Journal of The Korea Society of Computer and Information
Vol. 20, No. 4, April 2015 www.ksci.re.kr
http://dx.doi.org/10.9708/jksci.2015.20.4.001

CL 트리: 낸드 플래시 시스템에서 캐시 색인 리스트를 활용하는 B+ 트리

황 상 호*, 곽 종 욱*

CL-Tree: B+ tree for NAND Flash Memory using Cache Index List

Sang-Ho Hwang*, Jong Wook Kwak*

요 약

낸드 플래시는 기존의 하드디스크와 다르게 지움 연산이 필요하고 제자리 갱신이 불가능한 특성을 가지고 있어 플래시 전환 계층(FTL: Flash Translation Layer)을 사용한다. 하지만 플래시 전환 계층을 이용하는 방법은 사상 테이블의 사용에 따른 메모리 소비량이 많은 단점이 있어서 최근에는 사상 테이블을 사용하지 않는 색인 구조에 대한 연구가 많이 이루어지고 있다. 하지만 이러한 연구들은 사상 테이블을 사용하지 않는 시스템에서 발생되고 있는 업데이트 파생문제를 해결하여야 한다.

논문에서는 이러한 업데이트 파생문제를 효과적으로 해결하고자 CL-트리(Cache List Tree)라 명명된 새로운 색인 구조를 제안한다. 제안하는 기법은 메모리상에 쓰기 연산이 이루어진 노드들의 주소를 다중 리스트로 이루어진 CL-트리에 저장함으로써, 추가적인 쓰기 연산을 줄일 뿐만 아니라 자주 접근되는 노드에 대하여 빠르게 접근할 수 있기 때문에 탐색 측면에서도 뛰어난 성능을 보인다. 성능평가 결과 제안하는 CL-트리 구조는 작업 수행 속도에서 기존의 B+ 트리와 주요 관련 연구에 비해 삽입 속도는 최대 173%, 탐색 속도는 179% 향상되었음을 보였다.

▸ Keywords : 낸드 플래시 메모리, B+ 트리, 색인 구조, 업데이트 파생, 다중 리스트 구조

Abstract

NAND flash systems require deletion operation and do not support in-place update, so the storage systems should use Flash Translation Layer (FTL). However, there are a lot of memory consumptions using mapping table in the FTL, so recently, many studies have been proposed to resolve mapping table overhead. These studies try to solve update propagation problem in the nand flash system which does not use mapping table.

•제1저자 : 황상호 •교신저자 : 곽종욱
•투고일 : 2015. 2. 5, 심사일 : 2015. 2. 26, 게재확정일 : 2015. 3. 16.
* 영남대학교 컴퓨터공학과(Department of Computer Engineering, Yeungnam University)
※ 이 논문은 2014년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임.
(No. NRF-2014R1A1A2057146)

In this paper, we present a novel index structure, called CL-Tree(Cache List Tree), to solve the update propagation problem. The proposed index structure reduces write operations which occur for an update propagation, and it has a good performance for search operation because it uses multi-list structure. In experimental evaluation, we show that our scheme yields about 173% and 179% improvement in insertion speed and search speed, respectively, compared to traditional B+tree and other works.

▸ Keywords : NAND Flash Memory, B+ tree, index structure, update propagation, multi-list structure

I. 서 론

낸드 플래시 시스템은 비휘발성, 빠른 접근 속도, 저전력, 내충격성, 작은 크기 및 적은 중량 등의 장점으로 인해 그 사용범위가 넓어지고 있는 추세에 있다. 이러한 낸드 플래시 메모리는 하드디스크와 다른 몇 가지 특징이 있다. 읽기 속도가 25㎲ ~ 60㎲이고, 쓰기 속도는 250㎲ ~ 900㎲로 동작 속도가 서로 다르다. 또한 읽기 및 쓰기 연산 외에도 지움 연산이 있으며, 읽기 및 쓰기 연산이 페이지 단위로 동작하는 것에 반해 지움 연산은 블록 단위로 동작을 한다. 지움 속도는 약 3.5ms로 다른 연산에 비해 상대적으로 느리다. 또한 블록당 최대 소거 연산수는 약 10,000 ~ 100,000회이고, 소거 연산을 하지 않으면 데이터의 제자리 갱신이 불가능하다[1].

지움 연산이 필요하고 제자리 갱신이 불가능한 특성으로 인해 낸드 플래시를 사용하는 시스템은 별도의 플래시 전환 계층(FTL: Flash Translation Layer)을 사용한다. 저장 시스템의 전반적인 성능은 플래시 전환 계층에 의해 결정되므로 이에 대한 많은 연구가 지금까지 이루어지고 있다[2,3]. 하지만 이러한 시스템은 플래시 전환 계층에 포함되어 있는 사상 테이블(Mapping Table)의 메모리 소모량이 많은 단점이 있다.

최근에는 플래시 전환 계층을 적용하기 힘든 시스템에서의 색인 구조에 대한 연구가 많이 이루어지고 있다[4-6,12-15]. 하지만 사상 테이블이 없는 시스템에서의 색인 구조들은 제자리 갱신이 불가능한 낸드 플래시의 특성으로 인하여 업데이트 파생 문제가 발생하여 추가적으로 쓰기 연산을 수행해야한다. 본 논문에서는 이러한 문제를 효과적으로 해결하기 위해서 CL-트리(CL-Tree: Cache List Tree)라 명명된 새로운 색인 구조를 제안한다. 제안하는 색인 구조는 기존의 트리와는 별도로 새롭게 갱신이 일어나는 노드들에 대한 새로운 링크를 제공하기 위해 메모리에 위치하는 다중 리스트를 이용하고 있다. 이 다중 리스트 구조를 통해 업데이트 파생문제를 해결하며, 또한 최근 업데이트가 이루어진 노드에 빠르게 접근할 수 있어 삽입 속도 및 탐색 속도가 기존의 색인구조에 비해 향상되었다.

이하 논문의 구성은 다음과 같다. 2장에서는 낸드 플래시와 B+트리 그리고 기존 연구에 대하여 기술한다. 3장에서는 본 논문에서 제안하는 트리에 대하여 설명하고, 4장에서는 실험결과를, 5장에서는 결론 및 향후 연구에 대하여 기술한다.

II. 관련 연구

기존의 하드 디스크 기반에서 만들어진 색인 구조를 낸드 플래시에 적용하게 되면 낸드 플래시 고유의 특성으로 인하여 메모리 수명이 단축되는 등의 성능저하가 발생한다. 이를 해결하기 위해 최근에는 낸드 플래시 전용 색인 구조에 대한 연

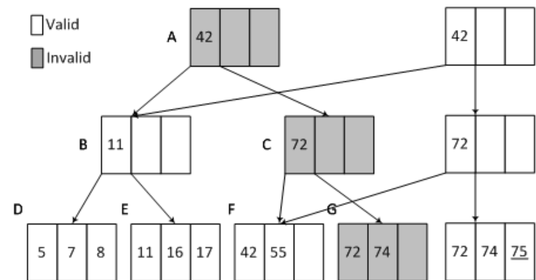


그림 1. 업데이트 파생문제
Fig. 1. Update propagation

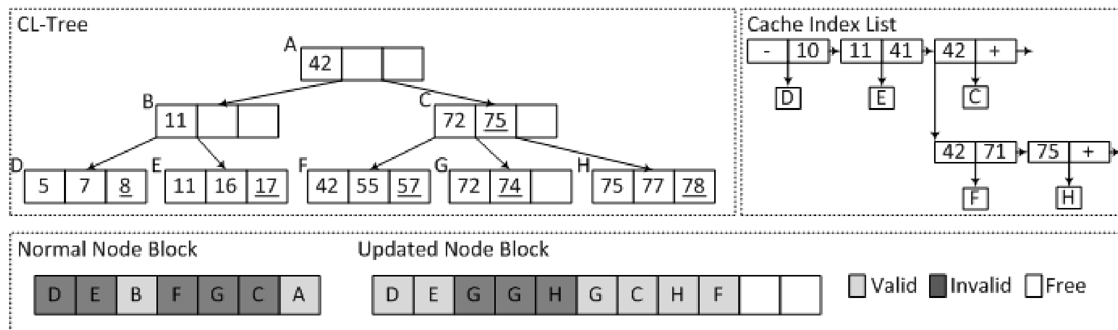


그림 2. CL-트리
Fig. 2. CL-Tree

구가 이루어져왔다[7-10,16,17]. 이러한 기존의 연구들은 플래시 전환 계층을 사용하고 대부분 메모리 버퍼를 이용하여 성능을 향상시켰다. 하지만 플래시 전환 계층을 이용하는 방법은 사상 테이블 의 사용에 따른 메모리 소비량이 많아서, 32GB를 기준으로 페이지 사상(page mapping)은 32MB, 블록 사상(block mapping)은 256KB의 메모리가 필요하다[12]. 따라서 최근에는 메모리에 여유가 없는 임베디드 시스템을 위해, 사상 테이블을 사용하지 않는 낸드 플래시 전용 색인 구조에 대한 연구가 활발히 이루지고 있다[4-6,12-15].

사상 테이블이 없는 시스템에서는 트리의 노드가 갱신되면 갱신이 이루어진 노드만 업데이트 되는 것이 아니라, 해당 노드를 참조하고 있는 조상 노드 및 형제 노드들까지 업데이트가 이루어져야한다. 예를 들어 그림 1과 같이 새로운 키 75가 삽입되면 G노드는 갱신이 되어 G'로 저장이 된다. 하지만 부모 노드인 C노드는 G'노드가 아닌 기존의 G노드를 참조하고 있어 트리가 유지되지 않는다. 따라서 이런 문제를 해결하기 위해B+트리를 색인 구조로 사용하는 JFFS3의 경우에는, 갱신이 발생하는 경우 단말 노드에서 루트 노드까지 갱신을 하는 배회 트리(Wandering Tree) 기법을 이용하여 색인 구조를 유지하고 있다[11]. 이렇게 구성하게 되면 색인 정보를 낸드 플래시 내에 저장할 수 있는 장점이 있지만, 읽기 연산에 비해 상대적으로 느린 쓰기 연산이 빈번하게 이루진다는 단점이 있다. 이러한 문제를 해결하기 위해 많은 연구들이 진행되어 왔으며 대표적인 연구는 다음과 같은 것들이 있다.

μ-트리[4]는 앞서 설명한 배회 트리의 단말 노드에서 루트 노드까지 이루어지는 쓰기 연산을 줄이기 위해 제안되었다. 그림 1과 같이 G노드의 갱신이 이루어지면 루트 노드까지 한 페이지에 저장함으로써 한 번의 쓰기 연산만 발생시킨다. 비록 이 기법이 쓰기 연산에 대하여 장점이 있지만, 한 페이지에 여러 레벨의 노드들을 한꺼번에 저장하기 때문에 한 레벨에서 사용할 수 있는 공간이 크지 않아 트리의 높이가 빠르게 증가하고 페이지 내의 공간을 효율적으로 사용하지 못해 페이지 사용량이 많은 단점이 있다.

AD 트리[5]는 같은 부모를 가지는 형제 노드들을 1개 이상의 페이지에 같이 저장하여 노드 분할시 발생되는 오버헤드를 줄이고 있다. 또한 기존의 노드들은 CNB(Cold Node Buffer)에 저장하고 새로 업데이트되는 노드들을 HNB(Hot Node Buffer)에 저장하는 방법으로 업데이트 파생 문제를 해결하였다. 비록 이 기법이 쓰기 연산에서 발생되는 업데이트 파생문제를 해결하였지만 새로 갱신된 노드의 경우 페이지를 여러 번 읽게 되는 단점이 있다.

LUB 트리[6]는 삽입 연산 시 업데이트 파생이 발생하는 단말 노드에서 루트 노드까지의 모든 노드들을 메모리에 저장하여 업데이트를 지연시키는 방법으로 쓰기 성능을 향상시키고 있다. 이 기법은 최근 갱신된 노드의 경우 메모리에 노드가 저장되어 있기 때문에 순차 삽입에서 좋은 성능을 나타내지만 무작위 삽입에서 여전히 쓰기 횟수가 많다.

LA-트리[9]는 트리의 각 레벨에 적응형 버퍼공간을 두고 있으며 키가 삽입되면 상위 버퍼에서 하위 버퍼로 순차적으로 이동하여 최종적으로 단말 노드에 저장되는 방법을 사용한다. 이렇게 업데이트를 지연시키는 방법으로써 읽기 및 쓰기 오버헤드를 줄이고 있지만 임베디드 시스템에 적용할 때에는 FTL 역할을 하는 노드 테이블(Node Table)을 메모리에 추가해야 한다.

FD-트리[10]는 삽입이 이루어지는 헤드 트리(Head Tree)와 합병으로 인하여 부분적 순차 쓰기가 이루어지는 하단 부분으로 이루어져있다. 이 기법은 무작위 쓰기보다 순차 쓰기가 빠른 낸드 플래시 메모리 특성 때문에 속도에서 좋은 성능을 보이지만, FTL을 기반으로 하고 있어 임베디드 시스템에 적용하기 위해서는 추가적인 사상 테이블이 필요하다.

LU-트리[16]는 메모리에 위치하는 LUP(Lazy Update Pool)에 삽입 및 삭제가 이루어지는 키를 로그형태로 저장한

다. 이 기법은 LUP가 가득차면 그때 희생자 선택 정책에 의해 가장 효율성이 높은 그룹을 선택하여 합병을 수행한다. 업데이트를 지연하기 위해 메모리 버퍼를 사용하는 LU-트리는 쓰기 성능을 향상시켰지만 갑작스러운 전원 차단 시 데이터 소실의 위험이 있으며 FD-트리와 같이 FTL을 기반으로 하고 있어 임베디드 시스템에 적용하기 힘든 단점이 있다.

표 1은 언급한 기법들의 평가 항목을 나타내고 있다. 표1에서 n은 1개 페이지에 들어갈 수 있는 엔트리를 나타내고, h는 트리의 높이를 의미하며, l은 선택된 노드의 높이, m은 페이지의 크기, p는 페이지의 수를 의미한다. 평가 항목에서 캐시는 메인 메모리에 위치하고 있는 저장 공간을 의미한다. 캐시를 사용하는 대부분의 기법들은 페이지 또는 키를 저장하기 위한 용도로 캐시를 활용하고 있으나 제안하는 기법은 업데이트된 주소를 저장하는 용도로 사용되고 있다.

이와 같은 주요 관련 연구의 공통된 문제를 해결하기 위해 본 논문에서는 업데이트가 이루어지는 노드에 대한 새로운 링크를 제공하기 위해 메모리에 위치하는 다중 리스트를 이용한다. 이 리스트 구조를 통해 업데이트 파생문제를 해결하며, 또한 제안하는 색인 구조는 최근 업데이트가 이루어진 노드에 빠르게 접근할 수 있어 삽입 속도 및 탐색 속도가 기존의 색인 구조에 비해 우수한 특성을 가지고 있다.

표 1 각 트리들의 평가 항목
Table 1. The evaluation list of each trees

기법	노드 엔트리 수	캐시	FTL
B+트리	n	×	-
μ-트리	n if $h=1$ $n/2^{l-1}$ if $h=l$ $n/2^{l}$ $otherwise$	×	×
AD-트리	$\sqrt{n}\times p$	×	×
LUB-트리	n	○	×
LA-트리	n	○	○
FD-트리	n	×	○
LU-트리	n	○	○
CL-트리	$n-2$	○	×

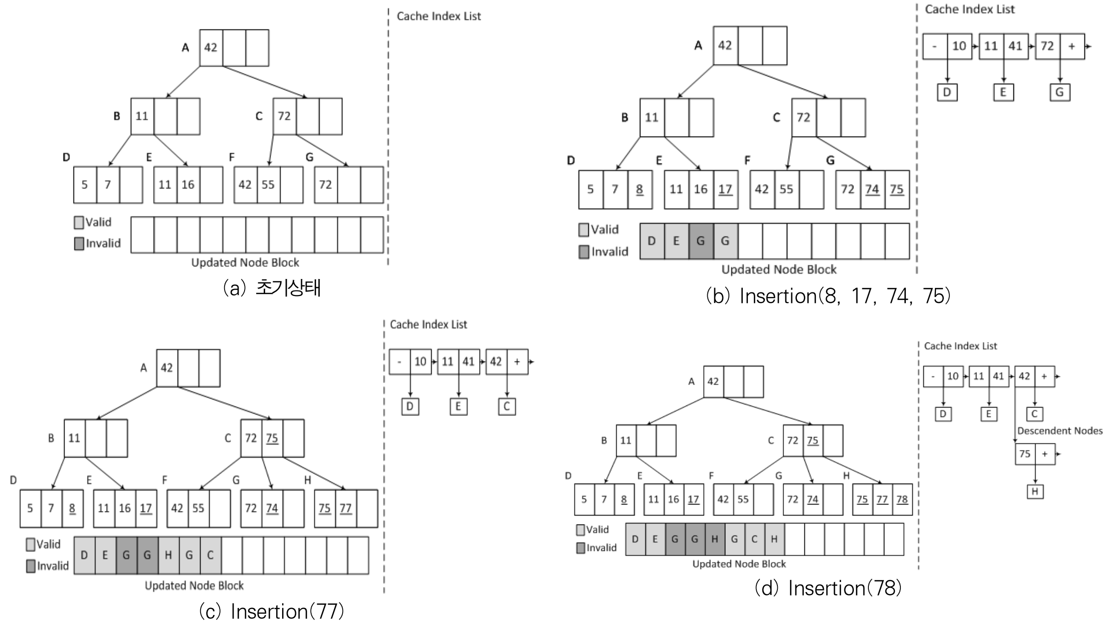
III. 다중 리스트 구조를 활용한 CL-트리

1. 주요 구성

본 논문은 메모리를 필연적으로 사용하는 사상 테이블을 배재한 형태의 CL-트리를 제안한다. 그림 2는 제안하는 CL-트리의 주요 구성을 보여주고 있다. 기본적으로 CL-트리의 노드들은 낸드 플래시 내에 저장되며, 갱신된 노드의 접근을 위해 메모리상에 CIL(Cache Index List)가 있다. 이 CIL은 노드의 주소, 노드가 가질 수 있는 값의 범위를 저장하고 있고, 같은 레벨로 연결하는 링크와 낮은 레벨로 연결하는 링크를 포함하고 있다. 같은 레벨로 연결하는 링크는 이 노드의 값 범위에 포함되지 않는 노드들이 연결되고, 낮은 레벨로 연결하는 링크에는 이 노드의 값 범위에 포함되는 후손 노드들이 연결된다. 낸드 플래시 저장 공간은 NNB(Normal Node Block)와 UNB(Updated Node Block)으로 구성된다. NNB는 갱신이 이루어지기 전의 초기 상태에 해당하는 CL-트리의 노드들이 저장되는 공간이고, UNB는 새로 갱신되는 노드들이 저장되는 공간이다. UNB는 1개 이상의 블록으로 구성되어 있으며, 전원 차단 시 CIL의 복구를 위해 사용된다. 삽입, 삭제, 탐색 모든 연산은 이 CIL에서 시작하고, 만약 키에 해당되는 리스트 노드가 CIL에 존재하면 저장된 주소를 따라 낸드 플래시 내의 노드에 직접적으로 접근을 수행한다. 만약 CIL에 키를 범위에 포함하는 리스트 노드가 존재하지 않으면, 모든 연산은 CL-트리에서 수행한다. 이는 키와 관련된 단말 노드가 루트 노드에서부터 연결이 이루어져있는 상태이기 때문이다. 각각의 주요 연산에 대해 살펴보면 다음과 같다.

2. 삽입 연산

삽입 연산은 CIL에서 관련된 노드를 찾는 것부터 이루어진다. 만약 CIL에 삽입되는 키와 관련된 리스트 노드가 없는 경우, 기존의 B+ 트리와 동일한 방법으로 키를 삽입한다. 삽입이 이루어진 새로운 페이지는 UNB에 저장된다. 그리고 UNB에 저장된 노드에 접근할 수 있도록 삽입 연산이 일어난 노드의 키 범위와 해당 물리 주소를 가진 리스트 노드를 생성한 다음 CIL에 저장을 한다. 이 때 저장되는 키의 범위는 현재 가지고 있는 키의 최소 값, 최대 값이 아닌 조상 노드의 키에 의해 해당 노드가 가질 수 있는 최소 값과 최대 값이 된다. CIL에 관련 리스트 노드가 있는 경우, 저장된 주소를 따라 페이지에 직접 접근하여 키를 삽입한다. 갱신된 페이지는 UNB에 저장되며, CIL에 있는 리스트 노드에 저장된 주소를 새로운 주소로 갱신한다. 삽입 과정의 자세한 동작은 알고리즘 1과 알고리즘 2에서 보여주고 있다. 그림 3(b)는 키 8, 17, 74, 75가 삽입되는 예를 보여주고 있다. 키 8, 17, 74는 페이지 D, 페이지 E, 페이지 G에 각각 저장되며, 갱신된 페이지에 대한 리스트 노드를 생성하여 CIL에 삽입한다. 이 때 저장된 72 ~ $+\infty$의 범위를 가지는 리스트 노드에 의해, 이어서 이루어지는 키 75 삽입 연산은 UNB의 G 페이지에 직접 접근하여 이루어진다. 삽입이 이루어지면 이전의 페이지는 무



(a) 초기상태

(b) Insertion(8, 17, 74, 75)

(c) Insertion(77)

(d) Insertion(78)

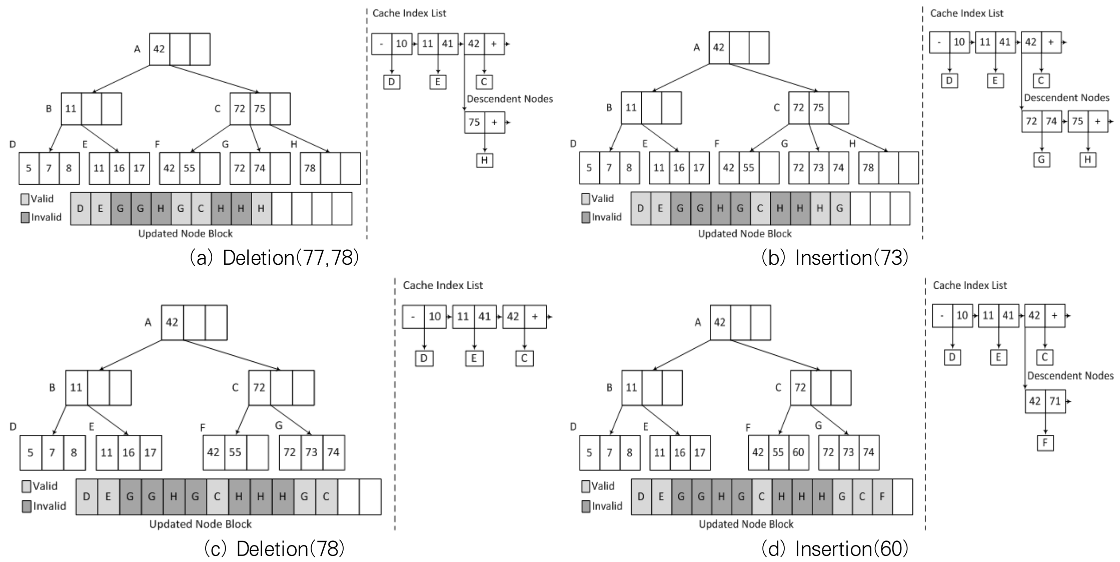
그림 3. 삽입 연산
Fig. 3. Insertion operation

효화되며, CIL 리스트 노드의 무효화된 페이지 주소는 새로운 페이지의 주소로 수정된다.

Algorithm 1: Insertion Algorithm
Input : Key K
N←Get Node N from CIL
(result, K')←TreeInsert(N,K)
while result is SPLIT do
N' ← GetParentNode(K', parentNodeLevel)
parentNodeLevel++
(result, K')←TreeInsert(N', K')
end while
if the CIL is full then
CIL reorganization
end if

키의 삽입으로 분할이 발생하면 분할이 일어난 페이지들은 UNB에 저장되며, 그 부모 페이지에 공간이 충분하면 분할키를 삽입한 후 UNB에 부모 페이지도 저장한다. 이 때 저장되는 3개의 페이지는 바로 UNB에 저장되는 것이 아니라 각 페이지들의 자식에 해당하는 CIL 리스트 노드의 주소를 반영한 후 UNB에 저장된다. 자신이 가지고 있는 주소가 반영된 CIL 리스트 노드들은 모두 삭제된다. 새로 저장된 부모 페이지가 루트가 아닌 경우 CIL에 새로운 리스트 노드를 생성하여 저장시킨다. 그림 3(c)는 그림 3(b)의 트리에서 키 77이 삽입되는 예를 보여주고 있다. 키 77의 삽입은 CIL 리스트 노드의 주소를 따라 페이지 G에 직접 접근하여 이루어진다. 페이지 G에 빈 공간이 없기 때문에 분할이 일어나며, 갱신된 페이지 G와 새로 생성된 페이지 H는 UNB에 저장되며 이 때 발생되는 분할키는 페이지 C에 삽입된다. 페이지 C의 자식 페이지에 해당하는 CIL 리스트 노드들의 주소를 페이지 C에 반영한 후, 페이지 C 역시 UNB에 저장한다. 주소가 반영된 CIL 리스트 노드들은 삭제가 이루지고 페이지 C와 관련된 CIL 리스트 노드는 새롭게 생성된 후 그림 3(c)와 같이 CIL에 저장된다.

만약 CIL에 저장하고자하는 노드의 조상 노드가 존재하는 경우, 저장되는 CIL 리스트 노드는 조상 CIL 리스트 노드의 낮은 레벨의 링크를 따라 자식으로써 저장된다. 그림 3(d)는 그림 3(c)에서 키 78이 삽입되는 예를 보여주고 있다. 키 78의 삽입은 탐색 정책에 따라 43 ~ +∞ 범위를 가지는 CIL 리스트 노드의 링크를 따라 페이지 C에 직접적으로 접근하여 이루어진다. 페이지 H에 공간이 충분하기 때문에 키 78이 삽입된 후, 페이지 H에 관련된 CIL 리스트 노드를 생성한다. 생성된 리스트 노드는 이미 존재하는 43 ~ +∞의 범위에 포함되는 노드이기 때문에 낮은 레벨의 링크에 연결된다.



(a) Deletion(77,78) (b) Insertion(73)

(c) Deletion(78) (d) Insertion(60)

그림 4. 삽입 및 삭제 연산의 복합시나리오

Fig. 4. The composite scenarios of Insertion operation and Deletion operation

Algorithm 2: TreeInsert Algorithm
Input : Node N, Key K
if N≠leaf node of the tree then
  (result, K)←TreeInsert($N.p_i$,K)
if result≠SPLIT then
  return result
if the key number of N 〈 the total key entry then
  N'←insert Key K in node N
  InsertToUNB(N')
  if LinkNode corresponding to Node N doesn't exist then
    LinkNode L←create new LinkNode which have the range of Node N' value and the pointer of Node N'
    Insert LinkNode L to CIL
  else
    LinkNode modify
  return NONE
else
  Split node N into Node N_1 and Node N_2
  N_1 or N_2←insert Key K
  Delete LinkNode corresponding to children Nodes of Node N_1 and Node N_2 in CIL
  Insert Node N_1 and Node N_2 to UNB
  if N is root then
    Create new root node
    return NONE
  return SPLIT
return result

3. 삭제 연산

삭제 연산은 삽입 연산과 동일한 방법으로 CIL에서 관련된 노드를 찾는 것부터 이루어진다. CIL에 리스트 노드가 존재하는 노드에서 키 삭제가 이루어졌다면, 키를 삭제한 노드는 UNB에 저장되고 CIL 리스트의 주소를 갱신한다. CIL에 리스트 노드가 없는 경우, 키를 삭제한 노드를 UNB에 저장한 후 삽입 연산과 동일하게 새로운 CIL 리스트 노드를 생성하여 CIL에 저장한다. 그림 4는 삭제 및 삽입이 복합적으로 이루어지는 예를 보여주고 있다. 그림 4(a)은 그림 3(d)에 이어서 키 75, 77가 삭제되는 것을 보여주고 있다. 키 75, 77이 삭제된 페이지 H는 UNB에 저장되고, 페이지 H에 해당하는 CIL의 리스트 노드에 저장된 주소는 새로운 주소로 변경된다. 그림 3과 그림 4에서 볼 수 있듯이 삽입과 삭제 연산이 인접한 노드에서 이루어질 때 단말 노드 접근 속도가 빠른 것을 볼 수 있다. 덮어쓰기가 불가능한 낸드 플래시에 CL-트리의 균형 분배를 사용하게 되면 많은 쓰기 연산이 일어나 낸드 플래시의 수명이 짧아질 수 있으므로, CL-트리는 균형 분배를 사용하지 않는다. 노드의 삭제는 노드에 키가 없을 때 수행된다.

4. 검색 연산

검색 연산은 CIL에서 키가 포함된 범위를 가진 노드를 찾는 것부터 이루어진다. 해당 리스트 노드를 찾지 못하면 CL-

트리에서 다시 탐색을 수행한다. CIL에서 키와 관련된 리스트 노드를 찾으면 검색은 해당 노드가 가지고 있는 주소로 직접 페이지에 접근한 뒤 이루어진다. 이렇게 함으로써 최근 업데이트된 노드들은 더 빠르게 접근이 가능하다. 자세한 동작은 알고리즘 3과 같다.

```
Algorithm 3: Search Algorithm
Input : Key K
Output : Node N (which points to the record
corresponding to key K)
N←Get Node N from CIL
while N≠ leaf node of tree do
  if K < N.K_1 (N.K_i refers to the i-th search field
value in Node N) then
    N←N.P_1
  else if K ≥ N.K_{q-1} then (q is the number of
pointers in Node N)
    N←N.P_q
  else then
    Search node n for an entry i such that K_{i-1}
≤ K < K_i
    N←N.P_i
end while
return N
```

5. 재구성

재구성은 단말 노드에서부터 루트 노드에 이르는 갱신된 모든 노드들에 대하여 이루어지는 전역 재구성과 CL-트리의 일부 가지들만 갱신을 수행하는 지역 재구성이 있다. 전역 재구성은 시스템 종료와 같이 CIL 전체를 비울 필요가 있을 때 수행되며, 전역 재구성이 종료되면 기존의 UNB를 해제하여 NNB로 설정하고 빈 블록들을 새로운 UNB로 설정한다. 지역 재구성은 일부 가지영역만 재구성을 수행하는 것으로 CIL이 사용하는 메모리양이 특정 임계값을 넘을 때 메모리 공간을 확보하기 위한 용도로 사용된다.

6. 전원차단에 따른 복구

제안한 CL-트리는 새로 삽입되거나 갱신되는 노드의 연결 정보를 메모리에 있는 CIL에 저장하고 있다. 따라서 갑작스러운 전원 차단에 대비하여 복구하는 기능이 필요하다. 새롭게 써지거나 갱신되는 노드들은 모두 1개 이상의 블록으로 구성된 UNB에 저장하고 있는데, 갑작스러운 전원 차단이 일어나면 UNB내에 있는 유효 페이지들을 삽입된 순서대로 읽으면서 CIL을 복구할 수 있다. 복구에 필요한 데이터를 위해 B+ 트리의 노드는 헤드를 가지고 있으며, 이 헤드에는 노드의 높이, 노드가 가질 수 있는 최소 값과 최대 값을 가지고 있다. 노드 높이 값은 단말 노드가 1이며 부모 노드로 갈수록 1씩 증가하는 형태로 저장된다. 복구를 수행할 때는 이 헤드의 정보를 이용하여 CIL을 재생성 한다. 이러한 경우들에 대하여 실험을 진행하여 CIL이 100%로 복구가 이루어짐을 확인하였다.

IV. 실 험

1. 환경 설정

본 연구는 가공되지 않은 낸드 플래시 칩(raw NAND Flash chip)을 사용하고 B+ 트리를 사용하는 임베디드 환경에 적합하다. 즉, 실험환경이 독립적이며, 제안한 기법은 어떤 낸드 플래시 칩에도 동일하게 적용된다. 성능 분석을 위하여 실제 낸드플래시 메모리가 있는 기기로 실험을 진행하였으며, 사용한 낸드플래시 메모리의 상세는 표 2과 같다. 각 실험은 20번을 수행한 평균값을 사용하였고, 평가를 위해 기존의 B+트리, μ-트리, LUB-트리와 성능을 비교하였다. 사용된 키는 무작위로 생성하였으며 삽입 연산 시 중복된 키 값을 사용하지 않았다.

표 2. 실험 환경
Table 2. Experimental environment

Chip	H27UCG8T2ATR-BC
Page Size	8192 + 640(Spare)byte
Block Size	2M + 160Kbyte, 256pages
Device Size	4096 + 84(Extended)blocks
Random Read Time	60µs
Page Program Time	1500µs
Block Erase Time	5.0ms

2. 쓰기 성능 비교

그림 5는 페이지 크기를 4KB로 설정한 환경에서의 레코드 삽입에 따른 쓰기 연산 발생 횟수를 보여주고 있다. B+트리의 경우 키 삽입이 일어나면 업데이트 파생이 발생하여 그림 5에서처럼 쓰기 연산이 많이 일어나는 것을 확인할 수 있다. LUB-트리는 루트 노드가 메모리에 항상 적재되어 있기 때문에 트리 높이가 높지 않을 때는 쓰기 연산이 적게 발생하였으나, 레코드 삽입에 따라 트리 노드 수가 증가할수록

쓰기 연산수가 μ-트리와 CL-트리에 비해 많이 발생하는 것을 확인할 수 있다. μ-트리와 CL-트리의 경우 업데이트가 파생되는 문제를 해결하여 쓰기 연산이 적게 발생하는 것을 확인할 수 있다. CL-트리는 B+ 트리에 비해 100만개의 키가 삽입이 이루어졌을 때 약 179%의 개선율을 보이며, LUB-트리에 비해 약 58%의 개선율을 보인다. 업데이트 파생으로 인한 추가적인 쓰기 연산이 없는 μ-트리는 1개의 노드가 전체 엔트리를 사용할 수 없어 다른 트리들에 비해 분할이 발생할 확률이 상대적으로 높아 CL-트리에 비해 약 1%의 쓰기 연산이 더 일어났다.

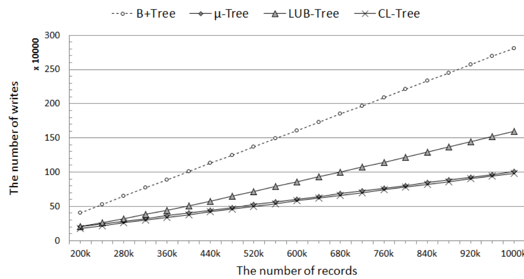
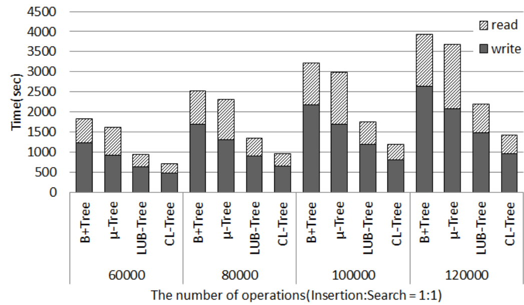
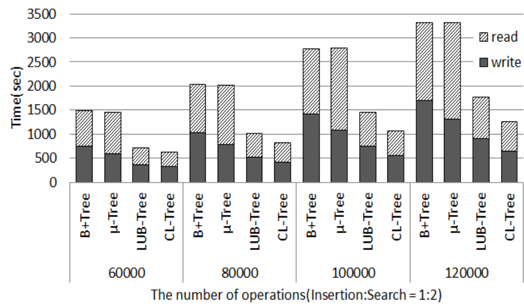


그림 5.삽입 연산에 따른 쓰기 횟수
Fig. 5. The number of writes per the insertion operation

3. 작업 수행 속도 비교

사용하는 어플리케이션마다 삽입 및 탐색 연산 비율이 달리 나타나므로 작업 수행속도 비교에서는 다양한 삽입 및 탐색 연산 비율로 실험을 수행하였다. 그림 6은 각 색인 구조에 대하여 무작위 키 삽입과 탐색을 1:1, 1:2, 1:4 비율로 수행한 시간을 나타내고 있다. μ-트리는 기존의 B+ 트리와 비교하여 쓰기 횟수가 많이 일어나지 않아 삽입 수행 시간이 적으나, 트리의 높이가 상대적으로 높아 탐색 수행 시간이 많은 것을 확인할 수 있다. 따라서 그림 6과 같이 탐색 비율이 높은 경우에는 전체 수행시간이 기존의 B+ 트리에 비해 더 느려진다. LUB-트리는 메모리를 사용하여 루트 노드를 상주시키고 있고, 최근 업데이트된 노드 역시 메모리에 있기 때문에 삽입 및 탐색 수행시간이 짧은 것을 볼 수 있다. CL-트리 역시 메모리에 최근 업데이트된 노드들의 주소를 저장하고 있고 삽입 및 탐색을 수행할 때 다른 노드들을 거치지 않고 저장된 주소를 이용하여 직접적으로 접근이 가능하여 작업 수행 시간이 다른 색인 구조에 비해 짧은 것을 확인할 수 있다. CL-트리의 삽입 속도와 탐색 속도는 기존의 B+ 트리에 비해 각각 약 173%와 179% 향상되었고, μ-트리와 비교하여 각각 114%, 110%, LUB-트리와 비교하여 각각 53%, 39% 향상되었다.





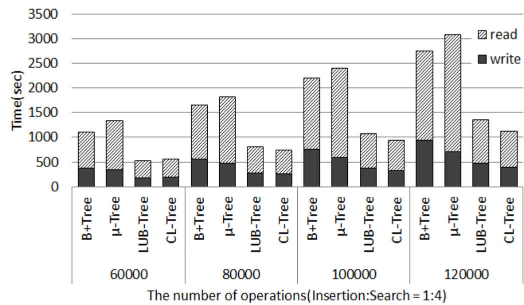


그림 6. 삽입 및 탐색 비율에 따른 수행시간 비교
Fig. 6. The comparison of run time against insertion and search operation ratio

4. 메모리 사용량

CL-트리는 메모리를 사용하고 있으며 이 메모리에 저장되어 있는 노드를 이용하여 업데이트 파생 문제와 작업 속도를 향상시키고 있다. 기본적으로 μ-트리는 삽입 연산 시 발생되는 쓰기 연산에서 우수한 성능을 보이지만, 또다시 참조 될 가능성이 있는 최근 업데이트된 노드에 대한 고려를 하지 않았다. 이러한 문제는 LUB-트리와 CL-트리에서 공통적으로 메모리 사용을 통해 문제를 해결하고 있으며, 사용되는 메모리 양은 블록 사상 기법에서 사용되는 것보다 적다.

만약 메모리를 충분히 활용할 수 없는 시스템에서 CL-트

리를 적용하는 경우에는 메모리의 소모량을 제한하여 사용할 수 있다. 그림 7은 CL-트리의 CIL 크기를 12KB로 제한하여 100만개의 레코드가 삽입되는 상황을 가정하여 이때 발생되는 평균 메모리 소모량과 그에 따른 쓰기 연산 횟수를 보여주고 있다. 메모리의 제한으로 추가적인 쓰기가 하였지만 100만개의 레코드가 삽입되었을 때 약 0.5%로 오버헤드가 매우 적다. 아울러 CL-트리는 LUB-트리와 같은 메모리를 소모하면서도 쓰기 연산이 57% 개선되었음을 확인하였다.

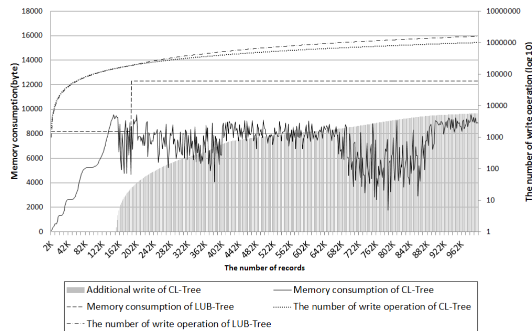


그림 7. 메모리 소모량 및 쓰기 횟수 비교
Fig. 7. The comparison of memory consumption and the number of write

V. 결 론

본 논문에서는 사상 테이블을 사용하기 힘든 낸드 플래시 시스템에서 사용할 수 있는 새로운 색인 구조를 제안하였다. 제안한 색인 구조는 제자리 갱신이 불가능한 낸드 플래시의 고유한 특성에서 발생되는 업데이트 파생 문제를 해결하기 위해 다중 리스트를 사용하고 있다. 실험을 통해 제안한 색인 구조가 기존의 연구와 비교하여 삽입 연산에 따른 쓰기 연산에서 좋은 성능을 나타냄을 보였다. 또한 최근에 접근한 노드들은 메모리에 상주하는 리스트를 통해 다른 페이지들을 거치지 않고 바로 접근할 수 있어 전반적인 작업 수행속도에도 강점을 보임을 실험을 통해 보였다. 또한 제안하는 색인 구조는 CIL이 사용하는 메모리 양을 조절할 수 있어 다양한 시스템에 따라 탄력있게 적용할 수도 있다. 아울러 갑작스러운 전원 차단 시에는 새로 업데이트되는 노드들이 저장되어 있는 UNB내에 있는 유효 노드들을 이용하여 CIL을 복구할 수 있다. 향후에는 가비지 컬렉션 및 낸드 플래시 수명을 더욱 늘리기 위한 마모도 평준화를 고려하여 연구를 진행할 예정이다.

REFERENCES

〔1〕 Chen, Feng, David A. Koufaty, and Xiaodong Zhang. "Understanding intrinsic characteristics and system implications of flash memory based solid state drives." ACM SIGMETRICS Performance Evaluation Review. Vol. 37. No. 1. ACM, 2009.

〔2〕 Chung, Tae-Sun, et al. "A survey of flash translation layer." Journal of Systems Architecture 55.5 (2009): 332-343.

〔3〕 Gupta, Aayush, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. Vol. 44. No. 3. ACM, 2009.

〔4〕 Dongwon Kang, Dawoon Jung, Jeong-Uk Kang and Jin-Soo Kim., "μ-Tree : An Ordered Index Structure for NAND Flash Memory", Proc. of the 7th Annual ACM Conference on Embedded Systems Software, 2007.

〔5〕 Hua-Wei Fang, Mi-Yen Yeh, Pei-Lun Suei, and Tei-Wei Kuo, "An adaptive endurance-aware b+-tree for flash memory storage systems", IEEE Transactions on Computers, 2013.

〔6〕 Bo-kyeong Kim, Min-hee Yoo, and Dong-Ho Lee, "An Efficient B-Tree Using Lazy Update on Flash Memory." Journal of KIISE : Databases (2012): 109-119.(in Korea)

〔7〕 Wu, Chin-Hsien, Li-Pin Chang, and Tei-Wei Kuo. "An efficient B-tree layer for flash-memory storage systems." Real-Time and Embedded Computing Systems and Applications. Springer Berlin Heidelberg, 2004. 409-430.

〔8〕 Nath, Suman, and Aman Kansal. "FlashDB: Dynamic self-tuning database for NAND flash." Proceedings of the 6th international conference on Information processing in sensor networks. ACM, 2007.

〔9〕 Agrawal, Devesh, et al. "Lazy-adaptive tree: An optimized index structure for flash devices."

Proceedings of the VLDB Endowment 2.1 (2009): 361-372.
[10] Li, Yinan, et al. "Tree indexing on solid state drives." Proceedings of the VLDB Endowment 3.1-2 (2010): 1195-1206.
[11] Artem B. Biryutskiy, "JFFS3 design issues", http://www.linux-mtd.infradead.org, 2005.
[12] Lee, Yong-Goo, et al. "μ-FTL:: a memory-efficient flash translation layer supporting multiple mapping granularities." Proceedings of the 8th ACM international conference on Embedded software. ACM, 2008.
[13] J. Ahn. D. Kang, D. Jung, J. Kim and S. Maeng, "μ*-Tree: An Ordered Index Structure for NAND Flash Memory with Adaptive Page Layout Scheme", IEEE Transactions on Computers, vol PP, no.99, p. 1, 2012.
[14] Fang, Hua-Wei, et al. "A flash-friendly B+-tree with endurance-awareness." Embedded Systems for Real-Time Multimedia (ESTIMedia), 2011 9th IEEE Symposium on. IEEE, 2011.
[15] NA, GAPJOO, Bongki Moon, and Sang-Won Lee. "IPL B-Tree for flash memory database systems." Journal of information science and engineering 27 (2011): 111-127.
[16] On, Sai Tung, et al. "Lazy-update b+-tree for flash devices." Mobile Data Management: Systems, Services and Middleware, 2009. MDM'09. Tenth International Conference on. IEEE, 2009.
[17] Yin, Shaoyi, Philippe Pucheral, and Xiaofeng Meng. "A sequential indexing scheme for flash-based embedded systems." Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology. ACM, 2009.

저 자 소 개

황 상 호
2009: 영남대학교 컴퓨터공학과 공학사.
2013: 영남대학교 컴퓨터공학과 공학석사.
현 재: 영남대학교 컴퓨터공학과 박사과정.
관심분야: 컴퓨터 구조, 영상처리
Email : snailcom@ynu.ac.kr

곽 종 욱
1998: 경북대학교 컴퓨터공학과 공학사.
2001: 서울대학교 컴퓨터공학과 공학석사.
2006: 서울대학교 전기컴퓨터공학부 공학박사
현 재: 영남대학교 컴퓨터공학과 부교수
관심분야: 컴퓨터 구조, 고성능 컴퓨팅, 임베디드 시스템
Email : kwak@yu.ac.kr