복수개의 페이지를 이용한 μ-Tree

*김태진, 신동군
성균관대학교 정보통신공학부
e-mail : taejin1999@hanmail.net, dongkun@skku.edu

Multi page-based μ-Tree

*Taejin Kim, Dongkun Shin
School of Information and Communication Engineering
Sungkyunkwan University

요 약

NAND 플래시 메모리가 임베디드 시스템용 data 저장장치로 널리 사용되면서 많은 파일 시스템과 데이터 베이스 관리 시스템들이 이를 기반으로 만들어 지고 있다. 이들은 거대한 양의 자료들로부터 특정한 item을 빨리 찾기 위해 index 구조를 필요로 하게 된다. 이 논문에서는 이러한 index 구조 중 하나로 leaf에서 root 까지의 경로 상의 노드들을 하나의 페이지에 저장하는 μ-Tree를 개선시킨 Multi page μ-Tree를 제안한다. Multi page μ-Tree는 저장단위를 1개의 페이지가 아닌 복수개의 페이지로 확대하여 기존의 μ-Tree가 가지고 있는 공간 낭비 문제를 개선하고 트리 높이 증가를 지연시킬 수 있다.

1. 서 론

플래시 메모리는 PMP, PDA, 디지털 카메라, 휴대폰 등과 같은 많은 이동 기기들에 저장장치로 널리 적용되고 있다. 그 주된 이유는 플래시 메모리의 선천적인 특성 즉, 비휘발성, 작고 가벼운 물리적 특징, 저전력소모, 안정적 상태의 신뢰도 때문이다.

저장장치에 필요한 index 구조중 가장 널리 사용되는 B+ Tree의 wandering tree 문제를 해결하기 위해 강동원 등은 한 페이지에 여러 개의 노드를 저장하는 μ-Tree를 제안했다.[3] μ-Tree는 갱신이 발생한 leaf 노드에서부터 root 노드까지 그 leaf 노드의 갱신으로 영향을 받은 모든 노드들을 하나의 page에 저장하므로 갱신에 필요한 쓰기 연산은 Tree의 높이에 관계없이 항상 한번이다.

이 논문에서는 이러한 트리구조를 개선시킨 복수개의 페이지를 이용한 μ-Tree를 소개한다. 하나의 페이지를 저장단위로 하는 기존의 μ-Tree와는 달리 복수개의 페이지를 이용한 μ-Tree에서는 노드들을 저장하는데 2개 이상의 페이지를 할당하여 사용하게 된다. 따라서 leaf 노드의 크기 역시 기존 μ-Tree에 비해 2배, 4배,... 등으로 늘어나게 되고 이는 공간 활용도를 더욱 높이는 결과를 가져다준다.

2. 배경 지식과 연구동기

2.1 μ-Tree

디렉토리 엔트리들을 구조화 하고 최근 접근된 파일의 위치를 찾는데 B+ Tree를 사용하는 파일 시스템들이 있다. 하지만 FTL의 부재로 인해 leaf 노드의 갱신작업은 root 노드까지 모든 index 노드들의 갱신을 야기한다. 이런 방식으로 트리를 갱신하는 방법을 wandering tree라고 한다[2]. 이런 방식과 달리 트리상의 어떠한 노드라도 갱신될 때 오직 한 번의 플래시 쓰기 연산이 필요한 index 구조를 μ-Tree라고 한다. μ-Tree는 기본적으로 root에서 leaf까지 갱신된 모든 노드들을 하나의 플래시 메모리 페이지에 저장한다.[3]

μ-Tree에서의 page 배치는 두 가지 이점을 가진다. 첫째, 이러한 배치가 μ-Tree의 높이에 관계없이 각 레벨에서의 동일한 노드의 크기를 보장하기 때문에 루트를 제외한 존재하는 모든 노드들을 그대로 다시 사용할 수 있다. 둘째, μ-Tree의 배치개념은 적어도 페이지의 절반을 leaf 노드에 할당한다. leaf 노드의 크기는 공간 이용에 상당한 영향을 가진다. 좀 더 구체적으로 말하면, leaf 노드의 크기가 커질수록 공간 낭비를 낮출 수 있다.[3]

2.2 연구 동기

μ-Tree는 이러한 장점에도 불구하고 몇 가지 개선점을 가지고 있는데 첫 번째로 트리의 전체 구조를 한 페이지 안에 담기 때문에 페이지의 크기가 정해지면 상위 레벨 노드로 올라갈수록 그 크기는 하위 레벨 노드의 절반으로 정해져 있기 때문에 트리가 가질 수 있는 최대 높이 역시 그 이하로 제한된다. 두 번째로 μ-Tree는 leaf 노드의 크기를 늘릴수록 공간 낭비가 줄어들어 공간 활용도를 높일 수 있다. 하지만 제안된 μ-Tree에서는 한 페이지에서 leaf 노드를 제외한 공간을 index 노드용으로 사용하기 때문에 index 노드를 갖지 않는 페이지들은 페이지 크기의 절반을 낭비하게 되고 따라서 공간 활용도를 개선하는데 어려움이 있다. 이 논문에서는 μ-Tree에 복수개의 페이지를 저장단위로 하여 공간 활용도 및 최대 트리 높이를 증가시킬 수 있는 방식을 제안한다.

3. Multi page μ-Tree

Multi page μ-Tree와 μ-Tree는 저장단위를 몇 개의 페

이지로 하느냐를 제외하고는 큰 차이가 없다. 복수개의 페이지를 저장단위로 한다는 것은 트리의 내용을 하나의 페이지가 아닌 여러 개의 페이지에 걸쳐서 저장한다는 것이다.

만약 저장 단위가 2 페이지이면 전체 저장단위 크기의 반을 leaf 노드로 하는 정책으로 인해 하나의 페이지가 leaf 노드를 저장하고 또 다른 하나는 index 노드만을 저장하게 된다. 저장단위가 4 페이지로 증가하면 leaf 노드를 저장하는 페이지의 개수는 2개가 되고 index 노드를 저장하는 페이지의 개수 역시 2개로 늘어난다. 이는 원래 방식에 비해 leaf 노드의 크기를 각각 2배, 4배 늘리는 효과를 가져온다. 따라서 leaf 노드에 저장할 수 있는 데이터의 개수가 많아지므로 트리 전체 높이의 증가를 지연시킬 수 있다. 또한 트리의 전체구조를 여러 페이지에 걸쳐 저장할 수 있으므로 트리가 가질 수 있는 최대 높이 역시 원래 방식에 비해 각각 2배, 4배 늘어나게 된다.

추가적으로 이러한 방식은 하나의 페이지가 leaf 노드를 저장하므로, index 노드를 가지지 않는 페이지의 경우 페이지 절반 크기의 leaf 노드만을 가지면서 나머지 절반은 낭비되던 기존의 μ-Tree에 비해 공간의 낭비가 거의 없다. 오히려 공간 활용도 면에서는 B+ Tree와 비슷한 성능을 보인다. 쓰기연산 횟수 측면에서 살펴보면 μ-Tree는 단 1번의 쓰기연산이 필요한데 반해 Multi page μ-Tree는 leaf 노드가 갱신된 후 이를 새로운 페이지에 쓰고 이전 leaf 노드를 가리키는 index 노드 역시 수정 되어야 한다. 하지만 이후 root 노드까지의 수정은 1번의 쓰기연산으로 가능하므로 총 2회(leaf page, index page)의 쓰기연산이 필요하다. 이는 Tree의 높이만큼의 쓰기연산이 필요했던 B+ Tree에 비해 쓰기연산 횟수 측면에서 더 나은 성능을 보인다. 즉, Multi page μ-Tree는 기존의 μ-Tree와 B+ Tree의 장점을 모아 놓은 index 구조이다.

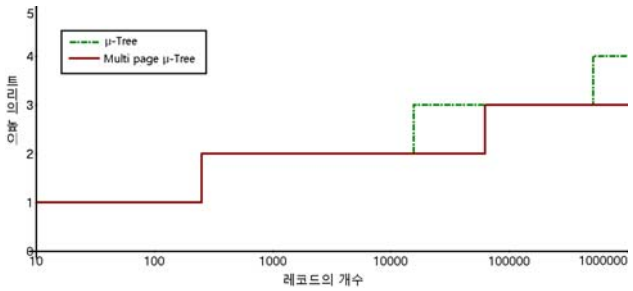
4. 실험적 평가

4.1. 평가 방법

이 논문에서 제안된 Multi page μ-Tree를 사용하여 레코드의 개수에 따른 트리의 높이 변화 및 삽입, 검색, 삭제 연산 등을 수행 했을 때 수행된 flash read, flash write, flash delete 횟수를 기준으로 평가하였다. Multi page μ-Tree의 경우 2 page를 할당하였다. 평가에서 사용된 페이지크기는 2KB이고 블록의 크기는 256KB를 사용하였다. 평가에 사용된 입력은 정수를 키로 갖는 레코드들을 사용했으며 다음과 같은 상황 하에서 진행되었다. 먼저 약 십만 개의 레코드가 삽입된 트리에 약 천 개의 숫자를 검색하고 같은 개수의 숫자를 트리에서 지우고 난 뒤 무작위로 생성된 키를 갖는 레코드를 다시 트리에 삽입하는 순서로 진행된다. 비교를 위해 기존의 μ-Tree에서도 같은 입력을 사용하여 평가를 진행하였다.

4.2. 평가 결과

그림 1은 레코드의 숫자가 늘어감에 따라 기존의 μ-Tree와 Multi page μ-Tree의 높이 변화를 보여준다. 윗절에서 기술한대로 Multi page μ-Tree는 트리의 높이 증가를 지연시켜서 거쳐야 할 노드의 수를 줄여준다. 표 1은 트



[그림 1] μ-Tree와 Multi page μ-Tree의 높이 비교

	검색		삽입		삭제	
	μ-Tree	Multi	μ-Tree	Multi	μ-Tree	Multi
read	4.00	3.00	5.43	4.04	5.45	4.05
write	0.00	0.00	1.44	1.85	1.43	1.84
access cost	0.311 (ms)	0.233 (ms)	0.786 (ms)	0.781 (ms)	0.785 (ms)	0.780 (ms)

[표 1] 작업 수행 시 호출되는 평균 flash 연산 횟수

리의 레코드 검색, 새로운 레코드 삽입, 기존 레코드 삭제 연산 수행 시 flash page를 read / write 한 평균 횟수를 나타낸다. 성능의 지표가 되는 access cost를 살펴보면 검색, 삽입, 삭제 연산에 대해 각각 25%, 5%, 6%의 성능향상을 보임을 볼 수 있다. access cost는 read latency : 77.8μs, write latency : 252.8μs를 갖는 large block SLC NAND flash memory의 특성을 기반으로 하여 계산되었다.[4]

모든 연산에 대해 flash read 연산의 횟수는 높이가 낮은 Multi page μ-Tree에서 더 적다. 하지만 flash write 연산 횟수는 기존 μ-Tree의 횟수가 더 적은데 이는 Multi page μ-Tree에서는 변경된 leaf와 이를 가리키는 index노드들을 따로 write해야 하기 때문이다. 하지만 Multi page μ-Tree의 노드들의 크기가 같은 레벨에서의 μ-Tree보다 더 크기 때문에 부모노드로 키를 추가 시키거나 부모노드의 키를 삭제하는 등의 flash write 연산을 유발시키는 횟수는 상대적으로 더 적다.

5. 결 론

이 논문에서는 root에서 leaf 노드까지 경로 상의 노드들을 하나의 플래시 페이지에 저장하는 μ-Tree를 개선하여 복수개의 페이지에 노드들을 저장하는 Multi page μ-Tree를 제안하였다. 3장에 나타난 대로 이러한 방식은 B+ Tree의 공간 활용성과 μ-Tree의 갱신 연산 시 효율성을 모두 가지는 index 구조임을 확인할 수 있다.

6. 참고 문헌

[1] Aleph One Limited. Yet Another Flash File System (YAFFS). http://www.aleph1.co.uk/yaffs

[2] A. B. Bityutski. JFFS3 design issues. http://www.linux-mtd.infradead.org

[3] D. Kang et al. μ-Tree : An Ordered Index Structure for NAND Flash Memory. EMSOFT'07

[4] Samsung Elec. 2Gx8 Bit NAND Flash Memory (K9GAG08U0M-P). 2006.