플래시 메모리상에 B+트리를 위한 효율적인 색인 버퍼 관리 정책[†]

이현섭* 강원식** 이동하** 이동호*
*한양대학교 컴퓨터공학과, **대구경북과학기술연구원 S/W연구팀
*{hyunseob, dhlee72}@cse.hanyang.ac.kr, **{wskang, dhlee}@dgist.ac.kr

An Efficient Index Buffer Management Scheme for B+tree on Flash Memory

Hyun-Seob Lee*, Kang Won-Seok**, Lee, Dong Ha**, Dong-Ho Lee*
*Dept. of Computer Science and Engineering, Hanyang University
**Daegu Gyeongbuk Institute of Science & Technology

요 약

NAND 플래시 메모리는 작고, 가볍고, 저 전력이라는 장점 때문에 휴대폰, MP3, PDA 등 이동 컴퓨팅 장치의 저장소로 많이 사용되고 있다. B+트리는 저장소에 있는 데이터를 효율적으로 접근하기 위한 색인 구조이다. 그러나 NAND 플래시 메모리의 다양한 특징들로 인해 기존의 디스크 기반의 B+트리를 플래시 메모리에 그대로 적용하는데 여러 단점들이 존재한다. 본 논문에서는 NAND 플래시 메모리 상에서 B+트리를 효과적으로 구축하기 위한 B+트리 색인 버퍼 관리 기법을 제안한다.

1. 서론

최근 휴대폰, PDA, 노트북, MP3, PMP등 임베디드 컴퓨팅 장비의 기술이 급속도로 발전하고 있다. NAND 플래시 메모리는 작고, 가볍고, 강한 내구력과 비휘발성이라는 특징을 가지고 있기 때문에 임베디드 컴퓨팅 장비의 저장소로 사용하기에 적당하다. NAND 플래시 메모리는 여러 개의 블록으로 이루어져 있고, 한 블록은 32개의 Page로 구성되어 있다(Page는 섹터와 동일하다). 하나의 Page는 512Byte의 데이터 영역과 16Byte로 이루어진 Spare영역으로 이루어져 있다. 그리고 플래시 메모리는 블록 당 10만 ~ 100만 번 쓰고 지울 수 있으며 그 이 후에는 블록에 결함(Bad)이 생길 수 있다. 또한 플래시 메모리는 디스크와는 다르게 읽고 쓰기 단위(Page)와 지우기 단위(Block)가 다르고 연산을 처리하는데 사용되는 시간적 비용이(읽기 : 10㎲, 쓰기 : 200㎲, 지우기 : 2㎳) 각각 다르다[7]. 플래시 메모리는 플래시 메모리가 갖고 있는 특성 때문에 Out-place update 정책을 사용한다. 따라서 디스크 기반으로 개발되어 온 시스템을 플래시 메모리 상에서 사용하는 것은 수명과 성능 면에서 문제가 있다. 이 문제를 해결하기 위해 FTL(Flash translation table)을 사용한다[2, 3, 4]. FTL은 논리적인 주소를 물리적인 주소로 변경해주는 역할을 담당한다. 그리고 플래시 메모리상의 모든 블록이 비슷한 횟수로 사용되도록 하기 위한 Wear-leveling 정책을 수행 한다.

B+트리[1]는 저장소에 있는 데이터를 효율적으로 접근하기 위한 자료 구조이다. 그러나 인덱스는 이것이 구축될 때 삽입(Insert)이나 갱신(Update) 노드분할(Split)등에 의해 많은 중첩 쓰기(Overwrite)를 발생한다. 따라서 기존에 사용해 왔던 디스크 기반 B+트리를 그대로 플래시 메모리상에 구축하는 것은 심각한 성능저하를 발생 시킨다. 이 문제를 해결하기 위해 BFTL[5]과 BFTL을 개량한 BOF[6]등의 방법이 제안되었다. BFTL이나 BOF는 버퍼를 사용하여 B+구축할 때 발생하는 많은 쓰기 연산을 개선하였다. 그러나 BFTL이나 BOF는 버퍼에서 플래시 메모리로 데이터를 기록하는데 초점을 맞추고 있고, 버퍼에 데이터를 유지 및 관리하는 것은 고려하고 있지 않다.

본 논문에서는 기존의 비효율적인 버퍼 사용의 문제점을 개선하여 보다 효율적으로 버퍼를 사용하는 방법을 제안한다.

본 논문의 구성은 다음과 같다. 2장에서는 플래시 메모리상에 B+트리를 구축하기 위해 제안된 BFTL(B-tree for flash memory translation layer)과 BOF(Btree on flash memory)를 기술하고 이것이 가지고 있는 문제점을 기술한다. 3장에서는 이 문제를 해결하기 위한 IBSF(Index buffer scheme on flash memory) 기법을 기술한다. 4장에서는 성능 평가를 하고 마지막으로 5장에서 결론을 맺는다.

2. Flash Memory 상에서 B+트리 구현 기법

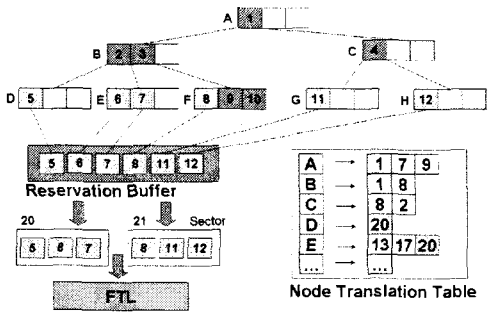
B+트리는 대부분의 응용프로그램에서 사용 되고 있다. B+트리는 저장소에 저장되어 있는 데이터를 효율적으로 접근하기 위해 사용되는 자료구조이다. 그러나 B+트리는 주로 디스크 기반의 저장소에서 개발되고, 사용되어 왔다. 따라서 현재의 B+트리는 플래시 메모리의 특성을 반영하고 있지 않는다. 플래시 메모리상에서의 데이터 갱신 방법은 디스크 상에서의 갱신 방법이 다르기 때문에 플래시 메모리 상에 디스크 기반 B+트리를 바로 적용하는 것은 심각한 성능저하를 일으킨다. 본 절에서는 이 문제를 해결하기 위한 선행연구인 BFTL 방법과 BOF 방법을 기술하고 이 방법들이 가지고 있는 문제점을 기술한다.

2.1 B-tree for Flash Memory Translation Layer (BFTL)

BFTL은 B트리의 노드에 삽입, 삭제, 갱신 등이 발생할 때마다 색인단위(Index unit)를 생성한다. 색인단위는 특정 노드에 속한 데이터(Entry)와 색인단위를 식별하기위한 식별자 그리고 이것을 처리하기 위한 명령어 코드 등의 정보로 구성된다. 노드의 변경으로 인해 발생한 색인단위는 일시적으로 Reservation buffer라고 불리는 버퍼에 유지된다. 그리고 버퍼가 다 차면 버퍼에 유지되고 있던 색인단위들을 플래시 메모리에 기록(Commit) 한다. 색인단위는 플래시 메모리의 섹터보다 상대적으로 작다. 따라서 BFTL은 한 개의 섹터에 여러 개의

[†]This work was supported by the Daegu Gyeongbuk Institute of Science and Technology and funded by Ministry of Science and Technology(MOST).

색인단위를 모아서 플래시 메모리에 기록한다.



[그림 1] BFTL의 구조

BFTL은 하나의 노드를 구축하기 위해서는 관련된 정보인 색인단위를 모두 읽어야 한다. 그러나 특정 노드에 관련된 색인단위는 플래시 메모리상에 여러 개의 페이지에 나뉘어 기록 될 수 있다. BFTL은 이렇게 흩어진 색인단위들을 관리하기 위해 노드변환테이블(Node translation table)을 사용한다. 노드변환테이블은 색인단위가 버퍼에서 플래시 메모리로 기록될 때 노드와 관련된 색인단위가 몇 번 섹터에 저장되는지 저장정보를 유지한다.

[그림 1]은 BFTL의 구조와 운용하는 방법을 보여주고 있다. 그림에서 노드 D, E, F, G, H에 변경(삽입, 갱신)이 일어났고 이 변경을 반영하는 색인단위 5, 6, 7, 8, 11, 12가 생성되었다. 그리고 색인단위들은 일시적으로 버퍼에 유지된다. [그림 1]의 버퍼는 색인단위들로 다 찼을 때 이 색인단위들이 플래시 메모리로 어떻게 기록되는지 처리하는 과정을 보여준다. BFTL은 노드 D, E, F, G, H에 일어난 변경정보인 색인단위를 2개의 섹터에 모아서 저장한다. 그리고 이 섹터를 플래시 메모리에 기록한다. 이때 노드변환테이블에 D, E, F, G, H노드에 관련된 정보가 갱신된다. [그림 1]의 예제에서 BFTL이 B+트리에서 발생한 6번의 쓰기를 2번으로 줄여서 쓰기의 성능을 개선하는 것을 알 수 있다. [그림 1]에서 D와 E노드에 관련된 색인단위가 20번 섹터에 저장 되었고 이 정보를 유지하기 위해 노드변환테이블의 D와 E노드 리스트에 관련 정보를 유지하는 것을 볼 수 있다.

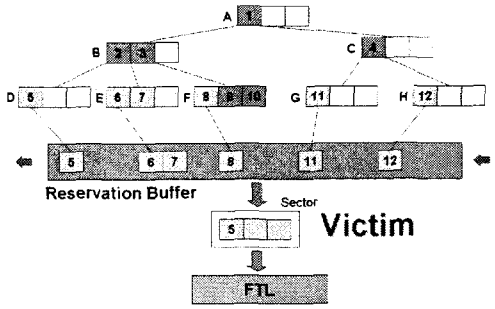
BFTL에서 색인단위를 관리하기 위해 사용하는 노드변환테이블은 RAM에 상주한다. 그러나 RAM은 제한된 자원이기 때문에 색인단위가 기록될 때 마다 유지하고 있는 정보를 무한히 늘려나갈 수는 없다. 따라서 노드변환테이블은 노드마다 일정 개수의 주소정보만을 유지한다. 만약 노드변환테이블이 유지할 수 있는 개수를 초과하는 주소정보가 발생한다면 압축을 수행한다.

BFTL은 노드에 변경을 반영하는 색인단위가 플래시 메모리 상의 여러 섹터에 저장된다. 따라서 하나의 노드를 구축하기 위해서는 여러 섹터에 저장되어있는 색인단위를 모아야 한다. 이것은 특정 데이터를 검색하는데 비효율적으로 많은 읽기연산을 발생시킨다. 또한 색인단위를 관리하기 위해 사용하는 노드변환 테이블은 B+를 구축할 때 발생하는 많은 색인구조 때문에 빈번한 압축을 수행한다. 압축은 많은 읽기와 쓰기 연산을 발생하기 때문에 성능을 나쁘게 한다.

2.2 B-tree on flashmemory (BOF)

앞 절에서 BFTL과 BFTL이 지니고 있는 문제점을 기술했다. 본 절에서는 BFTL의 문제점을 개선한 BOF 방법을 기술한다. BFTL은 하나의 노드에 관련된 색인단위를 여러 섹터에 저장하기 때문에 노드를 구축할 때 많은 읽기 연산이 발생한다. 또한 노드변환테이블을 관리하는데 압축연산의 수행으로 인해 성능이 나빠지게 한다. BOF는 이 문제를 해결하기 위해 하나의 노드에 관련된 색인단위는 하나의 페이지에만 저장하도록 하였다. BOF는 노드를 구축할 때 하나의 섹터만 읽으면 되기 때문에 메모리 자원을 차지하는 노드변환테이블을 사용하지 않아도 된다. BOF의 구조는 [그림 2]에서 볼 수 있다.

[그림 2]는 BOF가 버퍼에서 플래시 메모리로 색인단위를 기록하는 과정을 보여준다. [그림 2]의 예에서 노드 D, E, F, G, H의 변경을 반영하기 위해 5, 6, 7, 8, 11, 12의 색인단위가 발생하였다. 이것은 발생한 순서대로 버퍼(Reservation buffer)에 유지된다. 그리고 버퍼가 다 차면 버퍼에 유지하고 있던 색인단위를 플래시 메모리로 기록한다.



[그림 2] BOF의 구조

BOF는 BFTL과 비교하여 읽기 성능을 증가 시켰고, 비효율적으로 운용해 오던 노드변환테이블을 제거하여 성능을 개선하였다. 그러나 BFTL이나 BOF는 버퍼에서 플래시 메모리로 데이터가 기록되는 순간에만 초점을 맞추었다. BFTL이나 BOF는 버퍼에 데이터가 입력되는 순간이나 버퍼에 데이터를 유지하는 동안 데이터를 효율적으로 관리 하지 못한다. BFTL이나 BOF에서 버퍼에 데이터를 유지하는 방법은 노드에 삽입, 삭제, 갱신 등의 연산에 의해 생성된 색인단위를 발생 순서대로 버퍼에 유지하는 것이다. 따라서 특정 노드에 반복적인 갱신이나 노드 분할과 같은 연산이 일어날 경우 노드의 특정 데이터에 관련된 색인단위가 중복되어서 버퍼에 유지될 수 있다. 이것은 버퍼의 공간사용률을 떨어트리고, 색인단위의 데이터를 플래시 메모리로 기록하는 주기를 빠르게 만든다.

3. Index Buffer Scheme on Flash Memory (IBSF)

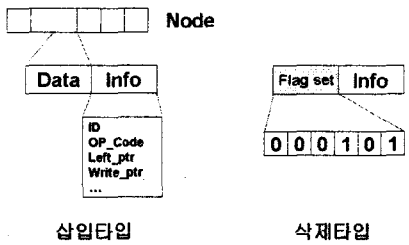
앞에서 플래시 메모리 상에 B트리를 구축하기 위한 BFTL 방법과 BOF 방법을 설명하고 이 방법들이 가지고 있는 문제점에 대해 기술했다. 본 장에서는 이 문제를 해결하기 위한 IBSF 기법을 설명한다.

3.1 색인 단위(Index Unit)

IBSF에서 색인단위는 노드의 삽입, 삭제, 갱신 등에 의해 발생된다. 색인단위는 데이터 영역과 색인 정보 영역으로 이루어져 있다. IBSF에는 두 가지 타입의 색인 단위가 존재한다. 한 가지는 삽입(Insert) 타입이고, 또 다른 한 가지는 삭제(Delete) 타입이다. 삽입 타입은 B+트리의 노드 상에 특정 데이터(Entry)의 삽입, 갱신을 처리하기 위한 색인단위이다. 삭제 타입은 B+트리의 노드 상에 특정 데이터가 삭제를 처리하기 위한 색인단위이다. 삽입 타입과 삭제 타입의 색인단위는 [그림 3]과 같다. 색인단위는 데이터 영역과 색인 정보로 구성된다. 삽입 타입의 데이터 영역은 변경된 노드의 데이터(Entry)를 유

지하는 영역이다. 색인 정보는 색인단위가 어느 데이터와 관련된 것인지를 식별하기 위한 ID와 색인단위가 삽입 타입인지, 삭제 타입인지 구분하기 위한 Op_Code 등을 포함하고 있다.

삭제 타입의 색인단위의 경우 색인정보는 삽입 타입과 같다. 그러나 데이터 영역을 구성하는 데이터(Flag set)는 삽입 타입의 데이터 영역을 구성하는 데이터(Entry)와 다르다. IBSF의 삭제 타입의 데이터 영역에는 노드의 몇 번째 데이터를 삭제해야 하는지에 관한 정보를 Flag형태로 유지한다. 만약 해당 위치에 Flag가 1로 설정되어 있다면 관련된 노드의 해당 데이터(Entry)는 삭제되어야 한다는 정보를 의미한다.



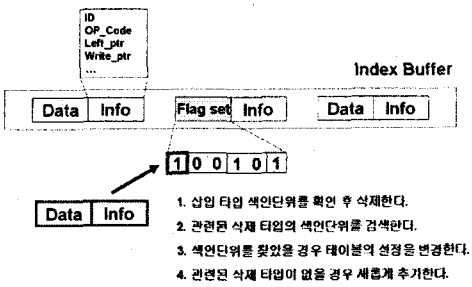
[그림 3] 색인단위

[그림 3]의 삭제 타입 색인단위는 해당 노드의 3번과 5번 데이터부분을 삭제해야 한다는 정보를 유지하고 있는 것을 나타낸다. BFTL과 BOF의 경우 B+트리의 노드에 삭제가 일어나면 삭제를 반영하는 색인단위가 삭제횟수 만큼 발생하였다. 그러나 IBSF는 노드에 관련된 여러 데이터에 대하여 삭제가 발생해도 한 개의 색인단위로 모든 삭제정보를 유지할 수 있다. 이것은 버퍼의 색인 버퍼의 공간 활용도를 높여준다.

3.2 IBSF의 데이터 삭제

삭제 타입의 색인단위는 노드에 데이터를 삭제하려고 할 때 발생한다. BFTL이나 BOF는 삭제타입의 색인단위를 삽입타입의 색인단위와 구분 없이 관리하였다. 그러나 삭제타입의 색인단위는 노드의 몇 번째 데이터를 삭제해야 하는지에 관한 정보를 유지하는 것만으로도 삭제명령을 수행할 수 있다. 따라서 삭제타입의 색인단위는 데이터 영역이 불필요 하다. IBSF에서 삭제타입의 색인단위의 데이터영역은 삭제할 데이터를 포함하는 대신 노드의 어떤 데이터를 삭제해야 하는지 Flag 정보를 유지한다. 삭제해야 할 대상이 되는 노드는 색인단위가 유지하고 있는 ID를 통해 식별된다. 그리고 해당 노드에서 삭제해야 할 데이터의 식별은 데이터 영역의 몇 번째 Flag가 1로 설정되어있는지를 통해 이루어진다. 하나의 노드에 여러 개의 데이터를 삭제해야 할 때 삭제타입의 색인단위에 여러 개의 Flag를 1로 설정한다. 따라서 IBSF에서는 한 개의 삭제 타입 색인단위로 하나의 노드에 관련된 여러 개의 삭제 정보를 유지할 수 있다. [그림 4]는 IBSF가 삭제 타입의 색인단위를 처리하는 것을 보여주고 있다.

IBSF에서 삭제 타입의 색인단위는 BFTL과 BOF에서 색인단위를 처리하는 방법과 다르다. IBSF에서 삭제 타입 색인단위를 처리하는 방법은 먼저 삭제하고자 하는 데이터의 색인단위가 버퍼에 저장되어 있는지 검색한다. 만약 검색한 색인단위가 버퍼에 존재한다면 해당 색인단위를 버퍼에서 삭제한다. 그 다음 버퍼에서 삭제하고자 하는 데이터와 관련된 노드의 삭제 타입 색인단위를 검색한다. 만약 해당 색인단위를 발견하게 되면 데이터 영역의 해당 Flag를 1로 갱신한다. 그러나 색인단위를 발견하지 못한다면 버퍼에 새로운 삭제 타입의 색인단위를 생성한다. [그림 4]의 예에서 삭제 타입의 색인단위는 노드의 0번, 3번, 5번의 데이터를 삭제하라는 삭제 정보를 유지하고 있다.

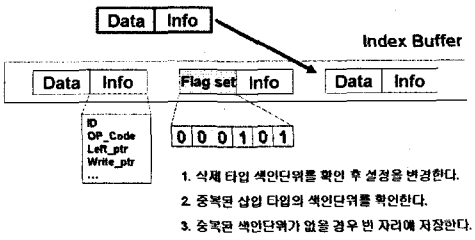


[그림 4] 삭제 정책

IBSF는 한 노드에 관련된 여러 개의 삭제정보도 하나의 색인단위에 유지할 수 있기 때문에 반복적인 데이터 삭제나 노드 분할 등으로 인해 발생하는 여러 개의 삭제 타입 색인단위를 적은공간에 유지할 수 있다. 이것은 BFTL이나 BOF 방법과 비교해 볼때 같은 크기의 버퍼를 이용해 더 많은 삭제 타입 색인단위를 처리할 수 있다. 따라서 IBSF는 BFTL이나 BOF와 비교해 버퍼의 공간 활용도가 좋다. 그리고 버퍼에서 플래시 메모리로 데이터를 기록하는 기록 주기를 늦추어서 플래시 메모리의 쓰기 성능을 향상시킨다.

3.3 IBSF의 데이터 삽입

IBSF의 삽입 타입 색인단위는 BFTL이나 BOF의 색인단위와 유사하다. 그러나 색인단위를 처리하는 방법은 BFTL이나 BOF와는 다르다. BFTL이나 BOF에서는 색인단위를 아무런 처리 없이 버퍼에 유지하였다. 그러나 IBSF에서는 발생한 색인단위를 처리 후 버퍼에 유지한다. IBSF에서 색인단위를 처리하는 방법은 버퍼 안에 중복된 데이터를 허용하지 않는 것이다. B+트리에서 특정 노드의 반복적인 데이터 갱신이나 노드 분할은 많은 중복된 색인단위를 발생시킨다. BFTL이나 BOF의 버퍼는 B+트리의 노드에 변경으로 인해 발생된 색인단위를 잉시로 유지하는 공간으로만 활용했다. 따라서 버퍼에는 중복된 색인단위가 존재할 수 있었다. 그리고 버퍼에 중복된 데이터는 버퍼의 공간 활용도를 나쁘게 할 것이다. 그러나 IBSF에서는 버퍼에 색인단위를 입력하기 전에 버퍼가 유지하고 있던 색인단위를 검사해 본다. 만약 IBSF가 버퍼에서 동일한 데이터의 변경을 반영한 색인단위를 발견한다면 기존의 색인단위를 새로운 색인단위로 갱신한다.



[그림 5] 삽입 정책

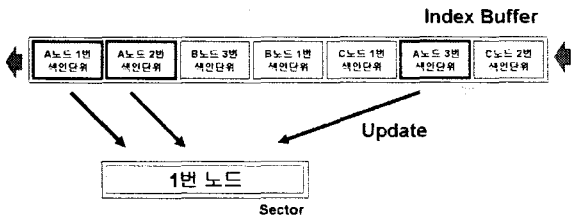
[그림 5]는 IBSF에서 색인단위를 처리하는 방법을 보여준다. [그림 5]에서 IBSF는 삽입타입의 색인단위가 발생했을 때 버퍼에서 삽입처리를 해야 하는 노드와 관련된 삭제 타입의 색인단위를 검색한다. 그리고 그리고 데이터 영역의 Flag set에서

해당 인덱스의 삭제 상태를 확인한다. 만약 해당 데이터가 삭제로 설정되어 있다면 이것을 무효상태로 갱신한다. 그 다음 버퍼에 삽입 타입으로 설정되어 있는 색인단위를 검색한다. 만약 관련된 색인단위를 찾는다면 그 색인단위를 새로운 색인단위로 갱신한다. 그러나 해당 색인단위를 찾지 못한다면 버퍼의 비어있는 공간에 새로운 색인단위를 입력한다. 만약 비어있는 공간이 없을 경우에는 버퍼가 유지하고 있는 색인단위를 플래시 메모리에 기록한다. IBSF는 BFTL이나 BOF는 버퍼에 색인단위를 저장하는 방법이 다르다. BFTL이나 BOF는 노드에 변경을 반영하는 색인단위를 발생한 상태 그대로 버퍼에 유지한다. 그러나 IBSF는 버퍼에 색인단위를 유지하기 전에 만약 버퍼가 중복된 색인단위를 유지하고 있다면 해당 색인단위를 새로운 색인단위로 갱신한다.

IBSF의 삽입 처리는 버퍼에 중복된 색인단위를 제거하기 때문에 버퍼를 BFTL이나 BOF보다 공간을 효율적으로 사용할 수 있다. IBSF의 삽입 정책은 버퍼에 있는 색인단위를 플래시 메모리로 기록하는 기록 주기를 늦추어 쓰기 성능을 향상시킨다.

3.4 기록 정책

IBSF에서 버퍼의 크기는 제한된 자원이다. 따라서 버퍼는 일정 개수의 색인단위만을 유지할 수 있다. 만약 유지해야 할 색인단위의 개수가 제한된 크기보다 클 경우 버퍼에 저장되어 있는 색인단위를 플래시 메모리로 기록해야 한다. 그러나 IBSF의 버퍼에서는 색인단위의 갱신이 가능하기 때문에 버퍼가 유지하고 있는 모든 색인단위를 플래시 메모리에 기록하는 것 보다는 몇 개의 색인단위를 선택하여 기록하는 것이 효율적이다.

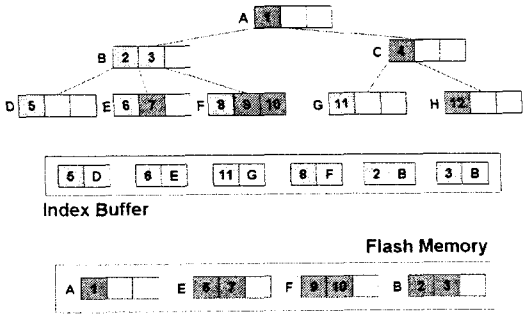


[그림 6] 기록 정책

IBSF에서 기록 정책은 [그림 6]과 같다. IBSF는 버퍼에서 플래시 메모리로 기록할 색인단위를 선정하는 방법은 FIFO(First in first out)를 사용한다. FIFO는 버퍼에 있는 색인단위를 플래시 메모리로 기록을 할 때 버퍼에 있는 모든 색인단위를 플래시 메모리로 보내는 것이 아니라 버퍼에 제일 먼저 유지되었던 색인단위부터 플래시 메모리로 보내는 방법이다. [그림 6]의 예에서는 색인단위가 왼쪽에서 오른쪽 순서대로 저장된다고 가정하였다. [그림 6]의 상태에서 버퍼의 색인단위를 플래시 메모리로 기록하기 위해서 IBSF는 제일 먼저 입력된 색인단위를 검색한다. [그림 6]의 예에서는 가정 먼저 입력된 색인단위를 A노드의 1번이라고 가정한다. 그 다음 IBSF는 A노드의 데이터를 저장하고 있는 섹터를 읽어서 메모리에 논리적으로 노드를 구축한다. 그리고 IBSF는 버퍼에 있는 A노드와 관련된 색인단위들을 노드에 반영한다. 색인단위의 반영을 통해 갱신된 노드는 다시 플래시 메모리로 저장된다. [그림 6]에서는 버퍼에 유지하고 있던 3개의 색인단위가 플래시 메모리로 처리되었다.

3.5 IBSF를 이용한 검색 및 읽기 정책

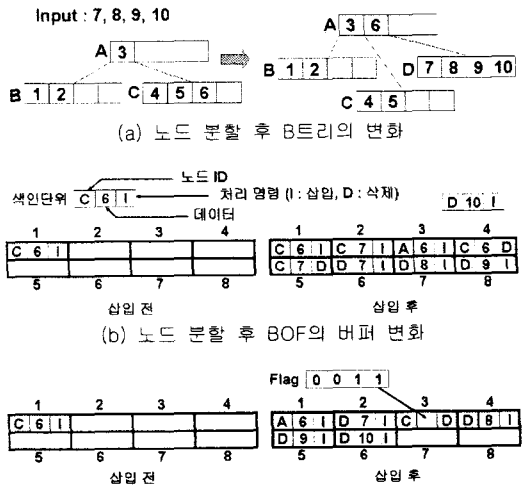
[그림 7]은 IBSF의 구조를 보여주고 있다. IBSF에서는 B+트리의 특정 데이터를 읽기 위해 버퍼를 먼저 검색한다. 그러나 버퍼에서 원하는 데이터를 찾을 수 없을 경우 플래시 메모리를 검색한다. 본 절에서는 IBSF에서 읽기가 어떻게 수행 되는지 예를 들어 설명한다. [그림 7]은 노드 D, E, G, F, B에 색인단위가 색인 버퍼에 저장되어 있는 상태이다.



[그림 7] IBSF의 구조

[그림 7]의 7번 데이터를 검색한다고 가정하자. IBSF는 버퍼에서 루트 노드인 A노드에 대한 데이터를 검색을 한다. 그러나 버퍼는 해당 노드에 대한 데이터를 가지고 있지 않다. 따라서 A노드를 읽어 들이기 위해 플래시 메모리를 검색하게 한다. 그리고 A노드에 대한 데이터를 메모리로 읽어 온다. A노드의 데이터에 의해 7번 데이터를 읽기 위해서는 B번 노드의 데이터가 필요하다. 그런데 B번 노드의 관련 데이터는 버퍼를 통해 알 수 있음으로 플래시 메모리에 대한 읽기 연산은 발생하지 않는다. IBSF는 버퍼에 있는 B번 노드의 데이터를 통해 7번 데이터는 E노드에 있다는 것을 알아낼 수 있다. 결국 IBSF는 플래시 메모리의 E노드를 한 번 더 읽어서 7번 데이터를 검색하게 된다. 그러나 만약 검색하고자 했던 데이터가 6번 데이터였다면 B+트리를 따라 검색해 나아가거나 플래시 메모리에 대한 읽기 연산을 수행하지 않아도 해당 데이터를 버퍼에서 찾을 수 있었을 것이다.

3.6 노드 분할에 따른 버퍼관리 정책



[그림 8] 노드 분할에 따른 색인 버퍼의 변화

앞 절에서 IBSF의 삽입, 삭제, 검색 및 읽기 정책을 기술했다. 본 절에서는 B트리에서 노드분할 수행시 BOF와 IBSF가 버퍼를 어떻게 관리하는지 예를 들어 설명한다.

[그림 8]는 노드 분할에 따른 BOF와 IBSF의 색인 버퍼 변화를 보여주고 있다. [그림 8]의 예제에서 B트리의 노드는 4개의 Entry를 유지할 수 있다고 가정했다. (a)에서 input은 입력된 데이터의 키 값이라고 가정한다. 색인단위 구조는 (b)의 색인단위로 가정한다. 예제의 설명을 위해 색인단위는 (b)에서 나타내는 것과 같이 3부분으로 나누었다. 앞부분은 색인단위가 어느 노드에 관련된 정보인지를 나타낸다. 두 번째 부분은 어떤 데이터에 관련된 색인단위인지를 나타낸다. 세 번째 부분은 색인단위가 어떤 연산을 수행해야 하는지를 나타낸다. Input 데이터를 처리하기 전 BOF의 색인버퍼는 (b)의 왼쪽 삽입 전 상태라고 가정하고, IBSF의 색인 버퍼는 (C)의 왼쪽 삽입 전 상태라고 라고 가정한다. Input 데이터 처리 전 B트리의 상태는 (a)의 왼쪽 트리라고 가정한다. 다음은 (a)의 B트리가 입력 데이터에 따라 왼쪽 트리의 모양에서 오른쪽 트리의 모양으로 변해갈 때 발생하는 색인유닛을 어떻게 처리하는지 BOF와 IBSF의 처리과정을 나타낸다. :

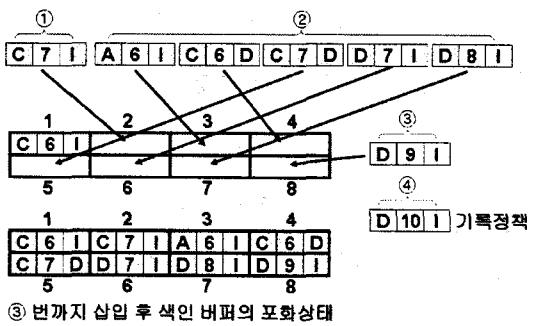
- BOF

① 7번 데이터가 입력되면 이것은 C번 노드에 삽입된다. 그리고 B트리의 삽입을 처리하기 위해 한 개의 삽입관련 색인단위가 발생한다. 발생한 색인단위는 [그림 9]에서 보여주는 것과 같이 색인버퍼의 2번 공간에 저장된다.

② 8번 데이터가 입력되면 C번 노드는 더 이상 데이터를 삽입할 수 없는 상태이다. 따라서 B트리는 노드 분할을 하게 된다. 노드 분할은 C번 노드의 6번 데이터를 C번 노드에서 삭제하고 A번 노드에 삽입한다. 그리고 7번 데이터를 C번 노드에서 삭제하고 D번 노드에 삽입한다. 새로 입력된 8번 데이터는 D번 노드에 삽입하게 된다. 이 연산은 C노드에 대한 두 개의 삭제관련 색인단위와 A노드에 대한 한 개의 삽입관련 색인단위 그리고 D번 노드관련 두 개의 삽입관련 색인단위를 발생한다. 발생한 5개의 색인단위는 [그림 9]에서 보여주는 것과 같이 순서대로 색인 버퍼의 3번 4번 5번 6번 7번 공간에 저장된다.

③ 9번 데이터가 입력되면 이것은 D번 노드에 삽입이 되고 B트리의 삽입을 처리하기 위해 한 개의 삽입관련 색인단위가 발생한다. 발생한 색인단위는 [그림 9]에서 보여주는 것과 같이 색인 버퍼의 8번 공간에 저장된다.

④ 10번 데이터가 입력되면 이것은 D번 노드에 삽입이 되고 B트리의 삽입을 처리하기 위해 한 개의 삽입관련 색인단위가 발생한다. 하지만 [그림 9]의 색인버퍼에서 보여는 것과 같이 BOF의 색인 버퍼는 이미 1번부터 8번까지 다 채워져 있음으로 버퍼를 비우는 기록 연산이 수행된다.



[그림 9] BOF의 색인버퍼 관리

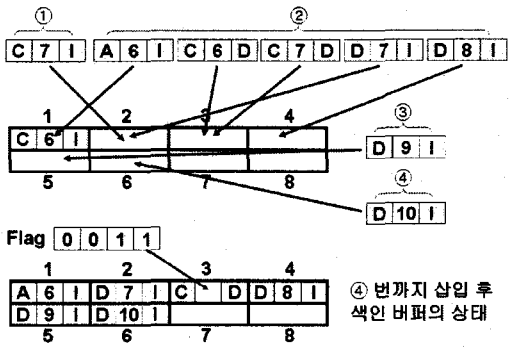
- IBSF

① 7번 데이터가 입력되면 이것은 C번 노드에 삽입이 되고 B트리의 삽입을 처리하기 위해 한 개의 삽입 타입 색인단위가 발생한다. 발생한 색인단위는 [그림 10]에서 보여주는 것과 같이 색인 버퍼의 2번 공간에 입력된다.

② 8번 데이터가 입력되면 C번 노드는 더 이상 데이터를 삽입할 수 없는 상태이다. 따라서 B트리는 노드 분할을 하게 된다. 노드 분할은 C번 노드의 6번 데이터를 C번 노드에서 삭제하고 A번 노드에 삽입한다. 그리고 7번 데이터를 C번 노드에서 삭제하고 D번 노드에 삽입한다. 새로 입력된 8번 데이터는 D번 노드에 삽입하게 된다. 이 연산은 C노드에 대한 두 개의 삭제 타입 색인단위와 A노드에 대한 한 개의 삽입 타입 색인단위 그리고 D번 노드관련 두 개의 삽입 타입 색인단위를 발생한다. 발생한 5개의 색인단위는 [그림 10]에서 보여주는 것과 같이 순서대로 처리된다. IBSF에서는 삭제 정책에 따라 한 노드에 관련된 삭제 타입의 색인단위는 하나의 색인단위에 유지된다. 노드 분할 시 IBSF가 삭제해야 할 색인단위는 노드의 3번째 4번째 데이터이다. 따라서 삭제 색인단위는 3번째 4번째 Flag를 1로 설정한다. [그림 10]에서 보이는 것과 같이 삭제 색인단위는 색인버퍼의 비어있는 3번 공간에 저장된다. 7번 데이터의 경우는 위 ①에서 이미 색인 버퍼에 입력했었다. 따라서 IBSF는 새로운 공간에 7번 데이터 관련 색인단위를 입력하지 않고 2번 공간에 7번 관련 색인단위를 새로운 색인단위로 갱신한다. 6번 데이터의 경우도 이미 색인버퍼의 1번 공간에 삽입관련 색인단위가 존재한다. 따라서 1번 공간에 유지하고 있던 6번 색인단위를 새로운 색인단위로 갱신한다. 그러나 8번 데이터관련 색인단위는 버퍼에서 유지하고 있지 않다. 따라서 버퍼에 비어있는 공간인 4번에 8번 데이터관련 색인단위를 입력한다.

③ 9번 데이터가 입력되면 이것은 C번 노드에 삽입이 되고 B트리의 삽입을 처리하기 위해 한 개의 삽입 타입 색인단위가 발생한다. 발생한 색인단위는 [그림 10]에서 보여주는 것과 같이 색인 버퍼의 5번 공간에 입력된다.

④ 10번 데이터가 입력되면 이것은 C번 노드에 삽입이 되고 B트리의 삽입을 처리하기 위해 한 개의 삽입 타입 색인단위가 발생한다. 발생한 색인단위는 [그림 10]에서 보여주는 것과 같이 색인 버퍼의 6번 공간에 입력된다.

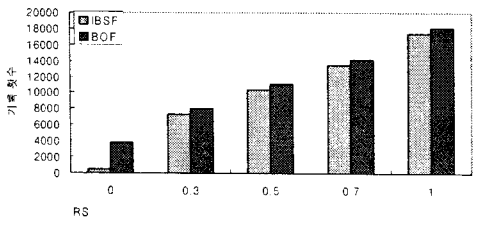


[그림 10] IBSF의 색인버퍼 관리

본 절에서는 BOF와 IBSF의 B트리의 분할 시 색인 버퍼 관리 정책의 차이에 대해 예를 들어서 설명했다. 예제에서 살펴본 것과 같이 색인단위가 중복되었을 경우 IBSF는 BOF와 같은 크기의 색인 버퍼를 이용하여 BOF 보다 많은 색인단위를 처리할 수 있다.

4. 성능 평가

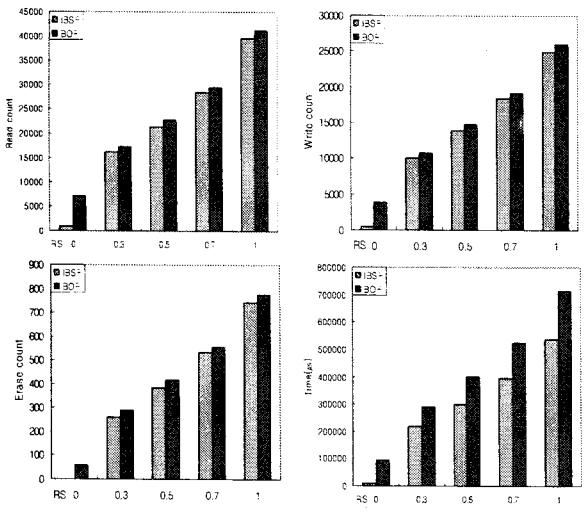
B+트리가 구축 될 때에는 삽입, 삭제, 갱신, 노드분할 등의 수행으로 인해 노드의 변경과 쓰기 연산이 많이 발생한다. 따라서 본 논문에서 제안한 IBSF를 BOF와 비교 평가하기 위해 B+트리가 구축될 때의 시점에서 실험을 수행하였다.



[그림 11] 버퍼의 기록 횟수

B+트리의 Fanout은 21로 설정하였고, B+트리를 구축하기 위해 입력한 사용한 데이터는 키 값이 순차적인 순서에 의한 입력 데이터와 랜덤한 순서의 입력 데이터를 조합하여 사용하였다. 실험은 64M NAND 플래시 메모리 환경에서 수행하였다. 그리고 실험에서 사용한 FTL은 Log-scheme기반의 FAST(Fully associative sector translation)[2]이다.

[그림 11]는 BOF와 IBSF가 순차적인 순서와 랜덤한 순서의 키 값을 조합한 입력데이터를 이용해 색인 단위를 생성하여 삽입하였을 때 몇 번의 기록이 일어났는지를 보여주고 있다. RS는 입력데이터의 키 값이 얼마나 순차적인 순서인지 혹은 얼마나 랜덤 한 순서인지를 나타내는 비율이다. RS가 0에 가까울수록 입력한 데이터의 키 값의 순서가 순차적인 것을 의미하고, 1에 가까울수록 입력한 데이터의 키 값의 순서가 랜덤하게 정렬된 것을 의미한다.



[그림 12] 성능 실험 결과

플래시 메모리는 중첩쓰기가 발생할 경우 내부적으로 무효화된 페이지가(Invalid Page) 발생하게 됨으로 성능저하가 일어난다. [그림 11]에서 IBSF와 BOF간의 기록 횟수는 큰 차이가 없어 보인다. 그러나 실제로 IBSF는 BOF보다 중첩쓰기의 횟수가 줄어든다. 따라서 플래시 메모리상에서 기록을 처리하는데 발생하는 성능은 큰 차이를 나타낼 것이다.

기록된 데이터를 FTL을 통해 플래시 메모리에 저장하는 실험의 결과는 [그림 12]에서 나타내었다. [그림 12]는 입력한 데이터의 RS의 변화에 따라 플래시 메모리의 읽기, 쓰기, 삭제 연산의 횟수와 이 연산을 수행하는데 소모된 시간비용을 보여주고 있다.

실험을 통해 IBSF가 BOF보다 좋은 성능을 보여주는 것을 확인하였다.

5. 결론 및 향후 연구

본 논문은 플래시 메모리상의 인덱스 버퍼를 효율적으로 사용하는 IBSF 방법을 제안하였고, 제안한 방법이 기존의 방법보다 우수함을 증명하였다. 플래시 메모리는 집중적인 중첩쓰기가 발생할 경우 심각한 성능저하를 일으킨다. 따라서 인덱스와 같이 구축 시 중첩쓰기가 빈번하게 일어나는 구조를 플래시 메모리상에 구축하기 위해서는 플래시 메모리의 특성을 보완해 줄 시스템이 필요하다. 특히 B+트리의 색인버퍼를 효율적으로 관리하는 버퍼관리정책은 플래시 메모리의 수명과 성능을 향상시킬 것이다.

본 논문을 통해 색인 버퍼를 어떻게 활용에 따라 달라지는 플래시 메모리의 성능을 확인했다. 앞으로는 플래시 메모리의 특성을 반영하는 향상된 색인 버퍼 관리 정책과, 데이터가 버퍼에 저장될 때의 처리뿐만 아니라 버퍼에서 데이터가 기록 될 때의 처리도 효과적으로 처리 할 수 있는 연구를 다양한 실험을 통해 진행할 예정이다.

참고문헌

[1] D. S. Batory, "B+trees and indexed sequential files: a performance comparison", SIGMOD, 30 - 39, 1981.

[2] Sang-Won Lee et al., "A Log Buffer based Flash Translation Layer using Fully Associative Sector Translation", ACM, 2005.

[3] Intel Corporation, "Understanding the Flash Translation Layer(FTL) Specification".

[4] Air Ban, Ramat Hasharon, "FLASH FILE SYSTEM OPTIMIZED FOR PAGE-MODE FLASH TECHNOLOGIES", Assignee: M-systems Flash Disk Pioneers Ltd., Patent Number: 5,937, 425, Date of Patent: 8/10/1999.

[5] Chin-Hsien Wu, Li-Pin Chang, Tei-Wei Kuo, "An Efficient B-Tree Layer for Flash-Memory Storage Systems", RTCSA, 2003.

[6] 남정현, 박동주, "플래시 메모리 상에서 효율적인 B-트리 설계 및 구현", 한국정보과학회, 2005

[7] 128M x 8 Bit NAND Flash Memory, 삼성전자, 2005