DOI: 10.3745/KIPSTD.2008.15-D.6.755

NAND 플래시 메모리를 위한 로그 기반의 B-트리

김 보 경[†] · 주 영 도[††] · 이 동 호[†††]

요 약

최근 NAND 플래시 메모리는 하드 디스크에 비해 작고, 속도가 빠르며, 저 전력 소모 등의 장점을 가지고 있어 차세대 저장 매체로 각광받고 있다. 그러나 쓰기-전-소거 구조, 비대칭 연산 속도 및 단위와 같은 독특한 특징으로 인하여, 디스크 기반의 시스템이나 응용을 NAND 플래시 메모리 상에 직접 구현시 심각한 성능저하를 초래할 수 있다. 특히 NAND 플래시 메모리 상에 B-트리를 구현할 경우, 레코드의 잦은 삽입, 삭제 및 재구성에 의한 많은 양의 중첩 쓰기가 발생할 수 있으며, 이로 인하여 급격한 성능 저하가 발생할 수 있다. 이러한 성능 저하를 피하기 위해 μ-트리가 제안되었으나, 잦은 노드 분할 및 트리 높이의 빠른 신장 등의 문제점을 가지고 있다.

본 논문에서는 갱신 연산을 위해 특정 단말 노드에 해당하는 로그 노드를 할당하고, 해당 로그 노드에 있는 변경된 데이터를 한 번의 쓰기 연산으로 저장하는 로그 기반의 B-트리(LSB-트리)를 제안한다. LSB-트리는 부모 노드의 변경을 늦추어 추가적인 쓰기 연산의 횟수를 줄일 수 있다는 장점을 가지고 있다. 또한 키 값에 따라 데이터를 순차적으로 삽입할 때, 로그 노드를 새로운 단말 노드로 교환함으로써 추가적인 쓰기 연산의 횟수를 줄일 수 있다. 마지막으로, 다양한 비교 실험을 통하여 μ-트리와 비교함으로써 LSB-트리의 우수성을 보인다.

키워드 : 플래시 메모리, 인덱스 구조, 색인 구조, B-트리, 로그 구조

Log-Structured B-Tree for NAND Flash Memory

Bo-kyeong Kim[†] · Young Do Joo[††] · Dong-Ho Lee[†††]

ABSTRACT

Recently, NAND flash memory is becoming into the spotlight as a next-generation storage device because of its small size, fast speed, low power consumption, and etc. compared to the hard disk. However, due to the distinct characteristics such as erase-before-write architecture, asymmetric operation speed and unit, disk-based systems and applications may result in severe performance degradation when directly implementing them on NAND flash memory. Especially when a B-tree is implemented on NAND flash memory, intensive overwrite operations may be caused by record inserting, deleting, and reorganizing. These may result in severe performance degradation. Although μ-tree has been proposed in order to overcome this problem, it suffers from frequent node split and rapid increment of its height.

In this paper, we propose Log-Structured B-Tree(LSB-Tree) where the corresponding log node to a leaf node is allocated for update operation and then the modified data in the log node is stored at only one write operation. LSB-tree reduces additional write operations by deferring the change of parent nodes. Also, it reduces the write operation by switching a log node to a new leaf node when inserting the data sequentially by the key order. Finally, we show that LSB-tree yields a better performance on NAND flash memory by comparing it to μ-tree through various experiments.

Keywords : Flash Memory, Index Structure, B-Tree, Log-Structure

1. 서 론

비휘발성 저장 매체인 NAND 플래시 메모리는 하드 디스크에 비해 작고, 가볍고, 데이터에 대한 접근이 빠르며, 전력 소모가 적고, 온도나 외부 충격에 뛰어난 내구성을 가지며, 소음이 없는 장점을 가지고 있다. 이같은 장점 때문에 휴대용 전화기, 개인용 단말기(PDA), 노트북 컴퓨터와 같은 이동 컴퓨팅 장비에 저장 장치로 많이 사용되고 있으며, 최근에는 하드 디스크를 대체할 저장 매체로 주목 받고 있다.

플래시 메모리는 하드 디스크와 다르게 다음과 같은 독특한 특징을 가지고 있다. 첫째, 제자리 갱신(in-place update)이 불가능하기 때문에, 갱신 연산이 일어나는 페이지에 중첩 쓰기(overwrite)를 수행할 경우 해당 블록을 소거한 후에 데이터를 기록하여야 한다. 둘째, 읽기·쓰기·소거 연산의 수행 속도가 다르다. <표 1>과 같이 읽기 연산이 가장 빠르며, 소

※ 본 연구는 정보통신부 및 정보통신연구진흥원의 IT신성장동력핵심기술개발사업[2006-S-040-01, Flash Memory 기반 임베디드 멀티미디어 소프트웨어 기술 개발]과 2007년 정부(교육인적자원부)의 재원으로 한국학술진흥재단의 지원(KRF-2007-313-D00757)을 받아 수행된 연구임.
† 준 회 원 : 한양대학교 컴퓨터공학과 석사과정
†† 정 회 원 : 강남대학교 컴퓨터미디어공학부 부교수
††† 정 회 원 : 한양대학교 컴퓨터공학과 조교수
논문접수 : 2008년 8월 25일
수 정 일 : 1차 2008년 9월 30일, 2차 2008년 10월 23일
심사완료 : 2008년 10월 23일

〈표 1〉 NAND 플래시 메모리 연산 속도[1,2]

연 산(단위) / 블 록	읽 기 (페이지)	쓰 기 (페이지)	소 거 (페이지)
소 블록 (64M × 8Bit)	15μs	200μs	2ms
대 블록 (2G,4G,8G × 8Bit)	25μs	200μs	1.5ms

거 연산이 가장 느리다[1, 2]. 셋째, 읽기·쓰기 연산은 페이지 단위로 수행하고, 소거 연산은 블록 단위로 수행한다. 이같은 특징을 고려하지 않은 채 읽기 연산에 비해 상대적으로 느린 쓰기·소거 연산이 많이 발생하는 시스템을 설계시 성능 저하를 초래하게 된다[7, 8].

하드 디스크 기반의 색인 구조인 B-트리는 구조적으로 균형이 잘 잡혀있어 삽입·삭제·갱신 연산이 빈번하게 발생하여도 빠른 검색 속도를 보장해주기 때문에 기존 시스템에서 널리 사용되었다[5, 6]. 그러나 플래시 메모리를 저장매체로 일정 페이지에 중첩 쓰기가 잦은 B-트리를 구축하게 되면 제자리 갱신이 불가능하여 읽기 연산에 비해 상대적으로 느린 쓰기·소거 연산이 많이 발생하게 된다. 이는 시스템 성능 저하의 주요 원인이 되기 때문에 읽기 연산을 많이 수행하더라도 쓰기·소거 연산을 줄이는 방식의 NAND 플래시 메모리에 특징을 고려한 새로운 B-트리 설계가 필요하게 되었다[3, 4, 9, 10, 11].

NAND 플래시 메모리를 위한 색인 구조로 BFTL[9], Wandering 트리[3], μ-트리[4] 등이 제안되었다. 여기서 BFTL은 B-트리 구축 시 발생하는 쓰기 연산을 줄이기 위해 변경되는 데이터를 예약 버퍼(reservation buffer)에 모아 두었다가 한 번의 쓰기 연산으로 처리한다. 이는 다수의 쓰기 연산을 한 번의 쓰기 연산으로 처리하여 시스템의 성능 향상시켰지만, 갑작스런 전원 차단 시 예약 버퍼에 저장된 키 값, 레코드 포인터 등과 같은 실제 데이터가 모두 지워지는 치명적인 문제점을 가지고 있다. 또한 예약 버퍼에 저장되는 실제 데이터와 노드 사상 정보를 값비싼 메모리에 유지하여야 하기 때문에 비용적인 측면에서도 문제점을 가지고 있다. 이러한 문제점을 가지는 BFTL은 메모리가 제한적인 임베디드 장비에 적합하지 못하여 실제 구현에 어려움을 가지고 있다. 그러나 Wandering 트리와 μ-트리는 BFTL과 다르게 버퍼를 사용하지 않고 실제 데이터를 플래시 메모리에 직접 저장하는 방식의 색인 구조이다. 여기서 Wandering 트리는 갱신 연산 시 변경되는 모든 노드를 플래시 메모리에 새로운 페이지를 할당하여 저장한다. 이처럼 변경되는 모든 노드를 저장하기 때문에 쓰기 연산이 많이 수행된다. 이러한 쓰기 연산을 줄이기 위해 μ-트리는 갱신 연산이 발생 시 변경되는 노드를 플래시 메모리의 한 페이지에 모아서 한 번의 쓰기 연산으로 저장한다. 그러나 μ-트리는 B-트리에 비해 페이지를 비효율적으로 관리하기 때문에 검색 성능 저하되고, 구조적으로 노드 분할과 트리 신장이 빈번하게 일어나 추가적인 쓰기 연산이 많이 발생한다.

본 논문에서는 이러한 μ-트리의 문제점을 해결하기 위해 갱신 연산이 일어나는 B-트리의 단말 노드에 로그 노드를 할당하여 변경되는 데이터를 한 번의 쓰기 연산으로 저장하고, 상위 부모 노드의 변경을 최대한 늦추어 시스템의 성능을 향상 시킨 NAND 플래시 메모리를 위한 로그 기반의 B-트리(LSB-Tree: Log-Structured B-Tree)를 제안한다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구인 NAND 플래시 메모리를 위한 색인 구조에 대해 알아보고 문제점을 살펴본다. 3장에서는 본 논문에서 제안하는 색인 구조인 LSB-트리를 기술하고, 4장과 5장에서는 관련 연구인 μ-트리와 수학적 모델을 통하여 비교하고, 성능을 측정한다. 마지막으로 6장에서는 결론과 함께 향후 연구 방향에 대해 기술한다.

2. 관련 연구

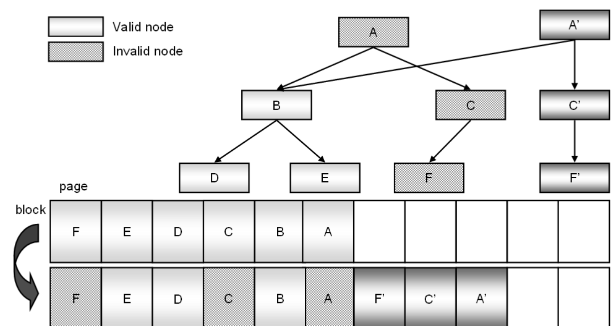
2.1. Wandering 트리

Wandering 트리[3]는 NAND 플래시 메모리 전용 파일 시스템 JFFS3의 색인 구조로, B-트리 형태를 가진다. 갱신 연산을 수행할 때, 변경되는 노드만을 플래시 메모리의 새로운 페이지에 할당하여 저장한다. (그림 1)은 트리가 갱신 연산을 수행할 때, 플래시 메모리에 저장되는 과정을 보여준다. 만약 노드 F에 갱신 연산이 발생하여 부모 노드인 노드 C, A에 변경이 발생하면, 플래시 메모리의 새로운 페이지에 변경되는 노드 F′, C′, A′를 각각 저장한다. 따라서 데이터 삽입시 변경이 발생하는 모든 노드에 대한 쓰기 연산이 불가피하다는 단점을 가지고 있다.

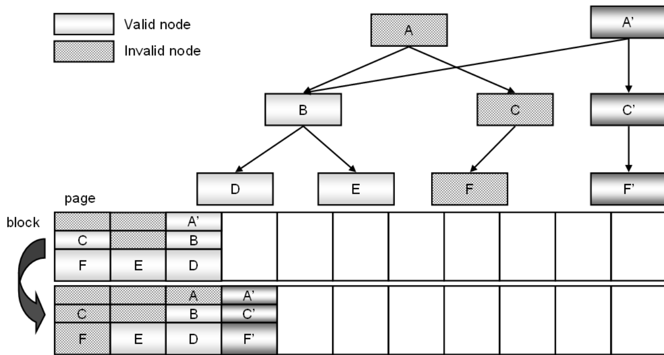
2.2 μ-트리

NAND 플래시 메모리를 위한 색인 구조인 μ-트리[4]는 Wandering 트리의 단점을 해결하기 위해, 변경되는 모든 노드를 플래시 메모리의 한 페이지에 모아서 저장한다. (그림 2)는 μ-트리 상에서 갱신 연산이 발생할 때, 플래시 메모리에 저장되는 과정을 보여준다. 노드 F에 갱신 연산이 일어나면 플래시 메모리에 변경되는 노드 F, C, A를 한 페이지에 모아서 저장한다.

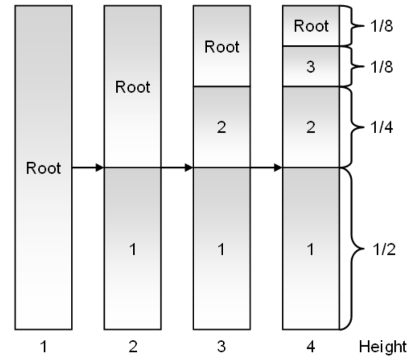
이같이 μ-트리는 한 페이지에 변경되는 모든 노드를 저장하기 위해서 (그림 3)과 같은 고정된 페이지 레이아웃 구



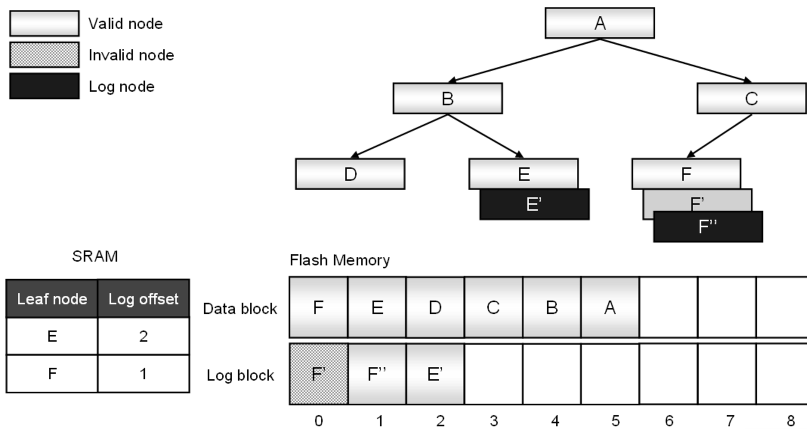
(그림 1) Wandering 트리 삽입 예제



(그림 2) μ-트리 삽입 예제



(그림 3) μ-트리의 페이지 레이아웃 구조



(그림 4) LSB-트리 개요

조를 사용한다. 트리의 높이가 2이상 일 때 모든 단말 노드는 페이지의 반을 사용하며, 트리가 신장할 때마다 루트 노드의 엔트리 개수는 절반으로 줄어든다. μ-트리는 페이지의 절반을 단말 노드로 사용하기 때문에 B-트리에 비해 단말 노드의 엔트리가 빨리 소모되어 노드 분할이 잦아지고, 루트 노드의 엔트리 또한 빨리 소모되어 트리 신장이 빨라진다. 이같은 잦은 노드 분할과 트리의 신장은 추가적인 쓰기 연산을 발생하여 시스템의 성능 저하를 초래하게 된다.

3. LSB-트리(Log-Structured B-Tree)

3.1 주요 아이디어

본 논문에서는 μ-트리의 비효율적인 페이지 관리로 인한 잦은 노드 분할과 트리 신장으로 발생하는 추가적인 쓰기 연산을 줄이기 위해 NAND 플래시 메모리를 위한 로그 기반의 B-트리(LSB-트리)를 제안한다. LSB-트리는 Wandering-트리와 유사하지만 갱신 연산이 발생하는 단말 노드에 로그 노드를 할당하여 변경되는 데이터를 한 번의 쓰기 연산으로 저장하는 것이 다르다. 만약 로그 노드가 가득 차게 되면, 합병 연산을 통해 단말 노드와 변경되는 부모 노드를 갱신하여 저장한다. 이처럼 LSB-트리는 로그 노드가 가득차기 전까지, 단말 노드의 갱신으로 변경되는 상위 부모 노드의 변화를 최대한 늦춤으로써 추가적인 쓰기 연산을 줄인다.

(그림 4)와 같이 단말 노드 E, F가 갱신되면 단말 노드와 함께 변경되는 부모 노드를 모두 갱신하는 것이 아니라, 변경되는 단말 노드 E, F에 대해 로그 노드 E′, F′를 할당하여 저장한다. 단말 노드 F에 한 번 더 갱신이 일어나면 로그 노드 F′를 무효화시키고 새로운 로그 노드 F′′를 할당하여 저장한다. 이때 변경되는 로그 노드에 대한 주소를 메모리의 로그 사상 테이블에서 관리한다. 만약 단말 노드 F의 로그 노드가 가득 차게 되면 해당 로그 노드와 단말 노드 F를 합병하고, 부모 노드 C, A를 무효화시키고 새롭게 변경되는 부모 노드를 데이터 블록에 할당하여 저장한다.

본 논문에서 제안하는 LSB-트리는 NAND 플래시 메모리상에서 B-트리를 구현할 때, 단말 노드의 갱신으로 인해 변경되는 모든 부모 노드의 변화를 즉시 반영하는 것이 아니라, 갱신되는 단말 노드에 해당하는 로그 노드를 할당하여 변경되는 정보를 한 번의 쓰기 연산으로 저장한다. 이때 로그 노드가 가득 차게 되면, 단말 노드와 해당 로그 노드를 합병하고 변경되는 모든 부모 노드를 새로운 페이지에 할당하여 저장한다. LSB-트리는 단말 노드의 변경을 로그 노드에 한 번의 쓰기 연산으로 수행함으로써 부모 노드의 변화를 최대한 늦추고, 불필요한 쓰기·소거 연산의 횟수를 효과적으로 줄인다.

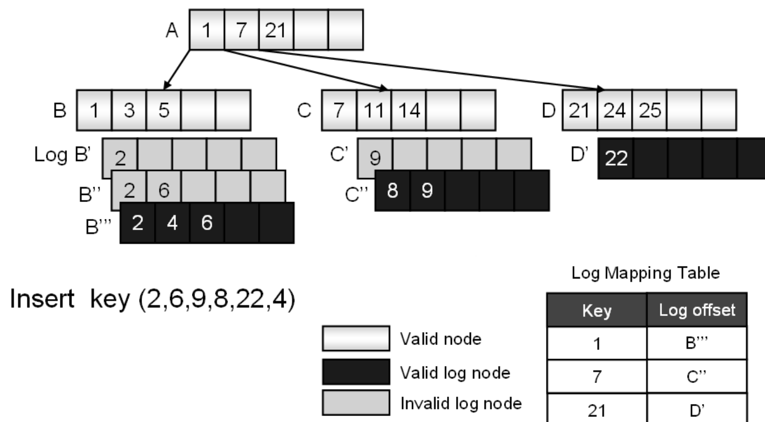
Algorithm 1 Insertion (*key*)

1: *offset* := *root node address*
2: allocate a new page *N*
3: **for** *i* = 1 to *height* - 1
4: *CurrentNode* ⇐ read the node from *offset*
5: *offset* := find child node address of *CurrentNode* with the *key*
6: **end for**
7: *ParentNode* ⇐ read the node from *offset*
8: *log_offset* := find log node address of *ParentNode* with the *key* in the log mapping table
9: **if** *log_offset* is NOT EXIST **then**
10: allocate a new log node *LogNode*
11: **else**
12: *LogNode* ⇐ read the node from *log_offset*
13: invalidate page of *log_offset*
14: **end if**
15: insert *key* into the log node *LogNode*
16: **if** *LogNode* is FULL **then**
17: *offset* := find child node address of *ParentNode* with the *key*
18: *LeafNode* ⇐ read the node from *offset*
19: merge *LogNode* with *LeafNode*
20: **else**
21: write node *LogNode* on page *N*
22: update the log mapping table
23: **end if**

3.2 삽입 연산

LSB-트리의 삽입 연산은 알고리즘 1의 의사 코드와 같다. 우선 3-6번째 줄과 같이 부모 노드를 재귀적으로 검색하여 삽입할 키 값의 자식 노드를 찾는다. 그러나 B-트리와 다르게 단말 노드를 찾기 전, 단말 노드의 로그 노드를 먼저 찾는다. 7-8번째 줄과 같이 로그 노드를 찾기 위해서 로그 사상 테이블에서 부모 노드와 검색할 키 값을 이용하여 로그 노드의 주소를 찾는다. 9-10번째 줄과 같이 로그 노드 주소를 찾지 못할 경우, 새로운 로그 노드를 메모리에 할당한다. 11-14번째 줄과 같이 로그 노드 주소를 찾을 경우, 로그 노드의 페이지를 메모리에 저장하고, 이미 읽은 로그 노드 주소의 페이지를 플래시 메모리에서 무효화시킨다. 15번째 줄과 같이 메모리의 로그 노드에 키 값을 삽입한다. 16-19번째 줄과 같이 로그 노드에 키 값을 삽입시 로그 노드가 가득 차게 되면 해당 로그 노드의 단말 노드를 찾아 합병 연산을 수행한다. 그렇지 않은 경우 20-23번째 줄과 같이 플래시 메모리에 할당된 새로운 페이지에 키 값을 삽입한 로그 노드를 저장하고, 로그 사상 테이블을 갱신하고 삽입 연산을 마친다.

(그림 5)와 같이 4개의 노드 A, B, C, D를 가지는 트리가



(그림 5) LSB-트리 삽입·검색 연산 예제

존재하고 키 값 2, 6, 9, 8, 22, 4가 순차적으로 삽입 된다고 가정하자. 키 값 2가 삽입되면 단말 노드 B에 갱신 연산이 수행되는데 로그 노드가 존재하지 않기 때문에 로그 노드 B′를 할당하여 키 값 2를 삽입하고, 로그 사상 테이블에 부모 노드의 키 값 1과 로그 노드의 주소 B′를 저장한다. 다음으로 키 값 6이 삽입되면 단말 노드 C에 로그 노드 C′를 할당하여 키 값 9를 삽입하고, 로그 사상 테이블에 부모 노드의 키 값 7과 로그 노드의 주소 C′를 저장한다. 그러나 키 값 9가 삽입되면 단말 노드 B의 해당 로그 노드 B′가 존재하기 때문에 로그 노드 B′를 무효화 시키고, 새로운 로그 노드 B′′를 할당하여 키 값 9를 삽입하고, 로그 사상 테이블의 키 값 1과 로그 노드의 주소 B′′를 저장한다. 키 값 8, 22, 4도 이와 마찬가지로 삽입 연산을 수행한다. 이처럼 대부분의 삽입 연산은 로그 노드의 엔트리가 가득차기 전까지 한 번의 쓰기 연산으로 수행하며, 엔트리가 가득 차게 되면 합병 연산을 수행한다.

3.3 검색 연산

LSB-트리의 검색 연산은 B-트리와 유사하게 루트 노드로부터 키 값을 검색하여 자식 노드를 찾고, 재귀적으로 키 값을 검색하여 단말 노드를 찾아낸다. 그러나 B-트리와 다르게 단말 노드를 찾기 전 단말 노드의 부모 노드를 찾게 되면 로그 사상 테이블을 검사하여 해당 단말 노드의 로그 노드의 주소를 얻고, 이 주소를 통해 우선적으로 키 값을 검색한다. 그러나 로그 노드에 검색할 키 값이 존재하지 않은 경우, 해당 단말 노드를 검색하여 키 값을 검색한다. 이 같이 로그 노드를 먼저 검색하는 이유는 로그 노드의 최신의 유효 데이터가 존재할 가능성이 높기 때문이며, 또한 불필요한 읽기 연산을 줄이기 위해서이다.

(그림 5)의 LSB-트리에서 키 값 9를 검색하게 되면 루트 노드 A를 검색하여 단말 노드 C를 찾게 되지만, 노드 C를 검색하기 전 로그 사상 테이블을 검사하여 노드 C의 로그 노드 C′′를 우선적으로 검색하여 해당 키 값 9를 찾아낸다. 이처럼 로그 노드를 먼저 검색하여 키 값을 찾게 되면, B-트리의 읽기 연산(트리 높이)과 동일한 횟수로 키 값을 찾아낼 수 있게 된다. 그렇지 않을 경우, 한 번의 추가적인 읽기 연산이 더 필요하게 된다.

3.4 삭제 연산

LSB-트리의 삭제 연산은 검색 연산을 수행하여 해당 로그 노드를 찾고, 해당 키 값을 제거하여 새로운 페이지에 저장한다. 이때, 해당 키 값이 로그 노드에 존재하지 않고 단말 노드에만 존재하는 경우, 삭제할 엔트리를 로그 노드에 삽입하여 새로운 페이지에 저장하고, 합병 연산 수행 시 단말 노드의 엔트리에서 제거하고 부모 노드를 변경한다.

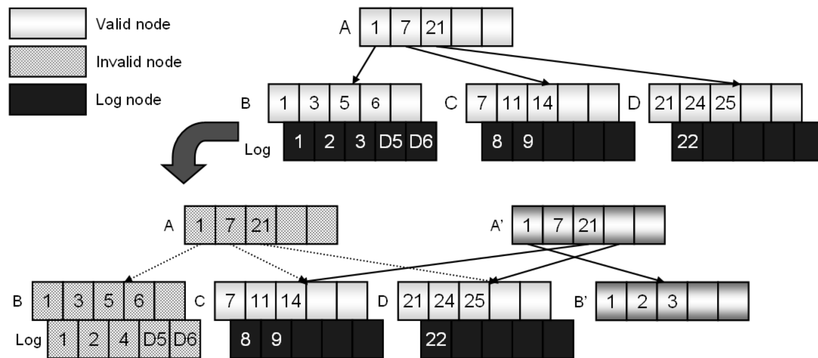
(그림 6)은 삭제 연산 시 발생하는 합병 과정이다. 만약 (그림 6)의 노드 B에서 키 값 5, 6을 삭제하면, 로그 노드에 엔트리에 삭제할 키 값 5, 6을 삽입한다. 그리고 로그 노드가 가득 차게 되어 합병 연산이 수행되면, 해당 단말 노드의 키 값 5, 6을 제거하여 새로운 노드 B′에 저장하고, 변경되는 부모 노드 A를 새로운 노드 A′에 갱신하여 저장한다.

3.5 합병 연산 및 교환 연산

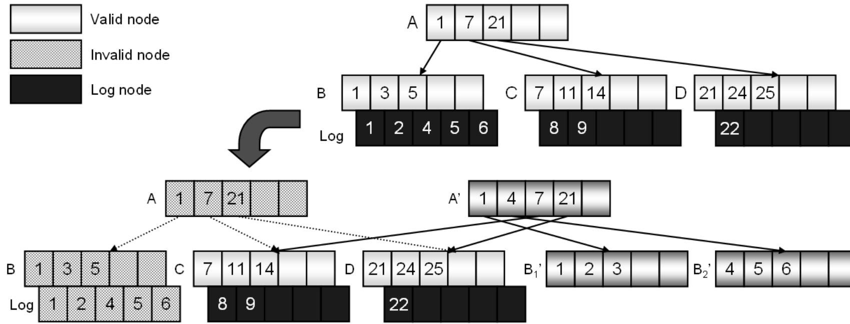
LSB-트리의 합병 연산은 삽입 연산을 수행할 때, 로그 노드가 가득 차게 되면 해당 단말 노드와 합병하여 저장하고, 변경되는 부모 노드를 갱신하여 저장한다. (그림 7)은 합병 연산 과정을 보여준다. 4개의 노드 A, B, C, D가 있는 트리가 존재할 때, 키 값 1, 2, 4, 5, 6이 갱신되어 로그 노드가 가득 차게 되면, 단말 노드 B와 해당 로그 노드를 합병하게 된다. 이때 로그 노드에 갱신된 최신 키 값 1, 2, 4, 5, 6과 단말 노드 B에 갱신되지 않은 유효한 키 값 3이 합병되어 키 값 1, 2, 3, 4, 5, 6이 된다. 이때 1, 2, 3, 4, 5, 6은 단말 노드의 엔트리 개수 보다 많기 때문에 노드가 두 개로 분할된다. 따라서 분할되는 노드 B_1'(1, 2, 3)와 B_2'(4, 5, 6)를 변경되는 부모 노드 A′와 함께 플래시 메모리의 새로운 페이지에 저장한다. 이 때 합병 연산은 '트리 높이+1'만큼의 쓰기 연산이 필요하다.

그러나 (식 1)의 두 가지 조건 중 하나라도 만족하게 되면 교환 연산을 수행한다. LSB-트리의 교환 연산은 해당 로그 노드를 단말 노드와 합병하지 않고, 로그 노드를 바로 단말 노드로 교환하는 것이다. 식 1의 E_{log}는 로그 노드의 엔트리가 가지는 키 값의 집합, E_{ext}는 단말 노드의 엔트리가 가지는 키 값의 집합이다.

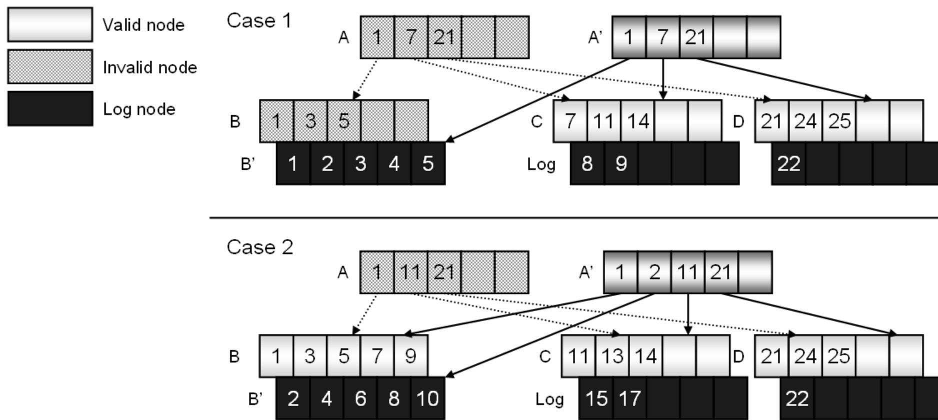
$$\begin{cases} Condition1: E_{ext} \subset E_{\log} \\ Condition2: E_{ext} \cap E_{\log} = \Phi \end{cases} \quad (식\ 1)$$



(그림 6) LSB-트리 삭제 연산 예제



(그림 7) LSB-트리 합병 연산 예제



(그림 8) LSB-트리 교환연산 예제

(그림 8)의 경우 1처럼 노드 B의 키 값이 1, 3, 5이고 로그 노드 B′의 키 값이 1, 2, 3, 4, 5이면 (식 1)의 조건 1을 만족하기 때문에 합병 연산시 로그 노드 B′를 단말 노드로 교환하고, 변경되는 부모 노드 A′를 플래시 메모리의 새로운 페이지에 저장한다. (식 1)의 조건 1은 갱신 연산이 빈번하게 발생하는 경우에 해당하게 된다.

(그림 8)의 경우 2처럼 노드 B의 키 값이 1, 3, 5, 7, 9이고 로그 노드 B′의 키 값이 2, 4, 6, 8, 10이면 (식 1)의 조건 2를 만족하기 때문에 합병 연산 시 로그 노드 B′를 단말 노드로 교환하고, 변경되는 부모 노드 A′를 플래시 메모리의 새로운 페이지에 저장한다. (식 1)의 조건 2는 순차적인 키 값의 삽입 연산이 잦은 경우에 해당하게 된다.

교환 연산이 수행될 경우 '트리 높이-1'만큼의 쓰기 연산을 하기 때문에 합병 연산('트리 높이+1'의 쓰기 연산)보다 추가적인 쓰기 연산을 2회 줄일 수 있게 되어, 시스템의 성능 향상을 가져오게 된다. 순차적인 키 값의 삽입 연산이 많을 경우, (식 1)의 조건 2를 만족하기 때문에 합병 연산으로 발생하는 추가적인 쓰기 연산은 교환 연산을 수행됨으로써 대폭 줄어들게 있게 된다. 또한 트리 생성시 키 값의 삽입에 중복이 없기 때문에 (식 1)의 조건 2를 대부분 만족하게 되어 교환 연산을 수행하게 된다. 이같은 교환 연산이 많아지면 트리의 생성 시간이 짧아지기 때문에 시스템의 성능 향상을 가져오게 된다.

3.6 전원 회복

전원 회복을 위해 기본적으로 검색 연산을 제외한 모든 연산 시 플래시 메모리 페이지의 부가 영역(spare area)에 트리의 높이, 논리 주소, 로그 노드 구분자, 시간 등의 메타 정보를 기록한다. 전원 차단 후 전원이 회복이 될 경우 플래시 메모리의 부가 영역을 모두 검사하여 로그 사상 테이블을 재구축한다.

4. 성능 분석

본 논문에서는 성능 분석을 위해 LSB-트리와 관련 연구인 μ-트리를 연산 비용 및 공간 효율을 수식을 통하여 비교하여 분석한다. 분석을 돕기 위해서, 본 논문에서는 다음과 같이 가정한다. 쓰레기 수집(garbage collection)이 발생하지 않도록 빈 블록이 충분히 있고, 주 메모리의 연산에 대한 비용은 계산하지 않는다.

4.1 연산 비용

<표 2>는 LSB-트리의 합병 연산이나 노드 분할이 일어나지 않았을 때의 각 트리의 기본적인 연산 비용이다. 여기서 사용된 T_R는 플래시 메모리의 읽기 연산에 소모되는 시간, T_W는 플래시 메모리의 쓰기 연산에 소모되는 시간, H_{LSB}는 LSB-트리의 높이, H_μ는 μ-트리의 높이, K_{log}는 해

〈표 2〉 연산 비용

연 산	LSB-트리	μ-트리
검 색	$R_{LSB}=\begin{cases} T_R\times H_{LSB} & \text{if } key\in K_{\log} \\ T_R\times(H_{LSB}+1) & else \end{cases}$	$R_\mu = T_R\times H_\mu$
삽입·삭제	$W_{LSB}=R_{LSB}+T_W$	$W_\mu = R_\mu + T_W$

당 단말 노드의 로그 노드 키 값의 집합이다.

읽기 연산 및 삽입·삭제 연산을 비교해 보면 LSB-트리와 μ-트리는 각 트리의 높이에 따라서 그 성능의 차이가 나타남을 알 수 있다. 이를 살펴보기 위해 LSB-트리에 n개의 키 값을 가질 때 의 높이를 구해보면 (식 2)를 통하여 높이 (식 3)을 구한다. 여기서 f는 한 페이지에 들어갈 수 있는 최대 엔트리의 개수, l은 트리의 깊이, d_l은 트리의 깊이 l일 때 들어갈 수 있는 최대 엔트리의 개수이다.

$$n=\prod_{i=1}^{H_{LSB}} d_i = f^{H_{LSB}} \qquad \text{(식 2)}$$

$$H_{LSB}=\log_f n \qquad \text{(식 3)}$$

d_l = f라고 가정하고, (식 4)와 (식 5)를 통하여 μ-트리에 n개의 키 값을 가질 때의 높이를 구하면 (식 6)와 같다[4].

$$d_l=\begin{cases} f/2^{l-1} & \text{if } root(l=H_\mu) \\ f/2^l & otherwise\,(l<H_\mu) \end{cases} \qquad \text{(식 4)}$$

$$n=\prod_{i=1}^{H\mu} d_i = 2\prod_{i=1}^{H\mu}(f/2^i) \qquad \text{(식 5)}$$

$$H_\mu=-\log_2\frac{\sqrt{2}}{f}-\sqrt{\log_2^2\frac{\sqrt{2}}{f}-2\log_2\frac{n}{2}} \qquad \text{(식 6)}$$

(그림 9)는 (식 3)과 (식 6)을 통하여 f가 256일 때의 각 트리의 높이를 구한 것이다. 여기서 플래시 메모리의 대 블록 페이지는 2Kbyte이고, 엔트리의 최소 크기는 키 값, 레코드 포인터만을 가졌을 경우 8byte가 되기 때문에, 최대 엔트리의 개수 f는 256이 된다. (그림 9)를 통하여 알 수 있듯이, 키 값이 많이 삽입되면 높이의 차가 1보다 더 커지게 된다. 실제적으로는 (식 5)와 같이 d_l이 f와 같지 않기 때문에 트리의 높이가 커질수록 d_l의 값이 작아지므로, (그림 9)보다 높이의 차가 더 커지게 된다. 따라서 각각의 연산에 소모되는 시간은 <표 2>의 수식과 같이 트리의 높이에 영향을 받기 때문에 (식 3), (식 6) 그리고 (그림 9)를 통하여 μ-트리보다 트리의 높이가 더 낮은 LSB-트리가 검색 연산 및 삽입·삭제 연산이 더 빠르게 수행됨을 알 수 있다.

4.2 공간 효율

μ-트리는 구조상으로 상위 부모 노드가 단말 노드보다 엔트리를 적게 가지고 있기 때문에, 노드 분할 및 트리 신장이 잦아 LSB-트리에 비해 같은 키 값의 삽입시 더 많은 페이지 할당이 필요하다. 만약 모든 단말 노드가 가득 차 있다고 가정하고, 페이지 할당량을 구하면 (식 7)의 P_{LSB}는 LSB-트리에 할당된 페이지 수이고. 여기서 P_{log}는 로그 노드에 할당된 페이지 수이다.

$$P_{LSB}=\left\lceil \frac{n}{d_1} \right\rceil+\left\lceil \frac{n}{d_1 d_2} \right\rceil+\ldots+\left\lceil \frac{n}{d_1 d_2 \ldots d_{H_{LSB}}} \right\rceil+P_{\log} \qquad \text{(식 7)}$$

(식 2)를 적용하면,

$$P_{LSB}=\left\lceil \frac{n}{f} \right\rceil+\left\lceil \frac{n}{f^2} \right\rceil+\ldots+\left\lceil \frac{n}{f^{H_{LSB}}} \right\rceil+P_{\log} \qquad \text{(식 8)}$$

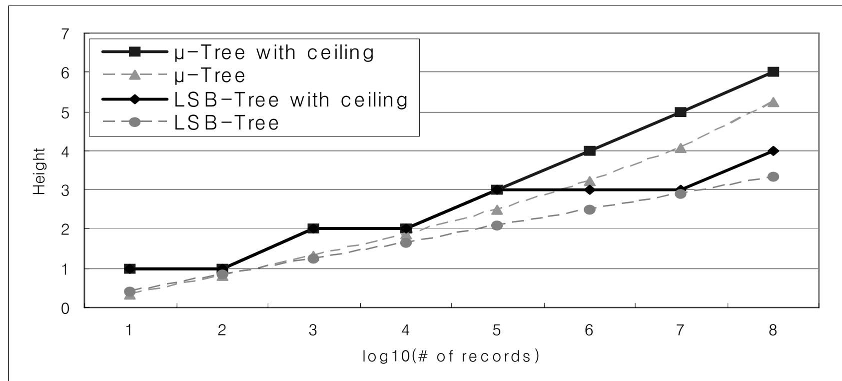
일반적으로 f가 커지게 되면(f>100), $\left\lceil \frac{n}{f^2} \right\rceil+\ldots+\left\lceil \frac{n}{f^{H_{LSB}}} \right\rceil$는 무시할 수 있다. 따라서 LSB-트리에 할당된 페이지 수 P_{LSB}는 (식 9)가 된다.

$$P_{LSB}\approx\left\lceil \frac{n}{f} \right\rceil+P_{\log} \qquad \text{(식 9)}$$

(식 10)의 P_μ는 μ-트리의 할당된 페이지 수이고, 계산상 어려움으로 단말 노드가 있는 페이지만을 계산한다[4].

$$P_\mu=\left\lceil \frac{n}{d_1} \right\rceil=\left\lceil \frac{n}{f/2} \right\rceil \qquad \text{(식 10)}$$

(식 9)와 (식 10)을 비교하여, (식 11)을 구한다.



(그림 9) μ-트리와 LSB-트리의 높이 비교

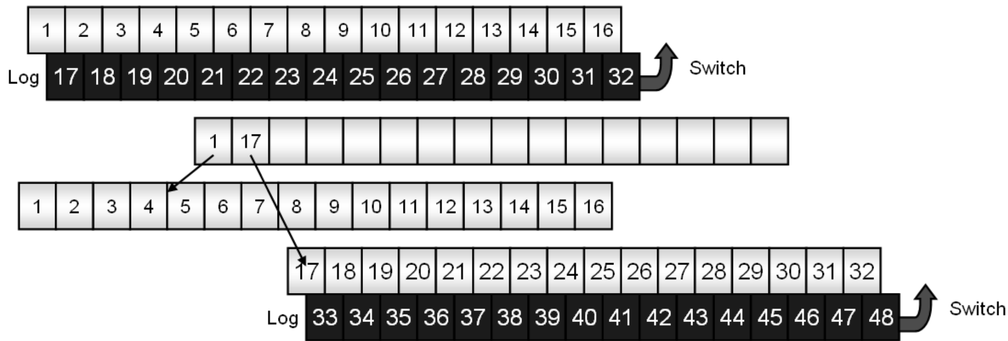
$$P_{LSB} \approx \frac{P_{\mu}}{2} + P_{\log} \qquad (식\ 11)$$

(식 11)에서 알 수 있듯이, LSB-트리에 할당된 페이지 수는 μ-트리에 할당된 절반의 페이지 수와 추가적으로 로그 노드에 할당된 페이지 수이다. 여기서 정렬된 키 값의 삽입으로만 트리가 생성된다면 교환 연산의 발생으로 로그 노드에 할당된 페이지 수 P_{log}는 0 또는 1이 되기 때문에 μ-트리에 할당된 절반의 페이지로만 LSB-트리를 생성할 수 있다. 또한 모든 단말 노드가 로그 노드를 가지고 있다고 하더라도, 부모 노드는 로그 노드를 가지고 있지 않기 때문에 P_{log}는 총 페이지 수 $\left\lceil \frac{n}{f} \right\rceil$ 보다 적어 $P_{LSB} < 2\left\lceil \frac{n}{f} \right\rceil = P_{\mu}$이 된다. 여기서 P_{log}가 최대값이 되더라도, 계산상으로 LSB-트리의 할당된 페이지 수가 μ-트리의 할당된 페이지 수보다 적음을 알 수 있다.

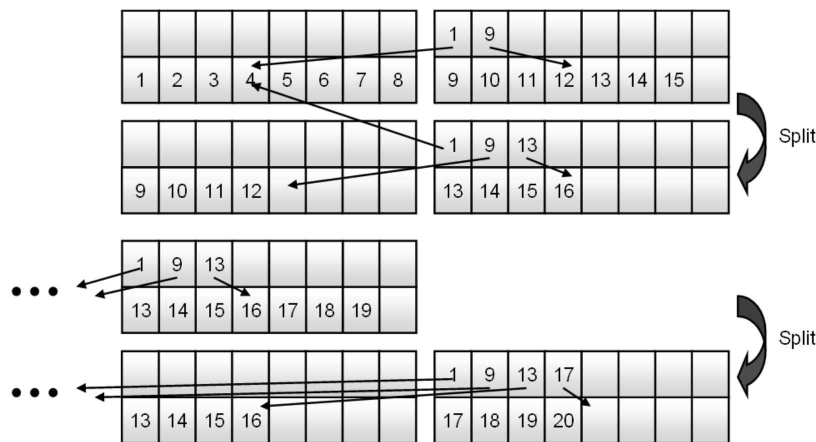
4.3 예제를 통한 비교 분석

(그림 10)은 페이지 엔트리 크기가 16인 μ-트리에 순차적인 키 값을 삽입 할 때의 과정으로 단말 노드 엔트리 수의 반인 4개의 키 값이 삽입될 때 마다 노드 분할이 일어나는 것을 알 수 있다. (그림 11)은 페이지 엔트리 크기가 16인 LSB-트리에 순차적인 키 값을 삽입 할 때의 과정으로 로그 노드에 16개의 키 값이 삽입될 때 마다 로그 노드가 단말 노드로 교환되어 부모 노드 변경이 일어나는 것을 알 수 있다. μ-트리는 노드 분할이 발생할 때 추가적인 쓰기 연산이 발생하지만, LSB-트리는 부모 노드 변경이 일어 날 때 추가적인 쓰기 연산이 발생한다. (그림 10, 11)에서 알 수 있듯이 LSB-트리는 μ-트리에 비해 4배의 키 값이 삽입될 때마다 추가적인 쓰기 연산을 하는 것을 알 수 있고, 이것은 LSB-트리의 트리 생성 시간이 μ-트리보다 적게 걸리는 것을 의미한다.

(그림 10, 11)과 같이 페이지 엔트리 크기가 16이고, 트리의 높이가 4일 때, μ-트리 단말 노드의 최대 엔트리 개수는 128개($8 \times 4 \times 2 \times 2$)이며, LSB-트리 단말 노드의 최대 엔트리 개수는 65536개(16^4)이다. 이처럼 LSB-트리는 높이 4에서 μ-트리의 512배 엔트리를 다룰 수 있으며, 128개의 엔트리(높이 4일 때의 μ-트리 최대 엔트리 수)는 높이 2(μ-트리의 반)로 표현이 가능하다. 이때 LSB-트리는 μ-트리보다 2배 정도 빠르게 키 값을 검색할 수 있다. 만약 128개의 순차적인 키 값을 각각의 트리에 삽입하게 되면, μ-트리는 160번($128+(128/4)+\alpha$)이상의 쓰기 연산을 하고, LSB-트리는 136번($128+1 \times (128/16)$)의 쓰기 연산으로 가능한 것을 알 수 있다(α는 트리 신장에 따른 추가 연산). 이와 같이 LSB-트리는 μ-트리보다 쓰기 연산이 적게 일어나 트리 생성이 빠르게 이루어지는 것을 계산을 통해 알 수 있다.



(그림 10) LSB-트리 순차적인 키 삽입 예제



(그림 11) μ-트리 순차적인 키 삽입 예제

5. 성능 평가

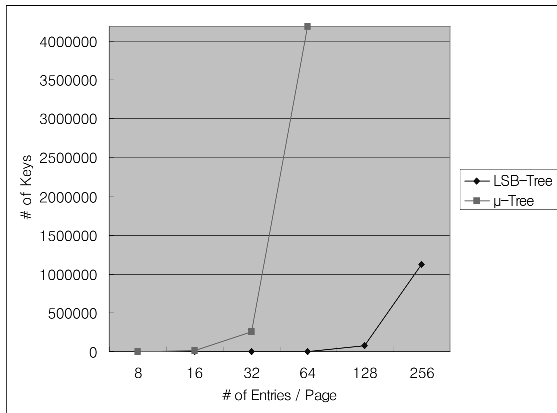
5.1 성능 평가 및 분석

본 논문에서는 LSB-트리를 평가하기 위하여 μ-트리[4]와 성능을 비교하였다. 각각의 색인 구조와 읽기·쓰기·소거 연산 횟수 측정이 가능한 플래시 메모리 시뮬레이터를 C언어로 구현하였다. 각각의 색인 구조에 다양한 패턴의 키 값을 삽입하고, 수행된 연산의 횟수를 계산하여 소모되는 시간을 측정하였다.

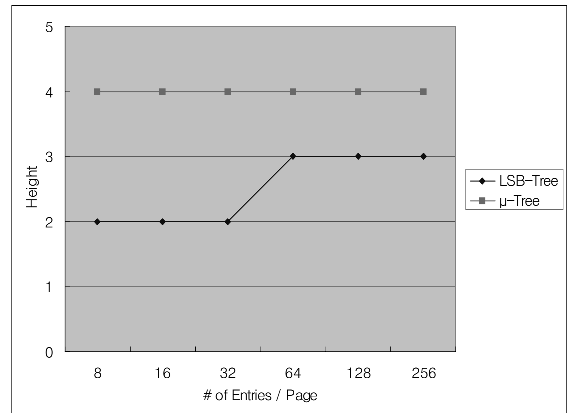
(그림 12)는 페이지 당 엔트리 수를 증가시키면서 순차적인 키 값을 삽입하여 높이 4일 때 삽입되는 키 값의 최대 개수를 구한 그래프이다. 그래프의 결과와 같이, 같은 높이에서 LSB-트리는 μ-트리에 비해 페이지 당 엔트리의 개수가 32개 일 때 약 518배 이상, 64개 일 때 약 755배 이상의 키 값을 삽입 할 수 있다. 이것은 두 트리의 높이가 같을 때, 같은 시간에 500배 이상의 키 값을 LSB-트리가 찾을 수 있다는 것을 의미한다.

(그림 13, 14, 15)는 (그림 12)에서 구한 μ-트리의 삽입된 순차적인 키 값의 최대 개수를 μ-트리와 LSB-트리에 삽입하여 각각의 트리의 높이와 사용된 페이지의 수를 구한 그래프이다. (그림 13)의 결과를 보면 μ-트리의 높이 4일 때, LSB-트리는 페이지 당 엔트리의 개수가 32개 이하일 때, 높이 2로 생성되고, 64개 이상은 높이 3으로 생성된다. 이 결과는 키 값을 검색할 때 μ-트리는 4번의 읽기 연산으로 검색 연산을 수행하지만, LSB-트리는 3번 이하의 읽기 연산으로 검색 연산을 수행하는 것을 나타낸다. (그림 14)와 (그림 15)에서는 LSB-트리가 같은 수의 순차적인 키 값을 삽입할 경우, μ-트리의 절반 이하의 페이지를 사용하는 것을 보여준다. 결과적으로 같은 수의 키 값을 순차적으로 삽입할 경우, LSB-트리는 μ-트리보다 트리의 높이가 낮기 때문에 검색 연산을 빠르게 수행할 수 있으며, 페이지의 효율성 또한 우수한 것을 보여준다.

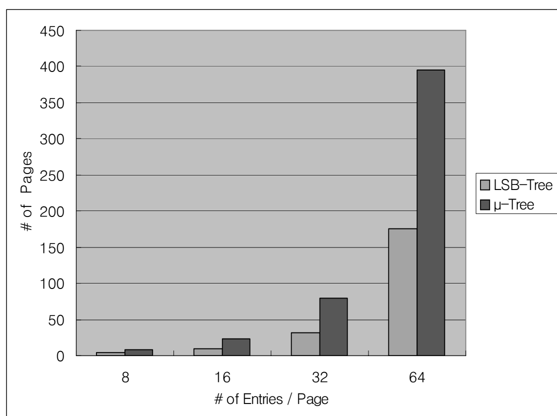
(그림 16)과 (그림 17)은 키 값의 랜덤 비율을 조절하여 각각의 트리에 삽입 연산을 수행하여 페이지 사용량과 트리 생성 시간을 비교한 그래프이다. 이 때 삽입된 키 값의 수는 24,000개이며, 0%는 순차적인 키 값을 삽입한 것이고, 100%로 갈수록 랜덤한 키 값을 삽입한 것이다. 그림 16의 결과를 보면, 키 값을 순차적으로 삽입할 경우 LSB-트리는 μ-트리의 페이지의 절반을 사용하여 트리를 생성하고, 랜덤한 키 값을 삽입할 경우 μ-트리의 약 80%의 페이지를 사용하여 트리를 생성한다. 따라서 LSB-트리는 전체적으로 더 적은 양의 페이지를 사용하여 트리를 생성하는 것을 알 수



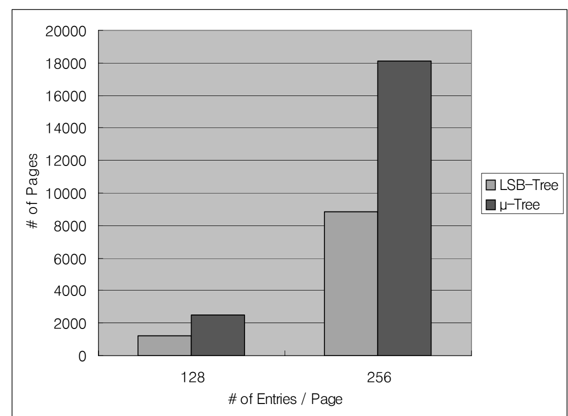
(그림 12) 삽입 가능한 키 값의 최대 개수



(그림 13) 삽입 연산에 대한 트리의 높이 변화



(그림 14) 삽입 연산에 대한 페이지 할당량-1

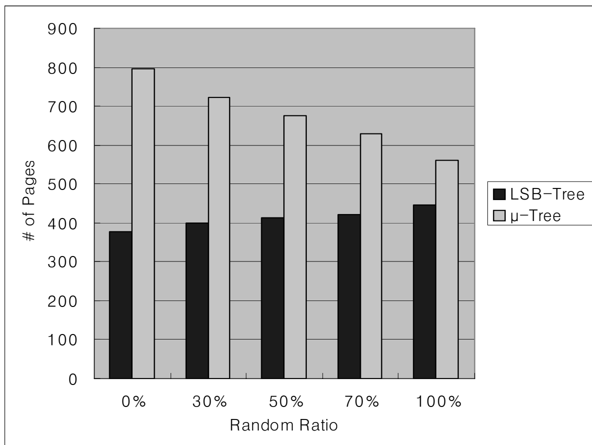


(그림 15) 삽입 연산에 대한 페이지 할당량-2

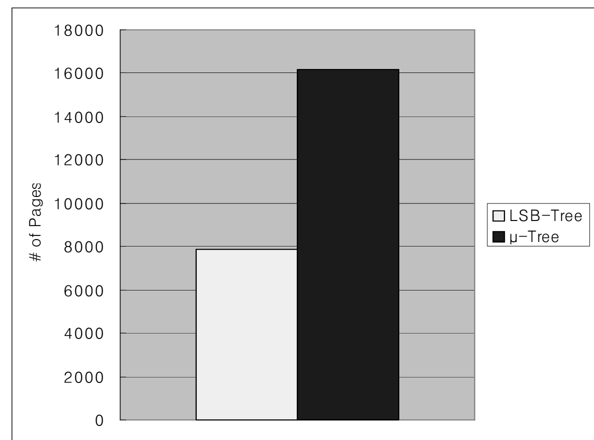
있다. 이와 함께 트리 생성 시간을 비교해보면 LSB-트리에 순차적인 키 값을 삽입하면 μ-트리에 비해 LSB-트리는 약 10% 정도 빠르게 생성되고, 랜덤한 키 값을 삽입하면 μ-트리에 비해 LSB-트리는 약 6% 정도 빠르게 생성된다. 이러한 결과를 분석해보면 LSB-트리는 순차적인 키 값이나 랜덤한 키 값의 삽입에서 페이지 사용 개수가 적고, 삽입 연산이 빠른 것을 보여준다. 높이 또한 임의 비율이 50% 이하일 때, LSB-트리는 높이 3, μ-트리는 높이 4로 이루어지며, 이후에는 두 트리 모두 높이 3으로 이루어진다. 트리 생성 높이에서 알 수 있듯이 LSB-트리가 μ-트리보다 더 낮기 때문에 검색 연산에서 읽기 연산이 적게 수행되어 더 빠르게 키 값을 검색할 수 있다.

(그림 18)과 (그림 19)은 순차적인 키 값 1,000,000개를 삽입하고, 플래시 메모리가 사용하는 최대의 블록 개수를 제한하여 실험한 그래프이다. (그림 18)의 결과는 LSB-트리에 1,000,000개의 순차적인 키 값이 삽입되면 μ-트리 페이지의 절반으로 트리를 생성하는 것을 보여준다. 그리고 그림 19의 결과는 LSB-트리가 μ-트리에 비해 삽입 연산이 빨리 수행되는 것을 보여준다. 그림을 보듯이 두 트리 모두 최대 블록 수가 작아질수록 트리가 생성되는데 소모되는 시간이 느려지는 것을 알 수 있다. 그 이유는 삽입 연산이 많아질수록 페이지를 빨리 소모되어 쓰레기 수집(garbage collection)을 많이 하게 되고, 이 쓰레기 수집은 쓰기·소거 연산을 많이 수행하여 시스템의 성능을 저하시키게 된다. μ-트리가 더 많은 페이지를 사용하기 때문에 블록의 최대 개수가 적을수록 쓰레기 수집을 많이 수행하여 더 느린 생성 시간을 보여준다. (그림 18)과 (그림 19)의 결과를 보듯이 최대 블록의 개수가 300개 일 때 약 38%, 1000개일 때 12% 정도 μ-트리보다 LSB-트리가 빠르게 삽입 연산을 수행하는 것을 알 수 있다.

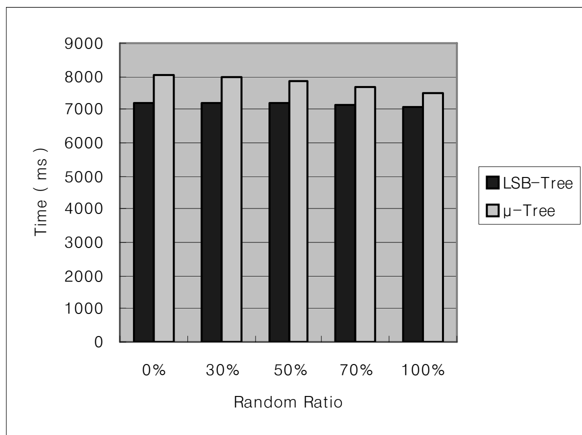
<표 3>은 (그림 16)과 (그림 17)의 실험에서 얻은 각 트리의 총 페이지의 할당량을 나타낸 것이다. 표 3에서 알 수 있듯이 LSB-트리가 μ-트리보다 입력된 키 값의 랜덤 비율에 상관없이 페이지를 적게 할당하는 것을 알 수 있으며, 100%의 랜덤 비율로 갈수록 페이지 할당량의 차는 줄어들지만 μ-트리에 할당한 561개보다 적은 447개를 할당한 것을 알 수 있다. 여기서 447개의 페이지 중 136개의 페이지를 로그 페이지로 할당하였으며, 이 136개의 페이지는 로그 사상 테이블에서 관리되는 것이다. 이 136개의 페이지를 관리



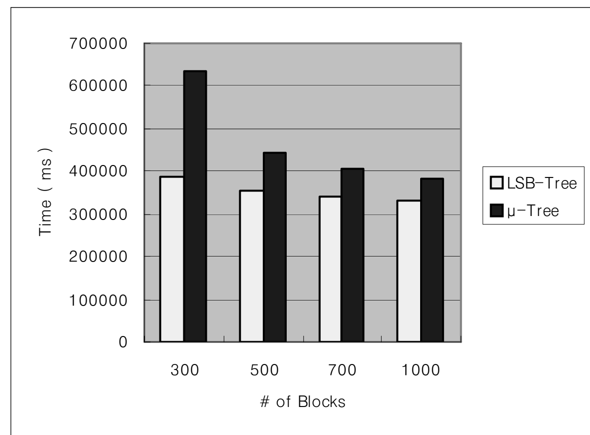
(그림 16) 랜덤 삽입 비율에 대한 페이지 할당량



(그림 18) 키 값 삽입에 대한 할당된 페이지 수



(그림 17) 랜덤 삽입 비율에 대한 트리 생성시간



(그림 19) 키 값 삽입에 대한 트리 생성 시간

〈표 3〉 랜덤 비율에 따른 할당 페이지 개수

랜덤 비율	Log Pages / Pages	
	LSB-Tree	μ-Tree
0%	1/378	795
30%	42/399	723
50%	69/414	677
70%	95/422	630
100%	136/447	561

하기 위해서 추가적인 메인 메모리와 연산 시간이 필요하다. 그러나 SRAM의 4byte를 랜덤하게 읽는데 소모되는 시간은 10ns으로 플래시 메모리 읽기 연산의 약 1/2000에 해당하는 시간이다[12]. 24,000개의 키 값을 정렬하여 삽입할 경우, 최대 1개의 로그 페이지가 필요하고 메인 메모리의 8byte에서 관리가 가능하며, 검색 연산 시 10ns의 추가적인 연산이 필요하다. 또한 100% 랜덤한 키 값을 삽입할 경우에도 100개 이상의 로그 페이지를 플래시 메모리에 할당하고, 메인 메모리 1Kbyte에서 관리가 가능하며, 100개의 키 값을 메인 메모리에서 검색하는 데 1μs가 추가적으로 필요하다. 이러한 추가적인 오버헤드(overhead)가 존재하지만, 검색 연산 시 추가적으로 소모되는 메모리 연산은 플래시 메모리의 쓰기 연산에 소모되는 200μs에 비해 매우 적은 양으로, LSB-트리보다 페이지를 많이 사용하여 발생하는 추가적인 쓰기 연산 보다는 적음을 알 수 있다. 또한 정렬된 키 값의 삽입에서는 매우 적은 양의 메인 메모리를 가지고 로그 페이지를 관리할 수 있고, 이때 발생하는 추가적인 메인 메모리의 연산은 무시할 수 있을 정도의 크기임을 알 수 있다.

6. 결론 및 향후 연구

본 논문은 NAND 플래시 메모리를 위한 새로운 색인 구조인 LSB-트리를 제안하였다. LSB-트리는 삽입·삭제·갱신 연산을 한 번의 쓰기 연산으로 처리하기 위해 로그 노드를 해당 단말 노드에 할당하여 변경되는 내용을 저장하였다. 또한 LSB-트리는 합병 연산을 수행하기 전까지 부모 노드의 변경을 최대한 늦추어 추가적인 쓰기 연산을 줄였고, 순차적인 키 값의 삽입이나 일정 노드에 대한 빈번한 갱신은 교환 연산을 수행하기 때문에 합병 연산에 발생하는 추가적인 쓰기 연산을 줄였다. 그밖에도 LSB-트리는 B-트리의 구조를 유지하고 있기 때문에 μ-트리보다 검색 성능이 뛰어남을 보였고, 페이지 전체를 단말 노드로 사용하기 때문에 μ-트리에 비해 공간 효율성이 우수함을 보였다. 마지막으로 다양한 실험을 통하여 관련 연구인 μ-트리보다 LSB-트리가 뛰어난 성능을 증명하였다.

향후 연구 과제는 전원 회복시 필요한 로그 사상 테이블의 저장 방법과 쓰기-전-소거 동작에서 발생하는 재복사(copy-back)의 쓰기 연산을 줄이는 로그 관리 기법에 대한 연구를 진행하는 것이다.

참 고 문 헌

[1] Samsung Electronics, "64M×8 Bits NAND Flash Memory (K9F1208X0C)," 2007.
[2] Samsung Electronics, "2G×8 Bit/4G×8 Bit/8G×8 Bit NAND Flash Memory (K9XXG08XXM)," 2007
[3] Artem B. Bityutskiy, "JFFS3 design issues," http://www.linux-mtd.infradead.org, 2005.
[4] Dongwon Kang et al., "μ-Tree: An Ordered Index Structure for NAND Flash Memory," Proceedings of the 7th Annual ACM Conference on Embedded Systems Software (ACM EMSOFT 2007), 2007.
[5] Patrick O'Neil, Edward Cheng, Dieter Gawlick, Elizabeth (Betty) O'Neil, "The Log-Structured Merge-Tree," Acta Informatica Vol.33, No.4, 1996.
[6] Douglas Comer, "The Ubiquitous B-Tree," ACM Computing Surveys. 11(2): 121-137, 1979.
[7] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, Ha-Joo Song, "A Log Buffer based Flash Translation Layer using Fully Associative Sector Translation," ACM Transactions on Embedded Computing Systems, Vol.6, No.1 Ariticle 5, 2007.
[8] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, Ha-Joo Song, "System Software for Flash Memory: A Survey.," International Conference on Embedded and Ubiquitous Computing, pp.394-404, 2006.
[9] Chin-Hsien Wu, Li-Pin Chang, Tei-Wei Kuo, "An Efficient B-Tree Layer for Flash-Memory Storage Systems," Real-Time and Embedded Computing Systems and Applications (RTCSA), pp.409-430, 2003.
[10] KIM, J. S., KIM, J. M.,NOH, S. H.,MIN, S. L., AND CHO, Y. K. 2002. A Space-Efficient Flash Translation Layer for Compactflash Systems. IEEE Transactions on Consumer Electronics 48, 366-375, 2006.
[11] M. Rosenblum and J. K. Ousterhout, "Thre Design and Implementation of A Log-Structured File System," ACM Trans. Computer Systems, 10(1):26-52, 1992.
[12] Henessy, J. L. and Patterson, D. A. "Computer Architecture: A Quantitative Approach 3rd ed.," Morgan Kaufmann, San Mateo, CA., 2003.

김 보 경

e-mail : bkhacker@hanyang.ac.kr
2007년 한양대학교 컴퓨터공학전공(학사)
2008년~현 재 한양대학교 컴퓨터공학과 석사과정
관심분야 : 데이터베이스, 플래시메모리용 시스템소프트웨어 등

주 영 도

e-mail : ydjoo@kangnam.ac.kr

1983년 한양대학교 전자통신공학과(학사)
1988년 미국 University of South Florida 컴퓨터공학과(석사)
1995년 미국 Florida State University 컴퓨터과학과(박사)
1984년~1985년 한국무역협회 전자계산소 시스템 프로그래머
1995년~2000년 KT 연구개발본부 연구팀/실장
2000년~2005년 시스코 시스템즈 코리아 통신사업부 담당 상무
2005년~2006년 화웨이 기술 코리아 부사장
2007년~현 재 강남대학교 컴퓨터미디어공학부 부교수
관심분야 : 데이터베이스, 지능형시스템, 초고속정보통신망, 통신망관리 등

이 동 호

e-mail : dhlee72@hanyang.ac.kr

1995년 홍익대학교 컴퓨터공학과(학사)
1997년 서울대학교 컴퓨터공학과(석사)
2001년 서울대학교 컴퓨터공학과(박사)
2001년~2004년 삼성전자 책임연구원
2004년~현 재 한양대학교 컴퓨터공학과 조교수
관심분야 : 데이터베이스, 멀티미디어 정보검색, 플래시메모리용 시스템소프트웨어 등