International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 3, Number 1, pages 153–169, January 2013

Analysis of the ARM Architecture's Ability to Support a Virtual Machine Monitor through a Simple Implementation

Akihiro Suzuki Shuichi Oikawa
Department of Computer Science
University of Tsukuba
Tsukuba, Japan

Received: February 9, 2012
Revised: April 20, 2012
Revised: September 25, 2012
Accepted: October 15, 2012
Communicated by Shuichi Ichikawa

Abstract

Since embedded systems are used for various purposes at a variety of places, their security, reliability, and availability are important concerns. Virtualization technology has been used for a long time in order to improve the security of systems, and the other advanced features recently become available also to improve the reliability and/or availability. In order to apply virtualization technology to systems, the detailed analysis of target processor architectures is required. Although such analysis was performed for x86 architecture, it is not available for the ARM architecture, which is the most widely used processor architecture for embedded systems. This paper focuses on such analysis of the ARM processor architecture. We also present the implementation details of a virtual machine monitor (VMM) based on the analysis. We implemented a VMM, called SIVARM, for the ARM architecture to perform and verify the analysis of its ability to support a VMM. We successfully booted Linux on SIVARM and performed the evaluation by executing several benchmark programs on the ARM 1136JF-S processor. It verifies the analysis of the sensitive instructions and also enables the analysis of the performance impact of the virtualization.

Keywords: Virtual Machine Monitor, ARM Processor Architecture.

1 Introduction

Virtualization technology [10] has been used for a long time in order to improve the security of systems by executing an operating system (OS) on a virtual machine monitor (VMM) [9, 15]. It becomes much more common nowadays as virtualization software becomes a commodity on x86 based systems [12]. VMMs can provide more advanced features, such as logging and replay [5], live migration [6], redundant execution [20], and so on, in order to improve the reliability and/or availability of systems.

Since embedded systems are used for various purposes at a variety of places, their security, reliability, and availability are important concerns. In order to apply virtualization technology to

[0]Akihiro Suzuki is currently with Toshiba. Work conducted when he was with University of Tsukuba.

improve them, the detailed analysis of target processor architectures is required. Although such analysis was performed for x86 architecture [21], it is not available for the ARM architecture [23], which is the most widely used processor architecture for embedded systems. While there have been research projects to develop VMMs on the ARM embedded processor [4, 7, 8, 13, 17], all of them failed to provide enough and verified analysis performed on real hardware. Therefore, we decided to perform the analysis by developing our own VMM on the ARM processor that can execute Linux as its guest OS on real hardware.

This paper focuses on such analysis of the ARM processor architecture. We also present the implementation details of a VMM based on the analysis. We implemented a VMM for the ARM architecture to perform and verify the analysis of its ability to support a VMM. We call our VMM SIVARM, a simple VMM for the ARM architecture, because it is based on the trap and emulation model to virtualize the ARM processor. Since the trap and emulation model requires the exact analysis of a target processor architecture, the successful execution of Linux on SIVARM verifies that our analysis is correct. We successfully booted Linux on SIVARM and performed the evaluation by executing several benchmark programs on the ARM 1136JF-S processor. It verifies the analysis of the sensitive instructions and also enables the analysis of the performance impact of the virtualization. SIVARM provides only a single virtual machine (VM); thus, it can accommodate only a single guest OS. Although such limitation makes SIVARM unsuitable for practical use, it is very much enough for the analysis and can rather clarify the performance impact because of its simple configuration.

The rest of this paper is organized as follows. Section 2 describes the related work. Section 3 describes the basics of the ARM architecture, and Section 4 describes its sensitive instructions. Section 5 describes the design and implementation of SIVARM, and Section 6 describes the experiment and evaluation. Section 7 discusses the issues related to the implementation and limitation of SIVARM. Finally, Section 8 concludes this paper.

2 Related Work

There have been a few efforts to port Xen [2] to the ARM architecture [8, 13]. These efforts take the paravirtualization approach [22]; thus, they do not require the precise analysis of the ARM architecture to realize a VMM on it. While [8] ported Xen to the ARM architecture, it could not complete the porting until it boots Linux on it. It could merely boot Mini-OS, which is a tiny OS accompanied with Xen for the testing purpose, on it. Xen ARM [13] completed the porting of Linux, but it required a lot of the modifications to the Linux kernel. It modified approximately 4500 lines of the Linux kernel source code. Such a large number of modifications make Xen ARM difficult to keep up with the version up. In contrast to that, our implementation modified only 81 lines at 48 places that are far less than Xen ARM, and the modifications are straightforward; thus, it is easy to apply the modifications to the latest kernel source code.

The work to build Choices Hypervisor on the ARM architecture [4] classified the sensitive instructions of the ARM architecture since they need to be emulated in the hypervisor layer. It used the QEMU processor emulator [3] to execute the whole system including the hypervisor and its guest OS. It modified QEMU to cause a trap when any of the sensitive instructions is executed. Such modifications change the processor behavior and make it possible to easily catch the execution of the sensitive instructions; thus, the work does not have to analyze the behavior of each sensitive instruction, and did not classify them into the privileged and non-privileged sensitive instructions. There is another ARM hypervisor [17] that took a similar approach to Choices Hypervisor. This hypervisor interprets the ARM instructions on the fly and translates the sensitive instructions to the trap instructions; thus, there is no need to distinguish between the privileged and non-privileged sensitive instructions, either. The classification of the sensitive instructions is required to realize a VMM using the trap and emulation approach, and this paper classifies them in detail. Moreover, no experiment results are described in both of the above efforts; thus, they could not verify the correctness of their classifications of the sensitive instructions. We successfully booted Linux on our VMM and could execute benchmark programs on real hardware; thus, our classification was verified by the actual execution of Linux.

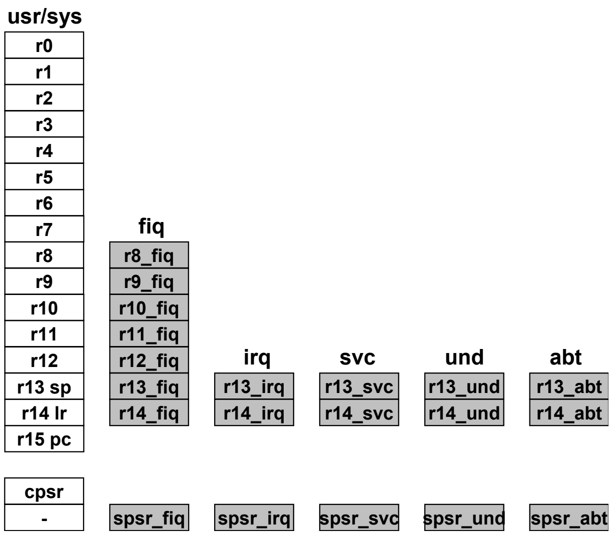

Figure 1: The registers of ARM architecture

KVM/ARM [7] brings a KVM-based virtualization solution to the ARM architecture. It works only on the Android emulator. The execution of the emulator significantly limits the capability to analyze the ability to support a VMM. There are at least the following two limitations; 1) the classification of the sensitive instructions into the privileged and non-privileged ones is not verified since there is no guarantee that the emulator behaves exactly the same as real hardware especially for the sensitive instructions, and 2) the execution of the emulator makes it impossible to analyze the performance impact. We also worked on the analysis of the sensitive instructions and the development of our VMM on the emulator, and reported the results [25, 26] earlier than KVM/ARM. We then moved our work on real hardware. Since we completed our work on real hardware, there are no limitations that the efforts on the emulator imply. We verified the correctness of the sensitive instructions classification on real hardware, analyzed the performance impact of the virtualization, and brought a solution to mitigate such impact.

There are several commercial solutions to virtualize the ARM architecture, such as VLX for ARM by VirtualLogix [27], OKL4 Microvisor by OK Labs [19], MVP by VMware [28], and INTEGRITY secure virtualization by Green Hills [11]. While these solutions are proprietary and their details are not available, there is a statement that all of them require paravirtualization [7].

As described above, there have been a number of efforts to realize the virtualization platform of the ARM architecture. Except for the commercial proprietary solutions, Xen ARM [13] is the only successful effort that works on real hardware, and all of them including Xen ARM and the commercial solutions require paravirtualization. Therefore, our work has the significance by presenting the analysis result of the ARM architecture's ability to support a VMM through a simple trap and emulate implementation on real hardware.

3 ARM Architecture

This section summarizes the basics of the ARM architecture, which are necessary to analyze the ability to realize a VMM on it.

3.1 Processor Modes and Registers

The processor modes of the ARM architecture are composed of the privileged mode (FIQ, IRQ, SVC, ABT, UND and SYS) and the non-privileged mode (USR). General application programs are executed in USR. The 5 modes of the privileged mode except for SYS are the exception modes.

Table 1: The exception types and their corresponding processor modes and high vector addresses

Exception type	Mode	Vector address
Reset	Supervisor	0xFFFF0000
Undefined instructions	Undefined	0xFFFF0004
Software interrupt (SWI)	Supervisor	0xFFFF0008
Prefetch Abort (instruction fetch memory abort)	Abort	0xFFFF000C
Data Abort (data access memory abort)	Abort	0xFFFF0010
IRQ (interrupt)	IRQ	0xFFFF0018
FIQ (fast interrupt)	FIQ	0xFFFF001C

When an exception is caused, the processor mode changes to the corresponding exception mode automatically.

The ARM architecture defines 37 registers as depicted in Figure 1. They are composed of the 31 general purpose registers and the 6 program status registers (PSRs). While R0 - R12 registers are used for general use, their usage is defined by ARM Thumb Procedure Call Standard (ATPCS) [24] as follows. R0 - R3 and R12 are used as the general scratch registers. R13 is the stack pointer (SP). R14 is the link register (LR). R15 is the program counter (PC). The current program status register (CPSR) holds the current PSR, and the saved program status resister (SPSR) holds the value of the CPSR saved when an exception is caused. The PSR contains condition the code flags, interrupt disable bits, and processor mode bits. The ARM architecture monitors and controls the internal behavior of a processor by using the CPSR.

Each of the exception modes utilizes several banked registers. When the processor mode changes to an exception mode, the corresponding banked registers for that mode are used. Linux executes its kernel in SVC and user processes in USR. When an exception is caused, the corresponding exception mode is used in the stub code of the exception handler.

3.2 Exception

The ARM architecture defines the 7 exception types. When an exception is caused, the execution jumps to the static address that corresponds to the caused exception type. The static address is called an exception vector. There are two locations where the exception vectors are placed. One starts at 0x0, and the other starts at 0xFFFF0000. Those at the first and the latter locations are called the normal and high vector address, respectively. Table 1 shows the exception types supported by the ARM architecture, and their corresponding processor modes and high vector addresses.

When an exception is caused, the execution jumps to the corresponding exception vector in the corresponding exception mode. At the time, the CPSR is automatically saved in the SPSR of the exception mode, and interrupts are disabled. When the exception handling is finished, the value of SPSR is restored to the CPSR, and the execution returns to the address saved in the LR.

The following describes how Linux uses the exception mechanism. Since there are only 4 bytes allocated for each vector address, only a single instruction can be placed for each vector; thus, Linux places a branch instruction at an exception vector in order to jump to the address of the stub of the exception handler because Linux needs to handle the different exception types. There is a stub for each exception type, but the stubs of different exception types are basically the same. The following shows a stub code:

```
1   vector_\name:
2     .if \correction
3     sub   lr, lr, #\correction
4     .endif
5     stmia sp, {r0, lr}
6     mrs   lr, spsr
7     str   lr, [sp, #8]
8     mrs   r0, cpsr
```

```
 9    eor   r0, r0, #(\mode ^ SVC_MODE)
10    msr   spsr_cxsf, r0
11    and   lr, lr, #0x0f
12    mov   r0, sp
13    ldr   lr, [pc, lr, lsl #2]
14    movs  pc, lr
```

The exception handler to be invoked is decided based on the processor mode when the exception is caused. The previous processor mode is saved in the SPSR; thus, the stub reads the SPSR at Line 6. At Line 13, the value of the SPSR is used to choose which exception handler is to be invoked. At Line 14, the chosen exception handler is then invoked. At this time, the processor mode becomes SVC. This is done by setting the value based on the CSPR at Line 8 - 10. The value is calculated by changing the processor mode bits to SVC. Since the MOVS instruction at Line 14 is with the S suffix, the SPSR set at Line 10 is moved to the CPSR while the execution jumps to the exception handler decided at Line 13.

3.3 Access Control

The ARM architecture defines a two-stage access control using a domain and an access permission (AP) bit. The domain is a mechanism that divides a virtual memory space and performs the access control of divided regions. By using the domain, a part of a shared virtual memory space can be isolated from the others. The domain access control register (DACR) controls the access to domains. The DACR stores a domain type for each domain. There are the three domain types, no access, client, and manager. The no access type denies any access to the domain. In contrast, the manager type allows all access to the domain. The client type bypasses the access control by a domain. Instead, the access permission (AP) bit of a page table entry is used for the access control.

Using the AP bit enables the access control for smaller memory spaces than using the domain. The privilege level of the current processor mode affects the access control when the AP bit is used.

4 Sensitive Instructions

This section describes the sensitive instructions of the ARM architecture. Sensitive instructions are the instructions that modify the state of system resources and processor modes; thus, the classification of the sensitive instructions is a must to develop VMMs on any processor architectures. Since a VMM needs to emulate such operations, all sensitive instructions have to be privileged and cause exceptions when executed in user mode [10]. If there are sensitive instructions that do not cause exceptions, special care must be taken to make them cause exceptions. Sensitive instructions that cause exceptions when executed in user mode are called privileged sensitive instructions. Those that do not cause exceptions are called non-privileged sensitive instructions. Table 2 and 3 show the privileged and non-privileged sensitive instructions, and their details are described below. This section focuses on the classification of sensitive instructions. The treatment of sensitive instructions is described later in Section 5.2.

4.1 Co-processor Register Access Instructions

The ARM architecture employs the notion of co-processors to extend the architecture in order to make it configurable. The co-processors provide important functions to manage system resources. For example, the MMU is one of them, and controls virtual memory and cache. The MRC, MRRC, MCR, and MCRR instructions shown in Table 2 are the co-processor register access instructions. Since these instructions are privileged instructions, they cause the undefined instruction exception when executed in non-privileged mode; thus, they can be properly emulated by the VMM.

Table 2: Privileged sensitive instructions

Instruction	Operation	User mode behavior
MRC, MRRC	Load from a co-processor register	Undefined instruction exception
MCR, MCRR	Store to a co-processor register	Undefined instruction exception

Table 3: Non-privileged sensitive instructions

Instruction	Operation	User mode behavior
MRS (CPSR)	Load from CPSR	Executable
MRS (SPSR)	Load from SPSR	Unpredictable
MSR (CPSR)	Store to CPSR	Disregarded
MSR (SPSR)	Store to SPSR	Unpredictable
MOVS	Move registers and copy SPSR to CPSR	Unpredictable
CPS	Change processor mode	Disregarded
LDM (2)	Load from the specified address to user-mode registers	Unpredictable
LDM (3)	LDM (2) and copy SPSR to CPSR	Unpredictable
STM (2)	Store to the specified address from user-mode registers	Unpredictable
RFE	Return from exception	Unpredictable
SRS	Store return state	Unpredictable

4.2 PSR Access Instructions

There are the several instructions that access the PSRs, such as CPSR and SPSR. They are sensitive instructions since the PSRs maintain the processor state and the state is modified by changing their values. While their execution must be emulated by the VMM, they require special attention because they behave differently when executed in non-privileged mode as shown in Table 3. The execution of MSR that stores a value to CPSR is ignored. The execution of MRS (SPSR) and MSR (SPSR), SPSR, MOVS, CPS, LDM (3) is unpredictable. MRS (CPSR) that loads the current value from CPSR is executable, but such behavior is not desirable if its execution in non-privileged mode needs to be hidden. Therefore, these instructions are non-privileged sensitive instructions, and special care must be taken to make them cause exceptions.

4.3 USR Registers Access Instructions

LDM (2) and STM (2) are the instructions that access USR banked registers in privileged mode. LDM (2) loads the value at the specified address to USR banked registers, and STM (2) stores the values of USR banked registers to the specified address. These instructions are supposed to be executed in privileged mode, and they behave differently when executed in non-privileged mode as shown in Table 3. The execution of these instructions in non-privileged mode is unpredictable. Therefore, these instructions are non-privileged sensitive instructions, and special care must be taken to make them cause exceptions.

5 Design and Implementation

This section describes the design and implementation of SIVARM, a simple VMM for the ARM architecture. SIVARM is based on the trap and emulation model to virtualize the ARM processor. Since the trap and emulation model requires the exact analysis of a target processor architecture, the successful execution of Linux on SIVARM verifies that our analysis described above is correct. SIVARM provides only a single VM; thus, it can accommodate only a single guest OS. Although such limitation makes SIVARM unsuitable for practical use, it is very much enough for the verification of the analysis.

5.1 Policies of the Design and Implementation

This section describes the policies of the design and implementation for developing SIVARM.

5.1.1 Detecting Sensitive Instructions

If a VMM controls a guest OS on its VM, the VMM must detect sensitive instructions that are executed by the guest OS. When detected, the VMM updates the state of the VM by emulating the detected instructions. If a VMM cannot detect sensitive instructions that are executed by a guest OS, the VMM cannot properly update the state of the VM according to the sensitive instructions; thus, the VMM cannot execute the guest OS correctly. In this study, we execute SIVARM in privileged mode and the guest OS in non-privileged mode. This design enables the detection of privileged sensitive instructions that are executed by the guest OS as exceptions; thus, SIVARM can emulate them. Non-privileged sensitive instructions described in Section 4.2 do not cause exceptions when executed in non-privileged mode, and special care must be taken to make them cause exceptions. We handle them by replacing them with representative privileged instructions described in Section 5.2.2.

5.1.2 Introducing Virtual Processor Modes

We execute the guest OS in non-privileged mode to detect sensitive instructions that are executed by the guest OS as described above. Since the ARM architecture does not provide multiple protection rings, unlike the x86 architecture, both the kernel and user programs of Linux must share the same non-privileged mode in order to execute them in non-privileged mode. This design, however, causes a problem that there is no way to distinguish if the execution is in the kernel or a user program by examining the processor mode bits of the PSR.

In this study, we solve this problem by introducing virtual processor modes in non-privileged mode. We construct 7 virtual processor modes that correspond to the real processor modes, and SIVARM manages them. SIVARM executes the guest OS kernel in virtual privileged modes and user processes in virtual non-privileged mode.

5.2 Emulation of Sensitive Instructions

This section describes how SIVARM detects and emulates privileged and non-privileged sensitive instructions executed in a guest OS.

5.2.1 Emulation of Privileged Sensitive Instructions

As described in Section 4, most sensitive instructions are privileged instructions. Since a guest OS is executed in non-privileged mode on SIVARM, the execution of a privileged sensitive instruction causes the undefined instructions exception. SIVARM emulates the privileged sensitive instruction that caused the undefined instructions exception. When the undefined instructions exception is caused, the processor mode changes to the undefined mode[1], and interrupts are disabled automatically. The execution jumps to the undefined instructions exception vector, and then jumps to the stub of the undefined instructions exception handler because SIVARM initializes the exception vectors to have branch instructions.

The stub of the undefined instructions exception handler saves the general scratch registers defined in ARM Thumb Procedure Call Standard (ATPCS) [24] on the stack, first. It then calls the the undefined instructions exception handler. By passing the pointer to the saved general scratch registers as its argument to the handler, the saved registers can be referred and updated during the emulation in order to reflect the result of the emulation. After the undefined instructions exception

[1]The ARM architecture defines a processor mode that is called the *undefined* mode.

is handled, the execution returns to the stub, and the saved registers are restored. Finally, the execution returns to the instruction next to where the guest OS caused the exception.

The following shows the stub of the undefined exception handler:

```
1   _vmm_und_handler_stub:
2           push    {r0-r3, r12, r14}
3           mov     r0, sp
4           bl      vmm_und_handler
5           pop     {r0-r3, r12, r14}
6           movs    pc, r14
```

R0 - R3, R12 are the general scratch registers. Since they can be corrupted in the callee functions implemented in the C language, the stub saves them on the stack with the push instruction at Line 2. R14 contains the instruction address next to where the guest OS caused the exception, and is used to return to the guest OS; thus, R14 is also saved here because the branch instruction at Line 4 corrupts it to save the return address in it. The undefined instructions exception handler takes the pointer to the saved general scratch registers as its argument. Since the registers are saved on the stack and the SP points to them; thus, the SP is copied to R0 at Line 3 to make it the first argument. Then, the execution branches to the undefined exception handler at Line 4. After the undefined instructions exception is handled, the execution returns to the stub at Line 5. The saved registers are restored there, and the execution returns to the guest OS at Line 6.

In the undefined instructions exception handler, it first obtains the instruction that caused the undefined instructions exception from the address where the exception was caused. By examining the obtained instruction along with the other information, the handler decides how it handles the exception. There are four courses to handle the exception, emulating the privileged sensitive instruction, handling the representative privileged instruction, handling the virtual banked registers, and having the guest OS handle the exception. The handling of the first case is described below, and the handling of the other three cases are described in Section 5.2.2, 5.3.1, 5.4, respectively.

If the handler finds that a privileged sensitive instruction caused the exception and that it needs to be emulated, SIVARM emulates it by executing it in the VMM instead. The execution of privileged sensitive instructions in non-privileged mode causes the undefined instructions exception because they are executed without sufficient privilege; thus, they are executed in privileged mode for emulation. The following shows the code that executes the privileged sensitive instruction that caused the exception:

```
1   __asm__ __volatile__(
2           "ldr r2, =0xFFF90000\n\t"
3           "str r0, [r2], #4\n\t"
4           "ldr r0, =0xE1A0F00E\n\t"
5           "str r0, [r2], #-4\n\t"
6           "push {r14}\n\t"
7           "blx r2\n\t"
8           "pop {r14}\n\t"
9   );
```

At the beginning of the code above, R0 contains the privileged sensitive instruction that caused the exception. The code uses the two word length area at 0xFFF90000 and 0xFFF90004 to place the instructions necessary for emulation; thus, it first loads 0xFFF90000 in R2 at Line 2. Next, it stores the sensitive instruction in the first word at Line 3, and the following instruction in the second word at Line 5.

```
mov   pc, lr
```

The binary representation of this instruction is 0xE1A0F00E shown at Line 4. R14 (ie. the LR) is saved at Line 6 because it is corrupted by the following branch instruction. The execution branches to 0xFFF90000 where the sensitive instruction was stored at Line 7, and returnes to Line 8 by executing the next instruction stored at Line 5. Finally, the LR is restored at Line 8. Therefore, the code shown above executes the privileged sensitive instruction, which caused the undefined instructions exception, in the VMM.

If the privileged sensitive instruction that caused the undefined instructions exception uses the general scratch registers as its operand, they need to be taken care of. If the sensitive instruction refers registers, the referred registers are restored from the saved registers before executing the instruction. Contrarily, if the sensitive instruction writes a value to a register, the corresponding register in the saved registers is updated to the value after the execution of the instruction. Moreover, the context affecting the execution of the instruction also needs to be restored correctly before the execution of the instruction. This is done by reflecting the condition code bits of the SPSR in the CPSR.

The steps described above emulate the privileged sensitive instruction correctly.

5.2.2 Emulation of Non-Privileged Sensitive Instructions

The execution of a privileged sensitive instruction in non-privileged mode causes the undefined instructions exception; thus, it can be emulated in the exception handler as described in Section 5.2.1. The execution of a non-privileged sensitive instruction in non-privileged mode, however, does not cause an exception. The MRS and the MSR instructions described in Section 4.2 are such instructions. These non-privileged sensitive instructions cannot be detected by using exceptions if they are left as they are; thus, special care must be taken to make them cause exceptions.

This study takes an approach to rewrite non-privileged sensitive instructions in the code of Linux to adequate privileged instructions statically. Since these substitutional instructions cause exceptions, their execution can be detected. We call them the representative privileged instructions. The privileged instructions of which operands are not used in Linux can be used as the representative privileged instructions. The following is an example:

```
mrc   p15, 2, r2, c2, c2, 2
```

Since representative privileged instructions cause an exceptions on behalf of the corresponding non-privileged sensitive instructions, each of them needs to be associated with one of the non-privileged sensitive instructions.

Since the execution of a representative privileged instruction causes the undefined instructions exception, SIVARM handles the exception in the same way as privileged sensitive instructions until it differentiates the courses of the emulation. If the undefined instructions exception handler finds that a representative privileged instruction caused the exception, it examines the instruction that caused the exception and emulates the corresponding non-privileged sensitive instruction. The following shows the code to emulate the representative privileged instruction shown above:

```
 1   if (inst == 0xEE522F52) { // mrc p15,2,r2,c2,c2,2
 2     /*
 3      * arch/arm/kernel/head.S
 4      * msr cpsr_c, #PSR_F_BIT | PSR_I_BIT | SVC_MODE
 5      */
 6     __asm__ __volatile__(
 7       "mrs r0, spsr\n\t"
 8       // PSR_F_BIT | PSR_I_BIT
 9       "orr r0, #0xC0\n\t"
10       "msr spsr_c, r0\n\t"
11     );
12   }
```

This representative privileged instruction is a replacement of the MSR instruction shown at Line 4. While this MSR instruction tries to change the processor mode to SVC, it cannot be properly executed in non-privileged mode. The representative privileged instruction causes an exception, and emulates it in the undefined instructions exception handler by modifying the SPSR not to set the SVC_MODE bit. After the exception handling, the SPSR is restored to the CPSR; thus, the emulation of the MSR instruction can be done appropriately.

The steps described above emulate the non-privileged sensitive instruction correctly by using the representative privileged instruction.

5.3 Virtual Processor Mode

This section describes the implementation of the virtual processor mode introduced in Section 5.1.2. The purpose of the virtual processor mode is the emulation of privileged mode in non-privileged mode. Such emulation requires the virtual banked registers and the virtual protection levels.

5.3.1 Virtual Banked Registers

The banked registers are the registers available for the exception modes as described in Section 3.1. While they can be used only in privileged mode, the guest OS kernel considers they exist; thus, SIVARM emulates and virtually provides them. For their emulation, SIVARM utilizes the general purpose registers of USR and the memory space to hold the contents of the virtual banked registers. The virtual banked registers consists of all the registers shown in Figure 1 except for the PC and the CPSR. When the virtual processor mode is changed, the contents of the general purpose registers are exchanged in accord with the mode before and after the transition.

Moreover, the SPSR for each mode needs to be handled appropriately. When an exception is caused, the value of CPSR before the exception is saved in the SPSR of the mode corresponding to the exception. While saving the SPSR is performed automatically for the transition of the real processor mode, it needs to be emulated by SIVARM for the transition of the virtual processor mode. Therefore, when an exception is handled by the guest OS kernel, SIVARM saves the CPSR in the virtual SPSR of the virtual processor mode that corresponds to the caused exception.

5.3.2 Virtual Protection Levels

While the kernel and user processes of the guest OS share the same real non-privileged mode, they need to have the same access right as they execute in the real processor mode; thus, the virtual privileged mode should have the access right to the all memory space of the guest OS, and the virtual non-privileged mode should not have the access right to the kernel memory space of the guest OS. Moreover, SIVARM is executed in real privileged mode, and should be protected from the guest OS.

In order to provide the protection for the virtual processor modes described above, we construct the virtual protection levels by using the domains described in Section 3.3. We configure the domains as shown in Table 4. D0, D1, and D15 are the domains where the guest OS kernel, user processes, and SIVARM belong to, respectively. Each line shows which component is being executed, and each column shows the setting of the domain. We confirm whether the domain configuration in Table 4 achieves the access control correctly. First, when our VMM is executed, both D0 and D1 are set to the manager. This setting enables the VMM to access all domains. When user processes are executed, D0 is set to the no access. This setting prohibits user processes to access the kernel memory space. When the kernel is executed, both D0 and D1 are set to the manager. This setting enables the kernel to access the memory spaces of both the kernel and user processes. The setting of D0 and D1 is the same setting in the original Linux kernel without a VMM. Moreover, by setting D15 to the client, the access control is enforced by the AP bit in the PTE. This setting successfully protects the VMM from the kernel and user processes.

5.4 The Exception Handling of the Guest OS

The Linux kernel needs to handle exceptions caused by its user processes and sometimes by itself even when it is executed as the guest OS on a VMM. The ARM architecture, however, defines the exception vectors at the fixed locations. When an exception is caused, the processor mode changes to privileged mode, and the execution jumps to the corresponding exception vector. Since only SIVARM executes in privileged mode, it handles the exception first.

Table 4: The access control for virtual processor mode using domain.

	VMM (D15)	User (D1)	Kernel (D0)
VMM (D15)	Client	Manager	Manager
User (D1)	Client	Client	No access
Kernel (D0)	Client	Manager	Manager

The exception handler of SIVARM decides how it handles an exception by examining the obtained instruction along with the other information as described in Section 5.2.1. If the cause of an exception is not to invoke SIVARM for emulation, the exception needs to be passed to the Linux kernel for its handling. In this case, SIVARM upcalls the corresponding exception handler of the Linux kernel by setting up the virtual processor mode including the virtual banked registers and the virtual protection level appropriately.

5.5 Emulation of Devices

When the guest OS executed in non-privileged mode accesses devices, it causes the data abort exception because of the lack of the access right. SIVARM needs to handle such an exception and to emulate the access to devices correctly. Since we support only a single VM on SIVARM, the access to devices can be emulated by simply executing the instruction for device access in SIVARM. Therefore, the emulation of device access is basically the same as the emulation of sensitive instructions. Only difference is that the type of the exception caused by device access is not the undefined instructions exception but the data abort exception; thus, the exception handler of the data abort exception of SIVARM and its stub are used for the emulation of devices.

6 Experiment and Evaluation

We performed several experiments in order to evaluate SIVARM by executing Linux on it. This section describes the experiments and their results.

6.1 Experiment Environment

The experiment environment used for the evaluation is shown below.

• OS	:	Linux 2.6.26
• Board	:	Atmark Techno Armadillo-500
• CPU	:	ARM 1136FJ-S (400MHz)
• Memory	:	64MB

The experiments were performed on the actual evaluation board that equips with the ARM processor.

6.2 Evaluation of the Size of Implementation

Constructing SIVARM based on the design and implementation described in Section 5 resulted in 6835 lines in the C language and 190 lines in the assembly language. The executable object of SIVARM consists of the 52.5KB of the text section and the 1.3KB of the bss section. It does not have the data section. When it executes, it takes the 16MB of the virtual address space at the bottom of it.

The modifications made to the Linux kernel are only 81 lines at 48 places. Among them, the modifications for the representative privileged instructions described in Section 5.2.2 are 55 lines at 39 places. Since the changes to the Linux kernel are very few, it is very easy to apply those changes to the Linux kernel source code.

Table 5: Comparison of Linux kernel sizes

	text	data	bss
Original Linux kernel	3336170	124472	93852
Modified Linux kernel	3330714	124472	93852

Table 6: Execution result of each benchmark

Benchmark	NativeLinux (usec)	SIVARM (usec)	SIVARM/Native
fork+exit	769.17	5,508.46	7.16
fork+exec	1,314.40	8,875.08	6.75
pipe	35.20	204.72	5.82
syscall	0.76	15.15	19.93

Table 5 shows the comparison of the original and modified kernel sizes. The text section is smaller for the modified kernel, and the data and bss sections remain the same. The reduction of the text size is mainly due to the two reasons. One major reason is that SIVARM takes some kernel functions that were originally in the modified kernel. Such functions include the setting up the first virtual address space and the initialization of the trap table. The other rather minor reason is that there are places where multiple non-privileged sensitive instructions are replaced with a single representative privileged instruction. For example, the original kernel requires the two instructions, MRS and CPS, in order to save the current state and to disable interrupts. Since those instructions appear contiguous, they can be replaced with a single representative privileged instruction that performs the two operations.

6.3 Evaluation of Preliminary Performance

We executed micro benchmark programs on the Linux kernel with SIVARM, and compared the results with those executed on the native Linux kernel. We used the benchmark programs, which are basically equivalent to the fork+exec, fork+exit, pipe, and syscall (getpid) programs included in the LMbench benchmark suite. Table 6 shows the results. The numbers in the column of SIVARM/Native shown in the table are the ratios of the execution times with SIVARM normalized to those of the native Linux. From the results, we can observe that there is severe performance degradation by introducing SIVARM. The performance of executing a simple system call (syscall) is the most severely impacted because its execution path on the native Linux kernel is very simple while that on SIVARM includes emulating several privileged instructions. The other benchmark programs involve fair amounts of work that is executed in the Linux kernel; thus, the performance degradation is not that severe.

We further conducted the performance evaluation by employing the performance monitoring counters available on the ARM processor. Table 7, 8, 9, and 10 show the numbers of instructions executed, instruction and data cache misses, and main TLB misses for each benchmark programs we used above, respectively. These results show that the numbers of instructions and cache misses increase on SIVARM while the number of TLB misses does not change. From these results, we can observe that the number of instructions executed on SIVARM increases significantly and that such increase affects instruction cache misses. The similarity can be found in the execution times of the benchmarks. Therefore, the performance degradation can be explained by the increase of the numbers of instructions and cache misses. For example, the performance of a simple system call (syscall) is the most severely impacted because the number of instructions executed on SIVARM is much greater than the native Linux.

We also measured the number of transitions to the VMM occurred during the execution of each benchmark program. Table 11 shows the results. The numbers are the average of several runs; thus, some numbers are non-integer values. In the table, und_emu and dabt_emu are the events caused

Table 7: The number of instructions executed

Benchmark	NativeLinux	SIVARM	SIVARM/Native
fork+exit	79,132	862,570	10.9
fork+exec	111,967	1,338,965	12.0
pipe	5,499	30,414	5.53
syscall	89	2,733	30.7

Table 8: The number of instruction cache misses

Benchmark	NativeLinux	SIVARM	SIVARM/Native
fork+exit	3,672	18,671	5.08
fork+exec	7,434	30,013	4.04
pipe	280	1,671	5.97
syscall	4	28	7.00

Table 9: The number of data cache misses

Benchmark	NativeLinux	SIVARM	SIVARM/Native
fork+exit	6,868	15,282	2.23
fork+exec	10,181	23,364	2.29
pipe	46	461	10.0
syscall	0	20	N/A

Table 10: The number of main TLB misses

Benchmark	NativeLinux	SIVARM	SIVARM/Native
fork+exit	111	114	1.03
fork+exec	118	119	1.01
pipe	11	11	1.00
syscall	0	0	N/A

Table 11: The number of transitions to the VMM

Benchmark	und_emu	dabt_emu	irq_svc	dabt_svc	und_usr	irq_usr	dabt_usr	pabt_usr	swi
fork+exit	2,610	4.1	0.2	0	0	0	28	13	3
fork+exec	3,946	5.9	0.3	1	2	0	26	19	7
pipe	72	0.1	0	0	0	0	0	0	4
syscall	8	0	0	0	0	0	0	0	1

Table 12: Share of execution addresses of sensitive instructions for fork+exit

Order	Function Address	Functionality	Share
1	cpu_v6_dcache_clean_area	Clean data cache line	27
2	cpu_v6_set_pte_ext	Set L2 page table entry	4
3	ret_to_user	Return to user from IRQ OFF	2

Table 13: Share of execution addresses of sensitive instructions for fork+exec

Order	Function Address	Functionality	Share
1	cpu_v6_dcache_clean_area	Clean data cache line	35
2	v6_flush_kern_dcache_page	Write back page	3
3	cpu_v6_set_pte_ext	Set L2 page table entry	3

Table 14: Improvement ratio for benchmark

Benchmark	Before	After	Improvement Ratio
fork+exit	7.16	4.60	35.8
fork+exec	6.75	3.89	42.4

Table 15: Improvement ratio for the number of instructions executed

Benchmark	Before	After	Improvement Ratio
fork+exit	10.90	6.54	40.0
fork+exec	11.96	5.31	47.2

Table 16: Improvement ratio for the number of transition to the VMM

Benchmark	Before	After	Improvement Ratio
fork+exit	2,610	1,479	43.3
fork+exec	3,946	1,986	49.7

by privileged instruction emulation and device emulation, respectively. The others are the events caused by the different types of exceptions. The results show that the numbers of the und_emu events for fork+exit and fork+exec are significantly large. Those numbers imply that the fork and exec system calls require the execution of privileged instructions many times. We further investigated in which functions privileged instructions are executed many times. Table 12 and 13 show the results. Surprisingly, the execution of the single function, cpu_v6_dcache_clean_area, causes 27% and 35% of privileged instructions executed for fork+exit and fork+exec, respectively.

6.4 Performance Improvement

From the preliminary performance evaluation and analysis described above, we found that the emulation of privileged instructions is a major cause of the performance degradation on SIVARM. Especially, the cpu_v6_dcache_clean_area function, which flushes and cleans the data cache lines, is the major source of privileged instruction emulation. The following shows its source code written in the assembly language:

```
1   ENTRY(cpu_v6_dcache_clean_area)
2   1: mcr  p15, 0, r0, c7, c10, 1
3      add  r0, r0, #D_CACHE_LINE_SIZE
4      subs r1, r1, #D_CACHE_LINE_SIZE
5      bhi  1b
6      mov  pc, lr
```

Line 2 is the privileged instruction for data cache line cleaning, and is executed repeatedly as specified by its argument. Since this function repeats the execution of the specific privileged instruction, each execution causes the transition to SIVARM and degrades the performance. Therefore, it should be effective to replace the whole function with a single representative privileged instruction.

Table 14, 15, and 16 show the results of the performance improvement by replacing the whole cpu_v6_dcache_clean_area function with its representative privileged instruction. The results show that the overhead introduced by SIVARM is cut to approximately a half by eliminating the single major source of privileged instruction emulation. By the performance improvement described above, some amount of overheads were eliminated. There are, however, large overheads still introduced by SIVARM. We need to identify and remove the sources of the overheads to further improve the performance of SIVARM.

7 Discussion

This section discusses the issues related to the implementation and limitation of SIVARM.

7.1 Support of Multiple Virtual Machines

SIVARM provides only a single VM; thus, it can accommodate only a single guest OS. It was designed in this way because it is very much enough for the analysis of the ARM architecture's ability to support a VMM although such limitation makes SIVARM unsuitable for practical use. Once a single VM was realized, it is straight forward to support multiple VMs.

In order to support multiple VMs, there is basically nothing specific to the ARM architecture. It is necessary to enhance SIVARM at least to support the context of each VM. The context consists of the contents of the general purpose and privileged registers along with the information of the resources allocated to its VM. When switching between VMs, the current context is saved, and the next context is restored. Since the ARM architecture defines the trap handlers at the specific addresses that are located in SIVARM, all interrupts go through SIVARM; thus, SIVARM can use interrupts for the timings to switch VMs. SIVARM does not provide virtualized devices, either. If a single device needs to be shared among multiple VMs, it needs to be virtualized for multiplexing. If it does not need to be shared, it can be accessed by simply mapping its I/O address rage to a VM since the ARM architecture employs memory mapped I/O.

Only caveat is the physical memory allocation to VMs. SIVARM achieves memory access protection using the domain mechanism. Shadow paging is, however, not implemented since memory virtualization is not necessary to accommodate only a single guest OS for the purpose of our analysis. Shadow paging enables multiple VMs to execute the same guest OS kernel image since it can provide the VMs with the same virtualized physical memory starting at the same address. Shadow paging then takes care of translating virtualized physical addresses to real physical addresses. There is the other way to accommodate multiple VMs without using shadow paging. Physical memory can be partitioned, so that each VM uses only the allocated physical memory region. The partitioned memory regions start at different addresses. Since embedded devices have various configurations, most OS kernels for those devices can take boot options that specify such memory information. Linux is one of such kernels; thus, it can boot using different physical memory start addresses.

7.2 Implementation Portability

The first target processor of our work was ARM926EJ-S, of which architecture is ARMv5TEJ [25, 26]. We successfully booted SIVARM and executed Linux on it using the QEMU system emulator [3] that emulates ARM926EJ-S. SIVARM currently runs on ARM1136JF-S, of which architecture is ARMv6. As far as the sensitive instructions are concerned, the difference between ARMv5TEJ and ARMv6 is the introduction of the CPS, SRS, and RFE instructions. The emulation code for those instructions were added to support ARM1136JF-S. Since the implementation of SIVARM started to support ARMv5TEJ and ARMv6 is defined as a superset of ARMv5TEJ, SIVARM should be portable enough to support both ARMv5TEJ and ARMv6.

The processors based on ARMv7, which is a newer architecture, became widely available recently. According to the architecture manual [1], there is no new system level instruction introduced for ARMv7. Since there are, however, the other architectural changes, SIVARM may require modifications to support ARMv7 based processors.

7.3 Usage Models

Our analysis showed that virtualizing the ARM architecture by simply trapping and emulating sensitive instructions imposes high runtime overhead. In order to address such a performance problem related to virtualization, ARM announced the new processor, Cortex-A15, that provides the hardware assisted virtualization mechanism comparable to Intel VT. Such a hardware assisted mechanism works better if virtualization is always required. The hardware assisted mechanism, however, makes processors much bigger and more complicated; thus, it is not very cost effective to apply it to smaller architectures.

We consider that there are a number of cases that virtualization can be used temporarily to help things better and that higher runtime overhead is not a big problem. For example, when security maintenance is performed, higher runtime overhead can be accepted [12, 14, 16, 18]. In such cases, our simple virtualization method enables the temporary introduction of virtualization by applying the patch to the kernel at the boot time.

8 Summary

This paper described the analysis of the ARM architecture's ability to support a VMM through a simple implementation. Since embedded systems are used for a lot of purposes at a variety of places, their security, reliability, and availability must be improved. Virtualization technology has been used for a long time in order to improve the security of systems, and the other advanced features recently become available to improve the reliability and/or availability of systems. In order to apply virtualization technology to improve them, the detailed analysis of target processor architectures is required. Such analysis was performed for x86 architecture, but it was not available for the ARM architecture, which is the most widely used processor architecture for embedded systems. This paper described such analysis of the ARM processor architecture and the details of a VMM implementation on it. We implemented a simple VMM, called SIVARM, for the ARM architecture to perform the analysis of its ability to support a VMM. We successfully booted Linux on SIVARM and performed the evaluation by executing several benchmark programs on the ARM 1136JF-S processor. It verified the analysis of the sensitive instructions and also showed the performance impact of the virtualization.

References

[1] ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, pages 164–177, 2003.

[3] F. Bellard. QEMU: a Fast and Portable Dynamic Translator. In *Proceedings of USENIX 2005*, pages 41–46, 2005.

[4] R. Bhardwaj, P. Reames, R. Greenspan, V. S. Nori, and E. Ucan. A Choices Hypervisor on the ARM Architecture. CS523 Course Project Report, Department of Computer Science, University of Illinois at Urbana-Champaign, 2006.

[5] T. C. Bressoud, F. B. Schneider. Hypervisor-Based Fault Tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1):80-107, 1996.

[6] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, A. Warfield. Live Migration of Virtual Machines. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation*, pages 273–286, 2005.

[7] C. Dall and J. Nieh. KVM for ARM. In *Proceedings of the Ottawa Linux Symposium*, pages 45–56, 2010.

[8] D. R. Ferstay. Fast Secure Virtualization for the ARM Platform. Master of Science Thesis, University of British Columbia, 2006.

[9] B. D. Gold, R. R. Linde, M. Schaefer, J. F. Scheid. VM/370 Security Retrofit Program. In *Proceedings of ACM Annual Conference*, pages 411-418, 1977.

[10] R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, 1974.

[11] Green Hills Software Inc. White paper: Integrity Secure Virtualization for ARM. 2010. http://www.ghs.com/ds/index.php?ds=integrity_virt_ARM.

[12] J. Hoopes. Virtualization for Security: including Sandboxing, Disaster recovery, High Availability, Forensic Analysis, and Honeypotting. Syngress Publishing, 2008.

[13] J-Y. Hwang, S-B. Suh, S-K. Heo, C-J. Park, J-M. Ryu, S-Y. Park, and C-R. Kim. Xen on ARM: System Virtualization using Xen hypervisor for ARM-based Secure Mobile Phones. In *Proceedings of IEEE Consumer Communications and Networking Conference*, pages 257-261, 2008.

[14] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. VMM-based Hidden Process Detection and Identification using Lycosid. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 91–100, 2008.

[15] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, C. E. Kahn. A VMM Security Kernel for the VAX Architecture. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 2–19, 1990.

[16] T. Kobayashi, H. Yamada, K. Kono. A Keylogger Detection System based on Virtual Machine Technology. IPSJ SIG Technical Report, IPSJ-OS08107001, Vol.2008-OS-107 No.9, 2008.

[17] D. Kudinskas. Virtualizing the ARM - the ARM Hypervisor. Final Year Project Report, School of Computer Science, The University of Manchester, 2010.

[18] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor Support for Identifying Covertly Executing Binaries. In *Proceedings of Conference on Security Symposium*, pages 243–258, 2008.

[19] Open Kernel Labs. OKL4 Microvisor. http://www.ok-labs.com/products/okl4-microvisor.

[20] H. P. Reiser, F. J. Hauck, R. Kapitza, W. Schroder-Preikschat. Hypervisor-Based Redundant Execution on a Single Physical Host. In *Proceedings of European Dependable Computing Conference*, pages 67-68, 2006.

[21] J. S. Robin and C. E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th USENIX Security Symposium*, pages 129–144, 2000.

[22] M. Rosenblum and T. Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. *IEEE Computer*, 38(5):39–47, 2005.

[23] D. Seal. ARM Architecture Reference Manual 2nd Edition. Addison-Wesley Professional, 2000.

[24] A. N. Sloss, D. Symes, C. Wright. ARM System Developer's Guide Designed and Optimizing System Software. Morgan Kaufmann Publishing, 2004.

[25] A. Suzuki, S. Oikawa. Development of Virtual Machine Monitor for ARM Architecture. IPSJ SIG Technical Report, IPSJ-OS09111012, Vol.2009-OS-111 No.12, 2009.

[26] A. Suzuki, S. Oikawa. Development of Virtual Machine Monitor for ARM Architecture. In *Proceedings of the 10th IEEE International Conference on Computer and Information Technology*, pages 2244-2249, 2010.

[27] VirtualLogix. Real-time Virtualization for Conencted Devices. http://www.virtuallogix.com/solutions/product/arm.html.

[28] VMware. VMware Mobile Virtualization Platform, Virtual Appliances for Mobile phones. http://www.vmware.com/products/mobile/.