ISSN 2383-6318(Print) / ISSN 2383-6326(Online)
KIISE Transactions on Computing Practices, Vol. 25, No. 3, pp. 209-214, 2019. 3
https://doi.org/10.5626/KTCP.2019.25.3.209

파일 수준 저널링에서의 fsync 지연시간 개선을 위한 fsync 병합 기법 제안

(fsync Merging for Improving the fsync System Call Latency in File-Level Journaling)

김 솜 † 박 대 준 † 신 동 군 ††
(Somm Kim) (Daejun Park) (Dongkun Shin)

요 약 대부분의 시스템에서 데이터 내구성을 보장하기 위해 fsync 시스템 콜을 사용한다. fsync 시스템 콜은 현재 수행 중인 프로세스와 동기적으로 동작하기 때문에 fsync 시스템 콜의 지연시간을 개선하는 것은 시스템의 응답시간 측면에서 중요하다. 기존의 컴파운드 트랜잭션으로 인한 fsync 시스템 콜의 지연시간을 파일 수준 저널링을 사용하여 개선한 연구가 있으나 짧은 시기에 여러 코어에서 fsync 시스템 콜이 발생할 경우 컴파운드 트랜잭션에 비해서 플러시 명령 횟수가 증가한다. 따라서 본 논문에서는 파일 수준 저널링에서 플러시 명령 횟수 증가로 인한 fsync 지연시간 문제를 해결하기 위해 짧은 시기에 여러 코어에서 발생한 fsync 시스템 콜들을 한꺼번에 묶어서 처리하는 fsync 병합 기법을 제안한다.

키워드: 저널링, fsync 시스템 콜, 파일 수준 저널링, fsync 병합

Abstract Most of the systems uses the fsync system call to ensure data durability. Since the fsync system call operates synchronously with the currently executing process, it is important to improve the fsync latency in terms of the response time of the system. There have been studies on the improvement in the latency of the fsync system call due to the compound transaction by using file-level journaling. However, when the fsync system call occurs on multiple cores in a short period of time, the number of flush command increases compared to compound transactions. Therefore, in the present work, we propose a fsync merging technique that combines fsync system calls generated from multiple cores in a short period of time in order to solve the latency problem arising due to increase in flush command count in file-level journaling.

Keywords: journaling, fsync system call, file-level journaling, fsync merging

· 이 논문은 2019년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임 (No.IITP-2017-0-00914, 지능형 IoT 장치용 소프트웨어 프레임워크)
· 이 논문은 2018 한국컴퓨터종합학술대회에서 'Fine-Grained Journaling에서의 fsync 지연시간 개선을 위한 선택적 Grouped fsync 기법 제안'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : 성균관대학교 전자전기컴퓨터공학과
sommkim@skku.edu
pdaejun@skku.edu
†† 종신회원 : 성균관대학교 소프트웨어대학 교수
(Sungkyunkwan Univ.)
dongkun@skku.edu
(Corresponding author임)

논문접수 : 2018년 9월 28일
(Received 28 September 2018)
논문수정 : 2018년 11월 28일
(Revised 28 November 2018)
심사완료 : 2018년 12월 13일
(Accepted 13 December 2018)

Copyright©2019 한국정보과학회 : 개인 목적이나 교육 목적인 경우, 이 저작물의 전체 또는 일부에 대한 복사본 혹은 디지털 사본의 제작을 허가합니다. 이 때, 사본은 상업적 수단으로 사용할 수 없으며 첫 페이지에 본 문구와 출처를 반드시 명시해야 합니다. 이 외의 목적으로 복제, 배포, 출판, 전송 등 모든 유형의 사용행위를 하는 경우에 대하여는 사전에 허가를 얻고 비용을 지불해야 합니다.
정보과학회 컴퓨팅의 실제 논문지 제25권 제3호(2019. 3)

1. 서 론

대부분의 파일 시스템은 시스템에 장애가 발생했을 때 데이터 일관성을 보장하기 위해 저널링을 사용한다. 저널링이란 파일 시스템의 변경사항을 실제 스토리지 영역에 기록하기 전에 먼저 저널 영역에 기록하여 시스템에 장애가 발생했을 때 저널 영역에 기록된 변경사항을 읽어 실제 스토리지에 반영하는 과정을 통해 데이터 일관성을 보장하는 방법이다.

리눅스 커널의 기본 파일 시스템인 ext4에서는 일정 주기 동안 발생하는 파일 시스템의 변경사항을 한꺼번에 묶어서 커밋하는 컴파운드 트랜잭션 방식으로 저널링을 수행한다. 하지만 fsync 시스템 콜을 컴파운드 트랜잭션 방식으로 커밋하면 해당 파일에 대해서는 즉시 내구성을 보장할 수 있지만, fsync 시스템 콜이 호출된 파일에 대한 변경사항뿐만 아니라 다른 파일에 대한 변경사항도 함께 커밋되므로 fsync 시스템 콜의 지연시간이 길어진다. 기존의 선행 연구인 iJournaling [1,2] 기법에서는 fsync 시스템 콜이 호출된 파일의 변경사항만을 커밋하는 파일 수준 저널링을 수행하여 컴파운드 트랜잭션으로 인해 발생하는 fsync 시스템 콜의 지연 문제를 해결하였고, 코어 별로 사용하는 저널 영역을 분리하여 여러 코어에서 발생한 fsync 시스템 콜이 동시에 수행될 수 있게 하였다. 그러나 짧은 시기에 여러 코어에서 발생한 fsync 시스템 콜은 서로 개별적으로 처리되므로 컴파운드 트랜잭션에 비해 플러시 명령 횟수가 증가한다. 플러시 명령의 수행시간은 그림 1에서 볼 수 있듯이 fsync 시스템 콜 지연시간의 90% 이상을 차지하기 때문에 플러시 명령 횟수의 감소는 fsync 시스템 콜의 지연시간을 감소시킬 수 있다.

따라서 본 논문에서는 파일 수준 저널링을 사용하여 fsync 시스템 콜이 호출된 파일에 대해서만 저널링을 수행하면서, 짧은 시기에 여러 코어에서 발생한 fsync 시스템 콜을 한꺼번에 묶어서 처리하여 플러시 명령 횟수를 줄이는 fsync 병합 기법을 제안한다.

2. 관련 연구

iJournaling [1,2] 기법은 컴파운드 트랜잭션으로 인한 fsync 시스템 콜의 지연 문제를 파일 수준 저널링을 사용하여 해결하였다. 또한 fsync 시스템 콜이 호출된 파일에 해당하는 데이터와 메타데이터만을 기록하고, 여러 파일과 관련되어 있는 블록 비트맵과 아이노드 비트맵, GDT(Global Descriptor Table)와 같은 메타데이터는 기록하지 않고, 하나의 파일에만 속하는 메타데이터인 아이노드와 익스텐트 블록(extent block)만을 저널 영역에 기록한다. 하지만 iJournaling 기법의 경우 짧은 시기에 여러 코어에서 fsync 시스템 콜이 발생했을 때 fsync 시스템 콜 하나 당 헤더 블록과 익스텐트 블록에 대한 플러시 명령과 커밋 블록에 대한 플러시 명령, 총 2번의 플러시 명령이 발생하게 되므로 컴파운드 트랜잭션 기법에 비해 플러시 명령 횟수가 증가하는 문제가 있다.

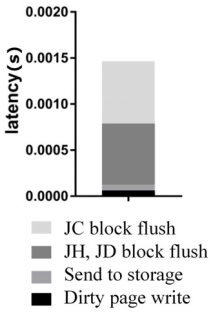


그림 1 fsync 지연시간
Fig. 1 fsync Latency

BarrierFS [3]는 IO 스택에서 스토리지 순서를 지키기 위해 사용하는 플러시 명령의 오버헤드를 제거하기 위해 배리어 커맨드를 추가한 기법이다. 배리어 커맨드를 추가함으로써 데이터, 헤더 블록, 익스텐트 블록과 커밋 블록 간의 스토리지 순서는 보장되므로 커밋 블록에 대한 플러시 명령만 사용한다. BarrierFS 또한 짧은 시기에 여러 코어에서 fsync 시스템 콜이 발생했을 때 fsync 시스템 콜 당 1번의 플러시 명령이 발생하므로 본 논문에서 제안하는 기법보다 플러시 명령 횟수가 많이 발생하는 상황이 존재한다.

3. 제안하는 기법

3.1 fsync 병합 기법

본 논문에서 제안하는 fsync 병합 기법은 짧은 시기에 여러 코어에서 발생한 fsync 시스템 콜을 묶어서 처리하여 플러시 명령 횟수를 감소시킨다. 또한 기존의 컴파운드 트랜잭션은 fsync 시스템 콜이 호출하지 않은 파일들의 변경사항도 한꺼번에 저널링하지만 fsync 병합 기법은 fsync 시스템 콜이 호출된 파일들의 변경사항만을 한꺼번에 묶어서 저널링한다. fsync 병합 기법에서 하나의 저널로 합쳐진 fsync 시스템 콜들에 해당하는 파일의 아이노드에 대한 정보는 헤더 블록에 추가되어 기록된다. 따라서 fsync 병합 기법의 저널 구조는 그림 2와 같다. 헤더 블록에는 저널 헤더, 하나의 저널로 합쳐진 fsync 시스템 콜 개수만큼의 아이노드 번호와 아이노드 구조체, 그리고 익스텐트 블록을 가리키는 블록 태그가 포함된다. iJournaling 기법의 저널 구조와 다른 점은 fsync 시스템 콜에 해당하는 아이노드 번호와 아이노드 구조체가 코어 개수만큼 존재한다는 점이다.

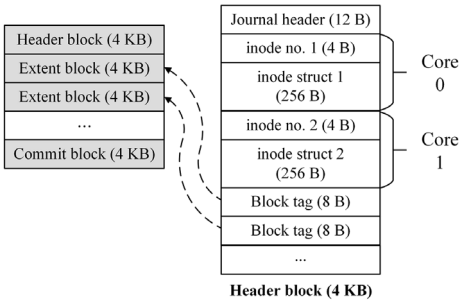
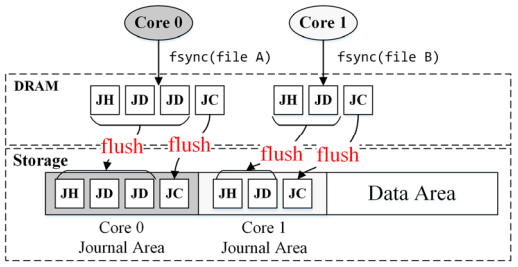


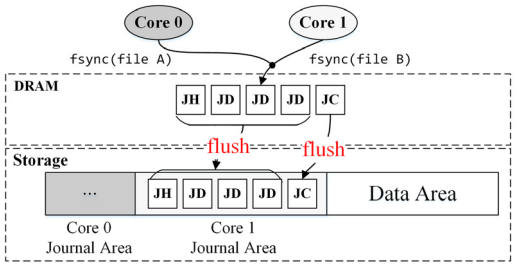
그림 2 fsync 병합 저널 구조

Fig. 2 fsync Merging Journal Format

본 기법에서는 아직 스토리지로의 쓰기 동작을 수행하지 않은 fsync 시스템 콜이 존재한다면 해당 fsync 시스템 콜과 함께 처리될 수 있도록 선택적으로 묶어서 하나의 저널로 만든다. 따라서 병합된 fsync 시스템 콜에 대해서 헤더 블록과 익스텐트 블록에 대한 플러시 명령과 커밋 블록에 대한 플러시 명령, 총 두 번의 플러시 명령만 발생한다. 그림 3은 두 개의 코어에서 fsync 시스템 콜을 호출하였을 때 iJournaling 기법과 fsync 병합 기법에서 몇 번의 플러시 명령이 발생하는지를 보여준다. fsync 시스템 콜은 특정 파일의 아이노드 정보가 저장된 헤더 블록와 익스텐트 블록, 그리고 fsync 저널의 트랜잭션이 끝남을 알리는 커밋 블록을 저널 영역에 기록한다. 이 때 fsync 시스템 콜 간의 순서를 맞추기 위해서 헤더 블록과 익스텐트 블록에 대해서 플러시 명령을 수행하여 실제 스토리지 영역에 반영하고, 데이터 내구성을 보장하기 위해서 커밋 블록에 대해 플러시 명령을 수행한다. 따라서 파일 수준 저널링을 사용하는 iJournaling에서는 fsync 시스템 콜 하나 당 헤더 블록과 익스텐트 블록에 대한 플러시 명령과 커밋 블록에 대한 플러시 명령, 총 2번의 플러시 명령이 발생하므로 fsync 시스템 콜이 호출된 횟수의 2배만큼의 플러시 명령이 발생한다. 반면 fsync 병합 기법에서는 스토리지로의 쓰기 동작을 수행하지 않은 fsync 시스템 콜과 묶어서 하나의 저널로 처리함으로써 헤더 블록과 익스텐트 블록에 대한 플러시 명령과 커밋 블록에 대한 플러시 명령, 총 2번의 플러시 명령만 발생한다. 전체적인 fsync 병합 동작은 그림 4와 같다. 하나의 저널로 합쳐진 fsync 시스템 콜의 처리를 맡아서 수행하는 fsync 시스템 콜을 리더 fsync라 칭하고, 리더 fsync에게 fsync 시스템 콜의 처리를 넘긴 fsync 시스템 콜을 멤버 fsync라 칭한다. 그림 4(a)의 merging 구간은 리더 fsync의 저널 블록 할당 시작 이후부터 멤버 fsync의 저널 블록 할당이 수행되기 전으로, 병합 가능한 구간을 나타내고, 그림 4(a)의 serialized 구간은 리더 fsync와



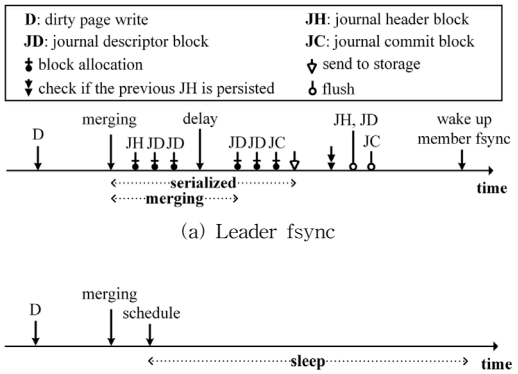
(a) iJournaling



(b) fsync Merging

그림 3 iJournaling과 fsync 병합 기법의 플러시 명령 횟수 비교

Fig. 3 Flush Count Comparison between iJournaling and fsync Merging



(a) Leader fsync

(b) Member fsync

그림 4 fsync 병합 기법의 동작 흐름

Fig. 4 fsync Merging Workflow

멤버 fsync의 저널 블록이 할당되는 구간으로, 같은 저널 영역을 사용하는 fsync 시스템 콜끼리는 블록 할당 순서가 유지되어야 하므로 동시에 처리될 수 없다. fsync 병합 기법에서는 fsync 시스템 콜 초반에 데이터 쓰기(dirty page write)가 끝나면 병합 가능한 리더 fsync가 존재하는지 확인하고, 병합 가능한 리더 fsync에 멤버 fsync의 fsync 시스템 콜 처리를 넘기고 리더

fsync의 수행이 끝날 때까지 기다린다. 병합 가능한 리더 fsync가 존재하지 않으면 해당 fsync 시스템 콜은 리더 fsync로써 동작한다. 리더 fsync는 fsync 시스템 콜이 호출된 파일을 저널링하기 위해 저널 블록을 할당받고 멤버 fsync가 존재할 경우 멤버 fsync 시스템 콜이 호출된 파일을 저널링하기 위한 저널 블록을 할당받는다. 또한 같은 저널 영역을 사용하는 fsync끼리 저널 블록이 순서대로 스토리지로 쓰여지도록 이전에 발생한 리더 fsync의 헤더 블록이 스토리지에 반영이 끝날 때까지 플러시 명령 수행을 대기한다. 플러시 명령의 수행이 끝나면 멤버 fsync를 깨우고 동작을 마친다.

fsync 병합 기법에서는 데이터 일관성을 유지하기 위해 iJournaling에서 제안한 복구 방법을 사용한다. fsync 병합 기법에서는 병합 도중에 시스템에 장애가 발생했을 때 해당 fsync 저널의 커밋 블록에 대한 플러시 명령이 수행되기 전이라도 데이터 일관성이 깨지지 않는다. 커밋 블록에 대한 플러시 명령이 수행되지 않았으면 병합된 fsync 저널이 온전히 쓰인 것을 보장할 수 없기 때문에 복구 작업 시 해당 fsync 저널의 데이터와 메타데이터는 복구하지 않기 때문에 데이터 일관성이 유지된다.

3.2 기다리는 시간을 부여한 fsync 병합 기법

fsync 병합 기법에서 실제로 병합 가능한 시간은 리더 fsync의 저널 블록 할당 구간(그림 4(a)의 merging 구간) 동안으로 실제 워크로드에서 병합으로 인한 fsync 시스템 콜의 지연시간 감소 효과를 보기에 충분하지 않다. 따라서 fsync 시스템 콜 수행 도중에 더 많은 병합이 발생할 수 있도록 기다리는 시간을 부여하여 fsync 시스템 콜의 지연시간을 감소시킬 수 있도록 한다. 기다리는 시간이 길수록 더 많은 병합이 발생하여 플러시 횟수를 줄일 수 있지만 병합으로 인한 이득보다 기다리는 시간으로 인한 손해가 커질 수 있다. 또한 워크로드마다 fsync 시스템 콜이 발생되는 주기가 다르기 때문에 기다리는 시간을 고정값으로 지정하면 모든 워크로드에서 최적의 성능을 발휘할 수 없다. 따라서 본 논문에서는 병합 개수와 기다리는 시간과의 트레이드오프를 고려하여 기다리는 시간을 조정할 수 있도록 fsync 병합 기법을 개선하였다.

병합된 개수($count_{merging}$)가 많을수록 플러시 명령 횟수가 줄어드므로 fsync 시스템 콜의 지연시간이 줄어들지만, 기다리는 시간($time_{delay}$)이 길수록 fsync 시스템 콜의 지연시간이 길어지므로 이득($benefit$)과 손실($cost$)을 아래 식 (1)과 같이 정의한다.

$$benefit = cost_{merging} \times time_{flush} \qquad (1)$$
$$cost = time_{delay}$$

$cost$가 $cost_{MAX}$보다 작고, 최대 병합 개수보다 작으면 기다리는 시간을 조금 더 부여하여 더 많은 병합이 발생할 수 있도록 한다. 기다리는 시간을 조금씩 증가시키면서 $benefit$ 변화 유무를 확인한다. $benefit$의 변화가 있으면 기다리는 시간을 더 부여하고, $benefit$의 변화가 없으면 기다리는 시간으로 인한 손실이 커지지 않도록 기다리는 것을 그만한다. $cost_{MAX}$값은 플러시 명령을 수행하는데 걸리는 시간으로 측정하여 플러시 명령 하나를 더 수행하는 것 이상의 오버헤드가 되지 않도록 한다. 기다리는 시간의 증가폭은 지연시간을 부여하지 않았을 때 병합이 발생한 시간의 평균값으로 설정한다.

4. 실 험

4.1 실험 환경

데스크탑 컴퓨터 환경은 Intel i5-6500(3.20GHz, 쿼드코어), DRAM 16GB, 삼성 850 EVO(256GB)이고, 운영체제는 ubuntu 16.04 LTS, 리눅스 커널 버전은 4.7.3을 사용하였고, 파일시스템은 ext4를 순차모드로 사용하였다.

4.2 실험 방법

표 1의 워크로드를 사용하여 ext4의 저널링 기법과 iJournaling 기법, fsync 병합 기법, 기다리는 시간을 부여한 fsync 병합 기법의 성능을 비교하였다. 기다리는 시간을 워크로드마다 달리 부여하는 fsync 병합 기법에서 사용하는 파라미터인 $cost_{MAX}$와 지연시간의 증가폭은 각각 400μs와 40μs로 설정하였다. 이 값은 그림 1과 같이 실험에 사용한 삼성 850 EVO SSD를 사용하여 fsync 수행 시간을 측정한 결과를 기반으로 설정하였다.

표 1 워크로드 구성
Table 1 Workloads Configuration

Sequential write	4KB write with fsync, Threads: 4
TPC-C	DBMS: MySQL, DB size: 10GB, Threads: 30, Rampup time: 120s, Runtime: 600s
Varmail	Filesize: 16KB, nFiles: 65536, Fileset: 1GB, Threads: 16, Runtime: 600s
LinkBench	DBMS: MySQL, DB size: 4GB, Threads: 30, Warmup time: 120s, Runtime: 600s

4.3 실험 결과

그림 5는 순차 쓰기 워크로드를 사용하여 ext4 저널링 기법과 iJournaling 기법, fsync 병합 기법에서의 fsync 시스템 콜의 지연시간을 누적분포함수(CDF)로 표현한 그래프이다. ext4 저널링 기법의 경우 컴파운드 트랜잭션으로 처리되므로 fsync 시스템 콜을 호출한 파일뿐만 아니라 일정 주기 동안 발생한 모든 파일의 변경사항까지도 저널링되어 각각의 fsync 시스템 콜의 지연시간이 길어져서 평균 fsync 지연시간이 길어진다.

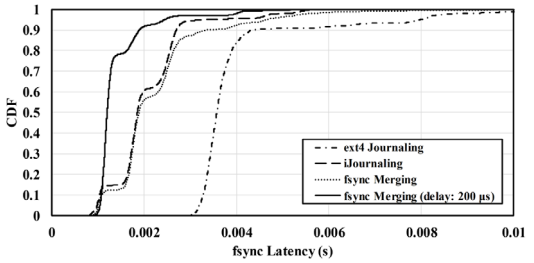


그림 5 ext4 저널링 기법과 iJournaling 기법, fsync 병합 기법의 성능 비교

Fig. 5 Performance Comparison between ext4 Journaling, iJournaling and fsync Merging

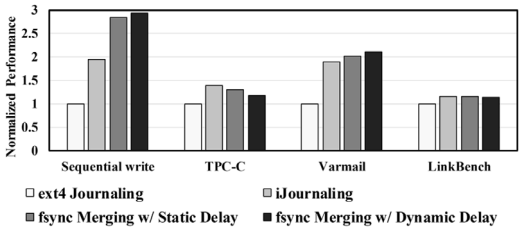
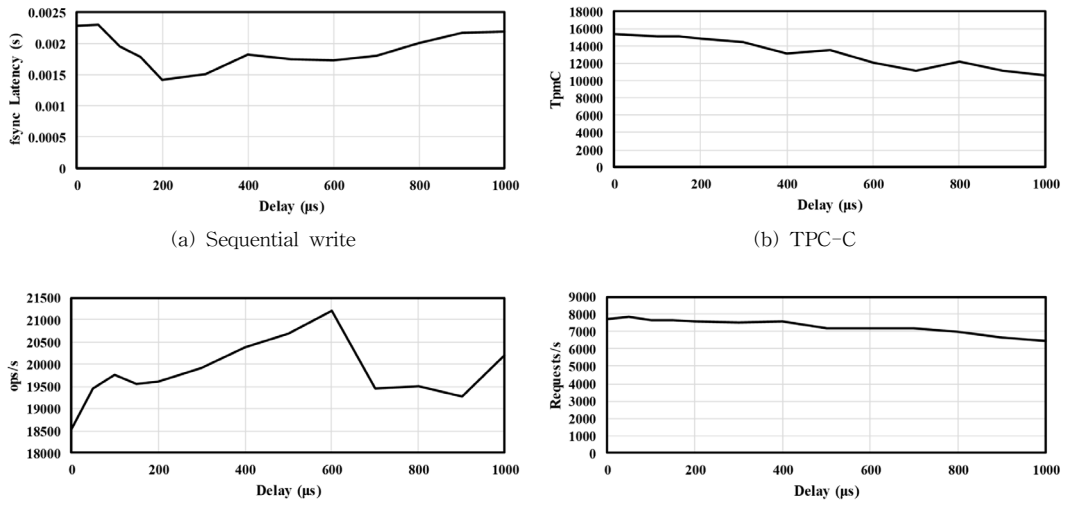


그림 6 워크로드별 각 기법의 성능 비교

Fig. 6 Performance Comparison of Each Technique by Workload

iJournaling 기법은 fsync 시스템 콜에 해당하는 파일만을 저널링하므로 ext4 저널링 기법에 비해 평균 fsync 시스템 콜의 지연시간이 향상된다. 하지만 짧은 시기에 여러 코어에서 fsync 시스템 콜이 호출되었을 때 각각의 코어에서 fsync 시스템 콜에 대한 저널링 동작이 개별적으로 처리되므로 플러시 명령 횟수가 증가하게 된다. 따라서 fsync 병합 기법에서는 플러시 명령 횟수를 줄이기 위해 짧은 시기에 여러 코어에서 발생한 fsync 시스템 콜을 병합하여 하나의 저널로 처리함으로써 iJournaling 기법에 비해 평균 fsync 지연시간이 줄어든 것을 볼 수 있다.

그림 6은 iJournaling 기법과 최적의 성능을 발휘했을 때의 기다리는 시간을 부여한 fsync 병합 기법, 기다리는 시간이 변하는 fsync 병합 기법의 성능을 ext4 저널링 기법의 성능을 1로 하여 표준화한 것이다. fsync 호출 주기가 상대적으로 짧은 순차 쓰기와 Varmail에서는 병합이 많이 발생하여 ext4 Journaling 또는 iJournaling에 비해 성능이 향상된 것을 볼 수 있다. 하지만 fsync 시스템 콜이 자주 발생하지 않는 TPC-C와 LinkBench 워크로드에서는 iJournaling에 비해 오히려 성능이 감소하였다. 이는 병합으로 인한 이득보다 병합을 위해 부여한 기다리는 시간으로 인한 손실이 더 컸기 때문이다. 그림 7은 기다리는 시간에 따른 fsync 병합 기법의 성능을 보여준다. 워크로드별로 fsync 시스템 콜이 호출되는 주기가 다르기 때문에 최적의 성능을 발휘할 때의 기다리는 시간이 다르다. 순차 쓰기의 경우 200μs, TPC-C의 경우 0μs, Varmail의 경우 600μs, LinkBench의 경우 50μs의 기다리는 시간을 부여하였을 때 최적의 성능을 발휘했다. fsync 시스템 콜 호출 주기가 짧은 순차 쓰기와 Varmail의 경우는 기다리는 시간을 증가시킬수록 병합으로 인해 플러시 명령 횟수가 감소하고 성능이 향상되는 것을 볼



(a) Sequential write (b) TPC-C

(c) Varmail (d) LinkBench

그림 7 기다리는 시간을 달리 부여한 fsync 병합 기법에서의 성능 비교

Fig. 7 Performance Comparison in fsync Merging with a Different Delay Time

수 있다. 그러나 TPC-C와 LinkBench와 같이 MySQL 데이터베이스 관리 시스템(DBMS)을 사용하는 데이터베이스 워크로드에서는 병합으로 인한 효과가 미미하다. 이중 쓰기 버퍼를 사용하는 MySQL DBMS에서는 두 종류의 fsync 시스템 콜이 존재한다. MySQL DBMS에서는 원자적 쓰기를 보장하기 위해 실제 스토리지 위치에 데이터를 쓰기 전에 메모리 내의 이중 쓰기 버퍼 영역에 데이터를 쓴 후 이중 쓰기 버퍼 영역이 다 채워지면 이를 스토리지 내의 이중 쓰기 버퍼 영역에 쓰기 위해 fsync 시스템 콜을 호출한다. 이후 실제 스토리지 위치에 쓰기 위해 각각의 페이지가 속한 DB 파일에 fsync 시스템 콜을 수행한다. 첫 번째 fsync 시스템 콜은 이중 쓰기 버퍼 영역이 다 채워진 후 동기적으로 수행되므로 병합이 발생하지 않지만, 두 번째 fsync 시스템 콜은 이중 쓰기 버퍼 영역에 포함된 각각의 페이지가 속한 DB 파일 개수만큼 병합이 발생할 수 있다.

5. 결 론

본 논문에서 제안한 fsync 병합 기법에서는 파일 수준 저널링에서 짧은 시기에 여러 코어에서 fsync 시스템 콜이 발생했을 때 플러시 명령 횟수가 증가하여 fsync 시스템 콜의 지연시간이 길어지는 문제를 병합을 통해 해결하였다. 또한 fsync 시스템 콜이 호출되는 주기와 병합된 개수를 고려하여 병합을 위해 기다리는 시간을 조정할 수 있도록 하였다.

References

[1] Park, Daejun, and Dongkun Shin. "iJournaling: Fine-grained journaling for improving the latency of fsync system call," *Proc. of the 2017 USENIX Annual Technical Conference*, 2017.

[2] Daejun Park and Dongkun Shin, "Fine-Grained journaling for Reducing Fsync System Call Latency," *Korean Institute of Information Scientist and Engineers*, pp. 1219-1221, Dec, 2015. (in Korean)

[3] Won, Youjip, et al., "Barrier-enabled IO stack for flash storage," *16th USENIX Conference on File and Storage Technologies*, 2018.

김 솜

2018년 경희대학교 컴퓨터공학과 졸업(학사). 2018년~현재 성균관대학교 전자전기컴퓨터공학과 석박통합과정. 관심분야는 임베디드 시스템, 파일시스템, 플래시 메모리

박 대 준

2013년 성균관대학교 컴퓨터공학과 졸업(학사). 2013년~현재 성균관대학교 전자전기컴퓨터공학과 석박통합과정. 관심분야는 임베디드 시스템, 파일시스템, 저널링, 플래시 메모리

신 동 군

1994년 서울대학교 계산통계학과(학사) 2000년 서울대학교 전산과학과(석사). 2004년 서울대학교 컴퓨터공학부(박사). 2004년~2007년 삼성전자 소프트웨어 책임연구원. 2007년~현재 성균관대학교 정보통신대학 부교수. 관심분야는 임베디드 시스템, 실시간 시스템, 저전력 시스템