FAST : 플래시 메모리 FTL을 위한 완전연관섹터변환에 기반한 로그 버퍼 기법

박 동 주[†] · 최 원 경[††] · 이 상 원[†††]

요 약

플래시 메모리가 개인 정보 도구, 유비쿼터스 컴퓨팅 환경, 모바일 제품, 가전 제품 등에 급속한 속도로 활용되고 있다. 플래시 메모리는, 이러한 환경에 저장매체로서 사용되기에 적합한 성질들 - 즉 저전력, 비휘발성, 고성능, 물리적인 안정성, 그리고 휴대성 등 - 을 갖고 있다. 그런데 하드디스크와 달리, 이미 데이터가 기록된 블록에 대해 덮어쓰기가 되지 않는다는 약점을 갖고 있다. 덮어쓰기를 위해서는 해당 블록을 지우고 쓰기 작업을 수행해야 한다. 이와 같은 성질은 플래시 메모리의 쓰기 성능을 매우 저하시킬 수 있다. 이와 같은 문제점을 해결하기 위해 플래시 메모리에는 FTL(Flash Translation Layer)라는 시스템 소프트웨어 모듈을 갖고 있다. 현재까지 많은 FTL 기법들이 제안되었는데, 그 중에서 대표적인 기법으로 로그블록 기법이 있다. 이 기법은 한정된 수의 로그블록을 쓰기 버퍼로 이용함으로써 쓰기에 따른 소거 연산을 줄임으로써 성능을 높인다. 그런데 이 기법은 로그블록의 활용률이 낮다는 것이 단점이다. 이러한 단점은 각 로그블록에 쓰여질 수 있는 섹터들이 블록 단위로 연관(Block Associative Sector Translation - BAST)되기 때문이다.

본 논문에서는 한정된 수의 로그블록들의 활용률을 높이기 위해 임의쓰기(random overwrite) 패턴을 보이는 섹터들을 전체 로그블록들에 완전 연관(Fully Associative Sector Translation - FAST)시킴으로써 활용률을 높이는 FAST 기법을 제안한다. 본 논문의 기여사항을 다음과 같다. 1) BAST 기법의 단점과 그 이유를 밝히고, 2) FAST 기법의 동기, 기본 개념, 그리고 동작원리를 설명하고, 3) 성능평가를 통해 FAST 기법의 우수성을 보인다.

키워드 : 플래시 메모리, FTL, 로그블록 기법, FAST 기법, 성능평가

FAST : A Log Buffer Scheme with *F*ully *A*ssociative *S*ector *T*ranslation for Efficient FTL in Flash Memory

Dong-Joo Park[†] · Won-Kyung Choi[††] · Sang-Won Lee[†††]

ABSTRACT

Flash memory is at high speed used as storage of personal information utilities, ubiquitous computing environments, mobile phones, electronic goods, etc. This is because flash memory has the characteristics of low electronic power, non-volatile storage, high performance, physical stability, portability, and so on. However, differently from hard disks, it has a weak point that overwrites on already written block of flash memory is impossible to be done. In order to make an overwrite possible, an erase operation on the written block should be performed before the overwrite, which lowersthe performance of flash memory highly. In order to solve this problem, the flash memory controller maintains a system software module called the flash translation layer(FTL). Of many proposed FTL schemes, the log block buffer scheme is best known so far.This scheme uses a small number of log blocks of flash memory as a write buffer, which reduces the number of erase operations by overwrites, leading to good performance. However, this scheme shows a weakness of low page usability of log blocks. In this paper, we propose an enhanced log block buffer scheme, FAST(Full Associative Sector Translation), which improves the page usability of each log block by fully associating sectors to be written by overwrites to the entire log blocks. We also show that our FAST scheme outperforms the log block buffer scheme.

Key Words : Flash Memory, FTL, Log Block Scheme, FAST Scheme, Performance Evaluation

1. 서 론

플래시 메모리는 소비전력이 적고, 전원이 꺼지더라도 저장된 정보가 사라지지 않은 특성을 지닌 비휘발성 기억장치로 하드디스크와 같은 기존의 디스크에 비해 데이터 접근 성능이 빠르고 크기가 매우 작다. 또한 물리적인 충격에 강하고 무게가 가볍다는 장점을 갖고 있다. 이러한 특성으로 인해서, 플래시 메모리는 최근에 MP3플레이어, 휴대 전화기, 개인정보단말기(PDA), 디지털 카메라/캠코더 등의 휴대용 정보기기들의 보조기억장치로 급속도로 사용되기 시작했다[1].

※ 본 연구는 숭실대학교 교내연구비 지원으로 이루어졌음.
† 정 회 원 : 숭실대학교 컴퓨터학부 전임강사
†† 준 회 원 : 삼성전자 연구원
††† 정 회 원 : 성균관대학교 정보통신공학부 조교수
논문접수 : 2005년 1월 9일, 심사완료 : 2005년 5월 2일

〈표 1〉 다양한 기억장치의 성능 비교[2]

	접근시간		
	읽기 (Read)	쓰기 (Write)	소거 (Erase)
DRAM	60ns(2B) 2.6μs(512B)	60ns(2B) 2.6μs(512B)	-
NOR형 플래시	150ns(1B) 15μs(512B)	211μs(2B) 3.5ms(512B)	1.2s(128KB)
NAND형 플래시	10μs(1B) 36μs(512B)	226μs(2B) 266μs(512B)	2ms(16KB)
하드디스크	12.4ms(512B)	12.4ms(512B)	-

플래시 메모리는 EEPROM의 한 종류로 NAND 형과 NOR 형 두 가지로 분류할 수 있다. NAND 형은 고집적이 가능하여 데이터를 저장하는 데 주로 사용되며, 섹터(sector)[1] 단위의 I/O를 지원한다. NOR 형은 처리 속도가 빨라 프로그램 코드를 저장하는데 주로 사용되며, 바이트 단위의 I/O를 지원한다. <표 1>은 DRAM, 플래시 메모리, 디스크 간의 읽기, 쓰기, 삭제 연산 성능을 비교한 것이다.

그런데 플래시 메모리는 하드 디스크와 달리, 특정 섹터(sector)에 쓰기연산(write)을 하기 위해서 해당 섹터가 깨끗한 상태로 비어져 있어야만 한다. 다시 말해서, 데이터가 쓰여져 있는 섹터에 대해 덮어쓰기(overwrite)가 허용되지 않는다. 따라서 이런 경우에는 섹터를 포함하고 있는 블록 전체를 소거연산(erase)을 통해서 깨끗이 지우고(이때, 소거연산의 단위는 블록) 쓰기연산을 수행해야 한다. 하지만 표1에서 보듯이 블록의 소거연산에 걸리는 시간은 쓰기연산과 읽기연산에 비해 훨씬 크다. 이러한 플래시 메모리의 특징은 플래시 메모리가 기존의 하드디스크를 대체하는 것을 어렵게 만들고, 플래시 메모리 시스템의 전체 성능의 저하를 가져온다. 즉, 쓰기연산에 의해 유발되는 소거연산의 횟수를 줄이는 것이 플래시 메모리 성능에 절대적인 영향을 미친다.

위와 같은 플래시 메모리의 문제점을 해결하기 위해서, 플래시 변환 계층(Flash Translation Layer, 이하FTL)이라 불리는 시스템 소프트웨어를 사용한다[1,2,8,9,10]. FTL은 플래시 메모리와 파일시스템 사이에 위치해서 파일 시스템이 플래시 메모리를 하드디스크처럼 블록 디바이스로 사용할 수 있게 해준다. FTL을 사용하게 되면 별도의 파일 시스템이 필요로 하지 않고 기존의 하드디스크를 가정한 파일시스템을 그대로 사용할 수 있다는 장점이 있다.

플래시 메모리를 위한 많은 FTL 기법들 중 대표적인 것이 로그블록 기법이다[2]. 이 기법은 한정된 수의 로그블록을 쓰기 버퍼(write buffer)로 이용하는 것이 핵심이며, 사진 저장이나 파일 복사 등을 위해 필요한 대량의 순차섹터연산(sequential writes, 이하 순차쓰기)과 소량의 임의섹터연산(random small overwrites, 이하 임의쓰기)에 대해서 좋은 성능을 보이는 것으로 알려져 있다. 그러나, 이 기법은 임의쓰기에 대해 사용된 로그블록의 활용률[2]이 좋지 않은 단점을 가지고 있다. 이 단점의 근본적인 이유는 특정 로그블록에 쓰여질 수 있는 섹터들이 블록 단위로 연관(Block Associative Sector Translation)되기 때문이다(이하, 이 로그블록 기법을 "BAST 기법"[3]이라 부름). 본 논문에서는, 임의쓰기 패턴을 보이는 섹터들을 전체 로그블록들에 완전연관(Fully Associative Sector Translation)시킴으로써 로그블록들의 활용률을 높이는 FAST 기법을 제안한다.

본 논문의 기여사항은 다음과 같다. 1) BAST 기법의 단점과 그 이유를 밝히고, 2) FAST 기법의 동기, 기본 개념, 그리고 동작원리를 설명하고, 3) 성능평가를 통해 FAST 기법의 우수성을 보이고, 다양한 설정(configuration)들에 대해 FAST의 성능을 분석한다. 특히, FAST 기법을 이용해서 대표적인 플래시 메모리용 응용분야의 특정 샘플 작업부하를 적용해본 결과, 많은 경우 소요시간 및 소거회수 측면에서 50% 이상의 성능 개선을 보였다.

본 논문의 구성은 다음과 같다. 2장에서는 플래시 메모리의 FTL의 기본 개념, 관련 용어, 그리고 지금까지 개발된 FTL 알고리즘들을 간략히 설명하고, 그 중 대표적인 BAST 기법의 기본적인 동작 원리와 문제점을 지적한다. 3장에서는 BAST 기법의 단점을 극복하기 위해 본 논문에서 제시하는 FAST 기법의 핵심 아이디어를 설명하고, 이의 구현을 위한 새로운 FTL 구조와 구현 알고리즘의 자세한 설계내용을 설명한다. 4장에서는 본 논문에서 제시한 FAST 기법과 기존BAST 기법의 성능을 비교 분석하고, 5장에서는 결론을 맺고 향후 연구 과제를 설명한다.

2. 관련 연구

2.1 FTL 배경지식

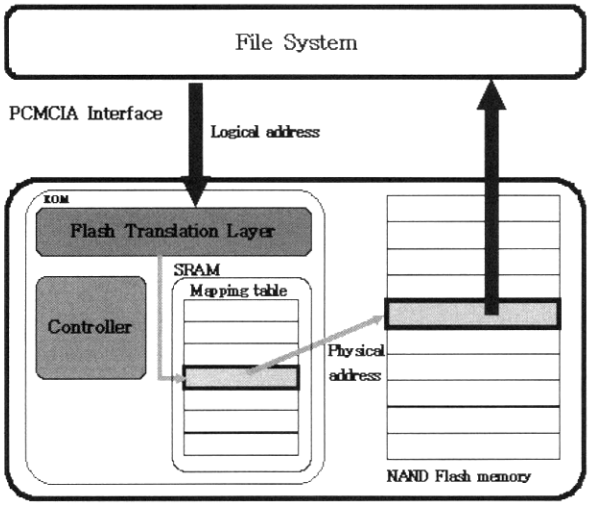
FTL의 가장 중요한 기능은 파일시스템으로부터 받은 논리 주소를 플래시 메모리의 물리 주소로 사상하는 것이다. (그림 1)은 컴팩트(compact) 플래시 시스템의 일반적인 내부 구조를 나타낸 것이다. 파일시스템에서 논리 주소에 대한 읽기 요청이 들어오게 되면 FTL은 사상 테이블을 통해 논리 주소에 해당하는 물리 주소를 찾아서 읽기연산을 수행하게 된다. 플래시 메모리에서 기존의 데이터를 변경하려는 경우에는(즉, 덮어쓰기) 데이터를 직접 수정할 수 없기 때문에 변경된 데이터를 저장할 새로운 공간(free space)을 찾아 확보하고 변경된 데이터를 새로운 공간에 저장하고 기존의 데이터는 소거한다. 그리고 사상테이블의 정보도 갱신한다.

지금까지 제안된 FTL의 주소변환 기법들에 대한 자세한 비교는 [1]의 논문을 참조하기 바란다. 본 논문에서는 새로

1) 플래시 메모리는 논리적으로 여러 개의 블록으로 이루어져있고, 하나의 블록은 여러 개의 섹터(이를 페이지(page)라고 부르기도 하지만, 본 논문에서는 섹터라고 부르겠다.)로 구성된다. 하나의 섹터는 보통 512 byte의 데이터 영역과 16 byte의 예비 공간을 갖는다. 하나의 블록은 32 개의 섹터로 이루어진다.

2) 로그블록의 활용률은, 현재의 로그블록이 교체되는 시점에서 이 로그블록을 구성하는 섹터들 중 사용된 섹터들의 비율을 의미한다.

3) 로그블록 기법의 저자들은 BAST라는 용어를 사용하지 않았지만, 본 논문에서 제안하는 기법과 기존 로그블록 기법의 본질적인 차이점이 섹터와 로그블록들과의 연관성(associativeness)이기 때문에 본 논문에서는 BAST과 FAST 기법 등의 약어로서 두 기법을 구분하고자 한다.



(그림 1) 컴팩트 플래시 시스템의 내부 구조

제안하는 FAST 기법을 이해하는데 필요한 범위 내에서 주소변환 기법과 그 과정에 대해서만 설명한다. 기본적으로 논리 주소와 물리 주소의 사상 단위는 섹터 또는 블록이 된다. 섹터단위 주소사상(sector mapping) 방식은 논리섹터와 물리섹터 간의 사상정보를 유지한다[1]. 섹터와 같이 작은 단위로 주소를 변환하므로 성능이 좋은 반면, 섹터 단위로 사상 테이블(mapping table)을 유지해야 하므로 테이블의 크기가 커지게 된다. 이는 많은 양의 SRAM을 필요로 하고, 결국은 플래시 메모리 시스템의 제작 단가를 높이게 된다. 블록단위 주소사상(block mapping) 방식은 논리블록과 물리블록 단위의 사상정보를 유지한다[1]. 블록 단위로 사상 테이블을 유지하기 때문에 테이블의 크기가 작은 장점이 있지만, 블록 내부의 특정 페이지를 수정해야 할 경우 블록 전체를 삭제하고 다시 사용하기 때문에 성능이 좋지 않다. 위의 두 방식은 서로 장단점을 갖고 있기 때문에 두 방식을 혼합한 방식을 사용한다. 그 예로, Mitsubishi, SSR, Lexar Media, 로그블록 방식 등이 있으며, 지면 관계상 각 방식에 대한 자세한 설명은 [1]의 논문을 참조하기 바란다. 이들 중에서 로그블록 방식(이후 BAST 방식)이 가장 뛰어나므로 [2], 이후에는 BAST 방식과 이 논문에서 제시하는 FAST 방식을 서로 비교한다.

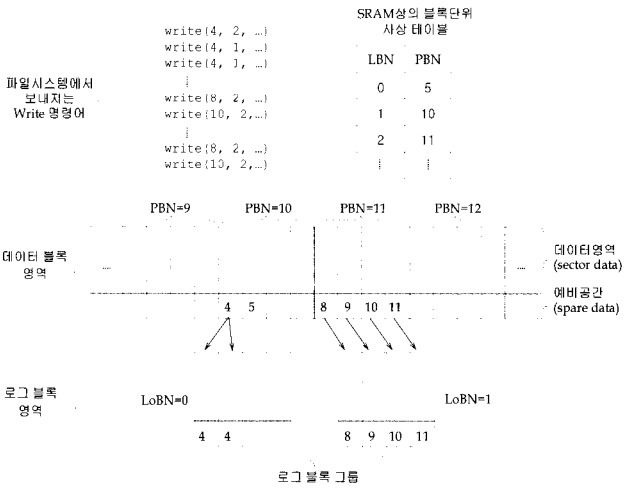
2.2 BAST 기법의 원리와 문제점

(그림 1)의 파일 시스템에서 플래시 메모리에 데이터를 기록하기 위해서는 중간의 FTL의 쓰기 인터페이스를 호출한다. 쓰기 인터페이스는 write(start_lsn, length, data[])로 정의되며, 논리섹터번호(Logical Sector Number, 이하 LSN) start_lsn부터 시작해서 start_lsn+length-1까지 주어진 데이터를 쓰라는 것을 의미한다. FTL의 쓰기연산이 호출되면, FTL은 주어진 시작 논리 섹터 번호를 이용하여 논리블록번호(Logical Block Number, 이하 LBN)를 계산하고 이 값을 이용하여 사상 테이블에서 물리블록번호(Physical Block Number, 이하 PBN)를 찾아낸다.[4] 그리고 나서, 찾은 물리블록에서 데이터를 기록할 시작 위치(offset)을 계산하고,[5] 시작 위치부터 주어진 길이(length)만큼 데이터를 기록한다.

만약 파일 시스템으로부터 이전의 쓰기연산에 의해 사용된 논리섹터에 또 다른 쓰기연산이 호출되면(즉, 덮어쓰기), FTL은 이전과 동일한 플래시 메모리의 물리 위치에 쓰기 작업을 수행해야 하지만 플래시 메모리의 특성상 같은 물리 위치에 데이터를 수정할 수 없다. 이러한 문제점을 해결하기 위해서, 현재의 쓰기연산을 예비 블록에 수행을 하고 예비 블록의 나머지 페이지들은 충돌이 발생한 물리 블록으로부터 복사하고 이 블록은 소거하여 예비 블록으로 설정해 주는 방법을 생각해 볼 수 있다. 그러나 이 방법은 같은 물리 블록에 충돌이 계속 발생하면 매번 많은 수의 복사연산(copy)과 한번의 소거연산을 유발하는 단점이 있다. 이러한 단점을 보완하는 한 방법이 BAST기법이다[2]. 이 방식을 설명하기 전에 먼저 다음과 같은 용어를 정의한다.

- 순차쓰기: 파일 시스템으로부터의 파일 복사 등에 의해 발생하며, 이때 대량의 순차적 쓰기연산이 발생한다 (large sequential writes).
- 임의쓰기: 순차쓰기 중간 중간에 몇 개의 특정 논리 섹터 영역에 쓰기연산이 발생하며, 이 영역들의 전체 크기는 아주 작다(small random writes). 이는 주로 윈도우 파일시스템의 FAT(File Allocation Table) 등의 정보를 기록하기 위해 발생한다.

(그림 2)는 BAST 기법에서 쓰기연산의 처리 과정의 예를 보여주고 있다. (그림 2)에서는 블록당 섹터의 수가 네 개라고 가정하며, 그림의 왼쪽 윗부분은 파일 시스템으로부터 호출된 쓰기연산의 순서를, 오른쪽 윗부분은 현재의 블록단위 사상테이블을 보여주고 있다. 이때, BAST 기법에서 쓰기연산 처리는 다음과 같다. 첫 번째 쓰기연산이 호출되



(그림 2) BAST 기법에서의 쓰기연산 처리

4) 논리 블록 주소 = start_lsndi 블록당 섹터 수.
5) 시작 위치 = start_lsnmo 블록당 섹터 수.

면, FTL은 사상 테이블에서 LBN 1에 사상되는 PBN 10을 찾고 이 블록의 시작 위치 0부터 두 개의 섹터 데이터를 기록한다. 즉, 물리 블록 10의 위치 0과 1에는 논리 섹터 4와 5가 각각 대응된다. 두 번째 쓰기연산에 대해서는 이미 같은 위치에 논리 섹터 4가 기록되어 있으므로, 로그블록관리자(Log Block Manger)[6]로부터 로그블록을 하나 할당 받고(즉, LoBN=0), 이 블록에 논리 섹터 4를 쓴다. 마찬가지로 세 번째 쓰기연산에 대해서도 논리 섹터 4를 같은 로그블록의 그 다음 빈 섹터에 쓴다. 이후의 쓰기 연산 처리의 결과는 그림의 로그블록 1(즉, LoBN=1)과 같다.

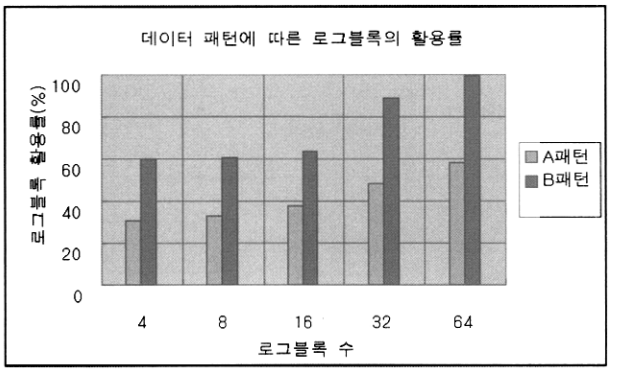
(그림 2)에서 로그블록관리자가 사용할 수 있는 로그블록 개수가 두 개라고 가정하고 두 개의 쓰기연산 write(12, 1,…), write(12, 1,…)이 호출되어 덮어쓰기 연산이 발생하면, 로그블록관리자는 이미 사용중인 로그블록 중 하나를 선택하여(일종의 victim) 이 블록을 새로 발생한 덮어쓰기 연산을 위해 할당하게 된다. 특정 로그블록이 선택되어서 다른 논리블록을 위한 예비 블록으로 사용되는 경우, 이 로그블록이 교체된다고 한다. 선택한 로그블록을 할당하기 전에 합병연산이 발생하는데, 예를 들면 (그림 2)의 PBN=10인 데이터블록과 LoBN=0 인 로그블록에 기록되어 있는 섹터 데이터 중 가장 최신의 섹터들만 예비 블록에 복사하는 작업이 이루어지며, 이 두 블록은 각각 소거되어 하나는 예비 블록으로, 나머지 하나는 새로운 덮어쓰기 연산을 위해 할당된다.

BAST에서는 로그블록의 상태에 따라 합병연산 시에 두 가지 최적화 기법을 제공한다. (그림 2)에서 LoBN=1의 경우, 최신의 섹터 데이터가 모두 로그블록에 기록되어 있으므로 굳이 예비 블록에 복사할 필요가 없이 해당 로그블록을 데이터블록으로 교환해 주면 된다. 이를 교환(switch) 기법이라고 한다. LoBN=0의 경우는, 최신의 섹터 데이터가 PBN=10인 데이터블록에도 존재할 수 있으므로 두 블록을 다 고려하여 복사 작업이 수행되어야 하며, 이것을 지능적 복사(smart copy)라고 한다. 지능적 복사와는 달리, 교환연산은 복사연산이 전혀 발생하지 않고, 또한 소거도 데이터블록에 대해서만 발생하므로 합병연산의 비용이 크지 않다.

2.3 연구동기

BAST 기법에서는, 대량의 순차쓰기에 해당하는 경우를 제외하고 동시에 여러 논리블록들에 임의쓰기가 발생할 경우, 이 논리블록들은 한정된 수의 로그블록들을 대상으로 경쟁(contention)을 한다. 따라서 할당된 로그블록의 모든 섹터들이 다 쓰여지고 교체되기보다는, 일부분의 섹터들만(일반적으로 60% 이하) 사용되고 교체되는 경우가 많다. (그림 3)은 플래시 메모리를 사용하는 샘플 작업부하(workload)에 대해, 로그블록의 개수에 따라 로그블록이 몇 % 정도의 섹터가 실제 사용되고 교체되는지를 보여주고 있다. 패턴 A는

6) FTL 계층 안에 있는 특정 소프트웨어 모듈, 로그블록들을 관리해주는 역할을 담당한다.



(그림 3) BAST 기법에서의 로그블록들의 평균 활용률

임의쓰기의 성격을 가진 패턴이고, 패턴 B는 디지털 카메라에서 보이는 순차쓰기의 성격을 가진 패턴이다. 표에서 알 수 있듯이, 패턴 A와 같은 임의쓰기에 대해서 로그블록들의 평균 활용률은 50%에도 미치지 못하고 교체됨을 알 수 있다. 뿐만 아니라, 순차쓰기 성격을 가진 패턴 B 에 대해서도 로그블록 개수가 16개 이하인 경우 평균 활용률이70%를 넘지 못한다. 이러한 현상은 교환연산의 감소와 합병연산의 증가를 초래하기 때문에 플래시 메모리의 성능이 저하되게 된다.

이와 같은 문제점 때문에, 여러 논리블록들에 대한 동시적인 임의쓰기가 발생하는 작업부하에 대해서는 BAST 기법은 좋지 않은 성능을 보일 수 있다. 따라서, 우리는 한정된 수의 로그블록들이 교체될 시점에 사용된 섹터들이 가능한 한 많도록, 즉 할당된 로그블록의 활용률을 최대한 높임으로써 성능 저하의 문제점을 해결하고자 한다.

3. FAST 기법: 완전연관섹터변환(Fully Associative Sector Translation)

3.1 기본 아이디어

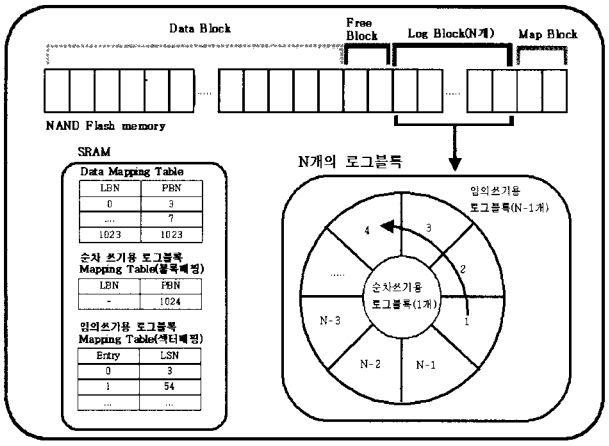
기존의 BAST에서는 하나의 로그블록에는 실제로 같은 논리 블록에 속하는 섹터들만 사상될 수 있다(그림 2 참조). 따라서 특정 시간윈도우(time window)[7]에서 로그블록들의 수보다 더 많은, 논리블록들에 대한 임의쓰기가 발생하게 되면, 논리블록들 사이에 한정된 수의 로그블록들에 대한 경쟁이 발생하게 되고 결국은 로그블록들에 대한 쓰레싱(thrashing) 현상[8]이 발생할 수 있다. 이러한 단점을 극복하기 위해, 로그블록들에는 임의의 논리블록에 속하는 섹터들을 완전연관(fully associative)[9] 사상할 수 있도록 하는 것이 핵심 아이디어다. 즉, 원본 데이터블록들에 대해서는 블록단위 주소사상을 하고, 로그블록에 쓰여지는 임의의 섹터들에 대해서는 완전연관 섹터단위 주소사상을 사용하는 것이다. 따라서 BAST 기법에서와는 달리, 각 로그블록은 블

7) 운영체제에서 작업부하에 시간윈도우 개념과 동일하게 생각하면 된다
8) 운영체제에서 버퍼의쓰레 현상과 유하다
9) 컴퓨터 구조에서 CPU 캐시에 메모리의 페이지를 사상하는 연관성과 유사하기 때문에 연관성(associatiit) 개념을 도입했다.

록 전체의 섹터가 모두 사용된 이후에 소거된다. 그 결과로서, 각 로그블록의 활용률이 높아지고 로그블록에 대한 합병연산의 수가 감소하므로 플래시 메모리의 성능을 향상 시킬 수 있다.

3.2 FAST-FTL 의 구조

(그림 4)는 FAST 기법을 구현하기 위한 FTL 모듈의 내부 구조를 보여주고 있다. N개의 물리블록이 로그 영역으로 할당되는데, 이중 한 개의 블록은 순차쓰기 전용으로 사용되고(이하, 이 블록은 "순차쓰기용 로그블록"), 나머지 N-1개의 로그블록들은 임의쓰기에 해당하는 섹터들을 위해 사용한다(이하, 이 블록은"임의쓰기용 로그블록"). 그림 4에서 오른편 하단 부분은 N개의 로그블록들을 사용하는 방식을 개념적으로 보여주고 있는데, 이에 대해서는 다음 절에서 자세히 설명하겠다.



(그림 4) FAST-FTL 모듈의 구조

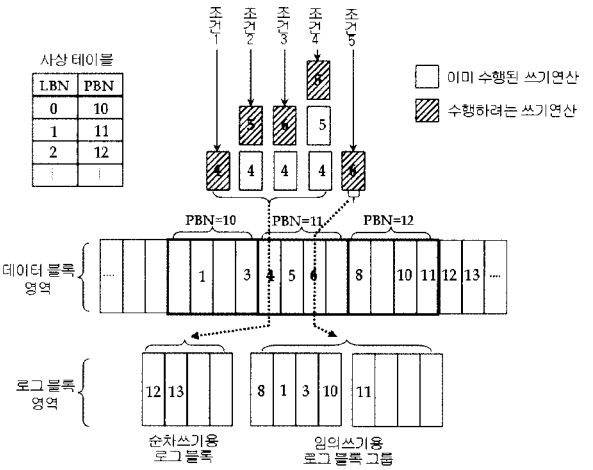
순차쓰기용 로그블록의 용도는, BAST 기법과 유사하게, 순차쓰기된 로그블록을 합병연산 시 교환연산(switch) 최적화 방법을 통해 로그블록에 대한 소거연산과 복사연산을 줄이기 위해서다. 순차쓰기용 로그블록에는 같은 논리블록 주소를 가지는 섹터들만 저장되므로 데이터블록과 마찬가지로 블록단위 주소사상 방식을 사용한다(이를 위해, "순차쓰기용 블록단위 주소사상 테이블"을 유지하며, 여기에는 현재까지 저장된 섹터의 개수도 기록함). 반면, 임의쓰기용 로그블록 그룹에 대해서는 현재 어떤 논리블록의 어떤 논리섹터가 저장되어 있는지를 유지하는 섹터단위 주소사상이 필요하다(이를 위해, "임의쓰기용 로그블록 섹터단위 주소사상 테이블"을 유지함). 원본 데이터블록들에 대해서는 BAST에서와 마찬가지로 블록단위 주소 사상 테이블을 유지한다.

3.3 FAST-FTL 알고리즘

이 절에서는 (그림 4)와 같은 구조를 가정했을 때, 읽기연산과 쓰기연산에 대해FAST-FTL이 내부적으로 어떻게 동작하는지를 논리적으로 설명한다.

읽기연산 : 논리 섹터m에 대한 읽기연산이 발생한 경우, 순차쓰기용 주소사상 테이블을 확인하고, 해당 섹터가 없을 경우 임의쓰기용 주소사상 테이블을 확인한다. 같은 논리블록 주소를 갖는 섹터들이 임의쓰기용 로그블록과 순차쓰기용 로그블록에 동시에 저장되는 경우는 없기 때문에 두 주소사상 테이블을 확인하는 순서는 중요하지 않다. 위의 두 주소사상 테이블에서 찾는 섹터가 없는 경우 데이터블록에서 해당 섹터를 읽어서 반환한다.

쓰기연산 : 쓰기연산을 위한 인터페이스는 2장의write(start_lsn, length, data[])과 같고, 또한 한 블록 안의 섹터 수는 32로 가정한다. 쓰기연산이 호출되면, 먼저 데이터를 써야 할 논리블록 번호(LBN)(= start_lsn *div* 32)를 계산하고, 주소사상 테이블에서 물리블록번호(PBN)를 얻는다. 이 데이터블록에서, 쓰기연산의 인자값으로 주어진 각 논리섹터[10]에 대해서 해당 자리에 최초 쓰기를 하는지 아니면 덮어쓰기를 하는지를 확인한다.[11] 만약, 최초 쓰기를 하는 경우는 해당 자리에 해당 섹터데이터를 기록하고, 그렇지 않고 덮어써야 하는 경우 아래의 조건들을 검사하여 적절한 연산을 수행한다.



(그림 5) FAST 기법에서의 쓰기연산 처리

조건 1. (start_lsn *mod* 32 = 0) & (lbn=(start_lsn *div* 32) ∉ rw_lbn_set)
조건 2. (lbn = sw_lbn) & (start_lsn *mod* 32 = sw_sec_num)
조건 3. (lbn = sw_lbn) & (start_lsn *mod* 32 > sw_sec_num)
조건 4. (lbn = sw_lbn) & (start_lsn *mod* 32 < sw_sec_num)
조건 5. 조건 1, 2, 3, 4를 모두 만족하지 않는 경우

우선 위 조건들에 사용된 용어 rw_set_of_lbns, sw_lbn 그리고 sw_lsn_num를 설명하겠다. 임의쓰기용 로그블록 그룹에 대해 FTL 모듈에서는 주소사상 테이블뿐만 아니라, 현재 임의쓰기용 로그블록 그룹에 기록된 섹터들의 논리블

10) 이때 쓰기연산의 논리 섹터 번호는 start_lsn, start_lsn+1,, start_lsn-length+1이 된다.
11) 이것을 검사하기 위해서는, 해당 페이지 내의 예비 공간에 기록되어 있는 논리 섹터 번호 0보다 같거나 큰 지를 비교한다. 만약, 작으면 최초 쓰기이며, 그렇지 않으면 덮어쓰기이다.

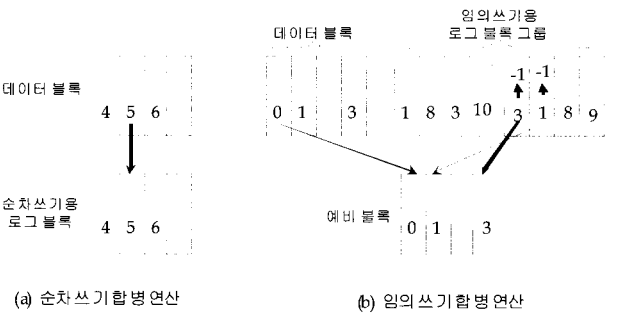
록들의 주소 정보 rw_set_of_lbns(rw는 random write의 줄임말)도 유지한다. 그리고, 순차쓰기용 로그블록에 현재 쓰여지고 있는 논리블록 주소가 sw_lbn(sw는 sequential write의 줄임말), 그 논리블록에 대해 순차적으로 쓰여진 섹터의 수가 sw_sec_num이다.

위의 조건들을 (그림 5)의 예제를 이용하여 자세히 설명한다. (그림 5)에서 블록당 섹터의 수는 4이고, 임의쓰기용 로그블록의 수는 2라고 가정한다. 조건 1은 교환연산이 일어날 가능성이 있는 쓰기연산, 즉 인자값의 시작 섹터 번호(즉, start_lsn)와 논리블록의 첫 번째 섹터 번호와 동일한 쓰기연산을 찾아낸다. 이때, 이 조건뿐만 아니라 동일한 논리블록 주소(즉, lbn=(start_lsn *div* 4))를 갖는 섹터 주소가 임의쓰기용 주소사상 테이블에 존재하지 않아야 한다. 다시 말해, 현재의 쓰기연산이 전체 로그블록 그룹에 최초로 섹터를 기록하는 조건을 만족해야 한다. 만약, 두 종류의 로그블록 그룹에 동일한 섹터를 유지하면, 읽기연산 시 최신의 섹터를 찾을 때의 오버헤드(overhead)를 피하기 위한 것이다. (그림 5)의 "조건 1의 쓰기연산"에서 start_lsn=4인 경우, (start_lsn *mod* 4)=0 이고, 동일한 논리 블록 주소 1(=4 *div* 4)를 갖는 섹터가 임의쓰기용 로그블록 그룹에 없기 때문에 조건 1을 만족한다. 따라서 섹터 4는 순차쓰기용 로그블록의 첫 번째 섹터에 기록된다. 그런데 (그림 5)의 순차쓰기용 로그블록에는 이미 다른 섹터들(즉, 12, 13)이 저장되어 있으므로, 이 로그블록과 원본 데이터블록 간의 합병연산이 발생하여 순차쓰기용 로그블록이 소거된 이후에 섹터 4가 기록된다.

조건 2는 순차쓰기용 로그블록에 섹터를 연속해서 추가(append)할 수 있는 쓰기연산을 찾는다. 즉, 인자값의 시작 섹터가 현재의 순차쓰기용 로그블록에 기록될 수 있고(즉, lbn=sw_lbn) 또한 로그블록의 마지막 섹터에 연이어 섹터들을 기록할 수 있어야 한다(즉, start_lsn *mod* 4 = sw_sec_num). 만약, 조건 2를 만족하여 쓰기연산에 의해 섹터들이 로그블록에 기록이 되어 로그블록의 다 차는 경우에는 바로 교환연산을 수행하고 소거된 순차쓰기용 로그블록을 만들어 준다. (그림 5)의 "조건 2의 쓰기 연산"에서, 섹터 4를 기록하는 쓰기연산을 수행하고 연이어 섹터 5를 기록하는 쓰기연산을 처리하려고 할 때, 섹터 5에 대한 쓰기 연산은 조건 2를 만족하므로 순차쓰기용 로그블록에 섹터 5가 기록된다(이때 로그블록에는 섹터 4, 5가 순차적으로 기록됨).

조건 3은 순차쓰기용 로그블록의 마지막 섹터에 연이어 섹터들을 추가할 수 없지만, 로그블록에 이들을 기록할 수 있는 쓰기연산을 찾는다. 즉, 조건 2와 같이 인자값의 시작 섹터가 순차쓰기용 로그블록에 기록될 수 있어야 하며, 또한 시작 섹터 번호와 로그블록의 마지막 섹터 번호 간의 차가 1보다 커야 한다(즉, (start_lsn *mod* 4) > sw_sec_num). (그림 5)의 "조건 3의 쓰기연산"에서, 이미 순차쓰기용 로그블록에 섹터 4가 기록되고 섹터 6을 기록하려는 쓰기연산은 조건 3을 만족시킨다. 이런 경우는, 가까운 미래에 섹터 5가 기록되어 로그블록에 순차적으로 섹터가 채워질 확률이 낮기 때문에 합병연산을 수행한다.

조건 4는 순차쓰기용 로그블록에 이미 기록되어 있는 섹터들 중 일부분에 대해서 덮어쓰기가 발생하는 쓰기연산을 찾는다. 즉, 조건 2, 3과 같이 쓰기연산의 섹터들이 현재의 순차쓰기용 로그블록에 기록될 수 있어야 하며, 또한 이들 섹터 중 일부분에 대해 덮어쓰기가 발생해야 한다(즉, (start_lsn *mod* 4) < sw_sec_num). 이 경우에는 데이터블록과 로그블록 간의 합병연산이 수행된다. 마지막으로, 위의 조건 1, 2, 3, 4를 모두 만족하지 않을 때는 쓰기연산의 섹터들을 임의쓰기용 로그블록 그룹에 기록하며, 기록할 위치는 로그블록에 마지막으로 기록한 위치 바로 다음이 된다. 만약, 로그블록 그룹에 더 이상의 빈 섹터가 없으면 첫 번째 로그블록에 대해 합병연산을 수행하여 공간을 확보한다.



(그림 6) FAST-FTL에서의 합병연산

3.4 FAST-FTL에서의 합병연산의 최적화

위에서 설명한 다섯 개의 조건에서 합병연산이 발생할 수 있으며, 크게 순차쓰기용 로그블록과 임의쓰기용 로그블록에서의 합병연산으로 나눌 수 있다. 합병연산을 간단히 설명하면, 데이터블록과 로그블록 그룹에서 가장 최신의 섹터를 예비 블록에 복사하여 데이터블록과 교환하고, 그 다음 해당 데이터블록과 로그블록을 소거하는 작업을 일컫는다. 합병연산의 최적화란 합병 시에 발생하는 복사연산과 소거연산의 횟수를 줄이는 것을 의미한다.

순차쓰기용 로그블록의 교환연산 : 순차쓰기용 로그블록의 모든 섹터가 순차적으로 다 기록된 경우 기존의 BAST 기법과 마찬가지로 교환연산을 한다.

순차쓰기용 로그블록의 합병연산 : 순차쓰기용 로그블록에 대해서는 BAST 방식에서 사용된 합병방식을 사용하지 않는다. BAST 방식의 경우에는 로그블록에 저장되는 섹터의 순서를 고려하지 않았기 때문에, 두 번의 소거연산과 32번의 섹터 복사가 필요한 합병연산을 수행해야 한다[2]. 하지만, FAST에서는 순차쓰기용 로그블록에 기록된 섹터들이 모두 최신의 데이터이고 또한 섹터들이 자기 위치에 기록되어 있으므로, 로그블록에서 비어있는 섹터들만 데이터블록에서 복사를 하면 된다. BAST 기법과 비교해서 복사연산의 횟수가 줄게 되며, 또한 데이터블록의 소거만 필요하므로 소거연산도 한번만 발생한다. (그림 6) (a)에서 순차쓰기용

로그블록에 섹터 4와 6이 있을 때, 데이터블록으로부터 최신의 섹터 5를 복사한다. 이 로그블록을 데이터블록으로 교환하고, 데이터블록은 소거하여 순차쓰기용 로그블록으로 설정한다.

임의쓰기용 로그블록의 합병연산 : 임의쓰기용 로그블록 그룹에 빈 공간이 없을 경우, (그림 4)의 원형 큐(circular queue) 방식으로 시작 로그블록을 합병연산의 대상으로 삼는다. 대상 로그블록에서는 서로 다른 논리 블록의 섹터들이 존재할 수 있기 때문에, 논리 블록의 개수만큼 합병연산이 발생한다. 매 합병연산마다 임의쓰기용 로그블록 그룹에서 최신의 섹터들을 예비 블록에 복사하고 그 다음으로 데이터블록에서 나머지 최신 섹터들을 복사한다. 특히, 임의쓰기용 로그 복사 그룹으로부터 최신 섹터들을 찾을 때는 원형 큐의 마지막 로그블록부터 역탐색(backward search)을 수행한다.[12] (그림 6) (b)는 쓰기연산에 필요한 임의쓰기용 로그블록의 빈공간을 만들기 위한 합병연산의 예를 보여준다. 그림에서 합병연산의 대상은 첫 번째 로그블록이며, 이 로그블록의 서로 다른 논리 블록의 개수는 두 개이므로(즉, (1, 3)과 (8, 10)) 두 번의 합병연산이 발생한다. 대상 로그블록의 섹터 1과 3에 대한 합병연산은 우선 임의쓰기용 로그블록 그룹에서 최신의 섹터 1과 3을 찾아서 예비 블록에 복사하고 나머지 최신 섹터들은 데이터블록에서 복사한다. 이때, 섹터 1이나 3과 같이 대상 로그블록 이외의 블록에서 최신 섹터가 발견이 되면, 사상 테이블에서 해당 섹터 번호를 -1로 설정하여 무효화(invalid) 시킨다. 이것은 추후에 섹터 -1이 포함되어 있는 블록이 합병 대상 로그블록이 되었을 경우, -1의 경우에 대해서는 합병연산을 무시하기 위해서이다.

위에서 설명한 합병연산의 비용 관점에서 FAST 기법과 BAST기법을 비교했을 때, FAST 기법은 다음 두 가지 이유로 성능 향상이 생긴다. 1) 합병연산 시점에서 임의쓰기용 로그블록들의 평균 활용률이 100%이므로, 평균 활용률이 낮은 BAST 기법과 비교해서 합병연산의 횟수가 줄어든다. 따라서 합병연산에서 필요한 복사연산과 소거연산의 횟수가 줄어든다. 2) BAST 기법에서 로그블록에 순차적으로 섹터들이 기록되다가 합병되는 경우에 비해 FAST 기법에서의 순차쓰기용 로그블록 합병연산에 비해 복사연산과 소거연산의 횟수가 작다. 4장의 성능 평가에서 BAST 기법 보다 FAST 기법이 우수한 이유는 이 두 가지 이유에 기인한다.

4. 성능 평가

본 장에서는 플래시 메모리의 전형적인 샘플 작업부하를 이용해서, BAST 와 FAST 두 기법의 성능을 분석한다. 로그블록의 수가 증가함에 따라서 BAST 와 FAST 두 기법의 성능을 전체소요시간과 소거회수를 중심으로 비교하고자 한다. 그리고, 극단적으로 임의쓰기로만 이루어진 쓰기부하[13]에 대해서도 FAST기법의 성능을 분석한다. 우선 4.1절에서는 실험환경을 설명하고, 4.2절에서는 주요한 샘플 작업부하들에 대한 BAST와 FAST 두 기법의 실험결과를 보여주고, 4.3 절에서는 FAST 기법이 더 좋은 성능을 보이는 이유에 대해 분석하고자 한다.

4.1 실험 환경

실험에서는 <표 2>와 같이 다섯 개의 작업부하들을 사용하였으며, 각 작업부하들은 논리섹터번호를 인자로 갖는 쓰기연산의 집합이다. <표 2>의 '패턴 E' 작업부하를 제외한 나머지 작업부하들은 [2]에서 사용한 것과 동일하며, 이들에 대한 자세한 설명은 [2]를 참조하기 바란다.

〈표 2〉 실험 환경

프로그래밍 언어	ANSI C		
운영체제	Red Hat Enterprise Linux AS release 3 (Taroon)		
컴파일러	gcc version 3.2.3		
추출 데이터	패턴	설 명	쓰기 연산 수
	A	Canon PowerShot G1의 디지털 카메라에서 추출	2,199,200
	B	Kodak DC290의 디지털 카메라에서 추출	3,144,800
	C	리눅스 운영체제 환경에서 추출(Andrew benchmark[6])	398,000
	D	심비안 운영체제 환경에서 추출	404,900
	E	자체 생성한 극단적 임의쓰기로만 구성된 작업부하	150,000

4.2 다양한 작업부하들에 대한 BAST와 FAST의 실험 결과

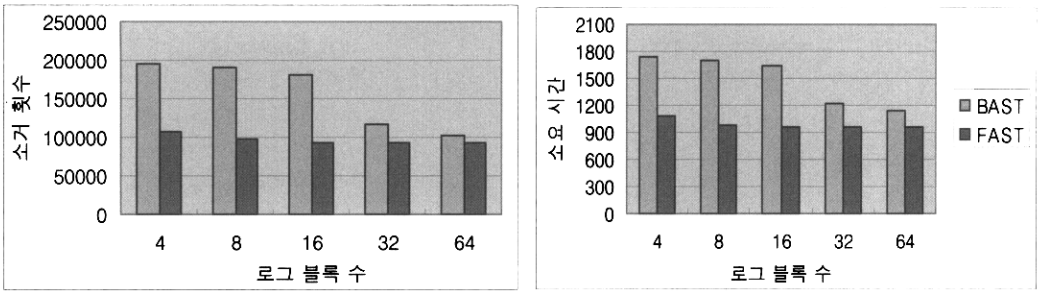
이 절에서는, 1장의 다섯 가지 추출(trace) 작업부하들에 대해 BAST 기법과 FAST 기법을 적용했을 경우 로그블록의 소거회수 측면과 전체 소요시간 측면에서 측정된 결과를 제시한다. 이 절의 성능 결과에 대한 분석은 다음 절에서 설명한다.

4.3 BAST와 FAST 간의 성능 분석

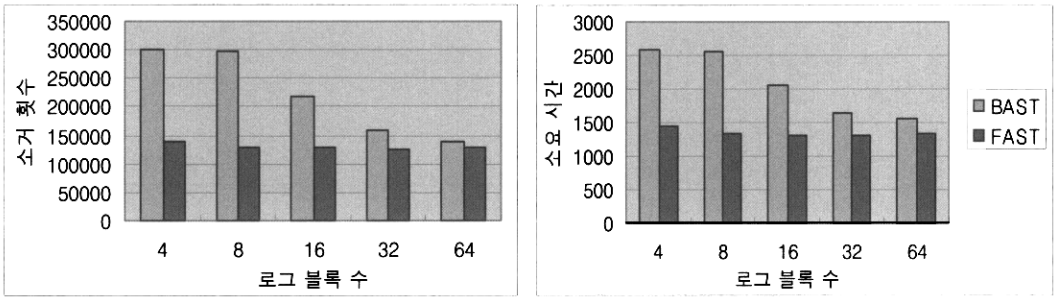
본 절에서는 4.2절에서 제시한 성능 측정 결과들에 대한 전반적인 성능 분석을 설명한다. 앞의 실험 결과에서도 볼 수 있듯이, 모든 실험 결과에서 FAST방식은 BAST방식에 비해 우수한 성능을 보여준다. 3장에서 설명했듯이, 성능의 차이는 두 방식간의 로그블록의 평균 활용률의 차이에 기인한다. 아래에서는 각 패턴에 대해 두 방식 간의 성능 평가

12) 물론, 임의쓰기용 로그블록 그룹에서 최신 섹터를 찾을 때는 로그블록들을 탐색하는 것이 아니라 임의쓰기용 사상 테이블을 이용한다.

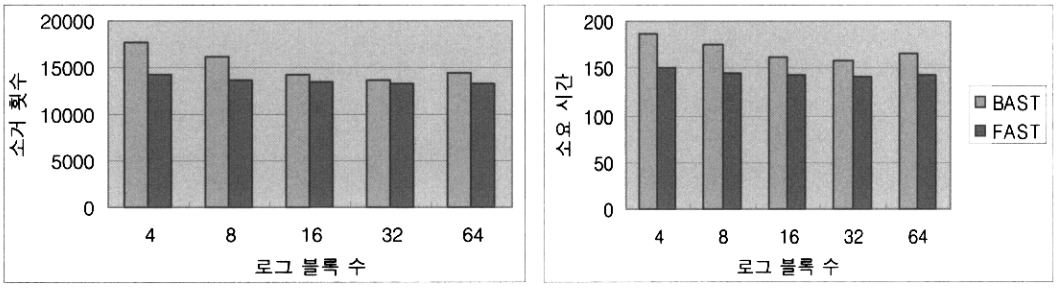
13) 우리는 향후에플래시 메모가 디지털카메라 등과 같이 대량의 순차쓰기가 많이 요구되는 기기 외도 활용되기 위해서는 임의쓰기로 구성된 쓰기부하에 대해서는 좋은 성능을 보이는 FTL 기법이 필수이라고 생각한다.



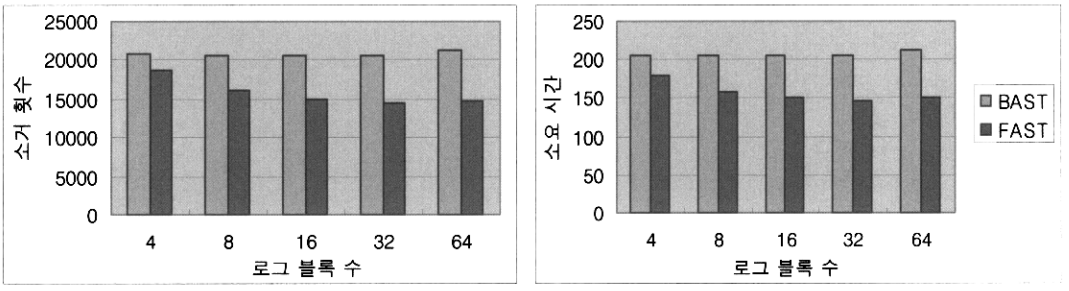
(a) 패턴 A



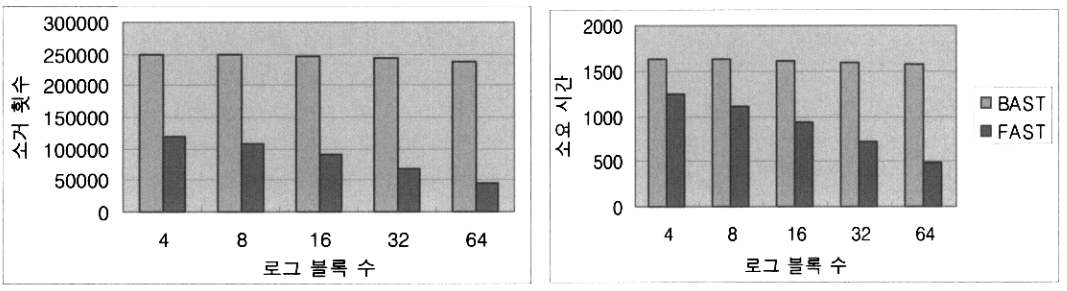
(b) 패턴 B



(c) 패턴 C



(d) 패턴 D



(e) 패턴 E

(그림 7) 작업부하에 따른 실험 결과

를 한다.

우선, (그림 7)의 패턴 E와 같이 작업부하에 임의쓰기만 포함하고 있는 극단적인 경우 FAST 방식의 장점을 극적으로 잘 보여준다. 그림에서 로그블록의 수가 증가하더라도 BAST 방식의 성능에는 변함이 없는 반면, FAST 방식은 성능이 향상됨을 알 수 있다. 그 이유는 다음과 같다. 극단적인 임의쓰기 환경에서는, BAST방식의 경우 하나의 로그블록에는 같은 논리블록 주소를 갖는 섹터들만이 들어갈 수 있으므로 계속해서 비용이 비싼 로그블록의 합병연산이 일어나게 된다. 반면, FAST 방식에서는 극단적인 임의쓰기가 일어나더라도 임의쓰기용 로그블록 그룹에 섹터들을 모두 채우고 난 후 더 이상의 빈 공간이 없을 때 합병연산이 일어나기 때문에, BAST 방식에 비해 매우 작은 수의 소거연산만 필요하다.

다음으로, 패턴 A와 B는 디지털 카메라로부터 추출되었기 때문에 다량의 순차쓰기를 포함하고 있다. (그림 7)에서 이 두 패턴 대한 실험 결과는 FAST방식이 임의쓰기뿐만 아니라 순차쓰기에서도 좋은 성능을 나타냄을 보여준다(로그블록의 수가 4일 때 BAST 방식보다 약 80% 향상). 또한, 패선 A와 B에 대한 실험 결과에서, BAST방식의 경우 로그블록의 개수가 증가할수록 그 성능이 FAST방식의 그것에 근접함을 알 수 있다. 이는 한 시간 윈도우 동안 기록된 섹터들에 대해 논리 블록의 수를 계산했을 때 그 값이 할당된 로그블록의 개수 이하이면 BAST방식도 충분한 로그블록의 활용률을 보이기 때문이다. <표 3>은 패턴 A와 B에 대한 실험 결과에서 부차적으로 얻어진 FAST와 BAST 각 방식에서 일어나는 교환연산의 수이다. 표에서 FAST와 BAST 두 방식에서 발생하는 교환연산의 수는 거의 동일함을 알 수 있다. 이것은 FAST방식에서 사용되는 순차쓰기용 로그블록이 BAST 방식과 같이 다량의 순차쓰기에도 잘 대처함을 알 수 있다.

〈표 3〉 패턴 A, B에 대한 FAST와 BAST에서의 교환연산의 수

패턴 A			패턴 B		
로그블록 수	BAST	FAST	로그블록 수	BAST	FAST
4	49,213	48,529	4	79,214	78,472
8	49,212	48,628	8	78,910	78,734
16	49,278	48,628	16	78,664	78,720
32	48,757	48,628	32	78,470	78,770
64	48,083	48,693	64	77,759	78,794

마지막으로, 패턴 C와 D의 경우 한 시간 윈도우 동안 임의쓰기에 의해 기록된 섹터들에 대한 논리 블록의 개수를 계산했을 때 그 값이 크지 않기 때문에(3~4개 이하), 로그블록의 개수에 상관없이 BAST와 FAST 두 방식의 성능에 큰 차이를 보이지 않는다. 하지만 FAST 방식이 BAST 방식보다 우위에 있다.

실험 결과에서 흥미로운 점은, BAST방식의 경우 로그블록 개수에 따라 성능이 민감하게 반응하는 반면, FAST방식에서는 로그블록의 개수가 작아도 잘 동작하고 그 값이 증가할수록 선형적으로 추가적인 성능 개선이 이루어짐을 알 수 있다. 즉, FAST방식은 작업부하의 패턴에 상관없이 좋은 성능을 보이고, 로그블록을 더 많이 사용함으로써 선형적인 성능 개선을 보이는 것이 아주 좋은 장점이라 할 수 있다.

5. 결론 및 향후 연구

FAST는 임의쓰기용 로그블록과 순차쓰기용 로그블록이라는 두 가지의 로그블록을 사용함으로써 임의쓰기와 순차쓰기와 같은 다양한 패턴의 쓰기에 대해서 높은 성능을 나타낸다. 전체 실험결과에서 볼 수 있듯이 FAST는 로그블록을 계속적으로 증가시켜도 큰 성능향상을 일어나지 않는다. 하지만 FAST는 4~8개 정도의 로그블록만으로도 30개 이상의 로그블록을 사용하는 BAST보다 우수한 성능을 보인다.

향후 연구로는 임의쓰기용 로그블록이 전부 찼을 경우, 지금처럼 무조건 큐의 가장 앞에 있는 블록만을 합병연산의 대상으로 하지 않고 좀 더 효과적인 방법을 찾아 볼 수 있을 것이다. 또한 데이터의 안정성과 일관성을 유지할 수 있는 트랜잭션 측면의 원자성(Atomicity)를 보장할 수 있는 방안 또한 과제라 할 수 있다.

참 고 문 헌

[1] Tae-Sun Chung, Dong-Joo Park, Sang-Won Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song,"System Software for Flash Memory: A Survey", Submitted for publication, 2004. http://www.mju.ac.kr/~tschung.

[2] 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.

[3] 배영현 외 4인, "플래시 메모리 파일 시스템을 위한 효율적인 소거 횟수 평준화 기법", 한국정보과학회 가을 학술발표논문집, pp.580-582, 2004년 10월.

[4] Amir Ban, "Flash File System", United States Patent, No. 5,404,485, 1995.

[5] Amir Ban, "Flash File System Optimized for Page-mode Flash Technologies", United States Patent, No.5,425,937, 1999.

[6] J. K. Howard, S. Menees, D. Michols, M. Satyanarayanan, N. Sidebotham, and M. West, "Scale and Performance in a Distributed File System", ACM Transaction on Computer Systems, Vol.6, No. Feb., 1988.

[7] Samsung Electronics, "Nand Flash Memory & Smartmedia Data Book", 2004.

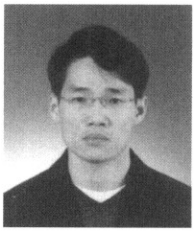
[8] Petro Estakhri and Berhanu Iman, "Moving Sequential

Sectors within A Block of Information in A Flash Memory Mass Storage Architecture", United States Patent, No. 5,930,815, 1999.

[9] Takayuki Shinohara, "Flash Memory Card with Block Memory Address Arrangement", United States Patent, No. 5,905,993, 1999.

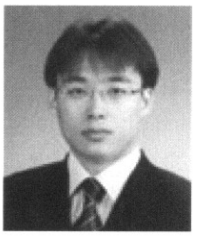
[10] Bum Soo Kim and Gui Young Lee, "Method of Driving Remapping in Flash Memory and Flash Memory Architecture Suitable Therefore", United States Patent, No. 6,381,176, 2002.

박 동 주



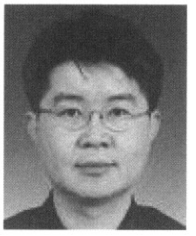
e-mail : djpark@ssu.ac.kr
1995년 서울대학교 컴퓨터공학과(학사)
1997년 서울대학교 컴퓨터공학과(공학석사)
2001년 서울대학교 컴퓨터공학부(공학박사)
2001년~2003년 삼성전자 책임연구원
2004년~현재 숭실대학교 컴퓨터학부 전임강사
관심분야 : 플래시 메모리, 내장형 소프트웨어, 멀티미디어 데이터베이스 등

최 원 경



e-mail : skkutop@skku.edu
2005년 성균관대학교 정보통신공학부(학사)
2005년~현재 삼성전자 연구원

이 상 원



e-mail : wonlee@ece.skku.ac.kr
1991년 서울대학교 컴퓨터공학과(학사)
1994년 서울대학교 컴퓨터공학과(석사)
1999년 서울대학교 컴퓨터공학과(박사)
1999년~2001년 한국오라클
2001년~2002년 이화여대 컴퓨터학과
2002년~현재 성균관대학교 정보통신공학부 조교수
관심분야 : XML, DB 튜닝, Data Warehouse/Data Mining