임베디드 시스템을 위한 신뢰성 있는 NAND 플래시 파일 시스템의 설계

이 태 훈[†] · 박 송 화[†] · 김 태 훈[†] · 이 상 기[†] · 이 주 경[†] · 정 기 동[††]

요 약

NAND 플래시 메모리는 저전력 소비, 비휘발성, 읽기 속도의 향상 등의 장점이 있다. 그러나 제자리 덮어쓰기(in-place-update)가 불가능하고 지우는 횟수에 제한이 있으며 페이지 단위로 연산이 수행되는 단점이 있다. 이러한 NAND 플래시 메모리를 위한 전용 파일 시스템으로 YAFFS가 개발되었지만 여러 가지 문제점이 존재한다. 본 논문에서는 빠른 복구를 위한 기법, 효율적인 데이터 갱신 기법 그리고 균등한 메모리 사용을 위한 플레인 지움 정책을 사용하는 파일 시스템을 제안한다. 전원 오류 발생시, 로그 정보를 사용하여 빠른 복구를 지원한다. 그리고 플래시 메모리의 효율적인 사용을 위해 데이터 쓰기 양을 최소화하고 이를 위해 새로운 메타 데이터 구조를 제안한다. 또한 플레인 지움 정책은 플래시의 균등 사용과 임베디드 시스템의 제한된 자원을 고려하여 연산을 최소화한다. 제안된 기법들의 성능을 실험을 통해 증명하고 그 결과를 분석한다.

키워드 : NAND 플래시 메모리, YAFFS, 임베디드 시스템, 파일 시스템, 저널링, 마운팅

RFFS : Design of a Reliable NAND Flash File System for Embedded system

Tae-hoon Lee[†] · Song-hwa Park[†] · Tae-hoon Kim[†] · Sang-gi Lee[†]
Joo-Kyong Lee[†] · Ki-Dong Chung[††]

ABSTRACT

NAND flash memory has advantages of non-volatility, little power consumption and fast access time. However, it suffers from inability that dose not provide to update-in-place and the erase cycle is limited. Moreover, the unit of read and write operations is a page. A NAND flash file system called YAFFS has been proposed. But YAFFS has several problems to be addressed. In this paper, the Reliable Flash File System(RFFS) for NAND flash memory is designed and evaluated. In designing a file system, the following four issues must be considered in particular for the design: (i) to minimize a repairing time when the system fault occurs, (ii) to balance the number of block erase operations by offering wear leveling policy, and (iii) to reduce turnaround time of memory operations by reducing the amount of data written. We demonstrate and evaluate the performance of the proposed schemes.

Key Words : NAND Flash Memory, YAFFS, Embedded System, File System, Journalling, Mounting

1. Introduction

최근 휴대폰, MP3, 디지털 카메라, PMP(Portable Multimedia Player)와 같은 휴대용 단말기를 위한 저장 공간으로 플래시 메모리가 각광받고 있다. 플래시 메모리는 비휘발성, 저전력 소비, 빠른 접근속도 등의 특징이 있으며 메모리 셀을 구성하는 게이트의 종류에 따라 NAND형과 NOR형 플래시 메모리로 구분된다[1]. 그 중에서 NAND 플래시 메모리는 집적도가 높아 대용량 데이터를 저장하기 위한 대부분의 기기에 사용되고 있으며 점차 NOR 플래시 메모리의 시장을 잠식해 가고 있다. 특히, 최근 16GB의 NAND 플래시 메모리의 개발로 2~3년 내에 하드 디스크를 대체할 것이라는 예상도 나오고 있다[2].

그러나 플래시 메모리는 기존의 저장매체에 비하여 다음과 같은 단점이 있다. 첫째, 플래시 메모리는 데이터 수정 시 본래 주소에 덮어쓰기가 불가능하다. 플래시 메모리의 각 비트가 단방향으로만 토글링(toggling)되기 때문에 쓰기 연산 시 초기화 연산을 선행해야 한다. 즉, 쓰기 연산 전에

※ 이 논문은 교육인적자원부 지방연구중심대학육성사업(차세대물류IT기술연구사업단)의 지원에 의하여 연구되었음.
† 준 회 원 : 부산대학교 대학원 컴퓨터공학과
†† 종신회원 : 부산대학교 전기전자정보컴퓨터공학부 교수
논문접수 : 2005년 9월 16일, 심사완료 : 2005년 11월 9일

〈표 1〉 NAND와 NOR 플래시 메모리의 성능 비교[7]

종류 / 연산	NAND	NOR
쓰기(us/byte)	0.4	6.0
읽기(us/byte)	69	45
지우기(us/byte)	0.12	15.2

해당 메모리 공간이 초기화되어 있어야 한다. 둘째, 플래시 메모리의 각 블록은 초기화 연산의 횟수가 제한되어 있기 때문에 플래시 메모리의 전체 공간이 균등하게 사용되지 못하는 경우에 사용가능한 메모리 공간이 급격히 줄어들게 된다[3]. 이러한 플래시 메모리의 단점을 극복하고 효과적으로 사용하기 위해 다양한 지움 정책(cleaning policy)과 균등 사용(wear-leveling) 기법이 적용된 파일 시스템이 제안되고 있다[3-6]. 또한, NAND 플래시 메모리는 비트 또는 바이트 단위가 아닌 페이지 단위의 접근만 허용되며 제품 생산 시부터 오류 블록(bad block)이 존재할 수 있다.

위에서 언급한 플래시 메모리의 특성으로 인하여 기존 파일시스템을 플래시 메모리에 바로 적용할 수 없으므로 플래시용 파일 시스템에 대한 연구가 활발히 진행되어 왔다. 대표적인 예로 FTL(Flash Translation Layer)[8], TrueFFS[9], JFFS(Journaling Flash File System)[10], YAFFS(Yet Another Flash File System)[11] 등이 있다. FTL은 순차적인 플래시 공간이 디스크의 섹터처럼 보이도록 하기 위해 매핑(mapping) 관리를 수행하는 드라이버 형식으로 구현되어 있다. TrueFFS는 VxWorks RTOS에 맞게 구현한 시스템으로 플래시에 대한 블록 디바이스 인터페이스를 제공하기 위해 Block-to-Flash 전환 시스템을 사용한다. JFFS2(Journaling Flash File System)는 플래시 공간을 순차적으로 저장하는 LFS(Log-structured File System)[12]처럼 플래시 메모리에 대한 갱신 연산을 추가 연산으로 변형하여 처리하며, 이를 통해서 플래시 메모리의 덮어쓰기가 허용되지 않는 문제를 해결하였다. 하지만 플래시 메모리의 이용률이 커지면서 쓰기 속도가 저하되고, 마운트 시간이 오래 걸리며, 메인 메모리를 많이 사용하는 단점이 있다. 이런 JFFS2의 단점을 해소하기 위해 개발된 것이 YAFFS이다. YAFFS는 NAND 플래시 메모리 전용 파일 시스템으로 마운트 속도와 NAND 유형 플래시 메모리의 입출력 속도에서 JFFS2보다 성능 면에서 우수하다. 그러나 YAFFS 역시 LFS 방식을 사용하므로 마운트 시에 플래시 메모리 전체를 스캔해야 하며 이로 인해 플래시 메모리가 클수록 마운팅 시간이 길어지는 단점이 있다. 또한 파일에 대한 수정이 발생하면 파일 전체를 다시 플래시 메모리에 쓰게 되므로 메모리의 낭비가 발생하게 되고 쓰기 연산중에 시스템 폴트가 발생하면 파일 일관성 유지를 위한 복구 작업을 수행하지 않는다. 따라서 마운팅에 소요되는 시간이 짧으며 메모리의 균등 사용을 지원하고 시스템 폴트 시 저장된 데이터의 일관성을 제공하기 위한 복구 기능을 포함하는 신뢰성있는 파일 시스템의 필요성이 대두되고 있다.

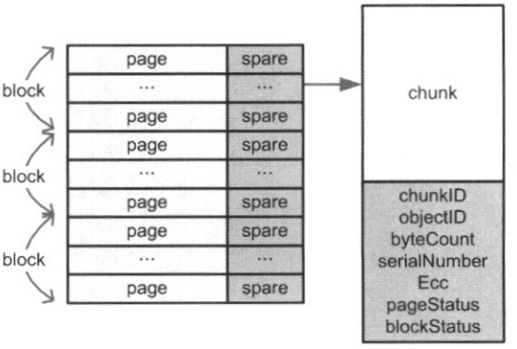
본 논문에서는 플래시 메모리의 크기에 관계없이 거의 일정한 마운팅 시간을 제공하면서 시스템 폴트 발생에 안정적으로 대응하는 신뢰성 있는 플래시 파일 시스템을 설계하고 각 모듈별 성능을 시뮬레이션을 통하여 분석하고 설계에 반영하였다. 현재 설계된 파일시스템을 구현중이다.

본 논문의 구성은 다음과 같다. 2장에서는 YAFFS에 대해 알아보고 3장에서는 제안하는 신뢰성 있는 플래시 파일 시스템의 구조를 기술하고 4장에서는 신뢰성 있는 플래시 파일 시스템의 구현 예와 시뮬레이션을 통한 모듈별 성능을 알아본다. 마지막 5장에서 결론과 향후 연구 과제에 대해 분석한다.

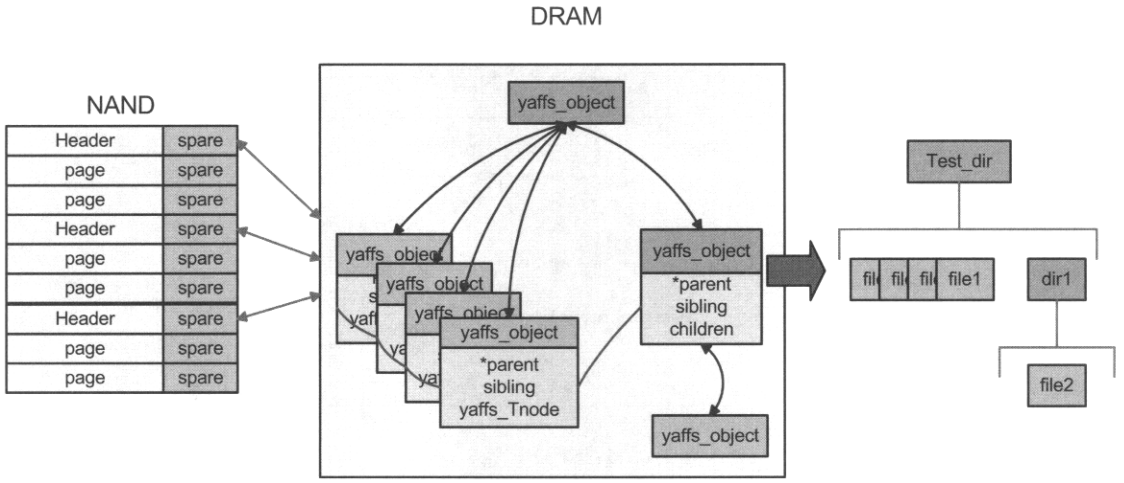
2. YAFFS

YAFFS는 2002년 5월 Aleph One사에서 개발하였으며 JFFS2의 높은 메인 메모리 사용률, 느린 마운팅 속도를 해결하기 위해 개발된 NAND 플래시 전용 파일시스템이다[11]. 일반적으로 'Small' 블록의 NAND 플래시 메모리는 512B의 페이지와 16B의 스페어 영역으로 구성되며 32개의 페이지가 모여 블록을 구성한다. 플래시 메모리에서 읽기와 쓰기는 페이지 단위로 수행되며, 삭제는 세그먼트 단위로 수행된다. YAFFS는 페이지와 스페어를 각각 'chunk'와 'tag'라는 용어로 관리한다. chunk에는 파일의 정보를 관리하는 헤더와 파일의 데이터가 저장된다. 헤더는 파일의 이름과 파일의 크기, 수정 시간, 상위 디렉터리에 포인터 등으로 구성된다. tag에는 chunk의 내용에 대한 메타 데이터가 저장된다. 파일의 헤더 는 chunkID가 0으로 설정된다[9].

JFFS2의 저널링 기법에서는 파일의 수정 시 기존의 파일을 직접 수정하는 것이 아니라 수정된 내용을 로그 형태로 저장한다. 이후 해당 파일을 메모리로 읽어 들일 때, 로그의 버전 정보를 이용해서 파일을 수정하게 된다[8]. 이와 같이 수정된 부분만 로그로 관리하여 제자리 덮어쓰기 문제는 해결하였으나 시스템 폴트 발생 시 파일 시스템 구성을 위해 메모리 전체를 순차적으로 읽어야 하므로 마운팅 시간이 길어지는 단점이 있다. YAFFS에서는 이러한 문제를 chunk와 tag를 이용하여 해결하였다. 즉, chunkID가 0인 chunk만 모



(그림 1) YAFFS의 플래시 메모리 구조



(그림 2) YAFFS의 DRAM상의 구조체

〈표 2〉 스페어 영역 tag 변수

필 드	설 명
objectID	chunk의 데이터가 속한 파일의 ID
chunkID	파일 내부에서 chunk의 인덱스
serialNumber	갱신 연산 시 증가되는 번호 (오류 복구 시 사용)
byteCount	chunk 내의 유효한 바이트의 수
ecc	에러 검출을 위한 비트

므로 JFFS2에 비하여 마운팅 시간이 줄어들게 된다. 그러나 YAFSS 역시 플래시 메모리의 모든 스페어 영역을 읽어야 하므로 플래시 메모리의 크기에 비례하여 마운팅 시간이 증가하게 된다.

또한 YAFFS는 오류 복구를 파일 시스템의 일관성 유지 측면만 고려하여 실제 파일 내용의 일관성을 지원할 수 없다. 즉, YAFFS에서는 갱신을 할 때, 이전의 스페어의 시리얼 번호를 저장했다가 1씩 증가시키며 마운트 과정에서 시리얼 번호를 검사하여 동일한 objectID와 chunkID를 가진 chunk가 있다면 오래된 chunk를 무효화하는 방식으로 동작한다. 이때, 여러 개의 chunk를 플래시 메모리에 쓰는 도중 시스템 폴트가 발생한다면 일부 chunk는 갱신된 내용을 반영하고 미처 플래시 메모리에 저장되지 못한 내용은 이전의 정보를 나타내므로 파일의 내용의 일관성을 보장하지 못하는 문제점이 발생하게 된다.

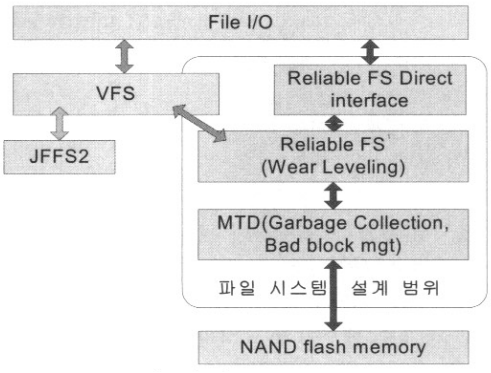
YAFFS는 마운팅 시 플래시 메모리의 모든 헤더를 메인 메모리로 읽어 파일 시스템 구조를 생성한다. 각 헤더마다 yaffs_Obejct 객체가 메인 메모리에 만들어지며, 이 객체들은 서로 연결되어 파일 시스템의 트리 구조를 형성한다. 또한 각 객체는 자신이 관리하는 파일에 속하는 데이터들의 위치를 yaffs_Tnode라는 트리로 관리한다. 각 파일의 데이터에 대한 트리를 구성하기 위해서는 모든 플래시 메모리의 데이터 chunk의 스페어를 읽어야 한다.

지금까지 설명에서 YAFFS의 두 가지 문제점을 발견할 수 있다. 첫째, 마운팅 과정에서 플래시 메모리의 모든 헤더와 스페어 영역을 읽어야 한다는 것이다. JFFS2의 느린 마운트를 해결한 것이 YAFFS이지만 모든 플래시의 스페어를 순차적으로 읽는 것은 마운트 시간이 플래시 메모리의 용량에 비례해서 증가하게 된다. 둘째, 파일 시스템의 일관성 유지만을 위한 기능을 제공한다는 것이다. NAND 플래시 파일 시스템이 주로 사용되는 이동형 정보기기는 배터리를 사용하고 외부환경에 쉽게 노출되기 때문에 전력중단(Power Fail)이나 예상치 못한 장애(Crash)가 생길 수 있다. 따라서 시스템 폴트로 인한 마운트 시간을 최소화하며 파일 내용의 일관성을 제공하는 신뢰성 있는 NAND 전용 파일 시스템이 요구된다. 본 논문에서는 위 두 문제의 해결을 위해 새로운 파일 시스템 구조를 제안한다.

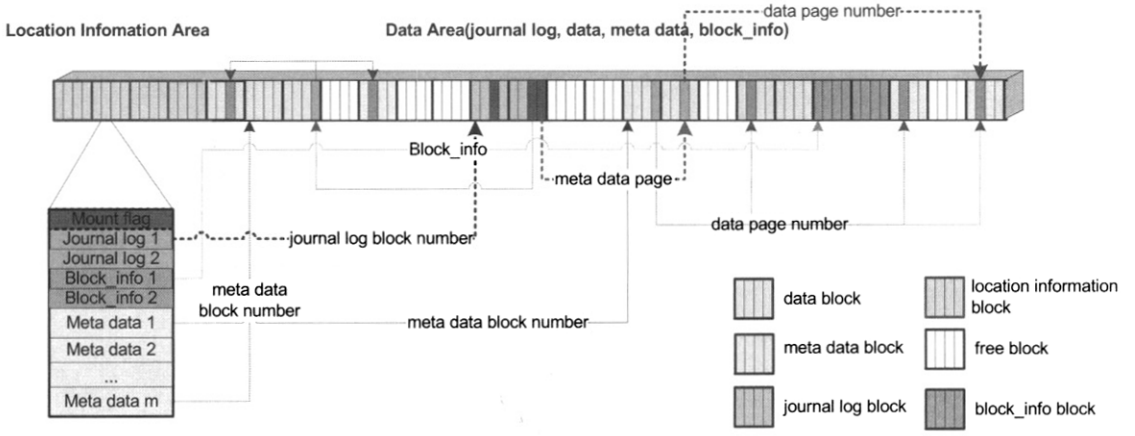
3. 신뢰성 있는 플래시 파일 시스템(RFFS)의 구조

3.1 파일 시스템의 설계 범위 및 구조

RFFS 파일 시스템은 빠른 마운팅과 파일시스템 및 파일 내용의 일관성을 제공하고 효율적인 균등 사용 정책을 적용하여 플래시 메모리의 가용성을 높이는데 목적을 두고 있다. 빠른 마운팅을 지원하기위해 파일 시스템의 구축에 필요한



(그림 3) RFFS 구조



(그림 4) 플래시 메모리 상의 파일시스템 이미지

정보를 메타영역, 블록 정보영역, 로그 영역으로 나누어서 관리한다.

RFFS의 구조를 살펴보면 (그림 3)과 같다. 상위 계층에서는 VFS(Virtual File System)와 Direct Interface를 통해서 접근할 수 있다. 기존의 파일 시스템과의 연동을 위해 VFS를 지원하며, 좀 더 빠른 접근을 위해서는 Direct Interface를 사용한다. 하단의 MTD(Memory Technology Device) 계층에서는 리눅스에서 일반적인 플래시 메모리나 램과 같은 다양한 메모리 장치를 쉽고 편리한 지원을 가능하게 하며, 오류 블록 관리 기능이 포함되어 있다.

(그림 4)는 플래시 메모리에 저장되는 파일 시스템의 전체 이미지를 나타내고 있다. 파일 시스템은 크게 위치 정보 영역(Location Information Area)과 데이터 영역(Data Area)으로 구성된다. 전체 파일 시스템의 메타 정보가 포함된 위치 정보 영역은 2개의 세그먼트로 구성되며 위치는 언제나 0번 블록으로 고정되어 있다. 데이터 영역은 메타데이터를 저장하는 메타 영역, 실제 파일이 저장되는 데이터 블록, 모든 블록의 상태를 저장하는 블록 정보 영역, 저널링 정보를 저장하는 로그 영역으로 구성되며 각 영역의 선택은 균등 사용 정책에 의해 결정된다. 이때, 데이터 블록만 하나의 블록 단위로 할당되며, 메타 영역, 블록 정보 영역, 저널링 영역은 하나의 세그먼트 단위로 할당된다. 위치 정보 영역은 512B의 Location_Info 객체로 구성되며, Location_Info 객체는 마운트 플래그와 로그 포인터, Block_Info 포인터, 메타 포인터로 구성된다.

3.1.1 위치 정보 영역

위치 정보 영역은 데이터 영역에 분산되어 있는 메타 영역과 로그 영역, 블록 정보 영역의 주소를 관리하는 영역이다. 이 영역의 정보를 이용하여 파일 시스템 전체를 스캔하지 않고 관련된 영역만을 읽어서 마운팅 속도를 향상시킬 수 있다.

NAND 플래시 메모리는 페이지 단위의 쓰기만을 허용하므로 위치 정보 영역의 쓰기도 페이지 단위로 수행된다. 페이지 내의 구성 요소는 (그림 4)와 같다. 그 중에서 마운트 플래그는 파일 시스템이 정상적으로 마운트 되었는지를 나타내는 것이다. 마운트 플래그가 1로 설정되어 있다면 비정상 종료를 나타내므로 마운트 과정에서 저널링 기능이 동작하여 파일 시스템의 복구를 수행하게 된다. 로그 포인터는 저널링 과정에서 참조할 로그 영역의 주소를 저장하고 있다. 로그 영역을 읽어 이전에 수행된 파일 연산의 일관성을 검사하고 복구를 수행한다. Block_Info 포인터는 전체 블록의 상태가 저장되어 있는 블록 정보 영역의 주소를 저장하고 있다. YAFFS에서 마운트 과정을 거치면서 메인 메모리에 파일 시스템의 구조를 구성하게 되는데 제안된 파일 시스템에서는 이 영역의 정보를 이용하여 플래시 메모리의 상태정보를 메모리상에 구축하게 된다. 메타 포인터는 각 파일의 Inode에 해당하는 메타 데이터를 모아둔 메타 영역의 첫 블록의 주소를 저장한다. 이 포인터를 이용하여 메인 메모리 상에 파일 시스템의 구조를 생성하게 된다.

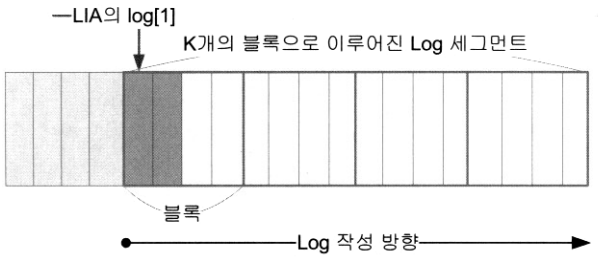
3.1.2 데이터 영역

데이터 영역은 로그, 메타, 블록 정보, 일반 데이터가 실제로 저장되는 공간이다. 일반 데이터를 제외한 로그와 메타, 블록 정보는 세그먼트 단위로 할당한다. 메타 영역은 파일의 Inode 정보를 저장하는 공간으로 상세 자료구조는 <표 3>에 표기되어 있다. 이 영역은 세그먼트 단위로 할당받고, 할당 받은 세그먼트의 주소를 위치 정보 영역의 메타 포인터 배열에 등록한다. 파일시스템에서 지원 가능한 최대 파일의 수는 블록 정보 영역의 메타 포인터 배열과 메타 세그먼트 내의 페이지 수의 곱으로 표시된다. YAFFS에서는 DRAM 상의 블록 상태 정보를 마운트할 때 전체 플래시 메모리를 스캔하여 구축하고, 전체 블록의 상태를 관리하지만 제안하는 파일 시스템에서는 Block_Info 배열을 플래시 메모리에서 바로 읽어 구축할 수 있으며 다음 마운트 시 사용하기 위해 주기적으로 쓰기 작업을 수행한다. 로그 영역은 저널링 기법에서 쓰이는 로그를 저장하는 공간이다. 로그는 트랜잭션 단위로 구분되며, 각 트랜잭션 끝나면 해당 로그는 무효화된다. 데이터 영역은 실제 파일의 내용이 저장되는 공간이며, 블록 단위로 할당되고 데이터 블록을 할당하

기 위한 균등 기법이 적용되는 영역이다.

3.1.3 로그

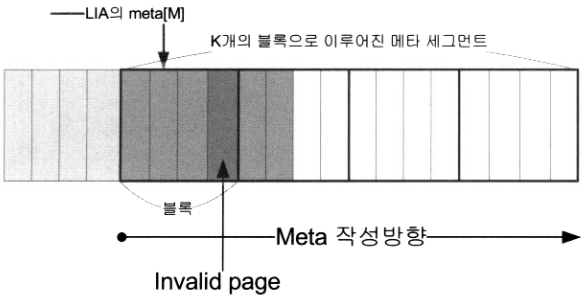
로그 영역은 파일 시스템의 일관성을 유지하기 위해 관리되는 영역이다. 로그에는 최근에 쓰기 연산이 일어난 파일의 메타 블록번호와 최신 데이터인지를 구분하기 위한 시리얼 번호, 파일연산이 완료되었다는 커밋 플래그 등이 저장된다. 로그는 세그먼트 단위로 할당받아 사용되고 쓰기 연산이 종료되면 무효화된다. 저널링 기능은 마운트 시, 위치 정보 영역에 저장되어있는 마운트 플래그를 검사하여 비정상 종료일 경우에만 동작을 한다. 두 개의 로그 포인터를 각각 따라가서 저장된 메타 정보와 데이터가 일치하는지를 검사하게 된다.



(그림 5) 로그 구조

3.1.4 메타 영역

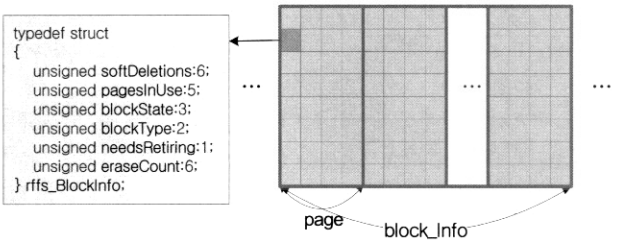
메타 영역은 객체의 정보가 저장되어 관리되는 영역이다. 객체의 타입에는 디렉터리, 파일, 하드링크, 심볼릭 링크 등이 있으며 각 타입에 따라 다른 멤버 변수를 가진다. 파일의 경우, 해당 파일의 페이지를 관리하기 위한 트리 구조체를 가지고, 디렉터리의 경우 하위 객체들의 포인터를 가지게 된다. 메타 영역은 (그림 6)과 같이 세그먼트 단위로 할당되며, 할당될 때마다 위치 정보 영역의 메타 포인터리스트에 등록이 되어 관리된다. 각 메타에는 파일의 데이터가 저장되어 있는 페이지들의 주소가 리스트로 관리되기에 메타를 읽어 시스템의 구조와 파일관리를 위한 트리를 구성한다.



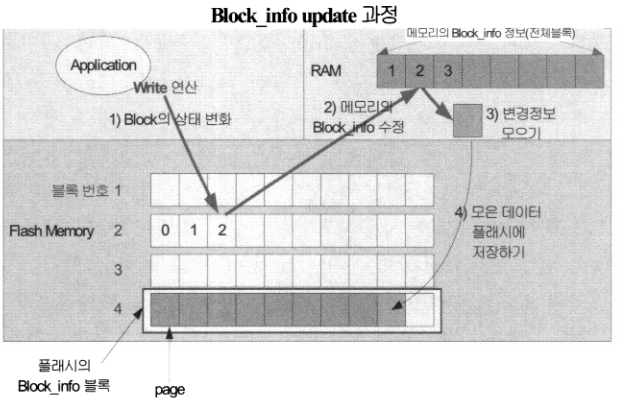
(그림 6) 메타 영역 구조

3.1.5 블록 정보 영역

블록 정보 영역은 전체 블록의 수만큼의 배열로 구성되며



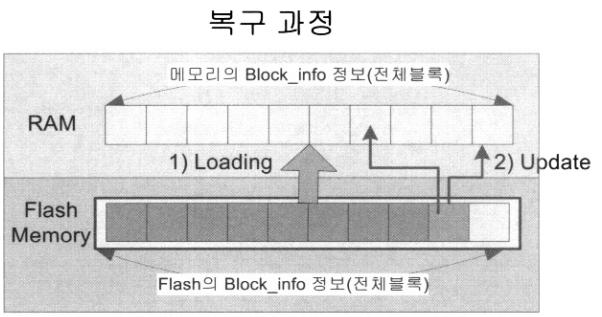
(그림 7) 블록 정보 영역 구조



(그림 8) 블록 정보 영역 갱신 과정

블록의 상태를 저장하고 있다. 사용되고 있는 페이지의 수를 나타내고 있는 pagesInUse, 블록의 상태를 나타내는 blockState, 블록에 저장되는 데이터의 타입을 나타내는 blockType, 오류 블록임을 나타내는 needsRetiring, 균등 사용 정책에 사용되는 블록의 지운 횟수 eraseCount로 구성되어 있다. (그림 7)의 자료구조를 파일 시스템 마운팅 시 메인 메모리로 읽어 들여 마운팅 속도를 향상시킬 수 있다.

(그림 8)은 블록 정보 영역의 갱신 과정의 예를 보여준다. 1)에서 응용 프로그램의 쓰기 연산을 수행하여 2번 블록의 3번 페이지의 상태가 변경되었다. 2) 이를 반영하기 위해 메인 메모리의 Block_Info 배열 중 2번 블록의 정보가 업데이트된다. 이때, 3) 변경된 정보를 플래시 메모리에 기록하기 위해 업데이트된 정보를 페이지 단위로 모아서 4) 플래시 메모리의 Block_Info 블록의 빈 페이지에 저장한다. 세그먼트를 모두 사용하면 새로운 세그먼트를 할당받는데 이때 메



(그림 9) Block_Info 배열의 복구 과정

인 메모리의 전체 Block_Info 배열을 적어주고 그 뒤부터 수정된 Block_Info 객체를 (그림 8)과 같은 방식으로 모아서 저장하게 된다.

(그림 9)는 비정상 종료 시 Block_Info 배열을 복구하는 과정을 보여준다. 마운팅 과정에서 먼저 위치 정보 영역에서 마운트 플래그를 검사하게 된다. 마운트 플래그가 1로 설정되어 있으면 정상적인 종료를 수행하지 않은 것으로 판단하고 저널링 기법의 회복 알고리즘이 수행되게 되고, 그 이후 파일 시스템의 복구 과정에서 Block_Info 배열의 복구가 수행되게 된다. 먼저 위치 정보 영역의 Block_Info 포인터를 따라가서 전체 Block_Info 배열을 읽어서 메인 메모리에 구성하고 수정된 정보만 모아둔 Block_Info chunk를 읽어서 갱신된 정보를 반영하게 된다.

3.2 파일 관리

플래시 메모리를 사용하는 파일 시스템에서는 제자리 덮어쓰기가 불가능하므로 데이터를 변경하고자 하는 경우, 변경하고자 하는 데이터가 들어있는 블록을 무효화하고 초기화된 블록을 할당받아 수정된 데이터를 저장한다. 이때, YAFFS와 같은 기존의 플래시 파일시스템은 파일 전체를 플래시에 새로 작성하는 단점이 있다. 이는 큰 파일의 일부 데이터를 변경할 경우 플래시 메모리의 공간 낭비와 쓰기 시간이 늘어나는 문제가 발생한다. RFFS에서는 이러한 비효율성을 막기 위해 새로운 파일 관리 및 쓰기 기법을 제안한다.

3.2.1 메타 데이터 구조

〈표 3〉 RFFS의 메타데이터 구조

Byte	정보
4	파일 아이디(id)
4	파일 타입
4	상위 디렉토리(directory)
2	파일 이름의 체크 합(check sum)
8	파일 이름
4	파일 권한
4	파일 소유자의 사용자 아이디
4	파일 소유자의 그룹 아이디
4	파일의 마지막 접근 시간
4	파일의 마지막 수정 시간
4	파일의 마지막 변경 시간
4	파일 오프셋(offset)
4	파일 크기
4	동일 파일 아이디
1	링크(link) 수
8	파일 별칭(alias)
4	이전 메타 데이터의 위치
4	파일의 데이터 위치 정보 1
…	…
4	파일의 데이터 위치 정보 n

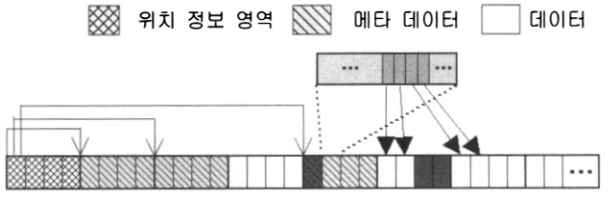
RFFS에서의 메타 데이터의 구조는 <표 3>과 같다. 메타 데이터는 파일에 대한 정보를 가지며 데이터의 페이지 수만큼 데이터의 위치에 대한 정보를 포함한다. 메타 데이터는 하나의 페이지에 기록되므로 하나의 메타 데이터가 나타낼 수 있는 데이터의 크기는 한정되어, 결과적으로 지원하는 파일의 크기에 제한을 두게 된다. 이러한 문제점을 해결하기 위해 하나의 파일에 대한 메타 데이터들을 링크드 리스트(linked-list) 구조로 유지하며 지원 가능한 파일의 최대 크기는 4GB이다.

3.2.2 데이터 수정

RFFS에서의 데이터 수정은 다음과 같은 절차로 수행한다.

(1) 수정되기 이전의 파일 데이터를 읽어온다.
(2) 수정하려는 파일의 데이터와 (1)에서 읽어온 데이터를 비교하여 수정되는 데이터의 위치와 크기를 조사한다.
(3) 수정되는 데이터를 기록할 영역을 할당한다.
(4) 메타 데이터의 정보를 갱신하여 플래시 메모리에 저장한다. 이때, 메타 데이터를 위해 새로운 세그먼트를 할당받은 경우에는 위치 정보 영역에 갱신된 메타 데이터의 위치를 기록한다.
(5) (3)에서 할당받은 데이터 영역에 수정된 데이터 기록이 완료되면 수정되기 이전의 데이터를 무효화한다.

(그림 10)은 파일의 일부분에 데이터를 수정하여 일부 데이터가 변경되고, 새로운 데이터가 추가되어 파일의 크기가 커지는 경우 일어나는 연산의 과정을 보여준다. 이러한 경우, 수정된 데이터와 추가된 데이터 및 갱신된 데이터만 플래시 메모리에 기록한다.



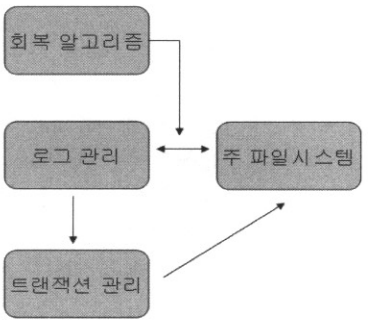
(그림 10) 데이터 수정 과정

3.3 로그 관리

3.3.1 구조

EXT3 파일 시스템의 경우 다양한 종류의 로그 옵션을 제공하지만 임베디드 시스템에서의 과도한 저널링은 파일 시스템 전체의 성능저하를 유발할 수 있으므로 최소한의 저널링 기능을 제공하여 시스템의 신뢰성을 제공하는 것이 바람직하다. RFFS에서의 저널링은 파일 시스템의 일관성 및 파일 내용상의 일관성을 유지하는 것을 목적으로 한다[13].

RFFS에서는 파일시스템의 정상적인 언마운트 확인을 위해 마운트 플래그를 사용한다. 즉, 정상적인 언마운트 수행 시 최종적으로 마운트 플래그를 0으로 설정하고 파일 시스



(그림 11) 저널링 파일 시스템 구성도

〈표 4〉 로그의 자료 구조

크기	항목	설명
32bit	meta	메타데이터의 페이지 번호
32bit	serialNumber	파일에 대한 버전 정보
32bit	commit	트랜잭션 완료
32bit	id	트랜잭션 ID
32bit	next	이전 로그 영역 위치
32bit	offset	변경되어 쓰여질 위치
32bit	asize	offset위치부터 추가되는 데이터 크기
32bit	rsize	offset위치부터 감소되는 데이터 크기
32bit	objId	객체 번호
32bit	renamed	이름 변경 연산 여부
32bit	checksum	오류확인

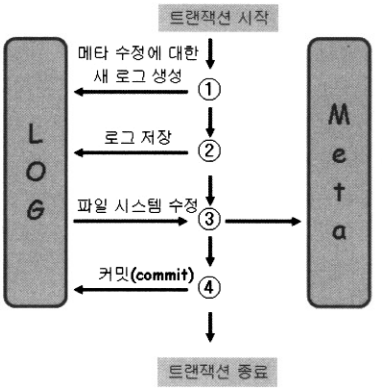
템이 언마운트되고 마운트시 이 플래그를 확인하여 파일 시스템 복구 수행 여부를 결정한다. 제안된 파일 시스템에서의 전체적인 저널링의 흐름은 (그림 11)과 같다.

쓰기 연산의 발생시 플래시 메모리에 쓰기 작업을 수행하기 전에 먼저 연산에 대한 로그가 생성되고, 각각의 로그는 트랜잭션에 의해 관리된다. 만약, 장애가 발생했을 경우 회복 알고리즘을 통해 복구한다.

RFFS에서의 로그 자료 구조는 <표 4>와 같다. 'meta'필드에서 메타데이터에 대한 위치 정보를 가지며 트랜잭션 ID를 통해 순차적으로 수행되는 트랜잭션을 관리한다. 로그내의 offset, asize와 rsize 필드는 서로 연관성이 있는데, 파일의 업데이트가 발생하게 되면 수정되어야 하는 해당 파일의 시작 위치인 offset을 받게 되고 offset을 기준으로 크기가 증가할 때는 증가하는 크기만큼 asize를 인자로 할당 받는다. 반대로 크기가 감소될 경우에는 rsize를 인자로 사용한다. 이 인자들을 통해 byte 단위로 변화된 크기에 대한 정보를 가지며, 메타데이터 정보를 통해 업데이트 부분만 다시 쓰기가 가능하다.

3.3.2 로깅 기법

RFFS는 시스템 폴트 시 빠른 복구를 위해 로깅을 사용한다[14, 15]. 로깅은 파일 시스템에서 임의의 연산이 수행될 때 그 연산에 대한 정보를 저장하는 것을 의미한다. 본 논문에서 제안하는 로깅은 다음과 같은 순서로 수행한다.



(그림 12) 제안하는 로깅 구조

1) 파일 시스템이 마운트 되고 파일 및 디렉토리의 생성, 삭제, 이름 변경과 같은 연산이 수행될 때마다 로깅이 수행되고, 로깅은 수행된 연산에 대해 락(lock)을 가진다.
2) 플래시 메모리상에서 로그영역을 사용하기 위해 k개의 블록을 할당받는다. k는 로그영역에 할당되는 블록의 개수이고, 전체 플래시 메모리의 크기에 따라 그 크기를 유동적으로 관리한다.
3) RAM에 요청된 연산의 로그를 생성하고 플래시 메모리의 로그 영역에 기록한다.
4) 로깅을 위해 획득한 락을 해제하고 연산을 수행한다.
5) 연산이 완료되면 commit이 이루어지고 해당 로그는 무효한 데이터(Invalid Data) 가 된다.
6) 연산이 수행될 때 마다 1)~5)의 과정이 반복적으로 수행되고, 할당 받은 로그 영역을 모두 사용했을 경우 새로운 k개의 블록을 할당한다, 그리고 이전 로그 영역에 대한 포인터를 소유한다. 연산의 수행이 commit 되지 않은 상태에서 새로운 연산이 발생하여 새로운 로그가 쓰여 질 경우 포인터를 통해 연속된 영역처럼 사용이 가능하다.

(그림 12)에서는 RFFS에서 로깅(logging)의 흐름을 표현하고 있다 그림에서와 같이 메타쓰기와 로깅은 트랜잭션 단위로 수행되며, 실제 파일 시스템의 연산이 수행되기 전에 로그 영역에 로깅이 완료된 후 사용자에 의한 연산을 수행한다. 따라서 로깅연산은 파일 시스템의 다른 연산에 간섭받지 않고 로그 데이터에 대한 무결성이 보장된다.

데이터의 복구 과정은 다음과 같다

1) 위치 정보 영역의 마운트 플래그가 1인 경우 비정상적 종료로 판단하고 복구 작업 수행한다.
2) 위치정보 영역의 로그 포인터를 통해 로그 영역을 찾고, 로그 영역을 검색(scan)하여 최근에 실패한 트랜잭션을 찾는다.
3) 로그의 메타 필드를 통해 메타데이터 영역을 찾아 데이터를 읽어온다. Size필드와 실제 데이터의 크기가 같은지 비교하고 파일을 구성하는 모든 페이지가 유효

(valid)한지 검사한다.

4) 모두 유효할 경우 이전 데이터는 무효(invalid)상태로 설정하고 새 데이터를 파일 시스템 구성에 포함한다. 유효하지 않거나 완료되지 않은 트랜잭션은 폐기하고 이전의 상태로 롤백(roll back)한다.

위와 같은 복구 과정을 통하여 비정상적인 종료 시 파일 시스템의 일관성 뿐 아니라 파일의 내용의 일관성도 유지할 수 있다.

3.4 지움 정책

플래시 메모리의 가장 큰 제약인 지우는 속도와 세그먼트당 지울 수 있는 횟수가 한정된다는 문제점을 극복하기 위하여 효율적인 지움 정책이 필요하다. 본 절 에서는 새로운 블록을 확보하기 위한 지움 정책과 쓰기를 수행하기 위한 블록 할당 정책을 제안한다.

3.4.1 기존 지움 정책

Greedy 정책은 유효 블록이 가장 적은 세그먼트를 선택하여 삭제하는 방법으로서 클리닝 횟수는 가능한 한 최소화하면서, 많은 공간을 확보하는 방법이다[5, 6]. 이 기법은 삭제를 위한 비용을 최소화하므로 삭제 효율은 높으나 삭제 연산이 특정 세그먼트에 집중될 수 있으므로 메모리의 균등 사용이 어려워 메모리의 수명을 단축시킬 수 있는 단점이 있다.

Cost-Benefit 방법은 특정 세그먼트에 집중해서 삭제연산이 발생하여 플래시 메모리의 수명을 단축시키는 문제점을 개선하기 위하여 고안된 정책이다[16]. (식 1)에서는 삭제의 효율성뿐만 아니라 특정 세그먼트에 대한 집중을 피하기 위해 세그먼트가 사용된 최근 시간을 고려하여, 클리닝할 세그먼트를 선택한다. 식에서 *age*는 세그먼트가 가장 최근에 수정된 이후의 시간이며, u는 세그먼트의 이용률, $1-u$는 활용 가능한 세그먼트의 비율을 의미한다.

$$\frac{benefit}{\cos t} = \frac{age \times (1-u)}{2u} \quad (1)$$

순위별 지움 정책(Ranking Cleaning Policy,RCP)은 세그먼트들의 등급을 계산하여 값이 큰 순서대로 세그먼트를 지우는 방법[5]으로 삭제를 위한 비용과 균등 사용을 모두 고려한 방식이다. (식 2)는 RCP의 계산 과정을 식으로 나타낸 것이다. 식에서 v는 세그먼트 내에서 유효 블록에 대한 비율, f는 비어 있는 블록에 대한 비율, i는 무효 블록에 대한 비율, e는 세그먼트의 전체 지움 가능 횟수에 대한 현재 지운 횟수의 비율, A는 e에 대한 가중치를 의미한다.

$$R = A \times \frac{i}{2v \times f \times e} \quad (2)$$

Greedy 방법은 균등 사용 정책을 무시하고 있으며 Cost-benefit 방법은 age를 이용하여 고려하고 있으나 age 자체가 삭제 횟수가 적다는 것을 의미하지는 않는다. 이러한 문제점을 해결하기 위해 RCP는 삭제 횟수를 사용하였다. 그러나 제한된 자원을 가진 임베디드 시스템에서는 Garbage Collection이 발생할 때마다 무효 데이터(invalid data)의 비율(i), 유효 데이터(valid data)의 비율(v), 빈 공간(free data)의 비율(f), 세그먼트의 전체 삭제 가능 횟수에 대한 현재 지운 횟수의 비율(e)을 각 세그먼트에 대해 구하고, 순위를 결정하는 많은 연산으로 인해 전체 시스템의 성능을 저하시키게 된다.

3.4.2 플레인 지움 정책(Plain Cleaning Policy, PCP)

본 절에서는 임베디드 시스템의 제한된 연산능력을 고려하여 연산을 최대한 줄이며 균등 사용을 지원하는 플레인 지움 정책(Plain Cleaning Policy)을 제안한다.

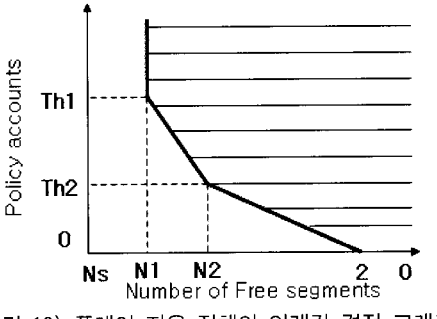
플래시 파일 시스템의 성능은 쓰기 연산에 의해 결정된다. 쓰기 연산은 데이터를 쓰는 연산뿐만 아니라 빈공간이 없을 경우 무효화된 데이터를 삭제하는 연산과 삭제되는 세그먼트에 유효한 데이터가 포함되어 있는 경우 해당 데이터를 빈 공간으로 옮기는 연산도 포함한다. 그러므로 파일 시스템의 성능을 향상시키기 위해서는 삭제 비용이 적은 세그먼트를 선택하여야 한다. (식 3)은 삭제 비용을 나타낸다.

$$C = (2 \times V) + F \quad (3)$$

(식 3)에서 첫 번째 항은 유효한 데이터를 다른 세그먼트로 옮기는 비용과 지우는 비용을 포함한다. 두 번째 항은 삭제될 세그먼트의 빈 공간을 삭제하여 낭비되는 비용이다. 삭제할 세그먼트는 (식 4)를 이용하여 결정된다.

$$R = \frac{EA}{C} \quad (4)$$

(식 4)에서 EA(ErasAble)는 플래시 메모리의 삭제 한계치와 현재 삭제 횟수의 차이 값으로 균등한 삭제를 위해 사용되며 블록 내의 페이지는 유효(Valid), 무효(Invalid), 빈(Free) 페이지 중에 한 가지 상태에 해당된다. 그러므로 두 가지의 상태를 가진 페이지의 비율을 알 수 있다면 나머지 한 가지 상태의 페이지의 비율을 알 수 있게 된다. 그러므로 플레인 지움 정책에는 V와 F의 비율을 만을 이용하여 계산 복잡도를 낮추었다.



(그림 13) 플레인 지움 정책의 임계값 결정 그래프

RCP 방법은 한 개의 임계값을 가지고 빈(Free) 세그먼트의 수가 임계값 이하로 떨어지면 지움 정책이 수행된다. 그리고 임계값 이상을 확보할 때까지 세그먼트를 지워준다[4]. 그러나 단일 임계값이므로 임계값 이하로 감소한 후에만 지움 정책이 동작하게 되어 시스템의 부하가 집중될 수 있다. 플레인 지움 정책에서는 지움 연산의 집중을 해결하기 위해 (그림 13)과 같이 두 개의 임계값을 사용하였다. 빈 세그먼트의 수가 Th1과 Th2의 사이에 존재할 때 세그먼트 내의 모든 블록이 무효이며 삭제 횟수가 최소인 세그먼트를 찾아서 삭제한다. 유효한 데이터가 한 개라도 존재하는 세그먼트는 지움 대상에서 제외한다. 따라서 Th1과 Th2사이에서는 유효한 데이터의 복사 비용이 없는 이상적인 지움 연산만을 수행 한다. 빈 세그먼트의 수가 Th2 이하인 경우는 RCP 방법과 같이 Th2 개의 빈 세그먼트를 유지하기 위한 차이만큼의 세그먼트를 삭제한다. 이와 같이 두 개의 임계값을 사용함으로써 지움 연산을 분산적으로 수행할 수 있다.

4. 실험 및 성능 평가

제안하는 RFFS의 성능 향상을 위하여 각 모듈별로 시뮬레이션을 통하여 성능을 분석하고 그 결과를 파일 시스템 설계에 반영하였다. 실험은 Pentium 4, 2.0GHz, Memory 256MB 사양의 하드웨어 환경에서 Red Hat 9.0 Linux 상에서 이루어졌다. 현재 설계된 RFFS를 구현중이다.

4.1 쓰기 연산 성능 실험

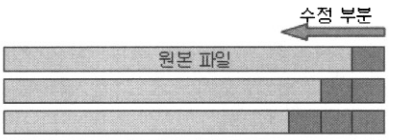
4.1.1 시뮬레이션과 작업부하의 설계

성능 분석을 위해 저장 매체로서 4,000개의 블록으로 구성된 64MB 용량의 플래시메모리를 기반으로 RFFS의 쓰기 연산을 시뮬레이션 하였다. 각 블록은 32개의 페이지로 구성되며 관련 실험 인자는 <표 5>와 같다.

본 실험에서는 수정된 데이터만 플래시 메모리에 쓰는 경우의 실험결과를 평가한다. (그림 14)는 쓰기 연산 성능 실험 과정을 보여주고 있다. 실험은 먼저 원본 파일을 생성하고, 그 원본 파일의 끝 부분의 데이터를 수정한다. 수정하는 데이터의 크기를 한 단위씩 증가시키면서 총 쓰기 양을 측정한다. 이때 파일의 크기에 대한 성능을 측정하기 위해서 원본 파일의 크기를 10KB, 100KB로 하였다. 그리고 수정되는 데이터의 크기의 증가 단위는 각각 1KB, 10KB로 하였다. 10번의 수정연산을 하여서 총 쓰기 양을 YAFFS와 비교해보았다.

〈표 5〉 시뮬레이션 파라미터

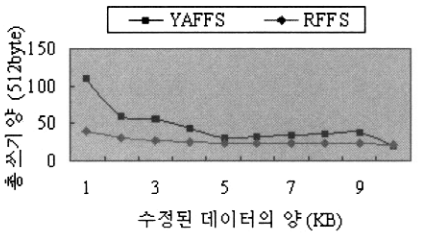
인자	값
플래시 메모리 용량	64 Mbytes
세그먼트 크기	64 Kbytes
블록 크기	16 Kbytes
페이지 크기	512 bytes



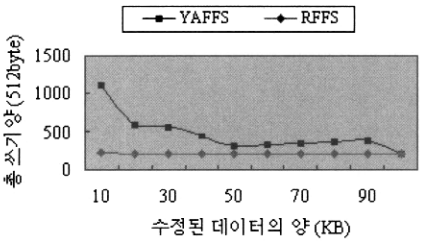
(그림 14) 쓰기 연산 성능 실험 과정

4.1.2 실험 결과

플래시 메모리의 쓰기 양을 감소시키는데 중점을 두고 있다. (그림 15)는 YAFFS와 RFFS에 대한 두 가지 작업부하에 대해 쓰기 성능 실험 결과이다. YAFFS의 쓰기양은 RFFS에 비해 추가적으로 적어주는 쓰기 양에 대해서만 나타내었다. 그림 (a)는 수정되는 데이터의 단위가 1KB인 경우이고, 수정되는 크기가 10KB가 될 때까지의 총 쓰기 양을 보여준다. 그림 (b)는 수정되는 데이터의 단위가 10KB인 경우이며, 파일의 크기가 100KB가 될 때까지의 총 쓰기 양을 보여준다. 두 경우 모두 RFFS가 우수한 성능을 보였다. 10KB와 100KB의 경우는 YAFFS는 헤더와 데이터를 적어주면 되지만 RFFS의 경우는 메타와 위치정보영역의 수정이 필요하기에 YAFFS가 나은 성능을 보여주게 된다. 전체적으로 파일의 부분에 대한 수정에서는 RFFS가 좋은 성능을 보이지만 파일 전체 수정에서는 위치 정보영역에 대한 추가 연산이 필요하게 된다.



(a) 수정된 데이터의 단위: 1KB



(b) 수정된 데이터의 단위: 10KB

(그림 15) 수정된 데이터의 양에 따른 총 쓰기 횟수

4.2 로그 모듈 성능 실험

4.2.1 시뮬레이션과 작업부하의 설계

로그 구조의 성능 평가를 위해서 메모리상의 힙 영역을 24MB 할당하여 플래시 메모리처럼 사용하였다. YAFFS는 저널링 기능을 제공하지 않기 때문에 JFFS2와 RFFS의 저널링 성능을 비교하였다. 두 가지 실험으로 저널링 성능을 비교하였다. 첫째, 플래시 메모리의 사용량에 따른 마운트 성능을 비교하였다. 파일 시스템의 사용량을 20%, 50%, 70%

로 늘려가며 마운트 시간을 측정하였다. 두 번째 실험은 로깅에 따른 오버헤드를 측정하기 위해 로깅 기능을 사용할 경우와 사용하지 않을 경우에서의 쓰기 속도를 비교하였다. 이 경우 512KB, 1024KB의 파일을 10회 복사하여 그 평균 시간을 측정하였다.

4.2.2 실험 결과

〈표 6〉 플래시 메모리의 사용량에 따른 마운트 속도

사용량	20%	50%	70%
JFFS	636ms	635ms	656ms
RFFS	136ms	209ms	320ms

첫 번째 실험결과 JFFS의 경우 사용량에 관계없이 전체 메모리를 스캔하므로 그 시간이 거의 일정했다. RFFS에서는 실제 유효한 로그 영역이나 메타 영역만을 스캔하며, 사용량에 따라 로그 영역이나 메타 영역의 양이 늘거나 감소하므로 그에 따른 약간의 성능 차이가 있다. 사용량이 20%인 경우에 5배, 50%인 경우 3배, 70%인 경우 2배의 성능향상이 있다.

〈표 7〉 로깅 유무에 따른 평균 복사 시간

	512Kb	1024Kb
RFFS(Logging)	67ms	102ms
RFFS(no Logging)	62ms	97ms

두 번째 실험에서는 로깅을 사용할 경우 로그 파일을 기록하기 위해 약간의 오버헤드가 있지만, 그 차이가 미미하기 때문에 파일 시스템의 성능에 크게 영향을 미치지 않음을 알 수 있다. 따라서 로깅을 수행함으로써 파일 시스템의 빠른 회복을 통한 일관성 유지를 보장한다.

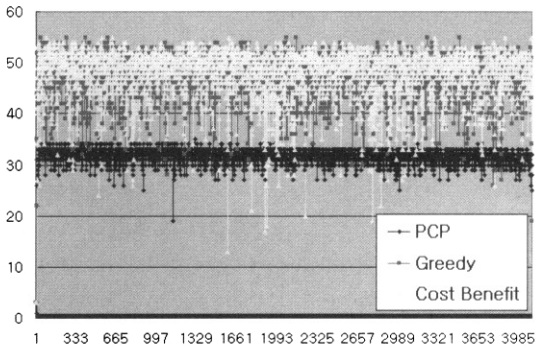
4.3 지움 정책 성능 실험

4.3.1 시뮬레이션과 작업부하의 설계

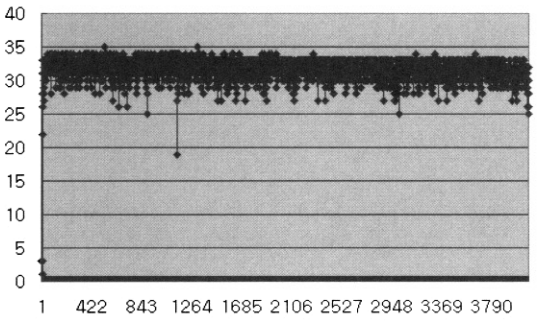
RFFS의 플레인 지움 정책에 대하여 성능을 평가하기위해 64MB 용량의 플래시메모리를 기반으로 시뮬레이션 하였다. 초기에 플래시 메모리 공간에 40%의 데이터를 저장시킨 후, 새로운 데이터를 갱신하는 방법으로 테스트를 수행하였다. YAFFS는 log-structured file system 방식으로 사용하기에 비교할 수 없으므로, 기존의 지움 정책인 Greedy와 Cost-benefit, RCP를 비교 대상으로 하였다. Greedy와 Cost-benefit 방법에서 임계값 T_h는 한 세그먼트에 저장 가능한 128개의 페이지 중 115개를 기준으로 하였으며, N = 12로 설정하였고, RCP 방법은 N1 = 2, N2 = 12로 두었다. PCP 방법에서도 다른 방법과의 동일한 실험 환경설정을 위해 Th1 = 15, Th2 = 12 로 설정 하였다. 데이터 갱신은 난수를 발생시켜 갱신하였다[6, 15].

4.3.2 실험 결과

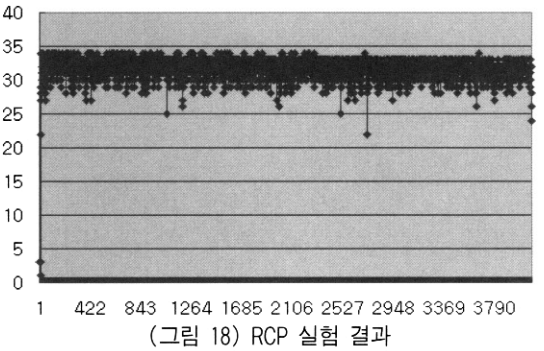
(그림 16)에서 각 블록의 지움 횟수에 대한 성능을 보여



(그림 16) Greedy, Cost-Benefit과의 성능 비교



(그림 17) PCP 실험 결과



(그림 18) RCP 실험 결과

주고 있다. Greedy 방법은 지움 횟수에 대하여 Cost Benefit 방식과 유사한 성능을 보이고 있으나, 균등 사용 정책에 있어서 좋은 성능을 보이고 있다.

(그림 17), (그림 18)은 40% 초기 데이터에 대한 PCP 방법과 RCP 방법에 대한 지움 회수의 성능평가를 보이고 있다. PCP 방법은 각 블록의 지움 횟수와 균등 사용 면에서 RCP와 매우 유사한 성능을 보이고 있으나, 계산 복잡도를 고려한 비용에 있어서 본 논문에서 제안한 기법이 우수한 성능을 보인다.

<표 8>은 플래시 메모리의 쓰기 작업인 write(), cleaning()과정을 나타낸다. PCP 방법의 계산 복잡도는 $\theta(N) = 6N + 1$, RCP방법의 경우는 $\theta(N) = 12N$ 으로 cleaning() 수행 시 약 2배의 차이를 나타낸다. 복잡도의 차이는 각 세그먼트의 비율에 대한 연산으로 인한 차이다. PCP에서는 Valid Page의 비율과 사용하지 않은 빈 공간의 비율을 이용하여, 최소의 지움 비용을 가지는 세그먼트를 선택한다. 이

러한 지움 비용을 고려한 계산 복잡도의 단순화는 플래시 메모리의 수명을 연장시킨다.

〈표 8〉 write(), cleaning() 과정

```
write(){
  Allocate a free block;
  If new write
    Write data into the free block
  else{
    Mark the obsolete data as invalid;
    Write data into the free block;
  }
}
cleaning(){
  Select a victim segment for cleaning;
  Identify valid data in the victim segment;
  Copy out valid data to another clean flash memory
  Erase the victim segment;
}
```

5. 결 론

본 논문에서는 휴대폰, MP3, 디지털 카메라와 같은 다양한 휴대용 기기의 특성을 고려하여 파일 시스템 마운팅 시간의 최소화, 시스템 폴트 발생 시 시스템의 일관성 제공, 플래시 메모리의 균등 사용을 위한 지움 정책 제공, 파일 쓰기 연산 수행 시 플래시 메모리의 사용량을 감소시키는 NAND 플래시용 파일 시스템 RFFS을 설계하고 각 기능별 성능을 비교 분석했다.

메타 데이터의 구조를 변경하여 일부만 수정되어도 모든 데이터를 쓰던 YAFFS에 비해 기 횟수가 최고 90%까지 감소하였고, 로그 구조로 YAFFS의 문제점인 파일 내용의 일관성을 지원했고, JFFS에 비해 좋은 성능을 보임을 확인했다. 그리고 임베디드 시스템의 제한적인 자원을 고려한 플레인 지움 정책으로 균등 사용을 지원하면서 전체 시스템의 부하를 줄였다.

시뮬레이션 결과를 바탕으로 설계된 각 모듈을 개발하였으며 모듈을 통합하여 전체 파일 시스템을 구축할 예정이다. 현재 설계된 플래시 메모리는 최대 4GB까지 지원 가능하므로 디스크를 대체하는 대용량 메모리를 지원하기 위해서 파일시스템을 확장할 예정이며, 효율적인 메모리 사용을 지원하는 FRAM(Ferroelectric RAM), MRAM(Magnetic RAM) 등의 차세대 비활성메모리를 이용하여 더욱 신뢰성 높은 파일시스템의 개발을 향후 연구 내용으로 남겨둔다.

참 고 문 헌

[1] White Paper: Two Technologies Compared: NOR vs. NAND, "www.m-sys.com/NR/rdonlyres/24795A9E-16F9-404A-857C-C1DE21986D28/229/NO R_vs_NAND5.pdf"

[2] 동아일보 2005년 9월 13일 "삼성전자 16기가 낸드 플래시메모리 세계 첫 개발 http://news.naver.com/news/read.php=LSD&office_id=020&article_id=0000315605&section_id=101&menu_id=101".

[3] 김한준, 이상구 "신뢰성 있는 플래시메모리 저장시스템 구축을위한 플래시메모리 저장 공간 관리방법", 정보과학회논문지: 시스템 및 이론, 제27권 제6호, 2000년 6월.

[4] 김정기, 박승민, 김채규, "임베디드 플래시 파일 시스템을 위한 순위별 지움 정책", 정보처리학회논문지 제9-A권 제4호, 2002년 12월.

[5] M. L. Chang, P. C. H. Lee, R. C. Chang, "Manageing Flash Memory in Personal Communication Devices," Proc. of IEEE Symp. on Consumer Electronics, pp.177-182, 1997.

[6] Mei-Ling Chiang, Paul C. H. Lee Ruei-Chuan Chang, "Cleaning Policies in Mobile Computers Using Flash Memory," journal of System and Software.

[7] A Space-efficient Flash Translation Layer for CompactFlash Systems, Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, IEEE Transactions on Consumer Electronics, May, 2002.

[8] Understanding the Flash Translation Layer(FTL) specification. Intel: 1997.

[9] WindRiver, "TrueFFS for Tornado Programmer's Guide 1.0," 1999.

[10] David Woodhouse, "JFFS: The Journaling Flash File System," Technical Paper of RedHat inc. Oct., 2001.

[11] YAFFS Spec, http://www.aleph1.co.uk/yaffs/yaffs.html.

[12] M.Resenblum and J.K.Ousterhout, "The Design and Implementation of a Log-Structured File System," ACM Transaction on Computer Systems, Vol.10, pp.26-52, 1992.

[13] Richard Menedetter, "Journaling Filesystems for Linux."

[14] Margo Seltzer, Keith Bostic "An Implementation of a Log-Structured File System for UNIX." Winter USENIX, January, 25-29, 1993.

[15] Mendel Rosenblum, "The Design and Implementation of a Log-Structured File System."

[16] Atsuo Kawaguchi, Shingo Nishioka, Hiroshi Motoda, "A Flash-Memory based File System," Proceedings of 1995 USENIX Technical Conference, pp.155-164, 1995.

[17] 이태훈, 이상기, 정기동, "임베디드 플래시 파일 시스템을 위한 플레인 지움 정책", 정보과학회 2005 가을학술발표논문집.

[18] 박송화, 이주경, 정기동, "NAND 플래시 메모리 파일 시스템의 설계", 정보과학회 2005 가을학술발표논문집.

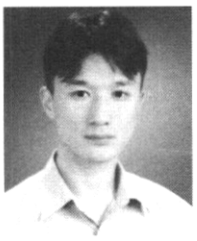
[19] 김태훈, 이주경, 정기동, "NAND 플래시 파일 시스템의 빠른 복구를 위한 로그 구조 설계", 정보과학회 2005 가을학술발표논문집.

이 태 훈

e-mail : withsoul@melon.cs.pusan.ac.kr
2004년 부산대학교 정보컴퓨터공학과(학사)
2004년~현재 부산대학교 정보컴퓨터공학과 공학석사
관심분야 : 임베디드 시스템, 무선 네트워크, 파일 시스템 등

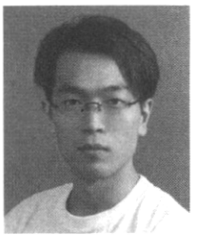
박 송 화

e-mail : downy25@melon.cs.pusan.ac.kr
2005년 부산대학교 정보컴퓨터공학과(학사)
2005년~현재 부산대학교 정보컴퓨터공학과 공학석사
관심분야 : 파일 시스템, 멀티미디어, 임베디드 시스템 등



김 태 훈

e-mail : rider79@melon.cs.pusan.ac.kr
2005년 동서대학교 컴퓨터공학과(학사)
2005년~현재 부산대학교 정보컴퓨터공학과 공학석사
관심분야 : 임베디드 시스템, 영상처리 등



이 상 기

e-mail : nick@melon.cs.pusan.ac.kr
2005년 부산대학교 정보컴퓨터공학과(학사)
2005년~현재 부산대학교 정보컴퓨터공학과 공학석사
관심분야 : 임베디드 시스템, 파일시스템, 네트워크 등

이 주 경

e-mail : jklee@melon.cs.pusan.ac.kr
1996년 부산대학교 전자계산학과(학사)
1998년 부산대학교 전자계산학과(석사)
1998년~2001년 한국전력공사 근무
2001년~2005년 부산대학교 컴퓨터공학과 (공학박사)
관심분야 : 멀티미디어 데이터 압축, 오류제어, 임베디드 시스템

정 기 동

e-mail : kdchung@pusan.ac.kr
1973년 서울대학교(학사)
1975년 서울대학교(석사)
1986년 서울대학교 계산통계학과(이학박사)
1990년~1991년 MIT, South Carolina 대학 교환교수
1995년~1997년 부산대학교 전자계산소 소장
1978년~현재 부산대학교 전자계산학과 교수
1997년~현재 부산대학교 멀티미디어 협동과정학과 교수
관심분야 : 병렬처리, 멀티미디어, 임베디드 시스템