ISSN 2383-630X(Print) / ISSN 2383-6296(Online)
Journal of KIISE, Vol. 42, No. 2, pp. 169-176, 2015. 2
http://dx.doi.org/10.5626/JOK.2015.42.2.169

데이터베이스 시스템의 원자성 쓰기 보장을 위한 스토리지 I/O 서브시스템

(Storage I/O Subsystem for Guaranteeing Atomic Write in Database Systems)

한 규 화 [†] 신 동 군 [††] 김 용 석 [†††]
(Kyuhwa Han) (Dongkun Shin) (Yongserk Kim)

요 약 최근 데이터베이스 관리 시스템에서는 이중 쓰기 버퍼 기법의 단점을 보완할 수 있는 원자성 쓰기 기법이 제시되었다. 원자성 쓰기 기법을 지원하기 위해서는 파일시스템과 I/O 스케줄러 등의 I/O 서브시스템과 SSD의 원자성 쓰기의 보장이 필요하다. 본 연구에서는 MariaDB의 데이터 쓰기의 기본 단위를 사용하여, 쓰기 단위 연속 블록 할당 기법, 플래그 전달을 통한 I/O 명령 병합 방지 기법, 원자성 쓰기를 지원하는 SSD를 통해 MariaDB의 원자성 쓰기를 지원하는 연구를 진행하였다. Tpcc-mysql와 SysBench를 사용하여 MariaDB에 제안된 기법을 적용한 결과, 원자성 쓰기 기법의 성능을 평가한 결과, 데이터베이스의 처리량이 이중 쓰기 버퍼 기법을 사용한 MariaDB 대비 약 40%~50% 향상된 것을 확인하였다.
키워드: 데이터베이스, 원자성 쓰기, I/O 서브시스템, SSD

Abstract The atomic write technique is a good solution to solve the problem of the double write buffer. The atomic write technique needs modified I/O subsystems (i.e., file system and I/O schedulers) and a special SSD that guarantees the atomicity of the write request. In this paper, we propose the writing unit aligned block allocation technique (for EXT4 file system) and the merge prevention of requests technique for the CFQ scheduler. We also propose an atomic write-supporting SSD which stores the atomicity information in the spare area of the flash memory page. We evaluate the performance of the proposed atomic write scheme in MariaDB using the tpcc-mysql and SysBench benchmarks. The experimental results show that the proposed atomic write technique shows a performance improvement of 1.4~1.5 times compared to the double write buffer technique.
Keywords: database, atomic write, i/o subsystem, solid state disk

· 본 논문은 2013년도 정부(교육부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 것임(2013R1A1A2A10013598)
· 이 논문은 2014 한국컴퓨터종합학술대회에서 '데이터베이스의 원자성 쓰기 보장을 위한 파일시스템 및 SSD 개발'의 제목으로 발표된 논문을 확장한 것임

† 비 회 원 : 성균관대학교 전자전기컴퓨터공학과
hgh6877@skku.edu
†† 종신회원 : 성균관대학교 컴퓨터공학과 교수(Sungkyunkwan Univ.)
dongkun@skku.edu
(Corresponding author임)
††† 비 회 원 : 성균관대학교 산학협력단 교수
yskimasi@skku.edu

논문접수 : 2014년 8월 22일
(Received 22 August 2014)
논문수정 : 2014년 10월 16일
(Revised 16 October 2014)
심사완료 : 2014년 10월 22일
(Accepted 22 October 2014)

정보과학회논문지 제42권 제2호(2015. 2)

1. 서 론

스마트폰과 휴대용 전자기기의 사용이 증가함에 따라 클라우드 서비스와 사용자의 정보를 관리하는 서버의 사용도 증가하고 있다. 클라우드 시스템과 서버와 같은 커다란 정보를 저장하는 시스템은 데이터를 무결성 있게 관리하는 것이 중요하다. 이러한 시스템은 데이터베이스 관리 프로그램을 사용하여 데이터의 ACID(원자성, 일관성, 고립성, 영속성)[1]를 보장한다. 오픈소스 데이터베이스 관리 프로그램의 대표적인 예로는 MySQL과 MariaDB가 있다[2,3].

MySQL과 MariaDB는 데이터의 무결성을 보장하기 위하여 이중 쓰기 버퍼 기법을 사용한다. 이 기법은 데이터 파일의 원자적인 변경을 위하여 이중 쓰기 버퍼 영역에 데이터의 쓰기가 완료된 후 파일의 원본 위치에 같은 데이터를 덮어쓴다. 이 기법을 사용하면 데이터의 원자성을 보존할 수 있으나, 같은 데이터를 서로 다른 두 영역에 써야 하기 때문에 쓰기 양이 2배로 증가한다.

이러한 이중 쓰기 버퍼 기법을 개선하기 위하여 원자성 쓰기(atomic write) 기법이 소개되었다[4]. 이 기법은 쓰기 명령에 대한 원자성을 보장해주는 SSD를 사용하여, 데이터 파일을 이중 쓰기 버퍼를 거치지 않고 다이렉트 I/O를 통해 데이터의 원본 위치에 바로 쓰기를 진행한다. 이 기법을 사용하면 이중 쓰기 버퍼 기법에 비해 쓰기 양을 반으로 줄일 수 있지만, 내부적으로 I/O 명령의 원자성을 보장하는 SSD 장치가 필요하다.

FusionIO에서는 SSD물리페이지의 스페어 영역에 쓰기 명령의 정보를 저장하는 방법을 통하여 쓰기의 원자성을 보장하는 SSD를 제안하였고, DFS[5]라는 독자적인 파일시스템을 통해 MariaDB에게 원자적인 쓰기를 제공하였으며, 그 결과 이중 쓰기 버퍼를 사용하는 기법에 대비해 1.1~1.2배의 성능향상을 보였다[4]. MariaDB는 DFS의 ioctl을 통해 특정 파일을 원자성 쓰기 파일로 세팅할 수 있으며, DFS와 SSD가 원자성 쓰기 파일에 대해 원자적인 쓰기를 보장한다. 그러나, MariaDB의 원자성 쓰기는 FusionIO의 SSD와 DFS에서만 지원이 가능하다. 또한 FusionIO의 DFS는 소스가 공개되지 않았고, SSD는 멀티 칩 환경에서 데이터를 병렬적으로 처리하기에 문제가 있다. 또한, 관련 연구에서 기법에 대해 자세하게 서술하지 않았다. 본 연구에서는 리눅스의 고유 파일시스템인 EXT4 파일시스템[6], CFQ I/O 스케줄러를 수정하여, 데이터의 원자성 쓰기를 보장하는 기법을 제시한다. 또한, 원자성 쓰기가 가능한 SSD의 데이터 처리 기법도 제시한다.

본 기법에서 제안한 스토리지 I/O 서브시스템을 적용한 결과, 이중 쓰기 버퍼 기법에 비해 1.5배의 성능을 보이는 것을 확인하였다. 2장에서는 본 논문의 기법과 관련된 연구들에 대해 소개한다. 3장에서는 원자성 쓰기를 지원하는 SSD의 디자인에 대해 설명한다. 4장에서는 데이터베이스의 쓰기 단위에 정렬된 할당과 원자성 쓰기 명령 전달을 위해 수정한 EXT4와 I/O 스케줄러의 구현 내용을 설명한다. 5장에서는 성능 평가 실험에 대해 설명하고, 6장에서는 결론을 제시한다.

2. 배 경

2.1 제안 기법의 동기

MariaDB의 데이터 원자성 쓰기 기법을 설계하기 위해, 우선 MariaDB의 데이터 파일의 쓰기 패턴을 분석하였다. 그림 1은 MariaDB에서 TPC-c 워크로드를 수행하면서 MariaDB의 데이터 파일에 대한 정보를 구분하기 위해, ftrace를 통해 측정한 DB의 데이터 파일에 대한 쓰기 명령의 크기를 보여준다. x축은 데이터 파일에 발생한 쓰기 명령의 크기를, 왼쪽의 y축은 데이터의 개수를, 오른쪽의 y축은 전체 쓰기량의 누적 분포를 보여준다. 쓰기 명령의 크기는 16KB의 배수로 나타나는 것을 확인할 수 있다. 그래프에서 512KB와 같은 큰 사이즈의 쓰기는 이중 쓰기 버퍼 영역으로 전달되는 쓰기 명령이며, 전체 쓰기 명령의 반을 차지한다. 데이터 영역에 적히는 쓰기 명령은 주로 16KB와 32KB 크기가 많은 것을 확인할 수 있다. 본 연구에서는 16KB를 기본 단위로 쓰기 명령을 수행하는 데이터 파일과 데이터 파일이 아닌 파일을 구분하고, MariaDB가 데이터 영역에 16KB단위의 쓰기 명령만을 요청하는 특성을 이용하여 호스트와 SSD 간의 인터페이스를 규정한다. 이를 이용하여, 적은 오버헤드로 MariaDB의 데이터의 원자성을 보장하는 기법을 제시한다.

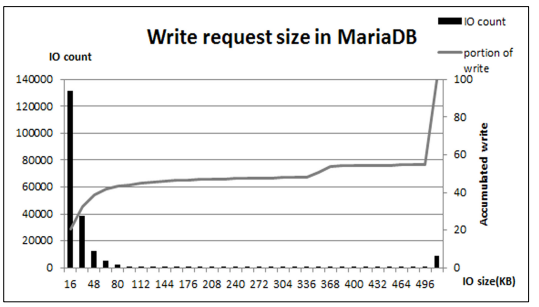

그림 1 MariaDB 쓰기 명령 크기
Fig. 1 Size of write request in MariaDB

2.2 관련 연구

데이터의 ACID를 보장하기 위한 연구들은 운영체제 영역에서부터 디바이스 영역에 걸쳐 다양하게 진행되어

왔다. 운영체제를 수정하는 연구들은 주로 파일시스템을 수정하여 데이터와 메타데이터를 트랜잭션 기반으로 처리하는 연구[7,8]들과 운영체제에 모듈을 추가하여 파일에 대한 접근을 트랜잭션 기반으로 처리하도록 하는 연구[9]들이 있다. 하지만 디바이스가 전통적으로 섹터 단위(512바이트)에 대한 원자성만을 보장하기 때문에, 위의 기법[7-9]들은 데이터 접근에 대해 섹터단위로 트랜잭션을 관리해야 하는 오버헤드가 존재한다.

FusionIO[4]에서는 SSD 장치 내부에서 플래시 메모리를 관리하는 FTL의 기능들을 호스트 시스템으로 가져오고, 플래시 메모리를 제어하는 기능만을 포함한 SSS (Solid State Storage)라는 장치를 사용한다. FusionIO의 호스트 시스템은 하나의 트랜잭션에 속한 서로 연속하지 않은 여러 페이지를 하나로 묶어 원자적 쓰기 명령을 생성한다. FusionIO의 SSD는 로그 구조의 FTL을 사용하여, 원자적 쓰기 요청을 물리적으로 연속한 페이지에 할당한다. 원자적으로 쓰여져야 하는 데이터의 부분적인 쓰기를 방지하기 위하여, 플래시 메모리 페이지마다 1개의 비트를 할당하여, 여러 페이지로 이루어진 쓰기 명령의 마지막 페이지의 해당 비트를 1로 세팅하는 방법을 사용하여, 모든 쓰기가 완료된 쓰기 명령을 구분한다.

FusionIO의 기법은 데이터의 원자성을 보장하기 위한 몇 가지 오버헤드가 존재한다. 원자적 쓰기 명령이 다른 명령과 섞이는 것을 방지하기 위하여, 원자적 쓰기 명령의 마지막 페이지가 쓰이기 전에 다른 쓰기 명령을 처리하지 못한다. 이로 인해, SSD의 모든 칩을 완전하게 사용하지 못하게 된다. 또한, FusionIO는 독자적인 파일시스템과 SSS에서만 동작하는 문제를 가진다. 본 기법에서는 원자성 쓰기 명령의 페이지들의 논리 주소를 연속하도록 제한하고, 물리 페이지 할당에 제한을 두지 않아 SSD의 모든 칩을 사용할 수 있으며 로그 구조의 FTL이 아닌 FTL에서도 사용이 가능하다. 또한 리눅스의 고유 파일시스템인 EXT4 파일시스템과 I/O스케줄러를 수정하여, 모든 시스템에서 사용이 가능하다.

LightTX[10]는 여러 쓰기 명령으로 구성된 트랜잭션의 원자성을 보장한다. LightTX는 트랜잭션의 정보를 스페어 영역에 저장하여, 트랜잭션 처리 중 전원이 차단되는 경우에도 트랜잭션의 원자성을 보장하도록 하였다. 또한, 플래시 메모리 영역을 모든 트랜잭션이 완료된 블록과 트랜잭션이 진행중인 블록으로 구분하여, 모든 트랜잭션이 완료된 블록들을 복구 작업에서 제외시킨다. 하지만, LightTX의 경우 트랜잭션의 정보(12bytes)를 플래시 메모리의 스페어 영역에 저장하는 방법을 사용하기 때문에, 스페어 영역이 부족한 SSD에는 적용하기 힘든 기법이다. 본 기법에서는 스페어 영역에 저장되는 데이터의 크기를 최소화하여 7개의 비트만을 사용하여, 데이터의 원자성을 보장한다. 또한, LightTX는 여러 쓰기 명령으로 구성된 트랜잭션 단위로 원자성을 지원하는 기법으로서 하나의 쓰기 명령에 대한 원자성을 지원하는 본 기법과 차이가 있다.

3. 원자성 쓰기 보장 SSD

3.1 기존 SSD의 부분적 쓰기 문제

데이터베이스에서 원자성 쓰기를 통한 데이터의 무결성을 보장하기 위해서는 SSD가 쓰기 명령을 원자적으로 처리하는 기능을 지원해야 한다. 그러나, SSD의 페이지 단위(4KB 또는 8KB)와 데이터베이스의 데이터 쓰기 명령의 기본 단위(MariaDB의 경우16KB)가 서로 다르기 때문에 MariaDB의 데이터는 여러 물리 페이지(2개 또는 4개)에 나누어져 저장된다. 이 때 여러 페이지들은 서로 다른 플래시 메모리 칩에 저장되며 동시에 처리된다.

여러 페이지로 이루어진 하나의 I/O 명령은 다음 두 가지 이유에 의해 부분적인 쓰기가 발생할 수 있다. 첫 번째는 SSD의 매핑 테이블의 갱신이 원자적으로 이루어지지 않기 때문이다. 페이지 단위 FTL의 경우, SRAM의 크기가 매핑 테이블 전체의 크기보다 작기 때문에, 매핑 테이블의 일부를 SRAM에 캐싱하여 사용한다. 이러한 방법으로 인해 SRAM이 부족하면 일부의 매핑 테이블만 플래시 메모리에 갱신이 발생할 수 있으며, 이로 인해 하나의 I/O 명령의 모든 페이지가 저장되기 전에 일부 매핑 테이블의 갱신이 진행될 수 있다.

두번째 이유는 갑작스럽게 전원이 차단된 경우 매핑 정보를 복구하는 SPOR (Sudden Power Off Recovery) 작업이 페이지 단위로 진행된다는 것이다. SSD는 일반적으로 SPOR을 위하여 물리 페이지의 스페어 영역에 논리 페이지 번호(LPN; Logical Page Number)를 저장한다. 그리고, 갑작스럽게 전원이 차단되어 매핑 정보가 플래시 메모리에 저장되지 못한 경우에는 스페어 영역에 저장해둔 논리 페이지 번호를 이용하여 매핑 테이블을 복구한다. 하지만, 여러 페이지로 이루어진 원자성 쓰기 명령의 일부 페이지 만 쓰기 동작이 완료되었을 때에는 쓰기 동작이 완료된 페이지에 대해서만 복구 작업이 진행된다.

그림 2(a)는 SSD의 SPOR 동작을 보여준다. 사용자는 SSD에 (페이지, 개수)로 이루어진 일반 쓰기 명령 W(300, 1)과 원자적으로 처리되어야 하는 쓰기 명령 aW(399, 4), aW(500, 8), aW(403, 4)를 요청하였다. SSD는 해당 명령들을 처리하던 중 갑작스럽게 전원이 차단되어 물리 페이지 107과 113은 쓰기가 완료되지 못하였으며, 물리 페이지 100~106, 108~112는 매핑의 갱

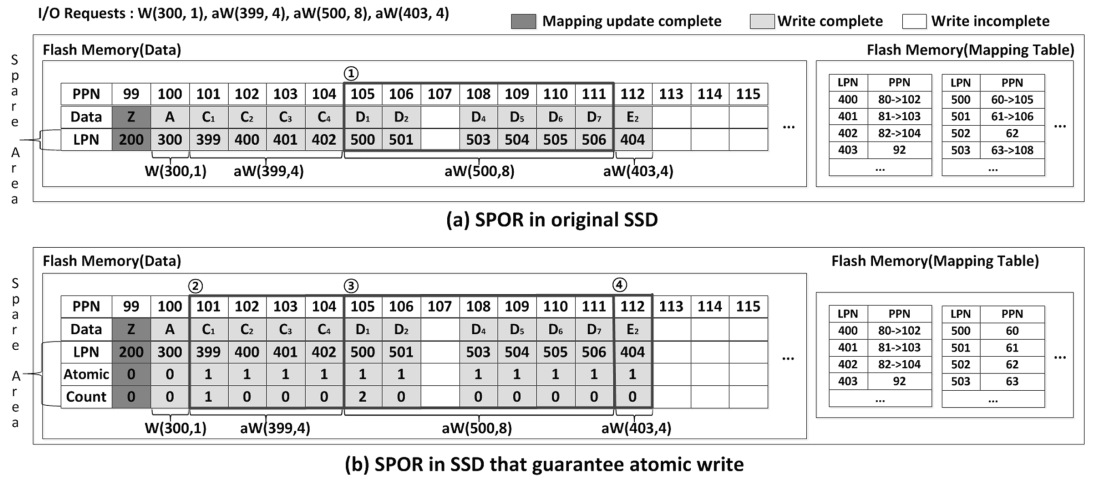

그림 2 원자성 쓰기 보장 SSD의 SPOR 과정

Fig. 2 SPOR in SSD that guarantee atomic write

신이 적용되지 않아 SPOR의 대상이 된다. 이에 대한 복구 작업은 물리 페이지 100의 스페어 영역을 확인하여 논리 페이지 300의 데이터가 저장되었음을 알고 매핑 테이블을 갱신한다. 또한, 논리 페이지 399~402의 매핑 정보도 같은 방법으로 갱신한다. 하지만, 그림 2의 ①과 같이 하나의 I/O 명령으로 전달된 32KB 데이터가 완전히 쓰이지 않은 경우, D3과 D8의 데이터를 제외한 데이터의 매핑 테이블 갱신만 진행되어 원자성 쓰기 명령 aW(500, 8)와 aW(403,4)의 부분적인 쓰기가 발생한다.

3.2 원자성 쓰기 인터페이스

본 연구에서는 위에서 언급한 기존 SSD에서 여러 페이지로 이루어진 I/O 명령에 대한 원자적인 쓰기를 방해하는 두 가지 원인을 해결한다. 먼저, SPOR의 문제를 해결하기 위하여, 원자성 쓰기 명령이라는 인터페이스를 생성하여 일반 쓰기 명령과 구분할 수 있도록 하였다. SSD는 SPOR이 두 가지 명령을 구분하기 위하여 스페어 영역에 1개의 비트를 할당하여, 원자성 쓰기 명령과 일반 쓰기 명령을 구분하는 플래그로 사용한다.

두 번째로, 매핑 정보의 갱신이 원자적으로 처리되지 않는 문제를 해결하기 위하여, 원자성 쓰기 명령이 플래시 메모리에 모두 적힌 것을 확인한 후, 매핑 테이블에 반영하도록 하였다. 이렇게 하면 매핑 테이블에 매핑 정보를 반영하는 중간에 전원이 차단된 상황에서, 데이터가 플래시 메모리에 모두 적혀있기 때문에 SPOR로 매핑의 복구가 가능하다.

본 연구에서 제안하는 원자성 보장 쓰기 명령은 연속된 논리 주소를 가진 페이지들에 대해 원자성을 보장한다. 파일시스템은 데이터베이스의 쓰기 단위를 고려하여 데이터가 두 영역에 나누어 할당되지 않도록 지원해야 한다. 또한, I/O 스케줄러는 데이터베이스의 I/O 명령이 다른 종류의 I/O 명령과 병합되어, I/O 명령의 크기가 데이터베이스의 쓰기 단위의 배수가 아니게 되는 것을 방지해야 한다. I/O 명령의 크기가 데이터베이스의 쓰기 단위의 배수가 아니게 되면, SSD는 해당 I/O를 SPOR을 통해 복구하지 못한다. 이 문제에 대해서는 3.3절에서 자세히 설명한다. 본 연구에서 제안한 인터페이스는 논리 페이지의 할당에만 제약을 두고, SSD의 물리 페이지 할당에 영향을 주지 않기 때문에 SSD의 성능에 영향을 주지 않는다.

3.3 I/O 명령 원자성 보장 SPOR

그림 2(b)는 원자성 쓰기 보장 SSD의 SPOR 모습을 보여준다. 아래의 설명은 MariaDB의 쓰기 단위인 16KB를 기준으로 설명한다. 본 기법은 16KB의 배수의 크기를 가진 원자성 쓰기 명령과 일반 쓰기 명령을 서로 다른 방법으로 복구 작업을 진행한다. 두 가지 쓰기 명령을 구분하기 위하여 페이지의 스페어 영역에 쓰기 명령의 종류를 구분하는 Atomic 플래그를 추가하였다. 물리 페이지는 Atomic 플래그가 1로 세팅된 경우 원자성 쓰기 명령의 일부이고, 세팅되지 않은 경우 일반 쓰기 명령이다. 예를 들어 논리 페이지 300은 일반 쓰기 명령이고, 논리 페이지 399~402는 16KB 크기의 원자성 쓰기 명령이다.

SSD는 원자성 쓰기 명령의 크기를 저장하기 위하여 물리 페이지의 스페어 영역에 count필드를 추가하였다. 원자성 쓰기 명령의 첫 번째 물리 페이지의 count 필드에는 데이터베이스의 데이터 개수가 저장된다. 예를 들어 그림 2(b)의 논리 페이지 399의 경우, 데이터베이스의 데이터 1개에 대한 원자성 쓰기 명령이므로 count

필드에는 1이 저장된다. 나머지 물리 페이지의 count 필드는 0으로 세팅하여, 첫 번째 물리 페이지와 구분하였다. 일반 쓰기 명령의 물리 페이지의 count 필드는 사용하지 않는다. Count 필드에 할당하는 비트는 데이터베이스의 쓰기 단위에 따라 개수가 달라진다. 데이터베이스의 쓰기 단위가 16KB인 경우 리눅스 I/O 서브시스템을 통해 전달되는 I/O의 최대 크기는 512KB이며, 이는 16KB 데이터 32개를 포함할 수 있다. 따라서 count 필드는 6개의 비트를 할당하면 리눅스 I/O 서브시스템의 모든 원자성 쓰기 명령의 크기를 표현할 수 있다. 만약 원자성 쓰기 명령의 크기가 16KB의 배수가 아닌 경우에는 본 기법을 통해서 정확한 I/O 명령의 크기를 표현하지 못한다.

SSD는 atomic 플래그를 확인하여 일반 쓰기 명령과 원자성 쓰기 명령을 서로 다른 방법으로 복구작업을 진행한다. 먼저, 일반 쓰기 명령에 대해서는 기존의 복구 방법을 그대로 사용한다. 하지만, Atomic 플래그가 1로 세팅된 경우의 복구 작업은 두 단계로 나누어 진행된다. 첫 번째 단계는 스페어 영역의 count 필드를 사용하여, I/O 명령의 헤드 페이지를 탐색하는 작업이다. 그림 1의 (b)의 경우 물리 페이지 101과 105가 헤드페이지임을 찾아낸다. 두 번째 단계는 해당 헤드페이지를 포함하는 I/O 명령이 모두 쓰였는지를 확인하는 작업이다. 그림 1의 ②는 논리 페이지 399~402의 물리 페이지들이며, 물리 페이지 101의 count 필드의 값이 1이므로 해당 I/O 명령의 데이터가 모두 적혔으므로 매핑 테이블을 갱신한다. 그림 2의 ③은 논리 페이지 500~507의 물리 페이지들이며, 물리 페이지 105의 count 필드의 값이 2 이므로 요청의 크기는 32KB이다. 논리 페이지 502와 507의 데이터가 모두 쓰여지지 않은 페이지이므로 논리 페이지 500~507의 매핑 테이블 갱신을 하지 않는다. 또한, 그림 2의 ④와 같이 원자성 쓰기 요청의 첫 물리 페이지의 쓰기가 완료되지 않은 경우에도 매핑 테이블 갱신을 하지 않는다. 이를 통해 그림 2(a)와 같은 부분적 데이터 갱신이 발생하지 않도록 보장한다.

SSD의 SRAM에 로딩된 매핑 테이블 페이지를 플래시 메모리에 갱신하지 않아도 SPOR 작업을 통해 매핑 테이블을 복구하는 것이 가능하다. 하지만, 매핑 테이블을 오랫동안 갱신하지 않으면, SPOR 작업을 통해 매핑을 복구하는 영역이 넓어지기 때문에 SPOR 작업에 많은 시간이 소비되는 문제점이 있다. 일반적으로 SSD는 사용하는 물리 블록이 변경될 때, 매핑 테이블 갱신 작업을 진행한다. 본 연구에서 제안하는 SSD도 기존의 SSD와 같은 시점에 매핑 테이블 갱신 작업을 수행한다.

본 연구에서는 호스트와 SSD 사이에 새로운 인터페이스를 제안하였다. 해당 인터페이스를 SSD에게 제공하기 위해, EXT4 파일시스템과 I/O 스케줄러를 수정하였다. EXT4의 수정 내용은 4.2절에서, I/O 스케줄러의 수정 내용은 4.3절에 서술하였다.

4. 리눅스 I/O 서브 시스템

4.1 기존 기법의 문제점

MariaDB는 데이터의 관리 단위로 16KB를 사용한다. 그림 3(a)는 EXT4의 블록 할당 기법을 보여준다. 데이터베이스에는 A, B, C, D, E라는 16KB 데이터가 5개 존재하며, MariaDB의 로그 데이터F와 G라는 4KB 데이터가 존재하며, MariaDB가 관리하지 않는 일반 파일에 H, I라는 8KB 데이터 2개가 존재한다고 가정한다. 데이터 B의 할당의 경우, 이미 할당된 일반 파일에 의해 분리된 2개의 영역을 할당받아 B1과 B2라는 2개의 데이터로 분리되어 저장된다. 이 경우, 데이터베이스가 데이터 B에 대한 I/O 명령을 발생하였을 때 앞에서 정의한 인터페이스를 지키지 못하는 상황이 발생한다.

또한 데이터 B1과 B2는 서로 다른 물리 영역에 할당되었으므로 스케줄러에 의해 병합될 수 없으며, 2개의 I/O 명령으로 처리된다. 본 연구에서 제시한 SSD는 연속된 논리 주소를 가진 데이터의 대한 원자성을 보장하지만, 여러 개의 I/O를 하나의 트랜잭션으로 관리하는 기능을 지원하지 않기 때문에 2개로 나누어진 I/O는 원자적으로 처리할 수 없다. 또한 일반 파일 데이터 I와

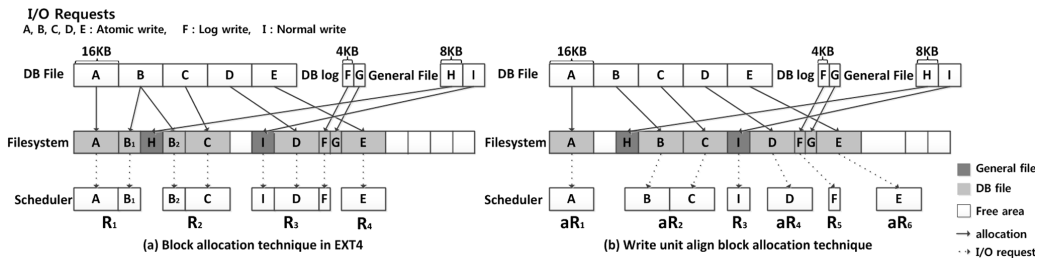

그림 3 쓰기 단위 블록할당 기법
Fig. 3 Write unit align block allocation technique

데이터베이스의 데이터 D 그리고 DB 로그 데이터 F가 같은 시점에 I/O 계층으로 전달되면, 스케줄러에 의해 병합되어 하나의 I/O 명령으로 SSD에 전달 된다. 이 경우 두 종류의 쓰기 명령이 병합되어 원자성 쓰기 명령으로 전송되지 못한다. 따라서 본 연구에서는 파일시스템과 스케줄러를 수정하여 본 연구에서 제시한 SSD가 원자성 쓰기를 보장할 수 있도록 수정하였다. 본 논문에서 제안한 원자성 쓰기를 만족하기 위해서는 원자성 쓰기를 지원하는 파일시스템과 I/O 스케줄러, SSD를 동시에 사용하여야만 보장이 가능하다.

4.2 원자성 쓰기 지원 파일시스템

EXT4 파일시스템을 사용하는 모든 파일에 대해 원자성 쓰기 요청이 수행되지는 않는다. 예를 들어, MariaDB의 경우 원자성 쓰기가 필요한 데이터 파일과 원자성 쓰기가 필요하지 않은 redo_log 파일을 동시에 사용한다. 본 연구에서는 원자성 쓰기가 필요한 MariaDB의 데이터 파일만을 원자성 쓰기 파일로 설정하며, 원자성 쓰기 파일과 일반 파일에 대해 서로 다른 방법으로 블록을 할당한다.

일반 파일의 경우, 기존 EXT4의 블록 할당 기법과 같은 방법으로 블록을 할당한다. EXT4는 큰 파일이 블록을 할당하는 경우, 할당 요청 크기보다 많은 영역을 미리할당하여 이후의 할당을 미리할당 영역에서 처리한다. EXT4는 ext4_mb_normalize_request 함수를 통해 새로 할당하는 영역의 시작 주소와 크기를 설정한다. 본 기법에서는 ext4_mb_normalize_request 함수를 수정하여 할당하는 영역의 크기를 16KB의 배수가 되도록 수정하였다. 실제 할당은 ext4_mb_simple_scan_group함수와 ext4_mb_complex_scan_group 함수를 통해 EXT4의 buddy 구조체를 탐색하여 이루어진다. EXT4의 buddy 구조체는 2^n개의 연속적으로 비어있는 영역에 대해 관리한다. 본 기법은 EXT4의 buddy를 그대로 사용하기 때문에 연속한 영역을 찾는 추가적인 오버헤드는 존재하지 않는다. ext4_mb_normalize_request 함수에서 원자성 쓰기 할당 요청을 구분하기 위하여, VFS의 do_direct_IO 함수에서 생성된 *atomic* 플래그를 전달한다. *atomic* 플래그는 ext4_allocation_cotext 구조체와 ext4_allocation_request 구조체를 거쳐 ext4_mb_normalize_request함수로 전달된다.

그림 3(b)는 수정된 블록 할당 기법을 보여준다. 기존의 EXT4의 블록 할당 기법에서는 미리 할당된 블록에 연속된 블록이 다른 파일에 의해 할당된 경우, 데이터 B가 연속하지 않은 2개의 구간에 나누어져 저장되었지만, 본 연구에서 제안하는 기법에서는 데이터 B를 연속된 영역에 할당한다. 위의 기법을 통해 본 연구에서 제시한 SSD의 원자성 쓰기가 가능하도록 하였다.

EXT4 파일시스템을 오래 사용하게 되면 파일의 할당과 해지가 반복되어 EXT4가 관리하는 데이터 영역의 단편화가 진행된다. 단편화가 심하게 진행된 경우, 원자성 쓰기 파일은 연속한 16KB의 영역을 할당 받지 못할 수 있다. 이러한 단편화 문제를 해결하기 위해서는 EXT4에서 제공하는 온라인 단편화 제거 기법을 사용하면 된다.

4.3 원자성 쓰기 지원 I/O 스케줄러

그림 3(b)는 수정된 I/O 스케줄러 기법을 보여준다. 16KB 원자성 쓰기 명령의 경우 특정 플래그를 가진 채 I/O 스케줄러로 전달된다. 플래그를 가진 명령은 원자적 명령(그림 3의 aR)으로 처리되고, 플래그를 가지지 않은 명령은 일반 명령(그림 3의 R)으로 처리된다. 스케줄러는 두 명령 간의 병합 처리를 하지 않는다. 예를 들어, C의 데이터는 일반 파일인 I와 주소가 연속되지만 병합 작업을 거치지 않고 독립적으로 처리된다. 또한 데이터 B와 C는 연속된 원자성 쓰기 명령이므로 병합을 통해 하나의 I/O 명령으로 처리된다.

본 기법에서는 VFS의 do_direct_IO함수에서 생성한 *atomic* 플래그를 dio 구조체를 통해 I/O 스케줄러에 전달한다. I/O 스케줄러는 blk_rq_merge_ok 함수를 통해 request 구조체와 bio (dio) 구조체의 병합이 가능한지를 판단한다. 본 기법에서는 blk_rq_merge_ok 함수를 수정하여, 원자적 명령과 일반 명령의 병합을 방지하고, 원자적 명령끼리의 병합과 일반 명령끼리의 병합을 가능하도록 하였다.

5. 실 험

5.1 실험 환경

실험에서는 Intel i5-4670 쿼드 코어CPU, 4GB 메모리를 가진 호스트 PC를, 커널 버전 3.8.0의 우분투 12.10 OS를 수정하여 사용하였으며, tpcc-mysql[11]과 Sysbench[12]를 사용하여 성능을 평가하였다. Tpcc-mysql은 percona에서 개발한 tpcc 워크로드를 테스트하는 벤치마킹 툴이다. SysBench는 OLTP(On-Line Transaction Processing) 워크로드를 기반으로 MariaDB의 성능을 측정하는 벤치마킹 툴이다. 실험에는 수정된 I/O 서브시스템을 사용하였으며, 매 실험을 진행할 때마다 EXT4 파일시스템을 포맷하여 단편화 발생을 억제하였다. 원자성 쓰기 지원 SSD가 특별한 성능 저하가 없으므로 기존 SSD(Samsung SSD 840 PRO)를 원자성 쓰기가 보장된다는 가정으로 사용하였다.

MariaDB는 버전 5.5.34를 사용하였고, 메모리에 1GB의 버퍼 영역을 할당하였으며, 트랜잭션의 정보를 저장하는 로그 파일의 크기는 512MB로, 실험 시간은 10분으로 설정하였다. 이중 쓰기 버퍼를 사용하는 *double*,

이중 쓰기 버퍼를 사용하지 않는 *no_double*, 그리고 원자성 쓰기를 사용하는 *atomic* 에 대해 실험을 진행하였다. *no_double* 의 경우 부분적인 쓰기를 해결하지 못하는 기법이다.

5.2 DBMS 성능

그림 4는 TPC-c 워크로드에서 측정한 *atomic*, *no_double* 기법의 처리량을 *double* 기법의 처리량에 정규화하여 보여준다. x축은 데이터베이스의 크기와 connection의 개수를, y축은 처리량의 상대값을 보여준다. 데이터베이스의 크기는 5GB에서 30GB까지, connection의 개수는 20, 50, 100까지 바꾸어가며 성능을 평가하였다. *no_double* 과 *atomic* 이 이중 쓰기 영역과 원본 위치에 2번의 쓰기를 하는 *double* 에 비해 1.2~1.5배의 높은 성능을 보이는 것을 확인하였다. 이중 쓰기 버퍼가 *no_double* 에 비해 쓰기량이 2배이지만 성능이 반으로 줄지 않는 원인은 이중 쓰기 버퍼 영역에 대해서 연속적 쓰기가 발생하기 때문이다.

MariaDB의 원자성 쓰기를 사용하는 경우 *no_double* 보다 최대 1.5배의 향상된 성능을 보인다. 이 성능 차이는 원자성 쓰기에서 사용하는 다이렉트 I/O로 인한 것이다. 다이렉트 I/O를 사용하면 페이지 캐시를 거치지 않고 디바이스로 바로 명령 전달이 가능하다. 또한 쓰기 명령이 페이지 캐시를 거치지 않음으로 인해 메모리 사용량 또한 줄일 수 있다. 추가적으로 *no_double* 에서 보장하지 못하는 데이터의 무결성의 보장이 가능하다. 데이터베이스의 크기가 작은 경우, *no_double* 의 성능이 *atomic* 보다 1.1~1.2배 좋다. 이것은 *no_double* 의 경우 페이지 캐시에서 읽기 요청이 처리되고 있기 때문이다.

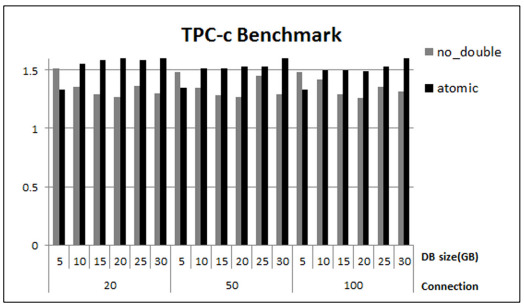

그림 4 TPC-c 처리량
Fig. 4 TPC-c throughput

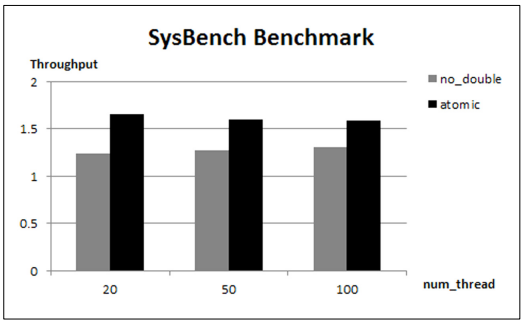

그림 5 SysBench 처리량
Fig. 5 SysBench throughput

그림 5는 50GB 크기의 DB에서 SysBench를 사용하여 측정한 *atomic*, *no_double* 기법의 처리량을 *double* 기법의 처리량에 정규화하여 보여준다. 쓰레드의 개수를 20, 50, 100으로 바꿔가면서 10분동안 OLTP 워크로드를 이용하여 성능을 측정하였다. *no_double* 의 경우 *double* 보다 1.25배의 성능을, *atomic* 의 경우 *double* 보다 1.6배의 성능을 보인다. 성능 차를 발생시키는 원인은 TPC-c 워크로드를 이용한 처리량 측정과 같다.

5.3 스토리지 쓰기량 감소

그림 6은 DB size 2.5GB, Connection 50에서 수행한 TPC-c 워크로드에 대한 *atomic*, *double*, *no_double* 기법의 트랜잭션당 디바이스의 쓰기량을 보여준다. 세 기법 모두 트랜잭션을 처리하는 동안 발생한 redo_log와 bin_log 파일에 대한 쓰기량은 동일하다. *Atomic* 과 *no_double* 의 데이터 쓰기량은 *double* 에 비해 1/2 정도로 작아진 것을 확인 할 수 있다. 스토리지 장치에 쓰여진 쓰기량의 경우, *atomic* 과 *no_double* 이 *double* 의 60% 보다 작음을 확인할 수 있다.

6. 결 론

데이터베이스에서 원자성 쓰기는 좋은 성능을 보이지만, 원자성 쓰기를 지원하는 공개된 파일시스템, SSD, I/O 스케줄러가 존재하지 않는다. 본 연구에서는 리눅스의 기본 파일시스템인 EXT4와 I/O 스케줄러를 수정하고, 원자성 쓰기를 지원하는 SSD를 디자인하여 데이터베이스의 원자성 쓰기의 ACID를 보장하였다. 본 연구에서는 호스트 시스템과 SSD 사이의 인터페이스를 정의하여, SSD 내부에서 적은 오버헤드로 I/O 명령의 원자성을 보장하도록 하였다. 실험 결과 본 기법은 이중 쓰기 버퍼 기법을 사용하는 기법에 비해 1.5배로 성능이 향상되었다.

References

[1] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.
[2] MySQL, Available: http://www.mysql.com/
[3] MariaDB, Available: https://mariadb.org/
[4] Xiangyong Ouyang, et al., "Beyond block I/O: Rethinking traditional storage primitives," *Proc. of the High Performance Computer Architecture (HPCA)*,

pp. 301-311, 2011.
[5] Josephson, William K., et al., "DFS: A file system for virtualized flash storage," *Proc. of the File and Storage Technologies (FAST)*, 2010.
[6] Avantika Mathur, et al., "The new ext4 filesystem: current status and future plans," *Proc. of the Linux Symposium*, pp. 21-33, 2007.
[7] E. Gal and S. Toledo, "A Transactional Flash File System for Microcontrollers," *Proc. of the USENIX Annual Technical Conference*, pp. 89-104, 2005.
[8] M. I. Seltzer, "Transaction Support in a Log-Structured File System," *Proc. of the Ninth International Conference on Data Engineering*, pp. 503-510, 1993.
[9] R. P. Spillane, S. Gaikwad, M. Chinni, E. Zadok, and C. P., "Enabling Transactional File Access via Lightweight Kernel Extensions," *Proc. of the FAST*, pp. 29-42, 2009.
[10] Youyou Lu, et al., "LightTx: A lightweight transactional design in flash-based SSDs to support flexible transactions," *Proc. of the 31th ICCD*, pp. 115-122, 2013.
[11] tpcc-mysql, Available: http://www.percona.com/about-us/mysql-white-paper/clustrix-tpcc-mysql-benchmark.
[12] SysBench, Available: http://sysbench.sourceforge.net

한 규 화

2013년 성균관대학교 컴퓨터공학과(학사) 2013년~현재 성균관대학교 전자전기컴퓨터공학과 석박통합과정. 관심분야는 임베디드 시스템, 파일시스템, 플래시 메모리

신 동 군

1994년 서울대학교 계산통계학과(학사) 2000년 서울대학교 전산과학과(석사). 2004년 서울대학교 컴퓨터공학부(박사) 2004년~2007년 삼성전자 소프트웨어 책임연구원. 2007년~현재 성균관대학교 정보통신대학 부교수. 관심분야는 임베디드 시스템, 실시간 시스템, 저전력 시스템

김 용 석

1983년 성균관대 전자공학 석사. 2002년 삼성전자 사내 명예박사. 2011년 동국대 정보통신공학과(박사). KAIST 반도체설계교육센터(IDEC) 운영위원. 2004년~2013년 삼성전자 상무. 2013년~현재 성균관대학교 산학협력중점교수. 관심분야는 임베디드 시스템, 파일 시스템, 저전력 시스템