Vol.6, No.10 pp.415~420 pISSN: 2287-5891 https://doi.org/10.3745/KTCCS.2017.6.10.415

A Method of Selecting Layered File System Based on Learning Block I/O History for Service-Customized Container

Chanho Yong† · Sang-Ho Na†† · Pill-Woo Lee††† · Eui-Nam Huh††††

ABSTRACT

Virtualization technique of OS-level is a new paradigm for deploying applications, and is attracting attention as a technology to replace traditional virtualization technique, VM (Virtual Machine). Especially, docker containers are capable of distributing application images faster and more efficient than before by applying layered image structures and union mount point to existing linux container. These characteristics of containers can only be used in layered file systems that support snapshot functionality, so it is required to select appropriate layered file systems according to the characteristics of the containerized application. We examine the characteristics of representative layered file systems and conduct write performance evaluations of each layered file systems according to the operating principles of the layered file system, Allocate-on-Demand and Copy-up. We also suggest the method of determining a appropriate layered file system principle for unknown containerized application by learning block I/O usage history of each layered file system principles in artificial neural network. Finally we validate effectiveness of artificial neural network created from block I/O history of each layered file system principles.

Keywords : Docker, Container, Layered File System, Docker Image, Machine Learning

서비스 맞춤형 컨테이너를 위한 블록 입출력 히스토리 학습 기반 컨테이너 레이어 파일 시스템 선정 기법

용 찬 호† · 나 상 호†† · 이 필 우††† · 허 의 남††††

요 약

OS-level의 가상화 기술은 애플리케이션을 배포하기 위한 새로운 패러다임으로, 기존의 가상화 기술인 가상 머신을 대체할 수 있는 기술로서 주목받고 있다. 특히 컨테이너는 기존의 리눅스 컨테이너에 유니온 마운트 포인트(Union Mount Point) 와 레이어 구조의 이미지를 적용함으로써 보다 빠르고 효율적인 애플리케이션의 배포가 가능하다. 이러한 컨테이너의 특징들은 스냅숏 기능을 제공하는 레이어 구조의 파일 시스템에서만 사용될 수 있으며, 애플리케이션의 특징에 따라 적절한 레이어 파일 시스템을 선택하는 것이 요구된다. 따라서 본 논문에서는 대표적인 레이어 파일 시스템들의 특징을 조사한 뒤, 레이어 파일 시스템의 동작 원리인 Allocate-on-Demand 및 Copy-up 방식에 따른 파일 시스템의 쓰기 성능 평가를 수행한다. 또한 각 레이어 파일 시스템 방식의 블록 입출력 사용 데이터를 학습한 인공 신경망을 통해 임의의 애플리케이션에 대해 적합한 레이어 파일 시스템 방식을 결정하는 방법을 제시하고 이에 대한 타당성을 검토한다.

키워드 : 도커, 컨테이너, 레이어 파일 시스템, 도커 이미지, 머신 러닝

1. 서 론

컨테이너 기술은 애플리케이션을 개발하고 배포하기 위한 새로운 관점을 제시하는 리눅스 컨테이너 기반의 가상화 기술로서, 기존의 가상화 기술인 하이퍼바이저 기반의 가상 머신에 비해 성능의 손실이 거의 존재하지 않아 이를 대체할 수 있는 가상화 기술로서 주목받고 있다[3]. 도커 컨테이너를 배포하기 위한 형식인 이미지는 운영체제를 구성하기 위한 커널을 포함하지 않기 때문에 가상 머신에 비해 이미지의 크기가 매우 작을 뿐만 아니라, 이미지를 레이어 구조로 배포함으로써 중복되는 레이어를 전송하지 않아 보다 효율적인 애플리케이션의 배포가 가능하다는 장점을 가진다.

이미지의 레이어 구조를 구성하기 위해 사용되는 핵심 기술인 스냅숏(Snapshot)은 RoW 또는 CoW를 지원하는 파일 시스템을 구성해야만 사용할 수 있으며, 도커 컨테이너가 지

※ 본 연구는 과학기술정보통신부 및 정보통신기술진흥센터의 대학ICT연구센터육성지원사업의 연구결과로 수행되었음(IITP-2017-2013-0-00717).
※ 본 연구는 한국과학기술정보연구원(KISTI)의 위탁연구 과제로 수행되었음.
† 준 회 원 : 경희대학교 컴퓨터공학과 석사과정
†† 비 회 원 : 한국과학기술정보연구원슈퍼컴퓨팅본부 선임연구원
††† 정 회 원 : 한국과학기술정보연구원슈퍼컴퓨팅본부 본부장
†††† 종신회원 : 경희대학교 컴퓨터공학과 교수
Manuscript Received : July 17, 2017
Accepted : August 19, 2017
* Corresponding Author : Eui-Nam Huh(johnhuh@khu.ac.kr)

원하는 대표적인 레이어 파일시스템으로서 AUFS(Advanced multi layered Unification Filesystem), Devicemapper, ZFS, Overlay 및 Overlay2 등이 있다[11]. 이러한 파일 시스템은 호스트의 운영체제, 커널 버전, 이미지 레이어 개수, 컨테이너의 읽기 및 쓰기 빈도뿐만 아니라 레이어 파일 시스템의 동작 방식인 Allocate-on-Demand, Copy-up 중 어느 방식을 사용하느냐에 따라 쓰기 성능이 상이하다. 따라서 애플리케이션의 개발 및 배포 시나리오, 서비스 환경과 같은 특징을 고려해 적합한 레이어 파일 시스템을 선택하는 것이 요구된다. 본 논문은 컨테이너가 사용할 수 있는 대표적인 레이어 파일 시스템들의 종류를 조사하고 성능평가를 진행한 뒤, 컨테이너의 블록 입출력 히스토리 데이터를 학습한 신경망을 기반으로 Allocate-on-Demand와 Copy-up 중 애플리케이션에 적합한 레이어 파일 시스템 방식을 도출하는 방법을 제시한다.

본 논문의 2장에서는 컨테이너의 이미지 구조에 대한 간략한 설명과 유니온 마운트 포인트를 사용하기 위한 스냅숏 기술들, 컨테이너에서 사용할 수 있는 대표적인 레이어 파일 시스템의 종류와 특징을 설명한다. 3장에서 각 파일 시스템의 컨테이너 실행 시간 및 쓰기 작업의 성능 평가를 진행하고 4장에서 블록 입출력 로그 데이터를 통한 신경망 학습 과정과 Allocate-on-Demand, Copy-up 방식의 결정 방법 및 인공신경망 모델의 타당성 검토 결과를 서술한다. 5장에서는 결론 및 향후 계획으로 논문을 마친다.

2. 관련 연구

2.1 도커 컨테이너와 이미지

도커 컨테이너는 리눅스의 자체 기능인 cgroup, namespace, chroot를 이용해 프로세스 단위의 가상화 공간을 제공하는 가상화 기술이다[2]. 도커 컨테이너를 구동하기 위한 도커 이미지는 컨테이너의 변경 사항을 저장하는 레이어 구조로 이루어져 있으며, 어떠한 경우라도 읽기 전용으로 사용됨으로써 불변(Immutable infrastructure)의 상태를 유지한다. Fig. 1은 읽기 전용의 이미지 레이어와 컨테이너 레이어, 그리고 이를 사용하기 위한 마운트 포인트의 구조를 나타낸다.

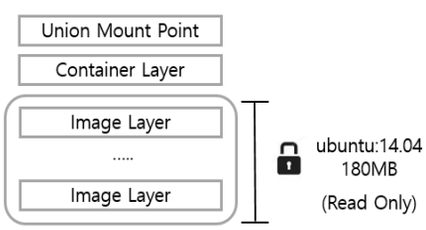


Fig. 1. Structure of Image and Container

컨테이너는 여러 개의 이미지 레이어와 컨테이너 레이어를 마운트한 유니온 마운트 포인트로부터 이미지의 파일들을 사용할 수 있으며[15], 컨테이너의 변경 사항은 이미지가 아닌 컨테이너 레이어에 저장된다.

RoW와 CoW는 파일 시스템의 스냅숏을 구성하기 위해 사용되는 기술이다. 두 방식 모두 파일들을 스냅숏으로 생성한 뒤 쓰기 작업으로 인해 파일에 변경 사항이 발생할 경우 원본 스냅숏 파일을 유지하고 변경된 사항을 스냅숏 풀(Snapshot Pool)에 저장하는 방식으로 동작한다. CoW는 스냅숏 파일에 쓰기 작업을 수행할 때 스냅숏 풀에 원본 파일을 복사하고 쓰기 요청을 반영하기 때문에 파일을 읽는 작업 1번과 파일을 스냅숏 풀에 복사하고 변경된 사항을 쓰는 작업을 포함해 총 2번의 쓰기 작업이 발생한다. 반면 RoW는 원본 파일을 묶은(Freeze) 뒤, 스냅숏 풀에서 블록을 할당 받아 변경점을 저장하기 때문에 1번의 쓰기 작업만이 발생한다[4]. Fig. 2는 RoW의 작동 원리를 도식화한 것으로, 스냅숏 파일의 변경점을 스냅숏 풀에 할당하는 것을 나타낸다.

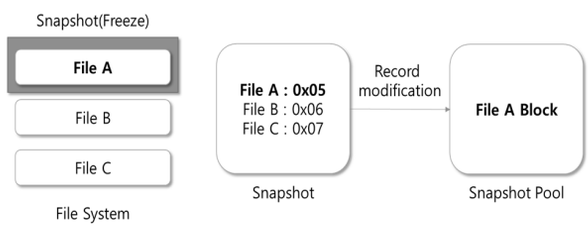


Fig. 2. Operating Principle of RoW

2.2 레이어 파일 시스템

도커 엔진이 사용하려는 파일 시스템은 도커 이미지를 사용하는 컨테이너 레이어를 구성하기 위해 기본적으로 RoW 또는 CoW[14, 15]를 지원해야 한다[1]. 도커 이미지를 스냅숏으로 사용하고 이를 불변의 상태로 유지하며, 이미지 레이어의 파일에서 변경되는 사항들은 RoW 또는 CoW의 원리에 의해 컨테이너 레이어에 별도로 저장된다. Table 1은 본 논문에서 설명하는 각 파일 시스템들의 특징들을 나타낸다.

컨테이너가 이미지 내부의 읽기 전용 파일을 수정할 때 동작하는 방식에 따라서 크게 Allocate-on-Demand와 Copy-up으로 구분된다[1]. Copy-up은 쓰기 작업으로 인해 변경된 이미지의 파일을 컨테이너 레이어로 전부 복사해 사용하는 반면, Allocate-on-Demand는 특정 크기의 블록을 할당함으로써 변경된 사항만을 저장한 뒤 블록을 포함해 하나의 파일로 인식해 사용한다. AUFS, Overlay는 Copy-up 방식을, Devicemapper와 ZFS는 Allocate-on-Demand 방식을 사용함으로써 컨테이너의 레이어 구조를 구성한다.

AUFS는 컨테이너가 이미지의 파일에 쓰기 작업을 수행하면 가장 상층에 위치한 레이어부터 이미지를 찾아 해당 파일을 컨테이너 레이어로 복사하는 Copy-up 작업을 수행한다. 따라서 이미지의 파일을 처음 변경할 때는 오버헤드가 존재할 수 있지만, 컨테이너 환경에서 비교적 오랜 기간 사용돼온 파일 시스템이기 때문에 안정성 및 컨테이너의 할당 속도 측면에서 PaaS에 적합한 파일 시스템으로 평가받는다[5]. AUFS는 데비안(Debian) 계열의 리눅스에서 도커 엔진을 사용하면 자동으로 사용하도록 설정되는 파일 시스

Table 1. Comparison of File Systems Supporting Layered Architecture

	AUFS	Devicemapper	Overlay, Overlay2	ZFS
Writing operation principle	Copy-up	Allocate-on-Demand	Copy-up	Allocate-on-Demand
Dependency of external modules	Yes	No	No	Yes
OS using as defualt file system	Ubuntu	-	CentOS, RHEL, Raspbian	-

템으로서, 리눅스 커널 모듈에 포함되어 있지 않기 때문에 레드햇 계열의 리눅스에서는 기본적으로 사용할 수 없다.

Devicemapper는 스토리지 풀(Storage Pool)로부터 Allocate-on-Demand 원리에 의해 이미지 레이어의 변경 사항을 블록 단위로 할당 받아 컨테이너의 레이어를 구성한다[5]. Fig. 3과 같이 Sparse 파일인 data 파일이 도커에 필요한 파일들을 저장하고 있으며, 기본적으로 100G 크기의 스토리지 풀에서 공간을 할당한다[12]. Devicemapper는 대부분의 리눅스 커널에서 사용할 수 있다는 장점은 있지만, 다른 파일 시스템과 달리 컨테이너와 이미지 파일을 분리된 디렉터리에 저장하지 않아 레이어 단위로 멀티 테넌시 환경을 관리할 수 없으며, 파일 시스템을 블록 디바이스로 사용하는 루프백(Loopback) 디바이스를 사용할 경우 일부 성능의 저하가 발생한다는 단점이 있다[6, 11]. Devicemapper는 이전 도커 릴리스에서 레드햇 계열의 리눅스를 위해 사용되었던 파일 시스템이며, 리눅스 커널에 기본적으로 탑재되어 있어 대부분의 리눅스 운영 체제에서 사용할 수 있다.

```
# ls -lash /DATA/devicemapper/devicemapper
total 2.6G
2.6G -rw------- 1 root root 100G Jun 29 02:55 data
2.7M -rw------- 1 root root 2.0G Jun 29 02:54 metadata
```

Fig. 3. Sparse File of Devicemapper

Overlay는 AUFS와 매우 유사한 구조를 가지고 있지만 수직적인 이미지 레이어 구조가 아니라는 점에서 좀 더 빠른 성능을 갖는다. AUFS와 동일하게 컨테이너가 변경하려는 이미지의 파일을 컨테이너 레이어로 복사해 사용하는 Copy-up 작업을 수행하지만, 이미지 레이어를 계층화된 구조로 사용하지 않기 때문에 파일을 찾는 과정에서 비교적 적은 오버헤드가 발생한다[11][1]. 레드햇 계열의 운영체제에 설치되는 도커 엔진은 Overlay를 자동으로 사용하도록 설정되며, 라즈비안(Raspbian)의 도커 엔진은 Overlay의 성능을 개선한 파일 시스템인 Overlay2를[1] 자동으로 사용하도록 설정된다.

ZFS[13]는 썬 마이크로시스템즈(Sun Microsystems)에서 개발한 128비트 파일 시스템으로, 리눅스 커널에 포함되어 있지 않아 별도의 설치가 필요하다. ZFS는 이미지의 파일에 쓰기 작업을 수행할 때 zpool에서 Allocate-on-Demand 원리에 의해 128kb 크기의 블록을 할당해 저장하며, 컨테이너와 이미지를 구성하기 위해 클론(Clone)과 스냅숏이라는 개념을 사용한다[7]. ZFS는 ARC(Adaptive Replacement Cache), 압축 등 다양한 기능을 제공해 반복되는 읽기 및 쓰기 작업에 대해 우수한 성능을 가지지만, 여러 개의 컨테이너를 동시에 사용할 경우 높은 메모리 점유율을 나타낼 수 있다는 단점이 있다.

2.3 인공 신경망

인공 신경망(ANN:Artificial Neural Network)은 생물의 신경망을 모사해 설계된 학습 알고리즘이다. 관측된 데이터를 신경망에 학습시키기 위해 역전파(Backpropagation)를 이용해 신경망의 가중치들을 최적화하며, 목적에 따라 지도학습, 비지도학습 등의 방법을 선택해 원하는 데이터를 예측하거나 분류할 수 있다[9]. 인공 신경망은 학습 데이터를 입력받는 입력 층(Input Layer), 데이터를 학습하고 데이터를 처리하는 은닉 층(Hidden Layer), 학습 및 데이터의 처리 결과를 출력하는 출력 층(Hidden Layer)으로 구성된다. 일반적으로 인공 신경망은 데이터를 입력 층에서 입력받아 은닉 층의 가중치를 통해 계산된 결과값을 출력 층으로 내보낸 뒤, 출력값을 통해 역전파를 수행함으로써 학습을 수행한다. 인공 신경망을 사용할 수 있는 대표적인 소프트웨어로서 구글이 오픈 소스로 공개한 텐서플로우가 있으며, 인공 신경망 학습을 비롯한 데이터 처리 및 분석 기능을 제공한다[8].

3. 파일 시스템 성능 평가

본 논문의 성능평가 지표는 (1) 서비스의 실시간성을 보장하기 위한 컨테이너 생성 시간, (2) 서비스의 지연을 최소화하기 위한 파일 입출력 수행 시간으로 설정한다. 이에 따라 각 파일 시스템의 성능을 평가하기 위해 (1) 컨테이너 생성 및 시작 시간, (2) 단일 파일에 대한 쓰기 및 읽기 시간, (3) 분할된 파일들에 대한 쓰기 및 읽기 시간을 측정한다. 이는 AWS(Amazon Web Service)의 t2.medium 인스턴스에서 SSD 저장 장치를 사용해 테스트했으며, Ubuntu 16.04 운영체제의 도커 엔진 버전 17.06.0-ce, build 02c1d87에서 수행되었다. Devicemapper 파일 시스템은 루프백 디바이스를 사용해 성능을 측정하였다.

3.1 컨테이너 생성 및 시작 시간

컨테이너를 연속으로 10개씩 10번 생성하고 시작하는 시간을 측정함으로써 컨테이너 1개에 소요되는 평균 시간을 계산하였으며, Fig. 4는 이에 대한 결과를 나타낸다. 컨테이너를 생성할 때 사용된 명령어는 다음과 같다.

```
# docker run -itd busybox
```

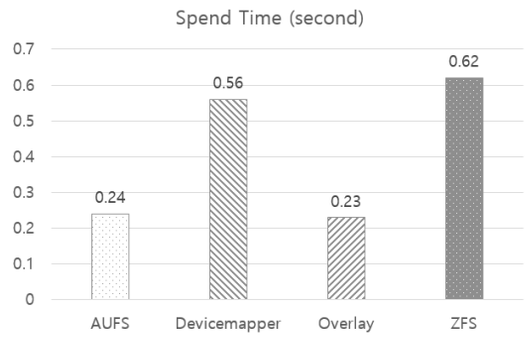


Fig. 4. Average Time of Creating and Starting 10 Containers

스냅숏의 구조가 유사하고, Copy-up 방식을 사용하는 AUFS와 Overlay는 거의 동일한 수행 결과를 얻었지만, Devicemapper와 ZFS 상대적으로 느린 수행 결과를 보여주었다. 그러나 ZFS는 각 수행 결과와 평균의 오차가 적은 반면, Devicemapper는 수행 결과가 최소 0.5초에서 최대 0.9초에 달하는, 일률적이지 않은 결과를 얻었다.

3.2 단일 파일에 대한 쓰기 및 읽기 시간

각 파일 시스템에서 이미지 내부에 존재하는 4G 크기의 파일에 쓰기 작업을 첫 번째로 수행한 시간을 측정하였으며, 동일한 작업을 새로운 컨테이너에서 10회 수행해 평균 시간을 계산하였다. Fig. 5는 이에 대한 소요 시간을 나타낸다.

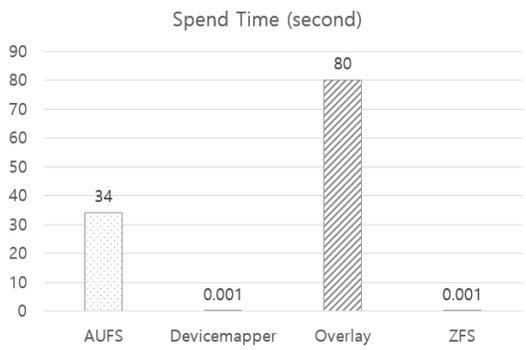


Fig. 5. Time of Conducting First Writing Operation to 4G Size File

AUFS와 Overlay는 이미지 레이어에 존재하는 읽기 전용 파일에 처음 쓰기 작업을 수행할 때 컨테이너 레이어로 파일을 복사하기 때문에 새로운 파일을 생성하는 시간이 각각 34초, 80초가 소요된 반면, Devicemapper와 ZFS는 블록 단위로 변경점을 저장해 쓰기 작업을 수행하기 때문에 시간이 거의 소요되지 않았다.

3.3 분할된 파일들에 대한 쓰기 및 읽기 시간

각 파일 시스템에서 이미지 내부에 존재하는 10개의 128MB 크기 파일에 첫 번째로 쓰기 작업을 수행하는 시간을 측정했으며, 동일한 작업을 새로운 컨테이너에서 10회 수행해 평균 시간을 계산하였다. Fig. 6은 이에 대한 소요 시간을 나타낸다.

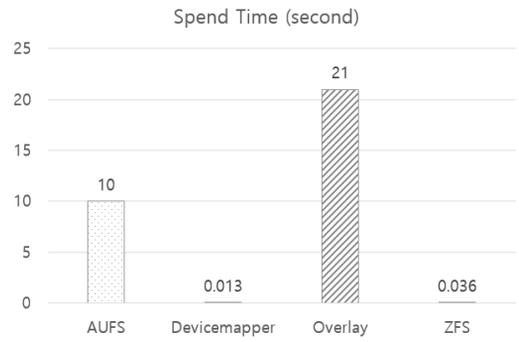


Fig. 6. Time of Conducting First Writing Operation to 128MB Size, 10 Files

Devicemapper와 ZFS는 4.2의 성능 평가에 비해 쓰기 작업의 소요 시간이 소폭 증가했으나, 이에 관련된 서비스의 실시간성을 보장하기에는 부족하지 않을 것으로 평가된다. 그러나 쓰기 작업을 수행하는 파일의 수와 크기가 늘어날수록 AUFS와 Overlay는 필연적으로 성능의 저하를 수반할 것으로 예상된다.

4. 파일 시스템 결정 기법

위의 성능 평가로부터 Copy-up 방식을 사용하는 AUFS, Overlay 파일 시스템은 많은 수의 컨테이너를 빠르게 생성하고 삭제해야 하는 웹 서비스 등의 환경에 적합하지만, 파일의 크기가 큰 .iso 확장자 또는 빅데이터 분석을 위한 덤프 파일 등이 이미지에 존재할 경우 쓰기 작업 수행에 있어 실시간성을 보장할 수 없다는 단점을 가진다는 것을 알 수 있다. 이러한 관점에서는 블록 단위로 이미지 레이어의 변경점을 저장하는 Allocate-on-Demand 방식의 Devicemapper, ZFS와 같은 파일 시스템이 이점을 가진다.

본 연구는 컨테이너의 블록 입출력 사용량의 관점에서 적절한 레이어 파일 시스템 방식을 선택하는 방법을 제시한다. Copy-up 방식의 레이어 파일 시스템에서 Allocate-on-Demand 방식에 적합한 애플리케이션을 구동할 경우, 파일을 컨테이너 레이어로 복사하는 작업으로 인해 일시적으로 높은 블록 입출력 패턴이 발생한다. 따라서 임의의 컨테이너 애플리케이션을 Copy-up 방식의 파일 시스템에서 구동한 뒤, 블록 입출력 로그 데이터를 수집해 패턴을 학습함으로써 Copy-up과 Allocate-on-Demand 방식의 레이어 파일 시스템으로 분류하는 인공 신경망을 구축한다.

4.1 블록 입출력 데이터 전처리

블록 입출력 데이터 수집 환경은 3의 성능 측정 환경과

동일하며, 데이터 학습을 위해 Tensorflow 1.0.0, Python 2.7.6을 사용하였다. 신경망을 학습시키기 위한 데이터는 (1) 각각 10개의 128MB, 32M, 8M, 1M 크기의 Chunk 파일을 이미지 내부에 가지고 있어 0.1초 간격으로 쓰기 작업을 수행한 경우, (2) 빈 파일에 0.1초 간격으로 쓰기 작업을 수행한 경우의 블록 입출력 데이터를 수집해 사용하였다. (1)은 Allocate-on-Demand로 결정하는 경우의, (2)는 Copy-up로 결정하는 경우의 학습 데이터로서 사용된다.

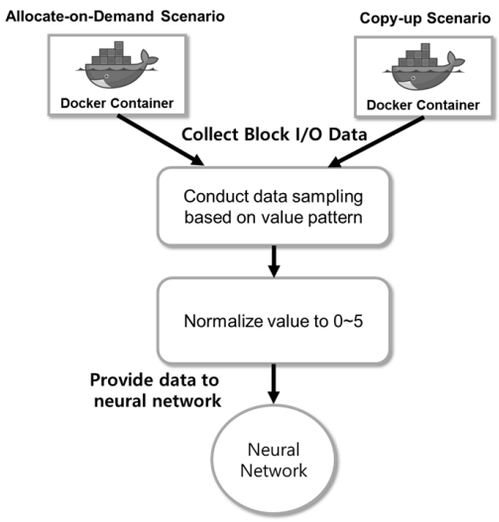


Fig. 7. Workflow of Preprocessing Block I/O Data Value to Provide Data to Neural Network

Fig. 7은 신경망 학습에 사용되는 블록 입출력 데이터의 전처리 과정을 나타낸다. 블록 입출력 데이터는 도커 엔진의 Remote API 중 /stats를 사용해 수집했으며, 1초간의 블록 입출력량을 계산하기 위해 1초 간격으로 API를 호출한 뒤 이전 데이터 값과 다음 데이터 값의 차를 구해 실제 입출력 값으로 사용한다. 수집된 데이터 중 일정 범위 이상의 변화가 발생하는 구간의 값만이 학습 데이터로서 의미를 가지기 때문에 임계값 이상의 변화량을 보이는 구간에 대해 샘플링 작업을 수행한다. 또한 Remote API로 수집된 입출력 값은 Byte 단위로서 매우 큰 값을 가지므로, sklearn.processing 패키지의 normalize 함수[10]를 통해 0에서 5 사이의 값을 가지도록 데이터 정규화를 수행한다.

4.2 데이터 학습 및 수행 결과

전처리를 마친 데이터는 선형 회귀 모델을 사용하는 신경망에서 학습 입력 데이터로서 사용되며, 앞서 언급한 각 시나리오에서 출력된 블록 입출력 데이터를 각각 사용해 신경망을 학습시킨다. 학습된 신경망에 임의의 애플리케이션에 대한 블록 입출력 데이터를 입력한 뒤, 인공 신경망의 최종 레이어에서 Softmax 함수[8]를 사용해 Allocate-on-Demand와 Copy-up 방식에 대한 적합도를 결과값으로 출력한다. 학습한 신경망의 적정성을 검토하기 위해 (1) 각 10개의 1MB, 8MB, 32MB 크기의 Chunk 파일에 0.1초 간격으로 쓰기 작업을 수행하는 경우, (2) 이미지 내부에 Chunk 파일이 존재하지 않는 MySQL 컨테이너에 데이터베이스의 Insert 쿼리를 수행하는 경우의 데이터를 신경망에 사용하였다. Fig. 8과 Table 5는 (1)과 (2)에 따른 결과값을 표현한 것으로, MySQL 컨테이너의 데이터는 Copy-up의 적합도가 0.89로 매우 높게 나타났으며 1MB 크기의 Chunk 파일에 쓰기 작업을 수행한 데이터는 Allocate-on-Demand의 적합도가 0.67로 나타났다. 그러나 8MB, 32MB 크기의 Chunk 파일에 쓰기 작업을 수행한 경우는 Allocate-on-Demand의 적합도 수치가 0.97, 0.99로 매우 높게 나타났다. 따라서 이미지 내부에 존재하는 파일의 크기가 클수록 Allocate-on-Demand로 분류할 확률이 더 큰 것을 알 수 있다.

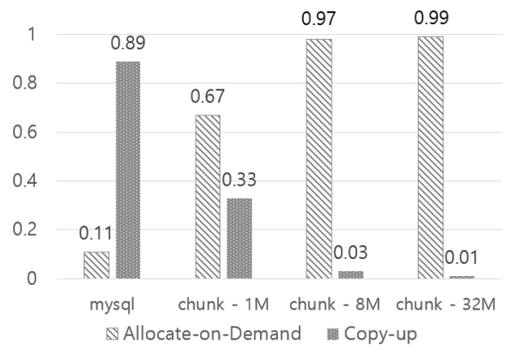


Fig. 8. Probability of Categorizing Layered File System Principles According to Writing Operation Scenario in Container (1 means 100%)

5. 결론 및 향후 계획

본 연구는 컨테이너에 적용할 수 있는 파일 시스템들의 특징을 조사하고, 레이어 파일 시스템의 특징 중 Allocate-on-Demand 및 Copy-up 에 초점을 맞춰 성능 평가를 진행하였다. Copy-up 방식을 사용하는 AUFS, Overlay 파일 시스템은 많은 수의 컨테이너를 빠르게 생성하고 삭제해야 하는 웹 서비스 등의 환경에 적합할 수 있지만, 크기가 큰 파일이 이미지 레이어의 내부에 존재할 경우 쓰기 작업 수행에 있어 실시간성을 보장할 수 없다는 단점이 있다. 이러한 관점에서는 블록 단위로 이미지 레이어의 변경점을 저장하는 Allocate-on-Demand 방식의 Devicemapper, ZFS와 같은 파일 시스템이 이점을 가진다. 본 연구는 이러한 두 특징 중 쓰기 작업으로 인한 블록 입출력 히스토리 데이터의 관점에서 적합한 레이어 파일 시스템을 선택하는 방법을 제시하였다. 각 레이어 파일 시스템 방식을 사용한 컨테이너의 블록 입출력 데이터를 신경망에 학습시켰으며, 이미지 내부의 파일 크기가 커짐에 따라 Allocate-on-Demand 방식으로 분류할 확률이 높아지는 것을 확인했다. 따라서 본 연구의 레이어 파일 시스템 방식 분류 기법을 통해 컨테이너 애플리케이션의 불필요한 블록 입출력을 크게 감소시킬 수 있을 것으로 예상된다.

그러나 본 연구의 레이어 파일 시스템 결정 기법은 블록 입출력 데이터만을 신경망에 학습시켰다는 한계점을 갖는다. 따라서 추후 연구에서 애플리케이션의 Scale-in/out 빈도 또는 필요성뿐만 아니라 쓰기, 읽기 작업에 대한 SSD 최적화, 캐싱, 데이터 압축, 블록 입출력의 QoS 등을 척도로 하는 신경망 학습 모델을 설계해 향상된 결정 기법을 연구한다.

References

[1] Docker Documentation [Internet], https://docs.docker.com/ 2017.06.29.

[2] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux Journal*, Vol.2014, Issue 239, Article No.2, 2014.

[3] Felter, Wes, et al., "An updated performance comparison of virtual machines and linux containers," *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*. IEEE, 2015.

[4] Xiao, Weijun, et al., "Design and analysis of block-level snapshots for data protection and recovery," *IEEE Transactions on Computers*, Vol.58, Issue 12, pp.1615-1625. 2009.

[5] P. Bellavista and Z. Alessandro, "Feasibility of Fog Computing Deployment based on Docker Containerization over RaspberryPi," *Proceedings of the 18th International Conference on Distributed Computing and Networking*, ACM, 2017.

[6] T. Inagaki, Y. Ueda, and M. Ohara, "Container management as emerging workload for operating systems," *Workload Characterization (IISWC), 2016 IEEE International Symposium on*. IEEE, 2016.

[7] Dawidek Pawel Jakub and Marshall Kirk McKusick. "Porting the Solaris ZFS file system to the FreeBSD operating system," *;login:: The Magazine of USENIX & SAGE*, Vol.32, No.3, pp.19-24, 2007.

[8] Abadi, Martín, et al., "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[9] 박중오 and 최도현, "클라우드 서비스 기술동향: 구글 머신러닝을 중심으로," 정보처리학회지, Vol.23, No.2, pp.4-12, 2016.

[10] Scikit-learn Documentation [Internet], http://scikit-learn.org /stable/modules/generated/sklearn.preprocessing.normalize. html, 2017. 07. 01, version 0.18.2.

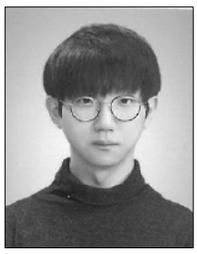
[11] Xu, Qiumin, et al., "Performance analysis of containerized applications on local and remote storage," *Proc. of MSST*, 2017.

[12] J. Eder, "Comprehensive Overview of Storage Scalability in Docker," https://developers.redhat.com/blog/2014/09/30/over view-storage-scalability-docker/.

[13] Bonwick, Jeff, et al., "The zettabyte file system," *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, Vol.215, 2003.

[14] X. Wu, W. Wang, and S. Jiang, "TotalCOW: Unleash the power of copy-on-write for thin-provisioned containers," *Proceedings of the 6th Asia-Pacific Workshop on Systems*, ACM, 2015.

[15] Dua, Rajdeep, et al., "Performance analysis of Union and CoW File Systems with Docker," *Computing, Analytics and Security Trends (CAST), International Conference on*. IEEE, 2016.



용 찬 호

e-mail : alice_k106@khu.ac.kr

2017년 경희대학교 컴퓨터공학과(학사)

현 재 경희대학교 컴퓨터공학과 석사과정

관심분야 : 클라우드 컴퓨팅, 고성능 컴퓨팅(HPC), 분산 컴퓨팅

나 상 호

e-mail : shna@kisti.re.kr

2010년 경희대학교 컴퓨터공학(석사)

2014년 경희대학교 컴퓨터공학(박사)

2014년~2016년 경희대학교 박사후연구원

2016년~현 재 한국과학기술정보연구원 슈퍼컴퓨팅본부 선임연구원

관심분야 : 클라우드 컴퓨팅, 고성능 컴퓨팅(HPC), IoT, 보안

이 필 우

e-mail : pwlee@kisti.re.kr

1998년 Univ. of Tsukuba 컴퓨터공학전공 (공학박사)

2000년 한국과학기술정보연구원 슈퍼컴퓨팅센터

2012년 한국과학기술정보연구원 정책연구실장

2015년~현 재 한국과학기술정보연구원 슈퍼컴퓨팅본부 본부장

관심분야 : 컴퓨터 아키텍처, 프로그래밍 언어, 분산컴퓨팅, 클라우드컴퓨팅

허 의 남

e-mail : johnhuh@khu.ac.kr

2002년 The Ohio University 전산학(박사)

2005년~현 재 경희대학교 컴퓨터공학과 교수, 실시간 모바일 클라우드 연구센터장, TTA STC2 의장, ISO/IEC JTC1 SC38 전문위원

관심분야 : 클라우드 컴퓨팅, 슈퍼컴퓨팅, 모바일 컴퓨팅, 빅데이터, IoT, 보안, WSN