2

플래시 파일 시스템 기술 소개

지인정보기술 ▌김 성 관

1. 서 론

플래시 메모리는 비휘발성의 메모리 반도체로 집적도가 높고 충격에 강하며 저전력으로도 동작 가능하기 때문에 최근 휴대용 단말기 및 임베디드 시스템 등을 위한 저장 매체로서 각광받고 있다. 단말기 메인보드에 직접 장착되는 플래시 메모리 칩, 메모리 카드 또는 USB 플래시 드라이브, 그리고 최근 시장에 선보이기 시작한 플래시 메모리 기반 SSD(Solid State Disk) 등의 형태로 광범위하게 사용되고 있다.

주로 사용되는 플래시 메모리에는 NOR형과 NAND형의 두 가지 종류가 있는데, 바이트 단위로 주소 지정이 가능한 NOR형은 실행 코드 저장용으로, 페이지 단위 또는 블록 단위로만 주소 지정이 가능한 NAND형은 데이터 저장용으로 주로 사용되고 있다. 앞서 언급한 여러 가지 형태의 플래시 메모리 기반 데이터 저장 매체들은 모두 NAND형 플래시 메모리를 사용한다.

플래시 메모리는 하드 디스크와 같은 일반적인 블록 장치와는 달리 읽기(read), 쓰기(write) 동작 이외에 소거(erase) 동작이 필요하다는 큰 차이점이 있다. 즉, 저장되어 있는 기존의 데이터를 새로운 값으로 갱신하기 위해서는 그 위치에 해당하는 메모리 셀(cell)을 먼저 전기적으로 소거한 다음, 새 데이터 값을 기록해야 한다. 그런데 읽기 및 쓰기 동작의 경우 페이지 단위로 이루어지는데 반해, 소거 동작은 페이지보다 큰 단위인 블록 단위로 이루어진다. 이렇게 소거 동작이 필요하다는 점과 쓰기와 소거의 단위가 일치하지 않는다는 점이 플래시 메모리 관리 소프트웨어를 복잡하게 만드는 근본 이유이다.

이 글에서는 NAND형 플래시 메모리를 저장 매체로 사용하기 위해 필요한 관리 소프트웨어가 어떻게 구성되는지 소개하고, 그 중에서도 특히 플래시 메모리 기반 파일 시스템과 관련된 여러 기술적인 문제와 최근 기술 동향에 대해서 살펴본다.

2. NAND 플래시 메모리 개요

그림 1은 플래시 메모리의 구조를 보여 준다. 보통 블록 크기가 128KB, 페이지 크기가 2KB인 것을 대블록(large block) 플래시 메모리라고 하며, 블록 크기가 16KB, 페이지 크기가 512B인 것을 소블록 플래시 메모리라고 한다. 한 메모리 셀 당 2비트를 나타낼 수 있는 MLC(Multi-Level Cell) NAND 플래시 메모리의 경우 블록 크기가 512KB, 페이지 크기가 4KB인 제품도 있다.

플래시 메모리의 기본 동작은 표 1에 요약되어 있다. 읽기, 쓰기, 그리고 소거 동작 이외에 복사(copy-back) 동작이 있는데, 이 동작은 어떤 페이지에 있는 데이터를 다른 블록에 있는 페이지로 복사하고자 할 때 매우 유용하다. 대부분의 NAND 플래시 메모리 제품에서 이 복사 동작이 지원되지만, 일부 MLC NAND 플래시 메모리에서는 지원되지 않는다. 참고로 읽기 및 쓰기 동작에 대한 수행시간에는 RAM과 플래시 메모리 사이의 데이터 전송시간이 포함되어 있지 않다[1].

3. 플래시 메모리 관리 소프트웨어의 구성

그림 2는 플래시 메모리를 저장 매체로 사용하는 시스템에서 플래시 메모리 관리 소프트웨어를 구성하는 세 가지 방법을 개략적으로 보여 준다.

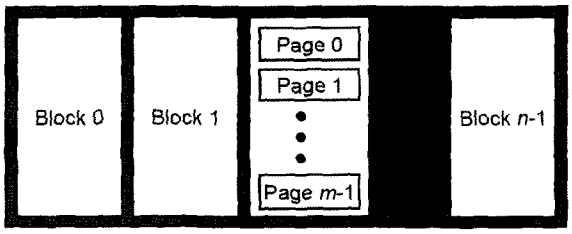
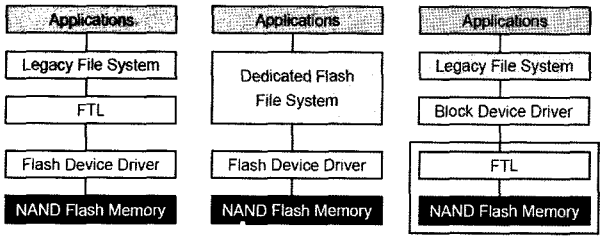


그림 1 NAND 플래시 메모리의 구조

† 본 연구는 정보통신부 및 정보통신연구진흥원의 IT신성장동력 핵심기술개발사업의 일환으로 수행하였음.
[2006-S-040-01, Flash Memory 기반 임베디드 멀티미디어 소프트웨어 기술 개발]

표 1 NAND 플래시 메모리의 기본 동작

동작	동작 단위	수행시간
읽기(read)	페이지	25 us
쓰기(write)	페이지	200~300 us
소거(erase)	블록	2 ms
복사(copy-back)	페이지	200~300 us



(A) FTL+기존 파일시스템 (B) 전용 플래시 파일시스템 (C) Flash Disk+기존 파일시스템

그림 2 플래시 메모리 관리 소프트웨어의 구성

그림 2에서 (A)와 (B)는 플래시 메모리 칩을 시스템 메인보드 상에 장착하는 경우이며, (C)는 플래시 디스크, USB 플래시 드라이브, 메모리 카드 등을 사용하는 경우에 해당된다. 그림 2에서 FTL이란 플래시 변환 계층(Flash Translation Layer)을 가리키는 것으로서, 플래시 메모리를 하드 디스크와 같은 일반적인 블록 장치로 변환해 주는 역할을 담당한다[2,3]. 그림 2(C)의 경우에는 FTL이 플래시 디스크 내에 펌웨어 형태로 내장되며 플래시 디스크 컨트롤러 상에서 수행된다.

플래시 메모리 기반 파일 시스템을 줄여서 플래시 파일 시스템이라 흔히 부르는데, 그림 2의 (A)와 (B)는 플래시 파일 시스템의 두 가지 구현 방법을 보여준다. 그림 2(A)는 별도의 FTL과 파일 시스템 모듈을 사용한 구성 방법으로 기존 파일 시스템을 그대로 활용할 수 있다. 대부분의 상용 솔루션들이 이러한 구성 방법을 따른다. 대표적인 FTL 솔루션으로 삼성전자의 PocketStore[4], M-Systems의 TrueFFS[5], DataLight의 FlashFX[6], 지인정보기술의 Z-FTL[7] 등이 있다. PocketStore와 TrueFFS 솔루션의 경우 그 위에 기존의 Microsoft FAT 파일 시스템을 그대로 사용하거나 또는 FAT 파일 시스템의 취약한 신뢰성 문제를 개선한 TFAT (Transactional FAT)을 사용한다. 지인정보기술 역시 Z-FTL과 함께 FAT 호환 파일 시스템인 Z-FAT[7]을 제공하는데 저널링(journaling)에 기반한 에러 복구 메커니즘을 사용하여 신뢰성을 확보하고 있다. DataLight는 FlashFX와 함께 높은 신뢰성이 주요 특징인 Reliance[6]라는 파일 시스템을 제공하는데, 이는 파일 입출력을 트랜잭션 기반으로 처리하는 FAT 비호환 파일 시스템이다.

반면, 그림 2(B)는 FTL과 파일 시스템이 합쳐져 있는 형태로 보통 플래시 메모리의 동작 특성을 고려한 전용 플래시 파일 시스템(dedicated flash file system)이 이런 형태를 갖는다. 리눅스 커널 배포판에 포함되어 있는 JFFS2[8]와 YAFFS[9], 그리고 상용 솔루션 중에서는 Qualcomm의 EFS 및 EFS2가 대표적인 예이다. 이들은 모두 LFS(Log-structured File System)[10] 형태의 파일 시스템 구조를 갖는데, 이는 플래시 메모리 블록을 소거한 다음에 그 블록 내에 있는 페이지들에 대하여 차례로 데이터를 기록해 나가는 플래시 메모리 고유의 사용 방식이 LFS의 그것과 매우 유사하기 때문이다.

신뢰성 측면에서만 본다면 그림 2(B)의 방식으로 구현된 LFS 형태의 전용 플래시 파일 시스템이 (A) 방식에 비하여 근원적으로 더 유리하다. LFS 형태로 파일 시스템을 관리하게 되면, 이전 데이터를 유지한 채 새 데이터를 추가하는 LFS 고유의 동작 방식으로 인하여 어느 순간 전원 중단 등의 돌발적인 오류 상황이 생기더라도 이전의 안전한 파일 시스템 상태로 복귀하는 것이 용이하기 때문이다. 물론 (A) 방식으로도 저널링이나 트랜잭션 기법을 기존 파일 시스템에 적용하여 보완함으로써 신뢰성을 높일 수 있지만 이 경우에는 추가의 오버헤드를 피할 수 없다.

그럼에도 불구하고 대다수의 상용 모바일 임베디드 제품들이 그림 2(A) 방식을 따르는 것은 다음의 두 가지 이유 때문이다. 첫째, (B) 방식의 경우 현실적으로 채택 가능한 솔루션으로서는 미흡한 측면이 많기 때문이다. 대표적으로 JFFS2와 YAFFS의 경우 오픈 소스 특성 상 최신 NAND 플래시 메모리 기술에 빠르게 대응하지 못해 왔으며 그로 인해 특정 플래시 메모리 칩을 지원하지 못하거나 또는 성능 상 제약을 가지는 경우가 많다. 둘째, 플래시 메모리가 많이 사용되는 모바일 제품군에 있어서는, UMS(USB Mass Storage) 서비스나 메모리 카드 등의 다양한 응용을 지원해야 하는 경우가 많은데 이를 위해서는 (A) 방식으로 소프트웨어를 구성하는 것이 훨씬 더 유리하기 때문이다. 그림 3에서 확인할 수 있듯이, 분리된 FTL 계층이 없다면 블록 장치 인터페이스를 제공할 수 없게 되어 UMS 서비스를 제공할 수 없게 된다. 또한 블록 장치 상에서 동작하는 기존 파일 시스템이 없다면 메모리 카드와 같은 외장 블록 장치를 관리하기 위한 별도의 파일 시스템을 두어야 한다. 특히, 메모리 카드의 경우 100% FAT 파일 시스템을 포맷을 갖기 때문에 FAT 파일 시스템을 사용하게 되면 하나의 파일 시스템으로 내장 플래시 메모리와 외장 메모리 카드를 같이 관리할 수 있게 되는 장점이 있다.

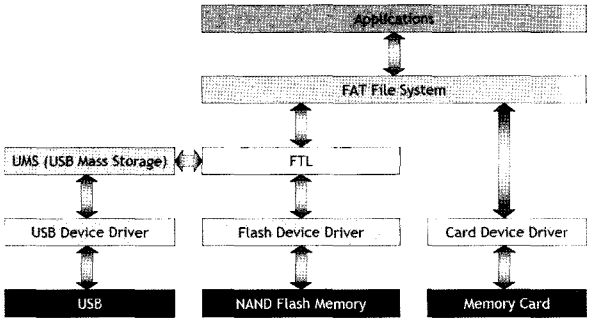


그림 3 다양한 응용에 유리한 소프트웨어 구조

한편, 삼성전자의 moviNAND, M-Systems의 DoC(Disk on Chip), 하이닉스의 DOC-H3 등은 칩 패키지 내에 컨트롤러가 있으며 펌웨어 형태로 내장된 FTL은 이 내장 컨트롤러 상에서 수행된다. 즉, 패키지만 반도체 칩 형태로 되어 있을 뿐, 사실상 메모리 카드와 동일한 성격의 제품군이다. 따라서 이러한 플래시 메모리 장치를 사용하는 경우에도 그림 2(C)와 같은 방식으로 기존 파일 시스템을 그대로 적용할 수 있다. 참고로 이런 형태의 제품이 출시되는 것은 FTL을 호스트 시스템의 OS에 넣지 않고 칩 내부에 넣음으로써 FTL을 호스트 시스템 OS에 이식하고 검증하는데 소요되는 부담을 줄일 수 있기 때문이다.

4. FTL의 동작 개요

전용 플래시 파일 시스템의 경우 FTL의 기능을 포함하고 있으며 기존 파일 시스템을 사용하는 경우에는 FTL 상에서 동작하게 되기 때문에, 플래시 파일 시스템에 대한 여러 가지 이슈에 대해서 이해하기 위해서는 FTL의 동작 원리를 이해하는 것이 필요하다.

4.1 주소 변환

주지하고 있듯이 플래시 메모리는 하드 디스크처럼 데이터를 현재 위치에서 바로 수정(in-place update)하는 것이 불가능하다. 앞서 언급하였듯이 한번 저장된 데이터를 수정하기 위해서는 그 데이터를 포함한 블록을 먼저 소거해야 하는데, 블록을 소거한 직후 새 데이터를 기록하기 전에 전원이 중단되면 이전 데이터마저 소실될 수 있기 때문이다. 따라서 항상 이전 데이터를 그대로 둔 채 새 데이터를 다른 곳에 기록한 다음, 마지막 단계에서 새 데이터가 기록된 곳으로 일종의 포인터를 변경해 준다. 이처럼 FTL에 의해서, 논리적인 주소인 섹터번호에 의해서 참조되는 데이터가 저장되는 실제 물리적인 주소인(블록번호, 페이지번호)가 실행 중에 계속 변경될 수 있는데, 이를 주소 변환(address translation, mapping)이라 하며 그림 4는 이 처리 과정을 보여 준다.

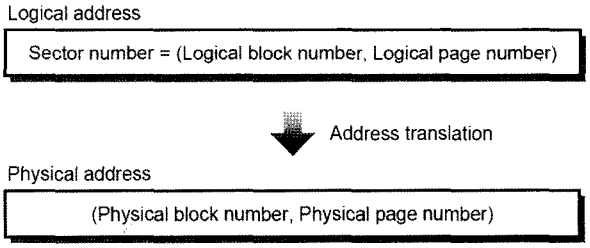


그림 4 FTL의 논리-물리주소 간 주소 변환

이 때 두 가지 선택이 가능한데, 하나는 블록번호 수준에서만 주소 변환을 수행하는 것이고(물리 페이지 번호는 논리 페이지번호와 동일), 다른 하나는 페이지 번호 수준에서 주소 변환을 수행하는 것이다. 보통 전자를 블록 수준 사상(block-level mapping)이라 하며 후자를 페이지 수준 사상(page-level mapping)이라 한다.

4.2 블록 수준 사상 FTL

블록 수준 사상 FTL의 경우 사상의 단위가 블록이기 때문에 사상 정보를 저장하는 사상 테이블(mapping table)의 크기가 페이지 수준 사상 FTL에 비해 작다는 장점이 있다. 사상 테이블 역시 플래시 메모리 상에 저장 관리되므로 사상 테이블의 크기가 커지게 되면 그만큼 FTL에 의해서 제공되는 가상 블록 장치의 크기가 줄어들게 된다.

블록 수준 사상 FTL에 있어서 어떤 블록의 데이터를 갱신하게 되면, 미사용 중인 빈 블록 하나를 잡고 새 데이터를 이 블록에 기록하게 된다. 그리고 기록이 끝나면 이전 블록 대신 이 새 블록을 가리키도록 사상 테이블을 갱신하게 된다. 그런데 이 과정에서 이전 블록에 있던, 갱신되지 않은 나머지 유효한 데이터가 존재하면 이를 새 블록으로 복사해 주어야 한다. 이를 흔히 블록 병합(merge) 동작이라고 하는데(그림 5 참조), 이 동작은 유효한 페이지 개수만큼의 데이터 복사와 한 번의 블록 소거 동작을 수반하기 때문에 큰 오버헤드가 될 수 있다. 따라서 블록 수준 사상 FTL의 성능을 높이기 위해서는 가능한 한 블록 병합의 횟수를 줄이는 것이 매우 중요하다(블록 병합의 횟수를 줄이면 각 블록 병합의 비용도 줄어드는 것이 일반적이다).

블록 병합의 횟수를 줄이기 위해 널리 사용하는 방법으로 쓰기 버퍼 블록(write buffer block) 기법이 있다. 어떤 블록의 일부 데이터가 갱신되자마자 바로 블록 병합을 실시하면 빈번한 블록 병합으로 큰 오버헤드가 발생하게 된다. 따라서 되도록이면 블록 병합을 최대한 뒤로 미루어 실시하는 것이 유리하다. 이렇게 할 경우 블록 병합이 되기 전까지는 하나의 논리블록

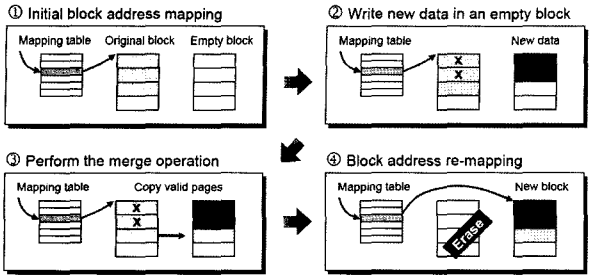


그림 5 블록 수준 사상 FTL의 블록 병합 동작

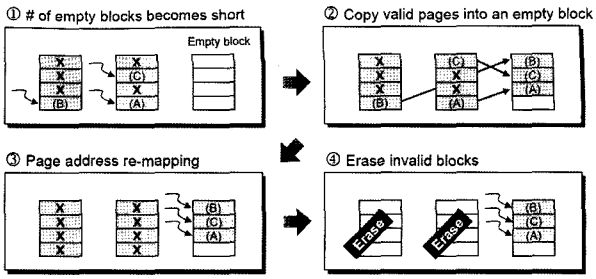


그림 6 페이지 수준 사상 FTL의 쓰레기 수집 동작

데이터를 두 개의 물리블록에 나누어 저장하게 되는데, 이 때 갱신되는 데이터가 기록되는 새 블록을 쓰기 버퍼 블록이라 부른다.

극단적인 예로 모든 논리블록에 대하여 쓰기 버퍼 블록을 한 개씩 할당하면 블록 병합의 횟수를 최소화할 수 있다. 그러나 이렇게 하면 파일 시스템에게 제공되는 블록 장치 공간이 원래 플래시 메모리 공간의 절반가량으로 줄어들기 때문에, 적절한 성능이 만족되는 선에서 쓰기 버퍼 블록의 개수를 최소화할 필요가 있다. 그리고 이 경우 한정된 쓰기 버퍼 블록 자원을 놓고 모든 논리블록들이 경쟁하게 된다. 그 결과 파일 시스템에서 FTL로 내려가는 어떤 한 그룹의 기록 요청들이 참조하는 논리블록의 개수가 쓰기 버퍼 블록의 개수를 초과하게 되면 강제로 블록 병합이 발생하게 된다. 블록 수준 사상 FTL이 사용되는 시스템에서 작은 단위의 임의적 쓰기(small random write) 요청이 많을 경우 성능이 저하되는 것은 바로 이런 이유 때문이다.

4.3 페이지 수준 사상 FTL

페이지 수준 사상 FTL은 블록 수준 사상 FTL과는 달리 자유롭게 각각의 논리페이지를 물리페이지에 배치할 수 있다는 장점이 있다. 따라서 쓰기 버퍼 블록을 한 개만 사용해도 되며, 어떤 한 논리블록 내에 있는 논리페이지들을 굳이 하나의 같은 물리블록 내로 사상할 필요가 없기 때문에 앞서 설명한 블록 병합 동작도 필요 없다. 그러나 블록 병합 오버헤드가 없는 대신 다음과 같은 쓰레기 수집(garbage collection) 오버헤드가 있다.

페이지 수준 사상 FTL은 하나의 쓰기 버퍼 블록 내의 페이지를 모두 사용하고 난 다음에 빈 블록을 찾아 새로운 쓰기 버퍼 블록으로 지정하고 사용하기 시작한다. 그 결과 이 동작을 지속적으로 반복하게 되면 궁극적으로는 모든 블록을 소진하게 된다. 물론 새로운 데이터가 쓰기 버퍼 블록에 기록되면서 더 이상 유효한 데이터를 가지고 있지 않은 페이지들이 기존 블록들 내에 생겨나므로 이들을 재활용할 수 있다. 그런데 보통 그런 무효화된 페이지들이 한 블록에 모여 있지 않기 때문에, 쓰기 버퍼 블록으로 사용하기 위해 어떤 블록을 소거하려면 그 블록 내에 있는 유효한 페이지들을 다른 블록으로 복사하여 이동시켜야 한다. 이를 흔히 쓰레기 수집이라고 하는데(그림 6 참조), 이 동작은 유효한 페이지 개수만큼의 데이터 복사와 한 번의 블록 소거 동작을 수반하기 때문에 큰 오버헤드가 될 수 있다. 따라서 페이지 수준 사상 FTL의 성능을 높이기 위해서는 가능한 한 쓰레기 수집 동작의 횟수와 각 쓰레기 수집 동작의 비용을 줄이는 것이 매우 중요하다.

참고로 그림 2(B)에서 언급되었던 전용 플래시 파일 시스템의 경우 보통 LFS 형태의 파일 시스템 구조를 갖는다고 하였는데, LFS에서 말하는 로그 블록(log block), 즉 갱신되는 데이터를 기록해 나가는 블록의 개념이 페이지 수준 사상 FTL에서 말하는 쓰기 버퍼 블록의 개념과 같다. 따라서 LFS와 비슷한 방식으로 동작하는 JFFS2, YAFFS, EFS 등은 모두 이 쓰레기 수집 동작을 필요로 한다.

4.4 블록 수준 사상 vs. 페이지 수준 사상

블록 수준 사상 FTL과 페이지 수준 사상 FTL 중에서 어느 쪽이 더 좋은 방식인가에 대해서는 명확하게 결론을 내리기 어렵다. 어떤 측면을 비교하느냐에 따라서 그리고 사용 환경에 따라서 우열이 바뀔 수 있기 때문이다.

자원 사용 측면에서는 블록 수준 사상 FTL이 더 유리하다. 한 블록에 64개의 페이지가 있는 대블록 플래시 메모리를 사용할 경우, 블록 수준 사상 FTL은 페이지 수준 사상 FTL에 비해서 사상 테이블의 크기가 1/64에 불과하다. 사상 정보의 총 개수가 작다는 것은 단위 데이터를 갱신할 때 함께 갱신해 주어야 하는 사상 정보의 개수가 작다는 뜻이다. 따라서 블록 수준 사상 FTL의 경우 사상 테이블을 유지 관리하는 오버헤드도 페이지 수준 사상 FTL에 비해 작다.

FTL이 제공하는 가상 블록 장치의 접근 성능에 대해서는 상황에 따라 우열이 바뀔 수 있다. 우선 읽기 접근 성능의 경우 두 가지 방식의 FTL에 대하여 별 차이가 없다. 그러나 쓰기 접근 성능의 경우 차이가 있을 수 있는데, 이 결과에 영향을 줄 수 있는 고려 사항들을 살펴보면 다음과 같다.

- 블록 장치의 섹터 공간이 각각의 물리적인 플래시 메모리 블록 내에서는 연속적으로 할당되어 있는가?
- 쓰기 요청이 작은 단위의 임의적 쓰기(small random write) 요청인가, 아니면 큰 단위의 순차적 쓰기(large sequential write) 요청인가?
- 블록 장치 전체 공간에서 유효한 데이터가 차지하고 있는 공간의 비율, 즉 사용률(utilization)이 몇 %인가?

먼저 블록 수준 사상 FTL에 대해서 생각해 보면, 물리 블록들 사이에서는 섹터 공간이 비연속적일 수 있지만 각 블록 내에서는 섹터 공간이 연속적이다. 따라서 큰 단위의 순차적 쓰기 요청이 있을 때, 쓰기 버퍼 블록 내의 모든 페이지들이 전부 사용된 다음에 블록 병합이 이루어지기 때문에 좋은 성능을 낼 수 있다. 그리고 이 결과는 블록 장치의 사용률에 관계없이 동일하다는 장점도 있다. 그러나 작은 단위의 임의적 쓰기 요청이 많으면 빈번한 블록 병합으로 인하여 좋지 않은 성능을 나타내게 된다.

페이지 수준 사상 FTL의 경우, 블록 장치의 사용률이 높지 않을 때에는 블록 수준 사상 FTL과 비교했을 때 유사한 성능을 보여 준다. 특히, 페이지 단위의 유연한 사상 관리가 가능하기 때문에 작은 단위의 임의적 쓰기 요청에 대해서는 더 좋은 성능을 나타낸다. 그러나 사용률이 높아지면 다른 결과가 나타날 수 있는데 이는 각 물리 블록에 할당된 섹터 공간의 연속성과 관련이 있다. 해당 블록 장치를 사용하는 동안 계속 큰 단위의 순차적인 쓰기 요청만 주어졌다면 각 물리 블록 내에서의 섹터 공간이 연속적일 가능성이 높다. 그러면 어느 정도 사용률이 높아져도 쓰기 성능에 큰 영향을 주지 않는다. 그러나 작은 단위의 임의적 쓰기 요청도 함께 주어지게 되면 각 물리 블록 내에서의 섹터 공간이 점점 비연속적으로 변하게 된다. 또한 유효한 페이지들과 무효화된 페이지들이 같은 종류끼리 서로 모여 있지 않고 점점 섞이게 된다. 그 결과 사용률이 높아짐에 따라 빈번한 쓰레기 수집 동작이 유발되어 쓰기 요청의 워크로드 특성에 관계없이 그 성능이 점점 나빠지게 된다.[1)]

플래시 메모리를 저장장치로 사용하는 많은 시스템에 있어서 성능과 관련된 가장 큰 문제는 저장장치의 사용률이 높아짐에 따라 쓰기 성능이 크게 저하된다는 사실이다. 블록 수준 사상 FTL의 경우 사용률이 높은 상태에서 작은 단위의 임의적 쓰기 요청이 많이 내려오는 경우가 최악의 상황이며, 페이지 수준 사상 FTL의 경우에는 각 물리 블록에 할당된 섹터 공간이 페이지 단위로 모두 비연속적인 경우가 최악의 상황이다. 주목할 만한 사실은 두 가지 방식 모두에 대해서 큰 단위의 순차적 쓰기 요청 위주로 쓰기 워크로드를 유지해 주면 최악의 상황을 피하고 상당한 성능 개선을 할 수 있다는 점이다. 다음 장에서 언급되겠지만 이는 플래시 파일 시스템 설계 시 중요한 힌트가 된다.

한편, 실제로 상용으로 사용되고 있는 FTL 솔루션들(메모리 카드 류 FTL 포함)을 살펴보면 대부분 블록 수준 사상 방식을 사용하고 있다. 이는 블록 수준 사상 FTL이 자원 사용 측면에서 더 유리하기 때문이다. 최근의 연구 동향을 보면, 블록 수준 사상 FTL을 사용하면서 작은 크기의 임의적 쓰기 요청에 대해서도 좋은 성능을 나타낼 수 있도록 부분적으로 두 방식을 혼용하는 기법도 제안되고 있다[11].

5. 플래시 파일 시스템에 대한 고려 사항

5.1 성능 개선을 위한 고려 사항

플래시 파일 시스템의 경우 읽기 성능보다는 쓰기 성능에서 문제가 되는 경우가 많다. 특히, 파일 시스템의 사용률이 높아졌을 때 쓰기 성능이 급격히 저하되는 경우가 많은데, 이 문제를 제거하거나 또는 완화시키기 위해서는 파일 시스템에 의해 FTL에게 전달되는 쓰기 워크로드의 특성에 대해서 분석해 볼 필요가 있다.

그림 7은 FTL 상에서 동작하는 파일 시스템과 전용 플래시 파일 시스템의 두 가지 경우에 대해서 어떻게 쓰기 워크로드가 생성되고 전달되는지를 보여 준다.

그림 7(A)는 분리된 FTL 계층 위에 플래시 파일 시스템(기존 파일 시스템도 가능)을 올린 형태이다. 응용 프로그램에 의해 생성되어 파일 시스템에게 전달되는 쓰기 요청의 경우, 파일 접근 지역성(file access locality)으로 인해 순차적인 특성을 갖는 경우가 적지 않다. 그런데 이와 같은 워크로드는 파일 시스템을 거

1) 페이지 수준 사상 FTL에 대한 이 같은 논의는 LFS 형태로 동작하는 전용 플래시 파일 시스템(예: JFFS2, YAFFS)에 대해서도 그대로 적용된다.

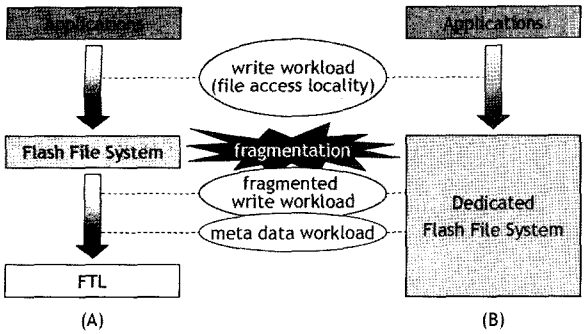


그림 7 쓰기 워크로드의 생성 및 전달 과정

치면서 단편화(fragmentation) 될 수 있다. 파일이 연속된 섹터 공간에 저장되지 못하고 여기 저기 흩어져 저장될 수 있기 때문이다. 그 결과 결국 FTL에게 전달되는 워크로드의 특성이 작은 단위의 임의적 쓰기 특성에 가까워져 좋지 않은 쓰기 성능을 초래하게 된다.

정확하게 기술하자면, 플래시 파일 시스템에 있어서 단편화의 의미는 하드 디스크 기반 파일 시스템에 대해서 통용되는 전통적인 단편화의 의미와는 다르다. 하드 디스크 기반 파일 시스템에서는 어떤 파일이 필요 이상의 실린더 그룹에 흩어져 있을 때 파일이 단편화 되어 있다고 하며, 그 결과 순차적인 파일 접근 워크로드에 대해 불필요한 디스크 탐색(seek) 시간이 유발될 수 있다. 플래시 파일 시스템에 있어서는 어떤 파일이 필요 이상의 논리블록[2]에 흩어져 있을 때 파일이 단편화 되어 있다고 정의할 수 있다. 그 결과 순차적인 파일 쓰기 워크로드에 대해 불필요한 블록 병합(블록 수준 사상 FTL을 사용한 경우)이나 쓰레기 수집(페이지 수준 사상 FTL을 사용한 경우) 동작이 유발될 수 있다. 따라서 플래시 파일 시스템의 섹터 할당 알고리즘을 설계할 때에는 이와 같은 단편화 문제가 가능한 한 최소화될 수 있도록 주의해야 한다[12].

FTL에게 전달되는 또 하나의 쓰기 워크로드는 파일 시스템 내부적으로 생성된 것으로, 파일 시스템을 관리하는데 필요한 이른바 메타 데이터(meta data)이다. 기존의 많은 파일 시스템에 있어서 메타 데이터에 대한 쓰기 요청은 작은 크기의 임의적 쓰기 특성을 갖고 또 빈번하게 발생되기 때문에 이로 인해 FTL의 성능이 크게 저하될 수 있다. 따라서 RAM 상에 메타 데이터 캐시(cache)를 두고 모아서 처리하는 등의 대책이 필요하다.

그림 7(B)에 나타나 있는 전용 플래시 파일 시스템의 경우에도 앞서의 논의가 거의 그대로 적용된다. 파일이 필요 이상의 플래시 메모리 블록에 흩어져 있음으로 해서 불필요한 쓰레기 수집으로 인한 성능 감소가 초래될 수 있다. 또한 메타 데이터 관리 상 문제도 마찬가지로 발생하는데, JFFS2의 경우 메타 데이터를 플래시 메모리의 스페어 영역(spare area)[3]에 함께 저장하도록 하여 메타 데이터를 기록하는데 소요되는 오버헤드를 줄이고자 한다. 그러나 이 경우 각 페이지의 스페어 영역에 흩어져 있는 메타 데이터 정보를 수집하는데 오랜 시간이 걸리기 때문에 파일 시스템의 마운트 시간이 길다는 단점이 있다.

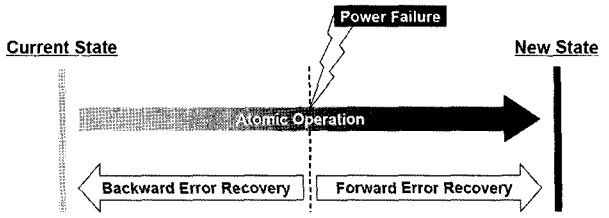


그림 8 원자적 연산과 에러 복구 과정

5.2 신뢰성 확보를 위한 고려 사항

다른 시스템 소프트웨어와 마찬가지로 플래시 파일 시스템에 있어서도 가장 중요한 고려 사항은 신뢰성이다. 특히, 플래시 메모리가 사용되는 제품 환경이 배터리로 운용되는 경우가 많아 불시의 전원 중단으로 인한 파일 시스템 손상 가능성이 높다. 따라서 그와 같은 상황에서도 파일 시스템이 보호될 수 있는 장치를 마련하는 것이 필요하다.

파일 시스템이 손상을 입고 메타 데이터에서 불일치성(inconsistency)이 발견되면 동작 불능 상태에 빠질 수도 있다. 예를 들어, 파일 삭제 동작 중 전원이 중단되어 파일이 사용하고 있던 섹터 공간은 반환되었는데 디렉토리에는 해당 파일에 대한 정보가 여전히 남아 있는 경우 문제가 될 수 있다. 이렇게 파일 시스템 상태에 변화를 주는 중요한 동작에 대해서는 해당 동작이 원자성(atomicity)을 갖는지 조사하고 그렇지 못하면 원자성을 갖도록 별도의 장치를 마련해 주어야 한다. 여기서 말하는 원자성이란, 해당 동작의 결과가 파일 시스템에 완전히 반영되거나 또는 전혀 반영되지 말아야 함을 의미하는 것이다(그림 8 참조).

LFS 형태의 전용 플래시 파일 시스템의 경우, 이전의 데이터를 그대로 둔 채 갱신되는 데이터를 로그 블록에 계속 추가해 나가는 형태로 동작하기 때문에,

2) 논리블록이란 FTL에 의해 제공되는 가상 블록 장치의 섹터 공간을 플래시 메모리 블록 크기 단위로 묶어 놓은 것이다.

3) NAND 플래시 메모리에서 데이터를 저장하는 영역을 메인 영역, 부가 데이터를 저장하는 영역을 스페어 영역이라 부르는데, 보통 512B의 메인 영역 당 16B의 스페어 영역이 있다.

갑자기 전원이 중단되더라도 이전의 안전한 파일 시스템 상태로 자연스럽게 복귀(rollback)할 수 있다. 앞서 3장에서 언급한 TFAT, Reliance 등의 파일 시스템은 트랜잭션 기반으로 파일 시스템 상태에 변화를 주는 주요 동작의 원자성을 보장한다. 트랜잭션이 완료되기 전에 전원이 중단되면 임시 공간에 저장해 두었던 트랜잭션 데이터를 버리고 이전의 안전한 파일 시스템 상태로 복귀하도록 한다. Z-FAT 파일 시스템의 경우에는 주요 파일 시스템 동작을 수행하기 전에 로그를 남기도록 한다. 중도에 전원이 중단되면 재부팅 후 남겨진 로그 정보에 기초하여 에러 복구를 시도하는데, 이 과정에서 에러 복구의 효율성을 위해 실패했던 동작에 대한 수행을 재시도 하기도 한다.

6. 결 론

이 글에서는 플래시 파일 시스템과 관련된 여러 기술적인 내용, 그 중에서도 특히 성능과 신뢰성 부분에 대해서 중점적으로 살펴보았다. 특히, FTL 고유의 동작 오버헤드와 연계된 플래시 파일 시스템의 단편화 문제가 어떻게 플래시 파일 시스템의 쓰기 성능을 저하시킬 수 있는지 분석하고 그 해결 방안을 제시하였다. 실제로 상용 제품에서 널리 사용되고 있는 FAT 파일 시스템 등의 기존 파일 시스템의 경우 FTL의 동작 특성을 전혀 고려하지 않고 있기 때문에, 이 글에서 언급된 보완 작업을 통하여 적지 않은 성능 개선을 할 수 있을 것으로 기대한다.

참고문헌

[1] NAND Flash Memory and SmartMedia Data Book, Samsung Electronics, 2006.

[2] Understanding the Flash Translation Layer(FTL) Specification, Intel Corporation, 1998.

[3] A. Ban, "Flash File System", U. S. Patent 5,404,485, 1995.

[4] PocketStore II, Samsung Electronics, http://www.samsung.com/Products/Semiconductor/Flashsw/pocketstore.htm, 2007.

[5] TrueFFS, M-Systems, http://www.m-systems.com/site/en-US/Technologies/Technology/TrueFFS_Technology.htm, 2007.

[6] FlashFX and Reliance, DataLight, http://datalight.com/products/, 2007.

[7] ZFS, Zeen Information Technologies, http://www.zeen.co.kr/, 2007.

[8] D. Woodhouse, "JFFS: The Journaling Flash File System", Ottawa Linux Symposium 2001, 2001.

[9] YAFFS(Yet Another Flash File System) Specification Version 0.3, http://www.aleph1.co.uk/yaffs/, 2007.

[10] 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.

[11] S.-W. Lee, "FAST: A Log Buffer Based FTL scheme with Fully Associative Sector Translation", UKC 2005, 2005.

[12] S.-K. Kim, "Performance Optimization Techniques for Legacy File Systems on Flash Memory", Ph.D Thesis, Seoul National University, 2007.

김 성 관

1991 서울대학교 컴퓨터공학과(학사)
1996 서울대학교 컴퓨터공학과(석사)
2007 서울대학교 전기컴퓨터공학부(박사)
1999~현재 지인정보기술(주) 대표이사
관심분야 : 임베디드 시스템, 파일 시스템
E-mail : skkim@zeen.snu.ac.kr