Prototype of Light-weight Hypervisor for ARM Server Virtualization

Young-Woo Jung, Song-Woo Sok, Gains Zulfa Santoso, Jung-Sub Shin, and Hag-Young Kim
Cloud Computing Research Department, ETRI, Daejeon, Republic of Korea

Abstract - *As ARM CPUs become increasingly common in the server world, virtualization technologies for mobile systems need to be extended for ARM server systems. However, because this system has the limited resources compared to the traditional x86 server system, new virtualization technologies should be considered to allow as many virtual machines as possible to run efficiently and simultaneously on single ARM server system. In this paper, we present the prototype of light-weight hypervisor for ARM server system which can minimize the performance degradation of the guest operating system running on the hypervisor and provide full virtualization. We explore how to achieve the light-weight ARM hypervisor by describing and analyzing its detailed implementation. Through a performance comparison between the native operating system and the guest operating system running on the proposed hypervisor, we show that the proposed ARM hypervisor guarantees minimal virtualization overhead.*

Keywords: ARM server, virtualization, ViMo-S, hypervisor, virtual machine, light-weight

1 Introduction

The number of ARM-based devices has grown tremendously across smart phones, tablets, laptops, and embedded devices. It is because ARM CPUs are more power-efficient than any other CPUs in the market. Nowadays, ARM CPUs also continue to increase performance and some of them is now within the range of x86 CPU performance. This drives the development of ARM-based microservers and pushes ARM CPUs into the traditional server world.

A microserver (also written as micro server or MicroServer) is a small server appliance that Intel introduced the concept around 2010. This inexpensive and energy-efficient server can be squeezed onto a small system board to obtain a blade system which may be smaller than the conventional blade but still powerful enough for data processing. [1] Although Intel has launched microserver products based on Xeon or Atom processors on the market, ARM CPUs have been also considered as another excellent choice, because ARM based SoCs have a better performance to build servers and clusters than x86 and Atom processors, especially considering their performance per Watt relation. [2]

On the other hand, virtualization has been adopted as an important key technology in the x86 server systems for many years and is now spreading to microservers. With ARM beginning to enter the server world, virtualization support is very critical and ARM CPUs of the ARMv7-A [3] and ARMv8-A [4] architectures now include hardware support for virtualization, ARM virtualization extensions, that lets multiple virtualized OSes run efficiently and simultaneously.

The current major hypervisor (also known as virtual machine monitor) technologies using the hardware virtualization extensions of ARM seem to be KVM/ARM [5] and Xen on ARM [6]. However, KVM and Xen was the original purpose of virtualizing x86 server systems, so both basic structures have been optimized in x86 architecture, not in ARM architecture. In the KVM/ARM approach which supports a full virtualization, the host operating system (OS) runs directly on top of the hardware, in which the hypervisor is implemented as a kernel module, and then the guest OSes run as processes on top of the host kernel. Although this kernel component of KVM is included in mainline Linux, KVM/ARM must leverage QEMU [7] in user space to virtualize I/O devices and QEMU is a rather heavy program to be installed in ARM server system which has restricted resources. [8] Xen on ARM has been used as one of leading para virtualization technologies for ARM-based devices. With ARM providing virtualization extensions, Xen on ARM has supported hardware virtual machine (HVM), rather than paravirtual machine (PV). Although Xen is a very mature virtualization technology, it has a very complex configuration which is not easy for common user and it needs to modify the guest OS which means its compatibility and portability is poor.

In this paper, we present the prototype of light-weight hypervisor for ARM server virtualization with ARM virtualization extensions, which support full virtualization and minimize the performance degradation of the guest OSes. On the beginning stage of the design and implementation, we focused only on the ARM architecture and have optimized it.

This paper is organized as follows. Firstly, we give the brief explanation about ARM virtualization extensions, which is main technology to make the hypervisor more efficient and light-weight. The detailed architecture of the proposed hypervisor is introduced in Section 3, where we describe how to virtualize each resource such as CPU, memory, interrupt, and I/O devices. Section 4 briefly shows the experimental results including the performance comparison between the

native OS and the OS running on the proposed hypervisor and the porting to the prototype of ARM server system. Finally, we conclude this paper and suggest some future works.

2 ARM Virtualization Extensions

Similar to x86 architecture, ARM virtualization extensions enable the efficient implementation of the hypervisor for ARM compliant processors to the latest ARMv7-A and ARMv8-A architectures. For example, the ARM Cortex-A15 [9] is a core of ARMv7-A architecture and ARM Cortex-A53/A57 [10, 11] are cores of ARMv8-A architecture, respectively. In this section, we describe a brief overview of ARM virtualization extensions

2.1 New privilege level for hypervisor

As shown in Figure 1, ARMv7-A architecture includes a new CPU mode called Hyp mode as well as TrustZone [12] as Security Extensions. TrustZone splits the modes into two worlds, secure and non-secure. A special mode, Monitor mode, is provided to switch between the secure and non-secure worlds. According to the typical booting sequences in ARMv7-A, ARM CPUs power up by reset starting in ARM secure SVC mode, execute boot and startup codes, and then transition to ARM secure Monitor mode, by which ARM non-secure Hyp mode for the hypervisor can be activated on.

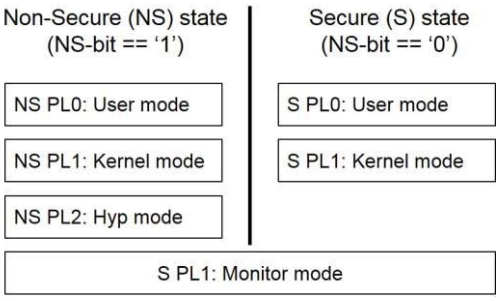

Figure 1. ARMv7-A processor modes

Hyp mode was introduced as trap-and-emulation mechanism to support virtualization in the non-secure world. It is more privileged than the existing non-secure kernel modes, the kernel and user modes, and leaves the guest OSes and applications unmodified. It has its own banked registers, as well as additional registers, such as SP, SPSR, and ELR, in which most of critical feature of hardware-assistant CPU virtualization is executed. Using this register set, the hypervisor software running in Hyp mode can configure hardware to trap into Hyp mode on several sensitive instructions and hardware interrupts.

2.2 Stage-2 translation

Virtualization requires that the guest OS cannot access to the hypervisor's memory space. Without virtualization extensions, a technique, for example, shadow page table which is maintained by the hypervisor, are enforced. In this technique, the guest OS kernel maintains its own page tables but the hypervisor should keep this OS kernel from setting the Memory Management Unit (MMU) registers. This approach makes the hypervisor complicated and causes performance overhead.

In ARM virtualization extensions, ARM provides hardware support to virtualize physical memory, two-stage memory address translation. When a virtual machine (VM) runs, the physical addresses managed by the VM are actually Intermediate Physical Addresses (IPAs) (also known as guest physical addresses) which are translated into physical addresses (PAs) (also known as host physical addresses). For the memory address translation in the guest OS, the stage-1 page tables using the translation table base register (TTBR) translate the virtual addresses (VAs) into IPAs, then stage-2 page tables using the virtual translation table base register (VTTBR) translates IPAs into PAs. This stage-2 translation can be enabled and disabled in Hyp mode.

2.3 Virtual interrupts

ARM defines the Generic Interrupt Controller (GIC) architecture. The GIC routes interrupts from devices to CPUs and CPUs discover the source of an interrupt through the GIC interfaces. The GIC architecture consists of two parts, the distributor and the CPU interfaces. There is only one distributer in a system, and each CPU core has a GIC CPU interface. The distributer is used to configure the GIC, for example, to configure the mapping of an interrupt to the CPU core, and the CPU interfaces are used to signal acknowledgment (ACK) and End-Of-Interrupt (EOI) to the corresponded interrupts.

If all interrupts are configured to be handled by the hypervisor, the hypervisor should generate virtual interrupts in software to signal them to VMs. This causes the interrupt processing in the hypervisor to be expensive, because even ACKs and EOIs for all virtual interrupts must be processed in the hypervisor. The next version of GIC introduced the concept of virtual interrupts which is supported by new hardware virtualization feature, virtual GIC (VGIC), which includes the virtual distributer and the virtual CPU interface. The virtual CPU interface can be mapped into the guest OS as the CPU interface, and can be used by the guest OS to signal ACKs and EOIs without trapping into the hypervisor, reducing overhead for manipulating interrupts on a CPU. The hypervisor generates virtual interrupts by writing to special registers in the virtual distributor, the list registers, and the virtual CPU interface signals these virtual interrupts directly to the guest OS's kernel mode. Nevertheless, the hypervisor must still emulate the distributor and all accesses by a guest OS will be trapped into the hypervisor.

2.4 Generic timer

ARM generic timer architecture provides virtualization support for physical timer resources by introducing virtual timers and virtual counters that measure the passing of virtual time, that is, the passing of time on a particular VM. While the hypervisor is configured to use the physical timer, the VM can be configured to use the virtual timer VMs can control their own virtual timers without any trap to the hypervisor. However, any access to the physical timer and counter by a VM arises trap to Hyp mode, in which the hypervisor only can control them.

3 Light-weight Hypervisor Architecture

The proposed hypervisor, ViMo-S, targets the virtualization of the ARM server system, which means it should be light-weight enough to provide the reasonable performance in the restricted resource environment. Originally, ViMo [13] was implemented by ETRI for mobile ARM processor, which doesn't support hardware virtualization extensions, so we expanded it for ARM server system with virtualization extensions. ViMo-S supports VM lifecycle management such as dynamic creation and destruction of VMs, which may be mandatory in the server virtualization. Another important feature of ViMo-S is to support the full virtualization for which ViMo-S completely virtualizes the physical hardware without any modification of the guest OS codes.

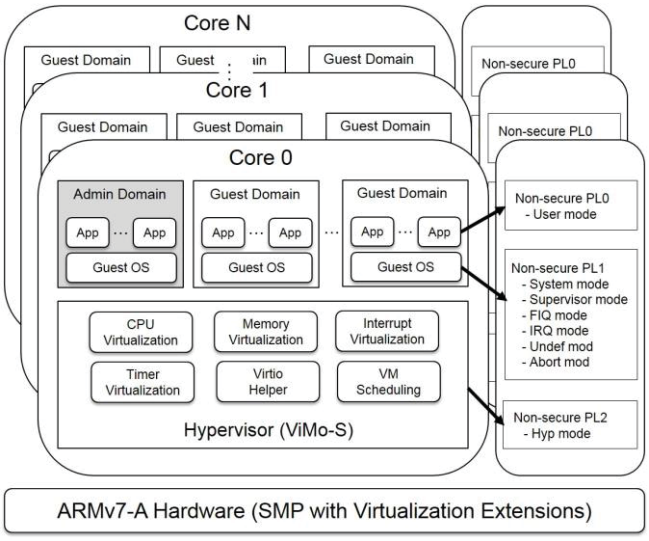

Figure 2. ViMo-S system architecture

As shown in Figure 2, ViMo-S runs in Hyp mode, with supporting the virtualization of CPU, memory, interrupt, and timer resources. It also supports Virtio-based I/O virtualization for full virtualization and can schedule multiple VMs at the same time. There are two kinds of domains running in user mode and kernel mode. While an admin domain knows the existence of the hypervisor and has interfaces to ViMo-S through the hyper calls for VM management and Virtio-based I/O operations, there is no interface to ViMo-S in the guest domains, so that they run in the same manner as the execution on the physical hardware.

In the following sections, we will describe in detail each core virtualization technology of ViMo-S.

3.1 CPU virtualization

To virtualize the CPU, ViMo-S must ensure that the guest OS running in the VM has the same access to the registers as the OS running on the physical CPU, while the hardware state controlled by the hypervisor is persistent across running VMs. With ARM virtualization extensions, a VM running in the kernel and user mode has same register state as register state without the hypervisor, and ViMo-S running in Hyp mode saves/restores the current VM context in/from the Hyp stack when a VM switches to ViMo-S and vice versa. ViMo-S configures all accesses to the other sensitive states such as WFI/WFE instructions, stage-2 page faults, and hyper calls for VM management and I/O virtualization, to be trapped and emulated in ViMo-S. Because trap-and-emulation may be expensive enough to affect VM performance, ViMo-S reduces the frequency of traps by leveraging ARM hardware virtualization support.

ARM boot loaders typically transition to the non-secure world at an early stage, which means there is no ways to switch on Hyp mode in which ViMo-S will execute, because Hyp mode can be activated only in secure Monitor mode. What we need to do is to trap into Hyp mode before uboot boots the guest kernel. In order to turn Hyp mode on, we used secure software, e.g. boot loader, running on secure state in which Monitor mode can activate Hyp mod. When uboot jumps to the entry point of ViMo-S in Monitor mode, ViMo-S performs the following actions in CPU core 0 to enable the hypervisor: (1) enable hyper call and disable secure monitor call, FIQ, IRQ and Abort of Monitor mode, (2) turn Hyp mode on and transition to Hyp mode, (3) activate other cores for SMP, (4) configure the exception vector table in Hyp mode, (5) set up the page table for the hypervisor and enable MMU by setting the page table base register of Hyp mode (HTTBR), (6) activate the hypervisor, (7) configure the distributor register (GICD), the CPU interface register (GICC), and the virtual interface control register (GICH), which are accessible only by the hypervisor, (8) configure the hypervisor timer (also known as Hyp timer) of the generic timer, then the hypervisor can receive the hypervisor timer interrupt, and (9) wait for creating a VM.

For the other CPU cores other than CPU core 0, after transitioning Hyp mode and configuring the exception vector table in Hyp mode, they wait for the event indicating that the page table setup for the hypervisor is complete in the CPU core 0, because all CPU cores share the page table for the hypervisor which is created by CPU core 0. And then, they

perform the same actions as in the CPU core 0, only except for configuration of GICD, because there is only one distributor in a system.

When ViMo-S creates a new VM by the request of the admin domain through the hyper call, it performs the following actions: (1) create and initialize the structure of the VM, which contains virtual CPU (VCPU) context, Hyp stack, MMU context including VTTBR, VGIC, virtual timer, vector floating point (VFP), and other necessary values for the VM, (2) allocate memory to the VM by unit of 2MB and configure the page table including I/O memory mapping for stage-2 translation, (4) activate the VM which is now schedulable. When the VM is scheduled, ViMo-S configures VTTBR and VTCR to enable stage-2 translation for the VM, and returns to the VM through ERET instruction, which performs the mode change to SVC mode and the program counter (PC) change at the same time.

After the VM is created, it runs in PL0 and PL1, and is trapped into ViMo-S only for the timer interrupts, the I/O interrupts, and the specified hyper calls by a VM of the admin domain.

3.2 Memory virtualization

ViMo-S supports memory virtualization of stage-2 translation in order that a VM cannot access physical memory belonging to ViMo-S or other VMs. ViMo-S controls all physical memory accesses and allows a VM only to access the memory regions allocated to it. If a VM tries to access the other memory regions, it causes stage-2 page fault and traps into ViMo-S. Actually we use this kind of stage-2 page fault mechanism for Virtio-based I/O virtualization, which will be explained in section 3.4. Since stage-2 page tables can be configured only in Hyp mode, they are completely transparent to each VM. When ViMo-S performs context-switching to the VM, it enables stage-2 translation and configures the stage-2 page table base register, VTTBR, of the VM. On the other hand, when switching back to ViMo-S, ViMo-S disables stage-2 translation and translates VA directly into PA by using HTTBR. After configuring stage-2 translation and the page table base registers, all memory translations are performed by the hardware without any intervention by ViMo-S, which gives better performance to VMs.

3.3 Interrupt virtualization

ViMo-S configures the CPU to trap all hardware interrupts to Hyp mode by setting GICD register, which enables the hypervisor to control hardware resources. While Hyp timer interrupt is processed only in ViMo-S for VM scheduling and maintenance, VMs must receive notification for other interrupts in the form of virtual interrupts for emulating devices. ViMo-S uses the VGIC to inject these virtual interrupts to VMs and reduce the number of traps to Hyp mode. Virtual interrupts are raised to VCPUs by configuring the list registers of the virtual distributors through GICH and VMs can access to the virtual CPU interfaces without being trapped to Hyp mode.

ViMo-S minimizes the execution of the interrupt hander in Hyp mode, because long execution in the hypervisor can affect the performance of the VM. For Hyp timer interrupt, it saves only the banked registers into the Hyp stack, executes its own jobs, and then transition to the VM. In the case of VM context switching, it additionally save the other contexts in the structure of the VM, such as MMU context, VGIC, virtual timer, and so on, which was explained in section 3.1. For the other interrupts, ViMo-S just injects the corresponded virtual interrupts to the VM using VGIC.

3.4 Virtio-based I/O virtualization

For the support of full virtualization, ViMo-S utilizes Virtio [14], the de-facto standard for I/O virtualization, to provide virtual devices in the guest domain. As described in Figure 3, we provide three key components for Virtio-based I/O virtualization; Virtio front-end, Virtio back-end, and Virtio helper.

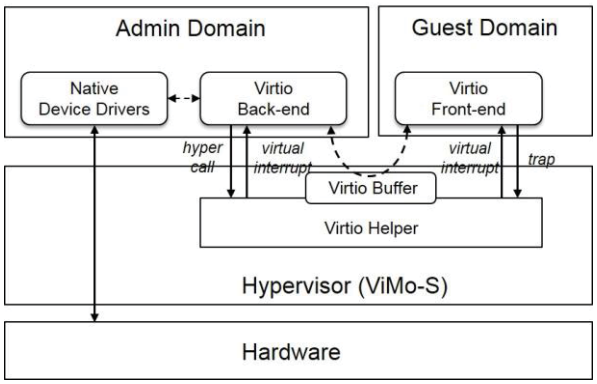

Figure 3. Virtio-based I/O virtualization

The Virtio front-end is located in the guest domain and considered as a normal device driver. There is no distinguishable difference between the Virtio front-end device drivers and other physical device drivers. Although the implementation of the Virtio front-end driver depends on the running OS of the guest domain, these drivers have been already included in Windows or Linux, and we just used it in the guest domain without any changes. That means ViMo-S can support the full virtualization without changing the codes of the guest OS. Through the configuration of the Virtio front-end driver, the access to the memory area for the transport via the memory-mapped I/O arises trap into ViMo-S, which can emulate I/O operations.

To serve I/O requests of the Virtio front-end, the Virtio back-end driver is necessary in the admin domain. The Virtio back-end driver leverages the already implemented virtualization programs in the OS running on the admin

domain. As an example, in Linux, the Virtio back-end driver uses TUN/TAP device to virtualize network devices for the guest domains. The Virtio back-end doesn't depend on the guest OS. A Virtio back-end driver can virtualize devices for multiple guest domains. The actions that can be done by the Virtio back-end may be limited due to the reasons of security and stability. For instance, the Virtio back-end cannot directly access to the memory area of the guest domain for reading and writing data. It means a module running on the hypervisor should support such operations. ViMo-S provides the Virtio helper which takes requests from the Virtio back-end as well as the Virtio front-end and processes them as necessary.

During the guest domain boots, the Virtio front-end asks the Virtio helper for the type, features and status of the Virtio device. After the Virtio helper gets the information from the Virtio back-end and sends it to the Virtio front-end, the Virtio front-end initializes the device accordingly. The initialization includes the buffer creation. The information about the size and location of the buffer is transferred to the Virtio helper which then map this buffer to the memory region in the admin domain, where the Virtio back-end driver can process the data directly in this buffer. By this way of conduct, we can be sure that all the virtual devices in the guest domain work under the capabilities of the admin domain's devices. As an example, the virtual block device in the guest domain can support SCSI device, as long as the admin domain supports it. Otherwise, the SCSI support in the guest domain is turned off.

For the Virtio operations, the Virtio front-end writes data into its buffer and the Virtio back-end reads the buffer and acts properly according to the request and the type of the device. As for the detailed explanation of the Virtio operations, please refer to [14]. There are several mechanisms to interact between each component for Virtio operations. The Virtio front-end interfaces with the Virtio helper through memory trapping. The access to the memory-mapped I/O region traps into ViMo-S where the Virtio helper handles it. As for the Virtio back-end, it interfaces with the Virtio helper through the hypervisor calls. The Virtio helper kicks both the Virtio front-end and the Virtio back-end by injecting virtual interrupts. The Virtio front-end and back-end drivers should register handlers to manage virtual interrupts sent by the Virtio helper. While the virtual interrupts injected to the Virtio front-end depend on which virtual devices are used, we use the virtual interrupt number 155 to be injected to the Virtio back-end, because 155 is not used by ARM v7-A architecture.

3.5 VM scheduling

Currently, ViMo-S provides the simple Round-Robin (RR) VM scheduler, which maintains its own VM queue on a CPU core. When a VM is created, it allocates a time quantum and the VM scheduler makes context-switching according to the specified time quantum. In order to schedule VMs in ViMo-S, ViMo-S always uses Hyp timer of ARM generic timer. In each core, the VM scheduler decreases the time quantum of a running VM whenever it receives the Hyp timer interrupt, and makes the context-switching of VMs when the remaining time quantum of the running VM is 0.

When the current VM traps into ViMo-S by the Hyp timer interrupt, the Hyp timer interrupt handler in ViMo-S saves the current VM contexts, such as all general purpose registers, R13 register of USR mode, and the banked registers of SVC, ABT, UND, IRQ, and FIQ mode, into the Hyp stack. In case of VM switching, ViMo-S additionally saves the MMU control registers including VTTBR, VGIC-related registers, virtual timer control registers, VFP registers, and fault state registers, into the data structure of the current VM. And, then it restores the saved registers and values from the Hyp stack and the saved data structure of the next VM, and traps into the next VM.

4 Experimental Results

In this section, we present some experimental results that can measure the performance of ViMo-S on ARM multicore hardware, and show how many VMs can be provided in an ARM server system. We evaluated the virtualization overhead of ViMo-S compared to native execution by running the AIM benchmark suite [15] within both a VM and directly on the hardware. The results provide the real measurements of the performance of ViMo-S with ARM hardware virtualization support. Moreover, we show the construction of a 32-bit ARM server system to run many VMs simultaneously on top of ViMo-S.

4.1 Methodology and measurement

For ViMo-S measurement, we used an Insignal Arndale board [16] with a dual core 1.7GHz Cortex-A15 CPU on a Samsung Exynos 5250 SoC, which has been the most widely used and commercially available development board supporting ARM virtualization extensions and multicore. It supports onboard 100Mb Ethernet, 2Gbyte memory, eMMC 4.5, SDIO 3.0, and SD 2.0.

We used the mainline Linux 3.8 kernel for our experiments, with several patches on top of the source tree. Although an OS running on ViMo-S was slightly modified to provide the hyper calls, we kept the software environments of both platforms as the same as possible to provide comparable measurements. Our focus was not on measuring absolute performance of ViMo-S, but rather the relative performance degradation between virtualized and native execution of OS. As the AIM benchmark suite, we used Re-AIM7 [17] open source software which is a rework of the AIM benchmark suite for the Linux community. Although it benchmarks several workloads such as CPU, disk, file server, database, and so on, we focused only on the measurement for CPU/disk-intensive workloads, because these workloads may be the important criteria to evaluate the virtualization environment.

As shown in Figure 4, two Arndale boards are connected to the desktop, using the serial ports. While one is for evaluating the performance of the native OS, the other one is for evaluating the performance of the guest OS running on ViMo-S. We continually executed the Re-AIM7 benchmark programs in both OSes by configuring the *-g* flag in order to increase the number of users up to 10. For each number of users, the Re-AIM7 runs until the maximum Job/Minute (JPM) is reached. For the CPU and disk performance, we repeated this benchmark in both OSes up to 10 times, and produced the average of the results.

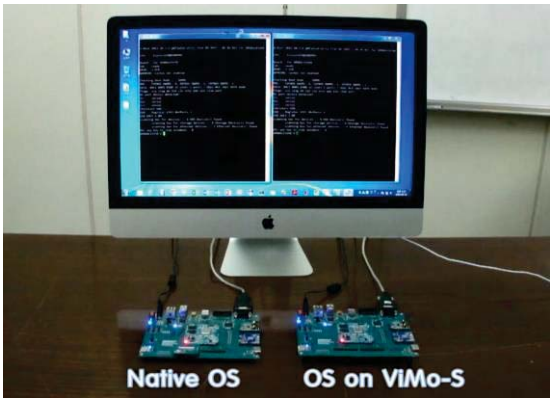

Figure 4. Test environment: native OS vs. OS on ViMo-S

For CPU-intensive workloads and disk-intensive workload, Figure 5 and 6 show respectively normalized performance for running application workloads in the VM versus running directly on multicore. The horizontal axis is essentially the number of simultaneous jobs (workloads), and vertical axis is the overall rate at which the jobs complete (throughput). Figure 5 shows that ViMo-S has minimum virtualization overhead across CPU-intensive workloads, despite the performance degradation of the maximum of 4.5% and the average of 2%. Although Figure 6 shows the substantial differences in virtualization overhead compared to CPU-intensive workload, the overhead by ViMo-S is less than 4%. In this evaluation, we found that ARM virtualization extensions significantly reduce complexity of ViMo-S and are also likely to reduce virtualization overhead.

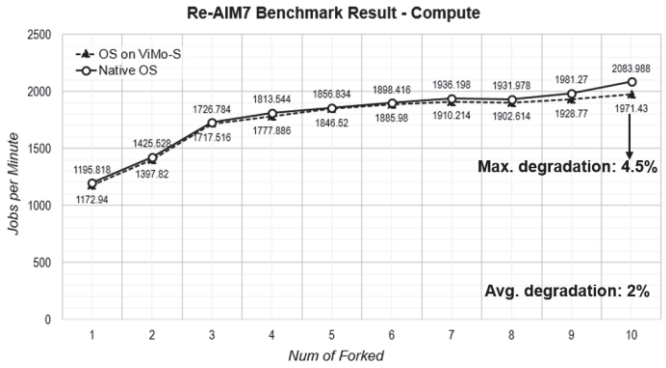

Figure 5. Re-AIM7 benchmark result - compute

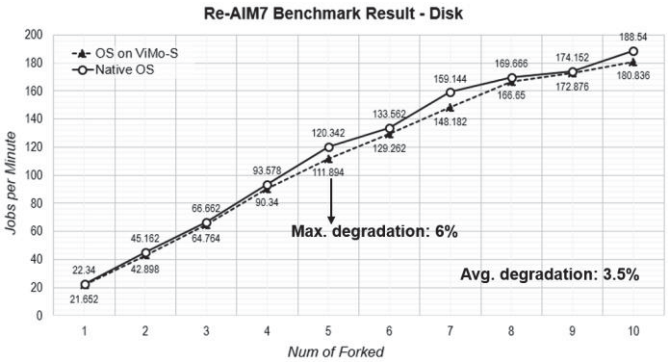

Figure 6. Re-AIM7 benchmark result - disk

4.2 Prototype of ARM server system

To evaluate ViMo-S as ARM server virtualization, we developed the reference platform of 32-bit ARM server system as shown in Figure 7, which consists of eight computing nodes, HDD pool, and power supplier. The system configuration may be very simple, but it is sufficient, because each ARM computing board provides CPU, memory, Gigabit Ethernet, USB 3.0, and so on. A computing board supports the two CPU sockets based on Samsung Exynos 5250 SoC, which is same as the CPU of the Insignal Arndale board.

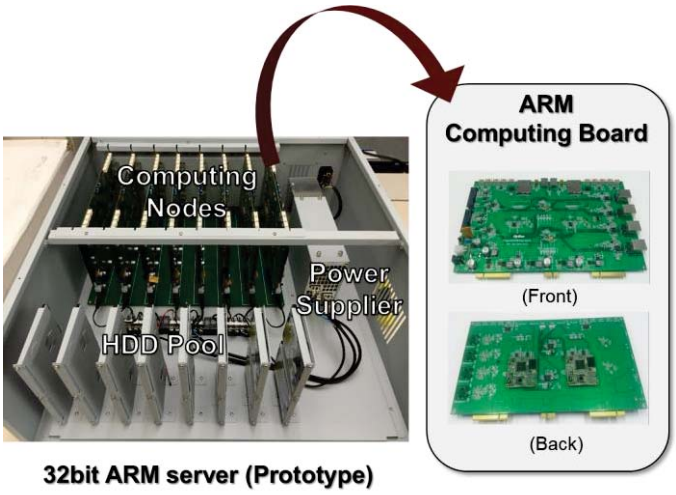

Figure 7. Reference platform of 32-bit ARM server system

In this single system, we generated and ran 64 VMs simultaneously, including the admin domains and the guest domains. The normal execution of each VM can be verified through the remote SSH connection to the SSH daemon running on each VM.

5 Conclusion and Future Works

In this paper, we presented the prototype of the light-weight hypervisor, ViMo-S, which targets the virtualization of the ARM server system. With the benefits from ARM hardware virtualization extensions, ViMo-S can minimize the

virtualization overhead on ARM multicore hardware. We also presented in detail how to use ARM hardware virtualization extensions to virtualize the hardware resources, such as CPU, memory, and interrupt. ViMo-S supports the full virtualization by using Virtio, the de-facto standard for I/O virtualization. Our experimental results show that ViMo-S incurs minimal performance impact and has modest virtualization overhead, within 4% of direct native execution on multicore hardware for CPU-intensive and disk-intensive workloads.

For the future works, we need to improve the performance of the Virtio-based I/O virtualization, because all I/O requests from the guest domains can be centralized into the admin domain. And then, we will expand ViMo-S to 64-bt ARM server system, for which we consider the X-Gene [18] development board with an octa core 2.4 GHz Cortex-A5x CPU on an Applied Micro APM883208 SoC, and the development of the reference platform for 64-bit ARM server.

6 Acknowledgements

This work was supported by the ICT R&D program of MSIP/IITP [R0101-15-237, Development of General-Purpose OS and Virtualization Technology to Reduce 30% of Energy for High-density Servers based on Low-power Processors].

7 References

[1] Hsieh, Wen-Hsu, et al. "Energy-saving cloud computing platform based on micro-embedded system." *2014 16th International Conference on Advanced Communication Technology (ICACT)*, 2014.

[2] Aroca, Rafael Vidal, and Luiz Marcos Garcia Gonçalves. "Towards green data centers: A comparison of x86 and ARM architectures power efficiency." *Journal of Parallel and Distributed Computing* 72.12 (2012): 1770-1780.

[3] ARM Ltd. ARM Architecture Reference Manual ARMv7-A DDI0406C.b, July 2012.

[4] ARM Ltd. ARM Architecture Reference Manual ARMv8-A DDI0487A.a, Sept. 2013.

[5] Dall, Christoffer, and Jason Nieh. "KVM/ARM: The design and implementation of the linux arm hypervisor." *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. ACM, 2014.

[6] Xenproject.org. ARM Hypervisor - Xen. http://www.xenproject.org/developers/teams/arm-hypervisor.html.

[7] Bellard, Fabrice. "QEMU open source processor emulator." *URL: http://www. qemu. org* (2007).

[8] Minnich, Ronald G., and Don W. Rudish. "Ten Million and One Penguins, or, Lessons Learned from booting millions of virtual machines on HPC systems." *Workshop on System-level Virtualization for High Performance Computing in conjunction with EuroSys*. Vol. 10. 2009.

[9] ARM Ltd. ARM Cortex-A15 Technical Reference Manual ARM DDI 0438C, Sept. 2011.

[10] ARM Ltd. ARM Cortex-A53 MPCore Processor Technical Reference Manual ARM DDI 0500D, Feb. 2014.

[11] ARM Ltd. ARM Cortex-A57 MPCore Processor Technical Reference Manual ARM DDI 0488C, Dec. 2013.

[12] Alves, Tiago, and Don Felton. "TrustZone: Integrated hardware and software security." *ARM white paper* 3.4 (2004): 18-24.

[13] Oh, Soo-Cheol, et al. "ViMo (virtualization for mobile): a virtual machine monitor supporting full virtualization for ARM mobile systems." *CLOUD COMPUTING 2010, The First International Conference on Cloud Computing, GRIDs, and Virtualization*. 2010.

[14] Russell, Rusty. "virtio: towards a de-facto standard for virtual I/O devices." *ACM SIGOPS Operating Systems Review* 42.5 (2008): 95-103.

[15] Wikipedia website. http://en.wikipedia.org/wiki/AIM_Multiuser_Benchmark.

[16] InSignal Co. ArndaleBoard.org. http://www.arndaleboard.org.

[17] Sourceforge.net. Re-AIM_7. http://re-aim-7.sourceforge.net.

[18] Applied Micro Circuits Co. X-Gene. https://www.apm.com/products/data-center/x-gene-family/x-gene/