# 스토리지 클래스 램을 위한 통합 소프트웨어 구조
## (A Unified Software Architecture for Storage Class Random Access Memory)

백 승 재 † 최 종 무 ††
(Seungjae Baek) (Jongmoo Choi)

**요 약** 바이트 단위 임의 접근이라는 램 특성과, 비휘발성이라는 디스크의 특성을 동시에 제공하는 FeRAM, MRAM, PRAM등의 스토리지 클래스 램(Storage Class Random Access Memory, SCRAM)이 소형 임베디드 시스템을 중심으로 점차 그 활용범위를 넓혀가고 있다. 본 논문에서는 SCRAM을 주 기억 장치 및 보조 기억 장치로서 동시에 사용할 수 있는 차세대 통합 소프트웨어 구조를 제안한다. 제안된 구조는 크게 스토리지 클래스 램 드라이버(SCRAM Driver)와 스토리지 클래스 램 관리자(SCRAM Manager)로 구성된다. SCRAM Driver는 SCRAM을 직접 관리하며, FAT이나 Ext2와 같은 전통적인 파일 시스템이나 버디 할당자와 같은 전통적인 메모리 관리자, 혹은 SCRAM Manager 등의 상위 소프트웨어 계층에 저수준 인터페이스를 제공한다. SCRAM Manager는 파일 객체와 메모리 객체를 통합하여 관리함으로써 이들 간에 부가적인 비용이 없는 변환을 가능케 한다. 제안된 기법은 FeRAM이 장착된 실제 시스템에서 실험되었으며, 실험 결과를 통해 SCRAM Driver가 효율적으로 전통적인 파일시스템과 메모리 관리자가 요구하는 기능을 제공할 수 있음을 보였다. 또한 기존의 파일 시스템과 메모리 관리자를 통해 각각 SCRAM을 접근하는 경우보다 SCRAM Manager가 수십 배 빠른 성능을 보임을 확인할 수 있었다.
**키워드** : 스토리지 클래스 램, 단일 객체, 객체 이동

***Abstract*** Slowly, but surely, we are seeing the emergence of a variety of embedded systems that are employing Storage Class RAM(SCRAM) such as FeRAM, MRAM and PRAM. SCRAM not only has DRAM-characteristic, that is, random byte-unit access capability, but also Disk-characteristic, that is, non-volatility. In this paper, we propose a new software architecture that allows SCRAM to be used both for main memory and for secondary storage simultaneously. The proposed software architecture has two core modules, one is a SCRAM driver and the other is a SCRAM manager. The SCRAM driver takes care of SCRAM directly and exports low level interfaces required for upper layer software modules including traditional file systems, buddy systems and our SCRAM manager. The SCRAM manager treats file objects and memory objects as a single object and deals with them in a unified way so that they can be interchanged without copy overheads. Experiments conducted on real embedded board with FeRAM have shown that the SCRAM driver indeed supports both the traditional FAT file system and buddy system seamlessly. The results also have revealed that the SCRAM manager makes effective use of both characteristics of SCRAM and performs an order of magnitude better than the traditional file system and buddy system.
**Key words** : SCRAM, Single Object, Object Migration

· 이 연구는 2008학년도 단국대학교 대학연구비 지원으로 연구되었음
† 정 회 원 : 단국대학교 컴퓨터학과
ibanez1383@dankook.ac.kr
†† 종신회원 : 단국대학교 컴퓨터학부 교수
choijm@dankook.ac.kr
논문접수 : 2008년 10월 16일
심사완료 : 2009년 1월 23일


정보과학회논문지: 시스템 및 이론 제36권 제3호(2009.6)

## 1. 서 론

컴퓨터 시스템의 기억장치는 크게 DRAM(Dynamic Random Access Memory)과 같은 주 기억 장치와 HDD(Hard Disk Driver)와 같은 보조 기억장치로 구분된다. 그러나 최근 FeRAM(Ferroelectric RAM), MRAM (Magnetoresistive RAM), PRAM(Phase-change RAM)과 같은 다양한 종류의 스토리지 클래스 램(Storage Class RAM, SCRAM)이 도입됨에 따라 점차 이러한 전통적인 메모리 계층 구조에 변화가 생기고 있다. SCRAM은 바이트 단위 임의 접근성(random-accessibility)과 비휘발성(non-volatility)이라는 특징을 모두 제공하는 차세대 메모리이다[1,2]. 따라서 SCRAM은 동적 프로그램 수행 공간을 위한 주 기억 장치로도 사용 가능하며, 혹은 파일을 저장하기 위한 보조 기억장치의 용도로도 사용가능하다.

최근 이러한 SCRAM을 활용하는 다양한 연구가 진행되고 있으며, 특히 램트론(Ramtron), 프리스케일(Freescale), 삼성, 후지쓰(Fujitsu), IBM, TI(Texas Instruments)등 주요 반도체 업체에서는 이미 SCRAM과 이를 활용한 제품을 판매 중이거나 예정하고 있다. 구체적으로 램트론 사에서는 FeRAM을 장착한 RAID controllers, Smart I/O modules, single board computers 등을[3], 프리스케일 사에서는 MRAM을 도입한 processors와 자동차 센서에 도입되는 controller chips 등을 이미 판매하고 있다[4]. 이러한 추세로 미루어 볼 때, 기존 저장장치 대비 다양한 장점을 가지고 있는 SCRAM을 도입함으로써 전통적인 컴퓨터 시스템의 저장장치 계층 구조에서는 이룰 수 없었던 수준의 성능과 신뢰성을 이루기 위한 연구가 이어질 것으로 예상된다.

SCRAM을 활용하여 성능과 신뢰성 향상을 도모한 많은 선행 연구가 존재한다[5,6]. 그러나 이들은 모두 SCRAM이 가지고 있는 주 기억 장치로써의 특성과 보조 기억 장치로써의 특성 중 한 가지만을 활용하고 있다. 이는 기존 연구가 SCRAM을 단순히 주 기억 장치의 확장, 혹은 보조 기억 장치의 확장으로만 취급하였으며, 따라서 SCRAM이 가지고 있는 고유의 장점을 제대로 활용하고 있지 못함을 의미한다. 본 논문에서는 SCRAM이 가지고 있는 두 가지 특성을 완벽히 활용하기 위한 새로운 소프트웨어 계층 구조를 제안한다.

제안된 소프트웨어 계층 구조는 *SCRAM driver*와 *SCRAM manager*로 구성된다. *SCRAM driver*는 SCRAM을 직접 관리하며, 상위 수준의 소프트웨어가 필요로 하는 인터페이스를 제공한다. 예를 들어, 파일 시스템이 SCRAM의 공간을 요청하면, *SCRAM driver*는 SCRAM의 일부 영역을 분할한 뒤, sector_read(), sector_write()와 같은 함수를 제공해 주며, 버디 시스템이 SCRAM의 공간을 요청하면, 역시 SCRAM의 일부 영역을 분할한 뒤 이를 버디 시스템이 사용할 수 있도록 해주는 역할을 수행한다. 즉, *SCRAM driver*를 통해 SCRAM의 공간을 분할하여 상위 수준에서 사용할 수 있으며, 파일 시스템은 파일 객체(file object)를 메모리 관리자는 메모리 객체(memory object)를 각각 자신의 SCRAM 공간에서 관리한다.

*SCRAM manager*는 *SCRAM driver*의 상위 수준에서 동작하는 소프트웨어로써, 상위 수준에 파일 객체와 메모리 객체의 개념을 제공하는 동시에 파일 객체나 메모리 객체를 각각 제공하는 기존의 파일 시스템이나 메모리 관리자와는 달리 *SCRAM manager*는 이 두 가지 객체를 단일 객체(single object)로 취급하고 관리한다. 이러한 관리가 가능한 것은 SCRAM 상에서는 malloc() 등의 함수로 생성되는 메모리 객체와 creat() 등의 함수로 생성되는 파일 객체는 이름(naming) 등의 메타 데이터를 유지하는가의 차이점 외엔 완전히 동일하기 때문이다. 이는 메모리 객체에 이름 등의 메타 데이터만 기록해주면 이를 파일 객체로 변환하는 것이 가능함을 의미한다.

파일 객체와 메모리 객체를 단일 객체로서 관리하는 정책은 *SCRAM manager*가 기존 기법에 비해 보다 단순한 구조와 효율적인 성능을 보일 수 있도록 한다. 전통적인 운영체제에서 파일 객체는 파일 시스템에 의해, 메모리 객체는 메모리 관리자에 의해 각각 관리 되었다[7]. 그러나 본 논문에서 제안한 *SCRAM manager*는 단일 객체 관리 기법을 통해 이 두 개념을 동시에 제공하며, 따라서 보다 단순화된 운영체제 구조를 유지할 수 있게 해준다. 또한 파일 객체와 메모리 객체간의 객체 이동(object migration) 역시 메타 데이터만 추가로 기록하거나 혹은 해제 시켜주면 실제 데이터의 복사 비용(copy overhead) 없이 이뤄질 수 있으며 이를 통해 성능향상을 얻을 수 있다.

본 논문에서 제안한 SCRAM 관리 소프트웨어 계층 구조, 즉 *SCRAM driver*와 *SCRAM manager*는 실제 SCRAM이 장착된 컴퓨터 시스템에서 구현되었다. 실험에 사용된 컴퓨터 시스템은 PXA255 프로세서를 장착하고 있는 범용 임베디드 보드이며, 본 연구진은 실험을 위해 총 32MB의 FeRAM을 장착할 수 있는 daughter board를 제작하였다. 이 시스템에는 리눅스 커널 2.6.21이 포팅되어 동작하며, *SCRAM driver*와 *SCRAM manager*는 리눅스에 동적으로 적재 가능한 모듈 형태로 구현되었다[8].

실험을 위해 *SCRAM driver* 모듈 상위에 FAT 파일 시스템과 버디 할당자, 그리고 본 논문에서 제안한

*SCRAM manager*를 동작시켰으며, 다양한 벤치마크를 수행하여 그 성능을 비교하였다. 실험 결과를 통해 본 논문에서 제안한 *SCRAM manager*가 기존의 FAT 파일 시스템 대비 1.3~20배, 기존의 버디 할당자 대비 1.3~3배의 성능향상을 보임을 확인 할 수 있었다.

본 논문의 구성은 다음과 같다. 2장에서 관련연구를 소개한 뒤, 3장에서 스토리지 클래스 램을 위한 소프트웨어 계층 구조에 대해 설명한다. 4장에서는 실제 구현 내용을 설명한 뒤, 5장에서는 실험결과에 대해 기술한다. 끝으로 6장에서는 결론 및 향후 연구 방향을 소개한다.

## 2. 관련 연구

이 장에서는 SCRAM을 이용한 기존의 연구 사례를 분석한다.

1990년대 초반 [5]의 연구에서는 배터리 백 램(battery-backed RAM)형태의 SCRAM을 이용하여 파일 서버로의 쓰기 트래픽(write traffic)과 디스크로의 접근을 효율적으로 감소시킬 수 있음을 보였다. 또한 [6]의 연구에서는 시스템의 중요한 정보를 SCRAM에 저장함으로써 추후 시스템을 빠르게 안정한 상태로 복원해 낼 수 있음을 보였다. 이러한 초창기 연구들은 시스템의 성능과 안정성을 향상시키기 위해 SCRAM이 사용될 수 있음을 보여준 중요한 연구이다.

SCRAM을 보조 기억 장치의 확장으로 간주한 연구도 진행된 바 있다. SCRAM 상의 공간을 절약하기 위해 데이터를 압축하여 저장한 MRAMFS[9], Ext2파일 시스템이 확장된 형태의 PRAMFS[10], 공간 효율성을 높이기 위한 익스턴트 기반의 할당을 하는 NEBFS [11]등의 SCRAM 전용 파일 시스템이 제안된 바 있으며, 일반 사용자 데이터는 기존의 보조 기억장치에 저장하고 메타 데이터를 SCRAM에 저장하는 하이브리드 구조의 저장장치가 소개되었다[12-14].

SCRAM을 주 기억 장치의 확장으로 간주한 연구 역시 활발히 진행되었다. [15]의 연구에서는 기존 휘발성 RAM과 SCRAM이 혼재된 형태의 버퍼 관리 구조에 대해 소개하였으며, [16]에서는 디스크로의 쓰기를 지연시키는 용도의 SCRAM 버퍼를 소개하였다. [17]의 연구에서는 UPS(Uninterruptible Power Supply) 혹은 SCRAM을 이용하여 기존시스템의 주 기억 장치를 안전하게 하기 위한 기법을 소개하였으며, [18]에서는 플래시 메모리를 도입하여 대용량의 비휘발성 주 기억 장치를 제안하였다.

그러나 이들 연구는 모두 SCRAM의 한 가지 특성만을 활용하였다. 따라서 본 연구는 SCRAM의 두 가지 특성을 모두 활용하는 첫 시도라는 점에서 기존 연구와 차별화 된다.

## 3. 스토리지 클래스 램을 위한 소프트웨어 계층 구조

이번 장에서는 SCRAM을 위한 소프트웨어 계층 구조(layered architecture)를 소개한다. 그런 뒤 소개된 소프트웨어 계층 구조의 두 가지 핵심 모듈인 *SCRAM driver*와 *SCRAM manager*의 구체적인 기능에 대해 설명한다.

소프트웨어의 계층 구조는 복잡한 소프트웨어를 구축할 때, 특히 기존 소프트웨어와의 호환성을 염두에 둘 때 매우 유용하다. 본 연구진 역시 그림 1에 나타낸 바와 같이 SCRAM을 위한 소프트웨어를 계층 구조로 설계하였다.

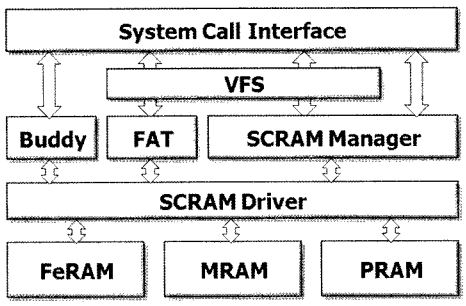



그림 1 SCRAM을 위한 소프트웨어 계층 구조

제안된 소프트웨어 계층 구조는 크게 4단계 - 시스템 호출 단계, VFS(Virtual File System) 단계, SCRAM 사용자 모듈(user module) 단계, 그리고 *SCRAM driver* 단계 - 로 구성된다. 각 SCRAM 칩은 그림 1의 가장 아래 단계에 존재하며, 하드웨어 적으로 DRAM처럼 접근하는 것이 가능하다[3,4]. *SCRAM driver*는 SCRAM을 직접 관리하며 SCRAM 사용자 모듈 단계에서 필요로 하는 인터페이스를 제공한다. *SCRAM driver* 상위에는 크게 세 가지 종류의 사용자 모듈을 동시에 수행시키는 것이 가능하다. 첫째는 FAT이나 Ext2와 같은 전통적인 파일 시스템이고, 둘째는 버디 할당자나 슬랩 할당자등의 전통적인 메모리 관리자이며, 셋째는 본 논문에서 제안한 *SCRAM manager*이다. SCRAM 사용자 모듈은 VFS와 시스템 호출 단계를 거쳐서 사용자 수준 응용 프로그램에게 메모리 객체와 파일 객체에 대한 인터페이스를 제공한다.

*SCRAM driver*가 수행하는 일은 구체적으로 다음과 같다. 우선 각 SCRAM 칩의 초기화와 주소 매핑 등의 작업을 담당한다. 또한 SCRAM 사용자 모듈의 요청에 따라 SCRAM을 연속된 구간으로 분할하여 관리해준다. *SCRAM driver*가 SCRAM을 분할하여 그 시작 번지

를 반환하면 버디 할당자 등의 메모리 관리자는 별도의 인터페이스 함수의 호출 없이 SCRAM을 그대로 접근할 수 있다. 그러나 파일 시스템의 경우 SCRAM을 블록 장치로 접근할 수 있어야 하는데 이 작업 역시 *SCRAM driver*가 담당한다. 구체적으로 파일 시스템은 섹터 단위로 읽고 쓸 수 있는 인터페이스를 요구하는데, SCRAM driver는 이러한 인터페이스 제공 및 SCRAM을 블록 장치로 에뮬레이션 하는 작업을 담당한다.

따라서 *SCRAM driver* 상위에 파일 시스템과 메모리 관리 사용자 모듈이 동작하는 경우 각각의 사용자 모듈에 의해 관리되는 파일 객체와 메모리 객체가 하나의 SCRAM 내에 동시에 존재할 수 있게 된다. 이러한 구조는 기존의 시스템에서 주 기억 장치와 보조 기억 장치에 분리되어 저장되던 메모리 객체와 파일 객체를 단일 객체로 취급할 수 있는 새로운 구조를 가능케 한다. 이러한 새로운 가능성을 실제 구현한 것이 SCRAM manager이다. 그림 2는 *SCRAM manager*와 기존의 저장 장치 관리자간의 차이를 개념적으로 보여준다.

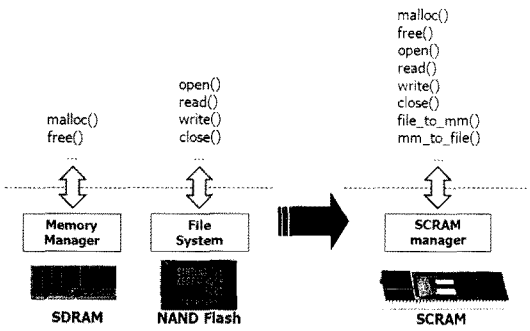



그림 2 SCRAM manager의 인터페이스

*SCRAM manager*는 open(), read()와 같은 파일 객체 인터페이스와 malloc(), free()와 같은 메모리 객체 인터페이스를 동시에 제공한다. 다시 말해 *SCRAM manager*는 사용자 수준 응용에게 파일 시스템인 동시에 메모리 관리자인 것이다. *SCRAM manager*는 메모리 객체와 파일 객체를 단일한 방식으로 관리하며, 유일한 차이점은 파일 객체는 이름 정보를 포함한 부가적인 메타 데이터가 존재한다는 점이다.

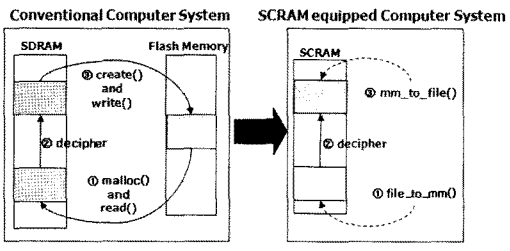
이러한 단일 관리 기법은 mm_to_file(), file_to_mm()과 같은 기존의 운영체제에서 제공하지 못했던 새로운 인터페이스의 제공을 가능케 한다. mm_to_file()은 실제 복사 작업 없이, 이름 등의 메타데이터만을 추가함으로써 메모리 객체를 파일 객체로 변환하는 인터페이스이며, file_to_mm()은 그 반대의 작업을 수행한다.

이 두 가지 새로운 인터페이스는 사용자 수준 응용 프로그램과 운영체제 내부에서 성능 향상을 위해 다양하게 사용될 수 있다. 암호화 되어 있는 파일을 복호화 한 뒤 이를 파일로 저장하는 프로그램을 예로 들어 보자. 기존의 구조에서는 총 두 번의 파일 접근(그림 3(a)의 좌측 편의 ①과 ③)과 한 번의 메모리 접근(그림 3(a)의 좌측 편에서 ②)이 필요한 반면, SCRAM이 장착된 시스템에 *SCRAM manager*를 도입한 경우 두 번의 파일 접근은 그림 3(a)의 우측 편에 나타낸 바와 같이 매우 적은 비용을 수반하는 mm_to_file()과 file_to_mm() 인터페이스로 대체 가능하다.

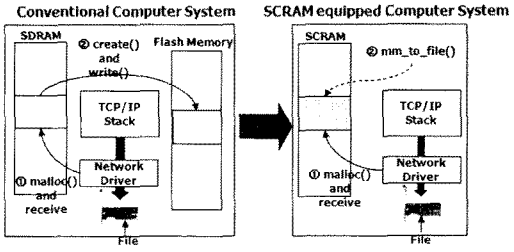
네트워크로부터 수신된 데이터를 파일로 저장하는 또 다른 예를 그림 3(b)에 나타내었다. 그림 3(b)의 좌측 편에 나타낸 기존 구조의 시스템에서, 네트워크로 수신된 데이터는 우선 메모리에 저장되며, 추후 다시 파일로 저장되어야 한다. 그러나 *SCRAM manager*를 도입한 시스템에서는 그림 3(b)의 우측 편에 나타낸 바와 같이, 네트워크로 수신되어 SCRAM에 저장된 데이터는 별도의 복사 없이 mm_to_file() 인터페이스를 통해 파일 객체로 저장될 수 있다.

## 4. 구현 내용

이번 장에서는 동적으로 적재 및 제거 가능한 모듈 형태로 리눅스 운영체제에 구현된 *SCRAM driver*와 *SCRAM manager*의 실제 구현에 대해 소개한다.




(a) 파일 복호화 작업의 예




(b) 파일 수신의 예

그림 3 새로운 인터페이스의 효율성

### 4.1 SCRAM driver

그림 4는 *SCRAM driver*의 기본적인 동작을 보여준다. 모듈을 적재하는 "insmod" 명령을 사용하여 우선 'scram.ko'라는 이름의 *SCRAM driver*를 커널에 적재한다. 적재되는 시점에 *SCRAM driver*는 '/dev/scram', '/dev/scramblock', '/dev/scramblock[no]' 등의 특수 장치 파일을 자동으로 생성한다. 이 파일들은 *SCRAM driver*가 제공하는 함수를 호출하기 위한 진입점(entry point)이 된다. 이때 '/dev/scram' 파일은 SCRAM의 분할에 대한 설정을 제공하는 문자(character) 장치 파일이며, '/dev/scramblock'과 '/dev/scramblock[no]'은 기존의 전통적인 파일 시스템이 사용할 수 있는 블록(block) 장치 파일이다.

*SCRAM driver* 모듈을 적재한 뒤, '/dev/scram' 파일을 이용하여 저수준 분할을 하였다. 그림 4의 예에서는 "FS=16, MM=0, SCRAM=16"으로 지정하였는데 이는 총 32MB의 SCRAM을 분할하여 이 중 16MB를 파일 시스템이 블록 장치파일을 통해 접근 할 수 있도록 제공하며, 나머지 16MB는 *SCRAM manager*가 사용할 수 있도록 함을 의미한다. 그런 뒤 'fdisk' 명령을 통해 할당 된 16MB의 파일 시스템 영역을 파티셔닝(partitioning)하였다. 이때 각각의 파티션은 '/dev/scramblock[no]'를 통해 접근 가능하다. 즉, 리눅스에서 사용가능한 어떠한 파일 시스템도 SCRAM 상에 구축하여 사용하는 것이 가능함을 의미한다. 그림 4의 예에서는 'mkdosfs' 명령을 통해 '/dev/scramblock1' 파티션에 FAT 파일 시스템을 구축하고 사용하는 예를 보이고 있다.

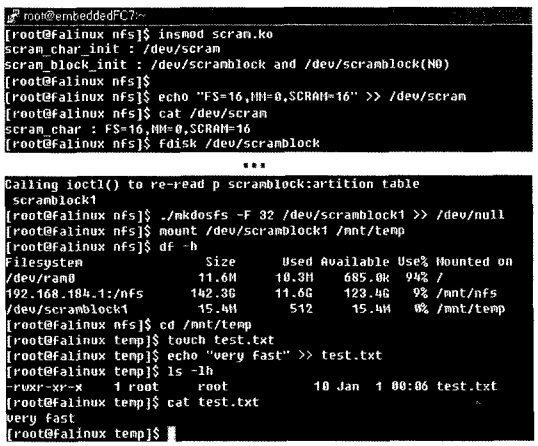



그림 4 SCRAM driver의 동작

### 4.2 SCRAM manager

그림 5는 *SCRAM manager*의 내부구조를 보여준다. *SCRAM manager*는 다시 세 개의 sub-manager로 나뉜다. 이들은 각각 *allocation manager*, *metadata manager*, *conversion manager*이다. *Allocation manager*는 기존의 메모리 관리 기법들과 마찬가지로 *SCRAM driver*로부터 분할받은 공간(그림 4의 예에서는 16MB)에 대한 할당/해제를 담당하며, 상위 계층으로부터의 다양한 크기의 요청에 대해 malloc(), free()와 같은 메모리 객체 인터페이스를 제공한다. 현재는 *allocation manager*를 구현하기 위해 버디 기법을 적용하였으며 32B부터 128KB까지의 객체 크기를 지원한다.

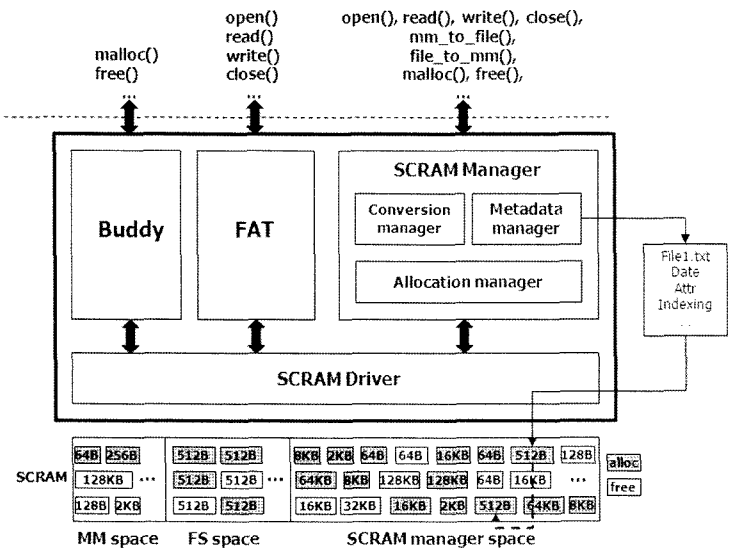



그림 5 SCRAM manager의 구조

*Metadata manager*는 파일의 이름, 속성, 데이터 블록에 대한 인덱싱 정보 등과 같은 파일의 메타데이터를 관리하며, open(), write()와 같은 파일 객체 인터페이스를 제공한다. *Metadata manager*는 filedata_object와 metadata_object 라는 두 종류의 메모리 객체를 관리한다. 새로운 파일이 생성되는 예를 들면, 파일의 실제 데이터를 담기위한 복수개의 filedata_object와, 메타 데이터를 기록하기 위한 하나의 metadata_object가 *allocation manager*로부터 할당된다. 파일의 이름과 FAT 테이블을 이용한 인덱싱 정보 등 필수적인 정보를 담고 있는 metadata_object는 현재 구현에서 64B로 고정되어 있다. 반면 filedata_object의 크기는 임의로 설정가능하며 실험에서는 FAT 파일 시스템과의 공정한 성능 비교를 위해 512B로 설정하였다.

본 논문은 전통적인 파일 시스템과 메모리 관리자가 SCRAM 상에서 동작할 때와 본 논문에서 제안한 단일한 관리 구조의 *SCRAM manager* 간의 공정한 성능 비교에 초점을 맞추고 있다. 따라서 비교 대상이 되는 FAT 파일 시스템과 버디 할당자와 가장 유사한 기법을 적용하여 SCRAM manager를 구현하였다. 이것이 부가적인 영향이 최소화된, 구조적인 차이를 보이기에 적합하다고 판단되었기 때문이다. 한편 추후 연구를 통해 본 연구진이 *SCRAM manager*의 구현에 도입한 FAT 구조와 버디 기법 외에 *SCRAM manager*에 적합한 관리 기법을 찾아낼 것이다.

마지막으로 *conversion manager*는 mm_to_file()과 file_to_mm()이라는 새로운 인터페이스를 제공한다. 전통적인 운영체제에서 객체의 변환은 주 기억 장치와 보조 기억 장치간의 실제 복사 작업을 필요로 했다. 그러나 *conversion manager*가 제공하는 이 두 가지 인터페이스를 이용하여 파일 객체와 메모리 객체는 실제 복사 작업 없이 metadata_object의 가감을 통해 즉각 전환하는 것이 가능하게 된다. 이 두 가지 새로운 인터페이스의 구체적인 내용을 표 1에 보였다.

mm_to_file() 함수는 세 가지 인자를 받는다. 첫째는 메모리 객체를 가리키는 포인터이고, 둘째는 파일 이름, 셋째는 O_RD나 O_WR과 같은 플래그이다. 이 함수는 추후 read(), write() 시에 사용될 수 있는 파일 디스크립터를 반환한다. file_to_mm() 함수는 두 가지 인자를 받는다. 첫째는 파일 디스크립터이고 둘째는 플래그이다. 이때 플래그는 해당 파일의 metadata_object를 삭제할 것인지 아닌지를 나타낸다. 예를 들어 플래그가 설정되어 있다면, 해당 파일의 metadata_object는 file_to_mm() 함수 호출 이후 삭제된다. 따라서 반환된 포인터를 free() 하게 되면 파일과 관련된 모든 데이터가 삭제되는 것을 의미한다. 반면 설정되어 있지 않다면 파일과 관련된 데이터는 free() 이후에도 남아있게 된다.

그림 6은 표 1에 나타낸 새로운 인터페이스가 동작원리를 나타낸다. 우선 그림 6의 ①은 2KB의 메모리 객체를 인자로 mm_to_file()함수가 호출된 것을 나타낸다. 그런 뒤 그림 6의 ②에 보인바와 같이 conversion manager는 metadata manager를 통해 metadata_object를 구축하고, 함수의 인자로 넘어온 파일 이름을 기록한다. 끝으로 그림 6의 ③에 나타낸바와 같이 인자로 넘어

표 1 Conversion Manager가 제공하는 새로운 인터페이스

| |
|---|
| *int file_descriptor = mm_to_file(void *memory_pointer, char * file_name, int flags);* |
| *void * memory_pointer = file_to_mm(int file_descriptor, int flag);* |

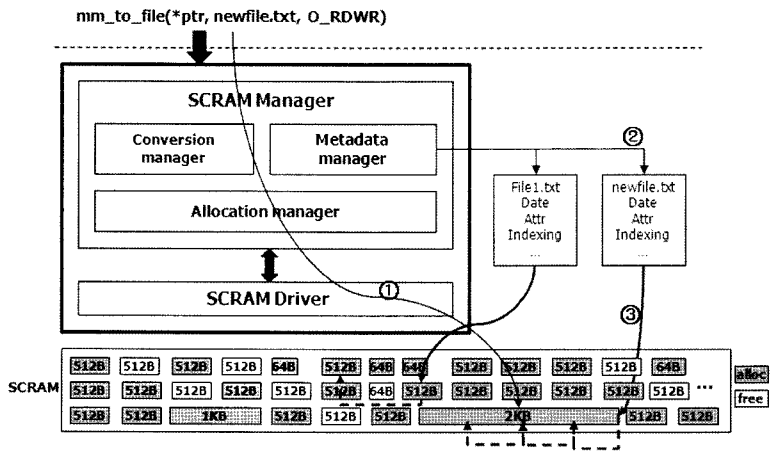



그림 6 객체 이동

온 2KB의 메모리 객체 공간을 가상적으로 512B 단위로 분할하여 FAT 테이블을 통해 연결시킨다.

이때 고려해야 할 사항은 크게 두 가지로 나뉜다. 첫 번째 고려사항은 mm_to_file()함수의 인자로 filedata_object 보다 적은 크기의 메모리 객체를 가리키는 포인터가 넘어온 경우이다. 이 경우 두 가지 기법을 적용하는 것이 가능하다. 첫째는 익스턴트 기법[19]을 활용하는 것이고, 둘째는 filedata_object를 새로 할당받고 이곳에 데이터를 실제 복사하는 방법이다. 본 연구진은 익스턴트 기법을 도입하여 실제 구현을 해 보았으나, 실험결과 심각한 오버헤드를 초래함을 확인할 수 있었다. 이에 따라 데이터가 추가 될 때까지 데이터의 복사를 미루는 변형된 복사 기법을 도입하였다. 구체적으로 metadata_object가 직접 메모리 객체를 가리킬 수 있도록 하였으며 이때 가리키고 있는 메모리 객체의 크기 정보를 기록해 두었다. 추후 새로운 데이터가 추가로 기록되면 새로운 filedata_object를 할당하여 이곳에 기존 데이터를 복사 후 요청된 데이터를 기록하였다. 이때 최대 복사량은 항상 filedata_object의 크기보다 적기 때문에 오버헤드는 매우 미미하였다. 두 번째 고려사항은 단일 매체에 메모리 객체와 파일 객체가 혼재함에 따라 객체의 보호(protection) 및 보안(security) 관련 문제점이 야기될 우려가 있다는 것이다. 예를 들어 할당된 메모리 객체에 대한 포인터를 이용하여 사용자 응용 프로그램이 잘못된 접근을 하면 최악의 경우 파일 객체를 찾기 위한 메타 데이터의 손상 등이 유발될 수 있다. 본 연구진은 추후 연구를 통해 보다 신뢰성을 보장할 수 있는 기법을 개발해 나갈 것이다.

## 5. 실험 결과

제안된 *SCRAM driver*와 *SCRAM manager*는 400 MHz XScale CPU와 64MB SDRAM, 64MB NAND Flash, 그리고 UART, LCD등의 주변 장치가 장착되어 있는 실제 시스템에서 실험되었다. 구체적인 하드웨어 제원은 표 2와 같다. 또한 32MB의 FeRAM이 장착된 daughter board를 제작하였으며 이를 그림 7에 보였다. 이 하드웨어 상에 리눅스 커널 2.6.21이 포팅되어 동작

표 2 하드웨어 제원

| 장 치 | 제 원 |
|---|---|
| CPU | PXA 255 400MHz |
| SDRAM | 32MB(K4S281632C)*2ea |
| NAND Flash | 32M(K9K1208U0A)*2ea |
| Boot Flash | 512KB(MX29LV400T/B) |
| SCRAM(FeRAM) | 0.5MB(FM22L16)*64ea |
| Peripherals | UART, LCD, JTAG, ... |

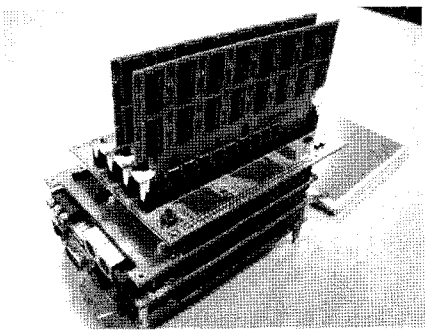


그림 7 SCRAM 실험 환경

되며, *SCRAM driver*와 *SCRAM manager*는 리눅스 커널에 동적으로 적재 가능한 모듈로 구현되었다. 또한 비교를 위해 FAT 파일 시스템과 버디 할당자를 *SCRAM driver*위에서 수행시켰다.

### 5.1 마이크로 벤치마크 수행 결과

표 3은 파일 시스템과 연관된 연산의 마이크로 벤치마크 결과를 보여준다. 이 실험에서는 creat(), open(), write(), read(), close()의 5개 함수에 대해 성능을 측정하였다. write()와 read()의 경우 괄호 내의 숫자는 각각의 연산을 수행한 바이트 수를 나타내며, 캐시의 영향을 최소화시키기 위해 각각의 수행 이후에는 시스템을 재부팅한 뒤 측정하였다. 표에 나타난 모든 결과 값은 각각의 함수를 10회 수행시킨 뒤 얻은 평균값이다. 실험의 비교 대상은 다음과 같다. (1) NAND 플래시 메모리위해 MTD의 NFTL을 동작시킨 뒤, FAT 파일 시스템을 구축하였고, (2) SCRAM상에 SCRAM driver를 사용하여 FAT 파일 시스템을 구축하였으며, (3) SCRAM 상에 SCRAM driver를 사용하여 SCRAM manager를 동작시켰다.

실험 결과에서 (1)과 (2)의 성능 차이는 NAND 플래시 메모리와 SCRAM의 저장 장치 특성에 의한 성능 차이이며, 이로 인해 수십 배의 성능차이를 보임을 알 수 있다. 특히 플래시 메모리의 덮어 쓰기 제약으로 인해 쓰기 단위가 커질수록 성능 차이도 급격히 커짐을 알 수 있다. (2)와 (3)의 결과 역시 수십 배의 성능 차이를 보이고 있다. 기존 파일 시스템의 최소 두 번의 복사-저장장치에서 버퍼로, 버퍼에서 사용자 공간으로-가 유발되는 반면 *SCRAM manager*는 이러한 복사를 제거한 것이 가장 주된 성능 향상의 요인으로 분석된다.

표 4는 메모리 관리자와 연관된 마이크로 벤치마크 결과를 보여준다. 이 실험에서는 malloc(), free() 두 개의 함수에 대해 성능을 측정하였고, 할당 받은 공간에 대해 기록하고 읽어오는데 걸리는 속도 역시 측정해 보았다. 표 4에 나타낸 결과 역시 10회 수행 된 값의 평

표 3 파일 시스템 관련 마이크로 벤치마크(단위: ms)

| | (1) NAND+ NFTL+ FAT | (2) SCRAM+ SCRAM driver+ FAT | (3) SCRAM+ SCRAM driver+ SCRAM manager |
|---|---|---|---|
| creat | 1837.6 | 325.9 | 135.5 |
| open | 74.2 | 74 | 29.5 |
| write(512) | 5737.8 | 225.9 | 12.8 |
| write(1024) | 17175.9 | 270.6 | 16.4 |
| write(4096) | 256936.9 | 825 | 43.5 |
| read(512) | 4685.7 | 251.5 | 11.9 |
| read(1024) | 4679.1 | 245.3 | 16.7 |
| read(4096) | 4698.4 | 299.7 | 51.4 |
| close | 13.1 | 7.5 | 5.6 |

표 4 메모리 관리자 관련 마이크로 벤치마크(단위: ms)

| | (1) SDRAM + Linux' Buddy | (2) SCRAM + SCRAM driver + SCRAM manager |
|---|---|---|
| malloc | 34.3 | 10.4 |
| free | 6.3 | 11 |
| load(512) | 8.9 | 4.2 |
| load(1024) | 9.4 | 4.7 |
| load(4096) | 32.2 | 10.4 |
| store(512) | 8.4 | 6.4 |
| store(1024) | 9.1 | 5.9 |
| store(4096) | 36.9 | 18.8 |

균치이며 괄호안의 숫자는 연산이 수행된 데이터의 바이트 수를 의미한다. 실험 대상으로는 (1)SDRAM 상에 리눅스의 버디 할당자를 동작시킨 것과 (2)SCRAM 상에 SCRAM driver를 사용하여 SCRAM manager를 동작시킨 뒤 비교해 보았다.

이때 (1)과 (2)는 동일한 버디 알고리즘에 기반하고 있기 때문에 비교적 성능 향상을 시도할 기회가 적다. 따라서 표 4에 기록된 결과는 표 3과 달리 대부분 비슷한 수준의 성능을 보이고 있다. 단, free()함수의 경우 (1)이 보다 좋은 성능을 보이는데 이는 리눅스의 버디 할당자는 free()되는 메모리 객체의 실제 병합 과정을 뒤로 미루는 Lazy Buddy 기법이 도입되어 있기 때문이다[8].

### 5.2 매크로 벤치마크 수행 결과

표 5는 매크로 벤치마크의 수행 결과를 보여준다. MAB는 수정된 Andrew 벤치마크 프로그램[20]을 의미한다. 구체적으로, 실험이 진행된 시스템에서 Andrew 벤치마크 프로그램의 정상적인 수행이 가능하도록 기존 벤치마크 단계 중 'Make' 단계를 제거하고 수행시켰다. Postmark[21]는 대용량 메일 서버의 동작을 모델링 하는 벤치마크 프로그램으로써 파일 시스템의 성능을 측정하기 위해 널리 사용되고 있다. 이 프로그램은 설정된

표 5 매크로 벤치마크(단위: s)

| | (1) NAND+ NFTL+ FAT | (2) SCRAM+ SCRAM driver+ FAT | (3) SCRAM+ SCRAM driver+ SCRAM manager |
|---|---|---|---|
| MAB | 112 | 6 | 5 |
| Postmark Sync (default) | 626 | 7 | 2 |
| Postmark Async (default) | 5 | 5 | 2 |

크기 범위의 복수개 파일과 디렉터리를 생성 한 뒤, 파일의 생성, 삭제, 추가, 읽기의 연산을 수행한다. 이와 관련하여 다양한 인자를 설정하는 것이 가능한데 본 실험에서는 Postmark의 기본 설정(500개의 파일, 500번의 연산, 500B~10K의 파일 크기)을 사용하였으며, 동기(Sync) 입출력과 비동기(Async) 입출력 실험 결과를 모두 기록하였다.

표 5에서 (1)과 (2)간의 성능차이는 저장장치의 변화로 인한 것임을 알 수 있다. 한편 (2)와 (3)간에는 미미한 성능차이만이 존재한다. 이는 SCRAM manager를 사용함으로써 얻을 수 있는 성능향상 즉, 단일 객체의 개념을 기존 벤치마크 프로그램들이 사용하고 있지 않기 때문이다.

표 6은 실제 응용프로그램을 수행시켜 그 속도를 측정한 본 논문의 마지막 실험결과를 보여준다. 각 결과는 모두 10회 반복 수행 후 그 평균값을 기록하였다. 실험에 사용된 프로그램은 Source Forge[22]에서 얻을 수 있는 'distillery', 'ftpget'이라는 프로그램과 본 연구진이 실험을 위해 제작한 합성 워크로드이다. 구체적으로 첫 번째 실험 프로그램인 'distillery'는 파일 압축 프로그램으로써, 실험에서는 다양한 크기의 파일을 압축한 뒤 512B를 압축하는데 필요한 평균시간을 기록하였다. 두 번째 프로그램은 ftp를 통해 파일을 전송 받은 뒤 이를 파일로 저장하는 프로그램으로써 역시 512B를 전송받아 저장하기 위해 필요한 평균시간을 기록하였다. 세 번째 프로그램은 malloc()함수를 통해 할당받은 공간에 데이터를 기록하고 이를 파일로 저장하는 작업을 반복 수행하며 512B를 작업하기위해 필요한 평균 시간을 나타내었다.

표 6의 (3)실험에서, 새로이 제공되는 mm_to_file()과 file_to_mm() 인터페이스의 효율을 알기위해 필요시 이들 함수를 호출하도록 소스를 수정한 뒤 측정하였다. 실험 결과를 통해 우리는 *SCRAM manager*가 새로이 제공되는 mm_to_file()과 file_to_mm()을 사용하는 경우 매우 높은 성능을 제공할 수 있음을 확인하였다. 이는 본 논문에서 제안한 메모리 객체와 파일 객체의 통합

표 6 실제 응용 프로그램 수행(단위: ms)

| | (1) (SDRAM+Linux'Buddy) + (NAND+NFTL+FAT) | (2) (SCRAM+SCRAM driver+SCRAM manager) + (SCRAM+SCRAM driver+FAT) | (3) SCRAM+ SCRAM driver+ SCRAM manager |
|---|---|---|---|
| compression | 110280 | 10736 | 7072 |
| ftpget | 8944.7 | 1852.1 | 175.1 |
| synth | $47.83 * 10^6$ | $5.09 * 10^6$ | $1.01 * 10^6$ |

관리 기법이 SCRAM의 두 가지 특성을 모두 활용할 수 있는 적합한 구조임을 증명한다.

## 6. 결 론

본 논문에서는 SCRAM의 두 가지 특성을 동시에 활용하기 위한 새로운 소프트웨어 계층 구조를 제안하였다. 제안된 소프트웨어 구조는 구체적으로 *SCRAM driver*와 *SCRAM manager*로 구분되며, *SCRAM manager*는 다시 *allocation manager, metadata manager, conversion manager*로 나뉜다. 특히 제안된 새로운 소프트웨어 계층 구조는 mm_to_file()과 file_to_mm()이라는 새로운 인터페이스를 제공한다. 이 인터페이스는 부가적인 오버헤드 없이 메모리 객체와 파일 객체간의 변환을 가능케 한다. 제안된 구조는 실제 SCRAM이 장착된 시스템에서 실제 구현되었으며, 실험을 통해 제안된 소프트웨어 구조가 획기적인 수준의 성능향상을 가져올 수 있음을 보였다.

추후 본 연구는 크게 세 가지 방향으로 확장해 나갈 것이다. 첫째는 FAT이나 버디 기법과 구분되는 *SCRAM manager*에 최적화된 관리 기법의 제안이다. 두 번째는 객체 이동시 발생할 수 있는 잠재적인 문제점을 해결하기 위해 SCRAM의 신뢰성을 보장할 수 있는 기법을 개발하는 것이며, 세 번째는 SCRAM의 특성을 고려한 새로운 운영체제를 구현하는 것이다.

## 참 고 문 헌


[1] G. B. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy, "Overview of Candidate Device Technologies for Storage-Class Memory", IBM Journal of Research and Development, 52(4):449-464, 2008.

[2] A. K. Sharma, "Advanced semiconductor Memories: Architectures, Designs, and Applications," Wiley Interscience, 2003.

[3] Ramtron's FeRAM-equipped systems, "http://www.ramtron.com/applications/computing.aspx"

[4] Freescale's MRAM Technology, "http://www.freescale.com"

[5] M. Baker, S. Asami, E. Deprit, J. Ouseterhout, and M. Seltzer, "Non-Volatile Memory for Fast, Reliable File Systems," In Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 10-22, 1992.

[6] M. Baker and M. Sullivan, "The Recovery Box: Using Fast Recovery to Provide High Availability in the Unix Environment," In USENIX Summer Technical Conference, pp. 31-44, 1992.

[7] W. Stallings, "Operating Systems: Internals and Design Principles," Prentice Hall, 5th Edition, 2007.

[8] D. Bovet, and M. Cesati, "Understanding the Linux Kernel." O'Reilly, 3rd Edition, 2007.

[9] N. K. Edel, D. Tuteja, E. L. Miller, and S. A. Brandt, "MRAMFS: A Compressing File System for Non-Volatile RAM, "In Proceedings of the 12th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, pp. 596-603, 2004.

[10] PRAMFS, "http://pramfs.sourceforge.net"

[11] S. Baek, C. Hyun, J. Choi, D. Lee, and S. H. Noh, "Design and Analysis of a Space Conscious Nonvolatile-RAM File System," In Proceedings of the IEEE TENCON, 2006.

[12] E. L. Miller, S. A. Brandt, and D. D. E. Long, "HeRMES: High-Performance Reliable MRAM-Enabled Storage," In Proceedings of the 8th HotOS, pp. 95-99, 2001.

[13] A. I. A. Wang, G. Kuenning, P. Reiher, and G. Popek, "The Conquest File System: Better Performance Through a Disk/Persistent-RAM Hybrid Design", ACM Transactions on Storage, 2(3):309-348, 2006.

[14] I. H. Doh, J. Choi, D. Lee, And S. H. Noh, "Exploiting Non-Volatile RAM to Enhance Flash File System Performance," In Proceedings of the 7th ACM & IEEE International Conference on Embedded Software, pp. 164-173, 2007.

[15] S. Akyurek and K. Salem, "Management of Partially Safe Buffers," IEEE Transactions on Computers, 44(3):394-407, 1995.

[16] T. R. Haining and D. D. E Long, "Management Policies for Non-Volatile Write Caches," In Proceedings of the IEEE International Conference on Computing and Communications Performance, pp. 321-328, 1999.

[17] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell, "The Rio File Cache:

Surviving Operating System Crashes," In Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systmes, pp. 74-83, 1996.
[18] M. Wu, and W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Storage System," In Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System, 1994.
[19] L. W. Mcvoy and S. R. Klieiman, "Extent-like Performance for a UNIX File System," In USENIX Winter Technical Conference, 1991.
[20] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and Performance in a distributed file system," ACM Transactions on Computer Systems, 6(1), 1998.
[21] J. Katcher, "Postmark: A New Filesystem Benchmark," Technical Report TR3022, Network Appliance, 1997.
[22] Source Forge, "http://www.sourceforge.net"

백 승 재

2005년 단국대학교 컴퓨터공학과 졸업(공학사). 2004년~현재 비트 컴퓨터 강사. 2007년 단국대학교 대학원 정보컴퓨터 과학과(이학석사). 2007년~현재 단국대학교 대학원 컴퓨터학과 박사과정. 관심분야는 운영체제, 임베디드 시스템, 차세대 저장장치 등

최 종 무

1993년 서울대학교 해양학과 졸업(이학사). 1995년 서울대학교 대학원 컴퓨터공학과(공학석사). 2001년 서울대학교 대학원 컴퓨터 공학과(공학박사). 2001년~2003년 유비쿼스 주식회사 책임 연구원 2003년~현재 단국대학교 공과대학 컴퓨터학부 컴퓨터공학 전공 조교수. 2005년~2006년 UC Santa Cruz 방문 교수. 관심분야는 운영체제, 임베디드 시스템, 차세대 저장장치 등