Journal of Korea Multimedia Society Vol. 12, No. 2, February 2009(pp. 165-177)

플래시 메모리 기반 임베디드 데이터베이스 시스템의 쓰기 성능 향상을 위한 지연쓰기 기법

송하주[†], 권오흠[††]

요 약

센서노드(sensor node)에서의 데이터 기록을 위해 NAND 플래시 메모리 기반의 임베디드 데이터베이스 시스템이 널리 사용되고 있다. 플래시 메모리의 쓰기 및 삭제연산은 읽기 연산에 비해 시간이 많이 소모되고 기억 소자를 마모시킨다. 따라서 이러한 연산들을 줄이는 것은 데이터베이스 시스템의 성능 향상과 메모리의 수명 증대 측면에서 중요하다. 본 논문에서는 이를 위해 지연쓰기 기법을 제안한다. 이 기법은 데이터페이스 페이지의 갱신 영역을 별도의 지연쓰기 레코드로 저장하여 데이터베이스 페이지 쓰기를 줄임으로써 플래시 메모리에 대한 쓰기연산과 삭제 연산을 감소시킨다. 따라서 제안하는 기법은 데이터 기록의 비중이 높은 센서노드 데이터베이스 시스템의 성능을 높이고 플래시 메모리의 수명을 늘리게 된다.

Delayed Write Scheme to Enhance Write Performance of Flash Memory Based Embedded Database Systems

Ha-Joo Song[†], Oh-Heum Kwon[††]

ABSTRACT

Embedded database systems (EDBMS) based on NAND flash memories are widely adopted for logging data on sensor nodes. Since write and erase operations of a flash memory are time consuming compared to read operations and wear memory cells, it is important to reduce these operations to enhance the EDBMS performance and to extend the memory life. In this paper, we propose a delayed write scheme to archive this goal. Proposed scheme stores updated parts of database pages into delayed write records to reduce the database page writes. By doing that, it decreases write and erase operations on a flash memory. Therefore, the proposed scheme enhances the logging performance of a write-intensive EDBMS on a sensor node and extends the flash memory life.

Key words: flash memory(플래시 메모리), embedded database systems(임베디드 데이터베이스 시스템), erase operation(삭제연산), delayed write(지연쓰기)

1. 서 론

플래시 메모리는 가볍고 물리적인 충격에 강하며 전원 소모가 작기 때문에 센서노드의 저장 매체로 각광받고 있다. 반면 플래시 메모리는 빠른 읽기연산 속도에 비해 쓰기연산 속도 [1,2,3]가 다른 메모리보다 느리고 덮어쓰기(overwrite) 연산이 불가능하다 [2]. 플래시 메모리에 쓰기를 수행하기 위해서는 먼저

※ 교신저자(Corresponding Author) : 송하주, 주소 : 부산시 남구 대연3동 599-1(608-737), 전화 : 051)629-6258, FAX : 051)627-6392, E-mail : hajusong@pknu.ac.kr
접수일 : 2008년 9월 3일, 완료일 : 2008년 11월 5일
[†] 부경대학교 전자컴퓨터정보통신공학부 조교수
[††] 정회원, 부경대학교 전자컴퓨터정보통신공학부 교수 (E-mail : ohkwn@pknu.ac.kr)
※ 본 연구는 지식경제부 및 정보통신연구진흥원의 IT핵심기술개발사업의 일환으로 수행하였음. [2006-S-040-03, Flash Memory 기반 임베디드 멀티미디어 소프트웨어 기술 개발]

해당 블록(block)에 대해 삭제연산을 수행해야 하는데 이것은 플래시 메모리의 연산 중 가장 비용이 많이 요구되는 연산이다. 이러한 문제점을 극복하기 위해 파일 시스템과 플래시 메모리 사이에 위치하는 FTL (flash memory translation layer, 플래시변환계층)[2] 소프트웨어가 사용된다. FTL은 다양한 블록 교체기법을 적용하여 이미 삭제 연산을 수행한 빈 블록으로 재사상(remapping)함으로써 플래시 메모리를 기존의 하드디스크와 같이 덮어쓰기가 가능한 블록 저장장치처럼 사용할 수 있게 한다. 그러나 FTL을 이용하여 덮어쓰기를 지원하는 것은 기존의 파일시스템 또는 데이터베이스 시스템에 플래시 메모리를 바로 사용할 수 있다는 장점이 있는 반면 실행 성능이 매우 떨어진다. 이를 개선하기 위해 기존 기법들은 주로 플래시 메모리상의 효과적인 인덱스 구조에 초점을 맞추었다. 이에 반해 본 논문에서는 데이터베이스 내의 인덱스 페이지가 아닌 일반 데이터 페이지들에 대해서도 플래시 메모리상에서 쓰기 성능을 높이기 위한 지연쓰기 기법을 제안하고자 한다. 먼저 제안하는 기법이 적용되는 센서노드용 임베디드 데이터베이스 시스템의 동작 환경은 다음과 같다.

- 데이터베이스 엔진은 라이브러리 형태로 제공된다. 단일 프로그램에 응용 코드와 데이터베이스 엔진 코드가 결합되어 사용된다.
- 트랜잭션 커밋(commit) 시에 페이지버퍼의 데이터를 저장장치에 모두 기록한다(flush)[1)]
- 읽기 보다는 쓰기 연산 위주로 트랜잭션이 수행되는 환경을 가정한다.

임베디드 데이터베이스가 적용되는 센서노드는 클라이언트-서버 구조보다는 데이터베이스 엔진 코드와 응용 코드가 단일 프로그램으로 동작하는 것이 일반적이다. 그리고 데몬 형의 서버가 없으므로 트랜잭션 종료시에는 갱신된 내역을 반드시 저장장치에 모두 기록해야만 데이터를 온전히 보전할 수가 있다. 센서노드는 또한 각종 감시 데이터에 대한 기록 용도로 자주 사용되기 때문에 임베디드데이터베이스 시스템의 다른 응용 분야보다 상대적으로 쓰기 연산의 실행 비율이 높다고 볼 수 있다[4].

본 논문은 다음과 같이 구성된다. 이어지는 2장에서는 플래시메모리 기반의 파일시스템과 데이터베이스 시스템 관련 기존 관련 연구를 보인다. 3장에서는 제안하는 지연쓰기 기법의 구조와 동작방법에 대해 설명한다. 4장에서는 지연쓰기 기법을 사용한 경우와 그렇지 않은 경우를 실험을 통해 성능을 평가하고 5장에서 결론을 맺는다.

2. 관련 연구

저장장치로서 플래시 메모리의 성능을 높이려는 연구는 크게 파일시스템에 기반한 것과 데이터베이스 시스템에 기반한 것으로 구분할 수 있다. 전자의 경우 FTL을 사용하지 않으면서 플래시 메모리에서 효율적으로 동작하는 파일시스템을 개발하려는 연구를 통해 JFFS[5], YAFFS[6]과 같은 새로운 파일시스템들이 제안되었다. 이들은 모두 로그 파일 시스템[7]에 기초를 두고 있다.

데이터베이스 시스템에 기반한 연구는 주로 색인의 성능향상에 중점을 두었다. 데이터베이스 시스템이 효과적으로 동작하기 위해서는 해쉬테이블(hash table) 또는 검색 트리(search tree) 와 같은 색인기법들의 성능을 높일 필요가 있기 때문이다. 데이터베이스 시스템에서 널리 사용되는B-트리(B^+, B^* 포함) 색인 구조는 데이터의 효율적인 삽입 삭제 검색이 용이하며 확장성이 우수한 색인기법이다. 하지만 하드디스크에서와 달리 플래시 메모리 상에서 구현한 B-트리는 빈번한 페이지 갱신에 따른 덮어쓰기 증가로 인해 인덱스 구축비용이 매우 커진다는 문제가 있다. 플래시 메모리 상에서 B-트리를 효과적으로 구현하기 위해 BFTL(B-tree flash translation layer)[8] 기법이 제안되었다. 그러나 BFTL은 거대한 사상 테이블을 유지하기 위한 추가적인 메모리 영역이 필요하며, 검색 시 읽기연산이 다수 발생하여 검색속도가 떨어지는 단점을 가지고 있다. 이러한 단점을 극복하기 위해 FlashDB[4]는 플래시 메모리 기반 데이터베이스 시스템에서 동적인 자동튜닝(tuning) 기법을 제안하고 있다. 이 기법은 저장 장치와 부하(workload)에 따라 B-트리의 페이지(노드)들의 특성을 구분한다. 쓰기 연산이 많은 페이지는 BFTL의 로그 페이지 구조를 적용하되 읽기 연산이 많은 페이지는 디스크 기반 B-트리의 페이지와 동일한 구조를 적용하였다. [9]은 센서 장치와 같은 매우 작은 저장공간을 사용하는 저장시스템을 위한 MicroHash와

1) 페이지버퍼 관리 기법 측면에서 FORCE 전략(strategy).

MicroGF라는 인덱스 구조를 제안하였다. 이 것은 해쉬테이블 구조에 기반한 것으로 소량의 에너지를 소모하면서도 동등비교와 시간순에 의한 데이터 검색이 효과적으로 이루어지도록 하였다. 반면 이 기법들은 대량의 데이터를 다루는 응용에는 적용이 어렵고 일반적인 데이터베이스 시스템에서는 적용하기 곤란한 가정들에 기반하고 있다. 이러한 제약점은 플래시 기반 스마트 카드용 파일 구조들을 제안하고 있는 [10]에서도 동일하다.

이와 같이 기존 기법들이 플래시 메모리상의 효과적인 인덱스 구조에 초점을 맞춘 것이라 볼 수 있다. 반면, 본 논문에서는 데이터베이스 내의 인덱스 페이지가 아닌 일반 데이터 페이지들에 대해서도 플래시 메모리상에서 쓰기 성능을 높이기 위한 기법을 제안하고자 한다.

3. 지연쓰기 기법

이 장에서는 제안하는 기법의 동작원리를 설명한다(그림 1참조). 제안하는 기법의 페이지버퍼(page buffer)는 버퍼 내의 페이지를 갱신할 때 마다 변경 내역을 포함하는 지연쓰기 레코드(delayed write record, DWR)를 생성한다. 지연쓰기 레코드는 메인 메모리에 유지되며 지연쓰기 버퍼가 가득 차거나 트랜잭션 커밋(commit)시에 지연쓰기 파일 (delayed write file)에 기록된다. 지연쓰기 되는 페이지들은 버퍼에서 페이지가 교체(replace)되더라도 데이터베이스(플래시 메모리)에 기록하지 않는다. 따라서 추후 해당 페이지들이 다시 페이지 버퍼에 적재될 때는 이전 페이지의 이미지(image)에 지연쓰기 레코드를 반영하는 병합(merge) 과정을 거치게 된다. 이 과정에서 그림 1과 같이 여러 페이지에 대한 지연쓰기 레코드들을 모아서 저장하면 갱신되는 플래시 메모리의 쓰기 연산을 줄일 수 있다. 요약하면 제안하는 기법은 데이터베이스에 대한 읽기 연산에 대해서는 부가적인 오버헤드가 발생하지만 쓰기 연산의 효율을 높이고 플래시 메모리의 내구성을 늘이는 효과가 있다.

3.1 지연쓰기를 위한 시스템 구조

그림 2는 지연쓰기를 위한 데이터베이스의 구조를 나타낸 것으로 일반적인 페이지 버퍼의 구조에지연쓰기 버퍼(delayed write buffer, DWB)를 추가한 형태이다. DWB는 DWR을 메모리상에 유지하며 페이지 해쉬를 통해 페이지 별로 그룹핑되어(grouping) 있다. 페이지 버퍼에 적재된 페이지에 갱신이 일어나면 갱신 내역에 대한 지연쓰기 레코드가 생성되고 지연쓰기 버퍼에 추가된다. 반대로 데이터베이스 파일로부터 페이지를 읽어 들이는 경우에는 해당 페이지의 페이지식별자(page identifier, PID)를 이용하여 지연쓰기 버퍼에서 관련된 지연쓰기 레코드의 존재 여부를 검사하여 존재하는 경우 그 내용을 읽은 페이지의 내용에 병합한다.

그림 3은 지연쓰기 버퍼 내부 구조를 나타낸 것이다. 지연쓰기 버퍼에는 그림과 같이 페이지식별자 해쉬를 통해 PageStatus 객체에 연결된다. PageStatus는 다음과 같은 정보를 포함한다.

- page_id: 페이지 식별자
- synced_current_trans: 현재 실행중인 트랜잭

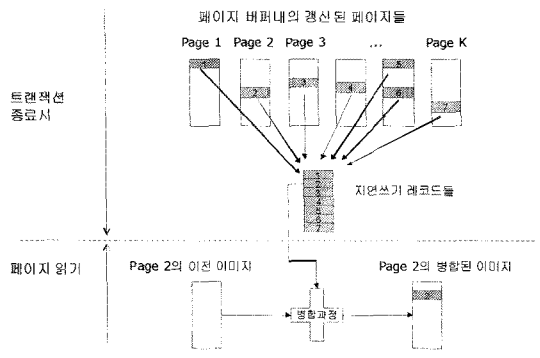


그림 1. 제안하는 기법의 동작 방식

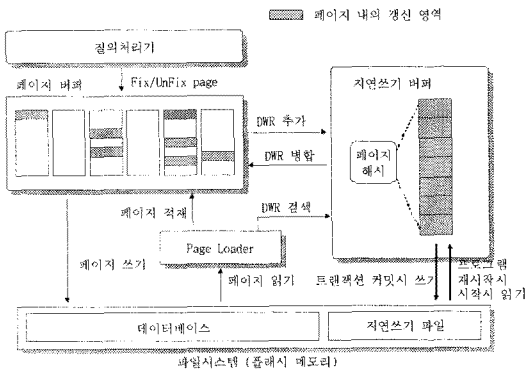


그림 2. 지연쓰기를 위한 데이터베이스 시스템 구조

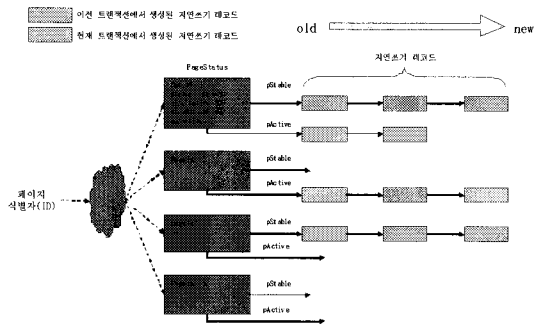


그림 3. 지연쓰기 버퍼의 구조

션에서 데이터베이스에 기록되었으면 True, 그렇지 않으면 False

- dw_status: 페이지에 대한 지연쓰기 상태
- num_active: 현 트랜잭션에서 생성된 DWR의 개수
- num_stable: 이전 트랜잭션들에서 생성된 DWR의 개수
- size_active: 현 트랜잭션에서 생성된 DWR의 크기
- size_stable: 이전 트랜잭션들에서 생성된 DWR의 개수
- size_obsolete: 페이지 전체가 기록되기 전에 생성된 무효한 DWR들의 크기
- save_header: 페이지헤더를 DWR로 기록했으면 true, 그렇지 않으면 false
- statble_list: 이전 트랜잭션에서 생성된 DWR들의 리스트
- active_lit: 현재 트랜잭션에서 생성된 DWR 들의 리스트

dw_status는 해당 페이지에 대한 지연쓰기 상태를 나타내는 것으로 다음과 같은 값을 가질 수 있다.

- DW_PURGED : 갱신 내역이 DB에 모두 기록되었음
- DW_DELAYED : Delayed Write 파일이 존재함
- DW_PURGEDELAYED : DB에 기록된 후, Delayed Write 파일이 새로 생성됨
- DW_NEEDSYNC : 트랜잭션 커밋시에 페이지를 DB에 기록해야 함
- DW_SYNCED : 현재 동작중인 트랜잭션에 의해 DB에 페이지 전체가 기록됨

지연쓰기 버퍼내의 지연쓰기 레코드는 활성 지연쓰기 레코드(active delayed write records)와 안정된 지연쓰기 레코드(stable delayed write records)로 구분된다. 활성 지연쓰기 레코드는 현재 진행중인 트랜잭션에 의해 생성된 것이고, 안정된 지연쓰기 레코드는 이전 트랜잭션에서 생성된 지연쓰기 레코드를 의미한다.

그림 4는 지연쓰기 레코드의 구조를 나타낸 것이다. 앞부분은 공통필드들로 이루어져 있으며 레코드의 크기(rec_size), 갱신이 되는 페이지에 대한 식별자, 연산의 종류(op_code)에 대한 정보를 포함한다. 공통 필드 이후로는 각각의 연산별로 필요한 정보를 기록하게 된다. 제안 기법을 SQLite[11]에 적용하였을 때 다음과 같은 5가지 종류의 연산을 사용하였다.

- MEMCPY: 페이지의 일정 영역에 대한 갱신 내역을 유지
- MEMSET: 페이지의 시작부터 일정크기의 영역을 주어진 값으로 설정
- PUT_BYTE/PUT_2BYTE/PUT_4BYTE: 오프셋(offset)으로 주어지는 페이지의 일정영역에 대해 주어진 1바이트/2바이트/4바이트 값으로 설정
- INSERT_CELL: 주어진 레코드를 지정된 페이지 위치에 삽입
- DROP_CELL: 주어진 위치의 레코드를 삭제

제안하는 기법에서 데이터 페이지는 임의의 순간에 오직 하나의 트랜잭션만이 갱신연산을 수행할 수 있는 것으로 하며 해당 트랜잭션이 종료되어야 다른 트랜잭션이 사용할 수 있는 것으로 가정한다. 아울러 지연쓰기의 대상은 인덱스 페이지와 페이지 크기보다 작은 레코드들 저장하는 페이지로 한정하고 BLOB(binary large object)을 저장하기 위한 페이지는 제외한다.

데이터베이스 저장시스템 수준에서 페이지에 대한 갱신은 매우 다양한 형태로 이루어 질 수 있으나 제안하는 기법에서는 앞서 언급된 5종의 연산으로 축약하여 갱신 내역을 기록하였다. 페이지 헤더 영역

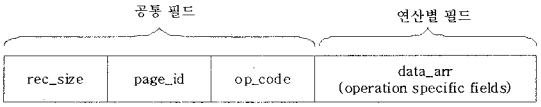


그림 4. 지연쓰기 레코드의 구조

은 트랜잭션 수행도중 동일 영역이 여러 번 갱신되는 경우가 흔히 발생한다. 따라서 매번 갱신 때마다 지연쓰기 레코드를 생성하는 대신 갱신되는 범위만을 유지하였다가 트랜잭션 종료시에 갱신 영역을 지연쓰기 레코드로 생성하였다. 페이지에 대한 갱신 영역의 크기가 작을수록 지연쓰기의 효과는 커지고 반대의 경우에는 효과가 줄어들 것이다. 따라서 제안하는 기법에서는 active_list에 지연쓰기 레코드를 추가하는 과정에서 페이지별로 size_active+size_stable의 크기가 페이지 크기의 일정 비율 (dwratio_per_page) 이상이 되면 지연쓰기를 수행하지 않고 페이지 전체를 데이터베이스에 기록하도록 하였다. 지연쓰기 버퍼가 계속해서 증가하는 것을 막기 위해 지연쓰기 버퍼가 가득 찬 이후에 발생하는 갱신 연산은 페이지 전체를 기록하도록 하였다(그림 5참조). 이 경우에는 해당 페이지에 대한 지연쓰기 레코드가 지연쓰기 버퍼에서 삭제되므로 버퍼의 유휴 공간을 늘리게 된다.

지연쓰기 파일(delayed write file)은 트랜잭션의 종료시에 지연쓰기 버퍼의 활성 지연쓰기 레코드들이 저장된다. 응용프로그램의 시작시에는 지연쓰기

```
DWRec alloc_dwrec (int opCode, int pageNo, int recSize)
1   PageStatus* ps = Look up hash with pageNo
2   if ps = NULL then
3       ps ← allocate a new PageStaus structure;
4       initialize ps;
5   else if ps = DW_NEEDSYNC then
6       return null; // whole page will be written to flash memory
7   else if (ps->dwStatus = DW_SYNCED) then
8       ps->dwStatus ← DW_PURGEDELAYED;
9   endif
10  if ps->size_active+ps->size_stable+recSize > dwratio_per_page * PAGE_SIZE
11      || DW buffer is full then
12      remove all in memory DW Records associated with ps;
13  ps->dwStatus = DW_NEEDSYNC;
14      return NULL;
15  endif
16
17  DWRec dwr ← allocate a new DW record.
18  initialize dwr;
19  add dwr into ps->activeList;
20  return dwr;
```

그림 5. 페이지 갱신시 지연쓰기 레코드의 할당 과정

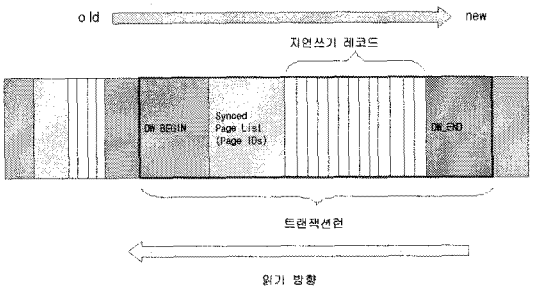


그림 6. 지연쓰기 파일의 구조

파일의 내용을 읽어들여 지연쓰기 버퍼를 메모리에 구성한다. 그림 6은 지연쓰기 파일의 구조를 나타낸 것이다. 지연쓰기 레코드들은 트랜잭션 단위로 기록이 되며, 트랜잭션의 경계를 나타내기 위해 경계레코드를 시작과 끝에 삽입한다. 경계레코드에는 다음과 같은 정보들이 유지된다.

- demarc : 지연쓰기 정보 시작(BEGIN) 또는 종료 (END)
- tx_id: 트랜잭션 식별자(transaction identifier)
- offset_begin: 경계시작 레코드까지의 오프셋 (경계 종료 레코드에만 사용)
- num_dwrecs: DW 레코드의 개수
- num_sync_pages: DW를 사용하지 않고 데이터베이스에 기록된 페이지의 개수
- pad_size: 페이지 정렬(alignment)을 사용하여 지연쓰기 레코드를 기록한 경우, 현재 지연쓰기에 추가된 더미 바이트의 크기
- dw_magic: 매직 문자열, 지연쓰기 기록의 무결성(integrity)을 검사하기 위해 사용

트랜잭션 시작 경계레코드 이후에는 페이지 전체가 기록된 페이지들에 대한 페이지식별자들이 위치한다(그림 6의 synced page list). 이어서 지연쓰기 레코드들이 기록되고 페이지 크기에 맞춰 정렬을 하기 위해 더미 데이터를 추가한다. 마지막으로 종료 경계레코드를 기록한다(그림 7 참조). 시작 경계레코드와 종료 경계레코드로 구분되는 레코드들의 집합을 트랜잭션런(transaction run)이라 하고 갱신 트랜잭션당 하나의 트랜잭션런이 할당된다. 지연쓰기 파일의 읽기는 데이터베이스 응용 프로그램이 시작되는 경우에 수행된다. 그림 8은 읽기 과정을 알고리즘 형식으로 나타낸 것이다.

```
flush_page_buffer
1   FILE cur_dwFile;
2   boolean shrinkMode = FALSE;
3   if (DW File Shrinking is needed) then
4       cur_dwFile  ← open new file
5       shrinkMode ← TRUE;
6   else
7       cur_dwFile ← current dwFile;
8   endif
9   write BEGIN run record;
10  foreach ps in dwBuffer do
11      if ps->dwStatus = DW_SYNCED
12        || ps->dwStatus = DW_PURGE_
                              DELAYED then
13          add ps->page_id into syced_page_list;
14          continue
15      end if
16      foreach dwRec in ps->activeList do
17          write dwRec into cur_DWFile;
18        if ps->dwStatus == DW_PURGED then
19            ps->dwStatus = DW_PURGEDELAYED;
20          else
21              ps->dwStatus = DW_DELAYED;
22          end if
23      end
24      if shrinkMode == TRUE then
25          flush all dw records in ps->stableList
            into cur_dwFile;
26      end if
27      move all DW records in  ps->activeList
        to ps->stable List;
28  end
29  write synced page list into cur_dwFile;
30  write END run record into cur_dwFile;
31  synchronize OS buffer and persistent storage;
32  close DW file;
33  if shrinke_mode then
34      delete old DW file;
35      rename new DW file as DW file
36  end
```

그림 7. 트랜잭션 종료시 지연쓰기 파일에 대한 쓰기 과정

트랜잭션런 내에서는 앞→뒤로 레코드를 읽되 최근 트랜잭션런→이전 트랜잭션런 순서로 읽는다. 개개의 트랜잭션런에 대해서는 다음과 같은 읽기 절차를 수행한다. 트랜잭션런에서Synced Page리스트를 읽은 후, 해당페이지의 PageStatus에 SYNCED로 표시한다. 만약 PageStatus에e 안정된 지연쓰기 레코드가 존재하면PageStatus는 SYNC_DELAYED로 표시하고 지연쓰기 레코드를 읽어서 지연쓰기 해쉬에 추가한다. 이 때, 만약 해당되는 페이지의 dw_status가 SYNCED 또는SYNC_DELAYED이면 지연쓰기 레코드를 해쉬에 추가하지 않는다. 이것은 페이지가 데이터베이스에 쓰여지면 그때까지의

```
build_dwbuffer(FILE dwFile, int curPos)
1   Transaction Record tr;
2   Delayed Write Record dwr;
3   PageStatus ps;
4
5   move to the last transaction run in dwFile
6   while there is transaction run to be processed do
7       foreach dwr in delayedwrite records in
        transaction run do
8         ps ← look up dwr.page_id from the hash;
9           if ps = NULL then
10            ps ← allocate new PageStatus object;
11              ps.page_id ←dwRec.page_id
12              ps.dw_status ←DW_DELAYED;
13              continue;
14        else if ps->dw_status = DW_PURGED or
                              DW_PURGEDELAYED
then
15        ps->size_obsolete += dwRec.rec_size;
16        else if ps->dw_status = DW_DELAYED
then
17              add dwRec into StableList;
18          endif
19      end
20      foreach page in syncedPages do
21        ps ← look up dwr.page_id from the hash;
22          if ps = NULL then
23          else if ps.dw_status = DW_DELAYED
then
24              ps ← DW_PURGEDELAYED;
25          end if
26      end
27      proceed to previous transaction run
28  end
```

그림 8. 지연쓰기 파일에 대한 읽기 과정

갱신내역이 모두 반영된 것이기 때문이다. 따라서 그 이전에 생성된 지연쓰기 레코드들은 사용할 필요가 없으므로 무효한(obsolete) 지연쓰기 레코드가 된다.

지연쓰기 파일은 시간이 경과될수록 그 크기가 늘어나게 되므로 유효한 데이터만을 선별하여 크기를 줄이는 축약(shrink) 작업을 수행한다. 축약작업은 새로운 지연쓰기 파일에 유효한 데이터만을 기록하고 이전 지연쓰기 파일을 삭제하는 것으로 완료된다(그림6 참조). 불필요한 데이터는 페이지 전체가 갱신되기 이전에 저장된 지연쓰기 레코드와 트랜잭션런에 저장된 Synced Page List이다. 무효한 지연쓰기 레코드는 지연쓰기 버퍼에 존재하지 않으므로 버퍼내의 지연쓰기 레코드들만을 저장하는 과정에서 자연스럽게 제거된다. 무효한 지연쓰기 레코드가 지연쓰기 파일에서 제거되므로 이들을 파악하기 위한 경계점으로 사용된Synced Page List는 필요 없게 된

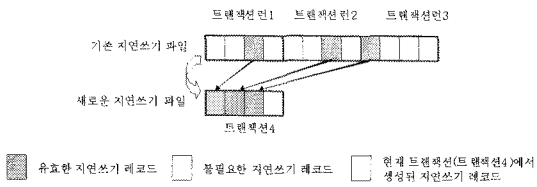


그림 9. 지연쓰기 파일의 축약 과정

다. 축약 작업은 현 트랜잭션에 대한 지연쓰기 레코드 쓰기 과정에서 수행된다. 만약 축약작업 도중 갑작스런 정전 등으로 인해 컴퓨터 가동이 중단된다면 트랜잭션은 취소(abort)되고 재가동시에는 기존의 지연쓰기 파일을 사용하고 새로운 지연쓰기 파일은 무시된다. 축약 작업의 수행 여부를 판단하기 위해서는 다음과 같은 선택 기준을 사용하였다.

- 데이터베이스 크기가 32페이지 이하인 경우
  - 유효한 DW 레코드의 크기의 합 * 10 < 불필요한 DW 레코드의 크기의 합
- 이외의 경우
  - 유효한 DW 레코드의 크기의 합 < 불필요한 DW 레코드의 크기의 합

지연쓰기가 효율적으로 동작하려면 단일 트랜잭션 내에서 다수의 페이지에 걸쳐 소량의 갱신이 이루어지는 경우에 적용되어야 한다. 따라서 트랜잭션 커밋시에는 지연쓰기를 적용한 경우의 페이지 쓰기 회수(P_d)와 그렇지 않은 경우의 회수(P_n)를 계산한다. P_n은 페이지 버퍼에서 트랜잭션 수행도중 갱신이 이루어진 페이지(dirty page)의 개수를 파악하면 간단하게 구할 수 있다. P_d는 데이터베이스 및 지연쓰기 파일에 대한 쓰기 이므로 다음과 같이 예측한다.

$$P_d = n_d + \left\lceil \frac{2R_{tr} + n_d \cdot sizeof(\text{int}) + \sum R_i}{ps} \right\rceil + 1$$

여기서 n_d, R_{tr}, R_i, ps은 각각, 전체가 기록되는 페이지의 개수, 트랜잭션 경계 레코드의 크기, i번째 지연쓰기 레코드의 크기 (지연쓰기 레코드가 다수인 경우), 데이터베이스 페이지의 크기를 나타낸다. 1은 지연쓰기 파일의 메타정보에 대한 갱신 비용을 나타낸 것으로 1회의 페이지 쓰기가 발생하는 것으로 가정하였다. 따라서 $P_d - P_n$이 양수가 되는 경우에만 지연쓰기를 적용한다. 이때 플래시 메모리의 섹터 크기를 ss라 하면, 섹터 단위의 쓰기는 $\frac{(P_d - P_n) \cdot ps}{ss}$ 만큼 줄어들 것이다.

제안한 기법은 다수의 페이지를 갱신하는 대신 지연쓰기 레코드들만을 저장함으로써 실제 갱신되는 페이지는 크게 줄어들 수 있다. 반면, 제안하는 기법은 CPU 사용 측면과 메모리 사용 측면에서 오버헤드가 따른다. 페이지를 읽을 때 마다 지연쓰기 버퍼를 검색해야 하며 지연쓰기 페이지로 판명되면 병합과정을 거쳐 페이지를 최신 상태로 변경해야 한다. 여기서 페이지식별자를 이용한 검색 과정에 소요되는 시간은 그다지 크지 않으며 코드 최적화를 통해 상당부분 줄일 수 있다. 그러나 병합 과정은 상당 시간이 소요될 수 있다. 이를 위해 제안하는 기법에서는 페이지에 대한 지연쓰기 영역의 크기가 일정 비율 이상이면 지연쓰기를 하지 않고 페이지 전체를 갱신하도록 하였다. 최근 CPU의 성능이 급격히 향상되고 있음을 감안했을 때 병합 과정에서 소요되는 시간은 점차 줄어들 수 있을 것으로 예상한다.

3.2 제안하는 기법과 기존 연구와의 비교

제안하는 기법이 다른 위치 갱신(out-of-place update) 방식으로 페이지 갱신을 수행하는 측면에서는 [6]과 동일한 접근 방법을 취하고 있다. 반면, [6]은 임의 접근 (random access) 보다 순차 접근(sequential access)가 빠른 디스크의 물리적인 특성을 최대한 살리기 위해 제안된 기법이다. 파일의 일부 블록들이 갱신되면 그것들을 제자리 갱신(in-place update) 하지 않고 로그 형식으로 순차적으로 기록한다. 이를 통해 임의 접근을 줄이고 순차접근을 늘려 디스크 대역폭(bandwidth)의 활용도를 높인다. 따라서 디스크 블록(512 바이트 이상) 단위로 쓰여지며 쓰기 연산의 양을 줄이지는 않는다. 반면 제안하는 기법은 데이터베이스 페이지의 일부가 갱신되는 경우에는 페이지 전체에 대한 갱신을 가능한 늦추고 여러 페이지에 걸친 소규모의 갱신 연산을 합쳐서 하나의 페이지에 기록한다. 즉, 쓰기 연산을 줄이는 목적으로 한다. 로그에 기록되는 지연쓰기 레코드(일반적으로 수십 바이트) 단위도 [6]보다 훨씬 작다고 할 수 있다.

BFTL 기법의 예약버퍼(reservation buffer)는 제안하는 기법의 지연쓰기 버퍼와 유사한 역할을 수행한다. 페이지에 대한 갱신 연산을 덮어쓰기로 수행하지 않고 별도의 플래시 공간에 저장한다는 점에서는 제안하는 기법과 동일한 접근 방법이라 할 수 있다.

반면 제안하는 기법에서는 페이지가 버퍼에 적재될 때 지연쓰기 버퍼에 저장된 레코드를 적용하므로 페이지 버퍼에는 최신의 페이지 이미지가 존재한다. 이에 비해 BFTL에서는 갱신 이전의 페이지와 갱신 데이터가 서로 독립적으로 존재한다. 그리고 예약버퍼 내의 데이터를 검색하면 페이지 버퍼의 데이터는 사용되지 않는다는 점에서 차이가 있다. 또한 BFTL이 색인 페이지에만 적용되는 것에 비해 제안하는 기법은 색인 및 데이터 페이지 모두 적용할 수 있다.

로그버퍼 FTL 기법(log buffer FTL)[12]은 플래시 메모리의 섹터에 대해 갱신이 일어나는 경우 플래시 메모리의 일부 영역을 로그버퍼로 지정하고 여기에 변경된 데이터를 저장하는 FTL 기법이다. 제안하는 기법과 로그버퍼 기법 모두 동일한 페이지가 연속적으로 갱신되는 경우에 유리하다는 점에서는 유사한 특징을 가진다. 반면 동작하는 수준이 섹터와 페이지라는 점에서의 구분된다. 또한 제안하는 기법은 페이지의 내용 중 변화된 부분만을 유지하는 반면 로그버퍼 기법은 변경된 전체 섹터를 유지한다는 점에서 차이가 있다.

제안하는 기법이 갱신되는 페이지의 로그 레코드를 이용한다는 점은 참고문헌[13]과 유사하다. 차이점은 [13]의 경우 Flash Memory Page (Erase Block 단위) 내에 갱신된 페이지의 로그 레코드를 위치시키는 반면 제안하는 기법은 별도의 파일에 기록한다는 점이다. 이것은 [13] 기법이 FTL 수준에서 지원되어야 하는 반면 제안하는 기법은 파일 시스템 상위 수준에서 지원될 수 있음을 뜻한다. 즉, 플래시 메모리라는 저장장치를 접근하는 방식에서 [13]이 하위 수준에서 접근하는 기법이라 할 수 있다. [13]에서는 서로 다른 두 페이지가 갱신되는 경우 각각의 로그가 별도의 섹터에 저장된다. 반면에 제안하는 기법은 [6]과 유사하므로 서로 다른 페이지의 로그가 동일한 플래시 메모리 섹터에 저장될 수 있고 이 경우에 대해서는 플래시 쓰기 회수를 플래시 쓰기 회수를 줄일 수 있다. [13]은 데이터 페이지를 읽는 과정에서 Erase Block 내의 로그 레코드를 연관(associative) 검색해야 할 필요가 있다. 반면 제안하는 기법은 각 페이지와 관련된 지연쓰기 레코드의 존재 여부를 초기에 미리 구축하는 접근 방식이라 볼 수 있다.

개념적으로 제안하는 기법과 가장 유사한 것은 레코드 레벨 REDO 로깅[14] 방법이라 할 수 있다. 양측 모두 페이지에 갱신이 이루어질 때에 갱신을 유발한 레코드 수준에서 로그 레코드를 생성하고 트랜잭션 커밋시에 이것을 지속성 저장장치에 기록하는 면에서 동일하다. 하지만REDO 로그는 데이터베이스 시스템 재가동시 고장회복 단계에서 커밋된 데이터가 데이터베이스에 반드시 반영되도록 하기 위한 용도로 사용되는 반면에, 지연쓰기 레코드는 순전히 페이지 갱신을 지연하기 위한 용도이다. 또한 지연쓰기 기법은 고장회복 단계가 아닌 정상적인 동작 중에 사용된다는 점에서도 차이가 있다.

4. 실험평가

제안하는 기법의 성능을 평가하기 위해 SQLite EDBMS에 제안기법을 구현하였다. 표 1은 제안하는 기법의 성능을 평가하기 위한 하드웨어 및 소프트웨어 환경을 나타낸 것이다.

센서노드의 임베디드 데이터베이스가 환경 감시 데이터를 기록하는 용도로 자주 사용되므로 데이터 삽입에 소요되는 시간을 위주로 실험을 수행하였다. 레코드의 크기가 각각 16바이트와 256바이트인 테이블을 사용하였고 빈 테이블에서 시작하여 임의의 키를 가지는 레코드 10,000개를 삽입하는데 소요되는 시간을 측정하였다.

레코드 삽입 종료 후, 레코드의 크기가 16 바이트, 256 바이트 일 때, 데이터베이스의 크기는 각각 241,664 바이트와 3,149,824 바이트가 되었는데 지연쓰기의 적용여부에 상관없이 모두 동일한 크기이다.

표 1. 실험 환경

임베디드 시스템 보드	Cirrus EDB9315A
CPU	ARM9 500MHz
OS	Linux 2.6.4
파일 시스템	YAFFS2
페이지 버퍼	40KB
플래시 메모리	NAND Flash 128MB중 4MB 파티션을 사용 (Large Block NAND Flash Memory)
트랜잭션의 종류	INSERT, TABLE SCAN
데이터베이스 페이지 크기	4096 bytes
레코드의 크기 (record size, rs)	16 bytes, 256 bytes

먼저 제안하는 기법을 사용하기 위해 부가적으로 필요한 저장공간의 사용량 및 메인메모리 사용량을 알아본다. 표 2는 삽입 트랜잭션의 유형별로 평균적인 지연쓰기 파일의 크기를 나타낸 것이다. 전체적으로 데이터베이스 저장공간 크기의 10% 수준을 넘지 않는 부가적인 저장공간(지연쓰기 파일)을 사용만으로도 제안하는 기법을 적용할 수 있음을 알 수 있다[2)]

트랜잭션에서 삽입하는 레코드의 개수가 작을수록 단일 트랜잭션런 내부에는 지연쓰기 레코드 보다는 트랜잭션 시작과 끝을 나타내는 레코드들이 차지하는 부피가 상대적으로 크고 전체 트랜잭션 시행회수도 많게 되므로 지연쓰기 파일의 평균 크기가 커지게 되는 것을 알 수 있다.

표 3은 지연쓰기에 따른 메모리 사용량을 측정한 것으로 제안 기법을 적용하기 전후의 프로세스 이미지의 크기 및 부가적인 메모리 사용량의 비율을 나타내었다. 지연쓰기 레코드의 양이 많아 질수록 메모리의 사용량도 늘어날 것이므로 표 2 **지연쓰기 파일의 평균 크기(바이트 단위) 및 DB파일 크기에 대한 비율**에서 제시된 지연쓰기 파일의 크기와 상관관계에 있는 것을 볼 수 있다. 전체적으로10% 내의 메모리를 부가적으로 상용하는 것을 알 수 있다.

표 2. 지연쓰기 파일의 평균 크기(바이트 단위) 및 DB파일 크기에 대한 비율

레코드/ 트랜잭션 (트랜잭션의 수)	1 (10000)	4 (2500)	16 (625)	64 (157)	**256 (40)**
rs=16	23,108 (9.56%)	19,018 (7.78%)	15,778 (7.53%)	11,917 (4.93%)	8,485 (3.51%)
rs=256	58,339 (1.85%)	56,484 (1.79%)	54,496 (1.73%)	50,908 (1.62%)	18,028 (0.60%)

표 3. 지연쓰기 적용 전/후의 메모리 사용량 비교

레코드/ 트랜잭션 (트랜잭션의 수)	*1 (10000)*	*4 (2500)*	*16 (625)*	*64 (157)*	**256 (40)**
rs=16	1812 /1880 (3.8%)	1832 /1888 (3.1%)	1868 /1918 (2.7%)	1928 /1976 (2.5%)	2016 /2044 (1.4%)
rs=256	1812 /1972 (8.8%)	1876 /2032 (8.3%)	1924 /2084 (8.3%)	2174 /2324 (6.9%)	2240 /2344 (4.6%)

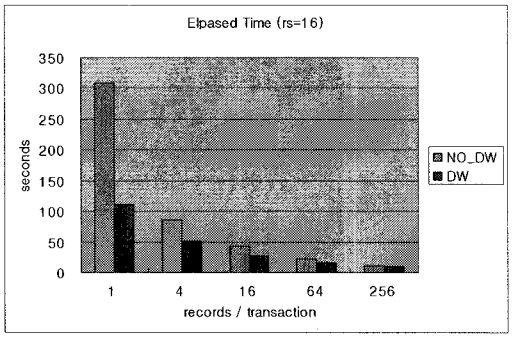
표 4. 실행되는 트랜잭션의 수 및 지연쓰기 파일 축약 회수 (괄호는 발생 빈도)

레코드/ 트랜잭션 (트랜잭션의 수)	*1 (10000)*	*4 (2500)*	*16 (625)*	*64 (157)*	**256 (40)**
rs=16	334 (3.4%)	169 (6.8%)	94 (15.0%)	46 (29.3%)	17 (42.5%)
rs=256	364 (3.6%)	195 (7.8%)	115 (18.4%)	70 (44.6%)	34 (85.0%)

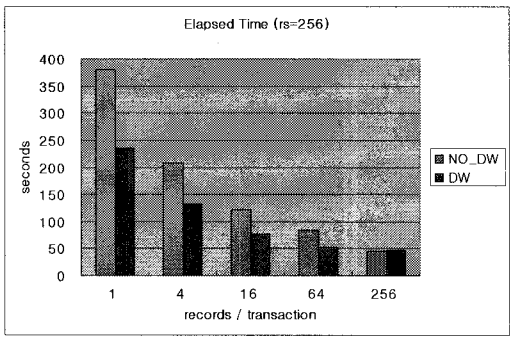
표 4는 트랜잭션 수행도중 발생하는 축약작업의 발생 회수를 나타낸 것이다. 축약작업은 지연쓰기 파일 내의 무효한 지연쓰기 파일을 추려내는 과정인데 수행성능에는 좋지 않은 영향을 미치게 된다. 트랜잭션 내에서 많은 수의 레코드를 삽입할수록 무효한 지연쓰기 레코드의 비율이 빠르게 증가하게 되는데 그에 따라 축약작업도 더 자주 발생하게 된다. 특히, 축약작업 수행조건에서 데이터베이스의 페이지의 개수가 32개를 넘는 경우에는 더 자주 축약작업을 수행하도록 하였는데 그에 따라 256바이트 레코드를 사용하는 경우에 축약작업이 더 자주 발생하는 것을 보여 준다.

그림10에서 가로축은 트랜잭션당 삽입한 레코드의 수, 세로축은 소요시간을 각각 나타내며 DW와 NO_DW는 각각 제안기법을 적용한 경우와 그렇지 않은 경우를 나타낸 것이다. 실험에서 사용한 파일시스템이 YAFFS2이기 때문에 NO_DW는 Log Structured File System[6]과 그 변종들[4,5]의 성능으로 볼 수 있다. YAFFS2는 NAND 플래시 메모리에서 효과적으로 동작할 수 있는 파일 시스템으로 제안된YAFFS를 개선한 것이다. 이것은 2K 바이트의 크기를 갖는 대블럭(Large Block) NAND 플래시 메모리에 적합하도록 개발된 것이다. 실험에서 레코드의 크기에 상관없이 대체적으로 제안 기법을 적용하는 편이 좋은 성능을 보였는데 특히 레코드의 크기가 작고 동시에 삽입되는 레코드의 개수가 적은 경우 상대적으로 더 좋은 효과를 보인다. 이것은 여러 페이지에 대한 갱신이 하나의 지연쓰기 페이지에 저장될 수 있으므로 전체적으로 페이지 쓰기 연산을 줄어들고 따라서 그 만큼 플래시 쓰기 및 지우기가 줄어

2) SQLite에서 사용할 수 있는 최소 크기인 8바이트 레코드(키 4바이트+데이터 4바이트)를 사용한 경우에는 데이터베이스의 크기(152KB)에 대해 18.4%(28KB)~4.93%(7.2KB) 크기의 지연쓰기 파일이 사용되었다

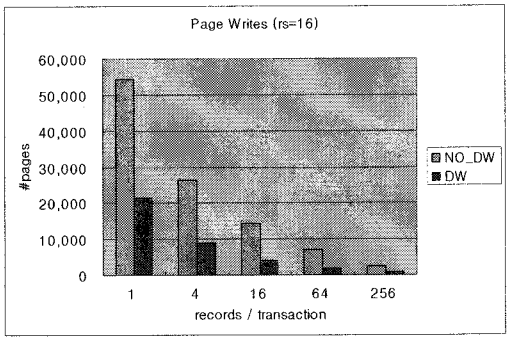


(a) 레코드 크기 16 바이트

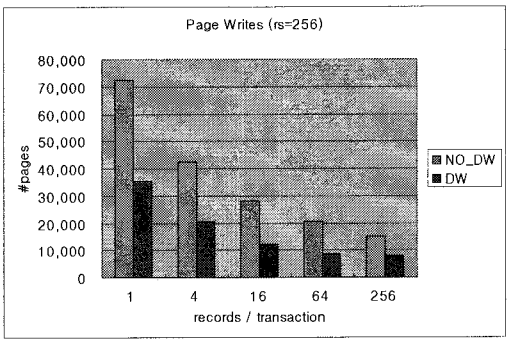


(b) 레코드 크기 256 바이트

그림 10. 레코드 삽입 소요시간 비교



(a) 레코드 크기 16 바이트



(b) 레코드 크기 256 바이트

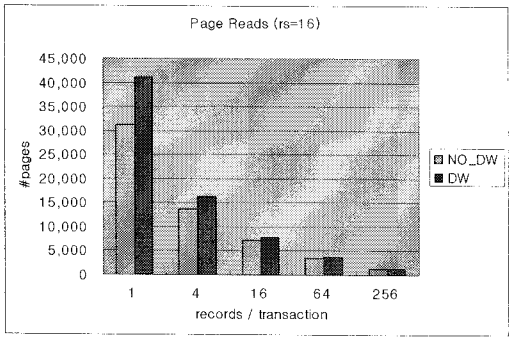
그림 11. 파일에 대한 쓰기 연산 횟수의 비교 (DB 페이지 단위)

들 수 있기 때문이다.

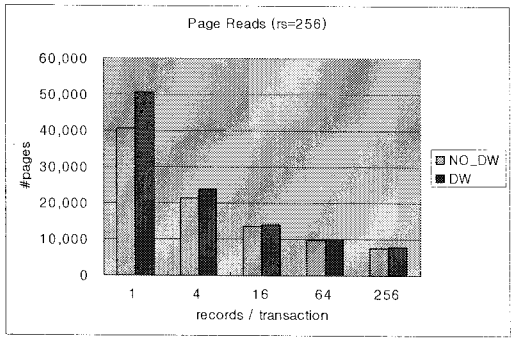
단일 레코드가 삽입되는 경우에도 데이터 페이지와 인덱스 페이지가 동시에 갱신되며 경우에 따라서는 B-트리 인덱스 관리를 위해 다수의 페이지가 조금씩 갱신될 수 있다. 이 경우에도 지연쓰기의 장점이 적용될 수 있다. 그러나 트랜잭션당 레코드의 크기가 일정 수준 이상이 되면 (그림 10에서는 64개 초과의 경우) 지연쓰기의 장점이 퇴색되는 것을 볼 수 있다. 이것은 일정 개수 이상의 레코드가 삽입되는 경우 지연쓰기에 의해 감소되는 페이지의 개수가 더 이상 늘어나지 않는 것과 상대적으로 축약연산이 더 자주 발생하기 때문인 것으로 볼 수 있다.

이러한 특징을 그림 11에서도 확인 할 수 있다. 그림 11은 삽입 실험을 수행하는 동안 발생하는 데이터베이스 페이지 쓰기 연산의 회수를 비교한 것으로 지연쓰기가 쓰기 연산의 회수를 감소시키는 것을 확인 할 수 있으며 앞서 언급된 것처럼 트랜잭션 내에서 삽입되는 레코드의 개수가 일정 수준을 넘어서면 쓰기 연산의 감소 폭이 크지 않게 됨을 알 수 있다. 레코드의 크기가 커질수록 그렇지 않은 경우에 비해 분산된 페이지 쓰기를 더 작은 개수의 페이지에 쓰기로 합쳐주는 효과가 감소하게 될 것이다. 이것은 그림 11-(b)가 그림 11-(a)보다 모든 영역에서 페이지 감소 비율이 더 작은 것을 통해서도 확인 할 수 있다.

그림 12는 삽입실험에서 발생하는 데이터베이스 페이지에 대한 읽기 횟수를 비교한 것이다. 어느 경우에서나 지연쓰기를 적용한 경우 더 많은 페이지 읽기가 발생한다. 제안하는 기법은 동일 프로세스에서 계속해서 트랜잭션을 수행하는 경우에는 이전 트랜잭션들에서 발생한 지연쓰기 레코드를 메인메모리에 계속 캐쉬하는 방식을 취하였다. 이 과정에서 캐쉬의 유효성을 확인하기 위해 매 트랜잭션의 시작시마다 지연쓰기 파일의 끝을 확인하는 과정을 거친다. 이 과정에서 부가적인 파일 읽기가 발생하는데 이것은 트랜잭션 시행회수와 비례하게 된다. 그림 12에서 트랜잭션 내에서 삽입하는 레코드의 개수가 많아질수록 트랜잭션의 횟수는 줄어들게 되고 그에 따라 읽기 또한 줄어들게 됨을 알 수 있다.

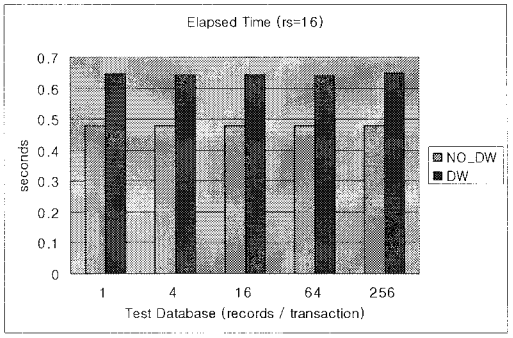


(a) 레코드 크기 16 바이트

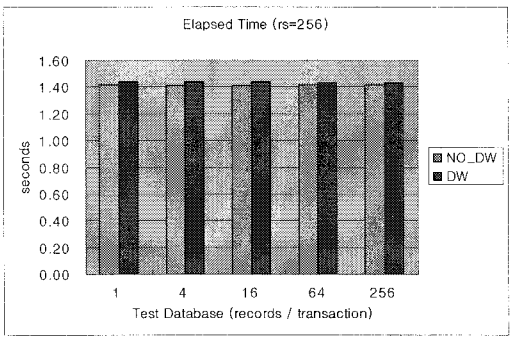


(b) 레코드 크기 256 바이트

그림 12. 파일에 대한 읽기 연산 횟수의 비교 (DB 페이지 단위)



(a) 레코드 크기 16 바이트



(b) 레코드 크기 256 바이트

그림 13. 테이블 스캔 성능 비교 (지연쓰기 레코드를 캐쉬하지 않음)

제안하는 기법은 데이터베이스 갱신 트랜잭션의 성능을 높이기 위한 것이나 읽기 트랜잭션에 대해서는 성능 저하를 가져온다. 특히 단일 프로세스 내에서 계속해서 트랜잭션을 수행하는 것이 아닌 서로 다른 프로세스에서 트랜잭션을 수행하는 경우에는 지연쓰기 레코드를 캐쉬할 수 없게 되므로 매번 지연쓰기 파일을 새로 읽어 들여야 하는 부담이 따르게 된다. 이를 확인하기 위해 10,000개의 레코드를 모두 삽입한 다음 메인 메모리상에 캐쉬된 지연쓰기 레코드를 모두 무효화한 후, 테이블 내의 모든 레코드를 스캔하기 위해 소요되는 시간을 측정해 보았다. 이 테스트는 지연쓰기에 따른 오버헤드가 가장 크게 발생하는 경우에 대해 범위 질의(range query)의 평균적인 성능 감소를 측정하기 위한 것으로 볼 수 있다. 그림 13에서 보듯이 레코드의 크기가 작은 경우에 오버헤드가 크게 발생하는 것을 알 수 있다. 그것은 레코드의 크기가 작은 경우 지연쓰기 파일의 크기가 데이터베이스 크기에 비해 상대적으로 크게 생성되기 때문인 것으로 볼 수 있다. 동일 크기의 레코드인 경우 트랜잭션에서 삽입하는 레코드의 개수와 오버헤드는 큰 차이가 없는 것으로 나타났는데 그것은 표 2에서 나타난 것처럼 8KB~24KB(rs=16의 경우) 또는 18KB~58KB(rs=256의 경우)에 대한 읽기 시간의 차이가 크지 않기 때문인(수십 ms 이내) 것으로 판단된다.

5. 결론 및 향후 연구

쓰기와 소거 연산은 플래시 메모리의 연산 중에 가장 큰 시간이 소요된다. 따라서 플래시 메모리 기반 데이터베이스 시스템에 있어 쓰기 연산을 줄이면 데이터베이스의 수행 성능을 높일 수 있다. 본 논문에서는 데이터베이스 페이지에 대한 갱신 내역을 별도의 지연쓰기 레코드로 저장하였다가 추후 해당 페이지가 다시 접근될 때는 지연쓰기 레코드의 내용을 병합하는 지연쓰기 기법을 제안한다. 제안하는 기법은 갱신되는 데이터베이스 페이지의 개수를 줄임으로써 플래시 메모리에 대한 쓰기와 소거 연산을 감소

시킨다. 공개소스 임베디드 데이터베이스 시스템 중의 하나인 SQLite Version 3에 구현하여 성능을 비교 분석한 결과 쓰기 연산의 감소를 통해 데이터베이스 갱신 성능을 향상시킬 수 있음을 확인하였다. 제안하는 기법은 지연쓰기 레코드의 저장을 위해 데이터베이스 파일과는 분리된 별개의 파일을 사용하고 있고 그로 인해 파일 개방(open)과 폐쇄(close), 파일 삭제, 이름 변경 등의 부가적인 오버헤드가 발생한다. 특히 지연쓰기 파일의 축약이 빈번하게 발생할수록 더 커지게 된다. 따라서 추후에는 데이터베이스 내에 지연쓰기를 위한 공간을 유지하여 이러한 오버헤드를 줄일 수 있도록 개선할 계획이다.

참 고 문 헌

[1] Michael Wu and Willy Zwaenpoel, "eNVy: A Non-Volatile, Main Memory StorageSystem," Proceeding of 6th Symposium on Architectural Support for Programming Languages and Operating System, pp. 86-87, 1994.

[2] Intel Corporation, Understanding the Flash Translation Layer (FTL) Specification, http://developer.intel.com/

[3] Intel Corporation, 3 Volt Synchronous Intel StrataFlash Memory, http://www.intel.com/.

[4] Suman Nath and Aman Kansal, "FlashDB: dynamic self-tuning database for NAND flash. Information Processing in Sensor Networks," pp. 410-419, Cambridge, Massachusetts, 2007.

[5] D. Woodhouse, JFFs: The journaling flash file system, http://sources.redhat.com/jffs2/jffs2.pdf

[6] Wookey, "Yaffs - A file system designed for nand flash," Linux Conference and Tutorials. Leeds, UK, 2004.

[7] M. Rosenblum and J.K. Ousterhout, "The Design and Implementation of a Log-Structured File System," ACM Transaction on Computer Systems, Vol.10, No.1, pp. 26-52, 1992.

[8] Chin-Hsien Wu, Li-Pin Chang and Tei-Wei Kuo, "An Efficient B-Tree Layer for Flash-Memory Storage Systems," Real-Time Computing Systems and Applications, pp. 409-430, 2003.

[9] Song Lin, Demetrios Zeinalipour-Yazti, Vana Kalogeraki, Dimitrios Gunopulos, and Walid A. Najjar, "Efficient indexing data structures for flash-based sensor devices," *ACM Transactions on Storage*, Vol.2, No.4, pp. 468-503, 2006.

[10] Christophe Bobineau, Luc Bouganim, Philippe Pucheral, and Patrick Valduriez, "PicoDBMS: Scaling down Database Techniques for the Smart Card," The 26th International Conference on Very Large Databases, pp. 10-20, Cairo, Egypt, 2000.

[11] Michael Owebs, The Definitive Guide to SQLite, Apress, 2006.

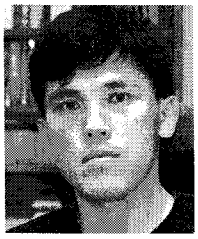
[12] J.Kim, J.M. Kim, S.H. Noh, S.L. Min, and Y. Cho, "A Space-Efficient Flash Translation Layer for Compact Flash System," *IEEE Transactions on Consumer Electronics*, Vol.48, No.2, pp. 366-375, 2002.

[13] Sang-Won Lee and Bongki Moon, "Design of flash-based DBMS: an in-page logging approach," Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 55-66, Beijing, China, 2007.

[14] Jim Gray and Andreas Reuter, Transaction Processing: Concepts and Techniques, Elsevier Science & Technology Books, 1992.

송 하 주

1993년 서울대학교 컴퓨터공학과 졸업 (공학사)
1995년 서울대학교 컴퓨터공학과 졸업 (공학 석사)
2001년 서울대학교 전기컴퓨터공학부 졸업 (공학 박사)
2003년 ㈜아이티포웹 부장
현재 부경대학교 전자컴퓨터정보통신공학부 조교수
관심분야 : 데이터베이스, 유비쿼터스 센서 네트워크



권 오 흠

1988년 서울대학교 컴퓨터공학과 졸업 (공학사)
1991년 KAIST 전산학과 (공학석사)
1996년 KAIST 전산학과 (공학박사)

현재 부경대학교 전자컴퓨터정보통신공학부 교수
관심분야 : 알고리즘 설계 및 분석, 분산 컴퓨팅, 유비쿼터스 센서 네트워크 등