재할당 블록을 이용한 플래시 메모리를 위한 효율적인 공간 관리 기법

(EAST: An Efficient and Advanced Space-management Technique for Flash Memory using Reallocation Blocks)

권 세 진 † 정 태 선 ††
(Se-Jin Kwon) (Tae-Sun Chung)

요 약 플래시 메모리는 전원이 끊기더라도 정보를 유지할 수 있는 비 휘발성 메모리로써 빠른 접근 속도, 저 전력 소비, 간편한 휴대성 등의 장점을 가진다. 플래시 메모리는 다른 메모리와 달리 "쓰기 전 지우기"(erase before write) 성질과 제한된 수의 지우기 연산을 수행할 수 없는 성질을 지닌다. 이와 같은 하드웨어 특성들로 인해 소프트웨어인 플래시 변환 계층(FTL: flash translation layer)을 필요로 한다. FTL은 파일 시스템의 논리주소를 플래시 메모리의 물리주소로 바꾸어주는 소프트웨어로써 FTL의 알고리즘으로 인해 플래시 메모리의 성능, 마모도 등이 좌우된다. 이 논문에서는 새로운 FTL의 알고리즘인 EAST를 제안한다. EAST는 재할당 블록(reallocation block)을 이용한 효율적인 공간 관리 기법으로 로그 블록의 개수를 최적화 시키고, 블록 상태를 사용한 사상 기법을 사용하며, 플래시 메모리의 공간을 효율적으로 관리한다. EAST는 특히 플래시 메모리의 용량이 크고 사용하는 용량이 작을 경우 FAST보다 더 나은 성능을 보인다.

키워드 : 플래시 메모리, 임베디드 시스템, 파일 시스템, FTL

Abstract Flash memory offers attractive features, such as non-volatile, shock resistance, fast access, and low power consumption for data storage. However, it has one main drawback of requiring an erase before updating the contents. Furthermore, flash memory can only be erased limited number of times. To overcome limitations, flash memory needs a software layer called flash translation layer (FTL). The basic function of FTL is to translate the logical address from the file system like file allocation table (FAT) to the physical address in flash memory. In this paper, a new FTL algorithm called an efficient and advanced space-management technique (EAST) is proposed. EAST improves the performance by optimizing the number of log blocks, by applying the state transition, and by using reallocation blocks. The results of experiments show that EAST outperforms FAST, which is an enhanced log block scheme, particularly when the usage of flash memory is not full.

Key words : flash memory, embedded system, file system, FTL

· 본 연구는 방위사업청과 국방과학연구소의 지원(UD060048AD)과 정보통신부 및 정보통신연구진흥원의 IT신성장동력핵심기술개발사업(2006-S-040-01)의 지원으로 수행되었습니다.

† 학생회원 : 아주대학교 정보 및 컴퓨터공학부
sejin1109@empal.com
†† 정 회 원 : 아주대학교 정보 및 컴퓨터공학과 교수
tschung@ajou.ac.kr

논문접수 : 2007년 9월 6일
심사완료 : 2007년 11월 29일

정보과학회논문지 : 컴퓨팅의 실제 및 레터 제13권 제7호(2007.12)

1. 서 론

플래시 메모리는 전원이 끊기더라도 정보를 유지할 수 있는 비휘발성 메모리로써 빠른 접근 속도, 저 전력 소비, 간편한 휴대성 등의 장점을 가진다. 이런 장점에도 불구하고 플래시 메모리는 "쓰기 전 지우기"라는 성질을 지니고 있다. "쓰기 전 지우기"성질이란 하드디스크와는 달리 데이타를 갱신하기 전에 지우기 연산을 먼저 수행해야 한다는 것이다. 최근 삼성 전자 낸드 플래시 메모리(Nand flash memory)[1]에 의하면, 읽기 연산과 쓰기 연산은 모두 섹터 단위이며 지우기 연산은 블록 단위이다. 읽기 연산은 15μs가 소요되고 쓰기 연산

은 200㎲이 소요된다. 지우기 연산은 블록(block) 단위로써 2ms이 걸리기 때문에 많은 지우기 연산은 성능에 지대한 영향을 줄 수 있다. 또한 한 블록에 대한 지우기 연산이 일정 횟수를 넘어가면 그 블록이 손상되어 플래시 메모리가 손상될 경우가 있다. 이를 위해 블록의 지우는 횟수를 평준화 시키는 마모도 평준화 기법(wear-leveling technique)이 요구된다. 이와 같은 성질들을 고려하여 플래시 메모리의 성능을 극대화시키는 것이 FTL의 역할이다. 기존의 FTL로는 기존의 파일 시스템을 확장하여 플래시 메모리의 기능을 추가한 기법인 [2-4]이 있고, 플래시 메모리를 위한 디바이스 드라이버를 제공하는 기법은 [5-10, 12-13]이 있다. 파일 시스템에 플래시 메모리를 위한 모듈을 첨가하는 기법은 최근 파일 시스템과의 호환성 문제로 인해 플래시 메모리를 위한 디바이스 드라이버를 제공하는 기법이 사용되는 추세이다.

플래시 메모리의 성능을 극대화시키기 위해서는 "쓰기 전 지우기"성질이 중요하다. 읽기와 쓰기의 횟수를 줄이는 것도 중요하지만 지우기 연산은 2ms이나 되기 때문에 지우기 연산을 줄이는 것이 플래시 메모리 성능 향상에 가장 중요한 요소이다. 지우기 연산의 횟수 외에도 플래시 메모리에서 중요한 사항은 RAM의 사용량이다. RAM은 휘발성 고가격대의 메모리로써 각 FTL 알고리즘에 필요한 사상 정보 테이블을 저장하고 사용한다. 사상 정보 테이블이란 논리 주소와 물리 주소를 섹터 또는 블록 단위로 사상한 정보를 담는 테이블이다.

따라서 플래시 메모리에 요구되는 FTL의 최종목표는 최소한의 RAM을 사용하면서 최소한의 연산 시간을 소요하여 연산을 수행하는 것이다. 본 논문에서 제안하는 EAST(Efficient and Advanced Space-management Technique)는 블록 사상 테이블과 비슷한 요구량을 사용하면서 기존의 알고리즘에 비해 고성능을 낸다는 특성을 지니고 있다.

본 논문의 2절에서는 플래시 메모리의 구성과 기존 연구를 분석하고, 3절에서는 EAST의 핵심 아이디어를 알아보며, 4절에서는 EAST의 알고리즘을 보고, 5절에서는 EAST와 FAST를 비교한 성능 평가 결과를 보이며, 마지막 6절에서는 결론을 맺는다.

2. 플래시 메모리의 구성과 기존 연구

2.1 플래시 메모리의 구성

플래시 메모리에는 한 블록의 크기와 한 섹터의 크기에 따라 소블록 플래시 메모리(small block flash memory)와 대블록 플래시 메모리(large block flash memory)로 나누어진다. 본 논문은 소블록 플래시 메모리를 대상으로 알고리즘이 설명됨을 가정한다. 하지만 본 논문의 아이디어는 블록의 크기와 한 섹터의 크기에 따른 영향을 받지 않기 때문에 대블록 플래시 메모리에도 적용될 수 있다.

삼성 전자 낸드 플래시 메모리[1]에 의하면, 읽기와 쓰기 연산은 섹터 단위이며, 지우기 연산은 블록단위이다. 한 블록이 32개의 섹터로 이루어져 있으며, 한 섹터의 크기는 528바이트이다. 528바이트 중 512바이트는 데이타를 적을 수 있는 영역이며, 나머지 16바이트는 섹터나 한 블록에 대한 정보(metadata)를 저장하는 여분 영역(spare area)이다. 데이타 영역과 여분 영역에는 부분 프로그래밍(partial programming)이 가능하다. 부분 프로그래밍은 섹터단위의 쓰기 연산이 아닌 바이트 단위의 쓰기 연산이다. 부분 프로그래밍의 횟수(NOP: number of partial programming)는 대개 데이타 영역에 2번, 여분 영역에는 3번이지만 이 횟수는 플래시 메모리마다 다를 수 있다.

파일 시스템에서 플래시 메모리의 논리 블록 수를 x개 있다고 간주한다면, 실제로 플래시 메모리는 y개의 물리 블록 수를 지닌다고 볼 수 있다. 이 때 사용된 y개의 물리 블록 수는 x개의 논리 블록 수보다 많거나 같다. 사용자에게 보여 지는 플래시 메모리의 용량은 한 섹터에서 데이타 영역인 512바이트만을 계산한 값이며, 블록의 숫자는 오직 논리 블록 수인 x만을 계산한 값이다.

2.2 기존연구

2.2.1 FAST

FAST[7]는 BAST[5]를 발전시킨 알고리즘으로 데이타 블록(data block), 프리 블록(free block), 로그 블록(log block)으로 이루어진 알고리즘이다. 데이타 블록은 논리 섹터의 오프셋(offset)과 물리 섹터의 오프셋이 일치하는 위치에(in-place) 데이타를 쓰는 블록이다. 여기서 논리 섹터의 오프셋은 논리 섹터 번호를 한 개의 블록이 지니고 있는 섹터 수로 나눈 나머지 값을 의미하며, 물리 섹터의 오프셋은 블록의 첫 번째 섹터를 0번 섹터로 시작해서 몇 번 섹터인지를 의미한다. 프리 블록은 아무 데이타가 없는 빈 블록이다.

로그 블록은 논리 섹터의 오프셋과 물리 섹터의 오프셋이 일치하지 않는 위치에(out-of-place) 쓰기 연산을 수행하는 블록으로 데이타 블록의 섹터에 덮어쓰기(over-write)가 일어날 때 사용되는 블록이다. 여기서 덮어쓰기란 이미 데이타가 존재하는 섹터에 쓰기 연산이 일어나는 것을 의미한다. FAST에는 두 가지 종류의 로그 블록이 존재한다. 순차적인 덮어쓰기 연산을 처리하기 위한 순차 로그 블록(sequential log block)과 랜덤 로그 블록(random log block)이 존재한다. 순차 로그 블록에 쓰기 연산이 수행될 수 있는 조건은 논리 섹터의 오프셋이 0번인 섹터가 오거나 순차적인 쓰기 연산이

들어올 때만 가능하다. 만약 순차 로그 블록에 적힐 수 있는 조건에 벗어난 경우에는 랜덤 로그 블록의 첫 빈 섹터부터 차례로 쓰기 연산을 수행한다.

FAST를 수행하기 위해서는 데이타 블록에 대한 블록 사상 테이블과 랜덤 로그 블록들에 대한 섹터 사상 테이블을 필요로 한다. 블록 사상 테이블은 논리 주소와 물리 주소를 블록 단위로 사상해놓은 테이블이며, 섹터 사상 테이블은 논리 주소와 물리 주소를 섹터 단위로 사상해놓은 테이블이다.

그림 1은 FAST의 쓰기 연산을 보여주고 있다. 그림 1을 비롯한 앞으로의 그림들은 이해를 돕기 위해 한 개의 블록이 6개의 섹터로 이루어져 있다고 가정하며, -1은 더 이상 섹터가 유효성이 없다는 것을 의미한다. 즉 그 섹터에 대한 갱신이 다른 섹터에 되었다는 것을 의미한다. 쓰기 연산들은 FAST의 쓰기 연산을 설명하기 위해 만들어진 임의로 만들어진 유형이며, 연산 명령은 명령어, 논리 섹터 번호, 데이타로 이루어져 있다. w란 write 명령어를 의미하며, 숫자는 논리 섹터의 번호를 의미하며, Dx란 논리 섹터 x에 해당하는 data를 의미한다.

FAST의 쓰기 연산은 먼저 논리 섹터 번호를 한 블록이 지닌 섹터의 수로 나눈다. 나눈 결과의 몫은 논리 블록 번호를 의미하고, 나머지는 논리 섹터의 오프셋을 의미한다. 결정된 논리 블록 번호와 사상된 물리 블록은 블록 사상 테이블을 통해 결정한다. ①의 쓰기 연산을 예로 들면, 논리 섹터 2번을 한 블록이 지니고 있는 섹터 수인 6으로 나눈다. 이 때 몫은 0이며, 나머지 값은 2이다. 논리 블록 0번과 사상된 물리 블록 번호는 블록 사상 테이블에서 0번이라고 결정된다. 사상된 물리 블록이 결정되면, 그 물리 블록의 섹터 상태에 따라 쓰기 연산이 수행된다.

- ①, ③, ⑤- 논리 섹터와 물리 섹터의 오프셋이 일치하는 섹터에 데이타가 없으므로 쓰기 연산을 수행한다.
- ②, ④- 논리 섹터와 물리 섹터의 오프셋이 일치하는 섹터에 데이타가 존재하므로 순차 로그 블록 또는 랜덤 로그 블록에 쓰기 연산을 수행한다. ②, ④의 쓰기 연산은 순차 로그 블록에 쓰기 연산을 수행 할 수 있는 조건이 아니므로, 랜덤 로그 블록에 쓰기 연산을 수행하고 여분 영역에는 논리 섹터 번호를 추가적으로 적는다.
- ⑥- 논리 섹터와 물리 섹터의 오프셋이 일치하는 섹터에 데이타가 존재한다. 논리 섹터 12는 오프셋이 0이므로 순차 로그 블록에 쓰기 연산을 수행한다.

FAST는 다른 알고리즘과 달리 로그 블록을 사용하기 때문에 합병 연산(merge operation)이 존재한다. 합병 연산은 랜덤 로그 블록이 꽉 찼을 경우에 발생한다. 합병 연산이란 유효한 섹터들을 빈 블록에 쓰기 연산을 수행한 뒤 지우기 연산을 수행하는 것이다.

그림 2는 합병 연산의 예제이다. 이 그림에서는 랜덤 로그 블록이 물리 블록 n+1번과 n+2번이라고 가정한다. 물리 블록 0번, n+1, n+2는 모두 데이타로 찼으며, -1은 유효하지 않는 섹터들이라는 것을 의미한다. 물리 블록 n+1번과 n+2번에 더 이상 빈 공간이 없으므로 합병 연산을 수행한다. 물리 블록 n+1번과 n+2번에 존재하는 유효한 논리 섹터들의 번호를 살펴보면, 논리 블록 0번에 관한 섹터들과 논리 블록 2번에 관한 섹터들로 이루어져 있다. 그림 2는 논리 블록 0번과 사상된 물리 블록과 랜덤 블록들과의 합병 연산을 나타낸 것이다.

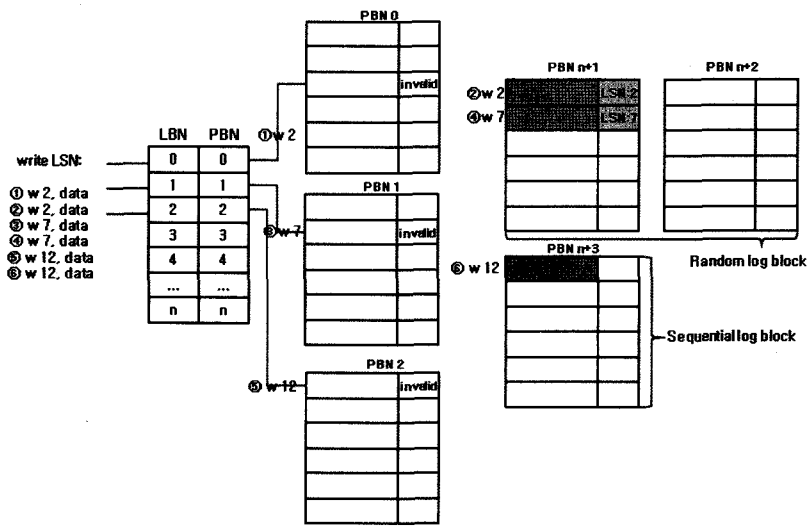

그림 1 FAST 쓰기 연산

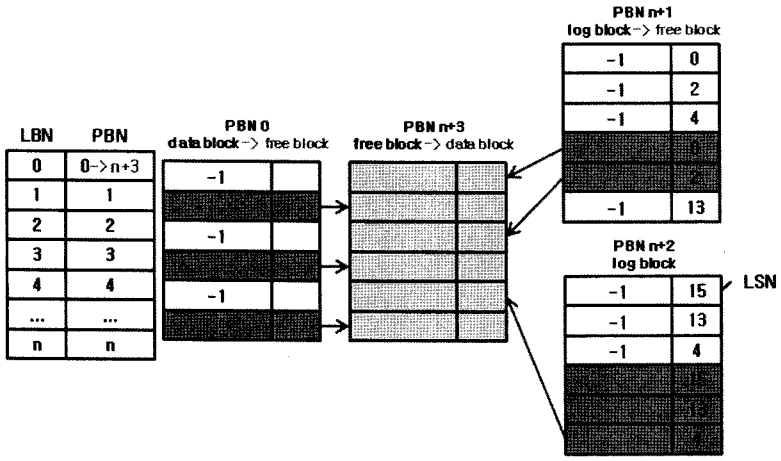

그림 2 FAST 합병 연산

3. EAST의 핵심 아이디어

EAST는 기존 연구인 FAST의 약점을 보안하고 보다 나은 성능을 위해 제안된 FTL 알고리즘이다. EAST는 동적 할당(dynamic allocation)으로 블록 상태를 기입하는 블록 사상 기법을 사용하면서, 하나의 논리 블록 당 네 개의 물리 블록을 가질 수 있는 알고리즘이다. 여기서 동적 할당이란 논리 주소와 물리 주소를 사상한 테이블을 처음부터 만들지 않고, 쓰기 연산이 수행되면서 동적으로 테이블을 만든다는 것이다.

3.1 로그 블록 개수의 최적화

FAST는 읽기 연산을 수행 할 때 데이타 블록의 섹터가 유효성이 없다면, 랜덤 로그 블록에서 섹터를 검색한다. 랜덤 로그 블록에서 섹터를 검색한다는 것은 랜덤 로그 블록의 끝에서부터 거꾸로 섹터의 여분 영역에 논리 섹터 숫자가 일치하는 것이 나타날 때까지 각 섹터에 대해서 읽기 연산을 수행하는 것이다.

검색을 위한 읽기 연산들을 배제하려면, 랜덤 로그 블록들에 대한 섹터 사상 테이블을 가지면 된다. 하지만 랜덤 블록의 숫자가 많아질수록 RAM의 용량이 커진다는 단점이 있기 때문에 오직 성능만을 위해 랜덤 블록의 수를 늘릴 수 없다.

EAST는 섹터 사상 테이블을 사용할 여유가 없는 환경도 고려하여, 읽기 연산을 위한 검색을 하더라도 지우기 연산의 소요 시간을 넘기지 않기 위해 다음 공식에 따라 로그 블록의 개수를 결정한다.

• 로그 블록의 개수 = 지우기 연산 시간/읽기 연산 시간/블록 당 섹터 수 (1)

지우기 연산 시간을 읽기 연산 시간으로 나눈 결과의 몫은 한 번의 지우기 연산이 소요되는 시간에 읽기 연산이 수행될 수 있는 섹터의 수를 의미한다. 섹터의 수를 한 개의 블록이 지닌 섹터의 수로 나누면, 로그 블록의 수가 나온다. 이 로그 블록의 수는 한 개의 논리 블록이 지닐 수 있는 물리 블록의 수를 의미한다.

3.2 블록 상태를 사용한 사상 기법

5절 성능 평가에서 실험된 표 1의 리눅스, 심비안, 코닥, 니콘과 같은 쓰기 연산 유형들을 FAST에서 수행하면, 덮어쓰기가 있는 블록들은 오직 33%의 섹터들만 채운 채로 지우기 연산을 수행해야 하는 경우가 많다. EAST는 연산에 관여하지 않는 섹터들의 블록 상태를 여분 영역에 기입하는 방법으로 성능을 높인다. EAST의 사상 기법과는 다르지만, 블록의 상태를 여분 영역에 저장하는 것은 이미 STAFF[6]에서 제안된 바가 있다.

EAST는 읽기 연산이 빠른 in-place기법의 장점과 블록을 효율적으로 채우는 out-of-place기법의 장점을 모두 지니기 위해 in-place/out-of-place 플래그를 여분 영역에 저장한다. in-place기법은 논리 섹터의 오프셋과 물리 섹터의 오프셋이 일치하는 위치에 데이타를 적는 것이며, out-of-place기법은 일치하지 않는 위치에 데이타를 적는 것이다. 모든 블록은 in-place기법으로 시작하여 쓰기 연산을 수행하다가 이미 데이타가 존재하는 섹터에 쓰기 연산이 수행될 때 플래그를 out-of-place로 바꾸고, 첫 빈 섹터부터 쓰기 연산을 수행한다.

그림 3은 EAST의 사상 기법을 보여준다. 먼저 논리 섹터 번호를 한 블록이 지니고 있는 섹터의 수로 나눈다. 나눈 결과의 몫은 논리 블록 번호를 의미하고, 나머지는 논리 섹터의 오프셋을 의미한다. 몫으로 결정된 논리 블록 번호와 사상된 물리 블록을 블록 사상 테이블을 통해 결정한다. 해당 물리 블록의 상태와 섹터의 상태에 따라 다음과 같이 다르게 쓰기 연산이 수행된다.

• ①, ⑤, ⑥- 현재 물리 블록이 어떤 기법을 사용하는

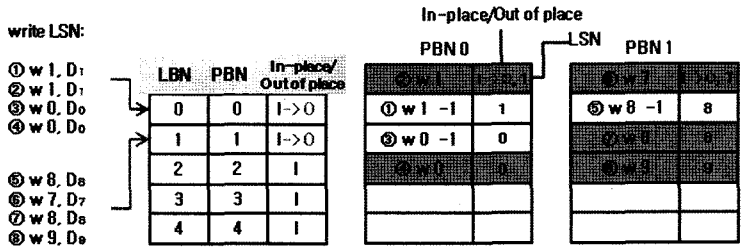

그림 3 블록 상태를 사용한 사상 기법

지 블록 사상 테이블을 통해 확인한다. 이 시점에서는 in-place기법이므로, 논리 섹터의 오프셋과 일치하는 물리 섹터에 데이타가 존재하는지 확인한다. 데이타가 존재하지 않으므로 쓰기 연산이 in-place기법으로 데이타 블록에 수행된다.

- ②, ⑦- 논리 섹터의 오프셋에 해당하는 물리 섹터에 데이타가 존재하는지 확인한다. ①, ⑤로 인해 이미 물리 섹터에 데이타가 존재하므로, 물리 블록의 첫 섹터의 여분 영역에 out-of-place블록이라는 O플래그로 표시를 한다. 블록 사상 테이블에도 O표시로 갱신한다. 그리고는 첫 섹터부터 빈 섹터를 검색하여 쓰기 연산을 수행한다.
- ③, ④, ⑧- 현재 물리 블록이 어떤 기법을 사용하는지 블록 사상 테이블을 통해 확인한다. out-of-place기법으로 첫 빈 섹터에 쓰기 연산을 수행한다.

그림 3과 같이 in-place/out-of-place플래그를 각 블록의 첫 번째 섹터에 저장을 한다고 가정한다면, 첫 번째 섹터의 여분 영역에 in-place/out-of-place플래그, LBN, LSN이 존재 할 수 있다. 세 가지 요소 중 LSN은 모든 섹터가 가지고 있고 나머지 두 요소인 LBN과 in-place/out-of-place플래그는 한 블록에 관한 정보이므로 한 섹터만 가지면 된다.

여분 영역에 부분 프로그래밍이 가능한 횟수는 기계마다 다르기 때문에 프로그래밍 또한 틀려진다. 여분 영역에 LBN과 in-place/out-of-place플래그를 부분 프로그래밍의 횟수가 충분치 않아 한 섹터의 여분 영역에 포함하는 것이 불가능하다면, LBN과 in-place/out-of-place플래그를 각각 두 개의 섹터로 나눌 수도 있다.

3.3 플래시 메모리의 효율적인 활용

플래시 메모리의 용량이 저용량일 때는 플래시 메모리 전체가 데이타로 꽉 차는 경우가 많을 수 있지만, 플래시 메모리의 용량이 고용량일 경우에는 사용자가 실질적으로 사용하는 용량이 전체를 다 채우지 않는 경우가 많다. 예를 들어, USB(Universal Serial Bus) 메모리가 4GB(Giga Byte)일 경우, 4GB를 꽉 채우기 보다는 대개 3.8GB나 3.9GB와 같이 4GB에 인접하게 사용하는 경우가 대부분이라는 것이다. 사용되지 않는 남은 데이타 블록의 공간을 효율적으로 사용하면 성능을 향상시킬 수 있다.

그림 4의 플래시 메모리는 논리 섹터 0번에서 11번까

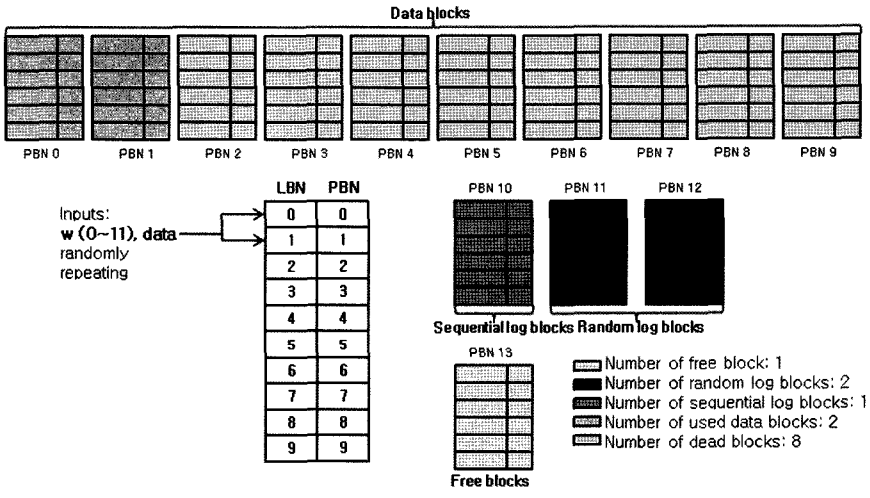

그림 4 FAST의 플래시 메모리 공간사용

지의 쓰기 연산만을 필요로 하는 저용량의 파일을 저장한다고 가정한다. 먼저 쓰기 연산의 논리 섹터를 한 개의 블록이 지닌 섹터 수로 나눈 결과의 몫으로 논리 블록 번호를 구한다. 논리 섹터가 0번에서 11번까지의 쓰기 연산만 사용됨으로 6으로 나눈 몫은 0이나 1이 된다. 따라서 논리 블록 0번과 1번에 사상된 물리 블록 0번과 1번을 데이타 블록으로 사용한다.

그림 4는 총 14개의 블록을 가지고 있지만, 실제적으로 사용하는 물리 블록은 0번, 1번, 10번, 11번, 12번이다. 물리 블록 0번과 1번은 데이타 블록으로 사용되고 물리 블록 10번, 11번, 12번은 데이타 블록 0번과 1번의 로그 블록으로 사용된다. 물리 블록 2번에서 9번까지는 아무 연산도 수행되지 않는 빈 데이타 블록으로 계속 남아있다. EAST에서는 2번에서 9번까지의 물리 블록과 같이 쓰기 연산의 유형을 수행 할 때 아무 관여를 하지 않는 빈 데이타 블록들을 재할당 블록(reallocation block)이라고 명시하겠다.

EAST는 재할당 블록들을 활용하여 성능을 향상시키고 있다. 재할당 블록들을 활용하기 위해서는 블록 사상 테이블을 동적으로 할당하고 로그 블록의 개수를 정하지 말아야 한다. 블록 사상 테이블을 동적으로 할당한다는 것은 블록 사상 테이블에 처음부터 논리 블록과 물리 블록을 사상해놓지 않고 쓰기 연산이 들어오는 순서대로 논리 블록과 물리 블록을 사상한다는 것을 의미한다. 로그 블록의 수를 정하지 않고 사용한다는 것은 처음부터 순차 로그 블록의 수와 랜덤 로그 블록의 수를 정하지 않고 재할당 블록의 수만큼 랜덤 로그 블록의 수로 최대한 사용한다는 의미이다.

동적 할당을 하는 블록 사상 테이블을 사용하고 로그 블록의 개수를 미리 정하지 않는 것은 FAST에도 이행시킬 수 있다. 그림 5, 6, 7은 FAST와 플래시 메모리 공간의 효율적 활용을 위해 변형시킨 FAST를 비교한

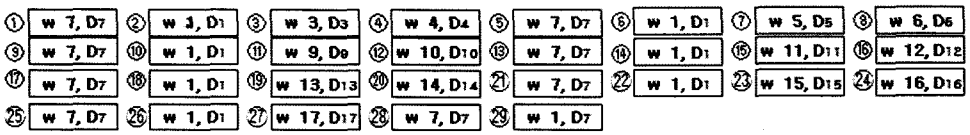

그림 5 쓰기 연산의 유형

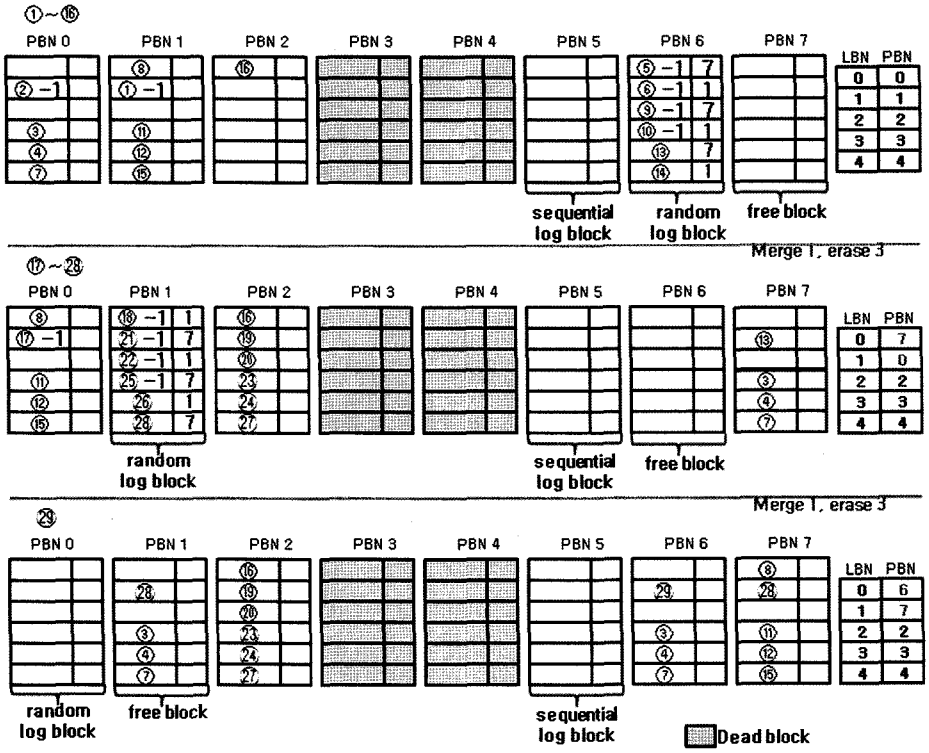

그림 6 FAST의 예시

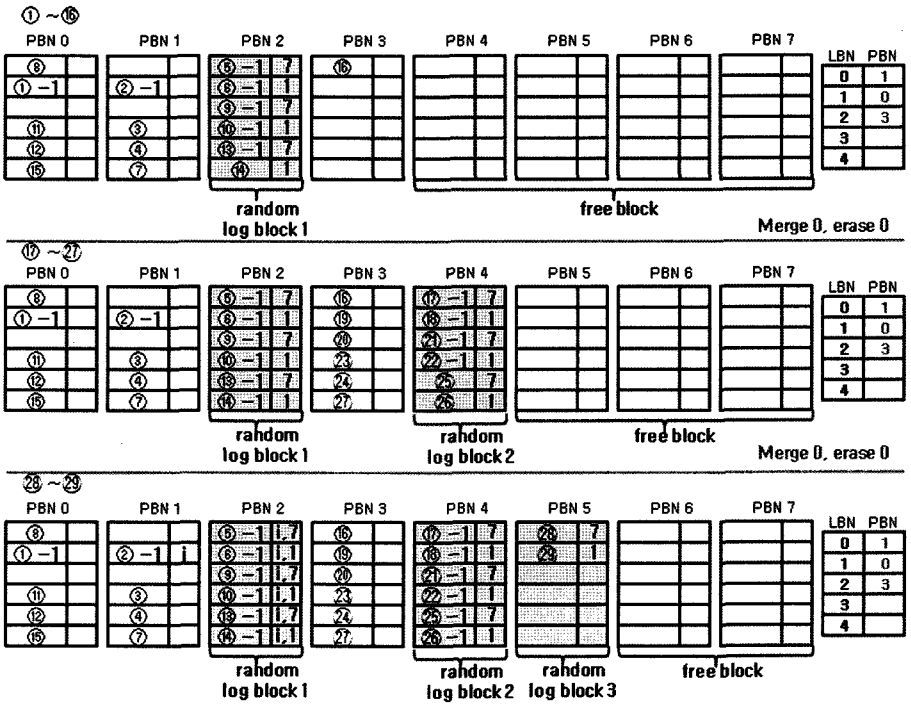

그림 7 변형된 FAST의 예시

결과이다. 그림 5는 용량이 작은 임의로 만들어진 쓰기 연산의 유형이다. 그림 6은 그림 5를 수행한 FAST의 결과이다. 그림 7은 동적 할당하는 블록 사상 테이블을 사용하면서 로그 블록의 수를 미리 정하지 않는 변형된 FAST 알고리즘을 보여준다. 그림 6, 7은 이해의 편의를 위해 한 블록은 6개의 섹터로 이루어져 있고, 플래시 메모리의 용량이 데이타 블록 5개라고 가정한다.

FAST 알고리즘을 사용한 그림 6은 데이타 블록이 5개, 순차 로그 블록이 1개, 랜덤 로그 블록이 1개, 프리 블록이 1개로 구성된다. ⑰, ㉙번째의 쓰기 연산이 수행되면서 두 번의 합병 연산으로 총 6번의 지우기 연산이 일어나며, 재할당 블록은 물리 블록 3번과 4번이 존재한다.

그림 7은 플래시 메모리의 공간을 효율적으로 사용하기 위한 변형된 FAST 알고리즘이다. 처음 상태에서는 모든 블록을 프리 블록으로 간주하기 때문에 블록 사상 테이블에서는 사상 정보가 없다. ①의 쓰기 연산에서는 논리 섹터 7번을 한 블록이 가지는 6개의 섹터로 나누어 몫이 1이므로, 논리 블록 1번과 새로운 물리 블록 0번을 사상하고 in-place기법으로 쓰기 연산을 수행한다. ⑰의 쓰기 연산에서는 논리 섹터 7번을 6으로 나눈 몫이 1이고 나머지가 1이므로 논리 블록 1번과 사상된 물리 블록 0번을 찾아낸다. 하지만 물리 블록 0번의 1번 섹터에 이미 데이타가 존재한다. 따라서 랜덤 로그 블록으로 할당되어 있는 물리 블록 2번의 빈 섹터를 찾아보지만 꽉 차있으므로, 프리 블록 중 새로운 랜덤 로그 블록을 할당하여 쓰기 연산을 수행한다. 이와 같은 연산의 결과로 지우기 연산은 0번 일어난다.

이와 같이 FAST도 로그 블록의 수를 정하지 않고 재할당 블록을 이용하여 성능을 더 발전시킬 수 있지만 FAST를 변형시키는 것은 다음의 단점을 가진다.

- 변형된 FAST는 재할당 블록 수만큼을 랜덤 로그 블록으로 사용한다. 따라서 랜덤 로그 블록에 대한 섹터 사상 테이블의 크기가 지나치게 커진다.
- RAM의 요구량으로 인해 랜덤 로그 블록에 대한 섹터 사상 테이블이 없을 경우, 3.1절에서 지적한 읽기 연산이 지우기 연산보다 큰 대가를 치르는 단점이 나타난다.

따라서 FAST 알고리즘을 변형시키는 것은 비효율적인 결과를 초래함으로 FAST로는 재할당 블록들을 효율적으로 사용할 수 없다. 따라서 이와 같은 단점이 나타나지 않는 4절의 EAST를 제안한다.

4. EAST 알고리즘

4.1 개요

EAST는 3.1절, 3.2절, 3.3절에서 제안한 핵심 아이디어들을 바탕으로 한다.

- EAST는 3.3절에 의해 블록 사상 테이블을 동적으로 할당하고, 사용할 수 있는 로그 블록의 수를 정하지 않는다.
- 한 물리 블록에 존재하는 모든 섹터들을 효율적으로 사용하기 위해 블록 상태를 사용하는 사상 기법을 3.2절과 같이 사용한다.
- 꽉 찬 데이타 블록에 추가적으로 사상될 수 있는 로그 블록의 수는 3.1절로 인해 최대 3개이다.

4.2 쓰기 연산

알고리즘 1은 EAST의 쓰기 연산 알고리즘을 보여준다.

```
알고리즘 1 쓰기 연산 알고리즘
1:  입력: 논리 섹터 번호(LSN), 데이타
2:  Procedure FTL_write(lsn, data)
3:  LBN = LSN/블록 당 섹터수;
4:  LBN과 사상된 물리 블록 번호들과 블록 상태 확인(in-place/out-of-place);
5:  if 물리 블록이 in-place기법을 사용함 then
6:    if 해당 섹터가 비어 있음 then
7:      해당 섹터에 씀;
8:    else
9:      out-of-place라는 표시를 물리 블록의 여분 영역과 RAM에 해줌 ;
10:     if 빈 섹터가 존재함 then
11:       빈 섹터에 데이타 쓰고 LSN을 여분 영역에 씀;
12:     else
14:       if 프리 블록이 2개 이상이다 then
15:         프리 블록 중 로그 블록을 ECN을 고려하여 통해 할당;
16:       else
17:         합병 연산 수행;
18:       end if
19:     end if
20:   else if out-of-place기법 && 논리 블록과 사상된 물리 블록이 4개이다 then
21:     if 빈 섹터가 존재함 then
22:       빈 섹터에 데이타와 쓰고 LSN을 여분 영역에 씀;
23:     else
24:       합병 연산 수행;
25:     end if
26:   else if out-of-place기법 && 논리 블록과 사상된 물리 블록이 4개 미만이다 then
27:     if 빈 섹터가 존재함 then
28:       빈 섹터에 데이타와 쓰고 LSN을 여분 영역에 씀;
29:     else
30:       if 프리 블록이 2개 이상이다 then
31:         프리 블록 중 로그 블록을 ECN이 가장 작은 것 할당;
32:       else
33:         합병 연산 수행;
34:       end if
35:     end if
36:   end if
```

그림 8은 알고리즘 1을 이용해서 나타낸 예제이다. 그림 8의 플래시 메모리는 그림 6, 7과 동일한 플래시 메모리를 대상으로 한다. EAST는 3.1절로 인해 한 개의 논리 블록 당 네 개의 물리 블록을 지닐 수 있지만, 그림 8은 이해를 돕기 위한 플래시 메모리의 축소판이므로, 한 개의 논리 블록 당 두 개의 물리 블록만을 지닐 수 있다고 가정한다.

쓰기 연산이 한 번도 수행되지 않은 플래시 메모리의 물리 블록들은 그림 8의 시작점에 나타난 블록들의 모습과 같이 모두 빈 블록이다. ①의 쓰기 연산에서는 먼저 논리 섹터 번호 7을 한 개의 블록에 지닌 섹터수인 6으로 나눈다. 몫은 1이며 나머지 값은 1이다. 논리 블록 1번과 사상된 물리 블록이 없으므로 새로운 물리 블록을 사상한다. 이 때 EAST에서는 지우기 연산이 가장 작은 블록을 선택한다. 지우기 연산의 횟수는 블록의 한 섹터 영역에 저장될 수 있다. 이 경우는 물리 블록 0번이 선택되었으며, ①의 논리 섹터 오프셋과 동일한 물리 블록 0번의 오프셋에 데이타가 없으므로 쓰기 연산을 수행한다.

⑤번의 시점에서는 논리 섹터 7번을 한 개의 블록이 지닌 섹터수인 6으로 나누어서 몫이 1이며, 나머지가 1이다. 논리 블록 1번과 사상된 물리 블록 0번의 1번 섹터에 데이타가 이미 존재한다. 따라서 논리 블록 1번과 사상된 물리 블록 0번을 out-of-place기법으로 사용한다는 것을 표시하고, 첫 빈 섹터에 쓰기 연산을 수행한다.

⑬번의 시점은 물리 블록 0번이 out-of-place기법으로 모든 섹터가 데이타로 차있다. 논리 블록 1번과 사상된 물리 블록의 수가 1개이므로 가장 지우기 연산이 작은 프리 블록을 선택하여 로그 블록으로 추가적으로 할당한다.

㉚번의 시점은 논리 블록 1번과 사상된 두 개의 물리 블록이 데이타로 꽉 찬 상태에서 쓰기 연산을 수행하는 경우이다. 그림 8에서는 한 논리 블록이 사상할 수 있는 물리 블록의 수는 최대 2개라고 가정했기 때문에 합병 연산을 수행한다. 합병 연산은 4.3절에서 자세히 설명된다.

4.3 합병 연산

EAST의 합병 연산은 다음 세 가지의 경우에 발생한다.

- 한 개의 논리 블록에 사상된 물리 블록의 수가 4개를 초과할 경우
- 프리 블록이 한 개 존재하고 로그 블록을 추가적으로 사상해야 할 경우
- 프리 블록이 한 개 존재하고 데이타 블록이 필요할 경우

그림 8의 ㉚번의 시점은 한 개의 논리 블록에 사상될 수 있는 물리 블록의 수를 초과할 경우이다. ㉚번의 시점에서 합병 연산을 수행한 모습을 그림 9에서 보여준

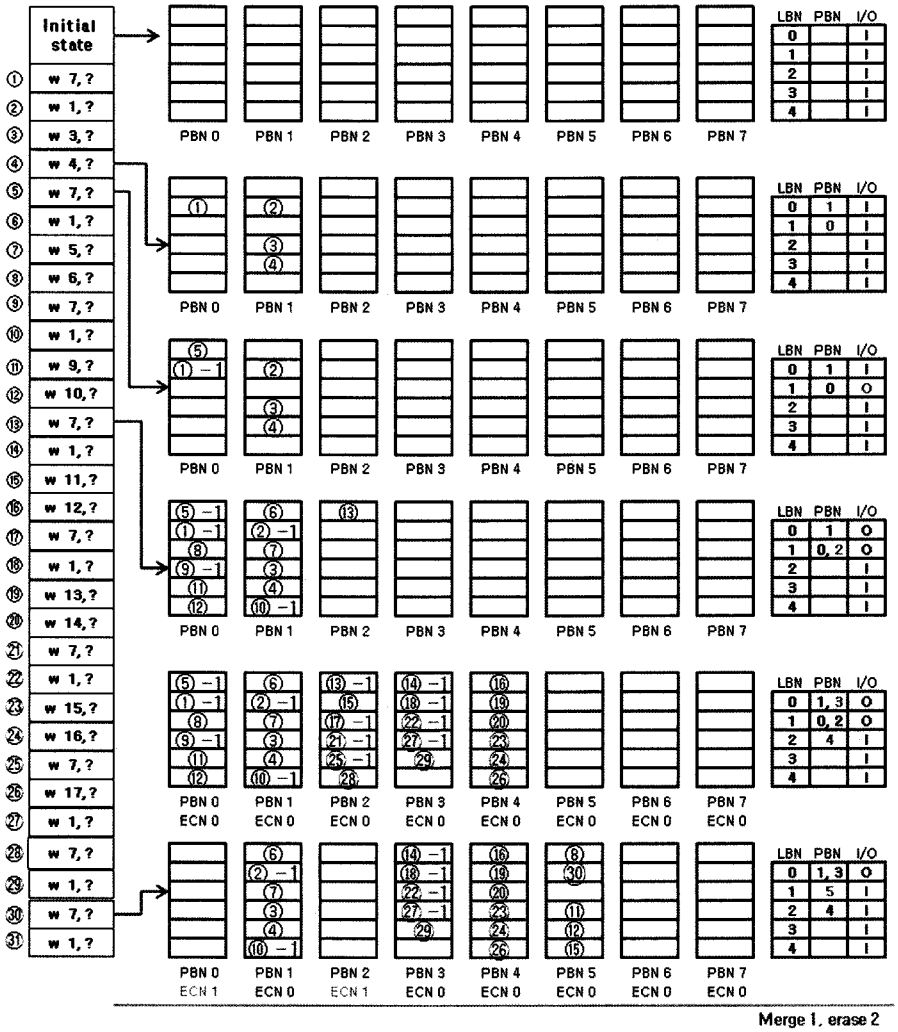

그림 8 EAST 쓰기 연산의 예시

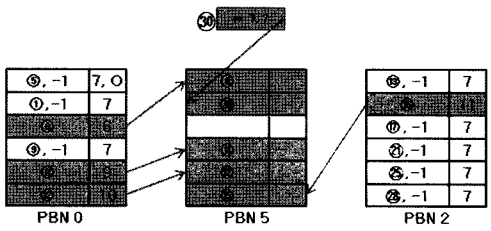

그림 9 합병 연산

다. 그림 9는 그림 8에 대한 합병 연산의 예로 한 논리 블록에 사상될 수 있는 물리 블록의 수가 2개라고 가정한다.

물리 블록 0번은 논리 섹터 6, 9, 10이 유효하고, 물리 블록 2번은 논리 섹터 11번이 유효하다. 지우기 연산 횟수가 가장 작은 프리 블록을 선택하여 새로운 데이타 블록으로 사상하고 유효한 섹터들에 대한 쓰기 연산을 수행한다. ㉚번의 시점에서는 선택된다. 쓰기 연산 후에는 물리 블록 0번과 2번에 지우기 연산을 수행한다.

프리 블록이 한 개 존재하고 로그 블록을 추가적으로 사상해야 할 경우에는 물리 블록이 2개 이상 사상된 논리 블록 중에서 회생될 블록을 선택한다. 회생될 블록을 선택하는 방법은 두 가지가 존재한다. 만약 오직 성능만을 생각한다면, 논리 블록에 사상된 물리 블록의 수가 가장 많은 것을 선택하면 된다. 만약 마모도 평준화도 고려했을 때는 각 논리 블록에 사상된 물리 블록들의 지우기 연산 횟수의 합을 구하여 이 값이 가장 작은 논리 블록을 선택한다. 논리 블록에 사상된 물리 블록들의 지우기 연산 횟수의 합은 RAM에 유지 될 수 있다.

세 번째의 경우는 프리 블록이 한 개밖에 존재하지 않지만 새로운 데이타 블록을 할당해야 될 경우이다. 세 번째의 경우는 두 번째의 경우와 같은 방법으로 합병 연산을 수행하면 된다.

4.4 읽기 연산

알고리즘 2는 EAST의 읽기 연산을 보여주고 있다. 알고리즘 2는 RAM의 여유가 없는 환경을 고려해서 로그 블록에 대한 섹터 사상 테이블을 사용하지 않았을 때의 알고리즘이다.

```
알고리즘 2 읽기 연산 알고리즘
1:  입력: 논리 섹터 번호(LSN)
2:  Procedure FTL_read(lsn)
3:  LBN = LSN/블록 당 섹터수
4:  LBN과 사상된 물리 블록 번호들과 블록 상태 확인(in-place/out-of-place)
5:  if 물리 블록이 in-place기법을 사용함 then
6:    LSN의 오프셋과 물리 섹터의 오프셋이 같은 섹터를 읽음;
7:  else
8:    LBN과 사상된 물리 블록 중 가장 나중에 사상된 물리 블록의 섹터들부터 검색
9:    for 사상된 물리 블록의 수만큼
10:     if 섹터의 여분 영역에 저장되어 있는 LSN가 읽고 싶은 LSN과 동일함 then
11:       섹터를 읽음;
12:       break;
14:   end if
15: end if
```

읽기 연산을 수행하기 위해서는 먼저 논리 섹터 번호를 한 개의 블록 가지는 섹터의 수로 나누어서 해당되는 논리 블록 번호를 구한다. 해당 논리 블록 번호가 in-place기법인지 out-of-place기법인지 RAM의 블록 사상 테이블을 통해 확인한다. in-place기법이라면 해당되는 섹터에 읽기 연산을 수행하면 된다. out-of-place 기법인 경우에는 섹터 사상 테이블의 사용여부에 따라 달라질 수 있다. 섹터 사상 테이블이 존재한다면, 섹터 사상 테이블을 통해 읽기 연산을 수행하고, 섹터 사상 테이블이 존재할 수 없는 환경이라면, 검색을 위한 읽기 연산을 수행한다. 하지만 검색을 위한 읽기 연산은 3.1절에 의해 지우기 연산을 넘지 않는 범위 내에서 일어나기 때문에 성능에 큰 영향을 주지 않는다.

5. 성능 평가

5절에서는 EAST과 현재 최고의 성능을 보이는 FAST와 비교하고 장단점을 분석하고 6절에서는 더 나은 방안을 모색하고 결론을 내린다. 표 1은 성능 평가에 사용된 쓰기 연산의 유형을 보여준다. 다양한 환경에서 실험하기 위해 디지캠과 리눅스와 같이 순차적인 쓰기 연산

표 1 성능 평가에 사용된 쓰기 연산의 유형

	쓰기 연산의 횟수
디지캠	69,575
코닥 A	21,991
코닥 B	10,221
코닥 C	5,110
리눅스	18,899
니콘	4,617

이 많은 유형과 그렇지 않은 코닥과 니콘과 같은 유형들을 실험했다. 또 EAST와 FAST의 공정한 비교를 위해 같은 블록의 수의 플래시 메모리를 대상으로 비교했다.

그림 10, 11, 12, 13, 14, 15는 표 1의 쓰기 연산의 유형들을 수행한 후 지우기 연산의 횟수를 나타낸 결과이다. 그림 10, 11, 12는 64MB 플래시 메모리를 대상으로 각 쓰기 연산의 유형을 한 번씩 수행한 결과이다. EAST는 3.2절과 3.3절에서 제안된 특성으로 인해 FAST보다 좋은 성능을 보이고 있다. 특히 3.3절의 특성은 EAST가 지우기 연산의 수를 FAST보다 절반이상으로 줄이는 가장 큰 원인이다. 이와 같은 성능 차이는 플래시 메모리의 용량이 크면 클수록 커진다.

그림 13, 14, 15는 64MB 플래시 메모를 대상으로 각 쓰기 연산의 유형을 15번씩 수행한 결과이다. 코닥 A, 코닥B, 코닥C, 니콘에서는 EAST가 더 좋은 성능을 보이지만, 디지캠과 리눅스에서는 FAST가 오히려 더 좋은 성능을 보인다. 디지캠과 리눅스의 쓰기 연산의 유형을 분석해보면, 이 두 쓰기 연산 유형은 다른 유형들보다 순차적으로 나열되는 쓰기 연산이 대부분이고 비교적 덮어쓰기가 다른 유형들에 비해 적은 편이다. 따라서 1번 수행했을 때는 없었던 순차적인 덮어쓰기가 15번을 서는 FAST가 더 좋은 성능을 보여준다.

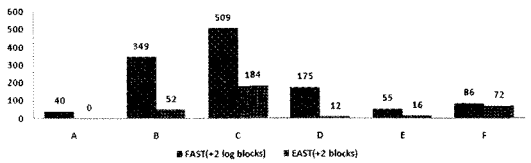

그림 10 실험환경: 빈 플래시 메모리 & 쓰기 연산 유형 1번 & 2개의 로그 블록

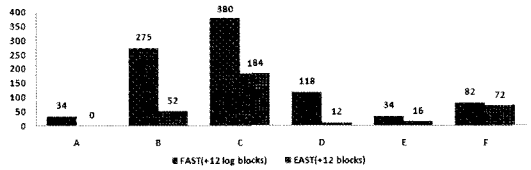

그림 11 실험환경: 빈 플래시 메모리 & 쓰기 연산 유형 1번 & 16개의 로그 블록

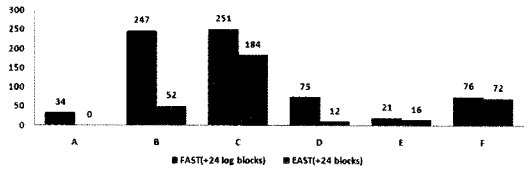

그림 12 실험환경: 빈 플래시 메모리 & 쓰기 연산 유형 1번 & 24개의 로그 블록

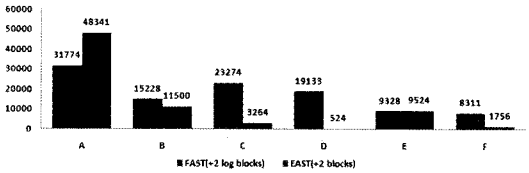

그림 13 실험환경: 빈 플래시 메모리 & 쓰기 연산 유형 15번 & 2개의 로그 블록

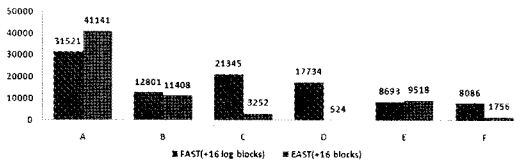

그림 14 실험환경: 빈 플래시 메모리 & 쓰기 연산 유형 15번 & 16개의 로그 블록

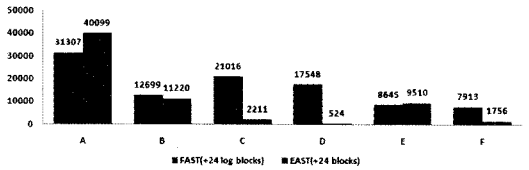

그림 15 실험환경: 빈 플래시 메모리 & 쓰기 연산 유형 15번 & 24개의 로그 블록

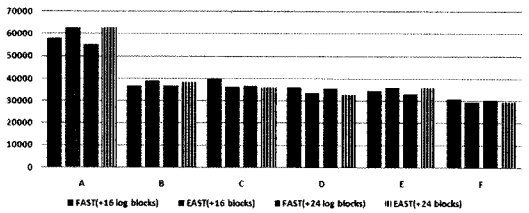

그림 16 실험 환경: 꽉 찬 플래시 메모리 & 각 유형 1회 수행

수행하면 많이 발생한다. FAST의 순차 로그 블록은 아무런 추가적인 쓰기 연산이 없이 단 한 번의 지우기 연산만으로 프리 블록을 만드는 효율적인 알고리즘이다. FAST의 순차 로그 블록의 활용으로 이와 같은 유형에

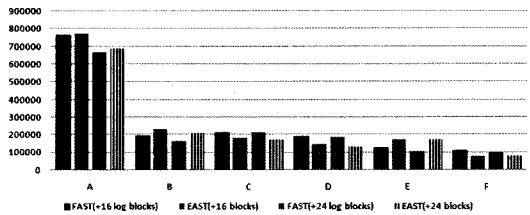

그림 17 실험 환경: 꽉 찬 플래시 메모리 & 각 유형 15회 수행

그림 16, 17은 64MB 플래시 메모리를 대상으로 플래시 메모리가 데이타로 모두 꽉 찬 상태에서 연산들의 시간을 구한 결과이다. 꽉 찬 상태에서 실험을 한 이유는 EAST의 3.3절에서 제안된 아이디어를 배제하고 서로 비교하기 위함이다. 이미 데이타로 모든 블록이 꽉 차있는 상태이기 때문에 쓰기 유형을 한번만 수행해도 순차적인 덮어쓰기가 수행된다. 따라서 디지캠과 리눅스 쓰기 연산의 유형에서는 FAST가 큰 차이로 더 좋은 성능을 보여줬고, 나머지 결과에서는 EAST이 3.2절에서 제안된 효율적인 블록의 섹터사용으로 조금 더 나은 성능을 보여줬다.

6. 결 론

EAST는 새로운 FTL로써 경우에 따라 블록 상태를 바꿀 수 있는 사상 기법을 사용하면서, 하나의 논리 블록 당 네 개의 물리 블록을 가질 수 있는 알고리즘이다. FAST는 현재의 FTL중 뛰어난 성능을 자랑하지만 블록의 활용성과 효율적인 용량사용 측면에서 문제점을 지니고 있다. EAST는 이와 같은 문제점들을 보안하기 위해 로그 블록의 개수를 최적화하고, 블록 상태를 저장하는 새로운 사상 기법을 사용하고, 재할당 블록을 효율적으로 사용하여 플래시 메모리의 효율성을 높였다.

5절의 성능 평가에서 EAST는 FAST보다 플래시 메모리의 용량과 블록의 활용성 면에서 더 효율적인 모습을 보여줬지만, 플래시 메모리가 데이타로 찬 상태에서 랜덤 쓰기 연산보다 순차 쓰기 연산에 대한 덮어쓰기가 더 많은 경우의 유형에서는 FAST가 더 나은 모습을 보여줬다. 따라서 EAST가 더욱더 발전하기 위해서는 EAST에 순차 로그 블록을 이행하는 것을 제안한다.

참 고 문 헌

[1] Samsung Electronics, "Nand flash memory" K9F5608X0D data book, 2007.

[2] M.Resenblum and J.Ousterhout, "The Design and implemtation of a log-structured file system," ACM Transaction on Computer Systems, Vol.10, No.1 Feb. 1992.

[3] M.Wu and W.Zwaenepoel, "eNVy: A non-volatile, main memory storage system," International Conference on Architectural Support for Programming Language and Operating Systems, 1994.
[4] A.Kawaguchi, S.Nishioka, and H.Motoda, "Flash memory based file system," USENIX 1995 Winter Technical Conference, 1995.
[5] Jesung Kim, et al., "A space-efficient flash translation layer for compact flash systems," IEEE Transactions on Consumer Electronics, 48(2), 2002.
[6] Tae-Sun Chung, Hyung-Seok Park, "STAFF: A flash driver algorithm minimizing block erasures," Journal of Systems Architecture, to appear, 2007.
[7] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, et. al., "A Log Buffer based Flash Translation Layer using Fully Associative Sector Translation," ACM Transaction on Embedded Computing Systems, Vol.6 issue3, 2007.
[8] Takayuki Shinohara, "Flash memory card with block memory address arrangement," 1999, United States Patent, no. 5,905,993.
[9] Amir Ban, "Flash file system optimized for page-mode flash technologies," 1999, United States Patent, no. 5,937,425.
[10] Petro Estakhri, "Moving sequential sectors within a block of information in a flash architecture," 1999, United States Patent, no. 5,930,815.
[11] Eran Gal and Sivan Toledo, "Algorithms and data structures for flash memories," ACM Computing Surveys, 37(2), 2005.
[12] Bum-soo Kim and Gui Young Lee, "Method of driving remapping in flash memory and flash architecture suitable therefor," 2002, United States Patent, no. 6,381,176.
[13] Amir Ban, "Flash file system optimized for page-mode flash technologies," 1999, United States Patent, no.5,937,425.

권 세 진
2002년 10월 아주대학교 정보및컴퓨터학사(B.S.). 2002년 10월~현재 아주정보통신전문대학원 석사 재학

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