매니코어 환경에서 Per-Core NAT를 통한 F2FS의 *create()* 확장성

황순, 이창규, 한정욱, 김영재
서강대학교 컴퓨터공학과
{hs950826, changgyu, immerhjw, youkim}@sogang.ac.kr

Enabling Scalability in F2FS *create()* by Per-Core NAT

Soon Hwang, Chang-Gyu Lee, Jungwook Han, Youngjae Kim
Department of Computer Science and Engineering, Sogang University, Seoul, Republic of Korea

요 약

매니코어 서버는 단일 서버 내부에 수백개의 Core를 가져 로컬 파일 시스템에 대해 병렬성이 높은 I/O 처리를 기대할 수 있다. 그러나 매니코어 환경에서 로컬 파일 시스템에 많은 Core들이 동시에 I/O를 수행하는 경우 성능 확장성이 크게 떨어진다. 기존 연구들은 데이터 I/O 성능 확장성을 위해 파일에 범위 기반 Locking을 적용하였다. 그러나 메타데이터 I/O는 I/O들이 공유하는 자료구조를 통해 처리되어 성능 확장성이 저해된다. 이 공유 자료구조에서 Thread 간 동기화를 위해 사용되는 Lock은 매니코어 환경에서 많은 Core들이 Blocking에 빠지게 하여 메타데이터 I/O를 효율적으로 처리하지 못하고 Core들의 CPU Cycle이 낭비되게한다. 특히 F2FS에서 각 개별 디렉토리에 File Create을 시도하는 워크로드를 살펴본 결과 1 Core와 20 Core 간의 Throughput은 204% 증가하였으나 20 Core와 40 Core간의 Throughput은 4% 감소하여 코어수 증가에 따른 확장성이 제한되었다. 따라서 본 논문에서는 매니코어 환경에서 F2FS의 *create()* 확장성 병목을 분석하고 이를 개선하기 위한 Per-Core NAT를 제안한다. 제안한 기법을 적용한 F2FS는 개별 디렉토리에서의 *create()* 워크로드에서 F2FS와 비교해 20 Core에서는 19%, 40 Core에서는 57% Throughput이 증가하여 확장성을 보였다.

1. 서 론

대량의 작은 파일들이 주를 이루는 데이터셋은 파일 시스템의 일반적인 워크로드 중 하나로 과학 데이터나 빅데이터 처리 시스템의 스토리지 시스템 등에서 흔히 볼 수 있다 [1, 2]. 이러한 대량의 작은 파일들을 효율적으로 수용하기 위하여 기존 분산 및 병렬 파일 시스템에서는 주로 메타데이터 서버의 복제 혹은 인덱싱 구조의 재설계를 제안하였다 [3, 4, 5]. 근래 단일 CPU에서 제공하는 Core의 수가 증가함에 따라 수백개의 Core를 가진 매니코어 서버가 데이터 처리 시스템에 사용되고 있다. 또한 매니코어 서버는 NVMe SSD 등의 고성능 저장장치를 동원하여 로컬 파일 시스템에 대한 병렬성이 높은 I/O 처리가 가능하게 되었다.

매니코어 환경에서 로컬 파일 시스템에 많은 Core가 동시에 I/O 수행하는 경우 성능 확장성이 크게 떨어지는 문제가 있다. I/O를 수행하는 Core 수 증가에 비례한 Throughput 증가인 성능 확장성은 크게 데이터 I/O의 확장성과 메타데이터 I/O의 확장성으로 나눌 수 있다. 이 중 데이터 I/O의 경우 파일에 범위 기반 Locking를 활용한 성능 확장성 연구가 수행되었다 [6, 7]. 그러나 데이터 I/O 못지않게 병렬 메타데이터 I/O 또한 성능 확장성이 크게 떨어진다 [8]. 이러한 낮은 확장성의 이유는 메타데이터 I/O들이 공유하는 자료구조에 파일 시스템 범위의 Lock이 사용되기 때문이다.

기존 멀티코어 환경에서는 공유 자료구조의 Lock에 대한 경쟁이 개별 메타데이터 I/O에 큰 지연시간을 부과하지 않았다. 그러나 매니코어 환경이 보다 많은 Core를 제공함에 따라 공유 자료구조의 Lock 오버헤드가 크게 증가하였고 각 Core들이 공유 자료구조의 Lock 획득을 위해 많은 CPU Cycle를 소모하게 되었다 [3]. 대표적인 메타데이터 I/O인 *create()*의 성능 확장성 개선을 위하여, 본 논문에서는 FxMark [8]를 사용해 Linux 파일 시스템인 F2FS의 *create()* 확장성을 분석하였다. 그 결과 *create()* 확장성 병목은 On-disk 자료구조인 NAT (Node Address Table)의 캐시인 Free NID Cache와 NAT Cache에 적용된 Lock으로 인한 Blocking에 기인함을 확인하였다. 이를 해결하기 위하여 각 Core가 NAT에 독립적인 접근을 가능케하는 Per-Core NAT를 제안하고 각 Core의 독립된 디렉토리에서 *create()* 수행하는 워크로드에서 확장성을 보였다.

이 논문은 2020년도 정부(과학기술정보통신부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임 (No.2014-3-00035, 매니코어 기반 초고성능 스케일러블 OS 기초연구 (차세대 OS 기초연구센터))

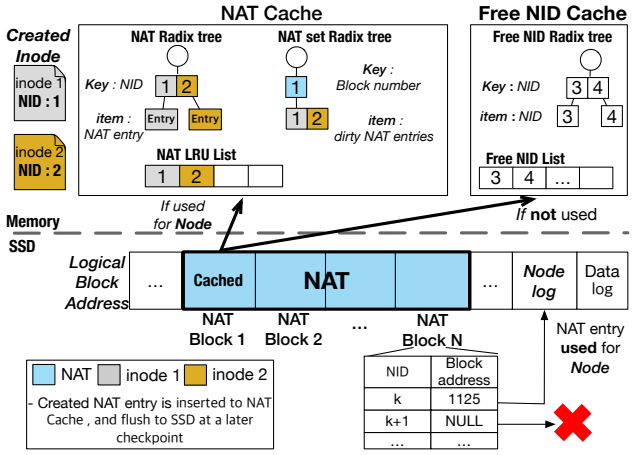


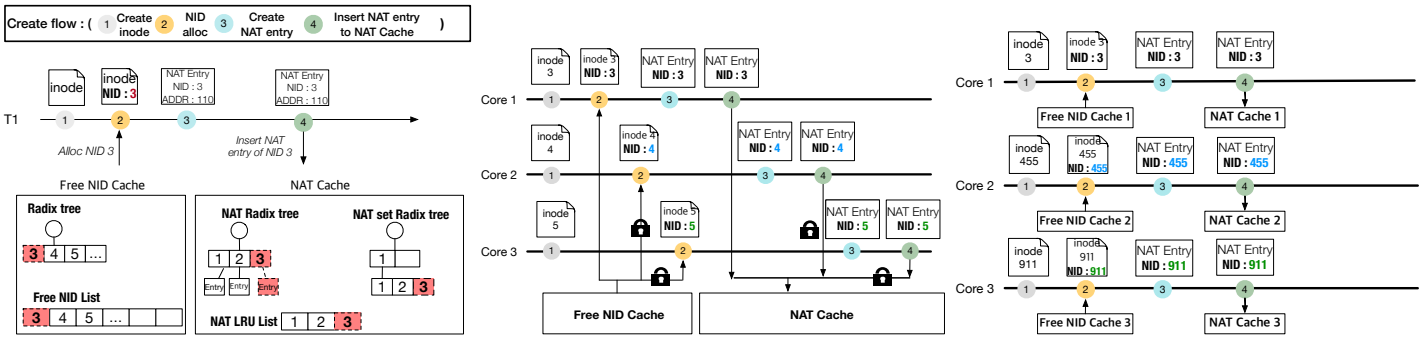
그림 1: NAT Cache in F2FS

2. 배경 지식

2.1 NAT Cache in F2FS

F2FS는 inode와 direct node, indirect node등의 파일 메타데이터를 모두 *Node*로 관리한다. 각 *Node*는 파일 시스템 내에서 고유한 ID (NID)로 구분된다. NID와 *Node*의 블록 주소간의 맵핑은 NAT Entry로 저장되고 이는 On-disk 자료구조인 NAT에 저장된다. 따라서 NID를 통해 *Node*의 Block 주소를 얻으려면 반드시 NAT를 참조해야한다. 그림 1은 F2FS의 NAT와 NAT Cache의 구조를 보여준다. NAT는 NAT Block들로 구성되어 있고 NAT Block은 다수의 NAT Entry를 담고 있다. 또한 NAT entry는 NAT Block의 단위로 메모리에 로드된다.

캐싱된 NAT Block에 속한 NAT Entry들은 해당 NID가 *Node*에 할당되어 있는지 여부에 따라 NAT Cache와 Free NID Cache 두 메모리 자료구조에 나누어 관리된다. NAT Cache는 *Node*의 주소를 가진 유효한 NAT Entry들을 NAT Radix Tree와 NAT Set Radix Tree, NAT LRU List를 사용해 관리한다. NAT Radix tree는 NID



(a) F2FS *create()* (b) Baseline F2FS *create()* (c) Per-Core NAT F2FS *create()*

그림 2: F2FS *create()* Analysis

를 Key로 사용해 NAT Entry를 검색하여 *Node*의 주소를 얻는데 사용되며 NAT Set Radix tree는 NAT Entry들을 속한 NAT Block 단위로 저장하여 수정으로 인해 Flush가 필요한 NAT Block들을 관리한다. NAT Cache는 NAT LRU List를 사용해 LRU (Least Recently Used) 방식으로 동작한다. Free NID Cache는 유효하지 않은 NID들을 Free NID List와 Free NID Radix tree를 사용해 관리한다. *Node* 생성시 새로운 NID는 Free NID List로 부터 얻어지며 Free NID List에 유일한 NID만을 담기위해 해당 List에 NID가 추가되기 전 Free NID Radix tree 사용해 중복 검사를 수행한다.

그림 1은 NAT Block 1이 메모리로 로드되어 NID 1,2,3,4가 캐시된 후 inode 1과 inode 2에 각각 NID 1,2가 할당된 모습을 보여준다. 이 때, 유효하지 않은 NID 3,4는 Free NID Cache에 적재된다. NID 1,2가 inode에 할당되어 이후 참조를 위해 NAT Radix tree에 해당되는 NAT Entry들이 삽입된다. 또한 새로 생성된 Entry들이 아직 Flush되지 않았으므로 NAT Set Radix tree에도 삽입되며 추후 Eviction을 위해 NAT LRU List에도 추가된다. NID 3,4는 *Node*에 부여되지 않아 Free NID Cache에 적재된다. 이후 Free NID 부여를 위해 Free NID Radix tree를 통해 이미 Free NID List에 존재하지 않음을 확인했다면 Radix tree와 List 모두에 삽입된다.

3. 설계 및 구현

3.1 F2FS에서 *create()* 과정

그림 2a는 F2FS의 *create()* 과정으로 ①-④의 순서대로 실행된다. *create()*는 가정 먼저 파일의 inode를 생성한다 (①). 생성된 inode는 *Node*로 관리되고 NID 3이 부여된다. 직후 NID 3은 Free NID Cache에서 삭제된다 (②). 이때 같은 NID가 중복 사용되지 않도록 Free NID Cache는 spinlock으로 보호된다. 생성된 *Node*가 이후에 참조되기 위해서 NID와 블록 주소를 가진 NAT Entry를 생성한다 (③). 마지막으로 생성된 NAT Entry는 NAT Cache에 삽입된다 (④). ③과 ④는 NAT Cache에 여러 Thread가 안전하게 NAT Entry를 삽입할 수 있도록 RW Lock (Reader-Writer Lock)로 같은 Critical Section으로서 보호된다. ④에서 NID 3의 NAT Entry가 새로 생성되었으므로 NAT Radix tree에 key가 3인 NAT Entry가 삽입된다. 또한 NID 3은 NAT Block 1에 속하므로 NAT set Radix tree에 key가 1인 item에 NAT Entry를 추가한다. 마지막으로 NAT LRU List에 NID 3을 삽입한다.

그림 2b는 Core 3개가 개별 디렉토리에 *create()*하는 경우의 동작을 보여준다. 각 Core들은 각자 다른 파일을 생성하므로 병렬적으로 inode를 생성한다 (①). 그러나 Free NID Cache로부터 NID를 부여받는 과정 (②)은 병렬적으로 실행되지 못한다. Core 1이 NID 3을 부여받는 동안 Core 2와 Core 3는 Block되고 Core들의 ②는 그림 2b와 같이 순차적으로 실행된다. 또한 Core 1이 NID 3을 가진 NAT Entry를 생성 (③)하고 삽입 (④)하는 동안 Core 2, Core 3는 Blocking되어 병렬적으로 실행되지 못한다. Core들의 ③-④ 실행은 그림 2b와 같이 순차적으로 실행된다.

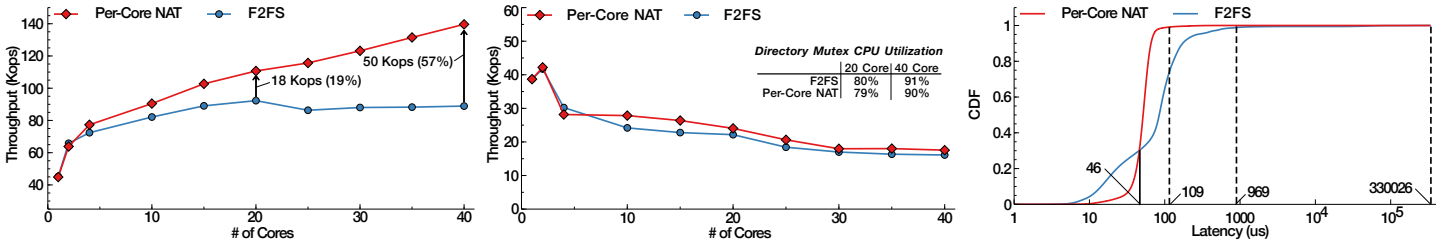
병렬적으로 실행되는 *create()*는 Free NID Cache를 보호하는 파일 시스템 범위의 spinlock과 NAT Cache를 보호하는 파일 시스템 범위의 RW Lock으로 인해 병렬성이 저하되어 확장성을 보이지 못한다. 이는 *create()*에서 발생하는 파일 시스템 범위의 Blocking으로 인해 파일 데이터 I/O 시 CPU Cycle을 충분히 활용하지 못하여 여러 자원들이 낭비될 수 있음을 시사한다. 본 논문은 병렬 파일 *create()* 시 확장성 병목인 NAT Cache와 Free NID Cache의 Lock으로 인해 발생하는 Core들의 Blocking을 개선하기 위해서 Per-Core NAT를 제안한다.

3.2 Per-Core NAT

Per-Core NAT는 Core들이 개별적으로 NAT Cache와 Free NID Cache를 가진다. 이때 Core들은 저장 장치 상의 NAT에서 서로 겹치지 않은 NAT Block들을 캐시해온다. 따라서 하나의 Core은 개별 NAT Cache와 Free NID Cache에 대한 spinlock과 RW Lock들을 사용하여 동시에 I/O를 수행하고 있는 다른 Core와 Lock 경쟁을 하지 않는다. 이때 NID는 자신이 속한 Block number와 Block 내부 offset을 이용해서 계산되므로 각 Core의 Free NID Cache와 NAT Cache 간에는 중복된 NID가 존재하지 않는다. 따라서 *create()* 시, Core들은 생성한 inode를 위한 NID를 다른 Core의 Free NID Cache와의 간섭 없이 동시에 부여받을 수 있다. 또한 Core들은 생성된 inode에 대한 NAT Entry들을 각자의 NAT Cache에 동시에 삽입할 수 있다. 결과적으로 *create()*는 Core들 간 독립적으로 진행되므로 전체 과정을 병렬적으로 수행할 수 있다.

Per-Core NAT에서 각 Core에 할당된 NAT Block들은 중복되지 않으며 따라서 기존 F2FS와 마찬가지로 함께 Flush 될 NAT Entry들은 같은 Block에 속한다. 이는 F2FS에서 메타데이터를 Flush하는 Checkpoint 과정에서 기존 F2FS가 Flush해야 할 NAT Block의 수가 Per-Core NAT를 사용했을 때와 차이가 없음을 의미하고 따라서 추가적인 I/O Overhead가 발생하지 않는다.

그림 2c는 Core들이 개별 디렉토리에 *create()* 하는 경우를 보여준다. Core 1이 inode 3에 NID 3를 할당함과 동시에 Core 2와 Core 3 또한 Per-core Free NID Cache에서 NID 455와 NID 911을 할당한다 (②). 각 Free NID Cache들은 개별 spinlock에 의해 보호되므로 (②) 모두 병렬로 실행된다. 이후 Core 1이 inode 3의 NAT Entry를 생성 및 삽입하는 동시에 Core 2, Core 3 역시 Per-core NAT Cache에 NAT Entry를 생성, 삽입한다 (③-④). 이 또한



(a) MWCL Throughput (b) MWCM Throughput (c) MWCL Latency

그림 3: MWCL and MWCM Scalability comparison between Per-Core NAT and Baseline F2FS with MWCL Latency in 40 Core

각 NAT Cache는 개별 RW Lock에 의해 보호되므로 (③-④) 병렬적으로 실행된다. 이와 같이 Per-Core NAT은 기존 *create()*에서 발생하는 Lock 경쟁으로 인한 Blocking time을 줄이고 매니코어 환경에서 각 Core들이 Lock 대기를 위해 CPU cycle을 낭비되지 않도록 하여 성능 확장성을 개선한다.

4. 실험 및 분석

제안하는 Per-core NAT는 Linux 5.13에서 구현되었으며 매니코어 환경에서의 *create()* 확장성 향상을 평가하기 위해 40 Core (2 socket) 서버에서 256GB의 DRAM과 Samsung EVO 970 250GB NVMe SSD를 사용하여 실험을 수행하였다. 워크로드는 FxMark [8]의 MWCL 워크로드와 MWCM 워크로드를 사용하였다. MWCL은 각 Core들이 개별 디렉토리에 *create()*를 수행하는 워크로드이며 MWCM은 각 Core들이 하나의 공유 디렉토리에 *create()*을 수행하는 워크로드이다.

4.1 개별 디렉토리에서의 *create()* 확장성

Throughput: 그림 3a에서 기존 F2FS는 NAT Cache의 RW Lock과 Free NID Cache의 spinlock으로 인해 20 Core이후 확장성을 보이지 못하고 40 Core까지 Throughput이 유지된다. 반면 Per-Core NAT는 각 Core들이 각각의 NAT Cache와 Free NID Cache를 가져 Core들간의 Blocking Time이 최소화된다. 전체 Core들이 *create()*을 병렬적으로 수행하므로 20 Core에서 18 Kops만큼 40 Core에서 50 Kops만큼의 성능 향상 및 확장성을 보인다.

Latency: 그림 3c은 MWCL에서 F2FS의 *create()* Latency를 표현한 CDF이다. 기존 F2FS에서 *create()*의 평균 Latency는 424 us이며 P99 Latency는 969 us이다. Per-Core NAT에서 *create()*의 평균 Latency는 103 us이며 P99 Latency는 109 us이다. 기존 F2FS의 높은 P99 latency는 파일 시스템 범위의 Blocking에 의해 이후 작업들이 직렬화될 수 있음을 시사한다. 그러나 Max Latency는 Per-Core NAT가 기존 F2FS에 비해 19% 높은 모습을 보인다. 이는 Threshold보다 더 많은 *Node*가 생성된 경우 SSD로 *Node*들을 Flush하는 Policy에 기인한다. Per-Core NAT가 기존 F2FS보다 더 많은 *Node*를 생성하므로 Flush될 *Node*가 더 많기 때문에 더 큰 Latency를 보인다. 제안한 기법을 통해서 Lock으로 인해 발생하는 Core들간의 Blocking time을 감소시켜 *create()*의 Latency가 감소했음을 알 수 있다. 기존 F2FS는 46 us까지 Per-Core NAT보다 낮은 Latency를 가진다. Per-Core NAT에서 각 Core별 NAT Cache와 Free NID Cache로의 분산을 위한 연산을 수행하나 기존 F2FS는 분산 과정이 없어 더 짧은 Latency를 가진다.

4.2 공유 디렉토리에서의 *create()* 확장성

그림 3b는 공유 디렉토리에서의 *create()* 실험 결과이다. 공유 디렉토리 상에서의 *create()*는 개별 디렉토리에서의 *create()*와 달리 F2FS와 Per-Core NAT 모두 확장성을 보이지 못하고 2 Core이후 성능이 감소한다. 이는 파일 시스템 범위의 Lock이 아닌, VFS (Virtual File System) 계층에서의 Lock으로 인해 발생하는 병목에 기인한다. 공유 디렉토리에서 여러 Core들이 *create()*를 할 때, VFS 계층에서 부모 디렉토리를 보호하기 위한 Mutex Lock이 강제된다. 따라서 공유 디렉토리에서 한 Core가 *create()*을 수행하는 동안 다른 Core들은 Mutex Lock로 인해 직렬화되므로 개별 NAT Cache 및 Free NID Cache를 통한 동시 수행이 불가능하다. Perf [9]를 사용하여 분석한 결과 공유 디렉토리에 대한 Mutex Lock이 20 Core에서는 전체 CPU Cycle의 80%, 40 Core에서는 91%를 소모하여 VFS 계층에서의 디렉토리에 대한 Mutex Lock이 확장성 병목임을 밝혔다.

5. 결론

본 논문에서는 매니코어 환경에서 F2FS의 병렬 *create()*시 성능과 확장성을 저해하는 원인을 밝혔다. 이는 NAT Cache와 Free NID Cache에서의 파일 시스템 범위의 Lock으로 인한 Core간의 Blocking이다. 이를 해결하기 위해서 Per-Core NAT를 적용하여 개별 디렉토리에서 *create()* 워크로드에서 확장성을 보였다.

참고 문헌

[1] NERSC, "Nersc storage trends and summaries," 2021.

[2] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch, "A five-year study of file-system metadata," ACM Trans. Storage, vol. 3, p. 9–es, Oct. 2007.

[3] Q. Zheng, K. Ren, G. Gibson, B. W. Settlemyer, and G. Grider, "Deltafs: Exascale file systems scale better without dedicated servers," in Proceedings of the 10th Parallel Data Storage Workshop, PDSW '15, (New York, NY, USA), p. 1–6, Association for Computing Machinery, 2015.

[4] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion," in SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 237–248, 2014.

[5] P. Matri, M. S. Pérez, A. Costan, and G. Antoniu, "Tỳrfs: increasing small files access performance with dynamic metadata replication," in 2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pp. 452–461, IEEE, 2018.

[6] C.-G. Lee, S. Noh, H. Kang, S. Hwang, and Y. Kim, "Concurrent file metadata structure using readers-writer lock," in Proceedings of the 36th Annual ACM Symposium on Applied Computing, SAC '21, (New York, NY, USA), p. 1172–1181, Association for Computing Machinery, 2021.

[7] 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, APSys '19, (New York, NY, USA), p. 1–7, Association for Computing Machinery, 2019.

[8] C. Min, S. Kashyap, S. Maass, W. Kang, and T. Kim, "Understanding manycore scalability of file systems," in Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '16, (USA), p. 71–85, USENIX Association, 2016.

[9] "perf: Linux profiling with performance counters," 2021.