KIPS Trans. Softw. and Data Eng.
Vol.12, No.3 pp.111~116
ISSN: 2287-5905 (Print), ISSN: 2734-0503 (Online)

https://doi.org/10.3745/KTSDE.2023.12.3.111

Boosting WiscKey Key-Value Store Using NVDIMM-N

Il Han Song† · Bo hyun Lee† · Sang Won Lee††

ABSTRACT

The WiscKey database, which optimizes overhead by compaction of the LSM tree-based Key-Value database, stores the value in a separate file, and stores only the key and value addresses in the database. Each time an fsync system call function is used to ensure data integrity in the process of storing values. In previously conducted studies, workload performance was reduced by up to 5.8 times as a result of performing the workload without calling the fsync system call function. However, it is difficult to ensure the data integrity of the database without using the fsync system call function. In this paper, to reduce the overhead of the fsync system call function while performing workloads on the WiscKey database, we use NVDIMM caching techniques to ensure data integrity while improving the performance of the WiscKey database.

Keywords : NVDIMM, Key-Value Database, Database Tuning

NVDIMM-N을 활용한 WiscKey 키-밸류 스토어 성능 향상

송 일 한† · 이 보 현† · 이 상 원††

요 약

LSM 트리 기반 Key-Value 스토어의 컴팩션(compaction)에 의한 오버헤드를 최적화한 WiscKey 데이터베이스는 값은 별도의 파일에 저장하고, 데이터베이스에는 키와 값의 주소만을 저장한다. 값을 저장하는 과정에서 데이터 무결성을 보장하기 위해 매번 fsync 시스템 호출 함수를 사용한다. 기존의 수행된 연구에서 fsync 시스템 호출 함수를 호출하지 않고 데이터 저장 작업을 반복적으로 수행해 본 결과 총 작업 수행시간이 최대 5.8배까지 줄어들었다. 하지만 fsync 시스템 호출 함수를 사용하지 않을 경우 데이터베이스의 데이터 무결성을 보장하기 어렵다. 본 논문에서는 WiscKey 데이터베이스에서 작업 수행 중 fsync 시스템 호출 함수의 오버헤드를 줄이기 위해 NVDIMM 캐싱 기법을 사용하여 데이터 무결성을 보장함과 동시에 WiscKey 데이터베이스의 성능을 향상시켰다.

키워드 : 비휘발성 메모리, 키-밸류 데이터베이스, 데이터베이스 튜닝

1. 서 론

지난 수십년 동안 SQL(Structured Query Language) 기반 관계형 데이터베이스 중심의 시장이 지속되어왔다. 그러나 대용량 데이터에 대한 요구가 급격히 증가하면서 최근 몇년간 대용량 데이터 처리에 강점을 가진 NoSQL과 같은 다양한 데이터베이스 모델이 등장하였다. NoSQL 데이터베이스 중 Key-Value 스토어는 전통적인 관계형 데이터베이스와는 달리 데이터를 키와 값 순서쌍 형태로 저장하며 Set()과 Get() 두 종류의 쿼리를 사용하여 데이터를 읽고 쓴다. 이와 같이 Key-Value 스토어는 데이터 저장 및 조회 원칙에 가장 초점을 맞추었기 때문에 간단한 데이터 모델을 대상으로 자주 읽고 쓰는 웹 인덱싱(indexing), e-커머스(commerce), 온라인 게임, 메시징(messaging), 소프트웨어 레포지토리(repository), 소셜 네트워크, 광고 등의 서비스를 제공하는 애플리케이션에 적합하다[1].

Key-Value 스토어 형태의 데이터를 저장하는 다양한 어플리케이션 중에서도 쓰기 집약적인 워크로드의 경우 LSM (Log-Structured Merge) 트리 구조를 사용한다[2]. Meta의 RocksDB, SQLite4, 그리고 구글의 LevelDB와 같이 LSM 트리 기반으로 구축된 다양한 분산 및 로컬 스토어는 대규모 프로덕션 환경에서 실사용 되고 있는데, 그 이유는 LSM 트리가 쓰기 연산에 최적화되어있기 때문이다. 업데이트 시 기존의 in-place 업데이트 방식이 아닌 out-of-place 업데이트 방식을 채택하여 별도의 공간에 순차적으로 쓴 뒤, 이후 컴팩션을 통해 하위 레벨에 flush를 일괄적으로 수행함으로써 변경사항을 반영하기 때문에 B-tree와 같은 다른 인덱싱 구조보다 쓰기 작업을 더욱 빠르게 수행하게 된다.

※ 이 논문은 2022년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임(No.2015-0-00314,비휘발성 메모리 기반 개방형 고성능 DBMS 개발).
※ 이 성과는 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. 2022R1A2C2008225).
† 비 회 원 : 성균관대학교 소프트웨어학과 석사과정
†† 정 회 원 : 성균관대학교 소프트웨어학과 교수
Manuscript Received : August 29, 2022
First Revision : September 29, 2022
Second Revision : November 2, 2022
Accepted : November 7, 2022
* Corresponding Author : Sang Won Lee(swlee@skku.edu)

LSM 트리 기반 LevelDB를 하드 디스크 위에서 사용할 때에는 컴팩션 작업이 큰 부하를 유발하지 않는다. 그러나 플래시 스토리지를 저장장치로 사용할 경우, 컴팩션 연산이 다음 두 가지 이유로 인해 큰 오버헤드를 유발한다. 첫 번째로, 컴팩션 쓰기 증폭(write amplification)이 악화된다. 쓰기 증폭은 제한된 NAND 플래시 셀의 수명 또한 비효율적으로 감소시킬 뿐만 아니라 과도한 가비지 컬렉션(Garbage Collection) 작업을 유발해 성능을 크게 저하시킨다[1]. 두 번째로, 컴팩션 수행 시간 동안 발생하는 읽기 및 쓰기 요청이 컴팩션 작업으로 인해 지연되어 트랜잭션 대기 시간을 증가시키며 트랜잭션 처리 효율 또한 감소시킨다. 이와 같은 구조는 SSD 내부 병렬성을 적극적으로 활용하지 못하게 된다[1].

이러한 문제점을 해결하기 위해 WiscKey에서는 키-값을 저장하는 공간을 분리하여 사용할 것을 제시하였다. WiscKey는 LevelDB의 LSM 트리에 키와 값의 주소만 저장하고 값은 별도의 vlog(value log) 파일에 저장한다. 그 결과, 정렬 과정 도중에 발생하는 불필요한 값 이동을 감소시켜 쓰기 증폭을 완화시켜 성능을 향상시켰다. 또한 LSM 트리 크기를 줄여 결과적으로 디스크 읽기 횟수를 줄이고 키 탐색 시 지연시간을 개선시킨다[1].

그러나 WiscKey에서는 vlog 파일에 저장되는 값의 무결성을 보장하기 위해 vlog 파일에 값을 쓸 때 마다 fsync 시스템 호출 함수를 사용한다. 하지만 매번 fsync 시스템 호출 함수를 사용할 경우 함수 자체의 처리 시간이 길어 WiscKey 데이터베이스에 워크로드 처리 시간 기준 최대 5.8배의 성능 저하를 초래한다[3].

본 논문에서는 fsync 시스템 호출 함수를 사용하지 않음과 동시에 무결성을 보장할 수 있는 NVDIMM 캐싱 기법을 제안한다. NVDIMM은 비정상적 전원 손실에도 내부 데이터의 손실이 없기 때문에 vlog를 저장하는 블록 저장장치와 LevelDB의 중간 계층으로써 활용하여 무결성을 보장하고, 쓰기 성능을 개선시켰다. Set 쿼리 수행 시 NVDIMM에 1차적으로 값을 쓴 후 일정 용량이 찰 때 마다 플러쉬를 수행하게 하였다. 그 결과, 무결성을 보장하며 동일 워크로드 수행 시 WiscKey보다 트랜잭션 수행 시간을 9.8배 감소시켰다.

2. 배 경

2.1 LSM 트리와 WiscKey

구글의 LevelDB와 같은 기존의 LSM 트리 기반 Key-Value 스토어는 SSD보다 하드디스크에 더 특화되어 있어 SSD의 특성을 충분히 활용하지 못한다. 이유는 LSM 트리 기반 Key-Value 스토어는 필연적으로 컴팩션 과정, 즉 트리 정렬 과정을 거쳐야 하는데, 이 과정에서 키와 값이 모두 포함된 SSTable을 정렬해야 하기 때문에 큰 오버헤드를 가지게 된다. 따라서 키와 값을 모두 담고 있는 SSTable을 정렬하기보다, 작은 크기의 키 만 정렬한다면 정렬하는 동안 수행되는 디스크 읽기/쓰기의 수가 줄게 될 것이다. WiscKey는 값을 vlog에 쓰고 해당 값이 쓰여진 주소만 Key-Value 스토어에 저장하여, 데이터베이스에 실제로 저장되는 Key-Value 의 크기를 줄임으로써 SSTable이 정렬되는 데에 걸리는 시간을 단축시켰다. 그러나 이와 같이 값을 vlog 파일에 별도로 저장할 경우, 데이터베이스의 무결성을 위해 값을 쓸 때 마다 응답속도가 느린 fsync 시스템 호출 함수를 사용해야 하는 단점이 있다.

2.2 fsync

fsync는 Unix 운영 체제의 표준 시스템 호출 함수이다. Linux에서는 해당 함수 호출 시 파일 기술자를 인자로 사용하는데, 해당되는 파일의 변경된 데이터 및 메타데이터를 비휘발성 저장장치에 쓰고 동기화함으로써 시스템의 고장이나 전원 손실에도 데이터의 무결성이 유지되게 하는 함수이다. 데이터베이스에서 fsync는 데이터의 무결성을 보장해주지만, 함수가 수행되는데 지연시간이 오래 걸리기 때문에 트랜잭션 처리 속도를 저하시킨다. 그러므로 DBMS(Database Management System)는 fsync할 데이터의 양과 호출 빈도수를 적절히 조절해야 한다.

2.3 NVDIMM

NVDIMM 기술은 기존의 휘발성 DRAM의 한계점을 극복하였기 때문에 새롭게 부상하는 기술 중 하나이다. 이러한 NVDIMM은 DRAM과 같은 낮은 지연시간을 가졌지만, SSD/HDD와 같이 비휘발성 메모리이다. DRAM에서 비휘발성을 구현하기 위해, NAND 플래시 메모리와 고용량 축전기, DRAM을 결합하였다. 만약 비정상적인 전원 손실이 발생할 경우, 고용량 축전기에서 공급되는 전력으로 DRAM에서 NAND 플래시로 데이터를 전송하는 방식으로 데이터 무결성을 보장한다. 이후 시스템이 다시 부팅되면 데이터가 다시 NAND 플래시에서 DRAM으로 복원된다. 이러한 NVDIMM은 현재 가장 빠른 I/O 지연시간을 가지고 있는 블록 장치(NVMe SSD)에 비해 I/O 지연시간이 30~240배 낮고, 대역폭과 IOPS(Input/Output Operations Per Second)는 동일 조건에서 약 20배 높다. 또한 MS Windows Server 2016부터 DAX(Direct Access)를 운영체제에서 지원하면서 관련 라이브러리 또한 충분이 지원되는 추세이다[4]. 하지만 NVDIMM은 1GB당 비용이 NVMe SSD보다 약 42배 더 비싸고 용량이 제한되어 있다[5]. 따라서 대규모 DBMS에서 NVDIMM을 기본 스토리지로 사용하는 것은 부적절하다. 이러한 사실은 NVDIMM이 디스크 기반 DBMS에서 DRAM과 스토리지(SSD/HDD) 사이의 캐싱 계층으로 NAND 플래시 스토리지를 대체할 수 있다는 것을 암시한다. 또한, NVDIMM은 NAND 플래시 스토리지와 달리 무한에 가까운 내구성을 가지며 NAND 플래시의 덮어쓰기 제한과 같은 제한점이 없다. 따라서 NVDIMM 캐싱 기법을 이용하면 무결성을 보장하며 안정적이고 높은 성능의 DBMS를 구현할 수 있다.

3. 관련 연구

본 장에서는 비휘발성 메모리를 응용한 데이터베이스 최적화와 관련된 연구를 소개한다.

3.1 PB-NVM

[5]는 현재의 SSD 기반 DBMS (Database Management System)이 OLTP 워크로드에서의 성능 병목 현상을 개선하기 위해 NVDIMM-N을 응용한다. 먼저 현재의 SSD 기반 DBMS이 OLTP 워크로드 수행에 있어 병목 현상이 심한 이유는 다음과 같다. 첫째, SSD의 입출력 대기시간이 느리다. 둘째, SSD의 읽기 요청 대기시간보다 쓰기 요청 대기시간이 길다. 셋째, DBMS가 원자성과 영속성을 보장하기 위해 과도한 쓰기 요청을 수행한다. 넷째, 메모리에 저장된 정보들을 SSD에 플러쉬를 수행할 때, 여러 플러쉬 쓰레드들이 하나의 버퍼를 공유하여 뮤텍스 락 경합이 심해진다. [5]는 NVDIMM의 메모리에 가까운 빠른 입출력 처리 속도를 응용하여 SSD의 입출력 대기시간과 읽기-쓰기 요청 대기시간의 불균형에서 오는 문제를 해결하였다. 또한 NVDIMM의 비휘발성을 응용하여 DBMS의 원자성과 영속성을 보장하였다. 마지막으로, 분할 버퍼를 이용하여 여러 플러쉬 쓰레드 들의 뮤텍스 락 경합을 완화하였다. [5]는 기존 InnoDB/MySQL에 비해 대역폭은 약 2.47배, 트랜잭션 당 플러쉬 대기시간은 약 7배 향상시켰다.

3.2 NVLSM

[6]은 비휘발성 메모리를 저장매체로 응용할 때, LSM (Log-Structured Merge) 트리를 사용하는 Key-Value 스토어의 쓰기 증폭 현상에 의한 성능 병목 현상을 개선하기 위해, 기존 연구된 누적 컴팩션 기법을 응용하였다. 하지만 분할 컴팩션 기법을 사용하면 읽기 증폭이 악화되기 때문에, 이를 개선하기 위해 비휘발성 메모리의 바이트 주소 지정 기능을 응용한다. 바이트 주소 지정 기능을 사용하여 여러 층에 분산되어 있는 컴팩션 필요 데이터의 주소를 기록해두었다가 접근함으로써 필요한 컴팩션의 수를 줄인다. 또한 분할 컴팩션 기법을 사용함으로써 악화된 읽기 증폭을 개선하기 위하여 계단식 탐색 기법을 제안하였다. [6]은 기존 Leveled 컴팩션 기법에 비해 쓰기 증폭을 최대 67%까지 감소시켰으며, 쓰기 위주의 워크로드에서 기존 컴팩션 기법에 비해 평균 쓰기 대기 시간을 최대 41.72% 단축하였다.

4. 디자인

4.1 개 요

Fig. 1은 WiscKey Key-Value 스토어에서 쓰기 쿼리가 매번 수행될 때 마다 fsync를 수행한 경우와 fsync를 수행하지 않은 경우 쓰기 워크로드 수행 시 워크로드가 완료되는데 걸리는 시간을 나타낸다[3]. 기 수행된 연구에서 vlog에 값을

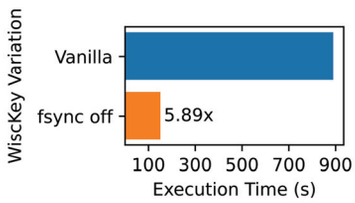


Fig. 1. fsync Overhead on WiscKey Key-Value Store [3]

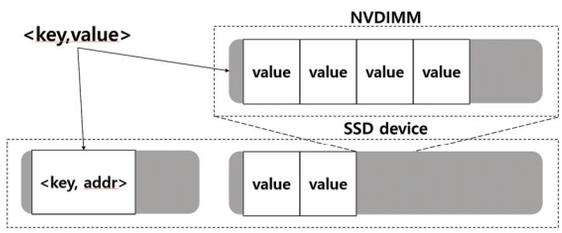


Fig. 2. WiscKey+NVDIMM Data Layout on NVDIMM and SSD

쓸 때마다 fsync 시스템 호출 함수를 사용할 경우, 그렇지 않을 경우보다 최대 5.9배의 성능 저하가 발생하였다[3]. 본 논문에서는 fsync 시스템 호출 함수의 사용을 지양하여 성능을 개선시키고 데이터 무결성을 보장하기 위해 NVDIMM 캐싱 기법을 사용하였다.

Fig. 2는 WiscKey+NVDIMM에 key와 value가 저장될 때의 데이터가 저장되는 방식을 나타낸 그림이다. 기존 WiscKey의 경우 쓰기 쿼리 수행 시 마다 Value Log에 fsync를 수행하여 성능 저하를 유발하였다. 하지만 쓰기 쿼리 수행 시 NVDIMM 버퍼에 먼저 저장한 뒤 버퍼가 차면 Value Log에 한번에 쓰고 fsync를 수행한다면 fsync에 의한 성능 저하를 개선할 수 있다. 이어질 3.2 단일 버퍼 캐싱, 3.3 이중 버퍼 캐싱에서는 NVDIMM 버퍼에 캐싱되는 과정을 보다 상세히 설명하였다.

4.2 단일 버퍼 캐싱

기존 WiscKey의 경우 Set() 함수를 수행할 때 먼저 vlog에 값을 쓴 후, fsync 시스템 호출 함수를 호출한 뒤 주소 값을 기존의 키와 함께 LevelDB에 Set() 요청을 하는 식으로 구현되어있다. 이 방식에서의 fsync 시스템 호출 함수의 호출 빈도를 줄이기 위하여, NVDIMM 상의 버퍼를 두어 버퍼가 다 찰 때까지 값을 저장한다(Fig. 3). 버퍼가 다 차면 일시적으로 I/O를 중단하고 블록 스토리지에 전체 데이터를 플러쉬한 뒤 버퍼 내용을 초기화하고 I/O를 다시 수행한다(Fig. 3). 예를 들어 8바이트 크기의 키와 4KB 크기의 값을 1024번 쓰는 상황을 가정해보면, 기존의 WiscKey는 4KB를 쓸 때 마다 fsync가 발생한다. 1024번의 쓰기동안 fsync에 의한 지연이 1024번 발생한다. 하지만 1MB 크기의 단일 버퍼를 추가한다면 256번의 쓰기마다 한번씩 fsync가 발생한다. 1024번의 쓰기동안 fsync에 의한 지연이 4번 발생한다. 따라서 fsync에 의한 지연시간이 감소해 워크로드 전체를 처리

Algorithm 1 WiscKey Single Buffer

```
1: procedure SET(key,value)
2:     if buf is full then
3:         flush buf to vlog
4:         fsync(vlog)
5:     end if
6:     buf ← value
7:     offset = vlog.length + buf.length
8:     LevelDB_Set(key,offset)
9: end procedure
1: procedure GET(key)
2:     offset = LevelDB_Get(key)
3:     if offset < vlog.length then
4:         value ← vlog[offset]
5:     else
6:         value ← buf[offset − vlog.length]
7:     end if
8:     return value
9: end procedure
```

Fig. 3. Pseudocode of WiscKey + NVDIMM Single Buffer

Algorithm 2 WiscKey Double Buffer

```
1: procedure SET(key,value)
2:     acquire cur_buf.lock
3:     if cur_buf is full then
4:         Send flush request of cur_buf
5:         release cur_buf.lock
6:         cur_buf ← another buf
7:         acquire cur_buf.lock
8:     end if
9:     buf ← value
10:    offset = vlog.length + buf.length
11:    LevelDB_Set(key,offset)
12: end procedure
1: procedure GET(key)
2:     offset = LevelDB_Get(key)
3:     if offset < vlog.length then
4:         value ← vlog[offset]
5:     else if offset > (vlog.length + BUF_SZ) then
6:         acquire cur_buf.lock
7:         value ← cur_buf[offset − (vlog.length + BUF_SZ)]
8:         release cur_buf.lock
9:     else
10:        acquire another buf.lock
11:            ▷ if buf.lock is free, it means flush process is end.
12:        value ← vlog[offset]
13:                    ▷ if flush process end, value is in vlog.
14:        release another buf.lock
15:    end if
16:    return value
17: end procedure
1: procedure FLUSHINGTHREAD()
2:     while True do
3:         Wait for flush request
4:         acquire buf.lock, vlog.lock
5:         flush buf → vlog
6:         fsync(vlog)
7:         release buf.lock, vlog.lock
8:     end while
9: end procedure
```

Fig. 4. Pseudocode of WiscKey + NVDIMM Double Buffer

하는 시간이 감소한다. 또한, 값을 개별적으로 쓰고 fsync 하는 것이 아니라 여러 값을 모아서 순차적으로 쓰기 때문에 순차쓰기의 효율이 좋은 SSD의 특성에 알맞다.

4.3 이중 버퍼 캐싱

단일 버퍼 캐싱 방식의 경우, fsync에 의한 오버헤드가 완전히 해소된 것은 아니다. 버퍼에 일정 용량 이상의 쓰기가 발생할 때 마다, 모든 데이터베이스 연산을 중지하고 NVDIMM에서 블록 스토리지 장치에 내용을 전부 옮겨 쓰는 것을 기다려야 하기 때문이다. 따라서 버퍼가 찰 때 마다 쓰기 지연시간이 증가하게 된다. 이러한 특정 조건에 의한 지연시간 증가는 데이터베이스의 최소 성능 보장을 어렵게 한다.

버퍼를 비울 때 마다 지연시간이 증가하는 것을 피하기 위하여, 버퍼가 다 찼을 때 백그라운드에서 플러쉬를 수행하게 하고 다른 버퍼에 계속 쓰게 한다(Fig. 4). 즉, Set() 함수를 수행할 때 먼저 현재 사용 중이던 버퍼가 다 찼는지 확인하고, 다 찼다면 백그라운드 플러쉬 쓰레드에게 플러쉬 요청을 보내고 다른 버퍼에 데이터를 쓴다(Fig. 4). 백그라운드 플러쉬 쓰레드는 플러쉬 요청을 기다리다 플러쉬를 수행한 뒤 fsync를 수행한다(Fig. 4). 백그라운드 플러쉬 쓰레드가 플러쉬, fsync를 수행하기 때문에 총 워크로드 수행시간 중 fsync에 의해 지연되는 시간이 감소한다.

5. 성능 평가

5.1 실험 환경

1) 하드웨어

성능 평가는 Ubuntu 20.04 LTS 운영체제 환경에서 수행되었다. 하나의 소켓에 CPU를 장착한 16코어(32 쓰레드) Intel Xeon Silver 4216 2.1Ghz CPU 환경에서 성능 평가를 진행하였다. 저장장치로는 Samsung SSD 970 Pro 1TB를 사용하였고 비휘발성 메모리로는 HPE 16GB NVDIMM Single Rank DDR4-2666을 사용하였다.

2) 워크로드

LevelDB는 한번에 한 프로세스만 접근 가능하기 때문에 쓰기 성능을 최대화하여 성능을 측정하기 위해 쓰기 위주의 임의의 워크로드를 제작하여 사용하였다. 미리 임의의 8바이트 키와 4KB 값 순서쌍을 8GB 로드 후, 해당 워크로드를 서

Table 1. Hardware Specification

OS	Ubuntu 20.04 LTS
CPU	Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz (16 Core)
Memory	32GB
Storage	Samsung SSD 970 Pro 1TB
NVDIMM	HPE 16GB NVDIMM Single Rank DDR4-2666

로 다른 WiscKey 버전에서 수행하였다. 기존 WiscKey 기준으로 fsync가 데이터 쓰기 쿼리가 262,144번 일어나 fsync 또한 262,144번 일어나게 되는 워크로드이다.

3) 성능 측정 도구

성능 측정 지표로 워크로드 수행시간, fsync 함수의 코어 사용량, 쓰기 지연시간을 사용하였다. 워크로드 수행시간 측정은 Linux의 time(1) 명령을 통해 측정하였다. time(1) 명령은 간단히 실행한 명령의 시간을 측정하거나 코어 사용 시간을 측정할 수 있는데, 본 성능 평가에서는 워크로드 수행 프로세스 명령의 시간을 측정하였다. fsync 함수의 코어 사용량은 perf(Performance Counter for Linux, PCL)을 통해 측정하였는데, 해당 도구를 사용하면 커널과 유저 프로세스에서 특정 심볼(__x64_sys_fsync 등)을 가진 함수의 오버헤드를 측정할 수 있다. 쓰기 지연시간은 sysstat 라이브러리에 iostat을 사용하여 성능 평가 동안 SSD에 발생하는 모든 쓰기 연산의 평균 지연시간을 측정하였다.

5.2 성능 평가 결과

1) 워크로드 수행시간 단축

Fig. 5는 WiscKey 버전에 따른 워크로드의 실행 시간을 나타낸다. 8GB의 쓰기 워크로드를 실행한 결과, 기존의 WiscKey에 비해 단순히 fsync를 수행하지 않는 WiscKey에서는 5.89배 적은 시간이 소요되었고, 단일 버퍼에서는 5.60배 적은 시간이 소요되었다. 이는 단순히 fsync를 수행하지 않은 것 보다 버퍼에 쓸 때 버퍼의 용량을 확인하거나 하는 몇 가지 연산을 추가하여 약간의 오버헤드가 발생한 것으로 보인다. 하지만 이중 버퍼에서는 기존 WiscKey에 비해 9.78배적은 시간이 소요되었는데 이를 통해서 이중 버퍼에서 값을 쓸 때 fsync에 의한 오버헤드가 매우 적음을 알 수 있다.

Fig. 6은 WiscKey 이중 버퍼 버전에서 버퍼 크기에 따른 워크로드의 실행 시간을 나타낸다. 버퍼를 1MB에서 8MB로 확장하면, 약 6%의 수행시간 성능 차이가 있지만 쓰기 위주의 간단한 워크로드를 수행할 때는 적은 버퍼 용량으로도 수행시간 성능을 크게 향상시킬 수 있어 효율적이다.

2) fsync cost 감소

Fig. 7은 워크로드를 수행하는 동안 perf를 사용하여 fsync에 의해 발생한 코어 오버헤드를 측정한 결과를 나타낸다. 기존 WiscKey의 경우 워크로드 수행 시 코어 연산 시간 중 37.38%가 fsync에 의해 지연되었다. 하지만 단일 버퍼를 추가한 경우 2.69%, 이중 버퍼의 경우 0.1%가 지연되었다. 단일 버퍼의 경우 fsync에 의한 코어 오버헤드가 13.90배 개선되었지만, 수행시간은 5.60배 줄어든 이유는 버퍼 관련 연산이 추가되어 해당 연산을 처리하는 코어 오버헤드와, LevelDB 데이터베이스의 기본적인 오버헤드 때문이다.

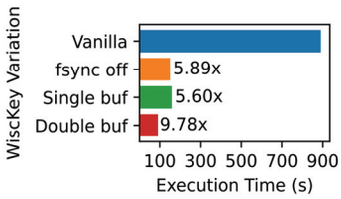


Fig. 5. Total Execution Time of the Workload by WiscKey Variation

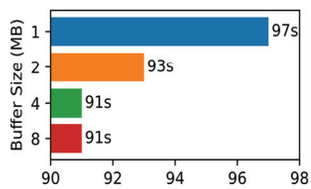


Fig. 6. WiscKey + Double Buffer Total Execution Time of the Workload by Buffer Size

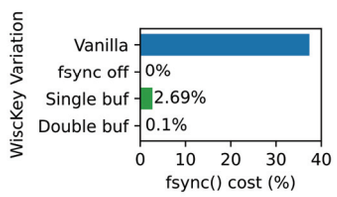


Fig. 7. fsync() Cost of the Workload by WiscKey Variation

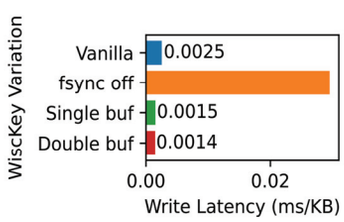


Fig. 8. Average Write Latency per 1KB I/O of the Workload by WiscKey Variation

3) 쓰기 지연시간 단축

Fig. 8은 워크로드를 수행하는 동안 블록 스토리지 장치에 발생한 쓰기 연산 1KB 당 평균 지연시간을 나타낸다. 버퍼를 사용한 경우가 그렇지 않은 경우보다 평균 지연시간이 짧다. 이유는 다음과 같다. 버퍼를 사용하여 데이터를 누적한 후에 플러쉬를 하게 되면 비교적 많은 양의 데이터를 한 번에 순차적으로 쓰게 된다. 이러한 패턴의 SSD 접근은 한번의 I/O 후에 fsync를 하는 경우보다 효율적이다. SSD의 특성상 작은 양의 데이터를 여러 번 요청하는 것 보다 많은 양의 데이터를 순차적으로 한 번에 쓰는 것이 더 효율적이기 때문이다. 따라서 버퍼를 사용하는 것이 SSD의 특성을 더 효율적으로 이용할 수 있다. 또한 특징적인 것은 fsync를 수행하지 않는 경우 지연시간이 다른 경우에 비해 길다는 점이다. 이유는 fsync를 수행하지 않으면 운영체제에 입출력 큐에 요청이 쌓여 비교적 늦게 처리되기 때문이다.

6. 결 론

본 논문에서는 WiscKey 데이터베이스를 사용할 때 NVDIMM 캐싱 기법을 사용하여 vlog에 쓰기 연산에 동반한 fsync 연산을 줄임으로써 성능을 개선시켰다. 본 논문이 공헌한 바는 다음과 같다.

- WiscKey 데이터베이스의 fsync 오버헤드를 개선하여 데이터 저장 워크로드 수행 시 성능을 9.78배 개선하였다.
- NVDIMM-N을 메모리와 블록 스토리지 장치의 중간 계층으로써 활용할 수 있는 방법을 제시하였다.

성능 평가 결과, 동일 워크로드 수행시간이 기존 WiscKey에 비해 단일 버퍼의 경우 5.60배, 이중 버퍼의 경우 9.78배 감소하였다. NVDIMM 구입 시 1GB당 비용이 NVMe SSD보다 42배 정도 더 비용이 소모되지만[4], 16MB 정도의 NVDIMM만 사용해도 성능을 크게 개선시킬 수 있다. 트랜잭션 대기 시간을 9.78배 개선시키는데 1/64 GB 정도의 용량을 소모한다. 즉, 비용 대비 성능 개선의 정도가 높아 Key-Value 스토어 사용 시 NVDIMM을 사용하는 것이 효율적이다. 기존 WiscKey 데이터베이스의 경우 LSM 트리의 용량을 줄여 효율적인 컴팩션 수행을 가능하게 했지만, vlog 쓰기 연산 시 마다 fsync를 수행하여 블록 스토리지의 성능을 전부 활용하기 어려웠다. 하지만 vlog 쓰기 연산 시의 NVDIMM을 활용한 중간 계층을 추가한다면 데이터의 무결성을 지키면서 WiscKey 데이터베이스의 성능을 크게 개선할 수 있다.

References

[1] L. Lu, T. Pillai, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "WiscKey: Separating keys from values in SSD-conscious storage," *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016.

[2] Google, LevelDB, [Internet], https://github.com/google/leveldb

[3] I. H. Song, J. h. Park, and S. W. Lee, "Performance degradation of WiscKey database due to fsync," *Korea Computer Congress 2022 (KCC 2022)*, 2022.

[4] Micron®, Windows Server 2016 Shows 75X IOPS Gain With NVDIMM-N, HPE, Micron® and Microsoft® Windows® Server 2016/NVDIMM-N Solution.

[5] T. D. Nguyen and S. W. Lee, "PB-NVM: A high performance partitioned buffer on NVDIMM," *Journal of Systems Architecture*, Vol.97, pp.20-33, 2019.

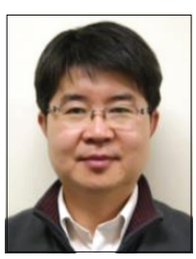
[6] B. Zhang and D. H. C. Du. "NVLSM: A persistent memory key-value store using log-structured merge tree with accumulative compaction," *ACM Transactions on Storage (TOS)*, Vol.17, No.3, pp.1-26, 2021.

송 일 한

https://orcid.org/0000-0003-4785-1978
e-mail : xmilex@skku.edu
2017년 성균관대학교 소프트웨어학부 (학사)
2021년 ~ 현 재 성균관대학교 소프트웨어학부(석사)
관심분야 : Flash Memory & Database System

이 보 현

https://orcid.org/0000-0002-8128-6164
e-mail : lia323@skku.edu
2018년 성균관대학교 소프트웨어학부 (학사)
2022년 ~ 현 재 성균관대학교 소프트웨어학부(석사)
관심분야 : Database System



이 상 원

https://orcid.org/0000-0001-8837-3255
e-mail : swlee@skku.edu
1991년 서울대학교 컴퓨터학과(학사)
1994년 서울대학교 컴퓨터학과(석사)
1999년 서울대학교 컴퓨터학과(박사)
1999년 ~ 2001년 한국오라클
2001년 ~ 2002년 이화여대 BK21 계약교수
2002년 ~ 현 재 성균관대학교 컴퓨터공학과 교수
관심분야 : Flash Memory & Database System