JDCS
디지털콘텐츠학회논문지
Journal of Digital Contents Society
Vol. 24, No. 11, pp. 2841-2847, Nov. 2023

Check for updates

컨테이너를 위한 고성능 오버레이 파일시스템의 설계 및 평가

한 혁*
*동덕여자대학교 컴퓨터학과 부교수

High-Performance OverlayFS for Containers

Hyuck Han*
*Associate Professor, Department of Computer Science, Dongduk Women's University, Seoul 02748, Korea

[요 약]

리눅스 도커 컨테이너는 기존의 파일시스템들을 가상 파일시스템으로 사상하는 오버레이 파일시스템을 통해 파일 접근을 지원한다. 컨테이너에서 동작하는 프로세스는 하위 파일시스템의 파일에 접근할 때 적절한 접근 권한이 필요하다. 이를 위해 오버레이 파일시스템은 입출력 연산 중에 마운트한 사용자의 자격 증명을 빌려 하위파일시스템의 파일에 접근하게 한다. 그러나 이러한 방식은 고성능 저장장치를 탑재한 서버 환경에서 입출력 중심적인 프로세스가 컨테이너에서 동작할 때 심각한 오버헤드를 보인다. 이러한 문제를 해결하기 위해 본 논문은 마운트한 사용자의 자격 증명을 빌리는 과정을 매 입출력 연산에서 수행하는 것이 아닌 컨테이너 상의 프로세스가 파일 연산을 시작할 때 한 번만 빌려 캐싱하는 방법을 제안한다. 그리고 제안하는 방법을 리눅스 상에서 구현하여 입출력 벤치마크를 이용해서 성능을 평가하였다. 성능 평가 결과는 본 논문에서 제안하는 방법이 기존의 방법보다 최대 23배 높은 성능을 보여준다.

[Abstract]

Docker containers in Linux support file access through OverlayFS, which maps backing file systems to a virtual file system. Processes running within these containers require appropriate permissions when accessing files in the backing file system. To achieve this, OverlayFS overrides the mounter's credentials during input/output (I/O) operations, allowing access to files in the backing file system. However, this approach involves a significant overhead when I/O-intensive processes run in containers on servers equipped with high-performance storage devices. To address this issue, this paper proposes a method to override and cache the credentials only once when a process on a container starts a file operation, instead of repeating the override process for every I/O operation. We implement the proposed approach on Linux, and evaluate our method using multiple I/O benchmarks. The performance evaluation results demonstrate that the proposed method achieves up to 23 times higher performance than the existing approach.

색인어 : 컨테이너, 도커, 오버레이파일시스템, 자격 증명, 참조 카운팅

Keyword : Container, Docker, OverlayFS, Credential, Reference Counting

http://dx.doi.org/10.9728/dcs.2023.24.11.2841

Received 19 August 2023; **Revised** 26 September 2023
Accepted 27 September 2023

***Corresponding Author; Hyuck Han**

Tel: +82-2-940-4687
E-mail: hhyuck96@dongduk.ac.kr

Ⅰ. 서 론

최근에 컨테이너는[1],[2] 클라우드에서 배포를 위해 널리 사용되고 있다. 컨테이너를 이용하여 서비스 제공자들은 런타임, 시스템 도구, 라이브러리 및 설정 등을 패키지화하므로 소프트웨어/서비스 배포 및 실행 프로세스가 간소화된다. 일반적으로 클라우드 제공자는 컨테이너화된 소프트웨어/서비스를 실행하거나 마이크로 서비스를 구축하기 위한 자체 서비스를 클라우드 사용자들에게 제공하고 있다.

도커[1]는 컨테이너화된 애플리케이션을 위한 사실상의 표준이며 오픈 소스 생태계와 협력하여 소프트웨어 개발자가 애플리케이션을 개발하고 쉽게 배포할 수 있게 한다. 도커 컨테이너 이미지는 일반적으로 기존 파일 시스템(예: ext4, XFS)을 지원되는 스토리지 드라이버(예: overlay 혹은 overlay2)를 사용하여 일련의 계층으로 만들어진다. 오버레이 파일시스템은 도커 애플리케이션을 실행하기 위해 여러 계층을 통합하여 단일 뷰를 제공한다.

그림 1은 오버레이 파일시스템의 개요를 보여준다. 오버레이 파일시스템은 마운트 위치의 역할을 하는 가상의 병합된 디렉토리를 이용하여 통합된 뷰를 제공하며, 이 디렉토리는 여러 계층(컨테이너 계층과 복수 개의 이미지 계층)으로 이루어진다. 도커 컨테이너에서 새 파일이 생성되면 그 파일은 컨테이너 계층에 저장된다. 이미지 계층은 읽기 전용이며, 이미지 계층의 파일을 업데이트는 다음과 같이 처리된다. i) 이미지 계층의 파일을 쓰기 가능 계층인 컨테이너 계층으로 복사하고, ii) 복사된 파일에 업데이트 연산을 수행한다. 파일을 읽을 때는 컨테이너 계층에서 이미지 계층으로 하향식으로 파일을 검색하여 가장 최신의 데이터를 읽는다. 따라서 이러한 동작은 여러 컨테이너가 동일한 이미지 계층을 공유할 수 있으면서 컨테이너 별로 컨테이너 계층이 있도록 하여 소프트웨어/서비스를 배포할 때 이미지 크기를 줄일 수 있게 한다.

오버레이 파일시스템은 기존의 리눅스 파일 시스템과 유사하게 액세스 제어(예, DAC와 MAC)를 지원하며 파일, 아이노드 및 디렉토리 연산의 경우 오버레이 파일시스템으로 마운트한 도커 데몬 프로세스(마운터)의 권한을 이용하여 수행한다. 이를 위해 파일 연산 전에 컨테이너에서 실행되는 프로세스의 자격 증명을 마운터의 자격 증명으로 대체하여 파일 연산을 수행한다. 그리고 파일 연산 후에 마운터의 자격 증명을 반환하여 원래대로 되돌린다. 이러한 자격 증명의 대체 및 반환은 컨테이너의 파일 연산에 추가적인 오버헤드를 발생시킨다.

컨테이너가 실행한 프로세스가 데이터 집중적이지 않으면 자격 증명과 관련된 오버헤드는 크게 드러나지 않는다. 그러나 하나의 컨테이너에서 여러 개의 프로세스(또는 스레드)들이 동시에 데이터 집중적인 작업을 실행한다면 오버헤드는 상대적으로 커지며 멀티코어 CPU와 고성능 SSD를 탑재한 서버 시스템에서 이 오버헤드는 더욱 커질 수 있다. 예를 들어 웹 서버는 많은 클라이언트의 요청들을 동시에 처리하기 위해 여러 프로세스/쓰레드를 실행한다. 이 때 실행된 프로세스/쓰레드들은 컨테이너에서 동시에 많은 수의 파일 연산들을 수행하여 자격 증명과 관련된 오버헤드를 노출시킬 수 있다.

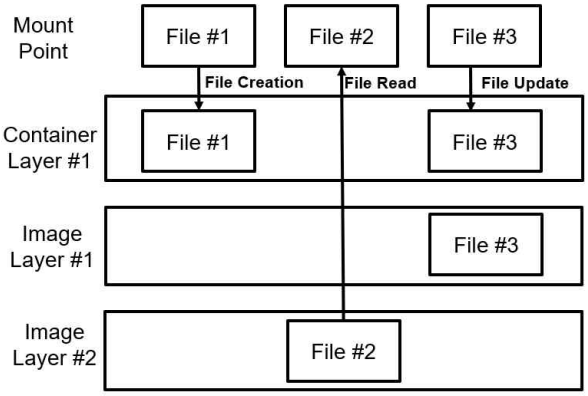


그림 1. 오버레이 파일시스템 개요
Fig. 1. Overview of OverlayFS

본 논문에서는 먼저 수십 개의 CPU 코어와 고성능 SSD가 장착된 현재 고성능 컴퓨터 시스템에서 자격 증명과 관련된 오버헤드가 오버레이 파일시스템의 확장성을 방해한다는 것을 보여준다. 그리고 자격 증명과 관련된 오버헤드를 줄이기 위한 새로운 기법을 제안한다. 제안하는 기법은 파일, 아이노드, 디렉토리 연산 전후에 자격 증명의 대체 및 반환 작업을 수행하는 것이 아니라 컨테이너에서 프로세스가 처음 파일 연산을 실행될 때 마운터의 자격 증명을 빌려와서 캐싱하고 프로세스가 종료될 때 자격 증명을 반환하는 것이다. 이를 통해 병목 지점인 자격 증명의 참조 카운터에 대한 경쟁을 크게 완화시킨다. 즉, 사용자 프로세스는 미리 빌려온 자격 증명을 사용하여 파일, 아이노드, 디렉토리 연산에 필요한 자격 증명을 대체함으로써 이러한 경쟁을 피한다. 또한 제안한 기법을 리눅스에서 구현하고 마이크로/매크로 벤치마크를 사용하여 성능을 평가하였다. 실험 결과를 통해 제안된 기법은 기존의 방법보다 마이크로 벤치마크에서 최대 23.7배 그리고 매크로 벤치마크에서 14배 정도 성능이 개선되었음을 확인하였다.

본 논문의 나머지 부분은 다음과 같이 구성되어 있다. Ⅱ장은 본 연구와 관련된 연구를 제시한다. Ⅲ장은 본 연구에 제안한 기법과 그 기법의 구현에 대해 설명한다. Ⅳ장에서 제안하는 기법의 성능을 평가하고, 마지막으로 Ⅴ장에서 본 연구를 마무리한다.

Ⅱ. 관련 연구

이 장에서는 본 연구와 관련된 선행 연구를 요약하고자 한다. 먼저 오버레이 파일시스템의 성능을 향상시키는 선행 연

구를 설명하고 컨테이너 및 파일 시스템의 성능을 향상 시키는 연구들을 제시하여 본 연구와의 차별성을 보인다.

2-1 오버레이 파일시스템과 관련된 선행 연구

입출력 측면에서 컨테이너에서 실행되는 소프트웨어/서비스의 성능을 분석하고 오버레이 파일시스템의 쓰기 혹은 복사 후 쓰기(Copy-on-Write) 연산을 최적화하는 많은 연구가 수행되었다.

BAOVERLAY[3]는 파일 단위가 아닌 블록 단위로 복사 작업을 수행하는 블록 수준 액세스가 가능한 오버레이 파일 시스템이며, 대용량 파일의 일부만 업데이트할 때 컨테이너 계층으로 복사하는 오버헤드를 줄였다. HP-Mapper 스토리지 드라이버[4]는 입출력 오버헤드를 줄이기 위해 2단계 매핑 전략을 통해 세분화된 복사 후 쓰기 작업을 제안하였다. Mizusawa et al.의 연구[5]에서 복사 작업 중에 발생하는 가상 파일 시스템의 메타데이터 동기화 오버헤드를 줄이기 위해 인터벌 기반 메타데이터 동기화 기법을 제안하였다. Wu et al.의 연구[6]에서는 응용 프로그램이 복사 작업을 많이 실행할 때 가상파일시스템의 컨테이너 간 글로벌 잠금에 대한 경합을 무시할 수 없음을 보였고, 이를 해결하기 위해 여러 컨테이너에서 잠금 없이 동시 복사 작업을 수행하게 해주는 계층 인식 가상파일시스템(LaVFS)를 제안하였다. Xu et al.의 연구[7]에서 고성능 SSD를 탑재한 시스템에서 컨테이너를 수행했을 때 입출력 집중적인 응용 프로그램의 성능을 평가하였다. 오버레이 파일시스템이 랜덤 읽기 워크로드에서 장치 성능과 거의 비슷한 성능을 보이지만 랜덤 쓰기 워크로드에서 성능이 50% 이상 감소한다는 것을 보였다.

위의 연구들과 달리 본 논문은 하나의 컨테이너에서 여러 프로세스/쓰레드들이 동시에 많은 양의 파일 연산들을 수행할 때 발생하는 오버레이 파일시스템의 확장성 문제를 다루고 있다.

2-2 컨테이너와 관련된 선행 연구

최근 다양한 종류의 입출력 워크로드를 고려하여 입출력 집중적인 응용들을 위한 효율적인 컨테이너 최적화 전략들이 [8]-[10] 제안되었다. Bhimani et al.의 연구[8]는 워크로드 특성이 다른 애플리케이션들을 컨테이너로 배포하여 동시에 실행하면 입출력 리소스의 활용률이 낮아지고 애플리케이션 간에 입출력 처리율이 불균형해질 수 있음을 보였다. 이 문제를 해결하기 위해 실행 시간에 동적으로 컨테이너를 예약하고 실행하여 동시에 실행할 컨테이너 그룹을 결정하는 컨테이너 컨트롤러를 제안했다. Yan et al.의 연구[9]에서 메타데이터를 메모리 페이지에 고정하고 로컬 디스크를 원격에 저장된 이미지용 캐시로 활용하여 입출력 지연 시간을 줄이는 2단계 캐싱 아키텍처를 제안했다. Xu et al.의 연구[10]는 다양한 도커 스토리지 드라이버의 성능을 비교하여 고성능 SSD를 가장 효과적으로 활용할 수 있는 설정을 제시하였다. 또한 컨테이너로 배포된 NoSQL 데이터베이스 확장성을 저해하는 입출력 리소스 병목 현상을 보이고 더 높은 성능을 위해 최적화된 리소스 할당 전략을 제시하였다.

위의 연구들은 컨테이너 수준의 입출력 작업 최적화에 초점을 맞추고 있고, 본 연구는 오버레이 파일시스템 수준의 성능 최적화를 다루고 있다.

2-3 파일 시스템의 확장성과 관련된 선행 연구

파일 시스템의 확장성 문제를 다루는 많은 연구들이 수행되었다[11]-[13]. ScaleFS[11]는 성능을 위해 CPU 코어마다 로그를 할당하여 사용한다. 또한, 인-메모리 파일 시스템과 디스크 기반 파일 시스템을 분리하고, 이들 사이의 데이터 교환 연산을 수행할 때 캐시 라인 충돌을 방지하여 성능을 효과적으로 확장할 수 있다. 파일 연산의 동시성을 높이기 위해 세분화된 범위 잠금을 사용하여 멀티 코어 서버에서 병목 현상을 완화할 수 있는 방법들도 이전 연구에서 제안되었다[12],[13].

입출력 집중적인 응용이 데이터에 접근할 때 최소한의 논리적 경합을 갖거나 전혀 하지 않는 경우에도 멀티 코어 시스템의 최신 파일 시스템은 확장성이지 않음을 최근의 연구에서 드러났다[14]. 또한, 확장성 문제의 근본 원인 중 하나가 공유 참조 카운터에 원자 연산들이 동시에 동일한 캐시 라인을 업데이트하려고 경쟁하는 것임을 보였다. 이에 시스템 소프트웨어 분야에서 참조 카운팅 기법 자체를 확장 가능하게 만들기 위해 많은 연구가 진행되었다[11],[15],[16]. Son et al.의 연구[17]에서는 잠금이 없는 데이터 구조와 멀티 스레드 기반의 저널 쓰기 연산을 이용한 고성능 저널링 기법을 제안했고, ext4 파일 시스템과 저널링 블록 장치(JBD2)에 구현했다.

위에서 제시된 연구들의 대상은 호스트 시스템에서 실행되는 파일 시스템이다. 반면, 본 연구는 앞서 설명한 것처럼 자격 증명과 관련하여 경합 특성을 가진 컨테이너가 사용하는 오버레이 파일시스템에 초점을 맞추고 있다.

Ⅲ. 최적화 기법 설계 및 구현

이 장에서는 오버레이 파일시스템의 병목 지점과 그 원인을 자세히 제시하고 병목 문제를 해소할 수 있는 방법을 제안하고 리눅스에서 구현한 것에 대해 설명한다.

3-1 연구 동기

오버레이 파일시스템에서 실제 입출력 연산을 수행하기 전에 자격 증명을 대체하고 입출력 연산이 끝난 후에 자격 증명을 반환하는 것은 컨테이너의 파일 연산에 추가적인 오버헤

드를 발생시킨다. 특히 데이터베이스 및 웹 서버와 같은 응용들은 다수의 클라이언트의 동시 요청들을 처리하기 위해 여러 입출력 작업 프로세스/쓰레드들을 생성하여 입출력 연산들을 수행하기 때문에 자격 증명과 관련된 오버헤드는 더 커진다.

자격 증명과 관련된 오버헤드를 검증하기 위해 FxMark 벤치마크[14]의 DRBL 워크로드를 사용하여 도커 컨테이너의 입출력 성능과 호스트 시스템의 입출력 성능을 본 논문의 4장의 실험 환경에서 비교하였다. DRBL 워크로드에서 각 프로세스는 전용 파일의 블록을 반복적으로 읽기 때문에 프로세스 간에 논리적 경합이 없다. 그리고 동시 프로세스들 사이에 논리적 경합이 없기 때문에 대부분의 파일 시스템은 이 워크로드에서 선형적으로 확장되는 것으로 알려져 있다[25]. 그러나 그림 2와 같이 DRBL 워크로드에서 오버레이 파일시스템은 입출력 프로세스의 수가 증가함에 따라 호스트 시스템 대비 성능이 확장되지 않는다. 그림에서 보는 바와 같이 컨테이너는 단일 프로세스 경우에서 호스트 처리량의 약 82%정도이지만 프로세스 수가 증가함에 따라 상대적인 처리량은 감소하여 64개 프로세스에서는 3% 정도로 감소한다.

프로세스의 수가 증가함에 따라 성능 격차가 증가한다는 것은 오버레이 파일시스템에 하나 이상의 병목 지점이 있다는 것을 의미한다. 병목을 확인하기 위해 응용 프로세스가 읽기 연산을 수행하는 경우의 함수 호출 흐름을 정리했다. 호스트 시스템에서의 읽기 연산은 다음과 같은 비교적 단순한 경로를 가지고 있다.

read() → sys_read() → vfs_read() →
generic_file_read_iter()

컨테이너의 읽기 연산은 가상 파일 시스템에서 오버레이 파일시스템의 읽기 함수를 호출하도록 되어 있다.

read() → sys_read() → vfs_read() → ovl_read_iter()

오버레이 파일시스템의 읽기 함수(ovl_read_iter)는 아래와 같이 자격 증명을 대체하고 복원하는 절차가 추가되기 때문에 호스트 시스템의 경우보다 더 복잡한 호출 흐름을 가진다. 오버레이 파일시스템의 다른 파일 연산들도 비슷한 절차가 추가되어 있다.

override_creds()→generic_file_read_iter()→
revert_creds()

override_cred/revert_cred 함수들이 자격 증명을 대체하고 복원한다. 자격 증명의 대체/복구를 구현하기 위해 오버레이 파일시스템으로 마운트하는 프로세스의 자격 증명에 대해 참조 카운팅 기법을 사용하고 있다. 리눅스의 참조 카운팅 기법은 카운터 변수에 프로세스 사이의 동기화된 액세스를 강제하는 atomic 연산, 즉 atomic_inc 및 atomic_dec_and_test를 사용한다. 이 atomic 연산들이 만들어내는 동기화 오버헤드는 동시 프로세스의 수가 증가함에 따라 현저하게 증가하며, 이는 프로파일링 도구로 확인할 수 있다.

그림 2의 컨테이너 결과에서 프로세스가 1개인 경우에 atomic 연산에 대해 작은 양의 CPU 시간(0.06%)을 사용하였다. 그러나 프로세스가 64개인 경우에는 자격 증명의 공유 참조 카운터를 원자적으로 업데이트하는 데에 약 68%의 CPU 시간을 사용했다. 모든 파일 연산에 자격 증명의 대체와 복원이 필요하므로 오버레이 파일시스템에서 이러한 경합의 강도는 접근하는 데이터의 유형(데이터 또는 메타데이터), 모드(읽기 또는 쓰기) 또는 데이터/메타데이터의 공유 수준(낮음에서 높음)에 관계없이 파일 연산의 빈도에만 결정된다. 이는 atomic 연산의 증가된 수행 시간(68%)은 컨테이너의 입출력 연산의 상대적 성능 저하(82%→3%)의 근본 원인임을 보여준다.

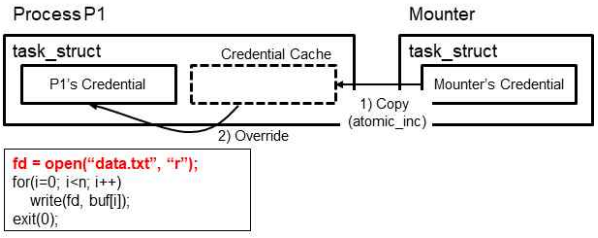
그림 2. 컨테이너 상의 파일 입출력 성능(FxMark의 DRBL, 64 core CPU, NVMe SSD)
Fig. 2. File I/O performance of container(FxMark DRBL, 64 core CPU, NVMe SSD)

3-2 최적화 기법

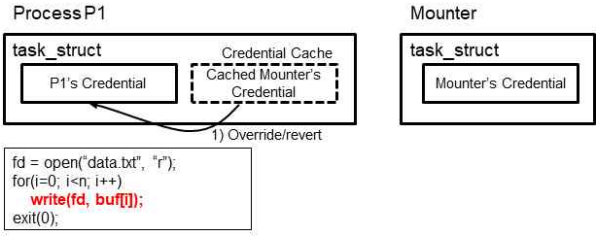
참조 카운팅 기법은 객체에 대한 참조가 없을 때(즉, 참조 카운트가 0일 때) 공유 객체를 안전하게 삭제할 수 있도록 해준다. 리눅스의 자격 증명 구조체는 해당 자격 증명에 대한 참조 카운터를 포함하며, 자격 증명의 참조 카운터가 0일 때 자격 증명 객체를 삭제할 수 있다. 오버레이 파일시스템으로 마운트 하는 사용자의 자격 증명은 컨테이너의 사용자 프로세스 간에 공유되기 때문에 프로세스가 자격 증명을 대체하거나 복원할 때마다 해당 자격 증명 구조체의 참조 카운터를 증가 혹은 감소시켜야 한다.

컨테이너에 오버레이 파일시스템이 마운트되면 마운터의 자격 증명에 대한 참조 카운터는 1로 설정되고 파일 연산들을 모두 종료하고 컨테이너가 종료될 때에 오버레이 파일시스템을 마운트 해제하여 0이 된다. 따라서 참조 카운터는 사용자

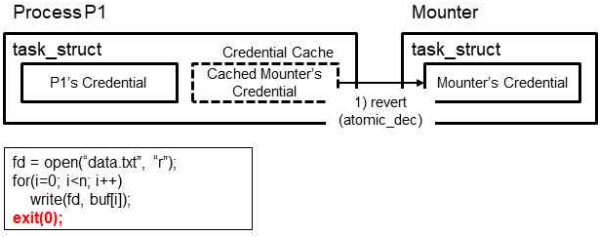
프로세스에 의해 0이 될 수 없으며 컨테이너가 동작하는 동안에 마운터의 자격 증명이 변경되지 않는다. 사용자 프로세스들은 파일 연산을 수행할 때마다 항상 동일한 마운터의 자격 증명을 이용하여 자격 증명을 대체/복원한다.



(a) 마운터의 자격 증명 캐싱
(a) Caching the credential of the mounter



(b) 캐시된 자격 증명을 이용한 대체 및 반환
(b) Override and revert with the cached credential



(c) 캐시된 자격 증명의 반환
(c) Revert the cached credential

그림 3. 프로세스 별 자격 증명 대체/반환
Fig. 3. Override/revert credentials per a process

본 연구에서 제안하는 최적화 기법은 컨테이너 내의 사용자 프로세스들이 공유하는 마운터의 자격 증명이 동일하다는 점을 이용한다. 즉, 프로세스들이 매번 마운터의 자격 증명으로 자신의 자격 증명을 대체/복원하는 대신 프로세스 내의 캐시된 자격 증명 버전을 생성하고 기존에 사용했던 자격 증명 대신 캐시된 자격 증명을 이용한다. 프로세스 내의 캐시된 자격 증명은 다른 프로세스와 공유되지 않기 때문에 참조 카운터를 관리하지 않아도 된다. 그리고 마운터의 자격 증명은 프로세스에서 캐시 버전을 생성할 때 참조 가운터를 증가시키고 (+1), 캐시 버전을 삭제할 때 참조 카운터를 감소시킨다(-1). 이러한 자격 증명 캐싱 방법은 마운터의 자격 증명에 대한 참조 카운터를 업데이트할 때 발생하는 캐시 경쟁을 크게 완화시켜 오버레이 파일시스템의 확장성과 성능을 개선한다.

그림 3은 제안하는 자격 증명 캐싱 기법의 동작 흐름을 보여준다. 자격 증명의 캐시된 버전을 관리하기 위해 리눅스의 프로세스에 대응하는 task_struct 구조체에 cached_cred라는 자격 증명 구조체 변수를 추가한다. 프로세스가 시작할 때 cached_cred 변수는 널(NULL)로 초기화된다. 그리고 프로세스가 파일 연산을 실행할 때 해당 프로세스의 task_struct의 cached_cred 변수를 이용하여 자격 증명 관련된 연산을 실행하도록 시도한다. 만약 task_struct 객체의 cached_creds 변수가 널(NULL)이면, i) 마운터의 자격 증명에 대한 참조 카운터를 원자적으로 1만큼 증가시키고, ii) 마운터의 자격 증명 구조체를 task_struct 객체의 cached_creds 변수에 복사하여 캐싱한다. 그림 3(a)는 이 과정의 예를 보여준다. P1 프로세스가 시작하여 data.txt 파일을 열려고 할 때 P1 프로세스에 대응하는 task_struct 객체의 cached_creds 변수가 널(NULL)이므로 위의 작업을 실행하고 캐싱된 마운터의 자격 증명을 이용하여 나머지 파일 열기 작업을 수행한다.

그림 3(b)에서는 P1 프로세스의 cached_creds 변수에 마운터의 자격 증명 객체가 복사된 후에 파일 연산을 수행하는 경우이다. 이 경우에는 단순히 cached_creds에 저장된 자격 증명을 이용하여 파일 연산을 수행하며 cached_creds에 대응하는 참조 카운터 변수가 없기 때문에 참조 카운터 변수에 대한 경쟁이 사라진다. 그리고 그림 4(c)와 같이 프로세스가 종료되는 시점에 마운터의 자격 증명에 대응하는 참조 카운터를 원자적으로 감소시킨다.

Ⅳ. 실험 및 평가

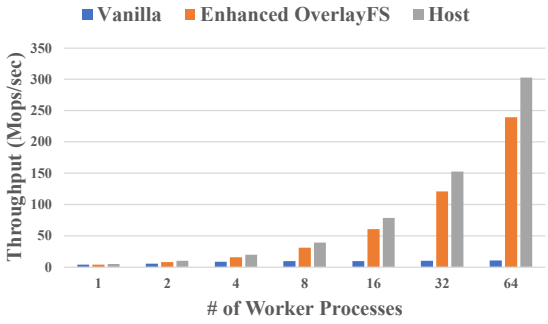
4-1 실험 환경

제안하는 자격 증명 캐싱 기법을 리눅스 커널 5.15.6에 구현하였다. Intel Xeon Platinum 8358 프로세서 2개가 장착된 64코어 시스템에서 모든 실험을 수행하였다. 이 시스템에 256 GiB DDR4 DRAM와 3.2 TB NVMe SSD(Intel SSD DC P4610)를 장착하였다. 오버레이 파일시스템의 하위 파일시스템으로 ext4를 사용하였으며, 컨테이너의 프로세스 수는 1 ~ 64개였으며 프로세스들을 서로 다른 코어에서 실행하였다.

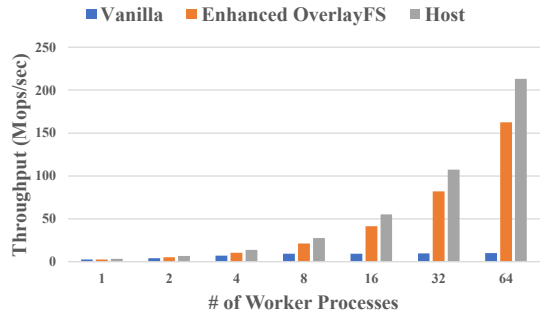
4-2 실험 결과

그림 4는 FxMark에서 두 개의 워크로드를 이용해서 파일 데이터를 입출력하는 벤치마크를 수행한 결과이다. DRBL(전용 파일에서 블록 읽기) 그리고 DWOL(전용 파일로 블록 덮어쓰기)의 자격 증명 대체 및 복원과 관련된 평가 결과이다. 이 결과는 64개의 프로세스에서 제안된 기법(Enhanced OverlayFS)이 호스트 시스템 성능의 약 75%를 보여준다.

이러한 안정적인 성능은 마운터의 자격 증명을 캐싱하여 참조 카운터에 대한 경합 오버헤드를 크게 줄였기 때문이다. 절대 성능 측면에서는 호스트 시스템과 제안된 기법 모두 64개 프로세스까지 선형적으로 확장되는 반면 기존의 오버레이 파일시스템의 경우(Vanilla)는 8개의 프로세스까지 느리게 증가하다가 그 후에는 더 이상 증가하지 않는다. 64 프로세스 사례에서 제안된 기법은 DRBL 및 DWOL 워크로드에서 기존의 오버레이 파일시스템보다 약 23배 더 높은 성능을 보여준다.



(a) DRBL 결과
(a) DRBL results



(b) DWOL 결과
(b) DWOL results

그림 4. 마이크로 벤치마크 결과
Fig. 4. Microbenchmark results

그림 5는 매크로 벤치마크인 FileBench의 OLTP에서의 성능 결과를 보여준다. OLTP 워크로드는 파일을 자주 접근하는 특징을 가지고 있으며 전체적으로 마이크로 벤치마크 경우와 유사한 결과를 보인다. 64개의 프로세스의 경우에 제안된 기법의 성능은 약 140Mops/s이며, 기존 오버레이 파일시스템(Vanilla)의 성능은 약 10Mops/s 정도이며 제안된 기법이 약 14배 정도 더 좋은 성능을 보여주고 호스트 시스템 대비 88% 정도의 성능을 보여준다.

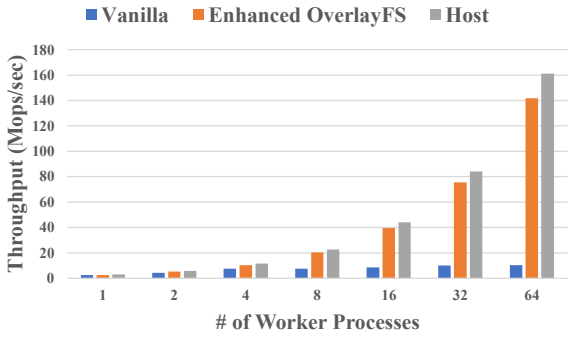


그림 5. 매크로 벤치마크 결과(FileBench, OLTP)
Fig. 5. Macrobenchmark results(FileBench, OLTP)

V. 결 론

오버레이 파일시스템 상의 컨테이너 환경에서 사용자 프로세스들이 파일 연산을 수행할 때 마운터의 자격 증명에 대한 참조 카운팅 때문에 성능 상의 문제가 발생한다. 이러한 성능 상의 문제를 해결하기 위해 본 논문은 마운터의 자격 증명을 프로세스 내부의 메모리 공간에 복사하여 참조 카운팅 횟수를 크게 줄이는 방법을 제안하였다. 제안한 방법을 리눅스 운영체제 상에 구현하였다. 마이크로/매크로 벤치마크를 이용하여 성능을 평가하였고, 평가 결과는 제안한 방법이 오버레이 파일시스템의 성능을 크게 개선할 수 있음을 보여주었다.

감사의 글

본 논문은 2022년도 동덕여자대학교 학술연구비 지원에 의하여 수행된 것입니다.

참고문헌

[1] D. Merkel, “Docker: Lightweight Linux Containers for Consistent Development and Deployment,” *Linux Journal*, Vol. 2014, No. 239, March 2014.

[2] A. Randazzo and I. Tinnirello, “Kata Containers: An Emerging Architecture for Enabling MEC Services in Fast and Secure Way,” in *Proceedings of the 6th International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, Granada, Spain, pp. 209-214, October 2019. https://doi.org/10.1109/IOTSMS48152.2019.8939164

[3] Y. Sun, J. Lei, S. Shin, and H. Lu, “Baoverlay: A Block-Accessible Overlay File System for Fast and Efficient Container Storage,” in *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC ’20)*, Online, pp. 90-104, October 2020. https://doi.org/10.1145/3419111.342

1291

[4] F. Guo, Y. Li, M. Lv, Y. Xu, and J. C. S. Lui, “HP-Mapper: A High Performance Storage Driver for Docker Containers,” in *Proceedings of the 10th ACM Symposium on Cloud Computing (SoCC ’19)*, Santa Cruz: CA, pp. 325-336, November 2019. https://doi.org/10.1145/3357223.3362718

[5] N. Mizusawa, J. Kon, Y. Seki, J. Tao, and S. Yamaguchi, “Performance Improvement of File Operations on OverlayFS for Containers,” in *Proceedings of IEEE International Conference on Smart Computing (SMARTCOMP)*, Taormina, Italy, pp. 297-302, June 2018. https://doi.org/10.1109/SMARTCOMP.2018.00019

[6] S. Wu, Z. Huang, P. Chen, H. Fan, S. Ibrahim, and H. Jin, “Container-Aware I/O Stack: Bridging the Gap between Container Storage Drivers and Solid State Devices,” in *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2022)*, Online, pp. 18-30, March 2022. https://doi.org/10.1145/3516807.3516818

[7] Q. Xu, M. Awasthi, K. T. Malladi, J. Bhimani, J. Yang, and M. Annavaram, “Docker Characterization on High Performance SSDs,” in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS),* Santa Rosa: CA, pp. 133-134, April 2017. https://doi.org/10.1109/ISPASS.2017.7975282

[8] J. Bhimani, Z. Yang, N. Mi, J. Yang, Q. Xu, M. Awasthi, ... and V. Balakrishnan, “Docker Container Scheduler for I/O Intensive Applications Running on NVMe SSDs,” *IEEE Transactions on Multi-Scale Computing Systems*, Vol. 4, No. 3, pp. 313-326, July-September 2018. https://doi.org/10.1109/TMSCS.2018.2801281

[9] B. Yan, H. Gao, H. Wu, W. Zhang, L. Hua, and T. Huang, “Hermes: Efficient Cache Management for Container-based Serverless Computing,” in *Proceedings of the 12th Asia-Pacific Symposium on Internetware (Internetware ’20)*, Singapore, pp. 136-145, November 2020. https://doi.org/10.1145/3457913.3457925

[10] Q. Xu, M. Awasthi, K. T. Malladi, J. Bhimani, J. Yang, and M. Annavaram, “Performance Analysis of Containerized Applications on Local and Remote Storage,” in *Proceedings of the 33rd International Conference on Massive Storage Systems and Technology (MSST 2017)*, Santa Clara: CA, pp. 24-28, May 2017.

[11] S. S. Bhat, R. Eqbal, A. T. Clements, M. F. Kaashoek, and N. Zeldovich, “Scaling a File System to Many Cores Using an Operation Log,” in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP ’17)*, Shanghai, China, pp. 69-86, October 2017. https://doi.org/10.1145/3132747.3132779

[12] 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 (SYSTOR ’19)*, Haifa, Israel, pp. 21-32, June 2019. https://doi.org/10.1145/3319647.3325828

[13] 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)*, Hangzhou, China, pp. 1-7, August 2019. https://doi.org/10.1145/3343737.3343748

[14] 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)*, Denver: CO, pp. 71-85, June 2016.

[15] S. Boyd-Wickizer, Optimizing Communication Bottlenecks in Multiprocessor Operating System Kernels, Ph.D. Dissertation, Massachusetts Institute of Technology, Cambridge, MA, February 2014.

[16] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, “An Analysis of Linux Scalability to Many Cores,” in *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI ’10)*, Vancouver, Canada, pp. 1-16, October 2010.

[17] Y. Son, S. Kim, H. Y. Yeom, and H. Han, “High-Performance Transaction Processing in Journaling File Systems,” in *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST ’18)*, Oakland: CA, pp. 227-240, February 2018.

한혁(Hyuck Han)

2006년 : 서울대학교 대학원 (공학석사)
2011년 : 서울대학교 대학원 (공학박사 -분산시스템)
1999년～2003년: 델타정보통신
2011년～2012년: 서울대학교
2012년～2014년: 삼성전자
2014년～현 재: 동덕여자대학교 컴퓨터학과 조교수/부교수
※관심분야 : 분산시스템 (Distributed System), 운영체제 (Operating System), 데이터베이스시스템 (DBMS) 등