NAND 플래시 메모리를 위한 효율적인 로그 기반의 B-트리†

김보경*, 이현섭**, 이동호***
한양대학교 컴퓨터공학과
e-mail:{bkhacker*, hyunseob**, dhlee72***}@hanyang.ac.kr

An Efficient Log-based B-Tree for NAND Flash Memory

Bo-kyeong Kim*, Hyun-Seob Lee**, Dong-Ho Lee***
Dept of Computer Science and Engineering, Hanyang University

요 약

NAND 플래시 메모리는 하드 디스크에 비해 작고, 빠르며, 저 전력 소모 등과 같은 장점을 가지고 있어 대체 저장 매체로 주목받고 있다. 그러나 제자리 갱신이 불가능한 특징을 가지고 있어 B-트리를 사용하면 갱신이 빈번하게 발생하여 읽기 연산에 비해 상대적으로 느린 쓰기 연산과 소거 연산이 빈번해져 시스템의 성능이 저하 된다. 이러한 성능 저하를 피하기 위해 μ-트리가 제안되었으나, 고정된 페이지 레이아웃 구조를 가지고 있어 노드 분할과 트리 신장이 빈번하게 일어난다. 본 논문에서는 NAND 플래시 메모리 상에서 B-트리 구현 시 발생하는 추가적인 쓰기 연산의 횟수를 줄이기 위해 갱신이 일어나는 단말 노드에 로그 노드를 할당하여, 갱신되는 내용을 저장한다. 따라서 부모 노드의 내용이 변경 되는 것을 늦추어 추가적인 쓰기 연산을 줄이게 되며, 순차적인 키 값의 삽입이나 일정 노드에 대한 빈번한 갱신은 로그 노드가 단말 노드로 전환되어 추가적인 쓰기 연산을 줄이게 된다. 이러한 방법으로 추가적인 쓰기 연산을 줄임으로써 시스템의 성능을 향상시키는 NAND 플래시 메모리를 위한 새로운 B-트리 구조를 제안한다.

1. 서론

플래시 메모리는 비휘발성 저장 매체로 하드 디스크에 비해 작고, 가볍고, 빠르며, 전력 소모가 적고, 충격 및 온도에 대한 내구성이 우수하고, 소음이 없다. 이 같은 장점 때문에 휴대 전화, 개인용 단말기(PDA), 노트북 컴퓨터 등 이동 컴퓨팅 장비에 많이 사용되며, 최근에는 하드 디스크를 대체할 대용량 저장 매체로 주목 받고 있다. 플래시 메모리는 읽기 연산과 쓰기 연산을 하는 하드 디스크와 다르게 소거 연산이 추가로 필요하다. 읽기, 쓰기, 소거 연산 단위가 각각 다르며, 수행 시간도 다르다. 표 1은 각 연산 단위와 연산 수행 시간을 보여준다[1,2]. 페이지 단위로 읽기, 쓰기 연산을 하며 블록 단위로 소거 연산을 한다. 그리고 읽기 연산이 빠르고 쓰기 연산과 소거 연산이 느리며 제자리 갱신이 불가능하기 때문에 한 페이지에 데이터를 겹쳐 쓰기 위해서는 해당 페이지의 블록을 소거하고 데이터를 기록해야 한다. 이 같은 쓰기 전 소거는 시스템 성능을 저하시키는 원인이 된다[7].

B-트리는 삽입, 삭제, 갱신 연산 빈번하게 일어나도 구조 상 균형이 잘 잡혀 있어 빠른 검색 속도를 보장해 주어 하드 디스크에서 널리 사용되는 인덱스 구조이다[5,6]. 디스크 기반의 B-트리를 제자리 갱신이 불가능하고 쓰기, 소거 연산이 느린 플래시 메모리에 사용하게 되면, 갱신이 잦은 특징 때문에 소거 연산이 많이 일어나 시스템의 성능 저하를 유발하게 된다. 따라서 읽기 연산을 많이 하더라도 상대적으로 느린 쓰기 연산과 소거 연산을 줄이는 새로운 방식의 B-트리 설계가 필요하다.

동작 / 블록	읽기(page)	쓰기(page)	소거(block)
소 블록 (64M x 8Bits)	15μs	200μs	2ms
대 블록 (2G,4G,8G x 8Bit)	25μs	200μs	1.5ms

<표 1> 플래시 메모리 동작 속도[5,6]

NAND 플래시 메모리를 위한 인덱스 구조인 μ-트리는 불필요한 쓰기 연산을 줄여 시스템의 전체적인 성능을 개선하였다[4]. 그러나 비효율적인 페이지 관리로 인해 노드 분할과 트리 신장이 빈번하다는 문제점을 갖고 있다. 이러한 단점을 극복하기 위해 본 논문에서는 B-트리의 단말 노드에 로그 노드를 두고 갱신되는 단말 노드를 로그 노드에 한 번의 쓰기 연산으로 저장하여 부모 노드의 변화를 늦추는 로그 기반의 B-트리(LSB-Tree:Log-Structured B-Tree)를 제안한다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구인

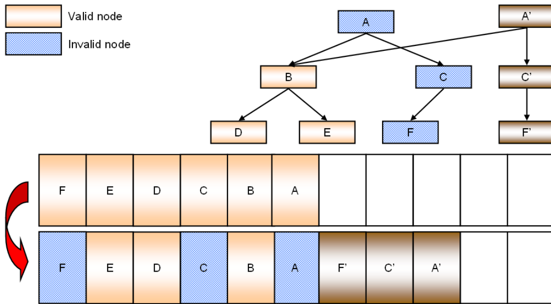
† 본 연구는 정보통신부 및 정보통신연구진흥원의 IT신성장동력 핵심기술개발사업[2006-S-040-01, Flash Memory 기반 임베디드 멀티미디어 소프트웨어 기술 개발]과 과학기술부 및 대구경북과학기술연구원의 연구개발사업의 일환으로 수행하였음.

NAND 플래시 메모리를 위한 B-트리 구조에 대해 알아보고 문제점을 살펴본다. 3장에서는 본 논문에서 제안하는 인덱스 구조 LSB-트리를 기술하고, 4장에서는 관련 연구인 μ-트리와 비교해 본다. 5장에서는 결론과 함께 향후 연구 방향에 대해 기술한다.

2. 관련 연구

2.1 Wandering 트리

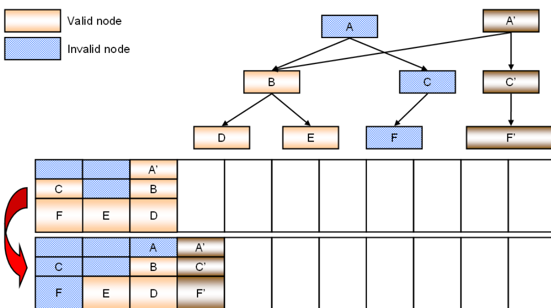
Wandering 트리[3]는 플래시 메모리 전용 파일 시스템 JFFS3의 인덱스 구조로, B-트리 형태를 가진다. 갱신 연산을 수행할 때, 플래시 메모리에 변경되는 노드만을 저장한다. 그림 1은 wandering 트리가 갱신 연산을 수행할 때, 플래시 메모리에 저장되는 과정을 보여준다. 만약 노드 F에 갱신 연산이 발생하여 부모 노드인 C, A 노드에 변경이 발생하면, 플래시 메모리의 각 페이지마다 변경되는 노드 F, C, A를 또 다른 페이지(즉, F′, C′, A′ 노드)에 각각 저장한다. 따라서 데이터 삽입시 변경이 발생하는 모든 노드에 대한 쓰기 연산이 불가피하다는 단점이 존재한다.



(그림 1) Wandering 트리 예제

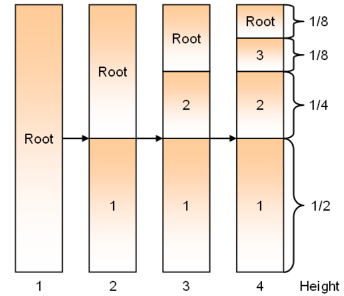
2.2 μ-트리

μ-트리[4]는 wandering 트리의 단점을 해결하기 위해, 변경되는 모든 노드를 하나의 페이지에 저장한다. 그림 2는 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 저장한다.



(그림 2) μ-트리 예제

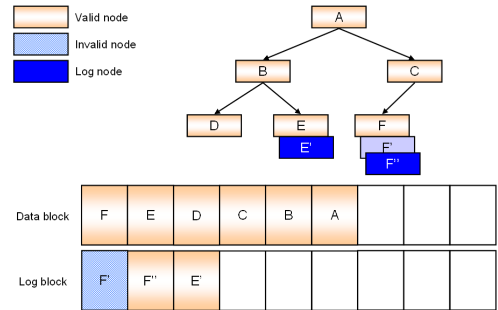
μ-트리는 한 페이지에 변경되는 모든 노드를 저장하기 위해서 그림 3과 같이 고정된 페이지 레이아웃 구조를 사용한다. 트리의 높이가 2이상 일 때, 모든 단말 노드는 페이지의 반을 사용하며, 트리가 신장할 때마다 루트 노드의 엔트리의 개수는 반으로 줄어든다. 따라서 μ-트리의 단말 노드는 B-트리에 비해 빨리 차기 때문에 노드 분할이 잦아지며, 부모 노드 또한 빨리 차게 되어 트리의 높이가 빨리 커진다. 노드 분할과 트리의 신장이 빈번해지면 추가적인 쓰기 연산이 발생하여 시스템의 성능을 저하시킨다.



(그림 3) μ-트리의 페이지 레이아웃 구조

3. LSB-트리(Log-Structured B-Tree)

3.1 주요 아이디어



(그림 4) LSB-트리 예제

μ-트리의 비효율적인 페이지 관리로 인한 잦은 노드 분할과 트리 신장을 해결하기 위하여 본 논문에서는 로그 기반의 B-트리(LSB-트리)를 제안한다. LSB-트리는 단말 노드에 로그 노드를 두어 갱신되는 단말 노드를 로그 노드에 한 번의 쓰기 연산으로 저장한다. 만약 로그 노드가 가득 차게 되면, 단말 노드와 부모 노드의 갱신 연산을 수행한다. 따라서 단말 노드의 변화로 인한 부모 노드 및 상위 노드의 변화를 늦춤으로써, 불필요한 쓰기 연산 횟수를 줄이게 된다. 그림 4와 같이 단말 노드 E, F가 갱신되면 단말 노드와 함께 변경되는 부모 노드를 갱신하는 것이 아니라, 변경되는 단말 노드에 대해서 로그 노드를 동적으로 할당하여 저장한다. 그리고 로그 노드 F가 가득 차게 되면, 해당 단말 노드 F와 변경되는 부모 노드 C, A를 데이터 블록에 새로운 페이지를 할당하여 각각 저장한다.

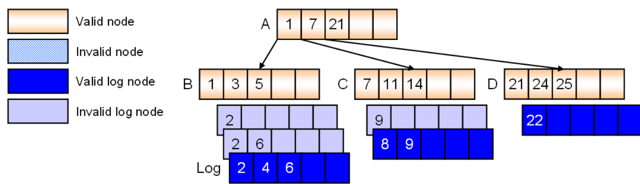
본 논문에서 제안하는 LSB-트리는 NAND 플래시 메모리상에서 B-트리를 구현할 때, 단말 노드의 갱신으로 인한 부모 노드 및 상위 노드의 변화를 즉시 반영하는 것이 아니라, 갱신되는 단말 노드에 해당하는 로그 노드를 두어

변경되는 정보를 저장한다. 그리고 만약 로그 노드가 가득 차게 되면, 해당 단말 노드와 변경되는 부모 노드에 대해서 새로운 페이지를 할당하여 저장한다. LSB-트리는 단말 노드의 변경으로 인한 부모 노드의 변화를 가능한 늦춤으로써 불필요한 쓰기 및 소거 연산의 횟수를 효과적으로 줄일 수 있다.

3.2 검색 연산

검색 연산은 부모 노드 엔트리의 키 값을 검색하여, 해당 단말 노드를 찾는다. 단말 노드의 키 값을 검색하기 전에 로그 노드를 먼저 검색한다. 만약 키 값이 존재하지 않으면 해당 단말 노드를 검색하고 키 값의 존재 유무를 알린다. 로그 노드를 먼저 검색하는 이유는 로그 노드의 유효한 데이터가 존재할 가능성이 높기 때문이다.

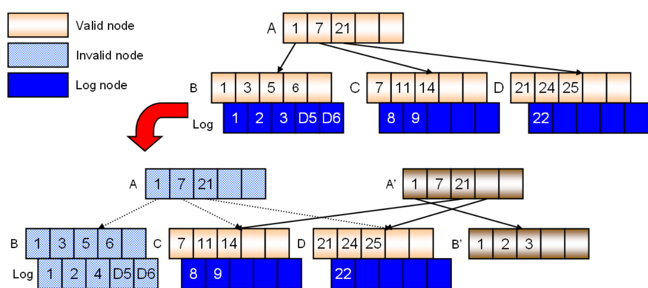
3.3 삽입 연산



(그림 5) LSB-트리 삽입 연산 예제

삽입 연산은 삽입할 키 값이 들어갈 단말 노드를 검색 연산을 통하여 찾는다. 검색된 단말 노드에 해당하는 로그 노드가 존재하지 않을 경우, 로그 영역에서 새로운 로그 노드를 하나 할당하여 키 값을 삽입한다. 그러나 로그 노드가 존재한다면, 로그 노드에 키 값을 삽입한다. 만약, 엔트리가 가득 차 더 이상 삽입 할 수 없다면 합병 연산을 수행한다. 그림 5는 삽입 연산 과정을 보여준다. 키 값 '2, 6, 9, 8, 22, 4' 순서로 삽입 연산을 하면 로그 노드에 6번의 쓰기 연산을 한다. 대부분의 삽입 연산은 로그 노드가 가득차기 전까지는 쓰기 연산 1회로 처리가 가능하다.

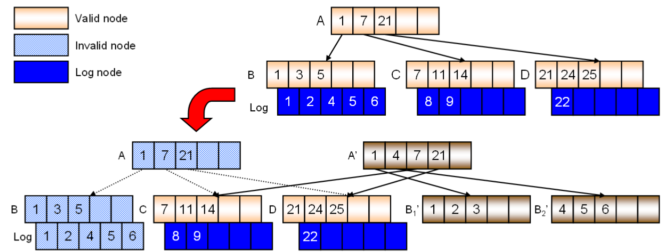
3.4 삭제 연산



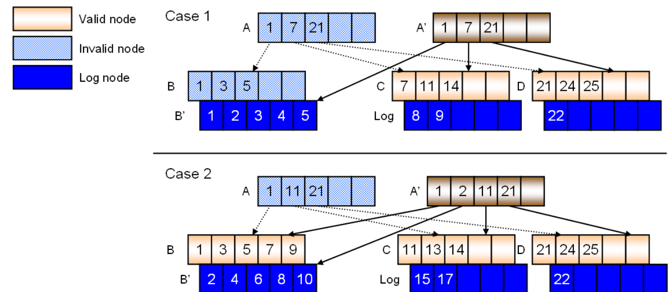
(그림 6) LSB-트리 삭제 연산 예제

삭제 연산은 검색 연산을 수행하여 해당 로그 노드를 찾아 해당 키 값을 제거한다. 이때, 해당 키 값이 로그 노드에 없고 단말 노드에만 존재하는 경우, 삭제 엔트리를 로그 노드에 삽입하여 나중에 합병 연산 수행 시 단말 노드에서 제거한다. 그림 6은 삭제 연산 과정 및 합병 과정을 간략하게 보여준다. 만약 그림 6에서 노드 B의 키 값 '5, 6'이 삭제된다면, 로그 노드에 삭제 엔트리 5, 6을 삽입한다. 그리고 로그 노드가 가득 차게 되어 합병 연산이 수행된다면, 해당 단말 노드의 키 값 '5, 6'을 제거하여 새로운 노드 B′로 저장한다.

3.5 합병 연산



(그림 7) LSB-트리 합병 연산 예제



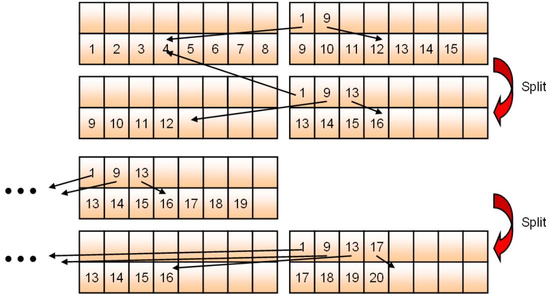
(그림 8) LSB-트리 전환 연산 예제

삽입 연산을 수행할 때 로그 노드가 가득 차게 되면 합병 연산을 수행한다. 그림 7은 합병 연산 과정을 보여준다. 키 값 '1, 2, 4, 5, 6'이 삽입되어 로그 노드 B가 가득 차게 되면, 단말 노드 B와 로그 노드 B를 합병하게 되는데, 합병되는 노드의 키 값 '1, 2, 3, 4, 5, 6'이 단말 노드의 엔트리 개수 보다 많기 때문에 노드가 2개로 분할된다. 따라서 분할되는 노드 B_1'(1, 2, 3)와 B_2'(4, 5, 6)를 변경되는 부모 노드 A′와 함께 새로운 페이지에 저장한다. 합병 연산은 '트리 높이+1'만큼의 쓰기 연산이 필요하다.

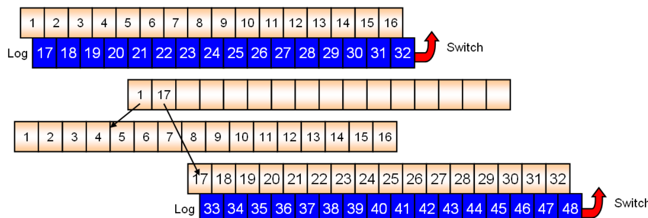
그림 8은 합병 연산 과정에서 발생할 수 있는 전환 연산 과정에 대해 보여준다. 그림 8의 사례 1과 같이 로그 노드의 키 값이 단말 노드의 키 값을 모두 포함하거나, 사례 2와 같이 로그 노드의 키 값이 단말 노드의 키 값과 모두 일치하지 않는 경우, 로그 노드를 단말 노드와 합병하는 것이 아니라, 로그 노드를 단말 노드로 전환한다. 사례 1처럼 노드 B의 키 값이 1, 3, 5이고 로그 노드 B의 키 값이 1, 2, 3, 4, 5 이면 합병 연산시 로그 노드 B를 단말 노드 B′로 전환하고, 변경되는 부모 노드 A′를 플래시 메모리의 새로운 페이지에 저장한다. 또한 사례 2처럼 노드 B의 키 값이 1, 3, 5, 7, 9이고 로그 노드 B의 키 값이 2, 4, 6, 8, 10이면 로그 노드 B를 단말 노드 B′로 전환하

고, 변경되는 부모 노드 A′를 새로 저장한다. 이러한 전환 연산은 '트리 높이-1'만큼의 쓰기 연산이 필요하기 때문에 합병 연산('트리 높이+1'의 쓰기 연산)보다 2회 적은 쓰기 연산을 한다.

4. μ-트리와의 비교



(그림 9) μ-트리 순차적인 키 삽입 예제



(그림 10) LSB-트리 순차적인 키 삽입 예제

그림 9는 페이지 엔트리가 16인 μ-트리에 순차적인 키 값을 삽입 할 때의 과정으로 단말 노드 엔트리 수의 반인 4개의 키 값이 삽입될 때 마다 노드가 분할이 되어 추가적인 쓰기 연산이 수행되는 것을 보여준다. 그리고 그림 10은 페이지 엔트리가 16인 LSB-트리에 순차적인 키 값을 삽입 할 때의 과정으로 로그 노드에 16개의 키 값이 삽입될 때 마다 로그 노드를 단말 노드로 전환하는 것을 보여준다.

그림 9, 10과 같이 페이지의 엔트리가 16이고, 트리의 높이가 4일 때, μ-트리의 최대 엔트리 수는 8*4*2*2 = 128개이며, LSB-트리의 최대 엔트리 수는 16^4 = 65536개이다. 이처럼 LSB-트리는 같은 높이에서 512배의 엔트리를 다룰 수 있으며, 128개의 엔트리는 높이 2(μ-트리의 반)로 표현 가능하다. 따라서 검색 연산이 μ-트리에 비해 약 2배 정도 우수한 것을 알 수 있다. 만약 128개의 순차적인 키 값을 각각의 트리에 삽입하게 되면, μ-트리는 128 + (128/4) + α ≑ 160번 이상의 쓰기 연산을 하고, LSB-트리는 128 + 1*(128/16) = 136번의 쓰기 연산으로 가능한 것을 알 수 있다(α는 트리 신장에 따른 추가 연산). 이와 같이 μ-트리보다 쓰기 연산이 적게 일어나므로, 쓰기 연산 역시 우수하다는 것을 알 수 있다.

만약 페이지의 엔트리가 32이고, 트리의 높이가 4이면, μ-트리의 최대 엔트리 수는 2048개, LSB-트리의 최대 엔트리 수는 1048576개이다. 엔트리가 16일 때와 마찬가지로 같은 높이에서 512배의 엔트리를 다룰 수 있으며, 높이 3의 LSB-트리로 높이 4의 μ-트리를 다룰 수 있어 검색 성능에서 우수함을 알 수 있다. 만약 순차적인 키 값 2048개를 삽입한다면, μ-트리는 2048 + 256 + α ≑ 2304 이상의 쓰기 연산을 하고, LSB-트리는 2048 + (1024/32) + 2*(1024/32) = 2144의 쓰기 연산을 한다.

두 가지 예제를 살펴보았듯이, 같은 수의 키 값을 각각의 트리에 삽입하고 검색한다면, μ-트리 높이보다 LSB-트리 높이가 낮기 때문에 빠른 검색을 할 수 있다. 또한 각각의 트리가 같은 높이를 가지고 있다면, LSB-트리의 공간 활용도가 높기 때문에 더 많은 엔트리를 사용할 수 있다.

5. 결론 및 향후 연구

본 논문에서 제안한 LSB-트리는 변경되는 단말 노드의 내용을 로그 영역에 할당한 로그 노드에 저장함으로써 단말 노드의 변경을 한 번의 쓰기 연산으로 처리한다. 그리고 합병 연산을 수행하기 전까지 부모 노드의 변경을 최대한 늦춤으로써, 추가적인 쓰기 연산을 줄인다. 또한 순차적인 키 값의 삽입이나 일정 노드에 대한 잦은 갱신은 합병 연산 대신 전환 연산을 수행하기 때문에 합병 연산에 발생하는 추가적인 쓰기 연산을 줄인다.

향후 연구 과제는 소거 동작을 줄이기 위한 로그 영역 관리 방법과 효율적인 쓰레기 수집(garbage collection)에 대한 정책을 수립하는 것이다. 그리고 LSB-트리를 플래시 메모리상에서 구현하여 실험하고 μ-트리의 성능과 비교 평가해 보는 것이다.

참고문헌

[1] Samsung Electronics, "64M x 8 Bits NAND Flash Memory (K9F1208X0C)", 2007

[2] Samsung Electronics, "2G x 8 Bit / 4G x 8 Bit / 8G x 8 Bit NAND Flash Memory (K9XXG08XXM)", 2007

[3] Artem B. Bityutskiy, "JFFS3 design issues", http://www.linux-mtd.infradead.org/

[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 (EMSOFT 2007)

[5] Patrick O'Neil et al., "The Log-Structured Merge Tree", Acta Informatica,33:351-385, 1996

[6] D. Comer., "Ubiquitous B-Tree", ACM Computing Surveys, 11(식 2):121-137, 1979

[7] Sang-Won Lee et al., "A Log Buffer based Flash Translation Layer using Fully Associative Sector Translation", ACM Transactions on Embedded Computing Systems, Vol. 6, No.1, Article 5, 2007