creative commons

COMMONS DEED

저작자표시-비영리 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.
- 이차적 저작물을 작성할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사학위논문

ZNS SSD를 이용한 F2FS의 효율적인 순서 보존 복구 기법

An Efficient Order-Preserving Recovery for F2FS with ZNS SSD

2023년 8월

서울대학교 대학원

컴퓨터공학부

이 의 동

공학석사학위논문

ZNS SSD를 이용한 F2FS의 효율적인 순서 보존 복구 기법

An Efficient Order-Preserving Recovery for F2FS with ZNS SSD

2023년 8월

서울대학교 대학원

컴퓨터공학부

이 의 동

ZNS SSD를 이용한 F2FS의 효율적인 순서 보존 복구 기법

An Efficient Order-Preserving Recovery for F2FS with ZNS SSD

지도교수 김 진 수

이 논문을 공학석사 학위논문으로 제출함

2023년 6월

서울대학교 대학원

컴퓨터공학부

이 의 동

이의동의 공학석사 학위 논문을 인준함

2023년 7월

위 원 장:	염 헌 영	(인)
부위원장:	김 진 수	(인)
위 원:	전 병 곤	(인)

초 록

현대 저장장치는 성능 향상을 위해 쓰기 버퍼를 사용하여 쓰기 요청을 병렬로 처리하고 임의의 순서로 완료한다. 그러나 갑작스러운 전원 장애가 발생할 경우 데이터 손실이 유발되는 위험을 내재하고 있다. 이에 따라 리눅스 파일시스템은 fsync() 시스템콜을 제공하여 데이터 손실을 방지하고 쓰기 순서를 보장한다. 하지만 널리 사용되는 리눅스 파일시스템 중 하나인 F2FS에서 fsync() 시스템콜은 데이터 일관성을 보장하지 못할 뿐만 아니라 효율적이지 못한 문제를 가지고 있다.

따라서 우리는 데이터 일관성을 보장하기 위한 파일시스템 복구 기법을 제안한다. 제안한 기법은 데이터의 영속성을 효율적으로 확인하기 위해 ZNS가 제공하는 쓰기 포인터를 활용하였고, 이를 통해 복구 시간에 미치는 영향을 최소화하였다. 더불어 이 기법은 일관성 수준을 높일 뿐만 아니라 fsync()의 성능도 개선하였다. 결과적으로 실제 워크로드에서 최대 1.2배 성능이 향상된다.

주요어: F2FS, ZNS, SSD, Filesystem

학 번: 2021-25814

차 례

제 1 장 서론 **1**

제 2 장 배경 지식 **3**

2.1 ZNS 인터페이스 3

2.2 쓰기 버퍼 4

2.3 fsync() 시스템콜 4

2.4 F2FS 파일시스템 5

2.5 파일시스템 일관성 7

제 3 장 연구 동기 **9**

3.1 fsync()의 데이터 손실 9

3.2 fsync()의 수행 시간 10

3.3 쓰기 포인터를 이용한 영속성 판단 10

제 4 장 관련 연구 **12**

제 5 장 설계 및 구현 **14**

5.1 데이터 일관성을 위한 파일시스템 복구 기법 14

5.2 fsync()의 성능 개선 16

5.2.1 PLP 미지원 장치를 위한 POSIX 모드 16

5.2.2 PLP 지원 장치를 위한 `nobarrier` 모드 17

5.3 ZNS 에뮬레이터 17

5.3.1 충돌 테스트 모델 17

5.3.2 쓰기 버퍼 모델 18

제 6 장 실험 결과 및 분석 **21**
6.1 실험 환경 21
6.2 데이터 일관성 실험 22
6.3 마이크로 벤치마크 22
6.4 매크로 벤치마크 24
6.5 복구 시간 25

제 7 장 결론 **27**

ABSTRACT **32**

표 차례

표 3.1 단계별 fsync() 수행 시간 10

표 6.1 실험 환경 21

표 6.2 데이터 일관성 실험 결과 22

표 6.3 전방향 복구 소요 시간 26

그림 차례

그림 2.1 F2FS의 fsync() 5

그림 5.1 OPRW의 복구 과정 15
그림 5.2 OPRW의 fsync() 16
그림 5.3 충돌 테스트 과정 18
그림 5.4 쓰기 버퍼 및 쓰기 포인터의 상태 변화 19

그림 6.1 마이크로 벤치마크 결과 23
그림 6.2 매크로 벤치마크 결과 24
그림 6.3 확장성 실험 결과 25

제 1 장 서론

현대 저장장치의 경우 높은 성능을 위해 쓰기 명령에 대한 버퍼링 기법을 필수적으로 사용한다. 즉 호스트는 저장장치에 쓰기 요청을 내린 후에 비휘발성 저장장치에 기록이 완료될 때까지 기다리지 않는다. 이 과정에서 저장장치는 여러 쓰기 요청을 병렬적으로 처리하여 시스템 성능을 향상시킨다. 한편 이러한 정책은 갑작스러운 전원 장애 상황에서 데이터의 손실을 일으키므로 호스트는 중요한 데이터를 보호하기 위한 별도의 메커니즘을 필요로 한다.

리눅스의 파일시스템들은 fsync() 시스템콜을 제공하여 중요한 데이터를 보존하고 쓰기 순서를 보장한다. 특정 파일에 대해 fsync()가 호출되면 파일시스템은 해당 파일에 대한 데이터와 메타데이터를 저장장치에 영속적으로 기록한 후 시스템콜을 반환한다. fsync()가 성공적으로 완료되면 사용자는 해당 파일이 장치에 실제로 기록되었으며 전원이 갑자기 꺼지는 상황에서도 데이터가 안전하다는 것을 기대할 수 있다.

우리는 널리 사용되는 파일시스템 중 하나인 F2FS에서 fsync()의 기능이 불충분할 뿐만 아니라 비효율적이라는 것을 확인하였다. F2FS는 파일의 변경된 데이터와 노드 블록을 차례로 기록하는 것으로 추가적인 메타데이터 쓰기 없이 fsync() 동작을 완료한다. 구체적으로 F2FS는 데이터 블록의 DMA 전송이 완료된 후 노드 블록의 쓰기 요청을 전송하여 둘 사이의 갱신 순서를 지키고자 한다. 그러나 DMA 전송을 순차적으로 수행하더라도 저장장치 수준에서 의도한 순서대로 쓰기가 완료되는 것은 보장하지 못한다. 따라서 F2FS에서 fsync() 도중 전원 장애가 발생하면 데이터가 손실될 위험이 있다.

우리는 차세대 저장장치 인터페이스인 ZNS(Zoned NameSpace) SSD[3, 11, 12]에서 이 문제를 해결할 방법을 설계하였다. ZNS SSD는 블록 인터페이스 기반 SSD와 달리, 호스트가 사용할 수 있는 명령을 제한하고 관리 부담을 일부 호스트에게 위

임하는 특징을 가지고 있다. ZNS SSD는 전체 공간을 고정된 크기의 존(zone)으로 나누고 존 내에서는 순차 쓰기만 허용한다. 이를 위해 존마다 쓰기 위치를 표현하는 쓰기 포인터(write pointer)를 관리하는데, 우리는 해당 정보가 갑작스러운 전원 종료 후 복구가 필요한 상황에서 유용하게 사용될 수 있다는 점에 주목하였다. 우리는 이러한 ZNS의 특성을 활용하여 데이터 손실 없는 효율적인 파일시스템 복구 기법을 제안한다. 제안한 기법을 통해 파일시스템의 안정성을 높임과 동시에 fsync()의 성능 또한 개선하였다. 결과적으로 fsync()가 주기적으로 발생하는 실제 워크로드에서 최대 1.2배의 대역폭 개선을 얻었다.

논문의 이어지는 내용은 다음과 같다. 2장에서는 배경이 되는 개념들을 설명하고, 3장에서 기존 F2FS의 문제를 제시하며, 4장에서는 관련 연구를 다룬다. 5장에서는 해결 방안과 추가적인 성능 이득을 검토하고, 6장에서 성능 평가를 진행한 후 7장에서 결론을 정리한다.

제 2 장 배경 지식

2.1 ZNS 인터페이스

전통적인 SSD는 고정된 크기의 블록 인터페이스를 이용하여 호스트 시스템과 통신한다. 블록 인터페이스는 제자리 덮어쓰기가 가능한 저장장치를 위해 설계되었으므로, SSD 컨트롤러는 주소 변환과 GC(Garbage Collection) 등의 다양한 내부 작업을 통해 덮어쓰기가 가능한 블록 주소 공간을 제공한다. 이러한 내부 작업으로 인해 컨트롤러의 비용과 관리 부담이 증가하고, 호스트가 예측할 수 없는 시점에 내부 동작이 발생하여 빠르고 일정한 응답속도를 확보하기 어렵다. 이를 극복하기 위한 새로운 인터페이스로 오픈 채널 SSD[13], 멀티 스트림 SSD[17] 등의 기법이 제안된 바 있으며, 현재 같은 목적으로 ZNS SSD가 상용화되어 있다.

ZNS SSD는 여러 개의 블록을 하나의 존으로 묶어서 관리하고, 각 존에 대한 쓰기 순서와 삭제를 호스트에게 위임한다. 이를 통해 장치는 자체적인 데이터 재배치와 삭제 연산을 제거하여 예측 가능한 응답속도를 제공할 수 있다. 반면 호스트 시스템은 새로운 인터페이스를 위해 새로운 소프트웨어 구조를 필요로 한다.

ZNS SSD의 존은 순차적으로만 기록될 수 있다. 따라서 각 존은 마지막으로 쓰인 위치를 유지해야 할 필요가 있으며, 이를 위해 장치는 쓰기 포인터를 관리하고 호스트와 공유한다[11, 12]. 만약 시스템 충돌이 발생하면, ZNS SSD는 내부 복구 메커니즘을 이용하여 가장 최근에 쓰인 위치를 찾아내고 쓰기 포인터를 해당 위치로 재조정한다. 이때, 쓰기 포인터를 기준으로 앞에 위치한 영역은 영속적으로 쓰인 상태이고, 반대로 뒤에 위치한 영역은 써지지 않은 상태가 된다. 쓰기 포인터가 존의 마지막에 도달하면 해당 존은 닫힌 상태가 되며 초기화할 때까지 추가적인 쓰기 동작은 허용하지 않는다. ZNS의 요구조건을 준수하기 위해 호스트는 같은 존에 대한 쓰기 순서를 강제해야 하는데, 이를 현재 커널에서는 데드라인 스케줄러가 담당한

다[2].

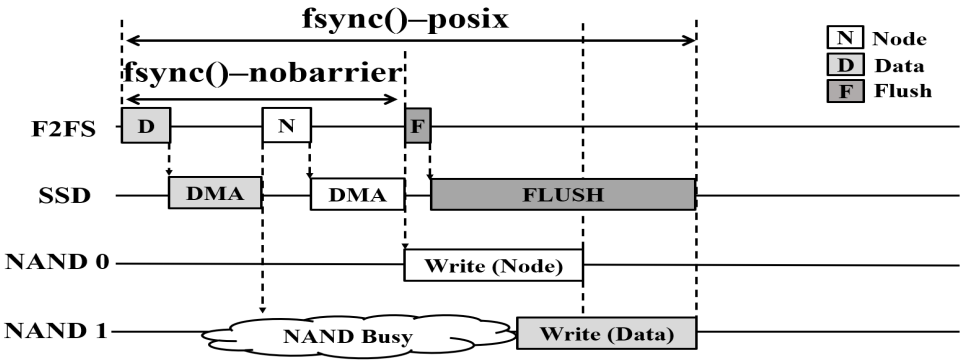
2.2 쓰기 버퍼

낸드 플래시(NAND flash)는 용량을 늘리기 위해 셀 당 비트 수를 늘리는 방향으로 발전하고 있다. 이에 따라 낸드 쓰기 시간은 점점 증가한다. 따라서 많은 SSD는 낸드 플래시의 수행 시간을 숨기기 위해 쓰기 버퍼를 지원한다. 해당 기능을 지원하는 경우, 쓰기 요청을 받은 SSD는 휘발성 메모리로 구성된 쓰기 버퍼에 사용자의 데이터를 전송한 후 해당 요청을 완료한다. 만약 낸드 플래시에 데이터가 저장되기 전에 전원이 꺼지면 사용자의 데이터는 손실된다. 데이터를 영속적으로 저장하기 위해서는 별도로 플러시 명령을 수행해야 한다.

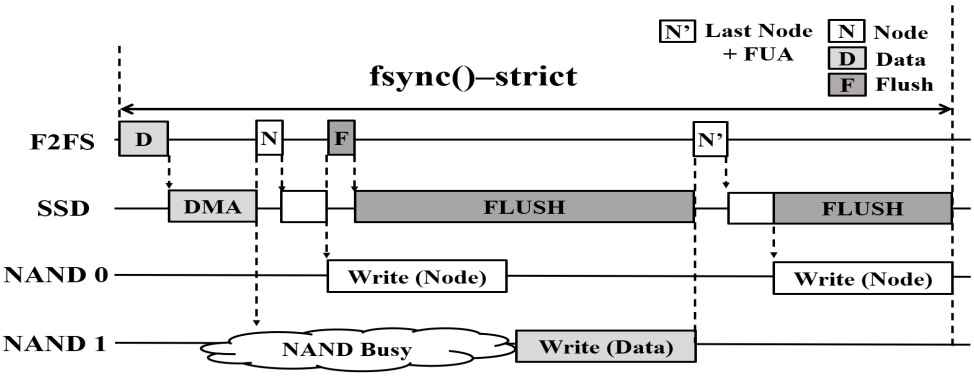
2.3 fsync() 시스템콜

일반적인 POSIX 호환 파일시스템에서 fsync()는 데이터의 영속성과 쓰기 명령의 순서를 보장하는 목적으로 사용된다. 파일시스템은 일관성을 위해 fsync() 명령을 주기적으로 사용한다. fsync()는 처리 과정에서 비휘발성 저장장치의 쓰기 응답시간을 노출하므로 매우 비싼 연산이다. 실제로 이 동작은 다양한 워크로드에서 성능의 병목을 일으킨다[21, 25, 26].

일부 SSD는 내장형 백업 배터리를 장착하여 갑작스러운 전원 손실에 대한 데이터 보호 기능을 제공한다. PLP(Power Loss Protection)라고 불리는 이 기능은 DMA 전송만으로 데이터의 영속성을 보장한다. 일부 파일시스템은 PLP를 지원하는 저장장치에서 fsync()의 성능을 향상시키기 위해 `nobarrier` 모드를 제공한다. 해당 모드에서 파일시스템은 비휘발성 저장장치의 쓰기 시간을 기다리지 않지만, 사용자는 여전히 fsync()를 통해 데이터의 영속성을 보장할 수 있다.



(a) POSIX 모드, nobarrier 모드



(b) strict 모드

그림 2.1: F2FS의 fsync()

2.4 F2FS 파일시스템

F2FS는 플래시 저장장치를 위한 리눅스 파일시스템이다[20]. 로그 구조 파일시스템(LFS, Log-structured File System)을 기반으로 하되, 일부 메타데이터를 임의 쓰기 방식으로 관리하여 메타데이터가 연쇄적으로 갱신되는 문제를 해결하고 높은 성능을 얻어냈다. F2FS는 현재 안드로이드 기반의 모바일 시스템에서 널리 사용되고 있다.

F2FS의 메타데이터는 기록되는 방식에 따라 크게 두 가지로 분류할 수 있다.

첫 번째, 주 영역의 세그먼트(segment) 안에서 순차적으로 공간을 할당하여 쓰는 메타데이터이고 노드(node)가 여기에 속한다. 노드는 특정 파일의 메타데이터를 관리하는 블록으로서 데이터의 위치 및 파일의 세부 정보를 담고 있다. 두 번째, 고정된 위치에 기록되는 메타데이터이다. 세그먼트 정보 테이블(SIT, Segment Information Table), 노드 주소 테이블(NAT, Node Address Table)과 같은 전역으로 관리되는 메타데이터가 여기에 포함이 된다. 해당 메타데이터들은 체크포인트(checkpoint)가 수행될 때 영속적으로 기록되고, 이때 임의 쓰기를 동반한다.

F2FS는 파일시스템의 일관성을 유지하기 위해 주기적으로 체크포인트를 수행한다. 체크포인트는 해당 시점의 노드 및 세그먼트 상태와 함께, 변경된 모든 파일의 논리적 구조를 보존한다. 시스템 충돌이 발생하면 F2FS는 최신의 체크포인트를 찾아 해당 상태로 돌아가는 것으로 복구 과정을 단순화할 수 있다. 하지만 fsync()의 경우 호출할 때마다 매번 체크포인트를 수행하면 성능이 하락하므로, 파일의 변경된 데이터와 다이렉트 노드만 기록하는 방법을 사용한다. 구체적으로, fsync()의 대상이 되는 노드들에게 특별한 플래그(flag)를 붙이고 이것들을 단일 세그먼트에 순차적으로 기록한다. 따라서 체크포인트를 복구하는 것만으로는 여전히 복구가 안 된 노드들이 존재하므로, 체크포인트 이후에 쓰인 노드 중 fsync() 대상이었던 것들을 순차적으로 복구하는 과정이 필요하고, 이를 전방향 복구(roll-forward)라고 부른다. 이 과정에서 노드가 가리키는 모든 데이터 블록들을 검사하면서 오래된 블록은 무효화하고 새로운 블록은 유효화하여, 이를 세그먼트 정보 테이블에 갱신한다. 또한 해당 노드의 위치 정보를 노드 주소 테이블에 등록한다.

그림 2.1a는 fsync()의 수행 과정을 보여준다. fsync() 시스템콜이 호출되면, 먼저 해당 파일의 변경된 데이터 블록들을 모아 저장장치로 쓰기 요청을 보낸다. 이후, 데이터 블록의 DMA 전송이 완료될 때까지 기다리고, 이어서 해당 파일의 변경된 노드 블록 중 다이렉트 노드를 모아 쓰기 요청을 보낸다. 모든 DMA 전송이 완료되면 POSIX 모드에서는 플러시(flush) 명령[6]을 통해 데이터와 노드의 영속성을 보장하는 과정을 거친다. 반면 `nobarrier` 모드에서는 즉시 fsync() 시스템콜을

반환한다.

`strict` 모드는 앞선 두 모드와는 다르게 원자성을 보장한다. 이를 위해 기존 F2FS의 원자적 쓰기 메커니즘을 이용한다. 그림 2.1b는 `strict` 모드의 수행 과정을 보여준다. fsync() 대상의 모든 노드 블록에 특별한 플래그를 붙이는 기존 방식과 달리, 가장 마지막 노드 블록에만 해당 플래그를 표시한다. 또한 마지막 노드 블록 앞뒤로 플러시 명령을 넣어 마지막 노드 블록과 그 외 블록의 순서를 보장한다. 마지막 노드 블록이 복구되어야지만 앞선 노드 블록들을 복구하는 방식으로 fsync()의 원자성을 보장한다.

F2FS는 로그 구조 파일시스템에 기반을 두기 때문에 큰 변경 없이 ZNS 인터페이스를 적용할 수 있다. 다만 기존 F2FS의 임의 쓰기 동작들은 ZNS SSD에서 수행될 수 없으므로 GC의 비용을 줄이기 위한 SSR(Slack Space Recycling) 기법은 사용할 수 없다. 또한 메타데이터 갱신을 위해 임의 쓰기가 가능한 존이 추가로 필요하다.

2.5 파일시스템 일관성

파일시스템에는 데이터와 메타데이터를 위한 여러 수준의 일관성이 존재하는데, 이는 메타데이터 일관성, 데이터 일관성, 버전(version) 일관성으로 구분될 수 있다[16].

메타데이터 일관성의 경우, 오직 메타데이터들 사이의 일관성만 유지하면 된다. 예를 들면, 아이노드를 할당하기 위한 비트맵과 아이노드의 개수를 나타내는 카운터가 일관성을 가지기 위해 비트맵의 유효한 비트의 수와 카운터의 값이 일치해야 한다. 반면 파일의 데이터가 손실되어도 메타데이터 일관성은 문제가 없다.

데이터 일관성은 메타데이터 일관성보다 더 높은 수준을 요구하는데, 메타데이터뿐만 아니라 파일에 속한 데이터도 그 일관성을 유지해야 한다. 데이터 일관성을 보장하려면 파일에는 새로 갱신된 데이터 또는 이전 데이터가 포함되어야 한다. 만약 다른 파일의 데이터나 쓰레기 데이터가 포함되어 있다면 데이터 일관성은 손상된

것으로 간주한다.

버전 일관성은 메타데이터와 데이터의 버전 일치를 요구한다. 데이터 일관성보다 더 강한 수준의 일관성으로 파일 내에서 서로 다른 버전의 데이터가 읽히면 버전 일관성을 만족하지 못한다.

제 3 장 연구 동기

3.1 fsync()의 데이터 손실

F2FS에서 fsync() 수행 도중 시스템 충돌이 발생하면 사용자 데이터가 손실되는 잠재적인 위험이 존재한다[9]. F2FS는 노드 블록의 쓰기 요청을 데이터 블록의 DMA 전송이 끝날 때까지 미루면서, 사용자 데이터가 낸드에 먼저 기록된 후 노드의 포인터가 변경되기를 기대한다. 그러나 DMA를 순서대로 처리하더라도 SSD 내부에서 낸드에 저장되는 순서는 뒤바뀔 수 있다. 그림 2.1a를 보면 데이터 D의 DMA 전송이 먼저 완료되었어도 데이터 D가 기록될 낸드 다이가 이미 동작 중이기 때문에, 뒤에 요청된 노드 N의 낸드 쓰기 동작이 먼저 시작하는 것을 확인할 수 있다. 데이터보다 노드가 먼저 낸드에 저장된 후 시스템 충돌이 발생하게 되면, 파일시스템 복구 시 데이터 일관성을 별도로 검사하지 않으므로, 해당 노드는 유효하지 않은 데이터를 가리키고 이는 기존 데이터의 손실로 이어질 것이다. 이것은 데이터 일관성을 훼손하는 동작이다[15, 16].

F2FS는 이러한 데이터 손실 문제를 `strict` 모드를 통해 해결하고자 하였다[9]. `strict` 모드는 플러시 명령을 이용하여 마지막 노드 블록과 나머지 블록 간의 쓰기 순서를 강제하는 방식으로 데이터를 보존한다. 하지만 이 과정에서 추가적인 플러시 동작으로 인한 성능 하락은 불가피하다.

여러 환경적인 요인들에 의해서도 이러한 일관성 문제가 회피될 수 있다. 첫 번째, PLP를 지원하는 SSD의 경우 DMA 전송만으로 영속성을 보장하기 때문에 이론적으로 데이터 일관성 문제가 발생하지 않는다. 그러나 실제 PLP를 지원하는 SSD 중에서 갑작스러운 시스템 충돌에 의해 데이터를 손실하는 제품도 일부 존재한다[14, 28]. 또한 적은 확률이지만 커널 오류(kernel panic), 칩 오동작 등으로 인해 DMA 전송이 완료된 데이터도 손실될 수 있다. 두 번째, 쓰기 버퍼의 크기가 작을수

표 3.1: 단계별 fsync() 수행 시간

Item	Latency(us)	Ratio(%)
Prepare & Dispatch (Data)	10.40	15.77
Wait for DMA (Data)	15.25	23.12
Prepare & Dispatch (Node)	16.16	24.50
Wait for DMA (Node)	16.38	24.83
Flush	2.12	3.21
ETC	5.65	8.57

록 쓰기 순서가 뒤바뀔 가능성이 작아진다. 하지만 저장장치는 성능을 높이기 위해 더 높은 병렬성을 추구하므로 쓰기 버퍼의 크기는 점점 증가할 것이고, 이에 따라 이러한 데이터 손실 문제는 더욱 대두될 것이다.

3.2 fsync()의 수행 시간

표 3.1은 PLP를 지원하는 저장장치에서 4KB 임의 쓰기 워크로드를 수행했을 때, fsync()의 단계별 수행 시간을 분석한 것이다. DMA 전송은 fsync() 수행 시간에서 약 50%의 비중을 차지하는데, 이는 작지 않은 수치이다. 해당 시간 동안 쓰레드는 다음 쓰기 요청을 보내기 위해 저장장치로부터 응답을 기다리고 있으므로 CPU 및 I/O의 활용도는 떨어질 것이다. 이는 점점 증가하는 SSD의 내부 병렬성을 충분히 활용하지 못하는 문제를 유발한다.

3.3 쓰기 포인터를 이용한 영속성 판단

앞서 상술한 fsync()의 어려움은 블록 인터페이스의 제한적인 정보 제공에서 기인한다. 블록 인터페이스에서 모든 블록은 임의 시점에 임의의 순서로 덮어쓰기가

가능하므로 블록 간에 명시적인 시간 순서를 파악하는 것은 어렵다. 또한 모든 블록에는 항상 데이터가 존재하는 것으로 가정하며, 특정 블록에 유효한 데이터가 존재하는지 확인하는 방법을 제공하지 않는다. 이러한 정보들은 일관성이 있는 파일시스템을 구동하는 데 필수적이기 때문에, 전통적인 파일시스템에서는 할당 비트맵과 같은 별도의 메타데이터를 사용하여 블록의 유효 여부를 판단한다. 하지만 이러한 기법은 추가적인 I/O를 발생시키며 공간 오버헤드를 유발한다.

반면 ZNS에서는 쓰기 포인터를 영속성 판단 용도로 사용할 수 있다. 쓰기 포인터는 항상 순차적으로 증가하므로 쓰기 포인터만 알고 있다면 별도의 I/O 없이 간단한 비교 연산만으로 블록의 영속 여부를 판단 할 수 있다. 구체적으로, 쓰기 포인터보다 전에 위치한 블록은 영속적으로 쓰여 있고, 쓰기 포인터와 같거나 뒤에 위치한 블록은 쓰여있지 않았다고 간주할 수 있다. 우리는 이러한 쓰기 포인터의 숨은 속성을 이용하여 쓰기 요청 사이의 순서를 빠르게 파악하고, 이를 통해 데이터 일관성 문제를 효율적으로 해결하고자 하였다.

제 4 장 관련 연구

많은 연구에서 fsync() 수행 시간이 길어지는 주된 원인을 쓰기 순서를 지키기 위한 DMA 전송과 플러시 명령 때문으로 분석하고 있다. 이에 따라 DMA 전송과 플러시 명령을 제거하는 방향으로 여러 연구가 진행되었다.

BarrierFS[26]는 I/O 스택의 계층별 부분 순서를 일치시켜 전체 I/O의 쓰기 순서를 보장한다. 다만 쓰기 명령을 보낸 순서대로 DMA를 전송하기 위해 SCSI의 단일 큐 구조를 사용하는 부분은 NVMe와 같이 여러 큐를 지원하는 인터페이스에서 적용하기 어렵다. 또한 DMA 전송 순서와 비휘발성 저장장치에 저장되는 순서를 일치시키기 위해 캐시 배리어(cache barrier)[8] 명령을 사용하는데, 이는 현재 eMMC 제품에서만 지원되는 제약이 있다.

HoraeFS[21], MQFS[23]는 PMR[10]을 활용하여 DMA 전송과 플러시 명령을 제거하였다. HoraeFS는 쓰기 명령의 순서 정보를, MQFS는 NVMe의 큐 정보를 PMR 영역에 관리하고, 갑작스러운 시스템 충돌이 발생했을 때 해당 정보들을 이용하여 쓰기 순서를 보존한다. 그 결과 BarrierFS가 가지고 있는 단일 큐 및 캐시 배리어의 제약은 해결하였지만, PMR을 지원하는 저장장치에서만 동작하는 제약이 있다.

OPTR[14]은 단일 슈퍼블록에 이어 쓰는 기존 FTL의 특성을 이용하여 저장장치 수준에서 쓰기 요청의 순서를 보장하는 FTL을 제안하였다. 물리적인 낸드 페이지마다 FTL의 메타데이터를 관리할 수 있는 작은 영역이 할당되어있는데, OPTR은 해당 영역에 명령의 순서를 나타내는 특별한 정보를 기록하여 쓰기 순서를 보존한다. 하지만 여러 존에 병렬적으로 기록할 수 있는 ZNS SSD에서는 존 간 시간 순서를 파악하기 어렵기 때문에 OPTR 기법을 적용하기 어렵다.

플러시 명령 자체의 비용을 줄이려는 노력도 존재한다. RFLUSH[27]는 기존 플러시 명령의 범위가 저장장치 전체였다는 것을 문제로 지적하고 플러시 명령의 범위를 파일에 한정시키는 기법을 제안하였다. fsync()를 수행하는 쓰레드와 그렇지

않은 쓰레드가 혼재하는 상황에서 필요한 부분만 저장장치에 영속적으로 저장하여, 플러시 명령의 수행 시간을 단축시키고 fsync()의 성능을 개선하였다.

제 5 장 설계 및 구현

5.1 데이터 일관성을 위한 파일시스템 복구 기법

3장에서 제시한 문제를 해결하는 간단한 방법은 fsync() 수행 시 데이터와 노드의 DMA 전송 사이에 명시적인 플러시 명령을 삽입하는 것이다. 이 경우 데이터 블록이 영속적으로 저장된 후 노드의 쓰기 명령이 시작되므로 노드가 가리키는 데이터는 항상 유효하다. 그러나 추가적인 플러시 동작은 fsync()의 수행 시간을 크게 증가시키므로 실용적이지 않다. 문제를 해결하는 다른 방법으로는 실제 시스템 충돌이 발생했을 때 쓰기 순서의 뒤바뀜을 확인하여 파일시스템을 복구하는 방법이 있다. 이는 데이터 보호를 위한 작업을 최대한 미루는 낙관적인 접근 방법으로, fsync() 과정을 간소화할 수 있지만 파일시스템 복구 시간이 길어지는 문제를 가지고 있다. 특히 기존 블록 인터페이스에서는 데이터 블록의 유효함을 판단하기 위해 추가적인 읽기 I/O가 필요하므로 복구 시간이 증가하는 것을 피할 수 없다.

따라서 본 연구에서는 ZNS SSD를 이용하여 데이터 손실 없이 효율적으로 손상된 파일시스템을 복구하는 OPRW(Order-Preserving Recovery by Write pointer) 기법을 제안한다. 우리는 이 기법을 통해 일관성 문제를 해결함과 동시에 fsync()의 성능 또한 개선하였다. OPRW는 앞서 설명한 낙관적인 접근 방법에 기반을 두고 있고, 복구 시간이 길어지는 문제를 쓰기 포인터를 통해 해결하였다. 그림 5.1은 fsync() 동작 중 전원이 차단되어 파일시스템이 손상된 상황에서 OPRW의 전반적인 복구 과정을 보여주고 있다. 문제가 되는 부분은 마지막 체크포인트가 완료된 시점 이후에 기록된 노드들이다. 3장에서 상술한 바와 같이 해당 노드에서 갱신된 데이터 블록은 유효하지 않을 가능성이 있다. 따라서 전방향 복구 과정에서 노드를 순차적으로 복구할 때마다 데이터 일관성을 먼저 검사한다. 만약 복구하고자 하는 노드의 데이터가 손실되었다면 해당 노드 블록의 복구를 중단한다. 이렇게 될 경우, 이전의

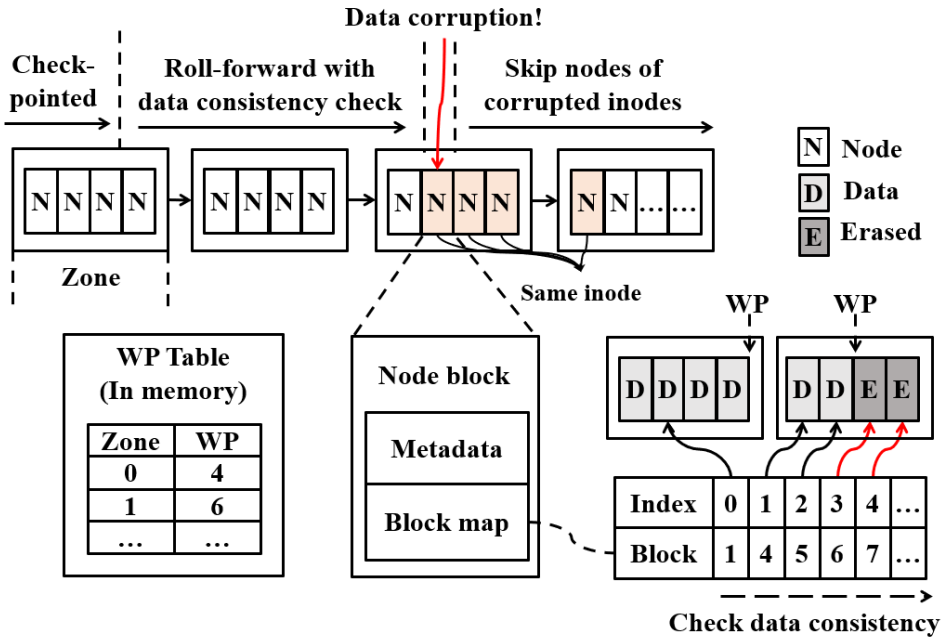
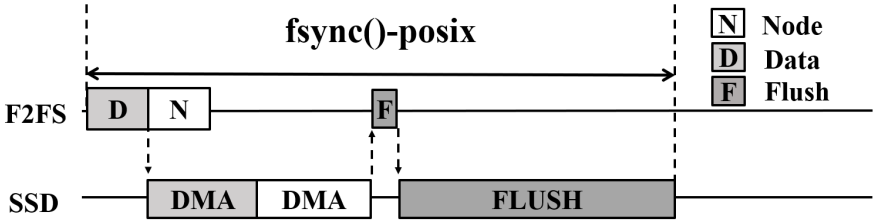


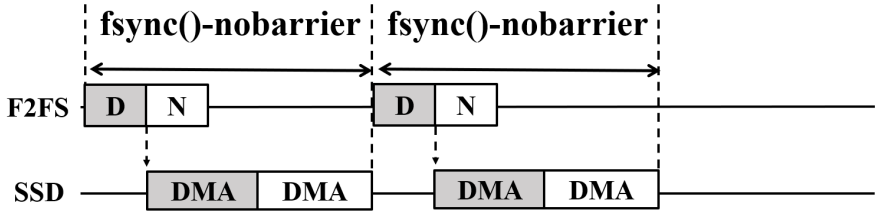
그림 5.1: OPRW의 복구 과정

온전한 노드를 유지하게 되므로 데이터 일관성을 보장할 수 있다. 노드는 순차적으로 기록되기 때문에 남아있는 노드 중 일관성이 훼손된 아이노드에 속한 것들도 복구 대상에서 제외한다.

알고리즘 1은 전방향 복구 과정 중 데이터 일관성을 검사하는 구체적인 과정을 서술한 것이다. 핵심 아이디어는 쓰기 포인터를 통해 I/O 동작 없이 오직 메모리 접근과 간단한 비교 연산만을 수행하여 신속하게 일관성을 검사하는 것이다. 우선 모든 존에 대한 쓰기 포인터를 단일 존 관리 명령[11]을 통해 한 번만 메모리에 적재한다. 존의 개수는 제한이 있으므로 쓰기 포인터를 위한 메모리 사용량은 미미하다. 쓰기 포인터가 모두 메모리에 적재된 후, 노드가 소유한 모든 블록 맵핑 엔트리를 검사한다. 노드가 가리키는 주소가 쓰기 포인터보다 작으면 유효한 데이터를 가리키는 것으로, 크거나 같으면 무효한 것으로 판단한다. 무효한 데이터를 가리키는 엔트리가 하나라도 존재한다면 해당 노드의 데이터 일관성이 훼손된 것이므로 복구



(a) POSIX 모드



(b) `nobarrier` 모드

그림 5.2: OPRW의 fsync()

대상에서 제외한다. 데이터가 손실된 아이노드는 *skip_list*에 삽입하여, 해당 아이노드에 속한 나머지 노드들을 복구에서 제외한다.

5.2 fsync()의 성능 개선

5.2.1 PLP 미지원 장치를 위한 POSIX 모드

POSIX 모드에서 F2FS는 데이터 일관성과 영속성을 목적으로, 그림 2.1a와 같이 두 번의 DMA 전송과 한 번의 플러시 명령을 순차적으로 수행한다. 그러나 DMA를 순서대로 수행하는 것은 데이터를 보호하는데 충분하지 않으며, 우리는 OPRW 기법을 통해 파일시스템 복구 시점에 데이터와 노드의 순서 역전 문제를 해결하였다. 그러므로 fsync() 과정에서 데이터와 노드 블록의 쓰기 순서는 보장하지 않아도 무방하다. 즉 그림 5.2a에서 보이는 것처럼 데이터의 DMA 전송을 기다리지 않고 즉

시 노드의 쓰기 요청을 전송하는 것이 가능하다. 결과적으로 본 논문에서 제안하는 기법은 일관성을 확보할 수 있을 뿐만 아니라 성능 역시 개선할 수 있다.

5.2.2 PLP 지원 장치를 위한 `nobarrier` 모드

POSIX 모드와 마찬가지로, `nobarrier` 모드에서도 OPRW 기법을 사용하면 노드 블록을 쓰기 전에 데이터 블록의 DMA 전송을 기다리지 않아도 된다. 반면 POSIX 모드와는 다르게, 영속성을 보장하기 위한 플러시 명령을 수행하지 않기 때문에 DMA 전송이 fsync() 수행 시간에서 큰 부분을 차지하게 된다. 결과적으로 그림 5.2b와 같이 호스트와 저장장치가 더욱더 병렬적으로 동작할 수 있다. 이를 통해 CPU와 I/O의 활용도가 증대되어 저장장치의 내부 병렬성을 효과적으로 활용할 수 있다.

5.3 ZNS 에뮬레이터

문제 재현 및 성능 평가를 위해 NVMeVirt[19] 에뮬레이터를 기반으로 충돌 테스트와 쓰기 버퍼 모델을 구현하였다.

5.3.1 충돌 테스트 모델

NVMeVirt는 휘발성 메모리에서만 동작하는 에뮬레이터이기 때문에 충돌을 모사하고자 전원을 재공급시키게 되면 파일시스템을 포함한 기존 데이터들이 사라지게 된다. NVMeVirt의 모든 데이터를 파일 형태로 저장하면 이를 해결할 수 있지만, 복잡도가 높아지고 재부팅으로 인한 테스트 시간이 증가하는 문제가 있다.

따라서 기존 파일시스템의 연결(mount)과 해제(unmount) 기능을 활용하여 파일 저장과 재부팅 없이 충돌 테스트가 가능하게 하였다. 그림 5.3은 시간에 따른 충돌 테스트의 수행 과정과 NVMeVirt의 상태 변화를 보여준다. 특정 시점에 시스템 충돌이 발생하면 NVMeVirt는 읽기 전용 모드로 바뀌고, 그 이후부터는 쓰기 요청이

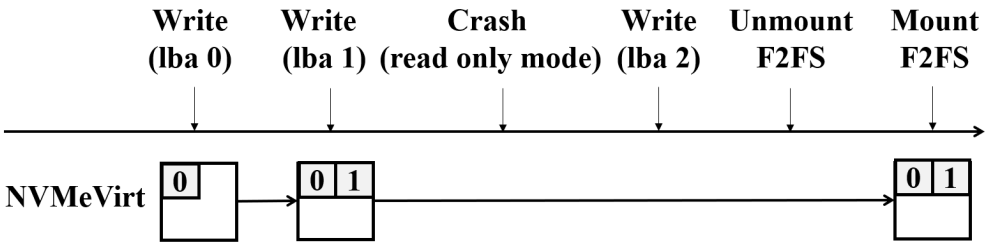


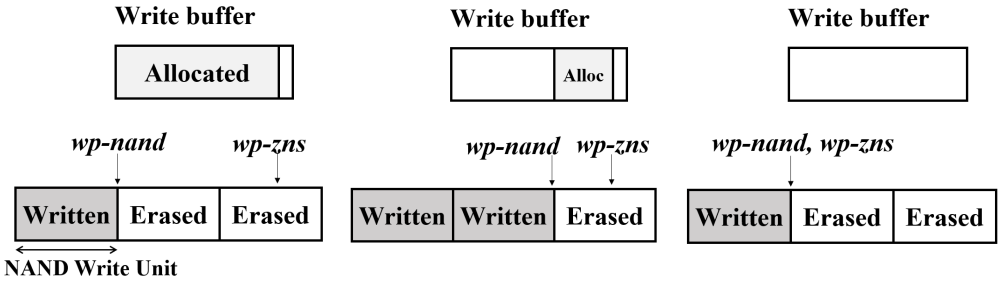
그림 5.3: 충돌 테스트 과정

들어와도 내부의 상태를 변경하지 않는다. 다만 시스템이 멈추지 않고 파일시스템을 성공적으로 해제하기 위해, 시스템 충돌 이후의 모든 쓰기 명령은 잘 처리된 것처럼 호스트로 응답한다. 이 상태에서 파일시스템의 해제와 연결을 순차적으로 수행하면 파일시스템은 충돌 직전의 데이터를 가지고 복구를 진행할 것이다. 이 과정을 반복하면 충돌 테스트를 재부팅 없이 빠르게 진행할 수 있다.

5.3.2 쓰기 버퍼 모델

우리는 쓰기 버퍼를 모사하기 위해 간단한 토큰(token) 방식을 채택하였다. 쓰기 요청이 오면 버퍼를 할당하기 위해 쓰기 요청의 크기만큼 토큰을 소진한다. 쓰기 요청의 지연시간에는 DMA 전송 시간과 약간의 내부 오버헤드만 포함한다. 낸드 플래시의 쓰기 단위만큼 버퍼에 데이터가 모이면 낸드 쓰기를 시작하고, 해당 동작이 완료되면 토큰을 증가시킨다. 만약 쓰기 버퍼의 공간이 부족하여 버퍼를 할당하지 못한다면 쓰기 버퍼가 충분해질 때까지 쓰기 요청을 수행하지 않고 기다린다. 이를 통해 SSD의 쓰기 버퍼 기능과 일정 단위만큼 모아서 써야 하는 낸드 플래시의 쓰기 동작을 모사할 수 있다. 또한 성능 관점에서 쓰기 버퍼에 의한 짧은 지연시간과 낸드 플래시에 의해 병목이 된 대역폭을 실제 SSD와 유사하게 만들 수 있다.

그림 5.4는 쓰기 버퍼의 구체적인 구현을 보여준다. 우리는 ZNS를 구동하기 위한 기존 *wp-zns*와는 별도로, 실제 낸드 플래시에 써진 정도를 나타내는 *wp-nand*를 추가하였다. 쓰기 요청이 들어오면 NVMeVirt는 버퍼링만 수행한 후 그림 5.4a와



(a) 버퍼링 완료 후 상태 (b) 낸드 쓰기 완료 후 상태 (c) 낸드 쓰기 완료 전 충돌 발생 시 이후 복구된 상태

그림 5.4: 쓰기 버퍼 및 쓰기 포인터의 상태 변화

같이 *wp-zns*를 증가시킨다. 쓰기 버퍼 내 데이터가 낸드 쓰기 단위만큼 쌓이면 낸드 쓰기를 시작하고, 해당 동작이 완료된 이후에 그림 5.4b와 같이 *wp-nand*를 증가시킨다. 또한 기존 SSD의 주기적인 내부 플러시를 모사하기 위해 일정 주기마다 *wp-zns*를 *wp-nand*에 복사한다. 그림 5.4c는 낸드 쓰기가 완료되기 전 충돌이 발생한 경우를 보여준다. 충돌 이후, *wp-nand*와 *wp-zns* 사이의 데이터는 손실되어야 한다. 따라서 모든 존의 *wp-nand*를 *wp-zns*에 복사하고 사이의 데이터를 초기화하여 충돌 이전의 낸드 플래시 상태로 복구한다.

Algorithm 1 데이터 일관성 검사

```
skip_list ← ∅
for each node ∈ roll forward target do
    inode ← find_inode(node)
    if inode ∈ skip_list then
        discard(node)
        continue
    end if
    for i = 0, 1, ..., N_ptrs − 1 do
        wp ← WP_Table[zone(pointer[i])]
        if pointer[i] < wp then                    ▷ Valid
            continue
        else                                       ▷ Invalid
            skip_list ← skip_list ∪ {inode}
            discard(node)
            break
        end if
    end for
end for
```

제 6 장 실험 결과 및 분석

표 6.1: 실험 환경

Processor	Intel Xeon Silver 4116 2.10 GHz
Memory	32GiB DRAM
OS	Ubuntu 18.04.6 LTS (kernel v5.14.4)
Storage	Western digital ZN540 4TB, NVMeVirt (emulator)

6.1 실험 환경

평가는 표 6.1에 상술한 바와 같이 Intel Xeon Silver 4116 프로세서, 32GiB 메모리가 장착된 서버와 5.14.4 버전의 커널에서 진행하였다. 저장장치로는 ZN540 4TB[1]를 사용하였다. ZN540은 PLP를 지원하므로 DMA 전송만으로 데이터의 영속성을 보장할 수 있다. 따라서 PLP가 없는 ZNS SSD를 평가하기 위해 NVMeVirt 에뮬레이터를 이용하였다. ZN540과 동등한 성능을 보이도록 NVMeVirt의 성능과 관련된 파라미터들을 조정하였고, 낸드 플래시의 쓰기 시간을 기반으로 플러시 지연시간을 적용하여 PLP가 없는 ZN540을 모사하였다.

우리는 최신 버전의 커널(6.3.3)에서 `strict` 모드의 F2FS가 ZNS SSD와 함께 실행될 때, 노드 블록 앞에 추가적인 플러시 명령을 삽입하지 않는 점을 인지하였다. 이 변경사항은 명령 재정렬로 인해 임의 쓰기가 발생하는 것을 피하고자 적용된 것으로 알려져 있다[18]. 하지만 플러시 동작 없이는 여전히 데이터 일관성 문제가 발생한다. 따라서 `strict` 모드의 올바른 성능 측정을 위해 데이터의 DMA 전송 후 플러시 명령을 삽입하였다.

	Total	Failed
F2FS-posix	1000	30
F2FS-strict	1000	0
OPRW-posix	1000	0

표 6.2: 데이터 일관성 실험 결과

6.2 데이터 일관성 실험

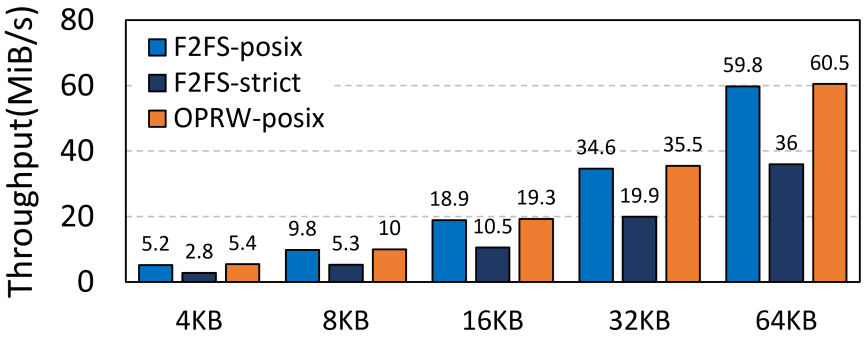
우리는 OPRW의 정합성을 검증하기 위해 PLP가 없는 NVMeVirt 환경에서 데이터 일관성 실험을 진행하였다. 데이터 일관성을 검증하는 순서는 다음과 같다. 특정 패턴으로 파일을 채운 후 동일한 패턴으로 쓰기 요청을 내린다. 이후, 임의의 시점에 시스템 충돌을 발생시키고 파일시스템을 다시 연결한다. 파일시스템이 복구되면 해당 파일을 읽어 데이터가 초기 패턴과 다른지 확인한다. 만약 파일에 다른 패턴이 포함되어 있다면 데이터 일관성이 훼손된 것으로 판단한다.

표 6.2는 데이터 일관성 실험 결과이다. 기존 F2FS의 경우 3% 실패하였는데, 이는 순차적인 DMA 전송만으로는 데이터 일관성을 보장하지 못한다는 것을 의미한다. 반면 기존 해결법인 `strict` 모드와 본 논문에서 제안한 OPRW 기법을 사용하면 데이터 손실 없이 파일을 복구할 수 있었다.

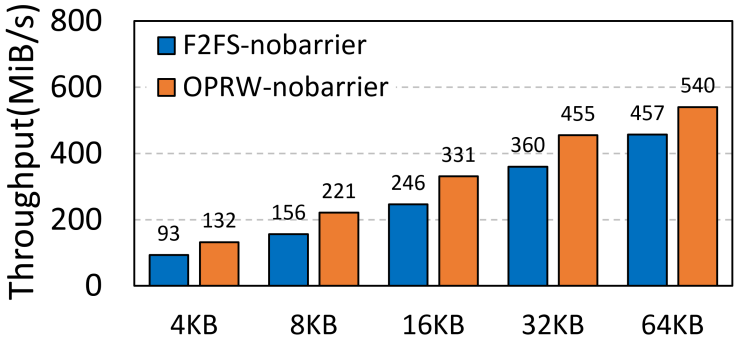
6.3 마이크로 벤치마크

그림 6.1은 FIO[7]를 사용하여 OPRW 기법의 성능 개선을 평가한 결과이다. fsync()를 동반한 임의 쓰기를 수행하였고, 단일 쓰레드에서 쓰기 요청의 크기를 변경해가며 성능을 측정하였다.

그림 6.1a는 PLP가 지원되지 않는 NVMeVirt에서 측정한 결과이다. fsync()의 처리 속도가 매우 느린 것을 확인할 수 있는데, 이는 F2FS와 OPRW 모두 데이터



(a) NVMeVirt (PLP 미지원)

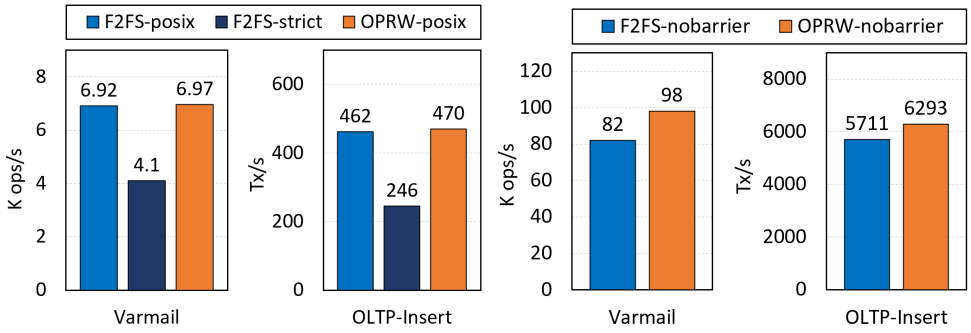


(b) ZN540 (PLP 지원)

그림 6.1: 마이크로 벤치마크 결과

의 영속성을 보장하기 위해 플러시 동작을 수행하기 때문이다. 기존의 해결책인 `strict` 모드는 POSIX 모드보다 44% 정도 성능이 하락하는데, 이것은 데이터 일관성을 유지하기 위한 추가적인 플러시 동작의 비용이 매우 높다는 것을 보여준다. 반면 OPRW는 데이터 일관성을 보장하면서도 기존 F2FS 대비 성능이 하락하지 않는다. 다만 OPRW가 상당한 성능 향상을 보여주지 못하는 이유는 fsync() 실행 시간에서 DMA 전송과 소프트웨어 오버헤드가 차지하는 비중이 플러시 동작에 비해 매우 작기 때문이다.

OPRW 기법은 PLP를 지원하는 SSD에서 더 큰 성능 개선을 보인다. PLP 지원 장치에서는 플러시 동작이 생략되어 fsync() 실행 중 다른 요소들의 비중이 증가하기



(a) NVMeVirt (PLP 미지원) (b) ZN540 (PLP 지원)

그림 6.2: 매크로 벤치마크 결과

때문이다. 결과적으로 그림 6.1b에서 보이는 것처럼 OPRW 기법을 사용하면 기존 F2FS보다 18% ∼ 42% 성능이 향상된다.

6.4 매크로 벤치마크

우리는 OPRW 기법을 fsync() 비중이 높은 워크로드로 잘 알려진 Varmail[5]과 OLTP-Insert[4]에서 단일 쓰레드를 사용하여 평가하였다. 그림 6.2를 보면 OPRW 기법의 개선 효과가 실제 워크로드에서도 유의미하게 나타나는 것을 볼 수 있다. PLP를 지원하는 장치의 경우 Varmail과 OLTP-Insert 각각 1.2배, 1.1배 향상되는데, Varmail에서 성능 향상이 더 높게 나타나는 것은 fsync()의 비중이 더 높기 때문이다[24].

우리는 Varmail 워크로드에서 쓰레드 증가에 따른 확장성을 평가하였다. 그림 6.3을 보면 쓰레드의 수가 적을 때는 의미 있는 수준의 성능 향상이 일어나지만, 쓰레드가 늘어날수록 OPRW의 성능 개선 정도가 줄어드는 모습을 볼 수 있다. 이는 세그먼트에 순차적으로 이어 쓰는 F2FS 구조상 쓰레드 간의 락 획득 경쟁으로 인해 확장성에 한계를 가지고 있기 때문이다[22]. 또한 ZNS의 요구조건을 준수하기 위해

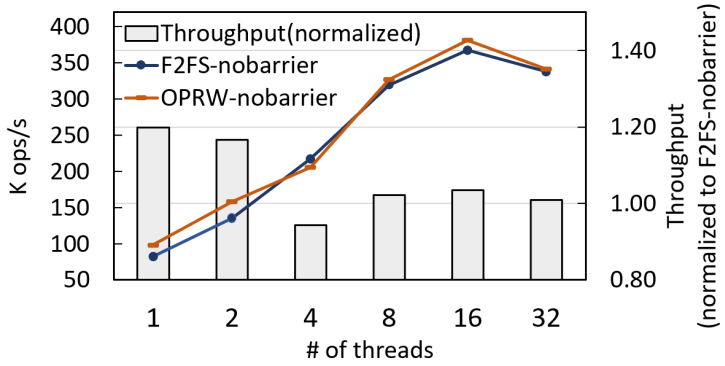


그림 6.3: 확장성 실험 결과

존 내 하나의 쓰기 명령만 허용하는 데드라인 스케줄러의 동작도 확장성을 저해한다. 이러한 확장성 문제들이 개선되면 OPRW 기법이 다중쓰레드 환경에서 더 의미있는 효과를 발휘할 것으로 기대한다.

6.5 복구 시간

우리는 데이터 일관성 검사가 파일시스템 복구 시간에 미치는 영향을 평가하기 위해 전방향 복구의 소요 시간을 측정하였다. 표 6.3은 Varmail 워크로드 동작 중 충돌이 발생한 상황에서 복구한 노드의 양에 따른 복구 시간을 보여준다. 체크포인트 되지 않은 노드가 많아질수록 전방향 복구를 수행해야 하는 양이 늘어나므로 복구 시간이 증가한다. 데이터 일관성 검사가 파일시스템 복구 과정에 포함되면 수행 시간이 203ms ∼ 342ms 늘어나는데, 전체 파일시스템 복구 시간이 수 초 이상 소요되는 것에 비교하면 상대적으로 미미한 수치이다. 복구 시간이 크게 늘지 않는 것은 OPRW 기법에서 어떠한 I/O 동작 없이 메모리 접근과 간단한 비교 연산만 수행하기 때문에 가능하다. 또한 일반적인 상황에서는 주기적인 체크포인트가 진행되기 때문에 시스템 복구 시점에 전방향 복구를 진행하는 양은 한정적이다. 즉 쓰기 포인터를 이용한 데이터 일관성 검사는 파일시스템 복구 시간을 크게 저해하지 않으면서도

# of scanned nodes	40791	50962	57500
F2FS (ms)	4177	7307	9751
OPRW (ms)	4380	7584	10093
Difference (ms)	203	277	342

표 6.3: 전방향 복구 소요 시간

파일시스템 일관성 수준을 높일 수 있는 점에서 실용적이고 효과적이다.

제 7 장 결론

본 논문은 F2FS의 fsync() 동작이 데이터 일관성을 보존하지 못하는 문제를 지적하고 실험을 통해 재현하였다. 이러한 문제를 해결하기 위해 차세대 인터페이스로 부상하고 있는 ZNS가 제공하는 쓰기 포인터에 주목하였다. 우리는 쓰기 포인터를 통해 데이터 블록의 영속 여부를 빠르게 판단할 수 있다는 점을 활용하여 데이터 손실 없는 효과적인 파일시스템 복구 기법을 제안하였다. 더불어 fsync() 성능의 병목이었던 불필요한 동작을 제거하여 성능을 개선하였다. 결과적으로 실제 워크로드에서 쓰기 성능이 최대 1.2배 향상됨을 확인하였다.

참고 문헌

[1] Western Digital DC ZN540. hhttps://www.westerndigital.com/ko-kr/products/internal-drives/data-center-drives/ultrastar-dc-zn540-nvme-ssd#0TS2094.

[2] Write Ordering Control. https://zonedstorage.io/docs/linux/sched.

[3] Zoned NameSpaces (ZNS) SSDs. https://zonedstorage.io.

[4] SysBench manual. https://imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf, 2004.

[5] Filebench. http://www.nfsv4bat.org/Documents/nasconf/2004/filebench, 2005.

[6] The end of block barriers. https://lwn.net/Articles/400541/, 2010.

[7] FIO: Flexible I/O Tester. https://github.com/axboe/fio, 2014.

[8] JEDEC Standard JESD84-B51. Embedded MultiMedia Card (eMMC) Electrical Standard (5.1), 2015.

[9] Force keeping write barrier for strict mode. https://lkml.org/lkml/2021/6/1/350, 2021.

[10] NVMe Specification. https://nvmexpress.org/wp-content/uploads/NVMe-NVM-Express-2.0a-2021.07.26-Ratified.pdf, 2021.

[11] Zoned Namespace Command Set Specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-Zoned-Namespace-Command-Set-Specification-1.1c-2022.10.03-Ratified.pdf, 2022.

[12] M. Bjørling, A. Aghayev, H. Holmberg, A. Ramesh, D. L. Moal, G. R. Ganger, and G. Amvrosiadis. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, Virtual, July 2021.

[13] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, February–March 2017.

[14] Yun Sheng Chang and Ren Shuo Liu. OPTR: Order-Preserving Translation and Recovery Design for SSDs with a Standard Block Device Interface. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[15] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, November 2013.

[16] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, San Jose, California, USA, February 2012.

[17] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The Multi-streamed Solid-State Drive. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, Philadelphia, PA, June 2014.

[18] Jaegeuk Kim. f2fs: use flush command instead of FUA for zoned device. https://lore.kernel.org/lkml/20220419215703.1271395-1-jaegeuk@kernel.org/, 2022.

[19] Sang-Hoon Kim, Jaehoon Shim, Euidong Lee, Seongyeop Jeong, Ilkueon Kang, and Jin-Soo Kim. Nvmevirt: A versatile software-defined virtual nvme device. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, February 2023.

[20] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *Proceedings of the 13rd USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, February 2015.

[21] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Write Dependency Disentanglement with HORAE. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, November 2020.

[22] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Max: A Multicore-Accelerated File System for Flash Storage. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, Virtual, February 2021.

[23] Xiaojian Liao, Youyou Lu, Zhe Yang, and Jiwu Shu. Crash Consistent Non-Volatile Memory Express. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual, April 2021.

[24] Joontaek Oh, Seung Won Yoo, Hojin Nam, Changwoo Min, and Youjip Won. CJFS: Concurrent Journaling for Better Scalability. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, February 2023.

[25] Daejun Park and Dongkun Shin. iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[26] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-Enabled IO Stack for Flash Storage. In *Proceedings of the 16st USENIX Conference on File and Storage Technologies (FAST)*, Oakland, California, USA, February 2018.

[27] Jeseong Yeon, Minseong Jeong, Sungjin Lee, and Eunji Lee. RFLUSH: Rethink the Flush. In *Proceedings of the 16st USENIX Conference on File and Storage Technologies (FAST)*, Oakland, California, USA, February 2018.

[28] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. Understanding the Robustness of SSDs under Power Fault. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, February 2013.

ABSTRACT

An Efficient Order-Preserving Recovery for F2FS with ZNS SSD

Euidong Lee
Department of Computer Science & Engineering
The Graduate School
Seoul National University

Storage devices use write buffers to improve performance, where multiple write requests are processed in parallel and completed in a random order. This may result in data loss in the event of a sudden failure. Therefore, Linux filesystems provide the fsync() system call to prevent data loss and ensure write order. However, the fsync() system call in F2FS, one of the most popular filesystems, is inefficient and insufficient for guaranteeing data consistency.

We propose a new technique called Order-Preserving Recovery by Write pointer (OPRW) to ensure data consistency during the filesystem recovery. OPRW utilizes the write pointer in Zoned NameSpace (ZNS) SSDs to efficiently determine the persistence of data without the need for I/O operations. This approach allows OPRW to provide a higher level of consistency and performance improvement for fsync(), while minimizing the impact on recovery time. As a result, our solution improves performance by up to 1.2x on realistic workloads.

Keywords: F2FS, ZNS, SSD, Filesystem

Student Number: 2021-25814