韓國컴퓨터情報學會 論文誌
第17卷 第2號, 2012. 2.

2012-17-2-1-1

컴파일 시간 명령어 디코딩을 통한 가상화 민감 명령어 에뮬레이션 성능 개선

신 동 하*, 윤 경 언**

Performance Improvement of Virtualization Sensitive Instruction Emulation by Instruction Decoding at Compile Time

Dongha Shin*, Kyung-un Yun**

요 약

최근 들어 ARM 구조에서 가상화를 구현하기 위해 다양한 연구들이 진행되었다. 현재의 ARM 구조는 전통적인 에뮬레이션 방법인 "trap-and-emulation"으로 가상화 할 수 없기 때문에, 게스트 커널 수행 시간에 가상화 민감 명령어를 탐지하여, 이를 직접 수행하는 대신 가상화 에뮬레이션 한다. 일반적으로 가상화 에뮬레이션은 이진 변환 또는 인터프리테이션 방법으로 구현한다. 본 연구는 인터프리테이션 방법을 기반으로 하는 가상화 에뮬레이션의 성능 향상에 관한 것이다. 인터프리테이션은 명령어 페치, 명령어 디코딩, 그리고 명령어 수행의 단계로 이루어진다. 본 논문에서는 게스트 커널의 컴파일 시간에 모든 가상화 민감 명령어를 디코딩하여, 게스트 커널의 수행 시간에 인터프리테이션 시간을 줄이는 방법을 제안한다. 본 연구의 방법은 인터프리테이션 기반의 가상화 방법에서 에뮬레이션 코드를 간단하게 하고, 에뮬레이션 성능을 향상시킨다.

▸ Keyword : ARM 리눅스, 가상화 에뮬레이션, 임베디드 가상화

Abstract

Recently, we have seen several implementations that virtualize the ARM architecture. Since the current ARM architecture is not possible to be virtualized using the traditional technique called "trap-and-emulation", we usually detect all virtualization sensitive instructions during the run-time of a guest kernel and emulate them virtually rather than executing them directly. The

• 제1저자 : 신동하 • 교신저자 : 신동하
• 투고일 : 2011. 11. 14, 심사일 : 2011. 11. 25, 게재확정일 : 2011. 12. 04
* 상명대학교 컴퓨터과학부(Division of Computer Science, Sangmyung University)
** 상명대학교 일반대학원 컴퓨터과학과(Department of Computer Science, the Graduate School, Sangmyung University)
※ 본 논문은 2010년 상명대학교 교내연구비에 의하여 지원되었습니다.

www.kci.go.kr

emulation for virtualization is usually implemented either by binary translation or interpretation. Our research is about how to improve the performance of emulation for virtualization based on interpretation. The interpretation usually requires a few steps: instruction fetching, instruction decoding and instruction executing. In this paper, we propose a method that decodes all virtualization sensitive instructions during the compilation time of a guest kernel and reduces the time required for interpretation during the run time of the guest kernel. Our method provides both implementation simplicity and performance improvement of emulation for virtualization based on interpretation.

▸ Keyword : ARM Linux, Virtual Emulation, Embedded Virtualization

I. 서 론

기업 및 개인 컴퓨팅 영역에서 인기를 얻고 있는 시스템 가상화는 최근 임베디드 분야에서도 주목받고 있다. 과거의 임베디드 시스템은 주로 특정 목적으로 사용하였지만, 최근 스마트 폰, 스마트 패드와 같은 스마트 기기의 대중화로 범용 목적으로도 사용되면서 소프트웨어의 복잡성이 증가하였다. 이로 인해 기존의 임베디드 시스템에서는 발생하지 않았던 다수의 운영체제를 동시에 수행하는 문제, 운영체제의 보안성 문제, 그리고 소프트웨어 라이선스 분리에 관한 문제가 발생하였는데, 이러한 문제는 가상화 기술을 통해 해결할 수 있었다[1].

최근 임베디드 시스템에서 많이 사용하는 ARM 구조에는 시스템의 모든 자원에 접근할 수 있는 특권 모드(privileged mode)와 자원 접근에 제약이 있는 비특권 모드(non-privileged mode)가 있다. 가상화 환경에서는 가상 머신 모니터가 게스트 운영체제에게 시스템의 자원을 적절히 할당하며 동작하기 위해서 가상 머신 모니터는 특권 모드에서 수행하고, 게스트 운영체제는 비특권 모드에서 수행한다. ARM 구조의 명령어에는 가상화 민감 명령어(virtualization sensitive instruction)가 존재하는데, 이 명령어는 비특권 모드에서 수행했을 때 예외(exception)가 발생하지 않으면서, 프로세서의 상태를 변하게 하고, 또한 프로세서의 모드에 따라 수행 결과가 달라지는 명령어이다. 만약 게스트 운영체제에서 가상화 민감 명령어를 직접 수행하면, 가상 머신 모니터의 프로세서 상태가 변하게 되므로 전체 시스템에 문제가 발생할 수 있다. 그러므로 ARM 구조에서 가상화를 구현하기 위해서는 가상화 민감 명령어를 에뮬레이션하는 과정(이하 가상화 에뮬레이션)이 필요하다.

최근 연구된 ARM 프로세서용 가상 머신 모니터인 KVM/ARM, SIVARM, 그리고 ViMo는 가상화 민감 명령어를 예외 발생 명령어로 변환하고, 변환한 명령어를 수행하면 발생된 예외를 통해 명령어 인출(fetch), 명령어 디코딩, 그리고 명령어 수행의 과정으로 구성된 인터프리테이션 방법으로 가상화 에뮬레이션을 수행한다. 하지만 인터프리테이션 과정에서 디코딩하는 명령어는 커널 컴파일 과정에서 생성되기 때문에, 명령어 디코딩을 커널 컴파일 시간에 미리 수행할 수 있다면 커널 수행 시간에 명령어를 디코딩하는 연산을 생략 할 수 있으므로 가상화 에뮬레이션의 성능을 향상 시킬 수 있다.

본 연구는 기존의 가상화 에뮬레이션 과정에서 커널 수행 시간에 수행하는 가상화 민감 명령어 디코딩 과정을 게스트 커널의 컴파일 시간에 수행하고, 디코딩한 정보를 저장해 두었다가 게스트 커널의 수행 시간에 저장해둔 디코딩 정보를 사용하여 가상화 에뮬레이션을 수행하는 방법을 제안한다. 본 연구에서 제안하는 방법은 커널 수행시간에 가상화 에뮬레이션 과정 중 명령어 디코딩을 생략할 수 있기 때문에 가상화 에뮬레이션 코드가 간단해지면서, 에뮬레이션 성능도 향상 시킬 수 있다.

II. 관련 연구

1. 가상화 민감 명령어

일반적으로 특권 모드(privileged mode)에서 수행하면 예외가 발생하지 않고, 사용자 모드(user mode)에서 수행하면 예외가 발생하는 명령어를 특권 명령어(privileged instruction)라고 하고, 명령어를 수행하였을 때 프로세서의 모드가 명령어를 수행하였을 때의 모드가 아닌 다른 모드로 변하거나 프로세서의 모드에 따라 명령어를 수행한 결과가 다른 명령어를 민감 명령어(sensitive instruction)라고 한다. 만일 민감 명령어의 집합이 특권 명령어 집합의 부분 집합이면, 예외를 발생시켜 에뮬레이션을 수행하는 방법(trap-and-emulation)으로 가상화를 구현할 수 있다[2]. 그러나 ARM 프로세서에서는 민감 명령어 중 특권 명령어에 포함되

www.kci.go.kr

지 않는 명령어인 가상화 민감 명령어가 존재하기 때문에 예외를 발생시켜 에뮬레이션하는 방법으로 가상화 에뮬레이션을 수행할 수 없다. 표 1은 ARMv6의 가상화 민감 명령어를 분류한 표이다.

표 1. ARMv6의 가상화 민감 명령어의 분류
Table 1. Categories of ARMv6 virtualization sensitive instruction

번호	민감 명령어 분류	명령어
1	데이터 처리 명령어	ADCS, ADDS, ANDS, BICS, EORS, MOVS, MVNS, ORRS, RSBS, RSCS, SBCS, SUBS
2	상태 레지스터 접근 명령어	MRS, MSR
3	변환 로드/스토어 명령어	LDRT, LDRBT, STRT, STRBT
4	다중 로드/스토어 명령어	LDM(2), LDM(3), STM(2)
5	예외 처리 명령어	CPS, RFE, SRS
6	코프로세서 접근 명령어	CDP, LDC, MCR, MCRR, MRC, MRRC, STC

데이터 처리 명령어(data-processing instruction)에서 S비트가 설정되고, Rd 레지스터가 프로그램 카운터(PC)이면 CPSR(current processor status register)에 SPSR(saved processor status register)을 복사하는 동작을 수행하는데, 사용자 모드와 시스템 모드는 SPSR을 가지지 않으므로 프로세서의 모드에 따라 수행 결과가 다르다. 상태 레지스터 접근 명령어(status register access instruction)는 CPSR이나 SPSR을 범용 레지스터에 복사하거나, 범용 레지스터의 값을 CPSR이나 SPSR에 복사하는 동작을 수행하는데 사용자 모드와 시스템 모드는 SPSR이 없으므로 프로세서의 모드에 따라 수행 결과가 다르다. 변환 로드/스토어 명령어(load and store with translation instruction)는 특권 모드에서는 사용자 모드의 레지스터를 접근하여 동작하지만, 사용자 모드에서는 다른 모드의 레지스터를 사용하지 않고 본래 사용자 모드의 레지스터를 사용하여 수행되기 때문에 모드에 따라 수행 결과가 다르다. 다중 로드/스토어 명령어(load and store multiple instruction)는 특권 모드에서 사용자 모드의 레지스터를 접근할 수 있지만, 사용자 모드에서 수행하면 수행 결과를 예측할 수 없으므로 모드에 따라 수행 결과가 다르다. 예외 처리 명령어(exception handling instruction)는 명령어가 수행되면 프로세서의 모드가 변하게 되고, 코프로세서 접근 명령어(coprocessor access instruction)는 코프로세서의 레지스터 값을 읽거나 쓰면서 프로세서 모드의 변화를 일으킨다[3][4]. 그러므로 가상화 민감 명령어의 정의에 따라 표 1에 속한 명령어들은 모두 가상화 민감 명령어이다.

2. 기존에 연구된 가상 머신 모니터의 가상화 에뮬레이션

2.1 KVM/ARM의 가상화 에뮬레이션

KVM/ARM[6]은 호스트 커널이 가상 머신 모니터의 역할을 수행하는 가상화 방법이다. KVM/ARM은 가상화 민감 명령어를 수행했을 때 예외를 발생하기 위해 게스트 커널의 컴파일 시간에 가상화 민감 명령어 앞에 ARM 구조의 예외 발생 명령어인 SWI 명령어를 삽입한다. 그 후, 게스트 커널의 수행 시간에 SWI 명령어를 수행하면 예외가 발생하여 호스트의 예외 처리기로 분기하게 되고, 분기한 예외 처리 루틴에서는 SWI 명령어가 가상화 에뮬레이션을 위해 발생한 예외인지 확인한다. 만일 발생한 예외가 가상화 에뮬레이션을 위해 발생한 예외라면 에뮬레이션 루틴으로 분기를 수행하고, 에뮬레이션 루틴에서는 예외를 발생한 SWI 명령어 다음에 위치하는 가상화 민감 명령어를 인출하여 에뮬레이션을 수행한다.

KVM/ARM의 가상화 민감 명령어 에뮬레이션 방식은 게스트 커널의 가상화 민감 명령어 앞에 SWI 명령어를 추가하기 때문에 게스트 커널의 크기가 커지고, 본래의 가상화 민감 명령어를 찾는 과정에서 수행하는 호스트 커널과 게스트 커널 사이의 제어를 교환하기 위한 월드 스위칭(world switching)에 의해 많은 오버헤드가 발생한다.

2.2 SIVARM의 가상화 에뮬레이션

SIVARM[7]은 게스트 커널인 ARM 리눅스에서 사용하지 않는 메모리 영역에 위치하면서 하나의 게스트 커널만 수행하는 가상 머신 모니터이다. SIVARM의 가상화 에뮬레이션은 비특권 민감 명령어(non-privileged sensitive instruction)와 특권 민감 명령어(privileged sensitive instruction)로 분류하여 수행한다. 비특권 민감 명령어는 가상화 에뮬레이션을 수행하기 위해 예외를 발생시켜야 하는데 명령어 스스로 예외를 발생시킬 수 없으므로 커널 컴파일 시간에 가상화 민감 명령어를 게스트 커널에서 사용하지 않는 코프로세서 접근 명령어로 변환한다. 특권 민감 명령어는 비특권 민감 명령어와는 달리 게스트 커널에서 명령어가 수행되면 스스로 예외를 발생하기 때문에 예외 발생 명령어로 변환하지 않는다. 커널 수행 시간에 특권 민감 명령어 또는 커널 컴파일 시 비특권 민감 명령어를 변환했던 코프로세서 접근 명령어를

www.kci.go.kr

수행하면 미확인 명령어 예외(undefined instruction exception)가 발생하여 예외 처리기로 분기하여, 발생한 예외가 가상화 에뮬레이션을 위한 것인지 확인한다. 만일 가상화 에뮬레이션을 위한 예외이면 수행한 명령어를 디코딩하여 에뮬레이션을 수행한다.

SIVARM은 다른 가상 머신 모니터와는 달리 메모리 가상화 방법이 구현되어 있지 않은데, 이는 가상 머신 모니터가 게스트 커널에서 사용하지 않는 메모리 영역에 존재하면서 하나의 게스트 커널만 수행하므로 게스트 커널에서 직접 메모리에 접근하여 연산을 수행하더라도 다른 게스트 커널에 영향을 줄 염려가 없기 때문이다. 하지만 SIVARM의 메모리 가상화가 구현되어 있지 않은 점에 의해 다수의 게스트 커널이 동작하는 가상화 환경에서는 사용할 수 없다.

2.3 ViMo의 가상화 에뮬레이션

ViMo[8]는 KVM/ARM과 SIVARM과는 달리 게스트 커널을 수정하지 않고 가상화를 수행하는 완전 가상화(full virtualization) 방식의 가상 머신 모니터이다. ViMo는 게스트 커널을 기본 블록(basic block) 단위로 읽어서 수행하는 이진 변환(binary translation) 방식을 사용하지만, 가상화 에뮬레이션은 인터프리테이션 방법을 사용한다. ViMo는 커널 컴파일 시간에 가상화 에뮬레이션을 위해 특별히 수행하는 동작은 없고, 커널 수행 시간에 게스트 커널을 기본 블록 단위로 읽어서 해당 블록에 가상화 민감 명령어가 포함되어있는지 스캔한다. 만약 기본 블록에서 가상화 민감 명령어를 발견하면 가상화 민감 명령어를 저장하는 RIT(replacement instruction table)에 해당 가상화 민감 명령어를 저장하고, RIT에서 명령어가 저장된 위치의 오프셋 값을 가지는 SWI 명령어로 가상화 민감 명령어를 변환한다. 모든 가상화 민감 명령어의 스캔을 마친 기본 블록은 기본 블록 캐시(basic block cache)에 저장한다. ViMo는 기본 블록 캐시에 저장된 내용을 수행하여 게스트 커널을 수행하는데, 게스트 커널이 수행 중에 가상화 민감 명령어를 변환한 SWI 명령어를 수행하면 예외를 발생하여 예외 처리기로 분기한다. 예외 처리기에서는 발생한 예외가 가상화 에뮬레이션을 위한 예외인지 확인하고, 가상화 에뮬레이션을 위한 예외라면 SWI 명령어에 저장된 RIT의 오프셋 값을 사용하여 본래의 가상화 민감 명령어를 인출하여 에뮬레이션을 수행한다.

ViMo의 방법은 커널 수행 시간에 수행할 코드가 기본 블록 캐시에 저장되어 있지 않으면 위와 같은 복잡한 과정을 수행해야 하기 때문에 처음 수행하는 코드에 대하여 오버헤드가 발생하고, 구현이 복잡하다는 단점이 있다.

3. 기존 가상화 에뮬레이션 방법의 문제점

기존에 연구된 KVM/ARM, SIVARM, 그리고 ViMo의 에뮬레이션 방법은 그림 1과 같이 디코드 앤드 디스패치 인터프리테이션[5]을 사용한다.

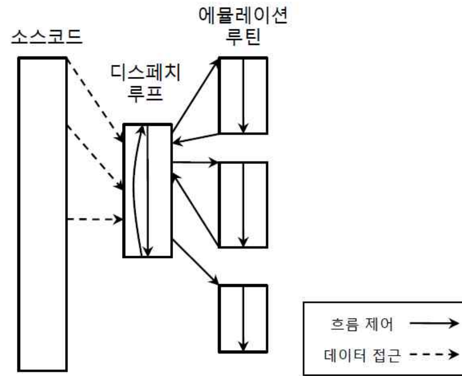


그림 1. 디코드 앤드 디스패치 인터프리테이션
Fig. 1. Decode and dispatch interpretation

그림 1의 디코드 앤드 디스패치 인터프리테이션은 명령어를 에뮬레이션 하기위해 디스패치 루프에서 소스코드의 명령어를 하나씩 읽어 명령어를 식별하는 과정을 수행하는데, 이 과정에서 경우에 따라 한 번의 비교 연산으로 명령어를 식별할 수 있지만 최악의 경우 마지막 비교 연산까지 모두 수행해야 할 수 있다. 또한 에뮬레이션을 수행할 때마다 '1)소스 코드의 명령어 인출 -> 2)디스패치 루프 수행 -> 3)에뮬레이션 루틴 수행-> 4)디스패치 루프로 복귀 -> 5)1)의 과정 수행'과 같은 과정을 수행하기 위해 다수의 분기를 수행한다. 그러므로 디코드 앤드 디스패치 인터프리테이션은 에뮬레이션 과정에서 발생하는 지연으로 인해 속도가 느리다는 단점이 있다.

본 장의 2.2절에서 설명한 가상 머신 모니터들은 예외를 발생하기 위해 가상화 민감 명령어 앞에 예외 발생 명령어를 추가하거나 가상화 민감 명령어를 예외 발생 명령어로 변환한다. 그 후 게스트 커널의 수행 시간에 '1)게스트 커널 수행 -> 2)예외 발생 명령어 수행 -> 3) 예외 처리기 수행 -> 4)본래의 가상화 민감 명령어 인출 -> 5)가상화 민감 명령어 디코딩 -> 6)에뮬레이션 루틴 수행 -> 7)게스트 커널로 복귀 -> 8) 1)의 과정 수행'의 과정으로 가상화 에뮬레이션을 수행하는데, 이 방법은 디코드 앤드 디스패치 인터프리테이션이므로 이에 대한 개선이 필요하다.

www.kci.go.kr

III. 커널 컴파일 시간 가상화 민감 명령어 디코딩

본 연구에서는 가상화 에뮬레이션을 수행하기 위해서 커널 컴파일 시간에 가상화 민감 명령어를 예외 발생 명령어로 변환하고, 커널 수행 시간에 변환한 명령어를 통해 예외를 발생시켜 에뮬레이션을 수행하는 방법을 사용한다. 기존의 방법과 다른 점은 ARM의 예외 발생 명령어인 SWI 명령어에 가상화 민감 명령어를 디코딩한 정보를 포함시킨다는 점이다. 그림 2는 본 연구에서 사용하는 SWI 명령어의 형식이다.

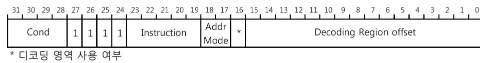


그림 2. 가상화 에뮬레이션을 위한 SWI 명령어의 형식
Fig. 2. Formats of SWI instruction at for virtual emulation

가상화 민감 명령어를 SWI 명령어로 변환할 때 SWI 명령어의 조건 비트인 [31:28]비트는 본래 가상화 민감 명령어와 동일한 값을 저장하여 SWI 명령어가 본래의 명령어와 동일한 조건일 때 수행되도록 하고, [27:24]비트의 OP 코드에는 SWI 명령어의 OP코드인 1111값을 저장한다. 명령어를 디코딩한 정보는 [23:19]의 5비트에 저장하고, [18:17]의 2비트에는 주소지정 방식을 디코딩한 정보를 저장한다. 단, 상태 레지스터 접근 명령어의 경우 주소지정 방식을 사용하지 않기 때문에 [18:17]비트에 CPSR 및 SPSR 사용 여부와 관련된 정보를 저장하여 에뮬레이션의 효율을 높인다. 나머지 [16:0]의 17비트에는 피연산자를 디코딩한 정보를 저장하는데, 피연산자 디코딩 정보는 명령어에 따라 [16:0]에 모두 저장할 수 없을 수 있다. 그러므로 16번 비트는 모든 피연산자를 저장할 수 있는지 여부를 기록한다. 만일 모든 피연산자를 저장할 수 없으면 피연산자를 특정 메모리 영역(이하 디코딩 영역)에 저장하고, 디코딩 영역의 시작 주소로부터 디코딩한 피연산자가 저장된 위치의 오프셋을 변환할 SWI 명령어의 [15:0]비트에 저장한다.

가상화 민감 명령어는 [27:26] 비트를 통해 명령어의 분류를 식별하는데 데이터 처리 명령어와 상태 레지스터 접근 명령어는 [27:26] 비트가 00으로 동일하므로 20번 비트를 추가적으로 확인한다. 그림 3은 [27: 26] 비트가 00이고, 20번 비트가 1인 데이터 처리 명령어의 인코딩 형식이다.

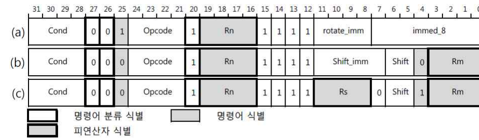


그림 3. 데이터 처리 명령어와 주소지정 방식1의 형식
Fig. 3. Formats of data-processing instructions and addressing mode1

데이터 처리 명령어는 가상화 민감 명령어가 되는 조건인 25번 비트의 값이 1인 것과 [15:12]비트가 1111인 것을 다른 값으로 수정하면 수정한 명령어를 직접 수행한 결과를 에뮬레이션의 결과로 사용할 수 있다. 그러므로 각 명령어에 대한 에뮬레이션 루틴은 필요하지 않고, 명령어 디코딩 정보는 00000 하나만 사용한다. 그림 3과 같이 데이터 처리 명령어는 25번 비트와 4번 비트를 통해 주소 지정 방식1의 형식인 (a)32-bit immediate, (b)immediate shifts, 그리고 (c)register shifts을 식별할 수 있는데, 주소지정 방식에 따라 사용하는 피연산자가 다르므로 명령어 디코딩 시 이를 충분히 고려해야 한다. 다른 가상화 민감 명령어에서는 디코딩 영역에 명령어의 피연산자를 저장하는 것과는 달리 데이터 처리 명령어는 직접 수행 가능한 형태로 변환한 명령어를 저장하고, 에뮬레이션 수행 시 해당 명령어를 직접 수행하면 되므로 주소 지정 방식에 대한 디코딩 정보는 필요하지 않다.

상태 레지스터 접근 명령어는 데이터 처리 명령어와 동일하게 [27:26] 비트가 00이지만 20번 비트가 0인 것을 통해 구분한다. 그림 4는 상태 레지스터 접근 명령어의 인코딩 형식이다.

그림 4. 상태 레지스터 접근 명령어의 형식
Fig. 4. Formats of status register access instructions

상태 레지스터 접근 명령어는 25번, 21번 비트를 사용하여 (a)MRS, (b)MSR(immediate operand), 그리고 (c)MSR(register operand)를 식별하고, 주소지정 방식은 사용하지는 않는다. 상태레지스터 명령어의 명령어 디코딩 정보는 (a)00001, (b)와 (c)는 00010을 사용하고, 22번 비트를 확인하여 (a)에서 SPSR관련 연산이면 00, CPSR 관련 연산이면 01을, (b)의 SPSR 관련 연산이면 10, CPSR 관련 연산이면 11을, 그리고 (c)의 SPSR 관련 연산이면 00, CPSR 관련 연산이면 01을 주소지정 방식의 디코딩 정보에 저장한다. 상태 레지스터 접근 명령어의 피연산자들은 총 16bit보다 적기 때문에, 디코딩 영역을 사용하지 않고, SWI

www.kci.go.kr

명령어의 [15:0]비트에 모든 피연산자를 저장할 수 있다.

그림 5는 변환 로드/스토어 명령어로 [27:26] 비트의 값이 01인 가상화 민감 명령어이다.

그림 5. 변환 로드/스토어 명령어의 형식
Fig. 5. Formats of load and store with translation instructions

그림 5의 변환 로드/스토어 명령어는 22번 비트와 20번 비트의 값을 사용하여 (a)LDRBT, (b)LDRT, (c)STRBT, 그리고 (d)STRB를 구분하고, 명령어 디코딩 정보는 (a)00011, (b)00100, (c)00101, 그리고 (d)00110을 사용한다.

변환 로드/스토어 명령어는 그림 6의 주소지정 방식 2를 사용한다.

그림 6. 주소지정 방식 2의 형식
Fig. 6. Formats of addressing mode2

그림 6의 주소지정 방식 2는 25번 비트와 [11:5]비트를 사용하여 (a)immediate post-indexed 방식, (b)register post-indexed 방식, 그리고 (c)scaled register post-indexed 방식을 식별한다. 주소지정 방식 2의 디코딩 정보는 (a)00, (b)01, 그리고 (c)10을 사용하는데, (b)register post-indexed 방식의 경우 피연산자 크기는 합이 16비트보다는 작지만 (c)scaled register post-indexed 방식과 연산 방법의 유사성으로 인해 코드 작성의 편의상 주소지정 방식 2는 모두 디코딩 영역을 사용하여 피연산자들을 저장한다.

다중 로드/스토어 명령어는 [27:26] 비트가 10인 명령어로 그림 7과 같은 형식을 가진다.

그림 7. 다중 로드/스토어 명령어의 형식
Fig. 7. Formats of load and store multiple instructions

다중 로드/스토어 명령어는 20번 비트와 15번 비트를 사용하여 (a)LDM(2), (b)LDM(3), 그리고 (c)STM(2)을 식별한다. 명령어 디코딩 정보는 (a)00111, (b)01000, 그리고 (c)01001을 가진다.

다중 로드/스토어 명령어는 그림 8과 같은 주소지정 방식 4를 사용한다.

그림 8. 주소지정 방식4의 형식
Fig. 8. Formats of addressing mode4

그림 8의 주소지정 방식4는 [24:23] 비트로 식별하고, (a)increment After, (b)increment before, (c)decrement after, 그리고 (d)decrement before가 있다. 주소지정 방식4의 디코딩 정보는 (a)00, (b)01, (c)10, 그리고 (d)11을 사용하는데, 모든 피연산자가 16비트를 초과하므로 디코딩 영역을 사용하여 피연산자들을 저장한다. 단 피연산자 중 21번 비트인 W비트는 LDM(2) 명령어와 STM(2) 명령어에서는 사용하지 않는다.

예외 처리 명령어는 그림 9와 같이 [31:28] 비트가 1111인 명령어들이다.

그림 9. 예외 처리 명령어의 형식
Fig. 9. Formats of exception processing instructions

예외 처리 명령어는 27번 비트와 20번 비트를 사용하여 (a)CPS, (b)RFE, 그리고 (c)SRS로 식별하고, 명령어 디코딩 정보는 (a)01010, (b)01011, 그리고 (c)01100을 사용한다. 예외 처리 명령어는 [24:23] 비트를 식별하여 그림 8의 주소지정 방식 4를 사용한다. 예외 처리 명령어는 피연산자의 크기가 16비트 보다 작으므로 SWI 명령어에 모든 피연산자를 저장할 수 있다.

가상화 민감 명령어에서 [27:26] 비트가 11인 경우 그림 10의 코프로세서 접근 명령어이다.

www.kci.go.kr

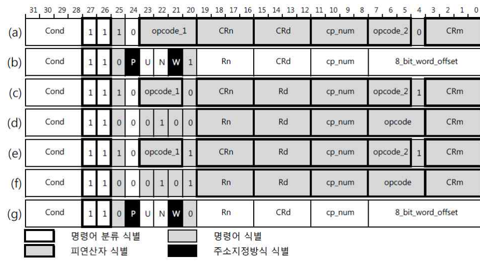


그림 10. 코프로세서 접근 명령어의 형식
Fig. 10. Formats of coprocessor access instructions

코프로세서 접근 명령어는 25번 비트와 4번 비트로 식별하는 (a)CDP, 25번 비트와 20번 비트로 식별하는 (b)LDC와 (g)STC, 25번, 20번 그리고 4번 비트로 식별하는 (c)MCR과 (e)MRC, 그리고 25번 비트와 [23:20]비트로 식별하는 (d)MCRR과 (f)MRRC가 있다. 코프로세서 접근 명령어의 명령어 디코딩 정보는 (a)01101, (b)01110, (c)01111, (d)10000, (e)10001, (f)10010 그리고 (g)10011을 사용하고, 코프로세서 접근 명령어의 피연산자들은 모두 16비트를 초과하므로 디코딩 영역을 사용하여 피연산자를 저장한다.

코프로세서 접근 명령어의 LDC와 STC는 그림 11의 주소지정 방식5를 사용한다.

그림 11. 주소지정 방식 5의 형식
Fig. 11. Formats of addressing mode 5

주소지정 방식 5는 24번 비트와 21번 비트를 사용하여 (a)immediate offset, (b)immediate pre-indexed, (c)immediate post-indexed, 그리고 (d)unindexed 방식으로 식별한다. 주소지정 방식 5의 디코딩 정보는 (a)00, (b)01, (c)10, 그리고 (d)11을 사용한다.

가상화 민감 명령어의 피연산자 중에는 특정 레지스터에 저장된 값을 사용하는 경우와 같이 게스트 커널이 수행되는 과정에서 값이 동적으로 변하는 형태가 있는데 이러한 경우 피연산자의 디코딩 없이 본래 가상화 민감 명령어의 레지스터 번호 그대로 사용해야 한다. 하지만 MSR 명령어의 field_mask와 관련된 연산에서 사용하는 특정한 마스크 값과 같이 디코딩 결과 값이 항상 동일함에도 불구하고 다수의 연산을 통해 해당 값을 생성하는 경우가 있는데, 이러한 경우는 연산의 특정 부분까지 미리 연산을 수행한 결과를 저장해 두고 사용하면 가상화 에뮬레이션의 성능을 향상할 수 있다.

3장에서 설명한 가상화 민감 명령어의 디코딩 정보와 주소지정 방식 디코딩 정보는 표 2와 표 3과 같이 요약할 수 있다.

표 2. 가상화 민감 명령어의 디코딩 정보
Table 2. Decoding values of virtualization sensitive instructions

명령어 분류	명령어	[23:19]
데이터 처리 명령어	-	00000
상태 레지스터 접근 명령어	MRS	00001
	MSR	00010
변환 로드/스토어 명령어	LDRBT	00011
	LDRT	00100
	STRBT	00101
	STRT	00110
다중 로드/스토어 명령어	LDM(2)	00111
	LDM(3)	01000
	STM(2)	01001
예외 처리 명령어	CPS	01010
	RFE	01011
	SRS	01100
코프로세서 접근 명령어	CDP	01101
	LDC	01110
	MCR	01111
	MCRR	10000
	MRC	10001
	MRRC	10010
	STC	10011

표 3. 주소지정 방식의 디코딩 정보
Table 3. Decoding values of addressing mode

분류	주소지정 방식	[18:17]
주소지정 방식2	Immediate post-indexed	00
	Register post-indexed	01
	Scaled register post-indexed	10
주소지정 방식4	Increment After	01
	Increment Before	11
	Decrement After	00
	Decrement Before	10
주소지정 방식5	Immediate offset	10
	Immediate Pre-indexed	11
	Immediate post-indexed	01
	Unindexed	00
MRS	SPSR 관련 연산	00
	CPSR 관련 연산	01
MSR(Immediate operand)	SPSR연산	10
	CPSR연산	11
MSR(Register operand)	SPSR연산	00
	CPSR연산	01

www.kci.go.kr

IV. 커널 수행 시간 가상화 에뮬레이션 방법 비교

그림 12는 2.2절에서 설명한 기존의 가상 머신 모니터에서 커널 수행 시간에 가상화 에뮬레이션을 수행하는 과정이다.

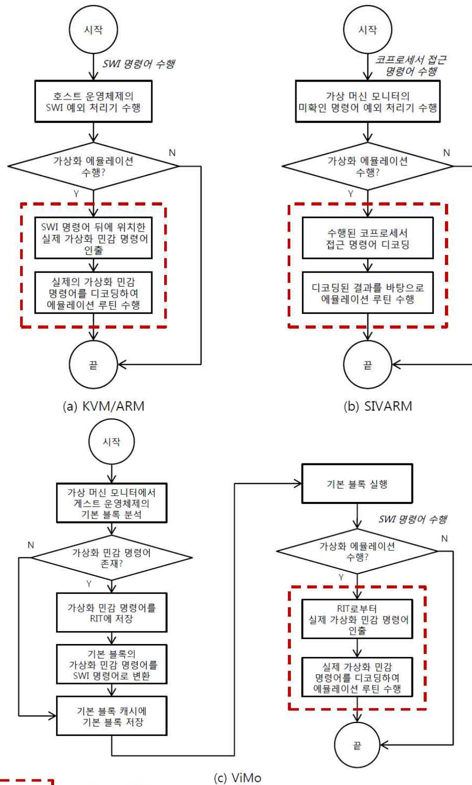


그림 12. 기존의 가상 머신 모니터에서의 가상화 에뮬레이션 과정
Fig. 12. Virtual emulation processes of introduced virtual machine monitor

그림 12의 기존에 소개된 가상 머신 모니터에서는 커널 수행 시간에 가상화 민감 명령어를 변환한 예외 발생 명령어를 수행하면 가상화 에뮬레이션을 수행하기 위해 본래의 가상화 민감 명령어를 인출한다. 그 후 인출한 명령어를 다수의 비교문과 시프트연산을 사용하여 명령어, 주소지정 방식, 그리고 피연산자를 디코딩한다. 하지만 디코딩한 명령어의 정보, 주소 지정 방식의 정보 그리고 특정한 피연산자의 정보는 디코딩할 때마다 결과가 동일하므로 명령어를 에뮬레이션 할 때마다 매번 디코딩 하는 것은 오버헤드이다. 이러한 과정은 그림 12의 오버헤드 발생 구간에서 수행된다.

그림 13은 본 연구에서 제안하는 커널 컴파일 시간에 명령어를 디코딩한 정보를 사용하여, 커널 수행 시간에 가상화 에뮬레이션을 수행하는 과정이다.

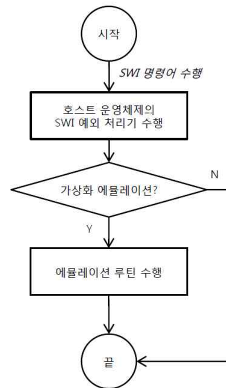


그림 13. 커널 컴파일 시간 디코딩 정보를 활용한 가상화 에뮬레이션 과정
Fig. 13. Virtual emulation process using kernel compile time decoding value

그림 13의 방법은 커널 수행 시간에 SWI 명령어를 수행하면 기존의 가상화 에뮬레이션 방법과 동일하게 SWI 예외처리기로 분기하여 가상화 에뮬레이션을 수행해야 하는지 확인한다. 만일 가상화 에뮬레이션을 수행할 필요가 있으면 기존의 가상화 에뮬레이션 방법과는 달리 본래의 명령어를 인출하지 않고, SWI 명령어에 포함된 명령어의 디코딩 정보, 주소지정 방식의 디코딩 정보, 그리고 피연산자의 디코딩 정보를 식별하여 에뮬레이션에 사용한다. 이러한 디코딩 정보를 사용함으로써 기존의 가상화 에뮬레이션 방식에서는 가상화 민감 명령어에서 '(마스크 연산의 수 + 시프트 연산의 수) * 디코딩하는 비트 수' 만큼의 연산을 수행하여 디코딩해야 했던 과정을 생략할 수 있다. 본 논문에서 제안하는 가상화 에뮬레이션 방식을 통해 그림 12의 오버헤드 발생 구간을 그림 13과 같이 단순화 시킬 수 있다.

V. 성능분석

본 연구에서 제안하는 방법의 성능을 분석하기 위해서 프로그램의 수행 시간에 가상화 민감 명령어를 디코딩(이하 기존의 방법)하여 가상화 에뮬레이션 하는 프로그램과 미리 디코딩한 가상화 민감 명령어의 정보를 사용하여 가상화 에뮬레이션(이하 본 논문의 방법)하는 프로그램을 구현하였다. 구현한 프로그램은 QEMU[9] 에뮬레이터에서 수행하였다. 본 연구에서 구현한 기존의 방법을 사용한 테스트 프로그램은 명령

www.kci.go.kr

어를 식별하는데 코드 길이가 109줄(line)인 함수를 사용했으나 본 논문의 방법을 적용하였을 때는 해당 함수의 코드 길이가 20줄로 줄었다. 이는 많은 비교문을 사용하는 명령어 디코딩을 커널 컴파일 시간에 수행하기 때문에 커널 수행 시간에는 이러한 과정이 생략되었기 때문이다.

표 4는 GDB 디버거[10]를 수정하여 측정한 본 연구에서 구현한 시험용 프로그램에서 가상화 에뮬레이션 수행 시 수행하는 명령어 개수의 명령어 분류별 평균이다.

표 4. 가상화 에뮬레이션 과정에서 수행하는 명령어 개수의 평균
Table 4. The average number of executed instructions in virtual emulation (단위: 개)

번호	명령어 분류	기존의 방법 (A)	본 논문의 방법(B)
1	데이터 처리 명령어	47	44
2	상태 레지스터 접근 명령어	62	52
3	변환 로드/스토어 명령어	69	70
4	다중 로드/스토어 명령어	282	279
5	예외 처리 명령어	58	58
6	코프로세서 접근 명령어	110	94

A: 커널 수행 시간에 명령어 디코딩을 수행하는 가상화 에뮬레이션 시험용 프로그램에서 수행한 명령어 개수의 평균
B: 미리 명령어를 디코딩한 정보를 사용하여 가상화 에뮬레이션을 수행하는 시험용 프로그램에서 수행한 명령어 개수의 평균

표 4의 수치는 가상화 민감 명령어와 주소지정 방식을 조합하여 생성할 수 있는 모든 경우인 55개의 가상화 민감 명령어를 테스트 프로그램에서 수행한 결과이다. 표 4에 따르면 번호 3과 5의 경우 본 논문의 방법이 기존의 방법에 비해 가상화 에뮬레이션 시 수행한 명령어의 개수가 더 많거나 동일하였다. 이는 해당 분류의 명령어는 기존의 가상화 에뮬레이션 방법을 사용하는 테스트 프로그램의 수행 시간에 가상화 민감 명령어를 디코딩하는 연산이 많지 않아서 본 논문의 가상화 에뮬레이션 방법을 사용하는 테스트 프로그램에서 SWI 명령어에 저장된 명령어 디코딩 정보, 주소 지정 방식 디코딩 정보, 그리고 피연산자 디코딩 정보를 식별하는 과정이 오버헤드로 작용했기 때문이다.

표 5는 KVM/ARM의 게스트 커널(linux 2.6.17)이 수행되어 커널을 무한 루프의 대기 상태로 만드는 함수 rest_init()을 호출하기 전까지 수행한 가상화 민감 명령어의 개수다.

표 5. KVM/ARM의 게스트 커널에서 수행하는 가상화 민감 명령어의 개수
Table 5. The number of executed virtualization sensitive instructions on KVM/ARM

번호	명령어 분류	게스트 커널에서 수행하는 가상화 민감 명령어의 개수(C)	비율 (%)
1	데이터 처리 명령어	48	0.1
2	상태 레지스터 접근 명령어	38,456	59.3
3	변환 로드/스토어 명령어	1,326	2.0
4	다중 로드/스토어 명령어	48	0.1
5	예외 처리 명령어	19,861	30.6
6	코프로세서 접근 명령어	5,119	7.9
총 계		64,858	100.0

표 5에 따르면 KVM/ARM의 게스트 커널이 수행한 전체 가상화 민감 명령어 중 상태 레지스터 접근 명령어와 예외 처리 명령어가 약 90%를 차지한다. 이는 비록 표 4의 3번과 5번의 명령어 분류의 경우 성능이 좋아지지는 않았지만 게스트 커널에서 수행 비율이 높은 상태 레지스터 접근 명령어와 그 외의 명령어 분류들에서 성능 향상이 있었기 때문에 본 논문에서 제안하는 방법을 사용하면 게스트 커널의 성능을 향상시킬 수 있다.

표 6은 기존의 가상화 에뮬레이션 방법과 본 논문에서 제안하는 가상화 에뮬레이션 방법을 사용할 때 게스트 커널에서 가상화 에뮬레이션 시 수행하는 명령어의 개수를 분석한 표이다.

표 6. KVM/ARM의 가상화 에뮬레이션 과정에서 수행하는 명령어의 개수
Table 6. Counts of executing instructions at virtual emulation on KVM/ARM

번호	명령어 분류	기존의 방법 (D)	본 논문의 방법(E)	명령어 감소비율(F)
1	데이터 처리 명령어	2,256	2,112	-6.4%
2	상태 레지스터 접근 명령어	2,384,272	1,999,712	-16.1%
3	변환 로드/스토어 명령어	91,494	92,820	1.5%
4	다중 로드/스토어 명령어	13,536	13,392	-1%
5	예외 처리 명령어	1,151,938	1,151,938	0%
6	코프로세서 접근 명령어	563,090	481,186	-14.6%
총 계		4,206,586	3,741,160	-11.1%

D: 게스트 커널에서 기존의 가상화 에뮬레이션 방법을 사용할 경우 수행하는 명령어의 개수

www.kci.go.kr

E: 게스트 커널에서 본 논문의 가상화 에뮬레이션 방법을 사용할 경우 수행하는 명령어의 개수
F: 본 논문의 방법이 기존의 방법보다 가상화 에뮬레이션 수행 시 감소하는 명령어 개수의 비율

표 6의 결과는 'D = (표 4의 A) * (표 5의 C)', 'E = (표 4의 B) * (표 5의 C)' 그리고 'F = -((D - E) / D)'의 수식을 사용하여 계산하였다. 본 연구에서 제안하는 방법은 변환 로드/스토어 명령어의 경우 수행 명령어가 1.5% 증가하였지만, 데이터 처리 명령어 6.4%, 상태 레지스터 접근 명령어 16.1%, 다중 로드/스토어 명령어 1%, 그리고 코프로세서 접근 명령어 14.6%의 게스트 커널의 수행 명령어 감소로 인해 전체적으로 게스트 커널의 수행 명령어가 11.1% 감소했으므로 본 논문의 방법을 게스트 커널에 적용하였을 때 가상화 에뮬레이션 성능향상을 기대할 수 있다.

VI. 결론

최근 임베디드 시스템이 범용적으로 사용됨에 따라 발생하는 다양한 문제들을 가상화 기술을 통해 해결할 수 있었다. 이러한 가상화 기술을 ARM 구조에서 사용하기 위해서 가상화 에뮬레이션의 과정이 필요한데, 기존의 가상 머신 모니터에서는 커널 수행 시간에 명령어 인출, 명령어 디코딩, 그리고 명령어 수행의 인터프리테이션 방법을 사용하였다.

본 연구에서는 기존에 연구된 가상 머신 모니터의 가상화 에뮬레이션에서 사용하는 인터프리테이션 방법의 명령어 디코딩 과정을 게스트 커널의 컴파일 시간에 수행하고, 디코딩 정보를 저장해 두었다가 커널 수행 시간에 저장해 둔 디코딩 정보를 사용하여 가상화 에뮬레이션을 수행하는 방법을 제안하였다. 본 연구의 방법을 사용하면 기존의 가상화 에뮬레이션 코드에서 수행되었던 복잡한 형태의 명령어 디코딩 과정이 생략되므로 에뮬레이션 코드 구현이 간단해진다. 성능적으로는 비록 일부의 명령어 분류에서 본 논문의 방법을 사용하였을 때 가상화 에뮬레이션 성능이 향상되지는 않았지만, 실제 게스트 커널의 수행 시간에 수행하는 가상화 민감 명령어의 약 59.3%를 차지하고 있는 상태 레지스터 접근 명령어에서 많은 성능 향상이 있었고, 이를 바탕으로 게스트 커널에서의 성능 향상 여부를 측정한 결과 본 논문의 가상화 에뮬레이션 방법을 사용하였을 때 기존의 가상화 에뮬레이션 방법을 사용하였을 때보다 가상화 에뮬레이션 시 수행하는 명령어의 개수가 11.1% 감소하였다.

본 연구의 결과는 게스트 커널의 수행 시간에 높은 비율로 수행되는 가상화 민감 명령어의 성능 향상을 통해 전체 게스트 커널의 성능 향상을 이룰 수 있었다. 하지만 특정 명령어의 경우 기존의 가상화 에뮬레이션 방법이 더 효율적이었는데, 이는 명령어에서 사용하는 피연산자의 특성에 따라 효율적인 에뮬레이션 방법이 다를 수 있기 때문이다. 그러므로 향후에는 각 가상화 민감 명령어에 대하여 효율적인 가상화 에뮬레이션 방법에 대한 연구를 통해 모든 가상화 민감 명령어의 가상화 에뮬레이션 성능 향상 방안을 모색할 필요가 있다.

참고문헌

[1] Gernot Heiser, "The Role of Virtualization in Embedded Systems," Proceedings of the 1st workshop on Isolation and integration in embedded systems, pp. 11-16, NY, USA, 2008.4.

[2] G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," Communications of the ACM, Vol. 17, Issue 7, pp. 312-421, 1974.7.

[3] ARM Limited, "ARM Architecture Reference Manual," ARM DDI 0100I, 2005.

[4] Dongha Shin and Changhoon Lee, "Encoding Virtualization Sensitive Instructions on ARM Architecture," International Transaction on Computer Science & Engineering, Vol.63, No.1, 2011.2.

[5] James E. Smith, Ravi Nair, "Virtual Machines Versatile Platforms for Systems and Processes," Elsevier Inc, pp. 29-46, 2005.

[6] Christoffer Dall and Jason Nieh, "KVM for ARM," Proceedings of the Linux Symposium, Ottawa, Canada, 2010.7.

[7] Akihiro Suzuki, Shuichi Oikawa, "Implementation of SIVARM: a Simple VMM for the ARM Architecture," Proceedings of First International Conference on Networking and Computing, Higashi-Hiroshima, Japan, 2010.11.

[8] Soo-Cheol Oh, KangHo Kim, KwangWon Koh, and Chang-Won Ahn, "ViMo(Virtualization for Mobile): A Virtual Machine Monitor Supporting Full Vir

www.kci.go.kr

tualization For ARM Mobile System," Proceedings of Cloud Computing 2010: The First International Conference on Cloud Computing, GRIDs, and Virtualization, Lisbon, Portugal, 2010.11.

[9] QEMU opensource processor emulator, http://wiki.qemu.org/Main_Page

[10] GDB: The GNU Project Debugger, http://www.gnu.org/s/gdb

[11] Sunghoon Son, Jaehyeon Lee, "Design and Imple mentation of Virtual Machine Monitor for Embed ded Systems," Journal of The Korea Society of Co mputer and Information, Vol.14, No.1, pp.57-64, 2009.

[12] In Hwan Doh, "Implementation of the Hibernati on-based Boot Mechanism on an Embedded Linux System," Journal of The Korea Society of Computer and Information, Vol16, No.5, pp.23-31, 2011.

저 자 소 개

신 동 하

1980년 : 경북대학교 전자공학과 전자계산기 전공 학사

1991년 : 서울대학교 전자계산기공학과 석사

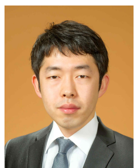
1994년 : University of South Carolina 컴퓨터과학과 박사

1982년 ~ 1996년 : 한국전자통신연구원 책임연구원

1997년 ~ 현재 : 상명대학교 컴퓨터과학부 교수

관심분야: 가상화, 임베디드 컴퓨팅, 논리프로그래밍, 리눅스 및 윈도우 시스템 프로그래밍

Email : dshin@smu.ac.kr



윤 경 언

2010년 : 상명대학교 컴퓨터과학과 이학사

현재 : 상명대학교 일반대학원 컴퓨터과학과 석사과정

관심분야: 가상화, 운영체제, 리눅스 시스템프로그래밍

Email : grans84@gmail.com

www.kci.go.kr

www.kci.go.kr