Fine-grained Autoscaling with In-VM Containers and VM Introspection

doi: https://doi.org/10.1109/UCC48980.2020.00034

Fine-grained Autoscaling with In-VM Containers and VM Introspection

Kohei Ueki
Kyushu Institute of Technology
u_kohei@ksl.ci.kyutech.ac.jp

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

***Abstract*—Clouds often provides a mechanism called autoscaling to deal with load increases of services running in virtual machines (VMs). When a VM is overloaded, scale-out is performed and automatically increases the number of VMs. However, when multiple services run in one VM, the entire VM is always scaled out even if only one service is over-utilized. In this case, only an over-utilized service should be scaled out, but it is not easy for clouds to accurately monitor the resource usage of services inside VMs. This paper proposes *Ciel*, which runs each service in a container created inside a VM for *separation of services* and enables fine-grained autoscaling of VMs. Using *VM introspection*, Ciel accurately monitors the resource usage of each in-VM container from the outside of a VM in a non-intrusive manner. If it detects an overloaded in-VM container, it creates a new VM of minimum size and boots only the container that needs to be scaled out in the VM. This can minimize both the cost of the VM and the time taken for scale-out. We have implemented Ciel using Xen and Docker and showed the effectiveness.**

***Index Terms*—virtual machines, containers, autoscaling, scale out, resource monitoring**

1. Introduction

Recently, cloud services such as Amazon AWS and Microsoft Azure are widely used. Infrastructure-as-a-Service (IaaS) clouds provide virtual machines (VMs) to users. The users can construct their own systems from scratch as needed. Most of the IaaS clouds provide several types of VMs of pre-defined sizes. In general, the cost of a VM is proportional to its size, e.g., the number of virtual CPUs, the amount of memory, the size of virtual disks. In IaaS clouds, *scale-out* is performed to deal with the load increase of services running in a VM. This method creates a new VM running the same services and suppresses the load increase of each VM by distributing the load to multiple VMs. IaaS clouds often provide a mechanism called *autoscaling*, which automatically performs the scale-out of VMs. This mechanism monitors the load of VMs and scales out a VM when it detects the load increase of it.

When multiple services run in one VM, traditional autoscaling performs scale-out of the entire VM even if only one service is over-utilized. As a result, the cloud creates a larger VM than necessary, which is more costly for the user. In addition, it takes a longer time to complete scale-out because unnecessary services also need to be started. In such a case, the cloud should scale out only an over-utilized service in a new VM. However, it is not easy for the cloud to accurately know which resources are used by each service from the outside of the VM. Processes and files used by each service are internal information in each VM and are tangled. Therefore, the cloud has to infer the resource usage of each service, but such inference is not accurate.

This paper proposes *Ciel* for fine-grained autoscaling of VMs using *in-VM containers*. Ciel runs each service in a container created inside a VM for *separation of services*. This in-VM container enables clouds to easily identify the resources used by each service. Then, it accurately monitors the resource usage of each service inside a VM from the outside of the VM. This is achieved in a non-intrusive manner using a technique called *VM introspection (VMI)* [1]. If Ciel detects an overloaded container in a VM, it creates a new VM of minimum size only for the container and boots only the container that needs to be scaled out in the VM. This can minimize the cost of the new VM, compared with that for a VM providing not only necessary services but also unnecessary ones. Also, the time taken for scale-out can be reduced by not starting unnecessary services.

We have implemented Ciel in Xen 4.6 [2] and Docker 17.05 [3]. Since Docker uses Linux cgroups for resource management, Ciel analyzes the hierarchy of cgroups in the kernel memory of a VM from the outside of the VM to monitor the resource usage of each container. For example, it can obtain CPU utilization, the amount of consumed memory, and the consumed bandwidths of disks and networks. To transparently translate virtual addresses used in the guest operating system into physical ones for this analysis, we have ported the LLView framework [4]. The modified LLView transforms a program so that references to the kernel memory are automatically redirected to the memory of a VM. Using Ciel, we measured the time taken for scale-out and the resource consumption and showed the effectiveness.

The organization of this paper is as follows. Section 2 describes traditional autoscaling of VMs and its issues. Section 3 proposes Ciel for fine-grained autoscaling of VMs with in-VM containers and Section 4 explains its implementation. Section 5 shows our experimental results using Ciel.

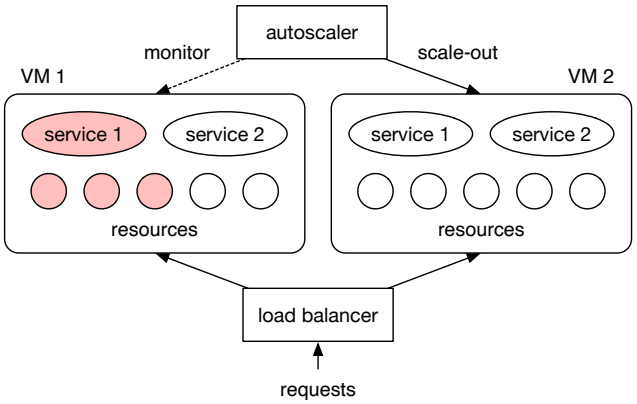


Figure 1: The traditional autoscaling of VMs.

Section 6 describes related work and Section 7 concludes this paper.

2. Autoscaling of VMs

In IaaS clouds, horizontal scaling such as scale-out and scale-in is used to flexibly deal with the load changes of VMs. For scale-out, the user creates new VMs to distribute the loads of existing VMs when some of the VMs are overloaded. For example, consider that the CPU utilization of a web server running in a VM exceeds 90%. The user creates a new VM running the same web server with the same contents and could reduce the CPU utilization of each VM to about 45%. Such scale-out is also useful on high memory pressure and excessive disk and network access in existing VMs. For scale-in, in contrast, the user stops several VMs to reduce the cost for the VMs when VMs are underloaded. For example, when the CPU utilization of a web server is 30% each in two VMs, one VM can be stopped. The CPU utilization of the other VM would be still about 60% even after that.

To automatically perform scale-out and scale-in, most of the clouds provide a mechanism called *autoscaling*. In this mechanism, the autoscaler always monitors the loads of VMs and performs scale-out when it detects that a VM is overloaded, as illustrated in Fig. 1. Then, the load balancer dispatches requests to the VMs including ones newly created for scale-out. In contrast, it performs scale-in when it detects that the average load of VMs is low enough. Since the fee of newly created VMs is charged to the users, the cost increases by scale-out and decreases by scale-in. The detection of the load change is based on the resource usage of a VM. The examples of the indicators are the CPU utilization of a VM, the amounts of consumed memory and disk access, and the number of connected clients. The condition for autoscaling can be configured by users.

However, the traditional autoscaling is not suitable for a VM that runs multiple services. In clouds, one VM often runs only one service, but multiple services can be consolidated into one VM. For an under-utilized service, it is wasteful even to use one VM of minimum size provided by clouds. Service consolidation can use the resources of a VM more efficiently and reduce the cost. When one VM runs multiple services, the entire VM is scaled out even if the loads of only one or several services increase. This means that the autoscaler creates a larger VM than necessary and therefore increases user's cost for the VM. In this case, it is unnecessary to scale out the other services in the VM. In addition to the cost issue, it takes a longer time to complete scale-out. When the newly created VM is booted, it has to start all the services including ones that do not need to be scaled out. The situation gets worse if resource conflicts occur especially for CPUs and disks between necessary and unnecessary services.

When the autoscaler scales out a VM running multiple services, it is desirable to create a minimum VM that includes only necessary services. However, this is troublesome for the user because the user has to prepare various types of VMs including all the possible combinations of services in advance. For example, if three services run in a VM, seven types of VMs are needed. In addition, it is often not easy for the autoscaler to accurately identify which services in a VM are over-utilized from the outside of the VM. Since the autoscaler can monitor only the resource usage of the entire VM, it is difficult to know how many amounts of resources are used by each service. Some of the clouds let the user to install agent software such as the Amazon CloudWatch agent [5] in a VM to monitor the loads of the internal services. Such an intrusive approach should be avoided because it could introduce several issues, e.g., vulnerabilities and instability. Therefore, the traditional autoscaler infers the resource usage on the basis of requests sent to each service, but this is not accurate.

3. Ciel

This paper proposes *Ciel*, which enables fine-grained autoscaling of VMs using containers in a VM, which is called *in-VM containers*. A container is a virtual execution environment provided by the operating system. In-VM containers are widely used [6], but Ciel uses them to achieve *separation of services* for resource monitoring. As shown in Fig. 2, it creates a container per service in a VM and runs only one service in one container. The in-VM container separates processes and resources, e.g., CPUs, memory, files, and networks, used by each service. This self-contained nature of an in-VM container makes it easier to accurately monitor the resource usage per service. Also, Ciel can scale out only necessary services in a VM by selectively booting in-VM containers in a newly created VM. The idea of separation of services was inspired by microservices running containers, but Ciel focuses on "legacy" services running in VMs.

To monitor the resource usage of in-VM containers from the outside of a VM, Ciel uses *VM introspection (VMI)* [1]. VMI is a technique for obtaining the internal states of a VM in a non-intrusive manner without agent software installed in the VM. This is often used for security monitoring of VMs, but Ciel uses it for resource monitoring of in-VM

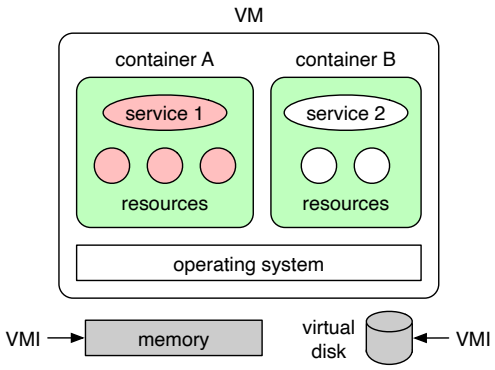


Figure 2: Separation of services with in-VM containers.

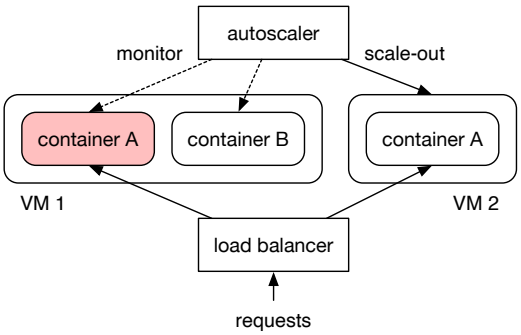


Figure 3: Fine-grained autoscaling of VMs in Ciel.

containers. For example, Ciel obtains the CPU utilization of each container, the amount of memory allocated to each container, and the bandwidths of virtual disk and network consumed by each container. To obtain that information, Ciel analyzes the memory of the guest operating system in a VM using the knowledge of its data structures. At this time, it transparently translates virtual addresses of obtained data into physical ones. In addition, Ciel obtains information on containers, e.g., their names and IDs, from the configuration files in a VM. For this purpose, it analyzes the filesystem used in the virtual disk of the VM.

Ciel performs resource monitoring at both the VM and in-VM container levels. Usually, it monitors the load of an entire VM like the traditional clouds. If it detects an overloaded VM, it inspects the load of each container inside the VM using VMI. One reason of using such two-level monitoring is that the monitoring overhead of in-VM containers is higher than that of a VM. Since VMI needs to analyze the memory of a VM, it is more time-consuming to continue monitoring the resource usage of all the in-VM containers. The other and fundamental reason is that a VM is often overloaded even when the load of any in-VM container is not high. For example, consider that two containers run in a VM. If each in-VM container consumes 45% of the CPU time, the CPU utilization of the VM reaches 90%. Therefore, Ciel identifies one in-VM container that causes the overload of a VM if possible. Otherwise, it selects one or a few in-VM containers for which scale-out is the most effective by considering the changes in resource usage.

Figure 3 illustrates fine-grained autoscaling of VMs in Ciel. Unlike the traditional autoscaling, the autoscaler monitors the resource usage of each container in VMs as well from the outside of the VMs using VMI. When it detects an overloaded in-VM container, it creates a new VM of minimum size only for that container. For example, that VM could equip with a smaller number of virtual CPUs, a smaller amount of memory, and a smaller virtual disk than those of the original VM running multiple containers. Since in-VM containers can be easily added and removed, the minimization of a VM is easier than in the traditional system that directly runs services without containers. The new VM selectively boots only a container that needs to be scaled out and the container starts a necessary service. The other containers running in the original VM are not booted. After the scale-out, the load balancer dispatches requests to multiple in-VM containers and reduces the load of each in-VM container.

As such, Ciel can minimize the cost for a VM that is newly created for scale-out by running in-VM containers only for over-utilized services. In general, that cost is proportional to the amount of resources assigned to a VM in clouds. If the cloud provides various types of pre-defined VMs, Ciel can select a best-fit one so that the amount of any resources does not become insufficient in the near future. If the cloud allows the user to freely adjust the amount of resources assigned to a VM, Ciel can prepare a custom VM. In addition, Ciel can minimize the time taken for scale-out because the new VM does not boot in-VM containers for unnecessary services. At the time of starting services, resource conflicts can be avoided between necessary and unnecessary services.

Readers may think that it is enough to use only containers without VMs. Recently, many clouds natively provide containers instead of VMs [7]. However, there are at least three reasons to use not only containers but also VMs. First, security of VMs is higher than that of containers because a container does not virtualize the operating system. Vulnerabilities of the operating system can affect all the containers in one host. Second, resource isolation of VMs is stricter than that of containers. For example, the resource usage of the operating system shared between containers cannot be separated exactly. Third, container migration is currently pre-mature [8], compared with VM migration. Therefore, it is difficult to dynamically perform load balancing between hosts. From these reasons, Ciel is useful for users who want to run more robust services using VMs.

4. Implementation

Figure 4 illustrates the system architecture of each host in Ciel. Ciel runs VMs using Xen [2] and in-VM containers using Docker [3]. In Xen, the virtual disks of VMs are located in the management VM called Dom0. It is possible to use KVM [9] as virtualization software and LXD [10] as container software, for example. Ciel supports Linux as the

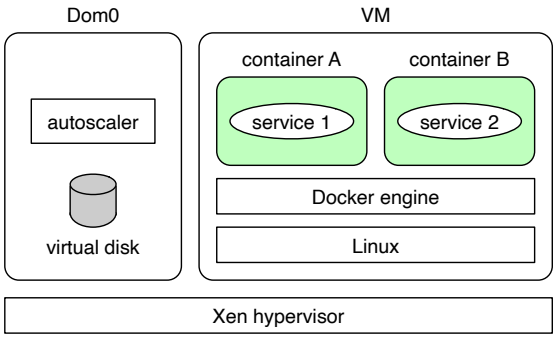


Figure 4: The system architecture of Ciel.

guest operating system running in each VM. The autoscaler runs in Dom0 and communicates with the autoscalers in the other hosts.

Ciel runs a private Docker registry and a load balancer called the Linux virtual server (LVS) [11] in VMs. The private registry is used to provide container images in each of which one service is installed. LVS is used to dispatch requests to multiple in-VM containers that provide the same service. Ciel uses the direct server return (DSR) mode of LVS, which returns a response directly from the in-VM container to which a request is forwarded.

4.1. Monitoring of In-VM Containers

Since Docker controls the resources of containers using control groups (cgroups) in Linux, Ciel analyzes the hierarchy of cgroups in the memory of the guest operating system and obtains the resource usage of each container using VMI. Part of the hierarchy is shown in Fig. 5. First, Ciel searches for the cgroup subsystems corresponding to the monitored resources. A cgroup subsystem represents one type of resources. For example, cgroups provide the CPU accounting (cpuacct) subsystem, the memory subsystem, the block I/O (blkio) subsystem, and the devices subsystem. Next, Ciel searches for the docker group that stores information on Docker containers.

Cgroups manages each Docker container using a UUID, which is a 128-bit number. However, Ciel cannot identify a service running in a container only from a UUID. UUIDs cannot include any information on services and are different even for containers running the same service. The name of a container is more useful, but it cannot be obtained from the guest operating system using VMI. It is managed by the Docker engine running on top of the operating system. To map the UUID of a container to the name of it, Ciel analyzes the configuration file of a container, which is stored in the virtual disk of a VM, using VMI for storage. It first creates the device maps corresponding to the partitions of the virtual disk using the losetup and kpartx commands. Then, it mounts the partitions in a read-only manner to prevent disk corruption. Next, it parses the configuration file named config.v2.json, which is written in a JSON format. This file includes the UUID and name of a container.

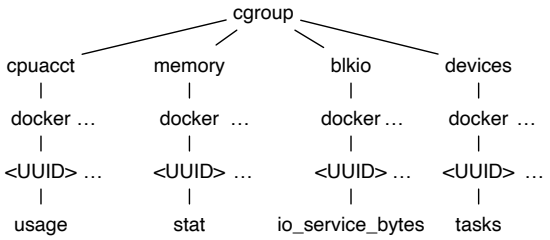


Figure 5: The hierarchy of cgroups.

For each container, Ciel obtains information specific to each subsystem. For the CPU accounting subsystem, Ciel obtains the consumed CPU time from statistical information on CPU usage. In a VM, this information is stored in the cpuacct.usage pseudo file provided by the cgroup filesystem. To obtain the same information from the outside of the VM, Ciel first finds the cpuacct data structure associated with the state of this subsystem in the kernel memory. Then, it accumulates the consumed CPU time for all the CPUs. It obtains the total CPU time every second and calculates the CPU utilization from the increase in CPU time.

For the memory subsystem, Ciel obtains statistical information on memory usage. In a VM, this information is stored in the memory.stat pseudo file. This file contains the resident set size (RSS) and the size of the page cache used by each container. RSS is the total size of anonymous memory and the swap cache. Anonymous memory is the memory used by processes running in a container. The swap cache is the memory that is cached in the memory of the guest operating system after page-out. The page cache is the memory that is allocated in the operating system when processes read and write files. To obtain this information using VMI, Ciel first finds the mem_cgroup structure from the state of this subsystem. Then, it accumulates the numbers of pages used for RSS and the page cache, respectively, for all the CPUs.

For the block I/O subsystem, Ciel obtains the amount of disk access from statistical information on disk usage. In an VM, this information is stored in the blkio.throttle.io_service_bytes pseudo file. To obtain this information with VMI, Ciel first finds the blkcg structure from the state of this subsystem. Then, it accumulates the amount of block I/O for all the CPUs and adds the amounts of block reads and writes for all the block devices. It obtains the total amount of block I/O every second and calculates the consumed disk bandwidth from the increase in disk access.

On the other hand, Ciel obtains the amount of network access from statistical information recorded in the network device of a container, instead of a subsystem of cgroups. Unlike the usage of the other resources, network usage is stored in the net/dev pseudo file provided by the proc filesystem inside a VM. To identify the network device used by a container, Ciel first obtains the task_struct structure for one process in a container from the devices subsystem. In a VM, the list of the processes running in a container is stored in the tasks pseudo file of the cgroup filesystem. Then, Ciel

```
#include <linux/cgroup.h>

void get_cpuacct(void)
{
  struct cgroup *cg, *child;
  struct cpuacct *ca;

  list_for_each_entry(child, &cg->self.children,
                      self.sibling) {
      :
    for_each_present_cpu(i)
      total += cpuacct_cpuusage_read(ca, i);
      :
  }
}
```

Figure 6: Code for obtaining CPU usage.

finds the network device for eth0 used by the process and accumulates the amounts of transmitted and received data, respectively, for all the CPUs. From the increase in network access, it calculates the consumed network bandwidth every second.

4.2. Transparent VMI

To perform resource monitoring using VMI, we have ported the LLView framework [4]. The original LLView is used for a GPU program to obtain system information stored in main memory. Unlike it, the ported LLView enables the autoscaler outside a VM to obtain the internal states of the guest operating system inside a VM. Using this framework, we could easily develop the function of resource monitoring in the autoscaler. Specifically, we included the Linux header files as they are and then used kernel data structures, inline functions, and macros. Also, we reused the source code of cgroups and the proc filesystem in the Linux kernel as much as possible. Fig. 6 shows an example code for obtaining CPU usage. Since LLView cannot transform assembly code, we needed to replace assembly functions with their C versions.

LLView compiles the source code of resource monitoring and generates the intermediate code using LLVM. Then, it transforms the load instructions in the code so that the code accesses the memory of a VM if the target address is of the guest operating system. For this code transformation, LLView uses the Pass framework in LLVM. Whenever LLView finds the load instruction, it inserts the invocation of the function for address translation before that instruction. This function translates the specified virtual address of kernel data into a physical one to access the memory of a VM. It first reads the CR3 register of one virtual CPU of the VM and obtains the physical address of the page directory. Then, it walks the page tables in the memory of the VM and obtains the corresponding physical address. Using the obtained physical address, it maps the memory pages that contain data to be accessed by invoking a hypercall of Xen.

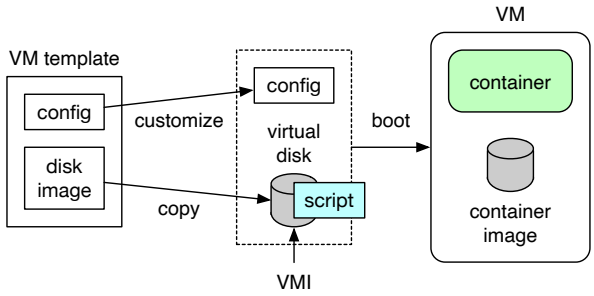


Figure 7: Cloning a VM from a VM template.

4.3. Resource Monitoring of VMs

To monitor the resource usage of the entire VM, Ciel uses the libvirt API [12]. For CPUs, it obtains the total CPU time consumed by all the virtual CPUs of a VM using the virDomainGetInfo function every second. From the increase in CPU time, it calculates the CPU utilization. For memory, it obtains the amount of memory actually used by a VM using the same function. This memory size is different from the maximum one configured to the VM in advance. Since a VM increases and decreases its memory size using the ballooning mechanism [13], the size of the used memory is useful to examine memory pressure inside a VM. For disks, Ciel obtains the amounts of read and written data using the virDomainBlockStats function and calculates the bandwidth every second. Similarly, it obtains the amount of transmitted and received data using the virDomainInterfaceStats function and calculates the bandwidth.

4.4. VM Cloning for Scale-out

When Ciel scales out a service, it clones a VM from a VM template, as illustrated in Fig. 7. A VM template consists of a configuration file of a VM and an image file for a virtual disk. Ciel first copies and parses the configuration file, which is described in the XML format. Then, it modifies a UUID, a MAC address, and the name of a VM. This is necessary because the same identifiers cannot be allowed for different VMs. In addition, Ciel modifies the number of virtual CPUs and the amount of memory assigned to a VM so that the size of a VM becomes minimum for a scaled-out service. Also, it modifies the name of an image file for a virtual disk so that the name becomes unique. Finally, it registers the modified configuration file to the libvirt system used in KVM.

Next, Ciel creates a virtual disk of the VM by copying an image file in the VM template to a file with the name specified in the new configuration file. That image file contains only the minimum system for Docker to run only necessary containers. For example, a lightweight operating system such as Barge [14] can be used. To reduce the size of an image file, the file is created as a sparse file. A sparse file is a special file that contains only blocks in which data is written. Its actual size is much less than the size of the

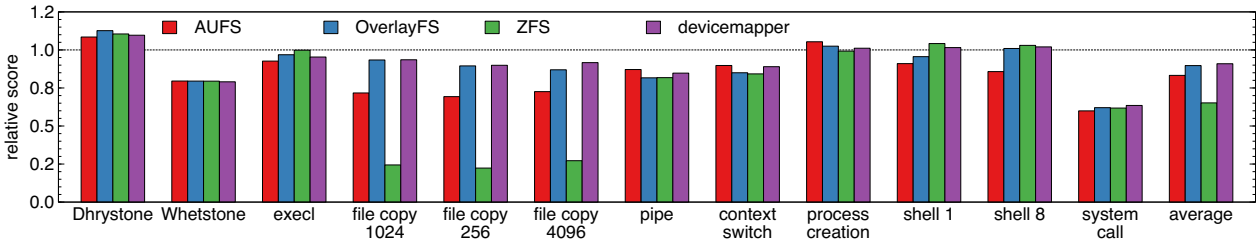


Figure 8: The performance of an in-VM container.

entire virtual disk. When Ciel creates a new virtual disk, it copies only blocks with actual data in an image file. This can also reduce the time taken to create a virtual disk.

Before Ciel boots the cloned VM, it embeds a shell script for booting necessary in-VM containers into the virtual disk of the VM using VMI. It mounts the virtual disk in a writable manner and writes a script file to the disk. This does not corrupt the virtual disk because the VM is still not running. The script is automatically executed at the boot time of the VM. It first sets up the network for Docker using the IP address dynamically allocated with DHCP. Then, it pulls necessary container images from the private Docker registry. In the private registry, container images for the services provided by the original VM services are registered in advance. Then, the script creates containers using the pulled images and boots them with IP addresses dynamically allocated by Ciel. At the same time, Ciel registers the IP addresses of the newly created containers to LVS. When LVS receives a request, it selects one of the registered containers and forwards the request.

Ciel can prepare VM templates in which typical combinations of containers have been installed in advance and clones a VM from an appropriate one. This can reduce the installation time of containers after the boot of a VM although it increases the size of an image file and the time taken to copy the file. This is a trade-off that should be considered by users. Also, Ciel can pool VMs that become unnecessary by scale-in and reuse them by simply booting them. This can reduce the time taken to clone a VM from a VM template as well. In either case, Ciel modifies the configuration of already installed containers in a virtual disk using VMI. It mounts the virtual disk and parses two configuration files, config.v2.json and hostconfig.json, used by Docker. Then, it sets the restart policies so that only necessary containers are automatically booted and the others are not. In addition, it replaces the IP address of each container with a newly allocated one.

5. Experiments

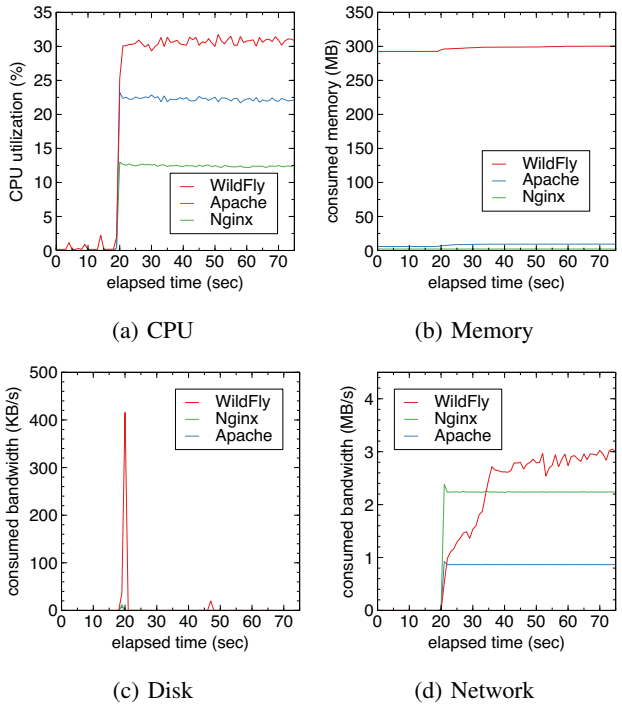
To show the effectiveness of fine-grained autoscaling of VMs in Ciel, we conducted several experiments. We used a PC with an Intel Xeon E3-1225 v5 processor, 12 GB of memory, 1 TB of HDD, and Gigabit Ethernet. We ran Xen 4.6.5 as virtualized software and Linux 4.4 in Dom0. For each VM, we assigned one virtual CPU, 1 GB of memory, and 50 GB of a virtual disk. We ran Linux 4.4 and Docker 17.05.0 in VMs.

5.1. Performance of an in-VM Container

To examine the overhead of running containers in a VM, we ran the UnixBench 5.1.3 benchmark in an in-VM container. As storage drivers of Docker, we used AUFS, OverlayFS, ZFS, and devicemapper in Linux. AUFS and OverlayFS are two different implementations of the union filesystem [15] and OverlayFS is simpler. Using these filesystems, Docker stacks a filesystem for a custom container image on top of a filesystem for a base image. ZFS is a filesystem that supports snapshots and clones. Docker creates a read-only snapshot for a base image and uses a writable clone for a custom image. Unlike the other drivers, devicemapper uses block devices and operates at the block level. Docker creates a device for a base image using thin provisioning and uses its snapshot for a custom image. For comparison, we measured the performance in a VM without a container. We ran UnixBench 10 times and calculated the average.

Figure 8 shows the results of UnixBench. The score of an in-VM container is normalized for that of only a VM. It was shown that the performance of an in-VM container degraded by 9-35% on average. The overhead was small for integer arithmetic (Dhrystone), program execution (execl and shell), and process creation. In contrast, that was not small for system calls, floating-point arithmetic (Whetstone), pipe, and context switching. In particular, the performance degradation of system calls was 40%. This is due to the extra virtualization overhead of a container in addition to that of a VM.

The performance of file copies largely depended on storage drivers. In particular, the overhead was large in ZFS and AUFS. Using ZFS degraded the performance by 75%. This is probably because ZFS follows block pointers more on block reads and needs more I/O for file operations. Using AUFS also degraded the performance by 30%. In addition, it caused performance degradation of 15% even for the concurrent execution of a shell script. When we used OverlayFS and devicemapper, the performance degradation of file copies was 10% on average. Since OverlayFS is more flexible than devicemapper, we used OverlayFS in the following experiments.



(a) CPU (b) Memory

(c) Disk (d) Network

Figure 9: The resource usage monitored with VMI.

As such, in-VM containers introduce extra overhead, but they are widely used commercially [6]. Therefore, we believe that it is acceptable to use in-VM containers for fine-grained autoscaling of VMs.

5.2. Resource Usage of in-VM Containers

To confirm that Ciel could monitor the resource usage of each in-VM container using VMI, we ran three containers for the Apache web server, the Nginx web server, and the WildFly application server in a VM. Then, we sent requests to the three servers at constant rates using the wrk2 benchmark [16] just after the three in-VM containers were booted. Ciel monitored the CPU utilization, the amount of consumed memory, and the consumed disk and network bandwidths of the in-VM containers from the outside of the VM every second. The overhead of VMI was negligible for this monitoring frequency.

Figure 9 shows the monitored resource usage when we started the benchmark at 20 seconds. Ciel could monitor the resource usage of each in-VM container separately. This result was consistent with the resource usage monitored inside the VM. As in Fig. 9(a), Ciel could detect that the WildFly container consumed a larger amount of CPU time. Without VMI, we could detect only that the CPU utilization of this VM was 100% and the VM was overloaded. For memory, Ciel could also monitor the usage of each in-VM container and detect that only the WildFly container consumed much more memory, as shown in Fig. 9(b). The amount of consumed memory slightly increased when the servers started request processing. After that, it gradually increased in the Apache and WildFly containers.

For disk usage in Fig. 9(c), each server accessed a disk only at 20 seconds. This is because the servers read files from disks to process requests and thereafter used the page cache in memory. In particular, the WildFly container consumed a much larger bandwidth. For network usage, Fig. 9(d) shows the total amount of transmitted and received data. The amount of transmitted data was much larger than that of received data. The bandwidths consumed by the Apache and Nginx containers were constant, while the consumption of the WildFly container increased gradually.

TABLE 1: Five methods for preparing a new VM for scale-out.

method	#services	container	image pull
clone A (Ciel)	1	✓	
clone B (Ciel)	1	✓	✓
clone C (traditional)	3		
reuse A (Ciel)	1	✓	
reuse B (traditional)	3		

5.3. Scale-out Time

We examined the time taken to scale out only one of the three services used in Section 5.2. We ran the three containers in a VM and scaled out only the Apache web server. In this experiment, we used five methods for preparing a new VM for scale-out, as shown in Table 1. The first three methods clone a VM to create a new VM. The first method (clone A) is to clone a VM from a VM template including the Apache container. This method can just boot the container to run Apache after the boot of a VM. The second one (clone B) is to clone a VM from a template that does not include the Apache container. In this method, the VM pulls the image of the Apache container from the private registry and creates a new container before the boot of it. The third one (clone C) is a traditional method for cloning a VM from a template that runs three services without in-VM containers. This template is the same as that used for the original VM to be scaled out.

The remaining two methods reuse a VM to prepare a new VM. The fourth method (reuse A) is to reuse a pooled VM including the Apache container. Without copying a large image file for a virtual disk, Ciel can just boot the VM and the container immediately. The last one (reuse B) is a traditional method, which reuses a pooled VM that runs three services without in-VM containers. This VM is the same as the original one. When the VM is booted, it starts all the three services. We did not use a method for reusing a VM including neither any containers nor services because reused VMs should run services before.

Figure 10 shows the clone time and the boot time of a new VM. Compared with the traditional method (clone C and reuse B), Ciel could reduce the boot time significantly. The boot time includes starting a VM, booting the operating system, and starting services. Even when Ciel used a VM that did not include the Apache container (clone B), the

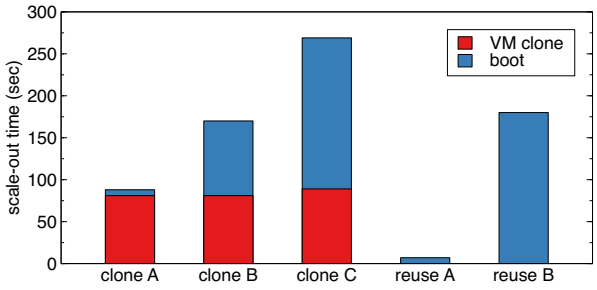
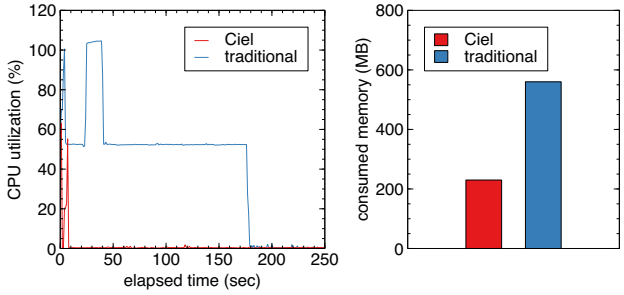


Figure 10: The time for scaling out one of three services.



(a) CPU (b) Memory

Figure 11: The resource usage of a new VM during its boot.

boot time was reduced by 91 seconds. This is because the VM started only one necessary service without two unnecessary services. In particular, the boot time of WildFly was very long in the traditional method. When Ciel used a VM including the Apache container (clone A and reuse A), the boot time was only 7 seconds. This means that it took a long time to pull the image of the Apache container and create a new container from it.

The clone time in Ciel (clone A and B) was 8 seconds shorter than that in the traditional method (clone C). This came from the difference of the number of allocated disk blocks because all the image files in the used templates were provided as sparse files. In Ciel, the size of the image file slightly increased when it included the Apache container, but the clone time did not increase. In the traditional method, two unnecessary services used more disk blocks. As a result, when we cloned a VM from a template including the Apache container (clone A), the time for scale-out was 33% of that for the traditional method. In contrast, when a pooled VM was reused, the clone time was zero. In this case, the time for scale-out in Ciel (reuse A) was only 4% of that for the traditional method (reuse B).

5.4. Resource Usage during VM Booting

We examined the resource usage of a new VM created for scale-out while the VM was being booted. As in Section 5.3, we ran the three services in a VM and scaled out only the Apache web server. Ciel cloned a VM from a VM template including the Apache container (clone A) and booted it. For comparison, we used the traditional method (clone C), which cloned a VM from a template including the three services without containers. We measured the CPU and memory usage of the new VM after we started to boot the VM.

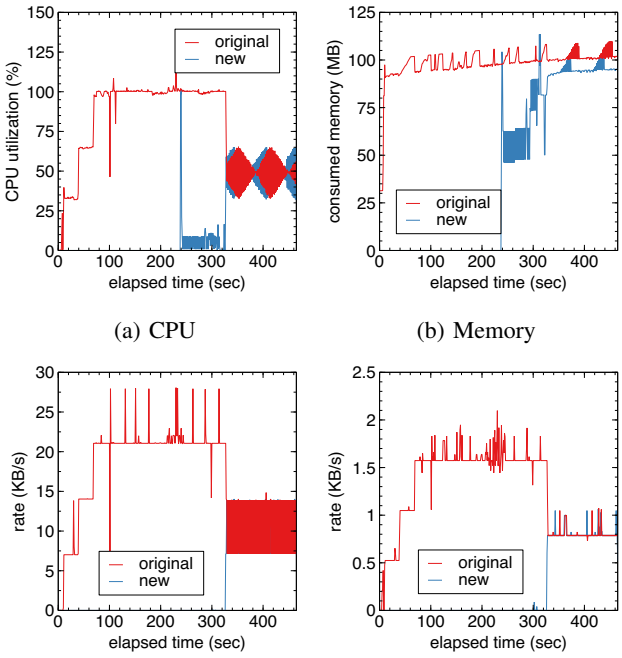
Figure 11(a) shows changes in CPU utilization of the VM. For Ciel, the CPU utilization was less than 63% and became zero after 8 seconds. This is due to booting only the Apache container in the VM. For the traditional method, in contrast, the CPU utilization sometime reached 100% and kept 52% for a long time to boot the three services. In particular, it took much time to start the WildFly application server. The CPU utilization became zero in 180 seconds. If we assigned more than one virtual CPUs to the VM, the boot time could be shorter even in the traditional method by booting the services in parallel. However, this would increase the cost for the VM due to extra virtual CPUs. The two unnecessary services needed CPUs at the boot time although they do not consume CPU time after that.

Figure 11(b) shows the memory usage of the VM after the boot. For Ciel, the amount of consumed memory was reduced by 330 MB, compared with the traditional method. Like the CPU usage, WildFly consumed much memory. This means that Ciel needs only a VM with a smaller amount of memory and can reduce the cost.

5.5. Autoscaling with Ciel

We examined that Ciel could effectively perform autoscaling and load balancing when a service is over-utilized. As in Section 5.2, we ran the three services in a VM. In addition, we ran a heavyweight web application that pixelated uploaded images on Apache, which was developed using the Bottle framework [17]. Then, we sent requests only to that heavyweight web application using the httperf benchmark [18]. The request rate was one request per second for the first 30 seconds. It was increased one by one every 30 seconds up to three requests per second. We configured the autoscale policy so that Ciel scaled out Apache when the CPU utilization of in-VM container exceeded 70%. In this experiment, Ciel cloned a VM from a VM template that did not include containers (clone B) to reveal the overhead of Ciel. We measured the resource usage of the Apache container in the original VM and that in a new VM created for scale-out.

Figure 12(a) shows changes in CPU utilization. When the CPU utilization exceeded 70% at 69 seconds in the original in-VM container, Ciel started scale-out of Apache. It cloned a VM from the template and started to boot the new VM at 150 seconds. Then, it pulled the image of the Apache container, create a new container from that image, and booted that container at 239 seconds. The CPU utilization was 5% on average during the first 85 seconds after the boot of the new container. This was due to the initialization of our heavyweight web application. After the new container



(a) CPU (b) Memory

(c) Network (TX) (d) Network (RX)

Figure 12: The resource usage of in-VM containers on autoscaling.

became ready, the CPU utilization of the original container was reduced from 100% to 49% on average. In contrast, that of the new container increased to 49% on average. As such, the loads of the two containers were balanced equally.

It should be noted that the CPU utilization of the original container kept 100% during scale-out. That duration can be reduced by cloning a VM from a VM template including the Apache container (clone A) or reusing a pooled VM (reuse A). The CPU utilization sometimes exceeded 100%, but this reason is that Ciel could not obtain the consumed CPU time at exact one-second intervals in some reason. When the actual interval was more than one second, Ciel estimated CPU utilization more largely.

Figure 12(b) shows changes in consumed memory. In the original in-VM container, the memory consumption gradually increased as the time elapsed. It was not affected by the load of the service except for the starting time of request processing. After the container in the newly created VM was booted at 239 seconds, the amount of consumed memory largely increased to almost the same as that in the original container. Even after load balancing was completed, the memory consumption of the original container did not decrease. This is because the used web application needed the almost fixed amount of memory regardless of the number of requests to be processed per second.

Figure 12(c) and Fig. 12(d) show changes in rates of network transmission and reception, respectively. In the original in-VM container, these rates increased step by step as the request rate increased. The reason why the transmission rate sometimes changed steeply was that it was too sensitive to calculate the average rate for one second. After the container started in the new VM, the rates decreased to the half on average successfully. Since our web application did not send or receive a large amount of data per second, the amount of increased network bandwidth did not raise performance issues. However, the network usage could be a bottleneck if a service consumes a larger amount of network bandwidth.

6. Related Work

Unlike Ciel for autoscaling of VMs, container orchestration platforms such as Kubernetes [19] provide autoscaling of containers. cAdvisor [20] is used for resource monitoring and is integrated into Kubernetes. It connects to the Docker engine and obtains the resource usage of all the containers using the RESTful API. For example, it can obtain CPU utilization, the amount of consumed memory, and disk and network usage. On the basis of the resource usage, Kubernetes horizontally scales out or vertically scales up a group of containers called a pod. It is possible to monitor the resource usage of in-VM containers by running cAdvisor inside or outside a VM and send it to the autoscaler outside the VM. However, Ciel does not adopt such an approach to prevent an overloaded VM from negatively affecting the performance of cAdvisor or the Docker engine inside it.

FlexCapsule [21] runs a service using a lightweight VM called an app VM inside a VM to enable flexible optimization of VM deployment. For example, it can consolidate under-utilized services into a small number of VMs by migrating app VMs. In contrast, it can seamlessly move over-utilized services to newly created VMs using the migration of app VMs. It runs app VMs inside a VM using nested virtualization [22] and uses a library operating system in an app VM to reduce the overhead. However, the overhead of nested virtualization is still large. Ciel can minimize that overhead by using containers inside a VM.

Several systems use in-VM containers for other purposes. Picocenter [23] uses a container for each service inside a VM to efficiently run mostly idle services in clouds. If a service is not accessed and the corresponding container becomes inactive, it is swapped out to storage. If a request is sent to the service later, the container is swapped in from the storage. Picocenter is similar to Ciel in that it uses one container for one service. However, it uses a container to easily save and restore the entire state of a service. In contrast, Ciel uses a container to exactly monitor the resource usage of each service and scale out only over-utilized services.

VCRecovery [8] runs services using containers inside VMs for low-cost and fast failure recovery. For low-cost warm-standby, it prepares one container for each VM running in the primary system, instead of a VM. Then, it runs multiple containers inside a smaller number of VMs in the secondary system. For fast cold-standby, it boots in-VM containers, instead of VMs, in the secondary system on a system failure. VCRecovery uses one container to run all the services that originally run in a VM, while Ciel uses one

container for each service. VCRecovery performs container migration if a VM running containers is overloaded after failure recovery. This mechanism can be also used in Ciel to decrease the load of a VM.

Docker enables users to run containers inside a container mainly for efficient development. Docker-in-Docker (DinD) [24] runs another Docker engine in a container and allows users to manage their own containers. Since clouds charge only for the parent container, users could reduce the cost by running multiple services in child containers. However, the isolation of the parent container from the host system becomes weaker because the parent container needs to be privileged. In contrast, Docker-outside-of-Docker (DooD) [25] shares the Docker engine in the host system between parent and child containers. The user in a container can manage their own containers, but all the containers are visible to the host system. Clouds would probably charge for such visible child containers as well.

7. Conclusion

This paper proposed Ciel for enabling fine-grained autoscaling of VMs using in-VM containers. Ciel runs each service using a container in a VM for separation of services. It analyzes the data of the guest operating system in the memory of a VM using VMI and exactly monitors the resource usage of in-VM containers from the outside of the VM. Then, it performs scale-out of only over-utilized services with in-VM containers. It can reduce the cost for a newly created VM for scale-out and minimize the time taken for scale-out. According to our experiments, it was shown that Ciel could monitor various resources of in-VM containers and that the scale-out time was dramatically reduced.

One of our future work is to develop sophisticated autoscaling policies considering not only CPU utilization but also various resources and other metrics, e.g., response time. It is also important to estimate the necessary amount of resources for running over-utilized services and create a new VM of minimum size. Another direction is to support fine-grained scale-in of VMs. If a service is under-utilized, we need to reduce the number of VMs and furthermore consolidate containers running in multiple VMs using container migration. In addition, we would like to apply Ciel to commercial clouds although it is not easy because Ciel requires VMI for memory and disks outside target VMs.

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] 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.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *Proc. Symp. Operating Systems Principles*, 2003, pp. 164–177.

[3] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux J.*, vol. 2014, no. 239, 2014.

[4] Y. Ozaki, S. Kanamoto, H. Yamamoto, and K. Kourai, "Detecting system failures with gpus and llvm," in *Proc. Asia-Pacific Workshop on Systems*, 2019, pp. 47–53.

[5] Amazon Web Services, Inc., "Amazon CloudWatch," https://aws.amazon.com/cloudwatch/.

[6] ——, "Amazon Elastic Container Service," https://aws.amazon.com/ecs/.

[7] ——, "Amazon Elastic Kubernetes Service," https://aws.amazon.com/eks/.

[8] T. Morikawa and K. Kourai, "Low-cost and Fast Failure Recovery Using In-VM Containers in Clouds," in *Proc. Int. Conf. Dependable, Autonomic and Secure Computing*, 2019, pp. 572–579.

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

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

[11] W. Zhang, "The Linux Virtual Server Project – Linux Server Cluster for Load Balancing," http://www.linuxvirtualserver.org/.

[12] Red Hat, Inc., "libvirt: The virtualization API," https://libvirt.org/.

[13] C. Waldspurger, "Memory Resource Management in VMware ESX Server," in *Proc. Symp. Operating Systems Design and Implementation*, 2002, pp. 181–194.

[14] A.I., "Yet Another Lightweight Linux Distribution for Docker Containers," https://github.com/bargees/barge-os.

[15] J. Pendry and M. McKusick, "Union Mounts in 4.4BSD-Lite," in *Proc. USENIX 1995 Technical Conf.*, 1995, pp. 25–33.

[16] G. Tene and M. Barker, "A Constant Throughput, Correct Latency Recording Variant of wrk," https://github.com/giltene/wrk2.

[17] M. Hellkamp, "Bottle: Python Web Framework," https://bottlepy.org/.

[18] D. Mosberger and T. Jin, "httperf – A Tool for Measuring Web Server Performance," Hewlett-Packard Company, Tech. Rep., 1998.

[19] Cloud Native Computing Foundation, "Kubernetes: Production-Grade Container Orchestration," https://kubernetes.io/.

[20] Google, Inc., "cAdvisor: Analyzes Resource Usage and Performance Characteristics of Running Containers," https://github.com/google/cadvisor.

[21] 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.

[22] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The Turtles Project: Design and Implementation of Nested Virtualization," in *Proc. Symp. Operating Systems Design and Implementation*, 2010, pp. 423–436.

[23] 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.

[24] J. Petazzoni, "Docker in Docker," https://github.com/jpetazzo/dind.

[25] A. Mouat, "Running Docker in Jenkins (in Docker)," https://blog.container-solutions.com/running-docker-in-jenkins-in-docker.