Journal of The Korea Society of Computer and Information
Vol. 22 No. 5, pp. 11-19, May 2017
www.ksci.re.kr
https://doi.org/10.9708/jksci.2017.22.05.011

A Hypervisor for ARM based Embedded Systems

Sunghoon Son*

Abstract

In this paper, we propose a hypervisor for embedded systems based on ARM microprocessor. The proposed hypervisor makes it possible to run several real-time kernels concurrently on a single embedded system by virtualizing its microprocessor. With assistance of MMU, it supports virtual memory which enables each guest operating system has its own address space. Exploiting the fact that most embedded systems use memory-mapped I/O device, it provides a mechanism to distribute an external interrupt to virtual machines properly. It also achieves load balancing through live migration which moves a running virtual machine to other embedded system. Unlike other para-virtualization techniques, minor modifications are needed to run it on the hypervisor. Extensive performance measurement studies are conducted to show that the proposed hypervisor has enough potentiality of its real-world application.

▸Keyword: Hypervisor, embedded system, virtual machine

I. Introduction

가상화는 컴퓨팅 자원에 대해 합병, 분할 또는 시뮬레이션 등의 기법을 적용하여 추상화된 가상의 실행 환경을 제공하는 기술을 일컫는다. 가상화 기술을 적용하면 하나의 물리적 컴퓨팅 환경에서 두 개 이상의 운영체제를 동시에 실행하거나 이종의 컴퓨팅 환경을 에뮬레이션하는 것이 가능해진다. 예를 들면 단일 프로세서의 컴퓨팅 환경에서 프로세서, 디바이스 등의 개별 자원을 시간적으로 분배하거나, 메모리와 같은 공간 자원을 분배하는 기술, x86계열 프로세서 컴퓨팅 환경에서 ARM 구조의 프로세서 환경을 구축하는 에뮬레이션 등이 대표적인 가상화 기술이라고 할 수 있다. 이와 같은 가상화 기술의 이용은 자원의 활용도를 극대화 할 수 있으며 컴퓨팅 환경의 유연성을 향상 시킬 수 있고 보안과 안정성의 측면에서도 많은 이점을 보장할 수 있다.

그간 서버나 PC 등 범용 컴퓨터 시스템을 대상으로 하는 가상화 기법에 대한 연구는 활발하게 이루어져 다양한 가상화 기술들이 발표되었고 이를 기반으로 하는 상용 가상화 제품들도 다수 출시되었다. 반면 임베디드 컴퓨팅 환경에서의 가상화 연구는 상대적으로 활발하지 못한 실정이었는데, 이는 주로 임베디드 시스템이 가지는 하드웨어 성능 상의 제약에서 기인한다.

하지만 임베디드 시스템 전용 프로세서의 성능은 갈수록 진화하고, SoC를 통해 주변 장치를 제어함으로써 부피와 무게가 감소하며, DSP의 포함으로 멀티미디어 처리 같은 작업에 대한 부담을 덜 수 있게 됨으로써 임베디드 시스템에서의 가상화가 가능해지고 있다.

임베디드 시스템에 대한 가상화 기술은 다양한 응용 분야를 가진다. 특히 하나의 하드웨어 상에서 같은 운영체제를 여러 개 실행시키는 서버 가상화와는 달리, 단일 임베디드 하드웨어 플랫폼에 다수의 다른 종류의 운영체제들을 동시에 실행시킴으로써 다양한 이점을 누릴 수 있다 [1]. 예를 들어, 스마트폰에 안드로이드 운영체제와 함께 보안 운영체제를 동시에 실행시킴으로써 일상적인 응용 프로그램과 보안이 필요한 프로그램을 동시에 안전하게 사용할 수 있도록 하거나, 안드로이드와 기존 회사 업무용 레거시 운영체제를 동시에 실행시켜 기존 소프트웨어를 쉽게 재사용하는 경우 등이 전형적인 사례이다. [1]에서도 언급한 바와 같이 두 가지 사례 모두 BYOD (Bring Your Own Device) 개념의 한 형태로, 임베디드 시스템 가상화는 한

• First Author: Sunghoon Son, Corresponding Author: Sunghoon Son
*Sunghoon Son (shson@smu.ac.kr), Dept. of Computer Science, Sangmyung University
• Received: 2017. 03. 29, Revised: 2017. 04. 07, Accepted: 2017. 04. 18.

단말 내에서 개인적인 컴퓨팅 환경과 업무용 컴퓨팅 환경의 논리적 구분을 가능하게 하여 추가의 하드웨어에 소요되는 비용을 절감하는 효과가 있다.

한편 실시간 운영체제를 수행하는 임베디드 시스템의 경우 대부분 특정 이벤트에 대한 처리를 목적으로 하기 때문에 자원의 활용도가 낮은 편이다. 이러한 측면에서 가상화 기술을 통해 한 운영체제가 이벤트에 대해 대기하거나 장치의 입출력 처리 과정에서 발생하는 프로세서나 장치의 유휴 시간을 다른 운영체제의 수행에 분배할 수 있기 때문에 자원의 활용도를 높일 수가 있다. 또한 이 과정에서 임베디드 시스템의 부피, 무게, 전력 소비량을 줄일 수가 있으며, 이러한 효과는 자동차 등 한 시스템 내에 여러 개의 임베디드 시스템을 운용하는 경우에 특히 의미가 있다.

그러나 기존의 범용 시스템을 위한 가상화 기법을 임베디드 시스템에 그대로 적용하는 것은 문제가 있다. 기존의 가상화 솔루션들을 오랜 시간 쌓은 기술력으로 뛰어난 성능과 높은 안정성을 제공하지만, 임베디드 시스템이 대부분 SoC를 사용하기 때문에 주변 장치의 제어 방식이 범용 시스템과 다르다는 점, 제한된 자원을 가진 임베디드 시스템과 달리 데스크톱이나 서버의 풍부한 자원을 가정하고 개발되었다는 점, 범용 가상화 시스템의 스케줄러와 입출력 처리 모델은 실시간성을 보장할 수 없다는 점 등을 고려해볼 때 기존 범용 시스템용 가상화 기법을 임베디드 시스템에 그대로 적용하는 것은 어렵다.

본 논문에서는 임베디드 시스템의 가상화를 지원하는 하이퍼바이저를 제안한다. 제안된 하이퍼바이저는 우선 단일 마이크로프로세서에서 여러 개의 실시간 운영체제의 수행을 지원한다. 이를 위해 하드웨어와 운영체제 사이에 하이퍼바이저 계층을 두고, 이 하이퍼바이저가 물리 자원들을 다수의 논리적인 자원들로 재구성하여 각 게스트 운영체제에 할당함으로써 하나의 물리적인 컴퓨팅 환경에서 다수의 운영체제를 동시에 수행할 수 있도록 한다. 제안된 하이퍼바이저는 마이크로프로세서의 MMU 기능을 사용하여 가상 메모리상에서 게스트 운영체제 간의 메모리 공간 동기화를 보장하며 이를 통해 운영체제 간의 독립성을 보장하도록 하였다. 또한 마이그레이션을 통해 네트워크상의 다른 임베디드 시스템으로 가상 머신의 동적인 이동을 가능하게 함으로써 부하 조절 등이 가능하게 하였다. 또한 기존의 반가상화 기법들과는 달리 게스트 운영체제 커널의 수정 없이 실행이 가능하도록 하였다. 실시간 운영체제가 갖는 특성을 고려하여 인터럽트와 시간에 대해 하이퍼바이저가 일방적으로 제공하는 구조를 갖추어 가상 머신에서 동작하게 될 운영체제의 수정이 불필요하게 하였다.

본 논문의 구성은 다음과 같다. 본 장의 서론에 이어 2장에서는 가상화와 임베디드 컴퓨팅 환경에 대한 일반적인 개념과 관련 선행 연구들에 대해 살펴본다. 3장에서는 제안된 하이퍼바이저를 구성하는 중심 기술인 예외 처리, 가상 머신 관리, 스케줄링, 메모리 관리, 마이그레이션 등에 대해서 서술한다. 4장에서는 하이퍼바이저의 성능을 가상화되지 않은 시스템과 비교 분석 한다. 마지막으로 5장에서 결론을 도출하고 이후 도전 과제들과 발전 방향을 모색한다.

II. Related Works

플랫폼 가상화 가법은 크게 반가상화 (paravirtualization), 전가상화(full virtualization)로 나눌 수 있다. 반가상화란 가상화 계층 위에서 동작하는 게스트 운영체제를 하이퍼바이저에 맞게 적절히 수정하여 동작시키는 방법을 말한다. 또 다른 가상화 방법인 전가상화는 실제 CPU의 명령어들과 하드웨어를 소프트웨어적으로 에뮬레이션하거나, Intel의 Intel-VT 등과 같은 하드웨어적으로 지원하는 특별한 기능을 사용하여 가상화된 환경을 제공하는 방법을 말한다.

현재 범용 PC에서 가장 많이 사용되고 있는 VMware Workstation은 x86기반의 구조 위에서 동작하는 운영체제를 수정 없이 동작시킬 수 있는 하이퍼바이저이다. VMware는 MS-Windows와 Linux 운영체제에서 응용 프로그램으로써 동작한다. 게스트 운영체제의 동작은 응용 프로그램 단계에서 동작하지만, 특권 레벨의 명령어들을 수행하기 위해 하이퍼바이저가 디바이스 드라이버로 동작하게 된다. VMware의 전형적인 사용 예는 하나의 주 운영체제 위에서 다른 운영체제에서 동작하는 응용프로그램을 수행시키거나, 운영체제 레벨에서 디버깅 하기 위한 용도로 많이 사용된다.

또 다른 하이퍼바이저로는 반가상화를 사용하는 Xen이 있다. Xen의 주 용도는 하나의 물리적인 컴퓨터에서 여러 개의 서버를 동시에 동작시킴으로써 시스템의 전체 처리량을 높이고 비용을 절감 하는데 있다.

Bochs는 x86기반 구조에서 동작하는 게스트 운영체제를 운영체제의 수정 없이 동작시키는 머신 에뮬레이터이다. Bochs는 비교적 정확한 에뮬레이션을 하기 때문에 시스템 레벨의 응용프로그램을 디버깅하는 용도로 많이 사용되지만 성능이 다른 가상화 기술을 사용한 하이퍼바이저에 비해 낮기 때문에 실제 게스트 운영체제의 사용에는 적합하지 않다.

임베디드 시스템을 가상화하면 하나의 임베디드 시스템에 다수의 실시간 운영체제를 동시에 실행하는 것이 가능하다. 임베디드 시스템을 가상화하면 다양한 이점을 얻을 수 있다[2]. 첫 번째로 복잡한 시스템을 분리할 수 있다. 최근의 임베디드 시스템은 다양한 기능들이 계속 추가되면서 시스템의 복잡도가 높아졌기 때문에 구현이 어렵고 오류가 발생할 확률이 높아진다. 임베디드 시스템을 가상화하면 시스템을 기능에 따라 분리하고 각각을 독립적인 시스템으로 구성할 수 있다. 두 번째로 하나의 기계에서 다수의 운영체제를 동시에 수행시킬 수 있다. 특히 실시간 운영체제와 범용 운영체제를 함께 수행시키면, 실시간 운영체제를 통해 실시간 작업을 처리하면서 동시에 범용 운영체제의 다양한 응용을 활용할 수 있다. 세 번째로 보안성과

안정성을 향상시킬 수 있다. 임베디드 시스템을 가상화하면 각각의 운영체제에게 독립적인 환경을 제공함으로써 한 운영체제가 오류를 일으켜도 전체 시스템에 영향을 미치지 않고, 다른 운영체제들을 안정적으로 계속 수행할 수 있다. 네 번째로 멀티코어 CPU로의 확장이 용이하다. 향후 멀티코어 CPU가 주류를 이루게 되면 싱글코어 CPU 기반으로 설계된 기존 임베디드 운영체제를 수정해야 한다. 임베디드 시스템을 가상화하면 멀티코어를 가상화하여 각 운영체제에게 제공하기 때문에 기존 운영체제와 응용의 수정 없이 사용할 수 있다. 다섯 번째로 라이선스를 분리할 수 있다. GPL이 적용된 Linux의 드라이버 코드를 사용한 코드는 GPL을 따라야 하지만, 가상 머신 상에서 Linux를 수행하고 드라이버를 사용하면 GPL을 따르지 않고 라이선스를 분리할 수 있다. 임베디드 시스템에서 가상화 기술을 사용하면 기본적으로 다수의 물리적인 시스템을 하나로 통합할 수 있기 때문에 부피와 무게가 감소한다. 대부분의 실시간 임베디드 시스템은 외부 이벤트를 기다리다가 이벤트가 발생할 경우 그 정보를 처리하여 결과를 출력하는 방식이다. 실시간 임베디드 시스템을 가상화 할 경우, 외부 이벤트를 기다리는 대기 시간 동안 다른 작업을 처리할 수 있기 때문에 시스템의 효율적인 사용이 가능하다.

그간 하드웨어 성능 상의 제약이 임베디드 시스템의 가상화를 가로막은 것이 사실이나, ARM virtualization extensions[3], Intel사의 VT[4]나 AMD사의 AMD-v 등과 같이 최근의 대부분 마이크로프로세서가 제공하는 하드웨어 가상화 확장 (hardware virtualization extension) 기능을 활용하면 임베디드 시스템에서도 성능 저하 없이 가상화가 가능하다.

최근에는 임베디드 시스템을 목표로 한 가상화 시스템에 대한 연구가 활발하게 진행되고 있다 [5, 6]. 모바일 장치 등의 가상화에 대한 기존 연구는 [1]에 잘 정리되어 있다. 특히 이 연구에서는 모바일 장치의 마이크로프로세서, 메모리, 입출력 장치 등 각 하드웨어 구성 요소 별로 가상화 이슈를 소개하고, 각 가상화 기법들의 효율성을 비교한 평가 결과가 제시되었다.

[7]에서는 반가상화 기법에 기반을 둔 임베디드 시스템을 위한 간단한 가상 머신 모니터를 제안하고 있다. 이 시스템은 프로세서 가상화, 인터럽트 가상화, 가상화 관리 기법 등을 통해 각 게스트 운영체제에게 가상화된 하드웨어를 제공한다.

본 논문에서는 임베디드 시스템의 가상화를 위한 새로운 하이퍼바이저를 제안한다. 제안된 하이퍼바이저는 마이크로프로세서의 MMU 기능을 활용, 가상 메모리상에서 가상 머신이 동작하게 함으로써 게스트 운영체제 간의 독립성을 보장하도록 하였다. 또한 마이그레이션 기능을 제공하여 가상 머신의 동적 이동을 가능하게 하였다. 또한 실시간 운영체제의 특성을 고려하여 인터럽트와 시간에 대해 하이퍼바이저가 일방적으로 제공하는 구조를 갖추어 가상 머신에서 동작하게 될 운영체제의 수정이 불필요하게 하였다. 제안하는 하이퍼바이저는 ARM 구조에 기반을 둔 Intel사의 PXA255 칩셋을 장착한 개발용 보드를 대상으로 하여 구현되었다. 하이퍼바이저는 uClinux, uC/OS-II 등의 실시간 운영체제를 게스트 운영체제로 지원한다.

III. Design and Implementation of Hypervisor

1. System overview

본 논문에서 제안하는 임베디드 시스템을 위한 하이퍼바이저의 구조는 Fig. 1과 같다.

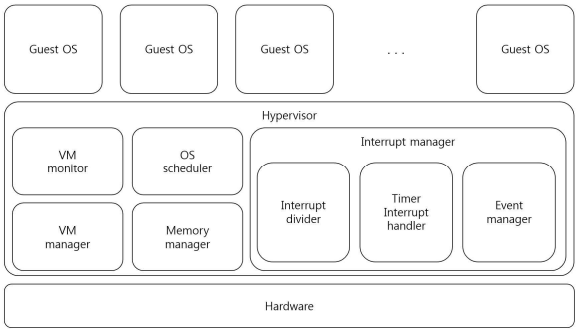


Fig. 1. System Architecture

그림에서 보는 바와 같이 하드웨어 바로 위에 하이퍼바이저가 위치하고, 이 하이퍼바이저가 메모리, Memory-Mapped I/O 장치 등과 같은 실제 물리 자원들을 관리하며 이를 가상 머신들에게 분배한다. 하이퍼바이저는 가상 머신들을 관리하기 위한 가상 머신 관리자와 물리/가상 메모리를 관리하기 위한 메모리 관리자, 각 게스트 운영체제에게 적절한 인터럽트를 전달하기 위한 인터럽트 제어 관리자, 각각의 가상 머신을 스케줄링 하기 위한 가상 머신 스케줄러, Memory-Mapped I/O 장치들의 동기화를 위한 이벤트 관리자, 가상 타이머로 구성된다. 이 가상 머신 상에서 여러 실시간 운영체제들이 커널 수정이나 포팅 작업 없이 동작하게 된다.

2. Virtual machine management

가상 머신들이 서로 영향을 받지 않고 독립적으로 동작하기 위해 각 가상 머신만의 고유한 정보를 관리할 필요가 있다. 이를 위해 Virtual CPU Info라는 객체를 두어 각 가상머신의 상태 정보, 레지스터 정보, 메모리 정보, 이벤트 정보, 스케줄 정보 등을 관리하도록 하였다. 가상 머신 관리자에서는 가상머신의 Virtual CPU Info를 이용해 생성, 제거, 일시 정지, 재개 등 가상 머신 제어에 필수적인 기능들을 수행할 수 있게 된다.

2.1 Managing virtual machines

제안된 하이퍼바이저는 가상 머신과 하이퍼바이저 간 제어 이동을 위해 가상 머신을 크게 두 개의 단계로 나누어 관리한다. 가상 하드웨어 환경을 제공해 주는 가상 CPU와 가상 CPU

상에서 동작하는 OS를 관리하기 위한 게스트 운영체제 Info라는 객체를 두어 관리한다(Fig. 2).

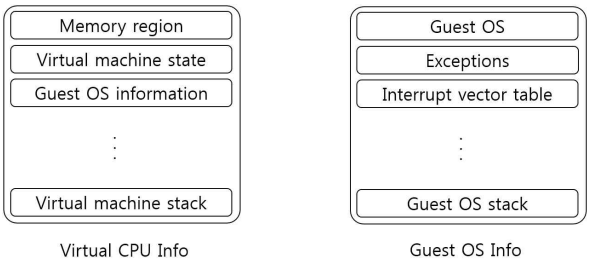


Fig. 2. Virtual CPU info and guest OS info

하이퍼바이저에서 스케줄링 단위는 가상 CPU이며, 스케줄링 시 가상 CPU 정보를 확인하여 가상 CPU에 요청된 이벤트나 OS 마이그레이션 요청 등이 있는지 확인한다. 만약 요청된 작업이 있다면 요청된 이벤트들을 처리한 후에 게스트 운영체제 Info 객체의 CPU 문맥을 복구하게 되고, 없다면 바로 게스트 운영체제의 CPU 문맥을 복구한다. 이 과정을 요약하면 Fig. 3과 같다.

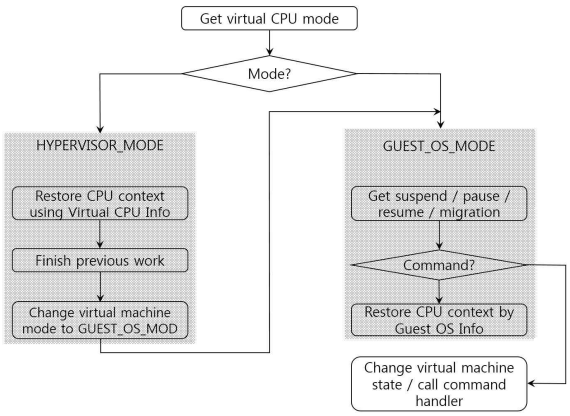
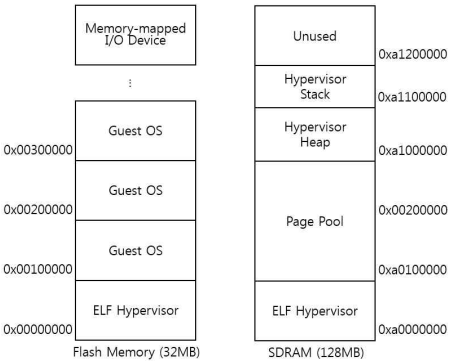


Fig. 3. Virtual machine state

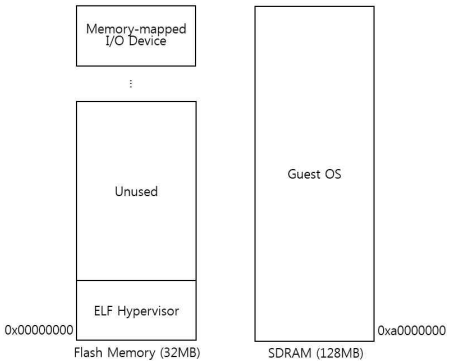
가상 머신이 동작하기 위해서는 가상 머신을 생성하기 위한 요청을 받을 수 있어야 한다. 본 논문에서 구현한 하이퍼바이저에서는 하이퍼바이저를 통해 사용자로부터 가상 머신 관련 요청을 받는다. 하이퍼바이저에서 사용자로부터 게스트 운영체제를 생성하라는 요청을 인식하면 먼저 새로운 가상 CPU를 생성하게 되고, 스케줄링이 가능한 상태가 되면 OS 스케줄링 큐에 들어가 실제 CPU시간을 부여받아 수행할 수 있게 된다. 실제 CPU를 할당받은 가상 CPU는 사용자로부터 얻은 게스트 운영체제정보를 통해 게스트 운영체제 Information 객체를 구성하고 페이지 테이블을 조작하여 메모리 주소 공간을 설정한다. 게스트 운영체제 Information 객체의 설정을 마치게 되면, 적절한 스케줄링 알고리즘에 따라 게스트 운영체제가 수행되게 된다.

2.2 Addressing of virtual machine

본 논문에서 구현한 하이퍼바이저는 모든 가상 머신에게 가상 메모리 주소 공간을 부여하여 각 게스트 운영체제가 독립적으로 수행하도록 한다. 제안된 하이퍼바이저는 MMU를 이용해 게스트 운영체제에게 보이는 메모리 영역을 가상화시킨다. 각 게스트 운영체제는 이 가상화된 메모리 영역을 사용하여 자기 자신만의 메모리 맵을 구축함으로써 정상적인 동작을 할 수 있게 된다(Fig. 4).



(a) Memory map seen by virtual machine monitor



(b) Physical memory map seen by guest os

Fig. 4. Memory map of virtual machine monitor

또한 게스트 운영체제가 하이퍼바이저에 의해 할당되지 않은 영역에 대한 접근을 시도할 때 예외를 발생 시킬 수 있도록 해당 페이지 테이블 엔트리 별로 특권 권한을 지정한다. 따라서 게스트 운영체제가 아직 할당되지 않은 메모리 영역이나 유효하지 않은 메모리 영역을 접근하려 하면 하이퍼바이저에서 이를 감지하여 처리해 줄 수 있다. 가상 머신의 가상 CPU에서 가상 머신이 사용할 메모리 공간에 대한 정보 또한 유지한다. 모든 가상 머신은 MMU를 통해 가상 메모리 주소 공간을 갖게 되며, 가상 CPU를 통해 현재 가상 머신에게 할당된 메모리 주소 공간이나, 유효한 메모리 주소 공간, 페이지 테이블 등에 대한 정보를 보관한다. 하이퍼바이저는 처음 가상 머신이 생성될 때, 메모리 공간을 효율적으로 사용하기 위해 OS를 수행시키기 위한 최소한의 메모리 영역만을 할당하여 페이지 테이블 엔트리를 구성하고, 그 외의 다른 주소공간은 페이지 테이블 엔트리를 구성하지 않은 상태에서 하이퍼바이저를 동작시킨다.

2.3 Address space of virtual machine

게스트 운영체제의 동작 시간이 길어짐에 따라 필요한 메모리 공간의 크기 또한 증가하기 때문에 동적으로 가상 머신의 메모리 주소 공간을 확장할 수 있는 메커니즘이 필요하다.

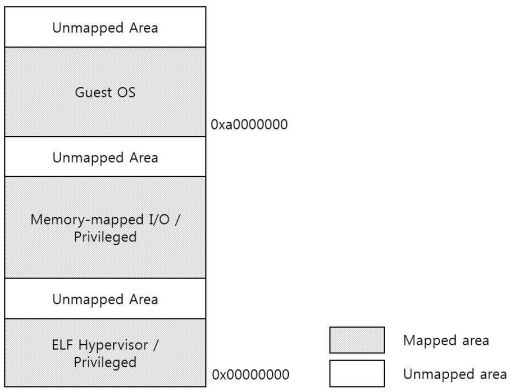


Fig. 5. Initial page table for virtual machine

Fig. 5에서 볼 수 있듯이 게스트 운영체제 Info에 구성된 페이지 테이블에 게스트 운영체제의 이미지 외의 다른 주소 공간은 아직 구성이 되지 않은 것을 확인할 수 있다. 게스트 운영체제가 아직 페이지 테이블 엔트리가 구성이 되지 않은 메모리 영역에 접근하려고 하면 페이지 폴트 예외가 발생하게 되는데 하이퍼바이저는 이 점을 이용해 동적으로 가상머신의 메모리 주소공간을 확장한다. 페이지 폴트 예외가 발생하게 되면, 인터럽트 서비스 루틴으로 분기가 되어 제어가 하이퍼바이저로 제어가 이동하게 된다. 접근하려는 메모리 주소가 유효한지 확인하고, 유효하다면 페이지 할당자를 사용해 적절한 페이지 테이블 엔트리를 설정해 주게 됨으로써 가상 CPU에 할당된 메모리 주소 공간을 확장하게 된다. 페이지 테이블 엔트리 설정이 끝난 후 제어를 다시 게스트 운영체제로 넘겨줌으로써 게스트 운영체제는 수행하고 있던 작업을 지속할 수 있게 된다.

3. Interrupt management

운영체제는 외부 장치로부터 인터럽트를 받을 수 있어야만 한다. 가상화 환경에서 동작하는 게스트 운영체제들은 하나의 물리적인 외부 인터럽트 장치를 공유하고 있기 때문에 인터럽트를 가상화시켜 적절히 분배해 줄 수 있는 메커니즘이 필요하다. 본 논문에서 제안한 하이퍼바이저는 대부분의 임베디드 시스템이 Memory-Mapped I/O장치를 사용한다는 점에 착안하여 인터럽트 가상화를 하였다.

MMU의 페이지 보호 기능을 이용해 실제 외부 장치가 연결되어있는 메모리 주소를 특권 레벨로 설정해 놓고 게스트 운영체제에서 해당 외부 장치에 접근을 시도할 때 접근 권한 예외를 발생시킴으로써 하이퍼바이저가 이를 감지하도록 하였다.

가상 머신이 동작하는 모드는 크게 두 가지가 있다. 하이퍼바이저로서의 역할을 하는 모드를 HYPERVISOR_MODE라 하는데, 이 상태에서는 인터럽트 처리나 운영체제 마이그레이션과 같은 일들을 하게 된다. 또 하나의 모드로 GUEST_OS_MODE가 있는데, 이 모드에서는 가상 머신이 게스트 운영체제를 수행하게 된다.

가상 CPU가 GUEST_OS_MODE로 동작하는 중 인터럽트가 발생하게 되면 하이퍼바이저에서 구성한 인터럽트 벡터로 분기하게 된다. 이 때 하이퍼바이저가 CPU의 제어권을 갖게 되며, 기존에 동작하던 게스트 운영체제의 문맥을 게스트 운영체제 Information 객체에 저장해 놓고 작업을 수행하게 된다. 하이퍼바이저로 제어가 넘어오게 되면 어떤 인터럽트가 발생했는지 알게 되고 인터럽트 종류에 따라 필요한 동작을 수행한 후, 다시 게스트 운영체제에 인터럽트를 전달하게 된다.

하이퍼바이저에서 인터럽트의 사전 처리를 마치고 제어가 가상 머신으로 넘어갈 때, 게스트 운영체제에서는 발생한 인터럽트가 하이퍼바이저로부터 전달된 것이라는 것을 알지 못하도록 해야 한다. 하이퍼바이저는 게스트 운영체제에서 사용하는 인터럽트 벡터 테이블의 위치를 게스트 운영체제 Information을 통해 알 수 있기 때문에, 인터럽트 플래그 레지스터를 설정한 상태로 게스트 운영체제 Context를 복구시키고, 적절한 인터럽트 벡터 엔트리로 분기시키게 된다.

가상 머신이 HYPERVISOR_MODE에서 동작 중 인터럽트가 발생할 수도 있다. 특정 게스트 운영체제에 대해 마이그레이션이나 CPU 사용이 큰 태스크 수행 시 가상머신이 HYPERVISOR_MODE로 동작하는 시간이 늘어나게 된다. 이 시간 동안 인터럽트를 금지시키면 시스템의 반응성이 현저히 느려지므로, 최소한의 임계영역을 설정한 후 임계영역 외에서는 인터럽트를 활성화한다. 따라서 가상 머신이 HYPERVISOR_MODE에서 동작하더라도 언제든지 인터럽트를 처리 할 수 있어야 한다. 이 경우 하이퍼바이저는 가상 CPU를 관리하는 자료구조에 HYPERVISOR_MODE로 동작 중이던 CPU의 문맥을 보관한다. 따라서 가상 머신의 CPU 문맥을 복구할 시기에 먼저 직전에 동작하던 가상 CPU의 모드가 HYPERVISOR_MODE인지 GUEST_OS_MODE인지를 확인 후에 적절히 문맥 복원을 함으로써 기존 동작을 이어나갈 수 있다.

4. Hardware synchronization

가상화 환경에서는 한정된 물리 자원을 다수의 게스트 운영체제가 사용하기 때문에 물리 자원들에 대해서 동기화를 지원해야 한다. 이를 위해 하이퍼바이저는 이러한 상황에 대해 각 운영체제의 독점권을 보장해준다. 하나의 게스트 운영체제가 특정 물리자원에 대한 사용을 요구하면 다른 게스트 운영체제에 대해 해당 장치에 대한 접근이나 사용을 제한하여 사용 중인 게스트 운영체제에 대한 물리자원의 점유 권한을 보장한다.

Fig. 6은 하드웨어 동기화에 대한 하이퍼바이저의의 관리구조를 보여준다. 그림 상에서 게스트 운영체제 1이 Device 1에 대해 독점권을 갖고 게스트 운영체제 2가 Device 2에 대해서 독점권을 갖게 되면, 게스트 운영체제 1이 Device 2의 제어 레

지스터(Control Register)에 대한 접근이 거부되고, 게스트 운영체제 2에서 Device 1의 제어 레지스터에 대한 접근은 제한되게 된다.

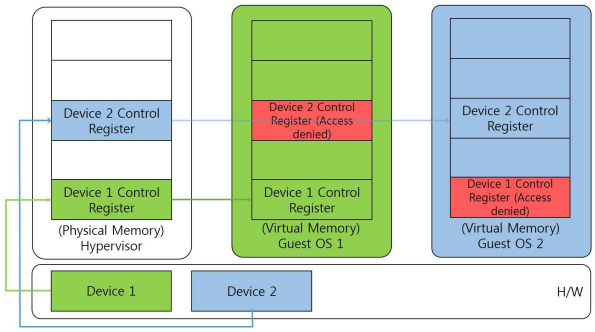


Fig. 6. Hardware synchronization

이처럼 하나의 게스트 운영체제가 특정 주변장치에 대해 사용 권한을 갖게 되면 해당 주변장치의 제어 레지스터의 메모리 주소에 대한 접근 권한을 해당 게스트 운영체제에게만 부여한다. 다른 게스트 운영체제가 접근하게 되면 해당 접근은 거부되어 독점 권한을 유지하게 된다.

5. Memory management

하이퍼바이저는 MMU 기능을 이용하여 메모리를 관리한다. 전체 물리 메모리 공간을 해당 영역의 성격과 용도에 따라 여러 영역(Region)으로 분류하고 각 영역을 페이지 테이블을 이용하여 단위 크기로 나누어 관리한다.

Fig. 7은 물리 메모리와 페이지 테이블, 그리고 각 가상 CPU의 가상 메모리 간의 관계를 보여준다. 우선 물리 메모리는 일정 단위의 페이지들로 나뉜다. 이런 페이지들이 모여 영역을 이루고, 영역의 각 페이지가 페이지 테이블 엔트리에 대응한다. 여기서 페이지 테이블 엔트리는 각 게스트 운영체제의 가상 메모리에 대응된다. 각 게스트 운영체제의 가상 메모리는 게스트 운영체제에 맞는 영역으로 분류되고 게스트 운영체제가 올라가는 영역은 모든 게스트 운영체제에서 같은 주소를 갖게 되어 일관된 메모리 관리가 가능하게 된다.

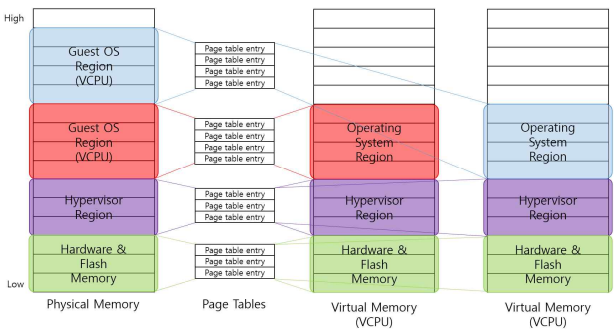


Fig. 7. Physical memory vs. virtual memory

5.1 Data structures

하이퍼바이저는 초기화 과정에서 MMU를 이용하여 물리 메모리와 가상 메모리의 초기화 및 설정을 수행한다. 이 때 물리 메모리와 가상 메모리간의 관계를 정의하고 동적인 메모리 관리를 위해 영역과 페이지 테이블 등 메모리 관리 자료 구조에 대한 초기화 동작을 수행한다. 이후 물리 메모리를 관리하기 위해 각 물리 메모리를 기본 메모리 관리의 구성단위인 페이지로 나누어 비트맵 구조로 관리한다.

하이퍼바이저는 페이지들을 관리하기 위해 페이지 할당자를 이용하여 새로운 게스트 운영체제를 생성하거나 게스트 운영체제 동작 중 새로운 메모리 공간을 요청하는 경우(페이지 폴트 예외 발생 시) 새로운 페이지 할당을 하게 된다. 또한 반대의 경우인 게스트 운영체제를 제거할 때 해당 페이지 반납을 위해 페이지 할당자의 동작이 이루어진다.

5.2 Page fault handling

먼저 게스트 운영체제에 대한 페이지 할당이 이루어지는 경우는 두 가지로 나누어 볼 수 있다. 첫 번째는 사용자의 요청이나 운영체제 이식과 같은 요청에 의해 이루어지는 경우가 있다. 이 경우에는 페이지 할당자를 호출하여 요청을 원하는 크기에 맞는 페이지 개수를 측정하여 할당하게 된다. 두 번째 경우는 운영체제의 메모리 확장에 따른 경우가 있다. 이 경우 운영체제는 자신에게 할당된 물리메모리 공간 이외의 공간에 대해 접근을 시도하기 때문에 MMU는 이에 대해 접근 위반에 의한 데이터 어보트 (Data Abort) 예외를 발생시킨다. 이 예외에 대해서 하이퍼바이저는 예외 처리기로 분기를 하고 해당 예외 발생에 대해 확인한 후 페이지 폴트인 경우에 대해서 페이지 할당자를 호출하여 새 페이지를 할당한 뒤 예외를 발생시킨 명령으로 돌아가서 다시 수행을 한다.

5.3 Memory allocation and deallocation

페이지 할당과 반납 과정은 다음과 같다. 새로운 페이지를 할당하기 위해 우선 동작중인 가상CPU의 유효한 주소 공간과 비트맵 테이블에 충분한 페이지가 있는지 확인한다. 만약 게스트 운영체제의 이미지 크기가 연속된 페이지보다 크면 할당 불가하므로 오류를 발생시킨다. 유효한 경우 가상CPU내의 메모리 관리 자료구조에 실제 페이지의 주소를 할당한다.

비트맵 테이블로 관리 중인 메모리에서 요청한 페이지 해당 위치의 비트를 사용불가(비트값 1)로 설정한다. 따라서 이 공간은 가상CPU에 할당한 페이지가 되고 다음 페이지 할당이나 반납 시에 유효 공간을 정확히 확인할 수 있다. 자료구조 상의 할당이 끝나고 나면 가상CPU의 메모리 관리 자료구조를 참조하여 MMU가 가상 메모리와 실제 메모리 간의 주소에 맞는 할당 작업을 수행한다. 마지막으로 페이지 프레임 시작 주소 반환하여 게스트 운영체제에 페이지 할당 작업을 완료한다.

메모리 반납 시에는 반납 요청 된 페이지를 가상CPU가 사용하고 있는지 여부를 판단한다. 만약 페이지를 소유하고 있었다면 비트맵 테이블에서 해당 페이지 인덱스를 사용가능(비트값 0)으로 설정한다. 따라서 이 공간은 비어있는 사용 가능한

공간이 되고 다음 페이지 할당이나 반납 시에 유효 공간을 정확히 확인할 수 있다. 이후 가상CPU의 메모리 자료상에서 명시된 매핑을 해지함으로써 게스트 운영체제의 메모리 반납이 완료된다.

6. Scheduling virtual machines

하이퍼바이저는 게스트 운영체제들의 실시간성을 보장하기 위해 게스트 운영체제에서 수행하게 될 작업들의 실시간성에 대해 파악을 해야 한다. 하이퍼바이저는 이를 게스트 운영체제 사용자에 의해 얻을 수 있으며 이를 기준으로 각 게스트 운영체제를 대상으로 스케줄링을 실시한다. 하지만 엄밀히 말하면 하이퍼바이저의 실제 스케줄링 대상은 게스트 운영체제가 아닌 가상 CPU이다. 하이퍼바이저와 접하여 수행하는 계층의 개념이 게스트 운영체제가 아닌 가상 CPU이고, 가상 CPU가 하이퍼바이저로부터 게스트 운영체제에게 할당 되는 논리적 자원이기 때문에 이를 대상으로 스케줄링을 한다.

6.1 Scheduling policy

가상 머신 스케줄링 시 각 가상 머신에 시간 할당량을 분배하는 정책은 다음과 같다. 가상 머신과 그 게스트 운영체제를 생성할 때 사용자가 제시하는 운영체제의 우선순위는 운영체제 내의 태스크들의 우선순위의 평균치로 계산되어 각 게스트 운영체제를 대표하는 우선순위(RV)로 나타낸다. 이 값은 일반화 과정을 거쳐 해당 게스트 운영체제의 대표 우선순위(NRV)가 되고, 이를 기준으로 기본 시간 할당량(DTQ)의 대표값의 배수를 해당 게스트 운영체제의 시간 할당량(TQ)으로 결정한다. 이를 통해 하이퍼바이저가 특정 게스트 운영체제를 스케줄링 할 때 해당 게스트 운영체제 내에 실행 중인 태스크들의 우선순위가 반영될 수 있도록 하였다.

P: 게스트 운영체제 내의 작업의 우선순위
RV_i: 게스트 운영체제 i의 우선순위의 대표값
NRV_i: 게스트 운영체제 i의 일반화된 대표값
DTQ: 기본 Time Quantum
TQ_i: 게스트 운영체제 i의 Time Quantum

$$RV_i = (P_1 + P_2 + \ldots + P_n) / n$$
$$NRV_i = RV_i / (RV_1 + RV_2 + \ldots RV_n)$$
$$TQ_i = NT_n \times DTQ$$

6.2 Scheduling mechanism

스케줄링을 수행하면서 가상 CPU간의 전환에 있어서 사용되는 방법은 기본적으로 Linux kernel에서 쓰이는 O(1) 스케줄링 방법을 사용한다. O(1) 스케줄링은 대상 개수에 관계없이 스케줄링에 할애하는 시간이 일정함을 보장한다. 현재 수행하는 대상의 지난 수행시간과 앞으로의 수행시간을 변동하여 해당 대상의 작업이 할당 받은 시간 할당량을 모두 사용하면 대상을 종료한 작업으로 분류하여 다음 스케줄링에서 스케줄링의 대상이 되지 않도록 한다. 이후 모든 대상이 시간 할당량을 소진하게 되어 스케줄링의 대상이 없어지게 되면 종료 작업으로 분류된 대상들을 모아둔 자료구조와 미종료 작업으로 분류하는 자료구조를 바꿔서 새로운 시간 할당량에서의 스케줄링을 수행한다. 여기서 자료구조를 바꾸는 동작은 스케줄링 대상의 개수에 관계없이 일정시간 안에 이루어지게 된다.

7. Migration of virtual machines

제안된 하이퍼바이저는 수행 중인 운영체제를 하나의 물리 기계에서 다른 물리 기계로 이동시키는 마이그레이션 [8] 기능을 제공한다.

7.1 Precondition for migration

마이그레이션 데이터를 송신하는 측의 하이퍼바이저는 마이그레이션을 진행하기 전, 일반적인 작업의 수행 도중에 네트워크상의 다른 물리자원의 하이퍼바이저들과 각자의 자원 환경에 대한 제약사항 정보를 공유한다. 각 하이퍼바이저는 주기적으로 자신들이 위치한 물리자원에서 메모리의 공간정보를 갱신하여 다른 하이퍼바이저에게 전송하여 알려주고 이를 받은 하이퍼바이저는 정보를 보낸 하이퍼바이저에 대해 자신이 보유하고 있는 정보를 갱신한다. 이러한 작업이 주기적으로 이루어져서 각 하이퍼바이저는 다른 공간에 위치한 물리자원의 제약 정보를 계속해서 갱신하여 갖고 있게 된다.

7.2 Sender hypervisor

이렇게 작업이 수행하던 도중 하이퍼바이저가 사용자나 내부에서 마이그레이션 요청을 받게 되면 대상이 되는 게스트 운영체제를 식별하고 이전에 갱신해왔던 다른 위치의 하이퍼바이저들의 제약정보들 중에서 마이그레이션 대상이 되는 게스트 운영체제의 데이터크기와 가장 적합한 정보를 가진 하이퍼바이저를 선정하고 해당 하이퍼바이저에게 마이그레이션의 허가 요청 신호를 전송하고 허가 신호를 기다린다. 수신하는 하이퍼바이저에서 허가 신호가 오게 되면 송신하는 하이퍼바이저는 해당 게스트 운영체제의 작업 상태를 보존하고 보존된 작업 상태 및 데이터를 하이퍼바이저가 다른 환경으로 전송 가능한 형태의 이미지로 변환한다. 해당 게스트 운영체제는 현재의 하이퍼바이저가 관리하는 수행 대상에서 빠지게 되고 메모리에서 차지하던 공간을 반납한다. 메모리 반납까지 마치면 하이퍼바이저는 변환된 게스트 운영체제 데이터를 이주 환경으로 전송을 실시한다. 전송이 완료되면 하이퍼바이저는 재정의 된 수행대상의 게스트 운영체제들을 계속해서 수행한다.

7.3 Receiver hypervisor

마이그레이션 데이터를 수신하는 하이퍼바이저는 마이그레이션이 이루어지기 전에 송신 하이퍼바이저로부터 마이그레이션 허가 요청 신호를 받는다. 이 신호를 받으면 하이퍼바이저는 현재 요청한 크기의 게스트 운영체제가 위치 가능한 메모리 공

간의 여부를 확인하고 허가 여부를 결정하여 허가 신호를 보낸다. 허가신호를 보내고 나서 하이퍼바이저는 수신대기 상태를 유지한다. 이후 송신 측에서 마이그레이션대상 게스트 운영체제의 이미지를 보내면 이를 수신하고 모든 데이터의 수신이 끝나게 되면 이전에 확인된 메모리 위치로 해당 게스트 운영체제의 이미지를 로드하고 하이퍼바이저는 새로운 게스트 운영체제를 수행대상에 추가하고 재정의된 수행대상들의 수행을 재개한다.

IV. Performance Evaluation

이 장에서는 본 논문에서 구현한 하이퍼바이저에 대한 성능 측정 결과를 소개한다. 성능 평가에서는 제안한 하이퍼바이저의 가상 머신 상에서 동작하는 uCLinux의 성능을 측정하고, 이를 가상화하지 않은 uCLinux와 비교함으로써 하이퍼바이저의 성능을 간접 평가하였다. 성능 측정 도구로는 널리 사용되는 LMbench[9] 벤치마크 프로그램을 사용하였다. 실제 성능 평가에서는 LMbench의 다양한 벤치마크 중 메모리 접근 대역폭, 시스템 콜 수행 시간, 문맥 교환 시간 등에 대한 측정을 주로 수행하였다.

Fig. 8은 LMbench 중 bw_mem 벤치마크를 실행한 결과이다. 이를 위해 마이크로프로세서가 메모리상의 1MB 크기의 데이터에 대해 읽기에 걸리는 시간, 쓰기 시간, 읽기/쓰기에 걸리는 시간을 측정하였다. 그림에서 보는 바와 같이 메모리 대역폭의 경우 가상화하지 않은 시스템과 비교해 성능 차이가 거의 없음을 확인하였다.

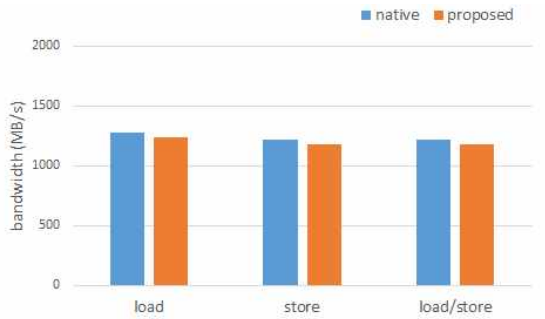


Fig. 8. Memory bandwidth

Fig. 9는 LMbench 중 시스템 콜 수행 시간을 측정하는 lat_syscall 벤치마크를 이용하여 가상 머신에서의 시스템 콜 수행 시간을 측정한 결과이다. 여기서는 write 시스템 콜을 이용해 /dev/null 파일에 한 바이트를 쓰는 과정에서 운영체제에 진입하는 시간을 측정하였다. 그림에서 보는 바와 같이 가상 머신 상에서의 시스템 콜 수행 시간은 가상화하지 않은 시스템의 경우에 비해 약 76% 정도 증가함을 알 수 있다.

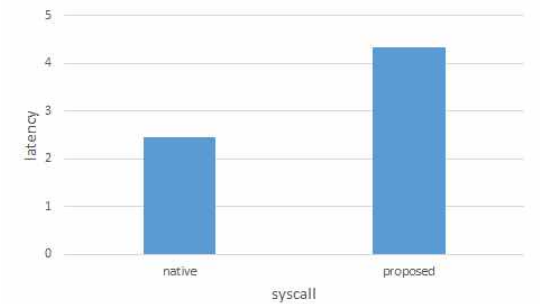


Fig. 9. System call latency

Fig. 10은 LMbench의 lat_ctx 벤치마크를 통해 가상 머신 상에서 프로세스 개수 변화에 따른 문맥 교환에 걸리는 시간을 가상화하지 않은 시스템과 비교한 결과이다. 그림에서 보는 바와 같이 가상 머신에서의 문맥 교환에는 가상화되지 않은 시스템에 비해 훨씬 많은 시간이 소요됨을 알 수 있다.

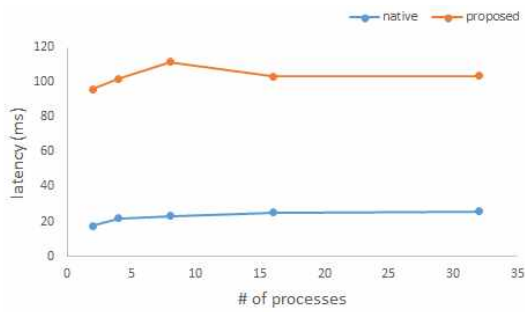


Fig. 10. Context switch latency

V. Conclusions

본 논문에서는 ARM 기반 임베디드 시스템의 가상화를 위한 하이퍼바이저를 제안하였다. 이 하이퍼바이저는 단일 마이크로프로세서에서 여러 개의 실시간 운영체제의 수행을 지원한다. 제안된 하이퍼바이저는 마이크로프로세서의 MMU 기능을 사용하여 가상 메모리상에서 게스트 운영체제 간의 메모리 공간 동기화를 보장하며 이를 통해 운영체제 간의 독립성을 보장한다. 이 외의 제안된 하이퍼바이저는 마이그레이션 기능을 제공한다. 마이그레이션을 통해 다른 물리 임베디드 시스템으로의 동적인 운영체제의 이동을 가능하게 함으로써 부하 분산이 가능하다. 또한 기존의 반가상화 기법들과는 달리 게스트 운영체제 커널의 수정 없이 실행이 가능하도록 하였다. 실시간 운영체제가 갖는 특성을 고려하여 인터럽트와 시간에 대해 하이퍼바이저가 일방적으로 제공하는 구조를 갖추어 가상 머신에서 동작하게 될 운영체제의 내부 수정이 불필요하도록 하였다. 향후 실시간 운영체제와 비실시간 운영체제를 동시에 수행될 때, 실시간 요구 조건을 충족할 수 있는 스케줄링 방식에 대한 연구도 진행되면 다양한 분야에서 사용될 수 있는 임베디드 가상화 솔루션이 될 것으로 기대한다.

REFERENCES

[1] J. Shuja, A. Gani, K. Bilal, A. Khan, S. A. Madani, S. U. Khan, and A. Y. Zomaya, "A Survey of Mobile Device Virtualization: Taxonomy and State of the Art," ACM Computing Surveys, Vol. 49, No. 1, pp. 1-36, July 2016.

[2] G. Heiser, "The Role of Virtualization in Embedded Systems," Proceedings of the First Workshop on Isolation and Integration in Embedded Systems, pp. 11-16, Glasgow, UK, April 2008.

[3] P. Varanasi and G. Heiser, "Hardware-Supported Virtualization on ARM," Proceedings of the Second ACM SIGOPS Workshop on Systems, pp. 1-11, Shanghai, China, July 2011.

[4] Intel Virtualization Technology(Intel VT), *http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html*.

[5] W. Chen, L. Xu, G. Li, and Yang Xiang, "A Lightweight Virtualization Solution for Android Devices," IEEE Transactions on Computers, Vol. 64, No. 10, pp. 2741-2751, October 2015.

[6] W. Li, L. Liang, M. Ma, Y. Xia, and H. Chen, "TVisor-A Practical and Lightweight Mobile Red-Green Dual-OS Architecture," Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services. pp. 485-485, Florence, Italy, May 2015.

[7] S. Son and J. Lee, "Design and Implementation of Virtual Machine Monitor for Embedded Systems," Journal of The Korea Society of Computer and Information, Vol. 14, No. 1, pp. 57-64, January 2009.

[8] R. W. Ahmad, A. Gani, S. H. A. Hamid, M. Shiraz, A. Yousafzai, and F. Xia, "A Survey on Virtual Machine Migration and Server Consolidation Frameworks for Cloud Data Centers," Journal of Network and Computer Applications, Vol. 52, No. 6, pp. 11-25, June 2015.

[9] L. McVoy and C. Staelin, "lmbench: Portable Tools for Performance Analysis," Proceedings of the USENIX 1996 Annual Technical Conference, pp. 279-294, San Diego CA, USA, January 1996.

Authors

Sunghoon Son received his B.S., M.S. and Ph.D. degrees in Computer Science from Seoul National University, Korea, in 1991, 1993 and 1999, respectively. Dr. Son joined the faculty of the Department of Computer Science at Sangmyung University, Seoul, Korea, in 2004. He is currently a Professor in the Department of Computer Science, Sangmyung University. He is interested in system software, embedded system, and virtualization.