플래시 메모리를 위한 효율적인 사상 알고리즘
(An Efficient FTL Algorithm for Flash Memory)

정 태 선 † 박 형 석 ††
(Tae-Sun Chung) (Hyung-Seok Park)

요 약 플래시 메모리는 비 휘발성(non-volatility), 빠른 접근 속도, 저전력 소비, 그리고 간편한 휴대성 등의 장점을 가지므로 최근에 많은 임베디드 시스템에서 많이 사용되고 있다. 그런데 플래시 메모리는 그 하드웨어 특성상 플래시 변환 계층(FTL: flash translation layer)이라는 시스템 소프트웨어를 필요로 한다. 이 FTL의 주요 기능은 파일 시스템으로부터 내려오는 논리 주소를 플래시 메모리의 물리 주소로 변환하는 일이다. 본 논문에서는 STAFF(State Transition Applied Fast Flash Translation Layer)라 불리는 FTL 알고리즘을 제안한다. 기존의 FTL 알고리즘에 비하여 STAFF는 적은 메모리를 필요로 하면서 기존 일반 방법인 블록 사상 방법에 비하여 5배 정도 좋은 성능을 보인다. 본 논문에서는 기존 FTL 알고리즘과 STAFF의 성능 비교를 보였다.

키워드 : 플래시 메모리, 임베디드 시스템, 파일 시스템

Abstract Recently, flash memory is widely used in embedded applications since it has strong points: non-volatility, fast access speed, shock resistance, and low power consumption. However, due to its hardware characteristics, it requires a software layer called FTL(flash translation layer). The main functionality of FTL is to convert logical addresses from the host to physical addresses of flash memory. We present a new FTL algorithm called STAFF(State Transition Applied Fast Flash Translation Layer). Compared to the previous FTL algorithms, STAFF shows five times higher performance than basic block mapping scheme and requires less memory. We provide performance results based on our implementation of STAFF and previous FTL algorithms.

Key words : Flash memory, Embedded System, File System

1. 서 론

플래시 메모리는 비 휘발성(non-volatility), 빠른 접근 속도, 저전력 소비, 그리고 간편한 휴대성 등의 장점을 가지므로 최근에 많은 임베디드 시스템에서 많이 사용되고 있다. 그런데, 기존의 메모리와는 달리 반도체 기반의 플래시 메모리는 다음과 같은 성질을 가진다[1].

- "쓰기 전 지우기" 성질: 기존의 비 휘발성 메모리인 하드 디스크는 데이타 갱신의 경우에도 바로 데이타가 갱신될 수 있지만 플래시 메모리는 데이타 갱신 연산을 수행하기 위해서는 반드시 그 데이타를 포함한 영역이 지워져 있어야 하는 성질을 가진다.
- 배드 섹터의 존재: 플래시 메모리는 출하할 당시 또는 데이타 쓰기 연산을 하는 중에 해당 섹터가 배드가 될 수 있는 성질을 가진다.
- 마모도 평준화(wear-leveling) 요구: 플래시 메모리는 특정 섹터에 대한 쓰기 연산이 일정 횟수를 넘으면 그 섹터의 데이타 정보가 손상될 수 있는 특징을 가진다.

따라서 플래시 메모리 기반의 시스템은 이와 같은 플래시 메모리의 본질적인 하드웨어적인 한계점을 극복하기 위하여 시스템 소프트웨어가 필요한데 이 소프트웨어를 플래시 사상 단계(FTL: Flash Translation Layer)[2-7]라고 한다.

특히 플래시 메모리의 "쓰기 전 지우기" 성질은 일반 플래시 메모리의 지우기 단위와 쓰기 단위가 다르기 때문에[1] 플래시 기반 임베디드 시스템의 성능 저하를 가져오는 주요한 성질이다.

기존에 제안된 FTL 알고리즘에서 플래시 메모리의 이러한 "쓰기 전 지우기" 성질을 극복하기 위하여 주로 사용한 방법은 논리 물리 사상 방법이다. 즉, 논리 물리

† 정 회 원 : 아주대학교 정보및컴퓨터공학부 교수
tschung@ajou.ac.kr
†† 비 회 원 : 삼성전자 소프트웨어센터
steinpark@samsung.com
논문접수 : 2004년 6월 7일
심사완료 : 2005년 2월 22일

1) Hitachi에서 생산되는 플래시 메모리는 지우기 단위와 쓰기 단위가 같다.

사상 테이블을 이용하여 해당 물리 주소가 이미 데이타가 쓰여져 있으면 비어있는 플래시 메모리의 공간에 먼저 쓴 후 논리 물리 사상 테이블을 변경하는 방법이다.

이러한 FTL 알고리즘을 실제 임베디드 시스템에 적용할 때 또 다른 중요한 점의 하나는 사상 정보를 위한 메모리 요구량이다. 즉, 사상 정보는 성능을 위하여 값비싼 RAM에 저장되는데 이 사상 정보의 양이 많아서 RAM 요구량이 많아지면 전체 비용 면에서 FTL 알고리즘이 실제 임베디드 시스템에서 적용되기 힘들기 때문이다.

본 논문에서 STAFF라는 FTL 알고리즘을 제안한다. STAFF는 기존 알고리즘에 비하여 고성능을 내면서 메모리 요구량이 작은 특성을 지닌다.

논문의 구성은 다음과 같다. 2절에서 문제 정의와 기존 연구를 다루고, 3절에서 STAFF 알고리즘을 다루고, 4절에서 성능 평가 결과를 보인다. 마지막으로 5절에서 결론을 맺는다.

2. 문제 정의와 기존 연구

2.1 문제 정의

본 논문에서 섹터(sector)는 읽고, 쓰는 연산의 기본 단위이고 블록(block)은 삭제 연산의 기본 단위라고 가정한다. 이때 블록의 크기는 섹터 크기의 배수이다.

그림 1은 플래시 기반 임베디드 시스템의 기본 구조를 나타낸다. 가장 상위 레벨에 응용 프로그램이 있으며, 그 밑에 파일 시스템, 그리고 플래시 메모리를 위한 디바이스 드라이버가 존재하며 마지막으로 플래시 메모리가 존재한다. 파일 시스템은 플래시 메모리에 있는 데이타를 읽거나 플래시 메모리에 데이타를 저장하기 위하여 논리 섹터에 대한 읽기 혹은 쓰기 요청을 연속적으로 하게 되며, FTL 알고리즘에 의하여 이러한 논리 주소는 실제 플래시 메모리의 섹터 주소로 변경된다.

따라서 FTL에 대한 문제를 다음과 같이 정의할 수 있다. 플래시 메모리가 n개의 물리 섹터로 구성되고 파일 시스템은 플래시 메모리를 m개의 논리 섹터로 간주한다고 가정한다. 이때 m은 n보다 작거나 같다.

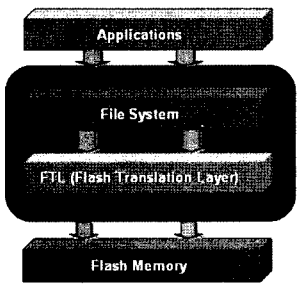


그림 1 플래시 파일 시스템의 아키텍처

정의 1. 플래시 메모리는 여러 개의 블록으로 구성되고, 각 블록은 여러 개의 섹터로 구성된다. 플래시 메모리는 다음의 성질을 가진다. 플래시 메모리의 물리 섹터 주소가 이미 데이타가 쓰여져 있으면 새로운 데이타가 그 위치에 다시 쓰여지기 위해서는 그 물리 섹터를 포함하는 블록이 지워져야만 한다. FTL 알고리즘은 파일 시스템이 보내는 논리 섹터 번호를 플래시 메모리의 물리 섹터 번호로 변경시키는 알고리즘이다.

2.2 기존연구

2.2.1 섹터 사상

섹터 사상 기법 [2]은 읽기 쓰기 단위인 섹터 단위로 논리 물리 사상 테이블이 존재하는 방법이다. 즉, 파일 시스템 관점에서 m개의 논리 섹터가 존재한다고 하면 논리-물리 사상 테이블의 행의 크기가 m이 된다.

그림 2는 섹터 사상 기법의 예를 보인다. 그림 2에서는 플래시 메모리는 총 4 개의 블록으로 구성되며 한 블록은 4 개의 섹터로 구성된다고 가정한다. 따라서 플래시 메모리에 존재하는 물리 섹터의 개수는 총 16 개이고 섹터 사상 기법은 이 16개 엔트리에 대한 사상 테이블을 가진다(그림 2에서는 13개만을 보였다).

예를 들어, 논리 섹터 번호 9번에 대한 쓰기 요청을 수행한다고 하면, 사상 테이블은 물리 섹터 번호 3을 리턴 하게 되고 물리 섹터 번호 3번에 데이타를 쓸 수 있다.

만일 같은 논리 섹터에 대한 갱신 연산이 요청되면 사상 테이블의 정보만 변경한 후 플래시 메모리의 비어있는 물리 주소에 데이타를 쓸 수 있으므로 삭제 연산을 최소화 할 수 있다.

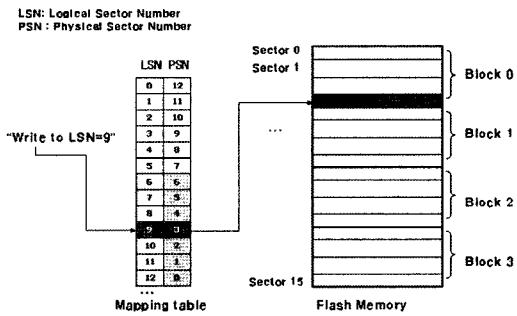


그림 2 섹터 사상

2.2.2 블록 사상

블록 사상 기법[3,4,6]은 섹터 사상 기법이 사상 테이블의 크기가 커지는 단점을 극복하기 위하여 제안된 방법으로 플래시 메모리의 삭제 단위인 블록 단위로 논리 물리 사상 테이블이 존재하는 방법이다.

그림 3은 블록 사상 기법의 한 예를 나타낸다. 사상테이블의 엔트리는 그림 3의 예에서 플래시 메모리가 블록 4개로 구성되므로 4개임을 볼 수 있다. 그런데 플래시 메모리의 읽기 쓰기 연산은 섹터 단위로 이루어짐으로 논리 주소의 섹터 오프셋과 물리 주소의 섹터 오프셋을 일치시킨다.

예를 들어 논리 섹터 번호 9번에 대한 쓰기 연산이 요청되면, 먼저 논리 섹터 번호 9번의 논리 블록 번호를 계산한다(9/4=2). 그리고 섹터 오프셋은 1이 된다. 블록 사상 테이블을 참조하면 물리 블록 번호 (1)을 얻을 수 있고 오프셋이 1이므로 해당 물리 섹터 위치에 데이타를 쓸 수 있다.

블록 사상 기법은 사상 테이블의 크기가 현격히 줄어들었지만 블록 단위로 사상 정보를 일치시켜야 하기 때문에 같은 논리 주소에 쓰기 연산이 많이 요청되는 경우 성능 저하를 발생시킨다. 예를 들어 논리 섹터 번호 9번의 쓰기 연산 요청 후 다시 논리 섹터 번호 9번에 쓰기 연산이 요청되면 사상 테이블을 변경하여 다른 위치의 플래시 메모리에 데이타를 쓴다고 할지라도 인접 섹터의 모든 정보가 옮겨지는 블록으로 복사되어야 한다.

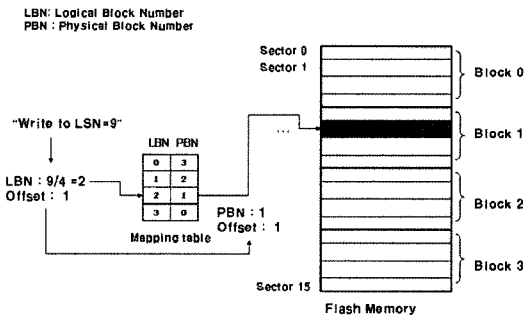


그림 3 블록 사상

2.2.3 혼합 사상

혼합 사상 기법[5,7]은 섹터 사상 기법과 블록 사상 기법을 혼합한 기법이다. 즉, 일단 블록 사상 기법과 같이 블록 단위의 사상 테이블이 존재하고, 블록 내에서는 섹터 사상을 하는 방법이다.

그림 4는 혼합 사상 기법을 보인다. 예를 들어 논리 섹터 번호 4에 쓰기 연산이 요청되면, 블록 사상 기법과 같이 사상 테이블을 참조하여 물리 블록 번호 (1)을 구한다. 다음으로 해당 블록에 대한 논리 블록과 물리 블록의 섹터 오프셋을 일치 시킬 필요 없이 섹터 정보를 저장한다. 예에서는 논리 섹터 번호 (9)를 플래시 메모리에 저장하였다.

플래시 메모리로부터 데이타를 읽을 때는 먼저 해당 논리 블록에 대한 물리 블록을 사상 테이블로부터 구한다. 다음으로 그 블록에 존재하는 논리 섹터 번호를 탐색함으로써 해당 데이타를 읽을 수 있다.

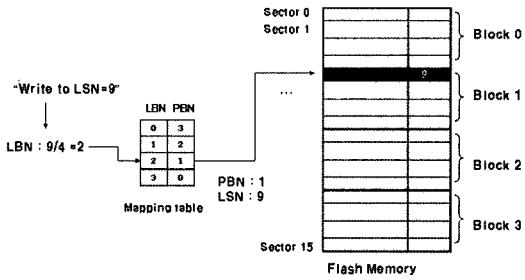


그림 4 혼합 사상

2.2.4 비교

기존의 FTL 알고리즘은 읽기/쓰기 성능과 사상 정보를 위한 메모리 요구량으로 비교될 수 있다.

읽기 쓰기 성능은 플래시 메모리의 I/O 횟수로 비교할 수 있다. 플래시 메모리의 I/O 시간에 비하여 RAM 상의 사상 테이블 연산 시간은 무시할 수 있다고 가정하면 읽기 쓰기 성능은 다음의 식으로 표현될 수 있다.

$$C_{read} = xT_r \quad (1)$$

$$C_{write} = p(T_r + T_w) + (1-p)(T_r + (T_e + T_w) + T_c) \quad (2)$$

위의 식에서 C_{read}, C_{write}는 각각, 읽기 쓰기 시간이고 T_r, T_w, T_e는 플래시 메모리의 읽기, 쓰기, 지우기 시간을 나타낸다. T_c는 복사 시간으로써 해당 블록을 지워야 하는 경우 빈 블록(free block)에 유효한 데이타를 복사한 후 해당 블록을 지운 후에 다시 유효한 섹터를 복사하는 필요한 시간을 의미한다. 변수 p는 해당 쓰기 연산이 삭제 연산을 필요로 하지 않을 확률을 의미한다. 여기서 하나의 논리 섹터는 한 물리 블록의 한 물리 섹터에 사상된다고 가정한다.

섹터 사상이나 블록 사상 방법에서는 식 (1)에서의 변수 x는 1이다. 왜냐하면 사상 테이블에 의하여 물리 섹터 위치가 바로 구해질 수 있기 때문이다. 혼합 사상에서는 변수 x의 값은 $1 \le x \le n$ 이다. 여기서 n은 블록 안의 섹터의 수를 의미한다. 혼합 사상에서는 물리 블록이 사상테이블에 의하여 결정되더라도 그 블록의 섹터를 탐색하여 해당 논리 섹터 번호를 읽어야 해당 데이타를 읽을 수 있기 때문이다. 따라서 혼합 사상은 섹터 사상이나 블록 사상에 비하여 읽기 시간이 많이 든다.

쓰기의 경우에는 해당 섹터가 쓰기 가능한 지를 알기 위하여 한번의 읽기 연산이 필요하다고 가정한다. 따라

서 식 (2)에서 T_r이 첨가되었다. 플래시 메모리의 읽기 연산 시간은 쓰기나 지우기 연산에 비하여 매우 작기 때문에 쓰기 연산 시간은 변수 p가 좌우한다는 사실을 알 수 있다. 섹터 사상이 삭제 연산을 필요로 할 확률이 가장 낮고 블록 사상이 그 확률이 가장 높다.

또 다른 비교 항목은 사상 정보를 위한 메모리 요구량이다. 표 1은 사상 정보를 위한 메모리 요구량을 보인다. 여기서 플래시 메모리는 128MB이고 총 8192 블록으로 구성되고 각 블록은 32 섹터로 이루어 진다고 가정한다[1]. 섹터 사상에서는 모든 섹터를 표시하기 위하여 3 바이트가 필요하고 블록 사상에서는 모든 블록을 표시하기 위하여 2 바이트가 필요하다. 혼합 사상에서는 블록 사상을 위하여 2 바이트가 필요하고 블록 내 섹터의 위치를 표시하기 위하여 1 바이트가 필요하다. 표 1은 메모리 요구량에 있어서 블록 사상이 섹터 사상이나 혼합 사상에 비하여 유리함을 알 수 있다.

표 1 사상 정보를 위한 메모리 요구량

	주소 지정 (B: Byte)	총 합
섹터 사상	3B	3B* 8192* 32 = 768KB
블록 사상	2B	2B *8192= 16KB
혼합 사상	2B+1B	2B*8192+1B*32*8192 = 272KB

3. STAFF(State Transition Applied Fast FTL)

STAFF는 제안하는 FTL 알고리즘으로서 디자인 목표가 우수한 성능을 내면서도 동시에 적은 양의 메모리를 사용하여 실제 임베디드 시스템에 적용될 수 있도록 하는 것이다.

3.1 블록 상태 오토마타

STAFF는 플래시 메모리의 블록에 다음과 같이 상태를 정의한다.

- F 상태 블록: F 상태 블록은 그 블록이 삭제되어 데이타가 한번도 쓰여 지지 않은 블록을 의미한다.
- O 상태 블록: O 상태 블록은 블록에 있는 데이타가 더 이상 유효하지 않음을 나타낸다.
- M 상태 블록: M 상태 블록은 논리 섹터의 오프셋과 물리 섹터의 오프셋이 같은 블록으로 그 블록의 일부는 데이타가 쓰여져 있고, 나머지는 비어있는 블록이다.
- S 상태 블록: M 상태 블록과 같이 논리 섹터의 오프셋과 물리 섹터의 오프셋이 같은 블록이면서 블록의 모든 섹터에 데이타가 쓰여져 있는 블록이다. S 블록은 교체 합병(swap merging) 연산에 의하여 M 블록에서 전이된다. 구체적인 교체 합병 연산은 3.2절에서 다룬다.
- N 상태 블록: N 상태 블록은 논리 섹터의 오프셋과 물리 섹터의 오프셋이 다를 수 있는 블록으로 M 상태 블록에서 전이된다.

본 논문에서는 앞서 정의한 플래시 메모리 블록의 상태와 FTL 연산에 대하여 다음과 같이 오토마타를 정의한다. 오토마타 관련 기호는 [8]을 사용한다. 오토마타는 $(Q, \Sigma, \delta, q_0, F)$로 표현되고 각 기호의 의미는 다음과 같다.

- Q는 블록 상태의 집합으로 Q = {F, O, M, S, N}가 된다.
- Σ은 오토마타의 입력 기호로 본 논문에서는 FTL 연산 중의 여러 이벤트에 대응된다.
- δ는 전이 함수로 $Q \times \Sigma$에서 Q로의 함수이다.
- q_0는 시작 상태로 F 블록에서 시작한다.
- F는 종료 상태의 집합이다.

그림 5는 블록 상태 오토마타를 나타낸다. 시작 상태는 F 블록 상태가 되는데 첫 번째 쓰기 요청에 의하여 그 F 블록은 M 블록으로 전이된다. M 블록은 FTL 연산에 의하여 S블록이나 N블록으로 전이된다. S블록과 N블록은 이벤트 e4 와 e5에 의하여 O블록으로 전이되고, O블록은 이벤트 e6에 의하여 F 상태로 변경된다. 구체적인 이벤트는 3.2절에서 자세히 설명된다.

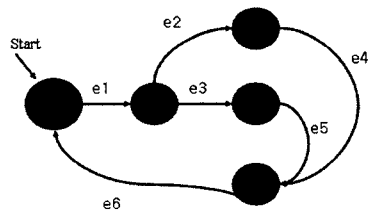


그림 5 블록 상태 오토마타

3.2 FTL 연산

기본 FTL 연산의 읽기와 쓰기 연산이다. 이 절에서는 STAFF의 읽기와 쓰기 알고리즘을 다룬다.

3.2.1 쓰기 알고리즘

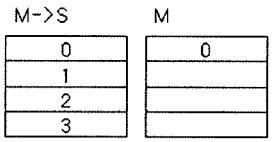
알고리즘 1은 STAFF의 쓰기 알고리즘을 보인다. 알고리즘의 입력은 논리 섹터 번호와 실제 쓸 데이터이다.

알고리즘 1 쓰기 알고리즘

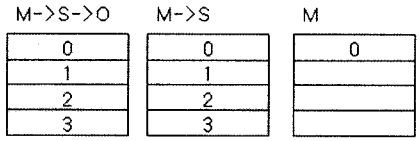
```
1:  입력: 논리 섹터 번호 (lsn), 쓰여질 데이타
2:  출력: 없음
3:  Procedure FTL_write (lsn,data)
4:  if 합병 연산이 필요 then
5:      합병 연산 수행;
6:  end if
7:  if lsn에 대한 논리 블록이 N 혹은 M 블록을
        가짐 then
```

```
8:      if 해당 섹터가 비어 있음 then
9:          입력 데이타를 M이나 N 블록에 씀;
10:     else
11:         if 블록이 M 블록 then
12:             블록의 상태가 N 블록으로 바뀜;
13:         end if
14:         입력 데이타를 N 블록에 씀;
15:     end if
16:   else
17:     F 블록을 할당 받음;
18:     F 블록이 M 블록으로 변경됨;
19:     입력 데이타를 M 블록에 씀;
20: end if
```

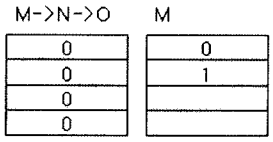
알고리즘은 먼저 해당 쓰기 연산이 합병 연산을 필요로 하는지 검사한다. STAFF의 합병 연산은 두 가지 종류가 있는데 치환 합병(swap merging)과 스마트 합병(smart merging)이다. 치환 합병 연산은 더 이상 빈 공간이 없는 M 블록에 데이타를 쓰려고 할 때 발생한다. 그림 6(a)는 치환 합병 연산을 보인다. 예에서는 한 블록이 4개의 섹터로 구성되는 것을 가정한다. M 블록은 S 블록으로 전이되고 한 논리 블록이 2개의 물리 블록에 사상된다. 그림 5에서 이벤트 e2가 치환 합병 연산에 해당한다.



(a) Isn sequence : 0, 1, 2, 3, 0



(b) Isn sequence : 0, 1, 2, 3, 0, 1, 2, 3, 0



(c) Isn sequence : 0, 0, 0, 0, 1

그림 6 여러 가지 쓰기 시나리오

스마트 합병 연산은 그림 6(c)에서 보여진다. 스마트 합병 연산은 더 이상의 비어 있는 공간이 없는 N 블록에 쓰기 연산이 요청될 때 발생한다. 스마트 합병 연산에서는 새로운 F 블록이 할당되고 N 블록의 유효한 데이타가 F 블록에 복사되고 그 블록은 M 블록이 된다. 그림 6(c)에서 논리 섹터 번호 0번에 대한 하나의 데이타만 유효함으로 그 데이타가 새로이 할당된 블록에 복사된다. 스마트 합병 연산은 그림 5의 이벤트 e5와 e1에 해당된다.

알고리즘 1의 7행에서 입력 논리 섹터 번호에 해당하는 논리 블록에 M 블록이나 N 블록을 가지지 않으면 해당 논리 섹터 번호에 해당하는 논리 블록은 한번도 쓰여져 있지 않음을 의미한다. 따라서 새로운 F 블록이 할당되어 입력 데이타는 F 블록에 쓰여지고 그 블록은 M 블록이 된다(17-19행).

만일 입력 논리 섹터에 대한 논리 블록이 M이나 N 블록을 가지면 쓰기 알고리즘은 해당 논리 섹터 번호와 오프셋이 일치하는 섹터가 비어있는지를 검사하게 된다. 만일 섹터가 비어있으면 논리 섹터에 해당하는 데이타가 쓰여진다. 그렇지 않으면 데이타는 N 블록에 쓰여지게 된다.

STAFF의 쓰기 알고리즘에서는 하나의 논리 블록은 최대 두 개의 물리 블록에 쓰여 지게 된다. 따라서 더 이상의 공간이 없을 때는 한 블록이 O 블록으로 전이된다. 그림 6(b)가 이 과정을 나타낸다.

17행에서 더 이상 할당할 F블록이 없을 때는 합병 연산이 수행되고 삭제 연산이 필요하게 된다.

3.2.2 읽기 알고리즘

알고리즘 2는 STAFF의 읽기 알고리즘을 보인다. 알고리즘의 입력은 논리 섹터 번호와 읽은 데이타를 저장할 버퍼이다.

알고리즘 2 읽기 알고리즘

```
1:  입력: 논리 섹터 번호 (lsn), 데이타 버퍼
2:  출력: 없음
3:  Procedure FTL_read (lsn,data buffer)
4:  If lsn에 해당하는 논리 블록이 M 블록이나 N 블록을 가짐 then
5:    if 그 블록이 M 블록임 then
6:      if 해당 섹터에 데이타가 존재함 then
7:        M 블록에서 데이타를 읽음;
8:      else
9:        if 논리 블록이 S 블록을 가짐 then
10:         S 블록에서 데이타를 읽음;
11:       else
12:         "ERROR: 해당 논리 블록이 쓰여진 적이 없음";
13:       end if
14:     end if
15:   else {블록이 N 블록임}
16:    if 해당 섹터가 N 블록에 존재 then
17:      N 블록에서 데이타를 읽음;
18:    else
```

```
19:       if 해당 논리 블록이 S 블록을 가짐 then
20:         S 블록에서 데이타를 읽음;
21:       else
22:         "ERROR: 해당 논리 블록이 쓰여진 적이 없음";
23:       end if
24:     end if
25:   end if
26: else
27:   if 해당 논리 블록이 S 블록을 가짐 then
28:     S 블록에서 데이타를 읽음;
29:   else
30:     "ERROR: 해당 논리 블록이 쓰여진 적이 없음";
31:   end if
32: end if
```

읽혀질 데이타는 M, N 혹은 S 블록에 존재한다. 만일 한 논리 블록이 두 개의 물리 블록에 사상된다면 그 두 개의 물리 블록은 S와 M 블록 혹은 S와 N 블록이 될 것이다. 만일 해당 lsn의 데이타가 S와 N(또는 M) 블록에 동시에 존재하면 N(또는 M) 블록에 있는 데이타가 유효한 데이타이다. 해당 lsn에 해당하는 데이타가 N 또는 M 블록에 존재하지 않으면 S 블록에 있는 데이타를 리턴 하거나 S 블록이 존재하지 않으면 에러를 리턴 한다.

N 블록에서 데이타를 읽을 때는 논리 섹터 번호의 오프셋과 물리 섹터 번호의 오프셋이 다를 수도 있으므로 쓰기 알고리즘에 따라 유효한 데이타를 찾는 과정이 필요하다. 구체적인 알고리즘은 간단함으로 생략한다.

4. 평 가

4.1 비용 계산

앞서 언급한대로 FTL 알고리즘은 사상 정보를 위한 메모리 요구량과 플래시 I/O 성능으로 비교할 수 있다.

STAFF는 블록 사상 기법에 기반하기 때문에 2.2절에서 언급한 대로 사상 정보를 위한 메모리 요구량이 작다. 2.2절의 1:1 블록 사상 기법과 비교하면 STAFF는 1:1과 1:2 블록 사상 기법의 혼합이라고 할 수 있다. 또한 N 블록은 섹터 사상 정보를 필요로 한다.

플래시 I/O 성능에 있어서는 읽기 쓰기 시간은 다음 식으로 계산될 수 있다.

$$C_{read} = p_M T_r + p_N k_1 T_r + p_S T_r \ (where\ p_M + p_N + p_S = 1) \tag{3}$$

$$C_{write} = p_{first}[(T_f + T_w)] + (1 - p_{first})[p_{merge}\{T_m + p_{e_1} T_w + (1 - p_{e_1})(k_2 T_r + T_w)\} + (1 - p_{merge})\{p_{e_2}(T_r + T_w) + (1 - p_{e_2})(k_3 T_r + T_w) + T_r + p_{MN} T_w\}] \tag{4}$$

여기서 $1 \le k_1, k_2, k_3 \le n$을 만족한다. 여기서, n은 한 블록에 존재하는 섹터의 수를 의미한다. 식 (3)에서 p_M, p_N, p_S은 각각 데이타가 M, N, S 블록에 저장될 확률을 의미한다.

식 (4)에서 p_{first}는 입력 논리 블록에 대하여 쓰기 연산이 첫 번째로 일어났을 확률을 의미한다. p_{merge}는 쓰기 연산이 합병 연산을 필요로 할 확률을 의미한다. p_{e_1}과 p_{e_2}은 각각 합병 연산을 동반하고 혹은 동반하지 않고 입력 논리 섹터 번호에 대한 쓰기 연산이 플래시 메모리의 섹터 오프셋이 일치하는 위치에 쓰여질 확률을 의미한다. T_f는 F 블록을 할당하기 위해 필요한 시간을 의미한다. 마지막으로 p_{MN}는 쓰기 연산이 M 블록을 N 블록으로 전이시킬 확률을 의미한다. 쓰기 연산이 M 블록을 N 블록으로 전이시킬 때는 상태를 표시하기 위하여 한번의 플래시 쓰기 연산이 더 필요하게 된다.

비용 함수는 M이나 S 블록에 대한 읽기 쓰기 연산보다 N 블록에 대한 읽기 쓰기 연산이 더 많은 읽기 연산을 필요로 함을 알 수 있다. 하지만 플래시 메모리에 대한 읽기 연산은 쓰기나 지우기 연산에 비하여 매우 작은 시간이 드는 연산이다. 따라서 T_f나 T_m이 플래시 삭제 연산을 필요로 할 수 있기 때문에 전체 시스템 성능을 결정하는 주요한 요소가 된다. STAFF는 삭제 연산을 야기하는 T_m이나 T_f의 값을 최소로 할 수 있도록 디자인되었다.

4.2 실험 결과

그림 1의 시스템 구조에서 FTL 알고리즘을 구현하여서 비교하였다. 실제 플래시 메모리는 플래시 에뮬레이터[9]를 이용하였다.

비교 대상이 된 FTL 알고리즘은 Mitsubishi FTL [6], SSR[7]과 STAFF이다. Mitsubishi FTL은 2.2절에서 소개된 블록 사상 기법에 기반하고 있고, SSR은 혼합 사상에 기반한다. 섹터 사상 방법은 구현하지 않았는데 섹터 사상 방법은 많은 메모리를 요구하므로 실제 적용되기 힘들기 때문이다.

임베디드 시스템에서 FAT 파일 시스템이 많이 사용되고 있기 때문에 Symbian[10] 운영체제가 1MB의 파일 쓰기 요청에 대하여 블록 디바이스 드라이버 단으로 보내는 접근 패턴을 실험에 이용하였다. 이 접근 패턴은 실제 임베디드 응용의 패턴과 유사할 것이다.

그림 7은 총 수행 시간을 보인다. x축은 실험한 횟수를 의미하고, y축은 총 수행시간을 나타낸다. 처음에 플래시 메모리는 비어 있으며 실험 횟수가 증가할수록 플래시 메모리에 데이타가 쌓이게 된다. 실험 결과는 STAFF는 혼합 사상과 비슷한 성능을 보이고 블록 사상에 비하여서는 약 다섯 배의 성능 향상을 보인 것을

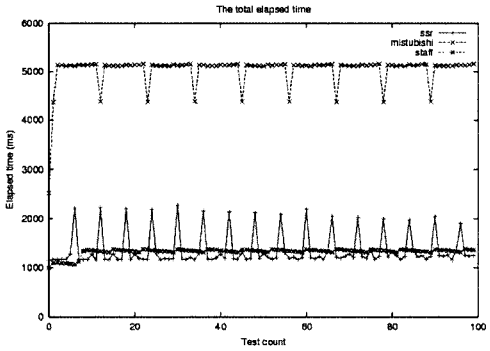


그림 7 총 수행 시간

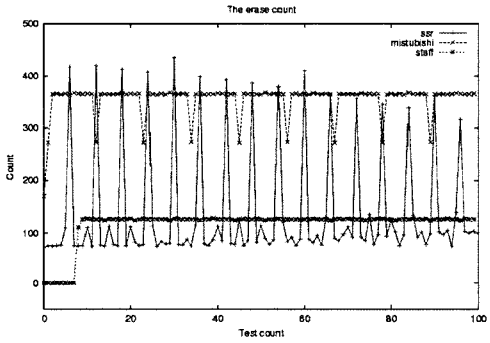


그림 8 삭제 횟수

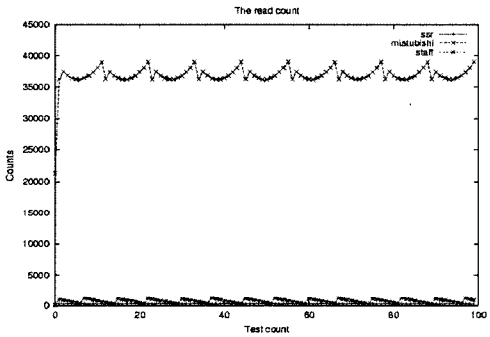


그림 9 읽기 횟수

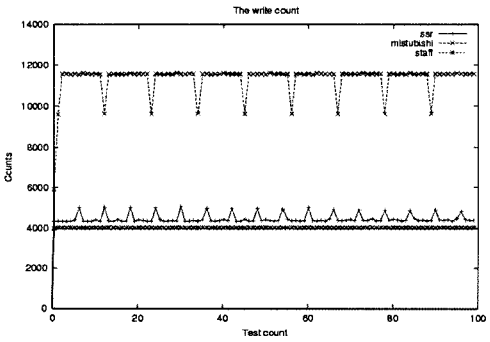


그림 10 쓰기 횟수

알 수 있다. STAFF는 평균적으로 혼합 사상에서 필요한 메모리 요구량의 약 10% 메모리를 필요로 함으로 임베디드 응용에 효율적으로 쓰일 수 있을 것이다.

그림 8은 삭제 횟수를 나타낸다. 삭제 횟수는 총 수행 시간과 비슷한 결과를 나타낸다. 이것은 삭제 횟수가 전체 시스템 성능의 주요 요소가 되기 때문이다. 특히 플래시 메모리가 비어있을 때는 STAFF가 더 좋은 성능을 나타내는데 이것은 O 블록을 이용하여 삭제 연산을 가능한 연기 시키기 때문이다. 만일 시스템이 병렬 처리를 허용하면 O 블록은 시스템의 다른 프로세스에 의하여 F 블록으로 전이될 수 있을 것이므로 성능 향상에 더욱 기여할 것이다. 플래시 메모리가 거의 다 차 있을 때도 STAFF는 일관된 성능을 보임을 알 수 있다.

그림 9와 그림 10은 읽기와 쓰기 횟수를 보인다. STAFF는 어느 정도의 읽기 횟수와 가장 좋은 쓰기 횟수를 보인다. [1]에 의하면 한 섹터에 대한 읽기와 쓰기 그리고 한 블록에 대한 삭제 연산의 비율이 1:7:63이다. 따라서 STAFF는 좋은 FTL 알고리즘임을 알 수 있다.

5. 결 론

본 논문에서 STAFF라 불리는 FTL 알고리즘을 제안하였다. STAFF의 주요 아이디어는 플래시 메모리의 삭제 단위인 블록에 대하여 상태 전이 개념을 도입하여 삭제 연산을 최소화하는데 있다. 즉, 입력 패턴에 따라서 블록의 상태가 전이되는데 이것이 삭제 연산을 최소화 하게 된다. 또한, 본 논문에서 고속의 치환, 스마트 합병 연산을 제안하였다. 기존 연구와 비교하여 비용 함수와 실험 결과에 의하면 STAFF는 적은 리소스를 소비하면서 좋은 성능을 보인다. 향후에는 실제 임베디드 응용에서 실험하여 결과를 분석하고 실제 워크로드에서 알고리즘을 최적화할 것이다.

참 고 문 헌

[1] Samsung Electronics, "Nand flash memory & smartmedia," data book, 2002.

[2] Amir Ban, "Flash file system," 1995, United States Patent, no. 5,404,485.

[3] Amir Ban, "Flash file system optimized for page-mode flash technologies," 1999, United States Patent, no. 5,937,425.

[4] Petro Estakhri and Berhanu Iman, "Moving sequential sectors within a block of information in a flash memory mass storage architecture," 1999, United States Patent, no. 5,930,815.

[5] Jesung Kim, Jong~Min Kim, Sam~H. Noh, San g~Lyul Min, and Yookun Cho, "A space-efficient flash translation layer for compactflash systems," IEEE Transactions on Consumer Electronics,

48(2), 2002.
[6] Takayuki Shinohara, "Flash memory card with block memory address arrangement," 1999, United States Patent, no. 5,905,993.
[7] Bum soo Kim and Gui young Lee, "Method of driving remapping in flash memory and flash memory architecture suitable therefore," 2002, United States Patent, no. 6,381,176.
[8] John~E. Hopcroft and Jeffrey~D. Ullman, "Introduction to automata theory, languages, and computation," Addison-Wesley Publishing Company, 1979.
[9] Sunghwan Bae, "SONA Programmer's guide," Technical report, Samsung Electronics, Co., Ltd., 2003.
[10] Symbian, http://www.symbian.com, 2004.

정 태 선

1995년 2월 KAIST 전산학과 학사(B.S.). 1997년 2월 서울대학교 전산과학과 석사(M.S.). 2002년 8월 서울대학교 전기컴퓨터공학부 박사(Ph.D.). 2002년 3월~2004년 2월 삼성전자 소프트웨어센터 책임연구원. 2004년 3월~2005년 8월 명지대학교 컴퓨터소프트웨어학과 조교수. 2005년 9월~현재 아주대학교 정보및컴퓨터공학부 조교수

박 형 석

1996년 2월 부산외국어대 컴퓨터공학 학사(B.S.). 1996년 3월~1997년 6월 Fuji Xerox Korea SEC소속 연구원. 1997년 7월~2000년 8월 EPSON KOREA 기술개발실 대리. 2000년 9월~현재 삼성전자 소프트웨어센터 책임연구원