호스트 수준 FTL 쓰기 동작의 병목 현상 분석

진주영 O, 김혁중, 신동군
성균관대학교 전자전기컴퓨터공학과
siennajjy@skku.edu, wangmir@gmail.com, dongkun@skku.edu

Analysis on Bottleneck of Write Operation at Host-level FTL

Jhuyeong Jhin O, Hyukjoong Kim, Dongkun Shin
Department of ECE, Sungkyunkwan University

요 약

서버 시스템 환경에서는 다중 사용자의 동시 접근을 효율적으로 처리해야 한다. 최근 대용량 서버 시스템들이 플래시 메모리 기반 저장장치인 SSD 장치를 많이 사용하고 있는데, SSD는 동시에 여러 개의 read/write 연산을 수행할 수 있는 장점이 있다. SSD의 병렬성을 최대한 활용하기 위해 NVMe 인터페이스는 멀티 큐 메커니즘을 제공하고, 블록 다중 큐는 CPU 코어 별로 큐를 생성하여 블록 레이어의 단일 큐에 의한 병목 현상을 제거하였다. 한편 호스트 시스템에서의 SSD 장치 관리에 대한 필요성이 부각되면서, 호스트 시스템이 사용자 워크로드나 SSD 상태에 따라 설정 가능한 환경을 구축할 수 있도록 하는 호스트 수준 FTL에 대한 많은 연구가 진행되고 있다. 그러나 현재 호스트 FTL은 단일 스레드 구조를 가진 FTL을 단순 계층 이동으로 구현하였고, 이로 인해 호스트 FTL이 다중 사용자 스레드에 의한 요청들을 동시에 처리하지 못해 병목 현상의 원인이 된다. 본 논문에서는 호스트 수준 FTL의 단일 스레드 구조로 인해 생기는 문제점을 실험을 통해 분석하였다.

1. 서 론

최근 SSD와 같은 플래시 저장장치가 모바일 시스템부터 서버 시스템까지 다양한 분야에서 사용되고 있으며, 성능 및 용량 역시 크게 향상되었다. 이러한 SSD는 낸드 플래시 메모리의 erase-before-write와 같은 특성으로 인해 flash translation layer (FTL)라는 시스템 소프트웨어 계층을 필요로 한다. FTL은 SSD의 플래시 메모리를 관리하기 위한 주소 변환, 가비지 컬렉션(GC), 웨어-레벨링 등의 역할을 담당한다.

기존 SSD 시스템에서 FTL은 SSD 장치의 내부 펌웨어로 구현되어 있었으며, 호스트 시스템에서는 SSD의 구조나 FTL의 내부 동작에 대하여 알 수가 없었다. 그래서 플래시 메모리의 연산(읽기, 쓰기, 지우기)들의 성능은 예측 가능함에도 불구하고, 호스트 시스템이 요청한 I/O에 대한 SSD의 성능을 예측하는 것은 불가능하였다. 또한, 플래시 스토리지가 사용되는 분야가 다양해지면서 호스트의 요구사항에 따른 개인화(customizing)가 필요해지게 되었다. 이에 따라 최근 연구에서는 FTL 계층을 SSD 내부가 아닌 호스트 수준에 구현하는 시도를 하였다[2]. 그림 1의 (a)는 SSD 장치 수준의 임베디드 FTL을, (b)는 호스트 시스템 수준의 호스트 FTL를 이용하는 시스템의 I/O 스택을 간략하게 나타낸다. 그림 1(a)의 임베디드 FTL은 SSD 장치 내부에 구성되어 있기 때문에 스토리지 내부 펌웨어에서 SSD 장치를 관리한다. 반면 그림 1(b)의 호스트 FTL은 스토리지 인터페이스 상단에서 SSD를 관리한다.

이 논문은 2016년도 정부(미래창조과학부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (No. 2016R1A2B2008672)

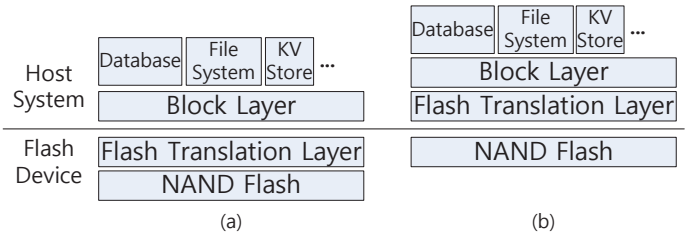

그림 1 FTL 계층의 위치에 따른 I/O 스택

한편, SSD의 성능이 크게 향상되어 수십만 IOPS의 대역폭을 제공하고, 대용량 서버 시스템에서는 다중 유저가 동시 다발적으로 스토리지에 많은 I/O를 발생시킨다. 이에 따라 최근 고성능 SSD는 I/O 명령을 병렬적으로 처리할 수 있는 인터페이스를 지원하며, 호스트 시스템의 I/O 스케쥴러 역시 멀티코어 시스템에서 I/O를 명령 처리속도를 향상시키기 위하여 멀티-큐 구조(blk-mq)를 차용하고 있다[3]. 하지만, 이에 비하여 FTL 소프트웨어에 대한 연구에서 다중 사용자 접근을 고려한 연구는 전무하였으며, 호스트 수준 FTL 역시 기존의 FTL 구조를 그대로 호스트 시스템에 구현했다.

본 논문에서는 이러한 멀티 유저 상황에서 현재 호스트 수준 FTL 구조의 문제점을 분석하도록 한다.

2. 호스트 FTL의 계층 구조

최근 스토리지 인터페이스는 SSD의 높은 IOPS를 최대한 활용하기 위하여 다중 하드웨어 큐를 지원한다. NVMe 인터페이스[4]는 기존 SATA 인터페이스에서 하나였던 하드웨어 요청 큐를 최대 64K개까지 제공하여 호스트 시스템의 멀티 코어가 동시에 여러 개의 명

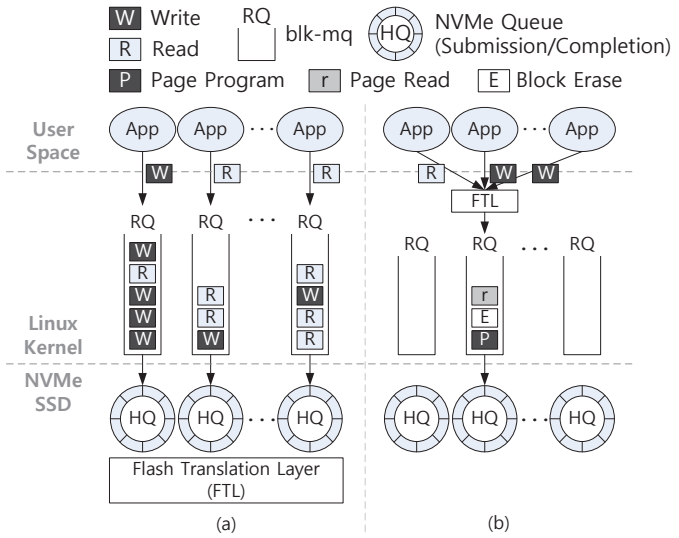

그림 2 FTL 계층의 위치에 따른 저장장치 스택 구조

령을 SSD 장치로 전달할 수 있다. 또한 blk-mq는 CPU 코어 별로 큐를 제공하여, 기존 블록 레이어의 단일 요청 큐에 대한 여러 스레드의 경쟁적 접근이 발생시키는 병목 현상을 제거하였다.

그림 2(a)와 2(b)는 각각 blk-mq와 NVMe 인터페이스를 사용하는 환경에서 임베디드 FTL과 호스트 FTL을 채택한 시스템에서의 I/O 처리 방식을 나타낸다. 그림 2(a)에서 사용자는 자신의 context가 실행 중인 코어의 큐에 I/O 요청을 삽입하기 때문에(blk-mq), 스레드 간의 충돌(contention)이 없이 코어 별로 동시에 I/O 처리가 가능하다. 이러한 I/O 요청은 NVMe 인터페이스의 다중 하드웨어 큐를 통해 SSD로 동시에 전달한다. 그러나 그림 2(b)에서 사용자의 I/O 요청은 다중 큐에 삽입되기 전 호스트 FTL 계층을 거쳐야 한다. 이 때, 기존의 FTL과 동일한 구조의 호스트 FTL는 매핑 테이블, 버퍼 등과 같은 글로벌한 자료구조를 싱글 스레드로 접근하면서 호스트 FTL에서 병목현상이 발생한다.

3. 관찰 및 분석

이 장에서는 리눅스 커널에 구현된 호스트 FTL 계층인 LightNVM에서 실험을 통해 현재 호스트 FTL 구조의 문제점을 파악하고, 여러가지 환경에 따른 성능 개선점을 관찰한다.

3.1 관찰 실험 환경

실험을 위해 본 논문에서는 LightNVM 기반 SSD 에뮬레이터를 구현하였으며, fio 벤치마크를 사용하였다[1]. fio 벤치마크의 워크로드는 4KB 단위의 임의쓰기를 direct I/O로 스레드 개수를 늘려가며(1~64) 요청하였다. SSD 에뮬레이터는 디바이스 드라이버 단에서 DRAM 메모리를 플래시 메모리로 에뮬레이션 하며, 상위 계층에서 LightNVM의 기본 호스트 FTL인 pblk를 사용한다. pblk에서는 모든 쓰기 데이터를 4KB 단위의 엔트리로 나누어 버퍼에 삽입하고, 싱글 writer 스레드가 버퍼에 쌓인 엔트리를 한 개씩 스토리지로 전달한다.

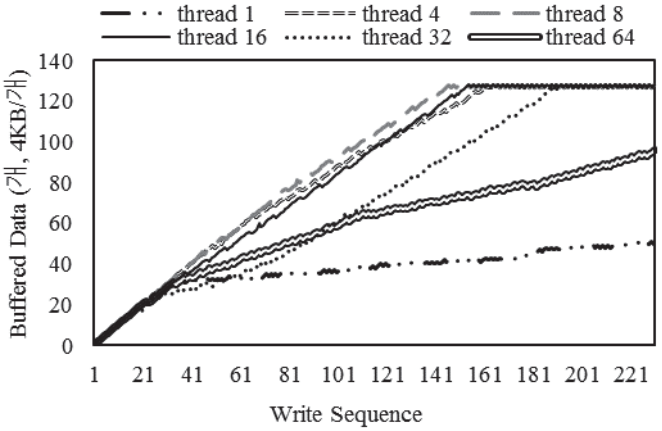

그림 3 사용자 스레드 개수에 따른 호스트 FTL 버퍼에 축적된 데이터 양의 변화

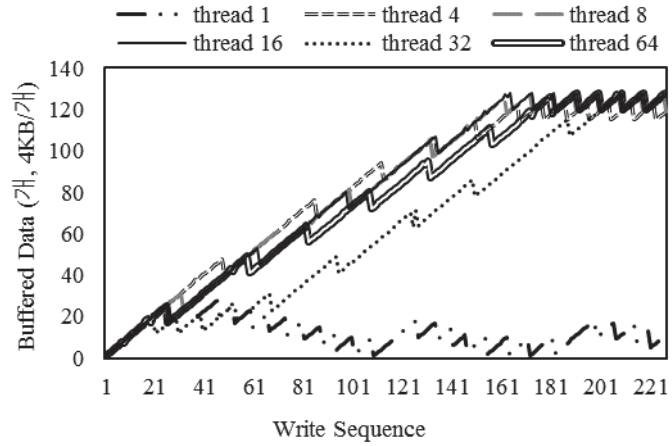

그림 4 사용자 스레드 개수에 따른 호스트 FTL 버퍼에 축적된 데이터 양의 변화 (WU=8)

버퍼는 최대 128개의 4KB 엔트리를 넣을 수 있다.

실험은 Intel Xeon 프로세서(2.40GHz, 16 cores) 기반의 서버에서 진행하였으며, 4GB의 DRAM 메모리를 스토리지 에뮬레이터를 위해 할당하였다.

3.2 단일 스레드 구조 FTL의 동작과 문제점

먼저 본 장에서는 유저 스레드 개수가 증가함에 따라서 싱글 writer 스레드에서 관리하는 버퍼에 I/O 요청이 쌓이는 양상을 관찰하였다. 그림 3은 I/O 요청의 진행에 따라서 버퍼에 쌓이는 I/O 양을 관찰한 그래프이다. 그래프에서 볼 수 있듯이 단일 유저 스레드가 쓰기 요청을 하는 환경에서는 버퍼에 I/O 요청이 쌓이는 속도가 멀티 스레드로 접근할 때보다 느린 것을 볼 수 있다. 이에 비해, 다중 유저 스레드가 I/O를 요청할 시에 단일 스레드에서 요청하는 것보다 버퍼에 I/O 요청이 쌓이는 속도가 빨라진 것을 확인할 수 있으나, 스레드의 개수에 비례하여 그 속도가 증가하지는 않았다. 본 실험을 통해 유저 스레드 개수가 증가할수록 요청속도는 빨라질 것이며, DRAM 메모리가 충분히 해당 대역폭을 지원할 수 있음에도 불구하고, 싱글 버퍼와 싱글 스레드로

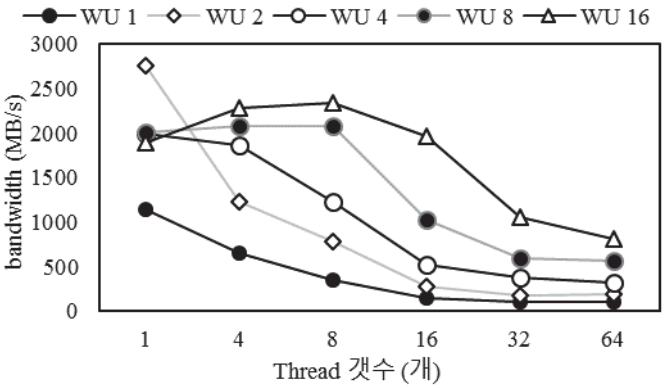

그림 5 사용자 스레드 개수와 WU에 따른 대역폭 비교

쓰기 요청을 처리하는 구조로 인해 버퍼에서 병목현상이 발생하는 것으로 예상할 수 있다.

그림 4는 버퍼에서 한번에 처리하는 엔트리의 개수(WU, write unit)를 변경하였을 때에 그림 3과 동일한 실험에 대한 결과를 보여준다. 본 실험에서는 WU를 기존 1개에서 8개로 변경하여, writer 스레드가 I/O를 처리하는 속도를 높이고자 하였다. 그래프를 보면, 싱글 유저 스레드에서는 버퍼에 I/O를 요청하는 속도보다 writer 스레드에서 I/O를 처리하는 속도가 빨라서 버퍼에 I/O가 쌓이지 않고 처리가 되는 것을 확인할 수 있다. 하지만, 스레드의 개수가 증가함에 따라 WU를 크게 설정하더라도 여전히 버퍼에 I/O 요청이 쌓이는 것을 관찰할 수 있었다.

그림 5에서는 스레드와 WU에 따라서 실제 대역폭이 달라지는 양상을 관찰하였다. 사용자 스레드가 1개인 경우를 제외하면, 모든 경우에 사용자 스레드 개수와 상관 없이 WU의 개수와 입출력 대역폭이 비례한다. 이는 writer 스레드가 I/O를 처리하는 양이 증가함에 따라 스레드들이 버퍼의 여유 공간을 기다리는 시간이 줄었기 때문이다. 그러나 일부 구간을 제외하고 WU를 증가시켜도 사용자 스레드 개수가 증가할수록 일부 구간을 제외하고 입출력 대역폭이 큰 폭으로 감소하는 양상이 동일하게 나타난다. 이러한 성능 저하의 원인은 pblk FTL의 구조에서 찾을 수 있다. pblk는 그림 2와 같이 단일 FTL 계층을 가지고 멀티유저가 하나의 버퍼와 스레드를 공유하며 쓰기 요청을 처리한다. 그래서 멀티 유저가 동시에 쓰기 연산을 요청하더라도, 버퍼에 저장할 때에는 해당 버퍼의 락(lock)을 보유한 단일 스레드만 한번에 접근이 가능하고, 또 버퍼에 쌓인 I/O 요청은 단일 writer 스레드를 통해서만 처리가 가능하다. 이러한 FTL 구조에서는 스토리지 전반에서 확장성(scalability)를 지원하고 있음에도 호스트 FTL에서 병목현상이 발생하여 멀티 유저에 따라 대역폭을 증가시킬 수 없다.

3.3 호스트 FTL의 writer 스레드 스케줄링

그림 6은 싱글/다중 유저 스레드가 쓰기 연산을 요청하는 상황에서 writer 스레드의 스레드 priority에 따른 처리 속도를 나타낸 그래프이다. 실험에서는 1개/64개의 유저 스레드가 쓰기연산을 요청하였다.

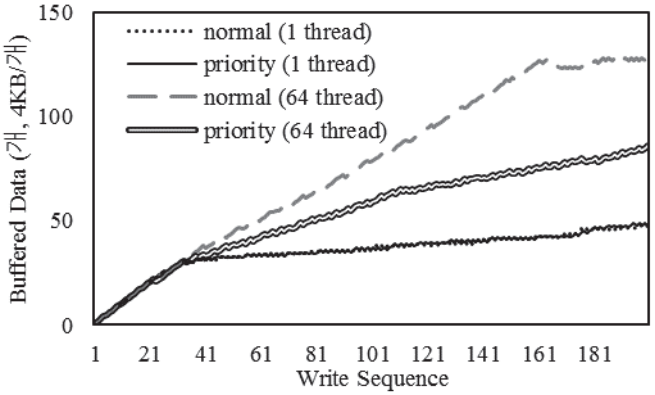

그림 6 Writer 스레드 priority에 따른 호스트 FTL 버퍼에 축적된 데이터 양의 변화 (WU=1)

그래프에서 볼 수 있듯이, 싱글 유저 스레드가 쓰기 연산을 요청할 때에는 writer 스레드의 priority를 변경하여도 쓰기 연산을 처리하는 속도에 영향이 없다. 이는 멀티 코어 환경에서 active하게 진행되는 스레드의 개수가 적어 priority에 따른 영향을 받지 않기 때문이다. 반대로 64개의 다중 유저 스레드가 쓰기 연산을 요청하는 경우에는 writer 스레드의 priority를 높임에 따라서 writer 스레드가 I/O를 처리하는 속도가 빨라져 버퍼에 I/O가 차는 속도가 느려 지는 것을 확인할 수 있다.

기존에 내부 펌웨어에서 FTL이 동작할 때에는 OS가 없는 환경에서 베어 메탈로 FTL 코드가 동작하였기 때문에 스레드 구조에 따른 스케줄링 이슈를 고려하지 않아도 되었다. 하지만, 호스트 FTL은 다양한 스레드와 함께 호스트 시스템 상에서 구동되어야 하기 때문에 스케줄링에 대한 고려가 필요할 것이다.

4. 결론 및 향후 계획

본 논문은 현재 단일 스레드 구조의 호스트 수준 FTL이 멀티 유저의 동시 접근을 처리하지 못해 병목현상을 발생시키는 것을 실험을 통해 관찰하였다. 실험을 통해 단일 스레드에서는 다중 입출력 요청들을 동시에 처리하지 못하기 때문에 사용자 스레드가 증가할수록 성능 저하가 심화되는 현상을 피할 수 없었다. 향후 연구에서는 이러한 호스트 FTL 스레드 구조를 개선하기 위한 연구를 진행할 계획이다.

참고문헌

[1] Flexible I/O Tester. https://github.com/axboe/fio.
[2] M. Bjorling, J. Gonzalez, P. Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In Proc. USENIX FAST'17, Santa Clara, CA, Feb. 2017.
[3] M. Bjorling, J. Axboe, D. Nellans, P. Bonnet. Linux block IO: Introducing multi-queue SSD access on multi-core systems., In Proc. SYSTOR '13. 2013.
[4] W. Choi, J. Zhang, S. Gao, J. Lee, M. Jung, M. Kandemir. An in-depth study of next generation interface for emerging non-volatile memories. NVMSA'16, 2016.