낸드 플래시 메모리를 위한 자기-서술 파일 시스템
(A Self-Description File System for NAND Flash Memory)

한 준 영 † 박 상 오 † 김 성 조 ††
(Jun-yeong Han) (Sang-oh Park) (Sung-jo Kim)

요 약 낸드 플래시 메모리는 하드디스크 드라이브와 물리적 특성이 다르기 때문에, 기존 하드디스크 드라이브를 위한 파일 시스템을 낸드 플래시 메모리에서 그대로 사용할 수 없다. 이 문제를 해결하기 위해 낸드 플래시 메모리 전용 파일 시스템이 개발되었으나 파일의 메타 정보를 파일 데이타와 분리하여 저장하는 구조 때문에, 파일이 쓰여질 때마다 파일의 메타 정보가 저장된 페이지를 갱신하는 오버헤드가 존재한다. 또한, 파일 시스템이나 파일 자체의 메타 정보가 저장된 페이지가 손실되었을 때, 파일 시스템이 실패하게 되는 안정성의 문제가 있다. 본 논문에서는 이와 같은 효율성 문제와 안정성 문제를 해결하기 위해 자기 서술 페이지(Self-Description Page) 기법과 메모리 상의 코어 파일 시스템(In Memory Core File System) 기법을 제안한다. 이 기법을 적용하여 새롭게 개발한 SDFS(Self-Description File System)에서는 낸드 플래시 메모리 내의 일부 페이지들이 실패하더라도 파일 시스템을 안전하게 복구할 수 있으며, YAFFS2보다 쓰기와 읽기 성능을 각각 평균 36%, 15% 향상시켰고, 마운트 시간을 최대 1/20까지 단축시켰다.

키워드 : 낸드 플래시 메모리, 파일 시스템, 자기-서술 페이지, 메모리 상의 코어 파일 시스템

Abstract Conventional file systems for harddisk drive cannot be applied to NAND flash memory, because the physical characteristics of NAND flash memory differs from those of harddisk drive. To address this problem, various file systems with better reliability and efficiency have also been developed recently. However, those file systems have inherent overheads for updating the file's metadata pages, because those file systems save file's meta-data and data separately. Furthermore, those file systems have a critical reliability problem: file systems fail when either a page in meta-data of a file system or a file itself fails. In this paper, we propose a self-description page technique and In Memory Core File System technique to address these efficiency and reliability problems, and develop SDFS(Self-Description File System) newly. SDFS can be safely recovered, although some pages fail, and improves write and read performance by 36% and 15%, respectively, and reduces mounting time by 1/20 compared with YAFFS2.

Key words : NAND Flash Memory, File System, Self-Description Page, In Memory Core File System

· 2007학년도 교내학술연구비 지원에 의한 것임

† 학생회원 : 중앙대학교 컴퓨터공학과
hanjunyeong@konan.cse.cau.ac.kr
sj1st@konan.cse.cau.ac.kr
†† 종신회원 : 중앙대학교 컴퓨터공학과 교수
sjkim@cau.ac.kr

논문접수 : 2008년 4월 28일
심사완료 : 2008년 12월 9일

정보과학회논문지: 컴퓨팅의 실제 및 레터 제15권 제2호(2009.2)

1. 서 론

낸드 플래시 메모리의 물리적 특성은 하드디스크 드라이브와 큰 차이가 있기 때문에, 기존의 하드디스크 드라이브 기반의 파일 시스템은 낸드 플래시 메모리상에서 그대로 사용될 수 없다. 이를 해결하기 위해 FTL (Flash Translation Layer)[1-3]이 소개되어, 플래시 메모리를 하드디스크 드라이브처럼 사용할 수 있게 되었고 FTL은 현재 상용 USB 메모리에 내장되어 널리 사용되고 있다. 그러나 낸드 플래시 메모리를 하드디스크 드라이브처럼 보이게 하는 변환 계층은 많은 CPU 시간과 RAM을 사용하기 때문에 추가적인 자원이 요구되며 성능 상의 오버헤드를 가지고 있다.

FTL의 성능 상의 문제는 낸드 플래시 메모리 전용 파일 시스템 개발의 동기가 되었고, 그 결과 JFFS[4], JFFS2[4]와 YAFFS[5], YAFFS2[5]와 같은 낸드 플래시 메모리를 위한 파일 시스템이 개발되어 널리 사용되고 있다. 또한 낸드 플래시 메모리의 특성을 고려하여 마운트, 읽기/쓰기 연산, 가비지 컬렉션, 마모도 평준화(wear-leveling) 등에서 성능을 개선시키기 위한 연구가 활발하게 이루어지고 있다[6-13]. 그러나 이런 낸드 플래시 메모리 전용 파일 시스템들은 파일 시스템에 대한 메타 정보를 낸드 플래시 메모리의 일정 페이지에 저장하고, 파일 자체에 대한 헤더 또한 낸드 플래시 메모리의 일정 페이지에 저장하는 구조에 기반을 두고 있는데, 이는 다음과 같은 두 가지 중요한 문제점을 가지고 있다.

첫째, 파일 시스템의 메타 정보, 또는 파일 헤더가 저장된 페이지가 손상되었을 때 파일 시스템 전체가 실패하거나 파일을 잃어버릴 수 있는 문제이다. 파일 시스템의 메타 정보가 손실 되었을 때, 낸드 플래시 메모리 전체를 스캔하여 파일 시스템을 복구[4]하거나 저널링 기법을 통해 빠른 시간에 파일 시스템을 복구하는 방법[13]이 연구되었지만 파일에 대한 메타 정보를 저장하고 있는 파일 헤더가 손상될 경우, 그 파일을 정상적으로 복구할 수 없다. 저널링을 위한 정보 역시 낸드 플래시 메모리의 특정 페이지에 저장되는데, 이 저널링 정보가 손상되면 파일 시스템을 복구할 수 없다. 또한 쓰기 연산이 일어날 때마다 저널링을 위한 정보를 낸드 플래시 메모리에 써야 하기 때문에 저장 공간을 충분히 활용하지 못하는 문제와 쓰기 연산 시 추가적으로 저널링 정보를 써야 하는 오버헤드 문제가 있다.

둘째, 쓰기 연산 시 파일의 데이타 정보뿐만 아니라, 파일에 대한 메타 정보, 그리고 파일 시스템에 대한 메타 정보까지 갱신해야 하기 때문에 오버헤드가 크다. 파일이 변경될 때마다 파일에 대한 메타 정보가 변하기 때문에 파일 헤더의 수정이 필요하고, 결과적으로 파일 시스템에 대한 메타 정보의 수정도 필요하다. 낸드 플래시 메모리는 제자리 덮어 쓰기가 불가능한 하드웨어적 특성으로 변경된 파일의 내용만을 낸드 플래시 메모리에 쓰는 게 아니라 수정된 메타 정보를 낸드 플래시 메모리의 새로운 페이지를 할당하여 써야 하기 때문에 메타 정보 갱신 오버헤드도 매우 크다.

위 두 가지 문제는 파일 시스템의 가장 큰 평가 요소인 안정성과 성능 두 가지 모두에 대한 문제이다. 본 논문에서는 기존 파일 시스템이 가지고 있는 안정성과 성능의 두 가지 핵심적인 문제를 해결하기 위해 자기 서술 페이지(Self-Description Page) 기법과 메모리 상의 코어 파일 시스템(In Memory Core File System) 기법을 제안한다. 자기 서술 페이지는 낸드 플래시 메모리 각 페이지에 데이타와 그 데이타를 설명할 수 있는 메타 정보를 함께 저장함으로써 하나의 페이지가 그 페이지를 완전히 서술할 수 있는 페이지이다. 어떤 페이지들이 실패했을 때 그 페이지의 정보만이 소실될 뿐 온전한 페이지들의 정보는 정확히 설명되기 때문에, 파일 시스템은 온전한 페이지로만 구성된 파일 시스템으로 복구한다. 메모리 상의 코어 파일 시스템은 마운트 시 RAM 상에 구성된 파일 시스템의 메타 정보와 VFS에서 dentry를 생성하여 사용을 요청한 파일의 메타 정보를 언마운트 시까지 RAM 상에만 존재하도록 하는 정책으로써, 파일 및 파일 시스템의 변경 내용을 낸드 플래시 메모리에 저장하는 것을 언마운트 시점까지 미루게 된다. 이를 통해 파일 I/O 연산 시 파일 메타 정보를 갱신하는 오버헤드를 제거한다. 또한 언마운트 시 RAM 상의 코어 파일 시스템의 현재 상태(Snapshot)를 낸드 플래시 메모리의 특정 영역에 저장함으로써, 마운트 시 속도를 획기적으로 단축시킨다.

본 연구에서는 자기 서술 페이지 기법과 메모리 상의 코어 파일 시스템 기법을 적용하여 SDFS(Self-Description File System)을 새로 설계하고 구현한다. 또한, 구현된 SDFS의 안정성을 테스트하고, 마운트 시간, 읽기 및 쓰기 시간, 그리고 쓰기 지연 시간을 측정하여, 상용 시스템인 YAFFS2와 비교한다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구로서 낸드 플래시 메모리의 특징과 낸드 플래시 메모리 파일 시스템에 대한 기존 연구에 대해 알아보고, 3장에서는 자기 서술 페이지 기법과 메모리 상의 코어 파일 시스템 기법을 적용한 SDFS를 설명한다. 4장에서는 SDFS의 안정성을 테스트하고 성능을 측정하여, 그 결과를 YAFFS2와 비교한다. 마지막으로 5장에서는 결론과 함께 향후 연구과제에 대해 서술한다.

2. 관련 연구

낸드 플래시 메모리 파일 시스템은 파일 시스템이 쓰이는 기기의 특성에 따라 빠른 마운트, 효율적인 읽기·쓰기 연산, 적은 RAM 사용, 적은 비용의 가비지 컬렉션, 그리고 균등 지움 연산 등의 기법에 대해 연구[6-13]되어 왔다.

JFFS2는 데이타를 로그 형태로 낸드 플래시 메모리에 순차적으로 쓰고, 읽기 연산은 로그를 역순으로 검색하여 가장 최신의 데이타를 읽어 들인다. JFFS2에서는 저널링 노드로서 next_in_ino, next_phys, flash_offset, totlen 값으로 구성된 jffs2_raw_node_ref라는 구조체(16Byte)를 사용하여 메모리에 저장한다. 또한, 낸드 플래시 메모리의 공간을 효율적으로 활용하기 위해 데이타

압축 기능을 사용한다. 그러나 SLC-large block 낸드 플래시 메모리가 128Mbyte일 때, 최대 218개의 페이지를 필요로 하기 때문에 jffs2_raw_node_ref를 위한 공간만 222Byte, 즉 4Mbyte를 할당해야 하는 문제점이 있고, 노드를 찾고 파일을 결정하기 위한 스캔 시간이 길다. 또한, 마운트 할 때 플래시 메모리 전체를 스캔 해야 하기 때문에 상당한 시간이 걸리는데, 128Mbyte 크기의 낸드 플래시 메모리를 마운트 할 경우 25초가 걸린다. 이와 같이 JFFS2는 노어 플래시 메모리를 기반으로 설계되었기 때문에 낸드 플래시 메모리용 파일 시스템으로 사용되기에는 RAM 사용량, 마운트 시간 그리고 가비지 컬렉션 시간 등에서 여러 가지 문제점이 있다.

YAFFS2는 낸드 플래시 메모리를 위해 개발된 파일 시스템으로써 낸드 플래시 메모리에 최적화 되었으며, 읽기/쓰기/마운트 시간과 RAM 소모량 등에서 JFFS2보다 우수하다. YAFFS2는 파일을 쓰기 위해 파일의 메타데이타를 가리키는 페이지를 먼저 쓰고, 그 파일에 속한 모든 페이지의 잉여 영역에 공통된 파일 ID를 기록하며, 이들 페이지의 연관 관계를 파악하기 위해 RAM에 트리 구조를 유지한다. 부팅 시 마운트 속도가 매우 빠르고, JFFS2보다 월등히 앞선 성능을 보인다. YAFFS2는 마운트 시 파일 시스템 초기화를 위해 스캐닝 할 때 잉여영역만 읽기 때문에 JFFS2보다 빠르다. 하지만, YAFFS2는 안정성에 큰 문제가 있다. 파일을 수정하는 도중에 파일 시스템이 실패하게 되면, 수정하기 전 파일의 페이지가 삭제되고, 갱신된 파일의 페이지가 남는다. 따라서, 파일에 수정된 페이지와 수정되지 않은 페이지가 존재하게 되어 파일의 일관성(Consistency)에 문제가 생긴다.

MNFS(Mobile Multimedia File System for NAND Flash based Storage Device)[9]는 휴대용 멀티미디어 기기의 반영구 저장소인 낸드 플래시 메모리를 위한 파일 시스템이다. 멀티미디어 파일은 그 크기가 크며, 주로 순차적으로 읽혀지고, 파일의 데이타는 자주 수정되지 않는다는 전제하에 블록 단위로 저장하고, 파일의 메타데이타는 페이지 단위로 저장한다. 그리고 파일 삭제 시 그 파일이 저장된 블록을 삭제하여 클린 상태로 만들어서 쓰기 연산 시 클린 상태의 블록을 확보하지 않아도 되기 때문에 일정한 쓰기 지연 시간을 보장할 수 있다. 블록 단위 읽기/쓰기 연산을 통해 성능을 향상시켰지만, 작은 파일을 하나의 블록에 할당하는 경우가 많이 발생하는 일반적인 모바일 기기를 지원하기에는 저장 공간 이용의 효율성 문제가 존재한다.

CFFS(Core Flash File System)[10]은 파일 시스템의 메타 정보와 파일 데이타 정보를 구분하여 서로 다른 정책에 따라 효율적으로 관리한다. 파일 크기에 따라 파일이 저장된 페이지의 인덱스를 트리 구조로 다루는데, 하나의 파일 인덱스를 저장하는 index entry의 크기는 낸드 플래시 메모리 페이지의 크기와 동일하고, 파일의 크기가 하나의 index entry로 나타낼 수 없을 때는, index entry를 트리 구조로 구성한다. 이 파일 시스템은 파일이 쓰여질 때, 수정된 index entry를 낸드 플래시 메모리에 써야 하는 오버헤드가 존재한다.

위 파일 시스템들은 어느 정도 안정적이고 효율적으로 작동하지만, 낸드 플래시 메모리 페이지들의 데이타가 깨졌을 때 파일 시스템 전체가 실패할 수 있다. 왜냐하면 안정성을 보장하기 위한 기법이 존재하지만, 그것은 낸드 플래시 메모리의 모든 페이지가 온전하여 모든 데이타를 정확히 읽을 수 있을 때에만 보장되기 때문이다. 그리고 안정성을 보장하기 위해 파일이 새로 쓰여지거나 수정될 때마다 수정 사항에 대한 로그 정보나 수정된 파일의 메타 정보를 갱신하는 오버헤드가 존재한다.

최근에는 NAND 플래시 메모리의 특성으로 인한 제약을 보완하기 위하여 덮어 쓰기가 가능한 비휘발성 메모리인 FRAM(Ferro-electric RAM)[14]을 이용한 NAND 플래시 메모리 전용 파일 시스템 [15]이 제안되었다. FRAM을 이용한 파일시스템들은 많은 수정이 이루어지는 파일 시스템의 메타 정보와 데이타 정보를 FRAM에 저장함으로써 마운트 시 FRAM에서 필요한 정보를 얻을 수 있어서 빠른 마운트를 제공할 수 있으며, 많이 수정되는 정보를 FRAM에 저장함으로써 NAND 플래시 메모리의 수명도 늘릴 수 있다. 그러나 FRAM을 이용하더라도 메타 정보와 데이타 정보 등을 저장하는 중 전력이 끊어졌을 경우 안정성을 보장하기가 어렵다. 따라서 추가 보안 체계가 필요로 한다. 또한, FRAM을 사용함으로써 NAND 메모리에 비해 추가 비용이 요구된다.

3. 안정하고 효율적인 파일 시스템

기존 낸드 플래시 메모리 파일 시스템에서 페이지에 저장된 데이타가 소실되는 경우가 발생했을 때, 그 페이지에 파일 시스템을 구성하는 핵심 정보가 저장되어 있다면 파일 시스템 전체가 실패하게 된다. 파일 시스템을 더욱 안정화 하기 위해 저널링에 필요한 정보를 낸드 플래시 메모리에 저장하더라도, 그 정보가 저장된 페이지가 실패한다면 역시 파일 시스템을 복구할 수 없다. 낸드 플래시 메모리 페이지 중 일부가 실패 하더라도 나머지 페이지 정보만을 통해 파일 시스템을 복구할 수 있기 위해서는, 페이지가 스스로 자신(Self)이 저장하고 있는 데이타를 서술(Description)함으로써 페이지에 저장된 정보의 의미를 그 페이지만 보고서 알 수 있어야 한다. 이러한 페이지를 자기 서술 페이지(Self-Des-

cription Page)라고 한다. SDFS(Self-Description File System)는 낸드 플래시 메모리 페이지에서 2Kbyte의 저장 영역에는 데이타를 저장하고, 64Byte의 잉여 영역에는 저장 영역에 저장된 데이타를 서술하는 메타 정보를 저장함으로써 페이지가 실패하더라도 실패한 페이지 정보를 빼고 전체 파일 시스템을 복구할 수 있다.

자기 서술 페이지는 inode 정보를 특정 페이지를 할당하여 저장하지 않고, 모든 페이지에 inode 정보를 분산하여 저장하는 구조이다. 따라서 파일 시스템을 관리하기 위해 낸드 플래시 메모리에 분산 저장된 inode 정보를 관리해야 하기 때문에 효율성이 떨어진다. 이를 해결하기 위해 SDFS는 파일 시스템 전체 정보와 파일들의 메타 정보를 RAM 상에 자료구조로 관리하고 이를 통해 연산을 수행하는데, 이렇게 RAM 상에 구성된 자료구조를 코어 파일 시스템(Core File System)이라고 한다. 코어 파일 시스템은 RAM의 용량이 허용하는 한, 마운트 시 RAM에 구성된 후 언마운트 시까지 RAM 상에서만 존재하게 되는 메모리 상의 코어 파일 시스템(In Memory Core File System)이다. SDFS는 마운트 시 파일 시스템의 구성에 필요한 정보를 낸드 플래시 메모리로부터 읽어서 RAM에 구성하는데 마운트를 빠르게 하기 위해 낸드 플래시 메모리에 언마운트 시 RAM 상의 코어 파일 시스템 정보를 그대로 저장(Snapshot)한다. 그리고 파일이 갱신될 때, RAM 상에 구성된 코어 파일 시스템 정보만을 수정하는데 이는 파일의 쓰기 연산 시 순수 데이타만 낸드 플래시 메모리에 쓰고 메타 정보는 쓰지 않기 때문에 파일 쓰기 연산을 최적화 할 수 있다. 비정상적인 언마운트로 RAM 상의 코어 파일 시스템을 낸드 플래시 메모리에 저장하지 못하더라도, 자기 서술 페이지 기법을 통해 파일 시스템을 완전히 복구할 수 있다.

SDFS는 자기 서술 페이지와 메모리 상의 코어 파일 시스템 기법이 최적으로 동작할 수 있도록, 기존 낸드 플래시 메모리 파일 시스템과 달리 전혀 새롭게 구현된 파일 시스템이다. 하지만 가독성(Readability)을 높이기 위해, yaffs가 jffs의 뼈대를 일부 사용한 것과 같이, SDFS는 yaffs2의 뼈대를 일부 사용하였다. 또한 SDFS는 페이지의 크기가 2110Byte(2048Byte(저장영역) + 64Byte(잉여영역))인 SLC-large block 낸드 플래시 메모리만을 지원하도록 구현되었다.

3.1 파일 시스템의 전체 구조

낸드 플래시 메모리의 각 페이지는 기본적으로 자기 서술 페이지 형식으로 저장되는데, 낸드 플래시 메모리 전체 블록 중 일부는 코어 파일 시스템을 저장하고 나머지 영역에는 파일 데이타를 저장한다. 그림 1은 낸드 플래시 메모리에 저장된 데이타의 구조를 설명한다.

파일 시스템에 존재하는 파일은 메타정보와 데이타를 가진 정규 파일로 가정한다. 코어 파일 시스템이 저장되는 영역의 크기는 아래 (1), (2), (3)으로부터 유도된다.

$$AP(F) >= size(P) + size(F) \quad (1)$$

$$CA(F) >= 44(Byte) + size(F)/(size(P)/4) \quad (2)$$

$$(AP(F) + CA(F))/CA(F) >= 86.3$$
(단, 파일의 크기는 1Byte 이상) (3)

AP(F): 정규 파일의 데이타가 할당되는 낸드 플래시 메모리 페이지의 크기
CA(F): 정규 파일의 메타 정보의 크기
size(F): 파일이 할당된 낸드 플래시 메모리 페이지들의 총 크기
size(P): 낸드 플래시 메모리에서 한 페이지 크기

(1)에서 AP(F)가 파일 크기에 상관없이 size(P)를 차지하는 이유는 파일의 이름을 낸드 플래시 메모리의 한 페이지에 저장하기 때문이다. 낸드 플래시 메모리의 잉여영역에 파일 이름을 저장할 수 있는 공간은 20Byte이기 때문에 긴 파일 이름을 지원하기 위해 파일 이름을 저장하기 위한 페이지를 할당한다. (2)에서 44Byte는 inode에서 이름을 뺀 정보의 크기이며, size(F)/(size(P)/4)는 파일 데이타가 낸드 플래시 메모리에 저장된 주소를 위한 공간이다. 주소의 크기가 4Byte이기 때문에 size(F)의 크기가 size(P)의 크기인 2KByte 커질 때마다 4Byte씩 증가하게 되며 이것은 size(F)/(size(P)/4)

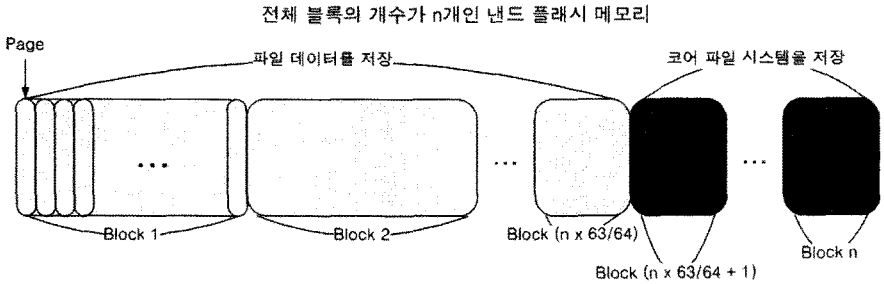


그림 1 낸드 플래시 메모리에 저장된 데이타의 구조

이다. (3)으로부터 낸드 플래시 메모리 전체 페이지 중에서 1/86의 공간만 있으면 코어 파일 시스템을 저장할 수 있음을 알 수 있고, 이를 2의 지수 승으로 비율을 조정하면 1/64가 된다.

SDFS가 마운트 되면 RAM 상에 코어 파일 시스템이 적재되어 파일 시스템을 관리하고, 연산을 수행한다. 그림 2는 SDFS가 마운트 되어 RAM 상에 코어 파일 시스템이 적재된 모습이다.

슈퍼 블록은 파일 시스템 전체에 대한 정보를 가지고 있는 자료 구조로서 낸드 플래시 메모리에 마운트 된 파일 시스템의 현재 정보를 저장한다. 또한 낸드 플래시 메모리의 페이지가 사용 중인지 아닌지를 나타내는 비트맵 정보를 저장하며, 파일 시스템 inode 리스트를 관리한다. 파일 시스템 inode는 VFS(Virtual File System) inode의 정보와 서로 변환이 가능하며, 추가로 파일의 데이타가 저장된 낸드 플래시 메모리의 페이지 주소 정보를 가지고 있다. 파일 시스템 inode가 낸드 플래시 메모리에 저장된 위치를 B-tree로 구성하여 RAM 상의 inode 정보를 낸드 플래시 메모리에 저장하거나 읽어올 수 있다.

그림 3과 그림 4는 정규 파일과 디렉토리의 파일 시스템 inode가 RAM 상에 적재되어 있는 상세 정보를 나타낸다. 정규 파일의 파일 시스템 inode는 파일의 속성에 대한 정보와 데이타가 저장된 낸드 플래시 메모리의 주소 정보를 저장한다. 디렉토리의 파일 시스템 inode는 데이타가 저장된 낸드 플래시 메모리의 주소와 자식 파일 시스템 inode들의 포인터를 저장하며, 항상

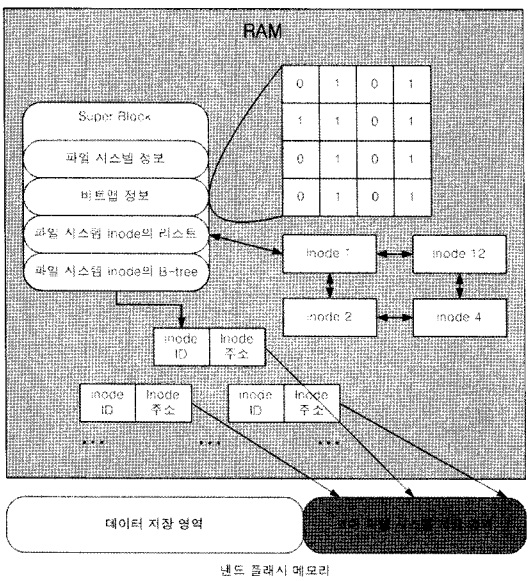


그림 2 RAM 상에 코어 파일 시스템이 적재된 모습

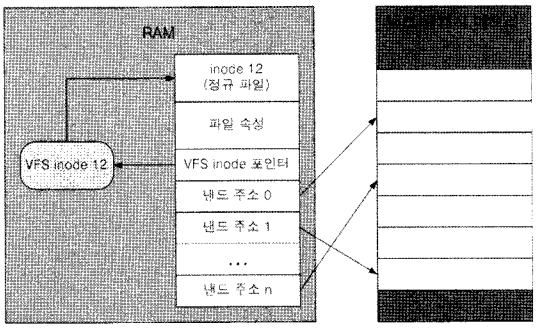


그림 3 정규 파일의 파일 시스템 inode 구조

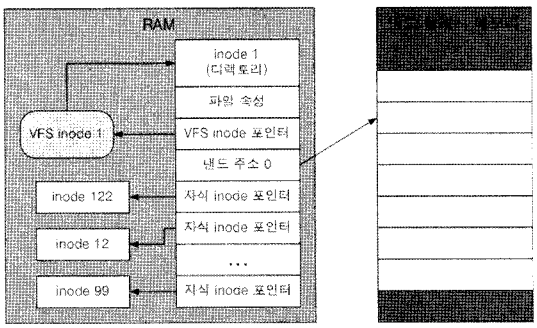


그림 4 디렉토리 파일의 파일 시스템 inode 구조

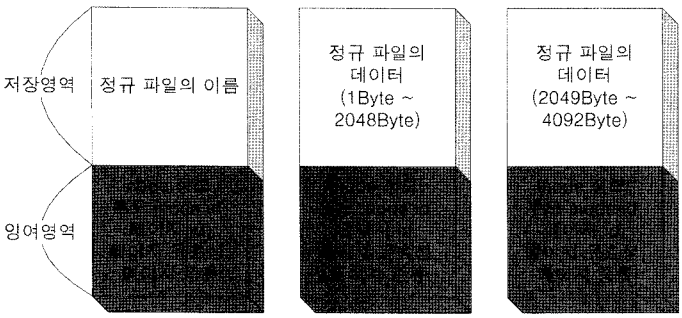
낸드 플래시 메모리의 한 페이지에 저장된다. 그리고 파일 시스템 inode와 대응하는 VFS inode의 포인터를 가지고 있어, VFS와 서로 정보를 주고받는다. SDFS에서 페이지(2Kbyte) 주소를 _u32형의 변수로 사용하기 때문에 약 (2Kbyte * (2^32-1))크기(약 8000Tbyte)의 파일까지 지원한다. 디렉토리의 경우도 _u32형의 변수를 사용하기 때문에 약 40억 개까지의 자식 파일을 가질 수 있으며, SDFS는 _u32의 inode id를 사용하기 때문에 약 40억 개의 파일이 존재할 수 있다.

3.2 자기 서술 페이지(Self-Description Page)

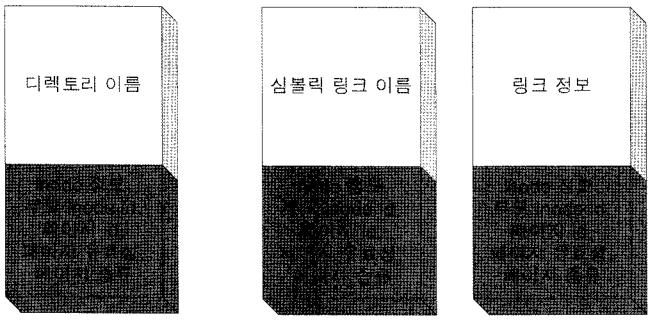
자기 서술 페이지(Self-Description Page)는 한 페이지의 저장 영역에 데이타를 저장하고, 그 페이지의 잉여 영역에 저장 영역에 저장된 데이타를 설명하는 정보를 저장하는 페이지 말한다. 즉, 자기 서술 페이지에는 그 파일에 대한 inode 정보, 페이지 저장 영역에 저장된 데이타가 파일의 어느 부분을 나타내는 지에 대한 정보, 페이지의 유효성, 그리고 페이지에 저장된 정보의 종류 등이 저장된다. 이와 같이 자기 서술 페이지는 페이지가 자신의 데이타를 설명함으로써 추가적인 메타 정보 없이도 파일 시스템에 관한 필수적인 정보를 제공한다.

3.2.1 구조

SDFS에 저장되는 파일은 그림 5와 같이 정규 파일(a), 디렉토리(b) 또는 심볼릭 링크(symbolic link)(c)에



(a) 정규 파일



(b) 디렉토리 (c) 심볼릭 링크

그림 5 SDFS에서 파일의 페이지 저장 구조

상관없이 자기 서술 페이지 형태로 저장된다. 그리고 모든 페이지의 잉여 영역은 파일의 종류와 저장된 데이타의 성격에 관계없이 똑같은 자료구조로 값만 다르게 저장된다.

페이지의 잉여 영역에는 inode 정보, 부모 inode의 번호, 페이지 id, 페이지의 유효성, 페이지의 종류에 대한 정보가 저장된다. 정규 파일(그림 5(a))의 경우, 하나의 페이지는 파일 이름을 위해 할당되고, 나머지는 데이타를 위해 할당된다. 하나의 정규 파일을 위해 할당되는 페이지의 개수(PN)는 파일 크기(FS), 페이지 크기(PS)를 이용하여 (4)와 같다.

$$PN=1+\lceil FS/PS \rceil \quad (4)$$

디렉토리(그림 5(b))의 경우, 모든 페이지의 잉여영역에는 페이지에 저장된 데이타의 부모 inode에 대한 정보가 저장되기 때문에 디렉토리 이름 저장을 위해 하나의 페이지가 할당된다. 디렉토리는 자식 inode 번호를 저장하지 않는다. 왜냐하면 자식 inode 번호를 저장하면 파일이 생성 및 삭제 될 때마다 디렉토리를 수정해야 하고, 이것은 한 페이지 이상의 쓰기 연산을 필요로 하기 때문이다. 따라서 각 페이지의 잉여 영역에 부모 디렉토리의 번호를 저장함으로써 디렉토리는 디렉토리 이름만을 하나의 페이지에 저장한다. 마지막으로, 심볼릭 링크(그림 5(c))의 경우, 심볼릭 링크 이름을 저장하기 위해 하나의 페이지가 할당되고, 링크 정보 저장을 위해 다른 하나의 페이지가 할당된다.

3.2.2 잉여 영역 자료 구조

특정 페이지에 저장된 데이타를 완전히 서술하는 메타 정보가 저장된 잉여 영역의 크기는 64Byte이다. 표 1은 잉여 영역에 저장되는 구조체이다.

i_ino 필드에는 inode 번호가 저장되며, i_mode에는 inode가 나타내는 파일 종류와 권한이 저장된다. i_atime, i_mtime 그리고 i_ctime 필드에는 각각 파일의 마지막 접근 시간, 마지막 수정 시간, 생성 시간이 저장한다. i_uid 필드에는 파일 주인의 권한, i_gid 필드에는 파일 그룹의 권한이 각각 저장되며, i_size 필드에는 파일의 크기, i_usage_size에는 페이지에서 실제 데이타가 쓰여진 공간의 크기가 저장된다. i_parent_ino는 부모 디렉토리의 inode 정보를 저장하며, 이 필드로 인해 디렉토리는 자식 inode를 저장할 필요가 없다. i_page_state 필드는 페이지의 상태(유효, 무효 또는 클린) 를 저장하고, type 필드는 inode의 타입(정규 파일, 디렉토리, 심볼릭 링크)을 저장한다. version은 해당 페이지가 갱신될 때마다 1씩 증가하여, mtime이 같을 경우 최근에 갱신된 페이지를 구분하는데 사용된다. reserved 필

표 1 잉여 영역 구조체

```
struct sdfs_Spare
{
    // inode 필드
    __u32 i_ino;
    __u32 i_mode;
    __u32 i_atime;
    __u32 i_mtime;
    __u32 i_ctime;
    __u32 i_uid;
    __u32 i_gid;
    __u32 i_size;
    __u32 i_usage_size;
    __u32 i_parent_ino;
    // inode에서 해당하는 페이지 번호
    __u32 i_page_id;
    // 페이지 상태 정보
    __u32 i_page_state;
    // 페이지 종류
    int type;
    __u8 version;
    __u8 reserved[4];
    __u8 ecc[7];
};
```

드는 64Byte의 자료 구조를 만들기 위해 추가되었고, ecc 필드는 에러 검출을 위해 할당되었다.

3.2.3 페이지 실패 시 복구 방법

파일 이름이 저장된 페이지가 소실되었을 때, 파일 이름을 inode의 ID로 설정하는데 이는 이 ID가 파일 시스템에서 유일한 값으로써 이름이 충돌하는 경우를 방지할 수 있기 때문이다. 정규 파일에서 데이타가 저장된 페이지가 소실되었을 때, 소실된 데이타의 영역이 차지하는 부분을 홀(hole)로 처리한다. 어떤 파일의 부모 디렉토리 정보를 찾을 수 없을 경우에는 root 디렉토리를 부모 디렉토리로 설정한다.

3.3 메모리 상의 코어 파일 시스템(In Memory Core File System)

코어 파일 시스템은 파일 시스템을 효율적으로 관리하기 위해, 파일 시스템 전체에 대한 메타 정보와 파일 시스템 inode들을 저장하는 자료구조이다. 이 코어 파일 시스템은 파일 시스템 inode를 제외하고, 마운트 시 RAM에 생성되고 갱신되며, 언마운트 시 낸드 플래시 메모리에 저장되기 때문에 RAM 상에 존재하는 코어 파일 시스템(In Memory Core File System)이다. 파일 시스템 inode들은 파일이 참조될 때 낸드 플래시 메모리로부터 읽혀진 다음, RAM 상의 inode 리스트에 추가되며 inode가 RAM에서 해제될 때 다시 낸드 플래시 메모리에 저장된다. 파일 시스템 inode가 저장된 낸드 플래시 메모리 주소는 B-tree로 관리된다.

RAM 상의 코어 파일 시스템 기법을 사용하는 SDFS는 파일 쓰기 시 수정된 메타 정보를 낸드 플래시 메모리에 저장하지 않고 실제 데이타만 쓰기 때문에 쓰기 연산을 빠르게 수행할 수 있다. 비정상적인 언마운트로 RAM상의 코어 파일 시스템이 낸드 플래시 메모리에 저장되지 못하더라도, 저널링 마운트를 이용해서 파일 시스템이 복구될 수 있다.

그림 6은 RAM 상의 코어 파일 시스템과 낸드 플래시 메모리에 저장되는 코어 파일 시스템의 구조이다. 마운트 속도를 향상시키기 위해 파일 시스템 언마운트 시 RAM 상의 코어 파일 시스템 정보를 그대로 낸드 플래시 메모리에 저장하는 스냅샷 기법을 사용하는데, 이는 다음 파일 시스템 마운트 시 저장된 정보를 빠르게 RAM에 복구할 수 있기 때문이다.

파일 시스템이 미래에 snapshot 연산을 할 동안에 idle한 경우를 알 수 있다면, 그 때 주기적으로 core 파일 시스템의 내용을 낸드 플래시 메모리에 저장하는 것이 효율적일 수 있다. 왜냐하면, 스냅샷이 일어나고 파일 시스템이 변경되지 않았을 때 파일 시스템이 crush 되면, 저널링 마운트를 통해 마운트 할 필요 없이, 코어 파일 시스템을 통해 마운트 할 수 있기 때문이다. 그러나 스냅샷을 수행하는 동안 파일 시스템이 idle할 지를 정확히 찾아내는 것은 불가능하며, 주기적으로 스냅샷을 하는 것은 실시간성을 크게 훼손시킨다. 그리고 수정이 빈번히 일어나는 경우는 짧은 주기 동안 스냅샷을 하더라도, 바로 파일 시스템이 변경되기 때문에 이익이 없다. 이런 이유로 SDFS에서는 언마운트시에만 스냅샷을 실행한다.

그림 6에서 RAM상의 코어 파일 시스템은 낸드 플래시 메모리에 저장되는 코어 파일 시스템에 대한 추가 정보를 가지고 있다. 이 정보는 MTD 정보, 블록 정보, 세마포어, 루트 디렉토리를 위한 자료구조로서 파일 시스템 연산을 위해 필요한 정보인데, 낸드 플래시 메모리에 저장되지 않더라도 마운트 시 쉽게 복구될 수 있는 정보이다. 따라서 이 정보는 낸드 플래시 메모리에 저장되지 않는다. 언마운트 시 RAM 상에 구성된 파일 시스템 inode는 낸드 플래시 메모리에 모두 저장되어야 하며, 파일 시스템 inode에 유일한 inode 번호를 할당하기 위해 현재 파일 시스템에 저장된 inode 번호의 최대값을 저장한다. 또한, 파일 시스템의 비트맵을 저장하지 않으면 낸드 플래시 메모리 모든 페이지의 잉여 영역을 읽어야만 구성될 수 있기 때문에 비트맵 정보를 저장한다. RAM상의 코어 파일 시스템의 정보들은 모두 YAFFS2에서도 사용하는 정보로써, SDFS의 RAM 사용량은 YAFFS2의 RAM 사용량보다 크지 않다.

파일 시스템의 inode를 저장할 때, 정규 파일인 경우, RAM 상의 파일 시스템 inode를 그대로 낸드 플래시

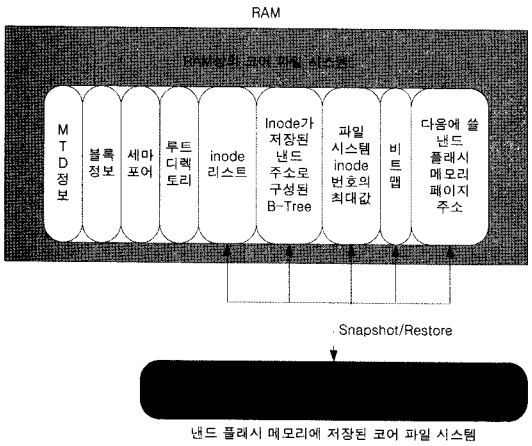


그림 6 RAM 상의 코어 파일 시스템 구조

메모리에 저장한다. 디렉토리 파일인 경우 RAM 상의 파일 시스템 inode는 자식 디렉토리 inode의 포인터를 가지고 있는데, 이것을 inode의 번호로 변경하여 저장한다. 자식 inode 포인터의 경우에는 파일 시스템이 다음 마운트 때 RAM이 재구성될 경우 유효하지 않기 때문에 inode의 번호를 저장해야 한다.

3.4 연산

3.4.1 저널링 마운트(Journaling Mount)

낸드 플래시 메모리의 모든 자기 서술 페이지를 읽어서 마운트를 하는 경우는 비정상적인 언마운트로 언마운트 시점에 RAM 상의 코어 파일 시스템을 낸드 플래시 메모리에 완전하게 저장하지 못했거나, 낸드 플래시 메모리가 실패했을 때이다. 이와 같은 경우, 파일 시스템을 RAM 상에 온전하게 복구하기 위해 낸드 플래시 메모리의 모든 자기 서술 페이지를 읽어야 할 필요가 있다. 저널링 마운트는 낸드 플래시 메모리의 모든 페이지를 읽어서 RAM 상에 코어 파일 시스템(In Memory Core File System)을 생성 한다. SDFS는 블록의 첫 페이지부터 할당이 된다. 따라서 블록의 첫 페이지가 클린 페이지가 아니면 그 블록의 다른 모든 페이지를 읽고, 클린 페이지이면 다음 블록으로 넘어간다.

표 2는 낸드 플래시 메모리 상의 자기 서술 페이지를 읽어서 파일 시스템을 마운트 하는 저널링 마운트 알고리즘이다. cfs는 RAM 상에 생성될 코어 파일 시스템이고, i는 n개의 페이지로 구성된 낸드 플래시 메모리 페이지 인덱스이다.

표 2의 줄 9에서는 이미 자기-서술 정보에 해당하는 inode가 코어 파일 시스템(cfs)에 존재하면, mtime이 큰 자기-서술 정보로 inode를 갱신한다. 그 이유는 최근에 변경된 페이지의 자기-서술 정보가 mtime이 크고, 이것이 그 파일의 최근 정보이기 때문이다.

표 2 저널링 마운트 알고리즘

```
MOUNT_FROM_SELF-DESCRIPTION-PAGE(n)
1   cfs ← null  /* 코어 파일 시스템(cfs) 초기화 */
2   i ← 0
3   while i < n
4     do
5       page_i ← read(i)  /* 플래시 메모리 i번째
        페이지 읽기 */
6       if (page_i = valid)  /* 플래시 메모리 i번째
        페이지가 유효 */
7       then
8         if (the inode of pagei exists in cfs)
9         then update information on pagei its inode
10        else add new inode to cfs
11      i ← i+1
12    end
13  return cfs
```

표 3 코어 파일 시스템 복구 알고리즘

```
RESTORE_CORE_FILE_SYSTEM
1  inode ← cfs->inode /* cfs의 inode 리스트에서 첫 번째
   원소를 inode에 저장 */
2  while (inode)
3    do
4      if (inode is incomplete)
5      Then
6        if (fail to get file name)
7        Then
8          set file name as the inode ID
9        end if
10       if (fail to get some file data)
11       Then
12         change some file data into hole
13       end if
14     else if (the parent directory of inode does not exist)
15     then
16       set parent directory as the root directory
17     end if
18     inode ← cfs->inode->next
19   end
20 return cfs  /* 완전히 복구된 cfs 리턴 */
```

저널링 마운트를 통해 구성된 코어 파일 시스템은 마운트 시 낸드 플래시 메모리의 페이지들의 결함으로 데이타를 읽을 수 없었다면, 코어 파일 시스템의 정보는 완전하지 못하다. 따라서 코어 파일 시스템이 완전한 지를 판단하고, 완전하지 못한 코어 파일 시스템의 정보를 수정하여 완전한 코어 파일 시스템으로 복구해야 한다. 표 3은 코어 파일 시스템이 완전한지를 판단하고, 완전하지 못한 코어 파일 시스템을 수정하여 완전한 코어 파일 시스템으로 복구하는 알고리즘이다.

표 3의 줄 1에서는 코어 파일 시스템의 inode 리스트 첫 번째 원소를 가져온다. 줄 2에서 inode가 존재하는지

를 판단되면 2가지에 대해 수행한다. 첫 번째는 해당 inode가 완전한지 줄 4~13에서 판단하여 복구한다. 줄 4에서는 inode자체가 완전하지 않으면, 즉 메타 정보들 사이에 일관성이 없으면 줄 6~13에서 일관된 최소의 정보만으로 inode를 복구한다. 줄 6~9에서는 파일 이름이 존재하지 않으면 그 파일의 inode ID로 파일의 이름을 설정하고, 줄 10~13에서는 정규 파일의 일부 데이타가 존재하지 않으면 그 부분을 물리적인 공간이 없는 홀(hole)로 처리하여 복구한다. 두 번째는 부모 디렉토리에 대해 완전한지 줄 14~17에서 판단하여 복구한다. 줄 14에서는 inode의 부모 디렉토리가 존재하지 않는다면, 줄 16에서 root 디렉토리를 부모 디렉토리로 대체한다. 복구 수행 과정이 끝나면 줄 18에서는 inode 리스트에서 다음 inode 리스트를 가져오고 다시 줄 2로 돌아가서 가져온 inode에 대한 완전성을 테스트한다. 리스트를 모두 순회하면 줄 20에서는 완전히 복구된 cfs를 리턴한다.

3.4.2 코어 파일 시스템 마운트

낸드 플래시 메모리가 정상적으로 언마운트 되었다면, 마운트에 필요한 정보가 낸드 플래시 메모리에 저장된다. 이 정보가 코어 파일 시스템인데 이를 이용하여 파일 시스템을 빠르게 마운트 할 수 있다. 표 4는 코어 파일 시스템 마운트 알고리즘이다. e(i)는 i번째 inode의 ID와 낸드 플래시 메모리 상의 주소 쌍으로 B-Tree를 구성하는 요소이다.

표 4 코어 파일 시스템 마운트 알고리즘

```
CORE_FILE_SYSTEM_MOUNT
1  cfs ← read a core file system from NAND. /* 낸드
   플래시 메모리에 저장된 코어 파일 시스템 정보를 메인
   메모리에 로드 */
2  if (cfs is valid) /* 코어 파일 시스템 정보가 유효한지를
   판단 */
3  then
4     restore cfs /* 코어 파일 시스템 정보로 파일 시스템의
      자료 구조를 구성 */
5     e ← read a element(inode ID, NAND address) from
      NAND /* 낸드 플래시 메모리에서 엘리먼트를 읽어옴.
      엘리먼트는 B-Tree의 노드 정보 */
6     i ← 1
7     while (e(i)) /* 코어 파일 시스템이 가지고 있는
      B-Tree에 엘리먼트를 삽입 */
8        do
9           cfs->B-Tree ← e(i)
10          i ← i + 1
11 else then /* 코어 파일 시스템 정보가 유효하지
   않다면 저널링 마운트를 수행 */
12    journaling mount
13 end if
14 return cfs
```

줄 1에서는 낸드 플래시 메모리로부터 코어 파일 시스템의 첫 페이지를 읽는다. 코어 파일 시스템은 낸드 플래시 메모리의 정해진 페이지에 저장되기 때문에 다른 정보 없이 코어 파일 시스템의 첫 페이지를 읽을 수 있다. 줄 2에서는 코어 파일 시스템이 유효한 지를 판단하고, 유효하다면 줄 4에서는 낸드 플래시 메모리에 저장된 코어 파일 시스템을 모두 읽어서 RAM 상에 자료구조를 구성한다. 줄 5에서는 낸드 플래시 메모리로부터 inode ID와 낸드 플래시 메모리의 코어 파일 시스템 영역에 inode가 저장된 주소의 쌍으로 구성된 요소를 읽어온다. 줄 6에서는 각 요소를 인덱싱 하기 위해 인덱싱 첨자 i를 1로 초기화한다. 줄 7에서는 i번째 요소가 존재하는지 판단하고, 존재한다면 줄 9에서 i번째 요소를 B-Tree에 추가한다. 줄 10에서는 다음 요소를 인덱싱 하기 위해 i를 1 증가시킨다. 모든 리스트를 순회하면 cfs의 구성이 완료된다. 줄 3에서 코어 파일 시스템이 유효하지 않으면 줄 13에서 저널링 마운트를 수행해서 cfs를 구성한다. 줄 12에서는 RAM상에 구성이 완료된 cfs를 리턴한다.

3.4.3 캐시

SDFS는 리눅스의 서브 시스템으로 그림 7에서와 같이 리눅스 VFS 계층과 MTD 계층 사이에 위치한다.

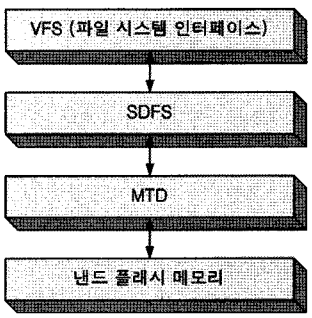
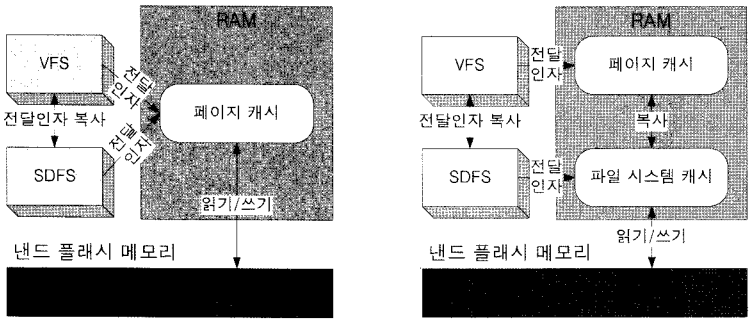


그림 7 SDFS가 적용된 전체 파일 시스템 구조

VFS는 파일 시스템과 리눅스의 인터페이스로 리눅스의 파일 I/O요청을 받아들인다. VFS는 이것을 SDFS에 요청하고, SDFS는 MTD 계층의 I/O 함수를 통해 직접 낸드 플래시 메모리에 파일을 읽고 쓰게 된다. VFS는 리눅스의 페이지 캐시[16]를 사용하여 I/O 효율을 높이는데 파일 시스템 자체에서 캐시를 추가적으로 가질 수 있다. 그림 8(a)는 SDFS가 캐시를 사용하지 않는 구조이고, 그림 8(b)는 캐시를 추가적으로 사용하는 구조이다.

그림 8(b)와 같이 파일 시스템이 자체적인 캐시를 가질 경우, 특별한 상황에서 좋은 성능을 낼 수 있는 캐싱 정책을 사용할 수 있는 장점이 있다. 그러나 이 경우에는 다음의 두 가지 단점이 존재한다. 첫째, VFS는 페이



(a) 파일 시스템 캐시 사용 안 함 (b) 파일 시스템 캐시 사용 함

그림 8 파일 시스템의 캐시 구조

지 캐시의 페이지를 전달인자로 사용하고, 파일 시스템은 자체적인 캐시의 페이지를 전달인자로 사용하기 때문에 I/O 연산 시 파일 시스템 캐시와 페이지 캐시를 서로 복사해야 하는 오버헤드가 존재한다. 둘째, 파일 시스템 캐시를 사용하기 위해 RAM을 할당할 경우, 전체 리눅스 시스템이 사용할 수 있는 RAM 용량이 그만큼 줄어들게 된다. 낸드 플래시 메모리는 모바일 기기에 주로 사용되는데, 모바일 기기에서 RAM 자원은 매우 제약되어 있다. 따라서 페이지 캐시와 파일 시스템 캐시를 모두 사용하는 것은 전체 시스템 성능에 저하를 가져올 수 있다. 따라서 SDFS에서는 전체 리눅스 시스템의 성능을 저하시키지 않기 위해 페이지 캐시만을 사용한다.

3.4.4 쓰기

파일 쓰기 시 일반적으로 두 가지 연산이 필요하다. 하나는 파일 데이타를 낸드 플래시 메모리에 쓰는 것이고, 다른 하나는 그 파일 inode의 갱신이다. SDFS는 RAM 상에 존재하는 코어 파일 시스템의 inode를 변경하고, 변경된 사항은 언마운트 시에 낸드 플래시 메모리에 저장된다. RAM 상의 코어 파일 시스템 정보가 분실되어도 자기 서술 페이지를 통해 파일 시스템은 안전하게 보호된다. 따라서 쓰기 연산 시 파일 데이타만을 낸드 플래시 메모리에 씀으로써 파일 쓰기 연산을 최적화한다.

파일 쓰기 연산 중 파일 수정은 낸드 플래시 메모리 상에는 제자리 덮어쓰기가 불가능하기 때문에 표 5와 같이 네 단계로 이루어진다.

표 5 파일 수정 단계

1. 수정된 페이지를 페이지에 씀.
2. 기존의 페이지를 무효화 시킴.
3. RAM 상에서 코어 파일 시스템의 해당 inode의 주소를 수정된 페이지를 가리키도록 변경하고, 파일 수정 시간, 파일 크기 등을 변경함.
4. 무효화된 페이지에 해당하는 무효화 비트맵을 1로 설정함.

파일 수정 시 파일 시스템이 실패할 경우, 원본 페이지 또는 수정된 페이지 중에 하나만 유효할 때는 파일 시스템이 실패할지라도 저널링 마운트로 쉽게 복구될 수 있다. 그러나 원본과 수정된 페이지가 둘 다 유효한 경우, 두 개의 유효한 자기 서술 페이지가 생기게 되는데, 이 순간 파일 시스템이 실패 한다면 같은 파일의 같은 페이지를 나타내는 유효한 두 개의 자기 서술 페이지를 가지게 된다. 이 경우는 어떠한 페이지를 통해 파일 시스템을 갱신할지를 결정해야 하는데, 수정 시간이 더 최근인 페이지를 찾아서 갱신하면 된다. 수정 시간이 같을 경우에는 버전 넘버가 큰(단 0은 255보다 나중 버전임) 페이지를 이용하여 갱신한다. 결과적으로, 파일 쓰기 연산 시 파일 시스템이 실패하더라도 저널링 마운트를 통해 파일 시스템을 완전히 복구할 수 있다.

3.4.5 읽기

파일의 읽기 시 역시 두 가지 연산이 필요하다. 하나는 파일의 데이타를 낸드 플래시 메모리의 어떤 부분에서 읽어야 하는지를 계산하는 것이고, 다른 하나는 계산된 주소의 낸드 플래시 메모리 페이지를 읽는 것이다. 따라서 읽기 성능을 향상시키기 위해서는 읽어야 할 주소를 빠르게 계산해야 한다.

3.2절에서 언급했듯이 파일은 여러 개의 낸드 플래시 메모리 페이지에 할당되는데 오직 파일의 마지막 부분만이 한 페이지의 저장 영역을 모두 사용하지 않을 수 있다. 이러한 구조는 읽을 데이타가 저장된 주소가 매우 빠르게 계산될 수 있게 한다. (5)와 (6)은 각각 페이지 주소가 저장되어 있는 inode의 주소 배열의 인덱스인 page_id와 전체 page_id의 개수를 구한다. 이 식에서 offset은 읽어야 할 파일 데이타의 시작 오프셋이고, size는 읽어야 할 파일 데이타의 크기(Byte)이며, page_size는 낸드 플래시 메모리의 페이지 크기이다.

$$page\_id = \lceil offset/page\_size \rceil \quad (5)$$

$$page\_id\ 개수 = \lceil size/page\_size \rceil \quad (6)$$

(5)에서 나누기 연산 한 번으로 읽어야 할 페이지 주소가 저장된 page_id를 구할 수 있으며, 실제 낸드 플래시 메모리 주소는 address = i_address[page_id] 형식으로 배열을 한 번 참조해서 얻을 수 있다. 따라서 파일 읽기 연산 시, 파일 크기에 상관없이 나누기 연산 한 번과 참조 연산 한 번으로 읽어야 할 주소를 구할 수 있기 때문에 이것의 시간 복잡도는 O(1)이다.

3.4.6 가비지 컬렉션

낸드 플래시 메모리는 한 번 사용되었던 영역의 재사용을 위해 무효화 상태의 페이지를 클린 상태로 변환하는 가비지 컬렉션(Garbage Collection)이 필요한데, SDFS는 데이타 저장영역을 대상으로 가비지 컬렉션을 수행한다. 가비지 컬렉션에서는 고려해야 할 사항은 다음 세가지 사항이 고려되어야 한다.

첫째, 가비지 컬렉션이 수행 되어야 할 시점이다. 쓰기 연산 중에 파일 시스템에 클린 페이지가 없다면, 가비지 컬렉션을 수행하여 클린 페이지를 확보한 후 다시 쓰기 연산을 해야 하기 때문에 일정한 쓰기 지연 시간과 실시간성을 보장할 수 없게 된다. 따라서 SDFS에서는 파일이 삭제되었을 때는 가비지 컬렉션을 즉시 수행[7]하여 무효화 상태의 페이지를 클린 상태의 페이지로 변환하여 쓰기 연산의 지연 시간을 최소화한다.

낸드 플래시 메모리는 제자리 덮어쓰기가 불가능한 특성을 가지기 때문에 하나의 파일이 계속 갱신되다 보면, 플래시 메모리의 저장 영역을 모두 사용할 경우가 존재한다. 따라서 파일의 업데이트 시에도 가비지 컬렉션을 수행해야 하는데, 하나의 페이지가 업데이트 될 때마다 가비지 컬렉션을 수행하는 것은 비효율적이다. 왜냐하면 하나의 페이지가 업데이트 되었을 때 하나의 무효화 페이지가 생기는데, 하나의 무효화 페이지를 회수하기 위해서 가비지 컬렉션을 수행하면 그 무효화 페이지와 함께 저장된 유효한 페이지를 복사해야 하기 때문이다. 무효화 페이지가 많이 일어났을 때 가비지 컬렉션을 수행하는 것이 유효 페이지의 복사 횟수를 줄이기 때문에 전체 파일 시스템의 성능을 향상시킨다. SDFS에서는 Linux에서 페이지 쓰기 요청이 오면, 이전 연산에서 썼던 페이지의 파일 아이디와 현재 쓰기 요청된 페이지의 파일 아이디를 비교해서 다르면 가비지 컬렉션을 수행한다. 즉, 새로운 파일에 대한 쓰기 요청은 이전 파일에 대한 연속된 갱신이 끝났다는 것을 의미하기 때문에, 이 때 가비지 컬렉션을 수행함으로써 파일 시스템의 전체 성능을 향상시킬 수 있다.

파일의 갱신 연산 중 파일의 내용이 삭제되면 파일의 크기가 줄어들 수 있다. SDFS는 이 경우 파일의 크기가 한 블록의 크기 이상 줄어들면, 가비지 컬렉션을 수행한다. SDFS에서 연속된 파일 데이타는 연속된 페이지에 할당되기 때문에 한 블록 크기 이상으로 파일 크기가 줄어들었을 때 가비지 컬렉션을 수행하면, 유효한 페이지의 복사를 최소화 할 수 있다.

둘째, 가비지 컬렉션을 수행할 블록을 지정해야 한다. 낸드 플래시 메모리의 지움 연산 단위는 페이지 단위가 아니라 블록 단위이다. 따라서 지움 연산을 수행하는 블록에 유효한 페이지가 존재한다면 이 페이지를 다른 블록으로 복사한 후 블록을 지워야 하는데 이것은 성능상의 오버헤드이다. 이 오버헤드를 줄이기 위해, 가급적 유효한 페이지 수가 적은 블록을 선택하여 삭제해야 한다. 그러나 SDFS에서는 일정한 쓰기 지연을 보장하기 위해 무효한 페이지를 한 개 이상 가지는 블록이 가비지 컬렉션 대상으로 설정된다. 이것은 무효화된 페이지가 발생하면 무효화 된 페이지를 클린 페이지로 변경함으로써 쓰기 연산 시 클린 상태로 변경하는 연산을 제거한다.

셋째, 가비지 컬렉션을 언제 멈춰야 할지를 결정해야 한다. SDFS는 파일 삭제로 생긴 모든 무효한 페이지가 회수될 때까지 가비지 컬렉션을 수행한다. 파일 크기가 줄어들었을 때도 파일 삭제 시와 유사한 경우이기 때문에 모든 무효한 페이지가 회수될 때까지 가비지 컬렉션을 수행한다. 파일 업데이트 시 파일 내의 클린 블록 비율이 5% 미만이면, 10개의 클린 블록이 회수할 때까지 가비지 컬렉션을 수행하는데, 이는 쓰기 연산 도중에 가비지 컬렉션을 수행함으로써 예상되는 쓰기 지연 시간을 최소화하기 위해서이다. 클린 블록이 5% 미만일 때 가비지 컬렉션을 수행하는 것은 Windows에서 디스크 공간이 부족함을 나타내는 기준을 따른 것이며, 10개의 클린 블록을 회수할 때까지 수행하는 이유는 평균적인 경우, 0.1초 미만의 쓰기 지연시간을 보장하기 위해서이다.

하나의 블록에 대한 가비지 컬렉션은 아래 표 6과 같이, 파일 수정과 유사한 단계를 거친다.

표 6 가비지 컬렉션 단계

1. 한 블록 내의 모든 유효한 페이지를 다른 블록에 복사한다.
2. 블록을 삭제한다.

표 6의 1을 수행하는 중에 파일 시스템이 실패할 경우, 하나의 파일에 속한 특정 페이지의 자기 서술 페이지가 두 개가 존재하게 된다. 이 경우 원본과 복사본의 자기 서술 페이지가 정확히 일치하기 때문에 저널링 마운트 시 두 개의 자기 서술 페이지 중 먼저 읽은 페이지를 선택하여 RAM 상의 코어 파일 시스템 구성을 완료한다.

3.4.7 언마운트

언마운트 시에는 파일 시스템에 할당된 자원을 해제

하여 시스템으로 반환한다. 그리고 코어 파일 시스템의 현재 상태를 낸드 플래시 메모리의 이전 마운트 시 사용했던 코어 파일 시스템이 저장된 코어 파일 시스템 블록 영역을 지운 후 새롭게 저장한다. 이를 통해 다음 마운트 시 낸드 플래시 메모리에 저장된 코어 파일 시스템을 이용하여 파일 시스템을 빠르게 마운트 할 수 있다.

4. 성능 평가

파일 시스템은 컴퓨터의 반영구 저장소로써, 저장된 정보를 잃어버리지 않아야 하는 안정성이 가장 큰 성능 평가 요소이다. 그리고 파일 입출력은 폰노이만 구조(von Neumann Architecture)에서 전체 시스템의 성능 병목이기 때문에 파일 시스템의 연산은 효율적이어야 한다. 본 절에서는 SDFS의 안정성과 연산의 효율성에 대해 평가한다. 그리고 파일 시스템 성능 측정 프로그램인 IOZone[17]과 cauben[18]을 이용하여 현재 상용으로써 가장 널리 쓰이고는 있으나 FRAM과 FTL을 사용하지 않는 YAFFS2의 성능을 SDFS와 비교한다.

4.1 성능 측정 환경

성능 평가를 위해 구축한 실험 환경은 크게 Host 컴퓨터와 Target 보드(표 7 참조)로 구성되며, 그림 9와 같다.

Host 컴퓨터는 Target 보드의 루트 파일 시스템을 갖고 있는 시스템으로 NFS(Network File System)를 이용하여 Target 보드를 부팅한다. Target 보드는 ARM(ARM9)과 DSP(C55x)로 구성된 저전력, 고성능의 OMAP 프로세서를 탑재한 OMAP 5912 OSK(OMPA Start Kit)[19]가 사용되었다. Host 컴퓨터와의 연결은 이더넷과 시리얼 포트로 연결된다. Target 보드의 부트로더는 환경 설정이 용이하면서 MCU의 레지스터 및 메모리 내용보기, 네트워크 부팅, 플래시 제어 등과 같은 다양한 기능을 지원하는 u-boot 1.1.1[20]을 선택하였다. 그리고 Target 보드의 커널 이미지로서는 Linux Kernel 2.6.10[21]이 Host 컴퓨터에서 크로스 컴파일 되어 TFTP를 이용하여 전송된다. Target 보드 콘솔로서

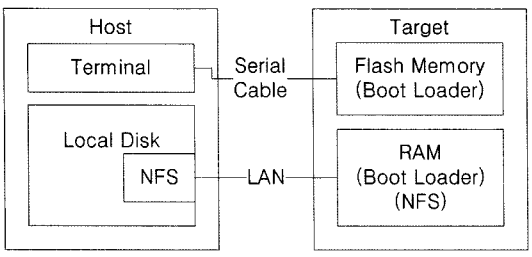


그림 9 Host와 Traget의 연결

표 7 Target 보드 사양

사양	Model	OMAP5912 OSK
	MCU	ARM926Ej-Sid(wt) rev 3 (v51)
	RAM	32MByte DDR SDRAM
	Flash	32MByte(NOR)
	기타	NAND Flash Test Board, RS-232 serial port, 10Mbps Ethernet port, USB port
Boot Loader		u-boot 1.1.1
Linux Kernel		Linux Kernel 2.6.10 for OMAP
Root File System		NFS

는 시리얼 케이블을 이용하여 연결된 Host 컴퓨터의 터미널이 이용된다. 실험에는 용량이 256MByte이고 페이지 크기가 (2048 + 64)Byte인 삼성 K9F2G08U0M[22]가 사용되었다. NAND 플래시 메모리는 64MByte 크기의 파티션으로 나누어 측정한다.

실험에는 용량이 256MByte이고 페이지 크기가 (2048 + 64)Byte인 삼성 K9F2G08U0M[22]가 사용되었다. NAND 플래시 메모리는 64MByte 크기의 파티션으로 나누어 측정한다.

4.2 안정성 평가

SDFS에서의 읽기 및 쓰기 연산 단위가 페이지이기 때문에 데이타는 페이지 단위로 잃어버리게 된다. 페이지 읽기/쓰기 실패의 원인은 하드웨어, 운영체제, 시스템 등의 잘못된 동작 때문에 발생할 수 있다. 예를 들어 데이타를 쓸 때, 하드웨어 또는 시스템의 오류로 주소가 잘못 계산되어 이미 데이타가 저장되어 있는 페이지에 쓰게 되면 그 페이지의 데이타가 소실되게 된다. 또한 낸드 플래시 메모리 블록이 사용 도중 배드 블록이 될 경우 그 블록의 페이지들은 실패의 원인이 된다. 따라서 낸드 플래시 메모리를 저장소로 사용하는 파일 시스템은 페이지들의 읽기/쓰기가 실패했을 때 마운트가 실패하지 않고 정상적인 페이지들에 저장된 정보들만을 이용해 마운트를 할 수 있으면 안정하다고 할 수 있다.

표 8은 SDFS의 안정성 테스트를 위한 시나리오이다. 안정성은 그 성격상 공인된 검증 프로그램이 없기 때문에 다양한 동작을 반복적으로 수행함으로써 검증한다. 특히 SDFS에서는 일부 페이지가 실패하는 상황에서의 안정성 보장을 검증해야 하기 때문에, 기존 안정성 테스트와는 다른 검증 시나리오가 필요하다. 데이타가 저장된 임의의 페이지들을 지워서 정보를 소실시킨 다음, 마운트의 성공 유무와 마운트 후 파일 시스템 연산이 정확히 동작하는지 테스트한다.

테스트 결과 디렉토리가 저장된 페이지가 소실된 경우, 그 디렉토리에 생성되었던 파일의 부모 디렉토리는 root 디렉토리로 설정되었다. 정규파일의 데이타가 저장

표 8 안정성 테스트 시나리오

시나리오 순서	내용
1	파일 시스템에 A~E까지의 5개 디렉토리를 생성한다.
2	각 디렉토리에 파일 크기를 4KByte부터 2배씩 증가시키면서 파일(최대 크기 : 1MByte)을 생성한다.
3	OMAP5912 OSK 보드를 재시작한다.
4	OMAP5912 OSK 보드의 NAND Erase 명령을 통해 낸드 플래시 메모리의 0번 블록부터 63블록까지를 지운다.
5	파일 시스템을 마운트하여 저널링 마운트, 읽기, 쓰기, 삭제 연산이 정확히 동작하는지 확인한다.
6	파일 시스템을 언마운트하고, 다시 마운트 하여 코어 파일 시스템 마운트가 정확히 수행되는지 확인한다.
7	1~5번의 과정을 7회(64MByte를 모두 지움) 더 반복하되, 매 테스트마다 시나리오 순서 4번에서 삭제된 블록부터 다음의 64개 블록을 지운다.

된 페이지가 소실된 경우, 그 정규파일은 소실된 데이타를 홀(holes)로 처리하였고, 파일의 이름이 저장된 페이지가 소실된 경우에는 inode id가 해당 파일의 이름으로 설정되었다. 차후 inode id로 복구된 파일은 사용자가 확인 후 변경할 수 있다. 그리고 파일 시스템의 읽기, 쓰기, 삭제 연산은 정확히 작동하였으며, 다음 언마운트 후 마운트 할 때는 코어 파일 시스템 마운트를 통해 빠르게 마운트 되었다. YAFFS2의 경우는 시나리오 순서 5의 마운트가 실패하여, 안정성 테스트 시나리오를 통과하지 못했다.

4.3 효율성 평가

컴퓨터에서 파일 입출력은 레지스터 접근이나 메모리 접근에 비해 그 속도가 매우 느리다. 따라서 컴퓨터 시스템 전체의 성능을 향상시키기 위해서는 파일 시스템에 저장된 데이타의 입출력이 최소화되어야 한다. SDFS는 파일 입출력 시, 파일의 메타 데이타를 항상 RAM에 유지하고 데이타만을 낸드 플래시 메모리에 쓰고, 읽기 때문에 파일 입출력 연산 횟수의 최소화를 통해 파일 시스템의 성능을 향상시킬 수 있다.

4.3.1 쓰기

SDFS는 새로운 파일을 쓰는 경우와 파일을 수정하는 경우 비용이 발생하는데, 후자의 비용이 더 비싸다. 아래 (7)과 (8)은 각각 파일 쓰기 비용 W와 삭제 비용은 고려하지 않은 파일 수정 비용 U를 정의한다.

$$W = (F(n) + w + a) \times (\lceil s/ps \rceil) \quad (7)$$

$$U = (F(n) + w + wi + a) \times (\lceil s/ps \rceil) \quad (8)$$

s: 쓸 데이타의 크기
ps: 낸드 플래시 메모리 페이지 크기
n: 낸드 플래시 메모리 전체 페이지 수
F(n): 낸드 플래시 메모리에서 데이타를 쓸 페이지를 찾는 비용
w: wi를 포함한 낸드 플래시 메모리 페이지 쓰기 비용(200μs(Typ.))
wi: 낸드 플래시 메모리 페이지 중 잉여영역 쓰기 비용
a: 파일 시스템 inode의 주소 갱신 비용

위 식에서 F(n)의 시간 복잡도는 O(k)(k는 사용중인 페이지 개수)이고 따라서 $n \geq k$이다. a의 시간 복잡도는 램에 접근하는 한번의 연산이기 때문에 시간 복잡도는 O(1)이며 특히 낸드 플래시 메모리 쓰기 비용에 비해 매우 적다. 따라서 (7)과 (8)에서 F(n)과 a는 w와 wi에 비해 매우 작은 값이기 때문에 제거하면 $W = w \times (\lceil s/ps \rceil)$이며, $U = (w + wi) \times (\lceil s/ps \rceil)$로 파일 쓰기 비용보다 파일 수정 비용이 크다. 이상적인 최적 쓰기 비용은 부수적인 연산 없이 낸드 플래시 메모리에 데이타를 쓰는 비용이고, 이 비용 $OW = w \times (\lceil s/ps \rceil)$이기 때문에 SDFS의 파일 쓰기 비용은 최적 쓰기 비용과 같다. 파일 수정 시에는 최적 쓰기 비용에 비해 $U - OW = wi \times (\lceil s/ps \rceil)$만큼의 오버헤드를 갖는다.

그림 10은 SDFS와 YAFFS2의 쓰기 단위에 따라 쓰기 및 다시 쓰기 속도를 비교한 그래프이다. SDFS의 쓰기와 다시 쓰기 성능을 비교해 보면 파일 쓰기의 속도가 파일을 다시 쓰는 속도보다 약간 빠른데 이는 파일을 갱신할 때 이전 페이지를 무효화해야 하는 오버헤드 때문이다. 이는 (7)과 (8)에서 파일의 쓰기 비용 W가 파일의 업데이트 비용 U보다 적은 결과와 같다. YAFFS2는 다시 쓰기 속도가 쓰기 속도보다 빠른데 이는 자체 캐시를 사용하기 때문이다. 쓰기 단위에 따라 SDFS는 균일한 쓰기 속도를 보이는데 이는 자체 캐시를 사용하지 않고, 쓰기 요청이 있을 때마다 낸드 플래시 메모리에 데이타를 쓰기 때문이다. 파일 쓰기에서 SDFS는 YAFFS2보다 평균 36%의 향상된 성능을 보였으며, 다시 쓰기의 경우 9%의 성능 향상을 보였다.

10MByte 파일을 쓸 때 (7)의 W는 1초인데 테스트에

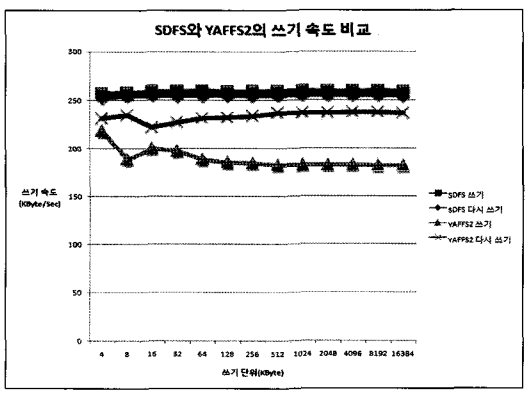


그림 10 SDFS와 YAFFS2의 쓰기 및 다시 쓰기 속도 비교

서는 약 38초가 걸리는 이유는 테스트 환경의 CPU처리 속도와 데이타를 낸드 플래시 메모리로 전송하는 버스의 용량이 부족하기 때문이다.

4.3.2 읽기

SDFS의 파일 읽기 비용은 R은 (9)와 같이 정의된다.

$$R = (F(a) + r) \times \lceil s/ps \rceil \quad (9)$$

s: 읽을 데이타의 크기(Byte)
ps : 낸드 플래시 메모리 페이지 크기
F(a): 파일에서 해당 데이타가 저장된 낸드 플래시 메모리 주소를 찾는 비용
r: 낸드 플래시 메모리 페이지 읽기 비용(25μs(Max.))

(9)에서 F(a)의 시간 복잡도는 읽을 파일의 크기에 관계없이 O(1)이며, 램에 접근하는 연산이기 때문에 무시될 수 있다. 최적 파일 읽기 비용은 부가 연산 없이 파일 크기만큼을 낸드 플래시 메모리에서 읽는 연산 비용이고, 이 비용 $R = r \times \lceil s/ps \rceil$ 이기 때문에 SDFS의 파일 읽기 연산 비용은 최적의 파일 읽기 연산 비용과 같다.

그림 11은 SDFS와 YAFFS2의 캐시 효과가 없을 때의 읽기 단위에 따라 읽기 속도를 비교한 그래프이다. 캐시 효과가 없기 때문에 두 파일 시스템에서 읽기 단위와 상관없이 균일한 읽기 속도를 보여준다. 일반적으로 SDFS가 YAFFS2에 비해 15% 빠른 읽기 속도를 보인다.

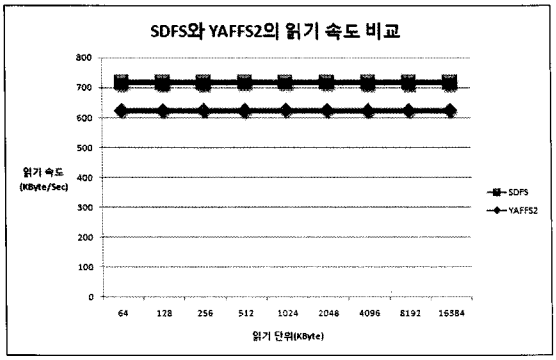


그림 11 SDFS와 YAFFS2의 읽기 속도 비교

4.3.3 일정한 쓰기 지연 시간을 보장하는 가비지 컬렉션

낸드 플래시 메모리를 반영구 저장소로 사용하는 기기에서는 대부분 실시간 보장이 매우 중요한 요소이기 때문에 SDFS에서는 가비지 컬렉션 비용이 많이 들더라도 일정한 쓰기 지연시간을 보장하는 방법을 사용한다. SDFS에서 최악 및 최적의 가비지 컬렉션 비용 GC_W와 GC_B는 각각 아래 (10), (11)과 같다.

$$GC_W = 63 \times (R + W + E) \times ip \quad (10)$$

$$GC_B = E \times (ip / 64) \quad (11)$$

ip: 무효화 페이지의 개수
E: 블록 삭제 비용

SDFS에서 GC_W의 경우 63개의 유효한 페이지를 복사해야 하는 오버헤드가 발생하고, GC_B의 경우 최적 가비지 컬렉션 비용과 같다.

SDFS와 YAFFS2에서 낸드 플래시 메모리 크기 보다 많은 데이타를 쓰고 지우기를 반복하여 낸드 플래시 메모리의 모든 블록을 사용해서 클린 상태인 블록을 제거한다. 그리고 나서 임의의 시점에 8개 블록을 사용하는 파일 크기인 1MByte를 쓰기 위해 4KByte의 데이타를 256번 쓰면서 지연시간을 측정하였다. 8개 블록 크기의 데이타를 쓰는 이유는 YAFFS2에서 5개 이하의 블록이 남았을 때 공격적 가비지 컬렉션이 수행되는데 그 블록 개수 이상을 써서 공격적 가비지 컬렉션을 최대 두 번 이상 수행하기 위해서이다. 표 9는 이 실험을 10회 반복하여 256개의 쓰기 지연 시간의 평균을 구하고, 그 쓰기 지연의 시간 평균, 표준편차, 최대값 그리고 최소값을 나타낸 표이다.

표 9 SDFS와 YAFFS2의 쓰기 지연 시간 비교

	SDFS	YAFFS2
최소값	15465μs	16040μs
최대값	17221μs	205010μs
평균값	15551μs	18510μs
표준 편차	158μs	20339μs

SDFS와 YAFFS2는 각각 최대값과 최소값의 차이가 1756μs와 188970μs 이고, 표준 편차는 158μs와 20339μs로 SDFS가 YAFFS2에 비해 일정한 쓰기 지연 시간을 보인다. SDFS의 쓰기 지연 시간의 평균값은 15551μs로 YAFFS2의 쓰기 지연 시간 평균값 18510μs에 비해 짧다. 결과적으로, SDFS는 YAFFS2보다 짧은 쓰기 지연 시간과 일정한 쓰기 지연 시간을 보임으로써 실시간을 필요로 하는 임베디드 기기의 파일 시스템으로 적합하다.

4.3.4 코어 파일 시스템 마운트

SDFS의 코어 파일 시스템은 낸드 플래시 메모리의 특정 영역에 저장된 마운트 관련 정보만을 읽어 마운트 한다. 이 방법은 낸드 플래시 메모리의 모든 페이지, 또는 일정한 페이지를 모두 읽어서 마운트 하는 방법에 비해, 마운트 시간을 획기적으로 단축시킬 수 있다.

파일 시스템의 메타 정보가 저장된 페이지의 개수(SP)는 슈퍼 블록 변수가 저장된 페이지 하나와, 유효 페이지에 대한 비트맵과 무효 페이지에 대한 비트맵이 저장된 페이지의 합이다. 유효 페이지 비트맵과 무효 페이지 비트맵의 크기는 같으며 그 크기는 전체 낸드 플래시 메모리의 페이지 수에 비례한다. 파일의 이름을 뺀

메타 정보가 저장된 페이지의 개수(FP)는 파일마다 300Byte의 저장공간을 사용하며 파일이 할당된 페이지 수마다 4Byte의 저장공간을 필요로 한다. 모든 파일은 그 파일의 이름을 저장하기 위해 파일의 크기와 상관없이 1페이지를 사용하며 이것은 코어 파일 시스템 영역이 아닌 데이타 영역에 저장되기 때문에 (13)에서는 고려하지 않는다. (14)는 코어 파일 시스템 마운트 시 파일 시스템에 저장된 파일의 개수와 크기에 따라 읽어야 할 페이지(MP)를 계산한 것이다.

$$SP = 1 + 2 \times \lceil (N/8)/ps \rceil \quad (12)$$

$$FP = \lceil \sum_{i=1}^{n} 304Byte + 4Byte \times (\lceil size(F_i)/ps \rceil))/ps \rceil \quad (13)$$

$$MP = FP + SP \quad (14)$$

N: 낸드 플래시 메모리의 페이지 개수
ps : 낸드 플래시 메모리 페이지 크기
F: 파일의 집합, 즉 $\{F_1, F_2, \cdots, F_m\}$ m은 전체 파일의 개수
$size(F_i)$: F_i 파일의 크기(Byte)

마운트 속도는 cauben[18]을 이용해 파일 시스템에 저장된 파일의 크기와 그 파일의 개수를 달리 하면서 측정한다. 그림 12는 SDFS와 YAFFS2의 마운트 속도를 비교한 결과이다. SDFS가 2배에서 최대 20배까지 YAFFS2보다 마운트 속도가 빠른데, 이는 SDFS가 마운트에 필요한 정보를 낸드 플래시 메모리의 일정 영역에 저장함으로써 마운트 시 그 정보가 저장된 페이지만 읽으면 되기 때문이다. 언마운트 시 파일 시스템 정보를 낸드 플래시 메모리에 저장해야 하기 때문에 언마운트 시간이 길지만 이는 1초 이내이다.

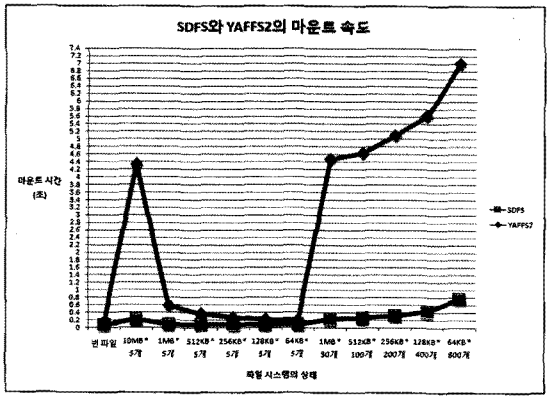


그림 12 SDFS와 YAFFS2의 마운트 속도 비교

5. 결론 및 향후 연구 과제

낸드 플래시 메모리가 모바일 및 임베디드 기기의 반영구 저장소로서 널리 사용됨에 따라 그 동안 낸드 플래시 메모리 전용 파일 시스템에 대한 많은 연구가 진행되어 왔고, 대상 기기의 성격에 적합한 파일 시스템이 개발되어 왔다. 그러나 이러한 파일 시스템은 낸드 플래시 메모리의 페이지가 실패하면, 최악의 경우 파일 시스템 전체가 실패하게 되는 안정상의 문제를 가지고 있다. 저널링 기법을 통해 파일 시스템의 안정성을 보장하는 방법도 저널링 정보를 저장한 페이지가 실패할 경우 파일 시스템을 복구할 수 없는 한계를 가지고 있으며, 저널링 정보 저장으로 파일 시스템 연산의 효율성이 떨어지는 문제점을 가지고 있다. 그리고 제자리 덮어쓰기가 불가능한 낸드 플래시 메모리의 특성 때문에 파일의 쓰기 연산 시 추가적으로 파일과 파일 시스템의 메타 정보도 복구해야 하는 성능상의 오버헤드도 존재한다.

본 논문에서는 페이지가 실패하더라도 남은 정상적인 페이지만을 가지고 파일 시스템을 복구할 수 있는 자기 서술 페이지 기법(Self-Description Page)을 제시하였다. 자기 서술 페이지는 낸드 플래시 메모리의 페이지가 스스로 페이지 저장영역에 저장된 데이타를 설명할 수 있는 페이지로 메타정보를 다른 페이지에 추가로 저장할 필요가 없다. 즉, 하나하나의 페이지가 완전히 설명되는 데이타이기 때문에 어느 페이지가 실패하더라도 정상적인 페이지들만으로 파일 시스템 복구가 가능하다.

이런 자기 서술 페이지를 이용한 마운트는 낸드 플래시 메모리의 모든 페이지를 읽어서 마운트 해야 하기 때문에 시간이 오래 걸린다. 따라서 메모리 상의 코어 파일 시스템 기법(In Memory Core File System)을 사용하여 파일 시스템과 파일에 대한 메타 정보를 낸드 플래시 메모리 일정 영역에 저장함으로써 마운트 시간을 단축시킬 수 있다. 마운트 시 RAM 상에 구성된 코어 파일 시스템은 언마운트 시에 낸드 플래시 메모리에 저장되기까지 RAM 용량이 허용하는 한 RAM 상에만 존재한다. 이는 파일 쓰기 연산 시 순수 데이타만을 낸드 플래시 메모리에 씀으로써 메타 정보를 추가적으로 써야 하는 기존 파일 시스템에 존재하는 오버헤드를 제거할 수 있다.

자기 서술 페이지 기법과 메모리 상의 코어 파일 시스템 기법을 적용시킨 SDFS(Self-Description File System)는 현재 상용으로 널리 사용되고 있는 YAFFS2보다 쓰기 성능과 읽기 성능을 각각 평균 36%, 15% 향상시켰다. 그리고 낸드 플래시 메모리가 사용되는 기기의 실시간 지원 요구를 일정한 쓰기 지연 시간을 통해 만족시켰으며 마운트 시간을 YAFFS2에 비해 2배에서 최고 20배까지 단축시킴으로써 기기의 빠른 부팅을 가능하게 하였다.

본 논문에서 구현한 SDFS 파일 시스템은 리눅스 캐시를 그대로 사용함으로써 전체 시스템과의 일관성을 유지하였다. 그러나 RAM 자원이 충분할 경우나 파일 시스템의 연산 속도가 중요한 시스템에서는 시스템 자

원을 할당 받아 파일 시스템 자체 캐시[6]를 사용해야 할 필요성이 있기 때문에 파일 시스템 자체 캐시를 사용하는 기능이 필요하다. 또한, SDFS에서는 일정한 쓰기 지연 시간을 보장하지만 가비지 컬렉션 비용을 고려하지 않았다. 가비지 컬렉션 시 페이지 복사 비용은 매우 크기 때문에 이를 최소화하기 위한 블록 할당 기법의 적용이 필요하다. 또한, 낸드 플래시 메모리는 블록의 지움 횟수에 제약이 있기 때문에 특정 블록에 계속 데이타를 쓰고 지우기를 반복하면 배드(Bad)블록이 되어 그 블록은 사용할 수 없게 된다. 따라서 페이지를 할당할 때 낸드 플래시 메모리의 지움 횟수를 일정하게 유지할 수 있도록 페이지가 할당되어야 하는데 이것도 SDFS에서 추가로 고려되어야 할 사항이다.

참 고 문 헌

[1] Intel Corporation, "Understanding the flash translation layer (FTL) specification," Application Note 648, 1998.

[2] M. Wu and W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Storage System," Proc. Of the 6th International Conference if Architectural Support for Programming Languages and Operating System, pp. 86-97, 1994.

[3] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory Based File System," Proceedings of the USENIX Technical Conference, 1995.

[4] D. Woodhouse, "JFFS: The Journaling Flash File System," Proc. Ottawa Linux Symp., 2001.

[5] Aleph One Company, "Yet Another Flash Filing System," http://www.aleph1.co.uk/yaffs/.

[6] 박상오, 김경산, 김성조, "NAND 플래시 메모리용 파일 시스템 계층에서 프로그램의 페이지 참조 패턴을 고려한 캐싱 및 선반입 정책," 한국정보처리학회 논문지(A), Vol.14-A No.4, 2007.08.

[7] 한준영, 김성조, "낸드 플래시 메모리 기반 멀티미디어 파일 시스템에서의 효율적인 페이지 할당 및 회수 기법", 2007 한국컴퓨터종합학술대회 논문집(B), Vol.34, No.1, pp. 680-682, JUNE 2007.

[8] 이현철, 김성조, "모바일 멀티미디어 기기를 위한 낸드 플래시 메모리 파일 시스템의 인덱싱 구조", 한국정보과학회 2007 추계학술대회, Vol.34, No.2(B), pp. 441~444, 2007.10.

[9] Hyojun Kim and Youjip Won, "MMFS: Mobile Multimedia File System for NAND Flash based Storage Device," in the IEEE CCNC 2006 proceedings, pp. 208-212, 2006.

[10] seung-Ho Lim and Kyu-Ho Park, "An Efficient NAND Flash File System for Flash Memory Storage," in IEEE TRANSACTIONS ON COMPUTERS, Vol.55, No.7, JULY 2006.

[11] 백승재, 최종무, "플래시 메모리 파일 시스템을 위한 순수도 기반 페이지 할당 기법에 대한 연구", 정보처리학회논문지 A 제13-A권 제5호, 2006.10.

[12] 박상오, 김성조, "NAND 플래시 메모리 기반 파일 시스템을 위한 빠른 마운트 및 안정성 기법", 정보과학회논문지 : 시스템 및 이론, Vol.34, No.12, pp. 683-695, DECEMBER 2007.

[13] 이태훈, 박종화, 김태훈, 이상기, 이주경, 정기동, "임베디드 시스템을 위한 신뢰성 있는 NAND 플래시 파일 시스템의 설계", 한국정보처리학회 논문지 A, Vol. 12-A, No.07, pp. 0571-0582, 2005.12.

[14] Ramtron, FRAM datasheets, http://www.ramtron. com/.

[15] 전병길, 김은기, 신형종, 한석희, "바이트 접근성을 가지는 비휘발성 메모리 소자를 이용한 낸드 플래시 파일 시스템의 부팅시간 개선 기법", 한국정보과학회 논문지 : 컴퓨팅 실제 및 레터, 제14권 제3호, 2008.05.

[16] Daniel P. Bovet and Marco Cesati, "Understanding the Linux Kernel," O'Reilly Media, 2006.

[17] IOzone Filesystem Benchmark, http://www.iozone.org/docs/IOzone_mswo- rd_98.pdf.

[18] 박상오, 김성조, "리눅스 기반의 NAND 플래시 메모리 파일 시스템에 대한 성능 측정 도구 설계", 한국정보과학회 2005 추계학술대회, Vol.32, No.02, 2005.11.

[19] Spectrum Digital Inc, "OMAP5912 Starter Kit (OSK)," http://www.spect- rumdigital.com/.

[20] DENX, "UBoot," http://www.denx.de/wiki/UBoot.

[21] Linux, "Linux Kernel v2.6.10," http://www.linux. org/.

[22] Samsung Electronics, "K9XXG08UXA.pdf," 2006.

한 준 영

2006년 중앙대학교 컴퓨터공학과(공학사) 2006년~현재 중앙대학교 컴퓨터공학과(석사과정). 관심분야는 NAND 플래시 메모리 파일 시스템, 임베디드 시스템, Linux Kernel

박 상 오

2005년 중앙대학교 컴퓨터공학과(공학사) 2007년 중앙대학교 컴퓨터공학과(공학석사). 2007년~현재 중앙대학교 컴퓨터공학과(박사과정). 관심분야는 NAND 플래시 메모리 파일 시스템, 임베디드 시스템, Linux Kernel

김 성 조

1975년 서울대학교 응용수학과(공학사) 1977년 한국과학기술원 전산과(이학석사) 1977년~1980년 ADD(연구원). 1980년~현재 중앙대학교 컴퓨터공학부 교수. 1987년 Univ. of Texas at Austin(공학박사). 1887년~1988년 Univ. of Texas at Austin(Research Fellow). 1996년~1997년 Univ. California-Irvine(Visiting Professor). 관심분야는 이동컴퓨팅, 임베디드 소프트웨어, 유비쿼터스컴퓨팅