논문 2013-08-18

ARMv7 구조를 위한 가상 머신 모니터 구현

(Implementation of a Virtual Machine Monitor for ARMv7 Architecture)

오 승 재, 신 동 하*
(Seung-Jae Oh, Dongha Shin)

Abstract : Virtualization technology has been applied in IA-32 based server or desktop systems. Recently it has been applied in ARM based mobile systems. Virtualization technology provides many useful features that are not possible in operating system level such as isolation, interposition, encapsulation and portability. In this research, we implement an ARM based VMM(Virtual Machine Monitor) by using the following techniques. First, we use "emulation" to virtualize the processor. Second, we use "shadow page tables" to virtualize the memory. Finally, we use a simple "pass-through I/O" to virtualize the device. Currently the VMM runs ARM Linux kernel 3.4.4 on a BeagleBoard-xM, and we evaluated the performance of the VMM using lmbench and dhrystone. The result of the evaluation shows that our VMM is slower than Xen on ARM that is implemented using paravirtualization but has good performance among the VMMs using full-virtualization.

Keywords : Virtualization, ARM, Linux, Embedded system

Ⅰ. 서론

가상화 기술은 실제 하드웨어에서 제공하는 자원이 아닌 다른 자원을 사용하여 구현한 추상화 된 인터페이스를 상위 계층에게 제공하는 기술로서, 기존 운영체제에서는 구현하기 어려웠던 하드웨어 자원의 격리(isolation), 캡슐화(encapsulation), 이식(portability) 및 중재(interposition)와 같은 기술들을 쉽게 구현하여 제공한다[1, 2]. 가상화 기술은 IA-32 구조를 사용하는 서버나 데스크톱 환경에서 주로 사용되었으나, 최근 ARM 구조를 기반으로 하는 다양한 스마트 디바이스들이 등장하면서 ARM 구조에서도 가상화 기술이 활발히 연구되고 있다[3-8].

* Corresponding Author (dshin@smu.ac.kr)
Received: 24 Jan. 2013, Revised: 11 Mar. 2013. Accepted: 21 Mar. 2013.
Seung-Jae Oh, Dongha Shin: Sangmyung University
※ 본 논문은 2013년도 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. 2010-0010901).

ISSN : 1975-5066

일반적으로 가상화를 구현하기 위해서는 하드웨어 자원인 프로세서, 메모리 및 디바이스 등을 가상화한다[9]. 본 연구에서는 ARMv7 구조에서 이러한 하드웨어 자원을 가상화하는 가상 머신 모니터(Virtual Machine Monitor, 이하 VMM)를 구현하였다. 본 연구에서 구현한 VMM은 Cortex-A8 프로세서가 탑재된 BeagleBoard-xM[10] 상에서 ARM Linux 커널 3.4.4를 수행시켜 성능을 시험하였으며, 기존에 연구된 ARM 기반 VMM의 성능과 비교한 결과 반 가상화 방식보다는 낮은 성능을 보였지만 완전 가상화 방식 중에서는 만족스러운 성능을 보였다.

본 논문의 구성은 다음과 같다. 먼저 2장에서는 ARM 구조를 기반으로 하는 가상화 연구들을 소개하고, 3장, 4장 및 5장에서 각각 프로세서, 메모리 및 디바이스를 가상화 하기 위하여 사용한 기술에 대하여 설명한다. 6장에서는 본 연구에서 구현한 VMM의 성능을 측정하여 관련 연구에서 소개한 가상화 연구들의 성능과 비교한 결과에 대하여 설명하며, 마지막 7장에서는 본 논문의 결론에 대하여 기술한다.

Ⅱ. 관련 연구

이 장에서는 기존 ARM 구조 기반의 가상화 연구인 Xen on ARM[3], KVM for ARM[4], ViMo[5, 6], SIVARM[7], 그리고 Xvisor[8]에 대하여 소개한다. 이들 연구 중 Xen on ARM, ViMo, SIVARM 및 Xvisor의 성능은 이후 6장에서 본 연구에서 구현한 VMM의 성능과 함께 비교한다.

1. Xen on ARM

Xen on ARM[3]은 공개 소프트웨어 기반의 x86 반 가상화(paravirtualization) 소프트웨어인 Xen을 ARM 기반 프로세서에서 수행하기 위한 연구이다. 이 연구는 게스트 운영체제의 소스 코드를 수정하여 VMM에 맞게 이식함으로써 가상화를 구현하는 반 가상화 방식을 사용하기 때문에 성능이 매우 좋지만 운영체제의 소스를 수정하여 이식하는 작업에 따른 공학적 비용이 많이 발생한다는 단점이 있다.

2. KVM for ARM

KVM for ARM[4]은 Linux 커널에서 제공하는 VMM인 KVM(Kernel based Virtual Machine)을 ARM 구조에서 동작하도록 구현한 연구이다. 이 연구는 가상화를 위하여 호스트 커널의 자원을 활용하는 hosted 가상화 방식을 사용한다. KVM for ARM은 hosted 가상화 방식을 사용하므로 호스트 커널에 이미 구현되어 있는 자원 관리 루틴을 사용할 수 있다는 장점이 있지만, 호스트 커널과 게스트 간의 월드 스위치(world switch) 시에 공유 메모리를 이용하여 상태 정보를 교환하는데 오버헤드가 발생하며, 호스트 커널과 게스트 커널을 동시에 수행하는데 따르는 하드웨어 제약 사항을 해결하는데 많은 오버헤드가 발생한다는 단점이 있다.

3. ViMo

ViMo[5, 6]는 ARM 구조에서 완전 가상화(full-virtualization)를 지원하는 VMM이다. 이 연구는 실행 시간에 게스트의 가상화 민감 명령어를 탐지하는 code tracer, 탐지한 가상화 민감 명령어를 에뮬레이션 하는 CPU virtualizer, 가상 머신 간의 메모리 관리를 위한 memory virtualizer, 가상 머신의 인터럽트를 제어하고 처리하는 interrupt controller virtualizer, 그리고 가상 머신 간의 전환을 처리하는 scheduler로 구성되어 있다. ViMo는 완전 가상화 방식을 사용하기 때문에 게스트 운영체제를 수정하지 않고 가상화를 구현할 수 있다는 장점이 있지만, 반 가상화 방식에 비하여 성능이 떨어진다는 단점이 있다.

4. SIVARM

SIVARM[7]은 ARM 구조에서 하나의 게스트 Linux 커널을 가상 수행시키기 위한 연구이다. 이 연구는 게스트 커널을 User 모드에서 동작시켜 발생하는 예외(exception)를 이용하여 프로세서 가상화를 구현한다. 메모리와 디바이스 가상화는 구현하지 않으며, VMM, 커널 및 사용자 모드 간의 메모리 접근만 제어한다. SIVARM은 메모리와 디바이스 가상화를 구현하지 않기 때문에 하나의 게스트 커널만 수행시킬 수 있다는 한계가 있다.

5. Xvisor

Xvisor[8]는 오픈 소스로 개발된 ARM 기반의 VMM으로서, 관련 논문 등은 현재까지 발표되지 않았기 때문에 본 논문에서는 공개된 소스 코드를 기반으로 가상화 구현 방법을 분석하였다. 이 연구는 프로세서 가상화 구현을 위하여 게스트 Linux 커널을 컴파일 하여 출력된 ELF 파일에서 가상화 민감 명령어를 찾아 예외 발생 명령어로 변환한 후, VMM의 예외 처리 핸들러에서 가상화 민감 명령어를 에뮬레이션 한다. 메모리 가상화는 섀도 페이지 테이블 기술을 사용한 VTLB라는 자료구조를 유지하여 게스트의 주소 변환을 관리하며, 게스트 커널이 수행하는 메모리 관리 명령어를 이용하여 VTLB의 내용을 관리한다. 디바이스 가상화는 VMM이 디바이스를 에뮬레이션 하여 게스트에게 제공하는 방식으로 구현한다. Xvisor는 본 연구에서 구현한 VMM과 가장 유사한 방법으로 구현된 VMM으로서, QEMU 및 BeagleBoard-xM에서 Linux 커널을 가상화하여 수행시킬 수 있는 데모를 제공하고 있다.

Ⅲ. 프로세서 가상화

이 장에서는 프로세서 가상화를 구현하기 위한 가상화 민감 명령어의 변환 방법과 가상 프로세서의 동작에 대하여 설명한다.

1. 가상화 민감 명령어 변환

가상화 민감 명령어는 프로세서 상태의 변화를 일으키거나 프로세서의 상태에 따라 명령어의 수행

표 1. ARMv7의 가상화 민감 명령어
Table 1. Virtualization Sensitive instructions of ARMv7

명령어 분류	가상화 민감 명령어
Standard data-processing instructions	ADCS, ADDS, ANDS, BICS, EORS, ORRS, RSBS, RSCS, SBCS, SUBS, MOVS, MVNS (단, Rd=PC 또는 r15)
Shift instructions	ASRS, LSLS, LSRS, RORS, RRXS (단, Rd=PC 또는 r15)
Status register access instructions	CPS, MRS, MSR
Load/store instructions	LDRT, LDRHT, LDRSHT, LDRBT, LDRSBT, STRT, STRHT, STRBT
Load/store multiple instructions	LDM (exception return/user registers), STM (user registers)
Exception-generating and handling instructions	RFE, SMC(SMI), SRS
Coprocessor instructions	CDP, CDP2, MCR, MCR2, MCRR, MCRR2, MRC, MRC2, MRRC, MRRC2, LDC, LDC2, STC, STC2

결과가 달라지는 명령어이다[11]. VMM 상에서 수행되는 게스트가 이러한 가상화 민감 명령어를 수행하면 VMM 및 VMM 상의 다른 게스트의 동작에도 영향을 미칠 수 있다. 이를 방지하기 위하여, 가상화를 구현할 때는 VMM이 게스트에서 가상화 민감 명령어가 직접 수행되지 않도록 관리하여야 한다.

본 연구에서는 먼저 ARMv7 구조의 가상화 민감 명령어를 식별하기 위하여 ARMv7 명령어 집합에 있는 명령어들의 동작 과정을 조사하였으며, 그 결과 Advanced SIMD 와 VFP 명령어를 제외한 총 15개 분류의 300여개 명령어들 중 표 1과 같이 7개의 분류에서 총 48개의 가상화 민감 명령어를 식별하였다[12].

프로세서 가상화를 구현하기 위해서는 전통적으로 트랩(trap) 발생 후 에뮬레이션(emulation) 하는 방법(trap-and-emulation)을 사용한다. 이 방법은 게스트 운영체제를 비 특권(non-privileged) 모드에서 수행시키고, 비 특권 모드에서 수행되는 게스트 운영체제가 특권(privileged) 명령어인 가상화 민감 명령어를 수행하여 트랩이 발생하면, VMM이 시스템의 권한을 가져와 특권 모드에서 해당 명령어에 대한 에뮬레이션을 수행하는 방법이다. 이 방법을 사용하여 프로세서 가상화를 구현하기 위해서는 가상화 민감 명령어 집합이 특권 명령어 집합의 부분 집합이어야 한다는 조건이 존재하는데, ARMv7 구조는 이 조건을 만족하지 않는다. 그러므로 ARMv7 구조에서 trap-and-emulation 방법으로 가상화를 구현하기 위해서는 특권 명령어 집합에 속하지 않는 가상화 민감 명령어를 예외 발생 명령어로 변환해주어야 한다. 본 연구에서는 게스트 Linux 커널에 존재하는 가상화 민감 명령어를 예외 발생 명령어인 SVC 명령어로 변환하여, 가상화 민감 명령어가 수행될 때마다 발생하는 SVC 예외를 트랩 함으로써 게스트에서 가상화 민감 명령어가 수행되는 것을 VMM이 감지할 수 있도록 하였다.

가상화 민감 명령어의 탐지 및 변환은 게스트 Linux 커널의 컴파일 시간에 본 연구에서 개발한 도구를 사용하여 자동으로 수행한다[13, 14]. 이와 유사한 방법을 사용하는 KVM for ARM[4]은 소스 코드로부터 가상화 민감 명령어를 탐지하여 변환하기 때문에 매크로 확장에 의하여 생성되는 가상화 민감 명령어를 변환할 수 없다는 문제점이 있다. 일반적으로 Linux 커널의 컴파일 과정은 소스 코드로부터 컴파일, 어셈블 그리고 링크 과정을 거쳐 최종적으로 실행 파일을 생성하는 과정으로 이루어져 있는데, 본 연구에서는 컴파일 과정 이후에 출력되는 어셈블리 코드로부터 어셈블리 매크로를 처리한 후 가상화 민감 명령어를 탐지하기 때문에 게스트 Linux 커널의 가상화 민감 명령어를 모두 탐지할 수 있다. 또한 이와 같이 가상화 민감 명령어를 자동으로 탐지하고 변환하는 과정을 통하여 Xen on ARM[3]과 같은 반 가상화 방식의 단점인 운영체제 수정에 따른 공학적 비용 발생을 최소화할 수 있다.

이렇게 탐지한 가상화 민감 명령어는 ARMv7의 예외 발생 명령어인 SVC 명령어로 변환하는데, 이 때 SVC 명령어의 오퍼랜드에는 가상화 민감 명령어의 종류를 식별하기 위한 값을 저장한다. 게스트 수행 중, SVC 명령어로 변환된 가상화 민감 명령어가 실행되면 VMM이 각 게스트의 가상 프로세서를 이용하여 에뮬레이션을 수행한다.

2. 가상화 민감 명령어 에뮬레이션

2.1 가상 프로세서

가상 프로세서는 VMM 상에서 동작하는 게스트의 상태를 저장하고 관리하기 위하여 VMM이 유지하는 자료 구조이다. 본 연구에서 구현한 VMM은 각 게스트마다 한 개의 가상 프로세서를 유지하며, 다음과 같은 방법을 사용하여 각 게스트의 가상 프로세서를 관리한다.

2.2 예외 처리

본 연구에서는 trap-and-emulation 방법으로 가상화를 구현하기 때문에 게스트 수행 중 발생하는 트랩, 즉 예외를 이용하여 VMM이 모든 게스트를 제어한다. 이를 위하여 VMM은 최초 시스템 초기화 단계에서 게스트를 제어하기 위한 예외 벡터 및 예외 핸들러를 설정한다. 본 연구에서 구현한 VMM의 예

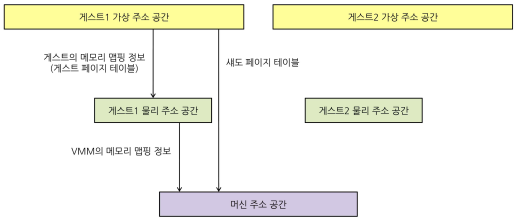

그림 1. VMM의 가상 주소 변환
Fig. 1 The address translation of VMM

외 벡터에서는 예외 발생 직후 게스트의 R0-R14 레지스터 및 SPSR(Saved Program Status Register)을 예외가 발생한 모드의 스택에 저장한다. 이렇게 저장한 레지스터 값들은 가상 프로세서의 상태를 유지하고 가상화 민감 명령어를 에뮬레이션 하기 위하여 사용된다.

ARMv7 구조에는 Reset, Undefined Instruction, Supervisor Call(SVC), Prefetch Abort, Data Abort, IRQ 및 FIQ 예외가 존재하며 [12], VMM은 다음과 같은 방법으로 각 예외를 처리한다. 먼저 SVC 예외가 발생하면 SVC 예외 핸들러에서 예외 발생 명령어가 가상화 민감 명령어인지 식별한다. 식별된 명령어가 가상화 민감 명령어이면 해당 명령어를 에뮬레이션 하고, 그렇지 않으면 게스트에게 가상으로 SVC 예외를 발생시킨다. Prefetch Abort 및 Data Abort 예외 발생 시에는 메모리 가상화를 위하여 섀도 페이지 테이블(shadow page table)을 제어한다. 섀도 페이지 테이블의 동작은 4장에서 자세히 설명한다. 나머지 Reset, Undefined Instruction, IRQ 및 FIQ 예외 발생 시에는 발생한 예외에 따라 게스트에게 가상으로 예외를 발생시킨다.

2.3 가상화 민감 명령어 에뮬레이션

SVC 명령어로 변환된 가상화 민감 명령어로부터 SVC 예외가 발생하면 해당 명령어를 에뮬레이션 한다. 가상화 민감 명령어의 에뮬레이션은 예외 발생 직후에 SVC 모드의 스택에 저장한 레지스터 값과 각 게스트의 가상 프로세서 상태를 기반으로 수행하며, 그 결과를 다시 가상 프로세서 및 SVC 모드의 스택에 저장된 레지스터 값에 반영한다.

에뮬레이션을 완료하고 SVC 예외 벡터로부터 원래의 수행 흐름으로 돌아갈 때는 SVC 모드의 스택에 저장된 레지스터 값들을 다시 하드웨어 레지스터로 복구하여 게스트가 수행을 재개할 수 있도록 한다. SIVARM[5] 및 Xvisor[8]도 이와 유사한 방법으로 가상화 민감 명령어의 에뮬레이션을 수행한다.

Ⅳ. 메모리 가상화

이 장에서는 메모리 가상화를 구현하기 위하여, 게스트의 주소 변환을 관리하기 위한 섀도 페이지 테이블의 구현 방법과 게스트의 메모리 접근을 제어하는 방법에 대하여 설명한다.

1. 섀도 페이지 테이블

일반적으로 가상 메모리 구조를 사용하는 운영체제는 페이지 테이블을 사용하여 가상 주소와 물리 주소의 변환을 관리한다. VMM 상에서 여러 개의 게스트 운영체제가 가상화되어 동작할 때에는 그림 1과 같이 각각의 게스트가 사용하는 동일한 물리 주소가 실제 머신의 메모리상에서는 각 게스트에 따라 서로 다른 주소에 맵핑된다. 예를 들어, 게스트 1이 생각하는 물리 주소와 게스트 2가 생각하는 물리 주소는 동일할 수 있지만 VMM은 각 게스트가 생각하는 동일한 물리 주소를 서로 다른 머신 주소로 맵핑하도록 관리한다.

섀도 페이지 테이블은 VMM 상에서 동작하는 여러 게스트의 가상 주소 변환을 관리하는 기술이다[9]. VMM은 게스트의 가상 주소를 머신 주소로 변환하기 위하여 게스트 페이지 테이블에 존재하는 게스트의 가상->물리 주소 맵핑을 가상->머신 주소 맵핑으로 변환하고 섀도 페이지 테이블에 적절히 동기화하여 유지해준다. 게스트 수행 시 하드웨어 MMU(Memory Management Unit)는 게스트의 페이지 테이블이 아닌 섀도 페이지 테이블을 기준으로 주소 변환을 수행하게 된다. 본 연구에서는 이러한 섀도 페이지 테이블 기술을 사용하여 VMM 상에서 동작하는 게스트의 메모리를 가상화하며, 다음과 같은 방법으로 섀도 페이지 테이블을 유지한다[15-17].

1.1 Abort 예외를 이용한 섀도 페이지 테이블 관리

본 연구에서 구현한 섀도 페이지 테이블은 게스트 페이지 테이블의 모든 주소 맵핑 엔트리를 섀도 페이지 테이블에 동기화하지 않고 게스트가 실제 접근하는 페이지에 대한 주소 맵핑 엔트리만을 섀도 페이지 테이블에 유지하는 요구 페이징 방식을 사용하며, 이를 위하여 게스트에서 발생한 Prefetch 및 Data Abort 예외를 이용한다. 그림 2는 섀도 페이지 테이블 엔트리의 생성 과정을 나타낸 그림이다.

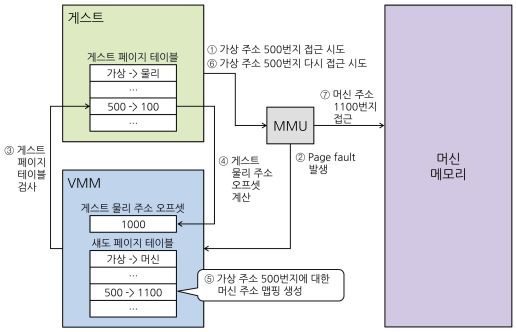

그림 2. 섀도 페이지 테이블 엔트리 생성 과정
Fig. 2 The process of creating shadow page table entry

게스트가 가상 주소를 사용하여 메모리의 특정 주소에 접근하였을 때 하드웨어 MMU는 VMM이 관리하는 섀도 페이지 테이블을 기준으로 동작하므로, 게스트가 사용하는 가상 주소에 대한 가상->머신 주소 맵핑이 섀도 페이지 테이블에 생성되어 있지 않으면 주소 변환에 실패(translation fault)하여 Prefetch 또는 Data Abort 예외가 발생한다. Abort 예외가 발생하면 VMM의 Abort 예외 핸들러에서는 게스트가 접근을 시도하였던 가상 주소에 대한 물리 주소 맵핑이 게스트의 페이지 테이블에 존재하는지 여부와 게스트의 접근 권한을 검사한다.

만약 게스트가 접근을 시도한 가상 주소에 대한 물리 주소 맵핑이 게스트 페이지 테이블에 존재하고, 게스트가 접근이 가능한 페이지이면 게스트 페이지 테이블의 가상->물리 주소 맵핑에 현재 게스트의 머신 주소 오프셋을 계산한 가상->머신 주소 맵핑을 섀도 페이지 테이블에 생성한다. 그리고 나서 게스트가 수행하려고 했던 명령어를 다시 수행하도록 하면 MMU는 섀도 페이지 테이블에 생성된 엔트리를 보고 정상적으로 주소 변환을 수행해주게 된다.

그러나 만약 게스트 페이지 테이블에 가상->물리 주소 맵핑이 존재하지 않거나 게스트가 접근할 수 없는 페이지인 경우, VMM은 게스트에게 가상으로 Prefetch 또는 Data Abort 예외를 발생시킨다. 게스트 운영체제는 가상 Abort 예외를 전달 받아 자신의 Abort 예외 핸들러에서 해당 주소에 대한 가상->물리 주소 맵핑을 자신의 페이지 테이블에 생성한 후, Abort 예외가 발생하였던 명령어를 다시 수행하게 된다. 이 때 섀도 페이지 테이블에는 게스트가 접근한 가상 주소에 대한 머신 주소 맵핑이 여전히 존재하지 않는 상태이므로 다시 MMU에서 Abort 예외가 발생한다. 이 때는 VMM의 Abort 예외 핸들러에서 이 예외를 처리하게 되며, 게스트 페이지 테이블에 가상->물리 주소 맵핑이 생성되어 있으므로 섀도 페이지 테이블에 해당 가상 주소에 대한 머신 주소 맵핑을 생성한 후 Abort 예외가 발생하였던 명령어를 다시 수행하도록 하면 MMU가 정상적으로 주소 변환을 수행해주게 된다.

1.2 메모리 관리 명령어를 이용한 섀도 페이지 테이블 관리

본 연구에서 구현한 VMM은 게스트에서 발생하는 Abort 예외 외에도 게스트 운영체제가 수행하는 메모리 관리 명령어를 이용하여 섀도 페이지 테이블을 관리한다.

게스트 운영체제는 자신의 페이지 테이블 엔트리를 수정하거나 삭제한 후 이를 TLB(Translation Lookaside Buffer)에 반영하기 위하여, 수정하거나 삭제한 주소에 대한 TLB 삭제(invalidate) 명령어를 수행한다. 또한 프로세스 간 문맥 전환을 위하여 ASID가 저장된 CP15의 CONTEXTIDR(Context ID Register)을 변경하는 명령어를 수행한다. 이러한 메모리 관리 명령어는 모두 가상화 민감 명령어이므로, 명령어가 수행되는 것을 VMM이 감지할 수 있다. 만약 게스트가 TLB 삭제 명령을 수행하는 경우에는 섀도 페이지 테이블에서 해당 주소에 대한 맵핑을 삭제하고, CONTEXTIDR을 변경하는 경우에는 섀도 페이지 테이블에 존재하는 게스트의 가상-머신 주소 맵핑을 모두 삭제함으로써 섀도 페이지 테이블을 관리한다.

2. 메모리 접근 제어

ARMv7 구조에는 User, FIQ, IRQ, Supervisor, Abort, Undefined 및 System 모드가 존재한다[12]. 본 연구에서 구현한 VMM 상에서 게스트로 동작하는 ARM Linux 커널은 일반적으로 특권 모드인 Supervisor 모드에서 동작하며, 사용자 프로세스는 비 특권 모드인 User 모드에서 동작한다. 본 연구에서는 trap-and-emulation 방법을 사용하여 가상화를 구현하므로, VMM은 Supervisor 모드로 동작하고 게스트 Linux 커널과 사용자 프로세스는 User 모드로 동작하도록 한다. 이 때 게스트로부터 VMM의 메모리 영역을 보호하고 게스트의 사용자 프로세스가 게스트 커널의 메모리 영역에 접근하는 것을 막기 위하여 각 메모리 영역에 대한 접근 제어 방법을 구현한다. Xen on ARM[3], KVM for ARM[4] 및 SIVARM[7]은 ARM 구조의 도메인과 접근 권한 메커니즘을 이용하여 VMM, 게

스트 커널 및 게스트 사용자 프로세스에 각각 다른 도메인을 설정함으로써 메모리 접근을 제어한다. 본 연구에서도 이러한 방법을 사용하여 메모리 접근 제어를 구현하였다.

Ⅴ. 디바이스 가상화

디바이스 가상화를 구현하는 방법은 일반적으로 에뮬레이션과 pass-through 두 가지의 방법이 있다[2, 18]. ARM 구조는 memory-mapped I/O를 사용하기 때문에 메모리의 특정 주소가 디바이스의 입출력을 위한 레지스터에 할당되어 있으며, 메모리의 주소에 따라 각 디바이스가 구분된다[19]. 또한 디바이스에 대한 접근은 메모리 접근 명령을 사용하여 처리한다.

본 연구에서는 게스트가 자신의 디바이스 드라이버를 통하여 디바이스가 맵핑된 메모리 주소에 직접 접근하게 하고, VMM이 게스트의 메모리 접근을 관리하는 pass-through 방식으로 디바이스 가상화를 구현하였다. 이를 위하여 디바이스가 맵핑된 메모리 주소 영역을 특권 모드에서만 접근 가능하도록 변경하고, 게스트의 디바이스 드라이버에서 디바이스가 맵핑된 메모리 주소에 접근할 때 Abort 예외가 발생하도록 한다. VMM의 Abort 예외 핸들러에서는 Abort 예외가 발생한 주소를 보고 게스트가 어떤 디바이스에 접근하는지를 감지할 수 있다. 본 연구에서는 이러한 방법을 사용하여 UART 디바이스를 가상화하였다.

Ⅵ. 시험 및 평가

본 연구에서는 ARMv7 구조에서 프로세서, 메모리 및 디바이스 자원을 가상화하는 VMM을 구현하였다. 본 연구에서 구현한 VMM은 아래와 같이 ARMv7 구조를 사용하는 Cortex-A8 프로세서가 탑재된 BeagleBoard-xM[10] 상에서 ARM Linux 커널 3.4.4를 수행하여 시험하였다.

- 하드웨어: BeagleBoard-xM
(ARMv7 Cortex-A8 프로세서, 메모리 512MB)
- 게스트 운영체제: ARM Linux 3.4.4

본 시험에서는 시스템의 특정 동작의 성능을 측정하는 lmbench[20]와 전체 시스템의 수행 성능을 측정하는 dhrystone[21]을 사용하여 가상화된 Linux 커널과 가상화되지 않은 Linux 커널 상에서 각각의 성능을 측정하여 그 비율을 계산하였으며, 시험 결과를 관련 연구에서 소개한 가상화 연구들의 성능 측정 결과와 비교하였다.

표 2. lmbench 대역폭 시험 결과
Table 2. The results of bandwidth test of lmbench

시험	Xen on ARM	ViMo	SIVARM	Xvisor	본 연구
bw_mem (512 rd)	1.041	-	-	0.943	0.901
bw_mem (512 wr)	0.985	-	-	0.944	0.921
bw_mem (512 rdwr)	0.985	-	-	0.943	0.911
bw_mem (512 cp)	-	-	-	0.938	0.910
bw_mem (512 bzero)	-	-	-	0.769	0.912
bw_mem (512 bcopy)	-	-	-	0.888	0.901
bw_pipe	0.893	-	-	0.227	0.202

단위: 비율 = (가상화 측정값(MB/sec) / 가상화되지 않은 측정값(MB/sec))
-: 발표된 결과 값이 없음

1. lmbench 시험

lmbench는 프로세서, 메모리, 네트워크, 파일 시스템 및 디스크 상에서 데이터의 이동에 따른 대역폭(bandwidth)과 지연시간(latency)을 측정함으로써 시스템의 기본적인 동작들의 성능을 측정하기 위한 벤치마크 프로그램이다. 본 시험에서는 lmbench-3.0-a9 버전에서 제공하는 6개의 대역폭 시험과 26개의 지연시간 시험 중, 파일 시스템 및 네트워크 관련 시험을 제외한 12개의 시험을 수행하였다. 본 시험의 결과는 관련 연구 중 성능 측정 결과가 발표된 Xen on ARM[3], ViMo[5, 6], SIVARM[7]의 측정 결과 및 오픈 소스로 공개된 Xvisor[8]를 직접 컴파일 하고 수행시켜 성능을 측정한 결과와 함께 비교하였다.

표 2는 lmbench를 이용하여 시스템의 대역폭을 측정한 시험 결과를 비교한 표이다. 대역폭 시험의 결과는 1 이하이면서 값이 클수록 성능이 좋음을 의미한다. 표 2의 결과 중 ViMo와 SIVARM의 결과 값은 발표된 자료가 존재하지 않아 기재하지 못하였다.

표 3은 lmbench를 이용하여 시스템의 지연시간을 측정한 시험 결과를 비교한 표이다. 표 3에서 Xen on ARM과 ViMo의 문맥 전환 시간(lat_ctx) 결과 값은 발표된 자료가 그래프로만 존재하기 때문에 그래프로부터 추정한 결과를 기재하였으며, 각 연구에서 발표되지 않은 결과 값은 기재하지 못하였다. 지연시간 시험의 결과는 1 이상이면서 값이 작을수록 성능이 좋음을 의미한다.

표 2와 표 3의 시험 결과에서, Xen on ARM은 관련 연구에서 설명한 바와 같이 반가상화 방식을

표 3. lmbench 지연시간 시험 결과
Table 3. The results of latency test of lmbench

시험	Xen on ARM	ViMo	SIVARM	Xvisor	본 연구
lat_ctx (32K 2P)	1.44‡	36.29‡	-	5.59	11.13
lat_ctx (32K 4P)	1.62‡	37.14‡	-	4.69	9.17
lat_ctx (32K 8P)	1.39‡	38.00‡	-	4.63	8.54
lat_ctx (32K 16P)	1.39‡	32.73‡	-	4.61	7.84
lat_ctx (32K 32P)	1.41‡	29.17‡	-	6.15	7.91
lat_fcntl	-	-	-	110.43	141.34
lat_mem_rd (1M 0.00586)	-	-	-	1.06	1.06
lat_mem_rd (1M 0.12500)	-	-	-	1.12	1.06
lat_mem_rd (1M 0.75000)	-	-	-	3.40	1.06
lat_pagefault	-	-	-	74.16	134.52
lat_pipe	1.735	-	5.82	54.48	48.62
lat_proc (fork+exit)	3.465	-	7.16	51.56	35.67
lat_proc (execve+exit)	3.385	-	6.75	138.54	35.39
lat_proc (fork+/bin/sh)	-	-	-	17.64	18.16
lat_select	-	-	-	14.74	7.37
lat_sem	1.77	-	-	107.59	66.78
lat_sig (install)	-	-	-	275.41	140.28
lat_sig (catch)	1.82	-	-	100.31	134.19
lat_syscall (null)	2.50	-	19.93	318.50	274.54
lat_syscall (read)	1.90	-	-	240.36	157.26
lat_syscall (write)	1.85	-	-	300.76	166.85
lat_syscall (open/close)	-	-	-	137.90	80.99

단위: 비율 = (가상화 측정값(seconds) / 가상화하지 않은 측정값(seconds))
-: 발표된 결과 값이 없음
‡ : 그래프로부터 추정한 값임

사용하는 VMM으로서, 시험 결과 가장 좋은 성능을 보이는 것으로 나타났다. 본 연구에서 구현한 VMM을 비롯한 나머지 VMM은 모두 게스트 운영체제를 수정하지 않고 가상화를 구현하는 완전 가상화 방식을 사용하기 때문에 Xen on ARM에 비하여 낮은 성능을 보인다.

ViMo는 대역폭 측정 자료가 존재하지 않고 지연시간 측정 자료도 프로세스 문맥 전환 시간을 측정한 자료만 존재하기 때문에 많은 비교가 이루어지지는 못하였다. 프로세스 문맥 전환 지연시간 시험 결과에서는 본 연구에서 구현한 VMM의 성능에 비하여 낮은 것으로 나타났다. ViMo는 완전 가상화 방식을 사용하며, 수행 시간에 게스트의 가상화 민감 명령어를 탐지하고 에뮬레이션 하기 때문에, 컴파일 시간에 게스트 운영체제 커널의 가상화 민감 명령어를 미리 변환하는 본 연구의 방식에 비하여 낮은 성능을 보인다.

SIVARM 또한 대역폭 측정 자료가 존재하지 않기 때문에 지연시간 측정 자료만을 가지고 비교하였으며, 그 결과 본 연구에서 구현한 VMM보다 좋은 성능을 보이는 것으로 나타났다. 그러나 SIVARM은 메모리 가상화를 구현하지 않기 때문에 메모리 가상화에 의한 오버헤드가 전혀 발생하지 않으며, 하나의 게스트 운영체제만 수행할 수 있다는 한계점이 존재한다. 본 연구에서 구현한 VMM은 여러 게스트 운영체제의 주소 변환을 처리해주기 위하여 섀도 페이지 테이블을 사용하여 메모리 가상화를 구현하고 있다. 따라서 게스트 운영체제 수행 시 섀도 페이지 테이블을 사용한 메모리 가상화에 많은 연산이 필요하므로, 본 연구의 시험 결과를 SIVARM에서 발표한 자료와 비교하기에는 무리가 있다.

표 4. dhrystone 시험 결과
Table 4. The result test of dhrystone

	가상화하지 않은 Linux	Xvisor상에서 수행한 Linux	본 연구의 VMM상에서 수행한 Linux
Dhrystones per Second	638841.6	578369.0	600600.6
비율(%)	-	90.53%	94.01%

비율(%) = (가상화한 Linux 측정값 / 가상화하지 않은 Linux 측정값 * 100)

마지막으로 Xvisor는 가장 최근에 개발된 ARM 기반의 VMM으로서, 프로세서 가상화 구현을 위하여 게스트 운영체제 커널의 컴파일 시간에 가상화 민감 명령어를 예외 발생 명령어로 변환하며 섀도 페이지 테이블 기술을 사용하여 메모리 가상화를 구현하는 등, 본 연구에서 구현한 VMM과 가장 유사한 방식을 사용하여 구현되었다. 하지만 아직까지 구현 방법이나 성능에 대하여 공식적으로 발표된 자료가 존재하지 않기 때문에, 본 연구에서는 Xvisor의 성능 측정을 위하여 Xvisor-0.2.0 버전의 소스 코드를 직접 컴파일 하고 본 연구에서 시험한 환경과 동일한 BeagleBoard-xM 상에서 수행시켜 그 성능을 시험하였다. Xvisor의 시험 결과 중 프로세스 문맥 전환 지연시간(lat_ctx)과 페이지 폴트 지연시간(lat_pagefault)에서는 본 연구에서 구현한 VMM보다 좋은 성능을 보였다. 그러나 프로세스 생성 지연시간(lat_proc) 및 시스템 콜 진입 지연시간(lat_syscall) 등의 시험에서는 본 연구에서 구현한 VMM이 더 좋은 성능을 보이는 것으로 나타났다.

2. Dhrystone 시험

dhrystone은 시스템의 전체적인 성능을 측정하는 벤치마크 프로그램이다. 본 시험에서는 dhrystone-2.1 버전을 사용하여, 가상화하지 않은 Linux 커널, 본 연구에서 구현한 VMM 상에서 수행되는 Linux 커널, 그리고 Xvisor 상에서 수행되는 Linux 커널 상에서 각각의 성능을 측정하여 그 비율을 계산하였다. 시험 결과는 표 4와 같다.

표 4의 시험 결과에 의하면 본 연구에서 구현한 VMM 상에서 수행되는 Linux의 전체 시스템의 성능은 가상화하지 않은 Linux에 비하여 약 5.99% 낮은 것으로 나타났다. 또한 Xvisor 상에서 수행되는 Linux의 성능은 가상화하지 않은 Linux에 비하여 약 9.47% 낮은 것으로 나타났다. 비교 결과, 본 연구에서 구현한 VMM의 성능이 Xvisor에 비하여 3.48% 높은 것으로 나타났다.

Ⅶ. 결론

기업과 개인 컴퓨팅 영역에서 주로 연구되던 시스템 가상화 기술이 최근 임베디드 시스템 영역에서도 연구되고 있다. 기존의 임베디드 시스템은 메모리, 프로세서 파워, 배터리 용량 등 하드웨어의 제약으로 인하여 상대적으로 작고 단순하며 주로 한 가지 목적을 위해 사용되었지만, 최근 임베디드 시스템에도 고성능의 하드웨어가 사용됨에 따라 점차 범용적인 목적을 지닌 시스템으로 변화하고 있다. 이에 따라 안정성, 신뢰성 및 보안 등에 대한 요구가 증가하고 있으며 이를 제공하기 위한 기술로서 가상화 기술의 필요성이 점차 증가하고 있다.

본 연구에서는 최근 스마트 디바이스에 주로 사용되는 ARM 구조의 가상화를 위하여 ARMv7 구조의 프로세서, 메모리 및 디바이스를 가상화하는 VMM을 구현하였다. 본 연구에서는 먼저 프로세서 가상화를 위하여 게스트 운영체제 커널의 컴파일 시간에 어셈블리 매크로가 확장된 어셈블리 코드로부터 가상화 민감 명령어를 탐지하여 예외 발생 명령어로 변환함으로써 VMM이 게스트에서 수행되는 가상화 민감 명령어를 에뮬레이션 하도록 하였으며, 또한 메모리 가상화를 위하여 게스트에서 발생하는 Abort 예외와 게스트 운영체제가 수행하는 메모리 관리 명령어를 이용하는 요구 페이징 방식의 섀도 페이지 테이블을 구현하였다. 마지막으로 pass-through 방식으로 UART 디바이스를 가상화하였다.

본 연구에서 구현한 VMM은 Cortex-A8 프로세서가 탑재된 BeagleBoard-xM 상에서 ARM Linux 커널 3.4.4를 가상화하여 수행시켜 성능을 시험하였다. 시험 결과, 반 가상화 방식을 사용하는 Xen on ARM보다는 성능이 낮게 측정되었지만 본 연구와 유사한 완전 가상화 방식을 사용하는 ViMo, SIVARM 및 Xvisor와 비교하여 만족스러운 성능을 보였다.

References

[1] J.E. Smith, R. Nair, "The Architecture of Virtual Machines," IEEE Computer, Vol. 38, No. 5, pp.32-38, 2005.

[2] VMware, Introduction to Virtual Machines, VMware Labs Academic Course Materials, 2010.

[3] J.-Y. Hwang, S.-B. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, C.-R. Kim, "Xen on ARM: System Virtualization using Xen Hypervisor for ARM-based Secure Mobile Phones," Proceedings on 5th IEEE Consumer Communications and Networking Conference, pp.257-261, 2008.

[4] C. Dall, J. Nieh, "KVM for ARM," Proceedings on the Linux Symposium, 2010.

[5] S.-C. Oh, K.H. Kim, K.W. Koh, C.-W. Ahn, "ViMo(Virtualization for Mobile): A Virtual Machine Monitor Supporting Full Virtualization For ARM Mobile System," Proceedings on International Conference of Cloud Computing, pp.48-53, 2010.

[6] S.-H. Jeon, C.-W. Ahn, C.-H. Lee, "Implementation of supporting out of synchronization of shadow page table in ViMo hypervisor based on ARM," Proceedings on The 35th Conference of the Korea Information Processing Society, Vol. 18, No. 1, pp.103-105, 2011 (in Korean).

[7] A. Suzuki, S. Oikawa, "Implementing a Simple Trap and Emulate VMM for the ARM Architecture," Proceedings on IEEE 17th International Conference of Embedded and Real-Time Computing Systems and Applications, Vol. 1, pp.371-379, 2011.

[8] Xvisor: eXtensible Versatile hypervISOR, http://xhypervisor.org

[9] M. Rosenblum, T. Garfinkel, "Virtual Machine Monitors: Current Technology and Future Trends," IEEE Computer, Vol. 38, No. 5, pp.39-47, 2005.

[10] BeagleBoard-xM Rev C System Reference Manual, Revision 1.0, BeagleBoard.org, 2010.

[11] G.J. Popek, R.P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures," Communications of the ACM, Vol. 17, No. 7, pp.412-421, 1974.
[12] ARM Limited, ARM Architecture Reference Manual (v7-AR), ARM DDI 0406B, 2010.
[13] J. Cho, J.-H. Kim, D. Shin, "A Method for Rewriting Sensitive Instructions to Implement ARM Linux Pre-virtualization," Proceedings on The Conference of Institute of Embedded Engineering of Korea 2010, Vol. 1, No. 1, pp.144-147, 2010 (in Korean).
[14] S.-J. Oh, K. Yoon, D. Shin, "A Method for Completely Detecting Sensitive Instructions on ARMv7 Linux Kernel," Prodceedings on The Conference of Institute of Embedded Engineering of Korea 2011, Vol. 1, No. 1, pp.164-167, 2011 (in Korean).
[15] J. Cho, C. Lee, D. Shin, "Implementing Shadow Page Table for Memory Virtualization on ARM Architecture," Proceedings on The Conference of Institute of Embedded Engineering of Korea 2011, Vol. 1, No. 1, pp.483-486, 2011 (in Korean).
[16] J. Cho, "Memory Virtualization for ARM based Virtual Machine," Master of Science Thesis, Sangmyung University, 2012 (in Korean).
[17] J. Cho, S.-J. Oh, D. Shin, "Implementing Shadow Page Tables using TLB Maintenance Operations for Full Virtualization of ARM Architecture," The Journal of Korean Institute of Next Generation Computing, Vol. 8, No. 6, pp.59-68, 2012 (in Korean).
[18] B. Liu, L. Yang, X. Qin, "Research on Hardware I/O Passthrough in Computer Virtualization," Proceedings on the Third International Symposium of Computer Science and Computational Technology, Vol. 2, No. 1, pp.353-356, 2010.
[19] S.-J. Lee, Y.-H. An, A.H. Han, Y.-S. Hwang, K.-S. Chung, "Virtual ARM Machine for Embedded System Development," Journal of IEMEK, Vol. 3, No. 1, pp.19-24, 2008 (in Korean).
[20] L. McVoy, C. Staelin, "lmbench: Portable tools for performance analysis," Proceedings on Annual Technical Conference of the USENIX, 1996.
[21] R.P. Weicker, "Dhrystone: a synthetic systems programming benchmark," Communications of the ACM, Vol. 27, No. 10, pp.1013-1030, 1984.

저 자 소 개

오 승 재

2011년 상명대학교 컴퓨터과학과 이학사.
2013년 상명대학교 일반대학원 컴퓨터과학과 석사.
관심분야: 가상화, 임베디드 컴퓨팅, 리눅스 시스템 프로그래밍.
Email: meganero@gmail.com

신 동 하

1980년 경북대학교 전자공학과 전자계산기 전공 학사.
1982년 서울대학교 전자계산기공학과 석사.
1994년 University of South Carolina 컴퓨터과학과 박사.
1982년 ~ 1996년 한국전자통신연구원 책임연구원.
현재, 상명대학교 컴퓨터과학부 교수.
관심분야: 가상화, 임베디드 컴퓨팅, 논리 프로그래밍, 리눅스 및 윈도우 시스템 프로그래밍.
Email: dshin@smu.ac.kr