Gen-Z 메모리 디바이스를 활용한 LevelDB의 성능 평가

강현구[1], 이창규[1], 이희락[1], 석성우[2], 마진석[2], 오명훈[2], 김영재[1]
[1]서강대학교 컴퓨터공학과, [2]한국전자통신연구원
{hyeongu, changgyu, heerock, youkim}@sogang.ac.kr, {majinsuk, mhoonoh, swsok}@etri.re.kr

Performance Evaluation of LevelDB on Gen-Z Memory Device

Hyeongu Kang[1], Chang-Gyu Lee[1], Heerock Lee[1], Seong-Woo Seok[2], Jinseok Ma[2], Myeong-Hoon Oh[2], Youngjae Kim[1]
[1]Department of Computer Science and Engineering, Sogang University, Republic of Korea,
[2]ETRI, Republic of Korea

요 약

메모리 중심 컴퓨팅은 CPU와 메모리 간의 병목 현상을 해결하기 위해 제안된 구조로, Gen-Z 프로토콜에 기반한 Fabric Attached Memory (FAM)을 구성하여 Shared Memory Pool을 제공한다. 본 논문에서는 FAM의 기반이 되는 최초의 Gen-Z 메모리 디바이스 프로토타입의 성능 평가를 수행하였다. Block 단위의 평가를 위해 FIO 벤치마크를 이용했고 Shared Memory Pool에서 오브젝트 관리 성능 평가를 위해 LevelDB에서 db-bench를 이용한 Key-Value Store 성능 평가를 수행했다. Gen-Z Memory Device (GZD) 프로토타입의 내부 캐시 성능 평가를 위해 CPU 캐시를 활용해 디바이스 캐시를 에뮬레이션했으며 캐시 정책은 WC (Write-Combined), WT (Write-Through), UC (Uncached)를 사용하여 실험을 진행했다. LevelDB의 db-bench를 이용하여 실험한 결과, 쓰기 성능은 WC 캐시 정책을 사용할 때 캐시를 사용하지 않는 UC보다 최소 11%, 최대 47%의 latency 감소를 보였으며 읽기 성능은 WT 캐시 정책을 사용할 때 UC보다 최소 31%, 최대 99%의 latency 감소를 보였다.

1. 서론

Machine Learning 기술의 발전으로 시스템이 처리할 데이터와 연산의 양은 점점 증가하고 있다. 이를 뒷받침할 프로세서의 성능은 지속적으로 발전하고 있으나 메모리의 성능 향상은 그에 미치지 못하고 있다. 또한 클러스터 환경에서 공유 메모리를 사용하는 코어 수가 증가함에 따라 노드 간 동기화 비용이 증가하고 메모리 모듈의 성능 향상을 온전히 이용하지 못하는 현상이 발생한다 [1].

메모리 중심 컴퓨팅 (Memory Centric Computing, MCC)은 Large-scale system에서 CPU와 메모리 간의 병목 현상을 해결하기 위해 제안되었다 [2]. 메모리 중심 컴퓨팅에서는 메모리가 패브릭 네트워크 (Fabric Network)에 연결되어 Shared Memory Pool을 구성하는데 이를 FAM (Fabric Attached Memory)이라 한다. 공유 메모리 접근 시 다른 CPU가 관여하는 기존 시스템과 달리 FAM을 사용한 시스템은 각 CPU가 패브릭을 통해 FAM에 직접 접근할 수 있다.

Gen-Z는 메모리 중심 컴퓨팅을 위한 Fabric 프로토콜로 FAM을 구성하는 기술이다 [3]. Gen-Z 프로토콜을 통해 연결된 FAM은 Global Address Space (GAS)를 통해 시스템 내의 모든 노드에서 접근할 수 있다. 시스템 내 모든 디바이스가 Gen-Z 프로토콜로 통신하기 때문에 시스템 확장에 따른 동기화 오버헤드가 적다. 즉 메모리 중심 컴퓨팅은 패브릭 네트워크에 새로운 메모리를 추가해도 Shared Memory Pool의 capacity 증가에 따른 오버헤드가 적어 bandwidth와 latency의 저하없이 시스템을 확장할 수 있다.

최근 Key-Value Store (KVS)는 간단한 Key-Value 인터페이스를 사용하여 NVM (Non-Volatile Memory)를 위한 인터페이스로 주목받고 있다. 이와 관련한 여러 연구는 작은 capacity의 NVM을 활용하여 Key-Value Store의 성능을 높이는데 초점을 맞췄다 [4, 5]. 반면 FAM은 큰 메모리 capacity를 제공하기 때문에 GZD (Gen-Z Memory Device)로 NVM을 사용한다면 기존의 스토리지 영역을 대체하여 Key-Value Store의 데이터를 FAM에 저장할 수 있다.

본 논문에서는 Key-Value Store를 사용하여 GZD 프로토타입의 성능을 평가한다. 이를 위해 FIO [6]를 사용하여 스토리지로서의 GZD의 성능을 평가하고, LevelDB를 사용하여 읽기, 쓰기 시의 latency를 측정했다. 파일 시스템은 Ext4-DAX [7]를 사용했으며, GZD에 내부 캐시가 없는 관계로 CPU 캐시를 사용하여 내부 캐시를 에뮬레이션했다. 캐시 정책은 WC (Write-Combined), WT (Write-Through), UC (Uncached)를 사용했다. 쓰기 워크로드는 db-bench [8]의 Sequential Put, Random Put 그리고 Random Sync를 사용했으며 읽기 워크로드는 Sequential Get과 Random Get을 사용하였다. 각 워크로드에 대한 특징은 4.1 실험 환경에 설명되어 있다. 실험 결과, 쓰기 성능은 WC를 사용할 때 캐시를 사용하지 않는 UC보다 11-47%의 latency 감소를 보였으며, 읽기 성능은 WT를 사용할 때 UC보다 latency가 31-99% 감소했다.

이 논문은 2020년도 정부(과학기술정보통신부)의 재원으로 정보통신 기획 평가원의 지원을 받아 수행된 연구임 (No.2018-0-00503, 메모리중심 차세대 컴퓨팅 시스템 구조 연구)

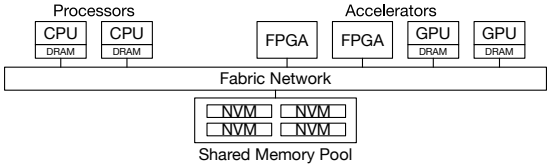


그림 1: 메모리 중심 컴퓨팅 구조

2. 연구 배경

2.1 메모리 중심 컴퓨팅과 Gen-Z

메모리 중심 컴퓨팅은 처리할 데이터가 증가하면서 데이터 접근으로 인한 병목현상을 해결하기 위해 제안된 구조이다. 그림 1은 메모리 중심 컴퓨팅 구조를 나타낸다. 기존 컴퓨팅 구조에서 메모리가 CPU에 종속되는 반면, 메모리 중심 컴퓨팅에서 모든 메모리는 FAM에 소속되며 CPU는 패브릭 네트워크를 통해 FAM에 접근한다. 따라서, FAM에 대한 접근은 패브릭을 통해서만 이루어지고 다른 CPU의 개입을 배재한다.

그러나 데이터 병목 현상을 해결하기 위해서는 패브릭 네트워크와 FAM의 성능이 뒷받침되어야 한다. Gen-Z는 메모리 중심 컴퓨팅 내 모든 장치의 커뮤니케이션에 사용되는 통합 인터페이스이자 프로토콜로 높은 bandwidth와 낮은 latency를 제공한다. CPU의 메모리 컨트롤러와 메모리의 미디어 컨트롤러는 Gen-Z 프로토콜을 사용하여 커뮤니케이션하므로 메모리는 더이상 특정 CPU에 종속되지 않는다. 즉, 메모리가 스스로 장치를 제어하며 모든 노드가 Fabric을 통해 메모리에 접근한다.

2.2 LevelDB on NVM

LevelDB [8]는 LSM-Tree (Log-Structured Merge-Tree) [9]를 기반으로 한 Key-Value Store이다. LevelDB의 구조는 그림 2와 같다. LevelDB는 DRAM 상의 자료구조로 Memtable을 사용한다. Memtable은 삽입된 Key-Value Pair를 일시적으로 저장하는 역할을 한다. LevelDB는 failure consisentcy를 위해 WAL (Write Ahead Logging)을 사용한다. Memtable 크기가 임계값을 넘으

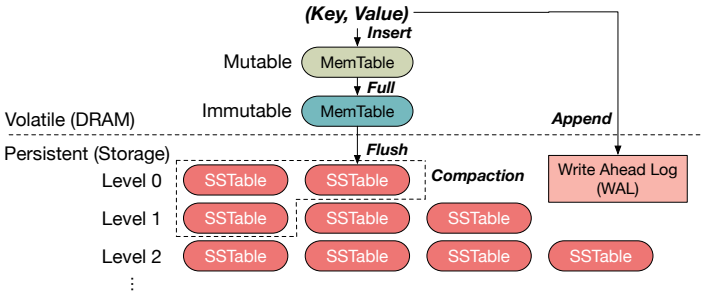
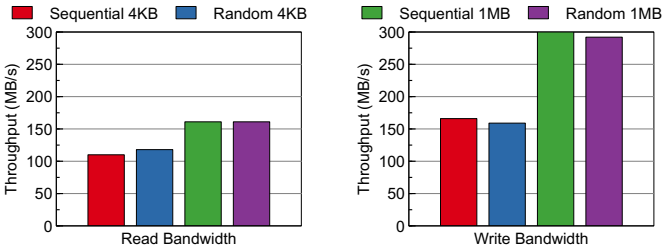
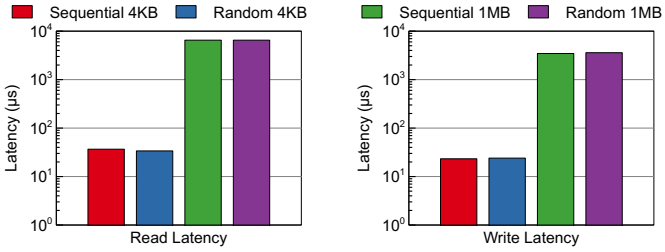


그림 2: LevelDB 구조



(a) Read (b) Write

그림 3: Bandwidth 측정 결과



(a) Read (b) Write

그림 4: Latency 측정 결과

면 Immutable Memtable이 되고, 이후 flush되어 SSTable (Sorted String Table)로 스토리지에 저장된다. LevelDB는 여러 level로 구성되어 있으며 각 level은 여러 SSTable로 구성된다. 각 level은 이전 level의 10배 크기의 공간을 할당 받고, level에 할당된 공간을 모두 사용할 경우 compaction이 실행된다. compaction은 상위 level의 SSTable들을 병합하는 과정에서 중복된 키나 삭제된 키를 제거하고 Key-Value Pair를 정렬하여 다음 level의 SSTable로 만든다.

LevelDB에서의 검색은 저장과 같은 순서로 수행된다. 우선 Memtalbe을 탐색하고 Key-Value Pair를 찾지 못하면 Level 0부터 Level N까지 SSTable을 탐색한다. LSM-Tree에서 상위 level에 존재하는 데이터가 최신 데이터이므로 원하는 Key-Value Pair를 찾는 즉시 사용자에게 리턴한다.

NVM의 등장 이후 Key-Value Store에서 NVM을 활용하려는 연구들이 있었다. 이 연구들은 Key-Value Store의 Memtable이나 인덱스 자료구조를 NVM에 적용하여 Key-Value Store의 성능을 최적화했다 [4, 5]. 본 연구에서는 LevelDB를 사용하여 FAM이 NVM을 사용하는 GZD로 구성된 경우 SSTable을 FAM에 저장했을 때의 Key-Value Store의 성능을 파악한다.

3. 실험 설계

3.1 Gen-Z Memory Device 및 캐시 효과 에뮬레이션

실험에 사용한 GZD는 FPGA를 통해 Gen-Z 프로토콜을 구현한 프로토타입으로 SDRAM을 사용한다. 본 실험에서는 DMA 인터페이스를 통해 GZD를 Block 디바이스로 사용했다. CPU 캐시로 에뮬레이션한 GZD의 내부 캐시에 대한 정책 세팅에는 리눅스 커널의 MTRR (Memory Type Range Register) [10]과 PAT (Page Attribute Table) [11]를 활용했다. MTRR과 PAT는 메모리 주소 공간에 대한 캐시 정책을 지정하는 기능으로, MTRR이 지정한 메모리 범위 내에서 PAT를 통해 페이지 단위로 캐시 정책을 세팅한다. 본 논문에서는 FAM으로 하나의 GZD를 사용하여 실험을 진행했으며 향후 여러 개의 GZD가 연결된 Shared Memory Pool을 구성하여 실험할 계획이다. 캐싱 정책으로는 메모리 영역을 캐싱하지 않는 UC (Uncached), 캐시와 메모리에 동시에 write하는 WT (Write-Through) 및 WCB (Write Combine Buffer)를 활용하여 메모리에 write하는 WC (Write-Combined) 정책 [12]을 사용했다.

3.2 Key-Value Store

본 논문에서는 그림 2의 storage 영역에 GZD를 사용할 때 Key-Value Store의 성능을 측정하고자 한다. FAM은 높은 capacity와 scalability를 제공하며, 패브릭을 통해 높은 bandwidth로 데이터를 제공한다. FAM이 NVM으로 구성된다면 Memtable이나 인덱스처럼 크기가 작은 DRAM 자료구조보다 데이터를 저장하여 크기가 큰 SSTable를 사용하는 것이 더 적합하다. 이처럼 Key-Value Store는 FAM에 적합한 특징을 가지고 있기 때문에 스토리지 Key-Value Store 중 대중적인 LevelDB를 사용하여 실험을 진행했다.

4. 실험 결과

4.1 실험 환경

실험은 AMD Ryzen Threadripper 1950X (16 Cores, 32 Threads) 프로세서와 32 GB의 메모리를 가진 시스템에서 진행했으며 Linux Kernel 5.0.21을 사용했다. 파일 시스템은 Ext4-DAX [7]를 사용했다. 성능 평가에는 FIO [6]를 사용했으며 읽기와 쓰기에 대해 bandwidth와 latency를 측정하였다. 디바이스는 multi queue를 지원하지 않으며, queue depth를 1로 설정하였고 queue depth 변화에 따른 성능 변화는 없었다. Block 크기는 4 KB와 1 MB를 사용하였다.

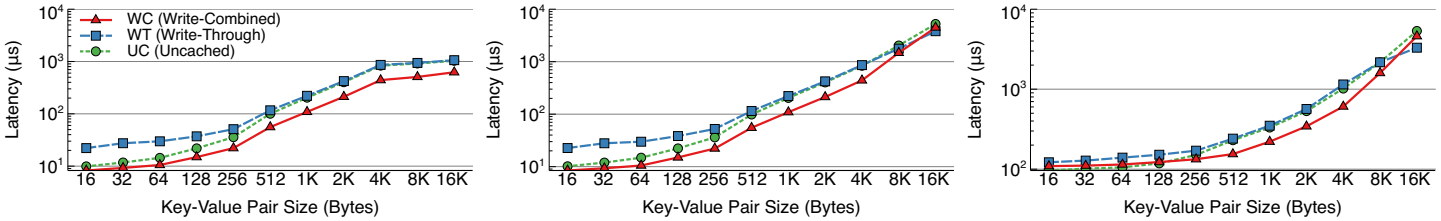
Key-Value Store 성능 평가는 LevelDB가 제공하는 db-bench [8] 벤치마크를 사용했다. 쓰기 워크로드는 Sequential Put, Random Put, Random Sync를 사용했으며 읽기 워크로드는 Sequential Get, Random Get을 사용했다. Sequential Put과 Random Put은 Key-Value Pair (KV Pair)에 대해 각각 sequential 및 random key 순서로 asynchronous한 쓰기를 수행하는 반면, Random Sync는 KV Pair에 대해 random key 순서로 synchronous한 쓰기를 수행한다. Sequential Get과 Random Get은 KV Pair에 대해 각각 sequential 및 random key 순서로 asynchronous한 읽기를 수행한다.

KV Pair 크기는 16 B부터 16 KB까지 2배씩 늘려가며 10,000개의 KV Pair를 사용했으며 GZD가 사용하는 메모리 주소 공간에 대한 CPU 캐시 정책 설정을 위해 MTRR과 PAT 값을 UC, WT 또는 WC로 세팅하여 캐시 사용 여부를 조정했다.

4.2 GZD 프로토타입 latency 및 bandwidth 측정

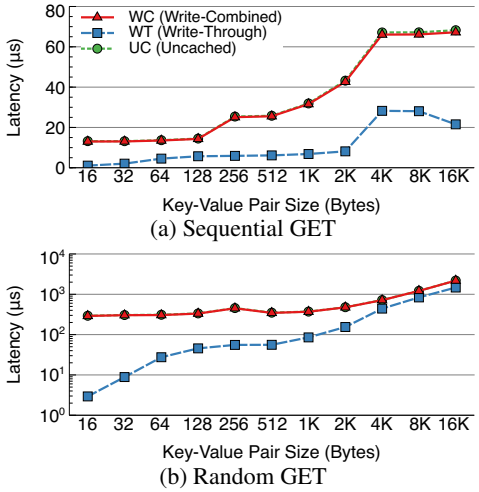
그림 3과 4는 GZD의 읽기 및 쓰기 성능을 보여준다. Sequential 및 Random I/O 실험 결과, I/O 패턴보다 크기가 성능에 더 큰 영향을 미쳤다. 읽기 bandwidth는 I/O 크기가 4 KB인 경우 약 110 MB/s, 1 MB인 경우 약 160 MB/s로 1.46 배의 향상을 보였다. 쓰기 bandwidth는 I/O 크기가 4 KB인 경우 약 160 MB/s, 1 MB인 경우 약 300 MB/s로 1.8 배의 향상을 보였다.

반면 latency는 I/O 크기가 커질수록 높아져 성능이 하락하는 경향을 보였다. 읽기 latency는 I/O 크기가 4 KB인 경우 약 35 µs, 1 MB인 경우 약 6500 µs로 177 배의 향상을 보였다. 쓰기 latency는 I/O 크기가 4 KB인 경우 약 25 µs, 1 MB인 경우 약 3500 µs로 148 배의 향상을 보였다.



(a) Sequential PUT (b) Random PUT (c) Random SYNC

그림 5: GZD에서 LevelDB의 쓰기 성능



(a) Sequential GET

(b) Random GET

그림 6: GZD에서 LevelDB의 읽기 성능

4.3 GZD의 LevelDB Latency 평가

그림 5는 GZD에서 LevelDB의 쓰기 성능을 보여준다. 캐시 정책으로 WC를 사용한 경우 WCB로 인해 모든 쓰기 워크로드에서 가장 낮은 latency를 보이며 UC에 비해 latency가 11-47% 낮다. 각 워크로드는 KV Pair 크기가 커질수록 DB 크기 증가로 인해 GZD 접근이 늘어나 latency가 증가한다.

모든 쓰기 워크로드에서 WT와 UC는 KV Pair 크기가 커질수록 latency 차이가 작아지는 경향을 보인다. KV Pair 크기가 16-256 B인 경우 전체 DB 크기가 작아 WAL (Write Ahead Log)에 write할 때만 GZD로의 접근이 발생하여 512 B 이상일 때보다 GZD 접근 횟수가 적다. 따라서 WT로 인한 오버헤드가 캐시 효과보다 상대적으로 커져 UC에 비해 높은 latency를 보인다. 그러나 KV Pair 크기가 512 B를 넘어가면 flush가 발생하고 4 KB를 넘어가면 compaction이 발생한다. 이에 따라 latency가 증가하며 WT의 오버헤드가 상대적으로 줄어들어 UC와 latency 차이가 감소한다.

Random Put은 KV Pair 크기가 8-16 KB일 때 Sequential Put보다 높은 latency를 보인다. Sequential Put은 key가 정렬된 순서로 삽입되므로 SSTable는 정렬된 상태로 저장되어 여러 SSTable을 읽고 중복 제거 및 정렬을 수행하는 compaction 비용이 크지 않은 반면, Random Put은 random order로 key가 삽입되므로 compaction 비용이 커져 latency가 증가한다.

Random Sync는 매 write마다 synchronous IO를 통해 GZD에 write를 수행한다. KV Pair 크기가 16-128 B일 때 WC가 UC보다 4-13% 높은 latency를 보이는데, 이는 매 operation마다 GZD에 데이터를 write하여 캐시 오버헤드가 캐시 효과보다 크기 때문이다.

그림 6은 GZD에서 LevelDB의 읽기 성능을 보여준다. 읽기 워크로드에서는 WT 정책이 가장 낮은 latency를 보이며 UC에 비해 latency가 31-99% 낮다. WC는 읽기에 대해 캐시 효과를 볼 수 없어 UC와 비슷한 latency를 보인다. 읽기 성능 역시 쓰기의 경우처럼 KV Pair 크기가 커질수록 latency가 증가한다.

Sequential Get은 KV Pair 크기가 작은 경우 Memtable과 LevelDB 캐시로 인해 GZD 접근 횟수가 적어 낮은 latency를 보인다. 하지만 KV Pair 크기가 커질수록 level이 깊어지고 탐색할 데이터가 증가한다. 즉 GZD 접근 횟수가 증가하여 latency가 증가한다.

Random Get은 random한 순서의 key를 찾기 때문에 탐색할 SSTable의 수가 증가하여 높은 latency를 보인다. WT는 작은 크기의 KV Pair에 대해 읽기에 대한 캐시 효과로 WC와 UC에 비해 낮은 latency를 보이지만, KV Pair 크기가 증가함에 따라 DB가 커지고, 캐시 미스의 증가로 GZD 접근 횟수가 증가함에 따라 WT의 latency가 WC와 UC에 수렴한다.

4.4 Discussion

실험에서 사용한 GZD 프로토타입은 사용된 메모리 컨트롤러 및 Gen-Z 프로토콜 처리 성능이 최적화되지 않아 전반적으로 낮은 성능을 보이며 드라이버에서 single queue만 지원하는 한계가 있다. 또한 Device DAX [13]를 지원하지 않아 Block 디바이스로만 활용 가능하다. 추후 프로토타입의 성능이 개선되고 multi queue와 Device DAX 기능이 제공되면 더 높은 성능을 보일 것으로 예상된다.

5. 결론

본 논문에서는 메모리 중심 컴퓨팅의 패브릭 네트워크를 구성하는 Gen-Z Memory Device를 사용하여 LevelDB의 성능을 평가했다. 디바이스에 캐시가 없는 관계로 CPU 캐시를 내부 캐시로 에뮬레이션하여 실험을 진행했다. LevelDB의 db-bench를 이용한 실험 결과, 쓰기 워크로드에서 WC 캐시 정책을 사용할 경우 캐시를 사용하지 않을 때보다 11-47% 낮은 latency를 보였으며, 읽기 워크로드는 WT 캐시 정책을 사용할 경우 캐시를 사용하지 않을 때보다 31-99% 낮은 latency를 보였다.

참고 문헌

[1] J. D. McCalpin, "Memory bandwidth and system balance in hpc systems," *Invited talk, SC*, 2016.

[2] K. Keeton, "Memory-Driven Computing," in *USENIX FAST)*, 2017.

[3] Gen-Z Consortium, "Gen-Z Overview." https://genzconsortium.org/.

[4] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning LSMs for nonvolatile memory with NoveLSM," in *USENIX ATC*, 2018.

[5] F. Xia, D. Jiang, J. Xiong, and N. Sun, "HiKV: A hybrid index key-value store for dram-nvm memory systems," in *USENIX ATC*, 2017.

[6] Jens Axboe, "Flexible I/O Tester." https://github.com/axboe/fio.

[7] "Direct Access for files." https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

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

[9] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, 1996.

[10] The Kernel Development Community, "MTRR control." https://www.kernel.org/doc/html/latest/x86/mtrr.html.

[11] The Kernel Development Community, "PAT." https://www.kernel.org/doc/html/latest/x86/pat.html.

[12] Advanced Micro Devices, "Amd64 architecture programmer's manual volume 2: System programming," 2020.

[13] D. Williams, "Device DAX for persistent memory." https://lwn.net/Articles/686664/.