creative commons

COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

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

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 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

공학석사 학위논문

오브젝트 스토리지 기반의 확장성 있는 분산 파일 시스템

2022년 2월

서울대학교 대학원

컴퓨터공학부

강 인 재

오브젝트 스토리지 기반의 확장성 있는 분산 파일 시스템

지도교수 김 진 수

이 논문을 공학석사 학위논문으로 제출함

2021년 12월

서울대학교 대학원

컴퓨터공학부

강 인 재

강인재의 공학석사 학위논문을 인준함

2021년 12월

위 원 장 염 헌 영 (인)

부위원장 김 진 수 (인)

위 원 이 재 진 (인)

초 록

오브젝트 스토리지는 확장이 쉽고 비용 효율적인 구조 덕분에 HPC 환경에서 널리 이용되고 있다. 기존의 연구들은 오브젝트 스토리지의 확장성을 활용하기 위해 오브젝트 스토리지 기반의 분산 파일 시스템을 제안하였으나 POSIX 표준을 준수하지 못하거나 낮은 확장성을 보이는 문제를 가지고 있다. 본 논문에서는 오브젝트 스토리지 기반의 분산 파일 시스템의 확장성을 개선하기 위한 메타데이터 구조와, 메타데이터를 효율적으로 저널링하기 위한 기법을 제안한다.

주요어 : 분산 파일 시스템, 오브젝트 스토리지
학 번 : 2020-29665

목 차

제 1 장 서론 ····· 1

제 2 장 배경 지식 ····· 2
제 1 절 오브젝트 스토리지 ····· 2
제 2 절 저널링 파일 시스템 ····· 2

제 3 장 관련 연구 ····· 4

제 4 장 오브젝트 스토리지 기반의
분산 파일 시스템의 설계 ····· 5
제 1 절 설계 목표 ····· 5
제 2 절 파일 시스템 구조 ····· 6
제 3 절 오브젝트 저장 방식 ····· 7
제 4 절 메타데이터 관리 ····· 8
제 5 절 저널링 ····· 10

제 5 장 실험 결과 ····· 14
제 1 절 실험 환경 ····· 14
제 2 절 메타데이터 I/O 성능 ····· 15
제 3 절 작은 파일에 대한 I/O 성능 ····· 17
제 4 절 클라이언트 수에 따른 I/O 성능 ····· 18
제 5 절 저널링 소요 시간 ····· 19

제 6 장 결론 ····· 20

참고문헌 ····· 21

Abstract ····· 22

표 목 차

[표 1] 실험 환경 ····· 14

그 림 목 차

[그림 1] ArkFS 클라이언트의 구조 ····· 6
[그림 2] 파일 시스템 계층 구조의 예 ····· 9
[그림 3] 파일 시스템 연산을 대신 수행하는 과정 ····· 9
[그림 4] 메타데이터의 저널링 과정 ····· 11
[그림 5] 클라이언트 간 연산의 커밋 과정 ····· 12
[그림 6] 메타데이터 I/O 성능 ····· 16
[그림 7] 작은 파일에 대한 I/O 성능 ····· 17
[그림 8] 클라이언트 수에 따른 I/O 성능 ····· 18
[그림 9] 저널링 소요 시간 ····· 19

제 1 장 서론

오브젝트 스토리지는 높은 안정성과 확장성, 비용 효율성을 가지고 있으며 대용량의 데이터를 저장하는 데에 적합하다. 이러한 장점 덕분에 AWS S3이나 Openstack Swift와 같은 다양한 오브젝트 스토리지들이 HPC(High-Performance Computing) 시스템에서 널리 이용되고 있다 [1], [4]. 그러나 인터페이스의 차이로 인해 파일 시스템상에서 작동하는 응용들이 오브젝트 스토리지 위에서도 직접 작동되기는 어렵다. 또한 기존의 많은 HPC 응용들은 파일과 디렉터리에 대한 접근 제어 목록 등의 기능을 제공하는 파일 시스템 인터페이스에 깊이 의존하고 있다.

따라서 오브젝트 스토리지의 높은 확장성을 활용함과 동시에 파일 시스템이 제공하는 기능을 사용하기 위해 오브젝트 스토리지를 기반으로 하는 다양한 분산 파일 시스템이 제안된 바 있다. 그러나 기존의 분산 파일 시스템은 `rename()`과 같은 파일 시스템 연산의 원자성을 보장하지 않는 등 POSIX 규격을 준수하지 못하거나, 메타데이터 관리 전용 노드에서 발생하는 병목으로 인해 낮은 확장성을 보이는 등의 문제점을 갖고 있다.

본 논문에서는 오브젝트 스토리지 기반의 분산 파일 시스템이 높은 확장성을 가질 수 있도록 하는 메타데이터의 관리 방식을 소개한다.

제 2 장 배경 지식

제 1 절 오브젝트 스토리지

오브젝트 스토리지는 블록 스토리지와는 다른 인터페이스를 갖는다. 전통적인 블록 인터페이스는 저장 공간을 고정된 크기의 블록들로 나누고 각 블록에 번호를 매긴다. 따라서 해당 번호를 사용하여 블록 단위의 읽기/쓰기를 할 수 있다. 이와 달리 오브젝트 스토리지는 데이터를 키-값의 형태로 저장한다. 키와 값은 임의의 바이트열로 구성되며, 그 길이는 오브젝트 스토리지에 따라 정해진 범위 내에서 가변적이다.

일반적으로 오브젝트 스토리지는 `get()`, `put()`, `delete()`를 포함하는 단순한 형태의 API를 제공한다. 사용자는 이를 사용하여 주어진 키에 대해 데이터를 읽거나, 변경하거나, 또는 삭제할 수 있다.

제 2 절 저널링 파일 시스템

파일 시스템은 블록 인터페이스상에서 디렉터리 구조를 추상화하기 위해 여러 종류의 메타데이터를 유지한다. 이러한 메타데이터는 파일 시스템 연산 중에 빈번하게 변경되며, 대부분의 파일 시스템 연산이 하나 이상의 메타데이터를 변경한다. 둘 이상의 메타데이터를 변경하는 파일 시스템 연산의 경우, 연산 중에 시스템이 실패하면 메타데이터 간의 일관성이 지켜지지 않을 수 있다. 예를 들어, 디렉터리 내에 파일을 생성하다가 시스템이 실패하면 디렉터리에는 파일 이름이 올바르게 쓰였지만 실제 파일의 아이노드는 쓰이지 않았을 수도 있다. 이러한 비정상적인 중간 상태를 방지하고자 많은 파일 시스템들은 저널링을 지원한다. 저널링은 메타데이터(또는 데이터)의 변경 사항을 반영하기 전에, 시스템이

실패할 것을 대비하여 미리 지정된 영역에 로그를 남겨두는 것을 의미한다. 만일 해당 로그가 완전히 쓰이기 전에 시스템이 실패했다면, 해당 연산은 수행되지 않은 것으로 간주한다. 만일 해당 로그가 완전히 쓰인 후 실제 메타데이터를 변경할 때 시스템이 실패했다면, 시스템을 재시작한 후 로그를 살펴봄으로써 복구할 수 있다.

제 3 장 관련 연구

기존의 많은 연구들은 오브젝트 스토리지의 확장성과 파일 시스템의 편리성을 결합하고자 오브젝트 스토리지 기반의 분산 파일 시스템을 제안한 바 있다.

SVFS, Goofys와 같은 파일 시스템들은 클라우드 오브젝트 스토리지 상에서 파일 시스템 인터페이스를 제공한다 [3]. 그러나 이들 파일 시스템은 POSIX의 표준을 준수하지 못하며 접근 제어 리스트와 같은 기능을 제공하지 않는다. S3FS 또한 클라우드 오브젝트 스토리지를 기반으로 하는 파일 시스템이다 [2]. S3FS는 앞서의 두 파일 시스템과 비교하여 POSIX 표준에 더욱 가까운 기능을 제공한다. 그러나 S3FS는 메모리를 활용하지 못하고 디스크를 데이터 캐시로 사용하기 때문에 성능이 좋지 않다. 또한 S3FS는 파일 또는 디렉터리의 절대 경로를 키로 사용하기 때문에 경로의 중간 컴포넌트가 변경되는 경우 함께 변경되어야 할 키가 많아지게 된다. S3FS는 `rename()`의 원자성을 보장하지 않는 단점도 가지고 있다.

MarFS와 CephFS는 높은 수준의 POSIX 호환성을 보장하는 오브젝트 스토리지 기반의 분산 파일 시스템이다 [5], [7]. 이 파일 시스템들은 메타데이터를 관리하기 위한 전용 노드를 배치하여 사용한다. 따라서 클라이언트의 수가 많아지게 되면 메타데이터 서버에서 병목 현상이 발생하므로 확장성이 낮다는 한계를 갖는다.

제 4 장 오브젝트 스토리지 기반의 분산 파일 시스템의 설계

제 1 절 설계 목표

본 연구에서는 다음과 같은 설계 목표를 달성할 수 있는 분산 파일 시스템 ArkFS를 제시한다.

- 높은 확장성

  메타데이터 서버는 분산 파일 시스템의 확장성을 저하시키는 주된 요인이다. ArkFS는 메타데이터 서버를 유지하지 않으며, 각 클라이언트가 파일 시스템 메타데이터의 일부를 담당하여 관리하도록 한다. 따라서 클라이언트의 수가 증가함에 따라 높은 확장성을 달성할 수 있다.

- 높은 수준의 POSIX 호환성

  ArkFS는 디렉터리 구조 및 접근 제어 목록을 포함하는 POSIX 표준에 명시된 파일 시스템의 기능을 지원한다. 또한, ArkFS의 파일 시스템 연산은 POSIX의 표준을 상당 부분 준수한다.

- 다양한 오브젝트 스토리지에서 작동 가능

  ArkFS는 다양한 오브젝트 스토리지 위에서 작동한다. 사용자가 새로운 오브젝트 스토리지를 사용하고자 할 경우, ArkFS에서 사용하는 오브젝트 관련 함수를 해당 오브젝트 스토리지에서 제공하는 API로 구현하여 사용할 수 있다.

제 2 절 파일 시스템 구조

ArkFS 클러스터는 다수의 ArkFS 클라이언트와 하나의 리스 관리자로 구성되어 있다. ArkFS 클라이언트의 전체적인 구조는 [그림 1]과 같다. ArkFS 클라이언트는 파일 시스템 연산을 수행하는 스레드, 저널링을 수행하는 스레드, 데이터 캐시를 관리하는 스레드 등으로 구성된다. 파일 시스템 연산을 수행하는 스레드는 로컬에 존재하는 메타데이터나 다른 클라이언트가 관리하는 메타데이터에 RPC(remote procedure call)을 통해 접근한다. 파일 시스템 스레드는 오브젝트를 읽거나 쓰는 과정에서 백엔드 오브젝트 스토리지의 API로 작성된 오브젝트 관련 함수를 호출한다. ArkFS 클라이언트를 구성하는 각 부분의 역할에 관해서는 이후의 절에서 더욱 자세히 설명하기로 한다.

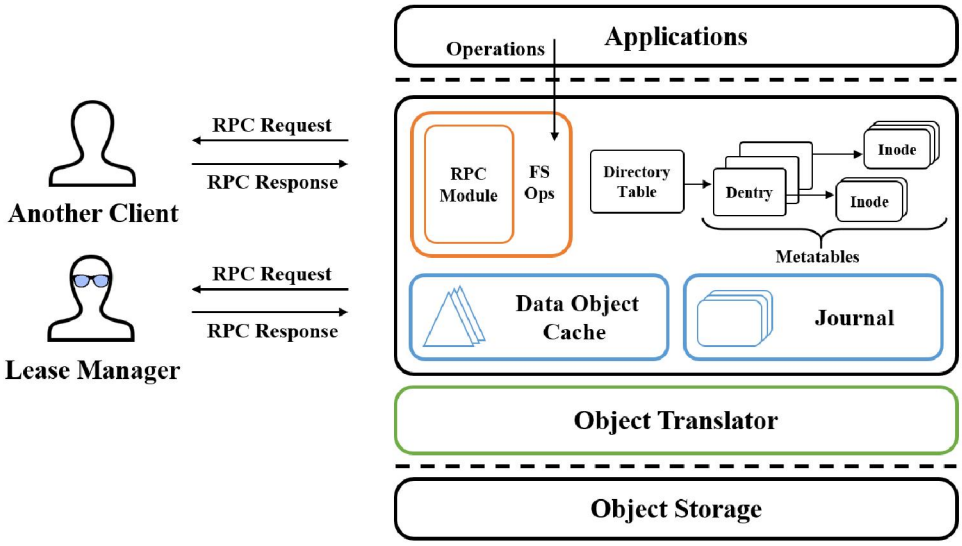


그림 1. ArkFS 클라이언트의 구조

제 3 절 오브젝트 저장 방식

ArkFS는 다양한 오브젝트 스토리지를 지원하며, 오브젝트 스토리지에 독립적으로 파일 시스템이 동작할 수 있도록 오브젝트 스토리지에 접근하는 부분을 모듈화하여 사용한다. 이는 [그림 1]에서 'Object Translator'로 표시된 부분에 해당한다. 파일 시스템 연산, 저널링, 캐시 관리 등을 수행하는 과정에서 오브젝트를 읽거나 쓸 때에는 오브젝트를 번역하는 모듈을 거치게 된다. 오브젝트 번역 모듈은 파일 시스템에서 공통적으로 사용하는 오브젝트 관련 함수를 오브젝트 스토리지에서 제공하는 API로 구현한 것이다. 만일 새로운 오브젝트 스토리지를 사용하고자 한다면 파일 시스템의 다른 부분을 수정하지 않고도 해당 오브젝트 스토리지의 API로 오브젝트 관련 함수를 등록하여 사용할 수 있다.

오브젝트 스토리지상에서 파일 시스템을 구현하기 위해서는 파일 시스템의 구성 요소, 즉 메타데이터 및 데이터를 오브젝트의 형태로 스토리지에 저장해야 한다. 파일 시스템은 각각의 파일 또는 디렉터리에 고유한 아이노드 번호를 할당하므로 이를 사용하면 파일 시스템의 구성 요소들을 고유한 키로 오브젝트 스토리지에 저장할 수 있다.

ArkFS는 파일 시스템의 구성 요소들을 '아이노드', '디엔트리', '데이터', '저널'의 네 가지 타입으로 구분하며, 각 요소의 타입에 따라 오브젝트 키의 접두어를 다르게 한다. 예를 들어, 아이노드의 접두어로는 'i$'를 사용하므로 어떤 파일의 아이노드 번호가 0x3ad1인 경우 오브젝트 스토리지에서 해당 파일의 아이노드를 접근하기 위한 키는 'i$3ad1'이 되는 식이다.

만일 오브젝트의 최대 크기가 제한되는 경우에는 접미어를 붙임으로써 구성 요소를 여러 오브젝트로 나누어 저장할 수 있다. 예를 들어, Ceph RADOS의 경우 오브젝트의 크기는 최대 4MB로 제한되므로, 파일의 데이터와 같이 4MB를 초과할 수 있는 요소에 대해서는 접미어를 붙여 'd$3ad1#0', 'd$3ad1#1', 'd$3ad1#2', ...와 같이 저장하면 된다.

제 4 절 메타데이터 관리

기존의 몇몇 연구에서는 메타데이터를 관리하기 위한 전용 노드를 배치하였는데, 이는 클라이언트의 수가 증가함에 따라 확장성의 저하로 이어지는 결과를 초래하였다. 이러한 구조와 달리 ArkFS는 클라이언트가 파일 시스템 계층 구조의 일부에 해당하는 메타데이터를 직접 관리하도록 함으로써 확장성을 개선한다. 이는 어떤 클라이언트가 파일 시스템 계층 구조의 어느 부분을 관리할 것인지에 관한 결정을 필요로 한다. 또한, 메타데이터를 관리 중이던 클라이언트가 실패했을 경우의 대책 역시 마련되어야 한다.

ArkFS는 이와 같은 문제를 해결하기 위해 리스 메커니즘을 이용한다. 리스는 일정 시간 동안 특정 데이터의 관리 권한을 특정 사용자에게 위임하는 방식이다 [6]. 주어진 시간이 끝나면 자동으로 권한이 회수된다는 점에서 동기화 메커니즘인 락(lock)과는 차이가 있다. ArkFS는 클러스터 내에 리스 관리를 수행하기 위한 리스 관리자를 둔다. 리스와 관련된 클라이언트의 요청은 리스 관리자를 통해 처리된다.

ArkFS의 파일 시스템 계층 구조는 각 디렉터리와 그 디렉터리 내에 포함된 정규 파일들을 하나의 단위로 하여 분할된다. 리스는 이 단위를 기준으로 관리되며, 한 번에 최대 하나의 클라이언트만이 특정 디렉터리의 리스를 얻을 수 있다. 이때 이 클라이언트를 해당 디렉터리의 리더라고 한다. ArkFS에서 디렉터리와 그 디렉터리 내의 정규 파일들에 대한 메타데이터의 관리는 해당 디렉터리의 리더가 수행한다.

[그림 2]는 파일 시스템 계층 구조의 예시이다. C1과 C2는 서로 다른 클라이언트를 의미하며, 빨간 점선과 파란 점선으로 표시된 영역은 현재 클라이언트 C1과 C2에 의해 관리되고 있는 부분을 의미한다.

ArkFS에서 디렉터리 또는 파일에 대한 접근이 어떻게 수행되는지 알아보자. [그림 2]에서 클라이언트 C2가 파일 `bar.txt`에 접근하는 과정을 예로 들어 설명한다. 그림에서 파일 `bar.txt`의 절대 경로는 `/home/doc/bar.txt`이며, C2는 해당 파일에 접근하기 위해 경로 내의

각 컴포넌트를 차례로 거쳐야 한다. 이를 위해 C2는 먼저 루트 디렉터리의 메타데이터에 접근해야 한다. C2는 루트 디렉터리와 디렉터리 home/이 다른 클라이언트에 의해 관리되고 있다는 사실을 현재로서는 알지 못한다고 가정한다. 현재 C2는 루트 디렉터리의 리더가 아니기 때문에 C2는 리스 관리자에게 접근하여 리스 획득을 시도해야 한다. 그러나 C1가 이미 루트 디렉터리를 관리하고 있기 때문에, 리스 관리자는 리스 획득 요청을 처리하는 대신 C1의 주소를 C2에게 전달한다. 그러면 C2는 전달받은 주소로 접근하여 C1에게 루트 디렉터리를 대상으로 하는 메타데이터 연산을 RPC를 통해 요청한다. C1은 요청된 파일 시스템 연산을 C2를 대신하여 수행하고, 그 결과를 C2에게 반환한다. [그림 3]의 ①~④는 이 과정을 나타낸다. 이후 C2는 home/의 아이노드 번호를 가지고 해당 디렉터리의 리스를 얻기 위해 리스 관리자에 다시 접근한다. 그러나 C1이 home/도 관리 중이기 때문에 앞서 설명한 과정을 다시 거쳐야 한다. 마지막으로 C1이 디렉터리 doc/의 아이노드 번호를 반환하면, doc/의 리더는 C2 자기 자신이므로 로컬에서 파일에 접근할 수 있다.

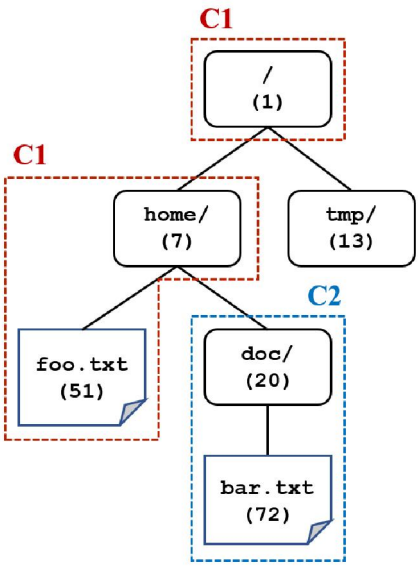


그림 2. 파일 시스템 계층 구조의 예

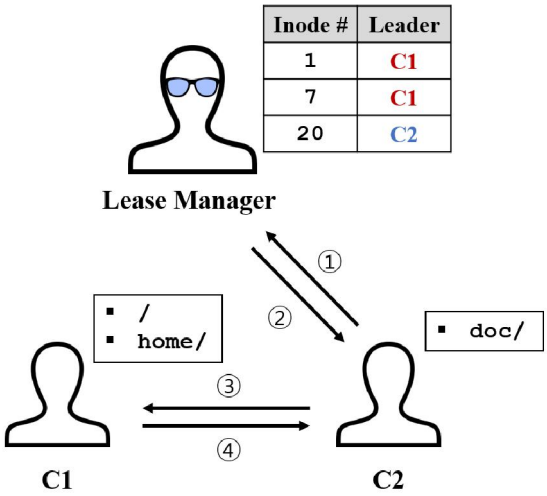


그림 3. 파일 시스템 연산을 대신 수행하는 과정

각 클라이언트는 이전에 접근했던 디렉터리에 대한 리스 정보를 리스 기간 동안 유지한다. 리스 기간을 포함한 리스에 대한 정보는 디렉터리의 리더에 관한 정보를 리스 관리자가 리스 획득을 요청한 클라이언트에게 전달할 때 함께 전달된다(예시에서는 [그림 3]의 ②에 해당). 따라서 리스가 아직 만료되지 않았다면, 이전에 리스 기간 내에 해당 디렉터리에 접근했던 다른 클라이언트는 리스 관리자에게 요청하지 않고도 디렉터리의 리더에게 접근할 수 있다.

이러한 동작 방식은 파일 시스템 연산을 수행할 때 각 컴포넌트에 해당하는 리더를 매번 접근해야 한다는 점에서 비효율적일 수 있다. 특히, 루트 디렉터리의 경우 경로에 항상 포함되어 있기 때문에 루트 디렉터리의 리더에게 연산 요청이 몰릴 가능성이 높다. 이를 완화하기 위해 ArkFS는 다른 클라이언트가 관리 중인 디렉터리의 리스 기간 내에서는 캐싱한 메타데이터를 사용할 수 있는 설정을 제공한다. 이 설정은 `rename()`, `chmod()`, `chown()`과 같은 연산이 리스 기간 내에 수행되었을 경우 디렉터리 또는 파일의 존재성이나 접근 제어 목록에 대한 정확성을 보장하지 않는다는 단점이 있지만, 그러한 연산이 자주 이루어지지 않는 환경에서는 성능상 이점이 있으므로 사용하는 것이 권장된다.

제 5 절 저널링

일반적인 파일 시스템에서 단일 저널 영역을 두는 것과는 달리, ArkFS는 전술한 바와 같이 전체 파일 시스템 계층 구조를 각 디렉터리와 디렉터리의 정규 파일을 포함하는 단위로 분할하고, 분할된 각 단위마다 하나의 저널 영역을 둔다. 저널 역시 하나 이상의 오브젝트의 형태로 관리되며, 디렉터리의 아이노드 번호로 저널 오브젝트에 접근할 수 있다. 디렉터리 또는 디렉터리 내의 정규 파일의 메타데이터 변경 사항은 해당 저널 영역에 기록되며, 저널 영역을 기록하는 주체는 디렉터리의 리더가 된다. 이처럼 디렉터리별로 저널 영역을 분할함으로써, 단일 저널을 두었을 때 발생하는 경합 문제를 자연스럽게 해소할 수 있다.

[그림 4]는 저널링 과정을 나타낸 것이다. 클라이언트가 파일 시스템 연산을 수행하는 과정에서 메타데이터에 변경 사항이 발생하면 러닝 트랜잭션의 형태로 메모리에 축적된다. 메모리상의 러닝 트랜잭션은 커밋 스레드에 의해 주기적으로 컴파운드 트랜잭션으로 전환되어 오브젝트 스토리지 내의 저널에 기록된다. 기록이 완료된 트랜잭션은 즉시 체크포인트 스레드에게 전달되며 체크포인트 후 해당 트랜잭션은 저널에서 제거된다. ArkFS는 writeback과 ordered의 두 가지 저널링 모드를 지원하며, ext4의 writeback 및 ordered 모드와 그 의미가 동일하다.

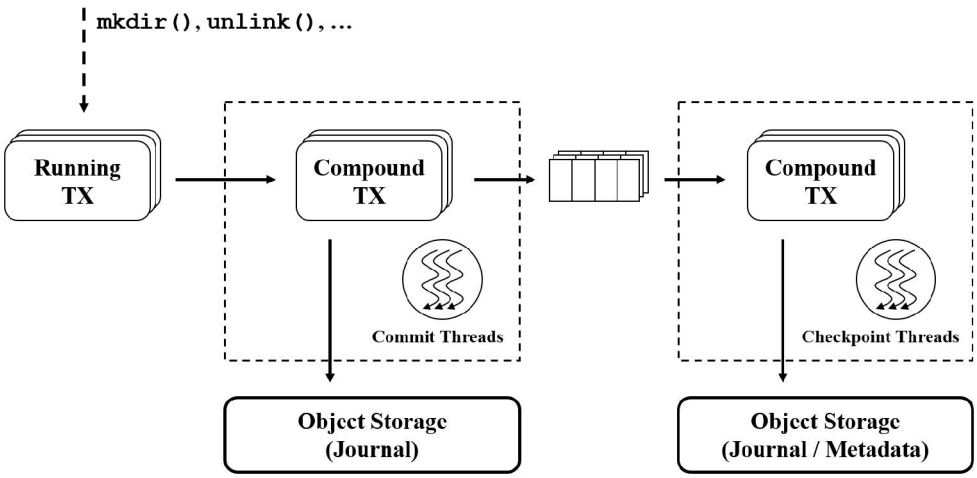


그림 4. 메타데이터의 저널링 과정

디렉터리별로 저널 영역을 분할하는 것은 그 장점이 명확하지만, `mkdir()`, `rmdir()`, `rename()`과 같이 여러 디렉터리에 걸친 연산의 원자성에 대한 문제를 유발한다. 예를 들어, 클라이언트 C1이 디렉터리 `src/`를, 클라이언트 C2가 디렉터리 `dst/`를 관리하고 있는 상황을 가정해 보자. 만일 또 다른 클라이언트 C3가 `src/` 내의 파일 `file`을 `dst/`로 옮기고자 한다면, 해당 `rename()` 연산은 C1과 C2의 저널 모두에 기록되어야 한다. 따라서 해당 연산이 원자적으로 모든 관련된 저널에 기록되기 위한 메커니즘이 필요하다. 이러한 문제에 대해 2단계 커밋 프로토콜(two-phase commit protocol) 등의 분산 알고리즘이 이미 존재하지

만, 잠재적인 모든 문제 구성에 대해 올바르게 동작하는 것은 아니며 원자적 연산의 실패 확률을 낮추는 것에 그 의미가 있다. 이론적으로, 신뢰할 수 없는 링크를 통해 연결된 여러 노드 간의 합의는 불가능하다는 사실이 증명된 바 있다.

ArkFS는 여러 디렉터리에 걸친 연산이 부분적으로 실패할 경우 복구할 수 있는 메커니즘을 제공한다. 앞서 예로 든 상황에서, 클라이언트 C3은 C1에게 rename() 연산을 요청한다. 그러면 C1은 자신의 저널에 rename() 연산을 기록하고, 파일의 목적지인 dst/의 리더인 C2에게 연산이 가능한지를 묻는다. C2는 해당 연산을 정상적으로 수행할 수 있다면 자신의 저널에 마찬가지로 rename() 연산을 기록하고, 그 결과를 C1에게 반환한다. 이 시점에서 C1과 C2가 저널에 기록한 트랜잭션은 아직 완료되지 않은 상태이다. C1는 C2으로부터 연산이 수행될 수 있다는 응답을 받았다면 전역 저널에 해당 연산의 결과를 기록한 후, C2와 C3에게 그 결과를 알린다. 이후 C1과 C2는 각자의 저널에 완료되지 않은 상태로 남아 있는 기록을 마무리한다. [그림 5]는 예시 상황에서 발생하는 커밋 과정을 나타낸 것이다.

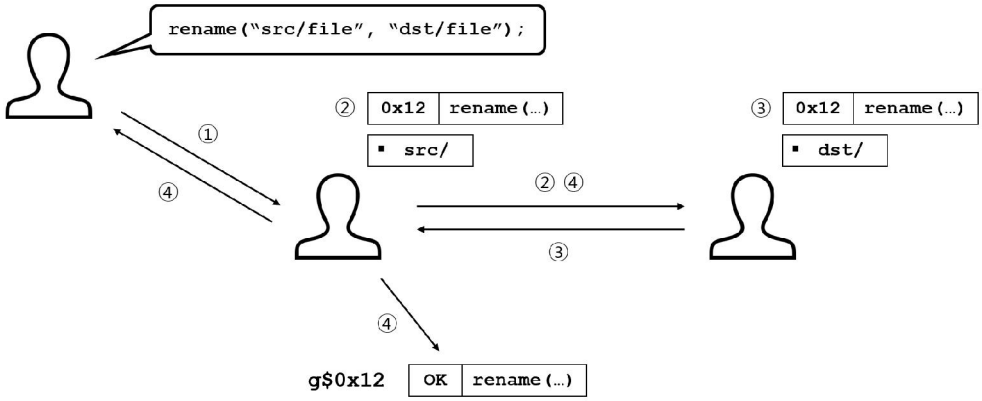


그림 5. 클라이언트 간 연산의 커밋 과정

만일 C1이 최종 결과를 C2에게 전송했을 때 C2가 실패하여 자신의 저널에 남아 있는 기록을 마무리하지 못했다면, 추후 dst/의 리스를 획득하는 클라이언트가 전역 저널을 확인하고 그 결과를 반영할 수 있다. 전역 저널은 여러 개의 오브젝트로 구성되며, 여러 디렉터리에 걸친 연산마다 하나의 오브젝트가 만들어진다. 해당 오브젝트는 각 디렉터리의 저널에 기록되는 트랜잭션 내의 트랜잭션 번호를 통해서 접근할 수 있다. [그림 5]에서 rename() 연산의 트랜잭션 번호는 0x12인 것을 확인할 수 있다.

제 5 장 실험 결과

제 1 절 실험 환경

실험은 Ceph RADOS와 AWS S3의 두 가지 오브젝트 스토리지를 백엔드로 하여 진행하였다. [표 1]은 분산 파일 시스템 클라이언트의 실험 환경을 나타낸다. Ceph RADOS를 백엔드로 하는 분산 파일 시스템의 경우 오브젝트 스토리지가 구동되는 서버의 스펙은 클라이언트와 동일하다. ArkFS는 각 오브젝트 스토리지 위에서 FUSE 프레임워크를 이용하여 구현하였다.

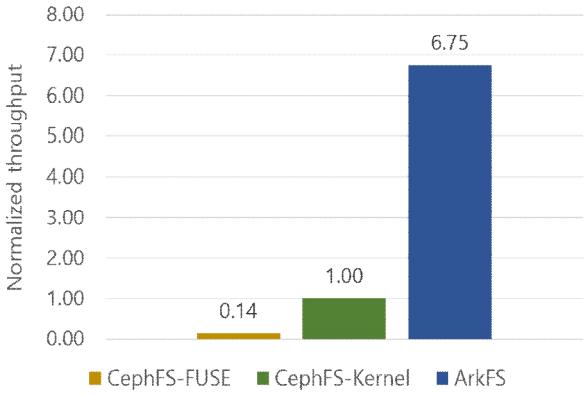
오브젝트 스토리지	Ceph RADOS	AWS S3
CPU	Intel Xeon E5-2640	c5n.18xlarge 72 vCPUs
메모리	128GiB	192GiB
네트워크	100Gbps	100Gbps

표 1. 실험 환경

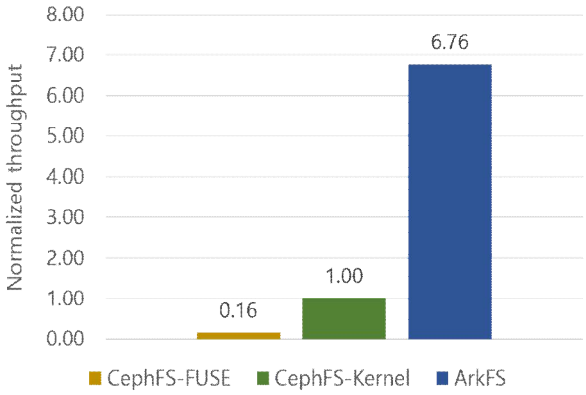
제 2 절 메타데이터 I/O 성능

메타데이터 워크로드에 대한 I/O 성능을 확인하기 위해 mdtest 벤치마크를 실행하였다. 구체적으로 mdtest의 여러 설정 중 mdtest-easy 벤치마크를 선택하였다. mdtest-easy의 경우 파일 데이터는 없으며 빈 파일을 생성하고, stat하고, 삭제하는 파일 시스템 연산으로 구성되어 있다. 백엔드 오브젝트 스토리지는 Ceph RADOS이다.

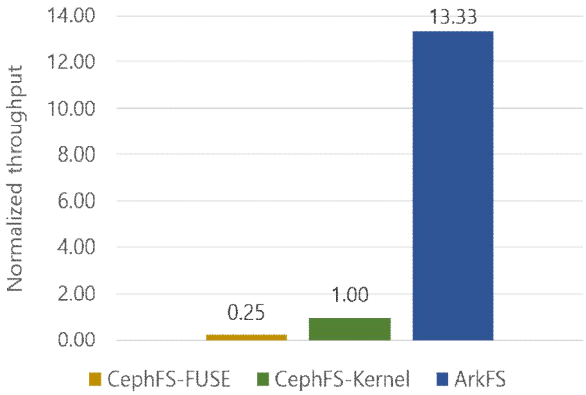
[그림 6]는 각각의 파일 시스템 연산에 대한 실험 결과를 나타낸다. 그래프의 값은 CephFS에서 측정한 결과를 기준으로 정규화된 값이다. CephFS의 경우 파일 시스템 연산을 처리하기 위해 메타데이터 서버에 접근해야 하지만 ArkFS의 경우 로컬 메모리에서 연산을 처리할 수 있기 때문에 더욱 나은 성능을 보인다. 커널 내에 구현된 CephFS (CephFS-Kernel로 표기)와 더불어 FUSE로 구현된 CephFS (CephFS-FUSE로 표기)에 대해서도 비교를 수행하였다. FUSE로 구현된 파일 시스템은 커널 내에 구현된 파일 시스템에 비해 커널 공간과 사용자 공간 간에 문맥 교환이 빈번하게 일어나며, 이로 인한 성능 저하를 그래프에서도 확인할 수 있다.



(a) create



(b) stat



(c) delete

그림 6. 메타데이터 I/O 성능

제 3 절 작은 파일에 대한 I/O 성능

작은 파일에 대한 I/O 성능을 확인하기 위해 Postmark 벤치마크를 실행하였다. 트랜잭션의 수는 5000이며 각각의 트랜잭션은 500개의 파일에 동시에 파일 시스템 연산을 수행한다. 트랜잭션은 create, read, append, delete로 구성되어 있다. [그림 7]는 실험 결과를 나타내며, CephFS 또는 S3FS에서 측정한 결과를 기준으로 정규화된 값을 의미한다. 메타데이터 I/O 성능을 측정한 실험과 마찬가지로 CephFS는 커널 내에 구현된 경우와 FUSE 프레임워크로 구현된 경우를 대상으로 실험하였다. 실험 결과 본 논문에서 제안한 파일 시스템은 두 경우보다 좋은 성능을 내는 것을 확인하였다. CephFS의 경우 이 실험에서도 역시 메타데이터 서버에 접근하는 것이 주된 오버헤드로 작용하며, 앞선 실험과 달리 메타데이터 뿐 아니라 데이터도 접근하므로 ArkFS와의 성능 차이가 줄어들었지만 여전히 낮은 성능을 보이는 것을 알 수 있다. AWS S3를 백엔드로 하는 파일 시스템의 경우에 대해서는 기존의 S3 기반의 오브젝트 스토리지인 S3FS와 성능을 비교하였다. S3FS의 경우 디스크를 데이터 캐시로 사용하기 때문에 기본적으로 성능이 현저히 낮은 것을 확인할 수 있다.

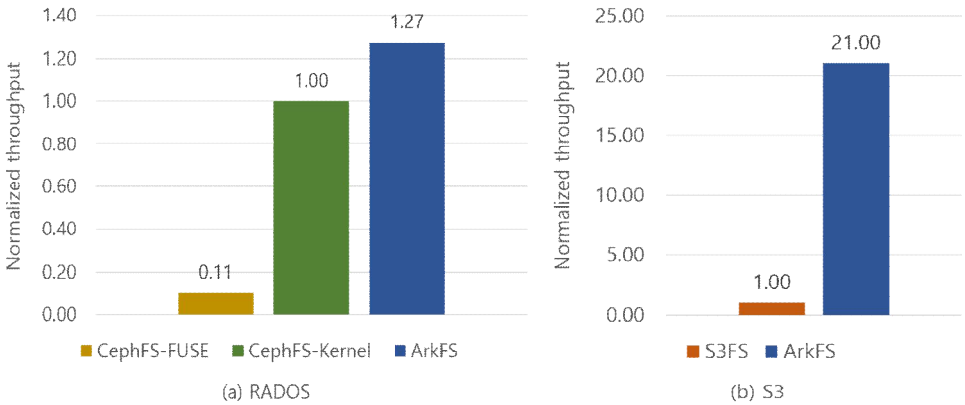


그림 7. 작은 파일에 대한 I/O 성능

제 4 절 클라이언트 수에 따른 I/O 성능

[그림 8]은 클라이언트 수를 1개에서 32개로 변화시켰을 때의 성능 변화를 나타낸 것이다. 오브젝트 스토리지는 RADOS이며, 클라이언트는 각자가 관리하고 있는 디렉터리를 대상으로 파일 시스템 연산을 수행한다. CephFS의 경우 일정 수준 이상으로는 클라이언트의 수를 증가시켜도 성능이 향상되지 않는다. 이는 클라이언트의 수가 증가함에 따라 메타데이터 서버로 요청이 집중되어 병목 현상이 발생하기 때문이다. 반면 ArkFS의 경우 메타데이터 서버가 없고 요청을 대부분 로컬에서 수행할 수 있으므로 클라이언트 수의 증가에 따라 I/O 성능이 함께 증가한다.

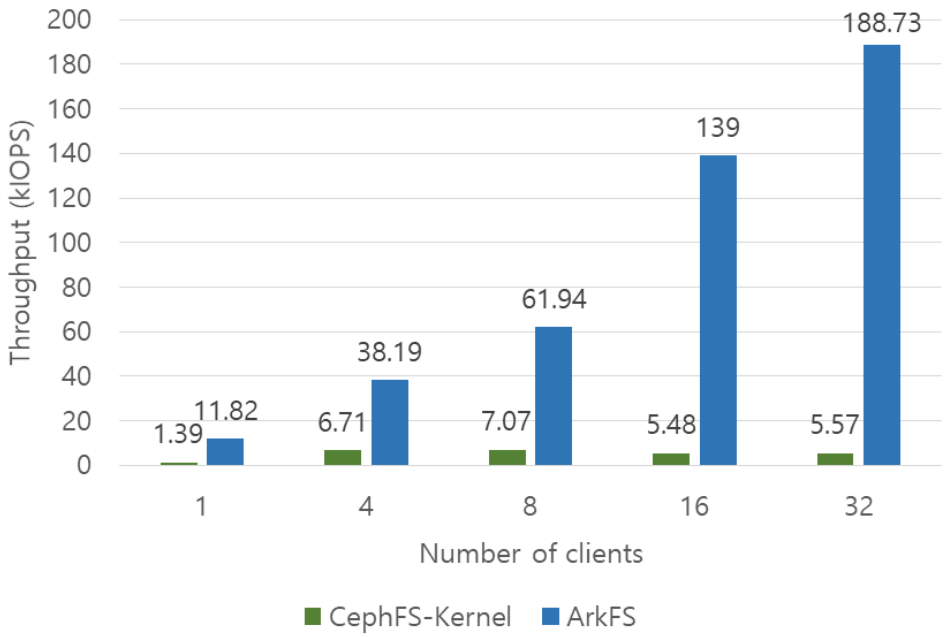


그림 8. 클라이언트 수에 따른 I/O 성능

제 5 절 저널링 소요 시간

저널링 파일 시스템에서 커밋 과정이 완료되면 원본에 반영되지 않은 파일 시스템의 변경사항이라도 저널을 참조하여 복구할 수 있으나, 시스템의 실패가 발생하지 않는다면 변경 사항이 체크포인트 과정까지 완료됨을 기대하는 것이 일반적이다. ArkFS에서는 리스가 만료된 후 다른 클라이언트에게 해당 디렉터리의 리스를 부여하기 전에 이전 리더가 체크포인트 과정까지 끝마치도록 대기하는데, 따라서 체크포인트 시간을 단축하는 것이 중요하다.

[그림 9]는 이에 대한 실험으로 RADOS를 백엔드로 하여 파일의 수를 1개에서 10000개까지로 변화시키면서 커밋 과정을 거쳐 체크포인트 과정까지 완료되는 데 걸린 시간을 나타낸 것이다. 실험은 하나의 디렉터리를 대상으로 하여 해당 디렉터리의 리더가 로컬에서 파일을 생성하는 과정을 수행하였다. 응용은 파일의 생성 시간 간에 지연이 없이 계속 파일을 생성하는 burst한 요청을 파일 시스템으로 전달한다. 실험 결과에서도 확인할 수 있듯이 ArkFS는 단시간 내에 요청되는 많은 수의 요청에 대해서도 짧은 시간 내에 커밋과 체크포인트 과정을 수행한다.

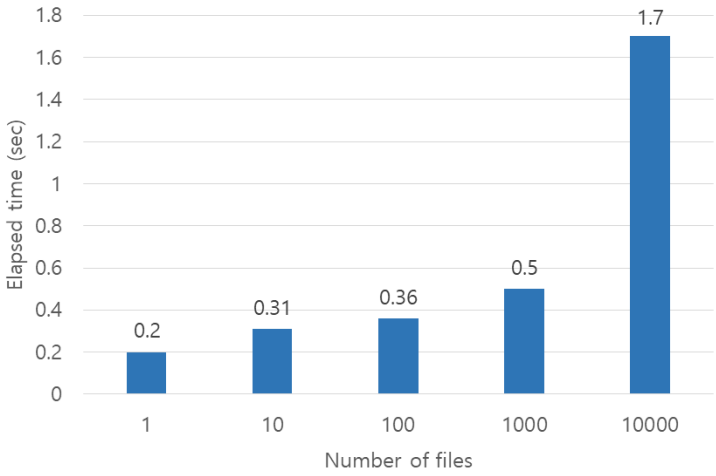


그림 9. 저널링 소요 시간

제 6 장 결론

본 연구에서는 오브젝트 스토리지 기반의 확장성 있는 분산 파일 시스템의 구조를 제안하였다. 기존의 분산 파일 시스템이 기능 부족 또는 확장성 저하의 문제를 겪는 것을 파악하였으며, 메타데이터를 효과적으로 관리하기 위한 파일 시스템 계층 구조의 분할 방법을 제시하였다. 이는 메타데이터 관리 전용 노드를 제거할 수 있도록 함으로써 확장성을 크게 개선하였다. 실험 결과를 통해 기존의 시스템 대비 높은 I/O 성능 및 확장성을 보이는 것을 확인하였으며, 이러한 결과는 특히 메타데이터 I/O에서 두드러지게 나타난다.

참 고 문 헌

[1] AWS S3, https://aws.amazon.com/s3/.

[2] S3FS. https://github.com/s3fs-fuse/s3fs-fuse.

[3] SVFS. https://github.com/ovh/svfs.

[4] Arnold, J. Openstack swift: Using, administering, and developing for swift object storage. O'Reilly Media, Inc. (2014).

[5] Inman, J., Vining, W., Ransom, G., and Grider, G. MarFS, a Near-POSIX interface to cloud objects. USENIX ;login: 42, 1 (Jan. 2017).

[6] Gray, C. G., and Cheriton, D. R. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP 89), pp. 202-210.

[7] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D., & Maltzahn, C. Ceph: A scalable, high-performance distributed file system. In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 06), pp. 307-320.

Abstract

A Scalable Distributed File System Based on Object Storage

Injae Kang
Department of Computer Science & Engineering
The Graduate School
Seoul National University

Object storage is widely deployed in HPC environments thanks to its highly scalable and cost-effective structure. Previous researches have proposed object storage-based distributed file systems to utilize the scalability of object storage, but they have problems of failing to comply with POSIX standards or showing low scalability. This paper proposes a metadata structure for improving the scalability of an object storage-based distributed file system and a technique for efficiently journaling metadata.

Keywords : Distributed file system, Object storage
Student Number : 2020-29665