낸드 플래시 메모리 상에서 쓰기 패턴 변환을 통한 효율적인 B-트리 관리
(Efficiently Managing the B-tree using Write Pattern Conversion on NAND Flash Memory)

박 동 주 [†] 최 해 기 [††]
(Dong-Joo Park) (Haegi Choi)

요 약 플래시 메모리는 하드디스크와 다른 물리적 특성을 가진다. 대표적으로 읽기연산과 쓰기연산의 비용이 다르고, 덮어쓰기(overwrite)가 불가능하여 소거연산(erase)이 선행되어야 한다. 이러한 물리적 제약을 소프트웨어적으로 보완해주기 위해서, 플래시 메모리를 사용하는 시스템은 대부분 플래시 변환 계층(Flash Translation Layer)을 사용한다. 현재까지 효율적인 FTL 기법들이 제안되었으며, 이들은 임의쓰기(random writes) 패턴보다 순차쓰기(sequential writes) 패턴에 훨씬 더 효율적으로 동작한다. 본 논문에서는 플래시 메모리 상에서 B-트리 인덱스를 효율적으로 생성, 유지하기 위한 새로운 기법을 제안한다. B-트리에 키의 삽입, 삭제, 수정 등의 연산을 수행하면 FTL에 비효율적인 임의쓰기 패턴을 많이 발생시키며, 결국 B-트리 인덱스 유지 비용이 커지게 된다. 제안하는 기법에서는 B-트리에서 발생되는 임의쓰기 패턴을 먼저 플래시 메모리의 쓰기 버퍼에 추가쓰기(append writes) 패턴으로 변환하여 저장하고, 추후 이를 FTL에 효율적인 순차쓰기 패턴으로 FTL에 전달한다. 다양한 실험을 통해 제안하는 기법이 기존의 기법보다 플래시 메모리 I/O 비용 측면에서 우수하다는 것을 보인다.
키워드 : 플래시 메모리, 임베디드 소프트웨어, 플래시 변환 계층, B-트리, 쓰기 패턴

Abstract Flash memory has physical characteristics different from hard disk where two costs of a read and write operations differ each other and an overwrite on flash memory is impossible to be done. In order to solve these restrictions with software, storage systems equipped with flash memory deploy FTL(Flash Translation Layer) software. Several FTL algorithms have been suggested so far and most of them prefer sequential write pattern to random write pattern. In this paper, we provide a new technique to efficiently store and maintain the B-tree index on flash memory. The operations like inserts, deletes, updates of keys for the B-tree generate random writes rather than sequential writes on flash memory, leading to inefficiency to the B-tree maintenance. In our technique, we convert random writes generated by the B-tree into sequential writes and then store them to the write-buffer on flash memory. If the buffer is full later, some sequential writes in the buffer will be issued to FTL. Our diverse experimental results show that our technique outperforms the existing ones with respect to the I/O cost of flash memory.
Key words : Flash memory, Embedded software, FTL, B-tree, Write pattern

† 정 회 원 : 숭실대학교 대학원 컴퓨터학과 교수
djpark@ssu.ac.kr
†† 정 회 원 : 숭실대학교 컴퓨터학부
seastand@naver.com
논문접수 : 2008년 4월 30일
심사완료 : 2009년 7월 30일

Copyright©2009 한국정보과학회 : 개인 목적이나 교육 목적인 경우, 이 저작물의 전체 또는 일부에 대한 복사본 혹은 디지털 사본의 제작을 허가합니다. 이 때, 사본은 상업적 수단으로 사용할 수 없으며 첫 페이지에 본 문구와 출처를 반드시 명시해야 합니다. 이 외의 목적으로 복제, 배포, 출판, 전송 등 모든 유형의 사용행위를 하는 경우에 대하여는 사전에 허가를 얻고 비용을 지불해야 합니다.
정보과학회논문지: 시스템 및 이론 제36권 제6호(2009.12)

1. 서 론

플래시 메모리는 크기가 작고 충격에 강하며 하드디스크에 비해 빠른 데이터 접근 속도를 가진다. 또한 소비전력이 적고 무게가 가벼우며 비휘발성의 성질 때문에 PDA, MP3, 핸드폰과 같은 이동기기의 저장장치로 널리 사용되고 있다. 최근에는 메모리 집적도의 향상으로 저장크기가 대용량화 되어감에 따라, 이동기기뿐만 아니라 개인용 컴퓨터나 노트북에서도 보조기억장치로 활용되고 있으며 플래시 메모리의 사용 범위가 점차 확

대되고 있다.

플래시 메모리는 하드디스크와는 다른 물리적인 특징을 가진다. 즉, 데이터의 읽기연산과 쓰기연산의 비용이 다르고, 하드디스크와 달리 덮어쓰기(overwrite)가 불가능하여 이를 위해 소거연산(erase)이 수반되어야 한다. 플래시 메모리의 물리적 제약을 소프트웨어적으로 해결하기 위해 플래시 메모리 저장 시스템에서는 플래시 변환 계층(Flash Translation Layer, FTL)을 사용한다 [1-3]. FTL은 기존 파일 시스템과의 호환성을 위해 플래시 메모리를 하드디스크처럼 사용할 수 있게 도와준다. 따라서 플래시 메모리에 덮어쓰기가 발생하더라도 FTL은 사용자에게 투명하게(transparently) 덮어쓰기가 된 것처럼 보여준다.

최근 플래시 메모리의 대용량화로 인해 플래시 메모리 저장 시스템은 더욱더 데이터를 효율적으로 검색하기 위한 B-트리 인덱스를 필요로 한다. 하드디스크 기반 B-트리 인덱스에서는 키의 삽입 또는 삭제 시 해당 인덱스 페이지를 버퍼에 적재하여 연산을 수행한 후 그 페이지를 하드디스크 상에 덮어쓰기를 수행하면 된다. 그러나 플래시 메모리에서는 덮어쓰기가 불가능하기 때문에 기존의 하드디스크 기반 B-트리 인덱스를 그대로 플래시 메모리에 적용시킬 수 없다. 플래시 메모리에서의 덮어쓰기가 발생하면 이를 해결하기 위해 FTL은 여러 번의 쓰기연산과 한번의 소거연산을 수행해야 하므로 그 비용이 매우 크다.[1] 따라서 플래시 메모리용 B-트리 인덱스를 개발할 때 B-트리 수정 시 발생하는 수정된 노드에 대한 덮어쓰기 연산을 효율적으로 처리하는 게 중요하다.

최근 하드디스크 기반 B-트리 인덱스를 플래시 메모리에 적용시키려는 연구가 이루어지고 있다[1,4]. 최초의 연구로서는 BFTL 기법[4]이 있으며, 이 기법은 B-트리에 키를 삽입하면 해당 노드를 수정한 후 바로 플래시 메모리에 덮어쓰기를 수행하는 것이 아니라 메인 메모리의 버퍼에 삽입된 키 값을 저장하고 추후 여러 개의 키 값들을 모아서 한번에 플래시 메모리에 저장한다. 따라서 키 삽입마다 수정된 노드에 대한 덮어쓰기를 수행하는 것이 아니라 여러 개의 키 삽입에 대해 한번의 쓰기연산으로 처리함으로써 덮어쓰기로 인한 비용을 크게 줄일 수 있다. 그러나 논리적으로 동일한 노드에 존재하는 키 값들이 하나의 물리적 페이지에 저장되는 것이 아니라 여러 개의 물리적 페이지에 분산 저장되기 때문에 키 검색에서는 그 성능이 매우 떨어진다.

BOF 기법[1]에서는 BFTL의 단점인 검색 성능을 향상 시키기 위해 버퍼에 존재하는 동일한 노드의 키 값들을 같은 물리적 페이지에 저장한다. 따라서 키 검색을 위한 B-트리 노드 탐색(traversing) 시 탐색 노드당 하나의 페이지만 읽게 함으로써 하드디스크 기반 B-트리 인덱스의 검색 성능을 보장해 준다. 그러나 이 기법은 B-트리 수정 시 많은 덮어쓰기를 발생시켜 B-트리 유지 비용을 증가시키는 단점을 가지고 있다.

본 논문에서는 플래시 메모리 상에서 B-트리 인덱스를 효율적으로 생성, 유지하기 위한 새로운 기법을 제안한다. 일반적으로 플래시 메모리 저장 시스템은 플래시 메모리의 상위 계층으로서 FTL을 두고 있으며 그 위에 B-트리 인덱스가 존재한다. 따라서 B-트리에 대한 키의 삽입 또는 삭제 등의 연산은 결국 FTL에서 제공하는 쓰기 인터페이스를 호출하여 수정된 노드를 플래시 메모리에 반영한다. B-트리에서 호출되는 FTL의 쓰기 함수는 인자로서 논리적 페이지 번호를 가지며, B-트리에서 전달되는 논리적 페이지 번호 시퀀스(sequence)가 순차쓰기(sequential writes) 보다는 임의쓰기(random writes) 패턴을 보인다. 그러나 본 논문의 예비 실험 결과에 의하면 FTL은 임의쓰기 패턴에서 그 성능이 현저히 떨어진다. 따라서 본 논문에서 제안하는 기법에서는 B-트리에서 발생되는 임의쓰기 패턴을 먼저 플래시 메모리의 쓰기버퍼에 논리 블록 번호로 구분되는 그룹 별로 저장하고, 추후 각 그룹 별로 순차쓰기 패턴으로 변환하여 FTL에 전달한다. 따라서 B-트리에서 발생하는 임의쓰기 패턴을 FTL에 효율적인 순차쓰기 패턴으로 변환하여 쓰기 연산을 수행함으로써 B-트리 인덱스 유지 비용을 크게 줄일 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 플래시 메모리 상에서의 B-트리 적용에 대한 관련 연구를 서술한다. 3장에서는 본 논문의 이해를 위해 예비 지식을 설명하며 4장에서는 제안 동기를, 5장에서는 본 논문에서 제안하는 기법을 소개한다. 6장에서는 실험결과를 분석하고, 마지막으로 7장에서 결론을 맺는다.

2. 관련 연구

플래시 메모리의 물리적 특성을 고려하여 B-트리를 저비용으로 관리하기 위한 연구로서 BFTL[4]과 BOF[1]가 존재한다. BFTL과 BOF는 각각 B-트리의 생성 비용과 검색 비용에 초점을 두고 제안되었다.

BFTL은 덮어쓰기가 되지 않는 플래시 메모리의 특성을 고려하여 서로 다른 노드 안의 새로 업데이트 된 키 값의 정보를 비어있는 섹터에 같이 저장한다. 이러한 정책으로 인해서 동일한 노드 안에 키 값들이 여러 섹터에 분산되어 저장되기 때문에 각각의 노드 정보를 관리하기 위해 노드 변환 테이블(Node Translation Table)을 사용한다. 이러한 기법은 하나의 노드를 검색하기 위해

1) 본 논문의 표 2 참조

서는 여러 개의 섹터를 읽어야 되기 때문에 B-트리의 검색 성능을 크게 악화시키는 문제점을 가지고 있다. 또한 추후에 동일한 노드 안의 키 값들이 일정 개수 이상의 섹터에 분산되어 저장되었을 경우, 이 노드 안의 키 값을 모아서 하나의 동일한 섹터에 저장시키는데 이러한 작업들(Compact, Garbage Collection)은 플래시 메모리에서 많은 비용을 초래시킨다. 또한 이러한 BFTL 기법은 노드 변환 테이블을 RAM에서 유지하기 때문에 B-트리의 크기가 커졌을 경우 필요로 하는 RAM 비용을 크게 증가시키고 부가적인 비용(Compact, Garbage Collection)도 비대해지기 때문에 실효성이 떨어진다.

BOF는 BFTL에서 많은 비용을 발생시키는 노드 변환 테이블을 없애고 검색 성능을 향상시키기 위해 동일한 노드의 키 값은 항상 같은 섹터에 저장하는 기법을 사용한다. 즉 하나의 노드를 검색하기 위해서는 한 개의 섹터를 읽으면 된다. 이러한 기법은 비록 B-트리 생성 시 BFTL에서 발생하지 않는 덮어쓰기 연산을 발생시켜 생성 비용을 증가시키지만 BFTL의 단점으로 여겨진 검색 성능을 크게 향상시키는 결과를 가져온다. 또한 노드 변환 테이블을 사용하지 않기 때문에 B-트리의 크기가 커졌을 경우에 발생되는 부가적인 비용(RAM, Compact, Garbage Collection)도 존재하지 않는다. 이러한 이유 때문에 BOF는 BFTL 보다 더 유용한 기법이라 할 수 있으며 따라서 이러한 BOF 기법의 기반에서 B-트리 생성 비용을 낮출 수 있는 방안을 연구해야 한다.

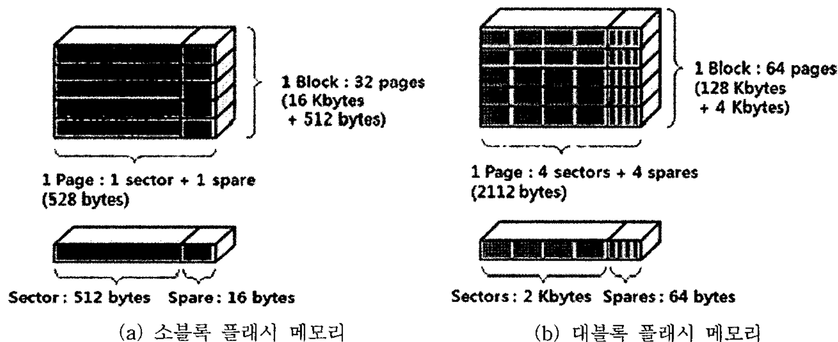
본 논문의 주요 아이디어는 B-트리에 키를 삽입할 때 발생하는 임의쓰기 패턴을 순차쓰기 패턴으로 변환시켜서 전체 B-트리 수정 비용을 줄이는 것이다. 이러한 쓰기 패턴의 변환은 로그 기반 파일 시스템(Log-Structured File System)[5]의 페이지 쓰기 기법과 유사하다. 로그 기반 파일 시스템은 페이지 쓰기 시 하드디스크의 탐색 시간(seek time)을 줄이기 위해 하드 디스크라는 로그에 추가 모드(append-only)로 페이지를 저장한다. 그러나 본 연구에서는 FTL에 유리한 쓰기 패턴이 순차쓰기라는 것을 실험을 통해 밝혀내고, 이를 바탕으로 새로운 B-트리 쓰기 기법을 제안한 점에서 다르다. 기존의 로그 기반 파일 시스템을 플래시 메모리에 적용한 파일 시스템으로 JFFS[6]나 YAFFS[7] 등이 있으나 기본 아이디어는 로그 기반 파일 시스템과 다르지 않다.

3. 예비 지식(Preliminaries)

3.1 플래시 메모리

NAND 플래시 메모리는 크게 "소블록 플래시 메모리"와 "대블록 플래시 메모리"로 나눌 수 있다. 그림 1(a)는 소블록 플래시 메모리로서, 하나의 블록은 32개의 페이지로 구성되며, 하나의 페이지는 데이터를 저장하는 512 바이트 크기의 섹터(sector) 영역과 부가정보(ECC 등) 저장을 위한 16 바이트 크기의 예비(spare) 영역으로 나뉜다. 대블록 플래시 메모리는 소블록 플래시 메모리에 비해 더 큰 단위의 페이지와 블록을 가지며, 그 구성은 그림 1(b)와 같다. 하나의 블록은 64개의 페이지를 가지며, 각 페이지는 네 개의 섹터와 네 개의 예비 영역으로 구성된다. 대블록 플래시 메모리는 페이지 단위뿐만 아니라 섹터 단위의 읽기/쓰기 연산도 지원한다. 따라서 페이지를 섹터 단위로 사용하는 경우 한 페이지에 최대 네 번의 쓰기연산이 가능하다.

하드디스크와는 달리 플래시 메모리의 섹터는 덮어쓰기가 되지 않는다. 따라서 데이터가 이미 존재하는 섹터에 새로운 데이터를 기록하기 위해서는 그 섹터를 포함하는 블록을 소거한 뒤 해당되는 섹터에 새로운 데이터를 저장해야 한다. 물론, 해당 블록을 소거하기 전에 그 블록의 나머지 유효한 섹터들을 다른 블록에 임시로 저장해 두고 소거연산이 끝난 후 다시 그것들을 해당 블록으로 복사해 와야 한다. 따라서 덮어쓰기에 수반되는 비용이 매우 클 수밖에 없다. 이러한 덮어쓰기를 저비용으로 처리하기 위해 파일 시스템과 같은 응용과 플래시 메모리 사이에는 FTL이 존재한다.



(a) 소블록 플래시 메모리 (b) 대블록 플래시 메모리

그림 1 플래시 메모리 구성

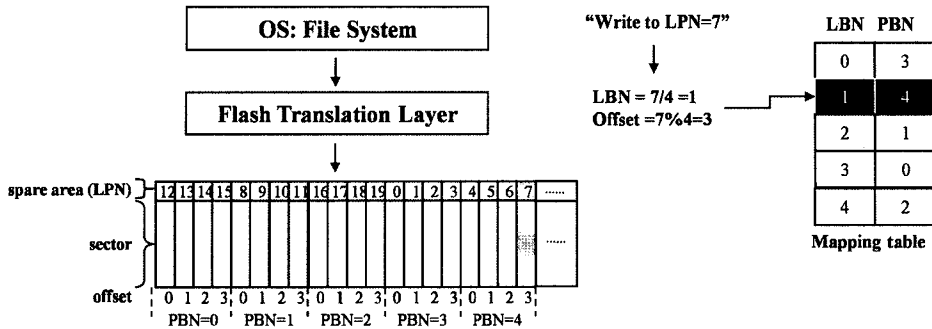


그림 2 FTL에서의 블록 사상 기법

3.2 FTL

플래시 메모리는 덮어쓰기가 되지 않고 데이터의 소거 단위와 읽기·쓰기 단위가 다르다. 이러한 물리적 제약 사항을 소프트웨어적으로 극복하기 위해서 플래시 메모리와 기존 파일 시스템 사이에는 FTL이 존재한다. FTL은 플래시 메모리를 하드디스크처럼 사용할 수 있게 도와준다. 예를 들면, 기존의 파일 시스템에서 플래시 메모리에 덮어쓰기를 발생시키더라도 FTL이 덮어쓰기가 발생한 데이터를 해당 섹터가 아닌 비어있는 다른 섹터에 저장함으로써 마치 파일 시스템에서는 덮어쓰기가 이루어진 것처럼 보여준다.

플래시 메모리 탑재 시스템에서 기존 파일 시스템(FAT 또는 Linux file system)을 그대로 사용하기 위해서는 사상테이블(address mapping table)이 필요하며, FTL에서 유지 관리한다. 사상테이블이란 파일 시스템의 읽기 또는 쓰기 연산에서 사용하는 논리 페이지 번호(Logical Page Number = LPN)를 플래시 메모리의 물리 페이지 번호(Physical Page Number = PPN)로 사상하여 놓은 것이다. 사상의 단위가 페이지일 경우 유지해야 할 테이블의 크기가 매우 크므로, 일반적으로 FTL에서는 블록 단위의 사상테이블을 사용한다. 이는 논리 블록 번호(Logical Block Number = LBN)를 플래시 메모리의 물리 블록 번호(Physical Block Number = PBN)로 사상한 테이블로서, FTL은 파일 시스템으로부터 내려온 논리 페이지 번호에 대응되는 물리 페이지 번호를 다음과 같은 방법으로 찾는다. 먼저 논리 페이지 번호를 통해 논리 블록 번호와 오프셋(offset)을 구하고, 그 다음 사상테이블에서 논리 블록 번호에 대응되는 물리 블록에서 오프셋만큼 떨어진 물리 페이지 번호를 찾는다.[2)]

이와 같은 블록 단위의 사상은 그림 2와 같이 이루어진다. 그림 2는 플래시 메모리의 블록은 4개의 페이지로 구성된다고 가정한다. 파일 시스템이 데이터를 쓰려고 하는 논리 페이지 번호가 LPN=7일 때 논리 블록 번호 LBN=1이고 오프셋은 3이 된다. 따라서 사상테이블에서 논리 블록 번호 LBN=1에 대응되는 물리 블록 번호 PBN=4를 얻고 최종적으로 이 물리 블록에서 오프셋 3에 해당하는 페이지가 논리 페이지 번호 LPN=7에 사상되는 물리 페이지이다.

3.3 B-트리의 논리적 수정에 따른 물리적 저장

B-트리에 키가 삽입(또는 삭제, 수정)되면 B-트리는 해당 노드를 수정한 후 FTL의 쓰기연산을 호출하고, FTL은 최종적으로 그 노드에 대응되는 플래시 메모리의 페이지에 수정된 노드를 저장한다. 그림 3의 B-트리는 노드 A, B, C, D로 구성되어 있으며, 각 노드는 플래시 메모리의 논리 페이지 0, 1, 2, 3에 저장되어 있다. 여기서 B-트리 노드는 플래시 메모리의 하나의 페이지에 저장된다고 가정한다.[3)] 만약 B-트리에 키 30이 삽입된다면 그 키는 논리적으로 노드 D에 저장된다. 수정된 노드 D를 물리적 페이지에 저장하기 위해서, B-트리는 FTL의 쓰기연산을 호출하는데 이때 논리 페이지 번호 LPN=3을 인자로 넘겨준다. FTL은 사상테이블을 이용하여 LPN=3에 대응되는 물리 페이지를 찾아서 수정된 노드를 저장한다. 그림 3에서 LPN=3에 대응되는 물리 페이지에 노드 D의 이전 키 값들이 저장되어 있으므로 FTL에서는 덮어쓰기가 발생한다.

4. 연구 동기

4.1 B-트리 쓰기 패턴 분석

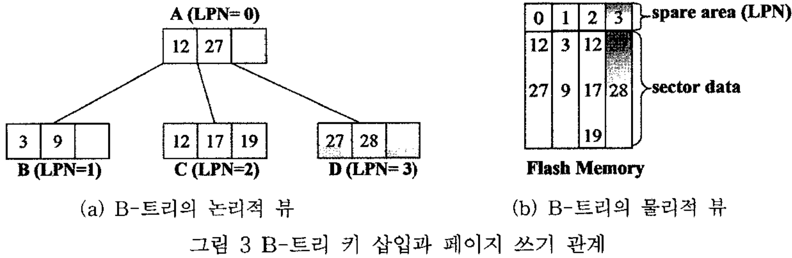
B-트리에 삽입 또는 삭제되는[4)] 키 값들이 임의적(random) 패턴을 따르면 수정되는 노드의 번호 순서도

---

2) LBN = LPN / #PAGES. offset = LPN % #PAGES. 여기서 #PAGES는 하나의 블록에 존재하는 페이지의 수이다.

3) 물론, B-트리 노드는 여러 개의 페이지에 저장될 수 있다.

4) B-트리 수정에는 키의 수정(update)도 포함되지만 이는 키의 삭제 후 삽입의 두 연산으로 처리할 수 있다.



(a) B-트리의 논리적 뷰 (b) B-트리의 물리적 뷰

그림 3 B-트리 키 삽입과 페이지 쓰기 관계

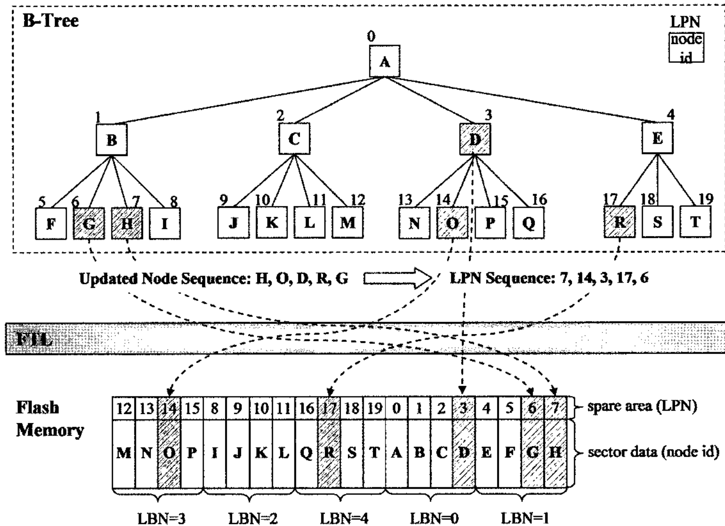


그림 4 B-트리 수정 시 발생하는 임의쓰기 패턴

같은 패턴을 보인다. 수정된 각 노드를 플래시 메모리에 반영하기 위해 B-트리는 FTL에 쓰기연산을 호출하며 이때 인자로서 이 노드에 대응되는 논리적 페이지의 번호를 전달한다. 결국 이러한 논리적 페이지의 번호 순서도 임의적 패턴을 보이며, 이를 임의쓰기 패턴이라고 부른다. 그림 4에서 B-트리에 임의의 순서로 키 값이 삽입 또는 삭제되었을 때, FTL에 임의쓰기 패턴이 발생함을 보여준다. 즉, 노드 H, O, D, R, G가 차례대로 수정되고 이를 반영하기 위해 B-트리는 차례대로 논리 페이지 7, 14, 3, 17, 6번에 대해 쓰기연산을 FTL에 발생시킨다. 여기서 논리 페이지 번호의 순서가 임의쓰기 패턴을 보인다. 참고적으로, 그림 4의 논리 페이지 번호의 순서가 total order 또는 partial order를 가지면 순차쓰기 패턴이고 random order를 가지면 임의쓰기 패턴이라고 한다.

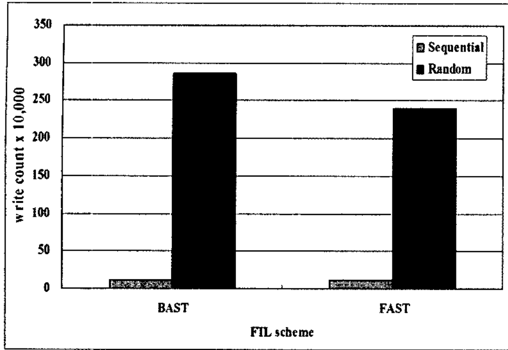
4.2 FTL에 효율적인 쓰기 패턴 분석

FTL은 로그버퍼(log buffer)을 유지하느냐에 따라 크게 로그버퍼와 비-로그버퍼 기반의 FTL로 나뉘어진다. 다양한 쓰기 패턴에 대해 일반적으로 로그버퍼 기반의 FTL이 비-로그버퍼 기반의 FTL보다 더 우수한 성능을 보인다. 로그버퍼 기반의 FTL 중 널리 알려진 기법으로 BAST 기법[2]과 FAST[3] 기법이 있으며, 이 중에서 효율적인 로그버퍼의 활용으로 인해 FAST 기법의 성능이 더 우수하다고 알려져 있다[3].

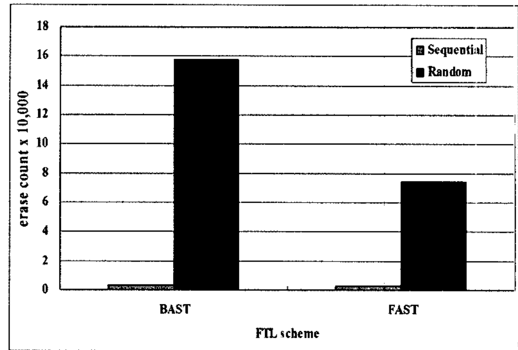
본 논문에서는 FAST 기법의 FTL에서 순차쓰기 패턴보다 임의쓰기 패턴의 비용이 훨씬 큼을 보이기 위해 다양한 실험을 수행하였다. 실험에서 순차쓰기 패턴용으로 FAST 기법에서 순차쓰기 패턴용으로 사용한 리눅스와 디지털 카메라 data set을, 임의쓰기 패턴용으로는 이들에 대해 임의쓰기 패턴으로 재구성한 data set을 사용하였다.[5] 그 결과, 그림 5와 6과 같이 임의쓰기 패턴이 순차쓰기 패턴보다 플래시 메모리에서 발생하는 비용이 훨씬 크다는 것을 확인할 수 있다. 여기서 비용은 주어진 data set이 갖는 논리 페이지 번호 시퀀스에 대해 FTL이 쓰기연산을 수행하였을 때 발생되는 전체 쓰기연산의 수와 소거연산의 수를 의미한다.[6]

5) 각 data set은 B-트리가 FTL에 쓰기연산을 호출할 때 인자로 전달하는 논리 페이지 번호의 시퀀스(sequence)이다.

6) 전체 쓰기연산의 수에는 덮어쓰기로 인해 발생되는 소거연산에서 발생되는 추가적인 쓰기연산도 포함된다.

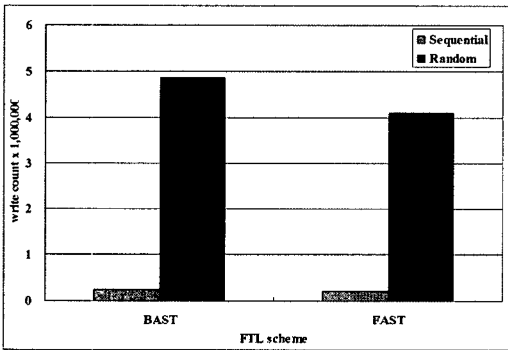


쓰기연산

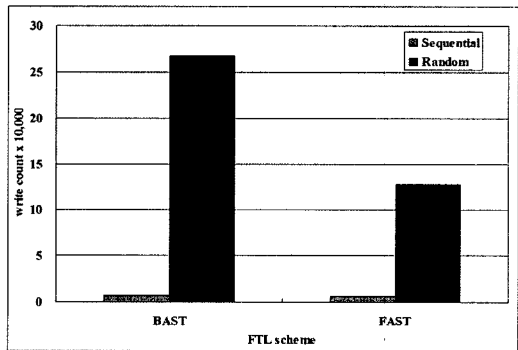


블록소거 연산

그림 5 리눅스 운영체제 data set



쓰기연산



블록소거 연산

그림 6 디지털 카메라 data set(Kodak DC290)

이 실험의 결과와 더불어 3.3절의 B-트리 쓰기 패턴 분석 결과를 종합해보면 B-트리에서 발생되는 임의쓰기 패턴은 결국 FTL에 비효율적인 쓰기 패턴이므로 플래시 메모리에서 발생되는 비용을 크게 증가시킨다. 따라서 B-트리에서 발생되는 임의쓰기 패턴을 FTL에 효율적인 순차쓰기 패턴으로 변환하면 플래시 메모리에서 발생되는 비용을 크게 감소시킬 수 있다. 다음 장에서는 이러한 실험 결과를 바탕으로 키 삽입 또는 삭제 시 B-트리를 효율적으로 관리할 수 있는 방법을 제시한다.

5. 쓰기 패턴 변환을 이용한 B-트리 관리 기법

본 장에서는 앞서 살펴본 4장의 실험 결과를 바탕으로 플래시 메모리 상에서 B-트리 수정 시 발생되는 임의쓰기 패턴을 FTL에 효율적인 순차쓰기 패턴으로 변환시켜 FTL에 전달함으로써 B-트리 유지 비용을 줄일 수 있는 방법을 제시한다.

5.1 전체 구조

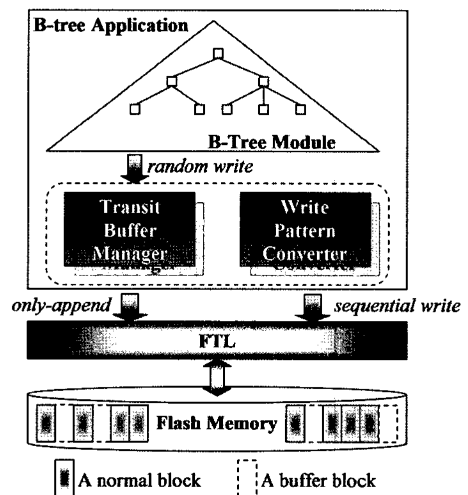


그림 7 전체적인 구조

그림 7과 같이, 전체적인 구조는 크게 B-트리 응용, FTL, 그리고 플래시 메모리로 나뉜다. B-트리 응용에는 기본적인 B-트리 모듈과 더불어 추가적으로 Transit buffer manager와 Write pattern converter를 포함한

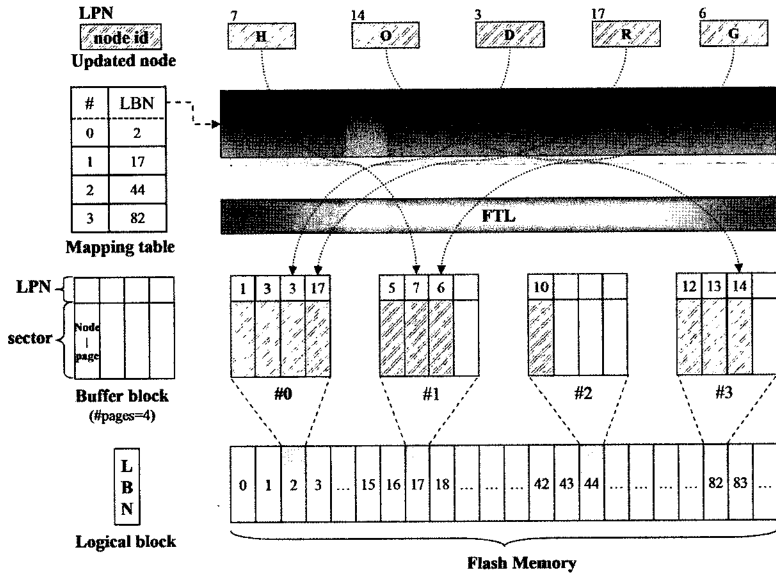


그림 8 Transit Buffer Manager

다. Transit buffer manager는 B-트리 수정 시에 수정 노드에 대응되는 페이지(이후 노드-페이지)를 플래시 메모리에 그대로 저장하는 것이 아니라 플래시 메모리 상의 버퍼에 노드-페이지를 임시로 저장하는 역할을 한다. Write pattern converter는 버퍼에 존재하는 노드-페이지들을 FTL에 효율적인 순차쓰기 패턴으로 변환하는 역할을 한다.

플래시 메모리 상의 버퍼는 임시 저장소로서 몇 개의 블록을 가지며, 각 블록에는 동일한 논리 블록 번호를 갖는 노드-페이지들이 저장된다. B-트리 수정에 따라 쓰기연산이 발생하면 각 노드-페이지는 버퍼의 해당 블록에 저장되며, 만약 이미 이 블록에 여유 공간이 없는 경우 Write pattern converter에 의해 이 블록에 존재하는 유효한 노드-페이지만을 FTL을 통해 플래시 메모리에 저장된다. 이때 이 블록의 페이지에 대해 FTL에 순차쓰기 패턴이 발생된다. 결국, B-트리의 임의쓰기 패턴에 의해 버퍼에 저장되는 노드-페이지들은 순차쓰기 패턴으로 변환되어 FTL에 전달된다. 따라서 4장에서 살펴본 실험 결과에 의하면 이 방식은 B-트리 수정에 따른 전체 유지 비용을 크게 줄일 수 있다.

5.2 Transit Buffer Manager

B-트리로부터 발생된 임의쓰기 패턴을 순차쓰기 패턴으로 변환하기 위한 중간 단계로, Transit buffer manager는 플래시 메모리의 일부 블록을 버퍼로 사용하며, 이 블록을 버퍼 블록이라 부른다. 본 절에서는 Transit buffer manager가 B-트리 수정 시 발생되는 수정 노드-페이지를 버퍼 블록에 저장하는 방법에 대해 설명한다.

B-트리 응용은 수정된 노드-페이지를 저장하기 위해 FTL에서 제공하는 쓰기연산 인터페이스를 바로 호출하지 않고 Transit buffer manager로 하여금 버퍼에 저장하도록 하며, 그 방법은 다음과 같다. Transit buffer manager는 수정된 노드-페이지의 논리 블록 번호 즉, LBN을 구하고 버퍼 블록 사상 테이블[7]에서 이것과 대응되는 버퍼 블록을 찾아 비어있는 최초의 페이지에 수정된 노드-페이지를 저장한다. 참고로, 이때 발생하는 쓰기연산은 추가쓰기(append) 모드, 즉 버퍼 블록의 첫 번째 페이지부터 순차적으로 다음 페이지를 사용하는 방식에 해당하므로 비용이 큰 덮어쓰기 연산을 유발시키지 않는다. 결국, 동일한 논리 블록 번호를 갖는 노드-페이지들은 동일한 버퍼 블록에 저장되며, 이 버퍼 블록이 다 차는 경우 Write pattern converter에 의해 순차쓰기 패턴으로 플래시 메모리에 저장된다.

그림 8은 플래시 메모리의 블록이 4개의 페이지로 구성되며, 버퍼 크기는 4개의 버퍼 블록에 해당한다고 가정한다, 그림 8과 같이, B-트리 응용에서 노드 H, O, D, R, G를 각각 수정하고 난 후 해당 수정 노드-페이지 7, 14, 3, 17, 6번에 대한 쓰기연산이 발생하였을 때 바로 FTL의 쓰기연산 인터페이스를 호출하지 않고 Transit buffer manager에 의해 플래시 메모리의 버퍼에 쓰여지는 과정을 보여준다. 예를 들면, 노드 H에 수

7) 6장의 실험 결과에 의하면 충분한 성능을 위해 요구되는 버퍼 블록의 수는 그리 크지 않으며(예를 들면, 16 또는 32), 따라서 B-트리 응용 내에 존재하는 버퍼 블록 사상 테이블 유지를 위한 오버헤드 또한 그리 크지 않다.

표 1 버퍼 블록 사상테이블

Buffer block #	LBN	Next Offset	LPN of node-page			
0	2	2	1	3	free	free
1	17	1	5	free	free	free
2	44	1	10	free	free	free
3	82	2	12	13	free	free

정이 발생하여 수정 노드-페이지 7번에 대한 쓰기연산이 발생하면, Transit buffer manager는 먼저 그 페이지에 대한 논리 블록 번호 즉, LBN = LPN / #PAGES = 7 / 4 = 1을 구한다. 그 다음 LBN=1에 버퍼 블록을 찾는데, 버퍼 블록 번호 = LBN % BUF_SIZE = 1 % 4 = 1이다. 표 1과 같은 버퍼 블록 사상테이블을 이용해서 버퍼 블록 1번에 최초로 비어있는 페이지의 오프셋이 0이라는 것을 알 수 있다. 따라서 최종적으로 노드 H의 수정 노드-페이지는 버퍼 블록 1번의 0번 페이지에 임시적으로 저장된다.

5.3 Write Pattern Converter

Transit buffer manager에 의해 동일한 논리 블록 번호를 갖는 노드-페이지들은 같은 버퍼 블록에 저장된다. Write pattern converter는 버퍼 블록 내에 존재하는 최신의 노드-페이지들에 대해 FTL로 순차쓰기 패턴의 쓰기 연산을 발생시킨다. 이 시점은 Transit buffer manager가 더 이상 여유 공간이 없는 버퍼 블록에 노드-페이지를 저장하려고 할 때이다. 이때 Write pattern converter는 그 버퍼 블록에 존재하는 노드-페이지들 중에서 가장 최신의 노드-페이지들을 순차쓰기 패턴으로 FTL에 쓰기 연산을 호출한다. 예를 들면, 그림 9에서 희생자 버퍼 블록에 존재하는 노드-페이지들 중에서 최신의 노드-페이지는 LPN=1, 0, 3(offset=3의) 이며, FTL에 쓰기 연산을 호출할 때 LPN=0, 1, 3의 순차쓰기 패턴으로 변환된다[8]. 희생자 버퍼 블록의 노드-페이지들은 동일한 논리 블록 번호를 가지므로 임의쓰기 패턴보다 순차쓰기 패턴에 훨씬 더 가까운 쓰기 패턴이 FTL에 발생된다.

플래시 메모리의 블록은 안정적으로 소거될 수 있는 횟수가 제한적이기 때문에 특정한 블록에서 많은 소거 연산이 일어날 경우 플래시 메모리의 내구성을 악화시킨다. 이러한 특성을 고려하여 희생자 버퍼 블록을 소거한 후 FTL에 이것을 반환한다. 그리고 FTL로부터 빈 블록을 할당을 받아서 버퍼 블록으로 사용한다.

8) 실험에 의하면 FTL에 전달되는 노드-페이지의 순서는 중요하지 않다. 즉, "1, 0, 3"의 경우나 오름차순정렬 후 "0, 1, 3"의 경우 모두 동일한 결과를 보인다. 따라서 정렬 비용과 같은 오버헤드가 거의 발생하지 않는다.

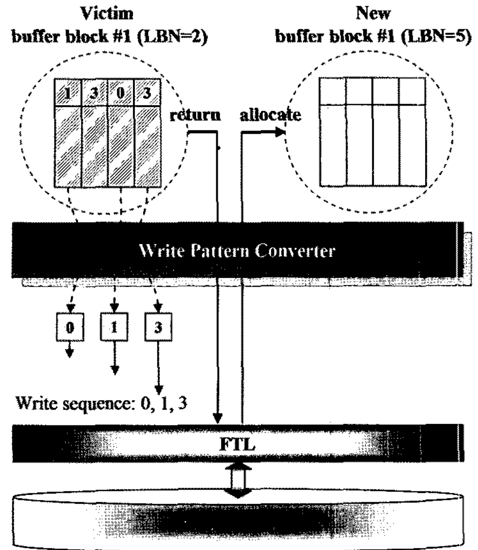


그림 9 Write Pattern Converter

6. 실험결과

본 장에서는 B-트리 관리 기법 중 쓰기 패턴 변환을 이용하는 경우와 그렇지 않은 경우에 대해 실험 결과를 보인다. 이를 위해 그림 7의 구조를 갖는 시뮬레이터(simulator)를 구현하였으며, 내부적으로 B-트리 응용, FTL, 플래시 메모리로 구성된다. B-트리 응용을 위해 기본적인 B-트리 모듈뿐만 아니라 Transit buffer manager와 Write pattern converter를 구현하였으며, FTL은 표 3에서 제시한 FAST와 BAST 기법을 각각 구현해서 사용하였다. 마지막으로, 저장 장치는 소블록 플래시 메모리를 가정한 에뮬레이터(emulator)를 구현하였으며, 디바이스 드라이버로서 내부적으로 read, write, erase 함수를 가진다.

플래시 메모리에서의 읽기, 쓰기, 및 소거 비용은 표 2와 같다. 표에서 읽기 보다는 쓰기 비용이 크며, 쓰기 보다는 소거 비용이 훨씬 크다는 것을 알 수 있으며, 따라서 플래시 메모리에서의 성능에서는 쓰기연산과 소거연산이 큰 영향을 미친다. 본 실험에서의 성능 측정 기준은 플래시 메모리에서 발생하는 쓰기 및 소거 연산의 횟수이며, 이를 통해 제안하는 방식의 우수성을 검증한다.

표 2 플래시 메모리 접근 속도(Samsung K9WAG08U1A 16 Gbits SLC NAND)

Operation	Access time
read (μs/page)	80
write (μs/page)	200
erase (μs/block)	1500

6.1 실험환경

실험환경은 표 3과 같으며 키 업데이트 횟수와 버퍼 블록의 수를 변화시켜 성능을 측정한다. FTL은 현재까지 성능이 우수하다고 알려진 BAST[2]와 FAST[3] 방식을 모두 사용하여 본 논문에서 제안하는 방식이 다양한 FTL에 적용될 수 있음을 보여준다.

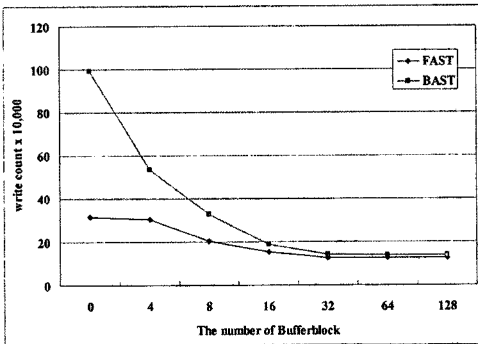
표 3 실험 환경

Programming Language	ANSI C
Operating System	Fedora 5
Compiler	gcc version 4.1.0
FTL algorithm	BAST, FAST
Number of Buffer Blocks	none, 4, 8, 16, 32, 64, 128
Number of key updates	50,000 ~ 500,000

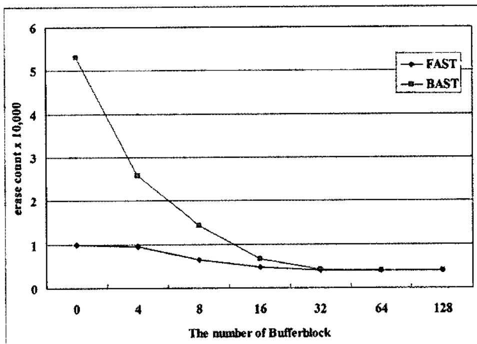
6.2 성능 분석

그림 10은 B-트리에 임의적으로 키를 50,000번 업데이트 하였을 경우 버퍼 블록의 수에 따른 플래시 메모리에서 발생되는 쓰기 및 소거 연산의 횟수를 나타낸다. 그림에서 X 축의 버퍼 블록의 수가 0일 때는 쓰기 패턴 변환 방식을 전혀 사용하지 않는 방식을 의미한다. 그림에서 알 수 있듯이, 쓰기 패턴 변환 기법을 사용한 경우가 그렇지 않은 경우(버퍼 블록의 수=0)보다 쓰기 및 소거 연산의 수에서 매우 우수함을 알 수 있다. 그 이유는 전자의 경우에는 B-트리 수정 시 발생하는 임의쓰기 패턴을 FTL에 유리한 순차쓰기 패턴으로 변환하여 FTL에 전달하기 때문이다. 그림에서 BAST 기법보다 FAST 기법이 우수한 성능을 보이는데, 임의쓰기 패턴뿐만 아니라 순차쓰기 패턴에서도 전자보다 후자가 더 우수한 쓰기 연산 알고리즘을 갖고 있기 때문이다[3].

그림 10에서 버퍼 블록의 수가 증가할수록 "total order"에 가까운 순차쓰기 패턴으로 변환할 수 있으므로 FTL에 더 효율적이며, 따라서 플래시 메모리에서 발생되는 비용을 크게 감소시킨다. 이와 같은 결과는 키 업데이트 수가 많아져도 비슷하게 나타나는데, 그림 11은 B-트리에 임의적으로 100,000번의 키를 업데이트 하였을 때 플래시 메모리에서 발생되는 쓰기 및 소거 연산 횟수를 보여준다.

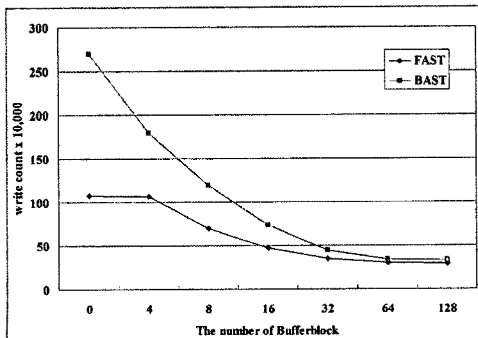


(a) 쓰기연산

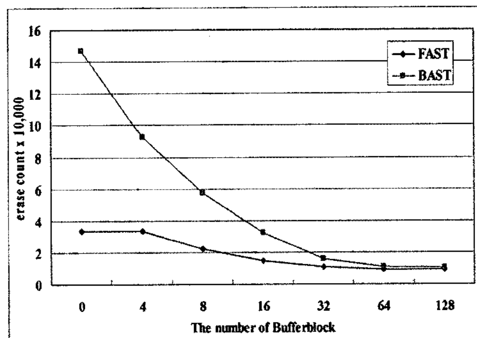


(b) 소거 연산

그림 10 키 업데이트 횟수 = 50,000

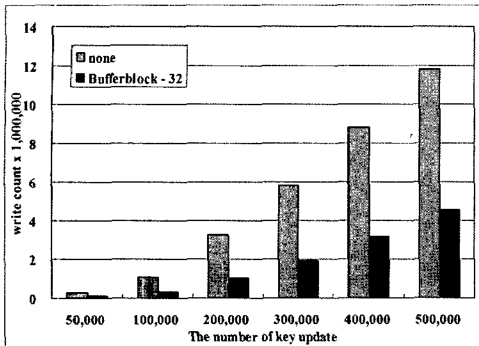


(a) 쓰기연산

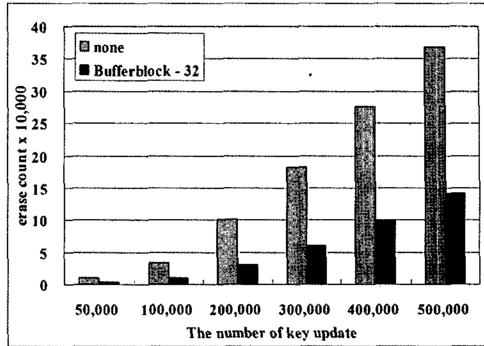


(b) 소거 연산

그림 11 키 업데이트 횟수 = 100,000

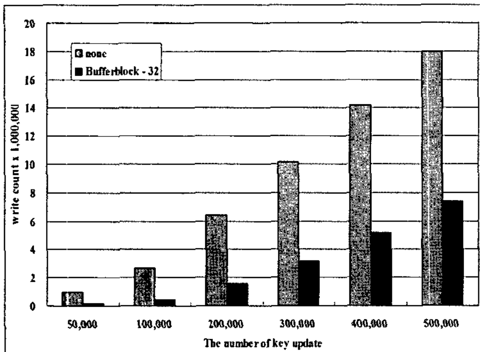


(a) 쓰기연산

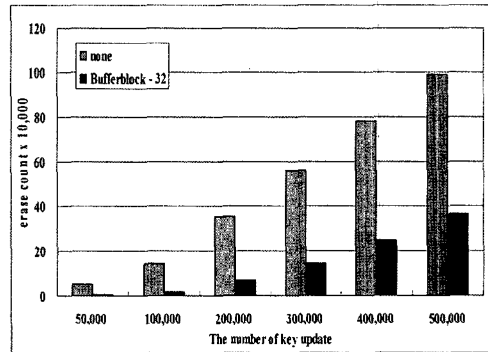


(b) 소거 연산

그림 12 키 업데이트 수에 따른 성능 비교(버퍼블록 = 32, FAST)



(a) 쓰기연산



(b) 소거 연산

그림 13 키 업데이트 수에 따른 성능 비교(버퍼블록 = 32, BAST)

그림 12는 쓰기 패턴 변환 방식을 사용하지 않은 경우와 버퍼 블록을 32개 사용했을 경우에 플래시 메모리에서 발생되는 쓰기 및 소거 연산의 횟수를 비교한 것이다. FTL은 FAST 방식을 사용하였으며, X축은 키 업데이트 수를 의미하는 것으로 50,000~500,000의 범위에서 실험한 것이다. 그림에서 키 업데이트 수에 따라 쓰기 연산의 횟수가 60~70%이상 감소하는 것을 알 수 있다. 이러한 쓰기 연산의 감소는 플래시 메모리에서 발생되는 소거 연산에도 영향을 미쳐 소거 연산도 60~70% 정도 감소함을 알 수 있다. 또한 이러한 소거 연산의 감소는 플래시 메모리에서 데이터가 처리되는 시간을 크게 단축시켜 줄 뿐만 아니라 각 블록이 안정적으로 소거될 수 있는 횟수가 제한적인 점을 감안할 때 플래시 메모리의 내구성을 크게 향상시켜 준다. 그림 13은 BAST FTL을 사용할 때 플래시 메모리에서 발생되는 비용을 나타낸다. 참고로, 그림 12와 13에서 FAST가 BAST보다 소거 연산의 수가 두 배 이상 차이가 발생하는 것은 순차 쓰기 패턴에서 FAST가 BAST보다 훨씬 유리하기 때문이다[3].

7. 결론 및 향후 연구

본 논문에서는 플래시 메모리 상에서 B-트리를 관리할 때 발생되는 비용을 낮추기 위해 쓰기 패턴 변환을 이용한 기법을 제안했다. 이 기법에서의 버퍼 블록 모듈은 B-트리를 수정할 때 발생하는 임의쓰기 패턴을 플래시 메모리의 일부 공간을 버퍼로 활용하여 순차쓰기 패턴으로 변환시켜 주는 역할을 한다. FTL에 효율적인 순차쓰기 패턴으로의 변환은 플래시 메모리에서 발생하는 비용을 크게 감소시켜 준다. 이러한 변환은 기존의 FTL은 임의쓰기 패턴보다 순차쓰기 패턴에 유리하다는 실험 결과에 바탕을 두고 있다.

참 고 문 헌

[1] Jung Hyun Nam, Dong-Joo Park, "Design and Implementation of the B-Tree on Flash Memory," *Journal of KIPS*, vol.34, no.2, April 2007.

[2] J.Kim, J.M. Kim, S.H. Noh, S.L. Min, and Y. Cho. "A Space-Efficient Flash Translation Layer for Compact Flash System," *IEEE Transactions on*

Consumer Electronics, vol.48, no.2, May 2002.
[3] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song, "A Log Buffer-based Flash Translation Layer using Fully-Associative Sector Translation," *ACM Transactions on Embedded Computing Systems*, vol.6, no.3, July 2007.
[4] Chin-Hsien Wu, Li-Pin Chang, Tei-Wei Kuo, "An Efficient B-Tree Layer for Flash-Memory Storage Systems," *The 9th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, 2003.
[5] Mendel Rosenblum and John K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, February 1992.
[6] David Woodhous, "JFFS : The Journalling Flash File System," RedHat homepage: http://www.redhat.com.
[7] YAFFS homepage: http://www.aleph1.co.uk/armlinux/projects/yaffs

박 동 주
1995년 서울대학교 컴퓨터공학과(학사) 1997년 서울대학교 컴퓨터공학과(석사) 2001년 서울대학교 전기전자컴퓨터공학부(박사). 2001년~2003년 삼성전자 책임연구원. 2004년~2005년 숭실대학교 컴퓨터공학부 전임강사. 2006년~현재 숭실대학교 컴퓨터공학부 조교수. 관심분야는 플래시 메모리, 임베디드 데이터베이스, 멀티미디어 데이터베이스 등

최 해 기
2005년 숭실대학교 컴퓨터학부(학사). 2007년 숭실대학교 대학원 컴퓨터학과(석사) 2007년~현재 메디슨 연구원. 관심분야는 데이터베이스, 플래시 메모리 기반 소프트웨어, DICOM 등