비휘발성 메모리를 이용한 빠르고 지속성 있는 저장장치 모듈 설계 및 구현

장형원*, 이상엽*, 조광일*, 정형수*
*한양대학교 컴퓨터공학부
e-mail : guddnjs91@gmail.com

Fast Durable Storage Module based on Non-Volatile Memory

Hyeongwon Jang*, Sang Youp Rhee*, Kwangil Cho*, Hyungsoo Jung*
*Department of Computer Science
Hanyang University, Seoul, Korea

요 약

데이터베이스 시스템의 트랜잭션 로깅이나 파일 시스템의 저널링에서 데이터 저장시 입출력 동기화(Synchronous I/O)는 올바른 프로그램 동작에 필수적이다. 하지만 입출력 동기화로 인한 프로그램의 지연 혹은 기다림은 응용 프로그램 성능의 저하를 가져온다. 본 논문에서는 차세대 저장장치인 비휘발성 메모리를 사용하여 지속성을 보장하며 쓰기 연산의 응답성을 개선하는 사용자 수준의 스토리지 모듈을 제안하고 기존의 동기화된 쓰기 연산과 성능을 비교하였다. 특히 멀티코어 환경에서 동시에 들어오는 여러 입출력 쓰기 연산 요청에 대하여 효율적으로 처리하였다.

1. 서론

현대의 컴퓨터는 저장매체의 성격에 따라 휘발성 메인 메모리와 비휘발성 저장장치의 계층적 구조를 이루고 있다. 휘발성 메인 메모리는 빠른 성능을 갖지만 저장공간 대비 가격이 비싸고 전원 공급이 끊기면 데이터가 손실되는 특징이 있다. 반면, 비휘발성 저장장치는 저렴한 가격과 무전원시에도 데이터 지속성을 지키는 특성이 있으나, 데이터 입출력 시 상대적으로 긴 지연 시간을 갖는다.

저장장치로의 입출력은 크게 동기 입출력과 비동기 입출력으로 나뉜다. 동기 입출력은 데이터가 저장장치에 완전히 저장된 후 응용의 수행이 재개될 수 있어서, 입출력에 의한 지연이 발생하고 그에 따라 응답성이 떨어지며 저장장치에서 제공하는 최대 대역폭을 활용하기 어려운 단점이 있다. 반면, 비동기 입출력은 데이터가 저장장치에 기록되는 것을 기다리지 않기 때문에 입출력에 대한 응답성은 개선되나 실제 데이터가 저장장치에 기록되는 것을 보장하지 않는다. 운영체제에서는 하드디스크와 같이 느린 저장장치를 사용하는 경우를 가정하여 비동기적 입출력을 우선으로 사용하고 있다[1]. 이에 반해 데이터베이스 로깅이나 파일 시스템의 저널링과 같이 동기화된 데이터 저장이 필요한 응용들은 동기 입출력이 필수적이다[2]. 특히 최근 멀티코어 컴퓨팅 환경이 일반화되면서 이러한 동시다발적인 입출력 요청의 빠른 처리는 고성능 데이터베이스 시스템에서 성능 개선을 위하여 반드시 해결되어야 할 문제사항이다.

최근 소개되고 있는 비휘발성 메모리(Non-Volatile Main Memory, NVRAM)[1] 는 전통적 메모리 계층구조에서 메인 메모리와 저장장치의 사이에 위치하며, 메인 메모리와 저장장치의 성능 간극을 줄이는 방향으로 활용할 수 있다[3][5]. 본 논문에서는 고성능 멀티코어 서버 환경에서 차세대 저장매체인 비휘발성 메모리를 활용하여 저장장치의 동기 입출력 성능을 개선한 시스템 모듈을 제안한다. 또한 동시적으로 발생하는 입출력 요청을 멀티코어 시스템을 활용하여 병렬적으로 처리하고, SSD 와 같은 고속 저장장치의 높은 대역폭을 최대한 활용할 수 있도록 하였다.

2 장에서는 본 논문에서 제안하고 있는 비휘발성 메모리를 이용한 스토리지 모듈의 시스템 설계에 대해서 설명하며, 3 장에서는 2 장에서 제안한 스토리지 모듈 시스템의 동작 메커니즘을 기술하였다. 4 장에서는 구현한 스토리지 모듈을 토대로 고성능 멀티코어 서버 환경에서 적용한 순차 쓰기 실험 결과를 동기적 쓰기 실험 결과와 비교하였다. 끝으로 5 장에서는 결론 및 본 논문의 향후 연구 방향에 대해서 다루겠다.

[1] 비휘발성 메모리(NVRAM)은 데이터를 읽고 쓰는데 걸리는 시간이 일반적인 DRAM 과 속도가 비슷하면서 동시에 비휘발성 성질을 가지기 때문에 전원이 나가도 데이터를 써진 그대로 보존할 수 있다.

2. 시스템 설계

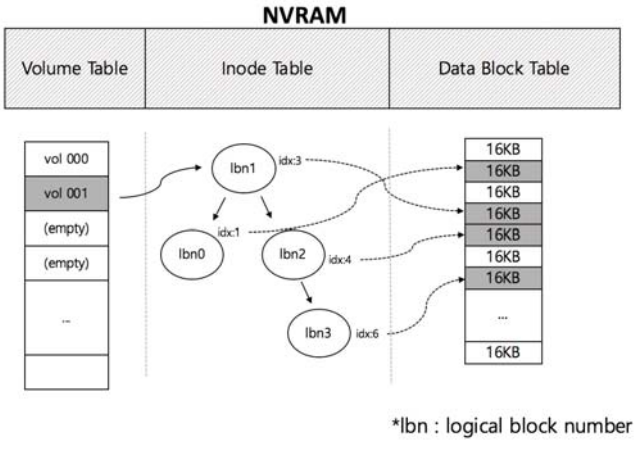
다음은 제안하는 스토리지 모듈에서 비휘발성 메모리를 활용하는 방법과 비휘발성 메모리에서 관리하는 자료구조에 대해 설명한다.

2.1 비휘발성 메모리를 이용한 스토리지 모듈

기존의 파일 시스템이 제공하는 동기적 쓰기 연산은 입출력 요청이 들어오면 파일을 저장장치에서 메인 메모리로 읽어오거나 새로 영역을 할당 받은 후, 메인 메모리 위에서 파일 데이터를 변경하고 변경된 데이터를 다시 저장장치에 쓰게 된다.

본 논문에서는 데이터를 저장장치까지 쓰는 과정에서 비휘발성 메모리를 이용하여 입출력 지연을 줄이고 그에 따른 성능 저하를 해소하였다. 비휘발성 메모리는 메인 메모리와 저장장치 사이에 존재하여 입출력 버퍼 역할을 하도록 구성하였다. 기존의 동기적 입출력 방식과 달리 데이터를 저장장치에 쓰기 전에 비휘발성 메모리까지 복사 과정을 거친다. 메인 메모리와 비휘발성 메모리 사이의 복사는 거의 DRAM 의 속도로 이뤄지며, 복사된 이후에는 전원의 공급여부와 무관하게 데이터의 접근이 가능하고, 회복 메커니즘에 의해 데이터의 지속성을 보장할 수 있다. 하지만 현재의 비휘발성 메모리는 메인 메모리보다 가격이 높으며 저장장치보다 사용 가능한 용량이 작으므로 효율적인 메모리 사용이 필요하다[4]. 따라서 비휘발성 메모리 버퍼에서 저장하고 있는 데이터를 재빨리 저장장치에 동기화 시켜주고 사용 공간을 회수해야 다음의 입출력 요청을 빠르게 처리할 수 있다. 해당 정책에 대한 설명은 3 장에서 다룬다.

2.2 비휘발성 메모리 버퍼 구조



(그림 1) 비휘발성 메모리 버퍼와 파일의 논리적 데이터 블록 관리

그림 1 은 제안하는 저장장치 모듈의 비휘발성 메모리에서 관리하는 세가지 데이터 - Volume Table, Inode Table, Data Block Table 에 대해 설명한다. 먼저, Volume Table 은 리눅스 커널의 파일 테이블과 같이 현재 입출력 요청중인 파일들의 목록을 파일 디스크립터로 관리한다. 파일 디스크립터는 런타임에 따라 달라지므로, Volume Table 엔트리에는 파일 디스크립터와 함께 파일 고유의 이름(경로)을 함께 기록한다. 하나의 Volume Table 엔트리는 파일을 관리하는 트리의 주소를 가지고 있어 이를 통해 파일 전체에 접근할 수 있다.

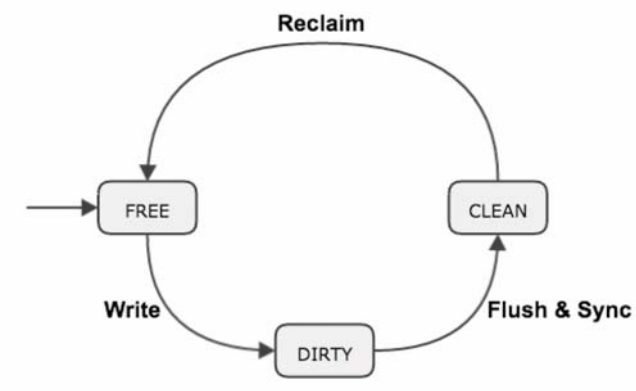
Inode Table 은 Data Block 의 색인 정보인 Inode 의 목록을 관리한다. Inode 는 Data Block 과 1:1 대응하며 각각의 Data Block 의 상태 및 위치 정보를 기록하고 있다. 예를 들어, 3 번째 Inode 는 3 번째 Data Block 에 대응한다(그림 1). 비휘발성 메모리 버퍼에서 동기적 입출력 요청의 한계 허용치는 비휘발성 메모리 버퍼의 Data Block 의 최대 개수로 제한된다.

Data Block Table 은 파일의 내용을 담고 있는 Data Block 을 관리한다. 하나의 파일은 저장장치의 쓰기 효율을 최대로 하기 위하여 저장장치의 블록 단위인 16KiB 로 나누어진다. 여기서 16KiB 는 저장장치의 쓰기 블록 단위를 의미하며 효과적인 저장장치 동기화를 위한 값으로 디바이스의 특성에 따라 달라질 수 있다. Data Block 은 파일의 오프셋을 블록 단위로 하는 Logical Block Number (LBN) 값을 부여받아 이를 키 값으로 하는 균형 트리에 저장된다. 파일의 크기에 비례하여 파일 트리의 크기도 증가하기 때문에 트리 구조는 저장공간이 작은 비휘발성 메모리에 저장하지 않고 메인 메모리에 저장하도록 설계되었다.

3. 구현

본 챕터에서는 제안하는 비휘발성 스토리지 모듈이 어떻게 동작하는지를 다룬다.

3.1 스토리지 모듈의 동작 메커니즘



(그림 2) 비휘발성 메모리 버퍼 내부 Data Block 의 상태 전이 도표

비휘발성 메모리에서 입출력 요청의 처리는 세 단계의 과정(Write, Flush & Sync, Reclaim)을 거치며, 각 단계에서의 Data Block 은 세 가지(Free, Dirty, Clean) 상태를 가진다(그림 2). 각 단계마다 별도의 요청 Queue 를 가지며 독립적인 작업 스레드에 의해 처리된다.

Write Stage

Write 단계에서는 가용 Data Block 을 가져와서 요청 데이터를 복사하고 이를 다음 스테이지의 Dirty Queue 에 입력한다. 데이터를 복사하는 과정에서 비휘발성 메모리의 Data Block 에 변경된 데이터가 저장되며, 이 때 Data Block 의 상태는 Free 또는 Clean 상태에서 Dirty 상태로 변화하며, 여러 개의 Dirty Queue 에 균등하게 배분된다. 사용할 Data Block 은 입출력 요청이 처리되기에 앞서, 요청에 해당하는 파일 트리를 확인하여 할당 받는다. 만약 파일 트리가 존재 하지 않는다면, 입출력 요청에 해당하는 크기의 Data Block 들을 Free Queue 로부터 할당 받고, Volume Tree 엔트리에 기록한다. 이미 존재하는 파일 트리에 대해서는 트리 검색을 통하여 해당 위치의 Data Block 을 재사용할 수 있다. 입출력 요청의 저장은 비휘발성 메모리를 사용하기 때문에, Write 단계의 완료만으로 데이터의 지속성을 보장할 수 있다. 하지만, 비휘발성 메모리의 공간 제약으로 인하여 요청된 Data Block 을 빠르게 저장장치에 동기화 시켜주고 재사용을 위해 회수해야 한다.

Flush & Sync Stage

Flush & Sync 단계는 비휘발성 메모리에서 업데이트 된 Dirty 상태의 Data Block 을 저장장치까지 동기화해 준다. 동기화를 마친 Data Block 의 상태는 Dirty 에서 Clean 상태로 변화한다. 이 과정은 운영체제의 비동기 입출력에 의존하는 Flush 부분과, 강제로 저장장치까지 동기화를 유발하는 Sync 부분으로 나뉜다. Write 단계에서 다중 Dirty Queue 로 배분된 Data Block 들은 Queue 별로 독립적인 Flush 스레드에 의해 처리된다. Flush 스레드는 다중 스레드로 구현되었으며, 각 스레드는 각각 할당된 Dirty Queue 로부터 Data Block 을 가져온 후 write 시스템 콜을 호출하여 저장장치에 기록한다. 현재 구현에서 사용한 저장장치는 16KiB 의 단위로 입출력을 처리하여 이보다 작은 크기의 입출력에 대해서는 성능저하가 발생한다. 이를 해결하기 위해 Flush 스레드는 Dirty Queue 가 일정 이상을 채워져야 작동하도록 Low Watermark 를 적용하여 작은 단위의 데이터 입출력 쓰기의 성능저하를 최소화하였다.

Write 시스템 콜에 의해 처리된 Data Block 은 Sync Queue 로 버퍼링된 후 Sync 스레드에 의해 저장장치까지 강제로 동기화 된다. Sync 스레드는 Flush 스레드와 달리 Free Queue 가 부족한 경우 활성화되며, sync 시스템 콜을 호출하고 빠르게 Dirty 상태의 Data Block 들을 Clean 상태로 변경시킨다. Sync 의 잦은 호출은 Flush 를 지연시키므로 Sync Queue 에 Data Block 이 충분히 채워지는 시점에 sync 시스템 콜을 호출해야 한다. Data Block 이 저장장치까지 동기화가 완료되면 Reclaim 스레드에 의해 다시 Free Queue 에 회수될 준비를 마치게 된다.

Reclaim Stage

Reclaim 단계에서는 동기화가 완료된 Data Block 을 회수하여 Write 단계에서 사용할 Data Block 을 준비해 둔다. 이때 Data Block 은 Clean 에서 Free 상태로 변한다. Reclaim 스레드는 Free 상태의 Data Block 이 부족하면 활성화되며, 전체 Volume Table 엔트리에서 각 트리를 탐색하여 대상이 되는 Clean 상태의 Data Block 을 찾아 회수한다. 이 때, Write 스레드가 해당 트리에 접근할 수 있기 때문에 Write 스레드와 Reclaim 스레드 사이에서의 스레드 동기화가 필요하다. 현재의 구현에서는 Read/Write Lock 으로 파일 트리를 보호하여 파일 트리 탐색의 안정성을 보장하였으며, 회수된 Data Block 에 대해서는 논리적 지우기와 가비지 컬렉션을 통해 파일 트리의 동적 변화를 최소화시켜 성능의 향상을 꾀하였다.

3.2 회복 메커니즘

제안하는 스토리지 모듈에서는 쓰기 요청을 비휘발성 메모리에 저장하는 것으로 데이터 지속성을 보장한다. 만약 시스템이 비정상적으로 종료되는 경우 쓰기 요청에 대한 데이터의 일부는 저장장치에 기록되지 않고, 비휘발성 메모리 상에만 존재할 수 있다. 따라서 시스템을 재시작하게 되면 회복과정을 통하여 아직 저장장치에 기록되지 않은 비휘발성 메모리 내부의 Data Block 에 대해서 쓰기 요청을 완료하여야 한다. 메인 메모리에 형성된 파일 트리가 관리하는 Data Block 에 대해서는 Reclaim 과정을 거치면서 비휘발성 메모리에 회수되는 시점부터 유효하지 않게 되므로 회복 과정에서 해당 쓰기 요청을 저장장치에 반영하지 않아도 되며, 비휘발성 메모리에 잔존하는 유효한 Data Block 에 대해서만 쓰기 요청을 완료해주면 된다. Dirty 상태의 Data Block 은 아직 write 나 sync 시스템 콜에 호출 대상이 되지 못하거나, 호출되었어도 저장장치까지 완전히 동기화가 이루어지지 못할 수 있다. 구현한 회복 메커니즘은 Dirty 상태인 Data Block 에 대하여 다시 동기화 시키고 Free 상태로 변경하고, 모든 Volume Table 엔트리를 초기화 및 파일의 모든 Data Block 의 상태를 Free 로 복원시켜 처음 시스템 상태로 되돌린다.

4. 실험 결과

본 논문에서 제안한 비휘발성 메모리를 이용한 스토리지 모듈을 토대로 실험을 하였다. 실험은 36 코어 2 소켓 CPU, 488 GiB DRAM 메인 메모리, 최대 쓰기 속도 4GiB/s 인 2.8 TB NVMe SSD 의 고성능 Linux 서버 환경에서 이루어졌으며, 실험은 파일을 생성하여 순차적으로 쓰기만 하는 연속 접근 쓰기를 진행하였다.

<표 1> 80 GiB 연속 접근 쓰기 실험 결과

Thread * File Size (GiB)	O_SYNC + 16KiB unit (sec)	NVRAM + 16KiB unit (sec)	NVRAM + 1KiB unit (sec)
1 * 80	730.663396	71.960662	81.187021
2 * 40	397.891570	33.659661	42.448661
4 * 20	221.099422	30.881170	40.837848
8 * 10	104.173094	29.625240	44.671247
16 * 5	77.308428	28.348618	44.845029
32 * 2.5	87.757776	29.853479	47.765922

실제 실험에서 사용된 비휘발성 메모리는 8GiB 공유메모리 상에서 진행하였다. 공유 메모리는 주 프로세스가 해제시킬 때까지 DRAM 상에 존재하면서 여러 프로세스가 접근을 할 수 있기 때문에 본 논문이 제안하는 비휘발성 메모리 버퍼 모듈을 모델링 할 수 있다. 각 스레드가 쓰는 파일의 크기는 80GiB 를 스레드 숫자로 나눈 값이다. <표 1> 에서 대조군인 Write + O_SYNC 결과는 파일을 O_SYNC 옵션을 주고 열어서 16KiB 씩 순차적으로 파일을 쓴 경우의 실험 결과이다. 한 스레드가 파일을 쓰는 크기가 클수록 많은 시간이 소요됨을 확인할 수 있으며, 최대 4GiB/s 의 쓰기 속도를 가진 고성능 SSD 저장장치를 이용해도 최대 1GiB/s 정도의 쓰기 성능을 보인다. 반면 비휘발성 메모리를 이용한 스토리지 모듈의 결과를 보면 동기화가 완료되는 시간을 앞당겨 성능을 크게 개선할 수 있었다. 한 스레드가 큰 파일을 쓰는 경우 10 배 가까이 쓰기 성능이 향상됨을 볼 수 있었으며, 16 스레드의 경우 약 2.6 배의 쓰기 이용률 향상을 낼 수 있었다. 또한 1KiB 단위의 작은 크기 입출력 연산에 대하여도 기본 16KiB 단위에 비해 성능에 차이가 거의 없이 비슷한 속도로 연산을 수행함을 볼 수 있다. 여러 스레드가 동시에 입출력 요청을 할 때 32 스레드의 경우 다중 Flush 스레드와의 경합으로 성능이 약간 저하되는 부분이 있지만 대체적으로 비례하여 성능이 증가하고 있다.

5. 결론 및 향후 연구

본 논문에서 제안하는 비휘발성 메모리를 이용한 스토리지 모듈을 통해 고성능 멀티 코어 서버 환경에서 동시다발적으로 들어오는 입출력 요청에 대하여 빠르고 지속성있는 동기적 쓰기가 가능함을 볼 수 있었다. 이는 슈퍼 컴퓨팅이나 대규모 데이터베이스 서버에서 동기적 입출력에 따른 지연으로 인한 성능 저하 문제를 해소해 줄 수 있다. 본 논문은 순차적 접근 쓰기의 환경을 구현해내었으며 앞으로 임의 접근 쓰기에 대해서 연구를 진행할 것이다. 또한 효율적인 메모리 관리를 위한 메커니즘 구축과 좀 더 병행성을 높인 자료구조를 도입하여 스레드 간 경쟁 상태를 없애고 확장성과 성능을 비약적으로 높인 스토리지 모듈을 구현하는 것이 향후 연구의 목표이다.

[Acknowledgement]

본 연구는 미래창조과학부 및 정보통신기술진흥센터의 SW 중심대학지원사업(IITP-2016-R7116-16-1027)의 연구결과로 수행되었음

[참고문헌]

[1] Bhattacharya, Suparna, et al. "Asynchronous I/O support in Linux 2.5." *Proceedings of the Linux Symposium*. 2003.

[2] 이상원. "NVRAM 과 데이터베이스 지속성." *정보과학회지* 33.2 (2015): 80-88.

[3] Kang, Sooyong, et al. "Performance trade-offs in using nvram write buffer for flash memory-based storage devices." *IEEE Transactions on Computers* 58.6 (2009): 744-758.

[4] Hyun, Choul-Seung, et al. "Design and Implementation of a File System that Considers the space efficiency of NVRAM." *Journal of KIISE: Computer Systems and Theory* 33.9 (2006): 615-625.

[5] Bailey, Katelin, et al. "Operating System Implications of Fast, Cheap, Non-Volatile Memory." *HotOS*. Vol. 13. 2011.