SCRAM을 위한 운영체제 구현 연구

단국대학교 ▌백승재* · 최종무**

1. 서 론

컴퓨터 시스템의 성능향상을 위해 지난 수십 년간 진행된 수많은 연구의 결과로, 컴퓨터 시스템의 메모리는 계층 구조(Memory Hierarchy)를 택하고 있다[1]. 계층 구조상에 존재하는 다양한 종류의 메모리들은 장치 자체가 가지고 있는 특성에 따라 최소의 비용을 통해 최적의 시스템 성능을 얻기 위하여 사용된다. 기존 컴퓨터 시스템의 메모리 계층 구조는 접근 단위와 비휘발성 여부에 따라 둘로 구분할 수 있다. 첫째는 일반적으로 램(RAM)이라 불리는 메모리로써 바이트(Byte) 단위의 임의 접근이 가능하며 접근속도가 빠르고 휘발성이라는 특징을 가진다. 이에 속하는 메모리로는 캐시(cache) 용도의 SRAM, 주 기억 장치 용도의 DRAM 등이 있다. 두 번째는 스토리지(Storage)로써, 블록 단위로 접근되며 비교적 접근 속도가 느리고 비휘발성 이라는 특징을 가진다. 보조 기억장치 용도의 하드디스크(Hard Disk)나 플래시 메모리(Flash memory) 등이 이에 속한다.

최근 이러한 전통적인 메모리 외에, 바이트 단위 접근 및 빠른 접근속도라는 램의 장점과 비휘발성이라는 스토리지의 장점 모두를 가지는 스토리지 클래스 램(Storage Class RAM)이 차세대 메모리로써 각광받고 있다[2,3]. 구체적인 스토리지 클래스 램의 예로는 FeRAM(Ferro-electric RAM), MRAM(Magneto-resistive RAM), PRAM(Phase-change RAM) 등이 있으며[4], 인텔(Intel), 삼성전자, 도시바(Toshiba), 프리스케일(freescale), 후지쯔(Fujitsu), 램트론(Ramtron), IBM 등 수많은 국내·외 대기업에서 이미 제품을 시판 했거나 시판 예정에 있다. 또한 2009년 2월 도시바는 128 Mbit의 용량과 1.6GB/s의 속도를 내는 DDR2 인터페이스 호환 FeRAM을 발표하였으며[5], 삼성전자는 현재 1Gbit인 자사 PRAM의 용량을 2011년까지 8Gbit로 증가시킬 계획임을 발표하였다[6]. 이는 스토리지 클래스 램 역시 황의 법칙에 예외가 아니며, 따라서 다양한 장점을 가지고 있는 대용량의 스토리지 클래스 램의 컴퓨터 시스템 적용이 더욱 현실화 되고 있음을 알 수 있다.

스토리지 클래스 램의 도입은 컴퓨터 시스템의 전통적인 메모리 계층 구조, 운영체제, 더 나아가 일반적으로 통용되던 수많은 개념에 변화를 야기한다. SRAM급 접근 속도를 가지는 대용량 스토리지 클래스 램의 도입으로 컴퓨터 시스템의 메모리는 더 이상 계층 구조가 아닌 단일한 구조를 가지게 되는 것이 가능해졌다. '파일은 장소를, 프로세스는 생명을 제공한다'는 운영체제의 기본 개념 역시 스토리지에 저장해야 하는 파일이라는 객체와 램(RAM)에 저장되는 프로세스라는 객체로 구분된 메모리 계층 구조에서 시작된 개념이다.

따라서 기존 운영체제의 램과 스토리지 관리 기법들은 스토리지 클래스 램에 적합한 구조가 아니다. 예를 들어 기존 운영체제에서 램은 슬랩 할당자(Slab Allocator), 버디 시스템(Buddy System) 등과 같은 메모리 관리(Memory Management) 기법을 통해 관리된다. 반면, 디스크는 FAT, Ext2, LFS와 같은 파일 시스템(File System)을 통해 따로 관리된다. 하지만 스토리지 클래스 램에서는 메모리 관리와 파일 시스템이 하나의 소프트웨어 모듈로 통합될 수 있다. 또한 이 통합은 빠른 프로그램의 수행, 파일의 접근, 그리고 저비용의 스레드 상태 전환을 가능하게 한다. 결국 스토리지 클래스 램에서는 프로세스이자 동시에 파일일 수 있는 단일 객체(single object)의 존재가 가능해 지는 것이다.

본 논문에서는 SCRAM을 활용함으로써 전통적인 컴퓨터 시스템의 저장장치 계층 구조에서는 이룰 수 없었던 수준의 성능과 신뢰성을 갖추기 위한 다양한 시도와 접근 방법에 대해 소개한다. 또한 스토리지 클래

* 정회원
** 종신회원
† 이 논문은 2008년도 정부(교육과학기술부)의 재원으로 한국과학재단의 지원을 받아 수행된 연구임(No. R01-2008-000-12028-0).

스 램이 가지고 있는 두 가지 특성을 통합하여 활용하기 위한 새로운 소프트웨어 계층 구조와 단일 객체의 개념, 이를 활용하여 '부팅(Booting)/사용(Running)/종료(Terminating)'가 아닌 '사용/비사용'이라는 개념만이 존재하는 이른바 영속적인 컴퓨팅(Permanent Computing) 환경을 제공하기 위한 기법에 대해 살펴본다.

2. 스토리지 클래스 램을 활용한 연구

2.1 스토리지 클래스 램 활용 시스템

스토리지 클래스 램의 급격한 기술 발전으로 인해 이를 컴퓨터 시스템에 도입하기 위한 많은 시도가 활발히 진행되고 있다. 구체적으로 램트론에서는 FeRAM을 도입한 8051 마이크로 컨트롤러를, 후지쯔에서는 FeRAM을 시스템의 유일한 메모리로 사용하는 RFID를 출시하였다. 또한 다양한 센서를 통해 정보를 수집하고 이를 안정적으로 저장해야 하는 자동차 시스템에서 스토리지 클래스 램 도입 역시 활발하게 이루어지고 있다. 램트론에서는 차량의 각 모듈에 FeRAM을 도입하는 연구를 진행 중이다.

램트론과 후지쯔가 센서 노드와 RFID 등에 스토리지 클래스 램을 도입한 것처럼, 삼성과 인텔에서도 휴대폰에 스토리지 클래스 램을 도입할 것을 발표하였다[7,8]. 기존 휴대폰의 아키텍처는 다양한 메모리들을 포함하고 있다. 예를 들면 부트 로더와 압축된 커널을 위한 NOR 유형 플래시 메모리, 파일과 바이너리를 위한 NAND 유형 플래시 메모리, 프로그램의 동적 수행을 위한 SDRAM 등으로 구성된다. 이러한 다양한 메모리들의 구성은 휴대폰 가격을 높이며, 동시에 휴대폰의 크기를 증가시킨다. 이에 따라 삼성, 인텔 등에서는 스토리지 클래스 램만으로 구성된 휴대폰을 연구하고 있다. 스토리지 클래스 램 관련 기술의 발전으로 그 용량이 점차 대용량화 되어 가고 있는 현실을 고려하면 스토리지 클래스 램이 점진적으로 컴퓨터 시스템의 메모리 계층구조에 도입될 것이며, 이에 따라 스토리지 클래스 램만으로 구축된 컴퓨터 시스템이 점차 현실화 될 것이다.

현재, 스토리지 클래스 램을 장착한 이러한 임베디드 시스템들은 스토리지 클래스 램을 램의 확장 공간으로만, 혹은 스토리지의 확장 공간으로만 이용하고 있다. 따라서 스토리지 클래스 램의 양면성을 활용할 수 있는 새로운 관리 소프트웨어의 필요성이 증가되고 있다.

2.2 램 특성 활용 연구

스토리지 클래스 램을 램의 확장 공간으로 활용한 대표적인 연구로는 [9, 10, 11, 12, 13]과 같은 것이 있다. [9]의 연구에서는 스토리지 클래스 램을 쓰기 캐시(write cache)로 사용하여 시스템이 붕괴 되었을 때 데이터의 일관성을 유지하기 위한 복구를 효율적으로 수행할 수 있음을 보였고, [10, 11, 12]에서는 시스템에 휘발성 메모리와 비휘발성 메모리가 동시에 존재하는 경우 캐시 관리 정책에 대한 연구를 수행하였으며, [13]의 연구에서는 분산 파일 시스템에서 스토리지 클래스 램을 쓰기 캐시로 사용하는 상황에 대한 연구를 진행하여 클라이언트에서 서버로 요청되는 쓰기 트래픽을 효과적으로 줄일 수 있으며 서버에서 디스크 접근을 효율적으로 관리할 수 있음을 보인 바 있다. 그러나 이들 연구는 모두 스토리지 클래스 램의 스토리지 특성을 활용하지 못하고 있다.

2.3 스토리지 특성 활용 연구

스토리지 클래스 램의 스토리지 특성 활용 방안에 대한 연구로는 [14, 15, 16, 17, 18]과 같은 것이 있다. [14]의 연구에서는 저장되는 정보를 압축하여 용량이 작은 스토리지 클래스 램의 공간을 효율적으로 사용하는 방법을 제안 하였고, [15]의 연구에서는 EXT2 파일 시스템에서 하나의 블록그룹을 파일 시스템 전체로 확장한 형태의 스토리지 클래스 램용 파일시스템을 제안하였다. [16]에서는 공간효율성을 높이기 위해 익스텐트(extent) 기반 할당을 도입한 파일시스템을 제안하였고, [17, 18]에서는 디스크와 스토리지 클래스 램을 동시에 사용하는 Hybrid 형태의 파일 시스템을 제안하였다. 한편, [19]에서는 스토리지 클래스 램에 메타데이터를 저장하여 플래시 메모리 파일 시스템의 성능 향상을 시도하였다. 그러나 이들 연구는 모두 스토리지 클래스 램의 램 특성을 활용하지 못하고 있다.

한편 스토리지 클래스 램의 양면성은 영속적인 컴퓨팅에 대한 새로운 가능성을 제시한다. 이러한 영속적인 컴퓨팅을 위한 기존의 연구로는 [20, 21]이 있다. 이러한 연구는 디스크 저장장치를 기반으로 프로세스의 수행 문맥을 저장함으로써 중단 없는 컴퓨팅 환경을 제안하였다. 그러나 디스크 저장장치의 지연(latency)으로 인해 구축이 용이치 않음을 기술한바 있다. 또한 스토리지 클래스 램을 고려하지 않은 디스크 기반의 연구였으므로 이를 스토리지 클래스 램 기반 시스템에 적용하는 경우 스토리지 클래스 램 고유의 장점을 살리지 못한다.

3. SCRAM을 위한 소프트웨어 설계

3.1 SCRAM 관리 소프트웨어 설계

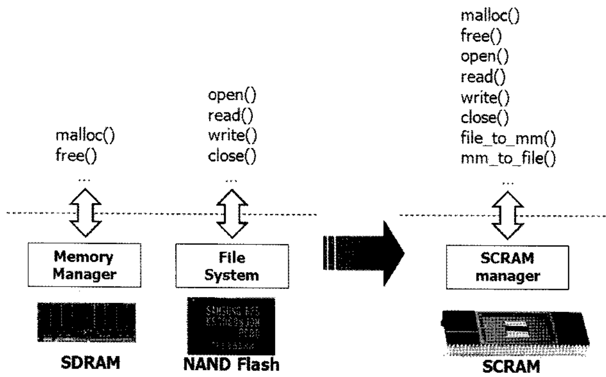


그림 1 SCRAM manager의 인터페이스

스토리지 클래스 램의 내용은 전원 공급이 중단되더라도 그대로 남아있기 때문에 파일 저장 공간으로 이용될 수 있다. 또한 바이트 단위의 접근이 가능하기 때문에 프로그램의 수행 공간으로도 이용될 수 있다. 따라서 스토리지 클래스 램 내에 저장된 객체를 효율적으로 관리하기 위해서는 기존 운영체제에서 보조 기억장치로써의 특성과 주기억장치로써의 특성 모두를 만족시켜줄 수 있는 메커니즘이 필요하다.

기존에는 휘발성 램을 관리하는 메모리 관리자(Buddy, Slab등)와 디스크 등의 저장장치를 관리하는 파일 시스템(Ext2, FAT 등)이 별도로 존재하였다. 그리고 이들은 malloc(), kmalloc()과 같은 메모리 접근 함수와 open(), write()와 같은 파일 접근 함수를 따로 제공해 주었다. 그러나 스토리지 클래스 램은 이 두 가지 기능을 모두 수행할 수 있으며, 이를 관리하기 위한 소프트웨어도 하나도 통합할 수 있다.

그림 1은 SCRAM을 관리하기 위한 통합된 소프트웨어인 SCRAM manager를 개념적으로 보여준다. 기존 시스템에서는 NOR, NAND, SDRAM 등의 독립된 메모리 모듈들을 관리하기 위해 펌웨어, 파일 시스템, 메모리 관리자 등 별도의 소프트웨어들이 필요하였다. 반면 SCRAM manager에서는 이러한 기능이 단일한 하나의 소프트웨어로 통합되어 있다.

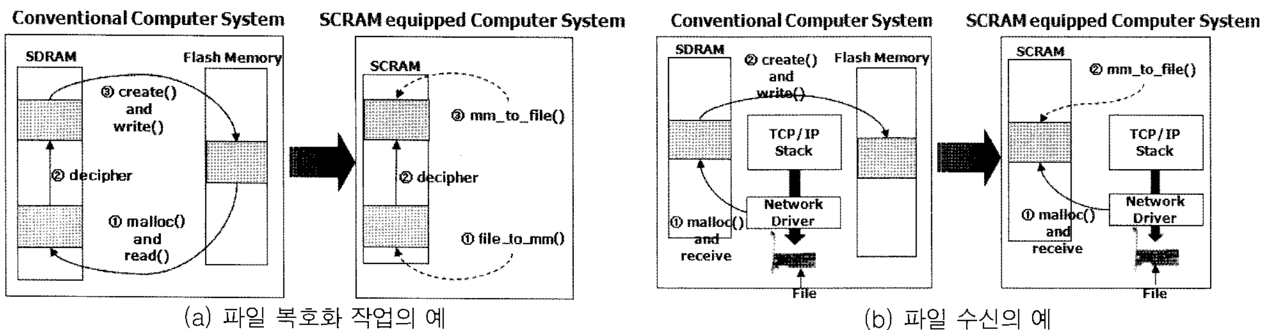
한편, SCRAM manager는 open()되는 파일 객체와 malloc()되는 메모리 객체를 동일한 공간에서 관리하기 때문에 이들 객체 간에 전환이 복사 없이 수행될 수 있다. 즉 mm_to_file(), file_to_mm()과 같은 기존의 운영체제에서 제공하지 못했던 새로운 인터페이스의 제공을 가능케 한다. mm_to_file()은 실제 복사 작업 없이, 이름 등의 메타데이터만을 추가함으로써 메모리 객체를 파일 객체로 변환하는 인터페이스이며, file_to_mm()은 그 반대의 작업을 수행한다.

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 2(a)의 좌측 편의 ①과 ③)과 한 번의 메모리 접근(그림 2(a)의 좌측 편에서 ②)이 필요한 반면, SCRAM이 장착된 시스템에 SCRAM manager를 도입한 경우 두 번의 파일 접근은 그림 2(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 2(b)에 나타내었다. 그림 2(b)의 좌측 편에 나타낸 기존 구조의 시스템에서, 네트워크로 수신된 데이터는 우선 메모리에 저장되며, 추후 다시 파일로 저장되어야 한다. 그러나 SCRAM manager를 도입한 시스템에서는 그림 2(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.

3.2 SCRAM Manager를 활용한 운영체제 설계

기존 시스템에서 실행 파일을 수행시키기 위해서는 우선 실행 파일의 내용을 읽어 메모리로 복사해야 한다. 그런 뒤 태스크에서 메모리 내용을 접근 할 수 있도록 페이지 테이블을 구축하여 메모리 공간과 연결시켜주는 작업이 필요하다. 그러나 SCRAM Manager를 활용한 운영체제에서는 실행 파일의 내용을 태스



(a) 파일 복호화 작업의 예 (b) 파일 수신의 예

그림 2. 새로운 인터페이스의 효율성

크의 페이지 테이블로 바로 연결함으로써 복사시간 등의 부가적인 오버헤드를 없앨 수 있다. 즉, 바이너리 이미지가 그 위치에서 그대로 수행(eXecution In Place, XIP)될 수 있는 것이다. 또한 모든 부팅 관련 코드 역시 스토리지 클래스 램에 저장되어 있고, 그 위치에서 바로 수행될 수 있기 때문에 Instant Booting, 즉 TV처럼 전원인가 즉시 동작하는 새로운 기능을 제공할 수 있다.

이러한 기법을 제공하기 위해서는 다음의 문제를 해결해야 한다. 첫째, 실행 파일의 수행 형식(execution format)이 인스턴트 수행을 지원할 수 있는 형식이어야 한다. 결국 기존의 ELF(Executable Linking Format)가 아닌 이진(binary) 형태를 고려해야 한다. 둘째, 프로그램의 수행 이미지가 큰 경우 처리 방법을 고려해야 한다. 셋째, 프로그램 수행 중에 바이너리 이미지를 수정하려는 경우에 대한 처리 방법이 필요하다.

한편 기존 운영체제에서 프로세스는 램 상에 유지되는 휘발성 객체였으나 SCRAM Manager를 활용한 운영체제의 모든 객체는 스토리지 클래스 램 상에 유지되므로 항상 영속적이라는 속성을 가진다. 즉 프로세스이면서 동시에 파일일 수 있는 단일 객체(single object)로 존재하는 것이다. 즉, 이 객체는 프로그램의 수행 환경 및 흐름을 제공한다는 측면에서 프로세스이며, 동시에 영속성을 제공하는 측면에서 파일인 것이다. 따라서 전원 on/off 이후에도 프로세스의 메모리 문맥(context)이 유지되며, 결국 CPU 문맥과 필요한 일부 주변 장치 문맥만 유지하면 기존 수행 상태를 그대로 복구할 수 있다. CPU 문맥과 장치 문맥은 전원 off 인터럽트 발생시 CPU 레지스터의 내용과 장치의 레지스터 내용을 SCRAM에 저장하는 것으로 유지 가능하다. 결국, 컴퓨터 시스템의 사용 개념을 '부팅(booting)/사용(running)/종료(terminating)'가 아닌 '중지(suspend)/재시작(resume)'으로 변화시키는 즉 영속적인(permanent) 컴퓨팅의 제공이 가능해지는 것이다.

영속적인 컴퓨팅은 효율성뿐만 아니라 전원 결함 복구 측면에서도 장점이 있다. 갑작스러운 전원 결함이 발생하더라도, 그 당시 동작하던 프로세스들의 메모리 문맥은 모두 SCRAM에 유지되어 있다. 따라서 CPU와 장치의 문맥을 유지할 수 있고, SCRAM에 유지된 메모리 문맥과 일관되게 연결할 수 있으면 전원 결함 복구가 가능하다. CPU와 장치의 문맥 유지는 하드웨어적인 접근 방법과 소프트웨어적인 접근 방법 모두 가능하다.

하드웨어적인 접근 방법은 전원 off 인터럽트가 발생했을 때 이를 탐지하고 실제 전원이 off 되는 시점까지 약간의 지연 시간을 제공하는 하드웨어 모듈을 이용한다. 이 하드웨어 모듈은 몇 개의 커패시터(capacitor)와 간단한 로직을 이용하여 제작할 수 있다. 이 지연 시간에 CPU와 필요한 장치의 레지스터 내용을 SCRAM에 저장하며, 이를 통해 전원 결함 복구가 가능하다. 본 연구진의 선행 실험 결과 ARM CPU의 경우 레지스터를 저장하기 위해 필요한 명령어의 개수가 10개 이하이며 몇 nano second 단위의 지연 시간이면 충분한 것으로 파악되었다.

소프트웨어적인 접근 방법은 부가적인 하드웨어 모듈 없이 소프트웨어만으로 전원 결함 복구 기능을 제공하는 것으로, 기본 아이디어는 체크 포인트(checkpoint)를 이용하는 것이다. 즉, 주기적으로 CPU와 장치의 상태를 SCRAM에 저장하고 이를 이용해 전원 결함에서 복구하는 것이다. 이때 문제가 되는 것은 주기적인 체크 포인트가 성능에 영향을 줄 수 있다는 것이다. 이를 극복하기 위해 본 연구진은 문맥 교환(context switch) 시점에서 저장되는 프로세스의 문맥을 체크 포인트로 사용하고 이를 이용하여 전원 결함에서 복구하는 방법을 적용하였다. 실제로, 이미 많은 운영체제에서 문맥 교환 시 CPU 레지스터 내용 등을 메모리에 저장하고 있으며, SCRAM을 사용하면 저장 내용들이 영속적이 되어 전원 결함 이후에도 유지된다. 따라서 전원 결함 이후 마지막 문맥 교환이 발생되었던 시점으로 복구하는 것은 가능하다. 이제 남은 문제는 문맥 교환이 발생된 시점부터 전원 결함이 발생한 시점까지 수정된 SCRAM의 내용을 어떻게 복원(redo 처리)하는가 인데, 이는 쓰기 시 복사(copy on write) 기법을 활용하여 해결 가능하다.

4. 구현 내용

4.1 SCRAM Manager

그림 3은 SCRAM manager의 내부구조를 보여준다. SCRAM manager는 다시 세 개의 sub-manager로 나뉜다. 이들은 각각 allocation manager, metadata manager, conversion manager이다. Allocation manager는 기존의 메모리 관리 기법들과 마찬가지로 SCRAM에 대한 저수준 할당/해제를 담당하며, 상위 계층으로부터의 다양한 크기의 요청에 대해 malloc(), free()와 같은 메모리 객체 인터페이스를 제공한다. 현재는 allocation manager를 구현하기 위해 버디 기법을 적용하였으며 32B부터 128KB까지의 객체 크기를 지원한다.

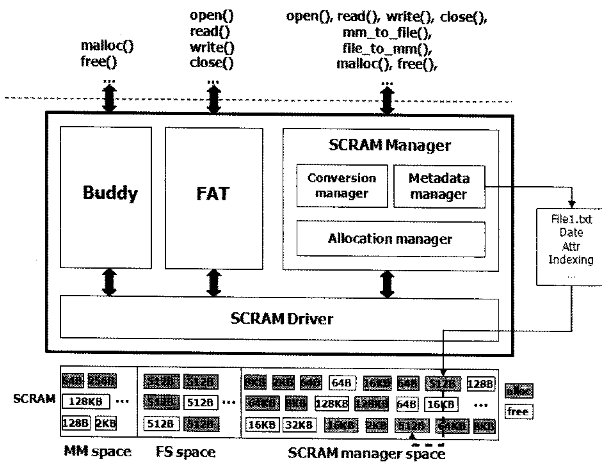


그림 3 SCRAM manager의 구조

Metadata manager는 파일의 이름, 속성, 데이터 블록에 대한 인덱싱 정보 등과 같은 파일의 메타데이터를 관리하며, open(), write()와 같은 파일 객체 인터페이스를 제공한다. Metadata manager는 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 manager로부터 할당된다. 파일의 이름과 FAT 테이블을 이용한 인덱싱 정보 등 필수적인 정보를 담고 있는 metadata_object는 현재 구현에서 64Byte로 고정되어 있으며, filedata_object의 크기는 임의로 설정가능하다.

마지막으로 conversion manager는 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 manager가 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

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);
```

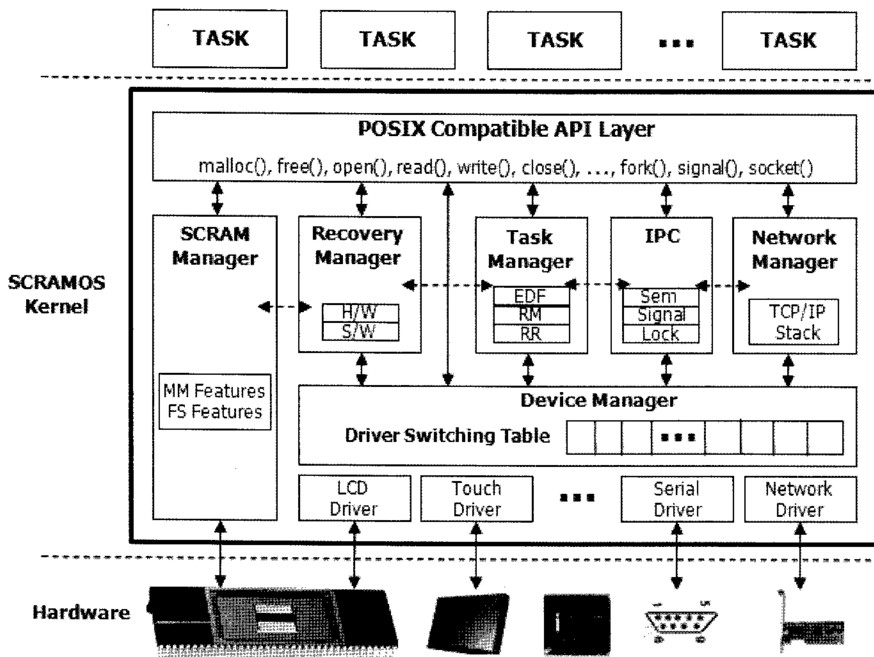


그림 4 SCRAMOS의 구조

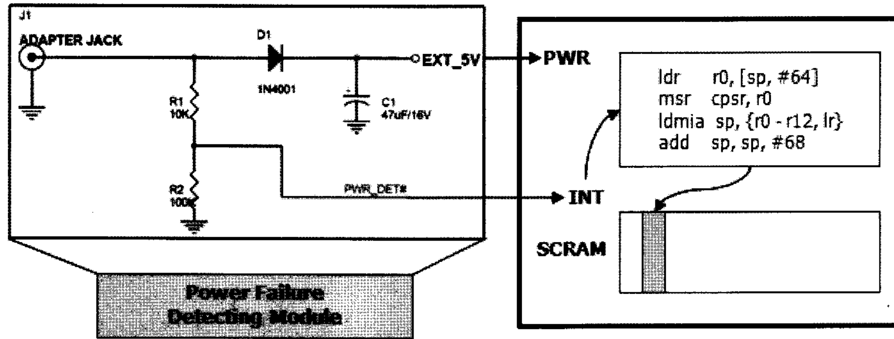


그림 5 전원 결함 탐지 H/W

모든 데이터가 삭제되는 것을 의미한다. 반면 설정되어 있지 않다면 파일과 관련된 데이터는 free() 이후에도 남아있게 된다.

4.2 SCRAMOS

현재 본 연구진은 SCRAM manager를 활용한 SCRAM 전용 운영체제를 구현 중에 있다. 이 운영체제는 그림 4에 보인바와 같이 태스크 관리자, IPC(Inter-Process Communication) 관리자, 네트워크 관리자 등의 일반적인 운영체제 구성 요소와 4.1절에서 소개한 SCRAM manager 그리고 영속적인 컴퓨팅 지원을 위한 Recovery Manager로 구성된다. 이 운영체제는 POSIX API 호환성과, EDF, RM, RR 등의 스케줄링 기법을 제공한다.

Recovery Manager는 S/W, H/W 전원 결함 복구 기법을 모두 지원할 수 있도록 구현되었다. S/W 전원 결함 복구는 부가적인 태스크의 문맥 교환(context switch) 시점을 체크포인트(check point)로 활용함으로써 부가적인 오버헤드 없이 전원 복구 이후 정상 동작이 가능하도록 구현되었다. H/W 전원 결함 복구는 그림 5와 같이 몇 개의 저항과 커패시터를 사용하여 저비용으로 구현 가능한 H/W를 이용함으로써 전원 결함이 발생하는 시점에 인터럽트를 발생시키고, 이때 필요한 정보를 저장함으로써 전원 결함을 해결한다.

5. 실험 결과

SCRAM manager와 SCRAMOS는 400MHz XScale CPU와 64MB SDRAM, 64MB NAND Flash, 그리고 UART, LCD 등의 주변 장치가 장착되어 있는 실제 시스템에서 실험되었다. 구체적인 하드웨어 제원은 표 2와 같다. 또한 32MB의 FeRAM이 장착된 daughter board를 제작하였으며 이를 그림 6에 보였다.

5.1 SCRAM Manager 성능 비교

다른 파일시스템 및 메모리 관리자와 SCRAM manager간의 성능을 비교하기 위해 그림 6의 하드웨어

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, ...

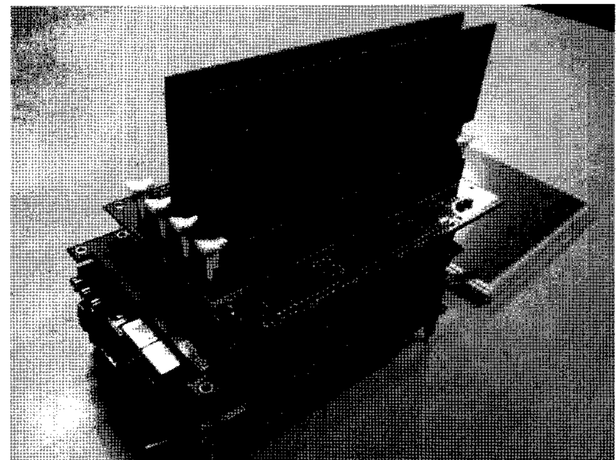


그림 6 SCRAM 개발 환경

상에 Linux 2.6.21을 포팅한 뒤 SCRAM manager를 Linux에서 동적으로 적재 가능한 모듈로 구현하였다. 또한 비교를 위해 리눅스의 FAT 파일시스템과 버디 할당자를 SCRAM 상에서 동작시켰다.

표 3은 마이크로 벤치마크 결과를 보여준다. 우선 파일시스템 관련 실험에서는 creat(), open(), write(), read(), close()의 5개 함수에 대해 성능을 측정하였으며 Write/Read는 512Byte를 단위로 수행하였다. 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

표 3 마이크로 벤치 마크 수행 결과

<table>
<tr><td rowspan="6">FS 관련연산
(us)</td><td></td><td colspan="2">(1) NAND+
NFTL+
FAT</td><td colspan="2">(2) SCRAM+
SCRAM driver+
FAT</td><td>(3) SCRAM+
SCRAM driver+
SCRAM manager</td></tr>
<tr><td>creat</td><td colspan="2">1885</td><td colspan="2">200</td><td>66</td></tr>
<tr><td>open</td><td colspan="2">71</td><td colspan="2">69</td><td>31</td></tr>
<tr><td>write(512)</td><td colspan="2">55739</td><td colspan="2">346</td><td>51</td></tr>
<tr><td>read(512)</td><td colspan="2">21198</td><td colspan="2">2004</td><td>64</td></tr>
<tr><td>close</td><td colspan="2">6</td><td colspan="2">6</td><td>5</td></tr>
<tr><td rowspan="5">MM
관련연산
(us)</td><td></td><td colspan="3">(1) SDRAM +
Linux' Buddy</td><td colspan="2">(2) SCRAM +
SCRAM driver +
SCRAM manager</td></tr>
<tr><td>malloc</td><td colspan="3">21.62</td><td colspan="2">10.55</td></tr>
<tr><td>free</td><td colspan="3">9.86</td><td colspan="2">6.36</td></tr>
<tr><td>load(512)</td><td colspan="3">10.65</td><td colspan="2">27.11</td></tr>
<tr><td>store(512)</td><td colspan="3">7.31</td><td colspan="2">20.16</td></tr>
</table>

파일 시스템을 구축하였고, (2) SCRAM상에 SCRAM driver를 사용하여 FAT 파일 시스템을 구축하였으며, (3) SCRAM 상에 SCRAM driver를 사용하여 SCRAM manager를 동작시켰다. (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. (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 manager는 이러한 복사를 제거한 것이 가장 주된 성능 향상의 요인으로 분석된다.

또한 메모리 관리자와 연관된 마이크로 벤치마크 실험 역시 수행하였다. 이 실험에서는 malloc(), free() 두 개의 함수에 대해 성능을 측정하였고, 할당 받은 공간에 대해 기록하고 읽어오는(512Byte)데 걸리는 속도 역시 측정해 보았다. 이때 (1)과 (2)는 동일한 버디 알고리즘에 기반하고 있기 때문에 비교적 성능 향상을 시도할 기회가 적다. 따라서 파일시스템 실험 결과에 기록된 결과와 달리 대부분 비슷한 수준의 성능을 보이고 있다.

표 4는 실제 응용프로그램을 수행시켜 그 속도를 측정한 실험결과를 보여준다. 각 결과는 모두 10회 반복 수행 후 그 평균값을 기록하였다. 실험에 사용된 프로그램은 Source Forge에서 얻을 수 있는 'distillery', 'ftpget'이라는 프로그램이다. 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 manager가 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 관리 기법이 SCRAM의 두 가지 특성을 모두 활용할 수 있는 적합한 구조임을 증명한다.

5.2 SCRAMOS 성능 결과

SCRAMOS와 다른 실시간 운영체제와의 성능 비교를 위해 잘 알려진 uC/OS-II를 그림 6의 하드웨어상에 포팅하여 실험을 수행하였다. 또한 비교를 위해 버디 할당자와 FAT 파일시스템을 uC/OS-II에 구동시켰으며, SCRAM의 용량을 정적으로 분할하여 버디 할당자와 FAT 파일시스템이 사용하도록 설정하였다. SCRAM manager가 제공하는 전통적인 파일시스템 연산의 성능을 측정하기 위해 postmark 벤치마크 프로그램을 수행시켰으며, file_to_mm(), mm_to_file()등의 인터페이스가 사용가능한 경우 이를 사용하도록 설정된 MPEG 디코딩 시간 측정 벤치마크를 제작하여 실험하였다.

표 4 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
distillery	110.280	10.736	7.072
ftpget	8.945	1.852	0.175

표 5 SCRAM OS의 성능 실험(단위: us)

	SCRAM Manager + SCRAMOS	Buddy and FAT + uC/OS-II
Postmark	11,000,000	49,000,000
MPEG decoding	86	4,105

표 6 부팅 시간 비교(단위: s)

	uc/OS-II	SCRAMOS
(Flash memory) Cold booting	2.33	2.41
(SCRAM) Cold booting	1.46	1.51
Instant booting	N/A	0.06

실험 결과 표 5에서 볼 수 있듯이 SCRAMOS는 기존 운영체제에 비해 좋은 성능을 보이며, file_to_mm(), mm_to_file() 등의 인터페이스를 사용하는 경우 월등한 성능향상을 가져올 수 있다.

한편 시스템의 부팅에 소요되는 시간과 SCRAMOS의 영속적인 컴퓨팅 기능의 성능을 비교해 보았다. 우선 uc/OS-II와 SCRAMOS의 운영체제 자체의 바이너리 파일을 Flash memory에 저장한 경우의 부팅시간(Flash memory Cold booting)을 측정하였고, 이 파일들을 SCRAM에 저장한 경우의 부팅시간(SCRAM Cold booting)을 측정하였다. 또한 SCRAMOS의 Recovery manager가 제공하는 전원 off 또는 전원 결함 시에 시스템의 상태를 저장하고 전원 on 또는 전원 결함 복구 시에 시스템 상태를 복원하는 경우의 부팅 시간(instant booting)을 측정해 보았다. 정상적인 부팅인 경우 uc/OS-II와 SCRAMOS는 유사한 시간이 소모되나, SCRAMOS에서만 제공 가능한 인스턴트 부팅의 경우 부팅작업에 소요되는 시간은 0.06초에 불과함을 확인할 수 있다. 물론 uc/OS-II에서도 동면(hibernation) 기능을 도입하면 콜드 부팅에 비해 부팅 시간이 줄어들 수는 있겠지만, 메모리 상태 저장을 위한 부하가 필요하므로 SCRAMOS의 인스턴트 부팅보다는 시간이 더 걸릴 것으로 예상된다. 결국 SCRAMOS에서는 불시에 시스템의 전원이 소실되더라도 전원인가 이후 바로 컴퓨팅 작업이 수행 가능한 영속적인 컴퓨팅이 가능해 진다.

6. 결 론

본 논문에서는 워드 단위의 임의 접근을 지원함과 동시에 비휘발성 특성을 갖는 SCRAM의 양면성을 효율적으로 활용하기 위한 SCRAM manager를 제안하였다. 기존 저장장치 관리 기법과 달리 SCRAM manager는 부가적인 오버헤드 없이 메모리 객체와 파일 객체간의 변환을 가능케 하는 mm_to_file()과 file_to_mm()이라는 새로운 인터페이스를 제공한다. 또한 제안된 SCRAM manager를 활용하여 성능 향상 및 영속적인 컴퓨팅 기능을 제공하는 SCRAMOS를 설계하였다. 제안된 기법은 실제 SCRAM이 장착된 시스템에서 구현되었다. 실험을 통해 기존 운영체제 대비 획기적인 수준의 성능향상을 가져올 수 있으며, 효율적으로 영속적인 컴퓨팅 기능을 지원함을 확인할 수 있었다.

추후 본 연구를 크게 세 가지 방향으로 확장해 나갈 것이다. 첫째, 가상 메모리 구조를 택하고 있는 Linux 등의 운영체제에 SCRAM manager를 도입하여 실행 파일의 바이너리 이미지가 저장 장치에서 그대로 수행(eXecution In Place, XIP)될 수 있도록 할 것이다. 둘째, 실생활에서 접할 수 있는 다양한 전자 제품에 SCRAMOS의 영속적인 컴퓨팅 개념을 도입함으로써 사용자가 원하는 시점에 바로 제품이 켜지는 인스턴트 부팅 제품을 개발할 것이다. 셋째, 운영체제의 일관성을 동적으로 탐지하고 문제를 해결할 수 있는 도구인 OSCK(Operating System Consistency checK)를 개발할 것이다.

참고문헌

[1] W. Stalling, "Operating Systems: Internals and Design Principles", Prentice Hall.

[2] Storage Class Memories, http://www.ssrc.ucsc.edu/proj/scm.html.

[3] S. Y. Lee and Kinam Kim, "Prospective of Emerging New Memory Technologies", 2004 IEEE International Conference on Integrated Circuit Design and Technology, 2004.

[4] http://www.neakorea.co.kr/article_view.asp?seno=4585.

[5] http://www.itechnews.net/tag/toshiba-feram/

[6] http://www.etnews.co.kr/newswire/press_view.html?id=0184587.

[7] http://techon.nikkeibp.co.jp/english/NEWS_EN/20070226/128173/.

[8] http://arstechnica.com/news.ars/post/20070307-intel-to-sample-pram-this-year.html.

[9] P. M. Chen, W. T. Ng, G. Rajamani, and C. M. Aycock, "The Rio File Cache: Surviving Operating System Crashes," In Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 74-83, Oct. 1996.

[10] S. Akyurek and K. Salem, "Management of Partially Safe Buffers," IEEE Transactions on Computers,

vol. 44, no. 3, pp. 394-407, Mar. 1995.

[11] T. R. Haining and D.D.E. Long, "Management policies for non-volatile write caches," In Proceedings of the 1999 IEEE International Performance, Computing and Communications Conference, pp. 321-328. Feb. 1999.

[12] Kyuhyung Lee, In Hwan Doh, Jongmoo choi, Donghee Lee, Sam H. Noh, "Write-Aware Buffer Cache Management Scheme for Nonvolatile RAM", International conference on Advances in Computer Science and Technology, 2007.

[13] M. Baker, S. Asami, E. Deprit, J. Ousterhout, 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, Oct. 1992.

[14] 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 IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, pp. 596-603, Oct. 2004.

[15] PRAMFS, http://pramfs.sourceforge.net.

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

[17] A. A. Wang, P. Reiher, G. J. Popek, and G. H. Kuenning, "Conquest: Better Performance Through A Disk/Persistent-RAM Hybrid File System," In the Proceedings of the 2002 USENIX Annual Technical Conference, pp. 15-28, Jun. 2002.

[18] E. L. Miller, S. A. Brandt, and D.D.E. Long. "HeRMES: High-performance reliable MRAM-enabled storage," In Proceedings of the 8th Workshop on Hot Topics in Operating Systems, pp. 83-87, May 2001.

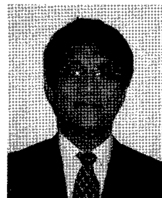
[19] In Hwan Doh, Jongmoo choi, Donghee Lee, Sam H. Noh, "Exploiting Non-volatile RAM to Enhance Flash File System Performance", ACM & IEEE International Conference on Embedded Softrware, 2007.

[20] A. C. Bomberger, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro, "The Key-KOS Nanokernel Architecture," In Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures, pp. 95-112, Apr. 1992.

[21] A. Dearle, R. d. Bona, J. Farrow, F. Henskens, A. Lindström, J. Rosenberg, and F. Vaughan, "Grasshopper: An Orthogonally Persistent Operating System," Computing Systems, vol. 7, no. 3, pp. 289-312, Summer 1994.

백 승 재

2005 단국대학교 컴퓨터 공학과 졸업(공학사)
2004 ~ 현재 비트 컴퓨터 강사
2007 단국대학교 대학원 정보컴퓨터 과학과(이학석사)
2007 ~ 현재 단국대학교 대학원 컴퓨터학과 박사 수료
관심분야 : 운영체제, 임베디드 시스템, 차세대 저장장치 등
E-mail : ibanez1383@dankook.ac.kr



최 종 무

1993 서울대학교 해양학과 졸업(이학사)
1995 서울대학교 대학원 컴퓨터 공학과(공학석사)
2001 서울대학교 대학원 컴퓨터 공학과(공학박사)
2001 ~ 2003 유비퀵스 주식회사 책임 연구원
2003 ~ 현재 단국대학교 공과대학 컴퓨터학부 컴퓨터공학 전공 조교수
2005 ~ 2006 UC Santa Cruz 방문 교수
관심분야 : 운영체제, 임베디드 시스템, 차세대 저장장치 등
E-mail : choijm@dankook.ac.kr