# NAND 플래시 파일 시스템의 빠른 복구를 위한 로그 구조 설계

김태훈*○ 이주경** 정기동*
부산대학교 컴퓨터공학과*
{rider7979*○ jklee**}@melon.cs.pusan.ac.kr, kdchung* @pusan.ac.kr

# Design of a Log Structure for Fast Recovery of a File System on NAND Flash Memory

TaeHoon Kim*○ , JuKyung Lee**, Kidong Chung*
Dept. of Computer Engineering, Pusan National University*
Pusan National University**

요 약

본 논문에서는 기존의 플래시 파일 시스템과 차별되는 빠른 복구를 위한 로그 구조를 제안한다. 로그와 메타데이터를 플래시 메모리의 특정 영역에 기록하고, 오류가 발생하게 되면 복구단계에서 이 영역만을 스캔하기 때문에 파일 시스템의 오버헤드를 줄인다. 그리고 로깅연산은 트랜잭션으로 관리되어 다른 연산에 간섭받지 않고 독립적으로 수행이 되므로 동시성을 높이고 로깅에 따른 오버헤드가 크지 않다. 따라서 로깅과 복구에 필요한 시간을 최소화하며 신뢰성 있는 파일 시스템 구성이 가능하다

## 1. 서 론

플래시 메모리는 기존의 하드 디스크를 대체할 만큼 활용빈도가 높은 기억장치로써 비휘발성의 특징을 가지는 메모리이다. 이를 다양한 용도로 활용하는 휴대전화와 같은 이동형 정보 기기의 사용이 일반화 되어 있고, 데이터를 저장하기 위한 보조기억장치로 플래시 메모리의 사용이 일반화 되어 있다.

플래시 메모리는 충격에 강하고 전력 소모가 적으며, 비휘발성 즉, 전력 공급이 중단된 상태에서도 데이터 유지가 가능하기 때문에 이동형 정보 기기의 보조 기억 장치로 적합하다.

이러한 플래시 메모리의 특징은 많은 임베디드 내장형 시스템에서 하드디스크와 같은 보조 기억장치의 역할을 수행할 수 있음을 의미한다

이동형 정보 기기는 빠른 시동과 신뢰성이 요구되는데, 배터리를 사용하고 외부환경에 쉽게 노출되기 때문에 전력중단(Power Fail)이나 예상치 못한 장애(Crash)가 생길 수 있는 단점이 있다. 이러한 상황이 발생했을 경우 기존의 플래시 파일시스템에서는 플래시 메모리의 전체 스캔을 통해 로그를 확인하고 복구하므로 많은 시간이 소모된다.

본 논문에서는 빠른 회복기능을 제공하기 위해 기본적으로 로그 구조 파일시스템(Log-Structured file system)[3]에서 제안한 로그 기법을 사용하며, Location Information 영역의 활용을 통한 복구 구조에 대해 제안한다.

본 논문에서는 한 가지 사항에 대한 전제를 둔다. 대표적 플래시 파일 시스템인 YAFFS[1]나 JFFS[2]에서는 파일의 일부분이 업데이트가 발생할 경우 파일 전체를 플래시 메모리에 다시 쓴다. 제안하는 플래시 파일 시스템에서는 파일에서 업데이트가 일어난 부분만 쓰기 연산을 수행하도록 설계되었다. 이 구조에 대해서는 타 논문에서 소개가 되므로 본 논문에서는 제안하는 플래시 파일 시스템인 RFFS( Robust Flash File System) 에서의 로그 구조와 기법에 관해서만 언급한다.

본 논문의 구성은 다음과 같다. 2장 관련연구에서는 플래시 파일 시스템과 빠른 오류 복구를 위한 저널링 파일 시스템(Journaling File System)에 대해 설명한다. 3장에서는 본 논문에서 제안하는 플래시 파일 시스템엔 RFFS(Robust Flash File System)의 설계 구조와 로깅에 대해 설명한다. 4장에서는 실험 및 성능평가에 대해 기술하고 5장에서 결론을 설명한다.

이 논문은 교육인적 자원부 지방연구중심대학 육성사업 (차세대물류IT 기술연구 사업단)의 지원에 의하여 연구되었다."

## 2. 관련연구

### 2.1 플래시 파일 시스템

플래시에 대한 연산은 하드디스크와는 달리 임의 접근이 가능하기 때문에, 읽기 연산은 파일의 분산에 따른 접근 시간의 차이가 없이 빠른 읽기 연산을 수행 한다

그러나 플래시 메모리의 쓰기 연산에 있어서는 몇 가지 제약사항이 있다. 기존에 사용되는 디스크 매체와 달리 데이터가 존재하는 영역에는 덮어쓰기(in-place update)가 불가능하다. 그러므로 기존에 데이터가 있던 영역을 쓰기 위해서는 삭제 연산을 수행하여 플래시 메모리의 모든 비트 값을 1로 초기화하는 작업이 선행되어야 하는데, 이러한 초기화 연산의 횟수는 10만번 ~ 100만번 정도로 제한되어 있다. 따라서 특정 위치에 대한 집중적인 사용은 플래시 메모리의 수명을 단축시키는 결과를 가져올 수 있으므로, 가비지 컬렉션(garbage collection)과 쓰기 평준화(wear leveling)와 같은 작업이 요구된다. 또한, 동작 중에 갑자기 전력 중단이 발생하거나 오류가 발생한 경우에는 시스템의 무결성을 회복하기 위해서 긴 시간 동안 파일

시스템에 대한 검사가 수행되어야 한다. 따라서, 이러한 특징을 가지고 있는 플래시 메모리를 이동형 정보 기기에서 보조 기억 장치로 사용하기 위해서 플래시 메모리를 최적화하고 단점을 보완할 수 있는 효율적인 플래시 파일 시스템이 필요 하며, 현재 이러한 파일 시스템에 대한 연구가 진행되고 있다

현재 널리 공개된 플래시 파일 시스템으로 NOR형 플래시 메모리에는 JFFS가 사용되고 있으며, NAND형 플래시 메모리에는 YAFFS가 사용되고 있다.

### 2.2 저널링 파일 시스템(Journaling File System)

파일 시스템에서는 입출력 성능을 향상시키기 위해 캐싱(caching) 기법이 사용 된다. 그러나 버퍼로부터 디스크에 데이터를 쓰는 작업을 완료하기 전에, 시스템에 장애(crash)나 전력중단(power fail)과 같은 문제가 발생하였을 경우 파일 시스템의 일관성을 보장할 수 없게 된다.

따라서, 파일 시스템의 일관성을 유지하기 위해 파일 시스템 검사기능(FSCK)이 전체 파일 시스템을 검사해서 파일 시스템의 일관성을 확인해야 한다. 이 단점을 보완한 빠른 복구를 위해 저널링 파일 시스템이 고안되었다.

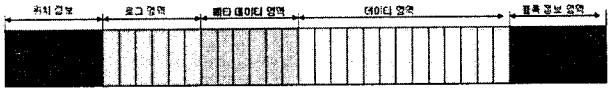
저널링 파일 시스템은 파일 시스템에 연산이 발생하기 이전에 연산에 대한 내용을 로그에 기록한다. 여기에는 연산명(operation name)뿐만 아니라, 연산을 수행하기 전의 인수 내용(argument content)에 대한 정보도 기록된다. 또한 하나의 작업을 위해 수행되는 원자적인 모든 작업을 모아 하나의 트랜잭션으로 관리하며, 수행이 완료되면 commit 연산이 이루어지고, 캐쉬의 내용이 디스크에 쓰이게 된다[4]. 따라서 파일 시스템에 장애가 발생해도 로그 파일을 추적하여 파일 시스템의 일관성을 유지하고 복구 가능성을 높일 수 있으며, 파일 시스템이 마운트(mount) 될 때 파일 시스템 전체를 검사를 하지 않아도 되기 때문에 좀 더 빠른 마운트가 가능하다

현재 저널링 파일 시스템을 지원하는 파일 시스템으로는 SGI의 XFS, IBM의 JFS, Namesys의 ResierFS, Ext2에 저널링을 적용한 Ext3가 있으며, 플래시 메모리의 특성에 맞게 수정하여 적용된 JFFS 등이 있다[5].

## 3. 설계 및 로깅

### 3.1 구조

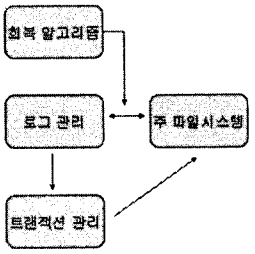
RFFS는 NAND 플래시 메모리 전용 파일 시스템으로 개발되었며, 메모리의 영역을 5개의 영역으로 나누어서 사용한다. 위치정보(Location Information)영역, 로그(Log)영역, 메타데이터영역, 데이터영역 그리고, 블록정보(Block Information)영역으로 구성되며, 그 구성은 [그림1]과 같다.




[그림1] RFFS의 플래시 영역 구성

위치정보 영역은 로그와 메타데이터 영역의 위치정보를 가지고 있고, 정상적인 언마운트 여부를 확인할 수 있는 언마운트 플래그를 가지고 있다. 정상적인 언마운트 수행후에 파일 시스템이 마운트 될 때는 위치정보 영역에서 메타데이터의 포인터를 통해 메타 데이터 영역의 정보를 스캔해서 빠른 마운팅을 수행한다. 만약, 마운트시에 언마운트 플래그를 통해 비정상적인 언마운트가 수행됐음이 확인되면 위치정보 영역에서 로그 포인터를 통해 로그 정보를 확인하여 복구를 수행한다. 로그영역과 메타데이터영역은 로그와 메타 데이터가 기록되는 영역이다. 이 영역들에 속하는 데이터는 플래시 전체에 분산되어 있지 않고, 각각의 영역에 연속적인 블록을 할당받아 저장 된다. 데이터 영역에는 사용자의 일반 데이터가 저장되고, 블록 정보 영역은 플래시 메모리 전체 블록에 대한 정보가 주기적으로 저장된다. 이 영역에는 블록 전체의 삭제횟수나 유효(valid), 무효(invalid)한 데이터 등의 정보를 포함한다.

RFFS에서 전체적인 저널링의 흐름은 [그림2]와 같다. 주 파일시스템의 연산이 일어나기 전에 먼저 연산에 대한 로그가 생성되고, 각각의 로그는 트랜잭션에 의해 관리된다. 만약, 장애가 발생했을 경우 회복 알고리즘을 통해 복구한다.




[그림2] 저널링 파일 시스템 배경도

RFFS에서 로그 데이터 구조는 [표1]과 같다. meta 필드에서 메타데이터에 대한 포인터를 가지고 있고, 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.

[표1] RFFS의 로그 구조 필드

| 크기 | 항목 | 설명 |
|---|---|---|
| 32bit | meta | 메타데이터의 청크 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 |

### 3.2 로깅 기법

RFFS는 오류가 발생했을 경우 빠른 복구를 위해 로깅을 사용한다. 로깅은 파일 시스템에서 임의의 연산이 수행될 때 그 연산에 대한 정보를 저장하는 것을 의미한다.

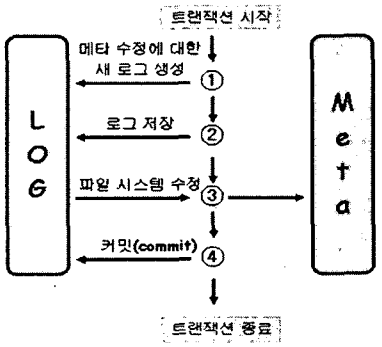
본 논문에서 제안하는 로깅은 다음과 같은 순서로 수행 한다.

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 수행될때 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 쓰여질 경우 포인터를 통해 연속된 영역처럼 사용이 가능하다.

[그림3]에서는 RFFS에서 로깅의 흐름을 표현하고 있다
그림에서와 같이 파일 시스템과 로깅은 트랜잭션에 의해 개별적으로 수행하고, 실제 파일 시스템의 연산이 수행되기 전에 로그 영역에 로깅(logging)이 완료된 후 사용자에 의한 연산을 수행한다. 따라서 로깅연산은 파일 시스템의 다른 연산에 간섭받지 않고 로그 데이터에 대한 무결성을 보장한다.




[그림3]RFFS의 로깅 구조

RFFS에서의 복구 수행 과정은 다음과 같다

1) 위치 정보 영역의 unmount_flag가 0인 경우 비정상적 종료로 판단하고 복구 작업 수행
2) 위치정보 영역의 log 포인터를 통해 로그 영역을 찾고, 로그 영역을 검색(scan)하여 최근에 실패한 연산을 찾는다
3) 로그의 meta필드를 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.

## 4. 실험 및 성능평가

본 논문에서 제안한 로그 구조의 성능 평가를 위해서 메모리상의 힙 영역을 24MB 할당하여 플래시 메모리처럼 사용하였다. JFFS2와 본 논문에 제안하는 로그 구조를 가지는 RFFS에서의 플래시 메모리의 사용량 따른 마운트 성능과 RFFS에서 로깅에 따른 오버헤드를 측정하기 위해 로깅 기능을 사용할 경우와 사용하지 않을 경우에서의 쓰기 속도를 비교하였다.
첫 번째 실험에서 파일 시스템의 사용량을 20%, 50%, 70%로 늘려가며 마운트 시간을 비교했다. 복구는 마운트 과정에서 수행되므로 마운트 시간을 통해 파일 시스템의 복구 성능을 측정할 수 있으며, 실험결과는 [표2]와 같다

[표2] 사용량에 따른 마운트 속도

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

두 번째 실험에서는 512Kb, 1024Kb의 파일을 10회 복사하여 그 평균 시간을 비교며, 그 결과는 [표3]과 같다.

[표3] 로깅 유무에 따른 평균 복사 시간

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

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

## 5. 결론

본 논문에서는 기존의 플래시 파일 시스템과 차별되는 빠른 복구를 위한 구조를 제안한다. 대표적인 플래시 파일 시스템인 JFFS는 에러 복구를 위해 메모리 전체에 분산되어 있는 로그와 데이터들을 반복적으로 스캔하여 복구를 수행하는 반면, 본 논문에서 제안한 구조에서는 로그와 메타데이터들을 일정 영역에 모으고, 이 영역을 찾기 위한 위지정보 영역을 사용함으로써 흩어져 있는 로그를 찾기 위해 플래시 메모리 전체를 스캔할 필요가 없이 복구에 필요한 로그와 메타정보를 빠르게 찾을 수 있다. 따라서 메모리 전체를 스캔하는 JFFS에 비해 빠르게 파일 시스템의 일관성을 회복할 수 있고, 스캔에 의한 오버헤드를 크게 줄임으로써 파일 시스템의 전체적인 성능을 향상 시킬 수 있다. 또한, 트랜잭션 관리를 통해 연산에 대한 무결성이 보장되며, 해당 연산이 실패했을 경우 롤백을 통해 데이터를 보호한다.

## 참고 문헌

[1] David Woodhouse,"JFFS:The Journalling Flash File System" Technical Paper of RedHat inc. Oct. 2001.
[2] "YAFFS Flash File System" http://www.aleph1.co.uk/yaffs/
[3] Margo Seltzer, Keith Bostic "An Implementation of a Log-Structured File System for UNIX" Winter USENIX, January 25-29, 1993
[4] Juan I. Santos Florido, "Journal File Systems", http://www.linuxgazzette.com/issue55/florido.html
[5] Richard Menedetter "Journaling Filesystems for Linux"
[6] Eran Gal, Sivan Toledo, "Algorithms and Data Structures for flash Memories" Tel-Aviv Univ. 2004
[7] Mendel Rosenblum "The Design and Implementation of a Log-Structured File System"