In-storage Processing을 활용한 LSM-tree 컴팩션 가속 기법

이영재°, 신동군
성균관대학교 전자전기컴퓨터공학과
yjlee4154@gmail.com, dongkun@skku.edu

Accelerating LSM-tree Compaction using In-storage Processing

Youngjae Lee°, Dongkun Shin
Department of Electrical and Computer Engineering, Sungkyunkwan University

요 약

Log-structured Merge tree(LSM-tree)는 쓰기 성능을 높이기 위해 최적화된 자료구조로 key-value 데이터베이스 시스템에서 널리 사용되고 있다. LSM-tree는 읽기 성능의 최적화를 위해 컴팩션 작업을 수행하며 이는 많은 양의 CPU 연산과 디스크 I/O를 사용하여 LSM-tree 기반 데이터베이스의 bottleneck이 되고 있다. 또한 데이터 처리량이 증가함에 따라 호스트와 저장장치 간의 데이터 이동에 드는 비용이 커지고 있어 이를 줄이기 위한 기법으로 저장장치 내부에서 데이터를 처리하는 In-storage Processing(ISP)이 활발히 연구되고 있다. 따라서 본 연구에서는 ISP-SSD 환경을 구축하고 이를 활용하여 LSM-tree 기반의 Key-value store의 컴팩션 성능을 최적화하고 효과를 분석한다.

1. 서 론

Log-structured Merge-tree(LSM-tree)[1]는 key-value 형태의 스토리지 시스템에서 쓰기 성능을 높이기 위해 최적화된 자료구조로 LevelDB[2], RocksDB, Cassandra 등의 많은 데이터베이스 시스템에서 사용되고 있다. LSM-tree는 주기적인 컴팩션 작업으로 데이터를 재정렬하여 읽기 성능을 최적화한다. 컴팩션 과정은 기존의 파일을 읽고 새로운 파일로 재생성 하는 과정에서 많은 CPU 자원을 소모하고 I/O 트래픽을 유발하여 LSM-tree의 쓰기 성능을 저하시킨다.

In-storage Processing(ISP)은 저장장치 내부에서 연산을 수행하고 결과 데이터만을 호스트로 전송하여 I/O와 호스트 리소스의 사용량을 줄이는 기법으로 데이터의 이동을 줄이기 위해 데이터가 있는 곳에서 연산을 수행하는 Near Data Processing(NDP)을 저장장치 수준에서 적용하는 것이다 [3]. 특히 NVMe SSD 등의 고성능의 저장장치가 등장하면서 호스트 소프트웨어 스택의 I/O 수행시간 비중이 커지고 있어 ISP를 통해 이를 완화할 수 있다.

본 연구에서는 LSM-tree의 성능을 최적화하기 위해 이를 저장장치에서 수행하도록 하는 In-storage Processing 기법을 적용하여 key-value store의 성능을 개선한다. ISP를 적용한 SSD 환경을 구축하여 LSM-tree의 컴팩션 작업을 오프로딩하고 이로 인한 성능 저하를 개선한다.

이 성과는 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (No. 2020R1F1A1073758).

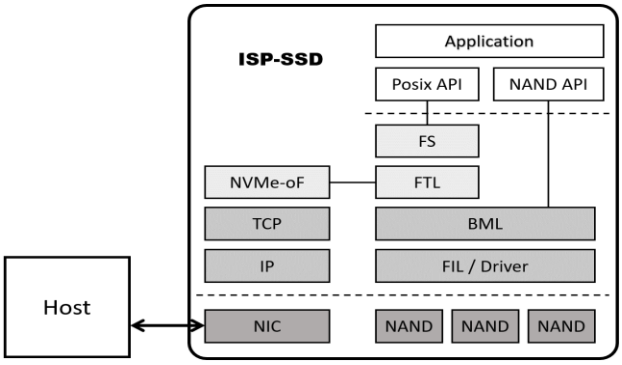
2. 배경 이론

LSM-tree 기반의 key-value store는 디스크에 데이터를 연속적으로 쓰기 위해 새로 입력되는 key-value 레코드를 메모리 자료구조인 MemTable에 일정량을 모아 주기적으로 이를 flush 작업을 통해 SSTable의 형태로 디스크에 저장한다. Key-value 레코드는 하나의 SSTable 내에서 오름차순으로 정렬되어 있으며 여러 레벨을 사용하여 이를 관리한다. 임의의 레코드를 탐색하기 위해서는 해당 key의 범위를 포함하는 모든 SSTable을 탐색해야 하므로 SSTable의 수가 증가할수록 많은 양의 디스크 읽기를 유발하여 읽기 성능이 저하된다. 따라서 주기적인 컴팩션 작업을 통해 SSTable을 재정렬하여 탐색 시간을 줄인다.

컴팩션 작업은 기존의 SSTable의 동일한 key 값을 갖는 중복된 레코드를 제거하고 레코드를 재정렬하여 새로운 SSTable을 생성한다. 하지만 컴팩션을 수행하기 위해 기존의 SSTable을 디스크에서 읽고 새로운 SSTable로 쓰는 과정에서 많은 양의 CPU 연산과 디스크 I/O가 발생하여 LSM-tree의 성능 저하를 유발하는 주요 원인이 된다.

3. 선행 연구

LSM-tree의 구조를 변경하여 컴팩션 성능을 개선한 여러 연구가 있었다. WiscKey[4]는 SSTable에서 key와 value를 분리하여 value를 별도의 위치에 저장하여 SSTable 크기를 줄이고 컴팩션 오버헤드를 감소시켰다. PebblesDB[5]는 동일한 레벨의 데이터를 다시 쓰지 않도록 하여 컴팩션 과정에서 발생하는 I/O를 줄이고



[그림 1] ISP-SSD의 구조

쓰기 성능을 개선시켰다.

컴팩션을 별도의 하드웨어에 오프로딩한 연구도 있었다. Zhang et al.[6]은 FPGA 가속기로 컴팩션을 수행하여 CPU 성능이 bottleneck이 되는 부분을 개선하였다. TStore[7]는 LSM-tree에 ISP를 적용한 것으로 컴팩션 작업을 나누어 호스트와 SSD에서 동시에 처리하여 수행 시간을 최소화하였다.

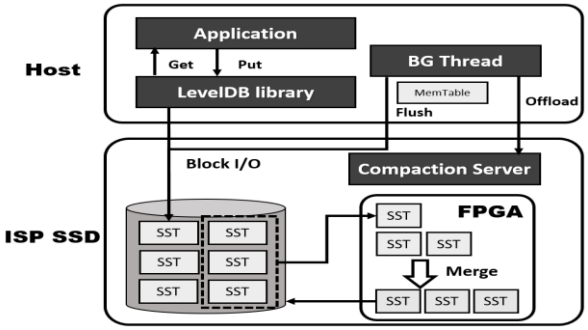
이외에 SSD에 ISP를 적용한 연구는 다음과 같다. Biscuit[8]은 ISP 개발을 위한 프레임워크를 제안하여 데이터 처리 애플리케이션이 호스트 및 SSD에서 분산 처리 형태로 수행되도록 하였다. YourSQL[9]은 Biscuit을 SQL 데이터베이스에 적용하여 쿼리 처리를 SSD에 오프로딩 하며 전용의 하드웨어를 사용하여 필터링을 수행한다. Catalina[10]는 고성능의 ARM CPU를 탑재한 Computational Storage Device(CSD)를 활용해 ISP의 분산처리 워크로드에서의 효과를 보여주었다.

4. ISP-SSD

ISP-SSD의 구현을 위해 ARM SoC를 탑재한 보드를 기반으로 환경을 구축하였다. ISP-SSD 내부 소프트웨어 스택은 TCP/IP, NVMe 인터페이스, 파일시스템 등의 사용을 위해 리눅스 운영체제를 기반으로 구현하였으며 전반적인 구조는 [그림 1] 과 같다.

SSD 기능을 제공하기 위해서 NVMe 인터페이스를 사용해 호스트에서 요청을 받아 I/O를 수행할 수 있어야 한다. 따라서 기존 SSD 펌웨어에서 사용되는 Flash Translation Layer(FTL) 등의 소프트웨어가 필요하며 이를 커널 영역에 구현하였다. 또한 ISP-SSD 내부의 애플리케이션이 NAND 플래시의 데이터를 파일 시스템 형태로 접근할 수 있도록 커널 레벨 FTL에서 블록 디바이스 인터페이스를 제공하도록 하였다.

또한 ISP-SSD의 애플리케이션이 호스트와 동일한 파일시스템에 접근할 수 있도록 OCFS2 (Oracle Clustered File System 2) 파일시스템을 사용하였다. OCFS2는 호스트 간 디스크를 공유하는 환경에서 사용되는 파일시스템으로 Distributed Lock Manager (DLM)를 통해 호스트간 동기화를 수행한다.



[그림 2] ISP를 적용한 LevelDB 구조

5. ISP 활용 Key-value Store 최적화

ISP 오프로딩을 적용한 LSM-tree 기반의 Key-value 데이터베이스는 LevelDB[2]를 기반으로 작성하였다. 클라이언트-서버 구조를 사용하며 호스트의 LevelDB가 클라이언트가 되어 SSD 내부의 서버에 오프로딩 한다. TCP/IP 기반 메시징 프로토콜을 사용하여 컴팩션 대상 SSTable의 목록을 전달하며 서버는 SSTable을 읽어 컴팩션을 수행하여 새 SSTable을 저장하고 결과를 호스트로 전송한다. 호스트 LevelDB는 컴팩션 결과를 받고 메타데이터에 반영한다.

LevelDB는 컴팩션 작업을 위해 단일 백그라운드 스레드만을 사용하지만 ISP를 적용한 LevelDB의 경우 컴팩션을 오프로딩하는 중에 백그라운드 스레드에서 MemTable flush 작업 등을 수행하여 호스트와 ISP-SSD를 병렬적으로 활용할 수 있다. 또한 다음 두 가지의 최적화 기법을 적용하였다.

① 파일시스템 동기화 최적화. OCFS2는 호스트 간 동일한 디렉토리나 파일에 대해 작업하는 경우 동기화 과정에서 오버헤드가 발생한다. 이에 따른 오버헤드를 줄이기 위해 호스트와 디바이스에서 SSTable 파일을 다른 디렉토리에 쓰도록 하였으며 로그와 메타데이터 또한 다른 디렉토리에 쓰도록 분리하였다.

② 호스트 Level-0 컴팩션. Level-0 SSTable의 경우 최근에 쓰인 것으로 호스트 메모리에 캐싱되어 있다. 따라서 Level-0 컴팩션은 추가 I/O 없이 호스트에서 기존의 방식대로 수행하며 Level-1 이상의 컴팩션 작업을 ISP-SSD에 오프로딩하여 처리하도록 하였다.

X86과 ARM CPU의 성능 차이를 보기 위해 메모리 상에서 컴팩션 성능을 비교해본 결과 ARM CPU에서 3.2배 정도 수행시간이 긴 것을 확인했다. 따라서 낮은 성능을 보완하기 위해 기존의 연구[6]를 참고하여 FPGA 기반의 컴팩션 가속기를 사용한 하드웨어 버전도 구현하였다.

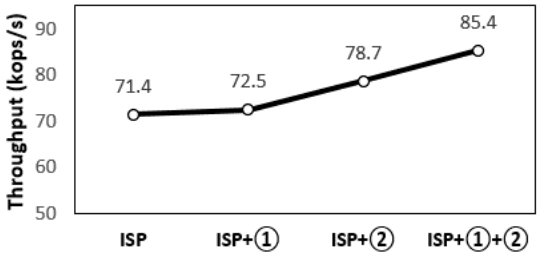
6. 실 험

6.1. 실험 환경

ISP-SSD 환경은 HoneyComb LX2K 보드를 사용하여 구현하였으며 ARM 기반의 LX2160A 프로세서 및 NAND 에뮬레이션을 위한 64GB의 DRAM을 탑재하였다.

	Host	ISP (SW)	ISP (HW)
Throughput(kops/s)	59.4	59.7	85.4

[표 1] 벤치마크 성능



[그림 3] 최적화 기법에 따른 성능

호스트는 Intel i5-4570, 8GB DRAM의 PC를 사용하였다.

사용된 개발보드는 NVMe 컨트롤러를 탑재하고 있지 않아 10기가비트 이더넷으로 호스트와 연결하고 TCP 기반의 NVMe-oF를 적용하여 NVMe SSD로 사용할 수 있도록 구성하였다. 또한 NAND 플래시 에뮬레이션을 위해 DRAM의 일부를 NAND 영역으로 할당하고 전용의 스레드가 I/O를 수행하도록 하였다.

실험은 기존의 LevelDB와 ISP를 적용한 LevelDB를 사용하였고, ISP는 CPU를 사용하는 소프트웨어 버전과 FPGA를 사용한 하드웨어 방식의 오프로딩을 사용하였다. 워크로드는 LevelDB db_bench의 랜덤 쓰기를 사용하였다. (50,000,000 ops)

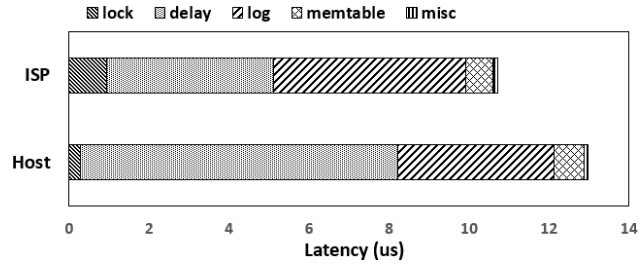
6.2. 실험 결과

[표 1]은 벤치마크의 쓰기 처리량을 보여준다. 소프트웨어 방식은 호스트 I/O를 줄이고 병렬적으로 컴팩션을 수행하여 ARM CPU 성능의 한계를 완화하여 LevelDB와 비슷한 성능을 보였다. 하드웨어 컴팩션 IP를 적용한 경우 약 44%의 쓰기 성능 향상을 보여주었다.

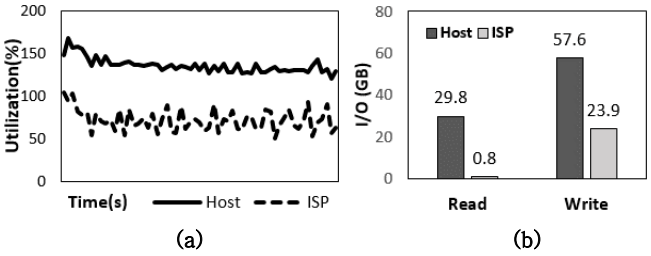
[그림 3]은 최적화 기법에 따른 성능 변화로 기본 ISP는 모든 컴팩션 작업을 오프로딩 하며 하나의 디렉토리를 사용하였고 여기에 ①,② 두 기법을 각각 또는 모두 적용한 경우의 성능 비교이다. 디렉토리 분리를 통한 동기화 오버헤드 감소와 Level-0 컴팩션의 호스트 수행 기법이 DB의 성능 개선에 도움이 되는 것을 보여준다.

[그림 4]는 LevelDB의 쓰기 동작의 각 부분에서 걸리는 시간을 분석한 것이다. ISP를 적용한 경우 백그라운드 컴팩션 작업에 의해 발생하는 지연이 48% 감소되어 전반적으로 쓰기 성능이 향상되었다.

[그림 5]는 벤치마크를 수행하는 동안 리소스 사용량 비교이다. (a)는 호스트의 CPU 사용량으로 ISP를 적용한 경우 대부분의 컴팩션 작업이 오프로딩 되어 LevelDB 대비 평균적으로 51%의 CPU 사용량을 보여주었다. (b)는 I/O 총량으로 ISP를 적용한 경우 디바이스 내부에서 컴팩션을 처리하기 때문에 이를 크게 줄일 수 있다.



[그림 4] LevelDB 쓰기 수행시간 분석



[그림 5] CPU 사용량 및 I/O

7. 결론 및 향후 계획

본 연구에서는 ISP-SSD 환경을 구축하고 이를 LSM-tree 컴팩션 오프로딩에 적용해 기존 대비 리소스 사용량을 줄이고 높은 성능을 제공할 수 있는 것을 보여주었다. 또한 본 연구의 ISP-SSD 환경을 추후 다른 워크로드에 적용하면 ISP 애플리케이션을 쉽게 개발하고 그 효과를 분석할 수 있을 것으로 기대한다.

참고 문헌

[1] Patrick O'Neil, et all. The log-structured merge-tree (LSM-tree). Acta Informatica. 1996

[2] Sanjay Ghemawat and Jeff Dean. LevelDB. http://code.google.com/p/leveldb, 2011.

[3] R. Balasubramonian et al., "Near-data processing: Insights from a micro-46 workshop," Micro, IEEE, 2014.

[4] Lanyue Lu, et al. WiscKey: Separating Keys from Values in SSD-conscious Storage. USENIX FAST. 2016.

[5] Pandian Raju, et al. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. SOSP

[6] Teng Zhang, et al. "FPGA-Accelerated Compactions for LSM-based Key-Value Store", USENIX FAST, 2020.

[7] H. Sun, et al., "NearData Processing-Enabled and Time-Aware Compaction Optimization for LSM-tree-based Key-Value Stores", ICPP, 2019.

[8] B. Gu et al., "Biscuit: a framework for near-data processing of big data workloads", Proceedings of the International Symposium on Computer Architecture, 2016.

[9] I. Jo et al., "YourSQL: a high-performance database system leveraging in-storage computing", Proceedings of the VLDB Endowment, 2016.

[10] M. Torabzadehkashi, et al., "Computational storage: an efficient and scalable platform for big data and hpc applications," Journal of Big Data, 2019.