현대 입출력 스택에서 쓰기 순서 효율적 보장에 대한 고찰

한국과학기술원 ▌김지은·유승원·원유집*

1. 서 론

저장장치에 일련의 데이터 블록들을 특정 순서대로 기록하는 것을 “쓰기 순서의 보장”이라 한다. 쓰기 순서의 보장은 데이터베이스 트랜잭션의 내구성과 원자성을 보장하기 위해서 뿐만 아니라[1][2][3], 파일 시스템 저널링[4][5][6][7], 소프트 업데이트[8][9] 그리고 카피 온 라이트 (CoW) 및 로그 구조 파일 시스템에서도[2][10][11][12] 중요한 역할을 한다.

현대 운영체제의 입출력 스택은 순서를 보장하는 기재가 존재하지 않는다. 입출력 스케줄러, 명령 큐 관리자, 그리고 쓰기 캐시 관리자 등 각 계층은 자신의 방식으로 입출력 요청을 재배열한다. 이로 인해, 쓰기 순서를 보장하기 위해서 매우 비효율적인 방식이 사용된다. 즉, 선행 데이터 블록을 안전히 저장장치에 저장한 후에야 다음 데이터 블록의 쓰기가 시작된다. 이 매카니즘을 전송 및 플러시 (Transfer-and-Flush)라고 한다[13]. 전송 및 플러시 기반의 순서보장이 입출력 병목의 핵심 원인이다. 스토리지 성능이 좋아지면서, 전송 및 플러시로 인한 성능 병목이 더욱 심해지고 있다.

스토리지는 멀티채널/웨이 컨트롤러[14][15], 대용량 스토리지 캐시[16], 그리고 깊은 명령 대기열[7][17][18] 등 병렬성을 증가시켜 성능 개선을 도모하여왔다. 그러나 플래시 스토리지가 더 높은 병렬성을 지원하고 집적도가 증가할수록, 전송 및 플러시 방식의 오버헤드는 더욱 커진다. 그림 1은 이러한 경향을 잘 보여준다. 이 그림은 일반적인 버퍼링 된 쓰기와 fdatasync()를 이용한 순서보장 쓰기 간의 성능 비율을 다양한 SSD에서 측정한 결과를 나타낸다. 고성능 SSD일수록 순서를 보장할 경우 성능 저하가 더욱 심화하는 것을

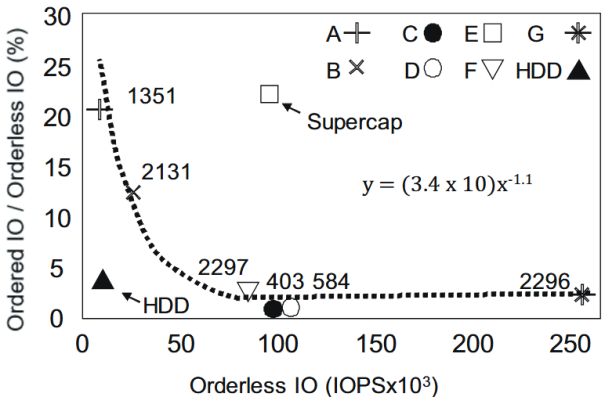

그림 1 Ordered write vs. Orderless write, HDD를 제외하면 모두 플래시 스토리지이다. A: (1채널)/eMMC5.0, B: (1채널)/UFS2.0, C: (8채널)/SATA3.0, D: (8채널)/NVMe, E: (8채널)/SATA3.0 (슈퍼캡 포함), F: (8채널)/PCIe, G: (32채널) 플래시 배열, 각 지점 옆의 숫자는 write() 후 fdatasync()를 수행한 IOPS를 나타냄.

확인할 수 있다. 예를 들어, 스마트폰용 단일 채널 SSD(SSD A)에서는 순서보장 쓰기의 성능이 순서 없는 쓰기의 20% 수준이지만, 32채널 플래시 어레이 (SSD G)에서는 이 비율이 1%까지 감소한다(2,296 IOPS vs. 230K IOPS).

스토리지 순서보장의 오버헤드를 줄이기 위해 다양한 연구가 진행됐다. 플래시 스토리지의 비휘발성 쓰기 캐시[19], EXT4 파일 시스템의 no-barrier 마운트 옵션[20], 그리고 트랜잭션 체크섬[21][22][23]과 같은 기법이 활용되고 있다. 또한, 스토리지 순서보장 오버헤드를 줄이기 위한 보다 근본적인 접근 방식도 제안되었다[24][25][26]. 그러나 이러한 기법들 역시 순서보장을 위해 여전히 전송 및 플러시 방식에 의존한다.

본 원고에서는 캐시 배리어 명령어를 사용하여 입출력 스택에서 순서를 보장하는 기법에 대해 살펴보기로 한다.

* 종신회원

† 본 원고는 한국연구재단 (NRF-2020R1A2C3008525) 그리고 IITP (RS-202400459026)의 지원을 받아 작성되었음.

2. 연구 동향

2.1 현대 입출력 스택의 쓰기 순서보장

응용 프로그램이 원하는 순서로 데이터 블록들을 저장하려면, 입출력 스택의 각 계층이 명시된 순서를 유지하며 요청을 처리해야 한다. 그러나 입출력 스택의 구조적 특성상, 한 계층에서 명시한 순서를 다른 계층이 준수하도록 강제하는 것이 불가능하다. 때문에, 쓰기 순서의 보장이 필요한 경우 응용 프로그램은 선행 요청의 데이터 블록이 저장장치에 안전히 저장된 후에 다음 요청을 디스패치 하는 방식을 사용한다. 하지만 이 방식은 대용량 쓰기 캐시, 멀티채널/웨이 컨트롤러 등 현대 저장장치가 제공하는 병렬 처리 기능을 효과적으로 활용하지 못한다.

2.2 캐시 배리어 명령어

캐시 배리어 (cache barrier) 명령어는 eMMC 4.4 [27]에 도입된 커맨드로, 데이터 블록들의 저장 순서를 제어하는 데 사용된다. 캐시 배리어 명령어가 저장장치 컨트롤러에 전달되면, 컨트롤러는 캐시 배리어 이전에 전송된 데이터 블록들이 캐시 배리어 명령어 이후에 전송된 데이터 블록들보다 먼저 영구 저장됨을 보장한다. 이를 통해, 호스트는 명시적으로 캐시 플러시를 수행하지 않고도 저장 순서를 제어할 수 있다.

2.3 다중 큐 기반 입출력 스택

리눅스는 4. x 커널부터 블록 입출력 계층에 다중 요청 큐 (multi-queue)를 도입하였다. 또한, 다중 커맨드 큐 (multi-command queue)를 지원하는 NVM Express (NVMe) 인터페이스 기반 저장장치가 여러 서버와 엔터프라이즈 환경에서 널리 사용되고 있다. 리눅스의 블록 입출력 계층에서는 CPU와 다중 큐가 결합하여 입출력 요청을 처리한다. 각 CPU마다 요청 큐가 할당되며, 각 요청 큐는 라운드 로빈 (round-robin) 방식이나 모듈러 연산을 통해 커맨드 큐와 연결된다. CPU에서 실행되는 스레드가 생성한 입출력 요청은 해당 CPU에 할당된 요청 큐에 삽입되며, 블록 계층은 요청 큐와 연결된 서브미션 큐를 통해 입출력 요청을 디스패치 한다.

입출력 스택의 각 계층이 상위 계층에서 명시한 쓰기 순서를 보장할 수 있으면, 전체 스택이 입출력 순서를 보장할 수 있게 된다. 이를 순서보장형 입출력 스택이라 한다. 순서보장형 입출력 스택은 순서보장형 블록 계층, 순서보장형 스토리지로 구성된다.

3. 순서보장형 블록 계층

순서보장형 블록 계층은 파일 시스템으로부터 입출력을 전달받아 스토리지 장치로 디스패치 하는 역할을 한다. 순서보장형 블록 계층은 요청 큐(request queue)와 입출력 스케줄러로 구성된다. 상위 계층에서 전달된 입출력 요청을 요청 큐에 삽입되며, 이후 스케줄러가 정해진 스케줄링 정책 (예: CFQ, BFQ 등)에 따라 요청을 처리한다.

리눅스 3. x 커널까지는 블록 입출력 계층에 단일 요청 큐만 존재하여, 시스템의 모든 입출력 요청이 하나의 큐에 등록되었다. 그러나 4. x 커널부터는 단일 큐에서 발생하는 락 경합과 그로 인한 오버헤드를 줄이기 위해 다중 큐(multi-queue) 구조가 도입되었다. 이 구조에서는 각 CPU 코어마다 요청 큐를 할당하며, 해당 코어에서 실행되는 애플리케이션이나 스레드가 생성하는 요청을 해당 큐에 삽입하는 방식으로 동작한다.

3.1 단일 큐 기반 순서보장형 블록 계층

순서보장형 블록 입출력 계층의 핵심 역할은 애플리케이션이나 파일 시스템에서 전달받은 순서 제약 정보를 기반으로, 쓰기 요청을 해당 제약에 맞춰 디스패치 하는 것이다. 이를 통해, 입출력 요청이 발행된 순서와 실제로 디스패치 되는 순서를 일치시킬 수 있다. 순서보장형 블록 입출력 계층은 순서보장형 파일 시스템과 상호작용하며 동작한다. 전송 및 플러시 오버헤드를 제거하기 위해, 순서보장형 파일 시스템은 전송 및 플러시를 캐시 배리어 커맨드로 대체한다. 순서보장형 블록 입출력 계층은 특정 파일 시스템에 종속되지 않고 다양한 파일 시스템과 결합하여 사용할 수 있다. 이를 위해, 파일 시스템은 캐시 배리어를 포함해야 하는 경우, 캐시 배리어에 선행되는 쓰기 요청에 이를 나타내는 플래그를 추가하여 전달하는 방식을 따른다.

에포크란 순서가 변경되어 기록되어도 무방한 쓰기 요청들의 집합이다. 순서보장형 입출력 스택은 파일 시스템으로부터 전달받은 입출력 요청을 에포크 단위로 구성하고, 순서 제약에 맞게 이를 순서대로 디스패치 한다.

블록 계층에서는 선입선출방식(NO-OP)[28] 이나 에포크 기반 입출력 스케줄러(Epoch-Based IO Scheduler)를 사용할 수 있다[13]. 에포크 기반 입출력 스케줄링은 발행 순서와 디스패치 순서 간의 부분 순서를 유지한다. 더욱 효율적인 스케줄링을 위해, 입출력 요청

을 순서보장 요청(order-preserving request)과 순서가 없는 요청(orderless request)으로 구분할 수 있다.

3.2 다중 큐 기반 순서보장형 블록 계층

블록 계층이 단일 큐 구조에서 다중 큐 구조로 변경되면서, 캐시 배리어 명령어만으로 저장 순서를 보장하는 것이 어려워졌다. 이는 순서 제약이 있는 쓰기 요청들이 여러 큐에 분산될 경우, 큐 간의 순서를 보장할 수 없기 때문이다. 각 큐 내에서는 저장 순서가 유지되지만, 큐 간 저장 순서는 보장되지 않으며, 스토리지 컨트롤러가 요청을 디스패치한 순서와 실제 서비스되는 순서가 달라질 가능성이 있다. 이를 큐 간 쓰기 순서(Inter-Queue Storage Order)라 한다[29] (그림 2).

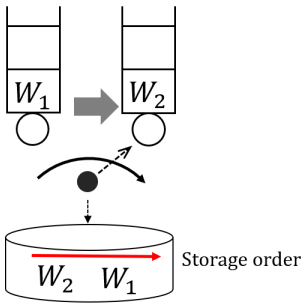

그림 2 큐 간 쓰기 순서, 쓰기 요청 W1이 W2보다 먼저 저장장치에 기록되어야 하지만, 다중 큐 구조로 인해 쓰기 요청 W2가 먼저 기록된다.

저장 순서보장을 최적화하는 다양한 기법이 연구됐다[13][29][30][31][32][33][44]. 그러나[33][44]의 연구는 단일 큐 저장장치를 기반으로 수행되어 큐 간 쓰기 순서를 보장해야 하는 상황을 고려하지 않았기 때문에 다중 큐 저장장치에 적용하기 어렵다. 다중 큐 저장장치에서 저장 순서보장을 최적화한 연구도 존재한다[30][31][32][34]. 하지만 이러한 연구들은 순서 의존성이 있는 요청들이 여러 큐로 분산되지 않도록 CPU에 스레드를 고정하거나[30], 특정 하드웨어 자원의 사용을 필수적으로 요구하거나[30][31][34], 모바일 스토리지 환경에서만 적용할 수 있다는[32] 한계를 가진다. 이처럼 현대 입출력 스택의 다중 큐 구조는 쓰기 순서보장과 관련하여 새로운 도전 과제를 제시한다. [29]에서는 큐 간 쓰기 순서를 보장해야 할 때 발생할 수 있는 문제점을 분석하고, 이를 해결하여 저장 순서보장을 최적화하는 다중 큐 기반 순서보장형 입출력 스택을 소개한다.

다중 큐 기반 순서보장형 블록 계층은 쓰기 요청을 생성된 스레드에 따라 여러 개의 스트림(stream)으로 분류하고, 각 스트림 내에서 캐시 배리어를 기준으로 에포크(epoch)를 구성한다. 블록 계층은 각 쓰기 스트림과 해당 에포크에 식별자를 부여하고, 입출력 요청에도 스트림과 에포크 ID를 포함하여 전달한다. 이를 통해 큐 간 순서가 변경되더라도 저장 순서를 보장할 수 있으며, FTL(Flash Translation Layer)와 협력하여 동작한다.

같은 스레드에서 생성된 쓰기 요청들이 여러 요청 큐에 분산되는 경우, 스트림 내 저장 순서(Intra-stream storage order)를 보장해야 한다. 로드 밸런싱이나 워크 스틸링으로 인해 스레드가 다른 코어에서 실행되면, 쓰기 요청들이 여러 큐로 나뉘는 에포크 분리(epoch split)가 발생할 수 있다.

에포크 분리가 발생하면, 캐시 배리어 명령어와 에포크 내 쓰기 요청 순서가 재정렬될 가능성이 있어, FTL이 저장 순서를 제대로 보장하지 못할 수 있다. 특히, 캐시 배리어가 먼저 서비스되면, 해당 에포크에 속한 모든 요청이 도착하지 않았음에도 미리 구획되는 현상(pre-delimit)이 발생할 수 있다.

이를 방지하기 위해, 에포크 피닝(epoch pinning) 기법을 적용한다. 블록 계층이 캐시 배리어를 디스패치할 때까지 해당 스레드가 사용하는 요청 큐를 고정하는 방식이다. 기존 연구[30]에서는 스레드의 코어 이동을 금지하는 방식도 사용하지만, 에포크 내부 요청이 많으면 특정 CPU에 부하가 집중되는 문제가 발생할 수 있다. 반면, 에포크 피닝은 스레드가 실행 중인 코어가 아닌, 디스패치 하는 요청 큐를 고정하므로 CPU 로드 밸런싱에 영향을 주지 않는다.

서로 다른 스레드가 생성한 쓰기 요청 간의 순서를 보장해야 하는 경우, 이를 스트림 간 저장 순서(Inter-stream storage order)라 한다. 스레드가 다른 코어에서 실행되면, 순서 제약이 있는 쓰기 요청들이 각 스레드가 실행 중인 CPU에 할당된 큐에 등록되므로, 큐 간 쓰기 순서를 보장해야 하는 문제가 발생한다.

이 문제를 해결하기 위해 이중 스트림 쓰기(Dual-stream write) 기법을 적용한다. 이중 스트림 쓰기는 두 개의 스트림에 속하는 쓰기 요청을 의미하며, 블록 계층은 각 요청에 두 쌍의 스트림 및 에포크 식별자를 부여한다.

첫 번째 쌍은 쓰기 요청이 본래 속하는 스트림과 해당 에포크의 식별자이며, 두 번째 쌍은 쓰기 요청이 순서 제약을 가지는 스트림과 해당 에포크의 식별자이다. 첫 번째 쌍을 주 스트림, 에포크 식별자라 하고 두 번째 쌍을 보조 스트림, 에포크 식별자라 한다. 이

를 통해, 블록 계층은 서로 다른 두 스레드 간의 순서 의존성 정보를 스토리지로 전달하고, 스토리지의 FTL은 이 정보를 기반으로 두 스트림 간 저장 순서를 보장한다.

4. 순서보장형 Flash Translation Layer

4.1 개요

순서보장형 FTL은 앞서 소개한 다중 큐 기반 순서보장형 블록 계층과 협업하여 저장 순서를 보장한다. 순서보장형 FTL은 저장 순서를 유지하는 과정에서 데이터를 캐시에서 플래시 칩에 기록하는 작업과 매핑 테이블에서 논리-물리 페이지 주소를 업데이트하는 작업을 분리하여 수행한다. 데이터 플러시는 순서와 관계없이 즉시 수행되지만, 매핑 테이블 업데이트는 에포크 식별자 순서대로 진행된다. 이를 통해, 호스트에서 도착한 에포크들이 순서 없이 저장 장치로 전달되더라도, 선행 에포크가 모두 처리되기 전에 후속 에포크의 데이터 플러시가 가능해진다. 즉, 에포크를 직렬적으로 처리하지 않고 SSD 내부의 병렬성을 극대화할 수 있다. 반면, 순서보장형 FTL은 매핑 테이블 엔트리의 업데이트 순서를 에포크 순서대로 유지하도록 제어하여, 데이터 블록이 올바른 순서대로 기록될 수 있도록 보장한다.

4.2 순서 보장형 매핑 테이블 업데이트

순서보장형 FTL은 특정 에포크의 모든 쓰기 요청이 저장장치에 도착하고, 해당 데이터가 영구적으로 저장되며, 논리 페이지의 매핑 정보가 매핑 테이블에 반영되었을 때 해당 에포크가 영구적으로 저장된 것으로 판단한다. 쓰기 요청이 도착하면, FTL은 해당 요청의 스트림 및 에포크를 확인한 후, 선행 에포크가 영구적으로 저장되었는지 검사한다. 선행 에포크가 영구적으로 반영된 경우에만 해당 쓰기 요청의 매핑 정보를 테이블에 업데이트할 수 있다.

그러나 다중 큐 구조로 인해 선행 에포크가 아직 영구적으로 반영되지 않은 경우, 쓰기 요청의 데이터는 플래시 메모리에 저장되지만 L2P 매핑 정보는 보류된다(그림 3). 보류된 매핑 정보는 각 스트림 별로 관리되는 지연 매핑(delayed mapping) 자료구조에 저장되며, 해당 엔트리는 논리 페이지 주소, 물리 페이지 주소, 에포크 ID, sibling 포인터로 구성된다. 순서보장형 FTL은 주기적으로 지연 매핑을 확인하여, 선행 에포크가 영구적으로 반영된 경우에 한해 지연된 매핑 정보를 테이블에 반영한다.

지연 매핑의 sibling 포인터는 스트림 간 저장 순서를 보장하는 데 사용되며, 이중 스트림 쓰기의 매핑 정보가 지연될 경우 활용된다. 이중 스트림 쓰기가 아닌 경우, sibling 포인터는 NULL 값을 가진다. 반면, 이중 스트림 쓰기의 경우, 지연 매핑 정보가 주 스트림과 보조 스트림의 지연 매핑에 모두 추가되며, 각 sibling 포인터는 서로의 지연 매핑 엔트리를 가리킨다.

순서보장형 FTL은 지연 매핑 엔트리의 선행 에포크가 영구적으로 반영된 후, sibling 포인터가 NULL인 경우에만 매핑 테이블을 업데이트한다. 만약 sibling 포인터가 존재하는 경우, 지연 매핑 엔트리는 삭제하지만, 매핑 테이블을 업데이트하지 않는다. 이렇게 하면, 두 스트림 간 저장 순서를 보장해야 할 때, 두 스트림 모두에서 선행 에포크가 영구적으로 저장된 후에만 매핑 업데이트가 수행되도록 하며, 하나의 스트림만 준비된 경우 매핑이 조기에 업데이트되는 문제를 방지할 수 있다.

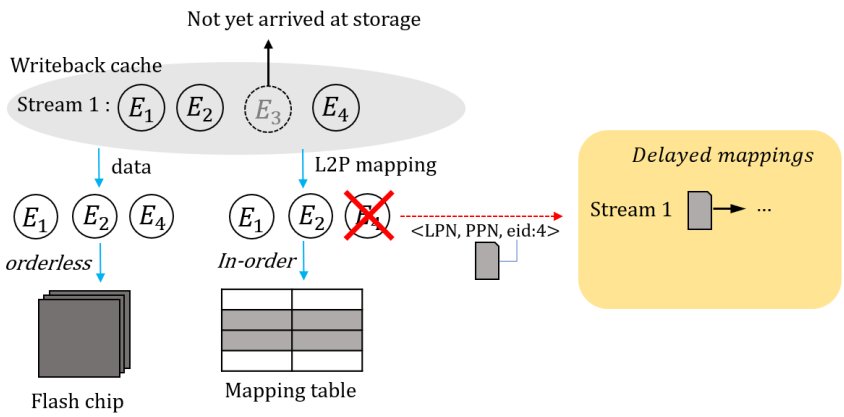

그림 3 순서보장형 FTL의 동작. E는 에포크를 나타낸다. 에포크 식별자 3이 아직 도착하지 않았으므로, 에포크 식별자 4에 해당하는 데이터는 플래시 칩에 기록되지만, 매핑 정보는 지연 매핑 자료구조에 저장된다.

5. 순서보장형 파일 시스템

5.1 프로그래밍 모델

순서보장형 입출력 스택은 동기화 인터페이스로 fsync(), fdatasync(), fbarrier(), fdatabarrier()를 제공한다. fbarrier()와 fdatabarrier()는 순서보장과 영속성 보장의 분리를 위한 새로운 동기화 인터페이스다. fbarrier()와 fdatabarrier()는 각각 fsync()와 fdatasync()와 같은 논리 블록 집합을 동기화하지만, 동기화 대상 블록들의 영속성을 보장하지 않고 리턴한다.

fdatabarrier()는 일종의 스토리지 배리어다. 응용프로그램은 쓰기 순서 보장이 필요한 두 write() 시스템콜 사이에 fdatabarrier()를 호출해 fdatabarrier() 호출 이전의 쓰기 요청으로 인한 갱신 사항이 fdatabarrier() 호출 이후의 쓰기 요청으로 인한 갱신 사항보다 먼저 디스크에 기록되는 것을 보장할 수 있다. fdatabarrier()의 동작은 메모리 배리어에서 mfence 명령어가 수행하는 역할과 동일하다[35]. 그림 4은 fdatabarrier() 기반 순서보장의 예제이다. 응용은 fdatabarrier()를 호출해 “Hello”가 “world” 이전에 디스크에 기록됨을 보장할 수 있다.

```
write(fileA, "Hello");
fdatabarrier(fileA);
write(fileA, "World");
```

그림 4 fdatabarrier() 사용 예제

5.2 저널링 파일시스템 – EXT4

EXT4의 기본 저널링 모드인 ordered 모드에서는 데이터 블록이 저널 트랜잭션보다 먼저 디스크에 기록된다. 그림 5은 EXT4에서 fsync()를 호출했을 때 각 계층의 디스크 쓰기 동작을 시간에 따라 표현한 그림이다. 파일 시스템은 갱신된 데이터 페이지 집합인 D로 구성된 쓰기 요청을 issue 한다. 쓰기 요청 issue 후, 응용 스레드는 쓰기 요청의 DMA 전송 완료를 기다린다. DMA 전송이 완료되면, 응용 스레드는 JBD 스레드를 깨우고 저널 커밋을 요청한다. JBD 스레드를 깨운 응용 스레드는 sleep 한다. JBD 스레드가 저널 커밋을 완료하면, fsync()는 리턴한다. 저널 트랜잭션은 보통 두 개의 쓰기 요청을 통해 커밋된다. 첫 번째 쓰기 요청은 저널 디스크립터 블록과 로그 블록이고, 두 번째 쓰기 요청은 저널 커밋 블록이다. 본문에서는 각각 쓰기 요청을 JD와 JC로 표시한다. 저널 커밋 시 보장되어야 하는 쓰기 순서는 두 가지로 (i) 트랜잭션 내의 JD가 JC보다 먼저 디스크에 기록되어야 하고, (ii) 트랜잭션 간 쓰기 순서가 보장되어야 한다. 두 쓰기 순서 중 하나라도 지켜지지 않으면 크래시 일관성이 보장되지 않는다[8][25]. JBD는 (i) 트랜잭션 내 쓰기 순서를 보장하기 위해 JD 쓰기 요청과 JC 쓰기 요청 사이에 전송 및 플러시(transfer-and-flush)를 대기하고, 트랜잭션 간 쓰기 순서를 보장하기 위해 JC의 영속성 보장 후 다음 트랜잭션의 저널 커밋을 시작한다.

저널 커밋은 두 개의 동작으로 구성된다. 각각 (i) JD와 JC 쓰기 커맨드 디스패치, (ii) JD와 JC의 영속성 보장이다. 순서보장과 영속성 보장 분리의 장점을 극대화하기 위해, [13]의 연구는 제어 동작(쓰기 요청 디스패치)과 데이터 동작을 분리하였다. 또, 각 동작에 별도의 스레드를 할당해 각 동작의 병렬수행이 가능하도록 하였다. [13]의 연구는 각 스레드를 커밋 스레드와 플러시 스레드로 정의하고, 상기 매카니즘을 듀얼 모드 저널링으로 칭한다. 듀얼 모드 저널링은 내구성 보장 모드와 순서보장 모드 두 가지를 지원한다. 커밋 스레드는 JD와 JC 쓰기 커맨드를 배리어 쓰기로 디스패치 해 JD와 JC 사이 쓰기 순서가 보장되도록 한다 (그림 6). JC 쓰기 커맨드를 디스패치한 커밋 쓰레드는 트랜잭션을 커밋팅 트랜잭션 리스트에 삽입하고 플러시 스레드를 깨운다. 플러시 스레드의 동작은

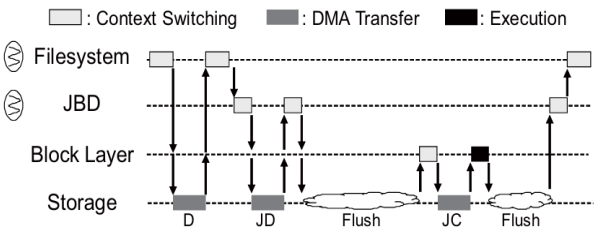

그림 5 fsync()의 DMA 전송, 캐시 플러시, 문맥 전환. ‘D’, ‘JD’, ‘JC’는 각 블록의 DMA 전송 시간이다. Flush는 플러시 요청 수행 시간이다.

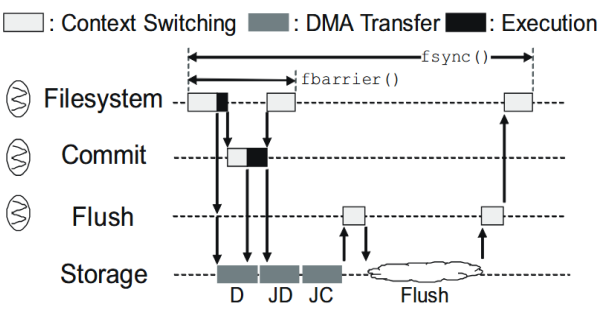

그림 6 듀얼 모드 저널링에서의 fsync()와 fbarrier()

저널 커밋이 fbarrier()에 의해 트리거된 경우와 fsync()에 의해 트리거된 경우로 나뉜다. 만약 저널 커밋이 fbarrier()에 의해 트리거된 경우, 플러시 쓰레드는 트랜잭션을 커밋팅 트랜잭션 리스트에서 제거한 후 리턴한다. 플러시 커맨드는 issue 하지 않고 리턴한다. 만약 저널 커밋이 fsync()에 의해 트리거된 경우, 플러시 쓰레드는 플러시 커맨드를 issue하고 캐시 플러시 완료를 기다린다. 그 후, 트랜잭션을 커밋팅 트랜잭션 리스트에서 제거한 후 리턴한다. 듀얼 스레드 구조는 (i) 순서보장과 내구성 보장을 분리할 수 있도록 하고 (ii) 트랜잭션 커밋을 병렬화 할 수 있다. 듀얼 모드 저널링 구조에서 여러 개의 커밋팅 트랜잭션이 존재할 수 있다. 실제로, 듀얼 모드 저널링 구조에서 동시에 커밋 중인 트랜잭션의 개수는 1개이다. 그 이유는 커밋팅 트랜잭션에 속한 페이지를 러닝 트랜잭션에 삽입해야 하는 경우 발생하는 트랜잭션 충돌 때문이다. 트랜잭션 충돌 발생 시, 충돌이 발생한 선행 트랜잭션 커밋 완료까지 후행 트랜잭션의 커밋을 시작할 수 없다. 상기 문제의 해결을 위해, [36]의 연구는 멀티 버전 쉐도우 페이징을 제안한다.

멀티 버전 쉐도우 페이징[36]에서는 커밋 스레드가 저널 커밋 시작 시, 트랜잭션 내 모든 페이지들의 쉐도우 페이지를 생성한다. DMA 전송에 원본 페이지 캐시 엔트리가 아닌 쉐도우 페이지가 사용된다. 따라서 추후 파일 연산이 원본 페이지 캐시 엔트리를 갱신할 수 있다. 멀티 버전 쉐도우 페이징은 저널 커밋 시 트랜잭션 내 모든 페이지들의 쉐도우 페이지를 생성한다. 따라서 각 메타데이터 페이지 캐시 엔트리들은 현재 커밋 중인 트랜잭션의 개수만큼 쉐도우 페이지가 존재할 수 있다.

그림 7는 트랜잭션 충돌 시의 멀티 버전 쉐도우 페이징 동작의 예제다. tx1,tx2,tx3는 순서대로 생성된다. tx1은 러닝 트랜잭션, tx2와 tx3는 커밋팅 트랜잭션이다. tx1에는 P1, P2, P3 페이지 3개가 속한다. 커밋 쓰레드가 tx1을 커밋할 때, 쉐도우 페이지 P'1, P'2, P'3가 생성된다. 추후 파일 연산이 P1, P2, P4를 갱신하고 저널 커밋이 트리거된다. 후행하는 트랜잭션 tx2는 P1, P2, P4로 구성된다. 커밋 쓰레드는 쉐도우 페이지 P"1 (P1의 두 번째 쉐도우 페이지), P"2 (P2의 두 번째 쉐도우 페이지), P'4를 생성한다. 추후 호출되는 파일 연산은 P1, P2, P3를 갱신해야 한다. P1, P2, P3는 갱신 가능하고, 러닝 트랜잭션에 삽입 후 갱신된다.

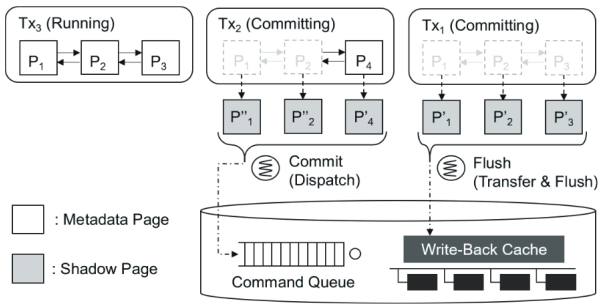

그림 7 멀티 버전 쉐도우 페이징

듀얼 모드 저널링의 병렬성을 위해서는 커밋 스레드와 플러시 스레드 모두 선행 트랜잭션이 존재하더라도 다음 트랜잭션을 디스패치 하거나 플러시 할 수 있어야 한다. 멀티 버전 쉐도우 페이징은 커밋 스레드가 선행 트랜잭션이 존재하더라도 다음 트랜잭션을 디스패치 할 수 있도록 한다. 하지만 플러시 동작은 여전히 직렬화된다. 플러시 쓰레드는 선행 트랜잭션의 영속성이 보장된 후 다음 트랜잭션을 플러시 한다. 상기 문제를 해결하기 위해, 우리는 컴파운드 플러시 기법을 제시한다. 컴파운드 플러시는 캐시 배리어 커맨드를 활용한다. 플러시 쓰레드는 플러시 명령어를 디스패치 하기 전 후행 커밋팅 트랜잭션의 존재 여부를 확인한다. 만약 후행 커밋팅 트랜잭션이 존재하지 않는다면 플러시 커맨드를 디스패치 한다. 후행 커밋팅 트랜잭션이 존재할 경우, 플러시 커맨드 대신 캐시 배리어 커맨드를 디스패치 한다.

5.3 로그 구조 파일 시스템 – F2FS

리눅스는 fsync() 호출 시, 단일 파일의 데이터와 메타데이타를 디스크에 기록한다[3]. 따라서 F2FS에서 fsync()를 호출하면 fsync() 호출 대상 파일 데이터와 파일 매핑이 담긴 노드 블락이 스토리지에 기록된다. 그림 8는 F2FS의 fsync() 동작이다. F2FS는 데이터 블록과 노드 블록 쓰기 간 DMA 전송만 대기하고 캐시 플러시를 수행하지 않는다. 따라서 fsync() 도중 크래시가 발생하면 노드 블록이 데이터 블록보다 먼저 써질 수 있다. 이 경우, 파일 매핑이 쓰레기 블록을 가리키게 되고 파일 일관성이 지켜지지 않는다.

F2FS가 파일 일관성을 희생하며 데이터 블록과 노드

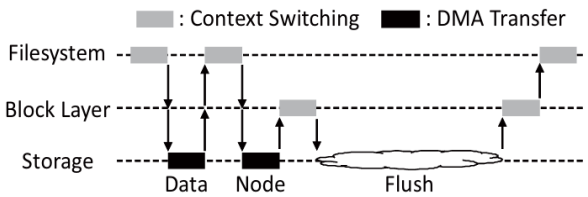

그림 8 F2FS의 fsync() 타임라인

```
1. creat(/d/manifest-new);
2. append(/d/manifest-new);
3. fsync(/d/manifest-new);
4. creat(/d/tmp);
5. append(/d/tmp);
6. fsync(/d/tmp);
7. rename(/d/tmp,
        /d/current);
8. fsync(/d);
```

그림 9 RocksDB의 manifest 교체

```
1. creat(/d/journal);
2. append(/d/journal);
3. fsync(/d/journal);
4. fsync(/d);
5. write(/d/journal);
6. fsync(/d/journal);
7. write(/d/db);
8. fsync(/d/db);
9. unlink(/d/journal);
```

그림 10 SQLite의 Persist 저널링

블록 간 쓰기 순서를 보장하지 않는 이유는 과도한 캐시 플러시로 인한 성능 저하를 방지하기 위함이다. 파일 일관성 보장이 필요한 경우를 위해 F2FS에 데이터 블록과 노드 블록 쓰기 사이에 플러시를 삽입하는 strict 모드가 추가되었다. 하지만 strict 모드는 과도한 플러시로 인해 기본 모드인 posix 모드 대비 성능 저하가 발생한다[37]. 입출력 스택이 캐시 배리어 기반의 경량 쓰기 순서보장 매카니즘을 지원하면 성능 저하 없이 F2FS의 파일 일관성을 쉽게 지원할 수 있다[37].

6. 순서보장형 응용 프로그램

많은 애플리케이션은 사용자 데이터 크래시 일관성을 보장하기 위해 atomic rename[38][39], 로깅[39][40] 등을 사용한다. atomic rename과 로깅은 일련의 인 메모리 파일 연산과 인 메모리 파일 연산 간 순서보장을 위한 fsync() 호출로 구성된다.

Atomic rename은 (i) 임시 파일 생성과 (ii) 임시 파일 이름 갱신으로 구성된다. Atomic rename 사용 시 파일 내용이 비어 있는 경우를 방지하기 위해 (i)과 (ii) 사이 fsync()를 호출한다[39]. 그림 9는 RocksDB가 manifest[41] 파일을 갱신하는 예제다. RocksDB는 (i) 새로운 manifest 파일을 생성하고 (Line 1~2), (ii) 새로운 manifest 파일의 이름을 current 파일에 기록한다 (Line 4~8). RocksDB는 상기 프로토콜에서 fsync()를 두 번 호출한다. 각각 (i)과 (ii) 사이 순서 보장 (Line 3), (ii)의 원자성 보장을 위해서다 (Line 6).

Logging은 (i) 저널 파일 갱신과 (ii) 원본 파일 갱신으로 구성된다. Logging의 (i)과 (ii) 사이 쓰기 순서는 보장되어야 하며, 이를 위해 fsync()를 (i)과 (ii) 사이에 삽입한다. 그림 10은 PERSIST 모드 SQLite의 데이터베이스 파일 갱신 프로토콜이다. SQLite의 데이터베이스 갱신 프로토콜은 (i) 저널 파일 갱신 (Line 1~6)과 (ii) 데이터베이스 파일 갱신 (Line 7)로 구성된다. SQLite fsync()를 두 번 호출하며, 각각 (i)과 (ii) 사이 순서보장 (Line 6), 트랜잭션의 원자성 보장을 위해 호출한다 (Line 3). 입출력 스택이 캐시 배리어 등으로 순서보장을 지원한다면 크래시 일관성을 보장하면서도 성능을 향상시킬 수 있다[13]. BarrierFS[13]는 그림 10의 fsync()를 fbarrier()로 교체해 SQLite 성능을 43배 개선한다.

7. 결 론

현대 저장장치 환경에서 쓰기 순서보장은 하드웨어의 발전과 병렬성 증가에 따라 중요한 연구 주제로 변화해왔다. 기존 입출력 스택은 전송 및 플러시 (Transfer-and-Flush) 방식으로 저장 순서를 보장하지만, 이로 인한 성능 병목이 부각 되면서 캐시 배리어 (cache barrier) 명령어와 같은 대안이 제시되었다.

블록 계층 측면에서는 단일 큐 구조에서 캐시 배리어를 활용한 순서보장 최적화 연구가 진행되었다. 그러나 단일 큐의 락 경합으로 인한 성능 저하 문제가 발생하면서 다중 큐 블록 계층이 도입되었고, 이에 따라 큐 간 쓰기 순서를 보장하는 추가적인 기법이 필요해졌다. 이를 해결하기 위해 다중 큐 입출력 스택을 위한 블록 계층, 순서보장형 FTL 등의 새로운 접근 방식이 연구되고 있다.

파일 시스템 측면에서는 EXT4와 F2FS가 순서보장을 위한 다양한 기법을 도입하였으며, 응용 프로그램 수준에서는 데이터베이스 시스템이 fsync() 호출을 최적화하거나 새로운 동기화 인터페이스를 활용하는 방식으로 대응하고 있다. 본 원고는 이러한 연구 동향을 정리하고, 저장장치 성능과 신뢰성 향상을 위한 향후 연구 방향에 대해 기술한다.

참고문헌

[1] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, “I/O Stack Optimization for Smartphones,” in Proc. USENIX Annu.

Technical Conf. (USENIX ATC), 2013.
[2] C. Lee, D. Sim, J. Hwang, and S. Cho, “F2FS: A New File System for Flash Storage,” in Proc. USENIX Conf. on File and Storage Technologies (FAST), 2015.
[3] MySQL AB, MySQL 5.1 Reference Manual, 2007.
[4] S. Best, “JFS Overview,” 2000, [Online] Available: http://jfs.sourceforge.net/project/pub/jfs.pdf
[5] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, “Scalability in the XFS File System,” in Proc. USENIX Annu. Technical Conf. (USENIX ATC), 1996.
[6] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, “The new ext4 filesystem: current status and future plans,” in Proc. Linux Symposium, 2007.
[7] S. Tweedie, “Journaling the Linux ext2fs filesystem,” in Proc. Annual Linux Expo, 1998.
[8] M. Seltzer, G. Ganger, M. McKusick, K. Smith, C. Stein, and C.A.N. Soules, “Journaling versus Soft Updates: Asynchronous Meta-Data Protection in File Systems,” in Proc. USENIX Annu. Technical Conf. (USENIX ATC), 2000.
[9] M. McKusick and G. Ganger, “Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem,” in Proc. USENIX Annu. Technical Conf. (USENIX ATC), 1999.
[10] R. Kesavan, R. Singh, T. Grusecki, and Y. Patel, “Algorithms and Data Structures for Efficient Free Space Reclamation in WAFL,” in Proc. USENIX Conf. on File and Storage Technologies (FAST), 2017.
[11] O. Rodeh, J. Bacik, and C. Mason, “BTRFS: The Linux B-tree filesystem,” ACM Trans. Storage, vol. 9, no. 3, 2013.
[12] M. Rosenblum and J. Ousterhout, “The Design and Implementation of a Log-structured File System,” ACM Trans. Comput. Syst., vol. 10, no. 1, 1992.
[13] Y. Won, J. Jung, G. Choi, J. Oh, and S. Son, “Barrier-Enabled IO Stack for Flash Storage,” in Proc. USENIX Conf. on File and Storage Technologies (FAST), 2018.
[14] F. Chen, R. Lee, and X. Zhang, “Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing,” in Proc. IEEE Int. Symp. on High Performance Computer Architecture (HPCA), 2011.
[15] S. Park, E. Seo, J. Shin, S. Maeng, and J. Lee, “Exploiting Internal Parallelism of Flash-based SSDs,” IEEE Comput. Architecture Lett., vol. 9, no. 1, 2010.
[16] D. Narayanan, A. Donnelly, and A. Rowstron, “Write off-loading: Practical power management for enterprise storage,” ACM Trans. Storage, vol. 4, no. 3, 2008.
[17] B. Dees, “Native command queuing-advanced performance in desktop storage,” IEEE Potentials Mag., vol. 24, no. 4, 2005.
[18] JEDEC, “Universal Flash Storage 2.1,” JEDEC Standard JESD220C, 2016.
[19] J. Guo, J. Yang, Y. Zhang, and Y. Chen, “Low cost power failure protection for MLC NAND flash storage systems with PRAM/DRAM hybrid buffer,” in Proc. Design, Automation & Test in Europe Conf. (DATE), 2013.
[20] “Barriers and journaling filesystems,” 2010, http://lwn.net/Articles/283161
[21] H. Kim and J. Kim, “Tuning the ext4 filesystem performance for android-based smartphones,” Comput. Educ., 2012.
[22] V. Prabhakaran et al., “IRON File Systems,” in Proc. ACM Symp. on Operating Systems Principles (SOSP), 2005.
[23] G. Shilamkar, “Journal Checksums,” 2007. [Online]. Available: https://wiki.old.lustre.org/images/4/44/Journal-checksums.pdf
[24] C. Frost et al., “Generalized File System Dependencies,” in Proc. ACM Symp. on Operating Systems Principles (SOSP), 2007.
[25] C. Vijay et al., “Optimistic Crash Consistency,” in Proc. ACM Symp. on Operating Systems Principles (SOSP), 2013.
[26] E. Nightingale, K. Veeraraghavan, P. Chen, and J. Flinn, “Rethink the Sync,” in Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI), 2006.
[27] Embedded Multi-Media Chip Card (eMMC) Electrical Standard (5.1), JEDEC Standard, 2015.
[28] J. Axboe, “Linux block IO present and future,” in Proc. Ottawa Linux Symp., 2004.
[29] J. Kim et al., “OPIMQ: Order Preserving IO stack for Multi-Queue Block Device,” in Proc. USENIX Conf. on File and Storage Technologies (FAST), 2025.
[30] X. Liao, Z. Yang, J. Shu, Z. Xu, and Y. Lu, “Crash consistent non-volatile memory express,” in Proc. ACM Symp. on Operating Systems Principles (SOSP), 2021.
[31] X. Liao, Z. Yang, J. Shu, Z. Xu, and Y. Lu, “Write dependency disentanglement with HORAE,” in Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI), 2020.

[32] Y. Zhang, W. Chen, S. Zhou, and X. Liao, “LazyBarrier: Reconstructing Android IO Stack for Barrier-Enabled Flash Storage,” in Proc. ACM Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2024.
[33] Y. Chang and R. Liu, “OPTR: Order-Preserving Translation and Recovery Design for SSDs with a Standard Block Device Interface,” in Proc. USENIX Annu. Technical Conf. (USENIX ATC), 2019.
[34] X. Liao, Z. Yang, and J. Shu, “Rio: Order-preserving and cpu-efficient remote storage access,” in Proc. European Conf. on Computer Systems (EuroSys), 2023.
[35] S. Palanca, S. Fischer, S. Maiyuran, and S. Qawami, “MFENCE and LFENCE micro-architectural implementation method and system,” U.S. Patent 9,383,998, 2016.
[36] J. Oh, S. Yoo, H. Nam, C. Min, and Y. Won, “CJFS: Concurrent Journaling for Better Scalability,” in Proc. USENIX Conf. on File and Storage Technologies (FAST), 2023.
[37] S. Park, “순서보장형 입출력 스택을 이용한 F2FS의 크래시 일관성 보장 기법,” KAIST, 2024.
[38] Y. Hu, Y. Bao, T. Wei, C. Wang, M. Zhao, and Z. Wang, “TxFS: Leveraging File-System Crash Consistency to Provide ACID Transactions,” in Proc. USENIX Annu. Technical Conf. (USENIX ATC), 2018.
[39] T. S. Pillai, D. Bittman, P. M. Chen, J. Flinn, and T. C. Ristenpart, “All file systems are not created equal: On the complexity of crafting crash-consistent applications,” in Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI), 2014.
[40] J. Oh, S. Ji, Y. Kim, and Y. Won, “exF2FS: Transaction Support in Log-Structured Filesystem,” in Proc. USENIX Conf. on File and Storage Technologies (FAST), 2022.
[41] “MANIFEST”-RocksDB Wiki, “2025, [Online]. Available: https://github.com/facebook/rocksdb/wiki/MANIFEST
[42] “fsync(2) - Linux manual page,” 2025, https://www.man7.org/linux/man-pages/man2/fsync.2.html.
[43] Q. Xu, E. Hwang, J. Yang, and Y. Zhang, “Performance Analysis of NVMe SSDs and Their Implication on Real World Databases,” in Proc. ACM Int. Conf. on Systems and Storage (SYSTOR), 2015.
[44] T. S. Pillai, D. Bittman, P. M. Chen, and J. Flinn, “Application Crash Consistency and Performance with CCFS,” in Proc. USENIX Conf. on File and Storage Technologies (FAST), 2017.
[45] T. Ts'o, “Using Cache barrier in lieu of REQ_FLUSH,” 2015. [Online]. Available: http://www.spinics.net/lists/linux-ext4/msg49018.html

약 력

김 지 은

2015~2019 이화여자대학교 전자공학과 졸업(학사)
2020~현재 한국과학기술원 전기 및 전자공학부 재학 (석박사 통합)
Email : pipiato@kaist.ac.kr

유 승 원

2017~2021 울산과학기술원 전기 및 전자공학과 졸업 (학사)
2021~2023 한국과학기술원 전기 및 전자공학과 졸업 (석사)
2023~현재 한국과학기술원 전기 및 전자공학과 재학 (박사)
Email : swyoo98@kaist.ac.kr

원 유 집

1986~1990 서울대학교 계산통계학과 졸업(학사)
1990~1992 서울대학교 계산통계학과 졸업(석사)
1992~1997 미네소타 주립대학교 컴퓨터과학과 졸업 (박사)
1997~2000 인텔사 서버 성능 분석가
1999~2019 한양대학교 컴퓨터과학과 교수
2019~현재 한국과학기술원 전기 및 전자공학부 ICT 석좌교수
Email : ywon@kaist.ac.kr