F2FS의 fsync latency 개선을 위한 file node structure 관리 최적화 기법

곽현호 O, 정지윤, 신동군
성균관대학교
gusghrhkr@skku.edu, wjdwldbs1@skku.edu, dongkun@skku.edu

Improving fsync latency of F2FS by optimizing file node structure

Hyunho Gwak O, Jeeyoon Jung, Dongkun Shin
Sungkyunkwan University

요 약

LFS의 out-of-place update scheme 특성은 사용자의 쓰기 요청을 저장장치에 항상 순차적으로 기록한다. 이러한 LFS의 특징은 플래시 저장장에 유리하며, 따라서 galaxy s10같은 최신 모바일 장치부터 ZNS까지 LFS가 사용되고 있다. F2FS는 플래시 메모리의 특징에 맞추어 설계된 LFS로써, 파일 시스템 성능 개선을 위한 다양한 기능을 포함하고 있다. 그런데, F2FS의 fsync 동작에서 수행하는 node 탐색 동작은 fsync latency를 크게 지연시키는 문제가 있다. Fsync latency가 길어질수록 사용자의 쓰기 요청이 처리되지 못하고 지연되는 시간도 길어지므로 파일 시스템 성능이 크게 감소된다. 우리는 F2FS의 fsync latency를 감소시키기 위하여 F2FS의 fsync 동작을 개선하였다. Fsync 수행 시 node 탐색 동작이 필요하지 않도록 변경된 node를 list structure로 관리하는 방법을 제안한다. 실험 결과, fsync latency를 33%까지 감소시켰다.

1. 서 론

NAND 플래시 메모리는 모바일 장치부터 서버 시스템에 이르기까지 다양한 시스템에서 사용되고 있다. 플래시 메모리는 높은 임의 접근 성능, 저전력, 그리고 높은 내구성과 같은 장점들을 가지고 있으며, 따라서 기존 주 저장장치로 사용되던 HDD를 대체하고 주 저장장치로 널리 사용되고 있다. 그런데, 플래시 메모리는 데이터를 기록한 블록을 덮어쓸 수 없고, 새로운 데이터를 기록하기 위해서는 블록을 erase해야 하는 제약을 가지고 있다. 따라서, SSD와 같은 플래시 메모리 장치는 flash translation layer (FTL)과 같은 임베디드 펌웨어를 사용하여 사용자의 쓰기 요청을 SSD 내부에서 재정렬하고 플래시 메모리에 순차적으로 기록한다.

Log-structured file system (LFS) [1]는 플래시 메모리에 사용하기에 적합한 파일 시스템이다. LFS는 out-of-place update scheme을 사용하며, 다른 파일 시스템들과는 달리 데이터가 업데이트 되었을 때 이전에 기록된 블록은 무효화하며 새로운 블록을 할당 받고 데이터를 기록한다. 따라서, LFS는 저장장치에 순차 쓰기만을 수행할 수 있으며, 이러한 특징으로 인하여 최신 모바일 장치 또는 ZNS SSD에서도 LFS가 채택되어 사용되고 있다 [2, 3].

Flash-friendly file system (F2FS) [4]는 플래시 메모리의 특성에 맞추어 설계된 LFS로써, 대부분의 LFS 연구에서 사용될 뿐만 아니라 linux kernel에서 활발하게 maintain되고 있다. F2FS는 파일 시스템 성능 개선을 위한 다양한 최적화 기법들을 포함하고 있다. F2FS의 메타데이터 관리 정책은 그 중 하나이다. F2FS는 LFS에서 자주 변경되는 메타데이터로 인한 쓰기 양 증가 문제를 완화하기 위하여, 파일 시스템의 메타데이터가 변경되더라도 저장장치에 바로 기록하는 대신 메모리에서 유지한다. 메모리에서 유지되는 메타데이터는 크게 두 가지 동작을 수행할 때 저장장치에 기록된다. 첫 번째는 파일 시스템의 체크포인트이다. 파일 시스템의 체크포인트는 현재 변경된 데이터 및 메타데이터 블록을 모두 저장장치에 기록함으로써 체크포인트 시점으로 복구할 수 있는 것을 지원하는 동작이다. 두 번째는 fsync이다. 사용자가 fsync를 요청할 경우 파일 시스템은 해당 파일의 변경된 블록들이 저장장치에 기록되는 것을 보장해야 한다. 그런데, 변경된 메타데이터를 기록하기 위하여 매 fsync 호출마다 체크포인트를 사용한다면 메타데이터 쓰기 양이 크게 증가된다. 따라서, F2FS는 fsync 수행 시 저장장치에 기록하는 블록 수를 최소화하기 위하여 해당 파일의 메타데이터만을 저장장치에 기록하도록 설계되었다.

하지만, 이러한 F2FS의 fsync 동작 최적화에도 불구하고 F2FS의 메타데이터 structure로 인하여 fsync latency가 증가하는 문제가 여전히 발생한다. F2FS는 메타데이터 중 데이터 블록의 블록 주소를 저장하는 node를 하나의 global structure로 관리한다. 체크포인트를 수행할 경우에는 모든 node를 저장장치에 기록해야 하기 때문에 global structure로 관리하는 것이 유리하다. 반면, fsync를 수행할 경우에는 fsync가 호출된 파일의 node만을 기록함에도 불구하고,

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

global structure에서 모든 node를 확인하고 각 node가 속한 파일의 정보를 검사해야 한다. 이와 같이 fsync 동작은 global structure를 사용하기에 적합하지 않으며, 불필요한 node 탐색 동작은 fsync latency를 지연시키고 파일 시스템 성능을 감소시키는 문제가 있다.

우리는 F2FS의 fsync 동작을 최적화하기 위하여 fsync 동작을 단계별로 분석하고 불필요한 동작을 최소화하였다. Fsync 분석 결과, F2FS의 fsync 단계 중 node를 기록하는 단계에서 큰 시간이 소모되는 것을 확인하였다. 또한, 이 시간은 node를 저장장치에 기록하는 시간이 아니라 파일 시스템에서 node를 탐색하기 위해 소모되는 시간인 것을 확인하였다. 본 논문에서는 이와 같은 node 탐색 시간을 최소화하고 fsync latency를 감소시킬 수 있는 새로운 메타데이터 관리 방식을 제안한다. 실험 결과, fsync latency를 최대 33%까지 감소시켰다.

2. 배경지식

F2FS는 다음과 같은 순서로 fsync를 수행한다. 첫 번째로, fsync가 호출되면 해당 파일의 변경된 데이터 블록을 저장장치에 기록한다. 메모리에서 해당 파일의 dirty 데이터 블록은 모두 저장장치에 기록된다. 그런데, LFS의 out-of-place update scheme에 따라 데이터 블록은 새로운 위치에 기록되므로, 데이터 블록을 기록하는 동작에서 데이터 블록의 주소를 저장하는 메타데이터인 node 블록이 변경된다. 두 번째 fsync 동작은 변경된 node 블록을 기록하는 것이다. Node 블록이 저장장치에 기록되어야 node 블록에 기록된 데이터 블록 주소를 기반으로 유효한 데이터 블록을 탐색하는 것이 가능하기 때문에 변경된 node 블록 또한 저장장치에 기록되어야 한다. Fsync의 마지막 동작은 flush command를 저장장치에 전달하는 것으로, flush command를 사용하여 변경된 데이터 및 node 블록이 저장장치에 완전하게 기록되는 것을 보장한다.

F2FS는 모든 node를 하나의 global structure에서 관리한다. F2FS는 별도의 *node inode*를 생성하고 새로운 node가 필요할 경우 node inode에 새로운 page를 추가한다. 이와 같은 메타데이터 관리 방식은 메모리 사용량을 최소화하는 장점이 있다. 또한, F2FS의 체크포인트 기반 메타데이터 기록 동작에서는 모든 dirty node를 저장장치에 기록하기 때문에 global structure로 node를 관리하는 것이 문제가 되지 않는다. 하지만, 이러한 구조는 fsync 수행 시 불리하게 동작한다. F2FS는 fsync의 두 번째 동작인 node 블록을 기록하는 동작에서 해당 파일의 dirty node만을 저장장치에 기록하도록 동작을 최적화하였음에도 불구하고, 파일별로 node를 관리하지 않기 때문이다. 따라서, F2FS는 모든 dirty node를 탐색해야 하며, 메모리에 존재하는 각 dirty node page를 읽고 해당 node page의 파일 정보를 확인한다. 파일 정보가 fsync가 호출된 파일의 정보와 동일할 경우, F2FS는 해당 node page를 저장장치에 기록한다..

3. Fsync 최적화

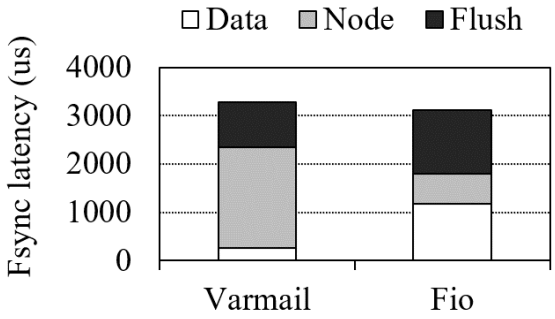

그림 1. Fsync의 단계별 소모시간

우리는 fsync latency를 지연시키는 원인을 확인하기 위하여 fsync 동작에서 단계별로 소모되는 시간을 분석하였다. 2장에서 설명한 것과 같이 fsync는 크게 세 가지 동작으로 구분될 수 있으며 우리는 각 동작에 소모되는 시간을 *Data*, *Node* 그리고 *Flush* 동작으로 구분하여 측정하였다. 그림 1은 두 가지 workload에서 fsync latency를 단계별로 분석한 것이다. 실험은 filebench [5]의 varmail workload와 fio benchmark [6]를 사용하였으며, 자세한 실험 환경은 4장에 서술하였다. *Data*는 fsync의 첫 번째 동작이며 변경된 데이터 블록을 기록하는 시간이다. *Node*는 fsync의 두 번째 동작이며 변경된 node 블록을 기록하는 시간이다. 마지막으로 *Flush*는 flush command를 처리하는 시간이다. 분석 결과, node를 기록하는 단계에서 많은 시간이 소모되는 것을 확인할 수 있다. Varmail workload에서는 동시에 많은 파일이 접근되기 때문에 메모리에 존재하는 node 개수가 많다. 따라서, Node 단계에서 소모되는 시간이 fsync latency의 63%를 차지한다. 반면, 실제로 기록되는 node의 수는 상대적으로 매우 작다.

표 1은 fsync에서 node 탐색에 소모되는 overhead를 확인하기 위하여 node 탐색 수 그리고 dirty node의 수를 측정한 것이다. N*ode search*는 fsync 수행 시 메모리에서 탐색하는 node page의 수 이다. *Node write ratio*는 탐색한 node 수 중에서 저장장치에 기록한 node 블록의 비율이다. 각 수치는 workload에서 발생한 총 누적 값을 fsync 수행 횟수로 나눈 값으로, 한 번의 fsync를 수행할 때 발생하는 평균 수치이다. 표 1에서 확인할 수 있는 것처럼, 탐색한 node 수에 비하여 fsync로 기록되는 node의 수는 매우 적으며, varmail workload의 node write ratio는 1%보다 낮았다.

그림 1에서 확인한 것과 같이 기존 fsync 동작에서 node 탐색을 위해 많은 시간이 소모된다. 또한, node 탐색 동작에서 fsync가 호출된 파일과 관계없는 다른 파일의 node를 탐색하기 위하여 대부분의 시간이 소모되는 것을 확인하였다. 이 문제를 해결하기 위하여, 우리는 dirty node를 관리하기 위한 별도의 *dirty node list* structure를 구현하였다.

표 1. Fsync의 평균 메타데이터 탐색 비용

	Node search	Node write ratio
Varmail	12100	0.01%
Fio	520	0.19%

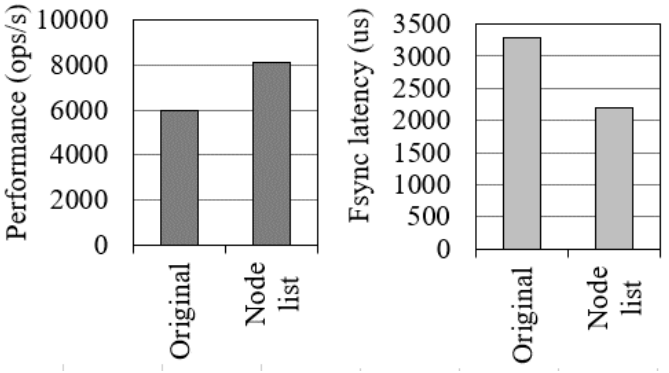

그림 2. Filebench-varmail workload에서 기법별 성능과 fsync latency

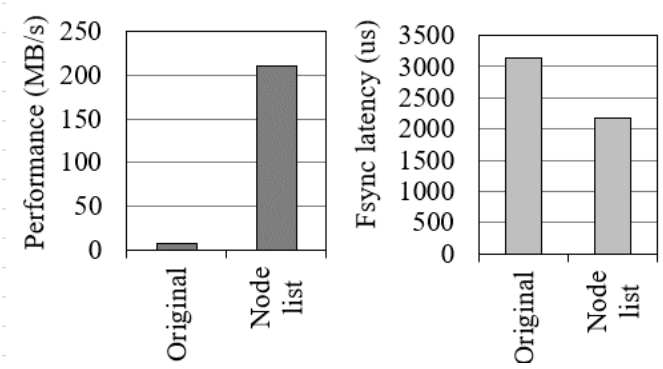

그림 3. Fio benchmark에서 기법별 성능과 fsync latency

Dirty node list에는 dirty node 블록의 ID가 list 구조로 저장된다. Dirty node list는 파일별로 관리되며, dirty node의 ID는 dirty node 블록의 메모리 주소를 나타낸다. Dirty node ID를 사용하여 메모리 공간에 저장된 dirty node 블록에 접근할 수 있으므로, 기존 F2FS의 dirty node 탐색 과정을 생략할 수 있다. 따라서, fsync가 호출된 파일의 dirty node를 탐색하기 위해서는 해당 파일의 dirty node list만 참조하면 된다.

기존 F2FS의 global structure는 dirty node list와 함께 사용된다. Global structure는 메모리에 존재하는 node page를 관리하기 위한 것으로 F2FS가 체크포인트를 수행하여 모든 dirty node를 기록할 때 활용된다. Dirty node list는 node page의 메모리 주소를 ID값으로 사용한다. 따라서, 기존 global structure는 그대로 유지하고 dirty node list를 참조하는 것만으로 fsync 수행 시 기록해야 할 dirty node page를 모두 탐색할 수 있다. 우리는 각 dirty node list의 entry 크기를 8 B로 구현하였으며, 이 크기는 하나의 node 블록 크기인 4 KB의 약 0.2%에 해당한다. 따라서, dirty node list를 유지하기 위한 추가적인 메모리 공간 소모는 매우 적다.

4. 실험 결과

4.1 실험 환경

우리는 kernel 4.15 버전의 F2FS를 수정하여 dirty node list를 구현하고 실험하였다. 실험은 3GHz 쿼드 코어, 8GB 메모리, 그리고 SAMSUNG 970 SSD를 사용한 환경에서 진행되었다. 실험에 사용된 workload는 filebench 그리고 fio benchmark 이다. Filebench에서는 varmail workload를 사용하였다. Fio는 write size는 4KB, thread 개수는 4, 파일 크기는 2GB, 그리고 32개의 write request마다 fsync를 호출하도록 benchmark를 설정하였다.

4.2 실험 결과

그림 2는 varmail workload에서 본 논문에서 사용한 기법을 적용하였을 때 성능 개선을 보여준다. *Original*은 기존 F2FS를 사용한 결과이며, *Node list*는 본 논문에서 제안하는 Dirty node list를 활용한 결과이다. 실험 결과, Original보다 Node list의 성능이 증가하였으며, fsync latency가 감소되었다. Varmail workload에서는 성능이 37% 증가하였으며 fsync latency는 33%가 감소하였다. 그림 3은 fio를 사용한 실험 결과이다. Varmail workload 실험과 마찬가지로 Original보다 Node list의 성능이 증가하였으며 fsync latency도 감소되었다. Fio에서는 성능이 170% 증가하였으며 fsync latency는 31%가 감소하였다. 이와 같이 dirty node list를 사용할 경우 불필요한 node 탐색을 제거할 수 있으며 fsync latency를 감소시키고 성능을 증가시킬 수 있다. Fio benchmark의 경우 write 크기가 작고 write 그리고 fsync만을 수행하도록 설정되었기 때문에, write 뿐만 아니라 read와 같은 다른 파일 동작을 같이 수행하는 varmail workload보다 성능 개선이 더 크게 측정되었다.

5. 결 론

본 논문에서는 F2FS의 fsync 동작을 단계별로 분석하였으며 fsync latency를 지연시키는 불필요한 node 탐색 문제를 확인하였다. 또한, node 탐색에 소모되는 시간을 최소화할 수 있는 dirty node list를 제안하였으며, 실험 결과 fsync latency를 최대 33% 감소시킬 수 있었다.

참고문헌

[1] M. Rosenblum and J. K. Ousterhout, “The design and implementation of a log-structured file system,” ACM Transactions on Computer Systems, vol. 10, no. 1, pp. 26-52, 1992.

[2] “Galaxy Note 10 uses F2FS.” https://www.sammobile.com/news/galaxy-note-10-uses-f2fs-not-ext4-file-system-whats-the-ifference.

[3] “F2FS ZNS support.” https://zonedstorage.io/linux/fs/.

[4] C. Lee, D. Sim, J. Hwang, and S. Cho, “F2FS: A new file system for flash storage,” in 13th USENIX Conference on File and Storage Technologies (FAST’15), pp. 273-286, 2015.

[5] Filebench, https://github.com/lebench/lebench/wiki.

[6] Fio benchmark, https://github.com/axboe/fio.