ViMo (Virtualization for Mobile) : A Virtual Machine Monitor Supporting Full Virtualization For ARM Mobile Systems

Soo-Cheol Oh, KangHo Kim, KwangWon Koh, and Chang-Won Ahn
Electronics and Telecommunications Research Institute
Daejeon, South Korea
{ponylife, khk, Kwangwon.koh, ahn}@etri.re.kr

Abstract—[*]**This paper proposes ViMo that is a micro virtual machine monitor for ARM mobile systems, which enables to run multiple OSes on single mobile system at the same time. ViMo does not require any modification or compilation of OS, so that time and cost developing a virtualized mobile system can be reduced. Isolation of each virtual machine is another major feature of ViMo and it prevents the other virtual machines from the malfunctions of contaminated virtual machine.**

Keywords-Full Virtualization; Virtual Machine Monitor; ARM; Mobile System; Isolation;

I. Introduction

Virtualization technology of server systems has been used widely due to advantages of increased resource utilization, power saving, space saving for servers and others. The virtualization technology is about to be used in mobile and embedded systems. Mobile system virtualization has advantages of security, reduction of mobile phone BOM (Bill Of Materials) and legacy software utilization.

The virtualization technologies can be categorized into full virtualization [1] and paravirtualization [1]. In the full virtualization, the instruction modifying system status is detected in runtime and emulated by a virtual machine monitor. This technology has the advantage that it is not necessary to modify source code of a guest OS. But it has the overhead of scanning and emulating the source code and this makes system performance lower. For the paravirtualization, the source code of the guest OS is scanned offline and the instruction modifying the system status is replaced with hypercall to the virtual machine monitor or a sequence of codes having same semantic. The advantage of the paravirtualization is higher performance than the full virtualization. But it has the problem that the source code of the guest OS must be modified offline by humans or tools.

In the server systems, the full virtualization has been used popularly because server system CPUs with 3GHz frequency and multi-core provide enough performance. Also hardware virtualization technologies such as Intel-VT [2] and AMD-V [3] accelerate the full virtualization. Representative systems supporting the full virtualization are VMware [4], Parallels [5], Virtualbox of Oracle [6] and KVM [7]. Representative paravirtualized virtual machine monitor is Xen [8].

A main technology for mobile system virtualization is the paravirtualization. The largest obstacle against the mobile system virtualization is weaker performance than the server systems. The latest CPU of the server systems has 3 GHz frequency and multi-core and this provides enough performance with the virtualization environment. However, the mobile systems have generally 500 ~ 800Mhz CPU with single core and doesn't provide enough performance with the virtualization systems. This limitation makes main virtualization trend of the mobile systems as the paravirtualization. But, the latest mobile systems including iPhone 4G of Apple and Galaxy S of Samsung are based on 1GHz ARM processor. Also the smart phones including HTC Desire, HTC HD2 with 1GHz Qualcomm SnapDragon are being delivered to the users. Thus the performance issue that is the main obstacle of the mobile system virtualization is being solved. Also, it is scheduled that new ARM core will support a hardware virtualization, and mobile systems with multi core will appear soon. Consequently, fundamentals for the full virtualization in the mobile systems will be concrete.

The paravirtualized mobile systems are MVP of VMware [9], VLX of VirtualLogix [10] and XenARM [11]. The full virtualizated mobile systems are QEMU [12].

This paper presents ViMo (Virtualization for Mobile) that is a virtual machine monitor based on the full virtualization for the mobile systems. ViMo scans binary code of OS in runtime and emulates critical instructions. Also ViMo allocates physical memory space to each virtual machine and provides memory isolation among the virtual machines.

II. ViMo Architecture

ViMo is the virtual machine monitor supporting the full virtualization for ARM-based mobile systems and the structure of ViMo is shown in Fig. 1. ViMo works on system hardware and multiple OSes are mounted on ViMo. ViMo creates one virtual machine per each OS and the OS runs on the corresponding virtual machine.

In typical systems, OS and application are located in supervisor (SVC) and user (USR) mode of ARM CPU, respectively. However, the OS in the ARM's SVC mode must be moved to the ARM's USR mode because ViMo is inserted into the ARM's SVC mode. In the virtualized systems based on ViMo, the USR mode of ARM, in which the OS is executed, is called logically as Virtual SVC

[*] This work was supported by the MKE [KI002088, Development of Virtualization Technology based on Open Source Software]

(VSVC) and the USR mode, in which applications are executed, is called Virtual USR (VUSR). The four modes used in ViMo are summarized as Table I.

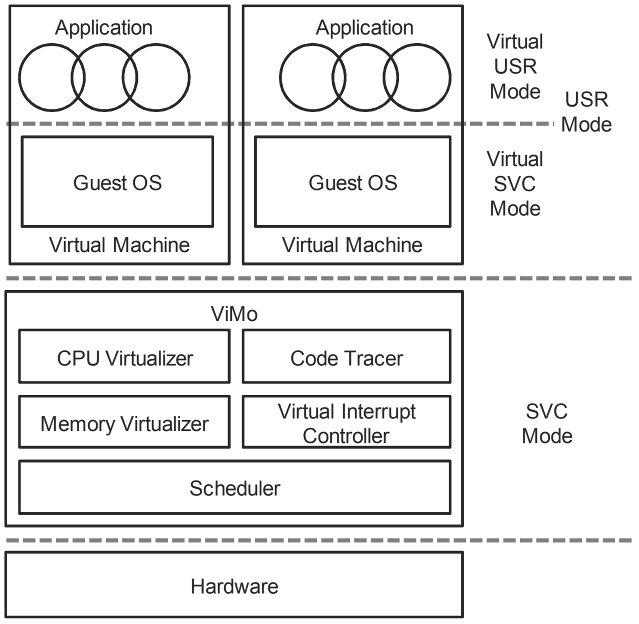


Figure 1. ViMo Architecture

TABLE I. CPU MODE OF VIMO

Mode	Descriptions
ASVC	(ARM SVC) SVC mode of ARM Hardware
AUSR	(ARM USR) USR mode of ARM Hardware
VSVC	Mode in that OS is executed in the ViMo-based virtualization system
VUSR	Mode in that applications are executed in the ViMo-based virtualization system

ViMo has mainly five components that are code tracer, CPU virtualizer, memory virtualizer, interrupt controller virtualizer and scheduler. The code tracer detects critical instructions on the virtual machines in runtime and the CPU virtualizer emulates the detected critical instructions. The memory virtualizer allocates memory space to the virtual machines and isolates the virtual machine from other virtual macines. The interrupt controller virtualizer builds a virtual interrupt controller for each virtual machine and processes interrupt controller accesses from the virtual machine. The scheduler switches the virtual machines periodically.

III. MODE TRANSITIONS

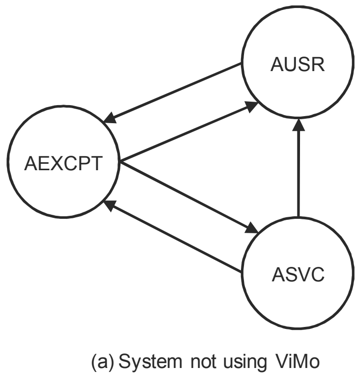
Typical systems not using ViMo have three modes. Two modes of them are ASVC and AUSR explained in table I and other mode is AEXCPT handling exceptions. Exception modes including interrupt (irq), fast interrupt (fiq), prefetch abort, data abort and SWI (software interrupt) can be explained as AEXCPT. As shown in Fig. 2-(a), the transition from ASVC running OS' kernel to AUSR running applications is performed directly. If AUSR wants to use kernel service using system calls, AUSR must be transited to ASVC through AEXCPT. Also ASVC or AUSR are transited to AEXCPT if exceptions happen, and after exception serving is completed, AEXCPT is changed directly to ASVC or AUSR.

In ViMo, VSVC, VUSR and VEXCPT are added instead AUSR as shown in Fig. 2-(b). The guest OS is executed in the virtual modes including VSVC, VUSR and VEXCPT which are located in AUSR. ViMo is executed in ASVC and AEXCPT.

The transitions among VSVC, VUSR and VEXCPT cannot be performed directly because these modes are located in AUSR. Thus, these mode transitions must be made by ASVC and AEXCPT.

The mode is transited directly to AEXCPT if an exception happens when CPU is in VSVC, VUSR or VEXCPT. In this situation, there are two processing options. If the exception is for the guest OS, this exception is passed to the guest OS and mode is transited from AEXCPT to VEXCPT. If the exception is for ViMo, the mode is transited to ASVC.

In ASVC, the transition to AEXCEPT is disabled. ViMo processes virtual machine management jobs including guest OS scheduling, memory management and critical instruction emulation, and these jobs have atomic characteristic that they must be processed without break. To preserve this characteristic, exception generation in ASVC is prohibited.



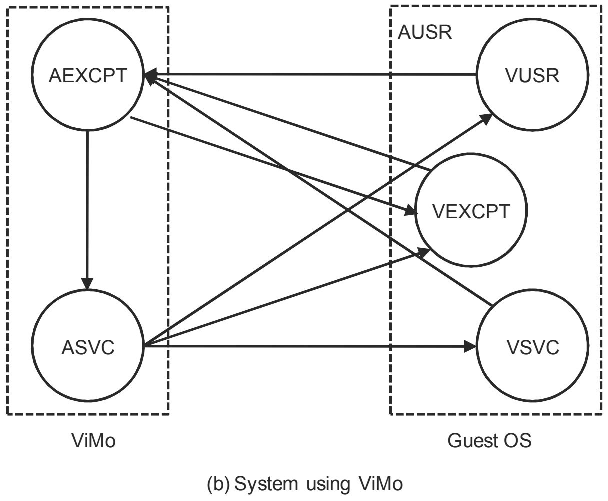


Figure 2. Mode Transition of ViMo

IV. CPU VIRTUALIZATION

This section explains CPU virtualization of ViMo. In ViMo, the guest OS is executed in VSVC and VUSR that belong to AUSR. VUSR doesn't make problems because both of VUSR and AUSR are user mode. However, the guest OS kernel running in VSVC doesn't generate the same result as ASVC because VSVC is located in AUSR. This problem must be solved by ViMo.

There are 148 instructions in ARM architecture v6 [13]. In these instructions, instructions related to the virtualization can be categorized as privileged, sensitive and critical instruction. The privileged instruction must be executed only in the privileged mode (ASVC). If this instruction is executed in AUSR, an exception is generated and control is taken by the privileged mode. The sensitive instruction generates different results when the instruction is executed in ASVC or AUSR. The critical instruction is the sensitive instruction but not the privileged instruction.

To virtualize a system using the full virtualization, the instructions that modify the system status must be executed under control of ViMo or be replaced with a sequence of instruction having same semantic. The instructions modifying system status are the privileged and the critical instructions. If the privileged instruction is executed in VSVC (AUSR mode), an exception is generated and control is taken by ASVC running ViMo. Thus, detection of the privileged instruction is performed automatically by ARM CPU.

If the critical instruction is executed in VSVC (AUSR mode), CPU makes no error and generates the result for AUSR mode that is different from intended result for ASVC mode. Thus, this is one of the most important problems to be solved in ARM CPU virtualization.

Table II shows instruction categorization for ARM CPU. There are 6 privileged instructions in ARM architecture with 148 instructions. MCR and MRC that belong to the privileged instructions have over 100 operand combinations. In detail, MCR and MRC have both characteristics of the privileged and sensitive instructions. Some operand combinations have the privileged instruction characteristic and other operand combinations have the sensitive instruction characteristic. Consequently, each operand combination of MCR and MRC must be handled as single instruction.

TABLE II. PRIVILEGED AND CRITICAL INSTRUCTIONS

Instruction	# of Inst.	Descriptions
Privileged instruction	6	CDP, LDC, MCR, MCRR, MRC, MRRC
Critical instruction-1	14	CPS, LDM(2), LDM(3), LDRBT, LDRT, MRS, MSR, RFE, SETEND, SRS, STC, STM(2), STRBT, STRT
Critical instruction-2	13	ADC, ADD, AND BIC, EOR, MOV, MVM, ORR, RSB, RSC, SBC, SUB, LDR

The critical instruction consists of the critical instruction-1 and the critical instruction-2. The critical instruction-2 can be critical instruction upon operands. Generally, the critical instruction-2 is not the privileged or the critical instruction. However, if the critical instruction-2 has S-bit option and PC as the destination operand, this instruction becomes the critical instruction. This operand combination tells that SPSR (Saved Processor Status Register) of current mode is copied to CPSR (Current PSR). However, if this operand combination is executed in AUSR, the instruction result is unpredictable because AUSR mode doesn't have SPSR.

Fig. 3 shows structure of the CPU virtualization. Basic Block Identifier (BBI) scans a binary image in runtime and identifies basic block. A basic block is the code block with single entry and single exit point. The exit point may be branch or the critical instruction. Suppose that a basic block begins at address 10 and a critical instruction is at address 20.

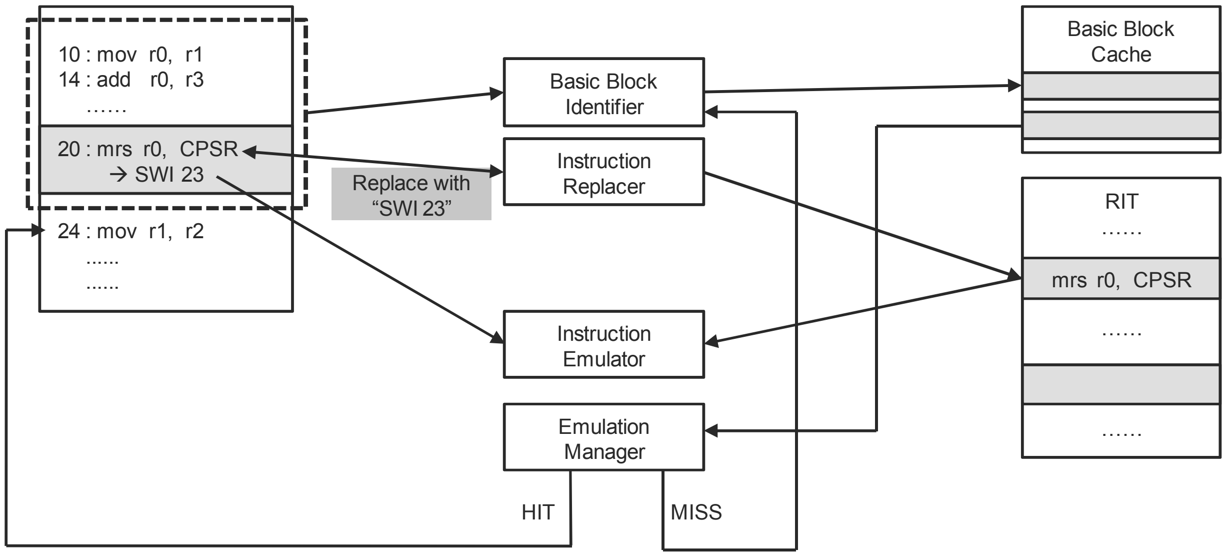


Figure 3. CPU Virtualization Architecture

BBI begins to scan the code at address 10. When BBI meets the critical instruction at address 20, it makes a basic block and stores this block at basic block cache. The basic block cache stores the basic block that is already scanned. When the basic block stored at the basic block cache is executed again, it is not necessary to scan the block.

Then BBI calls instruction replacer. The instruction replacer stores the critical instruction at Replacement Instruction Table (RIT) and replaces the critical instruction with single SWI instruction having index to RIT. For example, the critical instruction is replaced with "swi 23" if the instruction is stored at 23th index of RIT.

The instruction replacer sets PC register as the entry point (address 10 in Fig. 3) of the basic block and executes the basic block. Then codes at address 10 to 1C are executed and the SWI instruction at address 20 generates an exception that delivers control to ViMo located in ASVC. This sequence was already explained in Fig. 2-(b). ViMo calls instruction emulator. The instruction emulator retrieves index to the RIT from the SWI instruction, looks up the RIT and loads the original instruction (mrs r0, CPSR) that is replaced with the SWI instruction. Then, the original instruction is emulated by the instruction emulator. After completion of the instruction emulation, ViMo calls BBI and continues to find next basic block beginning at address 24. BBI skips to find the basic block if the basic block is cached in the basic block cache. The sequence explained above is repeated until the corresponding virtual machine is shutdown.

V. Memory Virtualization

ViMo virtualizes main memory on system and provides memory space with each virtual machine. ViMo also isolates virtual machine from each other by preventing accesses to memory of other virtual machines without permission of ViMo.

The memory virtualization architecture of ViMo is shown in Fig. 4. For example, suppose that a system has a main memory with 128MB located at address 0x50000000. In ViMo, address space is categorized as three address spaces that are virtual address, physical address and machine address. The virtual address space is used by the guest OS and the physical address space is recognized as real physical address by the guest OS. The machine address space is maintained by ViMo.

ViMo allocates a machine address space to each virtual machine and this memory space is contiguous physically. This address space is not changed until the corresponding virtual machine is shutdown.

Suppose that ViMo allocates 32MB region of the machine address space at 0x51000000 ~ 0x52FFFFFF to the guest OS 0. In this situation, ViMo provides the illusion, that this address is located at 0x50000000 ~ 0x51FFFFFF, with the guest OS 0. The guest OS 0 maps the physical address to virtual address and uses this address space.

The same memory allocation is applied to the guest OS 1. ViMo allocates the machine address space at 0x53000000 ~ 0x54FFFFFF to the guest OS 1 and the guest OS 1 thinks that it uses the physical address space at 0x50000000 ~ 0x51FFFFFF.

ViMo uses shadow page table (SPT) [14] for the memory virtualization. The guest OS has its own guest page table maintaining the memory mapping from the virtual address to the physical address. ViMo knows memory mapping from the physical address of each virtual machine to the machine address. Thus, ViMo can create the memory mapping from the virtual address of each virtual machine to the machine address and this memory mapping is maintained by SPT.

The guest page table is modified continuously by the guest OS. ViMo should detect modification of the guest page table and update SPT dynamically to reflect this modification. To solve this problem, ViMo modifies an attribute of the memory region that stores the guest page table from read-write to read-only. Memory write to the guest page table by the guest OS generates an exception and control is taken by ViMo. Thus ViMo can capture the page table write and maintain SPT.

VI. Interrupt Controller Virtualization

Fig. 5 shows virtualization architecture of interrupt controller in ViMo. ViMo builds a Virtual Interrupt Controller (VIC) based on HW Interrupt Controller (HWIC) and provides it to the guest OS. ViMo makes one VIC per

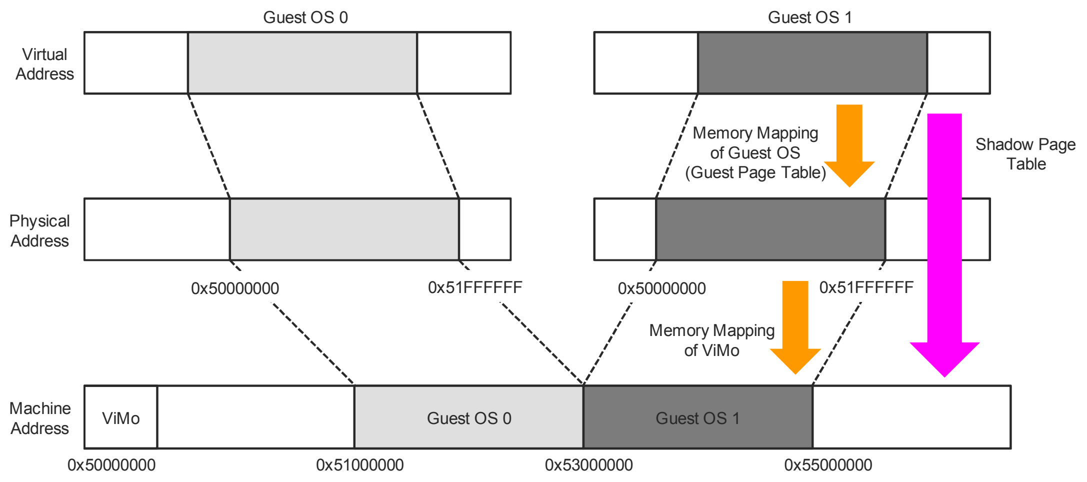


Figure 4. Memory Virtualization Architecture

each guest OS. Thus, the guest OS uses VIC instead of HWIC.

The guest OS controls HWIC by accessing registers in HWIC that are mapped to address space of CPU. Thus it is necessary to virtualize the registers for HWIC virtualization. VIC has same register configuration as HWIC. The access controller of VIC handles access from the guest OS and maintains consistency between VIC and HWIC.

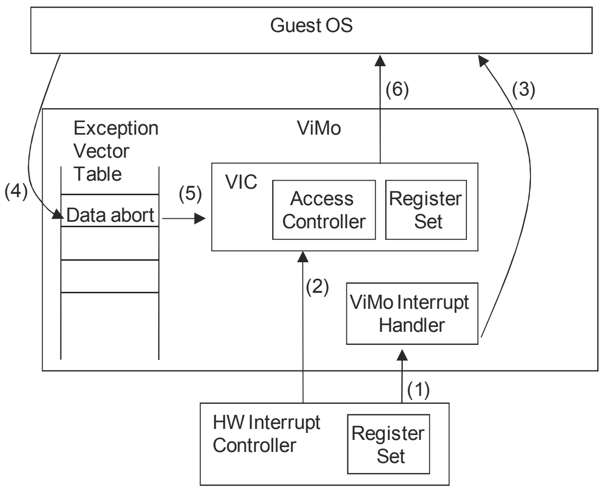


Figure 5. Interrupt Handling

If a HW interrupt is generated, ViMo interrupt handler is activated (Fig. 5-(1)) and copies interrupt information from HWIC to VIC (Fig. 5-(2)). In this situation, the ViMo interrupt handler checks interrupt mask of VIC and doesn't copy the interrupt that is masked on VIC. Then, ViMo calls interrupt handler of the guest OS (Fig. 5-(3)).

The interrupt handler of the guest OS tries to access HWIC for getting what kind of interrupt is generated. When the guest OS accesses HWIC, ViMo intercepts this access (Fig. 5-(4)). HWIC has some internal registers that are mapped to address space of CPU and the guest OS tries to access HWIC using this registers. ViMo modifies memory mapping between the guest OS and HWIC, so that the guest OS has no memory mapping for HWIC. In this case, the access to HWIC by the guest OS generates a data abort exception. If the data abort exception is generated and the exception handler of ViMo confirms that this exception is for HWIC, the access controller of VIC is activated (Fig. 5-(5)). The access controller processes the access request using VIC and maintains the consistency between HWIC and the VIC. After processing the access from the guest OS, control is returned to the guest OS that calls HWIC access (Fig. 5-(6)).

VII. SCHEDULER

Scheduler of ViMo switches virtual machines periodically. The scheduler stores status of virtual machine executed during previous time quantum at main memory and loads status of next virtual machine. The used scheduling algorithm is round-robin.

VIII. IMPLEMENTATION

ViMo proposed in the paper was implemented on an ARM11-6410SYS board developed by Huins[15] that is a embedded development board based on ARM11. The CPU is Samsung S3C6410 using ARM1176JZF-S core and frequency is 533MHz. The board has 128MB DDR SDRAM, 128MB NAND flash and 1MB NOR flash. The Board also has 4.3 inch wide color TFT LCD with 480x272 resolution. We executed Linux with 2.6.21 kernel and uC/OS-II as the guest OS. Binary code size of ViMo is 34KB that is very small.

IX. EXPERIMENTAL RESULTS

The experiments were performed with three cases that are RawLinux, ViMo-Single and ViMo-Dual. RawLinux is a case not using ViMo. ViMo-Single and ViMo-Dual are ViMo systems with single guest OS and two guest OSes, respectively.

A. *DhryStone*

We measured performance of ViMo using DhryStone [16]. DhryStone is the benchmark measuring the CPU performance developed by Dr. Reinhold P. Weicker. The number of run in DhryStone is 10,000,000.

RawLinux, ViMo-Single and ViMo-Dual show 432, 271, and 150 VAX MIPS, respectively. We know that ViMo-Single is 37% slower than RawLinux and this overhead is from ViMo. ViMo-Dual is 44% slower than ViMo-Single because ViMo-Dual has two guest OSes.

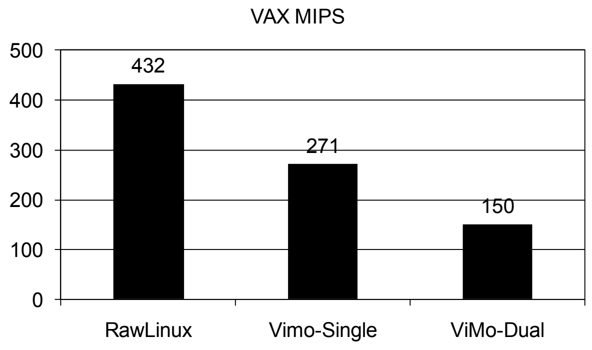


Figure 6. DhryStone Benchmark

Figure 7. Moive Play Capture

B. *Play Movie*

This experiment is to play a movie in ViMo-Single. The profile of the movie is 480x272 resolution and 30 frame/s. By using ViMo, the total play time of ViMo-Single increases compared to RawLinux. The play time of the movie file is 30 seconds but the play time in ViMo-Single increases to 40 seconds. Thus, ViMo produces 33% overhead.

C. *Virtual Machine Isolation*

ViMo provides the isolation of the virtual machines and the experiment shows that a malfunction from one virtual machine is not propagated to other virtual machines. In ViMo-Dual, movie play is executed in the guest OS 0 and a kernel module generating kernel panic is loaded in the guest OS 1. The kernel modules generates the kernel panic by writing memory region that doesn't belong to the guest OS 1. Fig. 8 shows that the guest OS 1 is stopped by the kernel panic. However, the movie play in the guest OS 0 still works without errors.

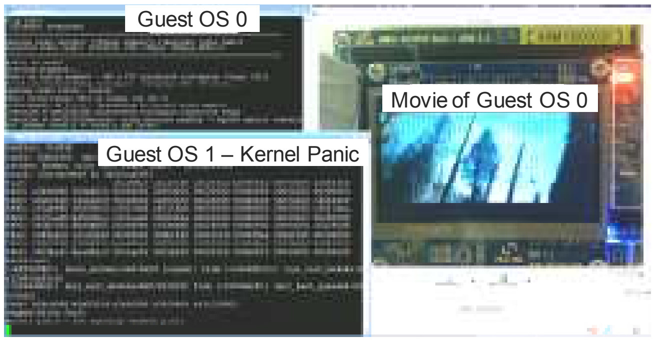


Figure 8. Virtual Machine Isolation

D. *Performance Analysis*

The experimental results of section A and B shows that ViMo is not acceptable for using real mobile systems because of about 33 ~ 37% overhead. The implementation presented in this paper is in its initial state and ViMo has many performance improvement points.

We estimate that the most time consuming part of ViMo is the critical instruction detection, the instruction emulation and related context switches between the guest OS and ViMo. ViMo generates many context switches that are from the critical instruction emulation, the basic block identification, the virtual interrupt controller accessing and the guest OS switching. These frequent context switches between the guest OS and ViMo make cache of CPU flushed and this makes the performance degradation.

We are improving the critical instruction detection and the instruction emulation algorithm to minimize the context switches between the guest OS and ViMo. We are also improving other components of ViMo including the memory virtualization and the interrupt virtualization. Futhermore, we are developing ViMo for ARM Cortex-A8.

Our aim is that the overhead becomes below 10% through these improvement works.

X. Conclusions and Future Works

This paper proposed ViMo that is the virtual machine monitor using the full virtualization for mobile systems based on ARM architecture. ViMo provides the virtualization for CPU, memory and interrupt controller. The binary image is scanned in runtime and the critical instructions are replaced with SWI to ViMo. The replaced instructions are emulated in runtime. Also ViMo presents the memory virtualization and each virtual machine has its own memory space provided by ViMo. A virtual machine cannot access the memory space of other virtual machines without permission of ViMo and no fault of one virtual machine is propagated to other virtual machines. VIC virtualize the HW interrupt controller and each virtual machine has its own VIC.

According to the experimental results, ViMo has 33 ~ 37% overhead. We are improving all components including the critical instruction detection, the instruction emulation and the memory virtualization algorithm. Our aim is that the overhead becomes below 10% through these improvement works. Additionally, we are under designing I/O virtualization

References

[1] "Understanding Full Virtualization, Paravirtualization, and Hardware Assist", White Paper, pp.4-5, VMware, 2007.

[2] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, Rich Uhlig, "Intel Virtualization Technoloyg: Hardware Support for Efficient Processor Virtualization", Intel Technology Journal, Vol. 10, Issue 03, pp. 170-175, Auguest, 2006.

[3] "AMD Virtualization Technoloyg", http://sites.amd.com/us/business/it-solutions/virtualization/Pages/amd-v.aspx, AMD, 2010.

[4] "Building the Virtualized Enterprise with VMware Infrastructure", White Paper, pp. 4-5, VMware, 2008.

[5] http://www.parallels.com, 2010.

[6] Virtualbox "Virtualbox Architecture", http://www.virtualbox.org/wiki/VirtualBox_architecture, Oracle, 2010

[7] AMIT SHAH, "Kernel-based Virtualization with KVM", Linux Magazine, Issue 86, p.37-39, Jan, 2008.

[8] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, lan Pratt, Andrew Warfield, "Xen and the Art of Virtualization", Proceedings of the nineteenthACM Symposium on Operating Systems Principles, pp. 164-177, 2003.

[9] "VMware MVP", http://www.vmware.com/products/mobile/index.html, VMware, 2010.

[10] "Meeting the Challenges of Connected Device Design", White Paper, VirtualLogix, pp. 8-10, 2006.

[11] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, "Xen on ARM: System Virtualization using Xen Hypervisor for ARM-based Secure Mobile Phones", IEEE 5th Consumer Communications and Networking Conference, pp. 257-261, 2008.

[12] Fabrice Bellard, "QEMU, a Fast and Portable Dynamic Translator", USENIX 2005 Annual Technical Conference, pp. 41-45, 2005.

[13] "ARM Architecture Reference Manual", ARM, pp. 152-435, 2005.

[14] James E.Smith, Ravi Nair, "Virtual Machines, Versatile Platforms for Systems and Processes", Morgan Kaufmann Publishers, p.399-402, 2005.

[15] http://www.huins.com, HUINS, 2010.

[16] Alan R. Weiss, "Dhrystone Benchmark, History, Analysis, Scores and Recommendations", White paper, November, pp. 1-5, 2002.