파일수준의 캐시기능을 통한 플래시 파일 시스템의 성능 향상 기법

이준희°, 서민열, 맹지찬, 유민수
한양대학교 실시간시스템 연구실
{jhlee, myseo, jcmaeng, msryu}@rtcc.hanyang.ac.kr

A Performance Improvement technique for Flash-based File System Using File-Unit Caching

Junhee Lee°, Min-Yeol Seo, Ji Chan Maeng, and Minsoo Ryu
Real-Time Computing and Communications Lab, Hanyang University

요 약

비휘발성 메모리의 하나인 플래시 메모리는 저전력 및 저비용 등의 장점으로 인해 임베디드 시스템에 필수적인 요소로 사용되고 있다. 이러한 장점에 반해, DRAM과 같은 휘발성 반도체 메모리와 비교하여 데이터 쓰기는 느리고, 데이터 업데이트를 위한 블록 소거 (erase) 동작의 오버헤드라는 단점이 있다. 특히 블록 소거 동작은 횟수가 제한되어 있으며, 이는 플래시 메모리의 수명을 결정하는데 중요한 요소이다. 본 논문에서는 플래시 메모리 기반 파일시스템에서 DRAM과 같은 동적 메모리를 사용하여 블록 소거의 횟수를 줄이고 입출력 속도를 향상시키는 기법을 제안한다.

1. 서 론

비휘발성 메모리의 하나인 Nand 플래시 메모리는 저전력, 저비용 및 고집적 등의 장점들로 인해 휴대용 단말기 및 임베디드 시스템 등의 저장매체로 폭넓게 사용되고 있다.

플래시 메모리는 기본적으로 그 내부구조에 따라 NOR와 NAND의 두 가지로 분류된다. NOR는 바이트 단위의 주소 지정이 가능하여 코드를 실행하거나 랜덤액세스가 가능하다는 장점이 있으나 제조단가가 높고 집적도가 낮다는 단점이 있다. 이와 반대로 NAND는 페이지 단위의 주소 지정만이 가능하므로 코드 실행 및 랜덤액세스는 불가능하나 제조단가가 낮고 집적도가 높다는 장점이 있다. 근래에는 저장매체로서의 이점을 가진 NAND의 성능 향상을 위한 연구가 활발히 진행되어왔으며, 산업계에서는 NAND상에 FAT 파일시스템을 구현하여 사용하는 것이 실질적인 표준처럼 사용되고 있다.

FAT 파일시스템은 원래 플로피 디스크와 같은 원형 자기디스크를 위한 파일시스템으로 플래시의 구조적 특성인 쓰기 전 소거 (erase-before-write)와 같은 기능은 전혀 반영되어있지 않다. 이렇게 기존의 자기디스크를 위한 파일시스템을 사용하기 위한 연구 결과의 대표적인 것이 FTL (Flash Translation Layer)이며 이를 통해 FAT와 같은 기존의 파일시스템을 플래시 메모리상에서 구현하여 사용하는 것이 가능하다.

앞서 언급한 바와 같이, 플래시 메모리에 데이터를 쓰기 위해서는 우선 해당 블록을 지우는 작업이 선행되어야 하며, 실제 쓰기 작업보다 수행시간이 긴 삭제 작업으로 인해 다른 메모리 소자에 비해 쓰기 속도가 매우 느리다는 단점이 존재한다. 또한 각 블록 당 삭제 횟수는 10만회 정도로 제한되어 있으므로 입출력이 많아지면 그만큼 수명도 줄어들게 된다. FTL은 이러한 물리적인 연산들을 감추어서 어느 정도 쓰기속도를 개선하고 삭제연산의 횟수를 줄여주는 이점을 제공하나 여전히 쓰기속도의 개선이 필요하다.

본 연구에서는 이러한 단점들을 개선하기 위해 사용가능한 성능 향상 기법을 제안한다. 우리는 FAT 파일시스템을 통해 플래시의 데이터를 읽고 쓸 때, 입출력 속도가 빠른 동적 메모리를 이용하여 사용 중인 데이터를 저장함으로써 플래시 메모리로의 입출력을 줄이고자 하였다. 이러한 입출력 과정은 FTL이 아닌 FAT 파일시스템 레벨에 포함되었으며, 파일시스템 API를 통해 개발자의 의도에 따라 선택할 수 있도록 하였다.

본 논문은 다음과 같이 구성되어 있다. 2장은 관련연구 및 배경에 대해 설명하고, 3장은 개발 환경에 대하여

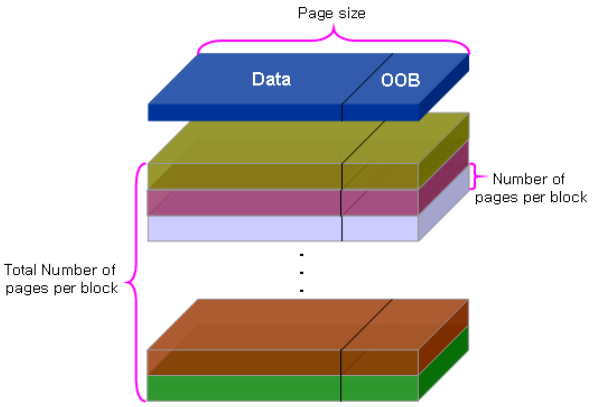
기술하였다. 4장은 제안하는 기법의 설계 구조와 상세 구현 내용을 설명한다. 5장은 성능측정에 대해 설명한다. 마지막으로 6장에서는 결론 및 향후 연구로 끝을 맺는다.

2. 관련 연구 및 배경

2.1. NAND 플래시 메모리 특성

NAND 플래시 메모리는 물리적으로 여러 개의 블록으로 이루어져 있고, 블록은 다시 여러 개의 페이지로 구성된다. 16MB 미만의 저용량 플래시의 경우, 블록 사이즈가 16Kbyte이며, 16MB 이상의 고용량 플래시는 128Kbyte 블록 사이즈를 갖는다.[1]

NAND 플래시 메모리의 블록 내에 존재하는 페이지는 데이터 영역과 OOB (Out Of Band) 영역으로 나누어져 구성되어 있다. 저용량 플래시는 512Byte를 갖는 16개의 페이지로 구성되며, 고용량의 경우에는 2Kbyte를 갖는 32개의 페이지로 구성 된다. 플래시의 블록과 페이지는 <그림1>과 같은 구조로 구성되어 있다.



<그림 1. FTL Page-Level Mapping>

2.2. FTL (Flash Translation Layer)

NAND 플래시 메모리는 페이지 단위와 블록 단위로 구분되어 동작한다. 플래시의 read와 write의 동작은 페이지 단위로 이루어지고, erase 동작은 블록 단위로 일어난다. 플래시의 수명은 erase 동작에 각 셀 당 보통 100,000번 정도의 제약을 가지고 있고, 한 블록이라도 제한된 횟수를 넘게 되면 플래시의 동작을 신뢰할 수 없다.

위와 같이 플래시 메모리 특성 때문에 기존에 개발된 플래시 메모리를 이용한 파일 시스템[2][3][4]에서는 LFS(Log Structured File System) 및 특정 영역에 대한 캐시[5][6]를 이용하여 성능을 향상 시키는 방식을 사용한다.

FTL은 파일시스템에서 사용되는 논리적 주소인 섹터 주소를 플래시의 물리적 주소로 변환하기 위해 주소 매핑 테이블 (address mapping table)을 구성한다.

주소 매핑 테이블을 구성하는 방법은 매핑의 구성단위에 따라 블록 수준 매핑 (block-level mapping) 기법과 페이지 수준 매핑 (page-level mapping) 기법으로 나눌 수 있고, 일반적으로 블록 수준 매핑 기법이 널리 사용되고 있다.[7]

페이지 수준 매핑의 경우 NAND 플래시 메모리 크기에 따라 매핑 테이블의 용량을 과도하게 사용하는 단점이 있다.

블록 수준 매핑의 경우는 작은 영역에 대한 빈번한 갱신 동작 시에 비효율적인 메모리 낭비가 발생한다.

NAND 플래시 메모리에 저장되어 있는 데이터는 플래시의 특성상 바로 갱신될 수 없고, 소거 동작 후 가능하다. 따라서 해당 블록에 있던 유효한 데이터를 캐시 영역으로 복사한 후 해당 블록을 소거 한다. 소거된 블록 주소는 주소 매핑 테이블에서 사용하지 않는 블록으로 매핑 한다. 비어 있는 새로운 블록을 찾아 캐시 영역에 저장되어 있는 데이터를 기록 한 후, 파일시스템에서 사용된 논리적 주소의 주소 매핑 테이블에 새로운 블록 주소를 저장함으로써 작업은 완료하게 된다.

이 과정에서 여러 번의 페이지 복사와 블록 소거 동작에 따른 오버헤드가 발생할 수 있다. 따라서 앞에서 기술한 오버헤드를 최소한으로 줄이고, 입출력의 속도를 향상 시키는 것이 플래시의 효율성을 극대화 시킬 수 있는 핵심이 된다.

플래시 메모리의 물리적인 동작 수행은 FTL을 통해 MTD (Memory Technology Device)[8]에서 이루어지며 이중 플래시 성능 향상 기법들은 주로 FTL 영역에서 활발히 연구가 이루어지고 있다.

2.3. FAT 파일시스템

FAT 파일시스템은 단순한 구조로 되어 있다. FAT 파일시스템은 MBR (Master Boot Record)의 파티션 정보에서 PBR (Partition Boot Record)의 정보를 얻을 수 있다.

PBR은 루트 디렉터리 정보가 저장되어 있다. 디렉터리는 저장되어 있는 파일 또는 서브 디렉터리에 대한 기본 정보를 담고 있는데, 그 안에는 파일이 저장되어 있는 첫 번째 클러스터에 대한 정보도 포함되어 있다. 해당 파일의 두 번째 클러스터부터는 FAT 테이블을 참조하여 그 위치를 파악할 수 있다.

FAT 파일 시스템에서 파일 생성은 비교적 단순하게 구현되고 있다. 즉, 파일 생성 시 파일 사이즈와 파일 이름, 속성 값 등을 디렉터리 엔트리에 저장한다.

해당 정보를 저장하기 위해선 소거 동작이 블록 단위로 일어난다. 기존의 PBR에 해당되는 섹터의 블록 정보를 캐시에 저장한 뒤 새로운 블록을 할당하여 저장한 후 기존 블록을 소거하게 된다.

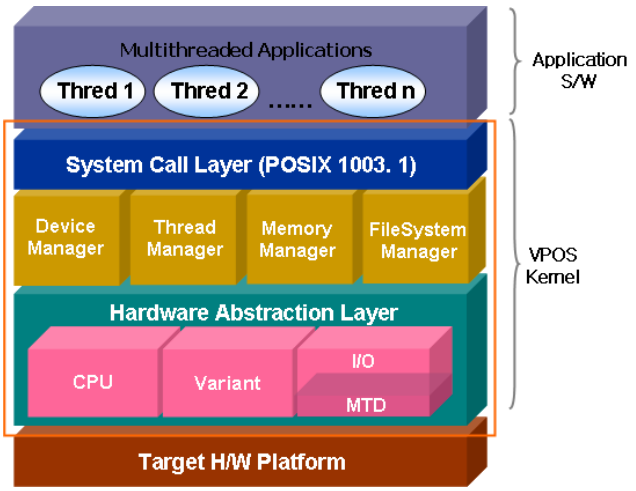
이와 같이 파일을 생성할 때 마다 블록을 소거해야 하는 단점이 있다. 이는 곧 NAND 플래시 수명을 단축시키는 요인이 되기 때문에 개선할 필요가 있다.

3. 개발 환경

본 논문에서의 구현 환경으로 하드웨어 플랫폼은 삼성 SMDK2410 평가 보드를 사용하였다. SMDK2410 보드는 ARM920T 코어를 적용한 S3C2410 32비트 RISC CPU와 삼성 K9F1208VOM NAND형 플래시를 채용한 SMC 카드를 포함하고 있다.

개발 운영체제로는 RTOS인 VPOS 2.0 (Verification Purpose OS)[9]을 사용하였다. VPOS 2.0은 다음과 같은 다섯 가지의 특징을 가지고 있다.

- 우선순위를 고려한 선점형 커널
- 커널의 실시간성 지원을 위한 동기화 기법인 PIP (Priority Inheritance Protocol) 적용
- HAL (Hardware Abstraction Layer) 도입
- 표준 및 POSIX 형식을 따르는 API 제공
- LINUX 호환성 지원을 위한 디바이스 드라이버 구조



<그림 2. VPOS의 구조>

4. 설계 구조 및 구현

이 장에서는 FAT 파일시스템에서 동적 메모리를 이용하여 플래시 메모리의 성능을 향상 시키는 기법에 대해 설명한다.

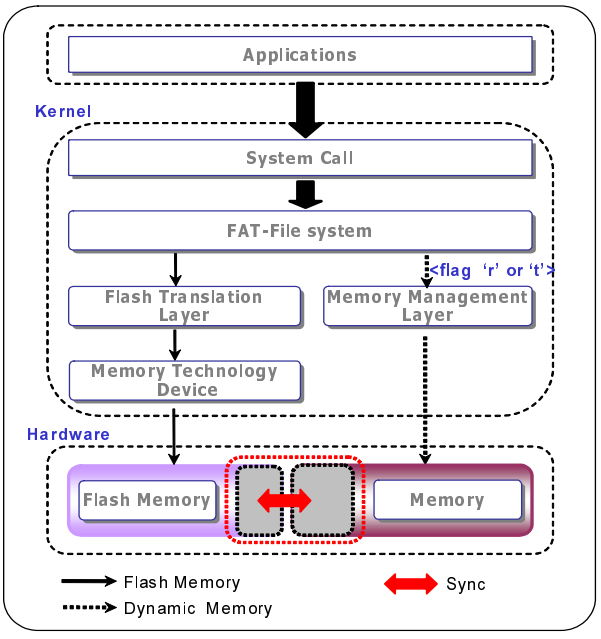
VPOS 2.0에 Block Device Driver와 MTD, FTL을 개발하였고, 파일시스템으로는 Simple-FAT를 개발하여 추가하였다. 그리고 가상의 동적 메모리 공간을 사용하기 위한 MML (Memory Management Layer)를 개발하여, FAT 파일시스템의 인터페이스에 추가하였다. VPOS 2.0의 구조는 <그림 2>와 같다.

4.1. 설계 구조

기존의 플래시 기반 파일 시스템에서 어플리케이션은 시스템 콜을 통하여 파일 시스템에 접근, 파일 등의 데이타 입출력을 수행하고, 파일 시스템은 FTL을 통하여 논리적 주소를 물리적 주소로 변환함으로써 MTD에 등록된 플래시에 접근하는 구조로 되어 있다.

본 논문에서는 <그림 3>와 같이 기존의 플래시 파일시스템에 MML (Memory Management Layer)을 추가하여, 동적 메모리의 일부분을 파일시스템 내부에서 사용 가능하게 하였다. 이를 통해 개발자는 동적 메모리의 특성을 고려할 필요 없이 동일한 방식으로 파일 시스템을 사용 할 수 있다. 또한 개발자의 필요에 따라 기존의 입출력 방법과 본 논문에서 제시하는 확장된 파일시스템 입출력 기법을 선택하여 사용하는 것이 가능하다.

MML은 파일 시스템과 메모리 사이에 존재하며 동적으로 메모리 공간을 할당 및 해제하며, 복사된 파일의 정보를 관리하는 소프트웨어 모듈이다.



<그림 3. 확장된 파일 시스템 구조>

기존의 플래시 성능 향상 기법은 주로 FTL 영역에서 이루어지며, 단순히 블록단위의 물리적인 플래시의 동작을 줄임으로써 기능을 향상시키고 있다. 하지만, 본 논문에서는 일반 운영체제에서 이미 존재하는 buffer에서의

캐싱을 하고 있지만, 새롭게 개발한 VPOS 2.0에는 아직 캐시 기능이 없어 VPOS2.0의 기반에 파일 단위 캐시 기능을 추가함으로써 파일시스템 혹은 어플리케이션에서의 파일 접근에 따른 오버헤드도 줄일 수 있다. 파일 시스템에서 MML을 통하여 메모리의 일부분을 플래시 메모리처럼 사용함으로써 메모리에 로딩된 파일은 재접근 시 플래시 메모리에 접근하지 않고 메모리에서 처리함으로써 파일 시스템의 성능 향상 및 플래시의 수명 연장 효과를 얻을 수 있다.

4.2. 상세 구현

4.2.1. File Open

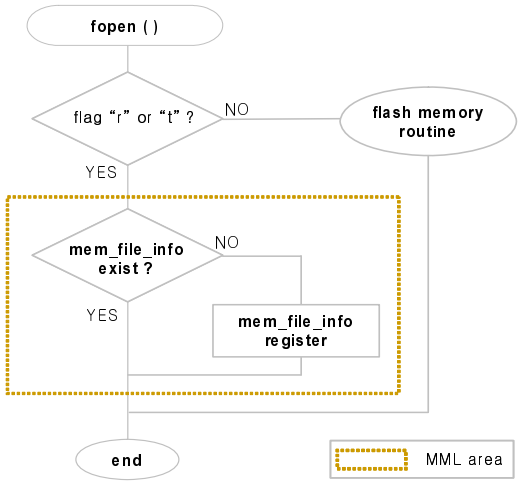
어플리케이션에서 파일 접근 시 플래시 메모리와 확장된 메모리 중 어느 것을 사용할지는 fopen() 시스템 콜의 argument를 통하여 개발자가 지정할 수 있다. 이를 위해 메모리 사용을 의미하는 “t" 플래그를 fopen() 시스템 콜에 추가하였다.

또한 읽기 전용을 의미하는 argument “r" 플래그인 경우 파일 정보 및 사이즈에 따라 플래시 혹은 메모리에서 파일을 처리할수 있게 하였다.

좀 더 상세히 살펴보면 fopen() 시스템 콜은 플래그 "t"가 존재하면 우선 MML에 fopen할 파일의 정보가 mem_file_info에 존재하는지의 여부를 검사한다. mem_file_info에 존재하지 않을 경우, 해당 file의 정보를 mem_file_info에 등록하며, 이 때에는 실제 파일의 데이터는 플래시에서 읽어오지 않는다. 그 이유는 실제 파일의 데이터를 읽거나 쓰기의 동작이 수행 시에 해당 파일에 데이터를 메모리로 가져와야만 플래시 수행에 따른 오버헤드를 줄일 수 있기 때문이다. 또한 전원 결함의 단점을 일부 보완하고자 플래시에 존재하는 파일을 메모리에 복사함으로써 파일 원본의 데이터는 유지 시킬 수 있다. 파일을 생성하기 위하여 “w+"와 같이 사용할 경우에는 mem_file_info에 등록 후 메모리에 파일을 생성하게 된다.

읽기 전용인 argument “r"을 선택하였을 경우 사용자는 플래시에서 1Mbyte 미만의 파일만 "t" 플래그를 선택하여 메모리 영역에서 읽기 전용을 수행한다. 메모리에서 처리되는 속도에 있어서 용량대비가 클 경우 효율성을 볼 수가 없기 때문에 현재는 사이즈를 1Mbyte로 정하였다. 추후 읽기 전용 파일에 대한 사이즈는 향후 개선 할 예정이다.

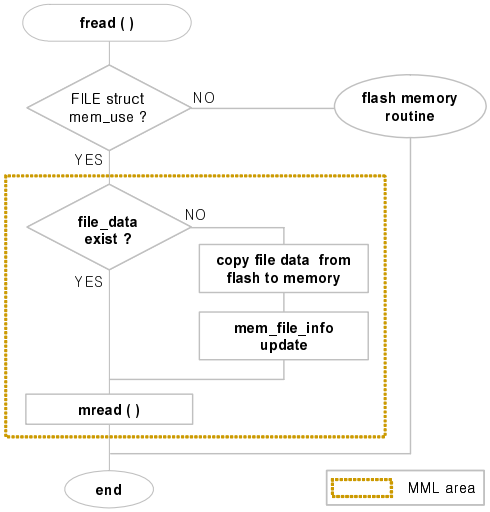
또한 fopen()시 개발자는 메모리의 용량이 open할 파일의 용량 보다 적을 경우, 메모리의 공간 확보 및 데이터의 일관성을 유지하기 위해 mem_file_info의 정보에 따라 적게 사용되어진 파일의 정보를 확인하여 삭제를 할 수 있다. 플래그에 ”t"가 없을 경우에는 파일의 정보를 읽는 기존의 FTL 루틴이 수행 된다. 이후는 기존의 파일 시스템의 처리루틴을 동일하게 수행한다. (<그림 4>)



<그림 4. fopen()시의 처리 흐름>

4.2.2. File Read

어플리케이션에서 fread()를 호출하면 파일 시스템은 fopen() 호출시 생성된 FILE 구조체내에 mem_use 항목에 따라 플래시 혹은 메모리의 접근 동작을 결정한다.



<그림 5. fread()시의 처리 흐름>

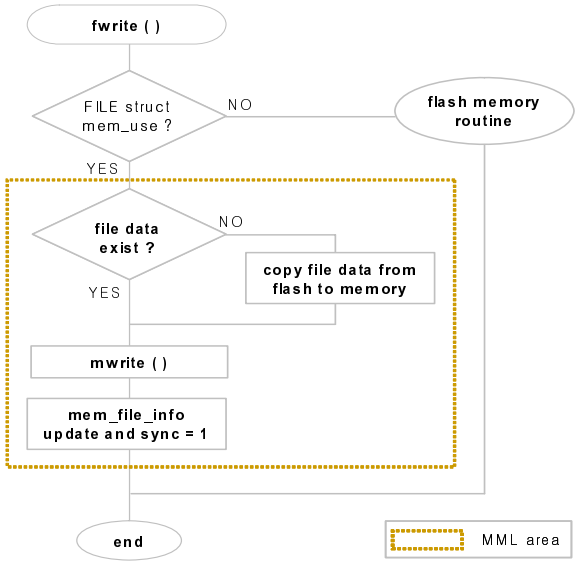
FILE 구조체내에 mem_use가 선택되어 있는 경우에는 파일의 정보가 MML의 mem_file_info에 등록되어 있는 것이므로 실제 해당 파일의 데이터가 메모리상에 존재하는지의 여부를 검사한다.

해당 파일의 데이터가 없을 경우, 기존 플래시 루틴을 호출하여 MML을 통하여 할당받은 메모리 주소에 플래시

로부터 파일 데이터를 복사한다. 그리고 mem_file_info의 파일 데이터 존재 여부를 갱신한다. mem_file_info에 파일 데이터가 존재하게 되면 플래시로 접근하지 않는 mread() 함수를 통하여 메모리의 파일 정보를 어플리케이션에 넘겨주게 된다. (<그림 5>)

4.2.3. File Write

어플리케이션에서 fwrite()를 호출 시 동작하는 흐름은 앞서 언급한 File Read의 경우와 동일한 흐름을 갖는다. 다른 점은 메모리에 데이터를 저장하기 위해 mwrite()가 수행 된다는 것과, 플래시의 파일 데이터와 메모리의 파일 데이터가 달라지므로 mem_file_info의 sync 값을 설정한다는 것이다. sync값을 설정하는 것은 플래시와 메모리간의 파일 sync를 해결하기 위한 것으로 아래에서 설명한다.(<그림 6>)

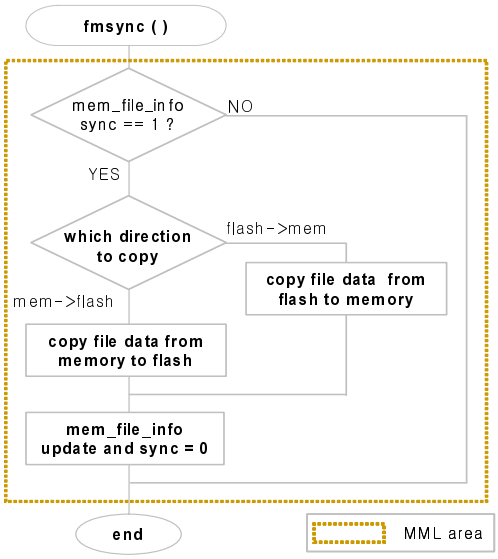


<그림 6. fwrite()시의 처리 흐름>

4.2.4. File Sync

확장된 파일시스템 처리 기능을 사용한 fwrite()가 호출되면 메모리에 존재하는 파일의 내용을 수정하거나 생성이 되고, 플래시의 파일 내용과 메모리의 파일 내용이 달라지는 문제가 발생하게 된다. 이때 개발자는 필요에 따라 플래시의 내용을 메모리로 재복사하거나 혹은, 메모리의 내용을 플래시로 저장할 수 있는 sync 기능을 수행하는 fmsync()를 호출할 수 있다.

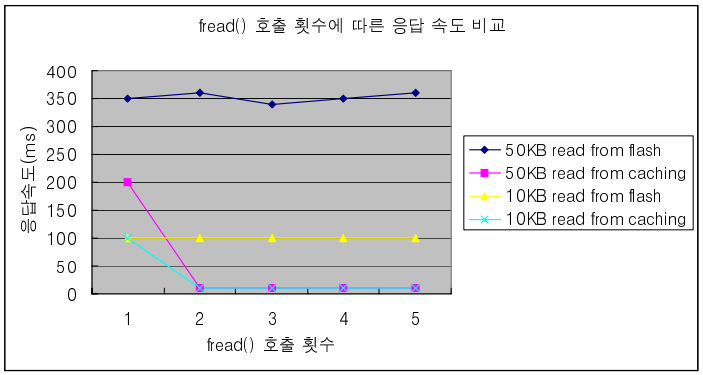
개발자가 fmsync()를 호출하면 파일 시스템은 플래시의 파일 내용과 메모리의 파일 내용이 서로 다른지의 여부를 mem_file_info의 sync값을 통하여 검사하게 된다. 파일의 내용이 같은 경우는 별도의 수행 작업이 없이 종료가 되지만, 파일의 내용이 서로 다른 경우에는 메모리의 내용을 플래시로 복사 하거나, 플래시의 내용을 메모리로 복사할지의 여부를 fmsync()의 argument를 통하여 선택할 수 있다.(<그림 7>)



<그림 7. fmsync()시의 처리 흐름>

5. 성능 평가

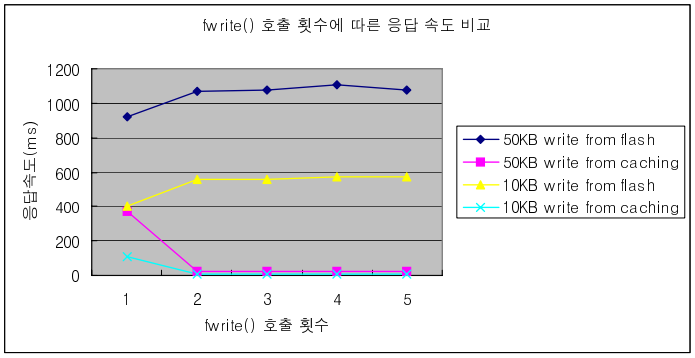
한 파일을 읽고, 쓰는데 있어서 블록의 사이즈인 16KB를 기준으로 측정하였다. 10KB, 50KB 파일을 읽고 쓰는데 있어 측정된 데이터는 다음과 같다.



<그림 8. 10KB, 50KB fread() 측정>

10KB와 50KB 파일을 fread()는 캐시로부터 데이터를 읽는 부분이 초기에는 플래시에서 읽는 응답 속도와 비슷하다. 이는 초기 캐시부분에 해당 파일이 없어 플래시로부터 데이터를 복사하기 때문에 걸리는 시간으로 읽기 속도가 비슷하다. 하지만 두 번째부터 캐시로부터 읽는 응답속도는 플래시 보다는 현저하게 빠른 것으로 알 수 있다.(<그림 8>)

fwrite() 또한 fread()와 동일하게 10KB와 50KB 파일을 쓰는데 있어 캐시로부터 데이터를 쓰는 부분은 플래시에서 쓰는 속도와 차이는 있지만, 두 번째부터 데이터를 쓰는데 있어서는 확연히 차이가 많이 난다. (<그림 9>)



<그림 9. 10KB, 50KB fwrite() 측정>

6. 결론 및 향후 연구

본 논문에서는 기존에 널리 사용되고 있는 FTL에서의 캐시를 이용한 성능 향상 기법 외에 동적 메모리를 이용한 플래시 메모리 파일 시스템에 대한 성능 향상을 위한 최적화 기법을 제안하였다.

제안된 기법은 기존의 FAT 파일 시스템 내에 구현되어 있는 플래시 메모리 처리 루틴을 확장하여, 시스템의 메모리를 동적으로 사용 가능도록 구현하였다. 이로써 플래시의 입출력 및 블록 소거 동작을 줄여 오버헤드를 감소시켰고, 그 결과 파일 시스템의 파일 처리 속도가 향상되고 플래시 메모리의 수명 또한 연장시킬 수 있다.

기존에 사용되고 있는 FTL에서의 물리적인 플래시 접근 감소 기법과 본 논문에서 구현한 파일 시스템에서의 파일 접근의 성능 향상 기법은 서로 통합하여 사용이 가능하다. 통합하여 사용 시 전체적인 파일 시스템을 통한 파일 접근 어플리케이션의 성능 향상을 얻을 수 있었다. 또한 파일 시스템 내에 구현함으로써 메모리 디스크를 사용하는 시스템에 비해 메모리 낭비의 오버헤드를 제거할 수 있다.

어플리케이션 측면에서는 시스템 파일 처리 함수인 fopen()의 플래그 하나를 추가함으로써, 기존의 어플리케이션 구조의 큰 변화 및 수정 없이 사용 가능하다.

현재는 파일 단위의 MML에서의 매핑 테이블을 플래시와 동일한 블록 단위로 관리를 하려고 한다. MML의 영역에서 MMT(Memory Mapping Table)을 추가하여 접근되는 플래시의 블록만을 메모리에 복사하고 MML에서 관리를 하려고 한다.

읽기 전용 파일인 경우 현재는 1Mbyte만 허용하고 있으나 향후 처리 기준을 규격화 하여 파일의 사이즈 및 메모리의 저장 공간 확보 마련과 파일의 이름에 대한 처리로 세밀한 결과 값을 얻고자 한다.

참고 문헌

[1] NAND Flash Memory and SmartMedia Data Book, Samsung Electronics, Co., 2003

[2] Atsuo Kawaguchi, Shingo Nishioka and Hiroshi Motoda, "A Flash-Memory Based File System," Proceedings of 1995 USENIX Techinical Conference, pp.155-164, 1995.

[3] 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.

[4] 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.

[5] 이정훈, 김신덕, "버퍼 시스템을 내장한 새로운 플래쉬 메모리 패키지 구조 및 성능 평가," 정보과학회논문지, 2005년 2월

[6] 송형근 차호정, "내장형 시스템에서 JFFS2 플래쉬 파일 시스템에 적합한 페이지 캐시 구조," 한국정보과학회 가을 학술발표논문집 Vol, 30,, No.2

[7] 박상호, 안우현, 박대연, 김정기, 박승민, "플래시 메모리를 위한 파일 시스템 구현," 정보과학 논문집, 2001년 10월.

[8] Memory Technology Device (MTD) Subsystem for Linux, http://www.linux-mtd.infradead.org

[9] 김지민, 유민수, "SoC 설계와 검증을 지원하는 실시간 운영체제," 한국정보처리학회 춘계학술발표회 논문집, 2005년 5월.

[10] Intel Co., "Understanding the Flash Translation Layer(FTL) Specification", AP-684