Journal of The Korea Society of Computer and Information
Vol. 20, No. 6, June 2015
www.ksci.re.kr
http://dx.doi.org/10.9708/jksci.2015.20.6.001

플래시 메모리 기반 인덱스 구조에서 대리블록 이용한 가비지 컬렉션 기법

김선환*, 곽종욱*

Garbage Collection Method using Proxy Block considering Index Data Structure based on Flash Memory

Seon Hwan Kim*, Jong Wook Kwak*

요 약

낸드 플래시 메모리는 빠른 접근 시간과 저전력의 특성을 가지고 있어 저장장치로 많이 사용되고 있는 추세이다. 하지만 저사양의 임베디드 장치에서는 메모리 요구사항과 구현상의 복잡성으로 FTL을 적용하기에는 비용이 많이 든다. 이러한 이유로 FTL을 구현하기 힘든 임베디드 장치에 적용할 수 있는 B+ 트리 연구들이 다수 제안되었다. 이런 연구들은 낸드 플래시 메모리에서 제자리 업데이트가 불가하다는 단점을 고려하여 삽입과 갱신의 성능을 최적화 하였다. 하지만 B+ 트리에 기존의 가비지 컬렉션 기법들을 적용하면 낸드 플래시 메모리의 페이지 위치를 변경하게 되고 B+ 트리의 재구성을 발생시켜 전체적인 성능을 저하시킨다. 이러한 문제를 해결하고자 본 논문에서는 낸드 플래시 메모리를 기반으로 하는 B+ 트리와 이와 유사한 인덱스 트리 구조에 적용할 수 있는 가비지 컬렉션 기법을 제안한다. 제안하는 가비지 컬렉션 기법은 블록 정보 테이블과 대리 블록을 이용하여 B+ 트리의 재구성을 발생시키지 않는다. 제안된 기법의 성능평가를 위해, 낸드 플래시 메모리가 장착된 실험 장치에 B+ 트리와 μ-Tree를 구현하고 제안된 기법을 적용하였다. 구현 결과 B+ 트리에서 제안된 기법이 GAGC(Greedy Algorithm Garbage Collection)보다 삽입된 키의 개수가 약 73% 많았으며, μ-Tree에서 제안된 기법이 GAGC보다 시간 오버헤드가 약 39% 적었다.

▸ Keywords : 플래시 메모리, B+ 트리, 가비지 컬렉션, 저장 시스템, 색인 트리 구조

Abstract

Recently, NAND flash memories are used for storage devices because of fast access speed and low-power. However, applications of FTL on low power computing devices lead to heavy workloads which result in a memory requirement and an implementation overhead. Consequently, studies of B+-Tree on

• 제1저자 : 김선환 • 교신저자 : 곽종욱
• 투고일 : 2015. 3. 4, 심사일 : 2015. 4. 30, 게재확정일 : 2015. 6. 8.
* 영남대학교 컴퓨터공학과(Department of Computer Engineering, Yeungnam University)
※ 이 논문은 2014년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임.
(No. NRF-2014R1A1A2057146)

embedded devices without the FTL have been proposed. The studies of B+-Tree are optimized for performance of inserting and updating records, considering to disadvantages of the NAND flash memory that it can not support in-place update. However, if a general garbage collection method is applied to the previous studies of B+-Tree, a performance of the B+-Tree is reduced, because it generates a rearrangement of the B+-Tree by changing of page positions on the NAND flash memory. Therefor, we propose a novel garbage collection method which can apply to the B+-Tree based on the NAND flash memory without the FTL. The proposed garbage collection method does not generate a rearrangement of the B+-Tree by using a block information table and a proxy block. We implemented the B+-Tree and μ-Tree with the proposed garbage collection on physical devices with the NAND flash memory. In experiment results, the proposed garbage collection scheme compared to greedy algorithm garbage collection scheme increased the number of inserted keys by up to about 73% on B+-Tree and decreased elapsed time of garbage collection by up to about 39% on μ-Tree.

▸ Keywords : Flash memory, B+-Tree, Garbage collection, Storage system, Index data structure

I. 서 론

낸드 플래시 메모리는 데이터가 저장된 위치를 기계적인 방식으로 탐색하는 일반 하드 디스크와는 달리 반도체 소자를 이용하여 데이터를 저장하고 주소를 통해 데이터가 저장된 장소를 바로 접근할 수 있다. 그래서 낸드 플래시 메모리는 하드 디스크에 비해 접근속도가 빠르고 소형화 및 경량화가 쉽다. 또한 플래시 메모리의 직접도가 지속적으로 높아지고 가격이 저렴해지고 있어 해마다 사용량이 늘어나고 있는 추세이다. 이런 특징으로 인하여 낸드 플래시 메모리는 일반 PC 이외에도 휴대폰, 태블릿 등의 임베디드 시스템의 데이터 저장장치로 많이 사용되고 있다.

하지만 낸드 플래시 메모리는 하나의 페이지에 제자리 덮어쓰기가 지원되지 않으며 하나의 블록에 지울 수 있는 연산의 횟수가 제한되어 있는 단점을 가지고 있다. 또한 쓰기 및 읽기는 페이지 단위이지만 지우기는 블록 단위로 이루어진다. 따라서 일반 저장장치와는 다르게 덮어쓰기가 발생하면 다른 페이지에 데이터를 저장하고 변경된 위치의 정보를 저장해야 한다. 이를 위해 논리적인 주소와 실제 데이터가 저장되어 있는 물리적인 위치의 정보를 주소 매핑 테이블에 저장한다. 이때, 덮어쓰기로 변경된 데이터 위치를 주소 매핑 테이블에 저장하면 기존의 페이지는 무효 페이지로 설정한다. 무효 페이지의 수가 많아지면 실제로 사용할 수 있는 메모리 공간이 부족해진다. 이를 해결하고자 무효 페이지를 수집하여 공간을 확보하는 가비지 컬렉션(garbage collection)을 수행한다. 이런 작업들을 수행해야 일반 하드 디스크를 기반으로 하는 파일 시스템과 응용 프로그램을 구동할 수 있다. 낸드 플래시 메모리에서 이런 작업들을 수행하고 관리하는 소프트웨어 계층을 FTL(Flash Translation Layer)라고 한다. FTL은 주요 목적인 주소 매핑 방법에 따라 크게 페이지 단위 매핑, 블록 단위 매핑, 하이브리드 매핑 등의 방법이 있다[1, 2].

FTL은 주소 매핑, 가비지 컬렉션 이외에도 마모도 평준화(wear leveling)와 복구 기능 등의 여러 가지 추가 작업을 수행해야 한다. 하지만 저사양 임베디드 시스템에 이를 모두 적용 및 구현하기에는 필요한 메모리와 연산처리 비용이 크다. 그래서 최근에는 저사양 임베디드 시스템에 초점을 두고 낸드 플래시 메모리에 필요한 기능만을 포함한 B+ 트리에 대한 연구가 활발히 진행되고 있다[3, 4, 5, 6, 7].

B+ 트리는 데이터가 갱신되면 제자리 덮어쓰기가 되지 않는 낸드 플래시 메모리의 특성 때문에 물리적인 위치가 변경된다. 이로 인해 변경된 위치를 다시 상위 노드에 저장해야 한다. 이러한 트리 재구성 및 갱신에 소요되는 작업으로 인하여 B+ 트리의 성능이 저하되는 특징이 있다. 많은 연구들이 B+ 트리의 삽입, 갱신 연산에 트리의 재구성과 상위 노드 갱신을

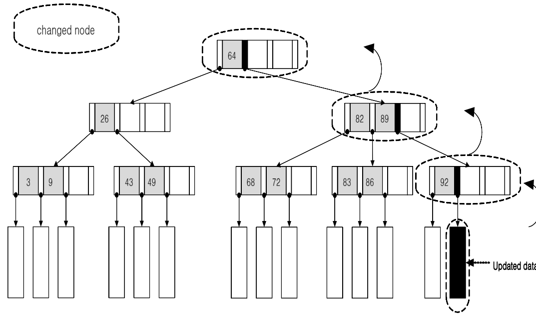


그림 1. 업데이트 파생 문제
Fig 1. Update propagation problem

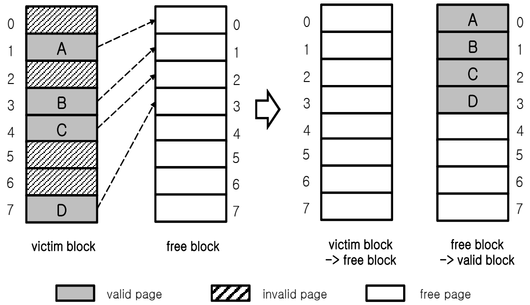


그림 2. 가비지 컬렉션 수행 과정
Fig 2. Garbage collection processing

줄여 속도를 향상시키는 기법들을 제안하였다. 하지만 이런 연구들에는 기존의 가비지 컬렉션을 적용할 수가 없다. 가비지 컬렉션 작업은 공간을 확보하는 과정에서 데이터가 저장되어 있는 페이지의 물리적인 위치를 변경하게 된다. 그래서 B+ 트리의 재구성을 유발하여 성능을 저하 시킨다. 이런 문제를 해결하기 위해 본 논문에서는 B+ 트리와 이와 유사한 인덱스 트리 구조에서 재구성을 발생시키지 않는 새로운 가비지 컬렉션 기법을 제안한다. 제안하는 기법은 블록 정보 테이블을 이용하여 블록 위치의 변경으로 발생되는 재구성을 해결하고, 대리블록을 이용하며 페이지 위치의 변경으로 발생되는 재구성을 해결한다.

이하 본 논문에서는 2장에서 관련연구와 B+ 트리상의 가비지 컬렉션 기법의 문제점을 지적하고, 3장에서 제안한 기법을 소개한다. 4장에서 낸드 플래시 메모리가 장착된 장치에 B+ 트리와 제안 기법을 적용하여 성능을 검증한다. 끝으로 5장에서 결론을 맺는다.

II. 관련 연구 및 배경지식

이 장에서는 낸드 플래시 메모리를 기반으로 하는 B+ 트리의 문제점을 설명하고, 이를 해결하고자 연구된 기법을 설명한다. 또한 가비지 컬렉션 수행의 기본 개념을 설명하고 이로 인해 B+ 트리에 발생하는 문제를 설명한다.

1. 업데이트 파생 문제

FTL을 탑재하기 힘든 임베디드 시스템이나 B+ 트리만을 고려해야할 시스템을 위해 여러 가지 기법들이 연구되고 있다. 낸드 플래시 메모리를 장착한 시스템에서 기존의 하드 디스크를 기반으로 하는 B+트리를 적용하면 업데이트 파생 문제(update propagation problem)가 발생하여 성능 저하가 발생한다. 업데이트 파생 문제는 다른 연구에서는 트리 배회 문제(wandering tree problem)로 언급되기도 한다[3, 4, 5].

그림 1은 업데이트 파생 문제를 나타내고 있다. 트리의 단말노드에서 가장 오른쪽에 있는 노드의 데이터를 갱신하게 되면 플래시 메모리의 특성상 덮어쓰기가 되지 않기 때문에 다른 페이지에 갱신된 데이터를 써야 된다. 페이지 주소가 변경된 해당 말단 노드는 B+ 트리의 특성상 부모노드가 자식노드의 주소를 저장해야 하기 때문에 부모노드 또한 데이터를 갱신해야 한다. 같은 방식으로 부모노드 또한 데이터가 갱신되면 덮어쓰기 문제로 인하여 상위노드가 모두 갱신되어야 한다. 따라서 그림 1의 예처럼 하나의 단말노드를 갱신하기 위해 총 4회의 쓰기가 발생하게 된다. 결국 B+ 트리의 키값을 갱신하기 위해서는 트리의 높이만큼 추가적인 쓰기 연산이 발생하게 되며, 이는 성능저하의 주요 원인이 된다.

2. 가비지 컬렉션

낸드 플래시 메모리는 무효 페이지를 수거하여 빈 공간을 확보하는 가비지 컬렉션을 수행해야 한다. 그림 2에서 가비지 컬렉션의 일반적인 수행 과정을 표시하고 있다. 무효 페이지가 많은 블록을 희생블록으로 설정하고 희생블록의 유효 페이지가 이주하게 되는 가용블록(free block)을 선택한다. 희생블록의 유효 페이지를 모두 가용블록으로 이주시킨 후, 희생블록을 지워 공간을 확보한다.

가비지 컬렉션 작업은 B+ 트리에서 업데이트 파생 문제를 유발하는 2가지의 물리적인 위치 변경이 발생한다. 첫 번째로 블록 번호가 변경되고, 두 번째로 페이지 번호가 변경된다. 이런 물리적인 위치 변경으로 가비지 컬렉션은 B+ 트리의 성능을 저하시킨다. 추가적으로 낸드 플래시 메모리에서 셀 당 저장되는 비트수가 한 개인 SLC와 달리 두 개의 비트를 셀에 저장하는 MLC는 블록에 데이터를 기록할 때 페이지를 순차적으로 써야 한다. 그래서 MLC의 이런 제한사항은

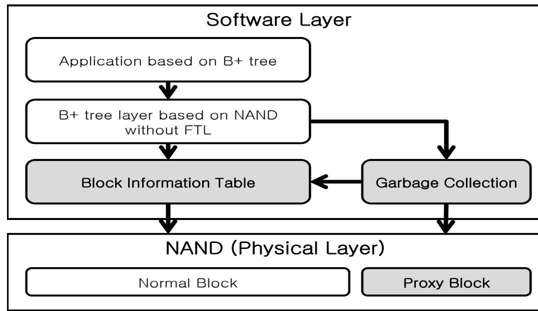


그림 3 전체 구조도
Fig 3. Overview of system

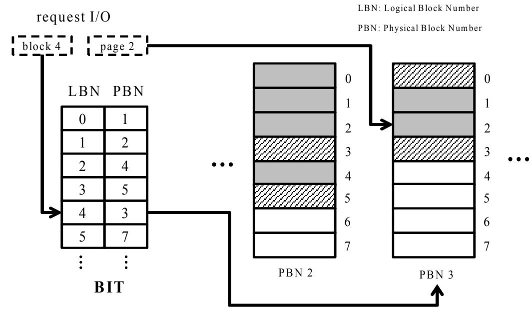


그림 4. BIT를 이용한 블록 주소 매핑
Fig 4. Block address mapping using BIT

가비지 컬렉션 작업을 더욱 복잡하게 만든다[8, 9].

3. 플래시 메모리를 위한 색인 자료구조

업데이트 파생 문제를 해결하기 위해 플래시 메모리의 특성을 고려한 인덱스 트리 구조들이 많이 연구되고 있다. μ-Tree는 업데이트 파생 문제를 해결하기 위해 갱신이 발생되면 관련 있는 상위 노드들을 한 개의 페이지에 모두 저장한다. 한 개의 페이지에 상위 인덱스 노드들을 모두 저장하기 위해서 페이지 영역을 트리의 높이에 따라 분할한다. 상위 노드를 저장할 때마다 페이지의 영역을 1/2로 나누어 저장한다. 이는 데이터 갱신 시 한 번의 페이지 쓰기만 이루어지기 때문에 속도가 빠르다는 장점을 가지고 있다[4]. μ-Tree에서 공간 효율성을 높이기 위해 제안된 μ*-Tree는 B+ 트리의 높이와 키의 개수 등을 분석하고 계산하여 페이지 영역을 할당하도록 제안되었다[5]. IPL은 B+ 트리에서 키의 값이 갱신되거나 삽입이 발생되면 각 블록의 일부를 로그 영역으로 활용하여 저장한다[6]. D-IPL은 MLC 낸드 플래시 메모리의 특성에 맞게 IPL을 수정하였다[7]. 플래시 메모리를 위한 동적 해쉬는 갱신 데이터를 로그 블록에 저장하고 해쉬 테이블의 분할 과정을 lazy 분할과 eager 분할로 나누어 해쉬를 확장한다[10].

III. 대리블록을 이용한 가비지 컬렉션

제안하는 기법의 전체적인 구조는 그림 3과 같다. 상위에 있는 B+ 트리 계층과 어플리케이션은 블록 정보 테이블(BIT: Block Information Table)을 이용하여 물리 계층에 있는 낸드 플래시 메모리에 접근한다. 가용할 수 있는 공간이 부족하면 제안하는 가비지 컬렉션 기법이 수행된다.

1. BIT

BIT는 가비지 컬렉션에서 발생하는 블록 단위 주소 변경을 해결하기 위해 사용된다. BIT는 블록 단위로 논리 주소와 물리 주소를 매핑하는 역할을 한다. FTL의 블록 단위 주소 매핑 기법과 유사하나 페이지 단위 위치는 고려하지 않는다. FTL에서 사용되는 블록 단위 주소 매핑은 페이지 위치를 찾기 위한 작업이 추가로 필요하다[1]. 하지만 BIT는 페이지 위치 검색에 대한 추가적인 작업이 없기 때문에 주소 매핑에 대한 성능 복잡도는 O(1)이다. 또한 BIT는 블록 수만큼만 공간이 필요하기 때문에 공간적인 비용도 적게 든다. B+ 트리 계층에서 노드에 대한 블록과 페이지의 위치를 요청하면 BIT는 중간 계층에 위치하여 논리블록에 해당하는 물리적인 위치를 변환하여 전달한다. 그림 4는 BIT를 이용한 주소 매핑을 나타내고 있다. 블록 4번과 페이지 2번에 쓰기나 읽기에 대한 요청이 발생하면 해당 블록만 BIT를 통해서 논리적인 위치 4번에 대응하는 물리적인 위치 3번으로 변환하고 페이지 위치는 그대로 사용한다. BIT에는 이런 매핑정보 이외에도 블록의 남은 페이지 개수, 무효 페이지 개수를 추가하여 사용한다.

BIT는 가비지 컬렉션에서 블록의 변경으로 인해 발생되는 트리 구조의 재구성을 방지한다. 그래서 제안된 기법은 BET, SGC와 같은 블록 단위로 데이터를 교체하는 마모도 평준화 기법에서도 트리 구조의 재구성 없이 적용될 수 있다[11, 12].

2. 대리블록

대리블록은 가비지 컬렉션에서 발생하는 페이지 단위의 주소 변경을 해결하기 위해 사용된다. 낸드 플래시 메모리의 가용블록들 중에서 하나의 블록을 대리블록으로 지정하며, 이는 가비지 컬렉션이 진행되는 상황에서만 사용되고 그 외의 경우에는 사용되지 않는다. 주요 역할은 가비지 컬렉션에서 희생블록의 유효 페이지가 이주될 블록으로 사용된다. 일반적인 가비지 컬렉션에서 희생블록의 유효 페이지를 가용블록으로 복사할 때 모든 유효 페이지를 한 번에 가용블록에 이주하고

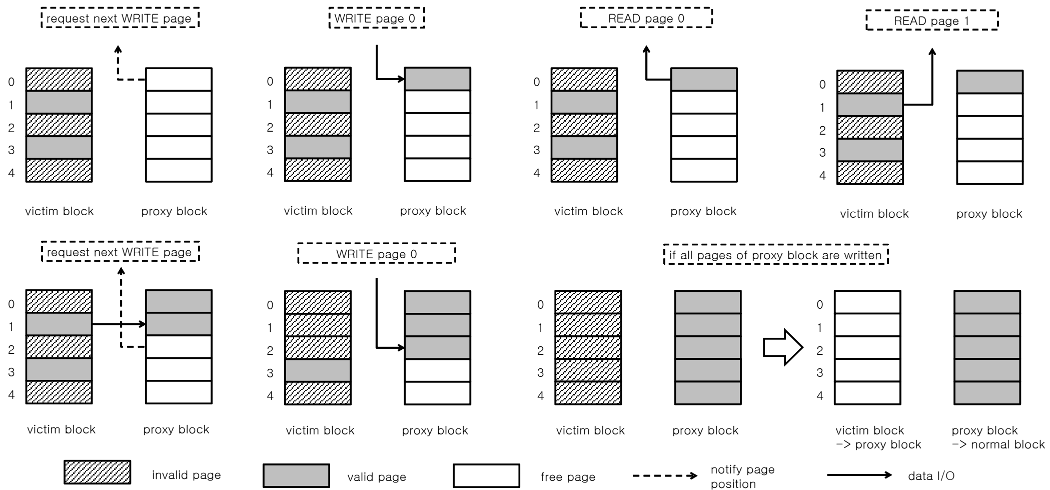


그림 5. 대리블록을 이용하는 가비지 컬렉션에서의 페이지 연산
Fig 5. Page operations on the garbage collection exploiting a proxy block

희생블록을 지워 공간을 확보한다. 하지만 제안 기법에서 희생블록의 유효 페이지를 대리블록으로 바로 이주하지 않고, 페이지 쓰기 요청이 발생할 때 마다 희생블록의 일부 유효 페이지들을 이주한다. 그리고 가비지 컬렉션이 수행 중이면 우선적으로 대리블록에 존재하는 가용 페이지를 사용하고, 희생블록과 대리블록이 하나의 가상블록이 되어 유효 페이지의 위치에 따라 각 블록을 참조한다.

3. 가비지 컬렉션 수행과 페이지 연산

제안하는 기법에서 수행되는 한 단위의 가비지 컬렉션은 일정 기간 동안 진행된다. 가비지 컬렉션이 수행되면, 다음 데이터가 저장되어야 하는 가용 페이지의 순서는 대리블록 내의 페이지들로 설정된다. 그리고 대리블록의 페이지들이 모두 사용되면 가비지 컬렉션이 완료된다. 가비지 컬렉션이 수행 중일 때는 희생블록과 대리블록이 하나의 가상 논리 블록으로 매핑된다. 이때 가상 논리 블록의 번호는 희생블록의 논리번호로 설정된다. 그리고 희생블록의 무효 페이지 개수만큼 대리블록에 가용 페이지가 존재한다고 인식한다. 그래서 가용 페이지 번호의 요청이 들어오면 희생블록의 무효 페이지 번호를 전달한다. 쓰기 연산이 발생하여 데이터와 해당하는 페이지 번호를 전달받으면, 해당 페이지의 번호와 희생블록의 정보를 참조하여 희생블록의 무효 페이지 번호와 같다면 대리블록의 페이지에 쓰기 연산을 수행한다. 만약 다음에 쓰기 연산을 수행할 페이지 번호가 순차적으로 진행이 되지 않고 2 단위이상으로 전달되면 이전에 쓰기 연산을 수행한 페이지 번호와 현재에 쓰기 연산을 수행하는 페이지 번호 사이의 페이지들은 희생블록의 유효 페이지가 된다. 그래서 현재 데이터를 페이지에 쓰기 전에 희생블록의 유효 페이지를 먼저 대리블록으로 이주한다. 가비지 컬렉션 수행 중에 읽기가 발생하면 대리블록에서 최근에 데이터가 저장된 페이지 번호를 기준으로 번호 이상이면 희생블록에서 읽기를 수행하고 아니면 대리블록에서 읽기를 수행한다.

그림 5는 가비지 컬렉션이 발생하였을 때 낸드 플래시 메모리의 페이지 연산을 도식화 한 것이다. 그림 5에서 가비지 컬렉션이 발생하면 희생블록과 대리블록은 하나의 가상 논리블록으로 형성되고 가용 페이지 요청이 들어오면 희생블록의 무효 페이지 번호인 0번을 알려준다. 0번 페이지에 쓰기 연산이 발생하면 해당하는 대리블록의 페이지에 데이터를 저장한다. 0번에 읽기가 발생하면 최근에 데이터가 저장된 대리블록의 페이지 위치를 기준으로 기준값 이하가 되기 때문에 대리

```
Algorithm 1: GC_Proxy
GC_flag : indicates whether GC(Garbage Collection) is processing
or not, global variable
V : victim block number, global variable
1   V <- get block which has maximum invalid page
2   if is V invalid block? then
3       erase block V
4       return
5   endif
6   if GC_flag = 1 then
7       GC_flag ← 1
8   end if
```

그림 6. 대리블록을 이용한 가비지 컬렉션 의사코드
Fig 6. Pseudo code of the garbage collection using a proxy block

블록의 0번 페이지의 데이터를 읽는다. 1번에 읽기가 발생하면 기준값을 초과하기 때문에 희생블록에서 읽는다. 그 후 다시 가용 페이지 요청이 발생하면 다음 희생블록의 무효 페이지 위치가 2번이기 때문에 2번을 알려주고 쓰기 연산이 발생하기 전에 0번과 2번 사이에 있는 희생블록의 1번 유효 페이지를 대리블록으로 이주한다. 그리고 희생블록의 1번 페이지는 무효 페이지로 처리한다. 최종적으로 대리블록의 모든 페이지가 사용되면 희생블록을 지워 가용블록으로 만들고 그 가용블록을 대리블록으로 지정한다.

그림 6은 제안한 기법의 가비지 컬렉션 알고리즘 의사코드이다. 알고리즘 1은 대리블록의 사용 여부를 알려주는 플래그

```
Algorithm 2: Page R/W
Input: block number B, page number P
BIT: Block Information Table
B_proxy: proxy block number, global variable
1   B_p ← get physical address from BIT
2   if GC_flag = 1 AND V = B_p then
3       call ReadOnGCP(B_p, P)
          or PhysicalWrite(B_proxy, P)
4       return
5   else
        /* read/write corresponding to B_p, P */
6       call PhysicalRead(B_p, P)
          or PhysicalWrite(B_p, P)
7   end if

Algorithm 3: ReadOnGCP
Input: physical block number B_p, page number P
P_last: last write page position
1   P_last ← get last write page position from B_proxy
2   if  P > P_last then
3       call PhysicalRead(V, P)
4   else
5       call PhysicalRead(B_proxy, P)
6   end if

Algorithm 4: RequestNextFreePage
Output: block number B, page number P
1   if GC_flag = 1 then
2       V_pos ← last write page position from B_proxy
3       while V_pos is valid page do
4            V_pos ← V_pos + 1
5            migrate page data to B_proxy from V_pos
6       end while
7       B ← V
8       P ← V_pos
9   else
10      B ← search block which has free page
11      P ← free page of B
12  end if
```

그림 7. 가비지 컬렉션 관련 함수들의 의사코드
Fig 7. Pseudo code of functions related to the garbage collection

변수를 설정하는 작업을 수행한다. 무효 페이지로만 구성된 무효블록은 추가적인 유효 페이지의 복사가 없어 트리의 구조에 영향을 주지 않는다. 그래서 우선적으로 무효블록을 선택하여 가용블록으로 만든다. 만약 무효블록이 없으면 플래그변수를 설정하여 대리블록을 사용하기 시작한다.

그림 7은 가비지 컬렉션 관련 함수들의 의사 코드이다. 알고리즘 2는 해당 블록과 페이지에 쓰기와 읽기 연산이 발생하면 BIT를 활용하여 물리적인 위치를 찾고, 가비지 컬렉션이 수행중이면서 희생블록에 읽기가 시도되면 대리블록을 이용한다. 알고리즘 3은 가비지 컬렉션이 진행 중이고 희생블록의 읽기가 발생하면 위치에 따라 대리블록의 페이지를 참조한다. 최근에 대리블록에 쓰기 연산을 수행한 페이지 번호를 변수에 저장한다. 그리고 이 값을 기준으로 하여 읽기 요청 페이지 번호가 기준값 이하이면 대리블록에 읽기 연산을 수행하고, 초과하면 희생블록에서 읽기 연산을 수행한다. 알고리즘 4는 다음에 쓰기 연산을 수행할 가용 페이지를 요청하는 함수이다. 응용 계층은 이 함수를 통해 데이터를 저장할 다음 가용 페이지의 위치를 알 수 있다. 가비지 컬렉션이 진행 중이면 대리블록에 있는 페이지를 우선적으로 할당한다. 이때 2~6줄에서 희생블록의 유효 페이지가 존재한다면 대리블록으로 이주시킨다. 이 작업은 무효 페이지가 나올 때까지 계속된다. 모두 완료되면 희생블록의 번호와 페이지 번호를 반환한다. 가비지 컬렉션이 수행중이 아니라면 10~11줄에서 가용 페이지가 존재하는 블록을 찾아 해당 블록 번호와 페이지 번호를 반환한다.

IV. 실험환경 및 평가

실제 낸드 플래시 메모리가 탑재되어 있는 실험 장치에 B+ 트리를 구현하고 제안하는 가비지 컬렉션 기법을 적용하여 성능평가를 실시하였다. 실험 장치로는 'Cubieboard 3'을 사용하였다[13,14]. 실험 장치의 상세사양은 표 1과 같다.

표 1. 실험 장치의 상세사양
Table 1. Specification of testbed

CPU		AllWinner A20 Cortex™-A7
RAM		2GB DDR3@480MHZ
NAND flash memory		H27UCG8T2ATR-BC
	Capacity	8GB
	Block count	4096
	Page count per block	256
	Page size	8KB
	Bock erase time	5ms
	Page write time	1500us
	Page read time	211us

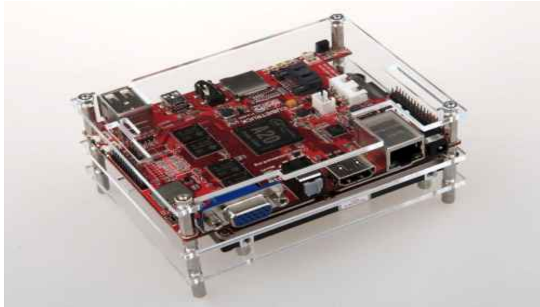


그림 8. 실험 장치
Fig 8. Testbed

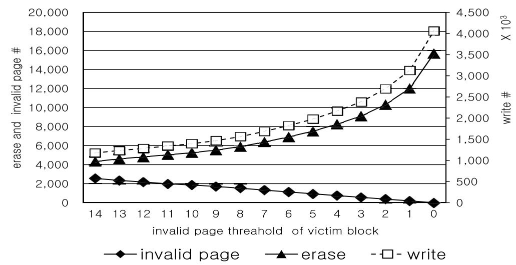


그림 9. 희생블록의 무효 페이지 임계값과 I/O 횟수
Fig 9. The number of I/O and invalid pages threshold of victim block

그림 8은 실험 환경에 사용하였던 실험 장치이다. 실험 장치의 낸드 플래시 메모리는 8GB를 지원하나 불량블록 발생 등의 상황에 대비하기 위해 여유블록을 두고 4096개의 블록 중에 2048+1개의 블록을 실험대상에 포함하였다. 블록들 중에 1개의 블록은 대리블록으로 사용하였다. 실험환경은 리눅스 Lubuntu 12.10을 사용하였으며 낸드 플래시 메모리 제어 함수는 커널 3.4.79버전에서 시스템 호출을 사용하여 구현하였다. B+ 트리에서 키를 삽입할 때 키의 값은 무작위로 선정하였다. 가비지 컬렉션 수행을 위한 희생블록 선정은 탐욕 알고리즘(greedy algorithm)을 사용하였다[15].

제안 기법이 적용된 B+ 트리의 전체 성능을 고려하기 위해 희생블록의 임계값 설정을 위한 실험을 먼저 수행하였다. 희생블록의 임계값은 정해진 무효 페이지의 개수가 되며, 가비지 컬렉션은 무효 페이지의 개수가 임계값 이하인 블록들을 희생블록으로 선정하지 않는다. 그림 9는 희생블록의 무효 페이지 임계값에 따른 지우기 및 쓰기 횟수이다. 임계값이 작을수록 사용할 수 있는 페이지는 많아지나 성능이 급격하게 떨어진다는 것을 알 수 있다. 이런 사실을 통해 B+ 트리의 전체 성능을 고려하기 위해 적절한 임계값의 설정이 필요하다. 임계값은 가용 공간이 적어질수록 레코드의 연산 시간이 높아지는 것을 적당한 시점에 방지하여 최소 연산 시간을 보장받기 위해 활용될 수 있다. 또한 시스템 관리자에게 경고를 보내어 확장이 필요하다는 것을 알릴 수 있다. 그래서 해당 시스템에 따라 임계값을 다르게 설정할 수 있다. 실험환경에서는 제안 기법의 최소 연산 시간과 실험의 전체 소요 시간을 위해 임계값을 8로 설정하였다.

그림 10은 각 상황에서 B+ 트리의 구성에 사용된 무효 페이지 개수, 유효 페이지 개수, 입력된 키의 개수를 보여주고 있다. 4가지의 상황에서 측정하였으며 각 상황은 가비지 컬렉션을 수행하지 않은 때(without GC), 무효블록만 사용하는 가비지 컬렉션(GC using invalid block), 탐욕 알고리즘을 사용하는 일반 가비지 컬렉션(GAGC: Greedy Algorithm Garbage Collection), 제안 기법(PBGC: Proxy Block allocated Garbage Collection)으로 구분된다. 그림 10에서 B+ 트리의 차수는 5로 설정하였으며, 키의 값은 무작위로 입력하였다. 각 기법을 비교한 결과 제안 기법이 다른 기법보다 최소 약 73% 이상의 키를 추가로 삽입하여 트리를 구성할 수 있었으며, 낸드 플래시 메모리의 약 96%를 활용하여 트리를 구성할 수 있었다.

그림 10에서 무효블록만 사용하는 가비지 컬렉션은 희생블록에 유효 페이지가 없기 때문에 페이지 이주 작업이 없고 트리의 재구성을 발생시키지 않는다. 하지만 이로 인해 희생블록 대상이 제한적이고 공간 확보 성능이 떨어진다. GAGC는 희생블록의 데이터를 이주하는 과정에서 발생하는 트리의 재구성으로 인하여 추가적인 페이지 쓰기 연산이 발생한다. 특히 GAGC는 키가 삽입 될수록 가비지 컬렉션의 수행에서 얻게 되는 가용 페이지의 개수보다 발생하는 무효 페이지의 개수가 많아져 가비지 컬렉션을 더 이상 수행할 수 없는 상황

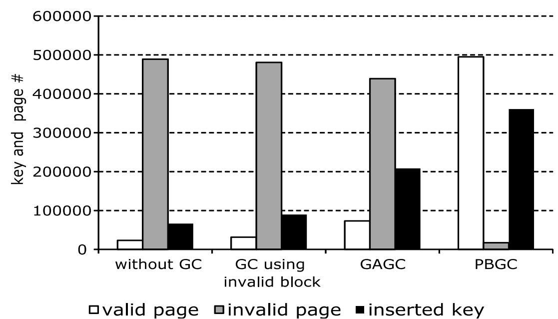
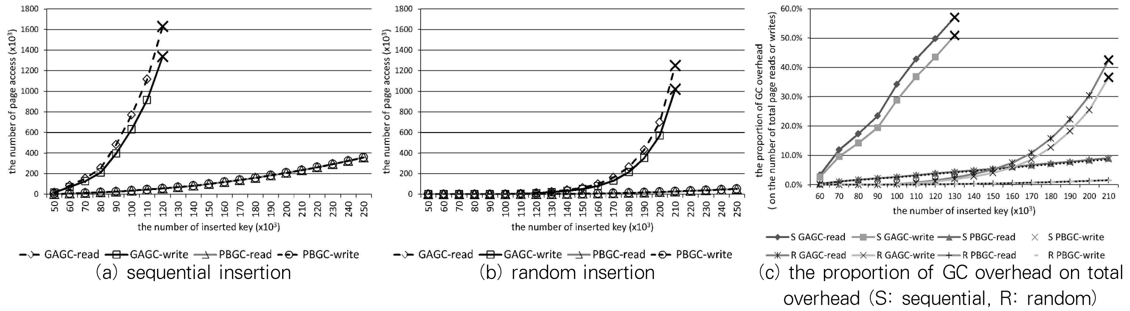


그림 10. B+ 트리에서 무작위 삽입 시 발생하는 각 기법별 유효 페이지, 무효 페이지, 키 개수
Fig 10. The number of keys, valid and invalid pages by each technique on B^+-Tree using random insertion



(a) sequential insertion (b) random insertion (c) the proportion of GC overhead on total overhead (S: sequential, R: random)

그림 11. B+ 트리에서 GAGC와 PBGC에 발생하는 오버헤드
Fig 11. The overhead of GAGC and PBGC on B+-Tree

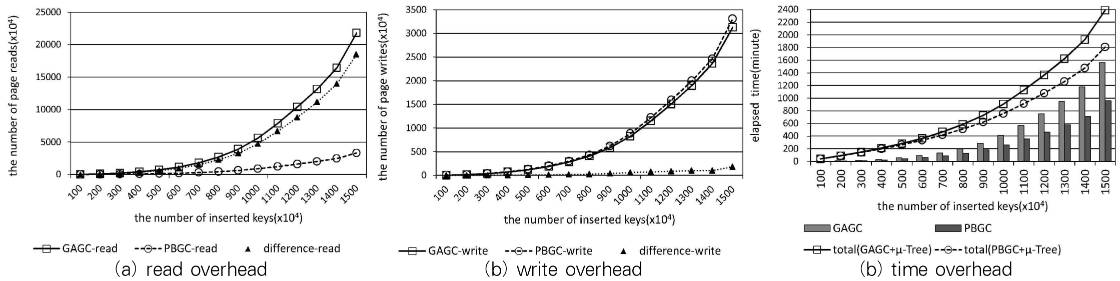
이 발생한다. 식 1은 GAGC에서 유효 페이지(v)의 이주로 인해 트리의 재구성에 발생하는 무효 페이지 개수(I_{count})를 나타내고 있다.

$$0 < l_v \leq H,\ 1 < m < P-1,\ \ I_{count} = \sum_{v=1}^{m} l_v \qquad \text{(식 1)}$$

I_{count}는 이주할 각 유효 페이지에 해당하는 노드의 트리 레벨(l_v)과 희생블록의 유효 페이지의 개수(m)에 따라 달라진다. l_v은 트리의 높이(H)를 초과할 수 없으며, m은 희생블록 선정 알고리즘에 의하여 '블록 당 페이지 수(P) -1' 미만이다. '$I_{count} > P$' 조건을 만족하면 가비지 컬렉션을 수행할 수 없다. 그 이유는 가비지 컬렉션의 수행으로 얻는 가용 페이지보다 발생하는 무효 페이지의 수가 더 많기 때문이다. 특히 l_v는 트리의 구조상 말단노드가 대부분이기 때문에 말단노드가 될 확률이 높다. 식 2는 말단노드가 트리를 구성하는 비율(r)을 노드의 차수(f)와 H를 통해 나타낸 것이다. H가 충분히 크다면 노드의 차수가 커질수록 말단노드의 비율이 높아진다.

$$r = \frac{f^H}{\frac{1-f^{H+1}}{1-f}} = \frac{f^H(f-1)}{f^{H+1}-1} \approx \frac{f-1}{f}\ \ , H > 0 \qquad \text{(식 2)}$$

그림 11은 B+ 트리에 GAGC와 PBGC를 적용하였을 때 발생하는 오버헤드를 측정한 것이다. 각 그래프에서 진한 엑스(×)로 표시된 부분은 앞서 설명한 가비지 컬렉션을 수행할 수 없는 조건이 되어 더 이상 측정을 할 수 없었다는 것을 의미한다. (a), (b)는 각각 순차 삽입과 무작위 삽입에서 각 기법별로 가비지 컬렉션 수행에서 발생하는 페이지 읽기와 페이지 쓰기의 횟수를 나타낸 것이다. 동일한 키의 개수가 삽입되었을 때 (a)가 (b)보다 GAGC와 PBGC의 오버헤드가 많고, GAGC에서 입력할 수 있는 키의 개수는 (a)가 (b)보다 적다는 것을 알 수 있다. 이는 B+ 트리의 구조상 순차적인 삽입이 무작위 삽입보다 인덱스 노드에 저장되는 키의 개수가 적어 인덱스 노드의 공간 효율성이 떨어지기 때문이다. 이러한 이유를 자세히 설명하면, 키의 값을 작은 값에서 큰 값으로 순서대로 입력하게 되면 B+ 트리의 가장 오른쪽 노드에 삽입이 발생한다. 이 때 인덱스 노드의 공간이 부족하게 되면 인덱스 노드의 분할 작업을 수행한다. 분할된 2개의 인덱스 노드는 분할되기 전에 저장된 키를 1/2로 나누어 저장한다. 그런 후, 다시 키가 삽입되면 2개의 노드 중에 오른쪽 노드에 삽입되어 결과적으로 대부분의 인덱스 노드는 1/2의 공간에 키를 저장하게 된다. 그래서 순차적인 삽입과 무작위 삽입에서 삽입된 키의 개수가 동일하다고 하여도 B+ 트리에 구성된 인덱스 노드의 개수는 순차적인 삽입이 무작위 삽입보다 많아진다. (c)는 B+ 트리의 전체 오버헤드에서 순차적인 삽입과 무작위 삽입에서 발생하는 GAGC와 PBGC의 오버헤드의 비율을 나타내고 있다. 각각의 오버헤드 비율에서 읽기는 전체 발생하는 페이지 읽기 횟수에서 각 가비지 컬렉션의 페이지 읽기 발생 비율을 의미하고 쓰기는 전체 발생하는 페이지 쓰기 횟수에서 각 가비지 컬렉션의 페이지 쓰기 발생 비율을 의미한다. 동일한 개수의 키가 삽입되었을 때 전체 발생하는 오버헤드에서 PBGC의 오버헤드 비율이 GAGC의 오버헤드 비율보다 현저하게 낮다는 것을 알 수 있다. (a), (b), (c)에서 PBGC는 읽기 오버헤드와 쓰기 오버헤드는 동일하지만 GAGC는 읽기 오버헤드가 쓰기 오버헤드보다 많다. 그 이유는 GAGC가 페



(a) read overhead (b) write overhead (b) time overhead

그림 12. μ-Tree에서 GAGC와 PBGC에 발생하는 오버헤드
Fig 12. The overhead of GAGC and PBGC on μ-Tree

이지 이주를 수행할 때, 해당 페이지에 저장되어 있는 인덱스 노드의 위치를 파악하기 위해 트리를 탐색하고 이주가 완료되면 변경된 페이지의 위치로 인해 업데이트 파생문제가 발생한다. 이 때, 페이지 읽기와 페이지 쓰기 횟수는 업데이트 파생문제의 특징을 고려할 때 동일해야 하지만 여러 개의 페이지 이주로 다수의 업데이트 파생문제가 발생하면 갱신되는 상위 노드들은 중복되는 경우가 있다. 그래서 이들의 쓰기 횟수를 줄이기 위해 해당 인덱스 노드들은 페이지 이주 작업이 모두 완료되는 마지막 시점에서 쓰기를 수행하기 때문이다. 결국 GAGC가 페이지를 이주할 때 업데이트 파생문제에서 발생하는 페이지 쓰기 작업을 감소시키는 경우가 발생한다. 예를 들면, 항상 페이지 이주에서 갱신이 중복되는 인덱스 노드는 최상위 인덱스 노드가 된다. 하지만 그림 11에서의 결과처럼 GAGC의 업데이트 파생문제의 페이지 쓰기 감소 효과는 크지 않다는 것을 알 수 있다.

그림 12는 μ-Tree에서 GAGC와 PBGC의 오버헤드를 나타내고 있다. 구현된 μ-Tree의 최대 차수는 128로 설정하였으며, 실험시간을 고려하여 사용하는 블록의 개수를 1024개로 제한하였다. (a), (b)의 오버헤드는 가비지 컬렉션 수행에서 발생하는 페이지 읽기 및 쓰기 횟수를 의미한다. (a)에서 GAGC가 PBGC보다 읽기 오버헤드가 많이 발생하였으며, 동일한 키의 개수가 삽입되었을 때 PBGC는 GAGC의 오버헤드를 최대 약 85%까지 줄일 수 있었다. GAGC의 읽기 오버헤드가 많은 이유는 μ-Tree도 B+ 트리처럼 이주된 페이지의 위치를 반영하고 이주할 페이지에 저장되어 있는 인덱스 노드를 확인하기 위해 트리 탐색이 필요하기 때문이다. (b)에서 PBGC가 GAGC보다 쓰기 오버헤드가 많이 발생하였으며, 동일한 키의 개수가 삽입되었을 때 PBGC가 GAGC의 오버헤드보다 최대 약 6% 많았다. 그 이유는 페이지가 이주할 때, μ-Tree는 페이지의 위치가 변경되어도 업데이트 파생 문제를 발생하지 않아 한 번의 페이지 쓰기로 각 페이지를 이주할 수 있다. 그리고 μ-Tree의 구조상 이주해야 하는 페이지들에 저장되어 있는 인덱스 노드가 중복 저장되는 경우가 있고 GAGC는 이런 정보를 트리의 탐색으로 알 수 있어 페이지 쓰기 횟수를 일부 줄일 수 있기 때문이다. 그러나 μ-Tree도 B+ 트리처럼 대부분의 페이지에는 말단노드가 저장되어 있기 때문에 이러한 이득은 적게 된다. (c)는 PBGC와 GAGC의 시간 오버헤드를 나타낸 것이다. 선형그래프는 각 기법을 적용하였을 때, 레코드 연산과 가비지 컬렉션 수행을 포함하는 전체적인 시간 오버헤드를 의미한다. 막대그래프는 각 기법별로 가비지 컬렉션의 수행시간만을 나타낸 것이다. 가비지 컬렉션의 수행시간만을 고려할 때, PBGC는 GAGC의 시간 오버헤드를 최대 약 39% 줄일 수 있었다. 결국 (b)에서 PBGC가 GAGC보다 쓰기 오버헤드가 많더라도 (a)에서 PBGC가 읽기 오버헤드를 현저하게 낮추어 (c)에서 전체적인 오버헤드는 PBGC가 적다는 것을 알 수 있다.

실험환경에서는 제안된 기법인 PBGC와 일반적인 가비지 컬렉션 기법이라고 할 수 있는 GAGC를 B+ 트리와 μ-Tree에 적용하고 평가하였다. 결론적으로 PBGC는 B+ 트리에서 이용공간의 효율성을 높여 삽입되는 키의 개수가 GAGC보다 약 73% 많았으며, B+ 트리의 가비지 컬렉션 성능을 고려하여 임계값을 설정하여도 플래시 메모리의 전체공간의 약 96%까지 사용할 수 있었다. μ-Tree에서는 PBGC가 GAGC의 시간 오버헤드를 최대 약 39% 줄일 수 있었다.

V. 결 론

FTL을 탑재하기 힘든 저사양의 임베디드 장치에서 낸드 플래시 메모리에 특화된 B+ 트리 또는 이와 유사한 인덱스 트리 구조들이 연구되었다. 하지만 이런 기법들은 일반적인 가비지 컬렉션을 수행할 경우 낸드 플래시 메모리의 물리적인 페이지 위치를 변경하며, 이는 트리의 재구성을 유발시켜 성능 저하가 발생한다. 이런 문제를 해결하기 위해 본 논문에서

는 BIT와 대리블록을 이용한 새로운 가비지 컬렉션 기법을 제안하였다. 제안된 기법은 BIT와 대리블록을 이용하여 트리의 재구성을 발생시키지 않았다. 일반적인 가비지 컬렉션으로 발생되는 블록 위치의 변경은 블록 단위 주소 매핑을 수행하는 BIT를 사용하여 해결하였으며, 페이지 위치의 변경은 대리블록과 희생블록을 하나의 논리적인 가상블록으로 설정하여 해결하였다. 낸드 플래시 메모리가 장착되어 있는 장치에 실험환경을 구축하여 제안한 기법을 적용한 결과, 트리의 재구성에 대한 추가적인 비용이 없었으며 B+ 트리에서 제안된 기법의 삽입된 키 개수가 GAGC보다 약 73% 많았으며, μ-Tree에서 제안된 기법의 시간 오버헤드가 GAGC보다 약 39% 적었다.

본 논문에서 제안한 기법은 이용공간의 효율성을 위해 트리의 재구성이 없이 무효 페이지를 수거하는 방법에 초점을 두었다. 하지만 가비지 컬렉션에서는 마모도 평준화 기법에 대한 고려도 필요하기 때문에 제안한 기법에 적용할 수 있는 마모도 평준화 기법을 향후 연구과제로 남긴다.

REFERENCES

〔1〕 Kwon, Se Jin, et al. "FTL algorithms for NAND-type flash memories." Design Automation for Embedded Systems, Vol. 15. No. 3-4, pp. 191-224, Dec. 2011.

〔2〕 `Ma, Dongzhe, Jianhua Feng, and Guoliang Li. "A survey of address translation technologies for flash memories." ACM Computing Surveys (CSUR), Vol. 46, No. 3, pp. 1-39, Jan. 2014.

〔3〕 Fang, H., et al. "An Adaptive Endurance-Aware B^+ -Tree for Flash Memory Storage Systems." Computers, IEEE Transactions, Vol. 63, No. 11, pp. 2661 - 2673, Nov. 2014.

〔4〕 Kang, Dongwon, et al. "μ-Tree: an ordered index structure for NAND flash memory." Proceedings of the 7th ACM & IEEE international conference on Embedded software. ACM, pp. 144-153, Sep. 2007.

〔5〕 Ahn, Jung-Sang, et al. "μ*-Tree: An Ordered Index Structure for NAND Flash Memory with Adaptive Page Layout Scheme." Computers, IEEE Transactions, Vol. 62, No. 4, pp. 784-797, Apr. 2013.

〔6〕 Na, Gap-Joo, Bongki Moon, and Sang-Won Lee. "In-page logging B^+-tree for flash memory." Database Systems for Advanced Applications. Springer Berlin Heidelberg, pp. 755-758, Apr. 2009.

〔7〕 Na, Gap-Joo, Sang-Won Lee, and Bongki Moon. "Dynamic in-page logging for flash-aware B^+-tree index." Proceedings of the 18th ACM conference on Information and knowledge management. ACM, pp. 1485-1488, Nov. 2009.

〔8〕 Lee, Sungjin, and Jihong Kim. "Improving Performance and Capacity of Flash Storage Devices by Exploiting Heterogeneity of MLC Flash Memory." Computers, IEEE Transactions, Vol. 63, No. 10, pp. 2445-2458, Oct. 2014.

〔9〕 H.-W. Fang, M.-Y. Yeh, and T.-W. Kuo, "MLC-Flash-Friendly logging and recovery for databases." in Proc.ACM Symp. Appl. Comput. (SAC), pp. 1541-1546, Mar. 2013.

〔10〕 Li, Xiang, Zhou Da, and Xiaofeng Meng. "A new dynamic hash index for flash-based storage." WAIM'08. The Ninth International Conference on. IEEE, pp. 93-98, Jul. 2008.

〔11〕 Chang, Yuan-Hao, Jen-Wei Hsieh, and Tei-Wei Kuo. "Improving flash wear-leveling by proactively moving static data." Computers, IEEE Transactions, Vol. 59, No. 1, pp. 53-65, Jan. 2010.

〔12〕 Chung, Ching-Che, Duo Sheng, and Ning-Mi Hsueh. "A high-performance wear-leveling algorithm for flash memory system." IEICE Electronics Express, Vol. 9, No. 24, pp. 1874-1880, Dec. 2012.

〔13〕 Hardware spec, *http://docs.cubieboard.org/tutorials/cubietruck/start.*

〔14〕 Product Feature, *H27UCG8T2ATR-BC Series 64Gb(8192M × 8bit) Legacy MLC NAND flash.*

〔15〕 Wu, Michael, and Willy Zwaenepoel. "eNVy: a non-volatile, main memory storage system." ACM SigPlan Notices. ACM, Vol. 29, No. 11, pp. 86-97, Nov. 1994.

저 자 소 개

김 선 환

2006: 영남대학교
컴퓨터공학과 공학사.
2009: 영남대학교
컴퓨터공학과 공학석사.
현 재: 영남대학교
컴퓨터공학과 박사과정.
관심분야: 임베디드 시스템,
무선 센서 네트워크
Email : amexist@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