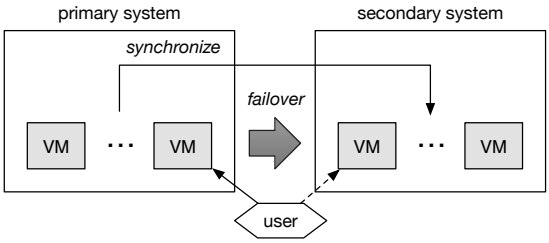


Figure 1: Hot standby.

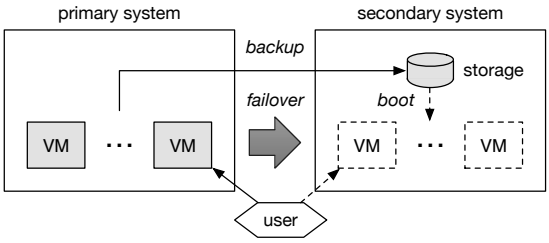


Figure 2: Cold standby.

is a kind of warm standby, as described below. In this case, users need to boot larger VMs and switch to them when a system load cannot be accommodated in the small VMs after recovery. This can result in longer downtime at the VM switch.

In contrast, cold standby periodically saves the backups of VMs in the primary system to remote storage, as illustrated in Fig. 2. It does not run any VMs in the secondary system. When a system failure occurs in the primary system, it boots VMs in the secondary system and restores the system state using the saved backups. As a result, the maintenance cost can be suppressed only to that for the remote storage, which is usually much less expensive than the cost for VMs. However, cold standby cannot fail over to the secondary system until it completes booting the same number of VMs as in the primary system and restoring their state. This results in a longer recovery time.

Warm standby is a system configuration between hot standby and cold standby. Like cold standby, it always runs VMs in the secondary system, but it does not run all the services as in the primary system. All or some of the services are started on failure recovery. Furthermore, the secondary system may run a smaller number of VMs than the primary system. For example, when the primary system runs multiple application servers for load balancing, the secondary system can run only one server. Therefore, the maintenance cost can be reduced less than twice. The recovery time can become longer than hot standby because several services have to be started before failover. As such, warm standby can take a trade-off between the maintenance cost and the recovery time. However, a lower cost results in a longer recovery time, while faster recovery results in a higher cost.

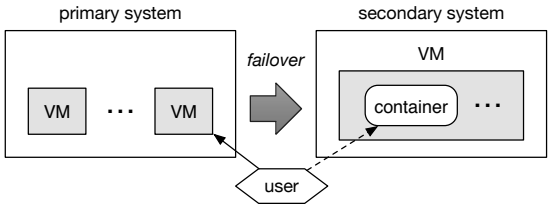


Figure 3: Failure recovery in VCRecovery.

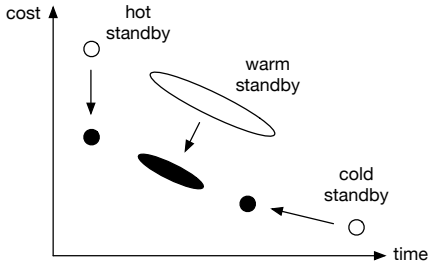


Figure 4: The improvement of the trade-off by VCRecovery.

III. VCRecovery

This paper proposes VCRecovery, which enables both low-cost and fast failure recovery using containers inside VMs (*in-VM containers*) for active/standby clustering. Unlike a VM virtualizing an entire computer, a container is a virtual execution environment provided by the operating system. Fig. 3 shows failure recovery in VCRecovery. VCRecovery runs multiple containers in each VM of the secondary system and runs one service in each container. Since clouds charge fees to VMs, the maintenance cost for failure recovery can be reduced by consolidating services using in-VM containers. In addition, the recovery time can be reduced because containers are more lightweight than VMs.

Using in-VM containers, VCRecovery can improve the trade-off between the maintenance cost and the recovery time, as shown in Fig. 4. First, it can achieve lower-cost hot standby. When several services run using multiple VMs in the primary system, VCRecovery runs those services using several containers in one VM of the secondary system. If a system failure occurs in the VMs of the primary system, VCRecovery rapidly fails over to the containers in the secondary system and seamlessly continues to provide the services. Thus, the additional maintenance cost is reduced only to one VM in the secondary system. The recovery time is almost the same as traditional hot standby because failure recovery is basically not affected by the difference between VMs and in-VM containers.

Second, VCRecovery can achieve more rapidly recoverable cold standby. Upon a system failure in the primary system, VCRecovery boots in-VM containers using backups in the secondary system. Since the boot time of a container is usually much shorter than that of a VM, the recovery time is reduced. Before booting the containers, it is necessary to boot VMs for running the containers although a smaller number of VMs are sufficient. To reduce the boot time of these VMs, VCRecovery can share idle VMs between multiple service providers in the secondary system. Each service providers can rapidly boot containers in these VMs. The maintenance cost for idle VMs can be split between multiple providers. In consideration of security, each idle VM is exclusively used by only one provider at one time.

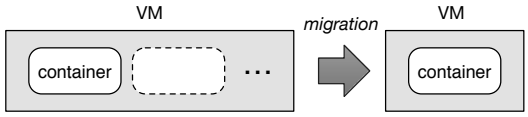


Figure 5: Load balancing with container migration.

Similarly, VCRecovery can achieve lower maintenance cost and faster recovery for warm standby. For services that always run in the secondary system, VCRecovery can consolidate containers running them into a smaller number of VMs. For services to be started on failure recovery, it can quickly boot in-VM containers. This can reduce the recovery time, compared with the traditional warm standby. VCRecovery needs shared VMs for the quick boots, but the increase in cost due to the shared VMs would be less than the cost reduction by the consolidation using in-VM containers.

When the load of VMs in the secondary system becomes high after failure recovery, VCRecovery can migrate several containers to other VMs. In the secondary system, multiple services are consolidated into one VM using containers. Since those services run using multiple VMs in the primary system, the VM in the secondary system can be overloaded by increasing the demand of system resources such as the number of virtual CPUs, the amount of memory, and disk and network bandwidth. In such a case, VCRecovery boots a new VM with appropriate resources and seamlessly migrates containers to it, as illustrated in Fig. 5. Thus, it can reduce the load of VMs without stopping services.

During normal operation, VCRecovery synchronizes data from a VM in the primary system to the corresponding in-VM container in the secondary system. For traditional synchronization between VMs, block-level synchronization is usually used and the two disk images are kept to be identical. However, disk images are different between a VM and a container in general. Therefore, VCRecovery uses file-level synchronization instead of block-level one. In addition, there are many files that are necessary in a VM but not in a container. For example, kernel-related files are unnecessary in a container because a container does not have a dedicated operating system kernel. Such files may be harmless even if they exist, but they increase the size of the disk image of a container and the synchronization overhead. To prevent unnecessary files from being synchronized, VCRecovery

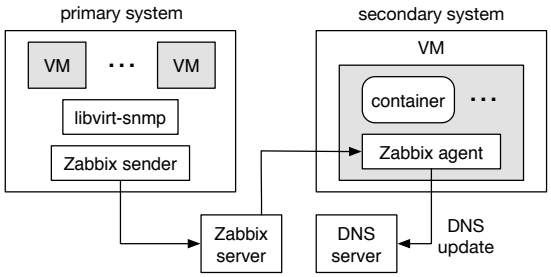


Figure 6: The architecture of VCRecovery.

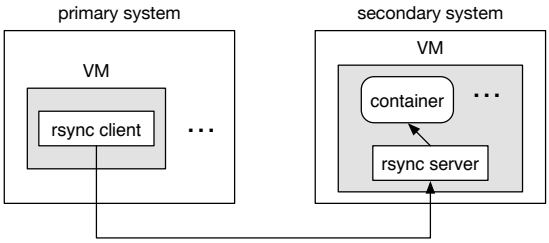


Figure 7: File-based synchronization between a VM and an in-VM container.

performs *package-based synchronization* and excludes the files included in specified software packages, e.g., kernel-related packages.

IV. Implementation

Fig. 6 illustrates the system architecture of VCRecovery. VCRecovery runs VMs using KVM and in-VM containers using LXD [3]. LXD is the container-type hypervisor developed mainly in Ubuntu. There are many container systems such as Docker and OpenVZ, but we adopted LXD because of the support for container migration.

For failure detection and recovery, VCRecovery uses libvirt-snmp [4] and Zabbix [5]. libvirt-snmp is a tool for monitoring the state of VMs using SNMP. When it receives a request from a management system, it returns information on the VM. When a system failure occurs in a VM, libvirt-snmp sends an SNMP trap to a management system. Compared with traditional SNMP, observable information is limited in libvirt-snmp, but it is not necessary to install SNMP agents inside VMs. Zabbix is an integrated monitoring tool and consists of the server, senders, and agents. It enables Zabbix agents to execute actions triggered by state changes.

A. VM-to-container Synchronization

VCRecovery performs file-based real-time synchronization using rsync and lsyncd [7]. An rsync client runs in each VM of the primary system, whereas an rsync server runs in a VM running containers, not in each in-VM container, of the secondary system, as illustrated in Fig. 7. This is because not all the in-VM containers run before a system failure in cold standby and warm standby. lsyncd also runs in each VM of the primary system and detects writes to files using the inotify mechanism in Linux. Upon write detection, lsyncd invokes an rsync client and performs synchronization. If only periodic synchronization is sufficient, VCRecovery does not use lsyncd.

If a simple directory is used as a storage backend of LXD, VCRecovery modifies the user and group IDs of synchronized files. LXD supports several types of storage backends, and this type of storage backend allocates a directory in a host system to a container on top of it as storage. To enable files of containers to coexist with those of a host system in one filesystem, LXD assigns different user and group IDs to files used in containers. Specifically, it adds 100,000 to the original user and group IDs used in containers. In containers, user and group IDs are virtualized and the original ones are provided. Therefore, we have modified the rsync server so as to add the same value to user and group IDs used in a VM of the primary system.

For efficient synchronization between a VM and a container, VCRecovery generates a list of excluded paths, which are contained only in unnecessary software packages. At this time, it optimizes the list so that only upper directories are included as much as possible to reduce the synchronization overhead. This is because it takes a longer time to process a longer list. For example, Fig 8(a) lists the files contained in the apparmor package. The files in the `/etc/apparmor` directory could be replaced with that directory if that directory includes only unnecessary files. However, they could not be replaced with `/etc` because that directory includes necessary files as well, e.g., `/etc/passwd`. As a result, the list is optimized as shown in Fig. 8(b)

The algorithm for this optimization is as follows. First, it extracts all the files contained in specified unnecessary packages and sorts them in dictionary order. This list includes all the directories used for the files. For example, the directories of `/`, `/etc`, and `/etc/apparmor` are included for the file `/etc/apparmor/subdomain.conf`. Then, VCRecovery checks a path in the list one by one. If the path includes the excluded directory, VCRecovery skips that path. Note that the excluded directory is empty at first. If the path is included only in unnecessary packages, VCRecovery adds that path to a list of excluded paths. In addition, if that path is a directory, VCRecovery regards it as the excluded directory in the next checks. If the path is included in necessary packages as well, VCRecovery skips that path.

When packages are installed or uninstalled in VMs of the primary system, VCRecovery reboots in-VM containers in the case of hot standby and, if necessary, warm standby. Ubuntu packages contain scripts executed before and after installation and uninstallation. These scripts are automatically executed in VMs, where packages are installed and uninstalled with the package management system. However, they are not executed in containers because VCRecovery

```
/
/etc
/etc/apparmor
/etc/apparmor/subdomain.conf
/etc/apparmor/parser.conf
/etc/init.d
/etc/init.d/apparmor
/etc/init
/etc/init/apparmor.conf
/etc/apparmor.d
/etc/apparmor.d/force-complain
  :
/lib
/lib/apparmor
/lib/apparmor/functions
  :
```

(a) Files contained in the package

```
/etc/apparmor/
/etc/init.d/apparmor
/etc/init/apparmor.conf
/etc/apparmor.d/
  :
/lib/apparmor/
  :
```

(b) Optimized paths

Figure 8: The excluded paths for the apparmor package.

synchronizes only installed and uninstalled files. This means that installed servers are not started and uninstalled servers are not stopped. To synchronize the running status of servers, VCRecovery periodically obtains the package list by executing the package management command outside an in-VM container. If the package list changes, VCRecovery starts and stops servers by rebooting the container.

B. Failure Detection

VCRecovery obtains the CPU usage of a VM from KVM using libvirt-snmp to detect overload and abnormal stop of the system in the VM. For this purpose, VCRecovery executes the snmpwalk command every 5 seconds and then calculates CPU utilization from current and previous CPU usage. Then, it sends the calculated CPU utilization with the UUID of the VM to the Zabbix server using the zabbix_sender command. If the CPU utilization keeps too high or 0%, it is possible that a system failure occurs in the VM.

To detect a crash of a VM, VCRecovery uses an SNMP trap. libvirt-snmp can issue an SNMP trap when the status of a VM changes. Since that SNMP trap is recorded in a log file, VCRecovery detects the change of the file using the inotifywait command. Then, it obtains the issue time, the UUID, and the status from the trap and sends that information to the Zabbix server. When the status of a VM changes to shutoff, the Zabbix server detects that the VM crashes.

To detect failures of hosts and data centers running VMs, VCRecovery performs alive monitoring between the Zabbix server and agents. The Zabbix server periodically sends heartbeats to a Zabbix agent running each host. When it cannot receive any response from a Zabbix agent, it is possible that a host failure occurs. When all of the Zabbix agents in a data center do not respond, a failure may occur in the entire data center.

C. Failure Recovery

When VCRecovery detects a system failure, it sends an action to a Zabbix agent in the secondary system. The Zabbix agent runs a recovery script as an action using the *remote command* function, which enables users to execute arbitrary commands. For cold standby, VCRecovery boots in-VM containers using backups saved from VMs of the primary system. If shared VMs in the secondary system are not used, VCRecovery also boots new VMs. For hot standby, it is not necessary to boot in-VM containers because containers always run.

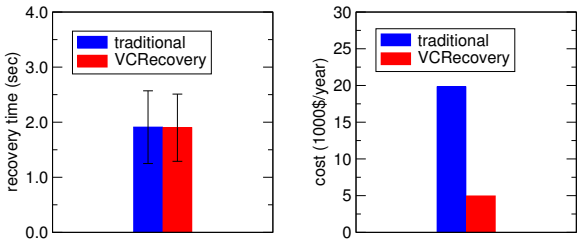
Next, VCRecovery attempts to connect to the network ports used by services in the in-VM containers. After all the connections are established, VCRecovery switches services to the in-VM containers using dynamic DNS update. The nsupdate command updates DNS records so that the DNS server returns the IP addresses of the in-VM containers for host names used by the services.

V. Experiments

We conducted several experiments to show the effectiveness of VCRecovery. We ran four VMs providing Web services in the primary system. Among them, three VMs ran the Apache Web server and one VM ran the MySQL server. For hot standby, we ran four containers providing the same Web services inside one VM in the secondary system. For cold standby, we did not run the four containers until failure recovery. For comparison, we used the traditional system running four VMs in the secondary system. For hosts in the primary and secondary systems, we used two PCs with an Intel Xeon E3-1226 v3 processor, 8 GB of memory, 500 GB of disk, and Gigabit Ethernet. We ran Linux 4.4 and KVM 2.5.0. We used VMs with two virtual CPUs, 2 GB of memory, 50 GB of disk and ran Linux 4.4 and LXD 3.7.

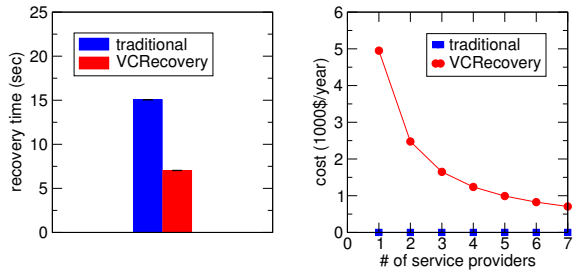
A. Recovery Time vs. Maintenance Cost

First, we compared the recovery time and the maintenance cost for hot standby. We forced VMs in the primary system to terminate and measured the time until Web services were switched to the secondary system. Fig. 9(a) shows the recovery time in VCRecovery and the traditional system. From this result, it was shown that VCRecovery could fail over in almost the same time as the traditional system although the performance of in-VM containers is lower than that of VMs. Fig. 9(b) shows the estimation of the additional



(a) Recovery time (b) Maintenance cost

Figure 9: The comparison for hot standby.



(a) Recovery time (b) Maintenance cost

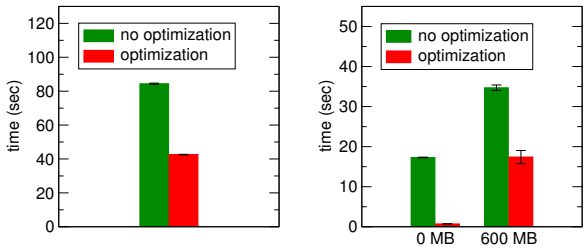
Figure 10: The comparison for cold standby.

maintenance cost. For this estimation, we used t3.2xlarge on-demand instances with Red Hat Enterprise Linux provided in Amazon EC2 and calculated the fee in one year. This result shows that VCRecovery can reduce the cost of about $15,000 per year by using only one VM instead of four VMs as in the traditional system.

Next, we compared the recovery time and the maintenance cost for cold standby. As shown in Fig. 10(a), VCRecovery could reduce the recovery time by 50%. This is because booting four containers in a shared VM took a shorter time than booting four VMs. Fig. 10(b) shows the estimation of the maintenance cost in VCRecovery when we used the same instances above. The additional cost is zero in the traditional system, while it is a fee of one VM in VCRecovery. However, the cost can be reduced as the number of service providers increases because one VM can be shared among multiple providers for recovery. From the result, it is shown that the cost can be suppressed when several providers share a VM.

B. Performance of Synchronization

To examine the impact of our optimization in package-based synchronization between a VM and an in-VM container, we first measured the generation time of a list of excluded paths. Before our optimization, we manually listed all the files contained in packages related to the operating system kernel. The number of those files was approximately 30,000. Fig. 11(a) shows the generation time without and with our optimization. Surprisingly, the time was shorter when the optimization was applied. This is because the



(a) Generation (b) Synchronization

Figure 11: The performance of disk synchronization.

number of generated paths was reduced only to 74 by our optimization. Compared with the execution time of our optimization algorithm, it took a longer time to write a long list of excluded paths to a disk.

Next, we measured the synchronization time using the generated list without and with the optimization. The total size of the target files was 11 GB. Fig. 11(b) shows the synchronization time when the target files were not modified at all and when 600 MB of them were modified. From these results, it was shown that our optimization could constantly reduce the synchronization time by 16.5 seconds. This is because it took a longer time to read the list of excluded paths from a disk and check the list for each target file. This is critical when we use lsyncd, which executes rsync whenever files are modified in a VM of the primary system.

C. Performance of in-VM containers

To examine the performance degradation of using containers inside a VM, we compared the performance of an in-VM container with that of a VM. As storage backends of LXD, we used a simple directory, Logical Volume Manager (LVM), Btrfs, and ZFS. We indicate four types of containers with these storage backends by C.dir, C.lvm, C.btrfs, and C.zfs, respectively.

Fig. 12 shows the results when we ran UnixBench 5.1.3. The score of an in-VM container is normalized for that of a VM. The performance of an in-VM container degraded in most operations. For example, the performance degradation in pipe throughput, context switches, and system calls was 13–17%. The performance of file copies largely depended on a storage backend used for the in-VM container. The performance degradation was only by 10% for C.dir and C.lvm, whereas that was 55–65% for C.btrfs and 85% for C.zfs. When using ZFS, in particular, the VM froze when UnixBench copied a file of 4 KB. As a result, we could not obtain the score of that operation and the total score, which were probably less than that for C.btrfs. From these results, we can conclude that the performance degradation of an in-VM container is 7–8% when we use C.dir or C.lvm.

For further investigation of file I/O in an in-VM container, we measured file access performance using fio 3.1. Fig. 13 shows the throughput of sequential and random read/write.

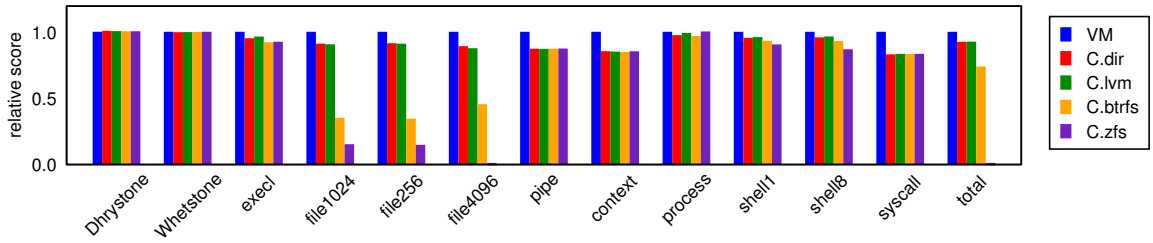
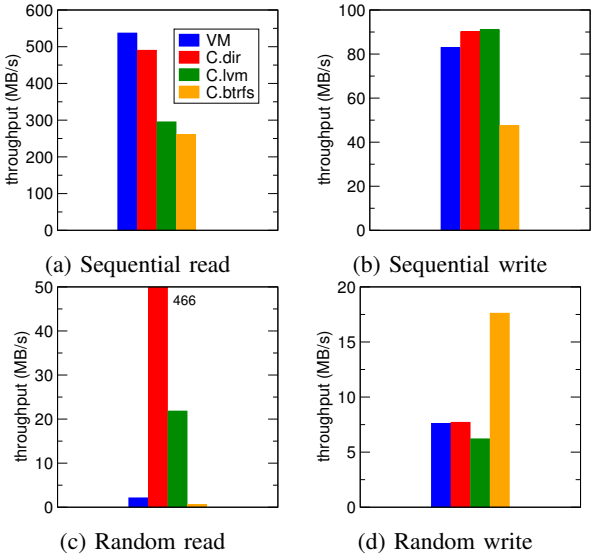


Figure 12: The scores of UnixBench.



(a) Sequential read (b) Sequential write

(c) Random read (d) Random write

Figure 13: The throughput of file access.

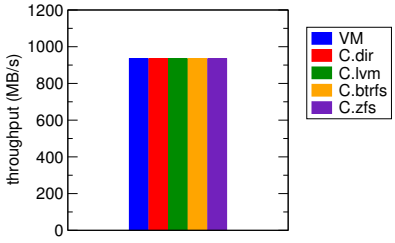


Figure 14: The network throughput.

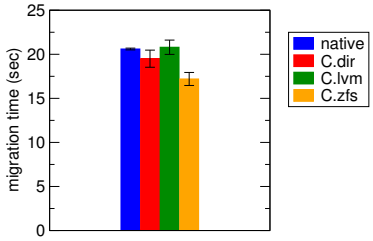


Figure 15: The migration time.

Like UnixBench, we could not obtain data for C.zfs. The throughput of sequential read is equivalent to the result of UnixBench. In contrast, the throughput of sequential write was largely different. The performance for C.dir and C.lvm was slightly higher than that for a VM. The throughput of random read was also largely different from that of sequential read. The performance for C.dir was too high. For random write, the throughput for a VM and C.dir was almost the same but that for C.btrfs was much higher. These reasons are under investigation.

To examine the overhead for network performance, we measured the TCP throughput using iperf 3.1.3. As shown in Fig. 14, the performance did not degrade at all even in an in-VM container.

D. Performance of Container Migration

To examine the migration performance of an in-VM container, we migrated a container between two VMs running different hosts and measured the migration time and the downtime. In this experiment, we used LXD 2.21 because LXD 3.7 could not migrate running containers. We used four storage backends, but we could not migrate a container using Btrfs. For comparison, we migrated a native container using a simple directory without using VMs between hosts. The number of files in the containers was the same, but the storage size of the container was 807–945 MB, depending on storage backends.

Fig. 15 shows the migration time. This result shows that the migration performance of an in-VM container is comparable to that of a native container. Rather, the migration of a container using ZFS was slightly faster. The downtime was 5.4–9.0 seconds because LXD does not support live migration, which migrates containers without stopping them as much as possible. Since CRIU [8] used by LXD can achieve process migration with negligible downtime, LXD could support live migration in the near future.

VI. Related Work

Wood et al. discuss economic benefits of disaster recovery as a cloud service [9]. They focus only on warm standby and compare the cost between a public cloud and a private data center. According to their analyses, much lower cost can be achieved by using a cloud for a multi-tier web application because only one small VM is necessary for synchronization during normal operation. In contrast, cost reduction depends

on synchronization frequency for data warehouse because of the I/O intensive nature.

Several systems have been proposed to consolidate multiple services into a small number of VMs. Picocenter [10] runs a mostly idle service using a container in a VM. When a service is not used and its container becomes inactive, the container is swapped out to storage and swapped in when a request is sent to the service again. Using this mechanism, VCRecovery could rapidly swap in in-VM containers on failure recovery for cold standby, instead of booting them. FlexCapsule [11] runs a service in a lightweight VM inside a VM using nested virtualization. A lightweight VM is achieved by using a library operating system and para-virtualization. FlexCapsule can dynamically optimize the number and the size of VMs in a fine-grained manner and reduce the cost. However, the overhead of nested virtualization is large.

Several systems can synchronize not only storage but also the other VM state such as CPUs and memory. Remus [12] frequently transfers the differences of the CPU state and the memory of VMs in the primary system to the secondary system. It stores network transmission and disk writes in buffers until the periodic synchronization is completed. Kemari [13] reduces the frequency of the synchronization by performing the synchronization only on network transmission and disk writes. COLO [14] delivers request packets to both VMs in primary and secondary systems and waits for the VMs until response packets from them match. Unlike these systems, VCRecovery synchronizes storage between a VM and a container.

There are several studies to reduce the boot time of VMs. Fast parallel VM setup [15] can rapidly boot multiple VMs at the same time by using fine-grained locks and caching XenStore data. LightVM [16] enables even a single VM to be booted rapidly by redesigning Xen's toolstack. Since cold standby needs to boot VMs on failure recovery, these mechanisms can reduce the recovery time. However, these techniques can reduce the boot time of VMs, but the boot time of the operating system is still long.

VII. Conclusion

This paper proposes VCRecovery, which enables both low-cost and fast failure recovery using in-VM containers. VCRecovery can consolidate lightweight containers in one VM and reduce the maintenance cost for hot standby. Since a container can boot more quickly than a VM, VCRecovery can rapidly recover from system failures in cold standby. We have developed VCRecovery using LXD and Zabbix and optimized VM-to-container synchronization based on software packages. According to our experiments, it was shown that VCRecovery could reduce the recovery time and the maintenance cost and that the performance overhead of in-VM containers was acceptable.

One of our future work is to run various services and cause various system failures. Failure recovery can depend on types of services and system failures. Another direction is to apply VCRecovery to multi-cloud environments. Since network latency is larger between clouds, storage synchronization and failure detection are challenging.

Acknowledgment

The research results have been achieved by the "Resilient Edge Cloud Designed Network (19304)," the Commissioned Research of National Institute of Information and Communications Technology (NICT), Japan.

References

[1] Amazon Web Services, Inc., "Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region," https://aws.amazon.com/message/41926/.

[2] ——, "Summary of the AWS Service Event in the Sydney Region," https://aws.amazon.com/jp/message/4372T8/.

[3] Canonical Ltd., "Linux Containers," https://linuxcontainers.org/.

[4] Red Hat, Inc., "Libvirt-snmp," https://wiki.libvirt.org/page/Libvirt-snmp.

[5] Zabbix LLC, "Zabbix :: The Enterprise-Class Open Source Network Monitoring Solution," https://www.zabbix.com/.

[6] B. Wellington, "Secure Domain Name System (DNS) Dynamic Update," IETF, RFC 3007, 2000.

[7] A. Kittenberger, "Lsyncd – Live Syncing (Mirror) Daemon," https://axkibe.github.io/lsyncd/.

[8] The CRIU Team, "CRIU," https://www.criu.org/.

[9] T. Wood, E. Cecchet, K. K. Ramakrishnan, P. Shenoy, J. V. der Merwe, and A. Venkataramani, "Disaster Recovery as a Cloud Service: Economic Benefits & Deployment Challenges," in *Proc. USENIX Conf. Hot Topics in Cloud Computing*, 2010.

[10] L. Zhang, J. Litton, F. Cangialosi, T. Benson, D. Levin, and A. Mislove, "Picocenter: Supporting Long-lived, Mostly-idle Applications in Cloud Environments," in *Proc. European Conf. Computer Systems*, 2016.

[11] K. Kourai and K. Sannomiya, "Seamless and Secure Application Consolidation for Optimizing Instance Deployment in Clouds," in *Proc. Int. Conf. Cloud Computing Technology and Science*, 2016, pp. 318–325.

[12] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High Availability via Asynchronous Virtual Machine Replication," in *Proc. USENIX Symp. Networked Systems Design and Implementation*, 2008.

[13] Y. Tamura, "Kemari: Virtual Machine Synchronization for Fault Tolerance using DomT," Xen Summit Boston 2008, 2008.

[14] Y. Dong, W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li, and H. Guan, "COLO: COarse-grained LOck-stepping Virtual Machines for Non-stop Service," in *Proc. Annual Symp. Cloud Computing*, 2013.

[15] V. Nitu, P. Olivier, A. Tchana, D. Chiba, A. Barbalace, D. Hagimont, and B. Ravindran, "Swift Birth and Quick Death: Enabling Fast Parallel Guest Boot and Destruction in the Xen Hypervisor," in *Proc. ACM SIGPLAN/SIGOPS Int. Conf. on Virtual Execution Environments*, 2017, pp. 1–14.

[16] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My VM is Lighter (and Safer) Than Your Container," in *Proc. Symp. Operating Systems Principles*, 2017, pp. 218–233.