An Architectural Perspective for Cloud Virtualization

Yiming Zhang  
School of Computer  
NUDT  
Changsha, China  
ymzhang@nudt.edu.cn

Qiao Zhou  
School of Computer  
NUDT  
Changsha, China  
zhouqiao3518@sina.com

Dongsheng Li  
School of Computer  
NUDT  
Changsha, China  
dsli@nudt.edu.cn

Kai Yu  
School of Computer  
NUDT  
Changsha, China  
yukaigoforit@126.com

Chengfei Zhang  
School of Computer  
NUDT  
Changsha, China  
chengfeizh@gmail.com

Yaozheng Wang  
School of Computer  
NUDT  
Changsha, China  
wyaozheng@gmail.com

Jinyan Wang  
School of Computer  
NUDT  
Changsha, China  
wjy0213@vip.qq.com

Ping Zhong  
Central South University  
Changsha, China  
ping.zhong@csu.edu.cn

Yongqiang Xiong  
MSRA  
Beijing, China  
yqx@microsoft.com

Huaimin Wang  
School of Computer  
NUDT  
Changsha, China  
whm_w@163.com

Abstract—Virtual machines (VMs) and processes are two important abstractions for cloud virtualization, where VMs usually install a complete operating system (OS) running user processes. Although existing in different layers in the virtualization hierarchy, VMs and processes have overlapped functionalities. For example, they are both intended to provide execution abstraction (e.g., physical/virtual memory address space), and share similar objectives of isolation, cooperation and scheduling. Currently, neither of them could provide the benefits of the other: VMs provide higher isolation, security and portability, while processes are more efficient, flexible and easier to schedule and cooperate. The heavyweight virtualization architecture degrades both efficiency and security of cloud services.

There are two trends for cloud virtualization: the first is to enhance processes to achieve VM-like security, and the second is to reduce VMs to achieve process-like flexibility. Based on these observations, our vision is that in the near future VMs and processes might be fused into one new abstraction for cloud virtualization that embraces the best of both, providing VM-level isolation and security while preserving process-level efficiency and flexibility.

1. Introduction

Currently, virtual machines (VMs) and processes are two important abstractions for cloud virtualization. Hardware virtual machines [1] have been widely used to guarantee isolation [2, 3] and improve system reliability and security [4]. VMs usually emulate existing hardware architectures that run a complete operating system (OS) providing the environment for executing various applications in the form of user processes.

Although existing in different layers in the virtualization hierarchy, VMs and processes have overlapped functionalities. They are both intended to provide execution abstraction (e.g., memory address space), and share similar objectives of isolation, cooperation and scheduling. On the other hand, however, neither of them could provide the benefits of the other: VMs provide higher isolation, security and portability, while processes are more lightweight and easier to schedule and cooperate (with multi-process APIs like fork, IPC and signals). Therefore, both VMs and processes are needed by the mainstream software stacks, which not only decreases efficiency but also increases vulnerability [4, 5].

Recently, commodity clouds (like Amazon’s Elastic Computing Cloud [6] and Alibaba’s Aliyun [7]) have completely changed the economics of large-scale computing [8], providing a public platform where tenants run their specific applications (e.g., a web server or a database server) on dedicated VMs. The highly-specialized VMs require only a very small subset of the overall functionality provided by a standard OS.

This has motivated two trends for cloud virtualization: the first is to enhance processes to achieve VM-like security, and the second is to reduce VMs to achieve process-like flexibility. For example, picoprocesses [9] augment processes so as to address the isolation and efficiency problem of conventional OS processes, and Unikernels [4] statically seal only the application binary and requisite libraries into a single bootable appliance image so that Unikernel VMs could be lightweight and flexible.

Based on these observations, our vision is that in the near future the two abstractions of VMs and processes might be fused into one new abstraction for cloud virtualization that...
embraces the best of both VMs and processes, providing VM-level isolation and security while preserving process-level efficiency and flexibility. We argue that a promising approach for this fusion is to start from the VM abstraction and add desirable processes’ features following the library operating system (LibOS) [10] paradigm, which has been widely adopted in the virtualization literature [4, 11, 9, 12, 13, 14, 15, 16].

We present a reference implementation, dubbed cKernel (customized kernel), for the new abstraction. cKernel is essentially a LibOS based virtualization architecture, which (i) removes traditional OS and process layers and retains only VMs, and (ii) takes the hypervisor as an OS and imitates processes’s dynamic mapping mechanism for VMs’ pages and libraries. By drawing an analogy between conventional processes and VMs, cKernel has partially (i) realized the fork API for VMs, a basis for conventional multi-process programming abstractions, and (ii) supported dynamic library loading and linking for the minimized VMs.

The rest of this paper is organized as follows. Section 2 further introduces the background and motivation. Section 3 introduces a promising approach (cKernel) to realizing the fusion of processes and VMs. Section 4 presents cKernel’s reference implementation. Section 5 introduces related work. Section 6 discusses future work. And finally Section 7 concludes the paper.

2. Background

2.1. VMs Vs. Processes

VMs rely on the virtual machine hypervisor, like Xen [1] and Linux KVM [17], to provide an application with all its dependencies in a virtualized, self-contained unit. As shown in Fig. 1 (left), the hardware dependency is provided through the abstraction of an emulated machine with CPU kernel mode, multiple address spaces and virtual hardware devices, which is able to provide the software dependency by running a conventional operating system together with various libraries, perhaps slightly modified for para-virtualization [11]. VMs have been widely used for cloud computing since they provide desirable compatibility of major applications by reusing existing OS functions including the support of various hardwares.

However, running a conventional OS in a VM brings substantial overhead, since the guest OS may run many duplicated management processes and each guest OS may demand gigabytes of disk storage and hundreds of megabytes of physical memory. This has recently motivated the revival of LibOS for VM appliances [4, 18, 11]. For example, Unikernel [4] implements the OS functions (e.g., networking system and file system) as a set of libraries that can be separately linked into the VM appliance at compile time. The Unikernel (system libraries), application libraries and application binary are statically sealed into a single bootable image, which can run directly on the virtual hardware provided by the hypervisor. These VM oriented LibOS designs effectively reduce the image size and memory footprint, taking the first step towards making a virtual machine more like a process.

Meanwhile, another trend is to improve processes with VM-like isolation and security. For instance, the picprocess [9, 19] is a process-based isolation abstraction that can be viewed as a stripped-down virtual machine without emulated CPU kernel mode, MMU or virtual devices. Picprocess-oriented library OSs [12, 15, 13, 14, 16] provide an interface restricted to a narrowed set of host kernel ABIs [12], implementing the OS personalities as library functions and mapping high-level APIs onto a few interfaces to the host OS.


2.2. Vision

Currently, neither VMs nor processes could provide the benefits of the other, and thus both of them are needed by the mainstream software stacks. This not only decreases efficiency but also increases vulnerability of cloud services. The two threads (of enhancing processes to achieve VM-like security and reducing VMs to achieve process-like flexibility) have motivated our vision that in the near future VMs and processes might be fused into one new abstraction for cloud virtualization, with the goal of making the best of both VMs and processes.

Considering the great success of VMs in security and backward compatibility, in this paper we will discuss a promising approach for the fusion. We propose to start from the VM abstraction, strip off any unused functions of the accommodated applications, and add desirable processes’ features like inter-process communication and dynamic library linking to the minimized VMs. We follow the technical
3. Fusion of VMs and Processes

This section introduces a promising approach (called cKernel) to realizing the new abstraction, a LibOS based virtualization architecture that fuses VM and process and makes the best of both for the cloud.

3.1. cKernel VMs

Conventional general-purpose OSs are heavyweight and contain extensive features, so processes have been proposed to act as “lightweight VMs” that are cheaper to create, schedule and configure. In cloud environments, however, most cloud services only run a specific application in a single-purpose VM, which needs only a very small portion of conventional OS supports. Therefore, we expect to reduce the OS and make the VM become (even) as lightweight as a process.

Based on this observation, cKernel proposes to start from the VM abstraction and take minimalism to the extreme, sealing only the application binary into a single VM. Previous work [5] has showed that a VM could be booted/run in less than 3 milliseconds, which is comparable to process fork/exec on Ubuntu Linux (about 1 millisecond). This proves the feasibility for cKernel to realize high-performance process-like VM creation and scheduling (such as VM fork, IPC-like inter-VM communication, and dynamic library linking) by taking the hypervisor as an OS. By drawing an analogy between VMs and processes, cKernel naturally relies on the hypervisor for multicore scheduling, following the multikernel mechanism [24] of running one VM per vCPU.

In the conventional virtualization architecture, threads have been proposed as “lightweight processes”. Multiple threads running in one process could provide more efficient scheduling and communication. However, many production systems have showed that threads are difficult to rein in large-scale distributed environments [25] due to problems like data race, deadlock, and livelock, which are usually nondeterministic and counter-intuitive. On the other hand, in modern clouds one VM usually runs only one service instance, making threads less attractive than before. Therefore, cKernel adopts the simplified coroutine model [26] for concurrency, where many logical coroutines (each of which serves one client connection) run in the same VM/process so as to achieve fast sequential performance while alleviating the problems of threads.

Next, we will in turn introduce how to add the two important features of conventional processes to cKernel VMs, namely, dynamic mapping of pages and dynamic linking of libraries.

3.2. Process-Like Fork of VMs

Since the process abstraction has been stripped off from the simplified cKernel architecture, cKernel VMs need to realize IPC (inter-process communication) to support conventional multi-process applications. cKernel lets a multi-process application have the view that its processes run without any change, while these processes are actually cKernel VMs collaboratively running on the hypervisor.

The fork API is the basis to realize IPC. It is straightforward to leverage the hypervisor’s memory mapping functions (like Xen’s page sharing mechanism) to implement fork. The fork API will create a child VM by duplicating the calling parent VM and return its ID to the parent. cKernel needs to maintain all the information of the parent and child VMs, including the IDs, offsets within shared pages, references of the grant table, etc.

3.3. Process-like Library Linking of VMs

Full VMs could easily realize dynamic library linking by leveraging their rich-featured OS functionalities. In the cKernel architecture, however, the minimized VMs are monolithically sealed at compile time and thus libraries cannot be dynamically linked as before. Static library linking brings great inefficiency to cKernel. For example, the entire VM image has to be re-compiled and re-deployed to update each of its libraries.

Therefore, it is necessary to design a dynamic LibOS for cKernel’s minimized VMs, which supports to map the shared libraries (along with their states) onto the VMs’ virtual address space at runtime. To achieve this goal, cKernel follows the “shell-core” mechanism [2], sealing only the application binary into a single VM (shell) and dynamically linking libraries (core) at runtime. It (i) adds a dynamic segment to its virtual address space and (ii) dynamically loads and maps requisite libraries, in a similar way to the procedure of dynamic linking of conventional processes by the language runtime (like glibc) [27].

Specifically, cKernel initializes a VM and then jumps to an entry function provided by the hypervisor. cKernel adopts a single virtual address space where the basic OS functions (e.g., bootstrap and memory allocation), system libraries (including the language runtime), application binary and libraries, and global data structures co-locate to run without kernel/user space separation or process scheduling.

4. Reference Implementation

We present a reference implementation of cKernel by modifying MiniOS [11], a para-virtualized LibOS for user domains on Xen [1], where the application binary and necessary libraries are statically sealed into a single bootable image like Unikernel [4]. MiniOS has implemented the basic functionalities needed to run an appliance on Xen. It has a single address space and a simple scheduler without preemptive scheduling/threading.
cKernel leverages the GNU cross-compiling development tool chain (including the GCC compiler) to support source code (mainly C in our current prototype) compatibility, so that a legacy Linux application could be re-compiled to a cKernel appliance that can run directly on Xen. It incorporates the lightweight RedHat Newlib [28] to provide the standard C library functions.

### 4.1. Fork

CKernel *fork* is designed to duplicate the VM that accommodates the caller application. When the *fork* API is invoked in the parent VM, cKernel uses inline assemblies to get the values of CPU registers. Dom0 is responsible for duplicating and starting the child VM.

The parent domain uses the grant table to selectively share its data with its child, including the stack, heap, .bss and data sections. This improves the security by avoiding unauthorized access. For instance, the parent may choose not to share its stack if the child never access it; and in a key-value store application a child domain servicing a specific user should be given access to only that user’s data. Writable data is shared in a copy-on-write mode.

The forked child VM can access the authorized data of the parent VM. The child and parent may also use shared pages to communicate with each other, i.e., realizing conventional inter-process communication APIs in the cKernel architecture, such as pipe, signal, and message queue. The Xen-based implementation of inter-VM page mapping will be studied in our future work.

### 4.2. Dynamic Library Linking

The original design of MiniOS cannot support dynamic libraries. Fig. 2 shows the augmented memory layout of cKernel, which is divided into multiple subspaces for regions of text and data, frontend driver info, grant table, external I/O, heap, and Xen-reserved.

The text and data region contains (i) the kernel segment including the pre-compiled basic OS functions, (if any) static libraries and application binary, (ii) the dynamic segment where all the dynamic libraries are loaded, and (iii) the structures needed by cKernel such as the physical-to-machine (p2m) mapping table, arch-specific structures, page tables and stacks.

The front driver info region is for realizing the split driver model of Xen. The grant table region is used to share pages with dom0 and other domUs. The external I/O region maps the external devices to the virtual address space to facilitate I/O operations. The heap region is used to allocate memory growing in 4KB page chunks. And the Xen-reserved region is used by Xen for hypervisor-level management and not available by cKernel.

By emulating the procedure of dynamic mapping in UNIX processes that supports shared libraries, we have implemented cKernel’s dynamic mapping mechanism of shared libraries in the Xen control library (libxc), which is used by the upper-layer control tool stacks like XL (XM), Libvirt/VIRSH, and XAPI/XE.

After creating the domain boot loader, XL will build a para-virtualized domU, during the process of which it (i) parses the kernel image file, (ii) initiates the boot memory, (iii) builds the image in the memory, and (iv) boots up the image for domU. We modify the above 3rd step (of building image) and add a function *xc_dom_map_dyn()*, which maps the dynamic libraries at runtime to the virtual address of dynamic segment in the text and data region depicted in Fig. 2.

### 5. Related Work & Discussion

This section discusses the related work of the fused virtualization architecture, including virtualization techniques of virtual machines and containers (Section 5.1), modular kernels and library OSs (Section 5.2), and security issues (Section 5.3).

#### 5.1. Virtual Machines & Containers

Currently full virtual machine hypervisors like Xen [1] and Linux KVM [7] are the dominant virtualization tech-
nique that provides an application with all its dependencies on both hardware and software in a virtualized self-contained unit [29].

The full VM and rich-featured OS architecture is heavy-weight and inefficient. Therefore, researchers have proposed container-based virtualization techniques, such as picoproces ses [9], Linux Container [20], Docker [21], OpenVZ [22], and VServer [23]. A container is essentially an OS process plus its executing environment, so it is much smaller than a traditional full VM and boots up much faster.

The picprocess [9] is a process-based isolation container. It is a highly restricted process that is prevented from making kernel calls, and provides a single memory-isolated address space with strictly user-mode CPU execution and a narrow interface for implementing specific OS personalities [12].

Linux Container [20] (LXC) runs multiple isolated containers over a single Linux host, each of which creates its own virtual spaces for its processes. It uses Linux kernel cgroups [30] to multiplex and prioritize resources such as CPU cycles and memory, and leverages Linux namespace to isolate an applications’ view of users, processes, file systems and networking [31].

A Docker [21] container is made up of a base image plus multiple committed layers, and also uses resource isolation features of Linux to allow multiple independent dockers to run within a single Linux host. Traditionally, Docker supports a single process only, so a separate process manager like supervisor is needed to run multiple processes in Docker.

Although containers are smaller and have better performance, full VMs are still dominant in public commodity clouds and production platforms due to better isolation, security, portability and compatibility.

5.2. Modular Kernels & Library OSs

A microkernel [32] provides the basic functions such as virtual memory management and interprocess communication (IPC), while many other functions like file systems, drivers and protocols, are run in the user space. Microkernels have mainly argued for extensibility and flexibility [33] [34] [35] [36]. For example, the SPIN [33] microkernel allows applications to make policy decisions by safely downloading extensions into the kernel, and Scout [34], Vino [35] and Nemesis [36] efficiently realize the communication, database and multimedia applications on top of microkernels, respectively.

The exokernel architecture [10] achieves higher flexibility than microkernels by providing application-level secure multiplexing of physical resources and enabling application-specific customization of conventional OS abstractions. A small exokernel securely exports all hardware resources through a low-level interface to untrusted library OSs, which use this interface to implement system functions. Exokernels provide applications with higher performance than traditional microkernel systems. For example, ExOS [10] and JOS [37] provide much more efficient application-level virtual memory and inter-process communication primitives, and SPACE [38] achieves better I/O performance while providing low-level kernel abstractions defined by the trap and architectural interface. Cache Kernel [39] provides a low-level kernel that supports multiple application-level OSs, and thus it can be viewed as a variation of the exokernel. The difference between Cache Kernel and Exokernel is that Cache Kernel mainly focuses on reliability rather than secure multiplexing of physical resources.

With the proliferation of cloud computing and single-purpose VM appliances, Unikernels [4] target the commodity clouds and compile the applications and libraries into a single bootable VM image that runs directly on a VM hypervisor (e.g., Xen). Mirage [4] is an OCaml [40] based unikernel system where the applications are written in high-level type-safe OCaml code and are compile-time statically sealed into VM appliances against modification, which provides better security than previous library OSs like Singularity [41]. Mirage eschews backward compatibility [12] to achieve higher performance, but it is nontrivial to rewrite the numerous legacy services and applications (that are written in C, Java, Python, etc.) using OCaml.

Jitsu [42] uses Unikernel to design a power-efficient and responsive platform for hosting cloud services in the edge network, providing a new Xen toolstack that satisfies the demands of secure multi-tenant isolation on resource-constrained embedded ARM devices.

MiniOS [11] designs and implements a unikernel-style library OS to run as a para-virtualized guest OS within a Xen domain. MiniOS has better backward compatibility and supports some single-process applications written in C. However, the original MiniOS also statically seals a monolithic appliance and suffers problems similar to Mirage.

Libra [43] provides a library OS abstraction for the JVM directly over Xen and relies on a separate Xen domain to provide networking and storage, which is similar to the design of MiniOS.

Library OSs may also run in picoproces ses [12] [15] [13] [14] [16]. The picprocess-based library OSs implement the OS personalities as library functions and map high-level APIs onto a few interfaces to the host OS. Picprocess-based LibOS is more like a traditional OS process and naturally supports dynamic shared libraries. However, currently picprocess-based library OSs run only a handful of custom applications on small research prototypes, mainly because they require much engineering effort to achieve the same security and compatibility as full VMS.

5.3. Security Issues

cKernel VMs are expected to provide network-facing services in the public multi-tenant cloud and run inside full virtual machines above the Xen hypervisor layer, therefore we treat both the hypervisor and the control domain (dom0) as part of the trusted computing base (TCB). Software running in cKernel VMs is under potential threat from both other tenants in the cloud and malicious hosts connected via Internet. The full VM architecture facilitates cKernel to
use the hypervisor as the guard for isolation against internal attacks from other tenants, and the use of secure protocols like SSL and SSH helps cKernel appliances trust external entities.

Some modern OSs apply type-safe languages (e.g., OCaml [4], Haskell [44], and C sharp [41]) to achieve type-safety. For example, Mirage [4] uses OCaml to implement a set of type-safe protocol libraries and provides static type-safety with a single address-space layout that is immutable via a hypervisor extension. But there are still non-type-safe components (e.g., the garbage collector) in Mirage. Clearly, cKernel is written in C and thus is not type-safe. However, the cKernel architecture is orthogonal to the language for implementation and it is possible to implement a “type-safe” cKernel using type-safe languages. Note that to achieve complete type-safety not only the OS and applications but also the hypervisor and dom0 should not allow the type-safe property to be violated.

Another approach to improving security is to reduce the TCB of a VM through disaggregation. For example, Anderson et al. propose to run security sensitive components in separate Xen domains for MiniOS [11], leveraging the fully virtualized processors and the Trusted Computing Group’s trusted platform module (TPM) provided by the hypervisor. Murray et al. implement “trusted virtualization” for Xen and improve the security of virtual TPM by moving the domain builder, which is the most important privileged component, into a minimal trusted compartment [45]. Currently the TCB of cKernel includes the hypervisor, the privileged dom0 and the necessary management tools that is hosted on top of a full Linux OS running dom0, therefore it is worth studying the disaggregation approach to reduce the TCB of cKernel in future work.

6. Future Work

The fusion of VMs and processes provides the cloud a desirable virtualization architecture that embraces the best of both, but a lot of work remains to be done in this promising direction.

First, currently cKernel supports fork but still cannot support IPC APIs like pipe, signal and message queue. So it is urgent to implement the complete IPC abstractions for cKernel. It is also necessary to study the subtle difference between VMs and processes. For example, it is common for a web server to fork a worker process to service a user request, but a worker VM will have a new IP and lose connection to the user. The fork API could also be extended to support the spawn() primitive for fast booting a large-number of VM appliances in OpenStack.

Second, the current cKernel implementation relies on the (unmodified) shared memory mechanism of Xen, whose networking system, file system and threading model are simple and inefficient. This might result in relatively poor performance [46]. For example, the current networking implementation of Xen is insufficient and we could use the ClickOS [46] modification to improve it. It is also desirable to incorporate the MUSL library [47] instead of RedHat Newlib for better runtime performance.

Third, it is important to develop demonstrating applications of the new architecture. For example, it is interesting to build a cloud platform on which multiple large-scale cKernel applications are deployed securely and quickly. It is challenging to build a cKernel pool for fast VM booting, which accommodates sleeping (empty) VMs that can be woken up by dynamically linking to the designated applications as shared libraries. It is also useful to build basic cloud services adopting the fused architecture, such as the large-scale persistent in-memory key-value store [48, 49].

Fourth, currently customers of a cloud must trust the cloud provider with both the hypervisor and the cKernel VMs, while recent advances in hardware protection (Intel SGX [16]) provides opportunity to shield cKernel VMs not only against unforeseen bugs of the compiler, runtime, hypervisor or toolchain, but also against attacks from malicious hosts and operators.

Last, it is also vital to implement the fused virtualization architecture for other VM hypervisors like Linux KVM and VMWare vSphere.

7. Conclusion

In this paper we propose a new virtualization architecture which fuses the abstractions of VMs and processes for the cloud. We present a LibOS based reference implementation (cKernel) by modifying MiniOS on top of Xen, which seals only the application binary into a bootable VM and supports dynamic library linking and conventional process-like fork. The new virtualization architecture has a number of advances. For instance, it (i) allows online library update by leveraging dynamic library mapping, (ii) supports conventional multi-process applications by realizing VM fork, and (iii) improves the flexibility of VMs’ management by emulating processes’ scheduling and operations. Some key components of cKernel are already available as open-source [50].

Acknowledgement

This work was supported in part by the National Basic Research Program of China (973) under Grant No. 2014CB340303, the National Natural Science Foundation of China (NSFC) under Grant No. 61772541, 61379055 and 61379053. This work was performed when the first author was visiting the Computer Lab at University of Cambridge between 2012 and 2013. The authors would like to thank Huiba Li, Ziyang Li, Yuxing Peng, Ling Liu, Jon Crowcroft, the iVCE Group, the NetOS group of Computer Lab at University of Cambridge, and the anonymous reviewers for their insightful comments.

References


[23] "http://www.linux-kvm.org/"


[38] D. Probert, J. Bruno, and M. Karaorman, “Space: A new approach to operating system abstraction,” in


[40] “https://ocaml.org/”


