하이퍼바이저 기술에 대한 고찰

장주현(JuHyeon Jang)
카이스트 운영체제 연구실
http://oslab.kaist.ac.kr
cjdool@kaist.ac.kr

ABSTRACT

Many modern systems use virtualization to subdivide the sufficient resources of a computing systems. The small part of systems is called *virtual machines*(VMs). Virtualization is designed to achieve a number of objectives such as isolation, high availability, workload balancing, and sandboxing. To achieve the virtualization, a machine introduces a new software layer called *hypervisor* which is often called as *virtual machine monitor*. Similar to operating systems, hypervisor provides CPU, memory, interrupt, timer, and IO virtualization. Basic principle of hypervisor is *trap-and-emulate* strategy, which is used by virtual machines to emulate the privileged instructions and the register and to pretend to the OS that it is still in kernel mode. There exist software-based hypervisors which use binary translation or adaptive binary translation instead of trap-and-emulate strategy. Another category of virtualization is paravirtualization. Paravirtualization modifies the operating system of virtual machines for improving the performance of the hypervisor. As virtualization is important from servers to embedded systems, hardware manufactures have supported hardware extensions for trap-and-emulate strategy. Intel adds VT-x and VT-d virtualization extensions, AMD adds AMD-V virtualization extensions, and ARM adds ARM virtualization extensions for hardware supported trap-and-emulate virtualization. In this paper, we examine the essential hypervisor techniques and hardware virtualization extensions. Hypervisor can be divided into two categories; *type 1* and *type 2*. Type 1 hypervisor runs directly on the host machine's physical hardware, and it is referred to as a bare-metal hypervisor. Type 2 hypervisor is typically installed on top of the existing operating systems, and it is sometimes called a hosted hypervisor. Type 2 hypervisor relies on the host machine's OS to manage calls to CPU, memory, storage, and network resources. Typical type 1 and type 2 hypervisors correspond to Xen and KVM, respectively. In this paper, we present Type1 and Type2 hypervisor architectures and virtualization techniques. Although hypervisor is responsible for IO virtualization, KVM hypervisor use different IO virtualization methods. In this work, we also deal with typical IO virtualziation methods such as QEMU full device emulation, virtio, vhost, and device direct assignments.

1 개요

본 문서는 하이퍼바이저와 가상머신 기술에 대한 전반적인 내용을 다룬다. 가상화란 컴퓨터 시스템에서 컴퓨터 리소스의 추상화를 일컫는 광범위한 용어이며, 물리적인 컴퓨터 리소스의 특징을 다른 시스템, 응용 프로그램, 최종 사용자들이 리소스와 상호 작용하는 방식으로 감추는 기술로 정의할 수 있다. 이 글에서는 가상화를 한개의 시스템에서 여러개의 운영체제를 컴퓨팅 환경을 훼손하지 않고 가동하는 방법으로 정의한다. 가상화를 활용하는 이유는 크게 4가지 이유가 존재한다. 먼저, Isolation 목적으로 각 운영체제간의 보안을 유지하며 구동하기 위해 가상화를 활용한다. 둘째는 High Availability 목적으로 유지 보수를 위해 머신이 가동 불가능 한 경우, 워크로드를 이전하기 위해 가상화를 사용한다. 셋째로, Workload Balancing 목적으로 하드웨어 사용률을 최대로 유지하기 위해 사용한다. 마지막으로 SandBoxing 목적으로 Legacy 프로그램을 사용 가능하기 위해 활용한다.

현대의 컴퓨터 시스템은 각각 운영체제 인스턴스를 운영할 수 있는 가상머신이라는 환영을 사용하기에 충분한 리소스를 가지고 있다. 가강머신을 적절하게 설계 및 운영하기 위해서는, 시스템에는 가상머신들의 관리자가 필요하다. 하이퍼바이저 또는 VMM (Virtual Machine Monitor)라고 불리는 소프트웨어는 이러한 목적을 위해 설계되었다. 현재 하이퍼바이저기술은 서버에서 임베디드 시스템에 이르기 까지 방대한 분야에서 사용되고 있다.

하이퍼바이저는 크게 2가지 종류로 구분된다. 먼저, 하드웨어에 대한 직접적인 접근권한을 하이퍼바이저만이 가진 Type 1 하이퍼바이저가 있다. Type1 하이퍼바이저는 하드웨어 계층위에 하이퍼바이저 계층이 존재하고, 하이퍼바이저 계층 위에 가상머신이 존재한다. 대표적인 Type1 하이퍼바이저는 Xen이 있다. 반면, Type 2 하이퍼바이저는 하드웨어에 대한 직접적인 접근권한을 호스트 운영체제와 하이퍼바이저가 가진다. Type2 하이퍼바이저는 하드웨어 계층위에 호스트 운영체제 계층이 존재하고, 그 위에 하이퍼바이저 계층이 존재한다. 하이퍼바이저 계층 위에는 가상머신이 존재한다. 대표적인 Type2 하이퍼바이저는 KVM이 있다.

이 글의 나머지 부분은 다음과 같이 구성된다. 먼저, 2장에서는 가상머신의 일반적인 개념과 활용에 대해서 알아본다. 3장과 4장은 하이퍼바이저 소프트웨어를 지원하기 위해서 추가된 하드웨어 기능에 대해서 알아본다. 3장은 CISC(Complex Instruction Set Computer) 머신인 x86 시스템에 추가된 가상화 하드웨어 기능을 알아본다. 4장은 RISC(Reduced Instruction Set Computer) 머신인 ARM에 추가된 가상화 하드웨어 기능을 알아본다. 5장은 Type1 하이버파이저에 대해서 서술한다. 6장에서는 Type2 하이퍼바이저인 KVM에 대해서 서술한다. 7장에서는 IO 장치에 대한 가상화 기법에 대해서 알아보고, 8장에서 요약과 향후 연구에 대해서 서술한다.

2 가상 머신 일반

이장에서는 가상머신의 일반적인 개념과 2가지 주된 사용 목적에 대해 서술한다.

2.1 Survey of Virtual Machine Research [11]

1개의 컴퓨터 시스템을 다른 시스템에서 구동하는 complete instruction-by-instruction simulation 기법을 이용해서, 시뮬레이션할 머신 X을 일반적인 머신 G에서 구동할 수 있다. 이때, X와 G가 동일한 경우, 소프트웨어의 개입없이 시뮬레이션할 머신 X의 소프트웨어를 하드웨어서 바로 구동할 수 있다. 이러한 시스템을 *virtual machine system* 이라고 부르며 시뮬레이션할 머신 X를 *virtual machine* (VM)이라고 하고 시뮬레이션 소프트웨어를 *virtual machine monitor* (VMM) 혹은 hypervisor이라고 한다.

privileged and non-privileged mode를 가지는 하드웨어 구성에 의한 Extended machine 형식에서는 특정시간에 오직 1개의 privileged 소프트웨어를 지닌다. 이에 따라, 동시에 여러 운영체제를 구동할 수 없다. VMM의 핵심적인 기능은 1개의 머신 인터페이스를 VMM을 이용해 여러개의 머신 인터페이스를 가진것 처럼하면서 동시에 여러개의 운영체제를 구동하는 것이다. 이러한 유용한 가상머신 개념에도 불구하고, 하드웨어의 구조는 이러한 가상머신들을 지원하되록 설계 되지 않았다. 특정한 명령어가 non-privileged mode에서 수행될때 자동적으로 trap되지 않는 문제, VMM이 가상머신이 지니는 페이지를 실제 물리주소로 매핑하기 위해 page table를 변경해야되는 문제를 극복하기 위해, 직접적으로 가상머신들을 지원하는 virtualizable architecture가 제안되었다.

어떤 작업을 가상머신위에서 구동하는 것은 실제 머신 위에서 동작시키는 것에 비해서 단점이 있다. 이러한 단점은 VMM이 사용하는 추가적인 리소스 사용에서 비롯되며 이를 오버헤드라고 부른다. 오버헤드의 원인은 가상 프로세서의 상태를 유지하는 문제, privileged 명령어들을 지원하는 문제, 가상머신의 페이징 기법을 지원하는 문제에서 비롯된다. 이를 위한 저자의 3개의 성능향상 기법은 정책, 가상머신 아키텍쳐와의 절충, 향상된 기법 또는 새로운 매커니즘이다. 위의 기법은 분명 유용하지만, 가상머신만이 가지는 문제는 여전히 존재한다.

2.2 Achieving High Performance via Co-Designed VMs [16]

가상머신의 원래 목적은 여러개의 다른 운영체제를 동시에 구동시키는 가상화를 하는 것이다. 이는 다른 명령어로 컴파일된 프로그램을 다른 명령어를 실행하는 프로세서에서 실행하게 할 수 있는 것을 말한다. 이는 가상머신을 또다른 목적인 virtual instruction set을 native instruction set으로 변환하여 이식성과 호환성을 얻는 것이 될 수 있다. 대표적인 예시로는 Java Virtual Machine (JVM)이 있다. JVM은 Java 컴파일러를 통해 컴파일된 Java Byte Code를 입력으로 받아서 시스템 명령어로 변환하여 실행한다. 이를 통해 Java는 플랫폼 독립성과 이식성을 가진다.

현재의 프로세서의 마이크로아키텍쳐는 더 많은 Instruction level parallelism (ILP)를 추구하거나 명령어 예측등의 기법을 사용해서 성능향상을 해왔다. 이러한 방벙은 4가지 방법의 문제가 크게 있다. 먼저, 명렁어셋의 호환성을 유지해주는 문제가 있다. 둘째로, 평균적인 성능향상에 중점을 두기 때문에 특정 프로그램에는 성능향상이 없을 수도 있는 문제가 있다. 셋째로, 프로세서가 제한된 ILP를 볼 수 있는 문제가 있다. 마지막으로, 특정성능향상기법이 전체 시스템을 복잡하게 만드는 문제가 있다. 이를 해결하기 위해 저자는 co-designed VM을 사용하는 방법을 제안한다.

그림 1a)는 일반적인 하드웨어와 소프트웨어가 분리된 일반적인 프로세서 구조를 나타낸다. 해당 구조에서 하드웨어와 소프트웨어 사이에 인터페이스를 제공하기 위해서 Instruction Set Architecture (ISA)가 존재한다. ISA가 한번 고정되는 순간부터 하드웨어를 파격적으로 개선하기는 힘들어진다.

그림 1b)는 가상머신이 적용된 프로세서 구조를 나타낸다. 가장 유명한 구조로는 Java Byte code와 Java virtual machine (JVM) 환경이 있다. 특정 어플리케이션은 가상 아키텍쳐 구성으로 컴파일 되는데, 자바의 경우 java byte code로 컴파일된다. 해당 컴파일과정은 일반적으로 Virtual ISA(V-ISA)로 컴파일된다고 취급된다. 하드웨어는 기존 ISA를 그대로 제공하는데, 이를 일반적인 프로세서 구조와 구별하기 위해 Native ISA라고 부른다. 가상머신은 어플리케이션과 하드웨어 사이에 추가적인 소프트웨어로써 존재하는데, 여기서 V-ISA를 interpretation, just-in-time (JIT) compilation, adaptive recompilation 또는 이러한 기법을 합친 방식을 통해서 Native ISA로 변환한다.

가상머신을 이용하는 방식은 플랫폼 독립성을 제공하는 것을 목표로 한다. 이는 V-ISA를 이용하는 Java bytecode는 어떠한 하드웨어 플랫폼에서도 사용할 수 있고, 해당 하드웨어에 맞는 가상머신을 제공함으로써 가능해진다. 하지만 가상머신이 interpretation이나 JIT 컴파일을 이용해서 V-ISA를 native ISA를 맞추는 과정이 효율성이 떨어

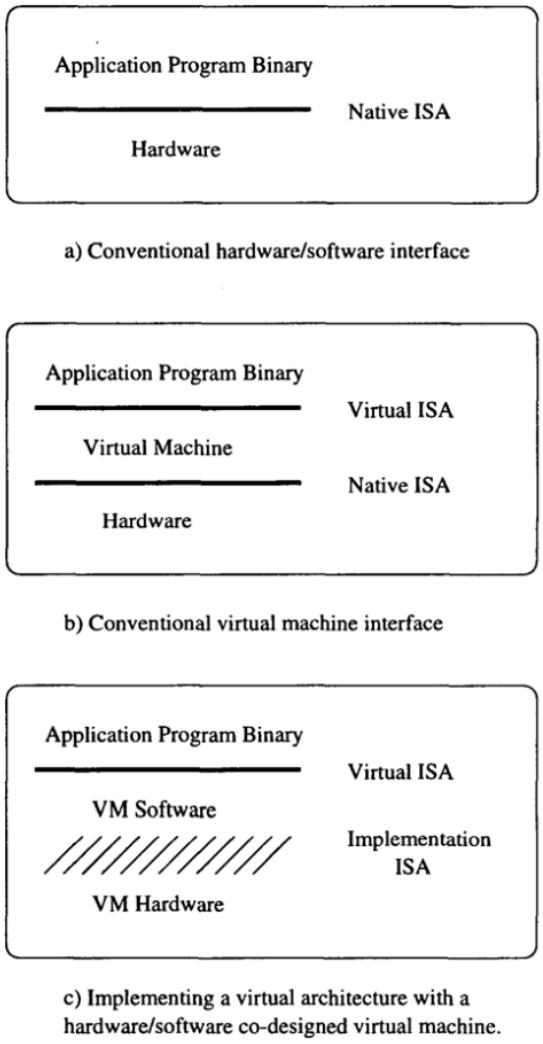


Figure 1: Virtual machine co-design.

지기 때문에 성능 감소가 있다. 이러한 환경에서는 일반적인 성능개선 목표로 native ISA에 근접한 성능을 내는 것으로 삼는다.

저자는 Co-designed VM을 사용해 다른 방식의 접근법을 제시한다. 그림 1c)에서 볼 수 있듯이, V-ISA는 가상머신 접근과 동일하게 고정되고, 가상머신 소프트웨어와 하드웨어는 co-design된다. 새롭게 co-design된 가상머신은 하드웨어에 종속되고 native-ISA와는 다른 implementation ISA(I-ISA)로 설계 된다. I-ISA를 통해서 가상머신은 하드웨어에 최적화되어 속도향상, 추가적인 하드웨어 기능 사용 및 유연성을 얻을 수 있다. 가상머신은 하드웨어에서 성능정보를 얻을 수 있기 때문에 dynamic compilation등을 사용한 최적화가 가능해진다. V-ISA는 고정되기 때문에 플랫폼 독립성은 그대로 유지된다. V-ISA는 Java bytecode 뿐만 아니라 일반적인 RISC나 CISC ISA도 사용될 수 있다.

Co-design 가상머신의 중요 특징중 하드웨어가 제공하는 것은 performance information과 configuration이다. 가상머신이 성능 정보(performance information)를 통해서 추가적인 코드 최적화를 할 수 있고, 어플리케이션의 실행 변화에 따른 최적화 조정도 가능해진다. 하드웨어가 가상머신이 프로세서를 재구성할 수 있는 특수 명령어를 제공하여 구동중인 어플리케이션에 맞게 프로세서를 변화(Configuration)시킬 수 있다. Co-design 가상머신의 중요

특징중 소프트웨어가 제공하는 것은 analysis와 optimization이다. Analysis은 하드웨어가 하는 것보다 많은 양의 코드 분석을 할 수 있음을 의미하며, optimization은 분석을 바탕한 런타임 최적화를 제공할 수 있는 점이다. Co-design 가상머신은 최적화에 의한 오버헤드보다 성능향상을 크게 하기 위해서, 단순한 소프트웨어 기법을 적용했다.

3 x86 CPU에서의 가상머신 지원 [1]

2005년에 Intel과 AMD는 각각 VT-x와 AMD-V라는 가상화지원 하드웨어를 추가했다. 기본적으로 하드웨어 지원 가상화의 경우, trap-and-emulate 기법을 지원하도록 설계됐다. Complex instruction set computer (CISC)와 Reduced instruction set computer (RISC)의 차이에 의해, 가상화 지원하는 방식 또한 다르게 설계 됐다. 이장에서는 대표적인 CISC 머신인 x86의 하드웨어 가상화 지원기술에 대해 서술한다. 2005년까지는 x86 아키텍쳐는 가상화 기능을 하드웨어적으로 지원하지 않았다. 이에 따라, trap-and-emulate라는 가상화 기법은 완벽하게 실현되지 못했다. Popek and Goldberg [14]의 1974년 논문에서는 특정 시스템 소프트웨어가 하이퍼바이저로 고려되기 위한 3가지 요소를 제시한다.

1. *Fidelity*: 시간 요소를 제외하고는 하이퍼바이저 없이 동작하는 것과 동일해야한다.
2. *Performance*: 대다수의 가상머신 명령어는 하이퍼바이저의 관여없이 동작해야한다.
3. *Safety*: 하이퍼바이저가 모든 하드웨어 리소스를 통제해야한다.

위의 3가지 요소에 따라, x86의 아키텍쳐의 경우 가상화 가능하지 않았다. 우선 privileged 상태를 가상머신이 확인할 수 있고, 유저레벨에서 동작하는 privileged 명령어가 trap이 되지 않는 문제가 있기 때문이다. 이러한 x86의 가상화 장애물때문에, 소프트웨어기법의 하이퍼바이저는 binary translation기법 혹은 adaptive binary translation 기법을 사용한다.

3.1 전통적인 가상화방식

전통적인 가상화 방식은 하드웨어 도움없이 trap-and-emulate 방식을 사용하는 가상화 방식이다. 이를 위해 필요한 아이디어는 아래와 같다.

De-privileging: privileged state에 관여하는 모든 명령어가 unprivileged context에서 수행되는 경우는 trap이 발생해야 한다. 하이퍼바이저는 게스트 운영체제를 reduced privileged level로 실행하여 trap이 발생하는 환경을 만든다. 하이퍼바이저는 게스트 운영체제의 trap을 가로채서 가상머신에 맞게 에뮬레이션하는 과정이 필요하다.

Primary and shadow structures: 가상머신이 가진 privileged state와 실제 머신의 privileged state는 다르다. 이러한 차이에도 가상머신이 예상하는 실행환경을 제공해주기 위해서 하이퍼바이저는 게스트 운영체제가 가지는 primary structure를 모방한 shadow structure를 가진다. On CPU privileged data로는 base page table register, processor status register등이 있고, 해당 복사본을 하이퍼바이저는 가지고 있다. 이를 활용해 가상머신 trap을 에뮬레이션한다. Off CPU privileged data로는 page table등이 있는데, 해당 정보에 접근시에는 trap이 발생하지 않을 수 있다. 이는 shadow structure의 incoherence를 만든다.

Memory traces: Off CPU privileged data의 coherence(일관성)을 보장하기 위해서 tracing 방식을 사용한다. Tracing 방식은 하드웨어의 페이지 보호방식을 이용해 메모리 접근시 trap을 유도하는 방법이다. 예를 들어, 게스트 운영체제의 page table entry(PTE)에 쓰기 보호를 걸어두어서, 이를 접근시 trap이 발생하도록 유도하는 방식이다.

Classic virtualization은 성능문제가 있어서, 이를 수정하는 2가지 방식이 존재한다.

1. *Paravirtualization*: 운영체제와 하이퍼바이저사이에 유동성을 추가하는 방식으로, OS를 하이퍼바이저에 맞게 수정하는 방식이다. 성능향상이 좋다. Xen이 대표적인 예시이다.

2. *HW supported trap-and-emulate*: 하드웨어와 하이퍼바이저사이에 유동성을 추가하는 방식이다. IBM 370 architecture interpretive execution이 시초이며, 게스트 운영체제를 위한 하드웨어 실행모드를 제공한다. 이는 후에 하드웨어 가상화기법으로 발전됐다.

3.2 소프트웨어 기반 가상화

Software virtualization은 binary translation 방식을 사용하는 가상화 방식이다. x86은 두가지 이유로 상기언급한 전통적인 가상화 방식을 사용할 수 없다. 먼저, x86에서 code segment selector (%cs)를 읽게 되면 current privilege level(CPL)을 알게 되어 privileged state가 노출되는 문제가 있다. 둘째로, dual-purpose instruction은 trap이 되지 않는 문제이다. 동일한 명령어라도 실행모드에 따라 다르게 동작하기 때문인데 예를들어, popf의 경우 user mode에서는 ALU flag만을 바꾸게 되나, kernel mode에서는 ALU flag와 system flag를 둘다 바꾸게 되고, 이러한 명령은 trap이 되지 않는다.

위와 같은 문제를 해결하기 위해서 guest의 명령어를 CPU위에서 바로 실행하는 것이 아닌 인터프리터를 두어서 해결 할 수 있다. Java byte code가 java virtual machine(JVM) 위에서 실행되는 것과 유사하다. 이에 해당되는 기술은 binary translation인데, 인터프리터의 경우 하이퍼바이저가 되기 위한 조건중 fidelity, safety를 만족하게 할 수 있으나, 성능이 매우 안좋다. 자바에서 Just-in-time(JIT) 컴파일러를 이용하는 것처럼 런타임에 번역을 하고 한번 번역된 것을 저장해 두어서 다시 사용하는 방법인 binary translation 이용한다.

Binary translation을 적용한 Software hypervisor의 크게 6가지 특성이 있다. 먼저, 입력이 binary x86 code라는 특성이다. 둘째로, 런타임에 번역을 하는 Dynamic 기법이라는 점이다. 셋째로, 실행하기 직전에만 번역하는 On demand 특성을 띈다. 넷째로, 원본 코드에 오류가 있더라도 x86 기준에 따라서만 번역한다. 다섯째로, 입력은 모든 x86 명령어이나, 결과는 안전한 x86 명령어만 출력하는 특성이 있다. 마지막으로, 번역은 상황에 따라 지속적으로 변경가능하며 이를 Adaptive 특성을 지닌다고 한다.

Binary Translation에서 대부분의 명령어는 동일하게 치환된다. Translation Unit(TU) 단위로 번역되고 실행되고, TU는 control flow 명령이나 12개의 명령어를 만나면 생성된다. 1개의 TU는 1개의 Compiled Code Fragment(CCF)를 생성하고, 실제로 실행되는 명령어만 번역한다. 또한 번역시 코드 최적화는 진행하지 않는다. 하이퍼바이저와 게스트 운영체제는 세그먼트로 분리하여 보호를 제공한다. 번역된 코드에서 하이퍼바이저의 자료에 접근하기 위해서는 %gs segment register를 따로 이용한다. 동일하게 치환되지 않는 명령어는 PC-relative addressing를 사용하는 경우, Direct control flow인 경우, Indirect control flow인 경우, Privileged instruction인 경우이다.

Adaptive binary translation은 번역시 trap에 최대한 빠지지 않도록 번역하는 것이 목표이다. 기본적으로는 innocent until proven guilty를 기준으로 하고 과도한 trap에 빠지게 되면 다른 번역 정책을 쓰도록 바꾼다. 동일하게 치환되지 않는 부분을 trap에 다시 빠지지 않도록 재번역한다. 동일하게 치환된 부분을 새로운 번역으로 forwarding하도록 변경한다. 과도한 trap이외에도 다른 환경에 따라 adaptive binary translation을 사용한다. 하지만 이에 따른 일정한 overhead도 있다.

3.3 하드웨어 기반 가상화

Hardware virtualization이란 하드웨어적으로 지원을 받아 trap-and-emulate를 활용한 가상화 방식이다. 대표적인 하드웨어 지원으로는 Intel VT-x, AMD-V, ARM virtualization extension등이 있다.

x86의 경우, virtual machine control block(VMCB)를 메모리에 두어서 게스트 운영체제의 virtual CPU상태를 관리할 수 있다. 추가적으로 게스트 운영체제를 구동하기 위한 guest mode를 따로 둔다. 하이퍼바이저를 위한 모드는 기존 모드 그대로 host mode라고 부른다. host mode에서 guest mode로 진입하기 위한 명령어로 vmrun이 있다. vmrun시 하드웨어가 자동으로 VMCB에서 데이터를 읽어서 게스트 운영체제의 상태를 복구하고 실행이 가능하게 한다. VMCB에서 설정한 특정한 조건이 만족되면, 하드웨어가 exit명령을 통해서 VMCB에 게스트의 상태를 자동으로 저장하고 하이퍼바이저에게 통제권을 넘기게 된다. VMCB의 control field를 통해서 exit이 되는 조건을 하이퍼바이저가 설정할 수 있다. 하이퍼바이저에게 통제권이 넘어오게 되면 exit이 된 이유에 맞게 하이퍼바이저가 처리를 하거나 에뮬레이션하고, vmrun을 통해 다시 게스트 운영체제를 구동하게된다. 이를 통해 기존 trap-and-emulate의 과도한 trap을 방지할 수 있다.

exit이 얼마나 자주 되는가와 exit시 host mode와 guest mode의 전환 속도가 하이퍼바이저의 성능을 크게 좌우한다. 전환속도는 하드웨어의 성능이기 때문에 하이퍼바이저의 성능을 좌우하는 가장 큰 요소는 exit의 주기에 있다.

3.4 Software hypervisor versus Hardware hypervisor

Binary translation를 사용하는 소프트웨어 하이퍼바이저의 장점은 3가지가 있다. 먼저, adaptive binary translation으로 대부분의 trap을 빠르게 대체 가능하다. 둘째로, 한번 binary translation을 한 부분(callout)을 계속 사용할 수 있으므로, 에뮬레이션 루틴을 제공할 수 있어 빠르다. 마지막으로, 자주 사용되는 경우에 대해서, callout을 translation cache(TC)에 저장할 수 있으므로 callout cost도 무시 가능하다.

Hardware trap-and-emulate를 사용하는 하드웨어 하이퍼바이저의 장점은 3가지가 있다. 첫번째로, 번역이 없으므로 코드의 사이즈가 변하지 않는다. 둘째로, 하드웨어가 자체적으로 exception시 상태 저장 및 복원을 해주므로 exception handling을 간단히 할 수 있다. 셋째로, 시스템 콜을 하이퍼바이저의 간섭없이 수행가능하다.

3.5 향후 개선방향

마이크로아키텍쳐의 개선으로 vmrun, vmexit같은 작업의 속도가 매우 빨라지고 있고, 이는 hypervisor의 성능개선에도 좋다. 하지만 하드웨어 하이퍼바이저는 태생적으로 stateless 성질을 가지고 있기 때문에 native에 비해서 성능의 제한을 두게 된다. 이는 하이퍼바이저가 vmexit이후 에뮬레이션을 위한 vcpu의 상태를 VMCB를 무조건적으로 통해 재구성을 해야하는 문제에서 발생하고 이를 완전히 제거하는 것은 불가능하다. 또 다른 하드웨어 하이퍼바이저의 문제로는 Memory Management Unit (MMU) 에뮬레이션으로 인한 성능저하가 있다. 이를 위해 메모리 trace를 적게 쓰게하는 최적화 기법, 하드웨어 하이퍼바이저와 소프트웨어 하이퍼바이저의 하이브리드 적용 같은 기법을 사용할 수 있다. 하지만 최종적으로는 하드웨어에서 MMU 가상화를 지원해야 한다. 이를 Intel은 Extended Page Tables (EPT), AMD는 nested paging이라는 기술을 지원하려고 한다. 해당 기술을 통해 메모리 tracing을 더이상 사용하지 않아도 되고, 게스트의 context switching을 하이퍼바이저의 간섭없이 가능하게 한다.

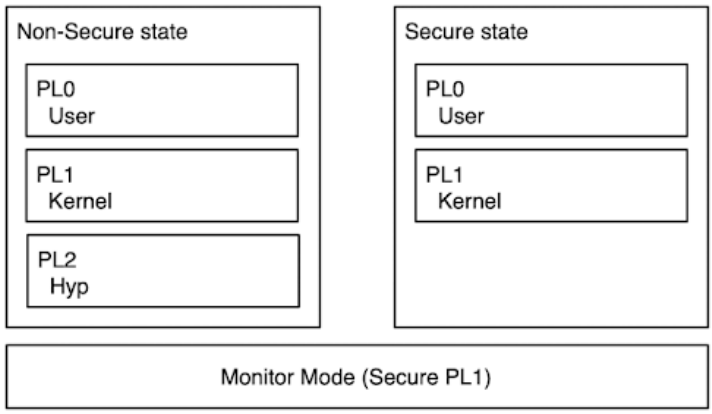


Figure 2: ARM CPU modes

4 ARM CPU의 가상머신 지원

ARM의 64비트 프로세서인 ARMv8에서 가상화지원 하드웨어가 추가됐다. 이장에서는 대표적인 RISC 머신인 ARM의 하드웨어 가상화 지원기술에 대해 서술한다.

4.1 Hardware-Supported Virtualization on ARM [17] [6]

ARM에서는 Xen과 같은 Type1 하이퍼바이저, KVM과 같은 Type2 하이퍼바이저에서 성능에 가장 큰 영향을 미치는 가상머신과 하이퍼바이저사이의 전환 주기와 비용을 줄이기 위해서 하드웨어 가상화 기능을 추가했다. ARM이 지원하는 가상화 하드웨어 부분은 4가지로 CPU, 메모리, 인터럽트, 타이머가 있다.

CPU Virtualization

그림 2처럼 Kernel mode보다 privileged level이 높은 Hyp mode 추가했다. Hyp mode는 General Purpose (GP) 레지스터를 제외한 별도의 레지스터 셋을 가진다. Hyp mode는 trap-and-emulate 방식을 지원하는데, kernel mode에서 특정한 조건이 만족되면 Hyp mode로 trap이 발생한다. 특정한 조건으로는 특정한 privileged 명령어을 수행하는 경우나 하드웨어 인터럽트가 발생하는 경우등이 있다. 구체적인 설정은 HCR_EL2 (Hyp Configuration register)의 bit 설정으로 가능하다. 해당 레지스터 설정을 통해 일반적인 시스템 콜등은 가상머신의 kernel mode로 trap 되도록 하여, 하이퍼바이저의 간섭을 줄일 수 있다.

일반적인 운영체제에서 프로세스가 전환되는 경우를 Context Switching이라 부르는 것처럼, 하이퍼바이저에서는 가상머신이 전환되는 경우를 World Switching이라 부른다. World Switching은 모든 ARM mode의 banked 레지스터, MMU 레지스터, 타이머 등 Context Switching에 비해서 저장 및 교체할 양이 많기 때문에 매우 무겁다. ARM은 world switching의 비용을 줄이기 위해서 여러 기술을 추가했다. 대표적으로 world switching시 Translation Lookaside Buffer(TLB)가 flush되는 현상을 막기 위해서 Address Space Identification (ASID)와 유사하게 Virtual Machine Identification (VMID)를 TLB에 추가하고 현재 VMID를 나타내는 레지스터를 추가했다. 그럼에도 World Switching 비용은 매우 크므로, World Switching을 최대한 줄이도록 하이퍼바이저를 설계하는 것이 중요하다.

Memory Virtualization

하이퍼바이저가 존재할 때, 가상머신이 보고 있는 물리 주소는 실제 메모리 장치의 주소 (PA)와는 다르다. 가상머신이 보고 있는 물리주소를 Intermediate physical address (IPA) 혹은 Guest physical Address (GPA)라고 부른다.

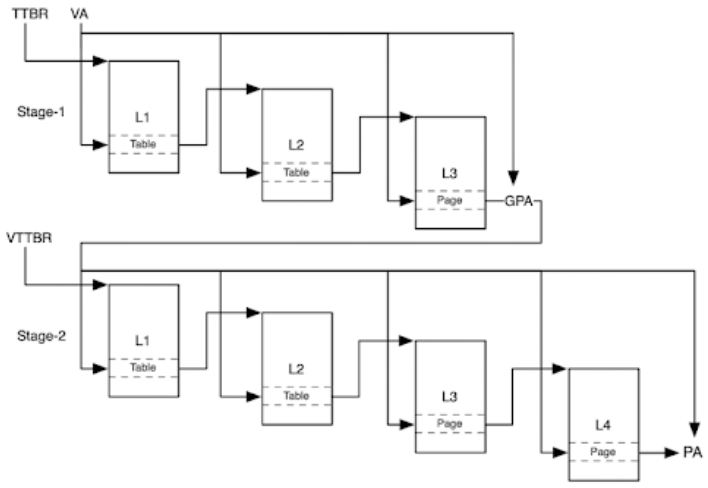


Figure 3: Stage-1 와 Stage-2 페이지 번역과정

이에 따라 일반적인 가상주소 (VA)는 2단계의 번역과정이 필요하다. VA를 IPA로 변환하는 과정을 Stage-1 page translation, IPA를 PA로 변환하는 과정을 Stage-2 page translation이라 부른다. ARM의 가상화 하드웨어는 2단계 번역과정을 지원한다.

그림 3은 Stage-1 과 Stage-2 페이지 번역시, page table walk 과정을 나타낸다. Stage-2 페이지 번역은 HCR_EL2 레지스터에서 설정가능하다. Stage-1 페이지 번역의 base register는 TTBR(Translation Table Base Register)이고 Stage-2 페이지 번역의 base register은 VTTBR(Virtualization Translation Table Base Register)이다. 중요한 점은 Stage-1과 Stage-2 사이의 page table level이 다른것이다. Stage-1은 3단계, Stage-2는 4단계 구성된다.

Interrupt Virtualization

그림 4와 같이 ARM은 GIC(Generic Interrupt Controller)에서 VGIC(Virtualization Generic Interrupt Controller)를 추가했다. GIC는 Distributor를 1개, CPU interface를 코어당 1개 가진다. VGIC는 virtual CPU interface를 코어당 추가로 가진다. Distributor는 PPI(Private Peripheral Interrupt), SPI(Shared Peripheral Interrupt), SGI(Software Generated Interrupt)를 전달받아 특정 코어에 IRQ (Interrupt Request), FIQ (Fast Interrupt Request)를 발생시킨다. Distributor는 Memory-mapped IO (MMIO)를 통해 접근가능하며, 이를 통해 인터럽트의 우선순위를 마스킹하거나 인터럽트를 켜거나 끌 수 있다. CPU interface는 distributor에서 인터럽트를 전달 받아, CPU에 인터럽트를 발생시키며, CPU는 MMIO를 통해 CPU interface에 접근하여 ACK(Acknowledge), EOI(End-Of-Interrupt)를 프로그래밍 가능하다.

하이퍼바이저가 하드웨어의 통제권을 쥐기 위해서는 CPU interface에 가상머신이 직접하는 것을 막아야하는데, 이에 따라 모든 인터럽트는 hyp mode인 하이퍼바이저에서 에뮬레이션 될 필요가 있다. 그런데 ACK와 EOI같은 작업을 에뮬레이션하는 것은 자주 반복되므로 오버헤드가 크다. VGIC는 virtual CPU interface를 추가해서, 하드웨어 인터럽트는 여전히 하이퍼바이저가 통제권을 가지지만, 가상 인터럽트는 virtual CPU interface에서 하이퍼바이저를 거치지 않고 ACK, EOI를 가능하게 한다. virtual CPU interface를 통제하기 위해서 하이퍼바이저는 List register를 프로그래밍한다. 이를 통해 가상 인터럽트를 List register를 통해 만들 수 있다. Distributor는 가상화지원이 되지 않기때문에, 무조건적인 hyp mode로의 trap이 발생한다. IPI(Inter-Processor Interrupt)를 발생시키기 위해서는 SGI

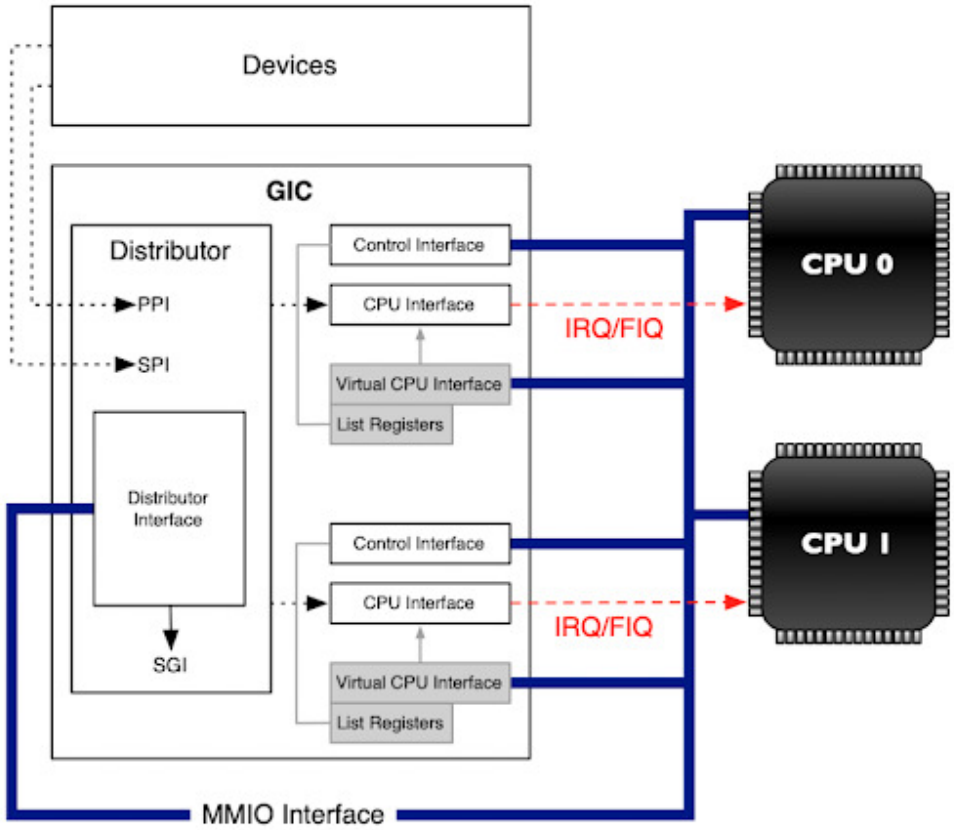


Figure 4: ARM Generic Interrupt Controller (GIC) overview

를 코어에서 MMIO로 distributor에 쓰는 과정이 필요한데, 이는 hyp mode로의 trap을 발생시키고 하이퍼바이저 에뮬레이션을 필요로 한다.

Timer Virtualization

타이머 또한 인터럽트와 마찬가지로 하드웨어 통제권을 하이퍼바이저가 쥐기 때문에, 타이머에 접근시 hyp mode로의 trap을 유발한다. 타이머 접근이 자주 발생하기 때문에, virtual timer, virtual counter를 추가해서 가상머신이 직접적으로 관리가 가능하게 만든다.

x86 versus ARM virtualization extensions

아래의 표 1는 x86과 ARM의 가상화지원 하드웨어의 비교이다.

	x86	ARM
CPU	root, non-root mode	hyp mode
Memory	nested page	stage-2 translation
Interrupt	vAPIC	vGIC
Timer	x	virtual Timer

Table 1: x86과 ARM의 가상화 하드웨어 비교

CPU는 x86과 ARM 모두 privileged 명령어의 trap을 지원한다. 실제 구현 방식에는 차이가 있는데, x86의 경우 가상머신을 위한 privileged level을 만들었다. 하이퍼바이저는 root mode로 동작하고, 가상머신은 non-root mode로 동작하는데, 둘은 동일한 privileged ring 구성을 가진다. root mode에서는 `vmrun`을 통해서 non-root mode로 진입할 수 있고, 이를 통해 가상머신은 기존과 동일한 구조의 privileged ring에서 동작할 수 있다. 특정 조건이 만족시 non-root mode는 `exit`명령으로 하이퍼바이저가 존재하는 root mode로 trap될 수 있다. 중요한 점은 `vmrun`, `exit` 명령시 하드웨어가 자동으로 Virtual Machine Control Structure (VMCS)라는 메모리상의 자료구조에 가상머신정보를 저장한다는 것이다. 반면 ARM은 Exception Level 2 (EL2)라는 추가적인 privileged level과 hyp mode를 도입했다. 유저레벨 프로세스는 EL0에서, 운영체제는 EL1에서, 하이퍼바이저는 EL2에서 동작하게 설계했다. 기존에 `svc`를 통해 EL0에서 EL1으로 전환할 수 있듯이, `hvc`을 통해 EL1에서 EL2로 전환할 수 있다. x86과 다르게 ARM에서는 전환시 하드웨어에서 자동적으로 가상머신 상태를 저장하지 않으므로 소프트웨어적으로 해당 과정을 구현해 주어야 한다.

메모리의 경우 x86과 ARM 모두 2단계 번역을 지원한다. 다만, x86에는 세그먼트에 대한 번역이 추가적으로 더 지원이 된다. x86은 경우 해당 번역과정을 nested page라고 부르고, ARM의 경우 해당 번역을 Stage-2 translation 이라 부른다.

인터럽트는 두 경우 모두 인터럽트 컨트롤러에 가상화 기능을 넣었다. 인텔의 인터럽트 컨트롤러인 Advanced Programmable Interrupt Controller (APIC)은 가상화 기능을 추가해서 virtual APIC (vAPIC)을 만들었다. ARM의 인터럽트 컨트롤러인 Generic Interrupt Controller (GIC)는 가상화 기능을 추가해서 virtual GIC (vGIC)를 만들었다. 반면, 타이머는 x86은 지원 되지 않으며, ARM만이 가상 타이머를 지원한다.

4.2 Generic Interrupt Controller Version 3 Virtualization [10]

이장에서는 ARM의 GIC와 시스템 타이머인 Generic Timer에 추가된 가상화 기능에 대해 자세히 서술한다.

GICv3 Architecture

ARM 아키텍쳐에서는 아래와 같이 4개의 인터럽트 종류가 있다.

1. Software Generated Interrupts (SGI): 운영체제가 주로 발생시키며, 프로세서간의 인터럽트이다. Inter-Processor Interrupt (IPI)라고 주로 불린다.
2. Private Peripheral Interrupts (PPI): 특정 CPU 코어와 통신하는 디바이스가 발생시키는 인터럽트이다. 타이머 인터럽트도 여기서 속한다.
3. Shared Peripheral Interrupts (SPI): I/O 장치가 발생시키는 인터럽트이다. 어떤 CPU 코어로도 갈 수 있다.
4. Locality-specific Peripheral Interrupts (LPI): message based interrupts이며, PCI Express 디바이스가 사용할 수 있는 인터럽트이다. Message-Signaled Interrupts라고 불린다.

인터럽트의 종류 이외에도 인터럽트 그룹, 인터럽트 우선순위등이 각 인터럽트가 가지는 특성이다. 인터럽트 그룹은 FIQ로 처리하는 group0, IRQ로 처리하는 non-secure group1, secure world용인 secure group1로 나뉜다.

GICv3는 Distributor, Redistributor, CPU interface로 총 3개의 부분으로 구성된다. Distributor는 GIC에 1개 존재하며, 글로벌 인터럽트인 SPI를 설정한다. enable/disable register가 존재하고, 디바이스당 1bit가 필요하다. priority register가 존재하고, 디바이스당 8bit가 필요하다. target register가 존재하고, 코어당 8bit가 필요하다. 추가적으로

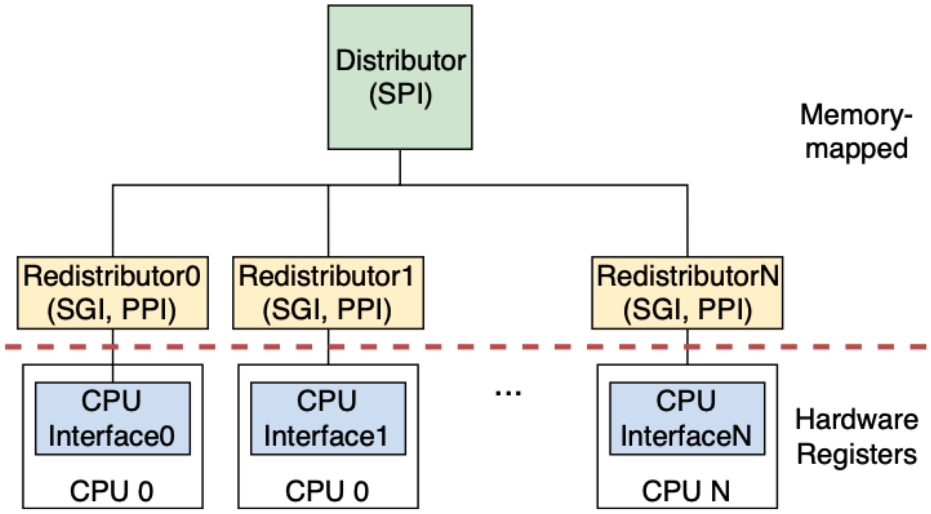


Figure 5: GICv3 구조

SGI register가 존재하고, SGI요청일 때 요청한 CPU정보를 저장한다. Distributor는 부팅 때 설정되고, 이후에 변경하는 경우는 거의 없다. 프로그램시 Memory mapped I/O (MMIO)로 설정한다.

Redistributor는 코어당 1개 존재하며, 코어당 적용되는 인터럽트인 PPI, SGI를 설정한다. enable/disable register가 존재하고, 디바이스당 1bit가 필요하다. priority register가 존재하고, 디바이스당 8bit가 필요하다. Distributor 처럼 부팅때 설정되고, 이후에 변경하는 경우는 거의 없다. 프로그램시 MMIO로 설정한다.

CPU interface는 코어당 1개 존재하며, 인터럽트 정보를 CPU에 전달하는 역할을 한다. Priority mask register가 존재하고, CPU interface가 전달할 우선순위 기준 설정한다. Interrupt Acknowledge Register (IAR)가 존재하고, 현재 인터럽트 상태를 설정한다. IAR은 소프트웨어는 읽기만 가능하고 하드웨어만 설정이 가능하다. End of Interrupt Register (EOIR)가 존재하고, 인터럽트 종료 여부를 설정한다. EOIR은 소프트웨어는 쓰기만 가능하고 하드웨어는 읽기만 가능하다. Running Priority register가 존재하고, 현재 인터럽트의 우선순위을 정한다. Highest Priority Pending Interrupt register가 존재하고, 다음 수행할 인터럽트 ID를 가진다. CPU interface는 매 인터럽트 처리마다, 레지스터 직접 접근방식을 취한다. 이는 GICv3에서 변경된 사항이다.

GICv3 Virtualization

모든 인터럽트가 EL2로 trap되는 것은 비효율적인데, 하이퍼바이저로의 전환을 해야하기 때문이다. 이는 인터럽트 ACK 과정이나 인터럽트 EOI 과정에는 더욱 비효율적이다. 인터럽트가 EL1으로 trap되는 것은 에뮬레이션 과정이 없어져 성능을 향상 시킬수 있으나, 가상머신이 하드웨어 통제권을 쥐는 것은 하이퍼바이저의 원칙에 위배된다. 이를 해결하기 위해, 자주 접근되지 않는 Distributor와 Redistributor는 MMIO를 이용해서 EL2로 trap되게 유도한다. 이를 통해 virtual distributor, virtual redistributor을 가상머신이 인식하게한다. memory mapped영역이 stage-2 translation에서 맵핑이 안되도록 설정하여, stage 2 page fault를 유도한다. 하이퍼바이저가 통제권을 쥐고 에뮬레이션하도록 유도한다. 소프트웨어적으로 trap and emulate 방식을 만드는 것이다.

자주 접근 되는 CPU interface는 하드웨어적으로 지원을 받는다. EL2 register set을 만든 것처럼 그림 4에서 볼 수 있듯이 virtual CPU interface, List Register를 만들어두었다. CPU interface는 EL2에서만 접근 및 수정 가능하지만, virtual CPU interface는 EL1에서도 접근 및 수정 가능하다. 인터럽트를 CPU interface를 통해 CPU에 전달하면,

하이퍼바이저가 EL2에서 실행되어 가상 인터럽트로 변경 후, virtual CPU interface에 전달한다. 이 과정을 virtual interrupt injection이라 부른다. virtual CPU interface는 하이퍼바이저가 직접 쓰는 것은 아니고 List register를 통해 프로그램 될 수 있다. virtual CPU interface에 전달된 가상 인터럽트는 EL2에서 실행되는 동안에는 무시된다. 이후, 가상머신이 실행되면 가상 인터럽트를 인식하고, virtual CPU interface에서 정보를 가져와서 처리한다. 가상 인터럽트가 완료되면 virtual CPU interface의 EOIR에 정보를 써서 하이퍼바이저의 간섭없이 인터럽트 처리가 가능하다. 이를 통해 모든 외부 인터럽트는 하이퍼바이저로 trap이 되어 하드웨어 통제권을 가질 수 있고, ACK와 EOI 같은 내부 접근은 하이퍼바이저로의 trap없이 실행가능하다.

Timer Virtualization

Generic Timer 내부의 타이머 종류는 physical timer, physical counter, virtual timer, virtual counter로 구성된다. virtual timer와 counter는 가상머신용 하드웨어이다. 인터럽트처럼 가상머신의 게스트 운영체제도 timer에 접근할 경우가 많은데, 이를 가상머신이 직접적으로 접근하는 것은 하이퍼바이저 원칙에 어긋나고, 매번 하이퍼바이저로 trap되면 오버헤드가 너무 크다. ARM에서는 이를 해결하기 위해 인터럽트처럼 하드웨어적으로 가상머신이 사용할 수 있는 timer, counter를 구현했다. virtual timer, counter를 EL1에서 접근하는 것은 EL2로의 trap이 발생하지 않는다.

그러나 EL1에서 physical timer, counter에 접근하는 것은 EL2로의 trap이 발생하여 여전히 에뮬레이션이 필요하다. 또한 타이머를 접근하는 것이 아닌 타이머를 설정한 가상머신이 아닌 다른 가상머신이 구동중일 때 발생하는 virtual timer 인터럽트는 여전히 하이퍼바이저의 virtual interrupt injection 과정이 필요하다.

5 Type 1 하이퍼바이저

이장에서는 대표적인 Type1 하이퍼바이저의 디자인과 구현을 살펴본다.

5.1 Disco [5]

기존 운영체제가 scalable한 머신을 지원하기 위해서는 상당한 양의 운영체제 수정이 필요하다. 현대의 운영체제가 커지고 복잡해짐에 따라, 이러한 수정작업은 상당한 시간과 비용이 든다. 기존운영체제가 scalable한 머신에 맞게 수정하는 것이 아닌, 소프트웨어 계층과 하드웨어 계층사이에 추가적인 계층을 추가하는 것이 효율적이다. 추가되는 소프트웨어 계층은 virtual machine monitor (VMM) 혹은 hypervisor라고 부르며, 머신의 리소스를 가상화한다. 이러한 가상머신들은 scalable한 머신에서 아직 scalable 하드웨어를 사용할 준비가 되지 않은 상용 운영체제가 여러개 구동 될 수 있게 해준다.

VMM을 사용하는 것은 장점도 있지만 기존 단점도 따르게 된다. 가상머신의 단점은 하드웨어 리소스를 가상화함으로써 따르는 오버헤드, 리소스 관리 문제, 공유 및 통신 문제가 있다. VMM에 약간의 기법을 사용하면, 이러한 단점을 크게 줄일 수 있다.

저자가 제시하는 VMM인 Disco는 기존 VMM이 지니는 문제를 해결하는 여러개의 특징이 있다. 전통적인 VMM처럼, Disco는 가상머신에서 수행되서는 안되는 명령을 감지 및 에뮬레이션하여 CPU 가상화를 한다. 메모리는 추가적인 주소 변환 층을 만들어 가상화한다. I/O 장치의 경우는 I/O 장치로 향하는 모든 통신을 가로채서 에뮬레이션을 하여 가상화한다. Disco는 또한 오버헤드를 줄이고 가상머신간의 메모리와 디스크 공유를 위해 NUMA 구조 메모리 관리 기법, copy-on-write 디스크 기법, 가상 네트워크 인터페이스 기법을 사용한다.

Disco는 가상화 기법을 통해 scalability, reliability, NUMA 메모리 구조 관리를 제공한다. NUMA관리기법는 가상머신이 암묵적으로 프로그램 코드와 같은 자료구조를 공유하는 것과 파일 시스템 버퍼를 활용하여 이루어진다.

5.2 Xen [3]

2003년 기준 x86이 가상화 기능을 제공하지 않아서 full virtualization을 구현하기 힘들었다. VMware ESX server 같은 경우는 binary translation 기법으로 구현했다. Xen은 다른 접근 방식을 취하는데, 게스트 운영체제가 하이퍼바이저 통제를 받아, 실제 하드웨어의 통제권을 지니게 하는 방식이다. 이를 *paravirtualization*이라 부른다. *paravirtualization*은 성능 개선이 좋다는 장점이 있고, 단점은 게스트 운영체제의 커널 코드에 수정이 필요한 점이 있다. 하지만 Application Binary Interface (ABI)는 수정되지 않기 때문에, 게스트 운영체제의 유저 어플리케이션은 수정없이 사용가능하다.

Xen Design

메모리 가상화: RISC머신의 경우, Software-managed TLB와 ASID (Address Space Identifier)를 가지고 있기 때문에, 메모리 가상화 과정이 쉽게 구현될 수 있다. x86의 경우, TLB가 하드웨어에 의해서 자동적으로 관리되고, ASID가 존재하지 않아서 context switching시 모든 TLB가 flush된다는 특징이 있다. 이에 따라 가상화 과정 구현이 힘들다.

x86의 한계 때문에 Xen의 메모리 가상화는 2가지 특징을 가진다. 먼저 게스트 운영체제는 페이지 테이블을 할당하고 관리할때, Satefy와 isolation을 보장하기 위해서 Xen의 간섭을 받는다. 둘째로 address space의 첫 64MB는 Xen이 존재하도록 만드는데, 이를 통해 Xen에 진입하고 나오는 과정에서의 TLB flush를 피할 수 있다.

새로운 프로세스가 만들어져서 새로운 페이지 테이블을 만드는 경우를 예로 보자. 게스트 운영체제가 자신이 가진 메모리 공간에서 메모리를 할당 및 초기화하고 이를 Xen에 등록한다. 이 순간부터 게스트 운영체제의 페이지 테이블 읽기는 Xen의 간섭없이 할 수 있으나 쓰기나 갱신은 Xen의 확인을 받아야한다. Xen의 간섭으로 게스트 운영체제는 자신의 메모리영역만을 사용할 수 있으며, 직접적인 페이지 테이블 쓰기를 할 수 없다. 게스트 운영체제는 매 페이지 테이블 업데이트마다 Xen의 간섭을 최소화하기 위해서, batch형식으로 한번에 업데이트를 요청하게 할 수도 있다.

Segment descriptor table 역시 동일한 기법이 적용된다. 세그먼트 테이블 읽기는 직접적으로 가능하고, 쓰기나 갱신은 Xen의 간섭을 받는다. address space마다 할당한 Xen영역에는 접근이 불가능하게 설정한다.

CPU 가상화: privileged level이 2개만 제공되는 경우, 게스트 운영체제와 유저 어플리케이션이 lower privileged level을 공유해야된다. 이 때, 하이퍼바이저는 higher privileged level을 지닌다. 또한 어플리케이션에서 운영체제로의 제어권 변경은 하이퍼바이저를 거쳐서 간접적으로 발생한다.

x86의 경우 그림 6처럼 4개의 privileged level이 제공되기 때문에 효율적인 가상화가 가능하다. 하이퍼바이저인 Xen이 ring0로 옮겨지고, 게스트 운영체제가 ring1으로 옮겨지면 privileged 명렁어의 직접 실행이 Xen만 가능하도록 수정가능하다. 이 경우 게스트 운영체제의 수정이 필요하다. 게스트 운영체제가 privileged 명령어를 실행할 때, 프로세서에 의해서 fault가 발생하고, 해당 fault를 처리하며 Xen이 검증 및 실행을 한다. 각 운영체제의 Exception handler table(Vector table)들은 Xen에 등록 되어야 한다. Exception handler가 privileged 명령어을 수행하지 않는 경우는 Xen의 간섭 없이 그대로 수행될 수 있다. 구체적으로는, handler code segment가 ring0에서 실행되는지를 확인하면 된다. 시스템 콜의 경우 fast exception handler로 등록하여, Xen의 간섭없이 수행가능하게 한다. Page fault handler는 CR2 레지스터 정보를 가져오는 privileged 명령어를 수행하기 때문에, ring1에서 게스트 운영체제가 수행

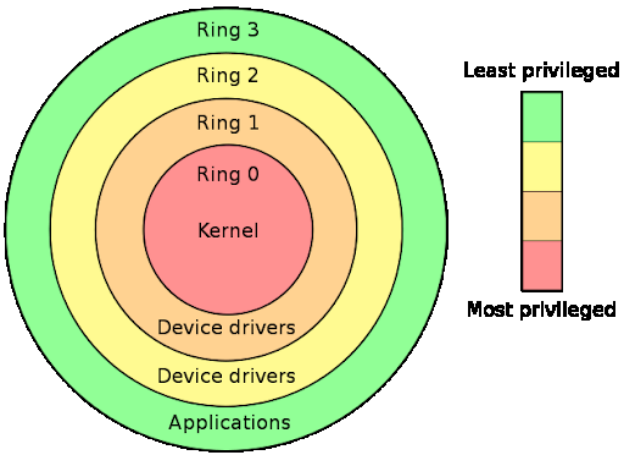


Figure 6: x86 Privileged Ring

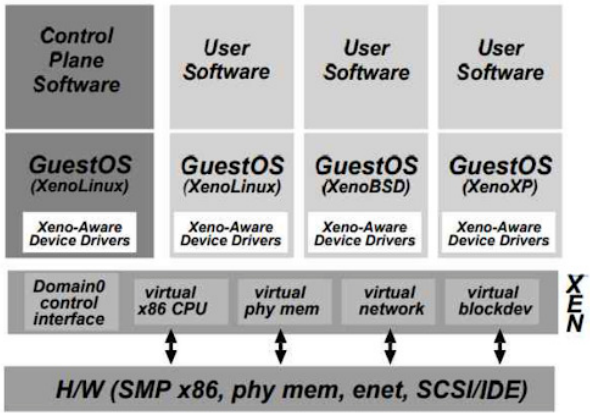


Figure 7: Xen의 일반적인 구조

하기 위해서는 Xen stack frame에서 정보를 복사해서 게스트 운영체제의 스택에 저장하는 과정이 필요하다. 이를 통해 게스트 운영체제가 CR2 레지스터에 직접 접근하는 것을 막는다. Exception handler가 fault를 처리중 다시 fault가 나는 상황인 double fault의 경우, Xen이 처리 후 Exception handler가 계속 처리한다. Double fault보다 더 많은 fault가 발생시 게스트 운영체제를 강제 종료한다.

I/O Device 가상화: 각 디바이스를 직접적으로 에뮬레이션하는 방식이 아닌, 기존 디바이스 드라이버를 활용할 수 있게 구현한다. Xen에 디바이스 드라이버의 backend를, 각 도메인에 있는 게스트 운영체제에는 디바이스 드라이버의 frontend를 구현한다. I/O 데이터는 각 도메인에서 Xen으로 Ring mechanism이라는 shared-memory와 asynchronous buffer를 이용해 전달된다. I/O 인터럽트는 lightweight event delivery mechanism으로 비동기적 알림을 도메인으로 보낼 수 있게 구현한다.

VM control & management: 그림 7은 Xen의 일반적인 구조를 나타낸다. Xen은 메카니즘과 정책을 최대한 분리하는데 초점을 둔다. 하이퍼바이저인 Xen 자체는 기본적인 제어 메카니즘만을 제공한다. 복잡한 제어 정책은 Domain0의 운영체제에서 제공할 수 있게 만든다. 가상머신의 생성, 가상머신의 삭제, 디바이스, 물리 메모리 할당 등은 Domain0 control interface를 사용할 수 있는 Domain0의 운영체제에게 책임이 있다.

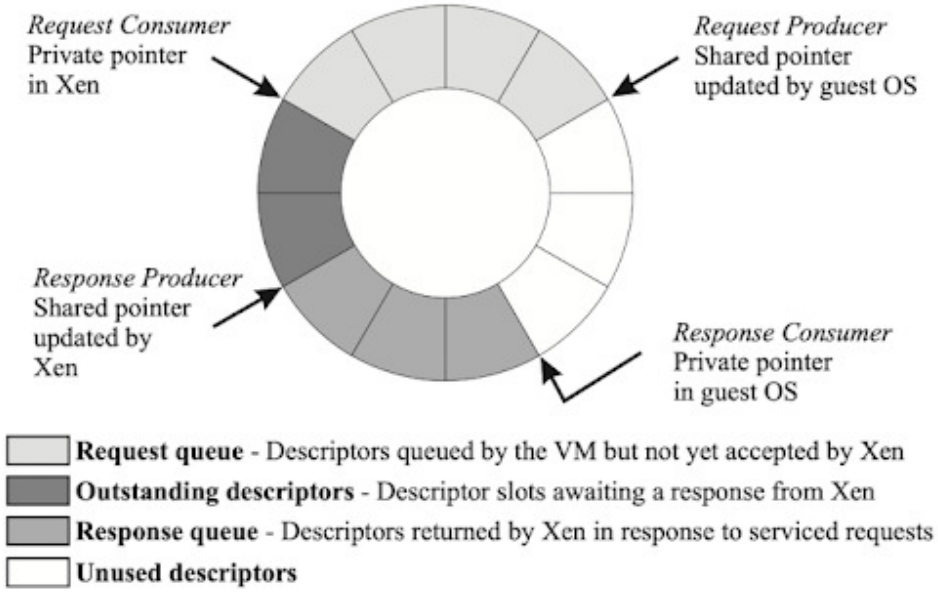


Figure 8: Xen IO ring

Detailed Design of Xen

CPU 가상화 디자인: CPU 가상화 디자인은 크게 4가지 영역으로 구성된다. 먼저 Xen과 도메인간의 컨트롤 전환 디자인이다. 도메인에서 Xen으로 컨트롤 전환이 일어 날때는 Hypercall을 이용한다. Hypercall은 하이퍼바이저로의 동기적 소프트웨어 trap이다. 일반적인 운영체제에서의 시스템콜과 같다고 볼 수 있다. 반대로 Xen에서 도메인으로 컨트롤 전환이 일어날때는 Asynchronous event (비동기적 이벤트)를 이용한다. 비동기적 이벤트는 디바이스 인터럽트를 대체하고, 도메인 종료와 같은 하이퍼바이저 이벤트를 전달할 수 있다. 구체적으로는 도메인마다 지정된 bitmask에 Xen이 이벤트를 쓰고 처리를 기다리는 방식으로 구현한다. 도메인마다 특정 이벤트를 거절하도록 설정 가능할 수도 있다.

둘째로 Xen과 도메인간의 데이터 전달 디자인이다. 실제 데이터의 저장은 도메인과 Xen의 공유 페이지를 사용하고, 이를 관리하는 grant table이 존재한다. I/O request / response의 경우, 그림 8과 같은 I/O ring을 활용한 비동기적 기법을 사용한다. I/O ring은 producer-consumer 모델을 따른다. I/O ring은 도메인마다 Circular queue로 구현되며, Xen과 도메인 모두 접근 가능하다. I/O ring의 요청정보에서 공유 페이지에서 구체적인 데이터 위치를 알 수 있다. I/O ring의 각 디스크립터에는 id가 부여되어 있어서, Xen이 out-of-order로 요청을 처리하여도 된다. 이는 강제적인 사항은 아니다. I/O 인터럽트 혹은 알림(notification)은 Hypercall과 비동기적 이벤트로 처리된다. 구체적으로는, 도메인1이 요청할 내용을 공유 버퍼에 쓰고 요청에 대한 정보를 I/O ring에 쓴다. 이후 hypercall을 통해 Xen에게 해당 사실을 알린다. 비동기적 이벤트를 통해 I/O 응답이 있음을 확인후, I/O ring에서 응답 정보를 가져올 수 있다.

셋째로 CPU 스케쥴링 디자인이 있다. Borrowed Virtual Time scheduling 기법 사용한다. 해당 스케쥴링은 이벤트를 받은 도메인이 빠르게 깨어나게 하기위해 사용한다. 최근에 깨어난 도메인에 좀더 선호도를 두어 일시적으로 fair sharing을 깨는 기법이다. 또한 도메인마다 다른 scheduling parameter를 가질 수 있고, 이는 도메인0의 관리 소프트웨어로 수정 가능하다.

마지막으로 시간 및 타이머 다자인이 있다. Xen이 게스트 운영체제에 제공하는 시간은 real time, virtual time, wall-clock time이 있다. real time은 machine boot 이후 시간을 나타내고, virtual time은 도메인이 실제로 실행된 시간을

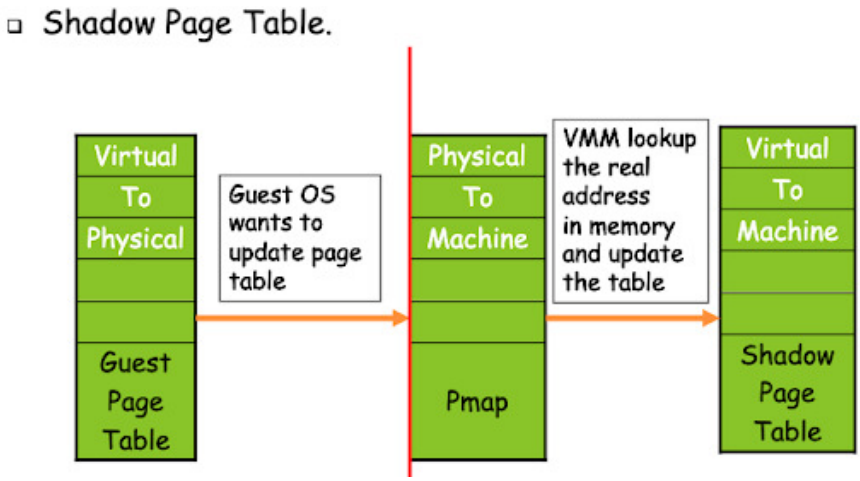


Figure 9: Shadow page table

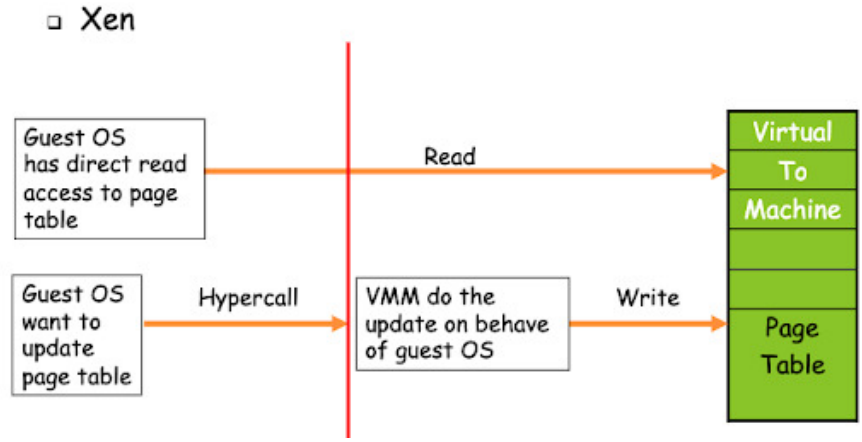


Figure 10: Xen memory virtualization

나타낸다. wall-clock time은 현재 real time에 더해지는 offset값이다. 게스트 운영체제는 real timer과 virtual timer을 각각 프로그램할 수 있고, timeout은 Xen의 이벤트 전달 방식으로 전달된다.

Memory 가상화 디자인: 먼저 가상 주소 번역 디자인이 있다. x86은 페이지 테이블이 하드웨어가 관리하기 때문에 가상화에 어려움이 있다. VMware는 그림 9과 같이 게스트 운영체제에는 virtual page table을 할당하고, 이를 하이퍼바이저상에서 실제 메모리 주소로 맵핑되는 테이블과 virtual page table의 복사본에 해당되는 shadow page table을 가지고 있다. virtual page table을 업데이트시, 하이퍼바이저상에서 물리 주소 맵핑을 찾아서 shadow page table과 virtual page table을 업데이트 하는 방식을 사용한다.

Xen은 그림 10과 같이 게스트의 페이지 테이블을 MMU에 등록할 수 있고, 읽기만 가능하도록 제한을 둔다. 갱신시 hypercall을 통해 Xen에 요청을 하여, 갱신을 할 수 있다. 실제 머신의 page frame에는 type과 reference count가 존재하게 구현한다. Type은 PD(page directory), PT(page table), LDT(local descriptor table), GDT(global descriptor table), RW(writable)로 나뉜다. Type을 활용해 이미 할당된 페이지를 추적할 수 있다. 게스트 운영체제는 1개의 hypercall에 여러개의 요청을 합쳐서 보낼 수 있다.

둘째로 물리 주소 디자인이 있다. 도메인이 만들어 질때 static하게 물리 메모리 실제할당량과 최대할당량이 정해진다. 만약 실제할당량이 최대할당량을 넘지 않았다면, 도메인의 메모리 사용량이 많아질 때 Xen에 추가 할당을

요청할 수 있다. XenoLinux상의 balloon driver가 해당 작업을 수행한다. Xen은 등록만 해주기 때문에, 실제 물리주소로의 맵핑은 게스트 운영체제에게 책임이 있다.

네트워크 디바이스 가상화 디자인: 네트워크 디바이스의 경우, VFR(Virtual Firewall-Router)를 Xen이 제공한다. 각 도메인은 VIF(Virtual Network Interface)를 가진다. VFR, VIF간의 전송을 위한 I/O ring 1개, 수신을 위한 I/O ring 1개가 존재한다. 도메인0의 실제 네트워크 컨트롤러에서 round-round packet scheduler로 처리된다. VFR, VIF간의 교환에서 실제 data는 전혀 copy되지 않고, 공유 버퍼가 packet buffer로 바뀌게 된다. 이는 page flipping기법이라고 불린다. 수신때도 page flipping을 이용하기 위해, 게스트 운영체제는 패킷을 받을 page frame을 지정해주어야 된다.

IO 디바이스 가상화 디자인: Disk와 같은 IO 디바이스의 경우, 도메인0이 실제 물리 디스크에 접근할 수 있는 권한이 있다. 나머지 도메인은 Virtual Block Device(VBD)를 통해 디스크에 접근한다. 도메인0이 VBD를 관리한다. VBD는 I/O ring 기법을 써서 Xen과 데이터를 전달 받는다. 네트워크와 마찬가지로 page flipping 기법을 활용해 데이터를 zero copy를 한다. 각 도메인에서 순서를 보장하기 위해서 reorder barrier를 사용할 수 있다.

Building new Domain: 초기 도메인을 만들기 위해서는 하이퍼바이저가 아닌 도메인0에서 수행한다. 이는 하이퍼바이저의 복잡도를 줄이고, robust하게 하기 위함이다. 도메인0에서 더 나은 진단툴이나 디버깅을 할 수 있는 이유도 있다.

5.3 Xen on ARM [12]

Xen을 x86 아키텍쳐가 아닌 ARM에서 구현시, 2가지 문제가 발생한다. 먼저, ARM에서는 unprivileged mode가 1개 밖에 제공되지 않는다. x86의 경우는 4 ring privileged level을 제공했는데, ARM에서는 Xen이 기존에 사용했던 게스트 운영체제와 유저 어플리케이션이 다른 mode에서 구동되는 방식을 사용하지 못하는 문제가 발생한다. 이를 해결하기 위해 privileged mode에서는 Xen이 구동되고, unprivileged mode에서는 guest OS와 application이 동시에 작동 되게 변경해야 된다.

둘째로 Cache/TLB flushing 오버헤드가 있다. ARM은 virtually indexed virtually tagged (VIVT) 형식의 캐시를 사용하고 있고, TLB에 address space ID (ASID)를 지원하지 않고 있다. 이에 따라 address space를 변경시 캐시와 TLB를 전부 flush해야 되는 문제가 있다. 게스트 운영체제와 유저 어플리케이션이 같은 mode에서 동작하므로 context switching이 동작하게 되면, cache와 TLB의 switching cost가 너무 커지게 된다.

CPU Virtualization

Virtual Privilege modes: 하이퍼바이저가 하드웨어 통제권을 쥐기 위해서 supervisor mode로 작동되며, 게스트 운영체제와 유저 어플리케이션이 user mode로 동작된다. ARM Linux kernel은 원래 supervisor mode에서 작동되게 구현 되었으므로, OS 수정이 필요하다. OS 수정을 최소화하기 위해, User mode를 user process mode, kernel mode 2가지로 분리한다. 위와 같은 가상 privileged mode 구현을 위해, user mode 레지스터를 2개의 banked 레지스터 그룹으로 쪼갠다. 그림 11는 새롭게 가상으로 분리된 mode들을 보여준다. 2개의 mode의 전환시에는 Xen을 거쳐야 하는데, User mode → Supervisor mode → Kernel mode를 거쳐야 게스트의 유저 어플리케이션에서 게스트 커널로 전환이 된다. 이는 Xen만이 가상 mode를 인식하기 때문이다.

Exception handling: mode전환에 Xen을 거치듯이 interrupt, fault, abort, software interrupt 같은 Exception들이 발생시 그림 12과 같이 Xen을 거쳐서 guest OS kernel로 진입한다. 진입 시나리오은 다음과 같다. (1) Exception 발생시 Supervisor mode인 Xen으로 진입한다. (2) ARM의 CPSR(Current Program Status Register)가 SPSR(Saved

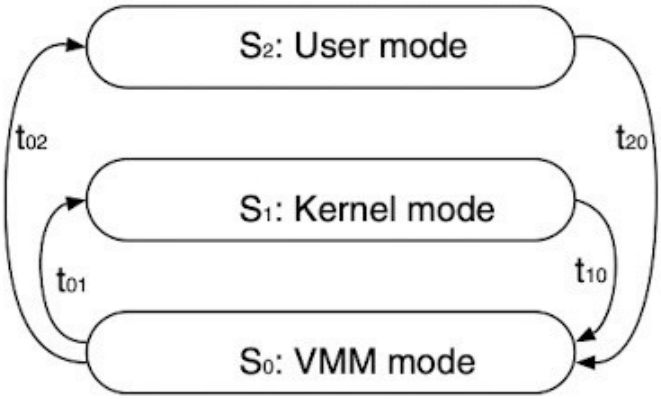


Figure 11: VCPU mode 전환. t_{10}: interrupts, faults, aborts, hypercall 시, t_{20}: interrupts, faults, aborts, system call 시, t_{01}: upcall, return from exception, t_{02}: return from exception

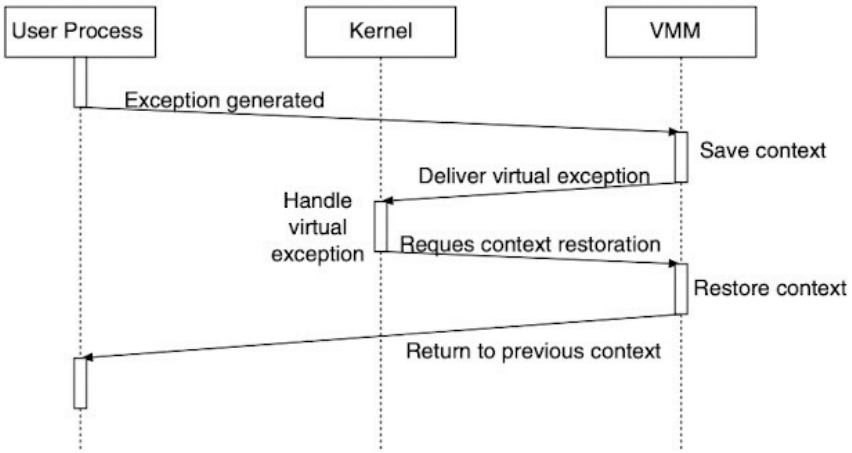


Figure 12: Exception handling in virtualization environment

Program Status Register)로 저장되고, Stack pointer등도 저장된다. (3) 게스트 운영체제의 커널로 진입하기 위해서, SPSR의 내용을 VSPSR(virtual SPSR)에 저장하고, Stack pointer도 VSP(virtual Stack Pointer Register)에 저장한다. (4) Xen이 virtual exception event로 guest OS kernel에 upcall을 요청하고, upcall시 VSPSR, VSP정보등을 kernel stack에 복사한다. (5) 게스트 운영체제의 커널에 진입시, 커널은 virtual exception을 보고 exception vector table을 활용해, exception handler로 점프한다. (6) Exception handler에서 kernel stack에 저장된 SPSR, VSP 값을 확인해서 exception을 처리한다. (7) 게스트 운영체제가 hypercall로 return from exception을 실행한다. (8) Xen이 유저 어플리케이션 context를 복원 후, return from exception을 실행한다. (9) Exception 이후 유저 어플리케이션이 실행 가능하다. 여기서 SPSR, Stack pointer, FAR(Fault Address Register), FSR(Fault Status Register)등은 user level에서 접근이 불가능하기 때문에 Xen이 kernel stack에 copy후 게스트 운영체제 커널로 진입하는 것이다.

Sensitive instruction: 모든 게스트 운영체제의 privileged 명령어을 사용하는 부분을 hypercall로 대체한다.

Memory Virtualization

기존 Xen의 기법과 동일하게, 페이지 테이블 접근은 게스트 운영체제에서 자유롭게 할 수 있으나, 갱신시에는

hypercall을 통해 Xen에 요청을 해야한다. ARM에서 x86과 차이가 나는 부분은 게스트 운영체제와 유저 어플리케이션이 동일한 mode에서 작동되기 때문에, 게스트 운영체제 커널과 유저 어플리케이션의 메모리 보호기법이 추가적으로 필요하다. ARM의 Domain protection mechanism을 이용하여 메모리 보호기법을 추가할 수 있다. Domain protection mechanism이란 page table entry(PTE)에 저장된 access level을 이용한 기법을 말한다. Domain value(access level)은 no access, client, manager 3종류로 나뉜다. no access는 무조건 접근 불가능한 접근 권한을 나타낸다. client은 접근이 가능하나 추가적인 검증이 필요하고, PTE의 AP bit로 read/write/no access 여부 확인해야 하는 접근 권한이다. manager는 무조건 접근 가능한 접근영역이다. Domain은 사용자가 임의로 특정 의미를 부여한 영역이며, Domain중에서 일부에 대한 접근권한을 다르게 주고 싶으면 AP bit를 이용한다. Domain은 16개까지 사용자가 설정가능하다. 예를들어 첫번째 domain은 IO를 위한 영역, 두번째 영역은 User 영역, 세번째 영역은 kernel 영역등으로 설정할 수 있다. Domain의 접근 권한을 설정하는 CP15(DACR: Domain Access Control Register) 레지스터가 존재한다. PTE에는 Domain 번호가 적혀있어 이를 통해 CP15에서 해당 Domain 접근레벨을 알아내어 접근 레벨이 맞지 않으면 access fault를 시킬 수 있다. 해당 논문에서는 16개의 Domain중 D0을 Supervisor mode용, D1을 Kernel mode용, D2를 User process mode용으로 구분한다. user process mode에서는 D2는 client, D1, D0는 no access로 설정되어, D1이나 D0 영역에 접근시 access fault가 발생되어 하이퍼바이저로 trap된다. kernel mode에서는 D0,D1,D2 모두 client로 설정한다. supervisor mode에서는 D0,D1,D2 모두 client로 설정한다. Context switching마다 CP15의 값을 변경해서 mode에 따른 접근 권한 통제가 가능하다.

ARM Virtual Cache optimization기술도 메모리 가상화에 사용된다. ARM의 캐시는 virtually indexed virtually tagged (VIVT) 형식의 캐시를 사용하고 있고, TLB에 address space ID (ASID)를 지원하고 있지 않아서 context switching마다 캐시와 TLB를 flush해주어야 한다. Flush비용이 크기 때문에 최대한 피하는 것이 좋다. 해당 논문에서는 해결책으로 기존 Xen의 기법인 Xen이 사용하는 공간을 process address space에 64mb 두는 것을 이용한다. Xen이 0xFC000000 - 0xFFFFFFFF, 게스트 운영체제 커널이 0xC0000000 to 0xFBFFFFFF, 프로세스가 0x0 to 0xBFFFFFFF으로 할당된다. 따라서 프로세스에서 하이퍼바이저로 변경시, 캐시나 TLB flush가 불필요하다. 단, 게스트 운영체제 도메인간의 전환이나 프로세스간의 전환이 있을 때는 cache/TLB flush가 필요하다. 추가적으로 ARM이 8개의 lockdown TLB entries를 제공하는데, 이는 TLB flush가 발생해도 invalid처리가 되지 않는 TLB entry이다. 8개중 2개의 entry를 Xen 메모리 영역에 고정시켜, TLB flush 오버헤드를 추가적으로 줄일수 있다.

Inter-Domain Memory Isolation기술도 메모리 가상화에 사용된다. Hypercall로 페이지 테이블 갱신를 요청하므로, Xen에서 도메인내부의 맵핑인지 타 도메인 맵핑인지 확인 후, 유효하지 않은 할당일 경우 차단하는 방법이다.

6 Type 2 하이퍼바이저

이장에서는 대표적인 Type2 하이퍼바이저인 KVM의 디자인과 구현을 살펴본다.

6.1 KVM/ARM: The Linux ARM Hypervisor [8] [9]

Split-mode Virtualization

KVM을 ARM의 새로운 가상화 mode인 Hyp mode에서 바로 구동하는 것이 가장 이상적이나, 2가지 문제가 존재한다. 먼저, 원본 Linux은 kernel mode에서 구동되게 설계되어서, 수정없이는 hyp mode에서 구동이 불가능하다. Hyp mode는 kernel mode와 다른 레지스터 세트을 쓰기 때문이다. 둘째로, hyp mode에서 구동되게 수정되어도, 성능 문제가 발생한다. Hyp mode는 page table register가 1개 밖에 없어서, 특정 주소 공간 이상은 쓸 수가 없다. 특정 공간을 쓰기 위해서는, page table register를 계속 바꿔야 해서 성능 문제가 발생한다.

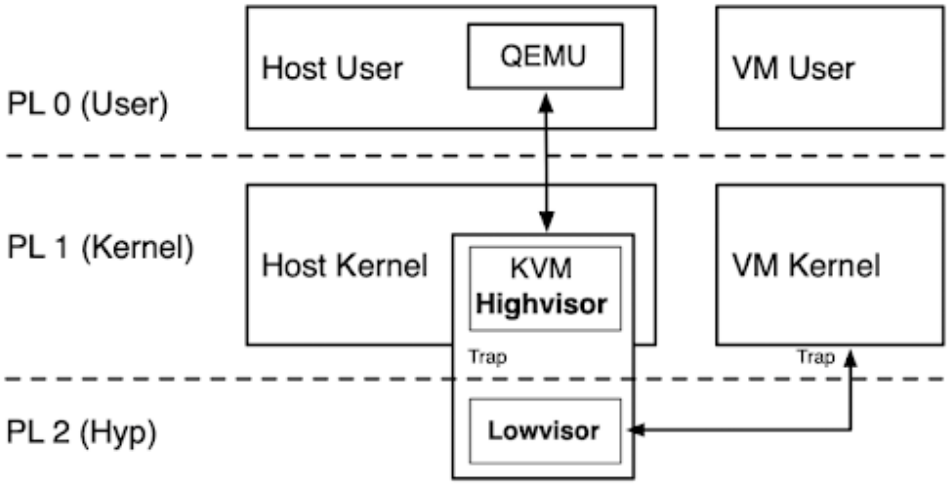


Figure 13: 분리된 KVM 구조

Hyp mode에서의 ARM virtualization extension 기능과 kernel mode에서의 Linux kernel 기능을 전부 사용하기 위해서 KVM은 그림 13과 같이 split-mode virtualization을 사용한다. Hyp mode에서 동작하는 Lowvisor는 3가지 기능을 제공한다. 먼저, ARM에서 제공하는 보호 기능을 이용하여 가상머신간의 보호를 제공한다. 둘째로, world switch 기능을 제공한다. 마지막으로, virtualization trap handler 기능을 제공한다. Kernel mode에서 동작하는 Highvisor는 Locking mechanism, memory allocation 등 기존 Linux 기능을 활용한다. ARM 머신 위의 KVM에서 하이퍼바이저로의 trap은 trap to lowvisor와 trap to highvisor가 합쳐서 double trap을 유발한다. Highvisor와 lowvisor의 데이터 공유를 위해 memory mapping이 동일한 주소로 되도록 설정한다. 주소의 맵핑은 기존 Linux memory management system을 그대로 활용한다.

CPU Virtualization

Lowvisor는 Hyp mode에서 동작하고, hyp mode용 register set을 따로 사용한다. 일반적으로는 kernel mode와 user mode에서는 hyp mode용 register set을 접근이 불가능 하다. Kernel mode와 user mode에서 소프트웨어적으로 접근 가능한 register set은 표 2와 같다.

VM isolation 이나 전체 머신에 영향을 미치는 레지스터가 아닌 경우는 가상머신 전환시 레지스터를 stack에 저장하고 복원하는 방식을 적용한다. 예를들어, TTBR_EL1같은 base page table register같은 경우는 hyp mode로의 trap 없이 kernel mode에서 접근가능하다. 하이퍼바이저 및 전체 system에 관련된 레지스터에 영향을 미치거나, 특정 명령어는 trap-and-emulate 방식을 사용한다. 예를들어, WFI 명령어로 kernel mode에서 CPU를 재우려고하면, 하이퍼바이저는 실제로는 해당 가상머신 할당을 하지 않는 방식으로 에뮬레이션한다. Floating point를 다루는 VFP 레지스터들은 hyp mode에서는 사용되지 않기 때문에 lazy context switch를 한다. Lazy context switch는 실제 world switch시에 VFP를 저장하는 것이 아닌, 다음에 다른 가상머신에 의해서 VFP가 실제로 사용될 때, hyp mode로 trap해서 hyp stack에 저장 후 복원 작업을 거친다. Lazy context switch를 적용할 때는 HCR_EL2을 이용해 VFP를 접근시 trap되게 설정해 둘 수 있다. 2가지 구체적인 world switch 과정을 아래에서 서술한다.

호스트에서 가상머신으로 world switch 과정 (1) 호스트의 GP register를 hyp stack에 저장한다. (2) 가상머신의 VGIC를 설정한다. (3) 가상머신의 timer를 설정한다. (4) 호스트의 configuration register를 hyp stack에 저장한다. (5) 가상머신의 configuration register를 복원한다. (6) Hyp mode로의 trap을 설정한다. (7) VFP lazy context switch를

Action	Nr.	State
Context Switch	38	General Purpose (GP) Registers
	26	Control Registers
	16	VGIC Control Registers
	4	VGIC List Registers
	2	Arch Timer Control Registers
	32	64-bit VFP Registers
	4	32-bit VFP Control Registers
Trap-and-Emulate	-	CP14 Trace Registers
	-	WFI Instructions
	-	SMC Instructions
	-	ACTLR Access
	-	Cache ops. by Set/Way
	-	L2CTLR / L2ECTLR Registers

Table 2: VM and Host State on a Cortex-A15

설정한다. (8) CPU halt 명령어(WFI/WFE) 사용시 하이퍼바이저로 trap이 되게 설정한다. (9) 인터럽트에 trap되게 설정한다. (10) SMC 명령어 사용시 trap이 되게 설정한다. (11) 특정 레지스터에 접근시 trap이 되게 설정한다. (12) 가상머신의 ID를 shadow ID register에 기록한다. (13) Stage-2 page table register(VTTBR)을 설정하고 및 Stage-2 translation 기능을 켠다. (14) 가상머신의 GP register를 복원한다. (15) 가상머신의 kernel mode나 user mode로 trap 한다.

가상머신에서 호스트로 world switch 과정 (1) 가상머신의 GP register를 저장한다. (2) Stage-2 translation 기능을 끈다. (3) Hyp mode로의 trap 기능을 끈다. (4) 가상머신의 configuration register를 저장한다. (5) 호스트의 configuration register를 복원한다. (6) 호스트의 timer를 설정한다. (7) 가상머신의 VGIC를 저장한다. (8) 호스트의 GP register 복원한다. (9) 호스트의 커널로 trap한다.

Memory Virtualization

Stage-2 translation 기능으로 가상머신과 KVM highvisor가 동일한 kernel mode로 구동되지만, 메모리 영역은 다르게 가져가는 것이 가능하다. 가상머신에게 유효하지 않은 메모리 영역 접근시 Stage-2 page fault가 발생한다. Stage-2 page fault handler는 highvisor에 존재하며, highvisor는 가상머신에게 할당된 메모리 영역인지 확인한다. 할당된 메모리 영역이 맞다면, 기존 Linux의 페이지 할당 시스템을 통해 페이지를 할당하고 맵핑해준다. 할당된 메모리 영역이 아니면, segmentation fault가 발생한다.

I/O Virtualization

ARM 아키텍처에서는 모든 I/O가 Memory Mapped IO (MMIO)를 이용하므로 Load/Store 명령어를 이용해서 I/O 작업이 가능하다. MMIO 메모리 영역은 가상머신에서는 접근 불가능하게 맵핑을 설정한다. MMIO 메모리 영역 접근시 Stage-2 page fault가 발생하여, fault address를 이용해서 호스트 운영체제의 QEMU나 Virtio를 통해 에뮬레이션된 디바이스에 접근가능하다.

Interrupt Virtualization

Linux의 interrupt handler를 그대로 이용한다. 하드웨어 인터럽트시 호스트 linux 운영체제로의 world switch가 발생하고 이를 interrupt handler로 처리한다. I/O 완료 같은 인터럽트는 가상머신이 받아야할 필요가 있는데, 이 경우는 virtual interrupt injection을 이용한다. Hyp mode에서 VGIC의 list register를 프로그래밍해서, virtual interrupt를 만들어 가상머신에게 전달 가능하다. 해당 virtual interrupt는 가상머신이 다음에 스케쥴링 되어 구동될 때 인식할 수 있다. List register는 hyp mode에서만 접근 가능하나, virtual interrupt가 저장되는 virtual CPU interface는 가상머신이 hyp mode로의 trap없이 접근 가능하다. 이후 hyp mode를 거칠 때, virtual CPU interface를 확인해서 인터럽트 ACK를 확인 할 수 있다. 가상머신이 GIC의 distributor를 설정하고 싶거나, IPI를 발생시키는 경우는 hyp mode로의 trap이 발생하게 된다. 해당 처리를 위해 소프트웨어적으로 virtual distributor를 highvisor의 기능으로 둔다. Virtual distributor는 인터럽트 정보를 소프트웨어적으로 저장하고, 다음 가상머신이 스케쥴링 될 때 해당 정보를 이용해서 virtual interrupt injection을 수행한다. 원래는 가상머신과 하이퍼바이저간의 전환 후 기존 가상머신이 그대로 할당되는 경우는 VGIC를 저장할 필요가 없으나, 구현을 쉽게 하기위해 모든 world switch시 VGIC정보를 저장하도록 구현했다. VGIC정보를 저장하는 과정이 KVM on ARM에서 world switch시 가장 큰 오버헤드를 차지한다. Lazy context switching 기법을 일부 VGIC에 적용시키면 오버헤드를 줄일 수 있을 것으로 예상한다.

Timer Virtualization

physical timer는 hyp mode에서 통제한다. virtual timer는 kernel mode에서 hyp mode로의 trap없이 접근 가능하다. virtual timer는 virtual interrupt를 바로 발생시키지 못하기 때문에, hardware interrupt를 발생시켜 lowvisor로의 trap이 발생하게 한다. 이후 virtual interrupt injection을 통해 가상머신 virtual timer interrupt를 전달한다. 해당 기법의 다른문제는 CPU core당 virtual timer가 1개 밖에 없기 때문에 core에 여러개의 가상머신이 존재하는 경우이다. 가상머신이 하이퍼바이저로 trap될때, 끝나지 않은 timer를 감지해서 software timer를 만들어 저장하고, 해당 timer가 만료될 때 callback function을 활용해 virtual interrupt injection을 발생시키게 설정해두어 문제를 해결했다. 해당 과정을 통해 가상머신이 할당되지 않았더라도 timer시간을 측정할 수 있게 한다.

KVM on ARM versus Xen on ARM Hypervisor

표 3은 ARM 머신상에서 KVM과 Xen의 구현 동작 차이를 보여준다. KVM의 경우 호스트 운영체제인 Linux가

	KVM	Xen
구성	Highvisor in EL1 Lowvisor in EL2	Hypervisor in EL2
가상머신 trap	EL1 가상머신 trap발생 KVM in EL2(Lowvisor) KVM in EL1(Highvisor) Host kernel in EL1	EL1 가상머신 trap발생 Xen in EL2
I/O 발생	VM I/O Trap to EL2(Lowvisor) HostOS EL1 switching HostOS가 I/O 처리	VM I/O Trap to EL2(Xen) Signal to Dom0 Scheduling Dom0 Dom0에서 I/O 처리

Table 3: KVM on ARM vs Xen on ARM

EL1에서 구동되기 때문에 EL1 레지스터를 context 저장 및 복원하는 과정이 추가적으로 존재해서, 가상머신에서

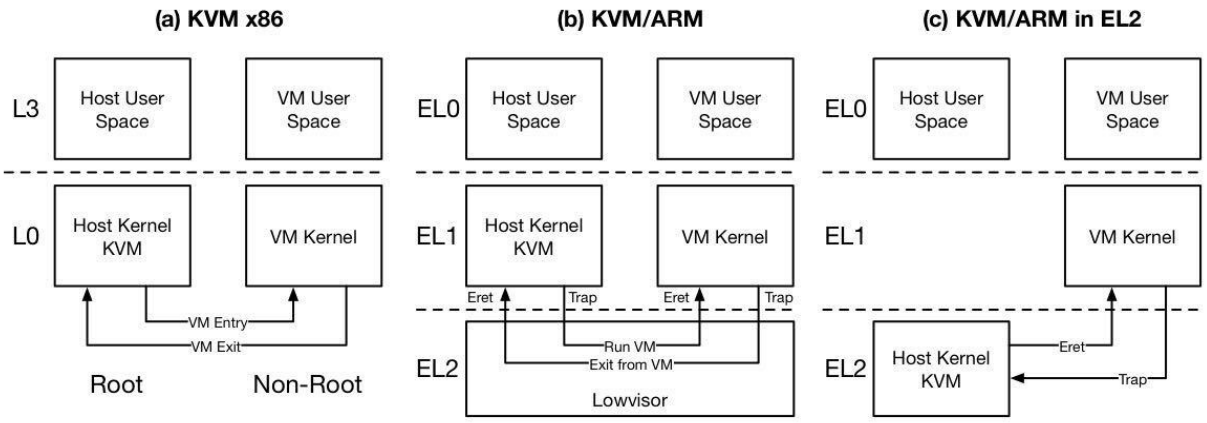


Figure 14: 하이퍼바이저 디자인과 CPU Privilege Level

하이퍼바이저로의 전환시간이 더 오래 걸린다. KVM on x86의 경우, root mode가 일반적인 명령어과 레지스터등을 사용할 수 있기 때문에 ARM처럼 highvisor, lowvisor를 분리할 필요가 없다. IO처리에서 2가지 경우 모두 하이퍼바이저로의 전환을 줄이는 방향으로 I/O를 진행하기 위해서 paravirtual I/O를 쓰는 것이 일반적이다. paravirtual I/O는 하이퍼바이저에서 구현된 가상 디바이스를 사용하는 가상머신안에 디바이스 드라이버를 이용한 것인데, virtual device front-end를 가상머신은 device driver로 인식하고 사용한다. KVM의 경우, Virtio protocol을 사용하며 virtual device backend를 호스트 운영체제에 구현한다. Xen의 경우, Xen PV를 사용하며 virtual device backend를 Dom0 커널에 구현한다.

6.2 Optimizing the Design and Implementation of the Linux ARM Hypervisor [7]

Original KVM Problem

KVM의 경우 linux의 기존 기능을 활용하기 위해서 하이퍼바이저 운영체제 커널을 EL1에 두게 되고, Xen의 경우도 Dom0의 기능을 활용하기 위해서 하이퍼바이저 운영체제 커널을 EL1에 두게 된다. 두 경우 모두, 하이퍼바이저 운영체제 커널과 게스트 운영체제 커널은 동일한 EL1에서 동작하게 되고, 하이퍼바이저와 가상머신 전환시에 EL1 레지스터의 저장과 복원과정이 동반된다. Intel VMX의 경우, 해당 과정을 하드웨어적으로 처리를 하고 ARM은 소프트웨어적으로 해당 과정을 처리하므로, ARM의 전환 오버헤드가 일반적으로 더 크다.

New Hypervisor architecture

그림 14c) 처럼 하이퍼바이저 운영체제 커널을 수정없이 EL2에서 하이퍼바이저와 동일한 레벨에서 동작시킨다면 해당 오버헤드를 상당히 줄일 수 있다. EL2에서 EL1의 전환, 혹은 그 반대 경우에는 EL1의 register를 저장하고 복원하는 과정이 필요없기 때문에, 빠른 전환이 가능하다. 해당 과정을 하드웨어적으로 지원하기 위해서, ARM은 Virtualization Host Extensions (VHE)를 추가했다. VHE 기능이 켜진 상태라면, 기존 운영체제를 수정없이 EL2에서 구동 가능하다. VHE 기능이 꺼진 상태 혹은 없다면, 새로운 하이퍼바이저 운영체제 커널을 EL2에서 동작시키기 위해서는 기존 하이퍼바이저 운영체제를 수정해서 EL2에서 동작되게 해야한다. 이는 크게 3가지 문제를 동반한다.

먼저, EL2가 EL1과 control register set이 다르고, 해당 레지스터를 접근하는 명령어가 다른 문제가 있다. 둘째로, EL1과 EL2의 페이지 테이블 구조가 다르고, address space 범위를 다르게 지니는 문제가 있다. EL2는 32bit TTBR 밖에 없으므로, 일정 주소공간이상을 활용 불가능하다. 셋째로, EL2 단독으로 EL0의 유저 어플리케이션을 구동할

수 있도록 address space를 구성하는 것이 불가능하다. EL1이 없으므로, system call, hardware interrupt, page fault 같은 exception들이 전부 EL2에서 처리되어야 하는데, HCR_EL2 register의 Trap General Exceptions (TGE) bit를 활용해 모든 exception이 EL2에서 처리되게 할 수는 있다. 하지만, TGE bit가 활성화 되면, EL0의 virtual memory기능이 꺼지게 되어 유저 어플리케이션의 address space를 정상적으로 구현하기 힘들다. 기존 KVM은 이를 해결하기 위해서 highvisor, lowvisor로 기능을 쪼개서 하이퍼바이저를 구현했다.

Advantages of New Hypervisor Architecture

새로운 하이퍼바이저 구조는 3가지 장점을 가진다. 먼저, 가상머신과 하이퍼바이저 사이의 전환시 레지스터의 저장 및 복원 과정이 없어도 된다. 둘째로, EL2에서만 접근가능한 정보를 EL1에서 보기 위해 두었던 intermediate data structure들을 더이상 복사할 필요가 없이, 하이퍼바이저 운영체제 커널이 직접접근 가능하다. 마지막으로, 하이퍼바이저와 하이퍼바이저 운영체제 커널이 다른 address space를 가질 필요 없이 동일한 address space를 사용 가능하다.

Virtualization Host Extensions (VHE)

ARMv8.1에서 추가된 기능으로, EL2에 레지스터들을 추가 및 수정해, EL1과 동일한 기능을 제공할 수 있게 한다. EL2에서 프로세서가 동작시, EL1 레지스터에 접근하는 명령어들을 자동적으로 EL2에 접근하게 만든다. EL2에서 직접적으로 EL0에서 virtual memory를 사용해 address space를 만들 수 있게 지원한다. EL1을 통하지 않고 모든 exception이 EL2로 통하게 설정 가능하다. EL1과 EL2의 page format을 통일한다. 부팅시에 VHE기능을 bit on시 적용된다.

가상머신의 EL1 정보에 하이퍼바이저가 접근하기 위해서는 기존 명령어로는 수행이 불가능하다. 하이퍼바이저도 EL2에 있으므로, 접근시 EL2에 자동접근 된다. _EL12 명령어들을 추가함으로 해결했다. VHE가 켜진 상태에서 _EL12명령어 사용시, EL2 상태에서 EL1 레지스터에 접근 가능하다. 단, 하이퍼바이저는 VHE가 켜진 상태에서는 _EL12 명령어를 사용하도록 수정이 필요하다.

el2Linux

VHE가 없는 상황에서, 소프트웨어적으로 EL2에서 linux를 구동하기 위해 수정된 운영체제이다. 먼저, EL1 레지스터에 접근하는 것을 전부 EL2 레지스터에 접근하는 것으로 수정한다. 둘째로, TGE bit를 사용하지 않고, EL1에 exception vector만을 두어서 hypercall을 활용해 exception을 EL2로 전달한다. 이때, EL1을 거치는 오버헤드와 EL1에 존재하는 runtime과 게스트 운영체제 커널간의 저장 및 복원 과정 overhead가 추가로 존재한다. 단, 이과정은 호스트의 유저 어플리케이션에 접근시에만 적용되므로, 단순한 가상머신과 하이퍼바이저 전환에서는 오버헤드가 아니다. KVM은 x86, ARM에 상관없이 호스트의 user space접근을 최대한 피하는 방향으로 최적화가 진행되었다. 셋째로, EL1와 EL2의 페이지 테이블구조가 다른 것을 해결하기 위한 구조 추가한다. EL1 페이지 테이블의 경우 executable이 EL0 bit과 EL1 bit으로 구분되어 권한 부여가 가능하다. EL2 페이지 테이블은 executable이 1개의 bit로 구성되기 때문에, EL0와 EL2가 둘다 executable하거나 둘다 안된다. EL2에서 커널 코드는 executable해야 하기 때문에 EL0도 커널 페이지를 executable하게 될 수 있다. 커널 페이지가 read, write되는 것은 막을 수 있으나 privileged user가 커널 페이지를 executable하는 보안 이슈가 생길 수 있다. el2Linux는 해당 보안 이슈를 허용하는 방향으로 EL1, EL2에서 페이지 테이블 구조가 유사하게 만든다. EL2 페이지 테이블은 Address Space Identifier(ASID)가 없어서 TLB invalidation이 자주 발생한다. 이는 프로세스 전환시에 EL2관련 entry들을 매번 invalid 처리를 한다. EL0는 ASID로 TLB에 tag되어서 프로세스 전환시 invalid 처리가 될 필요가 없다. el2Linux는 TLB invalidation을 감수하기로 한다. EL2 페이지 테이블은 virtual address로 48bit주소를 받는데, 이를 47bit 2개로

쪼개서 최상위 bit가 0일때를 user space영역, 최상위 bit가 1일때를 kernel space영역으로 포인팅하도록 설정한다. 이를 통해 EL2 페이지 테이블만으로 EL1/EL0 페이지 테이블을 가진 것처럼 위장한다.

Hypervisor Redesign
하이퍼바이저 또한 EL2에서 구동되는 linux를 위해 수정이 필요하다. 첫번째로, 가상머신과 하이퍼바이저사이의 단순 전환시, EL1 레지스터를 저장 및 복원하는 과정 삭제한다. 즉, 다른 가상머신이 구동될때에만, EL1 레지스터를 저장 및 복원한다. 둘째로, 가상머신과 하이퍼바이저사이의 자료 교환을 위해 EL1/EL2 정보를 저장하는 과정을 삭제한다. 이를 통해 같은 자료를 복사하는 것도 해결할 수 있고, EL1과 EL2가 address space가 달라서, TLB miss가 나는 문제도 해결된다. EL2에 있는 호스트 커널이 직접 모든 하드웨어 및 레지스터를 접근 가능하다. 셋째로, 가상머신과 하이퍼바이저간 전환시, 가상화 기능을 켜고 끄는 과정 삭제한다. 기존에는 EL1의 호스트 운영체제가 하드웨어 통제권을 쥐기 위해서, 가상화 기능을 꺼야 했지만 EL2에서 구동되면 해당 과정이 불필요하다. 또한 stage 2 translation, virtual interrupts, traps on sensitive instruction등은 EL1과 EL0에만 해당되므로 끌 필요 없다. 가상화 기능이 꺼질 때는, 호스트 유저 어플리케이션이 구동될 때만 적용한다. 마지막으로, EL2에서 구동되지 않는 linux 또한 지원하기 위해서 코드에 조건문을 추가하는 것이 아닌, static key infrastructure 이용한다. 런타임에 instruction flow를 정해서 실행한다. unconditional jump는 no-ops로 대체한다.

7 IO 가상화

이장에서는 IO 장치에 대한 가상화 기법들을 살펴본다.

7.1 QEMU, a Fast and Portable Dynamic Translator [4]

QEMU는 기본적으로 full system emulator라고 볼 수 있다. 에뮬레이터라는 말은 기본적으로 실제 하드웨어가 아닌 것을 실제 하드웨어에서 동작시키는 시스템을 의미한다. 예를 들어, ARM기반으로 작성된 안드로이드 어플리케이션은 x86 시스템에서는 바로 구동이 불가능하나, Emulator를 통해서 구동이 가능하다. QEMU는 특정 어플리케이션(프로세스) 뿐 아니라 운영체제 또한 full virtualization을 할 수 있다. QEMU는 CPU emulator, Emulated device, Generic device, Machine description, Debugger, User interface 등으로 구성된다. 가장 유명한 QEMU-KVM 시스템에서는 QEMU는 device emulation으로 I/O 가상화에 사용되고, KVM은 CPU, memory, interrupt, timer를 가상화하는데 사용된다. QEMU가 에뮬레이션을 하는 방식은 dynamic translation을 이용한다. 이는 런타임에 기존 명령어셋을 호스트의 명렁어셋으로 번역하는 것을 말한다. 번역결과는 캐시에 저장되어, 특정 명령어가 2번이상 fetch 및 decode 되지 않게한다. QEMU는 다른 dynamic translator와 다르게 이식성을 위해 2단계 번역을 한다. 먼저, Target 명령어를 우선 C code로 변환을 하고, 이를 GNU C 컴파일러를 활용해 machine code로 만든다. 이후, 번역된 machine code는 concatenate하여 관리한다.

Portable Dynamic Translation
번역기 제작과정 먼저, target CPU 명령어를 micro operation이라는 간단한 명령어 집합으로 바꾼다. 이 과정에서 micro operation은 C 코드로 구성되고, 모두 hand code로 실행된다. 또한 속도가 아닌 가독성과 압축성을 기준으로 최적화한다. 둘째로, micro operation이 든 c code는 GCC에 의해 object 파일 된다. 셋째로, dyngen이라는 컴파일 도구를 사용해 micro opertaion이 포함된 object파일을 dynamic code generator로 만든다. 마지막으로, Dynamic code generator는 런타임에 target CPU 명령어들을 micro operation들이 연결된 host function으로 변경한다. QEMU는

micro operation에 constant parameter를 줄 수 있고, 이를 활용하면 GCC를 통해 dummy code relocation을 할 수 있다. 최종적으로 dyngen에 의해 relocation을 할 수 있다.

예시 `addi r1,r1,-16` 이라는 PowerPC code를 x86 code로 변환하는 경우를 보자. 첫번째 단계에서 해당 명령어는 아래와 같이 c code로 구성된다.

```
movl_T0_r1;
addl_T0_im;
movl_r1_T0;

void op_movl_T0_r1 (void)
{
    T0 = env->regs[1];
    // env는 target CPU state를 담고 있는 array
}

extern int __op_param1;
void op_addl_T0_im (void)
{
    T0 = T0 + ((long) (&__op_param1));
    // runtime에 결정
}
```

여기서 모든 move를 고려하는 것이 아닌 특정 temp register만 사용된다. 예시에서는 T0, T1, T2만을 사용하고, `register` 변수 사용으로 변수를 레지스터로 고정한다. 두번째 단계를 거쳐 c code는 object file이 되고, object 파일은 세번째 단계를 거쳐 dyngen에 의해 dynamic code generator가 된다. code generator는 opc_ptr로 현재 operation을 가리키고, gen_code_ptr로 output host code를 가리킨다. 예시의 -16 같이 runtime에 결정되는 파라미터는 opparam_ptr에서 가리키고 있다. 최종적인 동작 과정은 아래와 같다.

```
[...]
for(;;) {
    switch (*opc_ptr++){
        [...]
        case INDEX_op_movl_T0_r1:
        {
            extern void op_movl_T0_r1 ();
            memcpy(gen_code_ptr,
            (char *)&op_movl_T0_r1+0, 3);
            gen_code_ptr += 3;
            break;
        }
        case INDEX_op_addl_T0_im:
```

```
        {
            long param1;
            extern void op_addl_T0_im ();
            memcpy(gen_code_ptr,
            (char *)&op_addl_T0_im+0, 6);
            param1 = *opparam_ptr++;
            *(uint32_t *)(gen_code_ptr + 2)
            = param1;
            gen_code_ptr += 6;
            break;
        }
        [...]
    }
    [...]
}
[...]
```

일반적인 경우는 gen_code_ptr로 code가 복사되지만, constant parameter를 사용하는 경우는 relocation을 하여 constant parameter를 적용한다. 위의 dynamic translator가 구동되면 `addi r1,r1,-16`는 아래와 같이 변한다.

```
# movl_T0_r1
# ebx = env->regs[1]
mov 0x4(%ebp), %ebx
# addl_T0_im - 16
# ebx = ebx - 16
add $0xfffffff0, %ebx
# movl_r1_T0
# env0>regs[1] = ebx
mov %ebx, 0x4(%ebp)
```

Dyngen 구현 object file을 파싱해서 symbol table, relocation entries, code section을 얻는다. symbol table을 통해 code section에 위치한 micro operation은 그대로 `memcpy`로 복사하는 방식을 적용한다. constant parameter를 사용하는 relocation이 필요한 micro operation은 __op_paramN을 사용하는 점을 이용해 감지하고, relocation을 한다. `memcpy`시 일반적으로 function prologue와 epilogue는 스킵된다. 즉, stack에 context 저장 및 복원을 하지 않는다. GCC에서 컴파일시, dummy assembly macro를 이용해 1개의 함수내에서 return이 1군데에서만 발생하게 강제한다.

Implementation details

Translated Blocks and Translation Cache: 일반적으로 QEMU가 target code를 번역할 때, 다음 `jump` 명령어나 static CPU state를 변경하는 명령어를 만날 때까지 번역한다. 이러한 단위를 Translated Blocks (TBs)라고 부른다. 또한 16MB의 캐시를 두어, 최근 사용된 TB를 저장한다. 간단한 구현을 위해, 캐시가 가득차면 전부 비운다. static

CPU state는 PC나 CPL(Current Privileged Level)등이 있으며, 이를 변경하는 경우 번역이 달라지므로 새로운 TB 단위를 할당한다. 여기서 PC변경은 자동적으로 되는 것이 아니라 `jump`등으로 달라지는 경우를 말한다.

Register Allocation: 특정 target CPU 레지스터는 호스트의 특정 레지스터나 메모리 주소에 고정된다. 주로 target 레지스터는 특정 메모리 주소에 맵핑되고, 호스트 레지스터에 일부만 저장된다. 이러한 고정은 hand code로 작성되어 있다.

Condition code optimization: condition code를 에뮬레이션할때는, lazy condition code evaluation을 사용한다. condition code를 바로 계산하지 않고, 실제로 사용되는 경우에 계산을 실행한다.

Direct block chaining: 다음 TB를 찾을 때, 시뮬레이션된 PC와 다른 static CPU state를 이용한 hash table을 통해 다음 TB를 찾는다. 다음 TB의 주소를 아는 경우는 direct jump를 하여 바로 실행하고, 다음 TB가 번역이 안된 경우는 새로운 번역을 한다.

Memory management: target MMU를 에뮬레이션하기 위해서 QEMU는 `mmap()` 시스템콜을 사용한다. 또한 Software MMU를 지원한다. Software MMU시 번역과정을 빠르게 하기 위해 캐시를 사용한다. MMU 맵핑이 변할때마다 cache flush하는 경우를 막기 위해, physically indexed translation cache를 이용한다.

Exception support: exception 발생시 target CPU state를 바로 저장해서 Exception이 처리 될 수 있게 한다.

Hardware interrupts: QEMU는 매 TB마다 하드웨어 인터럽트가 대기하는 지 확인하지 않는다. 유저가 비동기적으로 특정함수를 불러서 인터럽트 대기를 확인해야된다. 특정함수를 부르면 현재 진행중인 TB의 연결이 끊기게 되어, 빠르게 CPU 에뮬레이터의 메인 루프로 돌아오게 되어 인터럽트 확인이 가능하다.

QEMU Architecture

KVM은 vCPU (virtual CPU), virtual memory, interrupt, timer만을 제공하기 때문에 온전한 가상머신을 만들 수 없다. 일반적인 디바이스는 하나의 시스템에서 이용하도록 설계되었는데, 이를 하이퍼바이저에서 이용하기 위해 에뮬레이션된 디바이스가 필요하다. 이러한 디바이스의 가상화를 위해 사용되는 것이 QEMU이다. QEMU 자체적으로 vCPU, 메모리도 가상화 할 수 있어서 KVM 없이도 하이퍼바이저로 동작가능 하나 성능이 매우 낮다. 따라서 KVM-QEMU 조합이 일반적으로 사용된다.

QEMU의 Architecture는 Parallel Architecture와 Event-Driven Architecture이 결합된 Hybrid Architecture를 취하고 있다. Parallel Architecture는 어떤 요청에 대한 처리가 있을때마다 Thread를 생성하여 병렬적으로 처리하는 Architecture이다. 요청에 대한 반응성은 좋지만 Thread의 개수가 늘어날 수록 Context Switching Overhead가 커지기 때문에, 전체적인 성능이 떨어지게 된다. Event-Driven Architecture는 하나의 Main Loop Thread가 돌면서 Event 발생을 검사하고 발생한 Event가 있으면 해당 Event의 Event Handler를 수행시키는 구조이다. 하나의 Thread만을 이용하기 때문에 Context Switching Overhead가 적다는 장점을 가지고 있지만, Event Handler에서의 Event 처리 시간이 길어지면 전반직인 Event의 반응성이 크게 떨어진다는 단점을 가지고 있다. QEMU는 하나의 Main Loop Thread에서 대부분의 Event 및 관련 연산을 처리하고, CPU Intensive한 처리는 별도의 Worker Thread에 할당하여 처리한다. Main Loop Thread에서는 File Descriptor를 이용하여 Event를 확인하고 처리한다. Worker Thread는 할당받은 일의 처리가 완료되면 Main Loop Thread에 처리 완료 Event를 전달하고 종료한다. QEMU는 가상 머신이 이용하는 vCPU 처리를 Main Loop Thread에서 처리하는 방식과 별도의 Thread에서 처리하는 방식 2가지를 선택할 수 있는데, 전자의 방식을 non-iothread 처리 방식이라고 하고 후자의 방식을 iothread를 이용하는 처리 방식이라고 한다.

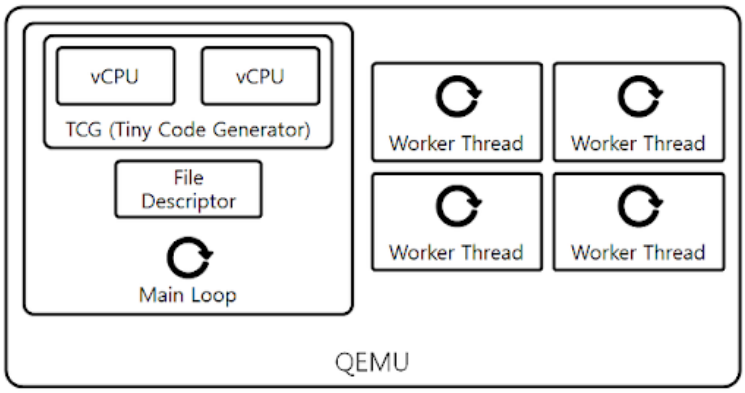


Figure 15: QEMU with non-iothread

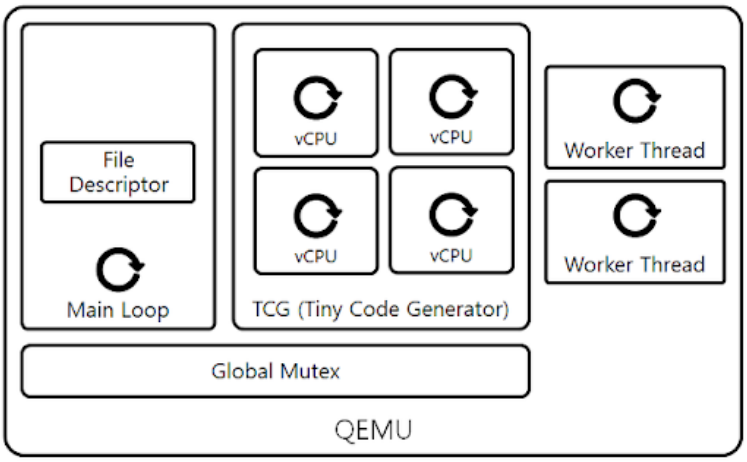


Figure 16: QEMU with iothread

QEMU with non-iothread: non-iothread 방식은 초기 QEMU Architecture으로써 Main Loop Thread에서 vCPU 처리와 Event를 같이 처리한다. 즉 하나의 Thread에서 vCPU 처리와 대부분의 장치 Emulation을 같이하는 구조이다. 그림 15에서 TCG(Tiny Code Generator)는 vCPU를 Emulation하는 QEMU의 모듈이다. 가상 머신의 2개의 vCPU를 가지고 있더라도 Main Loop Thread에서만 Multiplexing되어 처리되기 때문에, 가상 머신이 여러개의 vCPU를 가지고 있더라도 실제로는 병렬적으로 처리되지 않는다. 따라서 non-iothread 구조의 가상 머신은 매우 느릴수 밖에 없다.

QEMU with iothread: iothread 방식은 그림 16처럼 Main Loop Thread에서는 Event만 처리하고 각 vCPU마다 Thread를 할당하여 처리하는 방식이다. Main Loop Thread뿐만 아니라 모든 vCPU Thread에서도 장치 에뮬레이션을 수행한다. 다수의 Thread를 이용하기 때문에 iothread 방식에서는 장치 에뮬레이션 과정이 병렬로 처리되는것처럼 보인다. 하지만 QEMU의 장치 에뮬레이션 코드는 대부분 Thread Safe하게 작성되어 있지 않기 때문에 Global Mutex를 이용하여 Serialization 되어 있고, 이에 따라 병렬로 처리되지 않는다. 이러한 장치 에뮬레이션의 Serialization은 가상 머신의 I/O 성능을 떨어트리는 주요 원인중 하나이다. vCPU들이 별도의 Thread를 이용하기 때문에 병렬적으로 처리되는 것처럼 보이지만 실제로 vCPU를 에뮬레이션하는 TCG의 Architecture 때문에 vCPU의 병렬 처리율이 매우 낮다. 따라서 iothread만 이용하는 방식 또한 가상 머신의 빠른 성능을 얻기에는 힘든 구조이다.

QEMU with iothread and KVM: 그림 17처럼 iothread 방식에서 TCG대신 KVM을 이용하여 vCPU를 구동하는

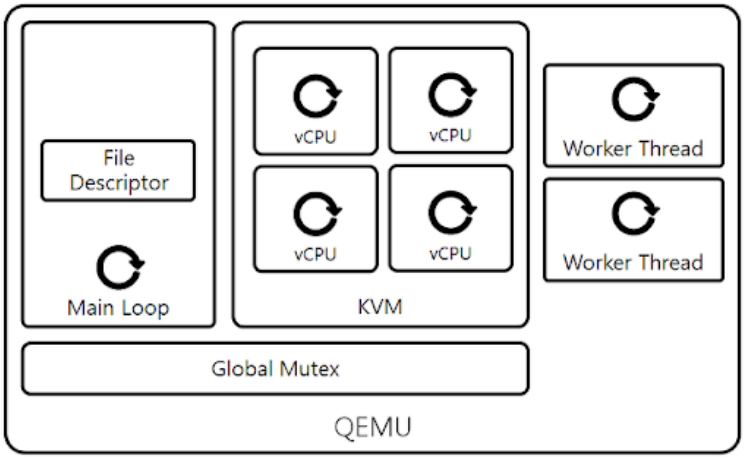


Figure 17: QEMU with iothread and KVM

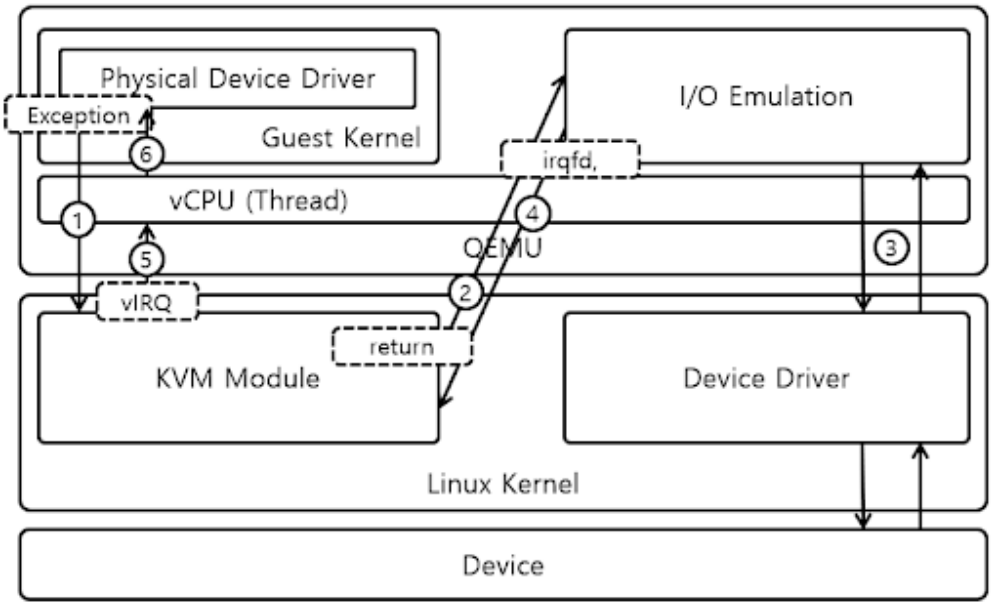


Figure 18: QEMU-KVM Full Device Emulation

방법이다. 장치 에뮬레이션의 Serialization 문제는 여전히 갖고 있지만, KVM에 의해서 각 vCPU는 병렬로 처리된다. 이러한 이유 때문에 가상 머신을 제대로 지원하기 위해서는 QEMU-KVM을 이용해야 한다.

QEMU-KVM IO virtualization (Full Device Emulation)

그림 18은 QEMU-KVM에서 full device emulation 과정을 보여준다. 게스트 운영체제가 자신의 디바이스 드라이버를 수행하다가 I/O 수행을 요청하면 Exception이 발생하여 KVM module로 trap된다. KVM은 QEMU를 스케쥴링하며, QEMU에게 Exception이 발생한 이유를 전달한다. QEMU는 I/O emulation을 하여 호스트 운영체제인 리눅스 디바이스 드라이버에 I/O 요청을 한다. 처리가 끝나면 QEMU는 완료를 전달받고, KVM에게 irqfd를 통해서 virtual interrupt injection을 요청한다. 가상머신은 virtual interrupt injection을 통해 I/O 요청이 끝났다고 인식한다. 위의 과정은 모든 I/O 마다 QEMU에서 장치 에뮬레이션을 하므로 context switching이 많이 발생하고 주요 오버헤드가 된다.

7.2 Virtio: Towards a De-Facto Standard For Virtual I/O Devices [15]

기존의 QEMU 기반의 디바이스 에뮬레이션 방식의 성능에는 한계가 있다. 성능 향상을 위해 Xen의 split device driver처럼 paravirtualization 기법을 적용하여 Linux의 가상 디바이스 드라이버를 만들면 성능향상을 할 수 있다. 또한 Xen의 split device driver의 단점인 Xen의 종속되는 문제를 해결하기 위해 공통 가상 I/O 메커니즘을 만드는 것 또한 중요하다. Virtio가 이루고자 하는 목표는 먼저 디바이스 드라이버 통합으로 서드 파티 드라이버 없이 리눅스에서 제공하는 API를 드라이버의 ABI와 연결하여 사용하도록 하는 것이다. 둘째로 공통 ABI를 제공하여, virtio의 ring mechanism (virtio_ring)을 이용하게 하는 것이다.

Virtio: a linux-internal abstraction API

Virtio는 기본적으로 Split driver 모델을 취하며, 게스트 운영체제에 front-end 드라이버가, 하이퍼바이저에 back-end 드라이버가 존재한다. KVM-QEMU 구조에서 QEMU는 backend driver 역할을 하며, 게스트 운영체제에 virtio device driver (frontend driver)를 존재한다. Virtio는 QEMU와 virtio device driver를 위한 통신 API 및 Framework로써 동작하는데, 이를 활용해 virtio device driver를 만들 수 있다. Virtio는 Device Configuration API와 Front-end driver와 back-end driver의 Data transport를 위한 API를 제공한다. Virtio에서 제공하는 API는 크게 보면 Configuration API와 드라이버간 통신을 위한 I/O API를 제공한다.

먼저 Configuration API는 아래와 같이 4가지가 존재한다.

1. Reading and writing feature bits
2. Reading and writing the configuration space
3. Reading and writing the status bits
4. Device reset

I/O API는 Virtqueue라는 추상계층을 사용해서 드라이버간 데이터 교환을 할 수 있다.

```
struct virtqueue_ops {
    int (*add_buf)(struct virtqueue *vq,
                   struct scatterlist sg[],
                   unsigned int out_num,
                   unsigned int in_num,
                   void *data);
    void (*kick)(struct virtqueue *vq);
    void *(*get_buf)(struct virtqueue *vq,
                     unsigned int *len);
    void (*disable_cb)(struct virtqueue *vq);
    bool (*enable_cb)(struct virtqueue *vq);
}
```

`virtqueue_ops`에서 `add_buf`는 queue에 새로운 버퍼를 추가할 때 사용된다. `kick`은 호스트에게 버퍼가 추가 되었음을 알리는데 사용되며, kick 동작을 연기시켜 여러개의 버퍼를 batch로 처리 가능하다. `get_buf`는 used buffer를 얻기위해 사용된다. `disable_cb`는 디바이스에 대한 인터럽트를 끄는 것처럼, virtqueue 작업에 대한 callback

을 끄는 작업이다. 게스트가 버퍼가 언제 사용되었지 몰라도 되는 경우에 사용한다. enable_cb는 callback을 켜는 작업으로, queue에 존재하는 버퍼를 전부 처리하고 다시 callback을 켜는 경우에 사용한다.

Virtio_ring: a transport implementation for virtio

virtio_ring은 virtio와 virqueue의 구현에 사용된 구조이며 3가지 형태로 구성된다.

1. descriptor array: 버퍼 주소, 길이를 모아둔 디스크립터

   ```
   struct vring_desc
   {
       __u64 addr;
       __u32 len;
       __u16 flags;
       __u16 next;
   }
   ```

   addr는 버퍼 주소이며, IPA(intermediate physical address)이다. len은 버퍼의 길이를 의미한다. next는 선택적으로 사용되며, 다음 버퍼 주소를 나타낸다. flag는 2개로 구성되는데, next가 valid값인지 아닌지 나타내는 값이며, buffer가 read-only, write-only인지 나타내는 값이다.

2. available ring: 게스트가 지정하는 ready 버퍼 디스크립터

   ```
   struct vring_avail
   {
       __u16 flags;
       __u16 idx;
       __u16 ring[NUM];
   }
   ```

   idx는 버퍼의 index 이며, flags는 인터럽트 suppression 옵션이다. flags는 추가적인 인터럽트가 필요 없음을 알리기 위해 사용한다. available ring은 descriptor table 배열의 지수(index)값들의 배열한다. available ring이 따로 분리된 이유는 virtqueue가 비동적으로 동작하기 때문이다.

3. used ring: 호스트가 지정하는 used 버퍼 디스크립터

   ```
   struct vring_used_elem
   {
       __u32 id;
       __u32 len;
   }
   struct vring_used
   {
       __u16 flags;
       __u16 idx;
       struct vring_used_elem ring[];
   }
   ```

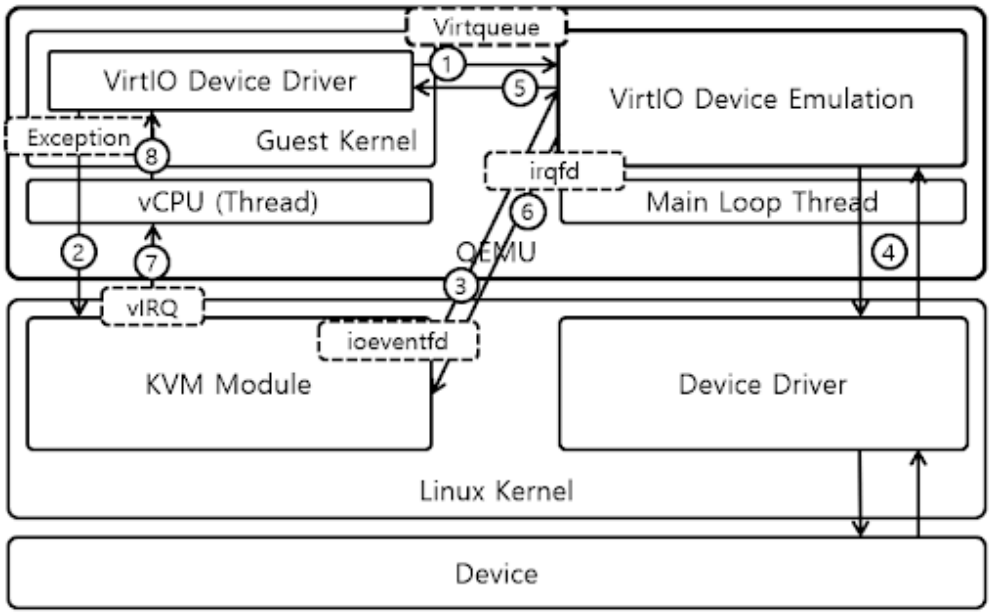


Figure 19: virtio device emulation

available ring과 유사하나 호스트에 의해서 쓰여진다. flags는 호스트가 게스트에게 더이상 버퍼를 추가할 때, kick 과정이 필요없을 때 사용한다.

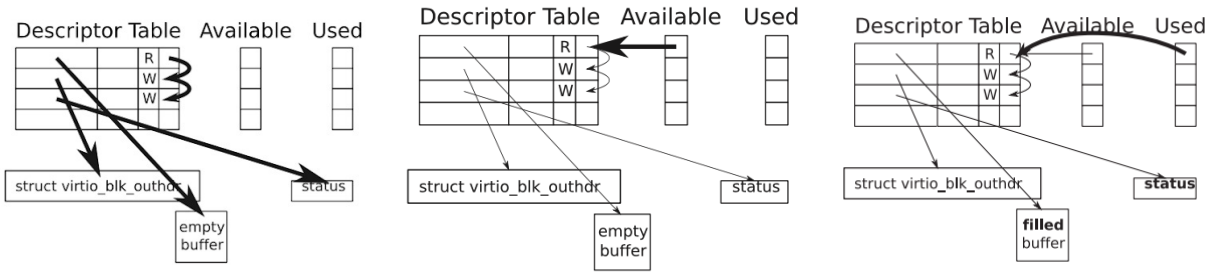
virtio_ring은 추가적으로 게스트 운영체제간의 zero-copy와 page flipping 기법을 활용해서 page 공유 성능을 향상시킬 수 있다.

Current virtio drivers

전체적인 virtio driver의 실행과정은 19과 같다. 먼저, 게스트 운영체제가 virtio device Ddriver인 front-end 드라이버를 수행 중, IO접근을 시도한다. virtio device driver가 virtqueue에 I/O 데이터를 옮기고, kick동작으로 KVM에게 IO 작업이 있음을 알린다. KVM이 QEMU를 스케쥴링해서 QEMU 실행하고, ioeventfd로 virtqueue에 데이터가 있음을 알린다. QEMU는 virtio의 back-end 드라이버로 virtqueue의 데이터를 처리하고, 호스트 운영체제에게 IO 요청을 한다. 실제로 호스트 운영체제의 디바이스 드라이버에 요청이 전달되고 데이터를 처리한다. 데이터 처리가 끝나면 처리 결과를 QEMU가 virtqueue에 쓰고, QEMU는 irqfd를 통해서 KVM에게 virtual interrupt injection을 요청한다. KVM이 virtual interrupt injection을 통해 가상머신에게 IO 완료를 전달한다. 가상머신의 virtio device driver는 가상 인터럽트를 받고 IO 요청이 완료되었다고 인식하고, virtqueue에서 데이터를 받아올 수 있다.

세부 구현 과정은 `virtio_blk_outhdr`를 활용하는데, `virtio_blk_outhdr`은 아래와 같다.

```
struct virtio_blk_outhdr
{
    __u32 type;
    __u32 ioprio;
    __u64 sector;
}
```



(a) virtio request placed into descriptor table (b) virtio block read ready to be serviced (c) virtio block request completed

Figure 20: Virtio descriptor process

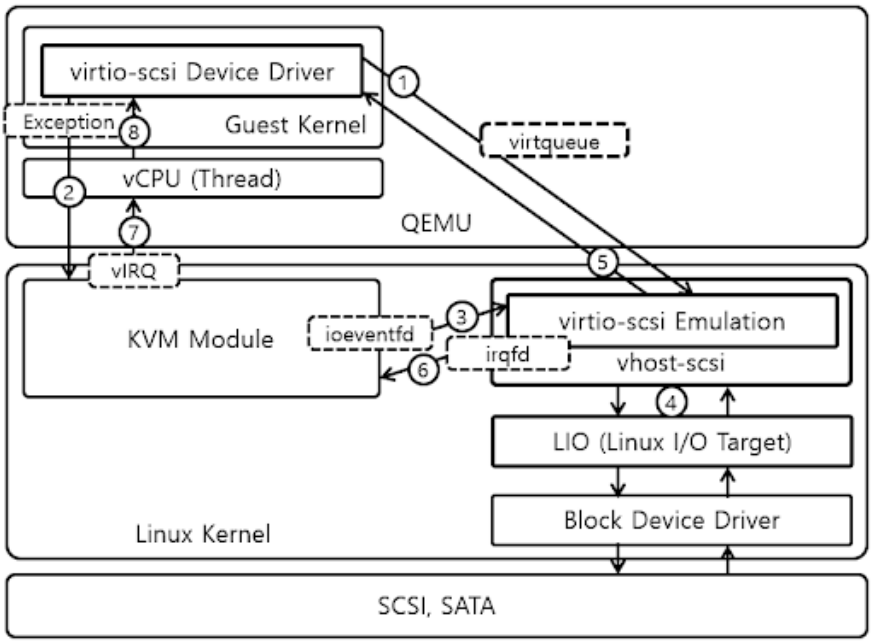


Figure 21: vhost

type은 읽기, 쓰기, generic SCSI 명령어, write barrier인지를 나타낸다. ioprio은 IO 우선순위이다. sector는 읽기 혹은 쓰기에서 512 byte offset을 나타낸다. `virtio_blk_outhdr`는 virtqueue에 들어가는 버퍼 요청의 첫 16 byte이며 metadata 파일을 나타낸다. `virtio_blk_outhdr`는 읽기전용이다. 그림 20는 실제 동작과정을 나타낸다. 디스크립터 테이블에서 `virtio_blk_outhdr` → empty buffer → status 순서로 자료가 연결되어 있다. 여기서 empty buffer, status는 쓰기 전용이다.

Vhost

vhost: 그림 21 처럼 QEMU의 virtio device emulation을 kernel 레벨의 모듈로 가져와서 수행하는 방식이다. QEMU로의 user level context swtiching이 줄어들어서 Context Switching 횟수를 줄일 수 있다. 또한 유저 레벨 어플리케이션인 QEMU가 스케쥴링이 뒤로 밀리는 문제를 방지 할 수 있다. QEMU의 global mutex를 활용해 serialize된 디바이스 에뮬레이션에서는 가상 머신이 동시에 많은 I/O 요청시 성능이 떨어지는 단점이 있는데, vhost는 global

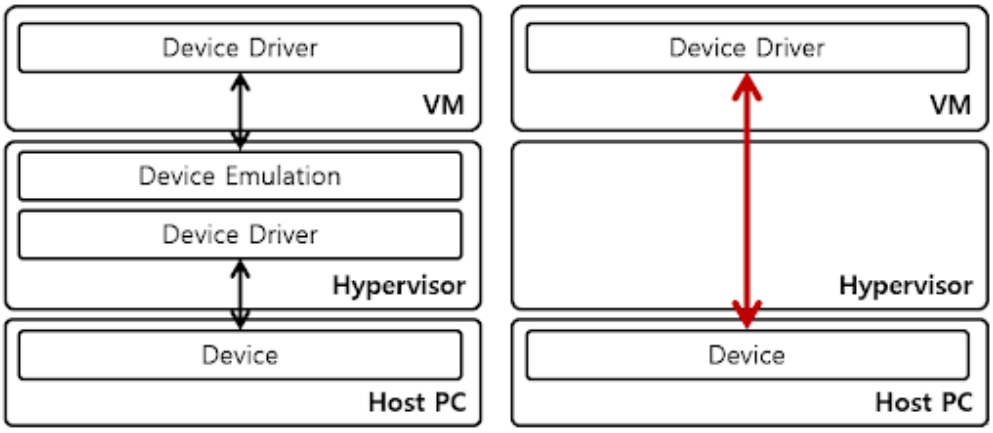


Figure 22: Device direct assignment

mutex를 이용하지 않기 때문에, 성능 향상이 있다. QEMU의 역할인 디바이스 에뮬레이션을 vhost-scsi + LIO를 통해 수행한다.

Vhost-user: 모든 드라이버가 kernel 영역에서 구동될 수 있는 건 아니기 때문에 user 영역에서 구동가능한 vhost 기법이 vhost-user이다. vhost-scsi 같은 backend 드라이버를 유저 영역로 올려서 사용한다. 드라이버는 QEMU와 별개의 프로세스로 존재한다. User space backend 드라이버가 에뮬레이션을 수행한다.

7.3 vIOMMU: Efficient IOMMU Emulation [2]

virtio, vhost보다 IO 가상화에서 최대 효율을 내기 위해서는 direct device assignment를 하는 것이 가장 좋다. Direct device assignment란 그림 22처럼 가상머신이 바로 디바이스를 이용하는 방식을 말한다. 이 경우 하이퍼바이저의 간섭이 전혀 없기 때문에, 가상머신에서 거의 bare-metal 시스템과 비슷한 IO 성능을 낼 수 있다. 하지만 Direct device assignment를 활용하기 위해서는 여러가지 제약사항이 있다.

먼저, 특정한 메모리 물리주소에 게스트 운영체제를 고정해야 되는 문제가 있다. 이는 IO 장치들이 DMA 컨트롤러를 통해 데이터를 교환하기 때문에 발생한다. DMA 컨트롤러는 물리 주소를 기준으로 프로그래밍 되는데, 가상머신은 DMA 요청시 자신이 물리주소라고 믿는 곳에 DMA 요청을 하기 때문이다. 이 주소는 실제 물리주소가 아니기 때문에 DMA page miss라는 심각한 문제가 발생한다. 이를 방지하기 위해서는 게스트 운영체제의 페이지를 특정한 물리 주소에 고정해야 된다.

둘째로, 가상머신이 IOMMU를 제대로 활용하지 못하는 문제가 있다. Bare-metal 시스템에서는 디바이스 드라이버의 버그를 방지하기 위해서, IOMMU는 IOVA (IO 가상 주소)를 PA (Physical Address)를 변경하는 방식으로 디바이스 드라이버가 특정 물리주소영역 이외를 사용하지 못하게 한다. 하이퍼바이저 시스템에서는 게스트 운영체제가 IOMMU를 직접적으로 맵핑하지 못하게 때문에, 이 과정에서 하이퍼바이저의 간섭이 필수적으로 발생한다. 이를 방지하기 위해서 paravirtualization 기법을 활용해서 IOMMU를 활용해야 하는데, 이는 full virtualization이 요구되는 상황에서는 사용되지 못한다.

셋째로, IOMMU를 다른 방식으로 활용하는 이점을 활용하지 못하는 문제가 있다. 32bit 주소를 사용하는 legacy 디바이스에서는 IOMMU를 통해서 64bit 물리 주소 공간에 맵핑을 할 수 있는데, 하이퍼바이저에서는 이를 활용하지 못한다.

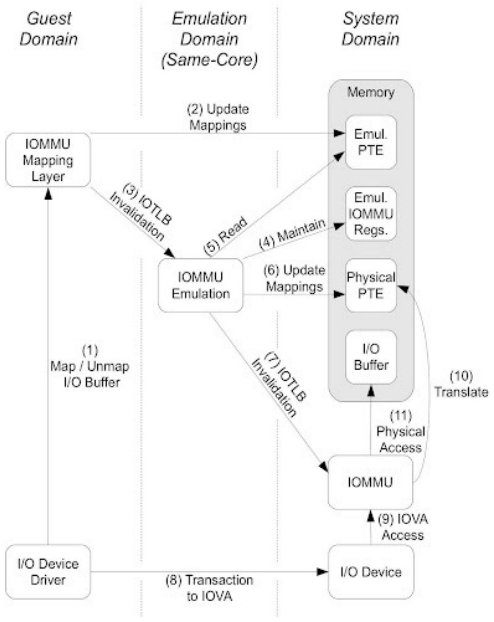


Figure 23: Samecore IOMMU emulation

결국 하이퍼바이저 상에서 IOMMU는 2가지 선택을 할 수 있다. 소프트웨어적인 IOMMU 에뮬레이션인 논문에서 제시하는 vIOMMU과 IOMMU가 하드웨어적 2단계 번역을 지원하는 방식이다. vIOMMU는 가상머신이 각자 IOMMU를 가진것 처럼 에뮬레이션해주는 방식이다. 2단계 번역을 지원하는 IOMMU는 IOVA를 가상머신의 물리주소로 변경하고 이를 최종적인 물리주소로 변경하여 문제를 해결한다. 단, 2단계 번역이 지원되는 하드웨어는 비싸다.

Samecore IOMMU Emulation

IO 장치 에뮬레이션은 기본적으로 디바이스 레지스터에 대한 접근을 trap후 에뮬레이션하는 방식으로 구현된다. 에뮬레이션된 IOMMU의 레지스터가 맵핑된 페이지를 하이퍼바이저가 not present로 설정해 두면, 게스트 운영체제가 IOMMU 변경작업을 할때 하이퍼바이저로의 trap이 발생하게 된다.

그림 23을 참조해서 실제 1개의 DMA transaction 요청 과정을 보면 다음과 같다. 가상머신의 디바이스 드라이버가 IO buffer를 물리 메모리에 맵핑하기 위해서 IOMMU mapping layer를 부른다. (1) IOMMU mapping layer는 IOVA 영역을 Guest 메모리 영역에 할당하고, 게스트 운영체제의 Page Table Entry(PTE)를 맵핑한다. (2) PTE가 새롭게 할당혹은 변경되었기 때문에, 게스트 운영체제는 IOTLB (I/O Translation Lookaside Buffer)를 invalidation 처리를 한다. (3) 해당과정은 IOMMU register에 쓰기 작업을 발생시키기 때문에, 하이퍼바이저로의 trap이 발생한다. 하이퍼바이저는 에뮬레이션된 IOMMU에 레지스터 상태를 업데이트 한다. (4) 이때, 실제 IOMMU 레지스터에 쓰는 것이 아닌 에뮬레이션된 IOMMU가 존재하는 메모리 영역에 쓴다. 하이퍼바이저는 IOMMU 에뮬레이션 과정에서 2과정에서 새롭게 할당한 게스트 운영체제의 PTE 정보를 읽어온다. (5) 해당 PTE 정보를 토대로 실제 물리 메모리 페이지를 pinning하고 IOVA to GPA(Guest Physical Address)를 IOVA to HPA(Host Physical Address)로 변환하여 실제 IOMMU에 업데이트 한다. (6) 이전에 맵핑이 없었던 경우 등 필요한 경우, 실제 IOMMU도 invalidation 과정을 실행한다. (7) 해당 과정후 IO 버퍼가 실제 물리 메모리 상에 맵핑이 된다. 3번 과정완료 이후의 게스트 운영체제로 돌아와서 IO 장치 드라이버가 IOVA를 토대로 DMA 요청을 시작한다. (8) 디바이스 DMA는 전달받는 IOVA를

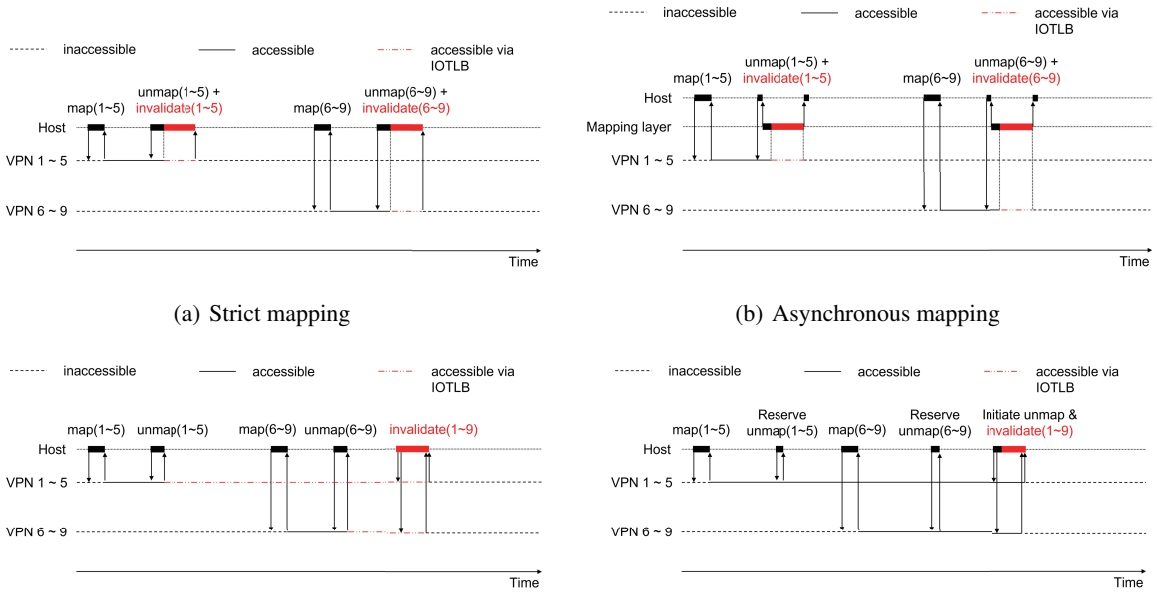


Figure 24: vIOMMU mapping strategy

토대로 메모리 접근을 시도하고, IOMMU에 의해서 IOVA는 HPA로 변경되어 실제 메모리 주소를 접근할 수 있게 된다. (9)

위의 과정을 통해 IOVA는 게스트 운영체제의 특정범위만 접근을 강제하여 안정성을 보장할 수 있다. 또한 모든 가상머신의 페이지가 아닌 특정 페이지만을 물리주소에 고정시킬 수 있다. 모든 가상머신에 에뮬레이션된 IOMMU를 제공함으로써, 각자 가상머신이 IOMMU의 추가적인 이점을 얻을 수 있다.

Optimizing IOMMU mapping

운영체제는 IOMMU 맵핑을 만들고 파괴하는 과정에서 다양한 맵핑 전략을 취할 수 있다. 이때, 성능과 메모리 사용량과 보호정도를 트레이드오프로 고려한다. 리눅스의 기본 모드는 IOMMU의 보호기능을 어느정도 완화해서 사용하고 있다. 이는 IOTLB invalidation을 매 10ms마다 batch 처리하기 것을 말하고, 이를 통해 성능향상을 취할 수 있다. 성능향상이 발생하는 원인은 IOTLB invalidation이 매우 비싼 작업이기 때문이다. 성능향상에 비해 10ms 시간 동안, 디바이스가 invalidation되지 않은 DMA 트랜잭션을 취할 수 있기 때문에 보호 기능이 완화된다. 해당 모드를 논문에서는 relaxed setting이라고 부르며, 그림 24(a)와 같이 IOTLB invalidation을 매번 취하는 것을 strict setting이라고 부른다.

Approximate Shared Mappings Shared mapping은 동일한 물리 페이지를 가리키는 다른 유효한 맵핑 자료를 같이 사용하는 방법이다. 해당 과정을 통하면, 새로운 맵핑을 만드는 setup과 teardown과정이 필요하지 않아서 성능향상을 할 수 있다. Willmann, Rixner, Cox의 논문 [18]에서는 모든 IOMMU 맵핑에 물리주소를 IOVA로 변환하는 역 변환 자료구조를 추가해서 shared mapping을 활용하는 방식을 제시했다. 하지만 최신 IOMMU의 경우는 각 IO 장치마다 IOVA를 두고 모든 물리주소를 맵핑할 수 있기 때문에, 역 변환 자료구조를 사용하는 것은 메모리 사용량이 너무 높아지게 된다. 메모리 사용량이 적지만 복잡한 자료구조인 Red-black tree를 사용하면 반대로 오버헤드가 너무 높아진다. Red-black tree 같은 복잡한 자료구조를 활용하기 위해서 저자들은 approximate shared mapping을 제시한다. Approximate shared mapping은 정확한 역 자료구조를 가지고 있지 않고, 휴리스틱 기법을 적용한

다. 이는 실제로 물리주소를 맵핑하는 정보를 가지고 있어도 발견하지 못할 수 있는 탐색 기법이다. 실제 구현은 소프트웨어 LRU 캐시를 사용하는데, IO 버퍼 할당에 대해 temporal locality, spatial locality를 활용하는 방식으로 구현한다.

Asynchronous Invalidation IOTLB invalidation은 bare-metal 시스템의 IOMMU unmap 처리과정의 40%의 시간을 차지한다. IOTLB를 batch처리하는 경우나 바로바로 처리하는 경우 모두, invalidation 과정을 IOMMU의 invalidation 레지스터나 invalidation 큐에 요청하는 순간 부터 요청한 thread는 IOMMU가 실제로 invalidation이 처리되기 전까지 block된다. 이를 synchronous invalidation이라고 부른다. Asynchronous invalidation은 IOMMU의 보호기능을 어느정도 희생해서, 성능향상을 취한다. 운영체제는 그림 24(b)와 같이 invalidation 요청한 이후 block되지 않고, 계속 진행이 가능하다. 정확한 처리를 위해서 invalidation을 요청한 IOVA 영역에 대해서는 invalidation이 완료될때까지 새로운 맵핑을 시도할 수 없다. 보호기능이 희생되는 순간은 몇 백 cycle정도 이다. 다만, 리눅스의 페이지 할당시 invalidation 과정인 IOVA에 대한 물리영역 재사용을 막을 방법이 없다. 실험상으로는 asynchronous invalidation 시, 동시에 2개이상의 invalidation이 대기하는 경우는 없다.

Deferred Invalidation 현재 리눅스에서 사용하는 IOTLB invalidation을 batch 처리하는 방식을 말한다. 그림 24(c)에서 처럼 10ms 동안 최대 250 invalidation을 합쳐서 요청할 수 있다. Asynchronous invalidation에 비해서 보호기능이 희생되는 기간이 더 길다. (10ms) 에뮬레이션된 하드웨어서는 성능향상이 뛰어난 편이다.

Optimistic Teardown IOVA 번역을 재사용하는 것이 IOMMU 성능에 결정적이다. 그 이유는 IOVA 번역을 재사용시, IOMMU의 페이지 테이블에서 맵핑 정보를 파괴하는 과정과 IOTLB를 invalidation하는 과정과 페이지 테이블을 재구성하는 과정과 IOTLB를 재삽입하는 과정이 필요없어지기 때문이다. Shared mapping을 사용하거나 Deferred invalidation을 사용하더라도, IOMMU 페이지 테이블의 맵핑은 unmap 요청시 즉시 삭제 되기 때문에, IOVA 번역을 재사용하는 것은 상당히 힘들다. Optimistic Teardown에서는 그림 24(d)처럼 IOTLB가 일정시간이후 삭제되는 것을 고려해서, IOMMU 페이지 테이블의 맵핑도 일정시간 이후 삭제되게 만든다. 이는 IOTLB invalidation을 대기하는 시간동안 IOVA 번역이 재사용될 수 있다는 기대에서 출발한다. 이때 보호기능이 희생되는 기간은 Deferred invalidation의 경우와 동일하다. unmap 요청시 해당 맵핑을 FIFO 큐에 넣어서 관리하다가 일정시간이 지나면 큐에서 빠져나오는 맵핑을 실제로 unmap 처리를 한다.

Sidecore IOMMU Emulation

디바이스 에뮬레이션을 다른 core에서 실행하는 방식 이때, 디바이스 에뮬레이션과 게스트 운영체제는 동시에 스케쥴링 되어야 성능이 좋다. VM-exit 혹은 전환과정을 피할 수 있어서 성능향상이 좋다. 기본적인 IOMMU 에뮬레이션 과정은 samecore IOMMU 에뮬레이션과 동일하다. 차이점으로는 먼저, 게스트 운영체제가 장치 레지스터를 접근할 때 발생하는 하이퍼바이저 trap과정의 비용이 싸진다. 이는, 하이퍼바이저가 게스트 운영체제와 장치 레지스터가 할당된 메모리 영역을 공유하며 하이퍼바이저가 갱신시 polling기법을 활용해 컨트롤 레지스터를 에뮬레이션하기 때문이다. 둘째 차이로, 하이퍼바이저와 게스트 운영체제가 다른 코어에서 구동되므로 cache pollution 현상이 적어진다. 하이퍼바이저가 polling 기법으로 레지스터 접근을 확인하므로 IO장치는 4가지 요구사항이 존재한다.

Synchronous Register Write Protocol: 디바이스 드라이버와 디바이스 레지스터간의 쓰기 작업이 동기화 작업이여야 한다. 디바이스 드라이버가 디바이스 레지스터에 쓰기 작업을 하고 완료 확인 등을 받아야 덮어쓰기를 방지할 수 있다.

Single Register Holds Only Read-Only or Write Fields: sidecore 하이퍼바이저에서는 게스트 운영체제의 디바이스 드라이버가 read-only field를 변경했는지 확신할 수 없기 때문이다.

Loose Response Time Requirements: 디바이스 드라이버에 특정 디바이스에만 적용되는 시간 가정등을 사용하게 되면 디바이스 에뮬레이션에서는 제대로 된 사용이 불가능하다. 디바이스 에뮬레이션은 일반적으로 하드웨어 사용보다는 느리기 때문이다.

Explicit Update of Memory-Resident Data Structures: sidecore 하이퍼바이저에 모든 메모리 영역을 전부 polling 하는 것은 비효율적이기 때문에, 특정 메모리 영역에 고정된 자료구조를 명시적으로 갱신하는 과정이 필요하다.

추가적으로 control 레지스터가 적을 수록 sidecore 에뮬레이션 성능이 올라간다. Intel IOMMU는 위 4가지 요구사항을 만족한다. 반면 AMD IOMMU는 운영체제의 mapping layer에서 특정 메모리 영역에 고정된 자료구조를 명시적으로 갱신하는 것을 요구하지 않기 때문에 4번특성을 만족하지 못한다.

Reasoning about risk and protection

위의 기법들에서 IOMMU의 보안기능을 완화하는 이유은 IO 디바이스 드라이버의 unmap function call 완료(logical unmapping)를 실제 물리 IOMMU의 페이지 테이블과 IOTLB의 invalidation을 완료(physical unmapping) 전에 수행하기 때문이다. 즉 logical unmapping이 진행되었지만 실제로 physical unmapping이 수행되지 않은 페이지가 존재하기 때문에 보안기능이 완화되는 것이다. 구체적으로 다른 게스트 운영체제간의 보호인 inter-guest protection의 관점과 특정 게스트 운영체제 내부의 보호인 intra-guest protection의 관점으로 다시 분석해보자. Physical unmapping이 유지되는 동안에는 페이지가 특정 물리주소에 고정되기 때문에, 다른 가상머신은 특정 가상머신의 물리주소에 접근할 수 없다. 이를 통해 어떤 IOMMU 맵핑 기법을 사용하던 간에 inter-guest protection을 유지할 수 있다. 반면 리눅스의 디바이스 드라이버는 logically unmapping이 된 경우와 physically unmapping이 된 경우를 동일하게 두기 때문에, logically unmapping이 된 페이지를 드라이버나 IO stack에서 다른 용도로 사용할 수 있다. 이 때문에 보호기능이 완화된 기법을 사용할 경우 intra-guest protection을 보장할 수는 없다.

그렇다면 실제 보호기능이 완화되었을때 위험정도는 얼마나 되는지 알아보자. 위험정도를 측정하기 위해서 window of vulnerability와 stale mapping bound에 집중해보자. window of vulnerability란 DMA가 logically unmapping이 되었지만 physically unmapping 되지 않은 stale mapping으로 IO buffer에 읽기 또는 쓰기가 가능해지는 시간단위를 말한다. stale mapping bound란 특정시간에서 최대 stale mapping의 숫자를 말한다. IOMMU 맵핑 기법에 따른 위험도를 분류하면 아래와 같다.

No Risk: Bare metal 시스템에서 운영체제가 strict mapping을 이용하는 경우나 가상머신 시스템에서 게스트 운영체제와 호스트 운영체제가 모두 strict mapping을 사용하는 경우이다. Bare metal, Samecore 에뮬레이션, Sidecore 에뮬레이션에 상관없이 strict mapping을 사용하는 경우는 보호가 완화 되지 않기 때문에 위험도가 없다. shared mapping의 사용 유무는 위험도와 상관 없다.

Nanosecond Risk: 호스트 운영체제가 실제로 invalidation request를 요청하고 실제 물리 IOTLB에 invalidation이 적용되는 시간이 ns단위이다. 해당시간 동안 보호가 되지 않는 순간은 Bare metal 시스템에서 asynchronous invalidation을 사용할 때가 유일하다. Samecore, Sidecore 에뮬레이션에서는 게스트와 호스트간의 통신과정이 해당 시간을 초과하기 때문에, 이 시간에는 위험도가 발생되지 않는다. 이 경우 stale mapping bound는 2개의 맵핑 정도이고, window of vulnerability는 page table entry당 평균 128 사이클 정도이다.

Microsecond Risk: Sidecore 에뮬레이션에서 게스트 운영체제가 Asynchronous invalidation을 사용하는 경우에 발생할 수 있는 위험도이다. window of vulnerability는 코어간 통신비용에 의해 결정된다. stale mapping bound는 실험상으로 128 entry정도이다.

Millisecond Risk: 모든 시스템에서 Deferred invalidation이나 optimistic teardown 기법을 사용하는 경우에 발생할 수 있는 위험도이다.

7.4 Platform Device Assignment to KVM-on-ARM Virtual Machines via VFIO [13]

네트워크 인터페이스나 스토리지 같은 IO 장치를 가상화하는 경우, user space에 대한 인터페이스가 존재하기 때문에 가상머신을 설정할 수 있는 유저 어플리케이션에서 에뮬레이션하는 방식으로 가상화를 진행한다. KVM의 경우, 해당 어플리케이션을 주로 QEMU를 사용하여 디바이스 에뮬레이션을 한다. QEMU는 KVM에서 제공하는 인터페이스를 통해 게스트 운영체제가 특정 메모리 영역을 읽거나 쓰는 경우 trap이 발생하도록 설정할 수 있다. 해당 trap이 발생하는 경우, QEMU가 실행되며 IO 장치 에뮬레이션을 실행한다. QEMU에서 인터럽트 컨트롤러를 에뮬레이션하는 방식으로 IO 장치 인터럽트(virtual interrupt injection)를 처리하는 경우도 있으나, ARM이나 x86 모두 인터럽트를 가상화하는 하드웨어를 제공하기 때문에, 이를 KVM에서 활용해서 장치 인터럽트를 처리하는 것이 일반적이다.

QEMU Device Emulation: QEMU-KVM에서 디바이스 에뮬레이션은 KVM의 기능을 유저 스페이스에서 활용할 수 있는 ioctl 시스템 콜 인터페이스를 활용해서 이루어진다. QEMU는 ioctl을 통해 KVM을 설정하고, 게스트 context로 진입가능하다. 게스트가 trap 이나 exit 된경우, QEMU는 KVM을 통해 게스트의 상황을 알고 에뮬레이션을 진행할 수 있다. 문제는 QEMU를 활용한 디바이스 에뮬레이션은 MMIO(Memory Mapped IO) 메모리 영역에 대한 접근이 많아서 KVM과 QEMU간의 전환이 많기 때문에 성능이 많이 느리다.

QEMU Device Paravirtualization(virtio, vhost): 위의 문제를 해결하기 위해서, QEMU에 paravirtualization 기법인 virtio를 적용했다. IO 장치의 에뮬레이션이 전부 QEMU에 의해서 발생하는 것이 아닌, QEMU와 게스트 운영체제간의 통신 인터페이스를 통해 이루어진다. 게스트 운영체제에 paravirtualization된 virtio driver (front-end driver)가 존재하고, QEMU는 back-end driver가 존재하여 디바이스 IO처리를 한다. virtio 장치 드라이버는 QEMU의 back-end driver와 바로 정보 교환을 할 수 있어 KVM의 MMIO 접근 횟수를 줄일 수 있고, 이를 통해 KVM과 QEMU간의 전환을 줄여서 성능향상을 취할 수 있다. 해당 방식의 문제는 게스트 운영체제에 미리 virtio driver를 구현해 두어야해서, 게스트 운영체제의 수정이 필요하다는 점이다. 어떤 full virtualization이 필요한 환경에서는 사용이 어렵다.

Device Assignment

게스트 운영체제가 특정 디바이스를 직접적으로 사용할 수 있는 것을 허용하면, 하이퍼바이저의 개입이 필요없기 때문에 native에 가까운 성능을 낼 수 있다. 해당 방식을 Device Assignment 방식이라고 부른다. 또한 게스트 운영체제가 native driver를 사용할 수 있어서 운영체제 수정도 필요없다. 단, 가상머신 환경에서는 IOMMU (IO Memory Management Unit)가 필수적으로 요구된다. 현재의 모든 hardware accelerator나 IO 장치들은 Direct Memory Access (DMA) 컨트롤러를 가지고 있다. DMA 컨트롤러는 기본적으로는 물리주소 (메모리주소)를 기반으로 설정된다. 이 때, 특정 디바이스가 bare-metal 환경에서 유저 어플리케이션에 할당되거나 또는 가상머신 환경에서 특정 가상머신에 할당된 경우가 문제가 된다. 이는 유저 어플리케이션이 보는 가상주소 IOVA(IO Virtual Address)나 가상머신이 보는 IPA (Intermediate Physical Address)가 실제 물리주소로 믿고 있기 때문에 DMA 요청을 해당 주소로 요청하기

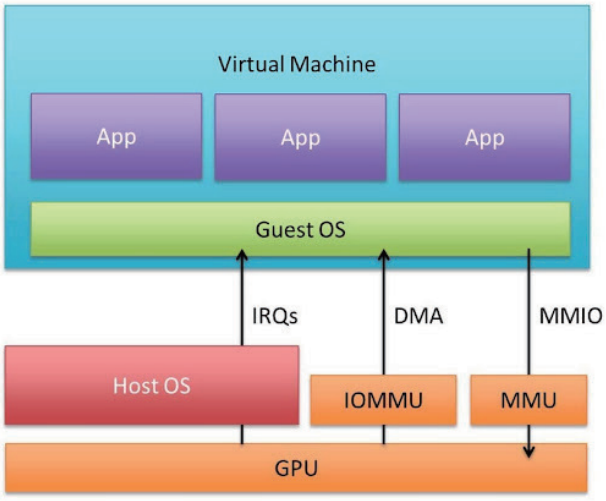


Figure 25: Device Assignment with IOMMU

때문에 발생한다. 또한 IOVA, IPA 모두 실제로 DMA 컨트롤러가 물리주소로 믿고 해당 주소로 작업을 실행할 경우, 다른 주소영역을 침범하여 작업하므로 보안이슈가 발생한다. IOMMU는 해당 과정을 방지하기 위해서 MMU (Memory Management Unit)처럼 디바이스가 볼 수 있는 메모리 영역을 제한한다. IOMMU는 virtual address를 physical address로 변환하는 page table을 가지고 있는데, 실제 구현은 하드웨어의 스펙마다 다르다. IOMMU는 디바이스가 접근하는 주소를 물리주소로 변경하여 DMA 요청이 원하는 공간 및 제한된 공간에서 사용할 수 있게 한다. 위의 과정을 요약하면 그림 25와 같다.

Device assignment에서 특정 디바이스는 다른 디바이스와 무조건적으로 같이 사용해야 되는 경우가 있는데, 이 때는 해당 디바이스들은 같은 가상머신에 할당해야 한다. 이를 위해서 리눅스의 IOMMU API는 IOMMU group이라는 것을 둔다. IOMMU group은 독립적으로 존재할 수 없는 디바이스 집합을 나타내고, 무조건 같은 가상머신에 할당되어야 한다.

Virtual Function IO

Virtual Function IO (VFIO)는 본래 kernel 디바이스 드라이버를 사용하지 않고 user space에서 디바이스를 직접적으로 이용할 수 있는 방식이다. 즉, MMIO와 DMA 영역을 유저 프로세스에 안정적으로 맵핑하는 역할을 한다. 가상화 환경에서 VFIO는 device assignment를 리눅스상에서 할 수 있는 모듈 역할을 한다. 따라서 QEMU와 같은 유저 프로그램는 그림 26처럼 VFIO API를 사용해서 QEMU 가상 디바이스 영역을 바로 실제 디바이스에 연결함으로써 가상머신이 디바이스를 직접적으로 사용하게 할 수 있다. 이를 virtual pass-through라고 부른다. 이때 호스트 운영체제의 커널 디바이스 드라이버가 사용되지 않으므로 성능향상을 여전히 취할 수 있고, 게스트 운영체제의 디바이스 드라이버도 수정할 필요가 없다. 인터럽트는 QEMU인 유저 프로세스로 전달되므로, 인터럽트 핸들러는 하이퍼바이저가 관여할 수 있다. 가상화 환경에서 VFIO를 제대로 사용하기 위해서는 IOMMU의 기능이 꼭 필요하다. IOMMU group이 동일한 가상머신에 할당되어야하는 디바이스 그룹을 나타낸다면, VFIO에서는 상위 개념인 container가 존재한다. Container는 동일한 configuration을 공유하는 디바이스 그룹과 디바이스를 나타낸다.

VFIO 핵심 모듈은 core VFIO module, VFIO bus driver, VFIO IOMMU driver가 존재한다. core VFIO module은 유저공간에 대한 인터페이스와 VFIO group, container 관리를 담당하며, 아키텍쳐에 독립적이다. VFIO bus driver은 VFIO를 통해 디바이스를 메모리 주소에 맵핑하고 등록하는 일을 담당하고, MMIO 맵핑과 디바이스 인터럽

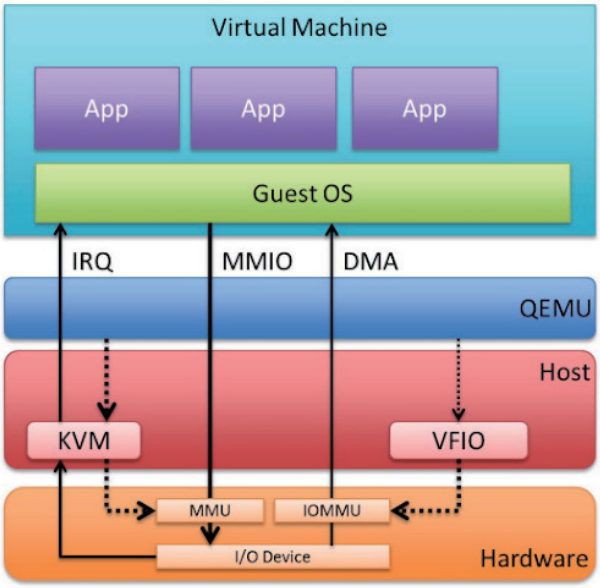


Figure 26: VFIO

트 포워딩 역할도 하고 있다. 일반적으로 VFIO를 이용해 특정 디바이스를 가상머신에 할당하기 위해서는 호스트 커널에 할당되어있는 디바이스의 IOMMU group을 비할당으로 바꾸는 것이 우선된다. VFIO IOMMU driver은 IOMMU를 활용해 DMA 맵핑을 담당한다. 이러한 VFIO를 활용하기 위한 3개의 중요한 ioctl 시스템콜이 있다. 먼저, VFIO_SET_CONTAINER은 Container를 설정하기 위한 시스템 콜이다. 둘째로, VFIO_SET_IOMMU은 DMA 메모리 영역을 준비하는 시스템 콜이다. 마지막으로, VFIO_IOMMU_MAP_DMA IOMMU에 DMA 맵핑을 시키는 시스템 콜이다. 메모리영역과 인터럽트 라인은 ioctl 시스템콜을 통해 VFIO device file descriptor로 맵핑될 수 있다.

VFIO on ARM

ARM 플랫폼은 주로 PCI interconnect을 사용하지 않고, on-chip interconnect나 network on chip을 사용하는 특징이 있다. on-chip interconnect는 PCI와 같은 auto configuration 인터페이스가 없기 때문에, 메모리 영역이나 할당된 인터럽트등 유효한 하드웨어 구성요소나 설정요소를 커널로 전달하는 다른 방법이 필요하다. 기존에 사용하던 방식은 2가지이다. 먼저, compiled device tree방법으로 타켓 시스템에 대한 정보가 들어있는 자료를 부팅시 리눅스 커널에 정보를 전달하는 방식이다. 둘째로, 커널에 직접적으로 해당 정보를 하드코딩하는 방법이 있다. 논문에서는 새로운 해결책으로 VFIO platform driver를 제시한다. ARM 아키텍쳐에서는 VFIO Platform bus driver와 ARM IOMMU 지원이 추가됐다.

VFIO Platform Bus Driver은 ARM과 같은 플랫폼 디바이스를 위한 bus driver로써, 플랫폼 디바이스의 정보를 커널에서 알 수 있게 제공한다. Bus driver는 MMIO 맵핑과 인터럽트 포워딩을 수행하기 때문에, 디바이스 아키텍쳐에 따른 특성을 알고 있어야 한다. 디바이스 특성을 알기 위한 몇가지 ioctl 시스템콜을 추가하여 이를 해결했다. 먼저, 디바이스에 대한 정보를 제공하는 VFIO_DEVICE_GET_INFO 시스템 콜을 추가했다. 둘째로, 디바이스 영역에 대한 정보를 제공하는 VFIO_DEVICE_GET_REGION_INFO 시스템 콜을 추가했다. 셋째로, 디바이스 IRQ 시그널 정보를 제공하는 VFIO_DEVICE_GET_IRQ_INFO 시스템 콜을 추가했다. 마지막으로, 인터럽트 마스킹용도로 디바이스 IRQ 시그널 갯수를 제한하는 VFIO_DEVICE_SET_IRQS 시스템 콜을 추가했다.

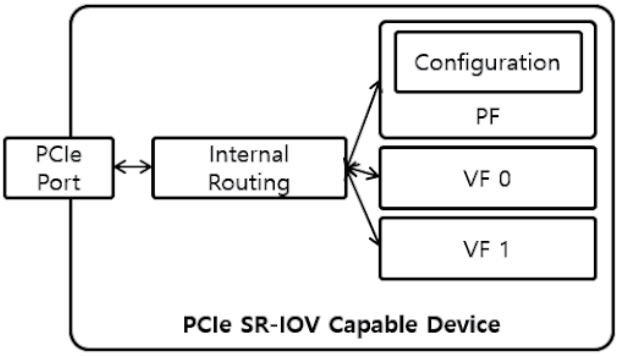


Figure 27: SR-IOV 장치

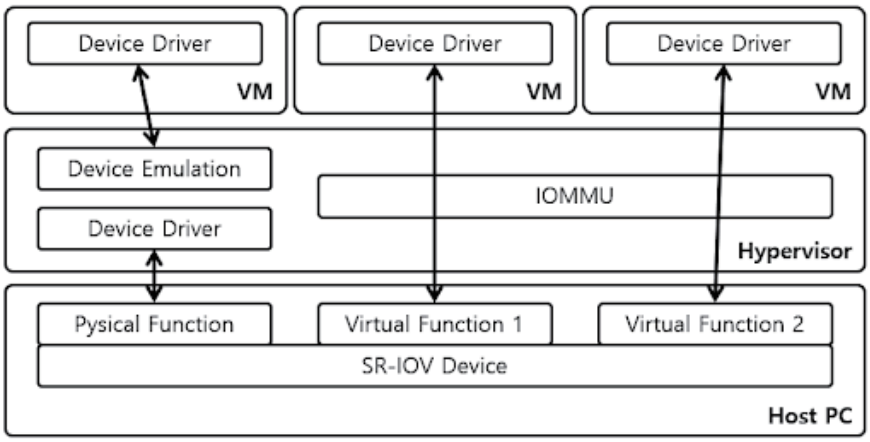


Figure 28: IOMMU + SR-IOV

IOMMU의 구현은 아키텍쳐마다 다른데, ARM의 경우는 IOMMU가 아닌 SMMU(System Memory Management Unit)이라고 불린다. 아키텍쳐마다 다른 구현을 지원하기 위해서 generic IOMMU를 설정하고 VFIO IOMMU 드라이버를 재구성한다. VFIO_TYPE1_IOMMU 드라이버가 generic IOMMU를 담당한다. Generic IOMMU는 IOMMU API를 이용할 때 가상주소를 어떠한 주소공간에 맵핑될 수 있게 한다. 내부적으로는 vfio_dma_map 함수에서 메모리영역을 개별 페이지로 쪼갠뒤, 고정하는 과정이 발생한다. 페이지를 고정하는 것은 DMA 과정중 VFIO가 사용하는 페이지가 스왑되는 것을 방지하기 위해서이다. SMMU가 generic IOMMU API를 지원하도록 하면, ARM 플랫폼에서도 VFIO IOMMU 드라이버를 사용할 수 있다.

Single Root IO Virtualization (SR-IOV)

Single Root IO Virtualization (SR-IOV)는 1개의 IO 장치를 호스트에게 여러개의 IO 장치처럼 보이게 하는 하드웨어 가상화 기법이다. 그림 27은 SR-IOV 디바이스의 예시를 보여준다. 여기서 Physical Function(PF)은 모든 기능을 가진 원본 디바이스를 의미하고, Virtual Function(VF)은 제한된 기능을 가진 디바이스를 의미한다. 이를 통해 호스트 운영체제는 1개의 실제 물리 장치가 여러개 존재하는 것처럼 인식을 할 수 있다. SR-IOV 디바이스에서는 PF의 레지스터를 통해 VF 생성 및 제거 가능하다. 그림 28은 IOMMU와 SR-IOV 디바이스를 활용한 예시를 보여준다. 1개의 SR-IOV 디바이스를 VF를 통해 가상머신에 직접할당 가능하고, 직접 할당된 디바이스는 IOMMU를 통해 가상머신 DMA 요청 가능하다. 이때, 하이퍼바이저는 인터럽트 전달만 처리한다.

8 요약 및 향후 연구

이제까지 하이퍼바이저 및 가상머신 기술에 대한 전반적인 내용을 다루었다. 2장에서는 가상머신의 일반적인 목적과 하이퍼바이저의 필요성에 대해서 다루었다. 하이퍼바이저는 1개의 머신 인터페이스를 여러개의 머신 인터페이스를 가지는 것처럼 구동하는 기능을 핵심으로 한다. 이를 통해 1개의 시스템에게 여러개의 운영체제를 구동할 수 있는 장점을 가지나, 이에 따른 오버헤드가 존재한다. 가상머신은 플랫폼 독립성과 이식성을 위해서 사용될 수도 있는데 이에 대한 대표적인 예시는 JVM (Java Virtual Machine)이다. Co-designed VM을 사용해 이러한 플랫폼 독립성 및 이식성을 요구하는 가상머신의 성능향상을 할 수 있는 방법을 알아보았다. 3장에서는 Classical Virtualization, Software Virtualization (binary translation), Hardware Virtualization에 대해서 알아보았으며, Hardware Virtualization에서는 Intel VT-x에서의 virtual machine control block(VMCB)를 활용한 하드웨어 가상화를 알아봤다. 또한 소프트웨어 가상화 기법인 binary translation과 하드웨어 지원을 받는 trap-and-emulate 방식을 비교하고 하드웨어나 소프트웨어 가상화에 대한 추가적인 기법 제공에 대해서도 알아봤다. 4장에서는 RISC 머신인 ARM에서의 가상화 지원 하드웨어를 알아봤으며, 크게 CPU, 메모리, 인터럽트, 타이머 가상화에 대한 지원을 알아봤다. ARM은 x86과 달리 새로운 CPU privileged level인 hyp mode를 추가하여 CPU 가상화를 한 점이 가장 큰 차이이며, 이는 6장에서 서술된 KVM 구조에 큰 영향을 미치게 되었다. 5장에서는 Type1 하이퍼바이저에대해서 알아보았는데, Disco와 Xen이 있었다. Disco는 본래 목적은 Non-NUMA 운영체제를 NUMA 시스템에 적용하는 것인데, 이를 가상머신 기법을 이용하여 해결하였다. Xen은 Disco와 다르게 하이퍼바이저의 구현을 목표로 하였으며, 이전과 다른 *paravirtualization*을 채용한 하이퍼바이저이다. Xen을 ARM에 이식하는 작업에 대한 내용도 5.3에서 알아보았다. 6장에서는 대표적인 Type2 하이퍼바이저인 KVM에 대해서 알아보았다. KVM은 ARM에서 Split-mode 구조인 Highvisor와 Lowvisor를 가지는 것이 큰 특징으로 하며, 이에 대한 ARM 하드웨어 지원을 받은 KVM 구조에 대해 알아보았다. 또한 ARM에서 구현된 Xen과 ARM에서 구현된 KVM을 비교하여 KVM 구조에 대한 이해를 도왔다. 6.2를 통해서 ARM의 Virtualization Host Extensions(VHE)를 활용한 성능개선 기법 및 하이퍼바이저 재설계를 알아보았다. 7장에서는 QEMU를 활용한 full device emulation, virtio와 vhost같은 paravirtualization 기법, viommu와 vfio를 활용한 direct device assignment 기법에 대해서 알아봤다.

향후 연구 방향으로 IOMMU의 새로운 트렌드인 2단계 번역 기술에 대해서 연구해보자한다. IOMMU 2단계 번역은 기존 IOMMU가 IOVA를 IPA로 변경하거나, IPA를 PA로 변경하는 1단계 번역과정만을 수행하는 것이 아닌 IOMMU자체가 IOVA를 PA로 번역할 수 있는 기술을 말한다. IOMMU가 2단계 번역을 지원하면, viommu같은 소프트웨어적인 기법을 하드웨어적으로 대체할 수 있고 이를 통해 성능향상을 이룰수 있을 것이다. 서버와 같이 컴퓨팅 리소스를 많이 사용하는 환경에 적용되는 하드웨어는 2단계 번역 기술을 활용할 가능성이 높지만, 임베디드 시스템같은 곳은 비용 문제로 해당 하드웨어를 여전히 지원하지 않을 가능성이 높다. 임베디드 시스템에서 주로 사용되는 ARM 아키텍쳐는 IOMMU를 SMMU(System Memory Management Unit)으로 부르며, 우리는 SMMU의 2단계 번역 기술에 대해서 연구 방향을 잡고자 한다. 2단계 번역 기술의 수치적 성능향상을 측정하고, 소프트웨어적으로 해당 성능향상을 이룰 수 있는 기법을 통해 1단계 번역만을 지원하는 하드웨어에 대한 성능향상을 이루는 것을 향후 연구 목표로 한다. 해당 기술을 2단계 번역 하드웨어에 적용시키면 부가적인 성능향상을 취할 수 있을 것으로 또한 기대한다.

References

[1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM Sigplan Notices*, 41(11):2–13, 2006.

[2] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, Assaf Schuster, et al. viommu: efficient iommu emulation. In *USENIX Annual Technical Conference (ATC)*, pages 73–86, 2011.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[4] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. Califor-nia, USA, 2005.

[5] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 15(4):412–447, 1997.

[6] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos. Arm virtualization: performance and architectural implications. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 304–316. IEEE, 2016.

[7] Christoffer Dall, Shih-Wei Li, and Jason Nieh. Optimizing the design and implementation of the linux {ARM} hypervisor. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 221–233, 2017.

[8] Christoffer Dall and Jason Nieh. Kvm/arm: Experiences building the linux arm hypervisor. 2013.

[9] Christoffer Dall and Jason Nieh. Kvm/arm: the design and implementation of the linux arm hypervisor. *Acm Sigplan Notices*, 49(4):333–348, 2014.

[10] Alexandru Elisei and Mihai Carabas. bhyvearm64: Generic interrupt controller version 3 virtualization.

[11] Robert P Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, 1974.

[12] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones. In *2008 5th IEEE Consumer Communications and Networking Conference*, pages 257–261. IEEE, 2008.

[13] Antonios Motakis, Alvise Rigo, and Daniel Raho. Platform device assignment to kvm-on-arm virtual machines via vfio. In *2014 12th IEEE International Conference on Embedded and Ubiquitous Computing*, pages 170–177. IEEE, 2014.

[14] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[15] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[16] James E Smith, Subramanya Sastry, Timothy Heil, and Todd M Bezenek. Achieving high performance via co-designed virtual machines. In *Innovative Architecture for Future Generation High-Performance Processors and Systems*, pages 77–84. IEEE, 1998.

[17] Prashant Varanasi and Gernot Heiser. Hardware-supported virtualization on arm. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, pages 1–5, 2011.

[18] Paul Willmann, Scott Rixner, and Alan L Cox. Protection strategies for direct access to virtualized i/o devices. In *USENIX Annual Technical Conference*, pages 15–28, 2008.