Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

EXTENDED CONFERENCE PAPER

Flexible Service Consolidation with Nested Virtualization and Library Operating Systems

Kenichi Kourai* | Kouta Sannomiya

[1]Department of Computer Science and Networks, Kyushu Institute of Technology, Iizuka, Japan

Correspondence
*Kenichi Kourai, 680-4 Kawazu, Iizuka, Fukuoka 820–8502, Japan. Email: kourai@ksl.ci.kyutech.ac.jp

Summary

In Infrastructure-as-a-Service clouds, users can reduce costs by scale-in or -down when running services are under-utilized. Since these optimizations of instance deployment require at least one minimum instance even for running an under-utilized service, cost reduction is limited. For further optimization, multiple services can be consolidated into one instance. However, services have to be stopped temporarily at the consolidation time and isolation between services becomes weaker after the consolidation. To solve these problems, this paper proposes FlexCapsule, which enables seamless and secure service consolidation in existing IaaS clouds. FlexCapsule runs each service in a lightweight virtual machine (VM) called an app VM, using a library operating system. An app VM runs inside an instance using a technique called nested virtualization. FlexCapsule can optimize instance deployment with negligible downtime by flexibly migrating app VMs. Thanks to strong isolation provided by app VMs, it can guarantee security between consolidated services. In addition, FlexCapsule provides multi-process support using app VMs by emulating process fork and process pools. We have implemented FlexCapsule in Xen using both fully virtualized OS[V] and para-virtualized MiniOS. Then we examined the effectiveness of FlexCapsule using several applications. Due to the premature implementation of nested virtualization in Xen, the performance of app VMs largely degraded, but we believe that the performance could be improved using several existing optimizations.

KEYWORDS:
IaaS clouds, service consolidation, nested virtualization, library operating system, process pool, virtual private network

1 | INTRODUCTION

Infrastructure-as-a-Service (IaaS) clouds provide users with instances, which are usually virtual machines (VMs), and users run their services 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 services become over-utilized. In contrast, they can decrease the number or the resource amount to reduce costs when their services 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 service 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 service. Although the cost of each instance may be low, the total cost could become high if users run many under-utilized services.

For further optimization, users can consolidate services running in multiple instances into one instance. When there are several under-utilized services, the user can run them in one instance and reduce costs. Later, when some of the services become over-utilized, the user can de-consolidate them to other instances. However, this service consolidation and de-consolidation cause service downtime when users move services between instances. This problem could be solved by using process migration[1], but a security issue arises due to consolidating services. Since multiple services 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 service consolidation for optimizing instance deployment. FlexCapsule runs each service in a lightweight virtual machine (VM) called an app VM, using a library operating system (LibOS)[2,3,4]. 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[5,6]. Since it constructs a virtual private network (VPN) for all app VMs across multiple instances, app VMs can be seamlessly migrated between instances. Thus FlexCapsule can optimize instance deployment without disrupting services enclosed in app VMs. In addition, it can guarantee security between services consolidated into one instance using strong isolation provided by app VMs. This strong isolation is useful particularly when multiple services of different users are run inside one instance, e.g., by service providers acting as an intermediary between IaaS clouds and multiple users.

We have implemented FlexCapsule in Xen 4.2.4[7]. FlexCapsule provides two types of LibOSes running in app VMs. One is based on OS^v 0.21[4], while the other is based on MiniOS in Xen. We have added migration support to these LibOSes because these LibOSes use para-virtualization techniques to reduce virtualization overhead and cannot be migrated without OS support. As a helper of the LibOS, FlexCapsule provides an OS server running inside each instance. For example, when the fork function in the LibOS is invoked, 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 emulate process pooling by creating process-like VM pools.

Our experimental results showed that FlexCapsule achieved flexible service consolidation. It could optimize instance deployment according to performance requirements of services. Migration performance of app VMs was better than the normal VM running Linux, thanks to smaller memory footprints. Unfortunately, the overhead of nested virtualization was large due to the premature implementation in Xen. We believe that this is not inherent in nested virtualization but that the performance could be improved using the optimizations proposed in several systems[8,9].

This paper is an extended version of our previous conference paper[10]. In this paper, we have developed the FlexCapsule LibOS based on MiniOS in addition to the OS^v-based one. Whereas OS^v is a fully virtualized LibOS with para-virtual device drivers, MiniOS is a completely para-virtualized LibOS. Therefore, MiniOS can further reduce the overhead and run in smaller memory footprint. Using the MiniOS-based FlexCapsule LibOS, we conducted several experiments and compared the results with those in the OS^v-based one.

This paper is organized as follows. Section 2 describes issues in optimizing instance deployment in existing IaaS clouds and related work. Section 3 proposes FlexCapsule for enabling seamless and secure service consolidation using app VMs. Section 4 describes its implementation and Section 5 shows experimental results. Section 7 concludes this paper.

2 | BACKGROUND

2.1 | Optimizing Instance Deployment

In IaaS clouds, the optimization of instance deployment is performed according to resource utilizations of instances, as illustrated in Fig. 1. The most popular optimization is scale-out and -in, which adjust the number of instances. When a service becomes over-utilized, the user can increase the number of instances by scale-out and distribute the load to more instances. In contrast, when a service becomes under-utilized, the user can decrease the number of instances by scale-in to reduce costs. However, if

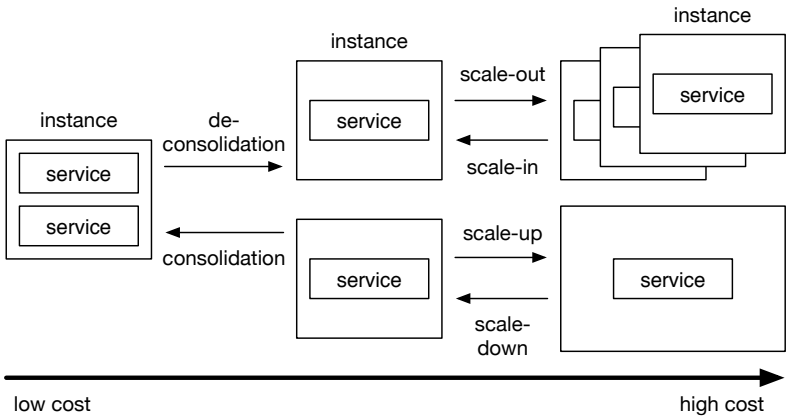


FIGURE 1 Various optimizations of instance deployment.

only one instance is deployed for an under-utilized service, the user cannot further reduce the number of instances. For example, consider intra servers that are rarely accessed during weekends and personal 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 services 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 a service 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 a service 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 services, move their data to the new instance, and restart these services in the new instance. This duration becomes downtime, for which services cannot be provided.

VMware vCloud Air Virtual Private Cloud OnDemand[11] 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 service demands. Unlike most of exiting IaaS clouds, they do not need to create a new instance of appropriate type and move services 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 services. This is one implementation of the Resource-as-a-Service cloud[12]. However, the number of vCPUs cannot be decreased less than one.

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

2.2 | VMs, Containers, and Serverless

IaaS clouds traditionally provide VMs as instances, but they also provide containers recently. Unlike a VM running on top of the hypervisor, a container runs on top of the OS and does not include the OS kernel. Since a container consumes a smaller amount of resources than a VM, it is possible to reduce costs. For example, Amazon ECS using Fargate[13] allows users to use less than one vCPU. Users can use containers with 0.25 vCPUs and 0.5 GB of memory at minimum. Like VMs, containers also support migration on the basis of process migration[1]. When multiple services are consolidated and de-consolidated and scale-up and -down, containers can be migrated between hosts or instances.

Zap[14] 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 enough because a pod is protected only by namespaces provided by the OS. Recently, there are many open-source projects for containers such as Docker[15] and LXD[16], but their migration support is still premature and unstable. Picocenter[17] runs one service in a Linux container and runs containers inside instances. Instead of process migration, it swaps out idle containers to cloud storage and swaps in them when the services in the containers are used. Since swapped-out containers cannot provide services at all, this mechanism is applicable only to mostly idle services.

Serverless computing is another execution model in recent clouds. Clouds provide server infrastructure, while users run only their services as functions running on it without server management. Unlike VMs and containers, users are charged only for the execution time of functions. This can reduce the cost for services that do not receive many requests. However, there are many limitations in serverless computing. Since services are stateless and the execution time is limited, users have to develop their services from scratch. For example, AWS Lambda[18] limits the execution time of a function to 15 minutes. In addition, the programming languages are also limited, e.g., JavaScript, Python, and Java. Therefore users cannot use legacy services written in C.

2.3 | Nested Virtualization

Nested virtualization[5] enables VMs and the hypervisor to run inside an outer VM. For Intel x86-based systems, nested virtualization has been developed without hardware support[6]. It virtualizes the CPU instruction set for virtualization by trapping and emulating those instructions. It also virtualizes MMU by using multi-dimensional paging. Nested virtualization is used for various purposes in clouds. For example, xCloud[19] and Xen-Blanket[8] allow users to use their own hypervisor and implement cross-platform live migration in public clouds. HVX[20] provides a thin virtualization layer using nested virtualization and enables users to run unmodified VMs in almost any existing IaaS clouds. Inception[21] takes one step further and enables users to build nested IaaS clouds on top of public clouds.

Nested virtualization causes larger overhead than traditional single-level virtualization. However, it is reported that the overhead is 6–8% for common workloads[6] , e.g., Linux kernel compilation and Java server processing. Special-purpose cloud hypervisors as used in CloudVisor[22] and TinyChecker[23] can improve the performance of nested virtualization more. Hardware support for nested virtualization has been also added. For example, Intel VMCS Shadowing[24] can eliminate VM exits due to VMREAD and VMWRITE instructions for accessing VMCS. The ARMv8.3 architecture[25] supports nested virtualization and its extension called NEVE has been proposed for coalescing and deferring traps[26]. We discuss further details of the performance of nested virtualization in Section 6.1.

2.4 | LibOS

A LibOS[2] enables each application to link most of the functions of the OS as a library and to perform its own resource management. Developers can customize a LibOS, considering characteristics of each application, and optimize application performance. There are many LibOSes running in VMs on top of the hypervisor. MiniOS in Xen[7] is a minimal OS and runs only one application in the kernel address space. Libra[27], GUK[28], and OSv[4] run the Java VM with the LibOS to optimize the execution of Java applications. Libra provides the LibOS with only functions that affect the performance of the Java VM and uses file systems and networks provided externally in Xen. GUK extends MiniOS and improves the memory management of MiniOS. It also adds support for SMP, memory ballooning, and VM suspension and resumption. OSv can run not only Java applications but also many existing C applications. MirageOS[3] specializes the MiniOS to OCaml applications and generates a unikernel directly running on the hypervisor.

Although a LibOS can run only one application, there are several studies on multi-process support. Xok/ExOS[29] can run the existing Unix applications without modifications using the nano kernel called Exokernel[2] and the LibOS. ExOS provides mechanisms for multi-process such as process fork and inter-process communication using shared memory. Graphene[30] supports multi-process for applications with the Linux-compatible LibOS. It provides an abstraction called a picoprocess[31], which runs on top of the host OS. Its applications can perform inter-process communication using RPC via the LibOS. In addition, Graphene achieves process fork and non-live migration by application checkpointing. KylinX[32] is based on MiniOS and achieves the fork function by VM fork[33]. Like normal OS processes, the memory of a VM running the LibOS is shared between parent and child VMs in a copy-on-write manner. KylinX also support shared libraries.

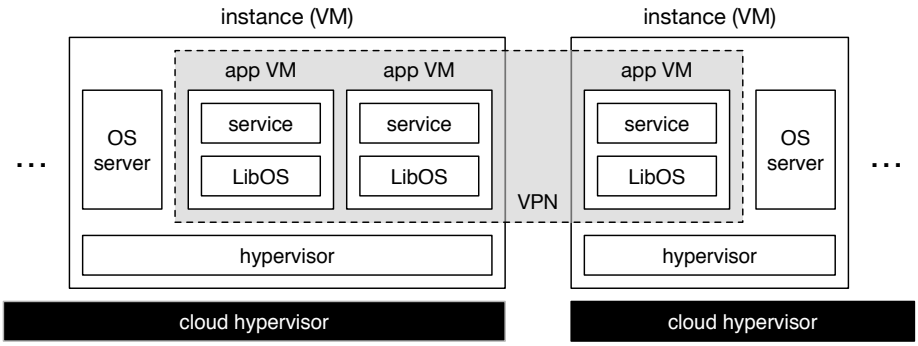


FIGURE 2 The system architecture of FlexCapsule.

3 | FLEXCAPSULE

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

3.1 | System Architecture

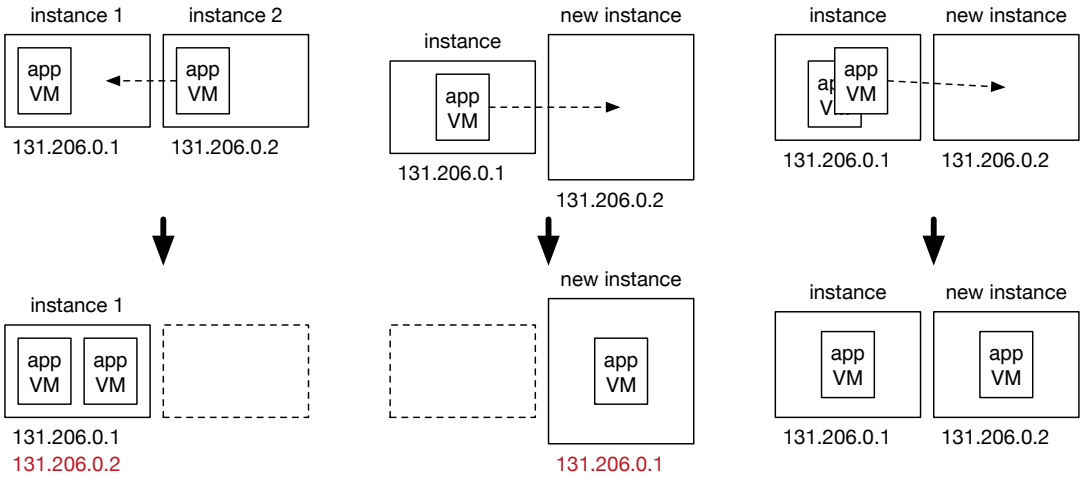
Fig. 2 illustrates the system architecture of FlexCapsule. FlexCapsule assumes IaaS clouds that run instances, which are VMs on top of the hypervisors. Using nested virtualization[6], FlexCapsule runs another hypervisor inside each instance and runs app VMs on top of the internal hypervisor. For simplicity, we hereafter omit the external hypervisor called the cloud hypervisor. Since the hypervisor is smaller and simpler than the OS, it is less vulnerable and more secure than the OS. Each app VM runs only one application process and a LibOS, 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 LibOS inside app VMs.

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.

3.2 | Optimization Using App VMs

When performing service consolidation, FlexCapsule migrates under-utilized app VMs to one instance, as illustrated in Fig. 3(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 as before service consolidation. In contrast, when performing service 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 a service up and down, FlexCapsule deploys a new instance of appropriate type and migrates app VMs in the original instance to the new one (Fig. 3(b)). Then it stops the original instance and re-assigns that public IP address to the new one. For scaling a service out, on the other hand, FlexCapsule deploys new instances, clones app VMs inside the original instances, and migrates them to the new ones (Fig. 3(c)). At this time, FlexCapsule assigns new private IP addresses to the cloned app VMs



(a) Consolidation **(b)** Scale-up **(c)** Scale-out

FIGURE 3 The optimization of instance deployment using the migration of app VMs.

but allows them to continue to use the original public IP addresses using NAPT. When scaling a service in, FlexCapsule simply stops several instances.

3.3 | FlexCapsule LibOS

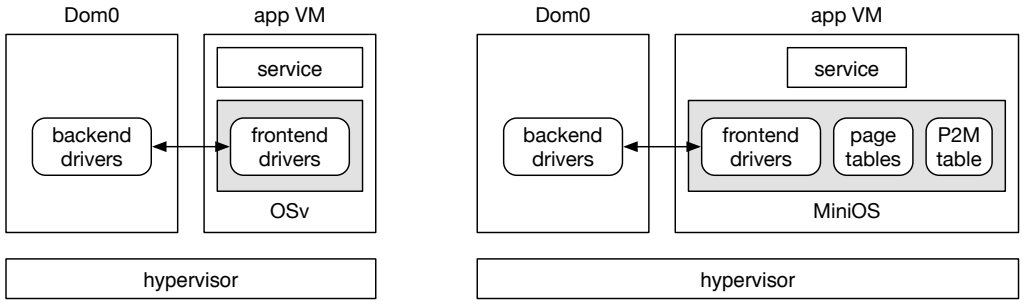
The FlexCapsule LibOS is a LibOS provided for services of FlexCapsule. This LibOS is a library linked to an application in an app VM and runs as part of the application. It provides the application with OS functions, which are traditionally provided through system calls. It can reduce the memory footprint of an app VM because only necessary functions in the FlexCapsule LibOS are linked to each application at compile time. Therefore running an app VM per service does not require extra memory so much, compared with using a general-purpose OS. The small memory footprint of an app VM enables more rapid VM migration by transferring only a smaller amount of memory. Similarly, cloning app VMs becomes faster.

The FlexCapsule LibOS reduces the overhead of extra virtualization due to app VMs by using para-virtualization. Para-virtualization is a technique that does not emulate real hardware completely to simplify hardware virtualization. Since full virtualization emulates real hardware as it is, the combination with nested virtualization poses larger overhead. Therefore, using para-virtualization can improve the performance of app VMs. In compensation for this performance gain, however, OSes with para-virtualization need to support VM migration by themselves. This is because such OSes are more tightly coupled with the hypervisor and virtual hardware. Specifically, the FlexCapsule LibOS enables itself to be suspended and resumed.

3.4 | FlexCapsule Server

The FlexCapsule server is an OS server running in each instance. It emulates functions related to process management and multi-process, which cannot be supported only by the FlexCapsule LibOS. For process management, users can obtain the list of running app VMs using the ps-like command and terminate specified app VMs using the kill-like command. For multi-process support, the FlexCapsule server is used for cooperation between app VMs. For example, when a service in an app VM invokes the fork function, the FlexCapsule server clones the entire app VM by VM fork[33]. At this time, the FlexCapsule LibOS 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 traditional process fork.

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 a service 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.



(a) OS^v-based **(b)** MiniOS-based

FIGURE 4 Two different FlexCapsule LibOSes.

For load balancing, the FlexCapsule server enables app VMs to emulate process pooling, which is a technique for preparing multiple processes that wait for the same port using process fork. The FlexCapsule server configures NAPT rules so that packets are delivered to one of the app VMs in a process-like VM pool.

4 | IMPLEMENTATION

We have implemented FlexCapsule in Xen 4.2.4[7]. A VM called DomU is an instance provided by a cloud and runs user's virtualized system including VMs and the hypervisor using nested virtualization. In the user's virtualized system based on Xen 4.2.4, DomU is used as an app VM and the FlexCapsule server runs on top of Linux in a privileged VM called Dom0. To share information between Dom0 and VMs, the database called XenStore runs in Dom0.

4.1 | Two Types of FlexCapsule LibOSes

We have implemented two types of FlexCapsule LibOSes based on OS^v 0.21[4] and MiniOS in Xen. The reason why we selected these two is that these LibOSes have different characteristics. The system architectures of the two LibOSes are illustrated in Fig. 4. OS^v is an OS optimized for virtualized systems. It is fully virtualized and is called an HVM guest. When using Intel's hardware virtualization extensions, CPUs and memory are virtualized using VT-x and Extended Page Tables (EPT), respectively. To reduce the overhead of device emulation, OS^v provides para-virtual (PV) device drivers to use PV devices. A PV driver running in the OS is called a frontend driver in Xen and communicates with a backend driver running in Dom0. For this communication, these two drivers establish event channels and shared memory at initialization time. OS^v uses musl libc[34], which provides all of the ISO C99 and POSIX 2008 APIs and non-standardized APIs in Linux, BSD, and glibc. It emulates most of the Linux API except fork and execve. Therefore, OS^v can run many existing C applications with no or slight modification. In addition, it can run custom applications more efficiently.

On the other hand, MiniOS is an OS used for separating components in Dom0 into independent VMs. It is para-virtualized to minimize the virtualization overhead and is called a PV guest. CPUs and memory are virtualized without using hardware extensions. The hypervisor assigns a VM running MiniOS part of the real physical memory called machine memory. MiniOS manages virtualized physical memory called pseudo-physical memory in the VM. To translate pseudo-physical memory frame numbers (PFNs) into machine memory frame numbers (MFNs), it maintains the P2M table. The reverse translation is performed using the M2P table. MiniOS configures its page tables so that virtual addresses are directly translated into MFNs instead of PFNs. Like OS^v, MiniOS also uses PV drivers. It uses newlib[35], which is the standard ANSI C library for embedded systems. Newlib provides only stubs for several system calls, e.g., fork and execve, while MiniOS does not implement them. In addition, MiniOS does not support file-related API. It provides not only the C runtime but also the OCaml runtime and the GHC runtime for Haskell. However, it cannot run the existing applications easily.

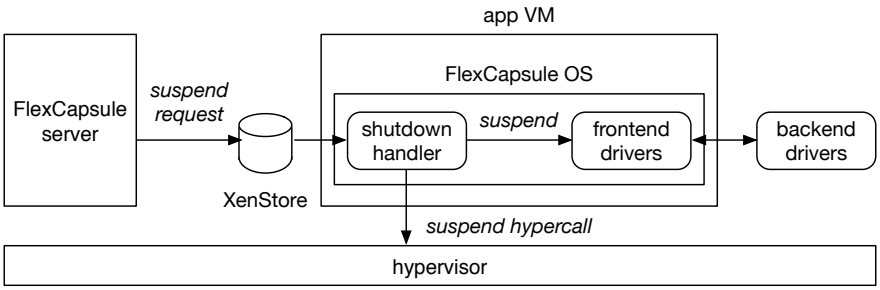


FIGURE 5 Suspending PV devices.

4.2 | Migration of App VMs

To migrate an app VM, the FlexCapsule server first transfers the memory contents of an app VM running in the source instance to the destination instance. Since the memory of the app VM continues to be modified during the memory transfer, the FlexCapsule server re-transfers modified memory contents. If the source and destination instances are in the same host, this memory transfer can be optimized[36]. It repeats the re-transfers until the amount of memory to be re-transferred is small enough and finally suspends the app VM. At this time, it saves the states of CPUs and virtual devices and transfers them. In the destination instance, it restores these states and resumes the app VM. The disk of the app VM is shared via NFS between instances as traditional.

Since the FlexCapsule LibOS uses para-virtualization techniques, it needs migration support at the OS level to migrate an app VM. When suspending an app VM at the final stage of VM migration, the FlexCapsule server writes a request for power management to the `control/shutdown` node in XenStore, as illustrated in Fig. 5. To monitor the node, the FlexCapsule LibOS starts a dedicated thread at boot time and registers the shutdown handler as a callback function. When the node value is changed, XenStore sends an event to the corresponding app VM using an event channel and the FlexCapsule LibOS invokes the registered shutdown handler.

The shutdown handler first suspends PV devices such as block and network devices. Event channels are established between the frontend and backend drivers, but they are not reused after VM migration. Therefore, the shutdown handler destroys all the event channels. Next, it invokes the `suspend` hypercall and then the hypervisor stops the app VM. In the destination instance, the hypercall returns when the app VM is restarted. Then the shutdown handler resumes PV devices and the frontend drivers re-establish new event channels with the backend drivers in Dom0.

In addition, the MiniOS-based FlexCapsule LibOS maintains bindings to the hypervisor after VM migration. Since MiniOS is a para-virtualized OS, it is more tightly coupled with the hypervisor than OS$^{\mathrm{v}}$. For example, MiniOS maintains a grant table for managing memory sharing between VMs. Since shared memory between an app VM and the other VMs is destroyed after VM migration, the FlexCapsule LibOS removes entries in the grant table on VM suspension and re-creates them on VM resumption.

Also, MiniOS manages the P2M table with a three-level tree structure, as shown in Fig. 6. Each tree node is one page and is linked to the lower nodes using the MFNs of those pages. The L1 lists form one contiguous array that contains MFNs corresponding to PFNs in order. Since MFNs are frame numbers of real physical memory, they are dependent on an instance. When a VM is resumed in the destination instance, the FlexCapsule server re-constructs the L1 lists by overwriting them using MFNs newly assigned to the VM. However, it is difficult to re-construct the L2 and L3 lists because the FlexCapsule server does not know how to replace old MFNs with new MFNs. Therefore, the FlexCapsule LibOS temporarily replaces MFNs in the L2 and L3 lists with machine-independent virtual addresses on VM suspension. Upon VM resumption, it replaces the virtual addresses with new MFNs using the memory information already re-constructed in the L1 lists.

When the MiniOS-based FlexCapsule LibOS issues a hypercall for suspending the app VM, it passes the `start_info` structure to the hypervisor. This data structure contains information on restarting the app VM such as a wallclock and the `shared_info` structure including the state of vCPUs. The OS$^{\mathrm{v}}$-based FlexCapsule LibOS does not need to pass this data structure.

In the current implementation, the state of the FlexCapsule server does not need to be transferred together with an app VM. For example, the NAPT rules for the app VM continue to be applied in the source instance, as explained in Section 4.4.

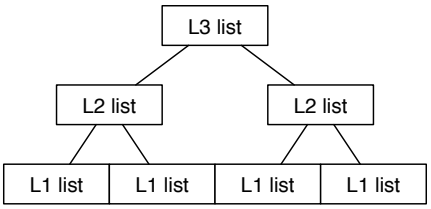


FIGURE 6 The P2M table with a tree structure.

4.3 | Fork of App VMs

Since the FlexCapsule LibOS provides the fork function compatible with the standard C library, a service in an app VM can invoke the function as done by traditional C applications. Then the FlexCapsule LibOS sends a fork request to XenStore in Dom0 using XenBus, which is established between the app VM and Dom0. The request is written to the fork node in XenStore. When the FlexCapsule server reads the request from XenStore, it suspends a parent app VM and creates a child app VM. It configures a newly allocated IP address for 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 starts to run the child app VM, whereas it resumes the parent app VM by canceling the VM suspension.

Our VM fork is similar to those of Potemkin[37], SnowFlock[33], and KylinX[32], but there are two differences. First, FlexCapsule supports cloning of VMs for not only para-virtualization but also full virtualization. In full virtualization, VM states to be copied are different from those in para-virtualization. Second, FlexCapsule eagerly copies the entire memory of a VM at fork time although the previous work copies it on demand. This is because the cost for constructing EPT for copy-on-write was similar to that for memory copies in our experiment.

FlexCapsule delegates the fork system call to the FlexCapsule server. Such system call delegation mechanisms have been developed in many systems: Proxos[38], the Libra LibOS[27], and multi-kernels such as FusedOS[39], McKernel[40], and mOS[41]. Compared with them, the communication via XenStore is not more efficient. This is because that communication uses message passing and increases the number of software interrupts and VM switches. However, the fork system call is not frequently invoked. Therefore, the overhead of using XenStore does not affect the performance of app VMs largely. To reduce the overhead, FlexCapsule can use the noxs (no XenStore) mechanism in LightVM[42].

4.3.1 | 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 MFN to the corresponding PFN. For PV guests, the FlexCapsule server needs special cares for the page tables and the P2M table, as illustrated in Fig. 7. When the hypercall copies pages used for page table entries (PTEs), it replaces MFNs stored in them because those MFNs are valid only for the parent app VM. It first canonicalizes an MFN for the parent app VM into the corresponding PFN using the M2P table. Then it uncanonicalizes the PFN into an MFN used for the child app VM using the P2M table. After all the PTEs are modified, the hypercall sets the page type to the L1 to L4 page table.

For the P2M table, the hypercall replaces MFNs similarly. Since the P2M table has a tree structure as in Fig. 6, the hypercall first finds the L1 lists by traversing the tree from the root. Then it replaces MFNs in the L1 lists using the M2P table. The M2P table is constructed by the hypervisor when the memory of the child app VM is allocated. The L2 and L3 lists also contain MFNs for the parent app VM, but the hypercall does not replace those MFNs. Those lists are re-constructed by the MiniOS-based FlexCapsule LibOS on resuming an app VM, as described in Section 4.2.

Next, the hypercall copies the states of the parent app VM to the child app VM. For HVM guests, it copies CPU states such as PAE, the TSS in the virtual 8086 mode, the identity-map page directory, and the location of ACPI control blocks. Also, it copies information on the ring buffers used for memory events, console, XenStore, and I/O requests to the child app VM. Then it saves the HVM context in the parent app VM and loads it to the child app VM. For PV guests, the hypercall copies the vCPU context of the parent app VM to the child app VM. At that time, it replaces the MFNs used for the start_info, GDT frames, and the page table base pointer for the child app VM. In addition, it copies the shared_info structure. For both HVM and PV guests, the hypercall copies the time stamp counter.

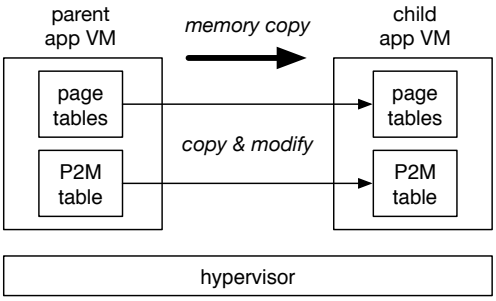


FIGURE 7 Memory duplication for a PV guest.

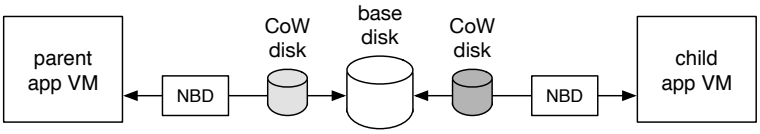


FIGURE 8 Dynamic copy-on-write sharing of a disk.

Finally, the FlexCapsule server duplicates device states only for HVM guests. It saves the device states in qemu-dm for the parent app VM by sending a QMP command. Xen-specific QEMU called qemu-dm is a device emulator and runs in Dom0. Then the FlexCapsule server starts qemu-dm for the child app VM with the saved states.

4.3.2 | 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 (CoW) disks for these app VMs, respectively, as illustrated in Fig. 8. A CoW 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 the qcow2 format, writes are performed to a CoW disk, whereas reads are performed from the disk if disk blocks 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)[43]. NBD transfers block data between the NBD device and the NBD server on demand. When an app VM accesses its disk via the NBD device, the device locally communicates with the NBD server and performs read and write to the disk on the remote NFS server. Using NBD, an app VM suffers only from the extra overhead of this local communication. For the child app VM, the FlexCapsule server connects one CoW 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 re-connects the other CoW disk to the original NBD device. As such, the parent app VM can use the CoW disk 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.

4.4 | Networking

Fig. 9 illustrates networking in FlexCapsule. When a service invokes the listen function to wait for new network connections, the FlexCapsule LibOS 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 Linux 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 library[44] for manipulating netfilter, which is a packet filtering framework in Linux. When a service invokes the close function for a socket, the FlexCapsule LibOS 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 pool of 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 a service invokes the fork function or the FlexCapsule server clones an app VM for scale-out, the FlexCapsule server examines whether the app VM is

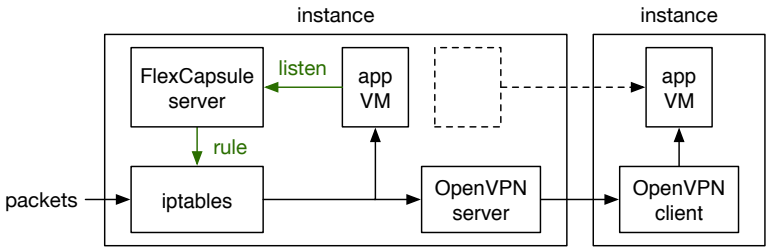


FIGURE 9 Networking in FlexCapsule.

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.

The FlexCapsule server constructs one VPN that connects all app VMs inside user's instances using Ethernet bridging of OpenVPN 2.3.2[45]. 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.

5 | EXPERIMENTS

We conducted experiments to examine the effectiveness of FlexCapsule. In our experiments, we used four app VMs running the OS^v-based FlexCapsule LibOS (OS^v VMs). For each app VM, we ran one of the four services: lighttpd 1.4.35[46], memcached 1.4.21[47], Redis 3.0.1[48], and Dhrystone 2.1[49]. lighttpd is a single-threaded web server, memcached is a memory object caching system, Redis is an in-memory database, and Dhrystone is a computing benchmark. For Redis, we used pipelining for sending new requests without waiting for responses. We measured the server throughput using httperf 0.9.0[50] for lighttpd, memaslap[51] for memcached, and redis-benchmark[52] for Redis. For an app VM running the MiniOS-based FlexCapsule LibOS (MiniOS VM), we ran only Dhrystone because it was difficult to port the other existing services.

We used two PCs with an Intel Xeon E3-1290 v2 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 and assigned four vCPUs and 2 GB of memory for each. Inside these instances, we ran several app VMs, each of which was assigned one vCPU and 4 to 1024 MB of memory.

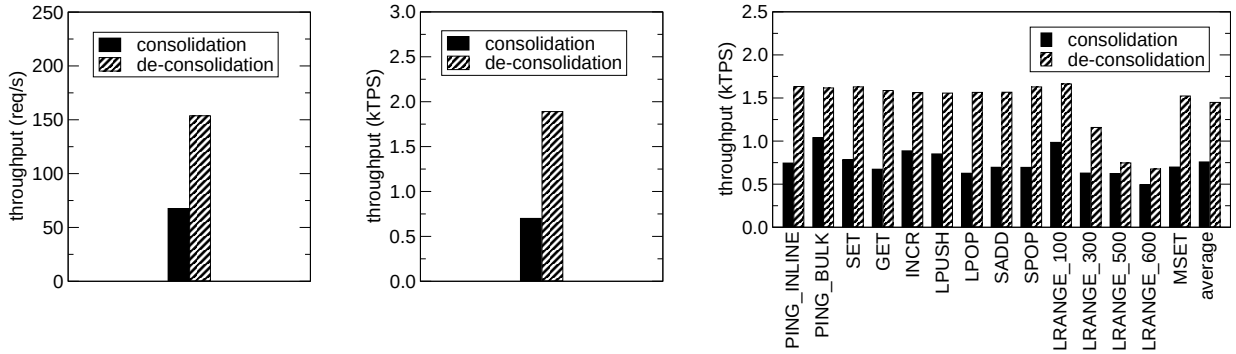
5.1 | Service Consolidation

To show the effectiveness of service consolidation using app VMs, we measured changes in service performance after consolidation and de-consolidation. In this experiment, we used three OS^v VMs running lighttpd, memcached, and Redis, respectively. When consolidating these app VMs, we ran them in one instance with one vCPU. When de-consolidating them, we used three instances, each of which had one vCPU.

Fig. 10 shows the performance after service consolidation and de-consolidation. In all the services, the performance after de-consolidation was 1.9 to 2.7 times higher than that after consolidation. This means that service de-consolidation can improve the service performance even when services run in app VMs. In other words, service consolidation is useful if each app VM does not use so large amount of resources. When only one of the three app VMs was busy in one instance, it could gain the performance similar to that after de-consolidation.

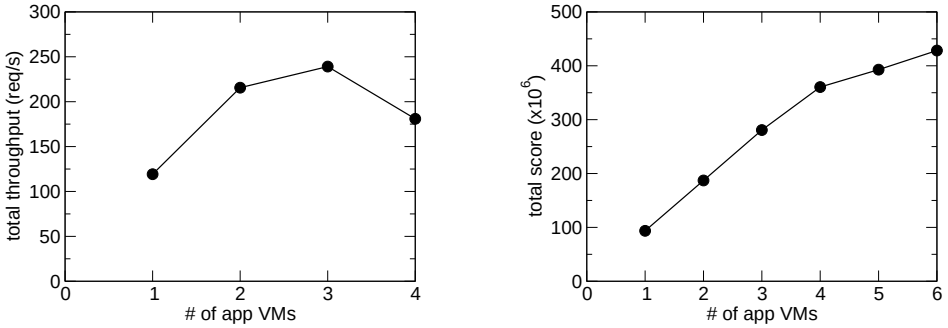
5.2 | Scale-out

First, we investigated whether app VM-level scale-out was effective. We increased the number of app VMs running in one instance and measured the total performance of all the app VMs. For OS^v VMs, we ran lighttpd and measured the throughput when we sent requests for 1 KB files. Fig. 11(a) shows the total throughput of app VMs. When we increased the number of app VMs, the total throughput also increased until three app VMs. It was doubled by using two app VMs, but the increase was small



(a) lighttpd **(b)** memcached **(c)** Redis

FIGURE 10 The performance after service consolidation and de-consolidation.



(a) lighttpd on OS^v **(b)** Dhrystone on MiniOS

FIGURE 11 The performance improvement by app VM-level scale-out.

in three app VMs. When we consolidated four app VMs, the throughput rather decreased. This is because three app VMs and one Dom0 inside the instance used up four vCPUs.

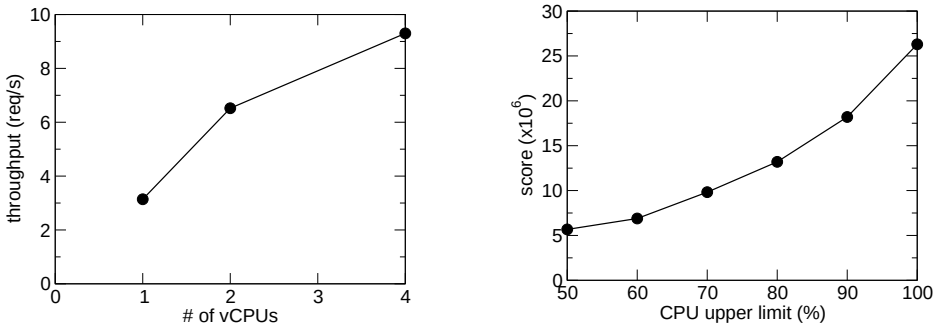
For MiniOS VMs, we ran Dhrystone and measured its score. Like lighttpd, Dhrystone uses a single thread. Fig. 11(b) shows the total score, which linearly increased to four app VMs because executing Dhrystone did not need Dom0. The total score continued to increase for more app VMs although the increase became smaller. This means that one app VM running Dhrystone could not use up one vCPU.

Second, we investigated whether instance-level scale-out was effective. We increased the number of instances and measured the throughput of lighttpd. When we ran one OS^v VM per instance, the total throughput in two instances became twice. Even when we fixed the number of app VMs to four in total, using two instances improved the total throughput. This is because each instance could run two app VMs and one Dom0 using four vCPUs.

5.3 | Scale-up

We investigated whether instance-level scale-up improved the performance of an app VM. We ran one OS^v VM running lighttpd in one instance and changed the number of vCPUs assigned to the instance. Fig. 12(a) shows the throughput of lighttpd when we sent requests for 1 MB files. As the number of vCPUs was increasing, the throughput was improved. For two vCPUs, the app VM and Dom0 could use one vCPU for each. The performance was still improved in four vCPUs although the app VM was assigned only one vCPU. This is because Dom0 used more than one vCPUs.

For a MiniOS VM, we ran Dhrystone and changed the CPU upper limit of an instance. Fig. 12(b) shows that the score was improved as the limit was relaxed. However, the performance was not proportional to the assigned CPU time. This is probably due to the vCPU scheduler in the hypervisor.



(a) lighttpd on OS^v **(b)** Dhrystone on MiniOS

FIGURE 12 The performance improvement by scale-up.

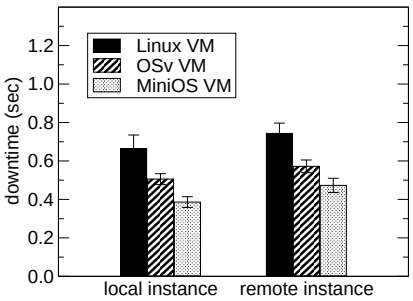


FIGURE 13 The downtime during VM migration between instances.

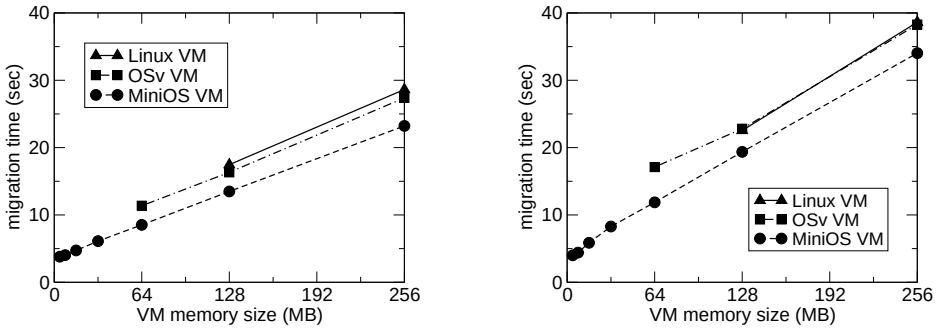
5.4 | Downtime

We measured the downtime during the migration of an app VM between instances. In this experiment, the downtime is the time until an app VM is restarted in the destination instance after the suspend request is sent to the app VM in the source instance. We migrated an OS^v VM and a MiniOS VM. For comparison, we ran a regular VM running Linux with PV drivers (Linux VM) in an instance and migrated it. We measured the downtime for the VMs with various memory sizes when we migrated a VM to another instance at the same host (local instance) and at the different host (remote instance).

Fig. 13 shows the average and standard deviation of the downtime. The downtime did not depend on the memory size of a VM and was almost the same. Overall, the downtime of the app VMs was sufficiently short. The downtime of the OS^v VM was shorter than that of the Linux VM. One reason is that OS^v supports only the smaller number of virtual devices to be suspended. Another is that suspending the state of PV devices was faster in OS^v. Compared with the OS^v VM, the downtime of the MiniOS VM was shorter. This means that complete para-virtualization makes the suspension of a VM faster. When VMs were migrated to a remote instance, the downtime became only 0.1 seconds longer.

5.5 | Migration Time

We measured the migration time between instances when we changed the memory sizes of app VMs. The migration time is the time needed for the execution of the migration command. Like the experiment in Section 5.4, we migrated the two app VMs and the Linux VM. Fig. 14(a) shows the results when we migrated a VM to another instance at the same host. The migration time was proportional to the memory size of a VM. The app VMs had an advantage over the Linux VM because they could run with the smaller amount of memory. The MiniOS VM needed only 4 MB at minimum, whereas the Linux VM needed 128 MB at least. In this case, the app VM could be migrated 4.6 times faster than the Linux VM. On the other hand, the OS^v VM needed 64 MB. In this minimal memory size, the migration time was still 1.5 times faster than the Linux VM. However, it was 3 times slower than that of the MiniOS VM.



(a) local instance **(b)** remote instance

FIGURE 14 The migration time between instances.

Fig. 14(b) show the migration time when we migrated VMs to another instance at a different host. Due to network overhead, the increase in migration time was more rapid. As a result, the migration of the MiniOS VM was 5.6 times faster than that of the Linux VM when these VMs were assigned the minimum amount of memory.

5.6 | Fork Time

We measured the time needed for the execution of the fork function in an app VM. The fork function in the FlexCapsule LibOS communicates with the FlexCapsule server and performs VM fork. For comparison, we measured the fork time when we naively achieved VM fork using only Xen's standard save and restore commands. Fig. 15(a) shows the comparison of the fork time in an OS^v VM and a MiniOS VM. Our VM fork was much faster than that using Xen's commands. With VM fork using Xen's commands, the fork time increased significantly as the memory size of an app VM was increasing. For app VMs with 256 MB of memory, our VM fork was 26 and 45 times faster for OS^v and MiniOS, respectively. Compared with the fork function in Linux, our VM fork needed much more time. In a Linux VM, it took only 5.4 ms because process fork is much faster than VM fork.

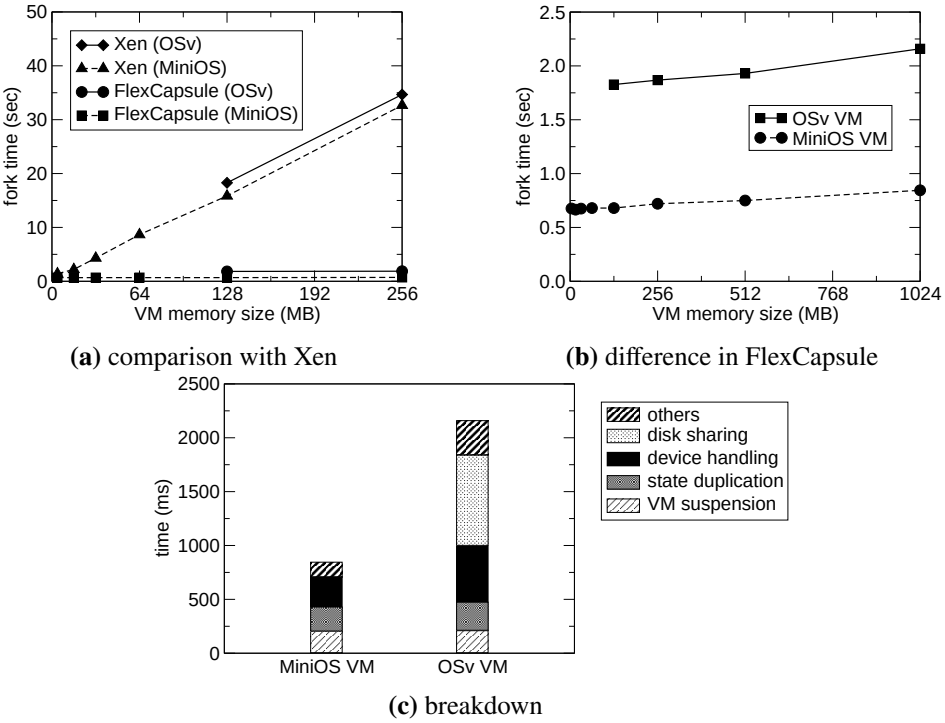
Fig. 15(b) compares the fork time in our VM fork in detail. The fork time did almost not depend on the memory size of an app VM. This means that even eager memory copies from a parent VM to a child app VM was very efficient. The fork of the OS^v VM was much slower than that of the MiniOS VM. Therefore we investigated the breakdown of the fork time for VMs with 1 GB of memory. As shown in Fig. 15(c), the largest difference was disk sharing. In the OS^v VM, it took 842 ms to share a disk using NBD. Since MiniOS does not support filesystems, we did not use a disk in the MiniOS VM. In addition, the time for device handling increased by 245 ms in the OS^v VM because device restoration was needed in qemu-dm. Another difference is the overhead of nested virtualization: 0.8 seconds for OS^v and 0.2 seconds for MiniOS. This is due to the difference between HVM and PV guests and that of the implementation complexity. It should be noted that we do not focus on implementing fast VM fork. Therefore, we do not optimize our implementation of VM fork. We can incorporate faster implementation in KylinX[32] with LightVM[42] to reduce the overhead.

5.7 | Listen Time

We measured the time needed for the execution of the listen function in an app VM. The listen function in the FlexCapsule LibOS communicates with the FlexCapsule server and registers a new NAPT rule. Fig. 16 shows the listen time in an OS^v VM, a MiniOS VM, and a Linux VM. The overhead of the registration was 5.6 and 8.3 ms in OS^v and MiniOS, respectively. This is not large for OS^v but is too large for MiniOS because the execution time of the listen function was very small in MiniOS. However, this does not degrade application performance because the listen function is not invoked so frequently.

5.8 | Service Performance

We ran various services in app VMs and compared its performance with that in a Linux VM. We ran Dhrystone in both an OS^v VM and a MiniOS VM, whereas we ran the other services only in OS^v VMs. For memcached, we used both an unoptimized



(a) comparison with Xen **(b)** difference in FlexCapsule

(c) breakdown

FIGURE 15 The fork time of app VMs.

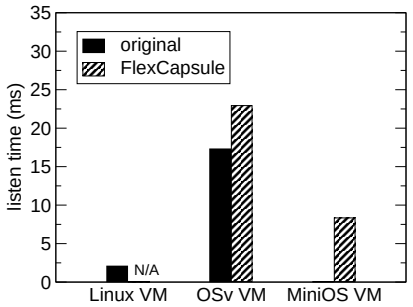
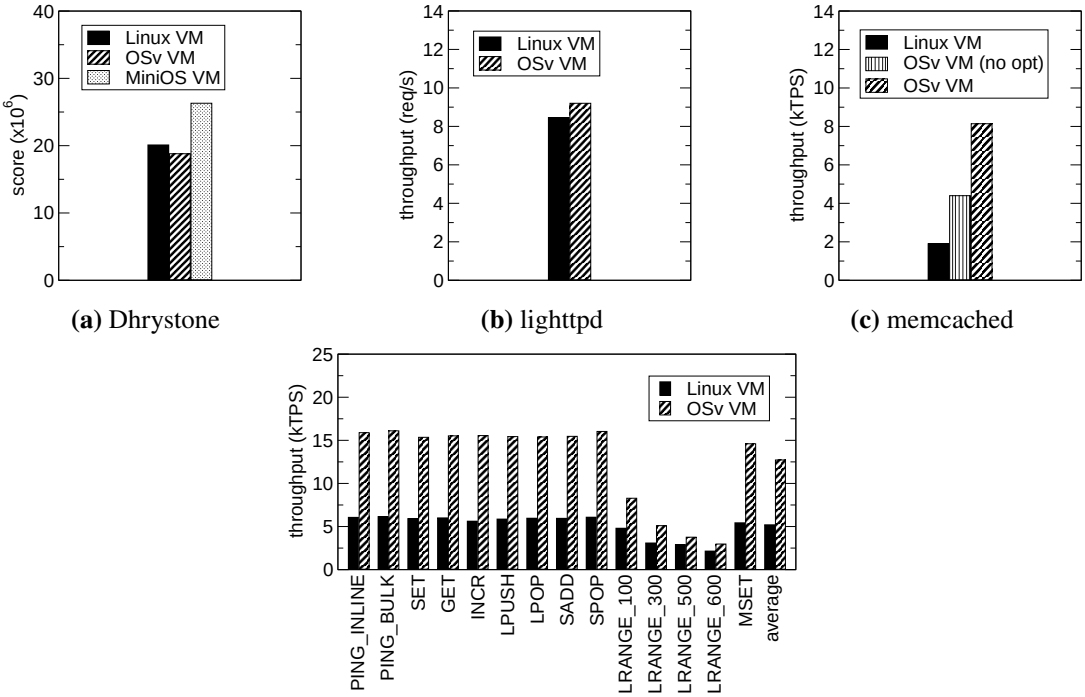


FIGURE 16 The listen time.

version and a version customized for OS^v. To examine the maximum performance, we assigned four vCPUs to each app VM. Fig. 17 shows the performance.

For Dhrystone, the performance of the MiniOS VM was the best and 31% higher than that of the Linux VM. This is probably because MiniOS did not suffer from process scheduling. In contrast, the performance of the OS^v VM was slightly worse than that of the Linux VM. The possible reason is that the implementation of string functions used by Dhrystone is slower in OS^v. Since Dhrystone mainly performs computation, it is not adequate to evaluate the differences between LibOSes. However, we need feature-rich LibOSes based on MiniOS, e.g., KylinX, to run various applications using MiniOS. Therefore, the detailed performance comparison between an OS^v VM and a MiniOS VM is our future work.

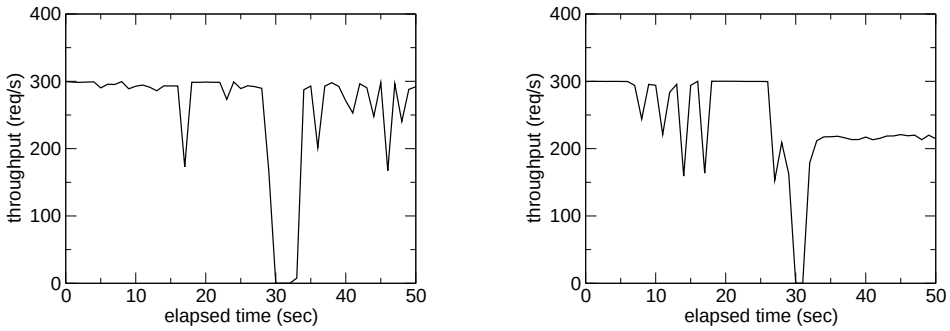
For lighttpd, the throughput of the app VM was 9% higher than that of the Linux VM. The performance degradation due to using NAPT was 3%. For memcached, the performance of the app VM was 4.3 times higher when we used a version customized for OS^v. Even using an unoptimized version, the performance was 2.3 times higher. For Redis, the performance of the app VM was 2.5 times higher on average.



(a) Dhrystone **(b)** lighttpd **(c)** memcached

(d) Redis

FIGURE 17 The performance of various services in app VMs.



(a) local instance **(b)** remote instance

FIGURE 18 Changes in throughput of lighttpd during VM migration.

5.9 | Overhead of the VPN

To examine the overhead of the VPN across instances, we migrated an OS^v VM to another instance and measured the throughput change of lighttpd in the app VM. After we migrated the app VM, packets were forwarded to the destination instance by the VPN server. Fig. 18 shows the changes in throughput when we sent 300 requests per second. When we migrated the app VM to the instance at the same host, the throughput was degraded by 8.7% on average and became unstable for a while. In contrast, the performance degradation was 28% but stable when we migrated the app VM to a remote instance. This is the overhead of packet forwarding. As described in Section 5.10, the network performance between remote instances was very low when nested virtualization was used in Xen.

Next, we measured the steady-state performance before and after the migration of app VMs running lighttpd, memcached, and Redis. After the migration, we waited for a while and measured the performance. As shown in Fig. 19, the performance of memcached and Redis degraded by 41% and 29%, respectively, even when the app VMs were migrated to a local instance. This is

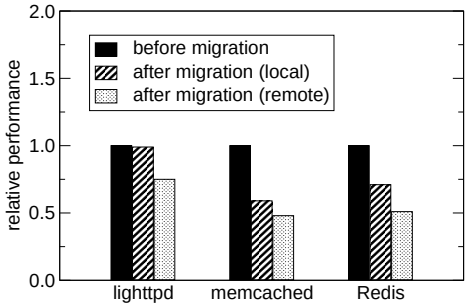


FIGURE 19 The performance after VM migration.

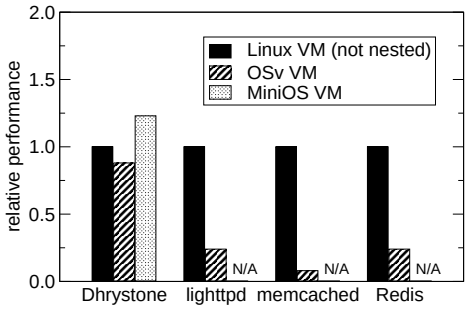


FIGURE 20 The overhead of nested virtualization in Xen.

because memcached and Redis were affected more largely by the overhead of packet forwarding. According to our analysis, the response size was 1.5 KB in lighttpd, while those were only 1 byte in memcached and 14 bytes in Redis. Therefore, memcached caused the largest overhead of nested virtualization.

5.10 | Overhead of Nested Virtualization

To examine the overhead of nested virtualization, we measured the performance of several services in the Linux VM using single-level virtualization. Then we compared it with the performance of app VMs using nested virtualization. Fig. 20 shows the performance of app VMs relative to the Linux VM using single-level virtualization. For Dhrystone, the performance of the OS^{v} VM degraded by 12%, while that of the MiniOS VM rather improved by 23%. This result is very similar to Fig. 17(a). On the other hand, the overhead was much large in the other services. It was 76% for lighttpd and Redis and 92% for memcached. We discuss this overhead in Section 6.1.

6 | DISCUSSION

6.1 | Performance of Nested Virtualization

As shown in Section 5.10, the overhead of nested virtualization is very large. However, we believe that the large overhead in our experiment is not inherent in nested virtualization. That is due to an issue of the premature implementation in Xen. In particular, Xen's low network performance in nested virtualization is critical to networked services. According to our experiments, the network throughput between remote instances was only 108 Mbps even using Gigabit Ethernet. Fortunately, this network performance can be largely improved by Xen-Blanket[8], which uses PV network drivers in Dom0 running in each instance. Xen-Blanket achieves network throughput comparable to single-level virtualization. VMs can receive network traffic at full capacity over Gigabit Ethernet.

For KVM, the overhead of nested virtualization is much smaller. It is reported that the overhead is 6–8% for common workloads[6]. Even the current implementation of KVM achieves 25 Gbps using 40-Gbps network[9], which is 69% of that in single-level

virtualization. HyperFresh[9] can further reduce the overhead by direct device assignment, dedicated physical CPUs, disabling the polling of virtual CPUs, and eliminating VM exits due to device interrupts. It improves network throughput to 36 Gbps, which is almost the same as that in single-level virtualization. These optimizations are not incorporated into FlexCapsule yet, but they could largely improve the performance of app VMs.

In contrast, CPU utilization is still high even in these systems. Compared with single-level virtualization, Xen-Blanket suffers from up to 15% larger overhead during network I/O using Gigabit Ethernet[8]. The overhead in HyperFresh is 22% larger in 40-Gbps network processing[9]. This can affect the service performance of app VMs in FlexCapsule. To prevent such a negative impact, users may need larger instances with more virtual CPUs, which leads to higher cost. However, this increase in cost can be smaller than cost reduction by service consolidation, scale-in, and scale-down.

6.2 | Nested Virtualization vs. In-VM Containers

In terms of performance, using containers in VM may be more promising than using nested virtualization. Users can freely run their own containers inside an instance (VM) provided by IaaS clouds. In our experiment, the performance overhead of such in-VM containers is only 7–10%, depending on the storage backend[53]. In this architecture, a container corresponds to an app VM in FlexCapsule. Container migration enables service consolidation, scale-up, and scale-down although its available implementation is currently premature. To migrate services flexibly, the entire container needs to be cloned when a process in the container issues the fork system call. However, directly using containers provided by IaaS clouds is neither cost-effective nor flexible. Since such clouds charge for containers, simply consolidating containers cannot reduce the cost. Therefore, multiple services in containers have to be moved to one container without any help of container's mobility. This makes seamless service consolidation, scale-up, and scale-down difficult.

The most critical drawback of using containers is that containers are less secure than app VMs. Since containers share the OS inside one instance, vulnerabilities in the OS can affect all the containers. Therefore, if unrelated services are consolidated or service consolidation is aggressively performed across multiple users, strong isolation of nested virtualization is indispensable. In contrast, weak isolation of containers may be sufficient for tightly coupled services. Recently, secure containers such as Kata Containers[54] and gVisor[55] are emerging. Kata Containers use optimized VMs, while gVisor restricts system calls issued by containers. These secure containers enhance the security, but the overhead also increases.

6.3 | Scheduling in Nested Virtualization

Using nested virtualization and LibOSes, FlexCapsule changes how to schedule services. In traditional clouds, the hypervisor schedules instances and the OS in each instance schedules application processes. In FlexCapsule, in contrast, the external cloud hypervisor schedules instances and the internal hypervisor in each instance schedules app VMs. In each app VM, the FlexCapsule LibOS schedules application threads if the only application in the app VM uses multiple threads. For efficient and controllable service consolidation, sophisticated scheduling algorithms are needed for the two hypervisors and the LibOS. For example, in the case of sharing one virtual CPU between multiple competing app VMs in an instance, the allocation of underlying physical CPUs to the app VMs has to be done in a fine-grained and timely manner. Soft real-time scheduling[56,57] may help such CPU allocation, but it needs to modify the hypervisor provided by existing IaaS clouds. In addition, it becomes more complex where to place services in FlexCapsule when app VMs are migrated to other instances. The cost of each instance can be different and therefore the migration strategy is important to reduce the cost. Since app VMs can be migrated even across multiple clouds[8], optimal placement algorithms are needed[58].

7 | CONCLUSION

This paper proposed FlexCapsule, which runs each service in a lightweight VM called an app VM using a LibOS. FlexCapsule can optimize instance deployment at service granularity. The migration of app VMs enables seamless service consolidation and de-consolidation and scale-up and -down. Strong isolation among app VMs enables secure service consolidation. We have implemented FlexCapsule in Xen using OS^v and MiniOS. The FlexCapsule LibOS cooperates with the FlexCapsule server to support VM migration, networking, VM fork, and process-like VM pools. Experimental results show that FlexCapsule could

optimize instance deployment using flexible service consolidation. At the same time, it is revealed that the performance of app VMs largely degraded due to the premature implementation of nested virtualization in Xen.

One of our future work is to enable various services to run in app VMs, particularly on MiniOS. We need to advance multi-process support such as the emulation of inter-process communication. Another direction is performance improvement of app VMs. Since OS^{v} running in app VMs is fully virtualized except for PV drivers, virtualization overhead is relatively large in nested virtualization. To reduce this overhead, we need to develop para-virtualized OS^{v}. Also, it is necessary to show that the performance of app VMs can be improved using the optimizations proposed in Xen-Blanket[8] and HyperFresh[9]. To reduce Xen's overhead, we are interested in running app VMs using Jitsu[59] and LightVM[42]. In addition, we plan to apply FlexCapsule to real public clouds such as Amazon EC2.

ACKNOWLEDGMENT

We thank anonymous reviewers for helpful comments. 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. Milojíčić DS, Douglis F, Paindaveine Y, Wheeler R, Zhou S. Process Migration. *ACM Computering Surveys* 2000; 32(3): 241–299.

2. Engler DR, Kaashoek MF, J. O'Toole J. Exokernel: An Operating System Architecture for Application-level Resource Management. In: Proceedings of the 15th ACM Symposium on Operating Systems Principles. ; 1995: 251–266.

3. Madhavapeddy A, Mortier R, Rotsos C, et al. Unikernels: Library Operating Systems for the Cloud. In: Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems. ; 2013: 461–472.

4. Kivity A, Laor D, Costa G, et al. OSv – Optimizing the Operating System for Virtual Machines. In: Proceedings of the 2014 USENIX Annual Technical Conference. ; 2014: 61–72.

5. Goldberg R. Architecture of Virtual Machines. In: Proceedings of Workshop on Virtual Computer Systems. ; 1973: 74–112.

6. Ben-Yehuda M, Day MD, Dubitzky Z, et al. The Turtles Project: Design and Implementation of Nested Virtualization. In: Proceedings of the 9th Symposium on Operating Systems Design and Implementation. ; 2010: 423–436.

7. Barham P, Dragovic B, Fraser K, et al. Xen and the Art of Virtualization. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles. ; 2003: 164–177.

8. Williams D, Jamjoom H, Weatherspoon H. The Xen-Blanket: Virtualize Once, Run Everywhere. In: Proceedings of the 7th ACM European Conference on Computer Systems. ; 2012: 113–126.

9. Doddamani S, Sinha P, Lu H, Cheng T, Bagdi H, Gopalan K. Fast and Live Hypervisor Replacement. In: Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. ; 2019: 45–58.

10. Kourai K, Sannomiya K. Seamless and Secure Application Consolidation for Optimizing Instance Deployment in Clouds. In: Proceedings of the 8th IEEE International Conference on Cloud Computing Technology and Science. ; 2016: 318–325.

11. VMware, Inc . VMware vCloud Air – Public Cloud Computing from VMware. http://vcloud.vmware.com; Accessed September 7, 2018.

12. Ben-Yehuda OA, Ben-Yehuda M, Schuster A, Tsafrir D. The Resource-as-a-service (RaaS) Cloud. In: Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Computing. ; 2012.

13. Amazon Inc . AWS Fargate – Run Containers without Having to Manage Servers or Clusters. https://aws.amazon.com/fargate/; Accessed March 14, 2019.

14. Osman S, Subhraveti D, Su G, Nieh J. The Design and Implementation of Zap: A System for Migrating Computing Environments. In: Proceedings of the 5th Symposium on Operating Systems Design and Implementation. ; 2002: 361–367.

15. Docker Inc . Docker: Enterprise Application Container Platform. https://www.docker.com/; Accessed March 14, 2019.

16. Canonical Ltd. . Linux Containers. https://linuxcontainers.org/; Accessed March 14, 2019.

17. Zhang L, Litton J, Cangialosi F, Benson T, Levin D, Mislove A. Picocenter: Supporting Long-lived, Mostly-idle Applications in Cloud Environments. In: Proceedings of the 11th European Conference on Computer Systems. ; 2016.

18. Amazon Inc . AWS Lambda – Serverless Compute – Amazon Web Services. https://aws.amazon.com/lambda/; Accessed March 14, 2019.

19. Williams D, Elnikety E, Eldehiry M, Jamjoom H, Huang H, Weatherspoon H. Unshackle the Cloud!. In: Proceedings of the 3rd USENIX Workshop on Hot Topics in Cloud Computing. ; 2011.

20. Fishman A, Rapoport M, Budilovsky E, Eidus I. HVX: Virtualizing the Cloud. In: Proceedings of the 5th USENIX Workshop on Hot Topics in Cloud Computing. ; 2013.

21. Liu C, Mao Y. Inception: Towards a Nested Cloud Architecture. In: Proceedings of the 5th USENIX Workshop on Hot Topics in Cloud Computing. ; 2013.

22. Zhang F, Chen J, Chen H, Zang B. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles. ; 2011: 203–216.

23. Tan C, Xia Y, Chen H, Zang B. TinyChecker: Transparent Protection of VMs against Hypervisor Failures with Nested Virtualization. In: Proceedings of IEEE/IFIP International Workshop on Dependability of Clouds, Data Centers and Virtual Machine Technology. ; 2012.

24. Intel Corp. . 4th Generation Intel Core vPro Processors with Intel VMCS Shadowing. 2013.

25. ARM Ltd. . *ARM Architecture Reference Manual – ARMv8, for ARMv8-A Architecture Profile*. 2017.

26. Lim JT, Dall C, Li S, Nieh J, Zyngier M. NEVE: Nested Virtualization Extensions for ARM. In: Proceedings of the 26th ACM Symposium on Operating Systems Principles. ; 2017: 201–217.

27. Ammons G, Silva DD, Krieger O, et al. Libra: A Library Operating System for a JVM in a Virtualized Execution Environment. In: Proceedings of the 3rd International Conference on Virtual Execution Environments. ; 2007: 44–54.

28. Jordan M, Roeck H. Guest VM Microkernel, GUK. https://kenai.com/projects/guestvm; Accessed March 8, 2017.

29. Kaashoek MF, Engler DR, Ganger GR, et al. Application Performance and Flexibility on Exokernel Systems. In: Proceedings of the 16th ACM Symposium on Operating Systems Principles. ; 1997: 52–65.

30. Tsai CC, Arora KS, Bandi N, et al. Cooperation and Security Isolation of Library OSes for Multi-process Applications. In: Proceedings of the 9th European Conference on Computer Systems. ; 2014.

31. Douceur J, Elson J, Howell J, Lorch J. Leveraging Legacy Code to Deploy Desktop Applications on the Web. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. ; 2008: 339–354.

32. Zhang Y, Crowcroft J, Li D, et al. KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization. In: Proceedings of the 2018 USENIX Annual Technical Conference. ; 2018: 173–185.

33. Lagar-Cavilla HA, Whitney JA, Scannell AM, et al. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In: Proceedings of the 4th ACM European Conference on Computer Systems. ; 2009: 1–12.

34. Felker R. musl libc. https://www.musl-libc.org/; Accessed March 14, 2019.

35. Red Hat, Inc . The Newlib Homepage. http://www.sourceware.org/newlib/; Accessed March 14, 2019.

36. Kourai K, Ooba H. VMBeam: Zero-copy Migration of Virtual Machines for Virtual IaaS Clouds. In: Proceedings of the 35th IEEE Symposium on Reliable Distributed Systems. ; 2016: 121–126.

37. Vrable M, Ma J, Chen J, et al. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In: Proceedings of the 20th ACM Symposium on Operating Systems Principles. ; 2005: 148–162.

38. Ta-Min R, Litty L, Lie D. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation. ; 2006: 279–292.

39. Park Y, Hensbergen EV, Hillenbrand M, et al. FusedOS: Fusing LWK Performance with FWK Functionality in a Heterogeneous Environment. In: Proceedings of the 24th International Symposium on Computer Architecture and High Performance Computing. ; 2012.

40. Soma Y, Gerofi B, Ishikawa Y. Revisiting Virtual Memory for High Performance Computing on Manycore Architectures: A Hybrid Segmentation Kernel Approach. In: Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers. ; 2014.

41. Wisniewski R, Inglett T, Keppel P, Murty R, Riesen R. mOS: An Architecture for Extreme-Scale Operating Systems. In: Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers. ; 2014.

42. Manco F, Lupu C, Schmidt F, et al. My VM is Lighter (and Safer) Than Your Container. In: ; 2017: 218–233.

43. Verhelst W. Network Block Device. https://nbd.sourceforge.io/; Accessed September 7, 2018.

44. Balliache L. Querying libiptc HOWTO. http://www.tldp.org/HOWTO/Querying-libiptc-HOWTO/; Accessed September 7, 2018.

45. OpenVPN Technologies, Inc . OpenVPN – Open Source VPN. https://openvpn.net/; Accessed September 7, 2018.

46. Lighty Team . Lighttpd - fly light. https://www.lighttpd.net/; Accessed September 7, 2018.

47. Fitzpatrick B. memcached – A Distributed Memory Object Caching System. http://memcached.org/; Accessed September 7, 2018.

48. Redis Labs, Inc . Redis. https://redis.io/; Accessed September 7, 2018.

49. Weicker R. Dhrystone: A Synthetic Systems Programming Benchmark. *Communications of the ACM* 1984; 27(10): 1013–1030.

50. Mosberger D, Jin T. httperf – A Tool for Measuring Web Server Performance. tech. rep., Hewlett-Packard Company; 1998.

51. Aker B. memaslap – Load Testing and Benchmarking a Server. http://docs.libmemcached.org/bin/memaslap.html; Accessed September 7, 2018.

52. Redis Labs, Inc . How fast is Redis?. https://redis.io/topics/benchmarks; Accessed September 7, 2018.

53. Morikawa T, Kourai K. Low-cost and Fast Failure Recovery Using In-VM Containers in Clouds. In: Proceedings of the 17th IEEE International Conference on Dependable, Autonomic and Secure Computing. ; 2019: 572–579.

54. The Openstack Foundation . Kata Containers – The speed of containers, the security of VMs. https://katacontainers.io/; Accessed March 14, 2019.

55. Google, Inc . Container Runtime Sandbox. https://github.com/google/gvisor; Accessed March 14, 2019.

56. Xi S, Wilson J, Lu C, Gill C. RT-Xen: Towards Real-time Hypervisor Scheduling in Xen. In: Proceedings of the 9th ACM International Conference on Embedded Software. ; 2011: 39–48.

57. García-Valls M, Cucinotta T, Lu C. Challenges in Real-time Virtualization and Predictable Cloud Computing. *Elsevier Journal of Systems Architecture* 2014; 60(9): 726–740.

58. Konstanteli K, Cucinotta T, Psychas K, Varvarigou T. Elastic Admission Control for Federated Cloud Services. *IEEE Transactions on Cloud Computing* 2014; 2: 348-361.

59. Madhavapeddy A, Leonard T, Skjegstad M, et al. Jitsu: Just-in-time Summoning of Unikernels. In: Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation. ; 2015: 559–573.