Seamless and Secure Application Consolidation for Optimizing Instance Deployment in Clouds

Kenichi Kourai
Kyushu Institute of Technology
Fukuoka, Japan
kourai@ci.kyutech.ac.jp

Kouta Sannomiya
Kyushu Institute of Technology
Fukuoka, Japan
kouta@ksl.ci.kyutech.ac.jp

***Abstract*—In Infrastructure-as-a-Service clouds, users can reduce costs by scale-in or -down when running applications are under-utilized. Since these optimizations of instance deployment require at least one minimum instance even for running an under-utilized application, cost reduction is limited. For further optimization, multiple applications can be consolidated into one instance. However, applications have to be stopped temporarily at the consolidation time and isolation between applications becomes weaker after the consolidation. To solve these problems, this paper proposes *FlexCapsule*, which enables seamless and secure application consolidation in existing IaaS clouds. FlexCapsule runs each application in a lightweight virtual machine (VM), called an *app VM*, using a library operating system. An app VM runs inside an instance using nested virtualization. Using VM migration, FlexCapsule can optimize instance deployment with negligible downtime. Thanks to strong isolation provided by app VMs, it guarantees security between consolidated applications. In addition, FlexCapsule provides multi-process support using app VMs such as process fork and process pools. We have implemented FlexCapsule using Xen and OS^v and confirmed its effectiveness.**

I. Introduction

Infrastructure-as-a-Service (IaaS) clouds provide users with instances, which are usually virtual machines (VMs), and users run their applications in instances. Since users can change instance deployment flexibly in IaaS clouds, they can respond to load changes rapidly. For example, users can use minimum instance deployment at start up and increase the number or the resource amount of instances when their applications become over-utilized. In contrast, they can decrease the number or the resource amount to reduce costs when their applications become under-utilized. Thus it is necessary to optimize instance deployment so that used instances are always sufficient but minimum.

However, it is not easy to perform such optimization in current IaaS clouds. If users adjust the number of instances by scale-out and -in, they need at least one instance even for an under-utilized application and cannot further reduce costs. As an optimization for one instance, users can adjust the amount of resources assigned to an instance by scale-up and -down. Unfortunately, most of the existing clouds achieve this optimization by switching instance types offline because they do often not provide the function for dynamically changing resource allocation to an instance. Therefore, users need at least one minimum instance even for a mostly idle application. Although the cost of each instance may be low, the total cost could become high if users run many under-utilized applications.

For further optimization, users can consolidate applications running in multiple instances into one instance. When there are several under-utilized applications, the user can run them in one instance and reduce costs. Later, when some of the applications become over-utilized, the user can de-consolidate them to other instances. However, this application consolidation and de-consolidation cause service downtime when users move applications between instances. This problem can be solved by using process migration [15], but a security issue arises due to consolidating applications. Since multiple applications run in the same instance, isolation among them becomes weaker than traditional instance-level isolation.

In this paper, we propose *FlexCapsule*, which achieves seamless and secure application consolidation for optimizing instance deployment. FlexCapsule runs each application in a lightweight virtual machine (VM), called an *app VM*, using a library operating system (OS). To enable an app VM to flexibly run with appropriate resources in existing IaaS clouds, FlexCapsule runs an app VM inside an instance using nested virtualization [4]. Since it constructs a virtual private network (VPN) for all app VMs across multiple instances, app VMs can be migrated between instances. Thus FlexCapsule can optimize instance deployment without stopping applications enclosed in app VMs. In addition, it guarantees security between applications consolidated into one instance using strong isolation provided by app VMs.

We have implemented FlexCapsule in Xen 4.2.4 [3]. The library OS used in an app VM is based on OS^v 0.21 [10]. We have added migration support to the library OS because the library OS itself has to suspend and resume para-virtual device drivers. As a helper of the library OS, FlexCapsule provides an OS server running inside each instance. For example, when the fork function in the library OS is invoked,

9781-5090-1445-3/16$31.00

the OS server clones the entire app VM. When the listen function is invoked, the OS server registers a rule for port forwarding to the app VM. Combining these mechanisms, the OS server enables app VMs to create a process pool. According to our experiments, it was shown that FlexCapsule could optimize instance deployment according to performance requirements of applications. Migration performance of app VMs was better than the Linux VM, thanks to smaller memory footprints.

This paper is organized as follows. Section II describes issues in optimizing instance deployment in existing IaaS clouds. Section III proposes FlexCapsule for enabling seamless and secure application consolidation using app VMs. Section IV describes its implementation and Section V shows experimental results. Section VI describes related work and Section VII concludes this paper.

II. Optimizing Instance Deployment

In IaaS clouds, the optimization of instance deployment is performed according to resource utilizations of instances. The most popular optimization is *scale-out* and *-in*, which adjust the number of instances. When an application becomes over-utilized, the user can increase the number of instances by scale-out and distribute the load to more instances. In contrast, when an application becomes under-utilized, the user can decrease the number of instances by scale-in to reduce costs. However, if only one instance is deployed for an under-utilized application, the user cannot further reduce the number of instances. For example, consider intra servers that are rarely accessed during weekends or vacations and personal servers and archive servers that are sometimes accessed. When there is almost no request to such a server, the system load becomes almost zero, but one instance is required if the server cannot be stopped. Thus the effectiveness of this optimization is limited when applications are almost not running.

The optimization for one instance is *scale-up* and *-down*, which adjust the amount of resources assigned to each instance. When an application becomes over-utilized, the user can increase the number of virtual CPUs (vCPUs), the performance of vCPUs, and/or the amount of memory of the instance by scale-up. In contrast, when an application becomes under-utilized, the user can decrease the amount of such resources by scale-down to reduce costs. However, most of the existing clouds like Amazon EC2 achieve scale-up and -down by switching instance types offline. Since the user has to choose one from several instance types, cost reduction is limited by the cost of the minimum instance type. In addition, when the user switches his current instance to a new one, he has to stop applications, move their data to the new instance, and restart these applications in the new instance. This duration becomes downtime, for which applications cannot provide services.

For further optimization, users can consolidate multiple applications running in multiple instances into one instance. This is called *application consolidation*. For example, consider a multi-tier application that consists of multiple applications such as a Web server, an application server, and a database. When these applications are running across multiple instances and all of them are under-utilized, the user can run these applications in one instance to reduce costs. Later, when the instance becomes over-utilized, the user can de-consolidate these applications to multiple instances again to perform load balancing. For further cost reduction, even different users could consolidate their applications into one instance. Like scale-up and -down, however, this also causes downtime when the user moves applications between instances. In addition, a security issue arises due to application consolidation. Since multiple applications run in the same instance, isolation among them becomes weaker than when using one instance per application.

To reduce downtime during application consolidation and de-consolidation and scale-up and -down, process migration can be used. For example, Zap [15] provides a thin virtualization layer between processes and the OS and runs a group of processes in a container called a pod. Using pods, Zap enables most of the process state to be maintained on process migration. However, isolation between pods is not strong because a pod is protected only by namespaces provided by the OS. For stronger isolation, DrawBridge [16] executes each application in a picoprocess, which is a process that runs a library OS. A picoprocess can restrict the interface between an application and the host OS more strictly. In its variant, Graphene [18], it is reported that most of the attacks against vulnerabilities of the system call interface can be prevented. However, isolation between picoprocesses is not sufficient because the host OS has a large number of other kinds of vulnerabilities.

III. FlexCapsule

This paper proposes *FlexCapsule*, which enables seamless and secure application consolidation for optimizing instance deployment in IaaS clouds. FlexCapsule runs each application in a lightweight VM, called an *app VM*, inside an existing instance. Using the technology of VM migration, FlexCapsule can move applications between instances with negligible downtime at the optimization time of instance deployment. Thanks to strong isolation between app VMs, FlexCapsule guarantees security between applications consolidated into one instance.

A. System Architecture

Fig. 1 illustrates the system architecture of FlexCapsule. Using nested virtualization [4], FlexCapsule runs the hypervisor inside each instance, which is usually a VM. It runs app VMs on top of the hypervisor, which is less vulnerable than the OS. Each app VM runs only one application process and

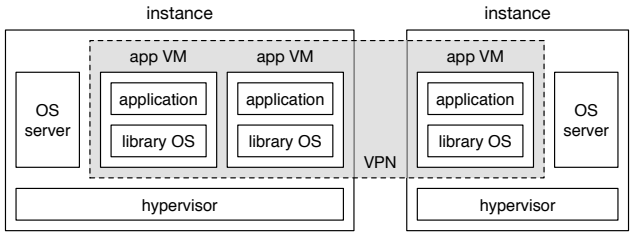


Figure 1. The system architecture of FlexCapsule.

a library OS, which is linked to the application to provide functions of the OS without any overhead of protection mechanisms. FlexCapsule also runs an *OS server* in each instance and provides functions that cannot be achieved only by the library OS inside app VMs. Our approach of using nested virtualization is feasible in terms of performance because it is reported that the overhead is 6-8% [4] for common workloads.

Since FlexCapsule assumes that public IP addresses are assigned only to instances, it assigns private IP addresses to app VMs. This reduces the cost for using public IP addresses in clouds. To provide services of app VMs to the outside, FlexCapsule uses network address port translation (NAPT). Thanks to NAPT, different app VMs can use the same public IP address. The public IP address that each app VM uses is determined at creation time and does not change. Also, it constructs a network with the same segment across multiple instances using a site-to-site virtual private network (VPN). This VPN enables app VMs to continue to use the same public and private IP addresses even after they are migrated to other instances. Each packet is first delivered to the instance with the specified public IP address. Then it is automatically forwarded to an appropriate instance running the target app VM by the VPN.

B. Optimization Using App VMs

When performing application consolidation, FlexCapsule migrates under-utilized app VMs to one instance, as illustrated in Fig. 2(a). As a result, if the source instances have no app VM, FlexCapsule stops them and re-assigns their public IP addresses to the destination instance, e.g., using Elastic IP addresses in Amazon EC2. Thus migrated app VMs can be reached using the same IP addresses before application consolidation. In contrast, when performing application de-consolidation, FlexCapsule deploys new instances and migrates over-utilized app VMs to those instances. Before the migration, FlexCapsule re-assigns one of the public IP addresses assigned to the source instance to the destination ones.

To scale an application up and down, FlexCapsule deploys a new instance of appropriate type and migrates app VMs in the original instance to the new one (Fig. 2(b)). Then it stops the original instance and re-assigns that public IP

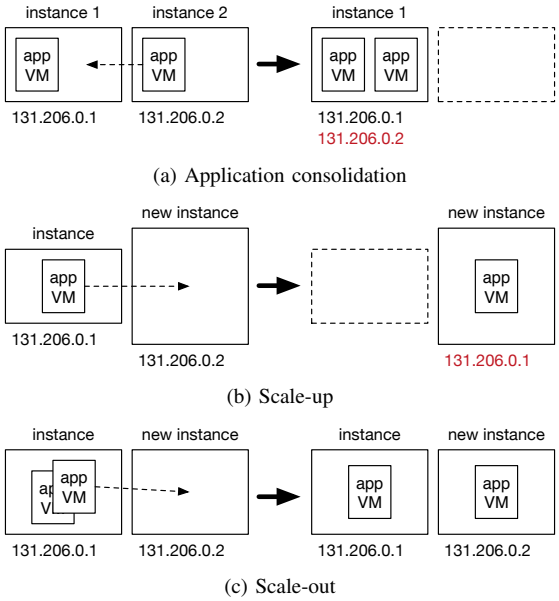


Figure 2. The optimization of instance deployment using the migration of app VMs.

address to the new one. For scaling an application out, on the other hand, FlexCapsule deploys new instances, clones app VMs inside the original instances, and migrates them to the new ones (Fig. 2(c)). At this time, FlexCapsule assigns new private IP addresses to the cloned app VMs but allows them to continue to use the original public IP addresses using NAPT. When scaling an application in, FlexCapsule simply stops several instances.

C. FlexCapsule OS

The FlexCapsule OS is a library OS running in an app VM. A library OS provides functions of the OS to applications as a library. Since only necessary functions are linked to an application at compile time, a library OS can reduce the memory footprint of an app VM. Therefore running an app VM per application does not require extra memory so much, compared with using a general-purpose OS. The small memory footprint of an app VM enables faster VM migration by transferring only a smaller amount of memory.

The FlexCapsule OS reduces the overhead of extra virtualization due to app VMs by using para-virtualization. Full virtualization in nested virtualization poses relatively large overhead because completely double virtualization is necessary. Since a para-virtualized OS simplifies virtualization by cooperating with the hypervisor, the performance of app VMs can be improved. In compensation for this performance gain, a para-virtualized OS needs to support VM migration by itself to disconnect from and reconnect to the tightly coupled hypervisor. Specifically, the FlexCapsule OS enables itself to be suspended and resumed.

D. FlexCapsule Server

The FlexCapsule server is an OS server running in each instance. It provides functions related to multi-process, which cannot be supported only by the FlexCapsule OS. Since FlexCapsule runs only one application process in an app VM, each app VM cannot achieve functions across multiple processes. The FlexCapsule server is used for cooperation between app VMs. For example, when an application in an app VM invokes the fork function, the FlexCapsule server clones the entire app VM. At this time, the FlexCapsule OS communicates with the FlexCapsule server and then the FlexCapsule server creates a child app VM from the parent app VM. The public IP address that the child app VM uses is the same as that used by the parent app VM. It returns an identifier of the child VM to the parent VM and zero to the child VM, as in the original fork. For inter-process communication, the FlexCapsule server mediates messages from one app VM to another one.

The FlexCapsule server also manages NAPT rules to forward packets to app VMs. Since an app VM communicates with the outside using NAPT, the FlexCapsule server registers a NAPT rule when an application invokes the listen function. For example, consider a Web server listening to TCP port 80 in an app VM. The FlexCapsule server registers a NAPT rule that translates a pair of the public IP address used by the app VM and port 80 into a pair of the private IP address of the app VM and port 80. For load balancing, the FlexCapsule server supports process pooling, which is a technique for preparing multiple processes that wait for the same port using fork. The FlexCapsule server configures NAPT rules so that packets are delivered to one of the app VMs in a process pool.

IV. Implementation

We have implemented FlexCapsule in Xen 4.2.4 [3]. DomU is an instance provided by a cloud and runs user's virtualized system using nested virtualization. In the user's virtualized system, DomU is used as an app VM and the FlexCapsule server runs in Dom0. We have implemented the FlexCapsule OS based on OS^v 0.21 [10], which is the OS optimized for virtualized systems. OS^v is fully virtualized but uses para-virtual (PV) device drivers to reduce virtualization overhead. OS^v can run many existing applications with no or slight modification. Since OS^v can run the Java VM, it also supports most of Java applications. In addition, OS^v can run custom applications more efficiently.

A. Migration of App VMs

Since the FlexCapsule OS uses the para-virtualization technology, it needs migration support at the OS level to migrate an app VM. When VM migration is performed, the FlexCapsule server writes a request for power management to the control/shutdown node in XenStore. XenStore is storage for sharing information between VMs and Dom0. To monitor the node, the FlexCapsule OS starts a dedicated thread at boot time and registers the shutdown handler as a callback function. When the node is changed, XenStore sends an event to the corresponding app VM and the FlexCapsule OS invokes the shutdown handler.

The shutdown handler first suspends PV devices. A PV device consists of a front-end driver in an app VM and a back-end driver in Dom0. A front-end driver communicates with a back-end driver using event channels, which are established at device initialization. Since it cannot use the established event channels after VM migration, the shutdown handler disconnects them. Next, the shutdown handler invokes the suspend hypercall, which returns when the app VM is resumed at the destination instance. Then the shutdown handler resumes PV devices to re-establish new event channels with a back-end driver in Dom0 at destination instance.

In the current implementation, the state of the FlexCapsule server does not need to be migrated together with an app VM. For example, the NAPT rules for the app VM continue to be applied at the source instance, as explained in Section IV-C.

B. Fork of App VMs

When an application invokes the fork function, the FlexCapsule OS sends a fork request to the FlexCapsule server via XenStore. The FlexCapsule server suspends a parent app VM and creates a child app VM. It configures a newly allocated IP address of the child app VM and registers new NAPT rules based on those for the parent app VM. Then it copies the states of the parent app VM to the child app VM. In addition, the FlexCapsule server makes the parent and child app VMs share the disk of the parent app VM in a copy-on-write manner. Finally, it resumes the parent app VM.

The implementation of VM fork is similar to SnowFlock [11], but there are two differences. First, FlexCapsule supports cloning of VMs for full virtualization. VM states to be copied are much different from those in para-virtualization. Second, FlexCapsule eagerly copies the entire memory at fork time, whereas SnowFlock copies it on demand. This is because the time needed for configuring the extended page table (EPT) for copy-on-write was similar to that for memory copies in our experiment.

1) Duplicating VM States: The FlexCapsule server issues a newly created hypercall to duplicate VM states. First, the hypercall copies the memory contents of the parent app VM to the child app VM. For each page of the parent app VM, it allocates a new page for the child app VM and registers the mapping from its host-physical page frame number to the guest-physical one. Next, the hypercall copies the CPU states of the parent app VM to the child app VM. It copies the time stamp counter, PAE, the TSS in the virtual 8086 mode, the identity-map page directory, and the location of

ACPI control blocks. Also, it copies the contents of the I/O rings used for memory events and sets the page frame numbers used for console, xenstore, ioreq, and bufioreq to the child app VM. Then it saves the HVM context of the parent app VM and loads it to the child app VM. Finally, the FlexCapsule server saves the device states in qemu-dm for the parent app VM and restores them to qemu-dm for the child app VM.

2) Sharing a Disk: To share the disk of the parent app VM with the child app VM in a copy-on-write manner, the FlexCapsule server creates two copy-on-write disks for these app VMs, respectively. A copy-on-write disk is a disk that stores only updates to the base disk, which is a disk used by the parent app VM before fork. In qcow2 used by qemu-dm, file writes to a copy-on-write disk are performed to the disk, whereas file reads are performed from the disk if files exist; otherwise, they are done from the base disk. To enable dynamically changing a disk of a running app VM, the FlexCapsule server indirectly attaches a disk to an app VM via a network block device (NBD) [19].

For the child app VM, the FlexCapsule server connects one copy-on-write disk to an unused NBD device and then attaches the device to the child app VM. For the parent app VM, the FlexCapsule server first disconnects the base disk from the NBD device already attached. Then it reconnects the other copy-on-write disk to the original NBD device. As such, the parent app VM can use copy-on-write disks seamlessly. If the NBD is not used for indirect disk attachment, this is difficult to achieve because a local disk directly attached to a VM cannot be detached as long as the VM is not stopped.

C. Networking

Fig. 3 illustrates networking in FlexCapsule. When an application invokes the listen function to wait for new network connections, the FlexCapsule OS obtains a listening port number from the specified socket and sends it to the FlexCapsule server via XenStore. Then the FlexCapsule server adds a new NAPT rule to iptables. The added rule forwards packets sent to the public IP address and port number used by the app VM to the app VM. We used the libiptc [2] library for manipulating netfilter. When an application invokes the close function for a socket, the FlexCapsule OS sends a listening port obtained from the socket to the FlexCapsule server via XenStore. Then the FlexCapsule server deletes the NAPT rule corresponding to the port.

To achieve load balancing with a process pool for app VMs, the FlexCapsule server uses the nth mode of the statistic module for iptables. The nth mode is used for simple stateful load balancing in a round-robin fashion. When an application invokes the fork function or the FlexCapsule server clones an app VM for scale-out, the FlexCapsule server examines whether the app VM is listening to network ports. If there are such ports, the FlexCapsule server translates the corresponding NAPT rules into rules for the nth mode and adds new rules for a child app VM. Using the nth mode, packets are delivered to one of the app VMs included in the same process pool. Note that all the packets in one connection are delivered to the same app VM.

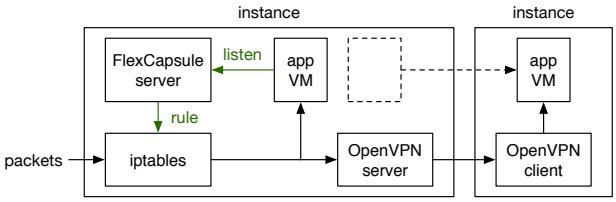


Figure 3. Networking in FlexCapsule.

The FlexCapsule server constructs one VPN that connects all app VMs inside user's instances using Ethernet bridging of OpenVPN 2.3.2 [14]. One instance runs the OpenVPN server, whereas the others run the OpenVPN clients. Thanks to the VPN, even after app VMs are migrated to other instances, the NAPT rules in the source instance are still applied. If the source instance has no app VM and is therefore stopped, the FlexCapsule server transfers the NAPT rules to the destination instance.

V. Experiments

We conducted experiments to confirm the effectiveness of FlexCapsule. We used two PCs with an Intel Xeon E3-1290v2 processor and 8 GB of memory. We ran Xen 4.2.4 for the hypervisor and Linux 3.13.0 in Dom0. We created several DomUs as instances in a cloud and assigned one to four vCPUs and 2 GB of memory. Inside these instances, we ran several app VMs, each of which was assigned one vCPU and 4 to 256 MB of memory.

A. Application Consolidation

To show the effectiveness of application consolidation using app VMs, we measured changes in application performance before and after de-consolidation. We used three app VMs, which ran lighttpd 1.4.35 [12], memcached 1.4.21 [7], and Redis 3.0.1 [17], respectively. For each server, we measured the throughput using httperf 0.9.0, memaslap, and redis-benchmark. When consolidating these three app VMs, we ran them inside one instance with one vCPU. When de-consolidating them, we used three instances, each of which has one vCPU.

Fig. 4 shows relative performance based on the throughput of each server when we consolidated the three app VMs. After de-consolidation, the application performance improved by a factor of 1.9 to 2.7. This means that application de-consolidation can improve the performance of app VMs. In other words, application consolidation is useful if each app VM is not used so much.

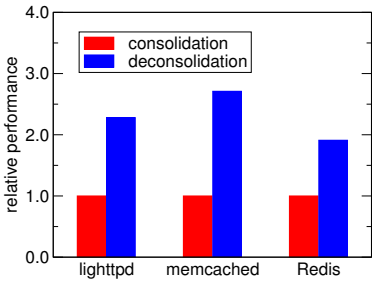
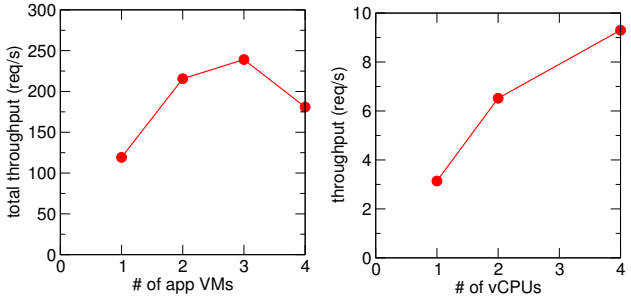


Figure 4. The performance improvement by application de-consolidation.



(a) scale-out (b) scale-up

Figure 5. The performance improvement by scale-out and scale-up.

B. Scale-out and Scale-up

First, we investigated whether app VM-level scale-out was effective. We increased the number of app VMs running inside one instance with four vCPUs and measured the total performance of all the app VMs. We ran lighttpd and measured the throughput using httperf when we sent requests for 1 KB files. Fig. 5(a) shows the total throughput. When we increased the number of app VMs, the total throughput was increased until three app VMs. This is because these three app VMs and one management VM inside the instance used up four vCPUs.

Second, we examined the effectiveness of instance-level scale-out with app VMs. We ran one app VM inside an instance and increased the number of instances. We assigned two vCPUs to each instance so that an instance could run one app VM and one management VM with maximum performance. Since the used PC had only four cores, we could run up to two instances without CPU contention. We measured the throughput of lighttpd when we sent requests for 1 KB files. The total throughput in two instances became twice of that in one instance.

Finally, we investigated instance-level scale-up when running lighttpd in one app VM. We changed the number of vCPUs assigned to the instance. Fig. 5(b) shows the throughput when we sent requests for 1 MB files using httperf. As the number of vCPUs was increasing, the throughput was improved.

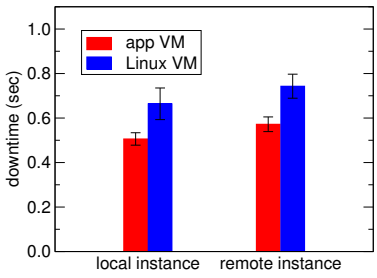


Figure 6. The downtime during VM migration between instances.

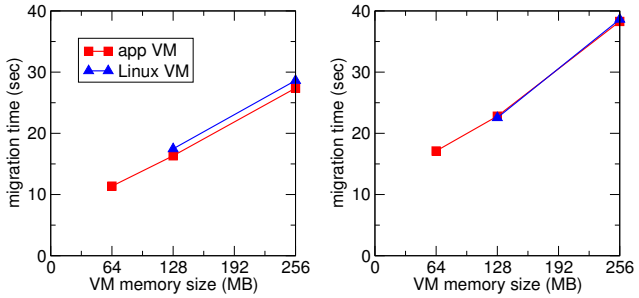
C. Performance of App VM Migration

First, we measured the downtime during the pre-copy live migration of an app VM between instances. The downtime is the time until an app VM is restarted in the destination instance after it is stopped in the source instance. For comparison, we migrated a regular VM that ran Linux. In this experiment, we did not run any active applications. Fig. 6 shows the downtime when we migrated VMs to another instance at the same and remote hosts, respectively. We measured the downtime for VMs of various memory sizes, but the downtime was almost the same. Therefore we show the average and the standard deviation. The downtime of the app VMs was sufficiently short and shorter than that of the Linux VM. This is partly because OSv supports only the smaller number of virtual devices to be suspended. When VMs were migrated to a remote host, the downtime became approximately 0.1 seconds longer.

Second, we measured the migration time when we changed the memory sizes of VMs. The migration time is the time needed for the execution of the migration command. Fig. 7(a) shows the results when we migrated an app VM and a Linux VM to another instance at the same hosts. Since the migration time is proportional to the memory size of VMs, the app VM has an advantage over the Linux VM because it can run with the smaller amount of memory. The app VM can run in only 64 MB, whereas the Linux VM needs 128 MB at least. In the minimal memory size, VM migration of the app VM was 1.5 times faster. Fig. 7(b) shows the results of VM migration to the remote host. The migration time became 10 seconds longer than VM migration to the same host. In this case, the migration time of the app VM was almost the same as that of the Linux VM.

D. Overhead of the VPN

To examine the overhead of the VPN across instances, we migrated an app VM to another instance and measured the throughput change of lighttpd. When we migrated the app VM to the instance at the same host, the throughput was degraded only by 1%. In contrast, the performance degradation was 24% when we migrated the app VM to a remote instance. This is because packets are forwarded to the destination instance by the VPN server.



(a) local instance (b) remote instance

Figure 7. The migration time between instances.

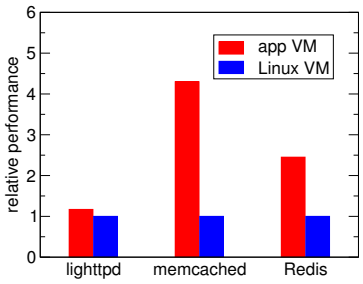


Figure 8. Application performance.

E. Fork Time

We measured the time needed for the execution of the fork function in an app VM. From our experimental result, it was shown that the fork time was slightly proportional to the memory size of the app VM. For an app VM with 256 MB of memory, it took 1.9 seconds. The time for restoring qemu-dm occupies a large portion of the fork time. In addition, the fork time includes the overhead of nested virtualization, which increases the time by 0.8 seconds. In other words, the fork time is shortened if the implementation of nested virtualization is improved.

When we achieved VM fork using Xen's standard save and restore commands, the fork time increased significantly as the memory size of the app VM was increasing. For an app VM with 256 MB of memory, our VM fork was 36 times faster.

F. Application Performance

We ran various applications in an app VM and compared its performance with that in the Linux VM. Fig. 8 shows the relative performance based on the performance of the Linux VM. For lighttpd, the throughput of the app VM was almost the same as that of the Linux VM. The performance degradation due to using NAPT was 3%. For memcached, the performance of the app VM was 4.3 times higher because OS^v provides an optimized version of memcached, which does not use the general-purpose socket API, for example. For Redis, the performance of the app VM was 2.5 times higher when Redis used pipelining, which sends new requests without waiting for responses.

VI. Related Work

Picocenter [22] runs applications in Linux containers inside instances of existing IaaS clouds. To optimize instance deployment, it swaps applications to and from cloud storage by checkpointing and rapidly restoring containers. Since Picocenter is designed for long-lived, mostly idle applications, it can cause too frequent swaps for periodically accessed applications. In addition, the hub manages IP addresses and ports and assigns available ones to swapped-in applications. Therefore, the IP address of an application may be changed whenever the application is swapped in.

VMware vCloud Air Virtual Private Cloud OnDemand [20] supports seamless and flexible scale-up and -down of instances. Users can dynamically increase or decrease the amount of resources assigned to their instances, according to application demands. Unlike most of exiting IaaS clouds, they do not need to create a new instance of appropriate type and move applications to it for scale-up and -down. Since users can pay only for assigned resources, not for instances, the cost can be reduced for under-utilized applications. This is one implementation of the Resource-as-a-Service cloud [5]. However, the number of vCPU cannot be decreased less than one. FlexCapsule allows multiple applications to share one vCPU by application consolidation.

There are several studies on multi-process support in a library OS. Xok/ExOS [9] can run the existing Unix applications without modifications using the nano kernel called Exokernel [6] and the library OS. ExOS provides mechanisms for multi-process such as process fork and inter-process communication using shared memory. Since Xok/ExOS does not consider the migration of applications, seamless application consolidation cannot be achieved. Graphene [18] supports multi-process for applications with the Linux-compatible library OS. Its applications can perform inter-process communication using RPC via the library OS. In addition, Graphene achieves process fork and non-live application migration by application checkpointing. However, isolation between applications is weaker than VM-level isolation because applications run on top of the large host OS.

Including OS^v [10] that FlexCapsule uses, there are several library OSes running on top of the hypervisor. Libra [1] and GUK [8] run the Java VM with the library OS on the hypervisor to optimize the execution of Java applications. Libra provides the library OS with only functions that affect the performance of the Java VM and uses file systems and networks provided by Dom0 in Xen. GUK extends Mini-OS in Xen to run the Java VM. It improves the memory management of Mini-OS and adds support for SMP, memory ballooning, and VM suspension and resumption. Mirage [13] specializes the library OS to OCaml applications and gen-

erates a unikernel directly running on the hypervisor. FlexCapsule can use these library OSes for running app VMs.

Xen-Blanket [21] enables VMs to be run inside instances using nested virtualization and to be migrated between instances with different network segments. To construct a VPN between two instances, it connects virtual switches running in instances using a layer-2 tunnel. This approach is similar to FlexCapsule. However, Xen-Blanket runs a gateway server VM inside an instance to perform routing between VMs and the outside. FlexCapsule does not run such an extra VM but use NAPT. In addition, Xen-Blanket assumes VM migration between different clouds, whereas FlexCapsule assumes that inside a cloud. Therefore packet forwarding between instances is realistic.

VII. Conclusion

This paper proposed FlexCapsule, which runs each application in a lightweight VM, called an app VM, using a library OS. FlexCapsule can optimize instance deployment at application granularity. The migration of app VMs enables seamless application consolidation and de-consolidation and scale-up and -down. Strong isolation among app VMs enables secure application consolidation. We have implemented FlexCapsule in Xen and OS$^{\text{v}}$. The FlexCapsule OS cooperates with the FlexCapsule server to support VM migration, networking, process fork, and process pools. Experimental results show that FlexCapsule is effective for the optimization of instance deployment.

One of our future work is to enable various applications to run in app VMs. For example, we need to advance multi-process support such as inter-process communication. Another direction is to reduce the overhead of nested virtualization. Since OS$^{\text{v}}$ is fully virtualized except for PV drivers, we need to develop para-virtualized OS$^{\text{v}}$ or use more lightweight virtualization such as containers. Also, the network performance of app VMs should be improved using network optimization in Xen-Blanket.

Acknowledgment

This research was supported in part by JSPS KAKENHI Grant Number JP16K00101.

References

[1] G. Ammons, D. D. Silva, O. Krieger, D. Grove, B. Rosenburg, R. W. Wisniewski, M. Butrico, K. Kawachiya, and E. V. Hensbergen. Libra: A Library Operating System for a JVM in a Virtualized Execution Environment. In *Proc. Int. Conf. Virtual Execution Environments*, pages 44–54, 2007.

[2] L. Balliache. Querying libiptc HOWTO. http://www.tldp.org/HOWTO/Querying-libiptc-HOWTO/, 2002.

[3] 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*, pages 164–177, 2003.

[4] 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.

[5] O. A. Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. The Resource-as-a-service (RaaS) Cloud. In *Proc. Workshop on Hot Topics in Cloud Computing*, 2012.

[6] D. R. Engler, M. F. Kaashoek, and J. J. O'Toole. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proc. Symp. Operating Systems Principles*, pages 251–266, 1995.

[7] B. Fitzpatrick. memcached – A Distributed Memory Object Caching System. http://memcached.org/.

[8] M. Jordan and H. Roeck. Guest VM Microkernel, GUK. https://kenai.com/projects/guestvm, 2009.

[9] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. B. no, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proc. Symp. Operating Systems Principles*, pages 52–65, 1997.

[10] A. Kivity, D. Laor, G. Costa, P. Enberg, N. HarEl, D. Marti, and V. Zolotarov. OSv – Optimizing the Operating System for Virtual Machines. In *USENIX Annual Technical Conf.*, 2014.

[11] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *Proc. European Conference on Computer Systems*, 2009.

[12] Lighty Team. Lighttpd - fly light. https://www.lighttpd.net/.

[13] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library Operating Systems for the Cloud. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pages 461–472, 2013.

[14] OpenVPN Technologies, Inc. OpenVPN – Open Source VPN. https://openvpn.net/.

[15] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proc. Symp. Operating Systems Design and Implementation*, pages 361–367, 2002.

[16] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the Library OS from the Top Down. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pages 291–304, 2011.

[17] Redis Labs, Inc. Redis. http://redis.io/.

[18] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter. Cooperation and Security Isolation of Library OSes for Multi-process Applications. In *Proc. European Conf. Computer Systems*, 2014.

[19] W. Verhelst. Network Block Device. http://nbd.sourceforge.net/.

[20] VMware, Inc. VMware vCloud Air – Public Cloud Computing from VMware. http://vcloud.vmware.com.

[21] D. Williams, H. Jamjoom, and H. Weatherspoon. The Xen-Blanket: Virtualize Once, Run Everywhere. In *Proc. European Conf. Computer Systems*, pages 113–126, 2012.

[22] 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.