# 디지털 컨버전스 기기를 위한 지역 버퍼 캐쉬 파일 시스템 설계 및 구현

## Design and Implementation of File System Using Local Buffer Cache for Digital Convergence Devices

**정근재, 조문행, 이철훈**
충남대학교 컴퓨터공학과

Geun-Jae Jeong(gjjeong@cnu.ac.kr), Moon-Haeng Cho(root4567@cnu.ac.kr),
Cheol-Hoon Lee(clee@cnu.ac.kr)

**요약**

오늘날 내장형 장치의 보급 증가와 반도체 및 저장장치 기술의 발달로 인해 디지털 컨버전스 기기들이 늘어나고 있다. 디지털 컨버전스 기기는 PMP, PDA, 스마트폰 등과 같이 하나의 디지털 기기에 통신, 동영상과 음악 재생, 전자 수첩, 전자 사전 등의 기능이 집적된 장치를 말한다. 따라서 이러한 장치에는 여러 기능을 관리하고 제어할 수 있는 효율적인 파일 시스템의 설계가 필요하다. 파일 시스템을 설계하는데 있어서는 소형화된 내장형 장치에 맞는 사이즈, 성능, 호환성 등을 고려해야 한다. 본 논문에서는 부분 버퍼 캐쉬 기법을 제안한다. 기존의 버퍼 캐쉬에 비해, 부분 버퍼 캐쉬는 FAT 데이터와 쓰기 전용 데이터를 저장한다. 시뮬레이션을 통해 파일 크기가 100KByte가 넘는 경우, 부분 버퍼 캐쉬를 사용함으로써 쓰기 성능을 30% 이상 증가시킬 수 있음을 보인다.

■ 중심어 : | FAT 파일시스템 | 부분 버퍼 캐쉬 | 디지털 컨버전스 기기 |

**Abstract**

Due to the growth of embedded systems and the development of semi-conductor and storage devices, digital convergence devises is ever growing. Digital convergence devices are equipments into which various functions such as communication, playing movies and wave files and electronic dictionarys are integrated. Example are portable multimedia players(PMPs), personal digital assistants(PDAs), and smart phones. Therefore, these devices need an efficient file system which manages and controls various types of files. In designing such file systems, the size constraint for small embedded systems as well as performance and compatibility should be taken into account. In this paper, we suggest the partial buffer cache technique. Contrary to the traditional buffer cache, the partial buffer cache is used for only the FAT meta data and write-only data. Simulation results show that we could enhance the write performance more than 30% when the file size is larger than about 100 KBytes.

■ keyword : | FAT File System | Partial Buffer Cache | Digital Convergence Devices |

* 본 연구는 정보통신부의 선도기반기술개발사업의 지원으로 수행되었습니다.

접수번호 : #070516-002
접수일자 : 2007년 05월 16일
심사완료일 : 2007년 06월 13일
교신저자 : 이철훈, e-mail : clee@cnu.ac.kr

## I. 서 론

디지털 컨버전스는 디지털 기술이 발전함에 따라 유선과 무선, 방송과 통신, 통신과 컴퓨터 등 기존의 기술·산업·서비스·네트워크의 구분이 모호해지면서 이들 간에 새로운 형태의 융합 상품과 서비스들이 등장하는 현상을 포괄적으로 일컫고, 하드웨어적 의미로는 여러 가지 기능을 하나의 기기에 구현한 다기능 디지털 기기를 말한다[1].

PMP, PDA 등 다양한 디지털 컨버전스 기기가 나오고 있으며, 이러한 내장형 기기들이 소형화됨에 따라 실시간 운영체제를 사용하게 되었고[2], 저장장치로는 크고 무거운 하드디스크 타입이 아닌 작고 가벼운 SD/MMC Card, Memory Stick등을 많이 사용하게 되었다[3].

또한 내장형 장치들의 기능이 다양화 되면서 동영상 재생과 촬영의 지원은 기본 기능이 되었다. 동영상 촬영은 영상을 바로 저장장치에 저장해야 하기 때문에 파일 시스템의 성능이 중요하다[4]. 따라서 이러한 응용프로그램이 오버헤드 없이 효율적으로 동작하도록 파일 시스템을 설계 및 구현하는 것이 필요하다.

본 논문에서는 실시간 운영체제인 UbiFOS™을 기반으로한 디지털 컨버전스 기기에서 파일 시스템의 성능 향상 기법을 설계 및 구현하였다. 파일 시스템의 경량화 를 위해 플래그를 사용하여 메모리의 사용을 최소화 하였고, 성능 향상을 위해 기존의 버퍼 캐시를 수정하여 부분적으로 버퍼 캐시를 사용하는 기법과 기존의 파일 시스템 API를 수정하였다. 각각의 기법을 사용하였을 때 파일 기록 크기를 다르게 하여 성능을 측정하였다.

본 논문의 구성은 2장에서 실시간 운영체제 및 파일 시스템의 전체적인 구성을 기술하고, 3장에서는 파일 시스템 성능 향상을 위한 설계 및 구현 기법에 대해 다룬다. 4장에서는 실험 환경 및 결과를 기술하고, 마지막으로 5장에서는 결론 및 향후 연구 과제에 대해서 기술한다.

## II. 관련연구

### 1. 실시간 운영체제 UbiFOS™

실시간 운영체제인 UbiFOS™은 우선순위 기반의 선점형 스케줄러를 제공하여 태스크의 우선순위를 0부터 255까지 256단계로 구분하고 있다. 그리고 실행 준비된 태스크들 중에서 정해진 시간 내에 가장 높은 우선순위의 태스크를 찾기 위해 별도의 테이블과 리스트를 관리하며, 같은 우선순위의 태스크 사이에서는 선입선출(FIFO)과 라운드-로빈(Round-Robin)을 지원한다. [그림 1]은 실시간 운영체제 UbiFOS™의 전체 구성을 나타낸 것이다[5].

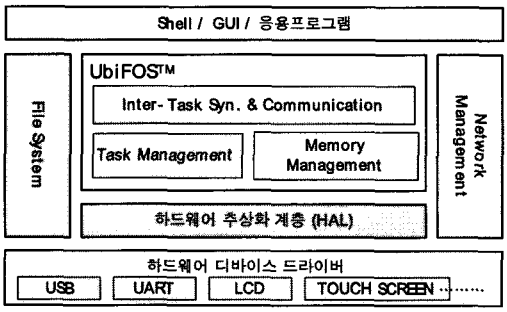



그림 1. 실시간 운영체제 UbiFOS™의 전체 구성도

### 2. 파일 시스템

데이터를 기록하기 위해서는 미리 저장장치에 데이터를 읽고, 쓰고, 찾기 위한 준비를 해두어야 한다. 파일 시스템은 그 준비의 규칙을 정리해 놓은 것으로써 파일에 이름을 붙이고, 저장이나 검색을 위해 파일을 어디에 위치시킬 것인지를 나타내는 체계이다.

UbiFOS™에서는 가상 파일 시스템 하에서 FAT 파일 시스템을 사용하고 있다.

#### 2.1 FAT 파일 시스템

FAT 파일 시스템은 마이크로소프트(MicroSoft)사의 MS-DOS에 의해 지원된 개인용 컴퓨터 파일 시스템이고, 클러스터(cluster)의 크기를 증가시키면 지원할 수 있는 저장장치의 크기도 증가하게 된다. FAT 파일 시스템은 윈도우즈 및 리눅스에서의 호환성을 위해 각종

미디어 카드(MMC, SD, Memory Stick)등에 가장 널리 사용되는 파일 시스템으로 각광 받고 있다. FAT 파일 시스템은 플로피 디스크를 위한 FAT12와 비교적 적은 용량 관리에 대한 FAT16, 그보다 큰 용량 관리를 위한 FAT32가 있다[6-8].

### 2.1.1 FAT 파일 시스템의 자료구조

FAT 파일 시스템은 마스터 부트 레코드(Master Boot Record), 파티션 부트 레코드(Partition Boot Record), 파일 할당 테이블 엔트리(File Allocation Table Entry), 디렉토리 엔트리(Directory Entry), 그리고 데이터 영역으로 구성되어 있다. [그림 2]는 FAT로 포맷한 32MB 메모리 스틱의 물리적인 데이터 저장 구조이다.

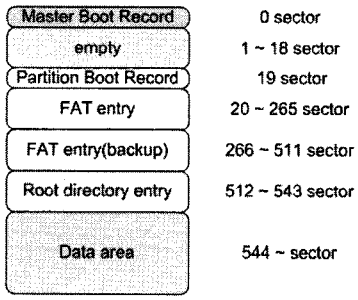



**그림 2. FAT 파일 시스템의 자료구조**

#### 2.1.1.1 마스터 부트 레코드

마스터 부트 레코드는 미디어의 용량, 파티션 부트 레코드의 시작 위치, 제조 회사 정보 등을 저장하며 1개의 섹터만 사용한다.

#### 2.1.1.2 파티션 부트 레코드

파티션 부트 레코드는 미디어에 대한 부가 정보를 저장하는데, 부트 섹터의 데이터는 저장장치를 포맷하는 시점에서 결정되고, 파일 시스템을 초기화하는 과정에서 수퍼 블록을 작성하고 루트 디렉토리를 생성할 때 사용된다.

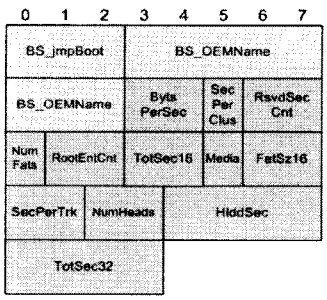



**그림 3. 부트섹터 및 BIOS Parameter Block 자료구조**

[그림 3]은 FAT12/16/32에서 공통적으로 사용되는 부트섹터의 데이터 구조이다.

#### 2.1.1.3 FAT(파일 할당 테이블)

FAT를 사용하는 미디어에 저장된 파일은 클러스터 단위로 분리되어 저장된다. 클러스터는 1~32개의 섹터로 구성되며 미디어를 포맷하는 과정에서 정해지고, FAT는 다음 클러스터에 대한 정보를 저장하고 있다.

FAT 디렉토리 엔트리는 파일에 할당된 첫 번째 클러스터 번호를 저장하고, 그 다음 클러스터들은 파일 할당 테이블을 통해서 알아낼 수 있다. 즉, 파일 할당 테이블은 하나의 파일이 할당 받은 클러스터의 범위를 단일 연결 리스크로 정의한 것이다.

파일 할당 테이블은 기본적으로 0x00으로 초기화되어 있다. 데이터 영역에 파일이 저장되면 해당하는 파일 할당 테이블에 다음 클러스터의 위치를 저장하게 된다. 해당 클러스터가 파일의 끝이면 0xFFF8(또는 0xFFFF)가 저장된다.

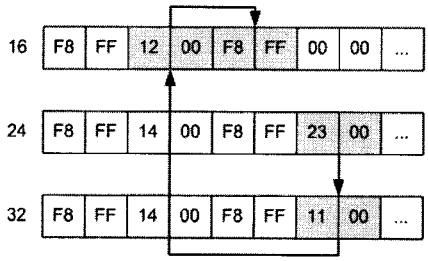



**그림 4. 파일 할당 테이블**

[그림 4]는 FAT16을 사용하는 미디어에 2KB 크기를 갖는 파일이 27, 35, 17, 18번째 클러스터에 분산되어 저장되어 있는 경우의 파일 할당 테이블 내용이다. FAT 파일 시스템의 디렉토리 엔트리에는 시작 클러스터인 27(0x1B)를 저장하고 다음 클러스터에 대한 정보는 그림 4와 같이 파일 할당 테이블에 저장된다.

## 3. 플래시 변환계층 (Flash Translation Layer, FTL)

FTL은 플래시메모리의 삭제연산을 감추기 위한 미들웨어로 파일 시스템과 플래시메모리 사이에 위치한다. 삭제연산은 쓰기연산 시에 파일 시스템이 생성한 논리주소를 플래시메모리상의 이미 삭제연산을 수행한 영역에 대한 물리주소로 변환함으로써 감춰진다. 삭제연산을 감추고 I/O를 하나의 단위로 처리해 하드디스크와 같은 단일 저장장치를 구성함으로써, 상단에서 일반 파일 시스템을 사용해 플래시메모리를 제어할 수 있다[9]. FTL은 크게 호스트시스템에서 독립된 하드웨어 형태[10-12]와 호스트시스템 내부의 디바이스드라이버 형태[13]로 구현할 수 있다.

FTL은 주소변환 단위에 따라 크게 페이지(쓰기)단위 주소변환과 블록(삭제)단위 주소변환으로 나뉜다. [그림 5(a)]의 페이지단위 주소변환은 정교하게 주소를 변환하기 때문에 성능은 좋은데 반하여 주소변환 테이블의 크기가 커 제작비용이 높다. 반대로 [그림 5(b)]의 블록단위 주소변환은 정교하게 주소를 변환하기 때문에 주소변환 테이블의 크기는 작지만, 내부의 하나의 페이지에 대한 수정연산이 발생해도 전체 블록을 삭제하고 갱신해야 하는 추가비용이 있다. 뿐만 아니라 이 과정을 수행하는 도중에 폴트가 발행하면 데이터의 일관성을 깨뜨릴 수 있다.

이러한 블록단위 주소변환 방식의 단점을 개선한 [그림 5(c)]의 교체블록 기법은 블록 내부의 페이지에 대한 갱신요청이 발생하면 교체블록을 할당해 쓰기를 수행하고 이를 연결리스트로 구성해 향후 읽기연산 시 이 리스트를 역순으로 검색해 최종적으로 수정된 데이터를 제공하는 방법이다.

마지막으로 [그림 5(d)]의 로그블록 기법은 페이지단위 주소변환과 블록단위 주소변환 기법을 병합한 형태로 비교적 큰 단위로 요청되는 순차 입출력은 블록단위로 처리하고, 작은 단위로 요청되는 임의 입출력은 페이지 단위로 로그구조 파일 시스템(LFS)[14][15]과 유사하게 로그형태로 저장하는 방식이다[12]. 이를 통해 주소변환 테이블의 크기를 줄이고도 높은 성능을 발휘할 수 있다.

하지만 FTL을 사용하여도 호스트시스템에서는 FAT와 같은 일반적인 자기디스크용 파일 시스템을 사용해야 하며, 하나의 작업을 두 개의 층을 두어 처리하는 구조가 된다. 따라서 NAND형 플래시메모리를 보조기억장치로 사용하는 경우에는 FTL을 활용해 저장장치를 설계하는 것이 플래시메모리 전용 파일시스템만을 사용해 설계하는 것에 비해 효율적일 수 있다[9].

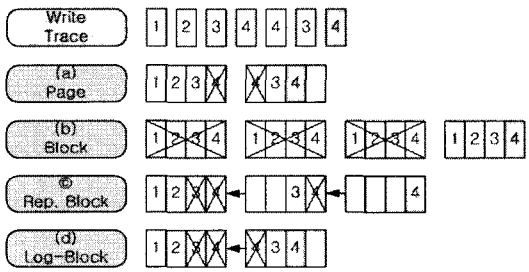



**그림 5. FTL의 설계기법에 따른 작동예제**

## 4. 플래시메모리 전용 파일 시스템

파일 시스템에서 효율적으로 플래시메모리를 제어하기 위해서 JFFS(Journaling Flash File System)와 FFS(Flash File System)와 같은 플래시메모리 전용 파일 시스템이 개발되었다[16][17]. JFFS는 GNU GPL[18] 아래 공포되어 주로 리눅스 기반 기기에서 사용하며, 비교적 간단한 구조의 FFS는 MS-DOS에서 사용한다.

JFFS[16][17]는 쓰기의 경우에는 데이터를 로그형태로 순차적으로 기록하고 읽기의 경우에는 로그를 역순으로 검색해 가장 최신의 데이터를 읽는다. 쓰기연산 이후에 임계용량 이상을 사용하게 되면 앞부분 블록부터 차례로 삭제연산을 수행해 빈 공간을 확보한다. 이때 삭제 블록내부에 존재하는 유효한 페이지는 다시 로

그형태로 기록해 보존한다.

하지만 이와 같이 삭제연산을 순차적으로 수행하는 것은 유효한 데이터의 비율 등을 고려해 최적의 블록을 선택적으로 삭제하는 방법에 비하여 비효율적이다. 이를 개선하기 위해 JFFS2가 개발되었다. 삭제연산을 효율적으로 수행하는 것 외에도 JFFS2에서는 단위 공간당 단가가 높은 플래시메모리의 공간을 효율적으로 활용하기 위해 압축 기능을 사용하고[19], 보다 다양한 종류의 inode구조를 지원해 몇 가지 장점을 취한다.

플래시메모리 전용 파일시스템과 FTL을 설계하는데 있어서 너무 일찍 삭제연산을 수행하면 삭제 블록내부에 유효한 페이지의 수가 적어 총 수행하는 삭제연산의 수가 늘어나게 된다. 반대로 삭제연산 수행시점을 최대한 지연시키면 공간 효율성 측면에서는 좋지만 이후에 쓰기연산이 요청되고, 기 삭제된 블록이 없는 경우에 그 동안 지연시킨 삭제연산을 한번에 수행해야 하기 때문에 입출력 성능을 크게 저하시킬 수 있다[10]. 플래시메모리 중에는 내부를 뱅크형태로 분할하고 서로 다른 뱅크에 대해 삭제와 쓰기 또는 쓰기와 읽기연산을 동시에 수행할 수 있는 기능을 지원하는데, 삭제연산을 지연해 특정시점에 동시에 수행하게 되면 이러한 구조적 병렬성을 활용하기 어렵기 때문이다[9].

## III. 디지털 컨버전스 기기를 위한 파일 시스템 설계 및 구현

오늘날 디지털 컨버전스 기기는 소형화되고, 다양한 기능들이 하나의 디지털 기기에 구현된다. 또한 저장장치 기술의 발달로 소형 및 경량의 저장장치가 가능하게 되었다. 따라서 응용프로그램을 효율적으로 동작시키기 위한 파일 시스템 설계가 필요하고, 이를 위한 고려사항은 다음과 같다.

- 경량화 : 제한된 크기의 메모리를 갖는 내장형 시스템에 탑재할 파일 시스템의 풋 프린트를 최소화해야 한다.
- 성능 : 디지털 컨버전스 기기의 다양한 기능들이 오버 헤드 없이 동작 하도록 향상된 성능이 요구된다.
- 호환성 : 디지털 컨버전스 기기의 호스트로 많이 사용되는 윈도우즈, 리눅스와의 호환성을 위해 FAT 파일 시스템을 기본으로 한다.

### 1. 경량화를 위한 기법 구현

디지털 컨버전스 기기는 메모리의 제한이 있는 소형화된 내장형 장치이기 때문에 파일 시스템의 크기를 고려해야 한다. 파일 시스템의 크기가 커지게 되면 메모리 사용량이 커지고, 그만큼 응용프로그램에게 할당할 수 있는 메모리의 크기가 줄어들기 때문이다.

본 논문에서는 파일 시스템의 경량화를 위한 기법으로 플래그를 사용하였다. 파일 시스템 함수 중 비슷한 기능을 수행하는 함수를 하나의 함수로 묶어 놓고 플래그를 설정하여, 동일한 함수 내에서 플래그의 셋팅에 따라 다른 동작을 수행하도록 설계 및 구현하였다. 이로 인해 다수의 함수들을 줄일 수 있었고, 결과적으로 코드 사이즈의 경량화를 가져왔다. 파일 시스템에서 사용되어지는 플래그는 [표 1]과 같다.

**표 1. 경량화를 위한 플래그**

| Flag | 기 능 |
|---|---|
| OpenFlag | 파일의 Open 여부 |
| RemFlag | 파일의 Remove 여부 |
| RenFlag | 파일의 Rename 여부 |

OpenFlag의 경우, 디렉토리 엔트리 영역에서 디렉토리 엔트리를 조합하는 MF_FatVnodeReaddirEntry() 함수에서 사용되어 진다. 파일을 생성할 경우는 위의 함수를 호출하여 비어있는 디렉토리 엔트리 영역을 반환하는데 사용하고, 파일을 여는 경우는 위의 함수를 호출하여 열고자 하는 파일 이름이 디렉토리 엔트리 영역에 존재하는지 확인하고 해당 엔트리 영역을 반환하는데 사용한다.

보통 FAT 파일 시스템에서는 파일을 열면 해당 파일이 디렉토리 엔트리에 존재하는지 여부만 확인하지만, UbiFOS™ 파일 시스템은 디렉토리 엔트리에 기록될 위치까지 계산하도록 설계 및 구현하였다.

RemFlag는 파일을 열여 삭제하고자 할 때 해당 파일명으로 저장되어져 있는 디렉토리 엔트리를 위의 함수를 통해 찾은 후 디렉토리 엔트리를 삭제하는데 사용한다.

RenFlag는 파일을 열어 파일명을 변경하고자 할 경우 위 함수를 통해 변경하고자 하는 파일의 디렉토리 엔트리를 찾은 후 엔트리를 삭제하는데 사용한다. 이와 같이 플래그를 사용하지 않을 경우 각각의 역할을 하는 함수를 각각 사용해야 하나, 플래그를 사용함으로써 하나의 함수를 플래그 셋팅값만 가지고 여러 가지 기능으로 사용할 수 있다.

## 2. 성능 향상을 위한 기법 구현

성능 향상을 위해서 기존의 버퍼 캐시가 가진 문제점인 메모리 사용량이 많은 점과 버퍼의 내용을 저장장치에 기록 시 발생하는 오버헤드 문제를 수정하여, 부분적으로만 버퍼를 사용하는 기법과 열린 파일의 포인터를 변경하는 기존의 lseek()함수를 수정하는 기법으로 크게 나누어서 설계 및 구현하였다.

### 2.1 부분적인 버퍼 캐시 사용

보조 저장장치는 전자적인 특성과 기계적인 특성을 동시에 가지고 있기 때문에 메모리를 읽고 쓰는 것에 비해 상당히 속도가 느리다. 두 계층 사이의 현저한 액세스 속도차로 말미암아 발생하는 비효율적인 문제를 개선하기 위해 캐시를 사용한다. 캐시에서 하는 일은 두 가지 이다. 하나는 이미 읽어 들인 데이터를 다시 읽어 들이지 않도록 하는 것이고, 다른 하나는 데이터가 기록될 때 매번 저장장치에 데이터를 기록하지 않고 모아두었다가 한꺼번에 기록하는 것이다[20].

접근 속도를 개선하기 위해서 캐시를 사용하지만, 버퍼 캐시를 사용함으로써 메모리 사용량이 많아진다는 단점이 있다. 그리고 버퍼 캐시를 언제 저장장치에 기록할 것인지를 결정해야 한다[20].

UbiFOS™의 파일 시스템에서는 버퍼 캐시를 위해 524KBytes의 메모리를 사용하고, 빈 버퍼의 개수가 5개 미만 남았거나 사용자의 명령이 있을 경우, DIRTY 플래그가 셋팅 된 캐시를 모두 저장장치에 기록하고 해당 버퍼를 Free 시킨다. 이 경우 버퍼 캐시를 모두 기록하는 과정에서 오버헤드가 발생하게 된다. 따라서 이러한 오버헤드를 줄이고 성능 향상을 위한 기법으로 파일 시스템 버퍼 캐시를 사용하지 않고 저장장치와의 접근이 자주 일어나는 부분에만 버퍼를 두어 관리하는 기법을 설계 및 구현하였다.

빈 클러스터를 얻을 때 저장장치의 FAT영역에 자주 접근하는 것을 막기 위해 한번 접근된 블록부터 8블록(4096Bytes)을 버퍼로 관리하여 저장장치의 잦은 접근을 방지하였다.

[그림 6]은 FAT영역을 버퍼로 관리하기 위한 구조체를 나타낸 것이다.

```
typedef struct FATTag {
        int         FatIndex;
        // 블록 인덱스
        char        FatTable[4096];
        // FAT영역을 저장하고 있는 버퍼
        int         start;
        // 시스템의 처음 시작 여부
}FatStt;
```

**그림 6. FAT 버퍼 구조체**

FAT 버퍼에서 빈 클러스터를 찾지 못할 경우, 버퍼를 디바이스의 FAT 영역에 기록한 후 다음 FAT 영역을 버퍼로 읽어 들여 관리한다. FAT 버퍼를 사용함으로써 빈 클러스터를 찾기 위해 저장장치의 FAT 영역에 대한 빈번한 접근을 줄일 수 있게 되었다. 버퍼 캐시를 사용할 경우, 버퍼에 512Bytes(한 블록)의 FAT 영역을 저장하고 있지만, 크기가 512Bytes인 버퍼가 기록할 수 있는 클러스터의 개수는 FAT32일 경우 하나의 클러스터를 나타내는데 4Bytes가 필요하므로 512 / 4 = 128개가 된다. 즉, 하나의 클러스터 크기가 4KBytes이므로 128 * 4KBytes = 512KBytes 만큼의 데이터를 512Bytes 버퍼로 관리할 수 있다.

본 논문에서는 성능 향상을 위해 FAT 영역 버퍼를 4096Bytes로 확장하여 이 버퍼에서 관리할 수 있는 데이터의 크기를 512KBytes * 8 = 4MBytes로 확장하였다.

또한 파일을 기록할 경우 크기가 블록 크기(512Bytes)보다 크면 여러 번 저장장치에 접근하여 기록이 된다. 이 경우에도 [그림 7]처럼 4KBytes의 부분적인 버퍼를 사용하여 저장장치에 접근하는 오버헤드

를 줄였다.

```
char        WriteBuffer[4096]
```

그림 7. Write 부분 버퍼

4KBytes 크기의 Write전용 버퍼를 생성하여 한 파일에 쓰려고 하는 사이즈가 4KBytes 이하일 경우 버퍼에 계속 쌓아두고 마지막 위치로 포인터만 변경 시켜 준다. WriteBuffer가 다 찼을 경우 저장장치에 접근하여 버퍼의 내용을 기록하고, 파일이 Close될 경우 WriteBuffer에 기록이 되지 않은 내용이 있을 때는 버퍼를 기록하고 파일을 Close하게 된다.

위와 같이 부분 버퍼를 사용할 경우 FAT버퍼와 Write버퍼의 8KBytes만 필요로 하게 되어, 기존 버퍼 캐시의 524KBytes보다 메모리 사용량을 줄일 수 있다.

### 2.2 lseek() 함수 수정

기존의 unsigned long lseek( int nHandle, unsigned long lOffset, int nOrigin ) 함수는 열린 파일에 대해서 인자로 주어진 lOffset으로 파일 포인터가 이동하는데, nOrigin인자에 따라 이동하는 기준점이 다르게 동작한다. [표 2]는 nOrigin 따른 기준점을 나타내고 있다.

표 2. 파일 포인터 기준점

| nOrigin | 설 명 |
|---|---|
| FSEEK_SET | 포인터를 lOffset로 설정 |
| FSEEK_CUR | 포인터를 현재위치 + lOffset로 설정 |
| FSEEK_END | 포인터를 파일의 크기 + lOffset로 설정 |

파일의 포인터를 변경하는데 있어서 lOffset에 따라 클러스터 번호와 클러스터 옵셋을 설정하게 된다. 이때 해당 파일의 처음 시작 클러스터부터 FAT 테이블에 접근하여 다음 클러스터 번호를 얻어 와야 한다. lOffset이 클 경우, 해당 옵셋의 클러스터로 이동하기 위해서 FAT 테이블에 접근하는 횟수가 많아지게 된다.

본 논문에서는 이러한 FAT 테이블에 대한 접근을 줄이기 위한 방법으로 클러스터 번호와 인접해 있는 클러스터의 개수를 저장하는 구조체를 생성하여 접근 오버헤드를 줄일 수 있게 설계 및 구현하였다.

```
typedef struct ClusterLogTag {
        int                 cnt;
        // 인접해 있는 클러스터 개수
        unsigned long       cluster;
        // 클러스터 번호
}ClusterLogStt;
ClusterLogStt       ClusterLog[100];
```

그림 8. 클러스터 로그 구조체

[그림 8]은 열린 파일에 대한 클러스터 정보를 저장하고 있는 구조체로 파일의 시작 클러스터부터 클러스터 번호와 다음 클러스터가 인접해 있다면 인접한 클러스터의 개수를 저장하는 변수로 이루어져 있다.

파일을 열 경우 파일의 마지막 클러스터와 파일 크기를 구하게 된다. 이때 파일 크기를 구하면서 시작 클러스터부터 시작해 다음 클러스터 번호를 얻어올 때 클러스터 로그 변수에 해당 내용을 먼저 기록하게 된다.

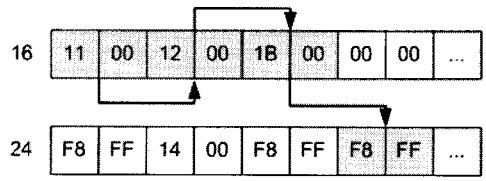



그림 9. 클러스터 체인

[그림 9]는 하나의 파일이 클러스터 번호 16, 17, 18, 27에 걸쳐서 쓰여 있을 경우 FAT 테이블을 나타내고 있다. 이 경우 파일을 오픈할 때 ClusterLog 변수에 다음 [표 3]과 같이 값이 저장된다.

표 3. 클러스터 로그 변수

| 구 분 | 개 수(cnt) | 클러스터 번호(cluster) |
|---|---|---|
| ClusterLog[0] | 2 | 16 |
| ClusterLog[1] | 0 | 27 |

[표 3]의 의미는 열린 파일이 클러스터 16에서 시작하여 연속된 2개의 클러스터를 가지며(17, 18), 그 다음 클러스터의 번호는 27번이고 마지막 클러스터를 나타낸다.

lseek() 함수를 호출할 경우 클러스터를 찾아 파일 포인터를 설정하게 되는데, 클러스터 정보를 가지고 있는 ClusterLog 변수를 사용하면 저장장치의 FAT 테이블에 접근하지 않아도 되므로 성능을 향상시킬 수 있다.

### 3. 호환성

대부분의 내장형 장치들이 윈도우즈와 연동하여 사용된다. 따라서 윈도우즈 FAT 파일 시스템과 호환성을 유지하기 위해 UbiFOS™ 파일 시스템을 FAT16/32로 구현하였다. 또한 내장형 장치에서 많이 사용되어지는 저장장치로 낸드 플래시가 있으나, 낸드 플래시에 적합하게 설계된 파일 시스템을 사용하지 않고 FTL(Flash Translation Layer)을 사용하여 윈도우즈와의 호환성을 유지하였다.

## Ⅳ. 실험 환경 및 결과

본 논문에서 구현한 파일 시스템은 실시간 운영체제 UbiFOS™을 대상으로, ARM920T 기반의 S3C2440 MCU가 탑재되고, 메모리가 32MBytes인 MBA2440 보드[5]에서 실험하였다. 저장장치로는 SanDisk의 1G Bytes SD Card를 사용하였다.

실험 방식은 SD Card에 사이즈가 10, 20, 40, 80, 100KBytes인 파일을 기록할 때 각각의 소요되는 시간을 텍트로닉스(Tektronix)의 TDS3054B 오실러 스코프를 사용하여 측정하였다. 버퍼 캐시를 사용할 경우와 부분 버퍼를 사용할 경우, 버퍼 캐시를 사용하지 않을 경우로 나누어 실험하였다.

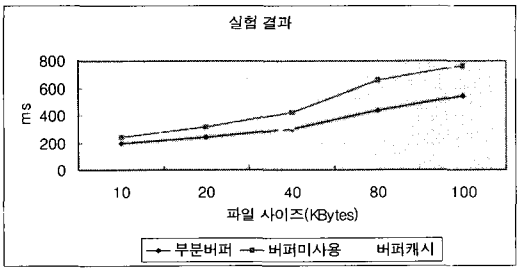



그림 10. Write 실험 결과

[그림 10]의 실험 결과를 보면 버퍼 캐시를 사용하지 않은 경우, 같은 크기의 파일을 기록할 때 소요되는 시간이 다른 두 경우에 비해 오래 걸렸다.

버퍼 캐시를 사용한 경우에는 파일 크기가 작으면 시간이 가장 적게 소요되었으나, 크기가 큰 파일을 기록할 시에는 메모리를 할당받는 시간과 일정량 이상의 버퍼가 찼을 경우 저장장치에 기록하는 오버헤드 등으로 인해 부분 버퍼를 사용했을 때보다 시간이 더 소요되었다.

부분 버퍼를 사용했을 경우는 파일 크기가 작을 시 버퍼 캐시를 사용했을 경우와 버퍼 캐시를 사용하지 않았을 경우의 중간 기록 시간 값을 확인할 수 있으나, 기록 파일의 크기가 클 시에는 가장 적은 시간이 소요되어 앞의 두 경우보다 성능이 향상됨을 확인할 수 있다.

[그림 11]은 lseek()함수를 수정해서 클러스터 로그를 사용했을 경우와 사용하지 않았을 경우의 실험 결과이다.

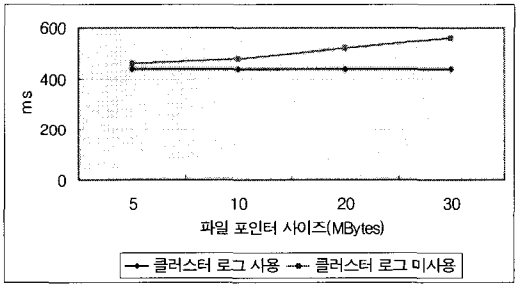



그림 11. lseek() 함수 수정 결과

[그림 11]의 X축은 열린 파일에서 lseek() 함수를 통해 이동할 파일 포인터의 크기이다. 실험은 35MBytes의 파일을 SD카드에 기록 후 100KBytes를 읽고 X축의 크기만큼 파일 포인터를 이동시킨 후 다시 100KBytes를 읽었을 시 소요되는 시간을 측정하였다.

클러스터 로그를 사용했을 경우엔 파일을 열 때 미리 클러스터정보를 유지하고 있기 때문에 파일 포인터를 이동시키는데 소요되는 시간이 일정하게 유지된다. 그러나 클러스터 로그를 사용하지 않았을 경우엔 lseek() 함수를 통해 파일 포인터를 이동 시킬 시, 저장장치로부터 클러스터 정보를 얻어 와야 하기 때문에 이동시키려는 파일 포인터의 크기가 클수록 소요되는 시간이 점차 증가한다.

위의 두 실험 결과를 통해 100KBytes의 파일 기록 시 버퍼 캐시를 사용할 경우 700ms가 소요되었고, 부분 버퍼를 사용했을 경우 540ms가 소요되었다. 이로 인해 100KBytes이상의 파일을 기록할 경우 30%이상의 속도 향상을 확인할 수 있다.

## V. 결 론

본 논문에서는 디지털 컨버전스 기기에서 파일 시스템의 경량화와 성능, 호환성을 향상시키는 기법을 설계 및 구현하였다.

버퍼 캐시를 사용했을 경우, 파일 크기가 작을 시 기록하는데 시간이 적게 소요되나 메모리 사용량이 많은 점과 버퍼의 내용을 저장장치에 기록할 때 오버헤드가 발생하였고, 버퍼 캐시를 사용하지 않을 경우 파일이 안정적으로 기록되나 잦은 저장장치 접근으로 인해 기록되는데 소요되는 시간이 오래 걸렸다. 그러나 부분 버퍼를 사용했을 경우, 위 두 가지의 장점을 이용하여 파일 기록 시 성능이 가장 좋게 나오는 것을 확인할 수 있었다. 또한 열린 파일의 클러스터 정보를 미리 유지함으로써 클러스터를 찾기 위한 저장장치 접근을 줄였고, 이로 인해 성능이 향상됨을 확인하였다.

크기가 작고 이동성이 편리한 디지털 컨버전스 기기의 특성상 메모리 크기에 한계가 있기 때문에 파일 시스템 경량화에 초점을 두어 파일 시스템이 차지하는 메모리의 사용량을 줄일 수 있도록 설계 및 구현하였다. 윈도우즈 운영체제에서는 대부분의 디지털 컨버전스 기기들을 이동식 디스크로 인식시켜 사용하기 때문에 윈도우즈와 호환성을 위해 FAT 파일 시스템을 사용하였다.

향후에는 버퍼를 통한 성능 향상이 아닌 저장장치에 접근하여 Read, Write 동작할 때 성능을 향상 시킬 수 있는 방안에 대한 연구가 진행되어야 하겠다.

## 참 고 문 헌


[1] http://www.irsin.net/archive/20060321

[2] D. Stepner, et. al, "Embedded Application Design Using a Real-Time OS," Design Automation Conference, pp.151-156, 1999.

[3] 박상호, 안우현, "플래시 메모리를 위한 파일 시스템의 구현", 한국정보과학회, 제7권, 제2호, pp.402-415, 2001(10).

[4] 장승주, "LZSS 압축 알고리즘을 적용한 PDA용 Embedded Linux 파일 시스템 설계", 한국정보처리학회, 제13권, 제2호, pp.95-100, 2006(4).

[5] http://www.aijisystem.com/

[6] http://www.walrus.com/~raphael/pdf/FatFormat.pdf

[7] 오재준, *OS 제작의 원리 그리고 Codes*, 가남사, 2004.

[8] 정준석, 정원용, *IT EXPERT 임베디드 개발자를 위한 파일 시스템의 원리와 실습*, 한빛미디어, 2006.

[9] 임근수, 고건, "플래시메모리 기반 저장장치의 설계기법", 한국정보과학회, 제30권, 제2호, pp.274-276, 2003.

[10] M. Wu and W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Stroage System," The International Conference on Architectural Support for Programming Languages and Operating Systems, pp.86-97, 1994.

[11] http://developer.intel.com

[12] J, Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A Space-Efficient Flash Translation Layer for Compact Flash Systems," IEEE Transactions on Consumer Electronics, Vol.48, No.2, pp.366-375, 2002.

[13] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash Memory Based File System," The USENIX 1995 Winter Technical Conference, pp.155-164, 1995.

[14] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," ACM Transactions on Computer

Systems, Vol.10, No.1, pp.26-52, 1992.

[15] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin, "An Implementation of a Log-Structured File System for UNIX," The USENIX Technical Conference, pp.307-326, 1993.

[16] http://www.linux-mtd.infradead.org

[17] http://developer.axis.com/software/jffs

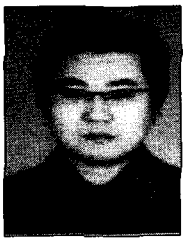
[18] http://www.gnu.org/licenses/gpl.html

[19] K. S. Yim, H. Bahn, and K. Koh, "A Compressed Page Management Scheme for NAND-type Flash Memory," The International Conference on VLSI, pp.266-271, 2003.

[20] D. P. Bovet and M. Cesati, *Understanding the LINUX KERNEL*, O'Reilly, Third Edition, 2005.

## 저 자 소 개

**정 근 재(Geun-Jae Jeong)** 준회원



- 2006년 2월 : 충남대학교 컴퓨터공학과(공학사)
- 2006년 3월 ~ 현재 : 충남대학교 컴퓨터 공학과 석사과정 재학

<관심분야> : 실시간 운영체제, 파일 시스템

**조 문 행(Moon-Haeng Cho)** 정회원

- 2004년 2월 : 충남대학교 컴퓨터공학과(공학사)
- 2006년 2월 : 충남대학교 컴퓨터공학과(공학석사)
- 2006년 3월 ~ 현재 : 충남대학교 컴퓨터공학과 박사과정 재학

<관심분야> : 실시간 컴퓨팅, 실시간 운영체제, 초소형 초절전 실시간 운영체제

**이 철 훈(Cheol-Hoon Lee)** 정회원

- 1983년 2월 : 서울대학교 전자공학과(공학사)
- 1988년 2월 : 한국과학기술원 전기및전자공학과(공학석사)
- 1992년 2월 : 한국과학기술원 전기및전자공학과 (공학박사)
- 1983년 3월 ~ 1986년 2월 : 삼성전자 컴퓨터사업부 연구원
- 1992년 3월 ~ 1994년 2월 : 삼성전자 컴퓨터사업부 선임연구원
- 1994년 2월 ~ 1995년 2월 : Univ. of Michigan 객원 연구원
- 1995년 2월 ~ 현재 : 충남대학교 컴퓨터공학과 교수
- 2004년 2월 ~ 2005년 2월 : Univ. of Michigan 초빙 연구원

<관심분야> : 실시간시스템, 운영체제, 고장허용 컴퓨팅