플래시 메모리를 위한 파일 시스템의 구현
(Implementation of a File System for Flash Memory)

박 상 호 † 안 우 현 † 박 대 연 †† 김 정 기 ††† 박 승 민 †††
(Sangho Park) (Woo Hyun Ahn) (Daeyeon Park) (Jeong-Ki Kim) (Sung-Min Park)

요 약 플래시 메모리는 기존의 회전식 자기 매체에 비해서 속도가 빠르고, 충격에 강한 장점이 있다. 이런 특성으로 인해 기존의 가전, 통신 기기, 휴대 기기에서 저장매체로써 플래시 메모리의 사용이 증대하고 있고, 더불어 저장 매체로 플래시 메모리를 사용한 파일 시스템의 필요성도 증가하고 있다. 저장 매체로써 플래시 메모리는 위와 같은 장점 외에 두 가지 문제점을 가지고 있다. 첫째, 데이타를 덧쓸 수가 없다는 점이다. 데이타를 덧쓰기 위해서는 데이타를 저장하기 전에 플래시 메모리를 지워야 하는데, 지우는 작업은 1초 정도의 시간이 소요된다. 따라서, 플래시 메모리에 저장된 데이타를 수정할 때, 시간이 오래 걸리게 되는데, 본 논문에서는 기존의 LFS(Log-structured File System) 방식으로 데이타를 저장하여 이와 같은 문제점을 해결하였다. 플래시 메모리의 두 번째 문제점은 수명이 제한되어 있다는 점이다. 본 논문에서는 cleaning policy를 통하여 수명을 최대한 연장시킬 수 있도록 하였다. 본 논문에서 구현한 플래시 파일 시스템은 소용량 저장매체에 적합한 FAT를 사용하여 성능을 향상시켰고, FAT를 구현할 때 발생할 수 있는 문제점을 해결하였다. 또한, 차례 쓰기, 무작위 쓰기의 실험을 통해서 성능을 분석하였다.

Abstract Advantages of flash memories are their shock resistance and fast read speed, which is much faster than that of a HDD. Because of these characteristics, they are increasingly used in the traditional household electric appliance and portable handset and ,therefore, development of file systems which use them as storage medium is increasingly needed. But they have two problems as storage medium. First, data stored in them cannot be overwritten: it must be erased before new data can be stored. Unfortunately, this erase operation usually takes about one second. Consequently, updating data in flash memories takes long time. In this paper, their problem is solved by using a data update mechanism like LFS(Log-structured File System). Second, their erase operations are restricted. We propose novel cleaning policy in order to increase the life cycle. We implemented FAT file system, which is suitable to small storage medium and solved problems, which usually happen in implementing FAT. We evaluated the performance of sequential writes and random writes on our implemented flash file system.

1. 서 론

Embedded system에서 저장매체로써 디스크 대신에 플래시 메모리(flash memory)가 점차적으로 널리 사용되고 있다. 또한, 기존의 가전, 통신 기기, 휴대 기기, 가정용 셋탑 박스(set-top box)나 인터넷 폰(internet phone)에서 플래시 메모리의 사용이 증대하고 있다. 이런 플래시 메모리 사용의 증대는 플래시 메모리가 기존의 회전식 자기 매체를 이용한 디스크에 비해서 다음과 같은 장점을 가지고 있기 때문이다. 첫째, 플래시 메모리는 외부 충격에 강하고, 비휘발성(non-volatile)이다. 따라서, 불안정한 환경에서도 동작하여야 하는 embedded system에 적합하다. 둘째, 기존의 디스크와 같은 회전식 자기 매체가 아니라 메모리이기 때문에 빠른 속도로 접근(access)할 수 있고, 높은 성능을 제공한다. 셋째, 저전력으로 구동이 가능하고, 크기가 작기 때

† 비 회 원 : 한국과학기술원 전기및전자공학과
shpark@sslab.kaist.ac.kr
whahn@sslab.kaist.ac.kr
†† 정 회 원 : 한국과학기술원 전기및전자공학과 교수
daeyeon@ee.kaist.ac.kr
††† 비 회 원 : 한국전자통신연구원 인터넷정보가전연구부 연구원
jkk@etri.re.kr
minpark@etri.re.kr
논문접수 : 2000년 10월 10일
심사완료 : 2001년 7월 5일

문에, 크기가 제한된 embedded system에 적합하다. 위와 같은 장점 때문에 플래시 메모리는 하드디스크에 비해서 메가 바이트(Mega byte)당 5배에서 10배정도 비싸지만, IC-card와 같은 형태로 휴대용 컴퓨터(portable computer)에서 보조 기억 장치로써 HDD(Hard Disk Driver)를 대체하고 있다.

또한, 롬(ROM:Read Only Memory)과 달리 전기적으로 지우고, 재사용할 수 있다. 많은 플래시 메모리 제품[2][3]이 출시되었고, 플래시 메모리의 일반적인 특징은 표 1과 같다.

저장매체로써 플래시 메모리를 사용할 때, 두 가지 문제점을 고려하여야 한다. 첫째로, 한 번 데이타를 쓴 영역에 바로 다시 쓸 수 없다는 점이다. 새로운 데이타를 쓰기 전에 플래시 메모리를 지워야 하고, 게다가 플래시 메모리를 지우는 작업은 EU(Erase Unit)단위로 이루어진다. 표 1에 나와 있듯이 EU은 보통 64 KBytes에서 256 KBytes 정도이고, 한 EU을 지우는데 1초가 소요된다. 따라서, 단지 몇 바이트를 바꾸기 위해서 한 EU을 지우는 것은 큰 성능 저하를 가져올 것이다. 둘째로, 플래시 메모리를 지우는 횟수가 제한되어 있다는 사실이다. 즉, 플래시 메모리를 영구적으로 사용할 수 없기 때문에 지우는 작업을 최대한 억제하여 수명을 늘려야 한다.

위와 같은 플래시 메모리의 특성 때문에 기존에 개발된 플래시 메모리를 이용한 파일 시스템[8][9][10]에서는 LFS(Log-Structured File System)을 사용한다. 즉, 데이타가 저장된 블록에 수정된 데이타를 기록하는 것이 아니라, 다른 빈 블록에 수정된 데이타를 기록한다. 본 논문에서 구현한 플래시 파일 시스템도 마찬가지로 LFS 방법을 사용하지만, 기존의 방법과는 다음과 같은 점에서 다르다.

표 1 플래시 메모리 특성

Read Cycle	80-150 ns
Write Cycle	10 μs/byte
Erase Cycle	0.5 ~ 1 s/block
Erase Cycle limit	100,000 times
Erase unit size	64 kbytes - 256 kbytes
Power Consumption	30-50mA in an active state 20-100 μA in a standby state

첫째, FAT 파일 시스템[5][6] 기반의 플래시 파일 시스템을 구현하였다. 기존의 방식은 inode를 사용하였는데, 소용량 저장 매체에서 FAT 파일 시스템은 inode를 사용한 UNIX 파일 시스템보다 더 좋은 성능을 보인다. Inode 방식은 각 inode 정보를 읽어야 파일 접근이 가능하지만, linked-list 형태의 FAT 파일 시스템은 FAT만 버퍼 캐쉬에 저장되어 있으면 어떤 파일이든 접근이 가능하다. 소용량 저장 매체의 경우 FAT 크기가 작기 때문에 모든 FAT 정보가 버퍼 캐쉬에 올라올 수 있다.

또한, FAT 파일 시스템을 구현할 때 일반적으로 발생할 수 있는 두 가지 문제점을 해결하였다. 파일 접근 시 FAT 파일 시스템은 언제나 linked-list 형태의 FAT를 검색해야 한다. 파일 내의 offset이 클수록 긴 시간이 소요되는데, 플래시 메모리는 접근 속도가 빠르기 때문에 FAT 정보 검색 문제점이 상대적으로 더 큰 문제가 된다. 본 논문에서 구현한 파일 시스템은 VFS (Virtual File System)에서 vnode내에 추가적인 table을 만들어서 검색 overhead를 줄였다. 둘째로 발생하는 문제점은 free 클러스터를 할당할 때, FAT를 차례대로 검색해야 하는 문제점이다. 이 문제점은 free 클러스터 할당 시 클러스터의 번호를 기억하고 있다가 다음에 free 클러스터를 할당할 때, 기억하고 있는 클러스터의 번호를 이용하여 free 클러스터가 어디 있는지 예측하여 해결한다.

둘째, 본 플래시 파일 시스템은 기존의 파일 시스템과 달리 오류 복구 능력을 가지고 있다. 플래시 메모리가 주로 사용되는 휴대용 장비 같은 embedded system을 비정상적으로 전원이 꺼지거나 예기치 않은 경우가 발생할 수 있다. 따라서, 비정상적으로 전원이 꺼졌을 경우에 발생할 수 있는 오류를 완벽하게 복구할 수 있어야 하는데, 본 논문에서 구현한 플래시 파일 시스템은 데이타 수정과 cleaning 과정을 세분화시켜 오류 복구 능력을 구현하였다.

셋째, 버퍼 캐쉬를 이용할 경우 meta-data 수정 문제점을 soft-update를 이용하여 해결하였다. Soft-update [11][12]는 정해진 순서로 블록을 HDD에 써서 dependency information을 유지하고, meta-data 수정 문제점을 해결한다.

본 논문에서 구현한 파일 시스템은 인터페이스, VFS, FAT, 버퍼 캐쉬, 플래시 메모리 관리자, 플래시 메모리 장치 드라이버로 구성되어 있다. UNIX 시스템와 동일한 인터페이스를 제공하여 응용 프로그램(Application Program)의 호환성을 유지하고, 사용자에게 편의성을 제공한다. VFS를 사용하여 다양한 파일 시스템을 같이 사용할 수 있다. 예를 들어, 작은 용량이나 성능에 큰 영향을 미치는 파일은 플래시 메모리에 저장하고, 대용량 데이타는 HDD에 저장하는 시스템에서 플래시 메모

리와 HDD를 동시에 사용할 수 있는데, 본 논문에서 구현한 파일 시스템은 VFS를 사용하여 이와 같은 확장성이 용이하게 설계되었다.

이후의 순서는 아래와 같다. 2장에서 본 논문에서 구현된 플래시 파일 시스템의 전체적인 구조에 대해서 설명한다. 3장에서 플래시 파일 시스템의 구현상 특징에 대해서 설명하고, 4장에서 실험을 통한 성능 분석에 내용을 실었고, 5장에서 마지막으로 결론을 말하겠다.

2. 플래시 파일 시스템

본 논문을 통해 구현한 플래시 파일 시스템의 구조를 그림 1에 나타내었다. 맨 상위 계층에 파일 시스템 인터페이스(interface)가 있고, 아래에 파일 시스템 관리자, 플래시 메모리 관리자, 플래시 메모리 장치 드라이버가 있다. 2.1 절에서 파일 시스템 인터페이스와 VFS에 대해서 설명하고, 2.2 절에서 FAT, 2.3 절에서 버퍼 캐쉬에 대해서 설명한다. 2.4 절에서 플래시 메모리 관리자에 대해서 설명하고, 2.5 절에서 플래시 메모리의 데이타 구조, 2,6 절, 2.7 절에서 데이타 저장과 cleaning 과정에 대해서 설명한다.

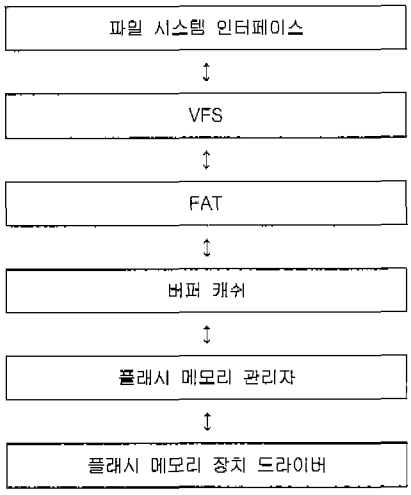

그림 1 플래시 파일 시스템의 전체적인 구조

2.1 파일 시스템 인터페이스와 VFS(Virtual File System)

본 논문에서 구현된 파일 시스템 인터페이스는 UNIX [4]와 동일한 인터페이스를 제공한다. 따라서, 기존의 응용 프로그램(Application Program)을 그대로 구동시킬 수 있는 확장성을 가지고 있다. 또한, UNIX에 익숙한 사용자에게 편의성과 일관성을 제공한다.

선 마이크로시스템(Sun Microsystem)에서 개발된 VFS(Virtual File System)[13][4]을 사용하여 복수의 파일 시스템 유형을 지원하도록 하였다. UNIX 파일 시스템(s5fs, ufs) 뿐만 아니라 non-UNIX 파일 시스템(FAT, A/UX)을 지원할 수 있도록 설계되어있다. VFS는 각각의 파일을 vnode로 관리한다. Vnode는 파일을 추상화시킨 구조체로써, 플래시 메모리 내에 저장된 파일 크기, 변경된 날짜, 파일의 permission 및 기타 정보를 포함하고, 파일 시스템 의존적인 함수들의 링크(link)를 가지고 있다. 파일에 대한 접근은 파일 시스템 관리자의 vnode를 통해서만 가능하다. 파일을 읽거나 쓸 때, 각 파일마다 vnode가 만들어지고, vnode를 통해서 파일의 위치를 알아내고, 읽거나 쓰게 된다. 시스템 초기화시 일정한 개수의 vnode만큼의 메모리를 미리 할당받아 linked list 형태로 관리하고, vnode가 필요하면, 시간 지연 없이 바로 이용할 수 있다. 또한, 각 파일의 vnode은 LRU(Least Recently Used) 알고리즘을 사용하여 관리되기 때문에, 별로 쓰이지 않는 파일은 vnode를 만드는데 시간이 소요되는 대신에, 자주 쓰이는 파일은 빨리 접근할 수 있다.

2.2 FAT(File Allocation Table)

실제 플래시 메모리에 파일을 저장하는 방식을 FAT 파일 시스템[5][6]을 이용하였다. FAT은 DOS에서 사용되는 파일 시스템으로 inode를 사용하는 기존의 UNIX 파일 시스템보다 소용량 저장 매체에서는 더 적합한 것으로 알려져 있다. 보통 제품으로 나와있는 플래시 메모리는 16 메가바이트이기 때문에, inode 방식보다는 FAT 방식이 더 적합하다.

FAT 파일 시스템은 몇 개로 섹터로 구성되는 클러스터 단위로 저장 매체를 관리하는데, 이와 같은 정보가 부트 섹터에 저장된다. 부트 섹터는 저장 매체의 맨 첫 번째 섹터에 위치한다. 부트 섹터는 섹터 당 바이트 수, 클러스터 당 섹터 수, FAT의 개수, 루트 디렉토리 내 엔트리 개수, 전체 섹터 수, FAT 당 섹터 수 등의 정보가 기록된다.

FAT는 부트 섹터 다음에 위치하게 된다. 엔트리의 크기가 12비트, 16비트 32비트인지에 따라서 FAT12, FAT16, FAT32로 나뉜다. 그림 2에 FAT를 이용한 파일 접근 방법을 나타내었다. 전체 클러스터의 개수가 256개일 때, FAT12의 예를 나타내었다. 각 파일의 디렉토리 엔트리 안에 시작 클러스터 번호가 저장된다. 시작 클러스터가 0x005라면, 해당 파일에 할당된 클러스터는 순서대로 0x005, 0x006, 0x007, 0x008, 0x00D,

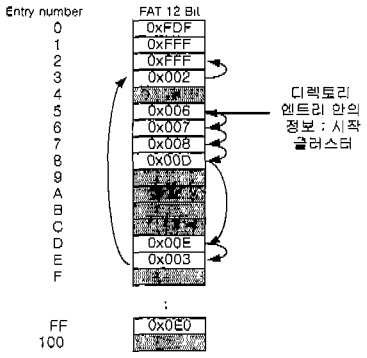

그림 2 FAT에서 파일 접근

0x00E, 0x003, 0x002이다.

FAT는 크기가 작기 때문에 테이블 전체를 메모리에 저장하면, 빠르게 파일을 접근할 수 있다. 단, HDD보다 접근 속도가 큰 플래시 메모리를 사용할 때, FAT를 검색하는데 소요되는 시간이 더 큰 문제로 부각된다. 이런 문제점과 해결 방안은 3.1절에서 설명한다.

2.3 버퍼 캐쉬(buffer cache)

버퍼 캐쉬(buffer cache)를 두어 데이타를 버퍼링한다. 기존의 UNIX 파일 시스템에서 버퍼 캐쉬는 HDD의 느린 속도를 감추어 시스템의 응답 시간과 처리 능력을 향상시키기 위해서 사용된다. 실제 컴퓨터와 HDD 사이의 데이타 전송은 한 섹터(sector) 단위로 이루어지게 되는데, 섹터 단위의 데이타를 버퍼 캐쉬에 저장하여 사용자는 문자(character) 단위로 데이타를 읽거나 쓸 수 있다. 내용이 변경된 버퍼 캐쉬의 블록은 버퍼 캐쉬에서 LRU에 의해서 제거될 때, HDD에 저장된다. 따라서, 여러 번 읽고 써야 하는 HDD 작업도 한 번의 읽기와 쓰기 작업으로 가능하게 된다. HDD 대신 플래시 메모리를 이용한다면 플래시 메모리의 빠른 속도 때문에 버퍼 캐쉬의 효과는 감소된다. 그러나, IC-card와 같은 형태로 플래시 메모리를 이용한다면, 플래시 메모리 접근 속도가 실제 플래시 메모리 읽기/쓰기 속도보다는 I/O 인터페이스의 속도에 의해 좌우될 것이고, 표 1에 예시한 속도보다는 많이 떨어지고, 버퍼 캐쉬에 의한 효과도 커지게 된다. 또한, 플래시 파일 시스템에서 버퍼 캐쉬를 이용함으로써 얻을 수 있는 큰 장점은 플래시 메모리의 수명을 연장시킬 수 있다는 것이다. 플래시 메모리의 가장 큰 단점 중 하나는 지우는 횟수가 제한되어 있다는 것이다. 버퍼 캐쉬를 이용하면, 플래시 메모리에 쓰는 것을 최대한 막을 수 있고, 따라서 플래시 메모리의 수명을 연장시킬 수 있다.

2.4 플래시 메모리 관리자

플래시 메모리 관리자의 기능은 크게 세 가지로 나눌 수 있다.

첫째, LFS 방식으로 데이타를 쓰고, 상위 파일 시스템에는 디스크와 똑같이 보이도록 에뮬레이트한다. 플래시 메모리는 물리적으로 덧쓸 수 없다는 단점이 있다. 한 번 데이타를 쓴 블록에 다시 데이타를 쓰기 위해서는 EU(Erase Unit)을 지운 다음에야 가능하다. 게다가 EU을 지우는데 소요되는 시간도 길기 때문에 매번 데이타를 쓸 때마다 EU을 지운다면 성능이 극단적으로 저하될 것이다. 따라서, 기존의 연구에서는 LFS(Log File System)[7]을 사용하여 효율적으로 데이타를 읽고 쓸 수 있도록 구현하였다. 기존의 플래시 메모리 파일 시스템은 대부분 LFS를 이용하여 구현하였고, 그 효율성이 증명된 상태이기[8] 때문에, 본 논문에서 구현한 플래시 파일 시스템에서도 LFS를 이용하였다. 플래시 메모리 관리자는 데이타를 쓸 때, 소요되는 시간을 줄이는 대신에 상위 파일 시스템에서는 디스크와 같이 보이도록 에뮬레이트(emulate)한다.

또한, 플래시 메모리 관리자는 플래시 메모리를 섹터별로 데이타가 들어있는지(valid), 비어있는지(free), 쓸모 없는 데이타가 들어있는지(invalid)를 구분하여 관리한다. 상위 파일 시스템이 새로운 데이타를 쓰려고 한다면, free 블록 S1를 찾아서 데이타를 쓰고, 이전의 블록은 invalid 상태로 무효화시키고, free 블록의 상태를 free에서 valid로 바꿔주는 일련의 과정을 처리한다. 상위 파일 시스템에서는 일관된 주소로 데이타를 접근하므로, 플래시 메모리 관리자는 상위 파일 시스템에서 넘겨받은 가상의 주소를 실제 주소로 바꾸는 매핑 테이블을 관리하여야 한다.

둘째, 플래시 메모리에 invalid 블록이 쌓이면, EU을 지워서 free 블록을 확보한다. 데이타를 쓰기 위해서는 free 블록이 있어야 한다. 따라서, 플래시 메모리 관리자는 언제나 일정한 양의 free 블록을 확보하고 있어야 한다. EU을 지우는 작업을 cleaning이라고 하고, cleaning policy에 따라 free한 EU의 개수와 invalid 섹터의 개수를 고려하여 cleaning 여부를 결정하게 된다.

셋째, 시스템의 전원이 예상치 못한 경우 꺼졌을 경우에 오류를 복구하는 기능이다. 예를 들어, 데이타를 쓰는 경우나 cleaning 도중에 전원이 꺼졌을 경우에 플래시 메모리의 데이타에 심각한 오류가 발생할 수 있다. Embedded system은 극한 환경에서도 구동될 수 있기 때문에 이러한 오류 복구 기능은 꼭 필요하다. 플래시 메모리 관리자는 시스템 초기화 시 오류가 발생했었는

지 검사하고, 오류를 복구한다.

2.5 플래시 메모리의 데이타 구조

CPU는 메모리 버스를 이용해서 플래시 메모리에 접근할 수 있다. 플래시 메모리로 Intel 사의 28F320S3[3]를 사용하였다. 28F320S3는 한 EU당 128 KBytes, 256개의 블록으로 구성된다. 한 블록은 512 바이트이다. 플래시 메모리에 저장되는 데이타 구조를 그림 3에 나타내었다. 각 EU마다 앞부분에 EU에 대한 정보와 각 블록들에 대한 정보가 저장된다.

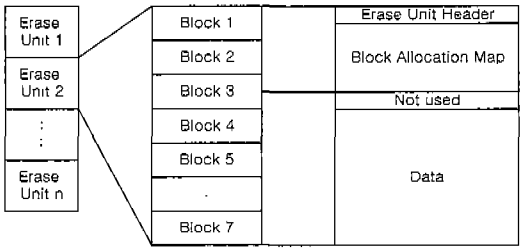

그림 2 플래시 메모리의 데이타 구조

EU의 앞부분에 각 EU의 정보를 기억하는 EUH(Erase Unit Header)가 저장된다. EUH는 64 바이트로 [9]의 형식 중에서 일부만을 사용하였다. Validation code, 현재 EU의 논리적인 번호, 한 블록의 크기, EU의 크기, 첫 번째 EU의 번호, EU의 개수, BAM(Block Allocation Map)의 offset, EU의 상태 정보, EU이 지워진 횟수 등이 EUH에 들어간다. EUH의 정보를 표 2에 나타내었다.

표 2 Erase Unit Header

Field	Size (bytes)	Description
Validation code	5	EU가 valid한지 검사하기 위한 code
LogicalEUN	2	현재 EU의 논리적인 번호
BlockSize	1	블록의 크기
EraseUnitSize	1	EU의 크기
FirstPhysicalEUN	2	플래시 메모리 관리자에서 사용되는 첫 번째 EU의 번호
NumEraseUnits	2	EU의 개수
BAMOffset	4	BAM의 offset
EUstatus	2	EU의 상태
NumErase	4	지워진 횟수
Reserved	41	

각 EU은 valid, free, invalid, prevalid, allocated, erasing 상태 중 하나 이다. Free 상태는 EU이 free 상태라는 뜻, 즉, EU 내의 모든 블록이 free임을 뜻하고, valid 상태는 현재 EU이 사용중이라는 뜻이다. Invalid, prevalid, allocated, eraseing 상태는 오류 복구를 위해서 cleaning 과정 중에서 사용되는 것으로 추후에 설명하겠다.

EUH 뒤에 블록들의 정보를 나타내는 BAM(Block Allocation Map)이 저장된다. 각 블록들이 free인지, invalid인지, valid인지 등의 상태 정보와 논리적인 주소가 무엇인지를 나타낸다. 논리적인 주소는 상위 파일 시스템에서 플래시 메모리 관리자에게 요청하는 주소인데, 플래시 메모리 관리자가 논리적인 주소를 실제의 물리적인 플래시 메모리 주소로 바꾸어야 한다. 한 BAM 엔터리에 4바이트씩 할당되고, 따라서 한 EU내의 BAM 크기는 1028 바이트이다. BAM 엔트리 중 상위 24비트는 논리적인 주소가 저장되고, 하위 8비트는 블록의 상태를 나타낸다. 블록의 상태는 free, invalid, valid, bad, control 중 하나이어야 한다. Valid 블록은 데이타가 저장된 블록이고, prevalid 블록은 데이타가 저장되어 있긴 하지만, 아직 인증이 되지 않은 상태이다. Allocated 블록은 데이타를 저장하기 전에 예약되었음을 나타낸다. Allocated, prevalid 상태는 오류 복구를 위해 데이타를 쓰는 중에 사용되는 것으로 자세한 설명은 뒤에서 하겠다.

BAM이 끝나는 다음 블록부터 실제 데이타가 저장된다. 시스템 초기화 시, 플래시 메모리 관리자는 각 EU의 EUH를 살피고, 전체 EU의 개수, 블록의 크기 등을 초기화한다. 그리고, 오류가 발생한 EU이 있는지 조사하여 오류를 복구한다. 그 다음에 각 EU의 BAM을 조사하여 논리적인 주소와 물리적인 주소의 매핑 테이블(mapping table)을 만든다. 상위 파일 시스템이 플래시 메모리 접근 요청 시 넘겨주는 논리적인 주소를 물리적인 주소로 바꿀 때, 매핑 테이블을 사용한다. 플래시 메모리 관리자는 매핑 테이블을 만드는 동시에 블록에서 오류가 발생하였는지 검사하고, 오류가 발생했다면 복구 작업을 수행한다. 즉, 본 논문에서 구현된 플래시 파일 시스템의 오류 복구 과정은 2단계로 이루어진다. 한 번은 EU 수준에서 이루어지고, 두 번째는 블록 수준에서 이루어진다. 거시적인 관점에서의 오류 복구와 미시적인 관점에서의 오류 복구를 통해서 완벽한 오류 복구 능력을 구현하였다. 상세한 오류 복구 과정은 3.2절에서 설명하겠다.

2.6 데이타 쓰기

데이타를 쓰는 과정은 아래와 같다.

① Free 블록 중 하나를 선택하고, 선택된 블록의 상태를 free에서 allocated 상태로 바꾼다.

② 새로운 블록에 데이타를 저장하고, allocated 상태

에서 prevalid 상태로 바꾼다.

③ 이 전의 블록을 valid 상태에서 invalid 상태로 바꾼다.

④ 새로운 블록의 상태를 prevalid 상태에서 valid 상태로 바꾼다.

한 EU을 active EU으로 선택하고, active EU내의 free 블록을 순차적으로 이용한다. 현재 active EU의 모든 free 블록을 사용하면, 새로운 active EU을 선택한다. 새로운 active EU 선택할 때, valid EU 들에게 우선권을 준다. Valid EU 중에서 지워진 횟수가 가장 적은 EU을 active EU으로 선택한다. 한 EU만 계속해서 사용되는 것을 막고, 모든 EU을 골고루 사용하기 위해서 이와 같은 방법을 사용한다. Valid EU이 없다면, free EU 중에서 지워진 횟수가 가장 적은 것을 active EU으로 선택한다.

플래시 메모리 관리자는 플래시 메모리를 HDD로 에뮬레이트하기 때문에 상위 파일 시스템에서는 블록 장치와 같이 접근하게 된다. 따라서, 한 섹터 단위로 데이타를 넘겨주고, 플래시 메모리 관리자는 한 섹터 단위로만 데이타를 읽거나 쓰게 된다. 따라서, 플래시 파일 시스템의 성능을 향상시키기 위해서는 EU의 블록 크기를 섹터 크기와 같거나 작아야 한다. 블록 크기가 섹터 크기보다 크다면, 바뀐 데이타를 저장하는 것 외에 바뀌지 않는 데이타도 복사하여야 하기 때문에 성능이 떨어지게 된다. 본 논문에서 구현한 플래시 파일 시스템은 플래시 메모리의 블록 크기와 섹터 크기는 512 바이트로 같다.

블록의 상태 정보는 BAM에 기록된다. 블록의 상태 정보를 이와 같이 세분화시킨 이유, 데이타를 쓰는 도중에 오류가 발생하여 플래시 메모리 데이타의 일관성(consistency)을 잃지 않도록 하기 위함이다. 시스템 초기화 시에 세분화된 블록 상태 정보를 이용하여 오류를 복구하게 된다.

2.7 Cleaning

데이타를 쓰는 과정에서 쓸모 없는 invalid 블록이 발생한다. 따라서, 플래시 메모리 관리자는 주기적으로 invalid 블록을 free 블록으로 바꾸어야 한다. Cleaning policy에 따라서 언제 어떤 EU을 cleaning할지 결정하게 되는데, 그것에 대해서는 3.3 절에서 자세히 다루고, 본 절에서는 cleaning이 어떤 과정으로 이루어지는지 설명하겠다.

Cleaning 과정은 데이타 쓰기 과정과 비슷하다.

① Free EU를 하나 고르고, free EU의 상태를 allocated 상태로 바꾼다.

② 백업(back-up) EU(새로운 EU)에 valid 블록의 데이타를 복사하고, BAM과 주소 매핑 테이블을 바꾼다. EU를 allocated 상태에서 prevalid 상태로 바꾼다.

③ 원본 EU(Cleaning될 EU)을 valid 상태에서 invalid 상태로 바꾼다.

④ 백업 EU의 상태를 prevalid 상태에서 valid 상태로 바꾼다.

⑤ 원본 EU의 상태를 erasing 상태로 바꾼 뒤에 지운다.

EU의 상태 정보는 EUH에 저장된다. EU를 지우는 작업은 플래시 메모리에서 가장 시간이 많이 걸리는 작업으로 플래시 파일 시스템의 성능을 저하시키는 가장 큰 요인이다. 그리고, 원본 EU에서 백업 EU로 valid 블록을 복사하는 작업도 성능 저하의 이유 중 하나 이다.

3. 플래시 파일 시스템의 특징

본 장에서는 플래시 파일 시스템의 특징과 구현하면서 발생한 문제점과 해결 방안에 대해서 설명하겠다. 3.1절에서는 FAT 구현시의 문제점 해결 방법에 대해서 설명하고, 3.2 절에서는 오류 복구 기능의 구현에 대해서 설명한다. 마지막으로 3.3 절에서 본 논문에서 제안한 cleaning policy에 대해 이야기하겠다.

3.1 FAT의 문제점과 해결 방법

FAT 파일 시스템 구현 시 두 가지 문제점을 간과할 수 있다. 두 가지 문제점은 모두 linked-list로 되어 있는 FAT의 특징 때문이다.

첫째, FAT 파일 시스템은 데이타의 정보가 linked-list 형태로 저장되기 때문에 파일 크기가 커지면 원하는 클러스터를 찾는데 소요되는 시간이 길어져서 파일 시스템 성능이 떨어지게 된다. 본 논문에서 구현한 파일 시스템에서는 이러한 문제점을 개선하기 위해서 inode 방법을 이용하였다. 즉, vnode에 별도의 field를 두고, 기존의 unix에서 사용하는 방식과 같은 inode 캐쉬(cache)를 vnode 내에 저장시키는 방법이다. 파일을 접근하기 위해서 FAT 테이블을 검색할 때 inode 캐쉬를 만든다. 따라서, 이전의 파일 접근의 최대 오프셋(offset)보다 작으면 바로 클러스터를 찾을 수 있게 된다. 예를 들어, 클러스터의 크기가 c인 FAT 파일 시스템에서 파일을 열고, 데이타를 읽는 경우를 생각해보자. 5*c 오프셋의 데이타를 읽는 경우, 처음에는 inode 캐쉬에 저장된 정보가 없으므로, FAT 테이블을 검색하여야 한다. FAT 테이블을 검색하는 과정에서 알아낸 0, c, 2*c, 3*c, 4*c, 5*c 오프셋의 위치 정보를 inode 캐쉬에 저장한다. 그 다음에 3*c 오프셋의 데이타를 읽는다면, 3*c 오프셋의

위치는 inode 캐쉬에 저장되어 있으므로, 바로 찾아 읽을 수 있다. Inode 캐쉬에는 5*c 오프셋까지 위치 정보가 저장되어 있으므로, 5*c 이전 오프셋의 데이타는 FAT 테이블 검색 없이 바로 읽을 수 있고, 5*c 이후 오프셋의 데이타를 읽을 경우에는 FAT 테이블 검색이 이루어지고, 검색 과정에서 얻을 수 있는 모든 위치 정보는 inode 캐쉬에 저장된다. 즉, 한번만 FAT 테이블을 검색하면 되기 때문에 빠른 FAT를 구현할 수 있다. 단, inode 방법은 메모리를 많이 필요로 한다.

메모리가 부족한 시스템에서는 간단하게 캐싱(caching) 방법을 사용할 수 있다. 최근에 접근한 클러스터를 파일의 vnode에 저장하여 상관성이 있는 파일 접근 시 성능을 향상시킬 수 있는데, 차례 쓰기(sequential write)에 효과가 극대화되지만, 무작위 쓰기에서는 효과가 별로 나타나지 않는다.

둘째로, 새로운 클러스터를 할당할 때에 발생하는 문제점이다. 새로운 클러스터를 할당할 때, FAT 테이블을 뒤져서 free 클러스터를 찾아야 하기 때문에 성능이 저하된다. 이 문제점은 free 클러스터의 위치를 예측하도록 구현하여 해결하였다. 어떤 파일이 차지하고 있는 클러스터는 순서대로 되어있거나 서로 가까운 곳에 할당되어 있을 가능성이 크다. 따라서, 파일이 지워진 후에 생기는 free 클러스터들도 서로 가까운 곳에 있을 것이다. 따라서, free 클러스터를 할당한 뒤에는 그 다음 클러스터를 free 클러스터라 예측할 수 있다. 만약, 그 다음 클러스터가 free 클러스터가 아니라도 가까운 곳에 free 클러스터가 있을 확률이 높으므로 빠른 시간에 free 클러스터를 검색할 수 있다.

3.2 오류 복구 기능

예기치 않은 경우에 전원이 꺼지면, 플래시 파일 시스템에는 두 가지 문제점이 발생한다.

첫째, 플래시 메모리 관리자가 플래시 메모리에 데이타를 기록하거나 cleaning 과정 중에 전원이 꺼지게 되면, 매핑 테이블에 이상이 발생하고, 기존의 데이타를 잃어버리게 될 수 있다. 더욱이, 잃어버린 데이타가 meta 데이타라면 플래시 파일 시스템이 회복 불능 상태가 될 수 있다.

둘째, 버퍼 캐쉬에서 플래시 메모리로 write-back 되지 않은 데이타를 유실하게 된다. 이와 같은 경우를 위해서 write-through 방식을 이용할 수도 있지만, 데이타를 쓸 때마다 플래시 메모리에 데이타를 쓴다면, 플래시 파일 시스템의 성능이 크게 저하된다. 이 경우에도 파일 시스템의 일관성(consistency)이 깨질 수 있다. 아래에서 각각의 문제점의 해결 방안에 대해서 논한다.

원본 EU	백업 EU	
Valid	Free	
Valid	Allocated	(1)
Valid	*Prevalid*	(2)
Deleted	Prevalid	(3)
Deleted	Valid	(4)
Erasing	Valid	(5)
Free	Valid	(6)

그림 3 Cleaning 과정

가. 플래시 메모리 관리자의 오류 복구 기능

데이타를 쓰거나 cleaning 작업 중간에 전원이 나가거나 기타 다른 이유로 오류가 발생하면, 같은 주소의 데이타가 두 개 이상의 블록에 존재하는 일관성(consistency) 문제와 기존의 데이타를 잃어버리게 되는 문제가 발생한다.

본 논문에서 구현한 플래시 파일 시스템은 오류 복구를 위해서 데이타를 쓰는 과정과 cleaning 과정을 세분화하였다. 오류 복구 작업은 시스템 초기화 시에 수행된다. 시스템 초기화 시 플래시 메모리 관리자는 먼저 각 EU의 상태 정보를 보고, 오류가 발생했었는지를 검사하고, 오류가 발생한 경우 복구 작업을 시작한다. 그 다음 플래시 메모리의 BAM을 읽고, 논리적인 주소를 물리적인 주소로 바꾸어 주는 매핑 테이블을 만든다. BAM을 읽으면서 각 블록들의 상태 정보를 보고 오류 발생 여부를 조사하고 오류가 발생한 경우 복구한다. 즉, 논문에서 구현한 플래시 파일 시스템은 2단계에 걸쳐서 오류 복구 작업을 수행한다. EU 수준에서의 오류 복구와 블록 수준에서의 오류 복구를 수행하여 오류에 대한 강인성을 증가시켰다.

오류가 발생하지 않았을 경우 일반적인 EU의 상태는 free, valid 상태이어야 한다. 시스템 초기화 시 allocated, prevalid, invalid, erasing 상태인 EU이 있다면, 시스템의 전원이 비정상으로 꺼져서 오류가 발생한 경우이다. 그림 5에 cleaning 과정을 나타내었다. Cleaning 중에 각 EU의 상태 변화를 세분화하여 원본 EU와 백업 EU의 상태 정보에 따라서 오류가 발생한 시점을 알 수 있도록 하였다. 즉, 예를 들어 어떤 EU의 상태가 prevalid 상태라면, (3)번 또는 (4)번 시점에서 오류가 발생한 것이다. 원본 EU과 백업 EU은 EUH의 LogicalEUN이 같으므로, 같은 LogicalEUN을 가진 EU를 찾아 상태를 조사하면 (3)인지 (4)인지 판단할 수 있

원본 블록	백업 블록	
Valid	Free	
Valid	Allocated	(1)
Valid	Prevalid	(2)
Deleted	Prevalid	(3)
Deleted	Valid	(4)

그림 4 데이타 쓰기 과정

다. (1)~(6)까지 오류가 발생한 시점에 따라 오류 복구 방법은 다음과 같다.

(1) Cleaning 작업이 시작되기 전이므로 그냥 둔다.

(2) 데이타 복사가 완료되기 전에 종료된 경우이다. 백업 EU의 상태를 invalid로 바꾸고, 지운다.

(3) 데이타 복사가 완료된 후에 종료된 경우이다. 원본 EU는 invalid 상태로 바꾸고 지운다. 백업 EU는 valid 상태로 바꾸어준다. 원본 EU와 백업 EU는 같은 LogicalEUN을 갖는다.

(4) 원본 EU를 지우고, 백업 EU는 valid 상태로 바꾸어준다.

(5)(6) 원본 EU를 지운다.

데이타를 쓰는 과정에 따른 오류 복구는 좀 더 간단하다. 오류가 발생하지 않았을 경우 일반적인 블록의 상태는 free, valid, invalid 상태이다. Allocated, prevalid 상태의 블록이 있다면, 오류가 발생했다는 뜻이다. 그림 6에 데이타 쓰기 과정을 나타내었다. EU과 마찬가지로 각 블록의 상태 정보를 이용해서 오류가 발생한 시점을 알아낼 수 있다. 원본 블록과 백업 블록은 같은 논리적인 주소를 가지기 때문에 BAM을 조사하여 찾아낼 수 있다.

(1) -

(2) 데이타 복사가 완료되기 전에 종료된 경우이다. 백업 블록의 상태를 invalid 상태로 바꾼다.

(3) 데이타 복사가 완료된 후에 종료된 경우이다. 원본 블록은 invalid 상태로 바꾸고, 백업 블록의 상태는 valid 상태로 바꾼다.

(4) 백업 블록의 상태를 valid 상태로 바꾼다.

이와 같은 과정을 통해서 오류를 복구하게 된다. 본 논문에서 구현한 플래시 파일 시스템은 데이타를 쓰거나 cleaning 과정 중에 발생한 오류를 완벽하게 복구하여 이전의 상태로 롤백할 수 있다.

나. Soft update

비정상적으로 전원이 꺼졌을 때, 버퍼 캐쉬에서 플래시 메모리로 write-back 되지 않은 데이타를 유실하게 된다. 이런 경우에 파일 시스템의 일관성이 깨질 수 있다. 예를 들어, 어떤 파일이 생성될 때, FAT 정보가 디렉토리 엔트리 정보보다 먼저 플래시 메모리에 저장되어야 한다. 만약, 순서가 바뀌어, 디렉토리 엔트리 정보만 플래시 메모리에 저장되고, FAT 정보가 미처 기록되기 전에 전원이 꺼진다면, FAT 정보가 이상한 데이타를 가지고 있기 때문에 문제가 발생한다. 파일 시스템의 메타(meta) 정보들 사이의 의존 관계에 따라서 순서대로 플래시 메모리에 저장되어야 한다.

Soft update[11][12]는 dirty 블록이 플래시 메모리로 저장될 때, 의존 관계에 의한 순서대로 저장되도록 하여 파일 시스템의 일관성을 지켜준다. 새로운 파일을 생성될 때, 디렉토리 엔트리를 포함한 블록이 FAT 정보가 저장된 뒤에 저장되도록 하고, 파일이 삭제될 때는 디렉토리 엔트리를 포함한 블록이 FAT 정보보다 먼저 저장된다. 따라서, 버퍼 캐쉬로 인해서 파일 시스템의 일관성이 깨지는 것을 막아준다.

3.3 Cleaning policy

Cleaning policy에 따라서 언제 어떤 EU를 cleaning 할 지 결정한다. Cleaning이 발생하는 경우는 일반적인 경우와 특수한 경우로 나눌 수 있다. 특수한 경우란 당연히 cleaning을 해야할 경우인데, 어떤 EU이 invalid 블록으로만 구성되어 있는 경우나 EUH의 validation code가 잘못 설정되어 있는 경우를 뜻한다. Invalid 블록만으로 구성된 EU나 EUH의 validation code가 잘못 설정되어 있는 EU가 있으면, 해당 EU를 cleaning한다.

Cleaning policy에 사용되는 각종 변수는 시스템 초기화 시에 정의된다. 플래시 파일 시스템의 성능 면에서는 cleaning은 최대한 연장되는 것이 좋다. 늦어지면 늦어질수록 invalid 블록이 많아지게 되고, 결과적으로

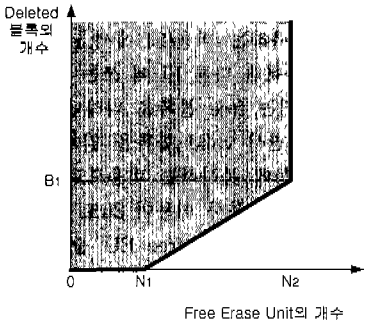

그림 5 Cleaning

valid 블록이 줄어들기 때문에 복사 비용이 낮아지게 된다. 반대로 플래시 메모리 관리자는 언제나 일정한 개수의 free EU을 가지고 있어야 한다. 이와 같은 점을 고려해서 *cleaning policy*가 결정되어야 하고, 다른 프로세스(process)와 병렬로 수행하여 전체 시스템 성능을 저하시키지 않도록 하여야 한다.

그림 4에 일반적인 경우 cleaning이 언제 발생하는지를 도시하였다.

- N_2개 이상의 충분한 free EU이 있으면, cleaning은 발생하지 않는다.
- Free EU의 개수가 N_2개이면, invalid 블록의 개수가 B_1이상인 EU을 cleaning한다. Free EU의 개수가 줄어들수록 invalid 블록의 개수 기준은 줄어들게 된다.
- Free EU의 개수가 N_1개 이하이면, 무조건 cleaning이 발생한다.

현재 구현에서 N_1=3, N_2=10, B_1=255를 사용하였다.

Cleaning될 EU의 선택은 두 가지 방법으로 구현되어 있다. 첫째로, 본 논문에서 제안한 방법으로 invalid 블록이 가장 많은 EU을 cleaning될 EU으로 선택하는 방법이다. Invalid 블록이 가장 많은 EU을 cleaning 한다면, free 블록을 가장 많이 얻을 수 있다. 기존의 "greedy" 방법[8]은 가장 적은 수의 valid 블록을 가진 EU를 cleaning될 EU로 선택한다. 만약, valid 블록은 적고, invalid 블록도 적지만, free 블록은 많은 EU를 cleaning한다면, cleaning 후에 얻을 수 있는 free 블록의 개수는 별로 없게 된다. 따라서, 본 논문에서는 cleaning 후에 최대의 효과를 얻을 수 있는 새로운 방법을 제안하였다.

둘째로, "cost-effective" 기법[7]은 아래의 공식에 따라서 가장 효과적인 EU을 선택하는 방법이다.

$$\frac{benefit}{cost} = \frac{age \times (1-u)}{2u}$$

여기서 u는 EU의 사용도(utilization)를 나타내고, *age*는 최근에 변경된 후의 시간을 나타낸다. $2u$와 $(1-u)$항목은 각각 복사하는 비용(EU에서 u개의 valid 블록을 읽고, u개의 블록을 다른 EU에 쓰는 것)과 어쩔 수 없이 같이 reclaim되는 free 블록을 나타낸다. LFS의 경우에는 cleaning하면서 valid 블록을 읽기 위해 전체 EU을 모두 읽기 때문에 복사하는 비용으로 $1+u$가 되는 것에 반해 이 경우에는 복사 비용이 2u로 줄어든다.

백업(back-up) EU는 free EU 중에서 erase 횟수가 가장 적은 EU를 선택한다. 따라서, 한 EU만 계속해서 사용되는 hot-spot 문제점을 막을 수 있다. 한 EU만 계속해서 사용하여 erase 횟수가 증가하면, 전체 플래시 메모리의 수명도 줄어들게 된다. 백업 EU를 erase 횟수를 고려하여 선택함으로써 이와 같은 문제점을 해결할 수 있다.

현재 구현된 플래시 파일 시스템은 항상 cleaning 여부를 조사하도록 되어있다. 그러나, free EU의 개수를 조사하여 free EU의 개수가 N_2개보다 훨씬 많다면, cleaning 여부를 조사하는 cleaner를 끄고, 파일 시스템의 로드를 줄일 수 있다.

4. 실험 결과

4장은 아래와 같이 구성되어 있다. 4.1절은 실험 환경에 대해서 설명하고, 4.2 절에서 FAT의 문제점이 성능에 미치는 영향을 비교 분석하였으며, 4.3 절에서 차례쓰기(sequential write), 4.4 절에서 무작위 쓰기(random write) 결과를 분석하였다.

표 3 실험 환경

플래시 메모리 시스템	
플래시 메모리	Intel 28F320S3 (32Mbit/chip)
EU 개수	32 개
EU 크기	128 Kbytes
블록 개수	8192 개
메타(meta) 블록	96 개
데이타 블록	8004 개
Erase cycle	0.5 초
Write performance	5.76 μs/word
Hardware	
CPU	SA-110S 233 MHz (Intel StrongARM RISC CPU)
메모리	32Mbytes
버스	PCI (Intel 21285)

4.1 실험 환경

실험 환경은 표 3에 나타낸 바와 같다. 실험에 사용한 하드웨어(Hardware)는 전자통신연구소(ETRI)에서 개발한 디지털-TV 셋톱 박스(set-top box)을 사용하였다. 셋톱 박스는 CPU로 인텔사의 SA-110S를 사용하고, 32Mbytes의 RAM을 사용하였다. 총 32개의 EU로 구성된 4 MBytes의 플래시 메모리를 이용하여 실험하였다.

4.2 FAT 성능 비교

FAT[5][6]는 MS-DOS에서 사용되는 파일 시스템으로 작은 용량의 HDD를 관리하는데, 좋은 성능을 나타내는 것으로 알려져 있다. 그러나, 큰 파일을 쓰거나 읽을 때, linked list 형태의 FAT 테이블을 차례대로 찾아가야 하기 때문에 큰 성능 저하가 발생할 수 있다. 또한, 어떤 파일에 클러스터를 할당할 때도 FAT를 차

표 4 FAT 성능 비교 (kb/sec)

FAT		FAT ver.1	FAT ver.2	FAT ver.3
Average Throughput	1 file	26.3	105.4	108.0
	2 files	43.4	107.2	109.7
	5 files	68.7	108.1	110.8
	10 files	84.1	108.7	111.3
	20 files	94.6	108.9	111.6

례대로 검색해야하는 단점이 있다. 따라서, FAT를 구현할 때는 위와 같은 점을 고려하여야 한다. 본 논문에서 구현한 플래시 파일 시스템은 위와 같은 두 가지 문제점을 해결하기 위해서 몇 가지 방법을 사용하였다. 먼저, 큰 파일 접근 시의 문제점을 해결하기 위해서 inode 방법을 사용하였고, free 클러스터 할당 시의 문제점을 해결하기 위해서 free 클러스터 예측 방법을 사용한다. 각 방법을 사용을 하였을 때, 성능 향상 정도를 표 4에 나타내었다. FAT ver. 1은 아무 방법도 사용하지 않았을 때이고, FAT ver. 2는 inode 방법을 사용하였을 때, FAT ver. 3는 inode 방법과 free 클러스터 예측 방법을 모두 사용하였을 때이다. 파일 개수를 증가시키며, 성능을 측정하였는데, 파일 크기의 합은 언제나 전체 플래시 메모리의 85%인 3.4 MBytes가 되도록 하였다. 플래시 메모리에 저장된 데이타의 크기가 성능에 영향을 미칠 수 있기 때문이다. 각각의 파일을 차례 쓰기하고, 모든 파일에 대한 처리가 끝나면, 다시 처음부터 차례 쓰기를 한다. 표 4에는 100 MBytes의 데이타를 쓰고 난 뒤의 평균 속도를 나타내었다.

파일 개수가 많아질수록, 파일 크기가 줄어들므로, FAT 테이블을 검색하는데 소요되는 시간도 줄어들게 된다. 따라서, FAT ver.1은 파일 크기에 따라서 성능이 크게 변한다. FAT 테이블 검색 시간은 파일 크기에 비례하기 때문에 큰 파일에 대한 성능이 크게 떨어진다.

반면에 FAT ver.2나 FAT ver.3는 파일 크기에 관계없이 거의 일정한 속도를 유지한다. Inode 방법을 사용하여 FAT 테이블 검색 시간이 크게 줄고, 일정하기 때문이다. 상대적으로 파일이 클 때, 큰 성능 향상이 있게 된다. FAT ver.2나 FAT ver.3에서도 파일이 작아질수록 성능이 아주 조금씩 좋아지는데, inode 방법에서도 최초의 파일 접근에서는 FAT 테이블을 검색해야하기 때문이다.

Free 클러스터 예측 방법을 사용하면, 약간의 성능 향상을 볼 수 있다. 이것은 맨 처음에 파일을 쓸 때에는 새로운 클러스터를 할당받지만, 그 다음부터는 이미 할당받은 클러스터를 이용하기 때문에 Free 클러스터 예측 방법의 효과가 처음에만 나타나기 때문이다. 새로운 파일을 만들거나, 지우는 실험을 한다면 더 큰 성능 향상을 볼 수 있을 것이다. 다음 절의 실험들은 모두 FAT ver.3를 사용한 결과들이다.

4.3 차례 쓰기

한 파일을 차례대로 순차적으로 쓸 때의 성능을 비교하였다. 차례 쓰기는 파일 시스템이 최대 성능을 얻을 수 있는 실험이다. 플래시 메모리에 valid 블록이 순서대로 위치하고, invalid 블록도 순서대로 발생하기 때문에 cleaning이 발생할 때, EU내의 모든 블록이 invalid 상태에 있게 된다. 따라서, cleaning 후에 얻을 수 있는 free 블록의 수도 많고, cleaning의 효율성이 최대가 된다.

차례 쓰기 실험 결과는 표 5와 그림 7과 같다. 파일의 크기를 전체 플래시 메모리 크기의 30%, 60%, 90%일 때의 성능을 측정하였다. 포맷(format)된 플래시 메모리에 맨 처음에 정해진 크기의 파일을 저장하고, 그 다음부터는 처음에 저장된 데이타 위에 덧쓰는 작업을 반복한다. 그림 7에서 x 축이 플래시 메모리에 쓴 데이타의 누적 양을 나타낸 것이고, y 축은 초당 저장한 Kbytes 수를 나타낸다. 버퍼 캐쉬의 크기는 512 KB로 하였다. 또한, 플래시 파일 시스템(FFS)의 성능과 상위 파일 시스템을 뺀 플래시 메모리 관리자(RAW)의 성능을 비교하여 상위 파일 시스템의 성능이 어떤 영향을 미치는지 살펴보았다.

표 5 차례 쓰기 결과 정리

파일 크기		평균 성능	지워진 EU의 개수	복사된 블록 개수
FFS	30%	110.7	795	0
	60%	109.5	788	225
	90%	21.1	4594	964952
RAW	30%	120.6	799	0
	60%	121.4	789	0
	90%	29.5	3364	653810

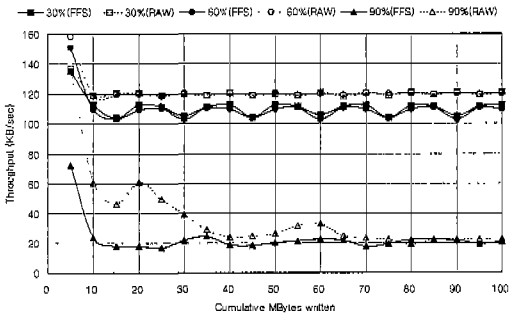

그림 6 차례 쓰기 결과

전체적으로 플래시 파일 시스템의 성능이 낮게 나왔는데, 차례 쓰기에서는 파일 크기가 버퍼 캐쉬보다 크면, 버퍼 캐쉬의 효과가 거의 나타나지 않고, 파일 시스템을 거치면서 CPU를 소모하기 때문이다. 따라서, 상위 파일 시스템에서 여러 계층을 거치기 것이 성능을 저하시킨다. 파일 크기가 30%, 60%일 때, RAW와 FFS의 지워진 EU의 개수와 복사된 블록 개수가 거의 비슷하지만, 성능에서 차이가 나는 점이 그런 특성을 말해주고 있다. RAW는 60%에서 복사되는 블록이 없지만, FFS는 60%에서 복사되는 블록이 있는데, 이것은 파일 데이타 외에 FAT 테이블과 같은 추가적인 정보가 플래시 메모리에 저장되어 실제 데이타의 크기가 더 크게되기 때문이다.

FFS, RAW의 경우 모두 파일 크기가 90%가 되면, 성능이 급격하게 떨어진다. 플래시 메모리에 valid 블록이 대부분을 차지하기 때문에 *cleaning* 작업에서도 복사되는 블록이 많아진다. 따라서, cleaning의 효율성이 떨어지고, cleaning 작업도 자주 발생하게 된다. 빈번한 cleaning에 의해서 성능도 저하되고, 복사되는 블록의 개수가 많기 때문에 cleaning에 소요되는 시간도 길어진다. FFS는 파일 데이타 외에 FAT 테이블이나 루트 디렉토리 정보 등이 추가적으로 저장되기 때문에 더 많은 cleaning과 데이타 복사가 발생하게 된다. 데이터 양에 따라서 성능이 떨어지는 시점은 cleaning policy의 변수에 의해 좌우된다.

본 논문에서는 기존의 greedy 방법이 아직 free 블록이 많은 EU를 선택할 수 있는 문제점을 가지고 있어서, 그러한 문제점을 없애기 위해 invalid 블록이 가장 많은 EU를 cleaning하는 새로운 cleaning policy를 제안하였다. 본 논문에서 제안한 cleaning policy와 기존의 greedy 방법과 cost-effective 방법들의 성능을 그림 8에서 비교하였다. 그림 8은 파일 크기에 따라 cleaning policy별로 차례 쓰기 성능을 나타낸 것이다. 'clean1'은 본 논문에서 제안한 cleaning policy이고, RAW는 상위 파일 시스템을 사용하지 않은 경우, clean2는 기존의 greedy 방법을 사용한 경우, clean3는 기존의 cost-effective 방법을 사용한 경우이다.

파일 크기가 80% 이상에서 성능이 떨어지기 시작하는 이유는 다음과 같다. 플래시 파일 시스템은 실제 플래시 메모리를 100% 모두 데이타를 저장하기 위해서 사용할 수 없다. 각 EU과 블록들에 대한 정보를 저장해야 하기 때문에 실제 데이타 블록은 98.83%만 사용할 수 있다. 또한, cleaning policy는 언제나 일정한 양의 free 블록과 free EU를 확보하려 하기 때문에 파일 크기가 커짐에 따라서 더 빈번한 cleaning이 발생할 수 밖에 없다. 파일 크기가 10% ~ 70%일 때는 거의 일정한 성능을 보인다. 상위 파일 시스템을 사용하면, 파일 크기가 10%일 때는 파일 크기가 버퍼 캐쉬 크기와 비슷해지기 때문에 성능이 조금 향상된다. 80% 이상에서 성능이 급격하게 떨어지는데, 85%나 90%에서는 EU을 지우거나 valid 블록을 복사하는데 소요되는 시간이 성능에 큰 영향을 미치기 때문에 FFS와 RAW의 차이가 점점 줄어든다.

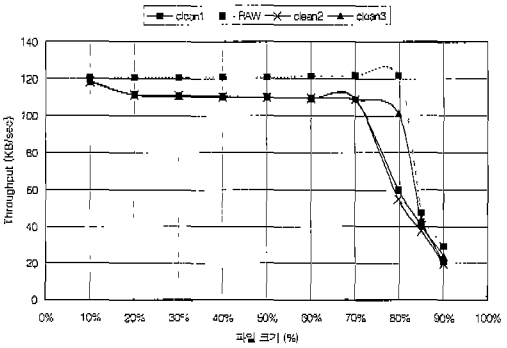

그림 7 파일 크기에 따른 cleaning policy별 차례 쓰기 성능 변화

파일 크기가 70% 이하일 때는 cleaning policy 별로 성능 차이가 거의 나지 않는다. 80% 이상에서는 성능이 차이가 나는데, clean3가 가장 좋은 성능을 보이고, clean1과 clean2는 거의 비슷한 성능이 나왔다. 이것은 clean3가 *age*를 고려하여 cleaning될 EU를 선택하기 때문이다. 가령, clean1이나 clean2에서는 active EU가 cleaning될 EU로 선택될 수 있다. (Invalid 블록을 많이 가지고 있는 경우) 따라서, 아직 free 블록이 있는 EU(active EU)를 선택하여 erase 하게 되고, cleaning 효율이 떨어진다. 반면에, clean3는 *age*를 고려하기 때문에 active EU가 선택될 확률이 크게 낮아지므로, 성능이 좋아진다. 특히, 차례 쓰기의 경우에는 invalid 블록이 두세 개의 EU에 몰려있기 때문에 이러한 성능 저하가 두드러지게 나타난다. 무작위 쓰기에서는 clean3가 오히려 나쁜 성능을 갖는 것을 볼 수 있었다.

Clean1은 기존의 greedy 방법보다 약 3%정도 성능이 향상된다. Free 블록이 있는 EU가 선택되는 확률을 줄이는 것이 약간의 효과를 본 수 있었다.

4.4 무작위 쓰기

무작위 쓰기는 파일 시스템의 최저 성능을 알아보기 위한 실험이다. 차례 쓰기에서는 invalid 블록도 순차적

으로 발생하고, 따라서 invalid 블록이 몇몇 EU에 몰려 있게 된다. 그러나, 무작위 쓰기에서는 invalid 블록이 무작위로 발생하고, 각각 EU은 valid 블록과 invalid 블록이 혼재해 있게 된다. Valid 블록과 invalid 블록이 같이 있으면, cleaning 작업 시 valid 블록 복사에 더 많은 시간이 소요되고, cleaning 후에 얻을 수 있는 free 블록의 수도 적어지게 된다. 즉, cleaning 비용은 높아지고, 효율성이 떨어지기 때문에 더 많은 cleaning 작업이 필요하게 되고, 성능이 저하된다.

그림 9와 표 6에 무작위 쓰기의 결과를 나타냈다. 차례 쓰기와 마찬가지로 FFS와 RAW의 결과를 비교하였다. 전체적으로 차례 쓰기에 비해서 성능이 떨어지는 것을 볼 수 있다. 파일 크기가 커질수록 invalid 블록이 더 많은 EU로 분산되기 때문에 속도가 더 떨어진다. 파일 크기가 30%일 때는 성능이 약간 향상되는데, 버퍼 캐쉬 때문에 실제 플래시 메모리 접근이 줄어들기 때문이다. 버퍼 캐쉬의 크기를 512 KByte로 하였으므로, 평균적으로 1/3 이상의 데이타가 버퍼 캐쉬에 존재하고, 쓰려고 하는 데이타가 버퍼 캐쉬에 존재하면, 상당한 속도 향상을 얻을 수 있다. 파일 크기가 커질수록 쓰려고 하는 데이타가 버퍼 캐쉬에 존재할 확률은 줄어들고, 성능 향상도 줄어들게 된다.

표 6 무작위 쓰기 결과 정리

파일 크기		평균 성능	지워진 EU의 개수	복사된 블록 개수
FFS	30%	107.5	826	22030
	60%	59.4	1548	195657
	90%	6.8	14451	3456211
RAW	30%	106.9	889	26057
	60%	61.1	1584	202656
	90%	6.4	15486	3720725

무작위 쓰기에서 cleaning policy별로 파일 크기에 따른 성능 변화 결과를 그림 10에 도시하였다. 시간이 어느 정도 지나면, 각 EU에 valid 블록이 고루 분포된다. 따라서, 파일 크기가 커질수록 각 EU 내 valid 블록은 많아지고, 그 외의 블록, free 블록과 invalid 블록은 줄어든다. 따라서, cleaning될 때, 파일이 커질수록 invalid 블록의 수가 줄어들므로, cleaning의 효율성이 떨어진다. 같은 이유로 cleaning시 블록 복사도 늘어나고, 빈번하게 cleaning이 발생한다. 따라서, 파일 크기가 커지면, 성능이 점점 떨어지게 된다.

파일 크기가 10%일 때, clean1, clean2, clean3의 성능이 150 KBytes/sec을 넘어 차례 쓰기 성능의 두 배 이상이 되는데, 이것은 버퍼 캐쉬 때문이다. FFS가 RAW보다 성능이 좋은 것도 버퍼 캐쉬로 인해 성능이 향상되기 때문이다. 그러나, 파일 크기가 버퍼 캐쉬보다 훨씬 커지게 되면, 거의 효과를 볼 수 없게 된다.

Clean1과 clean2의 성능 차이가 차례 쓰기에 비해서 줄어들었고, clean3은 제일 성능이 좋지 않았다. 무작위 쓰기에서는 *age*가 거의 영향을 미칠 수 없고, 오히려 부적절한 EU(invalid 블록이 적은 혹은 valid 블록이 많은 EU)를 cleaning할 수 있기 때문이다. 결과적으로 *age*를 이용하여 cleaning될 EU를 선택하는 것은 쓰기 패턴에 따라서 득이 될 수도 있고, 실이 될 수도 있다. 본 논문에서 제안한 방식에 *age*에 대한 개념을 추가하고, 실제 파일 접근 패턴에 따라서 *age*의 weight를 조

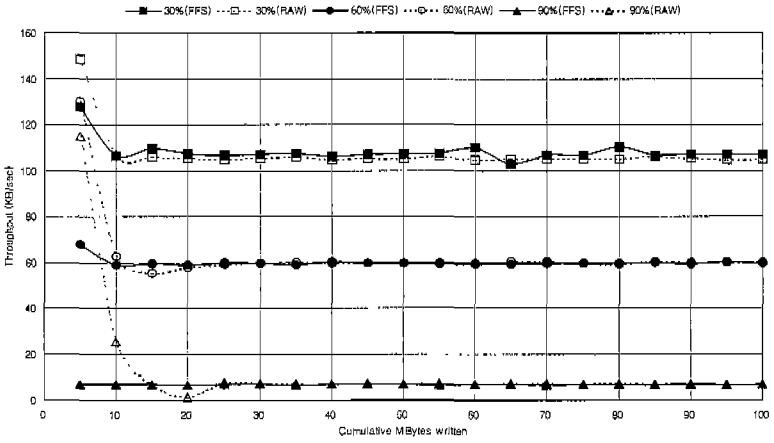

그림 8 무작위 쓰기 결과

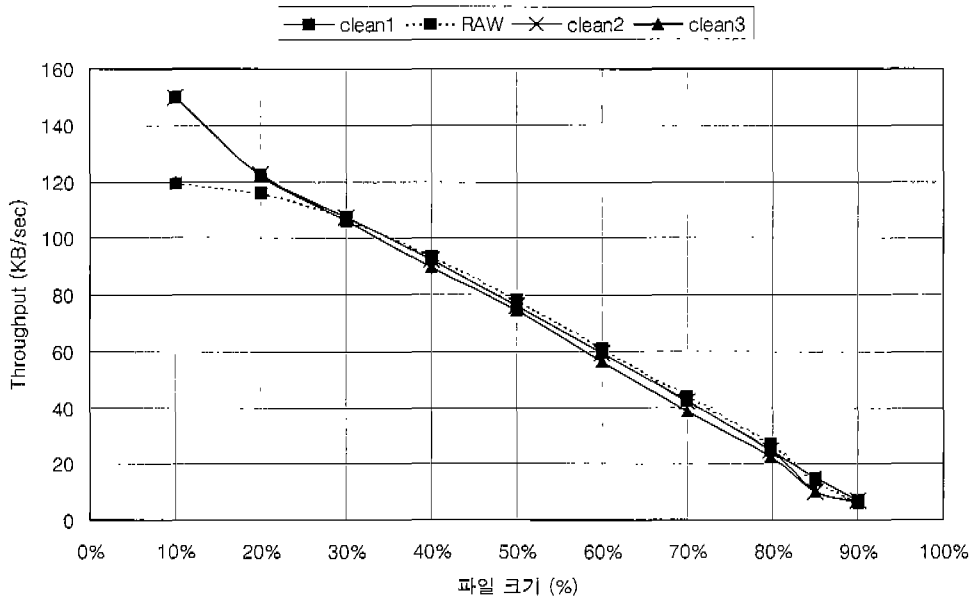

그림 9 파일 크기에 따른 cleaning policy별 무작위 쓰기 성능 변화

절하면, 좋은 성능을 볼 수 있을 것이다.

5. 결론 및 추후과제

본 논문에서는 플래시 메모리를 이용한 플래시 파일 시스템을 구현하고, 성능을 평가하였다. 플래시 파일 시스템은 VFS와 FAT를 이용하여 상위 파일 시스템을 구현하였으며, FAT를 그대로 적용하였을 때, FAT 검색, free 클러스터 검색에서의 문제점을 해결하였다. 또한, 효율적인 플래시 메모리 접근을 위해서 플래시 메모리 관리자를 설계하였고, 효율적인 cleaning policy를 제안하였다.

실험을 통해서 플래시 파일 시스템의 성능을 평가하였다. 성능 비교를 통해서 FAT의 문제점이 얼마나 성능을 저해시키는지 살펴보았으며, 차례 쓰기와 무작위 쓰기를 통해서 플래시 파일 시스템과 상위 파일 시스템이 없을 때의 성능을 비교 평가하였다. 또한, 본 논문에서 제안한 cleaning policy와 기존의 방법들의 성능 비교를 통해 본 논문에서 제안한 cleaning policy가 기존의 greedy 방법보다 좋은 성능을 갖는 것을 알 수 있었다. 나아가, 본 논문에서 제안한 cleaning policy에 *age* 개념을 도입한다면 더 좋은 성능을 가질 수 있을 것이다.

플래시 메모리 관리자는 논리적인 주소와 물리적인 주소의 매핑 테이블을 메모리에 저장한다. 플래시 메모리의 크기가 커지면 매핑 테이블이 커지게 된다. 현재 삼성 전자에서 1 기가 플래시 메모리가 개발되었고, 상용으로 512M, 128M 플래시 스마트 카드가 시판되고 있는 시점이다. 예를 들어 512 MByte의 플래시를 사용할 경우 매핑 테이블만 4 MByte에 달하게 된다. 인텔사의 FTL[1]은 이러한 문제점을 해결하기 위해서 replacement page와 virtual map page를 사용한다. 그러나, 매핑 테이블이 바뀌면 성능이 크게 떨어질 수 있는 단점이 있다. 우리는 대용량 플래시 메모리를 이용할 경우 효율적인 플래시 메모리 관리자의 설계를 추후 과제로 삼고 있다.

참 조 문 헌

[1] Kirk Blum, "Software Concerns of Implementing a Resident Flash Disk," Inter corporation, 1995.
[2] "Flash memory," Intel corporation, 1994.
[3] "3 volt FlashFile Memory 28F160S3 and 28F320S3," Intel corporation, 1998.
[4] Uresh Vahalia, "UNIX Internals : the new frontiers." pp. 220-288, Prentice-Hall, 1996.
[5] Michael Tischer and Bruno Jennrich, "PC INTERN : the Encyclopedia of System Program-

ming," pp.377-387, Abacus, 1996.
[6] Steven Holzner and The Peter Norton Computing Group, "Advanced Assembly Language," pp.214-225, 영진출판사, 1991.
[7] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," ACM Transactions on Computer Systems, Vol. 10, pp.26-52, 1992.
[8] Atsuo Kawaguchi, Shingo Nishioka and Hiroshi Motoda, "A Flash-Memory Based File System," Proceedings of 1995 USENIX Techinical Conference, pp.155-164, 1995.
[9] M. Wu and W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Storage System," Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, pp.86-97, 1994.
[10] Chiang, M.-L, Lee, P. C. H and Chang, R.-C, "Managing Flash Memory in Personal Communication Devices," Proceedings of the IEEE International Symposium on Consumer Electronics, pp. 177-182, 1997.
[11] Gregory R. Ganger and Yale N. Patt, "Metadata Update Performance in File Systems," USENIX Symposium on Operating Systems Design and Implementation, pp.49-60, 1994.
[12] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein, "Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems," Proceedings of 2000 USENIX Technical Conference, pp. , 2000.
[13] Kleiman, S.R., "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," Proceedings of the Summer 1986 USENIX Technical Conference, 1986.

박 상 호
1997년 한국과학기술원 전기 및 전자공학과 학사 졸업. 1999년 한국과학기술원 전기 및 전자공학과 석사 졸업. 1999년 ~ 현재 한국과학기술원 전기 및 전자공학과 박사 과정 재학. 관심 분야는 파일 시스템, RTOS, Ad-hoc network, Multicast Routing Protocol

안 우 현
1996년 2월 경북대학교 전자공학과 공학사. 1998년 2월 한국과학기술원 전기 및 전자공학과 공학석사. 1998년 3월 ~ 현재 한국과학기술원 전기 및 전자공학과 박사과정. 관심분야는 컴퓨터구조, 병렬처리 및 운영체제

박 대 연
1989년 12월 University of Oregon 전산학과 학사. 1991년 6월 University of Oregon 전산학과 석사. 1996년 5월 University of Southern California 전산학과 박사학위 취득. 1996년 9월 ~ 1998년 1월 한국외국어대학 제어계측공학과 조교수. 1998년 2월 ~ 현재 한국과학기술원 전기 및 전자공학과 조교수. 관심분야는 컴퓨터구조, 병렬처리, 운영체제 및 분산처리

김 정 기
1992년 전북대학교 컴퓨터공학과 학사 졸업. 1994년 전북대학교 컴퓨터공학과 석사 졸업. 1999년 전북대학교 컴퓨터공학과 박사 졸업. 1996년 시스템공학연구소 연구원. 1998년 ~ 현재 한국전자통신연구원 선임연구원. 관심분야는 병렬 정보검색, 파일시스템, RTOS

박 승 민
1981년 울산대학교 전자공학과 학사 졸업. 1983년 홍익대학교 대학원 전자공학과 석사 졸업. 1998년 3월 ~ 현재 충남대학교 대학원 전자공학과 박사과정. 1983년 ~ 1984년 9월 (주)LG전자. 1984년 9월 ~ 현재 한국전자통신연구원 인터넷정보가전연구부 실시간미들웨어연구팀장/책임연구원. 관심분야는 인터넷 정보가전, RTOS