매니코어 시스템을 위한 리눅스 파일 시스템의 확장성과 고성능 데이터 처리에 관한 연구

서강대학교 ▌황 순·변홍수·김영재*·박성용**
성균관대학교 ▌김종석·서의성
국민대학교 ▌정지현·주용수*

1. 서 론

빅데이터는 대용량의 정형 데이터 및 비정형 데이터의 집합으로 전통적인 데이터베이스 처리 방식으로 처리하기 힘든 큰 데이터 집합이다. 빅데이터의 출현으로 인해 전체 데이터의 90%가 최근 수년간 생산되었으며 이 생산량은 점차 가속화 될 것으로 예상된다[1]. 이러한 대용량의 데이터는 이미지 처리, 그래프 처리, 그리고 기계학습과 데이터베이스에 이르기까지 다양한 분야에서 사용되고 있다. 빅데이터 처리를 위해 GPU 등 기존 프로세서와 다른 특성을 가진 프로세서들이 출현하였으나, I/O가 많은 작업을 하기에는 적합하지 않다. 따라서 I/O 집약적인 작업 처리를 위해 매니코어 서버가 주목받고 있다.

매니코어 환경은 단일 서버 내부에 수백 개의 코어와 대용량의 메모리를 가지고 있다. 이에 따라 수백 개의 스레드를 동원하여 대용량의 데이터 I/O를 병렬적으로 처리할 수 있다. 또한, NVMe SSD와 PM 등의 기존 저장장치보다 높은 대역폭을 가지는 고성능 저장장치를 이용하여 I/O 성능이 저장장치로 인해 병목이 생기는 문제를 해결할 수 있다. 고성능 저장장치를 충분히 활용하기 위해서 디바이스의 특성을 고려하여 파일 시스템들이 개발되고 있다[2, 3]. 매니코어 환경의 높은 병렬성과 고성능 저장장치 및 적합한 파일 시스템을 사용한다면 기존에 단일서버에서 처리하기 어려웠던 고성능 데이터베이스 시스템 등의 I/O 집약적인 작업을 효율적으로 처리할 수 있을 것으로 기대된다.

그러나 매니코어 환경에서 고성능 저장장치를 이용해도 I/O 작업량이 코어 수에 비례하여 증가하는 확장성을 보이지 못한다. 이는 리눅스 파일 시스템들의 디자인이 매니코어 환경을 고려하지 않았기 때문이다. 많은 수의 스레드들이 리눅스 파일 시스템들의 데이터 일관성 유지를 위한 동기화 메커니즘과 인메모리 자료 구조의 구조적인 문제로 인해 병렬적으로 작업을 처리하지 못하여 확장성이 저해된다. 이에 따라 많은 연구자가 데이터 일관성을 유지하면서 매니코어 환경에서 파일 시스템의 성능 확장성을 보일 수 있도록 연구를 진행 중이다.

매니코어 환경에서의 확장성을 저해하는 병목 요인은 전체 I/O 스택 중 한 계층에 국한되어 있지 않다. 이는 파일 시스템들이 공유하는 저널 계층, 개별 파일 시스템 내부, 그리고 블록 디바이스에서 수행되는 I/O 명령을 관리하는 블록 계층 등 리눅스 파일 시스템에서 I/O에 관여하는 여러 계층에 존재한다. 본 고는 각 계층에서 매니코어 환경에서의 확장성을 향상시키기 위한 연구를 다룬다. 본 고의 2장은 리눅스의 저널 계층 확장성에 다루고, 3장에서는 F2FS 파일 시스템의 확장성, 4장은 NOVA 파일 시스템의 확장성을 다룬다. 마지막으로 5장에서는 리눅스 블록 계층 확장성에 다룬다.

2. 리눅스 저널 계층 확장성

2.1 배경

일반적으로 파일 시스템의 오퍼레이션(operation) 수행은 여러 블록(block)의 수정과 관련이 있지만, 기존 저장장치들은 여러 블록에 대한 수정이 원자적(atomic)으로 반영되는 것을 보장하지 않는다. 그러므로 파일 시스템의 오퍼레이션 수행 중 갑작스런 충돌(crash)이 발생하면 수정 내용의 일부만이 저장장치에

* 정회원
** 종신회원

반영될 수 있어 파일 시스템 메타데이터와 실제 데이터가 서로 일치하지 않을 수 있다. 이렇게 메타데이터를 부분적으로 업데이트하거나 메타데이터와 데이터가 일치하지 않으면 파일 시스템의 유효성을 손상시킬 수 있다.

저널링 메커니즘(journaling mechanism)은 미리 정해진 위치에 오퍼레이션으로 인한 파일 시스템 변경 사항을 기록하여 일관성을 보장하기 위한 조치이다. 저널은 파일 시스템이 일련의 변경 사항을 미리 정해진 영역에 커밋(commit)한 후 체크포인트(checkpoint) 작업을 통해 기록된 변경 사항을 파일 시스템에 반영한다. 파일 시스템 작업이 커밋되면 저널은 시스템 오류 후에도 커밋된 변경 내용을 알 수 있기 때문에 변경 내용이 파일 시스템에 안전하게 반영되는 것을 보장한다.

JBD2(Journaling Block Device 2)[4]는 리눅스(Linux) 커널에서 사용되는 일반적인 저널링 계층이다. JBD2는 일정 시간 동안 발생한 일련의 파일 시스템 변경 사항을 트랜잭션(transaction)이라는 단위로 그룹화한다. 트랜잭션은 kjournald라고 불리는 단일 스레드에 의해 저널 영역에 주기적으로 커밋된다. 트랜잭션이 저널에 커밋되면 JBD2는 해당 커밋 기록의 끝에 커밋 블록을 남긴다. JBD2는 충돌 후 복구를 수행할 때 커밋 블록이 있는 트랜잭션들을 체크포인트하고 커밋 블록이 없는 트랜잭션들을 폐기한다.

현재의 JBD2에는 다음과 같은 다수의 확장성 병목 지점들이 존재한다. 먼저, 일정 시간 동안 수정되는 블록을 모아두는 실행(running)상태의 트랜잭션은 전체 파일 시스템에서 동시에 오직 하나만 존재할 수 있다. 따라서, 다수의 코어가 병렬로 파일 오퍼레이션을 수행할 때, 실행상태의 트랜잭션에 대한 잠금(lock) 획득을 위해 경쟁해야 하기 때문에 확장성 저하의 요인이 된다[5].

둘째로, 실행상태의 트랜잭션과 커밋팅(committing) 상태의 트랜잭션은 모두 전체 파일 시스템에 한 개만 존재할 수 있다. 실행상태의 트랜잭션이 일정 시간이 지났거나 가득 찼을 때, 현재 커밋팅 트랜잭션이 완료된 후에만 실행상태의 트랜잭션이 커밋팅 상태로 진입할 수 있으며, 그제야 실행상태의 트랜잭션을 새로 생성할 수 있다. 결과적으로 실행상태의 트랜잭션이 커밋 트랜잭션이 완료될 때까지 기다릴 때 파일 작업을 실행하는 모든 코어는 모두 대기해만 한다. 이는 전체 시스템에 코어의 수가 많을수록 전체 파일 시스템 처리량에 더 큰 악영향을 미친다[6].

마지막으로, kjournald 스레드가 모든 쓰기 오퍼레이션을 직렬화(serialization)하여 혼자 작성하기 때문에 현재 JBD2는 최신 NVMe SSD에서 제공하는 내부 병렬성[7]을 완전히 활용하지 못한다[6]. 최신 저장장치의 고성능을 활용하려면 저널 메커니즘을 병렬로 적용할 수 있어야 한다.

기존 중앙화된 저널링의 문제점을 해결하기 위해 코어별로 저널링 과정을 병렬화시키는 것은 가능한 병렬적인 커밋 과정과 독립적인 트랜잭션 관리를 허용하면서 버퍼(buffer)에 대한 쓰기 순서를 보존할 수 있는 새로운 일관성 메커니즘을 필요로 한다.

2.2 디자인

본 논문에서 우리는 확장성 있는 코어별 저널 스킴(scheme)인 Z-Journal[6]을 제안한다. Z-Journal에서 각 코어들은 저장장치에 각자 자신만의 저널 영역이 있으며, 코어들은 서로 독립적으로 본인만의 저널에 커밋 작업을 수행한다. kjournald는 각 코어에 바인딩(binding)되어 로컬(local)로 실행되므로 불균형 기억장치 접근(NUMA) 시스템에서의 메모리 영역 접근에 대한 지역성도 개선될 수 있다. 각 코어의 저널들은 실행상태의 트랜잭션, 커밋팅 상태의 트랜잭션 및 체크포인트해야 하는 트랜잭션의 리스트(list)를 JBD2와 동일하게 가지고 있으며, 트랜잭션들의 라이프 사이클(life cycle) 또한 JBD2와 동일한 과정을 거친다.

이 코어별 저널 디자인은 앞서 언급한 저널 계층에서의 직렬화 지점들을 제거하여 확장성을 확보한다[6]. 먼저, 각 코어가 실행상태의 트랜잭션을 가지고 있기 때문에 스레드는 실행상태의 트랜잭션에 대한 접근 권한을 얻기 위해 다른 스레드와 경쟁할 필요가 없다. 둘째로 코어의 실행상태인 트랜잭션이 끝나고 코어가 실행상태의 트랜잭션을 새로 생성하기 위해 현재의 커밋팅 트랜잭션이 완료될 때까지 기다릴 때, 이러한 대기 상태는 해당 코어에만 적용된다. 따라서, 시스템의 코어가 증가하여도 처리량에 추가되는 악영향은 존재하지 않는다. 마지막으로, 여러 kjournald를 병렬로 사용할 수 있기 때문에 Z-Journal은 최신 고성능 저장장치를 충분히 활용할 수 있다.

그러나 이러한 코어별 저널에서는 두 개 이상의 스레드가 동일한 파일 시스템 블록에 대해 쓰기 작업을 동시에 수행할 경우 메모리 내 버퍼에 대한 쓰기 순서와 저장장치의 저널 영역에 대한 쓰기 순서 간에 불일치 문제가 발생할 수 있다. 예를 들어, 코어 0이 블록 0과 블록 1을 수정한다고 가정하자. 마찬가지로, 그다음 순서로 코어 1이 블록 0과 블록 2를 수정한다. 이때, 코어 1에서 발행한 저널 커밋은 확장성을 위해

독립적으로 운용되기 때문에 코어 0의 커밋보다 일찍 종료될 수 있다. 코어 1이 커밋되었지만 코어 0이 커밋되지 않았을 때 시스템 충돌이 발생한다면 복구 중에 코어 0과 코어 1의 수정 사항이 모두 포함된 상태로 블록 0이 복원된다. 코어 0이 수정 사항을 커밋하지 않았기 때문에 복구 후 올바른 블록 0의 데이터는 코어 0에 의한 수정 사항만을 제외하거나 수정 사항이 전혀 없어야 한다. 이런 종류의 문제는 기존의 중앙화된 저널에서는 발생할 수 없다.

일반적으로 다수의 스레드가 파일 오퍼레이션들을 수행한다면 코어들은 서로 메타데이터 혹은 파일을 공유한다. 파일이 동일한 블록 그룹[7]에 저장되는 경우 두 코어가 파일을 공유하지 않더라도 메타데이터 블록을 공유해야만 한다. 그러므로 여러 코어가 동일한 파일 시스템에 접근할 경우 코어 간에 블록을 공유하는 것이 불가피하다. 그렇기 때문에 확장성 있는 코어별 저널링은 공유 블록 수정에 관해 쓰기 순서를 유지할 수 있는 저널 일관성 메커니즘을 가져야 한다.

Z-Journal도 저널 일관성 메커니즘을 가지고 있으며, 이는 각 코어가 다른 코어의 활동에 의해 방해받지 않고 트랜잭션의 커밋을 수행할 수 있도록 체크포인트 과정에서만 트랜잭션들 사이에 쓰기 순서를 부과한다. Z-Journal에서 커밋된 트랜잭션은 쓰기 순서를 확인했을 때 해당 트랜잭션 이전의 모든 트랜잭션이 커밋된 이후에만 유효한 것으로 간주한다. Z-Journal은 이런 유효한 커밋들만을 체크포인트한다.

이를 실현하기 위해 Z-Journal은 트랜잭션들 사이의 순서 관계를 식별하고 트랜잭션 커밋 시에 순서 관계를 함께 기록할 수 있어야 한다. 그래서 우리는 이에 대한 순서 보존 트랜잭션 체인(order-preserving transaction chain)을 제안한다.

그림 1은 Z-Journal에서 순서 보존 트랜잭션 체인이 동작하는 것을 나타낸 그림이다. 트랜잭션들은 자신만의 고유한 식별자를 트랜잭션 체인 리스트(chained-transaction list)에 기록함으로써 순서 관계를 가진 트랜잭션에 대한 정보를 유지한다. 하나의 트랜잭션에 존재하는 리스트의 수는 시스템에 존재하는 코어 수와 같으므로 각 코어는 잠금을 획득하지 않고 모든 트랜잭션에서 자신의 코어에 해당하는 리스트를 수정할 수 있다. 트랜잭션은 각 저널 내에서 고유한 아이디(ID)를 가지고 있다. 따라서 트랜잭션의 고유 식별자는 코어 아이디와 트랜잭션 아이디의 쌍으로 구성할 수 있다. 트랜잭션이 체인 트랜잭션 리스트에서 (j, t)라는 체인 트랜잭션 마크(chained-transaction mark)를 가진다면, 해당 트랜잭션은 코어 j의 트랜잭션 t가 커밋된 후에만 유효하다.

만약 실행상태인 서로 다른 저널의 두 트랜잭션이 버퍼를 공유할 경우 양방향 트랜잭션 체인을 형성해야 한다. 그렇지 않고 실행상태의 트랜잭션이 커밋팅 트랜잭션에 속하는 버퍼에 쓰려고 한다면, 두 트랜잭션은 실행상태인 트랜잭션이 커밋 트랜잭션을 따르는 단방향 체인을 형성해야 한다.

예를 들어, 그림 1에서 코어 2의 트랜잭션 2에 의해 수정된 버퍼 4는 코어 1의 트랜잭션 7에 의해 수정될

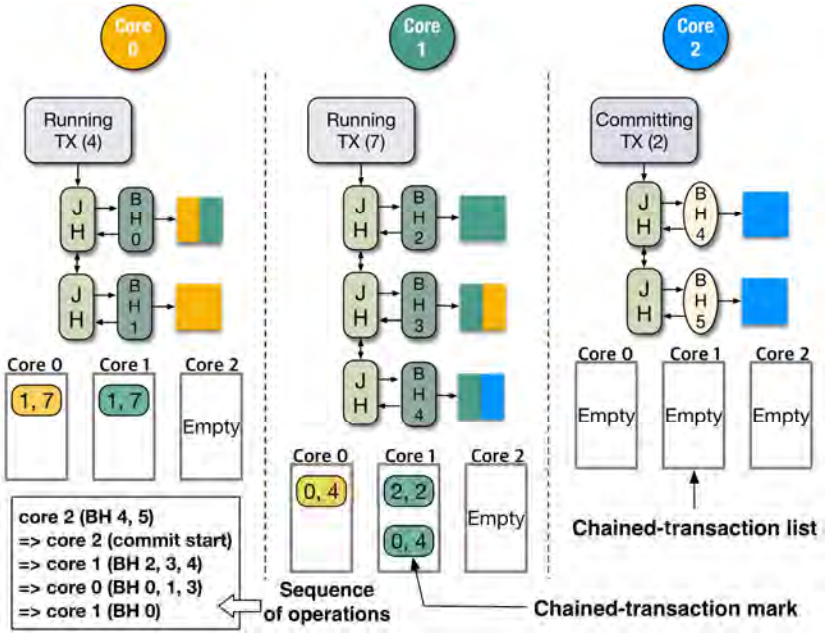


그림 1 순서 보전 트랜잭션 체인 예시 (참고문헌[6]의 그림)

예정이다. 코어 2의 트랜잭션 2가 이미 커밋 상태에 있으므로 코어 1은 코어 2의 트랜잭션 2를 의미하는 (2, 2)를 코어 1의 트랜잭션 7 목록에 남긴다. 이를 통해 (1, 7)과 (2, 2) 사이에 단방향 체인이 형성된다. 이 단방향 체인의 의미는 코어 1이 가진 버퍼 4의 데이터가 코어 2의 트랜잭션 2와 연관이 있으므로 이러한 순서를 보존하기 위함에 있다.

코어 0이 버퍼 3을 수정하려 하는 경우에 버퍼 3은 아직 코어 1의 실행상태 트랜잭션에 남아 있기 때문에 해당 상태 그대로 수정할 수 있다. 다만 이러한 경우에는 버퍼 3과 연관된 데이터를 가진 새로운 버퍼가 코어 0의 실행상태 트랜잭션에 존재할 수 있기 때문에, 실행상태인 이 두 트랜잭션은 마치 하나의 트랜잭션인 것처럼 간주되어야 한다. 따라서 이러한 의미의 순서 보존을 위해 코어 0은 저널 1의 트랜잭션 체인 리스트에는 (0, 4)를 남기고, 저널 0의 트랜잭션 체인 리스트에는 (1, 7)을 남긴다.

Z-Journal은 제안하는 순서 보존 트랜잭션 체인을 통해 트랜잭션 간의 쓰기 순서를 확장성 있고 효율적인 방법으로 추적할 수 있게 해준다. 이를 바탕으로 Z-Journal은 쓰기 순서의 적용을 체크포인트 시간까지 미루고 코어가 공유 상태에 놓여 있는 것과는 관계없이 트랜잭션을 독립적으로 커밋할 수 있도록 해준다. 일반적인 체크포인트 간격이 트랜잭션의 커밋 간격보다 길기 때문에 체크포인트 시점에 이르러서는 거의 모든 트랜잭션이 유효해질 가능성이 높다. 따라서 체크포인트 수행 시점에서 순서 제약으로 인해 발생하는 오버헤드는 커밋 시점에 준수하는 것에 비해 월등히 적을 것으로 예상할 수 있다.

앞에서 언급한 것처럼 Z-Journal에서는 커밋된 트랜잭션이 체크포인트의 대상이 되는 것은 아니다. 이를 구현하기 위해 kjournald가 트랜잭션의 상태를 체크포인트 상태로 변경할 때 트랜잭션 버퍼의 더티 플래그(dirty flag) 설정을 건너뛴다. 대신에 kjournald가 현재 실행 중인 트랜잭션을 커밋 트랜잭션으로 변환할 때마다 체크포인트 트랜잭션 리스트에 존재하는 트랜잭션들이 유효한지 확인한다. 만약 트랜잭션이 유효한 것으로 확인되면 그때 해당 트랜잭션에 속한 버퍼들에 더티 플래그를 설정한다. 그렇게 한다면 추후 kworker 스레드에 의해 백그라운드(background)에서 더티 플래그인 버퍼들이 체크포인트 되며, 이는 주기적으로 반복된다.

커밋 완료된 특정 트랜잭션이 유효하려면, 해당 트랜잭션보다 순서상으로 앞선 버퍼들을 가진 모든 트랜잭션이 커밋 완료되었을 뿐만 아니라 유효해야 한다. 그러므로 커밋된 트랜잭션의 유효성을 검사하기 위해 kjournald는 해당 트랜잭션과 그 상위 트랜잭션의 체인 리스트 마크로부터 생성된 트랜잭션 체인 그래프를 순회하고 모든 선행 트랜잭션들이 유효한지 여부를 확인한다. 모든 트랜잭션에는 유효성을 나타내는 필드가 존재한다. 탐색 중 트랜잭션의 모든 선행 트랜잭션들이 유효한 것으로 확인되면 kjournald는 이후 중복 탐색을 방지하기 위해 해당 유효성 필드를 설정해준다.

2.3 성능평가

표 1은 성능평가에 사용한 시스템 구성을 나타낸다. 우리는 리눅스 커널에 Z-Journal을 구현하고, 파일 오퍼레이션 수행 시 코어별 저널을 인식하고 JBD2 대신 Z-Journal을 사용하도록 하기 위해 ext4 파일 시스템을 수정하였다. 또한, mke2fs를 수정하여 ext4 파일 시스템이 여러 저널을 가질 수 있도록 포맷하였다. 추가로 ext4 파일 시스템의 슈퍼 블록(super block)이 여러 저널 구조체를 가질 수 있도록 하였고, 마운트(mount) 작업 시 이를 인식하도록 수정하였다.

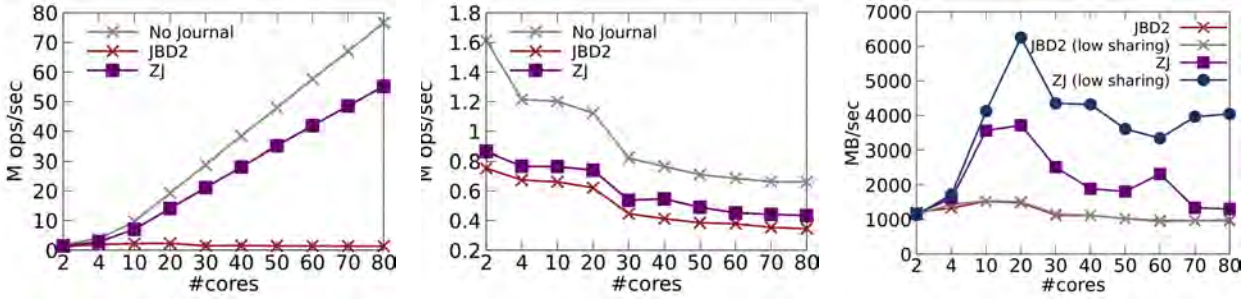
우리는 Z-Journal(그래프에서는 ZJ로 표시)의 성능을 JBD2(그래프에서는 JBD2로 표시)의 ext4와 저널이 없는 ext4(그래프에서는 No Journal로 표시)의 성능을 비교하였다. 전체 파일 시스템의 확장성을 목표로 하는 것이 아니기 때문에 저널이 없는 ext4의 성능은 Z-Journal이 달성할 수 있는 최고의 성능으로 생각할 수 있다.

먼저 우리는 다양한 블록 공유 조건에서 파일 오퍼레이션을 수행할 때 Z-Journal의 확장성을 평가하기 위해 FxMark[8]를 사용하여 성능을 평가하였다. 그림 2 (a)와 (b)은 각각 코어별로 서로 다른 폴더의 파일에 덮어쓰기(overwrite)하는 작업(DWOL)에 대한 결과와 모든 코어가 하나의 파일의 서로 다른 블록에 덮어쓰기하는 작업(DWOM)에 대한 결과를 보여주고 있다.

Z-Journal은 DWOL에서 저널에 추가로 써야 하는 오버헤드로 인한 성능 저하를 제외하고 저널이 없는 ext4에 근접한 성능을 보였으며 코어 수가 증가할수록 성능이 향상하는 모습을 보였다. 결과적으로 Z-Journal은 80코어의 JBD2에 비해 41배 높은 처리량을 보였다. 그러나 DWOM의 경우 파일의 공유로 인해 파일 시스템 자체의 확장성이 떨어지는 모습을 보였고, 저널 계층도 같은 이유로 성능이 떨어졌다. 그러나 이 경우에도 Z-Journal은 코어별 저널링 메커니즘을 통해 JBD2 대비 80코어에서 30%의 성능 향상을 달성했다.

표 1 성능평가를 위한 시스템 환경 구성

		Specification
Processor	Model	Intel Xeon Gold 6138 × 4 sockets
	Number of Cores	20 × 4
	Clock Frequency	2.00 GHz
Memory		DDR4 2666 MHz 32 GB × 16
Storage		Samsung SZ985 NVMe SSD 800GB
OS	Kernel	Linux 4.14.78



(a) FxMark DWOL 실험 결과(참고문헌[6]의 결과 재구성)

(b) FxMark DWOM 실험 결과(참고문헌[6]의 결과 재구성)

(c) Filebench의 fileserver 실험 결과(참고문헌[6]의 결과 재구성)

그림 2 Z-Journal의 성능 평가

우리는 읽기, 쓰기 및 메타데이터 작업이 혼합된 환경에서 Z-Journal이 성능에 미치는 영향을 평가하기 위해 Filebench[9]의 fileserver 워크로드를 사용하여 성능을 평가하였다. Fileserver는 파일 생성, 덮어쓰기, 덧붙여 쓰기(append), 읽기 및 파일 삭제 작업을 반복적으로 수행한다.

그림 2 (c)는 fileserver의 처리량을 나타낸다. 다수의 코어가 적은 수의 폴더에서 반복적으로 파일을 만들고 수정했기 때문에 JBD2의 처리량은 10개의 코어 수에서 가장 높은 성능을 보이고 코어 수가 증가함에 따라 점차 성능이 감소했다. Z-Journal은 모든 경우에 JBD2보다 훨씬 더 좋은 성능을 보였다. 그러나 코어 수가 증가함에 따라 코어 사이의 블록 공유가 급격히 증가하였고, 트랜잭션의 순서 보전을 위한 오버헤드로 인해 Z-Journal과 JBD2의 성능 격차가 점차 줄어들었다.

그래서 우리는 블록의 공유가 줄었을 때의 Z-Journal과 JBD2에서의 성능 경향을 확인하기 위해 간단히 모든 코어가 최대한 자기만의 블록 그룹에 블록들을 할당하도록 수정(low sharing)하여 같은 실험을 반복하였다. JBD2의 경우 블록의 공유가 줄어들었음에도 성능에 차이가 없었으나, Z-Journal의 경우 순서 보전의 오버헤드가 줄어 성능이 향상하였으며 코어 수가 증가하는 것에 따라 성능이 점차 감소하는 경향 또한 줄어드는 모습을 확인할 수 있었다.

3. F2FS 파일 시스템 확장성

3.1 배경

매니코어 서버는 수백 개의 코어를 가지고 있어 많은 스레드들을 병렬적으로 I/O에 동원할 수 있다. 또한, SSD 등 기존 저장장치 대비 우수한 성능을 가진 고성능 저장장치를 사용한다면 데이터베이스와 같이 대용량의 데이터를 관리하는 I/O 집약적인 응용에서 저장장치의 성능으로 인하여 발생하였던 병목을 제거할 수 있다. 따라서 매니코어 서버의 높은 병렬성과 SSD의 높은 성능 및 내부 병렬성을 활용 가능한 파일 시스템을 사용한다면 I/O 집약적인 응용에서 전체 I/O 처리량이 코어 수에 비례하여 증가하는 확장성을 보일 수 있을 것으로 기대된다. 저장장치의 특성을 고려한 대표적인 파일 시스템으로는 SSD와 같은 블록 디바이스에서 사용되는 F2FS[2]가 있다. F2FS는 SSD의 특성을 고려하여 만든 로그 구조 파일 시스템이다. F2FS는 기존 로그 구조 파일 시스템의 wandering tree 문제를 해결하였다. 또한, hot/cold 데이터의 분리를 적용하여 각자 다른 로그에 데이터를 저장하여 SSD 내부 병렬성을 높이고 Garbage Collection의 성능을 향상시켰다.

그러나 매니코어 서버에서 F2FS 파일 시스템을 사용하여도 F2FS의 매니코어 환경을 고려하지 않은 디자인으로 인해서 I/O가 병렬적으로 실행되지 못하여

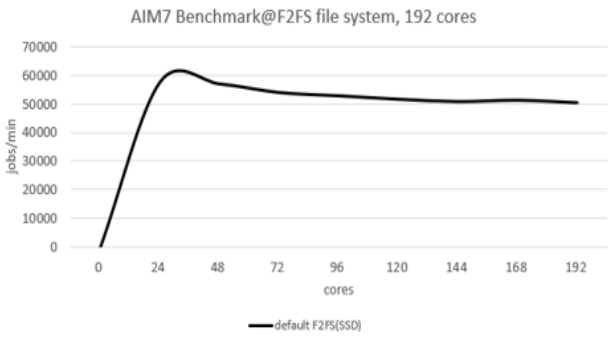


그림 3 매니코어 서버에서 F2FS의 AIM7 처리량

확장성을 보이지 못한다. 그림 3은 복합적인 패턴을 가지고 매니코어 서버에서 파일 시스템의 확장성을 평가할 수 있는 AIM7 워크로드[18]에서 F2FS의 성능이 확장성을 보이지 못하는 모습을 보여준다. 이때 F2FS가 I/O가 병렬적으로 실행하지 못하는 병목은 두 가지가 있다. 첫째는 여러 스레드가 각각의 서로 다른 파일에 대해 I/O를 수행하는 경우이다. 이 경우 스레드들이 각각의 파일 시스템 자원을 사용하므로 I/O를 병렬적으로 수행하여 확장성을 기대할 수 있다. 그러나 F2FS 파일 시스템 내부 구조로 인해 파일 I/O에 fsync가 주기적으로 수반되는 경우 추가적인 I/O가 발생한다. fsync가 호출될 때, 파일 시스템의 데이터 일관성을 위해 체크포인팅을 호출한다. 이때 일관성을 위해 다른 I/O들을 막아 성능 저하의 원인이 된다. 둘째, 여러 스레드가 공유 파일에 대해 I/O를 수행하는 경우이다. 이 경우, 여러 스레드가 파일 시스템의 자원을 공유하므로 확장성을 보이지 않는다. F2FS에서는 I/O 수행 시 파일 단위의 inode mutex 잠금으로 인해 동일 파일에 대해 병렬적으로 실행되는 I/O들이 직렬화된다. 따라서 본 논문에서는 매니코어 환경에서 F2FS의 확장성을 위한 디자인인 hybridF2FS를 제안한다. hybridF2FS는 다음과 같은 세 가지 기술을 포함한다. 첫째로, NVM과 DRAM의 하이브리드 메모리 구조를 이용하여 체크포인팅 시 디스크로 flush되는 메타데이터들을 NVM에 flush하는 NVM 노드 로깅 [10]이다. 둘째, 여러 쓰레드들이 공유 파일 I/O할 때, 서로 겹치지 않는 영역에 대해 병렬적으로 I/O를 할 수 있도록 하는 범위 잠금[11]이다. 셋째, 여러 쓰레드들이 병렬적으로 파일 메타데이터에 접근 할 수 있도록 하는 nCache[11]이다.

3.2. 디자인

3.2.1 노드 로깅

F2FS에서 fsync가 호출되면 체크포인팅을 위한 추가 I/O가 발생하며 파일 시스템의 일관성을 위해 다른 모든 I/O들이 블록되게 된다. hybridF2FS는 이와 같은 오버헤드를 개선하기 위하여 비휘발성 메모리(non-volatile memory, NVM)를 DRAM과 함께 하이브리드 메모리 구조로 사용한다. NVM은 DRAM 수준의 처리량 및 지연시간을 가지며 전원이 꺼져도 데이터가 보존되는 비휘발성 메모리이다. hybridF2FS는 NVM 노드 로깅을 사용하여 데이터 로그는 SSD에 저장하며 노드 로그 및 파일 시스템 메타데이터를 NVM에 저장한다. 기존 F2FS에서 노드 로그와 파일 시스템 메타데이터를 NVM에 저장하여 쓰기 및 체크포인팅 시간을 개선하여 fsync로 인한 지연시간을 감소시킬 수 있다.

그림 4는 기존 F2FS와 NVM 노드 로그의 fsync 과정을 보여준다. fsync 시 기존 F2FS에서는 ① 데이터 로그를 저장하고, 일관성 보호를 위해 다른 I/O를 블록하면서 체크포인팅을 통해 ② 노드 로그, ③ 파일 시스템 메타데이터 순서대로 저장한다. NVM 노드 로깅 역시 동일한 순서로 수행하나 노드 로그와 파일 시스템 메타데이터를 NVM의 빠른 쓰기 속도를 이용해 NVM에 저장한다. NAT에 수행되는 노드의 블록 주소 업데이트를 즉시 NVM에 저장하므로 체크포인팅 시 발생하는 쓰기를 감소시켜 전체 체크포인팅 시간을 감소시킨다. 따라서 fsync 시 발생하는 체크포인팅 시간을 감소시키므로 fsync의 지연시간 또한 감소하며 다른 I/O를 블록하는 시간 또한 감소한다.

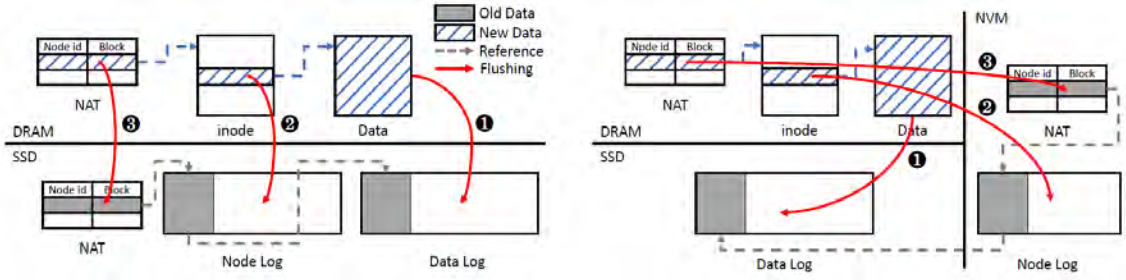


그림 4. 기존 F2FS(좌)와 NVM 노드 로깅 기법(우)의 fsync 과정 비교(참고문헌 [10]의 그림)

3.2.2 범위 잠금

여러 스레드가 병렬적으로 공유 파일에 I/O를 수행하는 경우 스레드의 I/O 수행 영역과 관계없이 파일의 inode의 mutex 잠금으로 인해 직렬화가 발생한다. 기존 F2FS에서는 I/O 영역이 겹치지 않더라도 쓰레드들이 병렬적으로 실행되지 못하고 직렬화된다. 이로 인해 공유 파일에 I/O를 수행하는 경우 치명적인 병목이 발생한다. 따라서 이를 개선하기 위해 인터벌 트리 기반 범위 잠금을 제안하였다. 인터벌 트리 기반 범위 잠금에서는 I/O를 수행하는 영역을 포함한 노드를 인터벌 트리에 삽입한다. 이때, 현재 영역과 겹치는 노드가 없다면 I/O를 할 수 있다. 그러나 인터벌 트리 기반 범위 잠금의 사용으로 I/O 성능은 개선되었으나 I/O가 빈번하게 발생하는 매니코어 환경에서는 인터벌 트리에 노드를 삽입/삭제 시 사용되는 mutex 잠금으로 인해서 확장성 있는 성능을 보여주지 못한다. 따라서 인터벌 트리 대신 원자성 연산 기반 범위 잠금을 개발하였다. 원자성 연산 기반 범위 잠금 사용 시 파일 내의 I/O 여부를 포함한 비트맵을 할당한다. 비트맵에 원자성 연산을 이용하여 현재 스레드가 I/O를 수행하려는 영역에 다른 스레드가 I/O를 하고 있는지 체크한다. 따라서 mutex로 인한 잠금 경합을 해결하였다.

그림 5는 범위 잠금을 적용한 F2FS에서 3개 쓰레드(T1, T2, T3)들이 공유 파일에 I/O를 수행하는 모습이다. 우선 스레드 T1가 파일의 시작 주소 0부터 4KB 만큼의 범위에 I/O를 수행한다. 겹치는 영역에 다른

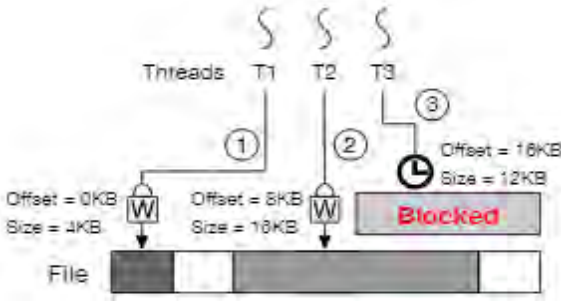


그림 5 범위 잠금을 적용한 F2FS의 I/O (참고문헌[11]의 그림)

스레드가 없으므로 T1은 블록되지 않고 I/O를 수행한다. 스레드 T2도 마찬가지로 시작 주소 8KB부터 16KB만큼의 범위에 I/O를 수행한다. 그 후 스레드 T3이 시작 주소 16KB부터 12KB만큼의 범위에 I/O를 수행하려고 하지만 T2와 범위가 겹치므로 블록된다. T2가 끝나면 T3이 I/O를 시작한다. 이를 통해 데이터 일관성을 지키면서 공유 파일 I/O를 병렬적으로 수행한다.

3.2.3 nCache

원자성 기반 범위 잠금을 도입하여 매니코어 서버에서 F2FS의 공유 파일 I/O 성능을 개선하였지만, 저장장치의 대역폭까지의 성능이 확장하지 못했다. 페이지 캐시 내부의 노드에 접근할 때, 상위 노드부터 탐색을 시작하여 자식 노드를 반복적으로 탐색한 뒤, 원하는 노드에 접근하게 된다. 이때 페이지 캐시에서 노드들의 일관성을 보호하기 위해 mutex 잠금이 사용된다. F2FS의 파일 메타데이터들에 접근 시, 페이지 캐시를 통해서 접근하므로 매니코어 서버에서 공유 파일 I/O를 할 때는 빈번하게 페이지 캐시 내부의 같은 노드들에 접근하게 된다. 이 mutex의 잠금 경합으로 인해 쓰레드들이 직렬화되어 확장성을 저해하였다. 페이지 캐시에서는 노드가 mutex 잠금으로 보호되어 있어 데이터 일관성을 해치지 않는 읽기 작업 시에도 한 스레드만 노드에 접근할 수 있다. 더욱이, 페이지 캐시의 노드는 파일 쓰기 작업 시에도 파일 메타데이터를 읽기 위해서 접근하게 되므로 노드 읽기로 인한 잠금 경합은 확장성을 크게 감소시킨다. 이를 해결하기 위해 nCache를 제안한다.

nCache는 페이지 캐시와 같은 역할을 하지만 R/W 세마포어로 구현되었다. F2FS가 특정 노드에 처음 접근하면 해당 노드는 nCache에 캐싱된다. 이후 캐싱된 노드에 대한 읽기/쓰기 작업은 페이지 캐시가 아닌 nCache에서 수행된다. nCache는 페이지 캐시와 다르게 R/W 세마포어를 사용하여 노드를 보호하여 노드를 동시에 읽는 작업을 병렬적으로 수행 할 수 있다. 공유 파일 I/O에서 범위 잠금과 nCache를 동시에 적

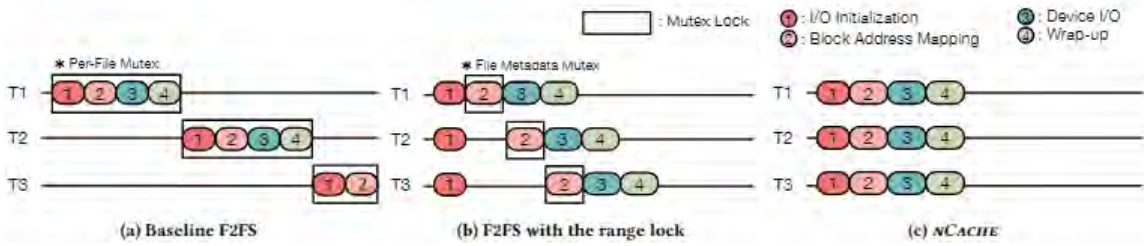
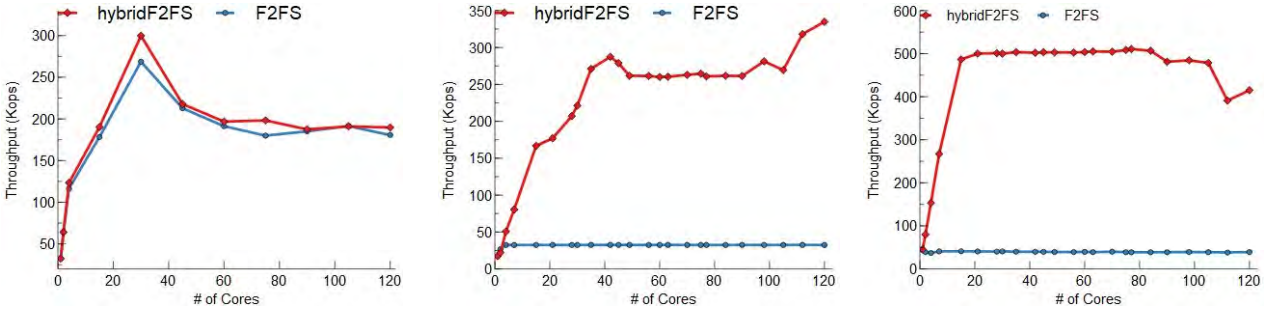


그림 6 F2FS(a), 범위 잠금을 사용한 F2FS(b), (b)에 nCache를 적용한 F2FS(c)의 파이프라이닝(참고문헌 [11]의 그림)

표 2 실험을 위한 환경설정

	Testbed I	Testbed II
Processor	Intel Xeon E7-8870 v2 (15cores × 8)	Intel Xeon E7-8890 v4 (24cores × 8)
Memory	740GB	740GB
Storage	Samsung EVO 970 250GB	Intel SSD 750
OS	Linux 5.7	Linux 5.7



(a) FxMark DWSL 실험결과.
(참고문헌 [10]의 결과 재구성)

(c) FxMark DRBM 실험결과
(참고문헌 [11]의 결과 재구성)

(c) FxMark DWOM 실험결과
(참고문헌 [11]의 결과 재구성)

그림 7 hybridF2FS의 FxMark 실험 결과

용하면 그림 6과 같이 공유 파일 I/O에서 파이프라이닝을 최적화할 수 있다. 기존 F2FS는 공유 파일 I/O 시 inode mutex 잠금에 의해 병렬적으로 수행이 불가능한 반면, 범위 잠금만 사용 시에 I/O 시작은 동시에 가능하며 페이지 캐시의 노드에 접근할 때의 페이지 캐시 내부의 mutex 잠금에 의해 I/O 주소 매핑이 직렬화된다. 반면 범위 잠금과 nCache 사용 시 병렬적으로 nCache에 접근하여 주소 매핑이 가능하므로 모든 I/O를 병렬적으로 실행할 수 있다.

3.3. 성능평가

표 2는 본 논문에서 제안한 기법들의 성능평가를 위한 환경이다. 매니코어 환경에서의 확장성을 평가하기 위해서 FxMark[8]를 사용해 Testbed I에서 평가하였다. 그림 7 (a)는 매니코어 서버에서 NVM 노드 로깅을 적용한 F2FS 실험 결과이다. FxMark 벤치마크 도구의 DWSL 워크로드는 각 스레드가 각자의 파일에 쓰기 작업을 수행하며 쓰기 작업이 끝난 뒤에는 fsync가 동반되는 업무량이다. hybridF2FS는 NVM 노드 로깅을 사용하여 체크포인팅 오버헤드가 감소하였기 때문에 fsync의 전체 지연시간이 감소하였다. 따라서 fsync를 동반한 쓰기 시 전체 처리량 또한 증가하였다. 그림 7 (b)와 (c)는 매니코어 서버에서 F2FS의 FxMark 공유 파일 읽기/쓰기인 DRBM, DWOM 실험 결과이다. 기존 F2FS의 경우 inode mutex 잠금으로 인해 모든 I/O가 직렬화되어 확장성이 없다. hybridF2FS는 원자성 연산 기반

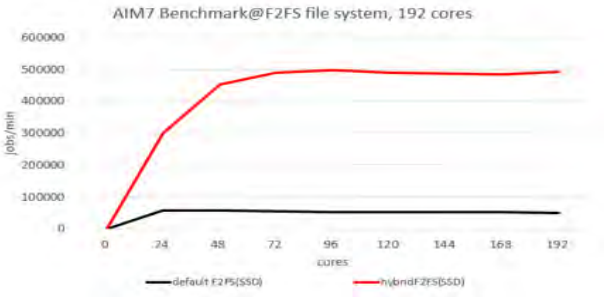


그림 8 매니코어 서버에서 F2FS의 AIM7 워크로드 성능

범위 잠금을 통해 I/O 범위가 겹치지 않는 스레드들의 병렬 실행을 가능하게 하고 nCache의 사용으로 페이지 캐시 내부의 잠금 경합을 해결하였다. 따라서 읽기와 쓰기에서 저장장치의 대역폭까지 확장성을 보였다.

그림 8은 hybridF2FS의 매니코어 서버인 Testbed II에서의 AIM7[18] 성능 그래프이다. 기존 F2FS는 fsync시 발생하는 오버헤드와 파일 읽기/쓰기를 병렬적으로 수행하지 못하기 때문에 확장성을 보이지 못한다. 반면, hybridF2FS는 fsync시 발생하는 오버헤드를 감소시켰으며 공유 파일 읽기/쓰기를 병렬적으로 수행 가능하므로 192코어까지의 확장성을 보인다.

4. NOVA 파일 시스템 확장성

4.1 배경

비휘발성 메모리(NVM, Non-Volatile Memory)는 기존

블록 기반 디바이스에 비해 낮은 지연시간과 높은 대역폭을 제공한다. 또한, DRAM보다 큰 용량과 데이터의 영구성을 보장하므로 차세대 저장장치로 활용하기 위한 파일 시스템 연구가 활발히 진행되고 있다. 그중 NOVA 파일 시스템은 높은 I/O 처리율을 보이며 데이터 원자성을 보장하는 대표적인 NVM 파일 시스템이다. NOVA는 로그 구조의 디자인을 사용해 파일 데이터와 메타데이터의 원자성을 보장한다. 하지만 NOVA 파일 시스템은 PM의 고성능을 최대한 활용하기 위해 설계되었음에도 불구하고 NUMA 기반의 매니코어 환경에서 확장성을 갖지 못하는 문제가 있다.

그림 9는 NOVA 파일 시스템에서 여러 I/O 쓰레드들이 동시에 같은 파일을 업데이트하려는 상황이다. 첫 번째, NOVA 파일 시스템은 NUMA 인지적이지 못하다. 그러므로 모든 파일의 데이터와 메타데이터를 단일 노드 PM 공간에 배치한다. 즉, 노드 1에서 실행 중인 스레드들은 노드 0에 있는 파일에 접근하기 위해 원격 메모리 접근을 통해 접근해야 하므로 I/O 성능이 크게 저하된다. 두 번째, NOVA 파일 시스템은 per-inode 로그 구조를 사용하는데, 로그 구조가 락으로 보호되기 때문이다. 따라서 공유 파일을 쓰기 하는 스레드들은 락에 의해 직렬화되어 매니코어 환경에서 확장성을 갖지 못하는 병목에 주요 원인이 된다.

본 학회지 해당 장에서는 기존 NOVA 파일 시스템을 NUMA 기반의 매니코어 서버에서 확장성을 가질 수 있도록 NUMA 인지적으로 개선한 pNOVA 파일 시스템에 대해 설명한다. pNOVA 파일 시스템이 NUMA 기반의 매니코어 환경에서 확장성을 가질 수 있도록 하기 위해 제안한 기술 3가지를 다음 절에서 설명한다.

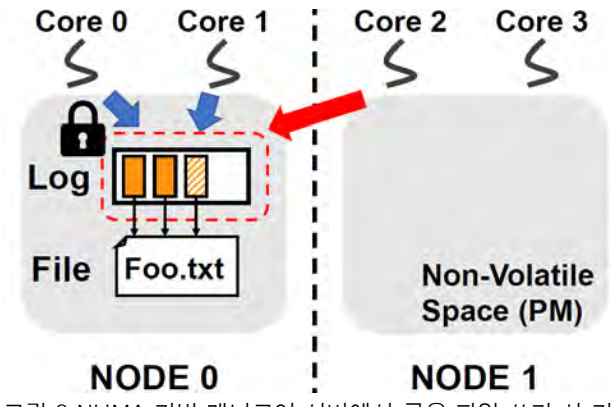


그림 9 NUMA 기반 매니코어 서버에서 공유 파일 쓰기 시 기존 NOVA 파일 시스템의 확장성 문제[12]

4.2 디자인

4.1.1 세그먼트 범위 기반 락

pNOVA 파일 시스템은 기존 NOVA 파일 시스템의 획일적인 락을 범위 기반 락으로 대체하였다. pNOVA 파일 시스템의 범위 기반 락은 파일을 세그먼트라는 단위로 나누고 각 세그먼트들을 관리하는 락 변수를 두어 읽기/쓰기 범위에 따라 해당 세그먼트들만 락을 한다. 그림 10는 pNOVA의 세그먼트 범위 기반 락의 구조이다. 파일을 연속적인 페이지들의 그룹으로 생각했을 때, 우선 연속적인 페이지들을 세그먼트라는 단위로 묶는다(그림에서는 두 페이지를 하나의 세그먼트로 정의함). 각 세그먼트는 32-bit의 세마포어를 가지는데, 세마포어의 가장 왼쪽 bit는 쓰기 플래그로 사용되며 남은 31-bit는 읽기 카운터로써 사용된다.

스레드들은 읽기/쓰기 범위에 따라 해당 세그먼트의 세마포어에 접근한다. 만약 가장 왼쪽의 bit가 1일 경우 해당 세그먼트에 대해 특정 스레드가 쓰기 중이므로 이후에 접근하는 읽기/쓰기를 블록킹한다. 만약 세마포어의 값이 0이 아니면서 가장 왼쪽 bit가 1도 아닌 경우 특정 쓰레드들이 읽기 중이므로 들어오는 읽기 동작에 대해서는 카운터를 증가시키고, 쓰기 동작에 대해서는 블록킹을 한다.

세그먼트의 세마포어를 변경하는 동작들은 모두 하드웨어가 지원하는 atomic-operation을 사용해 원자성을 보장한다. 또한, pNOVA 파일 시스템의 범위 기반 락은 세그먼트마다 참조 계수 카운터를 가지고 있기 때문에 다른 범위의 읽기 동작에 대해서는 참조 계수 카운터 문제가 발생하지 않는다. 병렬적 쓰기 동작에 대해서는 각각의 스레드가 쓰기 범위에 해당하는 세그먼트만 락을 걸어 범위가 다른 경우 서로에 의해 블락킹되지 않는다.

4.2.2 NVM 장치 가상화

pNOVA 파일 시스템에서는 NUMA 인지적인 파일 시스템 설계하기 위해 그림 11과 같이 여러 NUMA 노드에서 연속되지 않은 NVM 주소 공간들을 연속적인 단일 가상 주소 공간으로 가상화하였다. 이를 위해 파일 시스템을 마운트 할 때 슈퍼 블록으로 모든 NUMA 노드에서 NVM 장치의 정보들을 가져온다. 결과적으로 NVM 장치의 물리적 주소가 선형으로 매핑되어 NUMA 인지적인 파일 시스템 사용자 데이터와 메타데이터를 여러 NVM 장치에 쉽게 배치할 수 있다. 이때 원격 메모리 접근을 완화하기 위해 메모리 할당 정책이 필요하다. 따라서 pNOVA 파일 시스템에서는 NUMA 인지적인 NOVA 파일 시스템에서 스레

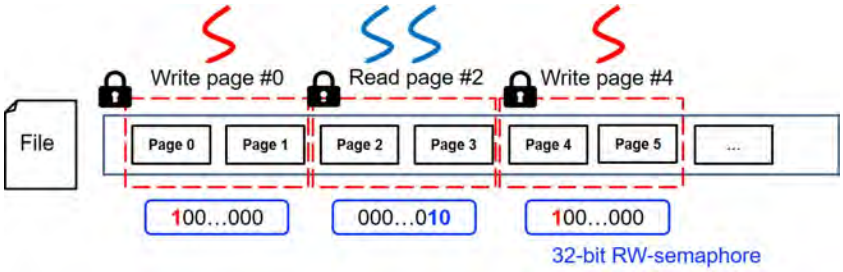


그림 10 세그먼트 범위 기반 락의 구조

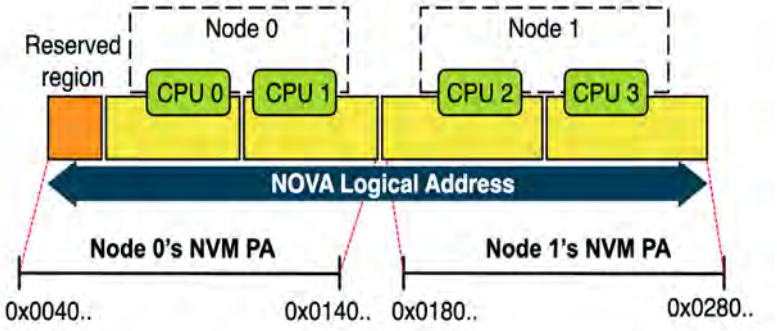


그림 11 NOVA 파일 시스템에서 NUMA 인지적인 NVM 장치 가상화 예시[13]

드가 로컬 NVM 장치에 파일을 먼저 쓰도록 하는 "Local Write First" 정책을 도입했다.

NUMA 인지적인 NOVA 파일 시스템은 NVM 장치의 크기에 따라 전체 논리 주소 공간을 분할하고 NUMA 노드 별 파티션을 생성한다. 또한, 파티션은 각 노드의 코어 수로 나뉘며 NOVA의 CPU 당 메모리 할당자에 의해 관리된다. 각 스레드의 파일 데이터 및 로그 페이지는 스레드가 실행 중인 코어에 할당된 영역에서 할당된다. 그림 11로 예를 들어 CPU-0 및 CPU-1에서 실행되는 스레드는 0x0040~0x0140 범위의 물리적 주소 공간에 파일을 배치하는 것을 선호하는 반면 CPU-2 및 CPU-3에서 실행되는 스레드들은 0x0180~0x0280 범위에 파일을 우선적으로 배치한다. 결론적으로 각 I/O 스레드는 우선적으로 로컬 NVM 공간에서 데이터 또는 로그 페이지를 할당받을 수 있다.

4.2.3 per-Core 로그 구조

여러 스레드 간에 공유되는 파일 데이터 구조는 NUMA 기반 시스템에서 파일 시스템의 확장성 병목 현상의 주요 원인이 된다. 서로 다른 NUMA 노드에서 실행되는 여러 스레드 간에 파일을 공유하면 잦은 원격 메모리 접근으로 인해 성능이 크게 저하된다. 또한, 공유자원은 일관성을 보장하기 위해 락을 통해 보호되므로 동시 업데이트 작업이 직렬화된다. 따라서 pNOVA 파일 시스템은 확장 가능한 NOVA 파일 시스템을 위한 per-core 데이터 구조를 제안하고 기존 NOVA 파일 시스템의 per-inode 로그 구조를 per-core 로그로 확장했다.

그림 12는 pNOVA 파일 시스템에 적용된 per-core 로그 구조를 보여준다. 여기서 두 가지 새로운 영구적

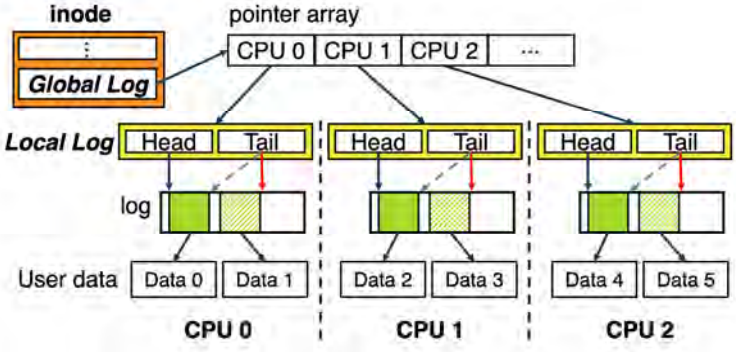


그림 12 per-Core 로그 구조를 적용한 pNOVA 파일 시스템[13]

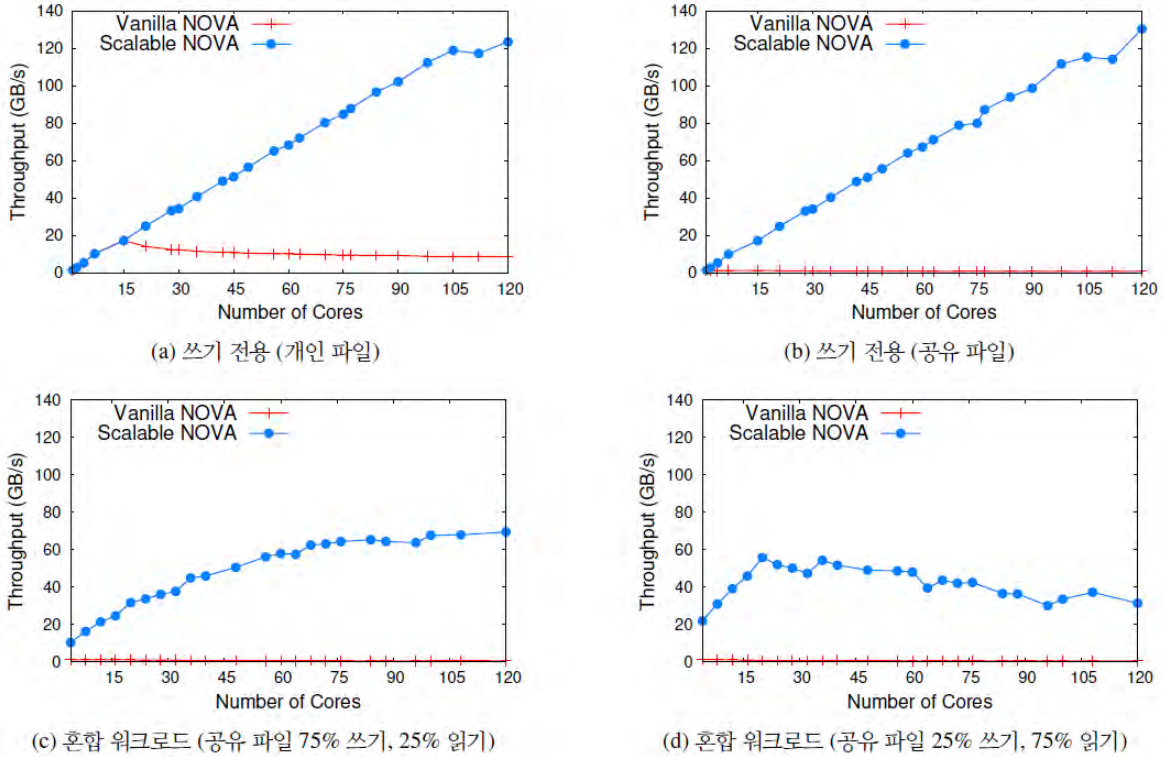
데이터 구조인 global log 및 local log를 소개한다. global log는 (그림 12)에 표시된 것처럼 inode에 포함되며 각 CPU의 local log를 인덱싱하는 포인터 배열이다. Local log는 per-core 로그에 대한 헤드 및 테일 포인터가 있는 CPU 별 개인 데이터 구조이다. 이 두 데이터 구조를 통해 서로 다른 NUMA 노드의 여러 스레드가 파일 I/O 동작을 동시에 수행하고 확장성을 향상시킬 수 있다.

4.3 성능평가

FxMark 벤치마크를 사용하여 최대 120코어까지 쓰레드들이 동시적으로 개인 파일 쓰기(DWOL), 공유 파일 쓰기(DWOM)를 수행하는 두 워크로드에 대해 실험을 진행하였다. 그림 13(a)의 개인 파일 쓰기의 경우 기존 NOVA 파일 시스템의 I/O 처리량은 15코어에서 최대 17.15 GB/s까지 증가하였다 이후로 감소하기 시작한다. 이는 기존 NOVA 파일 시스템이 모든 파일 데이터가 단일 노드에 배치하여 서버의 NUMA 바운더리인 15코어 이후로 쓰레드들이 원격 메모리 접근을 통해 쓰기를 수행하기 때문이다. 특히 기존 NOVA 파일 시스템은 그림 13(b)의 공유 파일 쓰기 시 파일 시스템 락에 의해 전혀 성능 확장성을 보이지 못한다. 기존 NOVA 파일 시스템은 1코어에서 1.29 GB/s의 I/O 처리량을 제공하며 코어 수가 증가할수록 성능이 약간 하락한다. 반면 pNOVA 파일 시스템의 I/O 처리량은 두 워크로드에서 모두 코어 수가 증가함에 따라 선형적으로 증가하였다. 그림 13 (c)와 (d)는 공유 파일 쓰기와 읽기를 혼합한 워크로드에 대한 실험 결과이다. 그림 13 (c)는 쓰기 스레드의 비율이 75%, 읽기 스레드의 비율이 25%일 때의 실험 결과이다. 기존 NOVA 파일 시스템은 마찬가지로 성능 확장성을 보이지 못한다. 반면 pNOVA 파일 시스템의 I/O 처리량은 코어 수가 증가함에 따라 선형적으로 증가하였지만, 쓰기 전용 워크로드들에 비해 증가 폭이 높지는 않았다. 이는 쓰기 쓰레드들이 지역 노드에 우선적으로 배치한 파일 데이터나 로그에 대해서 다른 노드들의 읽기 쓰레드들이 원격 메모리 접근을 통한 성능 감소가 있기 때문이다. 그림 13 (d)는 쓰기 스레드의 비율이 25%, 읽기 스레드의 비율이 75% 일 때의 실험 결과이다. pNOVA 파일 시스템의 I/O 처리량이 15코어 이후로 성능이 감소한다.

표 3 실험 테스트베드

CPU	Intel(R) Xeon(R) E7-8870 v2 2.3GHz CPU 소켓(노드) 수(#) : 8 소켓(노드) 당 코어 수(#) : 15 총 코어 수(#) : 120
Memory	DDR3, 96GB * 8개 (=968GB)
PM	32GB(DRAM 기반 에뮬레이션)



(a) 쓰기 전용 (개인 파일)

(b) 쓰기 전용 (공유 파일)

(c) 혼합 워크로드 (공유 파일 75% 쓰기, 25% 읽기)

(d) 혼합 워크로드 (공유 파일 25% 쓰기, 75% 읽기)

그림 13 120코어 서버에서 FxMark 쓰기 워크로드에 대한 기존 NOVA 파일 시스템과 pNOVA 파일 시스템의 확장성 평가[12]

읽기 스레드의 비율이 높아짐에 따라 원격 메모리 접근으로 인한 성능 저하가 심해짐을 확인할 수 있다.

5. 리눅스 블록 계층 확장성

리눅스의 블록 계층은 HDD의 동작 특성, 즉 회전하는 원판 위에서 서보 암을 움직여 데이터를 읽고 쓰는 구조를 고려하여 개발되어 왔다. 최근 들어 SSD가 널리 보급됨에 따라 리눅스의 블록 계층도 이를 반영하여 다중 큐(blk-mq)[13], 하이브리드 폴링[14] 등 SSD의 동작 특성에 최적화된 새로운 기능을 도입하기 시작하였다. 본 장에서는 리눅스 블록 계층의 매니코어 확장성 분석을 위한 고려 사항 및 본 연구진이 최근 수행한 블록 계층 매니코어 확장성 연구[15-17]를 소개한다.

5.1 다중 큐(blk-mq) 확장성 분석

최근의 컴퓨터 하드웨어는 내부 병렬성을 강화하는 방향으로 발전하고 있다. 4코어 이상의 멀티코어 CPU가 일반화되는 한편 120코어 이상의 CPU를 가진 매니코어 시스템도 등장하였다. 저장장치 분야에서도 다수의 플래시 칩으로 구성되어 병렬로 읽기 및 쓰기가 가능한 SSD가 HDD를 대체하고 있다.

SSD가 도입되면서 리눅스의 단일 큐 기반 블록 계층은 SSD의 병렬성을 활용하지 못하는 한계를 드러내었다. 다수의 CPU 코어가 SSD를 동시 접근할 때 발생하는 큐 잠금이 성능 저하의 주요인으로 밝혀짐에 따라 이를 극복하기 위하여 소프트웨어 큐와 하드웨어 큐의 2단계로 구성된 다중 큐가 새롭게 도입되었다.

다중 큐는 CPU 코어마다 1개의 소프트웨어 큐를 할당하며 하드웨어 큐를 저장장치의 장치 큐 개수만큼 생성하여 각 하드웨어 큐에 1개 이상의 소프트웨어 큐를 사상한다. I/O 병합, I/O 재정렬 등 큐 잠금이 필요한 작업은 소프트웨어 큐에서만 처리하도록 한다. 따라서 소프트웨어 큐가 하드웨어 큐로 I/O를 전달할 때 큐 잠금이 필요치 않다. 이러한 큐 무잠금 구조는 SSD의 내부 병렬성의 효과적 활용을 가능케 한다.

다중 큐의 매니코어 확장성을 평가하기 위한 방법으로 다음 방식을 고려할 수 있다.

가상 블록 장치(null block device): 가상 블록 장치는 하드웨어 큐가 제출한 I/O 요청에 대한 완료 신호를 즉시 반환하는 루프백(loopback) 방식으로 구현되어 있다. 따라서 저장장치의 성능 병목에 제한을 받지 않기 때문에 CPU 코어 수가 많은 매니코어 시스템에 대한 확장성 평가가 용이하다. 가상 블록 장치를 이용하여 80코어 시스템의 매니코어 확장성을 검증한 연구 결과가 소개되기도 하였다[13]. 하지만 가상 블록 장치에서는 데이터 입출력 및 저장이 실제로 발생하지 않기 때문에 이 과정에서 발생할 수 있는 확장성 저하를 파악하기 어렵다. 또한, 파일 시스템 설치가 불가능하여 AIM7[18] 등의 파일 시스템 기반 매크로 벤치마크를 사용할 수 없다.

실제 저장장치: NVMe SSD와 같은 실제 저장장치를 사용하면 데이터 입출력의 전 과정을 실제로 수행하면서 확장성 분석을 수행할 수 있다. 하지만 저장장치의 입출력 성능 한계로 인해 확장성 평가가 불가능한 단점 또한 존재한다. 예를 들어 활성 CPU 코어 수를 늘림에 따라 IOPS(초당 I/O 개수) 측정치가 저장장치의 사양 상 최대 IOPS에 수렴할 경우 그 원인이 저장장치의 성능 한계인지 아니면 다중 큐의 확장성 한계인지 판단하기가 어렵다. 또한, 실제 저장장치의 하드웨어 큐 개수가 고정되어 있어 하드웨어 큐 개수에 따른 확장성 검증 실험에 제약이 따른다.

메모리 기반 파일 시스템: tmpfs와 같은 메모리 기반 파일 시스템은 가상 블록 장치보다는 느리지만 플래시 기반 SSD에 비해 매우 빠른 입출력 속도를 가진다. 따라서 활성 CPU 코어 수를 충분히 늘리면서 다중 큐에 대한 확장성 평가를 수행할 수 있다. 하지만 메모리 기반 파일 시스템은 입출력 성능 최적화를 위해 블록 계층을 경유하지 않고 메모리에 직접 입출력을 수행한다. 따라서 다중 큐 확장성 분석에 사용할 수 없다.

블록장치 기반 램디스크: tmpfs는 메모리에 직접 입출력을 수행하는 반면, 블록장치 기반 램디스크는 블록 계층에 연결되는 가상 블록 장치를 모사한다. 따라서 사용자는 직접 원하는 파일 시스템을 선택하여 설치할 수 있고 모든 데이터 입출력은 블록 계층을 거치게 된다. 대표적인 블록장치 기반 램디스크로는 RapidDisk[19]가 있는데, 단일 큐 기반으로 구현되어 다중 큐 확장성 분석에는 사용할 수 없다.

5.2 ramdisk_mq 기반 다중 큐 확장성 분석

본 연구진은 최근 수행한 연구[15]에서 다중 큐에 연결되는 블록장치 기반 램디스크인 ramdisk-mq를 구현하였다. ramdisk-mq의 주요 특징은 다음과 같다.

- 가상 블록 장치를 모사(emulation)
- 크기 설정, 실제 데이터 입출력 및 저장 지원
- 다중 큐 사용
- 블록 장치 큐 개수 변경 가능

ramdisk-mq를 사용하여 120코어가 장착된 매니코어 머신[1]에 대해 활성 CPU 코어 개수 변화에 따른

1) 상세 사양은 [15] 참조

다중 큐의 IOPS를 측정하였다. 그 결과 다음 두 가지 변수가 다중 큐의 확장성에 영향을 미치는 주요인으로 확인되었다.

- 다중 큐 설정 변수 중 nr_request의 크기
- ramdisk-mq의 장치 큐 개수

첫째, nr_request는 다중 큐 내부의 각각의 하드웨어 큐가 관리할 수 있는 I/O 요청의 최대 개수이며, 기본 설정값은 64이다. 둘째, 장치 큐 개수는 저장장치의 사양에 따른 고윳값이어서 임의 변경이 불가능하다. 하지만 ramdisk-mq는 가상 블록 장치이기 때문에 이 값을 자유롭게 설정할 수 있다. 이때 다중 큐의 하드웨어 큐 개수는 ramdisk-mq의 장치 큐 개수와 동일하게 설정된다.

그림 14는 nr_request가 1024일 때 하드웨어 큐 개수를 15개 이상으로 늘리면 매니코어 확장성이 확보됨을 보여준다. 하지만 하드웨어 큐가 8개인 경우 활성 CPU 코어가 40개를 넘어서면서 성능개선이 더뎌지며, 60개 이상에서는 성능이 오히려 저하되었다.

nr_request가 확장성에 미치는 영향을 확인하기 위해 하드웨어 큐 개수를 8개로 고정하고 nr_request 값을 변경하면서 IOPS를 측정하였다. 실험 결과 그림 15와 같이 nr_request 값이 4096 이상일 때 80코어까지 확장성을 확보할 수 있었다.

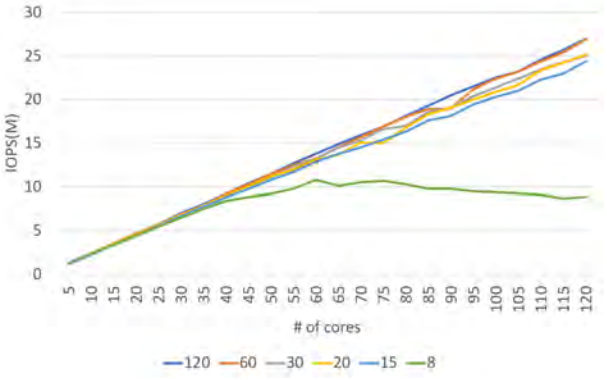


그림 14 하드웨어 큐 개수와 코어 수에 따른 FIO Benchmark I/O 처리량 (nr_requests = 1024)[15]

위의 실험 결과는 다음과 같이 요약할 수 있다.

- 장치 큐 개수가 15개 이상인 경우 120코어까지의 매니코어 확장성을 보장한다.
- 장치 큐가 8개인 경우 다중 큐 기본 설정 상태에서 매니코어 확장성은 60코어까지 확보할 수 있다.
- 이때 NVMe 장치의 nr_request 값을 기본값 1024에서 4096 이상으로 변경할 경우 매니코어 확장성을 80코어로 개선할 수 있다.

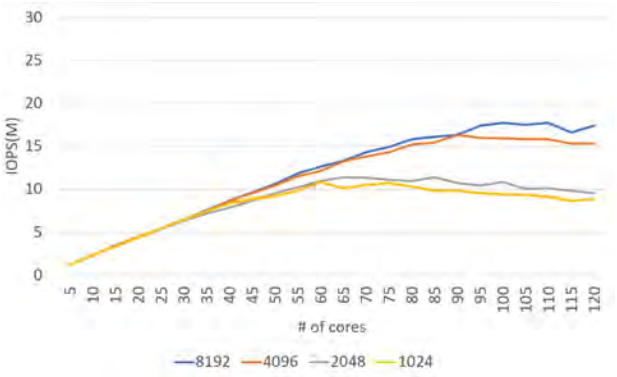


그림 15 nr_requests와 코어 수에 따른 FIO Benchmark I/O 처리량(하드웨어 큐 8개)[15]

5.3 파일 시스템과 블록 계층 간 오버헤드 분석

저장장치 시스템의 매니코어 확장성 확보를 위해서는 저장장치 시스템 구성 요소의 계층별 최적화뿐만 아니라 계층 간 최적화도 고려해야 한다. 본 연구진은 최근 수행한 연구[16]에서 실제 저장장치, 다중 큐, 파일 시스템으로 구성된 저장장치 시스템의 매니코어 확장성 분석을 위해 Samsung 960 Pro SSD에 EXT4와 F2FS 파일 시스템을 각각 설치하고 FIO[20]를 사용하여 임의 읽기 입출력 성능을 측정하였다.

I/O 요청 프로세스 개수 증가에 따른 각 파일 시스템의 IOPS를 측정한 결과는 그림 16과 같다. EXT4 파일 시스템은 8코어부터 960 Pro SSD의 사양 상 최대 성능을 유지하였다. 반면 F2FS는 CPU 코어 수를 늘려도 960 Pro의 최대 성능을 끌어내지 못하는 확장성 한계를 보였다(그림 16 F2FS 변경 전). 특히 15코어 이후 오히려 IOPS가 감소하는 양상을 보였다.

표 4는 I/O 요청 프로세스 개수에 따른 osq_lock CPU 점유율을 측정한 결과이다. 임의 읽기 패턴에 대해 과도한 잠금이 발생하는 원인을 분석한 결과 F2FS에서 입출력 요청을 최종적으로 블록 계층에 전송하는

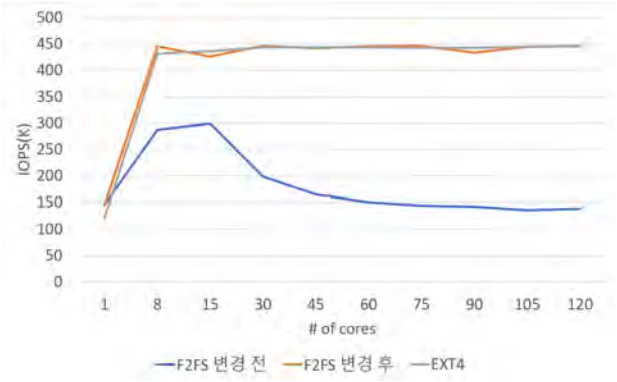


그림 16 IO 요청 프로세스 개수에 따른 파일 시스템별 IOPS 측정 결과[16]

표 4 IO 요청 프로세스 수에 따른 읽기 성능 및 osq_lock CPU 점유율 변화[16]

I/O 요청 프로세스 개수	KIOPS	osq_lock CPU 점유율 비중
8	287.6	54.2%
15	150.6	72.1%

단계에서 쓰기 요청뿐만 아니라 읽기 요청에 대해서도 잠금을 적용한 것을 확인하였다. 따라서 구현을 변경하여 읽기 요청 시에는 잠금을 적용하지 않도록 수정하고 재실험을 수행한 결과 EXT4와 마찬가지로 8코어부터 저장장치의 최대 성능을 끌어낼 수 있었다(그림 16 F2FS 변경 후). 이는 파일 시스템 내부 최적화를 통해 매니코어 확장성을 확보하더라도 계층 간 데이터 전송 과정의 오버헤드로 확장성이 저하될 수 있음을 보여주는 사례다.

5,4 입출력 인터페이스 확장성 분석

최근의 저장장치 기술 발전은 대역폭 증가뿐만 아니라 응답시간 개선 측면에서도 큰 진전을 보이고 있다. 특히 Z-NAND, 3D Xpoint와 같은 소자 기술의 도입으로 수십 마이크로초 이하의 응답시간을 가지는 저지연 SSD가 등장하였다. 이로 인해 기존에는 크게 주목받지 않았던 입출력 인터페이스의 성능 최적화가 중요한 문제로 떠오르게 되었다.

현재 리눅스에서 사용 가능한 주요 입출력 인터페이스의 특징을 표 5에 정리하였다. io_uring, SPDK 등 최근에 제안된 인터페이스는 커널 영역과 사용자 영역 사이의 데이터 복사 오버헤드를 줄이기 위해 입출력 버퍼를 두 영역 간 공유메모리로 설정하거나 아예 사용자 영역에 배치한 것을 볼 수 있다. 또한, 저지연 SSD에서 급격하게 증가하는 인터럽트 처리 오버헤드를 줄이기 위해 폴링 방식을 선택하고 있다.

그림 17은 본 연구진이 최근 수행한 연구[17]에서 Z-NAND 기반 저지연 SSD인 Samsung 983 ZET SSD를 대상으로 4KB 임의 읽기를 수행하면서 이들 입출력 인터페이스의 매니코어 확장성을 분석한 결과이다.

표 5 입출력 인터페이스별 I/O 완료 감지 방식 및 I/O 버퍼 구현 위치

종류	I/O 완료 감지	I/O 버퍼 위치
libaio	인터럽트	커널 영역
pvsync2	폴링	커널 영역
io_uring[21]	인터럽트/폴링 선택 가능	커널/사용자 영역 공유
SPDK[22]	폴링	사용자 영역

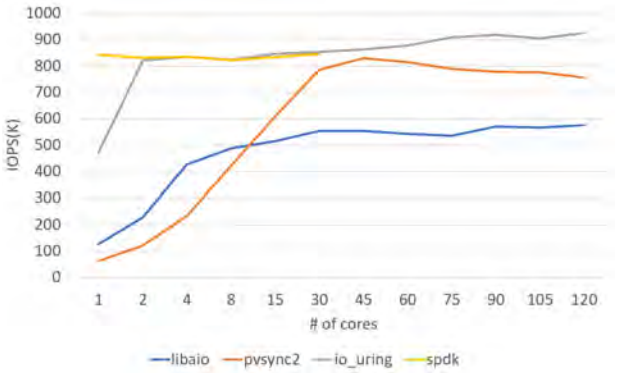


그림 17 Job 개수 증가에 따른 4KB 임의읽기 성능[17]

기존의 libaio나 pvsync2는 job 개수를 충분히 늘리더라도 저장장치의 최대 성능을 끌어내지 못함을 확인할 수 있다. 반면 최근 새롭게 제안된 SPDK와 io_uring 인터페이스는 job 개수가 각각 1개와 2개 이상인 구간에서 저장장치의 최대 성능을 확보하여 매니코어 확장성을 보여준다.

6. 결 론

현재, 기존의 리눅스 파일 시스템들은 매니코어 환경의 특성을 고려하지 못해 매니코어 서버의 성능을 충분히 활용하지 못하며, 저장장치 성능으로 인한 병목을 해결하여도 성능 확장성을 보이지 못한다. 매니코어 서버에서의 기존 리눅스 파일 시스템의 확장성을 제한하는 병목 요인은 어느 한 계층에 국한된 문제가 아니다. 따라서 본 고에서는 현존하는 문제점을 해결하기 위해서 저널 계층 확장성을 위한 Z-Journal, 개별 파일 시스템의 병목을 해결한 hybridF2FS와 pNOVA, 그리고 블록 계층 확장성을 위한 연구들을 살펴보았다.

각 연구에 대한 개별 성과는 매니코어 서버의 파일 시스템 성능 확장성 개선 연구에 도움이 되기 위해 Github(https://github.com/oslab-swrc/jet-journal)(https://github.com/oslab-swrc/hybridF2FS)(https://github.com/oslab-swrc/pNOVA)(https://github.com/oslab-swrc/blk-mq-analyzer)에 코드가 공개되어 누구나 자유롭게 확인, 수정, 배포할 수 있다.

참고문헌

[1] B. Marr. “How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read.” [Online]. 2019. Available: https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-datado-we-create-every-day-the-mind-blowing-stats-everyone-should-read/

[2] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. “F2FS: a new file system for flash storage.” In Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15). USENIX Association, USA, 273-286. 2015.

[3] Jian Xu and Steven Swanson. “NOVA: a log-structured file system for hybrid volatile/non-volatile main memories.” In Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST'16). USENIX Association, USA, 323-338. 2016.

[4] Stephen C Tweedie. “Journaling the Linux ext2fs filesystem.” In The Fourth Annual Linux Expo. Durham, North Carolina, 1998.

[5] Yongseok Son, Sunggon Kim, Heon Young Yeom, and Hyuck Han. “High-performance transaction processing in journaling file systems.” In Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18). USENIX Association, USA, 227-240. 2018.

[6] Jongseok Kim, Cassiano Campes, Jooyoung Hwang, Jinkyu Jeong and Euiseong Seo, “Z-Journal: Scalable Per-Core Journaling.” In proceedings of The 2021 USENIX Annual Technical Conference (ATC’21), 2021.

[7] 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," 2011 IEEE 17th International Symposium on High Performance Computer Architecture, 2011, pp. 266-277, doi: 10.1109/HPCA.2011.5749735. 2011.

[8] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim. “Understanding manycore scalability of file systems.” In Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '16). USENIX Association, USA, 71-85. 2016.

[9] Andrew Wilson. “The new and improved filebench.” In Proceedings of 6th USENIX Conference on File and Storage Technologies (FAST), 2008.

[10] Chang-Gyu Lee, Hyunki Byun, Sunghyun Noh, Hyeongu Kang, and Youngjae Kim. “Write optimization of log-structured flash file system for parallel I/O on manycore servers.” In Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR '19). Association for Computing Machinery, New York, NY, USA, 21-32. 2019.

[11] Chang-Gyu Lee, Sunghyun Noh, Hyeongu Kang, Soon Hwang, and Youngjae Kim. “Concurrent file metadata structure using readers-writer lock.” In Proceedings of the 36th Annual ACM Symposium on Applied Computing (SAC '21). Association for Computing Machinery, New York, NY, USA, 1172-1181. 2021.

[12] 변홍수, 김준형, 김영재, 박성용, "NUMA 인지 PM 파일 시스템의 성능 확장성 평가 및 분석," 한국정보과학회 2020 한국소프트웨어종합학술대회 (KSC 2020), VOL 47 NO. 02 PP. 1584 ~ 1586. 2020.

[13] June-Hyung Kim, Youngjae Kim, Safdar Jamil, and Sungyong Park, "A NUMA-aware NVM File System Design for ManycoreServer Applications," In 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). 2020.

[13] M. Bjorling, J. Axboe, D. Nellans, and P. Bonnet, “ Linux Block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems”, in Proc. of the 6th International Systems and Storage Conference, SYSTOR’13, pp. 22:1-22:10, 2013.

[14] D. L. Moal, “I/O Latency Optimization with Polling”, In Proceeding of USENIX Conference of Linux Storage and Filesystems Conference (VAULT), pp. 1-25, 2017.

[15] 윤명식, 김성곤, 주용수, 임성수, “다중 큐 블록 계층을 이용하는 RAM 기반 블록 디바이스 구현,” 한국정보과학회 학술발표논문집, pp. 1474-1476, 2018.

[16] 김성곤, 주용수, 임성수, 임은진, “리눅스 커널 블록 계층의 매니코어 확장성 분석,” 한국정보과학회 학술발표논문집, pp. 1149-1151, 2019.

[17] 정지현, 서동주, 주용수, 임성수, 임은진, “고성능 NVMe 장치를 위한 입출력 인터페이스의 읽기 성능 분석,” 한국정보과학회 학술발표논문집, pp. 1757-1759, 2020.

[18] AIM Benchmarks, https://sourceforge.net/projects/aimbench

[19] Rapiddisk, https://github.com/pkoutoupis/rapiddisk

[20] J. Axboe, “Fio-Flexible IO Tester,” http://freecode.com/projects/fio, 2014.

[21] J. Corbet, “Ringing in a New Asynchronous I/O API,” https://lwn.net/Articles/776703, 2019.

[22] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul, “SPDK: A Development Kit to Build High Performance Storage Applications,” in 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp. 154-161, 2017.

약 력

황 순

2020 서강대학교 컴퓨터공학과 졸업 (학사)
2020~현재 서강대학교 컴퓨터공학과 석사과정
관심분야 : 시스템 소프트웨어, 스토리지 시스템, 운영체제
Email : hs950826@sogang.ac.kr

김 영 재

2001 서강대학교 컴퓨터공학과 졸업(학사)
2003 KAIST 전산학과 졸업(석사)
2003~2004 한국전자통신연구원 (ETRI) 연구원
2009 펜실베니아주립대학교 컴퓨터공학과 졸업 (박사)
2009~2015 미국 US Department of Energy's Oak Ridge National Laboratory (ORNL) Staff Scientist
2016 ~ 현재 서강대학교 컴퓨터공학과 조교수/부교수
관심분야 : 운영체제, 파일 시스템, 스토리지 시스템, 데이터베이스 시스템, 시스템 보안,분산 시스템
Email : youkim@sogang.ac.kr

변 홍 수

2021 서강대학교 컴퓨터공학과 졸업(학사)
2021~현재 서강대학교 컴퓨터공학과 석사과정
관심분야 : 시스템 소프트웨어, 파일 시스템, 컴퓨팅 스토리지 시스템
Email : byhs@sogang.ac.kr

박 성 용

1987 서강대학교 전자계산학과 졸업(학사)
1987~1992 금성통신연구소(현 LG전자) 연구원
1994 Syracuse University Computer Science 졸업(석사)
1998 Syracuse University Computer Science 졸업(박사)
1998~1999 Bellcore Research Scientis
1999~현재 서강대학교 컴퓨터공학과 교수
관심분야 : 클라우드 컴퓨팅, 고성능 컴퓨팅 시스템, 빅데이터 및 머신러닝 플랫폼
Email : parksy@sogang.ac.kr

김 종 석

2018 성균관대학교 소프트웨어학과 졸업 (학사)
2020 성균관대학교 소프트웨어플랫폼학과 졸업 (석사)
2020~현재 성균관대학교 소프트웨어학과 박사과정
관심분야 : 메모리 관리, 플래시 메모리, 파일 시스템
Email : ks7sj@skku.edu

서 의 성

2000 한국과학기술원 전산학과 졸업 (학사)
2002 한국과학기술원 전산학과 졸업 (석사)
2007 한국과학기술원 전산학과 졸업 (박사)
2007~2009 미국 Pennsylvania State University Research Associate
2009~2012 울산과학기술원 조교수
2012~현재 성균관대학교 교수
관심분야 : 시스템 소프트웨어, 임베디드 시스템, 클라우드 컴퓨팅
Email : euiseong@skku.edu

정 지 현

2020 국민대학교 소프트웨어학부 졸업(학사)
2020~현재 국민대학교 컴퓨터공학과 석사과정
관심분야 : 메모리 및 스토리지 시스템, 임베디드 시스템
Email : jungjh@kookmin.ac.kr

주 용 수

2000 서울대학교 컴퓨터공학과 졸업(학사)
2002 서울대학교 전기컴퓨터공학부 졸업(석사)
2007 서울대학교 전기컴퓨터공학부 졸업 (박사)
2007~2008 서울대학교 전기컴퓨터공학부 BK21 정보기술사업단 박사후연구원
2009~2010 미국 Penn. State Univ. Postdoctoral Research Associate
2010~2015 이화여자대학교 임베디드소프트웨어연구센터 연구교수
2015~현재 국민대학교 소프트웨어학부 부교수
관심분야 : 메모리 및 스토리지 시스템, 임베디드 시스템
Email : ysjoo@kookmin.ac.kr