# 매니 코어 환경에서 F2FS 파일시스템의 단일 파일 I/O 병렬화

노성현, 이창규, 김영재
서강대학교 컴퓨터공학과
{nsh0249, changgyu, youkim}@sogang.ac.kr

# Parallelizing Shared File I/O Operations in F2FS on Manycore Servers

Sunghyun Noh, Changgyu Lee, Youngjae Kim
Department of Computer Science and Engineering, Sogang University, Seoul, Republic of Korea

요 약

SSD는 내부 병렬성을 활용하는 고성능의 저장장치이다. 이에 최적화된 파일시스템인 F2FS를 활용하여 I/O의 병렬 수행을 기대할 수 있다. 이전의 많은 연구들에서는 파일시스템에 Range lock을 적용하여 단일 파일 I/O를 병렬화 하였다. 그러나 매니코어 환경에서 단일 파일 I/O를 수행하는 경우 F2FS는 Range lock을 적용하여도 성능이 확장하지 않는다. 본 논문에서는 이러한 문제점의 원인으로 두가지를 밝혔다. 첫째, 여러 쓰레드가 단일 파일에 대해 I/O를 수행하는 경우 커널 쓰레드가 페이지 캐시에 있는 F2FS의 인덱스 자료구조에 접근하는데 이 때 인덱스 자료구조에서 lock contention이 발생한다. 둘째, Range lock에서 사용하는 Tree에서 lock contention이 발생한다. 본 논문은 이에 대한 해결책으로 다음과 같은 설계를 제안한다. 첫째, 페이지 캐시에서 발생하는 lock contention을 해결하기 위해 F2FS의 인덱스 자료구조를 캐싱하는 Node cache를 제안한다. 둘째, Range lock에서 발생하는 lock contention을 최소화하기 위해 Atomic 연산 기반 Range lock을 적용하였다. 이 두가지 기법을 적용한 F2FS를 120개 코어의 IBM 서버에서 FxMark 벤치마크를 사용하여 확장성을 실험하였다. 실험 결과 수정한 F2FS는 기존 F2FS에 비해 최대 24.6배, Interval tree Range lock을 적용한 F2FS에 비해 최대 10.5배의 성능 증가를 보였다.

## 1 서 론

NAND Flash 기반의 SSD는 높은 대역폭, 낮은 Latency 그리고 높은 병렬성을 보여주기 때문에 고성능의 저장장치로 널리 사용되고 있다. SSD는 블록 디바이스로 제공되고 파일 시스템을 사용하여 사용자의 데이터를 저장할 수 있다. 그 중에서 F2FS [1]는 SSD에 최적화된 파일 시스템으로 멀티 헤드 로깅 기법을 사용하여 SSD의 내부 병렬성을 활용한다. 반면 매니코어 서버는 수백개의 코어를 가지고 있고 이를 통해 높은 병렬성을 제공한다. 매니코어 환경에서 I/O intensive한 어플리케이션이 확장성 있는 성능을 내기 위해서는 파일시스템의 I/O 병렬 수행여부가 중요하다.

기존 연구에서는 매니코어 환경에서 F2FS의 단일 파일 I/O를 확장성 있게 만들기 위해 Inode mutex lock 대신 I/O를 수행하는 범위에만 lock을 획득하는 Range lock을 적용하였다 [2]. 이러한 Range lock은 단일 파일에 대한 I/O 성능을 크게 증가시켰지만 확장성 있는 성능을 보여주지 못한다. 본 논문에서는 매니코어 환경에서 F2FS의 단일 파일 I/O 수행시 확장성을 저해하는 원인을 밝혀내고 이를 해결하기 위한 방법을 제안한다. 본 논문이 기여하는 바는 다음과 같다.

첫째, 우리는 매니코어 환경에서 F2FS의 단일 파일 I/O의 확장성을 저해하는 원인이 페이지 캐시에서 발생하는 lock contention이라는 것을 밝혔다. 특히, F2FS의 Index structure인 Node를 접근할 때 얻는 mutex lock이 심각한 lock contention을 발생시킨다. 만약 여러 쓰레드가 하나의 Node를 읽기만 한다면 이를 mutex lock으로 보호할 필요가 없다. 따라서 우리는 불필요한 lock contention을 줄이기 위해 R/W Semaphore을 사용하여 여러 쓰레드가 Node에 동시에 접근하는 것을 허용하는 Node cache를 제안한다.

둘째, 우리는 F2FS의 단일 파일 I/O 확장성을 저해하는 다른 원인으로 Range lock에서 사용하는 Tree 자료구조에서 노드를 삽입/삭제 할 때 호출하는 mutex lock임을 밝혔다. 이를 해결하기 위해 우리는 비트맵과 Atomic 연산을 사용하는 Atomic 연산 기반 Range lock [3]을 F2FS에 적용하여 확장성을 향상시켰다.

우리는 제안한 기법들을 F2FS에 적용하여 120코어의 IBM 서버에서 Fxmark [4] 벤치마크를 사용하여 실험하였다. 단일 파일 쓰기 워크로드에서는 수정된 F2FS가 Baseline에 비해 최대 24.6배, Range lock을 적용한 F2FS에 비해 최대 10.5배의 성능 증가를 보였다. 단일 파일 읽기 워크로드에서는 수정된 F2FS가 Baseline에 비해 최대 8.1배, Range lock을 적용한 F2FS에 비해 최대 1.9배의 성능 증가를 보였다. 또한 수정된 F2FS의 I/O 성능이 코어수가 증가함에 따라 확장성 있음을 실험을 통해 알 수 있었다.

## 2 배경 지식

### 2.1 F2FS

그림 1는 F2FS의 On-disk 구조를 보여준다. F2FS의 On-disk 구조는 크게 두개의 영역으로 나누어진다. 데이터와 파일의 Index structure를 저장하는 메인 영역과 파일 시스템의 메타데이터를 저장하는 메타데이터 영역이 있다. 메인 영역은 파일의 Index structure(Inode, Direct node, Indirect node)를 저장하는 Node 로그와 데이터 블록을 저장하는 데이터 로그로 나누어진다. 각 Node는 유니크한 Node id를 할당받는다. 메타데이터 영역에는 NAT(Node Address Table) 자료구조가 있다. F2FS에서는 NAT를 통해 Node를 logical 블록 주소로 매핑한다. 그림 1에서는 실제 파일 데이터에 접근하기 위한 과정을 보여준다. Inode의 id는 *7*이고 NAT를 통해 *7*번 Node의 블록 주소인 *412*를 얻을 수 있다. 그 다음 주소 *412*에 있는 Inode에 접근한 후 Offset 정보를 이용해 데이터 블록의 주소를 얻는다. 얻으려는 데이터 블록은 주소 *600*에 있으므로 주소 *600*

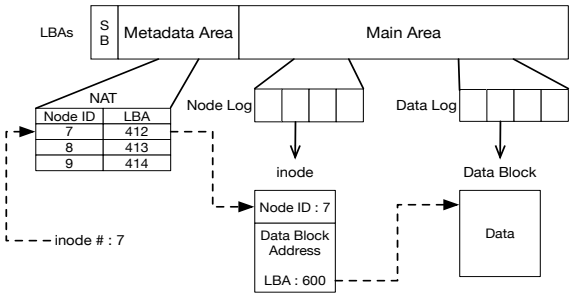



그림 1: F2FS 파일 시스템의 구조.

에 접근하여 데이터 블록을 읽을 수 있다.

### 2.2 파일 단위의 Range lock

Ext4, XFS, F2FS와 같은 POSIX 기반의 Native 파일시스템에서는 단일 파일에 대한 I/O를 Serialize 하게 설계되었다. 이러한 Serialization은 매니코어 환경에서 단일 파일에 대한 I/O 수행시 성능을 심각하게 저해하는 주요 원인이다. 따라서 이를 해결하기 위해 기존의 많은 연구들은 Inode mutex lock을 Range lock으로 대체하였다 [2, 3]. Range lock은 파일의 쓰려는 범위에만 lock을 얻을 수 있게 설계되었다. 따라서 여러 쓰레드가 단일 파일의 겹치지 않는 영역에 동시에 I/O 하려고 하면 쓰레드를 Block하지 않고 병렬적으로 수행할 수 있다.

## 3 매니코어 확장성을 지원하는 F2FS 설계 및 구현

본 섹션에서는 매니코어 환경에서 F2FS의 성능 확장성을 저해하는 원인을 분석한다. 또한 F2FS의 확장성을 높이기 위해 논문에서 제안하는 Node cache 설계와 Atomic 연산 기반 Range lock에 관하여 설명한다.

### 3.1 Node cache의 설계 및 구현

F2FS에서 단일 파일 I/O 수행시 페이지 캐시 내부의 Node를 접근하는데 이 때 사용하는 mutex lock 때문에 확장성 저하가 발생한다. 그림 2은 F2FS에서 커널 쓰레드가 I/O를 수행할 때 페이지 캐시에 있는 Node를 접근하는 과정이다. F2FS에서 Node를 접근할 때, 해당 Node를 포함하는 Inode로부터 탐색을 시작한다. 파라미터로 얻은 Offset값을 이용하여 Inode부터 자식 Node를 반복적으로 탐색하여 원하는 Node에 접근할 수 있다. 이 과정에서 Node에 접근할 때 페이지 캐시를 참조하게 된다. 접근하려는 Node id를 얻으면 페이지 캐시에서 이에 해당하는 Node가 있는지를 체크한다. 만약 페이지 캐시에 Node가 없다면 Block I/O request를 통해 Disk로부터 해당 페이지를 가져온다. Node가 존재하면 해당 페이지 캐시 엔트리에 접근한다. 이 때 Node를 보호하기 위해 페이지 캐시에서는 mutex lock을 사용한다. 여러개의 커널 쓰레드가 각각 다른 데이터 페이지에 대해 I/O를 수행하면 Range lock을 통해 병렬 I/O 수행은 허용하지만 여러 데이터 페이지들이 Node를 공유하기 때문에 Node에 접근하는 과정에서 lock contention이 발생한다.

F2FS는 Node를 lock/unlock 하기 위해 페이지 캐시 API를 사용한다. 그러나 Linux의 페이지 캐시는 mutex lock/unlock API만을

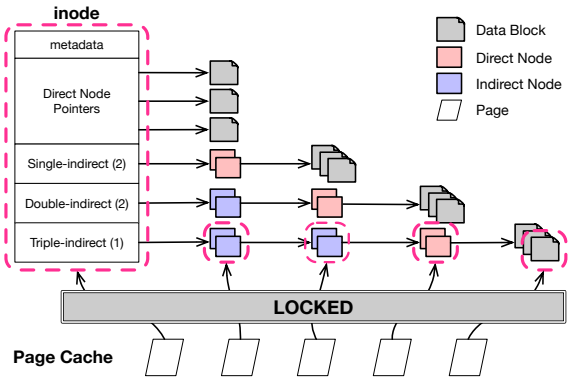



그림 2: 데이터 블록 접근시 mutex lock이 필요한 Node 페이지.

제공한다. 따라서 Node를 읽기만 하는 경우에도 mutex lock을 얻어야 한다. Node를 읽는 작업은 File read 뿐만 아니라 File write 호출시에도 수행된다. Node를 읽는 과정에서 발생하는 lock contention은 I/O 확장성을 크게 감소시킨다. 따라서 lock contention을 줄이기 위해 Node를 캐싱하는 Node cache를 설계하였다. Node cache는 페이지 캐시와 비슷한 역할을 한다. F2FS가 특정 Node에 처음 접근하면 해당 Node는 Node cache에 캐싱이 된다. 그 이후 캐싱된 Node에 대한 읽기나 쓰기 요청이 들어오면 해당 연산은 페이지 캐시를 거치지 않고 Node cache에서 수행된다. Node cache는 R/W Semaphore을 이용하여 Node를 보호하기 때문에 Node를 동시에 읽는 작업은 병렬적으로 수행할 수 있다.

### 3.2 Range lock의 확장성 문제 해결

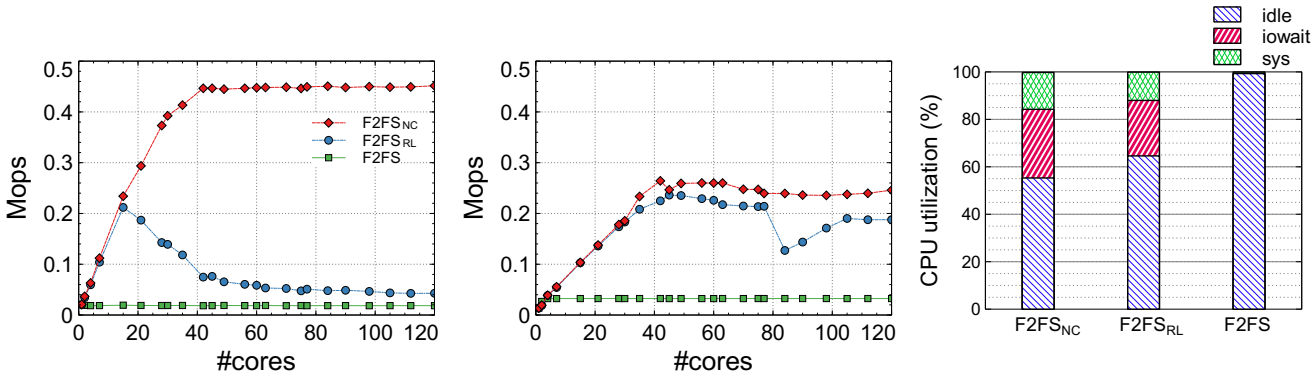
F2FS에서 단일 파일 I/O를 병렬적으로 수행하기 위해 사용하는 기존의 Range lock에서도 확장성 저하가 발생한다. Range lock의 경우 파일 하나당 Interval tree 하나를 가지고 있고 lock instance를 Tree의 노드로 관리한다. 각 노드는 커널 쓰레드가 I/O를 하려는 Interval을 가지고 있다. 커널 쓰레드가 I/O 수행을 위해 lock을 얻으려면 Interval tree에서 겹치는 영역을 가진 다른 노드가 있는지를 확인하고 겹치는 다른 노드가 없으면 노드를 삽입한 후 I/O를 수행한다. 커널 쓰레드가 I/O를 종료하면 Interval tree에서 노드를 제거함으로써 I/O 수행이 완료된다. Range lock에서는 노드를 삽입/삭제 할때 Tree 전체를 보호하기 위해 mutex lock을 얻어야 한다. I/O가 빈번하게 발생하는 매니코어 환경에서는 Range lock에서 사용하는 mutex lock이 F2FS의 확장성을 저해한다.

Kim et al. [3]에서는 Atomic 연산 기반 Range lock을 제안하였다. Atomic 연산 기반 Range lock은 Bitmap과 Atomic 연산들을 이용하여 범위를 체크하기 때문에 lock contention 없이 Range lock을 관리할 수 있다. 본 논문에서는 Atomic 연산 기반 Range lock을 구현하여 F2FS에 적용하였다.

## 4 실험 및 분석

### 4.1 실험 환경

표 1는 본 논문에서 제안한 기법을 평가하기 위한 실험 환경이다. 매니코어 환경에서 F2FS의 단일 파일 I/O 확장성 향상을 보기




(a) DWOM 처리율 (b) DRBM 처리율 (c) CPU Time Breakdown (DWOM)

그림 3: (a) DWOM과 (b) DRBM에 대한 기본 F2FS, Range lock 적용 F2FS와 Node cache와 Atomic 연산 기반 Range lock을 적용한 F2FS의 처리율 확장성 실험. (c) 120코어에서 DWOM 수행시 CPU Time을 Breakdown한 결과.

| | |
|---|---|
| CPU | Intel Xeon E7-8870 v2 2.3GHz ($15cores \times 8$) |
| RAM | 740GB |
| SSD | Samsung 970 EVO |
| Kernel | Linux 4.14.11 |

표 1: 실험 환경 설정.

위하여 FxMark 벤치마크의 DWOM, DRBM 워크로드를 사용하였다. DWOM은 병렬 단일 파일 쓰기이고 DRBM은 병렬 단일 파일 읽기 워크로드이다. 두 워크로드 모두 쓰레드들이 겹치지 않는 영역에 I/O를 수행한다. 그래프에서 F2FS는 기존 F2FS, F2FS$_{RL}$의 경우 Interval tree 기반의 Range lock을 적용한 F2FS, F2FS$_{NC}$의 경우 본 논문에서 제안한 기법을 적용한 F2FS를 나타낸다.

### 4.2 실험 결과 및 분석

그림 3(a)는 DWOM 실험 결과이다. F2FS의 경우 Inode mutex lock이 모든 I/O를 Serialization하기 때문에 코어 수에 따른 성능 변화가 없다. F2FS$_{RL}$의 경우 I/O를 병렬적으로 수행하기 때문에 15코어 까지는 성능이 확장한다. 그러나 섹션 2에서 언급한 페이지 캐시와 Interval tree에서의 lock contention 때문에 15코어 이후로는 성능이 감소한다. F2FS$_{NC}$의 경우 lock contention의 감소로 40코어까지 성능이 확장한다. 40코어 이후로는 성능이 확장하지 않는데 이는 F2FS가 Disk Bandwidth를 전부 다 소모했기 때문이다.

그림 3(b)는 DRBM 실험 결과이다. F2FS의 경우 Inode mutex lock이 모든 I/O를 Serialization하기 때문에 코어 수에 따른 성능 변화가 없다. F2FS$_{RL}$의 경우 40코어까지 성능이 확장한다. 그러나 40코어 이후로는 성능이 감소한다. 읽기의 경우에는 Node를 논리 주소에 매핑할 필요가 없기 때문에 쓰기 워크로드에 비해 Node에 접근하는 빈도가 낮다. 따라서 lock contention이 쓰기 워크로드에 비해서 자주 발생하지 않기 때문에 성능의 증가폭이 DWOM에 비해 낮았지만 확장성이 전체적으로 향상하였다.

그림 3(c)는 120코어에서 DWOM 수행시 CPU Time을 Breakdown한 실험 결과이다. *idle*은 CPU가 유휴한 상태, *iowait*은 Storage device의 I/O 수행을 기다리는 시간, *sys*는 Kernel에서 소모한 시간을 의미한다. F2FS에서는 대부분의 커널 쓰레드가 Inode mutex lock에 의해 Block되어서 자신의 차례를 기다리기 때문에 대부분의 CPU time이 *idle*임을 볼 수 있다. F2FS$_{RL}$의 경우 커널 쓰레드의 병렬 수행을 허용하기 때문에 *idle*의 비율은 감소하고 *sys*의 비율은 증가하였다. 또한 스토리지 디바이스에 부하가 증가하여 *iowait* 비율이 증가하였다. F2FS$_{NC}$는 lock contention을 최소화하여 병렬성을 더욱 향상하였기 때문에 *idle*의 비율은 감소하고 *sys*와 *iowait*의 비율은 더욱 증가하였다.

## 5 결론

본 논문에서는 매니코어 환경에서 F2FS의 단일 파일 I/O 수행 시 확장성을 저해하는 원인을 다음과 같이 밝혀내었다. (i) 페이지 캐시에서 F2FS의 Node 페이지를 접근할 때 발생하는 lock contention. (ii) 기존 Range lock의 Interval tree에서 발생하는 lock contention. 이런 문제점을 해결하기 위해 F2FS의 Node를 위한 R/W Semaphore 기반의 Node cache와 Atomic 연산 기반 Range lock을 적용하였다. 제안한 기법은 기존 F2FS와 Range lock을 적용한 F2FS에 비해 매니코어 환경에서 확장성 있는 성능을 보였다.

## 6 사사 표기

이 논문은 2019년도 정부(과학기술정보통신부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임(No. 2014-3-00035, 매니코어 기반 초고성능 스케일러블 OS 기초연구 (차세대 OS 기초연구센터)).

## 참고 문헌

[1] C. Lee, D. Sim, J. Y. Hwang, and S. Cho, "F2FS: A New File System for Flash Storage," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pp. 273–286, 2015.

[2] C.-G. Lee, H. Byun, S. Noh, H. Kang, and Y. 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*, pp. 21–32, ACM, 2019.

[3] J.-H. Kim, J. Kim, H. Kang, C.-G. Lee, S. Park, and Y. Kim, "pNOVA: Optimizing Shared File I/O Operations of NVM File System on Manycore Servers," in *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, pp. 1–7, ACM, 2019.

[4] C. Min, S. Kashyap, S. Maass, and T. Kim, "Understanding Many-core Scalability of File Systems," in *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (ATC)*, pp. 71–85, 2016.