플래쉬 메모리에서 Shadow 버전을 이용한
B-트리 인덱스 관리

온영오  조병래
영남대학교 컴퓨터공학과
(onda@, hrcho@yu.ac.kr)

A B-Tree Management Scheme Exploiting Shadow Version on Flash Memory
Kyungho Ohn  Haengrae Cho
Dept. of Computer Engineering, Yeungnam University

요 약
플래쉬 메모리의 비휘발성, 저전력, 경량, 내구성 등의 장점으로 인해, PDA나 스마트카드, 후드폰, 혜대폰, 혜대폰 등은 고장 수명 기등과 같은 이동 컴퓨터 장치의 저장소로 많이 사용되고 있다. 최근 들어 대용량의 플래쉬 메모리가 출시되고 휴대용 컴퓨터용 B-트리와 같은 인덱스 기법이 요구되고 있다. 하지만, 현재 사용되고 있는 NAND 플래쉬 메모리는 기존의 하드 디스크와 같은 메모리 특성에 적합하지 않다. 본 논문에서는 데이터에 바이너리 데이터를 저장하는 영역에서 16 바이트의 데이터로 이루어진 데이터의 B-트리와 같은 인덱스 기법의 특성을 이용하였다. 본 논문에서는 shadow 버전을 이용한 플래쉬 메모리의 B-트리 인덱스 기법을 제안한다.

1. 서 론
플래쉬 메모리는 비휘발성, 저전력, 경량, 내구성 등의 장점으로 인해 PDA나 스마트카드, 휴대폰, 혜대폰 등은 고장 수명 기등과 같은 이동 컴퓨터 장치의 저장소로 많이 사용되고 있다. 플래쉬 메모리의 제조사들이 대용량의 플래쉬 메모리를 출시하고 발표하는 B-트리와 같은 인덱스 기법이 요구되고 있다. 하지만, 현재 사용되고 있는 NAND 플래쉬 메모리의 제조사들이 제시하는 특성에 적합하지 않다. 본 논문에서는 데이터에 바이너리 데이터를 저장하는 영역에서 16 바이트의 데이터로 이루어진 데이터의 B-트리와 같은 인덱스 기법의 특성을 이용하였다. 본 논문에서는 shadow 버전을 이용한 플래쉬 메모리의 B-트리 인덱스 기법을 제안한다.

---

그림 1. NAND형 플래쉬 메모리 구조

---

2. 관련 연구

B-트리 인덱스는 대용량의 데이터를 효율적으로 액세스하기 위해 많이 사용되며, 데이터의 삽입/삭제에 의해 B-트리 노드는 반복하게 변경된다. 일반적으로 체계적인 B-트리를 RAM에 두는 것은 불가능하기 때문에 생성된 B-트리 노드들은 플래시 메모리에 저장해야할 경우가 발생하고 이를 위해 FTL의 쓰기연산이 수행된다. 플래시 메모리에서 섹터 단위의 쓰기연산은 비어있는 섹터 (free sector)를 빠르게 소모화하여 플랫에서 더 이상 유효하지 않은 섹터들 (dead sector)을 찾아서 유효한 섹터 (live sector)들만 새로운 복록에 저장하여 반 섹터를 생성 (garbage collection)하는 별도의 연산이 반복으로 발생하게 된다. 결국 플래시 메모리의 수행이 단축되고 추가적으로 이동 컨트롤의 장치의 전력 소모도 증가하게 된다. 이러한 문제점을 해결하기 위해 제안된 플래시 메모리에 대한 기존의 B-트리 인덱스 기법에는 BFTL[4]과 BOF(B-Tree on flash memory)[5]가 있다.

그림 3는 BFTL 기법의 구조를 나타낸다. BFTL 기법은 RAM에서 유효비퍼 (reservation buffer)의 노드 번환 테이블 (node translation table)을 유지한다. B-트리 액세스가 필요한 서비스가 요청되면 BFTL에 의해 플래시 메모리에 대한 액세스 요청으로 사전되어 FTL에 전송된다. 이에 요청된 서비스에 의해 삽입, 삭제, 또는 생성된 레코드들 (dirty records)은 유효비퍼에 기록된다. 유효비파가 가득 찼을 때, BFTL 기법의 완료정책 (commit policy)은 모든 생성된 레코드들에 대해 인덱스 유니트 (index unit)를 생성하여 섹터에 저장한다. BFTL 기법의 완료정책은 다른 B-트리 노드에 속하는 인덱스 유니트들을 동일한 섹터에 저장하는 것을 허용하여 플래시 메모리에 쓰는 섹터의 수를 최소화한다. 그림 3은 BFTL 기법의 완료정책을 위해 유효비퍼의 데이터와 인덱스 유니트가 관리되는 것을 나타낸다.

BFTL 기법은 하나의 B-트리 노드에 속하는 인덱스 유니트들이 여러 개의 섹터에 흩어져 저장될 수 있기 때문에 B-트리를 순회하기 위해서는 B-트리 노드에 속하는 모든 인덱스 유니트를 유지하기 위해 노드 레벨이 필요하다. 정확한 B-트리 순회를 위해 노드 번환 테이블은 인덱스 유니트를 저장한 섹터들은 연결 리스트로 유지를 하고, 연결된 섹터 수가 미리 정의된 리스트의 최대값보다 클 경우 해당 B-트리 노드와 인덱스 유니트들을 병합하여 B-트리 노드를 재구성한다. BFTL 기법은 섹터의 쓰기연산을 최소화하였지만, 노드 번환 테이블에서의 연결 리스트가 길어지고 이에 따라 액세스 여러 개의 섹터들에 흩어져 있는 인덱스 유니트를 찾아야 하므로 검색비용이 증가한다는 단점을 가지고 있다.

새로운 레코드가 삽입될 때, B-트리 노드와 인덱스 유니트를 추가할 공간이 부족하면 노드 분할 (split)이 수행된다. 노드 분할은 새로운 노드를 생성하여 분할이 필요한 노드와 새로운 노드에 인덱스 유니트를 나누어 저장하고 부모 노드에 구분자를 삽입한다. BFTL 기법의 노드 분할은 분할된 두 노드의 인덱스 유니트들을 하나의 섹터에 저장한다. 그림 4는 BFTL 기법에서 노드 분할을 나타낸다. 그림 4의 (a)에서 노드 B1에 새로운 인덱스 유니트가 삽입되어 노드 B1, B2로 분할이 수행될 때 노드 B2가 저장되어 있던 농리 주소 11인 섹터에 분할된 노드들을 담아 보낸다. BFTL 기법의 노드 분할은 두개의 노드를 저장하기 위해 두개의 섹터를 이용하는 것을 하나로 줄여서 테이온이의 수를 감소 시킬 수 있다. 하지만, 그림 4의 (b)와 같이 분할된 노드가 두 분할된 경우 두개의 분할된 노드는 새로운 섹터에 흩어 쓰기고 기존의 섹터에 대해서는 무효화와 같은 처리과정이 필요하다. 즉, 분할된 노드 B1의 둘 번할될 경우 농리 주소 13의 새로운 섹터에 분할된 두 노드 B11, B12를 저장하고, 농리 주소 11의 섹터에 있는 노드 B1에 대해서는 무효화를 해야 하지만, BFTL 기법에서는 그러한 과정이 존재하지 않는다. 이 경우 노드 B2가 분할되어 새로운 섹터에 저장되더라도

그림 2. BFTL 구조

그림 4. BFTL의 노드 분할
도 논리 주소 11의 섹터는 무효화될 수 없기 때문에 쓰기 값이 저장된 섹터가 늘어나는 문제점이 발생한다.

BOF 기법은 BFTL 기법의 이러한 문제가 극복하기 위해 제안된 알고리즘으로 BFTL 기법과 같이 유보바리지를 사용하고 인덱스 유닛을 생성하지만, 동일한 B-트리 노드에 속하는 인덱스 유닛들은 하나의 섹터에 저장함으로써 하나의 B-트리 노드를 액세스하기 위해 하나의 섹터만 읽기연산을 수행하면 된다. 이를 위해 BOF 기법은 유보바리지를 생성한 인덱스 유닛 읽기어배치에 오클레에 저장할 때 플래시 메모리에 저장된 해당 B-트리 노드와 인덱스 유닛의 병합을 수행한다.

BOF 기법은 RAM에서 노드 변경 태이블을 생성하는 오버헤드와 검색비용을 줄일 수 있다. 하지만, 생성된 모든 인덱스 유닛에 대해 플래시 메모리에 저장된 B-트리 노드를 읽고 병합한 결과를 다 플래시 메모리에 저장함으로써 발생될 쓰기연산이 발생한다. 최악의 경우, 유보바리지를 사용하는 모든 레코드들이 모두 다른 B-트리 노드에 속한다면 발생된 레코드의 수만큼 쓰기연산이 발생할 수 있다. 쓰기연산이 증가함에 따라 쓰기 비율은 생성하기 위한 블록 지우기 연산이 반복되고 전력 소모도 증가하게 되며, 결국 플래시 메모리의 수명이 단축된다.

3. Shadow 버전을 이용한 B-트리 관리 기법

BFTL 기법은 생성된 인덱스 유닛은 B-트리 노드에 상관없이 플래시 메모리에 저장한다. 이로 인해 쓰기연산을 줄일 수 있지만, B-트리 노드 재구성 과 검색을 위한 오버헤드가 증가하는 단점이 있다. 또한 노드 분할 기존 섹터에 대한 무효화 처리과정이 필요하지만 BFTL 기법에서는 안급되지 않았다. 이러한 BFTL 기법의 단점을 보완한 BOF 기법은 쓰기연산이 증가하는 단점이 존재한다.

본 절에서는 플래시 메모리에서 shadow 버전(shadow version)을 이용한 B-트리 인덱스 관리 기법을 제안한다. 제안하는 알고리즘은 기본 개념은 B-트리 노드에 shadow 버전을 별도의 별도의 반으로 사용하는 것이므로 shadow 버전으로 저장하는 것이다. 이때 이전 버전을 저장하고 있는 섹터는 즉 무효화 될 수 있고, 노드 분할 레코드의 변경 로직의 기법을 최소화하여 B-트리 인덱스 재구성 간을 단축 할 수 있다. 이를 위해 인덱스 유닛을 저장하고 있는 섹터들에 대해 논리 주소와 해당하는 B-트리 노드의 수로 구성된 인덱스맵(logical index map)을 유지한다고 가정한다.

3.1 담고리즘

플래시 메모리에 정의한 기존의 B-트리 인덱스 관리 기법들은 플래시 메모리에서 B-트리 인덱스를 위한 인덱스 유닛들을 저장한다. 스템이 켜질 때 RAM에서 B-트리 노드를 구성하고 레코드의 삽입, 삭제 또는 수정에 의해 생성되는 인덱스 유닛을 각 기법의 완료정책에 따라 플래시 메모리에 저장한다.

본 논문에서는 제안하는 기법은 플래시 메모리에서 shadow 버전의 B-트리 노드와 인덱스 유닛을 함께 유지하는 것이다. 제안한 기법은 인덱스 유닛의 쓰기연산을 최소화하기 위해 BFTL 기법의 유보바리지를 위한 완료정책을 사용하고, RAM에 캐시되는 B-트리 노드는 분할될 때 플래시 메모리에 shadow 버전으로 저장한다. B-트리 노드가 분할될 때 플래시 메모리에 캐시하기로 하여 각 최근에 저장된 shadow 버전을 임의로 한 노드 변경 태이블의 연결된 섹터에서 인덱스 유닛들을 임의로 B-트리 노드를 구성함으로써 B-트리 재구성 비용을 줄일 수 있다.

RAM에 캐시된 B-트리 노드가 shadow 버전으로 저장될 때 노드 변경 태이블의 연결 리스트는 삭제된다. 또한 해당 노드의 인덱스 유닛을 저장하고 있는 섹터들에서 부분적인 무효화가 필요하다. 예를 들어, 그림 5의 (b)에서 논리 주소 34의 섹터에는 노드 B와 C의 인덱스 유닛들이 저장되어 있다. 이때, 노드 B가 분할되어 저장될 때 노드 B의 인덱스 유닛을 저장하고 있는 연결 리스트는 삭제되고, 논리 주소 42, 34, 100, 53의 섹터에서 노드 B에 해당하는 인덱스 유닛들은 폐기되어야 한다. 이를 위해 개선된 버전의 섹터는 인덱스 유닛이 저장된 B-트리 노드의 수를 표 하고 있는 인덱스맵을 유지한다. 즉, 논리 주소 34에 해당하는 섹터에는 노드 B와 C의 인덱스 유닛들이 저장되어 있으므로 인덱스맵의 34에는 2가 표 된다. 또한 노드 B가 분할되어 플래시 메모리에 저장됨 경우 노드 변경 태이블에서 노드 B에 연결된 리스트를 찾고 논리 주소에 해당하는 인덱스맵의 값을 1 감소 한다. 그리고 노드 C가 분할되어 플래시 메모리에 저장되면 인덱스맵에서 34의 값이 0으로 논리 주소 34에 해당하는 섹터는 무효화된다. 무효화된 섹터는 빈 블록을 생성하는 BFTL의 블록 지우기 연산에 포함되어야 하기 때문에 섹터의 해제에 있는 "valid/obsolete" 비트를 0으로 변경한다.

그리고 유보바리지에는 인덱스 유닛도 shadow 버전에 반영된 것이기 때문에 폐기 가능하지만, 어디서 폐기할 인덱스 유닛을 찾아야 하는 오버헤드가 존재한다. B-트리 노드의 shadow 버전이 저장된 후 유보바리지에 남아있던 인덱스 유닛이 완료정책에 의해 플래시 메모리에 저장될 수 있지만, B-트리 노드의 타임스탬프와 인덱스 유닛의 타임스탬프를 이용하여 인덱스 유닛이 재 반영되는 것을 방지할 수 있다. 그림 6은 B-트리 노드가 분할되어 플래시 메모리에
1. RAM에서 새로운 B-트리 노드를 생성하고 분할하는 노드에서 전체 키들의 항목을 새로운 노드로 옮긴다. 두 노드를 구분할 수 있는 식별자를 부모 노드에 삽입하고 분할된 두 노드를 플래쉬 메모리에 저장한다.
2. 노드 변환 데이터에서 분할된 두 노드의 타임스탬프를 최신의 값으로 변경한다.
3. 노드 변환 데이터에서 변경된 섹터의 년오 주소를 삭제하고 인덱스맵에서 해당 섹터의 값이 감소 한다.
4. 부모 노드에서 구문자 삽입으로 노드 분할이 필요한 경우 부모 노드의 노드 분할을 수행한다.

그림 6. B-트리 노드 분할

저장될 때 노드 변환 데이터와 인덱스맵이 수정되는 과정을 나타낸다. 그림 6의 과정 3에서 노드 변환 데이터의 타임스탬프에 관한 것과 유보버퍼에 존재하는 인덱스 유니트에 대한 최적화 방안은 3.2절에서 설명한다.

3.2 오버헤드 분석
본 논문에서 제안한 기법은 플래쉬 메모리에서 B-트리 인덱스를 위해 shadow 버전을 이용한다. B-트리 노드가 분할될 때 플래쉬 메모리에 저장되며 플래쉬 메모리에 저장된 B-트리 노드의 인덱스 유니트들은 폐기된다. 이를 위해 노드 변환 데이터의 추가로 인덱스맵이 RAM에서 유지된다. 인덱스맵은 인덱스 유니트가 저장된 섹터의 논리 주소와 그 섹터에 저장된 인덱스 유니트에 해당하는 B-트리 노드의 수를 저장한다. 생성된 B-트리 노드를 shadow 버전으로 저장할 때 플래쉬 메모리에 저장된 인덱스 유니트의 섹터를 폐기하고 부모 노드 변환 데이터에서 유지해야 하는 각 B-트리 노드의 연결 리스트의 수가 감소한다. 따라서 인덱스맵에서 유지하는 섹터 수가 작아지므로, 인덱스맵을 위한 공간 오버헤드는 크지 않다.

B-트리 노드가 shadow 버전으로 플래쉬 메모리에 저장될 때, 유보버퍼에 있는 인덱스 유니트 shadow 버전에 변환된 것이기 때문에 폐기할 수 있는 인덱스 유니트 수가 감소한다. 또한 shadow 버전에 변환된 B-트리 노드의 인덱스 유니트를 폐기할 경우 갱신된 인덱스 유니트의 수를 감소시켜 비용을 줄일 수 있다. 이는 B-트리 노드의 폐-아웃을 줄여 갱신된 인덱스 유니트의 수를 감소시켜 비용을 줄일 수 있다. 또한 섹터에 저장되는 인덱스 유니트의 수를 줄여 갱신된 인덱스 유니트의 수를 감소시켜 비용을 줄일 수 있다. 또한 섹터에 저장되는 인덱스 유니트의 수를 줄여 갱신된 인덱스 유니트의 수를 감소시켜 비용을 줄일 수 있다.

Shadow 버전에 변환된 B-트리 노드의 인덱스 유니트를 폐기하기 위해서는 그림 6의 과정 3에서 B-트리 노드에 대해 타임스탬프를 유지해야 한다. 즉, 그림 6의 과정 3에서 B-트리 노드가 shadow 버전으로 플래쉬 메모리에 저장될 때 노드 변환 데이터의 타임스탬프를 갱신하고, 유보버퍼의 갱신에 의해 인덱스 유니트를 플래쉬 메모리에 저장할 때 인덱스 유니트의 타임스탬프와 노드 변환 데이터의 타임스탬프를 비교하여 shadow 버전이 저장된 후 생성된 인덱스 유니트들만 플래쉬 메모리에 저장하는 과정이 필요하다. 유보버퍼의 크기를 크게 할 경우 많은 레코드를 캐싱하므로 응답 간을 단축 할 수 있지만, 이동 컴퓨터장치의 경우 불안정한 전원 때문에 대부분 유보버퍼의 크기를 작게 설정한다. 따라서 타임스탬프를 비교하기 위한 오버헤드도 크지 않다.

4. 결론
플래쉬 메모리는 데이터가 저장된 섹터에서 싸이징이 발생할 때 개설된 섹터를 플래쉬 메모리에 쓰기 위해서는 그 섹터를 포함하고 있는 레코드를 RAM으로 가져와서 해당 섹터를 변환하고, 플래쉬 메모리에 쓰기 후에 RAM으로 분할된 레코드를 저장해야 한다. 다만, 플래쉬 메모리의 각 레코드는 쓰기 지연이 있으므로 쓰기연산을 줄이는 방안이 필요하다. 플래쉬 메모리를 탐색하는 동안 컴퓨터환경에서 데버그가 대응할 수 있어 효율적으로 예측하기 위해 B-트리와 같은 인덱스가 필요하다.

본 논문에서는 플래쉬 메모리의 특성을 고려하여 대부분의 데이터를 효율적으로 인덱싱할 수 있는 shadow 버전을 이용한 B-트리 관리 기법을 제안한다. 제안한 기법은 기본적으로 BFTL 기법의 완료정책을 사용하여 인덱스 유니트를 저장할 때 필요한 섹터의 수를 최소화하여 쓰기연산을 줄일 수 있다. 또한 B-트리 노드의 shadow 벽을 플래쉬 메모리에 저장함으로써 쓰기-아웃을 줄길 수 있으며, B-트리 노드의 shadow 벽을 저장할 때 유보버퍼의 인덱스 유니트를 선택적으로 쓰기함으로써 쓰기연산을 더욱 감소한다. B-트리 노드를 shadow 버전으로 저장할 때 노드 변환 데이터에서 연결 리스트를 줄임으로써 갱신할 때 B-트리 노드 벽 변환 데이터량을 참조한 B-트리 노드 재구성을 간단히 할 수 있다.

본 논문의 장점과는 제안한 알고리즘의 성능을 정량적으로 분석하는 것이다.

5. 참고문헌


127