스토리지 내부에서의 처리를 통한 Trim 최적화 기법

유태관[0], 한규화, 신동군
성균관대학교 전자전기컴퓨터공학과
tk1star2@skku.edu, hgh6877@skku.edu, dongkun@skku.edu

Optimizing Trim Operations by In-Storage Processing

Taekoan Yoo[0], Kyuhwa Han, Dongkun Shin
Department of Electronical and Computer Engineering, Sungkynkwan University

요 약

플래시메모리 기반의 저장장치인 SSD(Solid State Drive)에서는 HDD(Hard Disk Drive) 기반으로 설계된 블록 계층을 그대로 사용하기 위하여, FTL(Flash Translation Layer)을 통해 논리 주소 공간과 물리 페이지 사이의 매핑을 관리한다. 플래시메모리의 덮어쓰기가 불가능한 물리적 특성으로 인해, FTL에서는 GC(Garbage Collection)를 통해 유효한 페이지를 다른 블록으로 옮기는 작업이 필요하다. Linux 파일시스템은 SSD의 효율적인 GC 작업을 위해 파일 삭제 시 trim 명령을 사용하여, 블록이 유효하지 않은 상태가 되었음을 SSD에게 전달한다. 본 연구에서는 trim의 오버헤드를 줄이기 위해, 여러 trim을 하나의 trim으로 모아서 처리하는 *vTrim*(vectored Trim)과, file의 inode 번호만을 보내고 SSD가 파일시스템의 정보를 참조해 trim 동작을 수행하는 *fTrim*(file Trim)을 제안한다. *vTrim*과 *fTrim*은 OpenSSD를 통해 구현하였으며, EXT4 파일시스템에서 파편화된 10MB 파일의 삭제 시간을 기존 trim 대비 *vTrim*과 *fTrim*에서 각각 63%, 42%의 처리시간이 감소하는 것을 확인했다. 또한, 삭제 요청과 읽기, 쓰기 명령을 함께 수행할 경우, *vTrim*과 *fTrim*에서 전체 수행시간이 EXT4 대비 36%, 29% 감소하는 것을 확인했다.

1. 서 론

최근 SSD(Solid State Drive)는 데이터센터, 스마트 폰, 그리고 개인용 PC에 이르기까지 스토리지 장치로 많이 활용되고 있다. SSD는 플래시메모리로 구성되어 있는데, 플래시 메모리는 기존의 저장장치인 HDD(Hard Disk Drive)와 다른 물리적 특성을 가지고 있다. 먼저, 플래시메모리는 read/program 명령의 기본 단위(4KB / 8KB - 페이지)와 erase 명령의 기본 단위(64pages / 128pages - 블록)가 서로 다르다. 또한, 한번 program 된 페이지는 erase되기 전까지 덮어쓰기가 불가능한 erase-before-write 특징을 가진다.

SSD는 HDD기반의 기존 블록 계층을 그대로 사용하기 위해 내부적으로 FTL(Flash Translation Layer)을 두어 논리 주소와 플래시메모리 페이지 사이의 매핑을 담당한다. FTL은 플래시메모리의 erase-before-write 특징으로 인해, 블록 내의 유효한 데이터를 다른 블록으로 옮기고 해당 블록을 erase하는 동작인 GC(Garbage Collection)를 수행한다. 기존 Linux 파일시스템은 파일을 삭제하는 경우, 파일시스템의 메타데이터 수정만을 진행하기 때문에, SSD 내의 FTL에서는 해당 파일과 연관된 페이지가 유효하지 않은 데이터인지 확인할 수 없다. 이를 해결하기 위해서 Linux에서는 유효하지 않은 블록의 정보를 trim[1]이라는 블록 인터페이스를 통해 SSD에게 전달한다. Trim 명령은 시작 섹터번호, 섹터 개수로 구성되며, FTL은 해당 명령을 전달 받은 경우, 내부의 매핑 테이블에 이를 기록하여 다음에 발생하는 GC시 해당 데이터를 옮기지 않도록 한다.

한편, Linux 고유의 파일시스템인 EXT4에서는 블록(4KB) 단위로 파일시스템을 관리하는데, 연속한 여러 개의 블록에 대한 매핑 정보를 <시작 논리 블록 주소, 연속한 논리 블록의 개수>형태의 익스텐트라는 구조체를 사용하여 관리한다. 또한, 파일의 데이터 페이지를 트리 구조로 관리하며 말단 노드를 익스텐트 블록, 나머지 노드를 인덱스 블록이라 한다. 익스텐트 블록(4KB) 하나에는 익스텐트들이 340개 존재한다. 인덱스 블록은 <키, 익스텐트 블록의 주소>로 이루어진 매핑 정보로 구성된다. 만약, EXT4에서 파일이 삭제되면, EXT4는 모든 익스텐트 블록을 확인하면서, trim 명령을 통해 해당 영역이 삭제되었음을 SSD에게 전달한다.

judicious trim[2]에서는 trim 명령을 시스템 이용률과 작업량에 따라 선택적으로 켜거나 끔으로써, trim이 읽기/쓰기 명령을 방해하지 않도록 하는 기법을 제안하였다. 하지만 trim을 끄는 경우, 이후에 발생하는 SSD 내의 GC 작업에서 유효하지 않은 페이지에 대한 옮기기 작업이 발생한다. 본 연구는 trim을 켜거나 끄는 것이 아닌, trim 자체의 오버헤드를 줄이기 위해서 vTrim (Vectored Trim)과 fTrim (File-based Trim) 기법을 제안한다. vTrim과 fTrim을 사용하면, 기존 trim 대비 파편화된 파일 삭제 시간이 각각 63%, 42% 감소한다.

본 논문은 다음과 같이 구성된다. 2절에서는 vTrim과 fTrim 기법의 구조와 알고리즘을 설명한다. 3절에서는 본 기법들을 OpenSSD[3]와 Linux 시스템에 구현에 대

해 설명한다. 4절에서는 vTrim과 fTrim 기법에서의 파일 삭제 시간을 측정한 결과를 보인다. 마지막 절인 5절에서는 결론을 제시한다.

2. vTrim과 fTrim

기존 trim 인터페이스는 <시작 섹터주소, 섹터 개수>라는 논리적으로 연속한 하나의 영역에 대한 정보를 SSD에게 전달한다. 이로 인해, 파일이 파편화된 상황 혹은 파일의 크기가 커서 여러 익스텐트에 할당되었다면, 파일 삭제 시 여러 개의 trim 명령을 SSD에게 전달해야 한다. 본 기법에서는 호스트와 SSD 사이의 인터페이스를 줄이기 위하여, 여러 개의 trim을 하나의 I/O 명령으로 전달하는 vTrim과 삭제한 파일의 inode 정보만을 전달하고 SSD에서 trim 동작을 처리하는 fTrim 기법을 제안한다.

2.1 Vectored Trim

EXT4 파일시스템은 파일을 지우기 위해 다음과 같은 작업을 수행한다. 먼저, 파일의 이름을 기반으로 inode 구조체를 호스트의 DRAM으로 읽어오는 작업을 수행한다. 다음으로 inode의 매핑을 참조하여 지우려는 영역에 해당되는 모든 익스텐트들의 할당을 해지하는 작업을 수행한다. vTrim에서는 각 익스텐트 단위로 trim을 수행하지 않고, 해지하고자 하는 모든 논리 주소를 별도로 할당한 4KB영역에 모은 후, 하나의 vTrim 명령을 사용하여 SSD에게 전달한다.

vTrim은 여러 개의 trim의 묶음으로 구성되어 있으며, 가장 앞의 4bytes는 해당 명령에 포함된 trim의 개수를 저장한다. SSD는 해당 쓰기 명령의 논리 주소를 보고 해당 명령이 일반 쓰기 명령인지 vTrim 명령인지 구분할 수 있다. SSD는 vTrim 명령을 받은 경우, 쓰기 명령으로 전달된 데이터를 읽어 내부의 trim 정보들을 추출한다. 추출한 trim 정보를 기반으로 SSD 내의 L2P (Logical to Physical) 매핑 테이블에 이를 반영하여 해당 페이지들을 유효하지 않은 상태로 만든다.

2.2 File-based Trim

EXT4는 fTrim 명령을 통해 지우고자 하는 파일의 inode 번호만을 FTL에게 전달한다. 그림1은 SSD가 inode 번호를 기반으로 inode의 매핑 정보를 얻는 과정을 보여준다. 먼저, 그림1_①과 같이 SSD는 파일시스템 영역의 가장 처음 페이지에 저장된 슈퍼블럭과 두번째 페이지에 저장된 그룹 디스크립터 테이블을 SSD 내의 DRAM으로 읽어 들인다. 다음으로 그림1_②와 같이 Inode 번호를 블록 그룹의 최대 inode 개수로 나누어 해당 inode가 속한 블록 그룹을 찾고, 그룹 디스크립터 테이블을 확인하여, 해당 블록 그룹의 inode 테이블의 위치를 얻는다. 마지막으로 그림1_③과 같이 Inode 번호와 블록 그룹의 최대 inode 개수를 기반으로 블록그룹 내에서의 오프셋 정보를 계산해 inode 테이블에서 inode 구조체를 DRAM으로 읽어오는 작업을 수행한다.

FTL은 위의 과정을 통해 얻은 inode 구조체를 통해 매핑을 탐색하여 익스텐트 정보를 얻고, 이를 SSD 내부의 L2P 매핑 테이블에 반영하여 해당 페이지들을 유효하지 않은 상태로 만든다. fTrim은 기존의 EXT4에서 수행하던 작업을 대신 수행함으로써, vTrim보다 인터페이스 오버헤드를 줄일 수 있다.

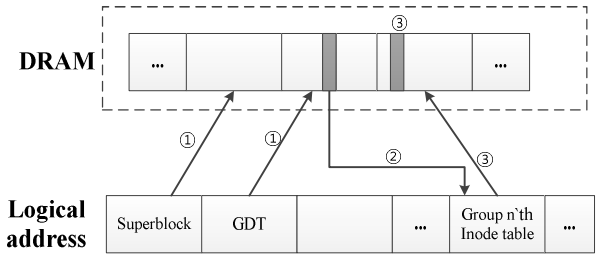

그림 1. SSD 내의 fTrim 처리 과정

2.3. 지연시간 감소를 위한 background 기법

vTrim과 fTrim은 기존의 trim과 달리, 여러 개의 Trim을 모아서 SSD가 한번에 처리하는 기법이다. vTrim은 전달된 모든 익스텐트들과 연관된 L2P 매핑 테이블을 수정하는 작업이 필요하며, fTrim은 inode 구조체를 찾은 작업이 추가로 필요하다. 이로 인해 vTrim/fTrim의 처리 시간은 기존의 Trim보다 많은 시간이 걸리고, Trim을 처리하는 동안 호스트의 다른 읽기/쓰기 명령을 처리하지 못한다. 본 연구에서는 이 문제를 해결하기 위해 백그라운드 처리기법과 preemption 처리기법을 제안한다.

vTrim과 fTrim 명령을 FTL이 받은 경우, trim을 처리시 필요한 정보를 SSD 내의 DRAM에 기록한 후, 다른 I/O 명령을 처리한다. 해당 정보는 vTrim의 경우에는 <시작 논리 블록 주소, 블록 개수>의 묶음이며, fTrim의 경우에는 inode의 매핑 정보이다. 이 후, SSD는 I/O 명령이 없는 시점에 DRAM에 기록한 정보를 참조하여 trim 동작을 수행한다. 본 연구에서는 해당 기법을 백그라운드 Trim 처리 기법이라 한다.

하지만, 백그라운드 작업을 처리 중 새로운 I/O 명령이 들어오는 경우, 해당 I/O는 trim 동작이 모두 처리된 이후에 수행할 수 있다. vTrim의 경우, 백그라운드 동작은 DRAM 상의 L2P 매핑 테이블 수정 작업이지만, fTrim의 경우에는 여러 번의 플래시메모리 접근을 필요로 하기 때문에 지연시간이 매우 길어질 수 있다. 이를 해결하기 위하여, 백그라운드 trim 처리 중에 호스트의 I/O 명령이 들어오는 경우, 백그라운드 trim 동작을 잠시 멈추고, 호스트 I/O를 먼저 처리하는 기법인 preemption trim 처리 기법을 제안한다. 이를 통해 호스트의 읽기/쓰기 명령은 trim 처리로 인한 오버헤드 없이 처리가 가능해진다.

3. 구현

본 연구에서는 호스트와 SSD 사이의 추가적인 인터페이스인 vTrim과 fTrim 인터페이스를 제안하였다. 파일 시스템은 두 인터페이스를 SSD 논리 주소의 마지막 페이지에 대한 쓰기 명령으로 SSD에게 전달하며, SSD는 해당 페이지에 대한 쓰기를 vTrim 혹은 fTrim으로 인식하고, 논리 영역에 대한 trim 작업을 수행한다.

vTrim 기법과 fTrim 기법은 OpenSSD에서 펌웨어를 수정하였다. OpenSSD는 8KB 크기의 페이지를 가지는 낸드 플래시를 사용하며, 내부 FTL에서는 16KB크기의 슈퍼페이지를 사용하여 L2P 매핑을 관리한다. EXT4의 기본 블록 크기인 4KB를 매핑의 단위로 사용하는 FTL을 구현하였고, 해당 FTL에서 vTrim과 fTrim을 처리할 수 있도록 수정하였다.

4. 실 험

4.1 실험 환경

본 연구에서는 *vTrim*과 *fTrim*의 성능을 평가하기 위해 인텔 i5-3580 쿼드코어 CPU와 8GB DRAM으로 구성된 호스트 PC를, 커널 버전 3.13.0의 우분투 14.04 OS를 수정하여 사용하였으며, 32GB 크기의 OpenSSD에 16GB EXT4를 구성하였다.

4.2 파일 삭제 시간

그림2는 160개의 익스텐트로 구성된 10MB 파일을 삭제하는데 걸리는 시간을 기존 trim을 사용한 경우에 대해 정규화한 결과를 보여준다. I2C(Insert to Completion)는 trim 명령을 SSD로 전달하고 SSD에서 완료 인터럽트를 받기까지의 시간을 의미하는데, 이는 blktrace[4]를 통해 얻은 블록 단위의 I/O 트레이스를 기반으로 계산한다. 그리고, SSD 내에서의 처리 시간은 trim 명령을 SSD내의 DRAM에 저장하는 I/O 수행 시간과 백그라운드에서 L2P 매핑 테이블을 수정하는 시간으로 구분된다. 각 단계에서 소요된 시간은 SSD 내의 펌웨어를 수정하여 측정하였다.

Trim 기법에서는 호스트와 SSD 사이의 명령어 전달에 전체 처리 시간의 43%를 소모하였으며, SSD 내부에서 trim을 처리하는데 57%의 시간을 소모하였다. 반면, *vTrim*과 *fTrim* 기법에서는 하나의 trim 명령을 통해 모든 정보를 전달하여 인터페이스 시간이 감소하였다. *fTrim* 기법은 전달받은 inode 번호에서 inode 구조체를 읽어와 해당 inode와 연관된 익스텐트 정보를 얻는 동작으로 인해 *vTrim*보다 긴 처리시간을 가진다. *vTrim* 기법과 *fTrim* 기법은 trim 기법 대비 각각 63%, 42%의 처리 시간을 단축하였다.

4.3 Mixed Workload Performance

그림3은 160개의 익스텐트로 구성된 10MB 파일을 삭제한 후, 여러 개의 읽기 명령을 수행하는데 걸린 시간을 trim의 수행 시간에 정규화한 결과이다. *vTrim* 기법과 *fTrim* 기법은 L2P 매핑을 수정하는 작업을 백그라운드에서 수행하는 기법이다. *fTrim_pre* 기법은 preemption trim 처리 기법을 추가한 *fTrim* 기법이다. *vTrim*과 *fTrim* 모두 trim의 처리 시간이 감소하였기 때문에, 기존 *trim* 기법 대비 전체 수행시간이 감소한다.

vTrim 기법은 160개의 trim을 하나의 *vTrim*으로 SSD에게 전달하기 때문에, *trim* 기법 보다 수행시간이 20번, 1000번의 읽기를 수행할 때 각각 36%, 12% 감소한다. 1000번의 읽기를 수행한 경우, 전체 수행시간에서 *trim* 동작이 차지하는 시간이 적어 성능 개선이 20번 read를 수행할 때보다 적다.

반면, *fTrim*의 경우, 20번, 100번 읽기를 수행할 때

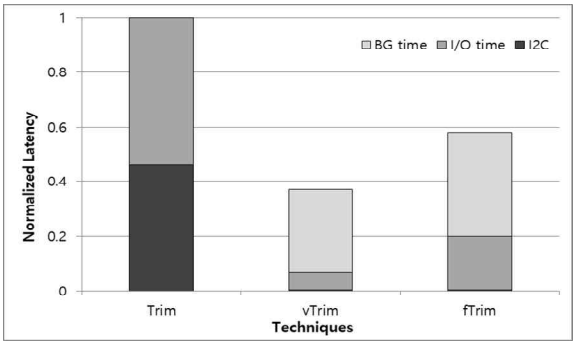

그림2. 파편화된 파일의 삭제 시간

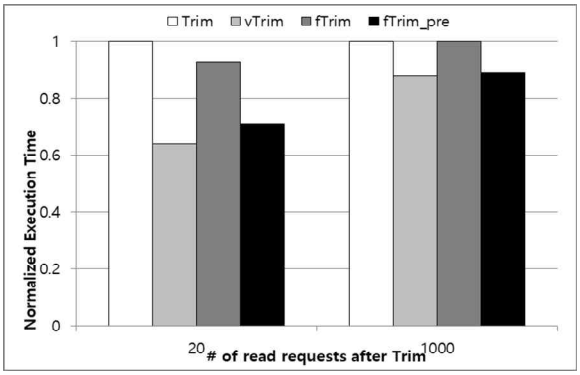

그림3. 파일 삭제 및 읽기 수행 시간

전체 수행시간이 각각 7%, 0.1% 감소한다. *fTrim*에서 성능 개선이 적은 이유는 SSD가 inode 번호를 기반으로 inode 구조체를 얻어 익스텐트 정보를 읽는 작업을 한번에 수행하기 때문이다. 해당 작업은 플래시메모리에 여러 번의 읽기가 필요하며, 이로 인해 호스트의 읽기/쓰기 명령의 처리가 지연된다. *fTrim_pre*는 *fTrim*의 백그라운드 작업을 처리 중, 읽기/쓰기 명령이 전달되었을 때 이를 먼저 처리하기 때문에, trim 이후 20번의 읽기를 수행하는 경우에 전체 수행시간을 trim 기법 대비 29% 줄일 수 있다.

5. 결론 및 향후 연구 계획

본 연구에서는 기존 trim을 개선한 vTrim과 fTrim을 제안한다. 본 기법은 trim의 인터페이스 오버헤드를 줄일 수 있으며, 해당 작업을 SSD의 백그라운드에서 수행하여 일반 읽기/쓰기 명령을 방해하지 않도록 하였다. 이를 통해 *trim* 명령의 처리 시간이 vTrim과 fTrim에서 각각 최대 63%, 42% 감소하는 것을 확인하였다. 향후 연구에서는 여러 개의 파일이 동시에 삭제되면서 사용자의 I/O가 많은 상황에서 trim 동작을 최적화 하는 연구를 진행할 계획이다.

참고문헌

[1] F. Shu. Notification of Deleted Data Proposal for ATA-ACS2. http://t13.org/, 2007.

[2] C. Hyun, J. Choi, D. Lee, and S. H. Noh, "To TRIM or not to TRIM:Judicious triming for solid state drives," *Poster presentation in the 23rd ACM Symposium on Operating Systems Principles*, 2011.

[3] OpenSSD project http://www.openssd-project.org

[4] A. D. Brunelle. "Block I/O Layer Tracing: blktrace.," 2006.