Portable Hypervisor Design for Commercial 64-Bit Android Devices Supporting 32-Bit Compatible Mode

Kangho Kim(✉), Kwangwon Koh, Seunghyub Jeon, and Sungin Jung

ETRI, Daejeon, South Korea
{khk, kwangwon.koh, shjeon00, sijung}@etri.re.kr

Abstract. We present a hypervisor design that can be applied to any commercial 64-bit Android devices without support of set makers. We achieved the portability by using pure software virtualization while preserving high performance. The contribution of the design is to put the guest OS and the hypervisor together into a single address space which results in avoiding the address space compression problem and reducing major virtualization costs, using 32-bit compatible mode. The design using the single address space makes the hypervisor simple and run fast even with pure software technologies. Prototypical implementation of the design is composed of one kernel module and one user-level program managing virtual machines for Android OS. We have evaluated our design on a commercial mobile phone, Nexus 6P. Since any Android device allows inserting kernel modules and installing user programs on it, we think that our hypervisor can be utilized on any 64-bit ARM-based mobile phones.

Keywords: Hypervisor · Virtual machine · Mobile phone · ARM · KVM

1 Introduction

Carriers and third-party mobile service providers aspire to secure their own space inside mobile phones to execute their core logics as well as to save data only for their own services. They expect that the space must be as independent as possible from the phone makers. They want to achieve their requirement without help of the phone makers. If reserving the space needs to modify the bootloader of phones, they cannot help asking the phone maker to add some functionalities into the bootloader.

The most common way to occupy their own independent space inside a real machine is to create a virtual machine (VM). We are able to put one's own data, service programs, and an OS in the VM and keep them separate from the environment the phone makers provide. We can create VMs with QEMU or KVM that are open source hypervisors freely available. QEMU without help of KVM is a hypervisor fully emulating processor instructions and peripheral devices, so that it may not meet performance requirements. KVM using hardware virtualization extension shows higher performance than QEMU. However, it depends on the bootloader because it expects the bootloader to set the processor to hypervisor mode (EL2) prior to kernel booting.

As far as we know, Samsung Galaxy S6 bootloader sets processor's mode to kernel mode (EL1) and transfers control to the Android kernel. The kernel disables KVM/ARM module after detecting it is running in the kernel mode [1, 2].

We present a portable hypervisor that provides the VM with 32-bit address space and expect its performance is as high as KVM, without cooperation of the bootloader. We assume that core part of both data and service program is not so large that 32-bit address space would be enough to accommodate those data and service in it. The 32-bit restriction help us to achieve efficiency of the hypervisor and keep it small even without using ARM virtualization extension.

2 Hypervisor Design

We also assume that ARM virtualization extension is not available, so that we have no choice but to resort to pure software hypervisor technologies. We prefer to full virtualization, which doesn't require any source code change for the guest OS(OS running inside a VM). The popular design for instruction virtualization is de-privileging; for memory virtualization, is the shadow page table. To keep the hypervisor simple and efficient, we adapted pre-virtualization, which achieves the same performance as para-virtualization, at a little amount of engineering cost [4]. By introducing the single address space design, we will argue that address space compression problem can be avoided, VM exit/entry/switching cost can be minimized, and hypervisor protection is achieved.

2.1 Address Space

Figure 1(a) shows KVM's address space layout in which the VM has independent space different from the hypervisor space. KVM hypervisor shares a single address

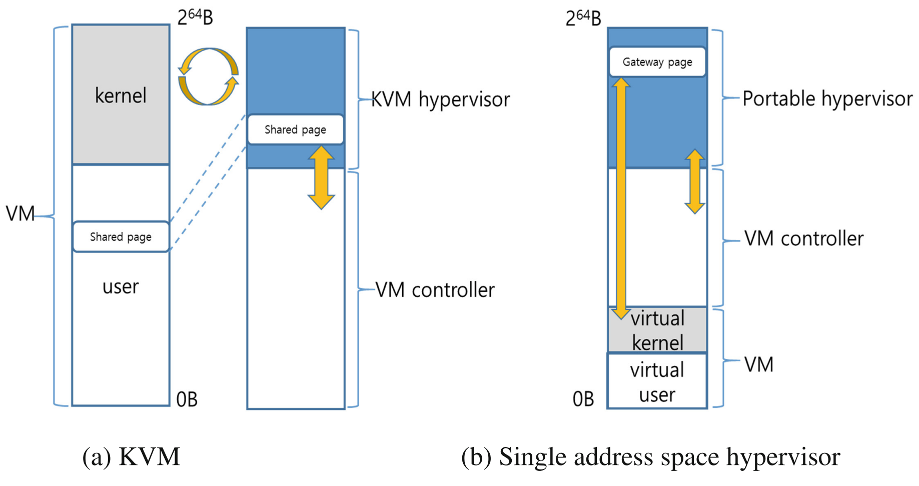

Fig. 1. Address space layout for hypervisors (a) KVM (b) Single address space hypervisor

space with VM controller, where VM controller resides in the user space, and the hypervisor in the kernel space. In this design, every VM-exit/-enter causes address space switching between VM and the hypervisor.

The VM controller is an Android(or Linux) task creating a VM, loading guest OS images inside the VM, patching the sensitive instructions of the guest OS, and starting/stopping/pausing/migrating the guest OS. The hypervisor is a kernel module trapping and emulating the sensitive instructions during VM execution.

Figure 1(b) represents our address space layout that enables to design and implement efficient pure software hypervisor. The unique feature in our design is that the VM address space is restricted to the first 4 GB range (0 ~ 4 GB), the first part of huge 64-bit address space. The VM controller is placed anywhere between above 4G and below hypervisor space and the hypervisor resides in the upper part of the 64-bit address space as KVM does. Since we restricted VM address space to 32-bit space, guest OS must be built for 32-bit ARM architecture target. However, the hypervisor and the VM controller should be 64-bit programs.

This address space layout illustrated in Fig. 1(b) looks similar to that of Xen, which is para-virtualization technology, placing Xen hypervisor at the upper part of the VM address space. The Xen guest OS should understand hypervisor resides somewhere within its virtual address space and modify OS code to reserve the upper space to the hypervisor. Our design is different from Xen's in that it does not require such a modification without scarifying performance.

2.2 Address Space Compression

We adopted pure software hypervisor design that has destined to suffer from address space compression problem if we just follow the conventional design. In that design, the hypervisor must use some amount of guest OS's virtual address space which stores VM context, hypervisor context, address space switching functions, and interrupt vector table in order to control the guest OS [3]. That *control space* is mapped into both the guest OS's address space and the hypervisor's, but it is not visible to the guest OS. KVM/ARM calls the control space as the shared page [2]. The address space compression problem is that the guest OS is not allowed to use the shared page space [3].

Under our design assumption, we have extra address space where guest OSs cannot access as well as recognize intentionally or accidentally, but the processor can do. We can avoid compressing the address space of the guest OS by using that extra address space, which implies that we don't need to penetrate into the guest OS's virtual address space and steal some amount of the address space. We place the control space outside of the guest OS's address space, and call it as *gateway* because the control goes to and comes from the guest OS though the gateway. Even though the gateway is not inside the guest OS, it can intercept exceptions occurring inside the guest OS and give a control to the guest OS.

2.3 Protection

The hypervisor including the VM controller must be protected from the guest OSs to keep an entire system safe. Our design presented in Fig. 1(b) ensures that it does not require any software mechanism to protect the hypervisor from the guest OS. The processor does not allow for the guest OS running in 32-bit compatibility mode to access to the gateway living in higher address than 4 GB because the processor overrides upper 32-bits of the address the guest OS generates. Thus, VM controller and hypervisor living in above 4 GB address can be protected from the guest OS by hardware.

Our design achieves VM isolation, protecting one VM from another VM, by switching only the 32-bit address space. It is like switching the address space of OS tasks to isolate one task from another. The hypervisor only needs to update VM address space of the shadow page table whose hypervisor address space is shared with all the VMs.

While the design confines the guest OS to the 32-bit address space, the processor can access the entire 64-bit address space including interrupt vector table address. When an exception occurs during VM execution, the processor changes its mode to 64-bit mode and moves control to the interrupt vector table entry residing outside of the guest OS in order to handle that exception, by hardware. If the control transfers to the hypervisor, it is allowed to access the VM address space without extra cost because the VM rents lower 32-bit address space that the hypervisor creates and manages.

2.4 VM Exit/Entry/Switch

In our design, no address space change is required but the hypervisor should save and restore VM context and its own context before and after handling the VM exit, and during VM switch since the hypervisor and the guest OS share a single address space. We can reduce the context switching cost due to the 32-bit restriction to the guest OS. That cost includes TLB invalidation, cache invalidation, address space identifier tracking as well as VM context save and restore. Due to 32-bit restriction and the single address space layout, the cost can be minimized and removed respectively.

Our design restricts the execution of the guest OSs to the user-level 32-bit compatibility mode, those OSs and their applications running inside the VM are forced to use user mode AArch32 registers which are subset of AArch64 registers [5]. Even though AArch64 provides 31 general purpose registers(x0 ~ x30), the VM is allowed to use only 16 registers(x0 ~ x15). This restriction makes VM context save/restore cost low. The hypervisor needs to save and restore only w0 ~ w15 general purpose registers for the VM exit and entry respectively. In addition to that benefit, the hypervisor context is not to be saved or restored when switching between the guest OS and the hypervisor because the hypervisor requires that only x19 ~ x29 registers be preserved across the guest OS execution and the guest OS is not allowed to use those registers at hardware level.

3 Prototype Implementation

3.1 Implementation on Nexus 6P

We make the best use of Android task's address space to implement the single address space. We divide 64-bit virtual address space into three parts: low-user(0 ~ 4 GB), high-user(4 GB ~ 512 GB), and kernel(upper 512 GB). We mapped the low-user, the high-user, and the kernel part to the 32-bit guest OS, the VM controller, and the hypervisor, respectively. The gateway will be installed in the kernel part where the hypervisor is responsible for managing.

As mentioned earlier, the portable hypervisor consists of two modules: user-level task (VM controller) and kernel module (Hypervisor). The controller creates a single address space, evacuates the first 4 GB of the address space and fully assigns to a VM. When a user wants to run a guest OS, the controller loads that guest OS's image into VM address space and patches the sensitive instructions. The hypervisor installs the gateway to catch exceptions during VM execution before the original Android kernel catches. To do that, the hypervisor module set VBAR(Vector Base Address Register) to our own vector table in the gateway area before entering VM. When the hypervisor catches an exception, it analyses the cause of the exception and handles the exception accordingly. If the exception belongs to the original Android kernel, the hypervisor delivers it to the Android kernel without any interpretation. Hardware timer interrupt could be that case.

We implemented our design on a Nexus 6P mobile phone. We have done rooting and installed our customized kernel that supports kernel module insertion. We just changed one line of kernel configuration, CONFIG_MODULES = y, for the kernel to support kernel modules. To the best of our knowledge, since most of the kernel supports kernel modules by default, the custom kernel would not be needed to insert our own kernel module.

3.2 Evaluation

We show the proposed hypervisor efficiency by comparing the message passing time between Linux processes through Linux domain socket and the time between Linux process and echo task over uCOS-II as 32-bit guest OS (Fig. 2). All experiments were done on a Nexus 6P(snapdragon 810 v2.1) and the result is as Fig. 3.

As shown in Fig. 3, in spite of the overhead of proposed hypervisor and uCOS-II, the message passing time is shorter in case that the message size is smaller than 1 K. It takes only 500 processor cycles for 1-byte message, which implies that virtualization cost will be minimized and our design has potential to achieve good performance competitive to KVM.

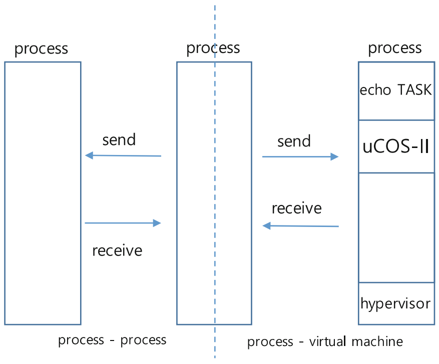

Fig. 2. Experiment environment

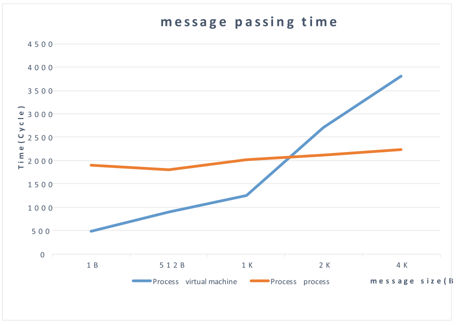

Fig. 3. Comparison of message passing time

4 Conclusion

We introduced the hypervisor design that has portability to any 64-bit Android devices while showing good performance. The main idea in the design is to confine the guest OS to the 32-bit address space. With this idea, we can avoid the address space problem, minimize major virtualization costs (VM exit/entry/switch cost), and achieve to protect the hypervisor from the guest OS.

We believe that the independent 32-bit address space (VM space) is enough to contain core data and service routines that need to be hidden from the main OS. We recommend that most of the service data and routines be placed in the tasks of the main OS and communicate with core routines inside the VM to execute service logics. The single address space design can be used for Intel and AMD processors as well as ARM processors.

Acknowledgments. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. B0101-16-0644, Research on High Performance and Scalable Manycore Operating System).

References

1. Dall, C., Nieh, J.: KVM/ARM: the design and implementation of the linux ARM hypervisor. In: 19-th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 333–347 (2014)
2. Dall, C., Nieh, J.: KVM for ARM. In: 12-th Annual Ottawa Linux Symposium, Ottawa, pp. 45–56 (2010)
3. Uhlig, R., Neiger, G., Rodgers, D., Santoni, A., Martins, F., Anderson, A., Bennett, S., Kägi, A., Leung, F., Smith, L.: Intel virtualization technology. IEEE Comput. Soc. **38**(5), 48–56 (2005)
4. LeVasseur, J., Uhlig, V., Yang, Y., Chapman, M., Chubb, P., Leslie, B., Heiser, G.: Per-virtualization: software layering for virtual machines. In: Computer Systems Architecture Conference, pp. 1–9. IEEE (2008)
5. ARM: ARM Cortex-A Series Programmer's Guide for ARMv8-A version 1.0 (2015)