In-Storage Processing을 활용한 LSM-tree의 컴팩션 가속

임민제 ○, 정지윤, 신동군

성균관대학교 정보통신대학

scvhero9607@gmail.com, jiyun710@gmail.com, dongkun@skku.edu

LSM-Tree Compaction Acceleration using In-Storage Processing

Minje Lim○, Jeeyoon Jung, Dongkun Shin

College of Information and Communication Engineering, Sungkyunkwan University

요 약

Log-structured merge Tree(LSM-tree)의 컴팩션 작업은 DB의 스토리지 쓰기 지연을 유발하여 성능에 악영향을 미친다. In-Storage Processing 플랫폼을 이용하여 컴팩션을 오프로딩 하면 호스트와 스토리지 간 IO traffic을 감소시키고 작업 속도를 증가시켜 DB 성능을 향상시킬 수 있다. 이 논문에서는 컴팩션 IP를 FPGA가 탑재된 OpenSSD 플랫폼에 통합하였고, 외장 가속기 환경과 비교를 진행하였다.

1. 서론

Log-structured Merge Tree(LSM-tree)[1] 기반의 Key-value Store는 내부 구조를 유지하기 위하여 컴팩션(Compaction) 작업을 수행할 필요가 있다. 컴팩션은 많은 양의 스토리지 IO와 CPU 연산을 발생시키므로 DB 쓰기 성능 하락의 주요한 원인이 된다. 기존에는 외장 가속기를 이용하여 컴팩션을 가속하는 기법[2,3]이 제안되었는데, 외장 가속기에 컴팩션을 오프로딩 할 경우, 호스트의 CPU 사용량을 감소시키고, 컴팩션으로 인해 발생하는 성능 저하를 완화하여 DB의 쓰기 성능을 개선할 수 있다. 하지만, 외장 가속기를 이용하여 컴팩션을 가속할 경우, 호스트와 스토리지 간 IO 양은 감소시킬 수 없으며, 추가적으로 외장 가속기와 호스트 간 IO가 발생한다는 한계가 있다. In-Storage Processing은 호스트의 작업을 스토리지 내부 가속기에 오프로딩 시키는 기법으로, 스토리지와 호스트 사이의 IO와 호스트와 외장 가속기 사이의 IO를 제거할 수 있어 외장 가속기를 사용할 때 보다 효율적으로 연산을 오프로딩 할 수 있다. 본 논문에서는 다음과 같은 작업을 수행하였다.

컴팩션 IP를 Cosmos+ [4] OpenSSD에 통합하고 컴팩션을 스토리지에 오프로딩 할 수 있도록 펌웨어와 LevelDB[5]를 수정하였다.

기존의 외장 가속기와 In-Storage Processing 플랫폼 간의 비교 실험을 진행하고, In-Storage Processing 플랫폼의 이점을 분석하였다.

호스트에서 처리할 경우와 비교하여 컴팩션 성능을 최대 71.2%, 벤치마크 전체 성능을 최대 51.4% 증가시켰다.

이 논문은 2021년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임 (No.IITP-2017-0-00914, 지능형 IoT 장치용 소프트웨어 프레임워크)

2. 관련 연구

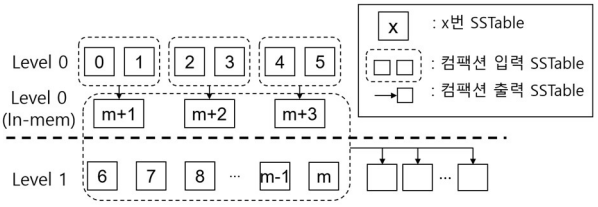
2.1 컴팩션 가속

LSM-Tree는 사용자가 삽입한 Key, Value 쌍을 In-memory 자료구조인 Memtable과 스토리지에 저장하기 위한 자료구조인 SSTable로 나누어 관리한다. Memtable의 크기가 임계점을 넘을 경우 Memtable을 SSTable로 변환한 뒤 스토리지에 기록하는 Memtable Flush가 수행되며, 새로운 SSTable은 Level 0에 기록된다, Level i 컴팩션은 Level i와 Level i+1의 서로 중복되는 Key range를 가지는 SSTable들을 정렬하고 병합하여 중복되는 Key, Value 쌍을 제거하고 독립적인 Key range를 가지는 새로운 SSTable들을 Level i+1에 생성하는 작업이다. Level 1 이상의 SSTable들은 항상 컴팩션을 통해서만 생성되기 때문에 동일한 Level의 SSTable들은 서로 독립적인 Key range를 가진다. 반면, Level 0의 SSTable은 Memtable Flush로 생성되기 때문에 서로 중복되는 Key range를 가질 수 있다. 따라서, Level 1 이상의 컴팩션의 경우에는 항상 동시에 비교하는 SSTable의 수가 2개지만, Level 0 컴팩션의 경우 정렬 과정에서 여러 SSTable들을 동시에 비교해야 한다.

기존에 컴팩션을 외장 가속기를 이용하여 가속한 연구는, SSTable 디코더의 수에 따라 동시에 비교할 수 있는 SSTable의 수가 제한되는데, 디코더의 수가 n 개일 경우, 컴팩션의 대상이 되는 Level 0 SSTable의 개수 m이 n-1 보다 클 경우, 하드웨어에 컴팩션을 오프로딩 할 수 없어 호스트에서 처리하는 방식으로 구현하였다.

단계적 컴팩션[6] 기법은 $n-1<m$ 인 상황에서도 컴팩션을 오프로딩하기 위해 제안된 기법으로, Level 0의 SSTable중 일

부만을 병합하여 merged SSTable을 Level 0에 추가하는 과정을 Level 0의 SSTable의 수가 n-1 이하가 될 때까지 반복한 뒤 Level 0와 Level 1의 SSTable을 대상으로 컴팩션을 진행하는 기법으로, merged SSTable은 실제 스토리지에 기록되지 않고 가속기 내부 DRAM에만 기록된다. merged SSTable을 생성하기 위해서는 컴팩션의 대상이 되는 SSTable를 모두 스토리지에서 읽어야 하는데, 병합의 대상이 되는 SSTable의 수를 항상 n으로 설정할 경우 병합을 시작하기 위한 대기시간이 지나치게 길어질 수 있으므로 대신 [m/n]개씩 병합을 수행한다, 그림 1은 디코더의 수 n이 4일 경우 Level 0의 6개의 SSTable과 Level 1의 SSTable들을 대상으로 단계적 컴팩션을 수행하는 과정을 보여준다. 먼저 Level 0의 SSTable들을 대상으로 컴팩션을 진행하여 3개의 merged SSTable을 생성하고, merged SSTable과 Level 1의 SSTable을 대상으로 컴팩션을 진행하여 최종 결과를 생성한다.

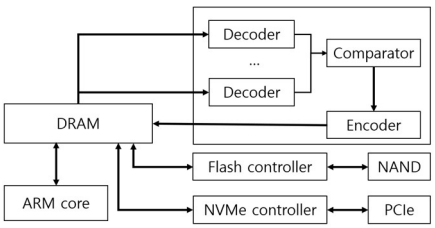


[그림 1] 단계적 컴팩션

3. In-storage processing 플랫폼 설계

3.1 하드웨어 구조

컴팩션 IP를 포함하는 In-Storage Processing 플랫폼의 설계는 그림 2와 같다. 컴팩션 IP는 단계적 컴팩션[5] 기법에서 제안한 IP의 구조와 유사한 구조를 가지고 있으며, DRAM에서 SSTable를 복사한 뒤 Key, Value 쌍 형태로 변환하는 디코더, 각 디코더에서 전달받은 Key, Value 쌍을 비교한 뒤 정렬하는 컴퍼레이터, Key Value 쌍을 SSTable 형태로 변환하여 DRAM에 기록하는 인코더로 구성된다. 플래시 컨트롤러와 NVMe 컨트롤러는 각각 플래시 IO와 호스트 IO를 담당한다. ARM core는 각 IP를 제어하는 펌웨어를 구동한다.

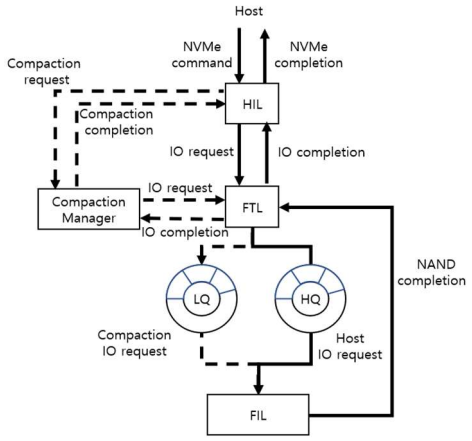


[그림 2] In-Storage Processing 플랫폼의 데이터 경로

3.2 소프트웨어 설계

3.2.1 펌웨어 설계

그림 3은 In-Storage processing 플랫폼에서 펌웨어의 데이터 제어 경로를 나타낸다. HIL(Host interface layer)는 호스트의 IO 명령 또는 컴팩션 오프로딩 명령이 담긴 NVMe 커맨드를 처리하는 역할을 수행한다. 일반적인 호스트의 IO는 FTL(Flash Translation Layer)에 전달되며, 오프로딩 명령은 컴팩션 매니저에 전달된다. 또한, FTL과 컴팩션 매니저에서 호스트의 명령이 완료되었다는 정보를 수신할 경우, 호스트의 NVMe 커맨드에 대한 반환을 수행한다. FTL은 컴팩션 매니저와 HIL에서 IO 명령을 수신하여 이를 플래시 명령으로 변환하여 FIL(Flash Interface Layer)에 전달한다. 컴팩션 매니저의 IO 명령은 LQ(Low Queue)에 삽입되며, 호스트의 IO 명령은 HQ(High Queue)에 삽입된다. 또한 FIL에서 플래시 IO가 끝났다는 정보를 수신할 경우 이를 HIL이나 컴팩션 매니저에 전달한다. FIL에서는 플래시 명령을 처리하는데, 오프로딩 된 작업에 의해 호스트의 IO가 방해받지 않도록 항상 HQ의 명령을 먼저 처리한다. 컴팩션 매니저는 컴팩션 IP의 제어와 컴팩션 도중 필요한 메모리의 관리, 그리고 FTL에 컴팩션에 필요한 IO 명령을 전달하는 역할을 수행한다. 먼저, IP에 전달할 SSTable에 필요한 버퍼를 할당하고, FTL에 SSTable을 적재할 것을 요청한다. FTL에서 IO 완료를 수신할 경우, IP에 해당 버퍼의 주소와 SSTable의 크기를 전송한다. IP의 동작에 필요한 최소한의 SSTable이 적재되었을 경우, 출력 버퍼를 할당한 뒤 IP를 동작 시키고 모니터링을 시작한다. 모니터링 중에 IP에서 읽기가 끝난 버퍼가 있을 경우 해당 버퍼에 다음 SSTable이 적재되도록 요청하고, 출력이 완료된 SSTable이 있을 경우 해당 SSTable을 플래시에 기록하도록 요청한다. 컴팩션이 완료되었을 경우, HIL에 컴팩션의 결과 생성된 SSTable들의 정보를 HIL에 전달한다.



[그림 3] 펌웨어의 제어 경로

3.2.1 LevelDB 수정

LevelDB에서 Memtable Flush와 컴팩션을 수행하는 *BackgroundCompaction()* 함수를 수정하여 구현하였다. 해당 함수가 호출되었을 경우, 기존의 컴팩션 과정을 수행하는 대신 별도의 컴팩션 스레드를 추가로 생성하며, 컴팩션 스레드는 컴팩션의 대상이 되는 SSTable의 정보를 NVMe 커맨드를 통해 SSD에 전달한 뒤 오프로딩의 결과를 수신할 때까지 대기한다. 결과가 반환된 다음에는 오프로딩의 결과 새로 생성된 SSTable들의 정보를 DB의 메타데이터에 업데이트한 뒤 스레드를 종료한다.

4. 평가

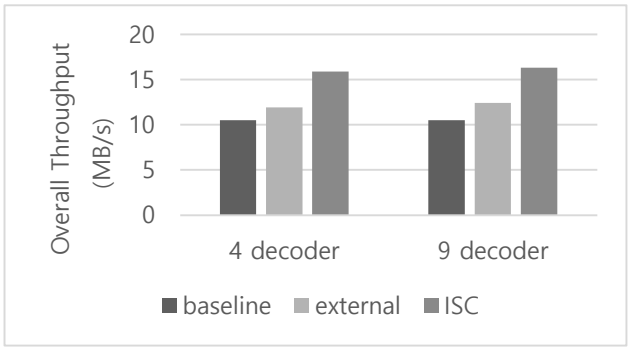
4.1 실험 환경

In-Storage Processing 플랫폼은 Cosmos+ OpenSSD에서 구현되었으며 200Mhz에서 동작한다. 플래시 컨트롤러는 4 Channel 4 Way로 설정되었다. 외장 가속기의 경우 Xilinx ZCU106[7]을 사용하였다. 호스트 PC는 I7 4790, 12GB

DRAM을 사용하였으며, 벤치마크에는 db_bench의 fillrandom workload를 이용하여 1억번의 레코드 삽입을 실험하였고, SSTable의 최대 크기는 32MB로 설정하였다

4.2. 벤치마크 전체 성능 평가

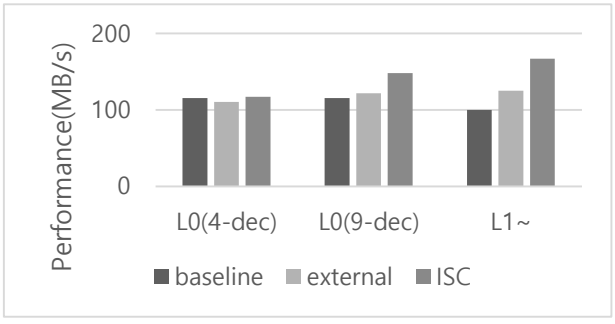
그림 5는 벤치마크의 전체 성능을 보여준다. 디코더의 개수가 4개일 경우, 외장 가속기 환경에서는 13.3% 성능 증가를 보였고, In-Storage Processing 환경에서는 51.4% 성능 향상을 보였다. 디코더의 개수가 9개로 증가할 경우, 외장 가속기 환경과 In-Storage Processing 환경에서 각각 18.1%, 55.2% 성능 향상을 보였다.



[그림 5] 벤치마크 성능

4.3 컴팩션 성능 평가

그림 6은 디코더의 수에 따른 컴팩션 IP의 처리량을 보여준다. 컴팩션의 결과 생성된 SSTable의 수를 각 컴팩션에 걸린 시간으로 나눈 값으로 계산하였으며, 단계적 컴팩션에 의하여 발생하는 성능 저하를 확인하기 위하여 Level 0 컴팩션과 Level 1 이상의 컴팩션을 별도로 집계하였다. Level 0 컴팩션의 경우, 연속 컴팩션으로 인한 성능 저하로 인하여 4개 디코더의 경우에는 외장 가속기의 경우 호스트에서 처리하는 것과 유사한 성능이 측정되었으며, In-Storage Processing 버전의 경우에도 6.3%의 성능 향상을 보였다. 9개 디코더의 경우 연속 컴팩션으로 인한 성능 저하가 완화되어 호스트와 비교하여 각각 9%, 32.9% 성능 향상을 보였다. Level 1 이상의 컴팩션의 경우, 단계적 컴팩션으로 인한 성능 저하가 없기 때문에 각각 28.6%, 71.2% 성능이 향상되었다.



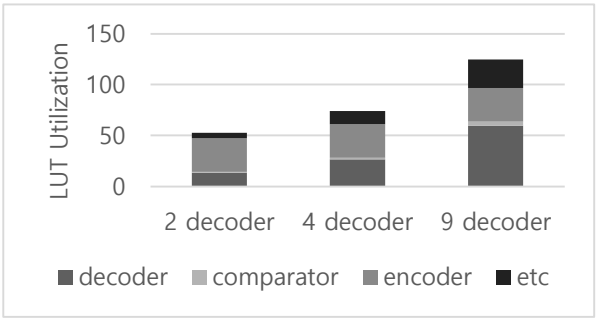
[그림 6] 컴팩션 처리량 비교

4.4 컴팩션 중 IO 속도 평가

컴팩션 중 SSTable의 읽기 속도와 쓰기 속도에 대한 측정을 진행하였다. In-Storage processing 버전의 경우에는 읽기/쓰기 속도가 평균 368MB/s, 271MB/s로 측정된 반면, 외장 가속기 버전의 경우 262MB/s, 243MB/s로 컴팩션 IP의 성능을 제대로 활용하지 못하였다.

4.5 리소스 사용량 평가

그림 7은 디코더의 수에 따른 LUT 사용량의 변화를 보여준다. 인코더에 필요한 리소스 사용량은 동일하지만, 나머지 부분은 디코더의 수에 비례하여 증가하는 것을 확인할 수 있다. 특히, 9개 디코더를 사용할 경우, Cosmos+ LUT의 90% 이상을 사용하였다.



[그림 7] 디코더 수에 따른 LUT 사용량 변화

4.6. CPU 사용량

컴팩션을 오프로딩 하지 않을 경우의 벤치마크 중 평균 CPU 사용량은 52.3%, 외장 가속기 버전은 39.9%, In-Storage Processing 버전은 33.3%로 측정되었다. 외장 가속기를 이용할 경우에는 연산에 필요한 CPU 사용량은 감소하는 반면 가속기의 제어에 CPU가 사용되지만, In-Storage Processing의 경우에는 호스트는 오프로딩 명령이 응답할 때까지 대기하고 가속기 제어 등의 작업은 스토리지 내부 펌웨어가 처리하기 때문에 추가적으로 CPU 사용량이 감소한다.

5. 결론

본 논문에서는 LSM-Tree의 컴팩션을 가속할 수 있는 FPGA Cosmos+ OpenSSD와 통합하였고, 외장 가속기 플랫폼과의 성능 비교를 진행하여 In-Storage Processing 플랫폼이 가지는 이점을 확인하였다.

참고문헌

[1] P. O'Neil et al, "The log-structured merge-tree (LSM-tree)", *Acta Informatica,* 33, 4(1996), 351-385

[2] Teng Zhang et al, "FPGA-Accelerated Compactions for LSM-based Key-Value Store", *18th USENIX Conference on File and Storage Technologies (FAST 20),* 2020, pp. 225-237

[3] Y. H. Song et al., "Cosmos+ openssd: A nvme-ba sed open source ssd platform", *Flash Memory Summ it*, 2016.

[4] X. Sun et al, "FPGA-based Compaction Engine for Accelerating LSM-tree Key-Value Stores", *2020 IEE 36th International Conference on Data Engineering (ICDE).* 2020, pp. 1261-1272

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

[6] 임민제, 신동군, "연속 컴팩션 기법을 이용한 LSM-tree의 FPGA 컴팩션 가속기", *한국정보과학회 2021 한국컴퓨터종합학술대회 논문집*, 2020.6, pp.1508-1510

[7] https://www.xilinx.com/products/boards-and-kits/zcu106.html