서버리스 환경에서 멀티스테이지 빌드 기반의 Dockerfile 최적화에 따른 컨테이너 이미지 풀링 시간 분석

김민창, 강민재, 유헌창
고려대학교 정보대학 컴퓨터학과

{naxcco, austin9228, yuhc}@korea.ac.kr

An Analysis of Image Pulling Time Based on Dockerfile Optimization Using Multi-Stage Builds in Serverless Computing

Min Chang Kim, Minjae Kang, Heonchang Yu
Dept. of Computer Science and Engineering, Korea University

요 약

서버리스 컴퓨팅은 클라우드 컴퓨팅 패러다임의 한 종류로, 마이크로서비스가 부상하며 함께 큰 주목을 받고 있다. 서버리스 컴퓨팅에서 가장 큰 문제점 중 하나는 콜드 스타트 문제를 해결하는 것으로, 함수를 실행할 컨테이너가 생성될 때 이미지를 풀링하는 데 걸리는 지연 시간은 콜드 스타트에 큰 비중을 차지한다. 본 논문은 컨테이너의 이미지 사이즈를 최적화하기 위해 Docker 멀티 스테이지 기법을 사용하였으며, 컨테이너 이미지 풀링 시간에 미치는 영향을 분석하였다. 이를 통해 컨테이너 이미지 크기가 풀링 시간에 미치는 영향을 알아보고, 멀티 스테이지 빌드 기법의 유용성을 검증한다.

1. 서론

클라우드 컴퓨팅이 많은 기업과 개발자에게 현실적인 대안으로 부상함에 따라, 서버리스 컴퓨팅 또한 비용 효율성 및 개발 용이성으로 주목을 받고 있다. 서버리스 컴퓨팅의 대표적인 구현 방식인 Function as a Service (FaaS) 모델은 특정 작업을 수행하는 함수를 작성하고, 여러 함수의 조합을 통해 애플리케이션을 구현한다. FaaS 모델은 이벤트 기반 방식으로, 코드를 실행하는 컨테이너는 사용자의 수요 급증에 맞춰 빠르게 생성되고 삭제되어야 한다. 웹 애플리케이션의 경우, 사용자 요청이 급증하는 시간대에 컨테이너를 신속하게 생성할 수 있어야 한다.

기존의 많은 연구는 서버리스 컴퓨팅의 요구사항을 해결하기 위하여 콜드 스타트를 줄이는 데 중점을 두고 진행되었다[1]. 그러나 컨테이너를 빌드하는데 소요되는 시간을 줄이기 위해 컨테이너 이미지 크기를 줄이는 방안에 대한 연구는 미비한 상태다. FaaS 워크로드는 불균형한 호출 패턴을 보이기 때문에, 요청이 집중되는 기간에 이미지 풀링에 사용되는 트래픽이 콜드 스타트를 유발하는 병목이 될 수 있다[2].

또한, 서버리스 함수를 실행하기 위해 사용자가 수정한 커스텀 컨테이너 이미지는 기반이 되는 이미지보다 크기가 큰 경향이 있다. 실제 Docker Hub 에 있는 컨테이너 중 10% 이상이 1.3GB 보다 크게 나타났다[3]. 크기가 큰 이미지를 캐싱하는 것은 클라우드 공급자의 비용(TCO)을 증가시킬 수 있기 때문에 공급자는 일부 필수적인 함수만 미리 준비(pre-warm)할 수 있고, 이는 콜드스타트 빈도의 증가로 이어진다.

본 논문은 Docker 의 멀티 스테이지 빌드(multi-stage build)를 활용하여 이미지 크기를 줄였을 때 발생하는 성능상의 이점을 분석하는 데 초점을 맞추었다. 실험은 AQUATOPE 에서 사용한 서버리스 벤치마크를 기반으로 하였으며, 벤치마크의 각 모듈에 사용되는 Dockerfile 은 멀티 스테이지 빌드 설계를 적용하여 수정하였다. 그 결과, 개별 컨테이너 이미지 및 전체 데이터 크기에 대해 평균 61% 차이가 발생하였으며, 이미지 풀링(pulling)에 소요되는 시간이 최대 28% 감소하였다.

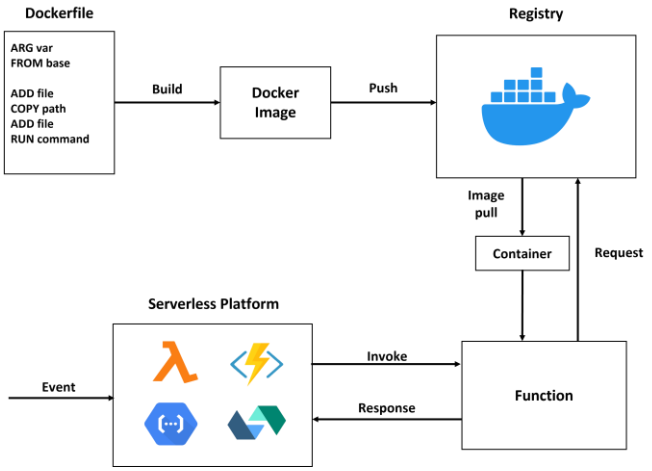
2. 배경

2.1. 서버리스 컴퓨팅

Function-as-a-service 는 함수 호출에 따라 필요한 컴퓨팅 자원이 실행되는 클라우드 서비스 모델이다[4]. FaaS 는 함수가 실행 되는 시간에만 과금되고, 서버 관리를 수행할 필요가 없으므로 실행 비용이 절감되고 개발이 용이한 장점을 지닌다. 서버리스 컴퓨팅 서비스 제공자는 일반적으로 Python 이나 Node.js 와 같은 다양한 프로그래밍 언어 또는 컨테이너 실행을 지원한다. 개발자는 자신이 작성한 코드를 해당 언어에 맞춰 업로드하거나, 컨테이너로 패키징하여 업로드할 수 있다. 본 논문에서는 컨테이너로 실행되는 서버리스 함수에 초점을 맞추었다.

콜드 스타트 문제는 서버리스 컴퓨팅에서 가장 대표적인 문제다. 콜드 스타트는 함수가 호출되었을 때 해당 요청을 처리할 컨테이너가 메모리 등에 적재되어 있지 않아 즉시 처리하지 못하는 경우를 의미한다. 콜드 스타트가 발생하는 경우, 새로운 컨테이너 인스턴스를 생성해야 하며 이는 함수의 실행을 큰 폭으로 지연시킨다. 서버리스 환경에서는 필요 없는 자원이 재활용되는 등 자원이 적극적으로 관리되고 이벤트는 비동기적으로 발생하기 때문에 콜드 스타트가 자주 발생하며, 콜드 스타트 발생을 줄이는 것이 중요하다.

(그림 1)은 서버리스 프레임워크에서 이벤트를 처리하는 과정을 보여준다. 이벤트가 발생하면, 서버리스 프레임워크는 이를 처리할 컨테이너 인스턴스가 로컬에 캐시되어 있는지 확인한다. 로컬에 캐시되지 않은 경우, 컨테이너 이미지를 저장하는 레지스트리에서 이미지를 풀링해야 한다. 이미지 풀링이 완료되면, 이를 기반으로 새로운 컨테이너를 생성하여 실행하게 된다.



(그림 1) 서버리스 컴퓨팅와 Docker 컨테이너

2.2. 멀티 스테이지 빌드

Docker 는 소프트웨어를 컨테이너화하여 배포, 실행하는 사실상 표준 플랫폼이다[5]. Docker 는 컨테이너 이미지를 빌드할 때, 레이어 역할을 하는 일련의 명령어들로 작성되는 Dockerfile 을 사용한다. 멀티 스테이지 빌드는 Dockerfile 을 작성할 때, 목적에 따라 여러 스테이지로 나누는 빌드 방법을 의미한다[6]. 일반적으로 '빌드(build)' 스테이지와 '릴리즈(release)' 스테이지로 나누지만, 필요에 따라 더 많은 스테이지로 구분하여 관리할 수 있다. 또한, 각 스테이지는 다른 스테이지를 기반으로 하거나 이전 스테이지에서 생성된 데이터에 접근할 수 있다.

멀티 스테이지에서 각 스테이지는 특정한 역할을 담당한다. 예를 들어, 빌드 스테이지는 개발 환경에서 필요한 도구와 종속성을 모두 포함하고, 릴리즈 스테이지는 실행 시 필요한 구성 요소만 포함하고 불필요한 다른 종속성을 제거할 수 있다. 따라서 릴리즈 스테이지를 통해 애플리케이션 실행에 필요한 사항만 포함된 간결한 런타임 이미지를 생성할 수 있다. 이는 컨테이너 이미지 크기를 줄이고, 결과적으로 이미지 풀링 시간을 감소시키는 데 효과적인 방식이다. 3 장에서는 멀티 스테이지 빌드 설계를 적용했을 때 발생하는 차이와 적절한 설계 방안을 제안한다.

3. 멀티스테이지 빌드 기반의 Dockerfile 최적화 실험

3.1. 실험 환경

실험을 위해, Intel® Core™ i9-10900K CPU 와 16GB 의 메모리로 구성된 서버에서 수행하였다. 실험은 AQUATOPE[7]에서 제공된 서버리스 애플리케이션 벤치마크를 사용하였다. 벤치마크에는 머신 러닝 파이프라인, 소셜 네트워크, 비디오 처리로 총 세 가지 애플리케이션이 포함되어 있다. 각 어플리케이션에는 Python 으로 작성된 여러 개의 모듈이 있으며 이는 각각 서버리스 함수를 나타낸다. 모듈에는 함수 코드를 패키징하는 컨테이너를 생성, 배포하는 데 필요한 구성요소가 포함되어 있다. 애플리케이션은 OpenWhisk 서버리스 프레임워크에서 작동하도록 작성되었으며, 사용된 Python 런타임 버전은 3.9.20 이다.

3.2 멀티스테이지 기반의 Dockerfile 최적화

본 연구는 모든 모듈의 Dockerfile 에 멀티 스테이지 빌드 기법을 적용하여 새로 작성하였다. 실험을 위해 수정을 가하지 않은 원본 벤치마크와 멀티스테이지 벤치마크 두 가지 Dockerfile 을 사용하였다.

멀티 스테이지 빌드 기법을 적용한 Dockerfile 에서는 스테이지를 '빌드' 스테이지와 '릴리즈' 스테이지

로 나누었다. 빌드 스테이지에서는 필요한 패키지를 설치하기 위한 'FROM' 명령어를 통해 원본 벤치마크에 제공된 Linux 기반 Debian OS, OpenWhisk 관련 종속성, Python 네이티브 라이브러리 패키지를 설치하였다. 해당 기반 이미지에 추가적인 Python 패키지를 컴파일하는데 필요한 리눅스 패키지 및 Python 종속성이 설치되었다. 이 방식을 사용하면 기반 이미지를 간결하게 유지할 수 있으므로, 최종 런타임 이미지에 Python 라이브러리를 컴파일할 때 필요한 Linux 패키지나 빌드 도구를 제외할 수 있다.

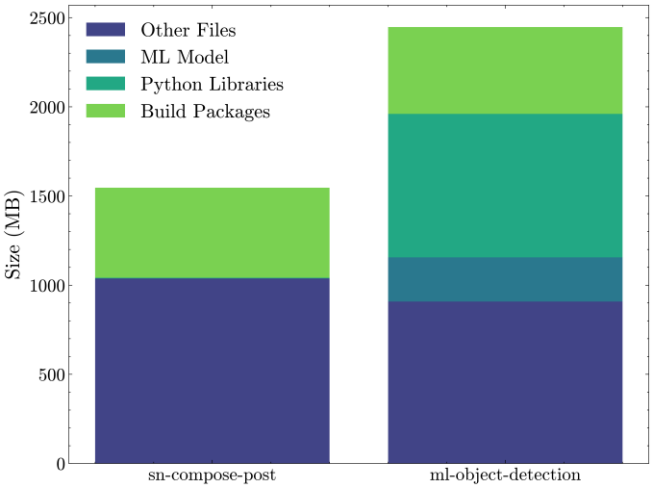
릴리즈 스테이지에서는 최소화된 Linux 기반 Debian OS 를 포함하는 슬림 버전의 기본 이미지를 사용하였으며, 모든 Dockerfile 에서 동일하게 사용되었다. 마지막으로 빌드 스테이지에서 빌드된 Python 종속성과 함수 코드를 복사하여 최종 이미지를 완성하였다.

릴리즈 스테이지에서는 'FROM' 명령어를 사용하여 벤치마크에서 사용된 원본 이미지와 유사하지만 이미지 크기가 작은 Debian OS 의 슬림 버전을 포함했다. 이후, 함수 코드와 빌드 스테이지에서 사용된 Python 종속성만을 복사하였다.

빌드가 완료된 이후 두 가지 버전의 함수 모두 Docker Hub 레지스트리에 업로드하였으며, 각 이미지가 빌드되는 데 걸리는 풀링 시간을 측정했다. 네트워크의 변동성으로 발생하는 영향을 최소화하기 위해 각 실험을 10 회 진행했으며, 풀링 시간의 측정 방식은 Linux 의 'time' 명령어를 사용했다. 풀링이 완료된 이후 다음 실험을 진행하기 전에 캐싱으로 발생하는 성능 간섭을 제거하기 위해 모든 이미지를 삭제했다. 실험 결과는 10 회 실험의 평균값을 제시하여 대표성을 확보하고자 했다.

3.3 실험 결과

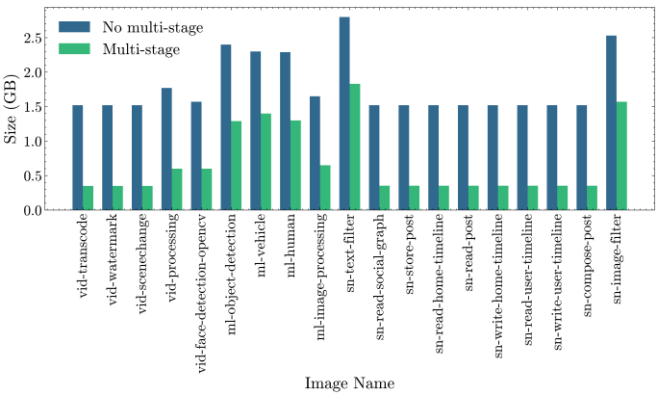
대부분의 함수가 비슷한 빌드 패키지와 적은 양의 Python 라이브러리를 포함하고 있기 때문에 이미지 크기가 비슷하게 축소되었다. (그림 2)는 두 가지 서버리스 함수 'social-network-compose-post'와 'ml-object-detection'에 멀티 스테이지 빌드 기법을 적용한 결과를 설명한다. 'social-network-compose-post'의 이미지는 build-essential, python-dev, libssl-dev, libffi-dev, ffmpeg 의 Linux 빌드 패키지를 포함한다. 이 빌드 패키지를 모두 합치면 504.91MB 의 크기가 되는데, 이는 원본 벤치마크 이미지의 전체 1.51GB 크기 중 약 33%에 해당한다.



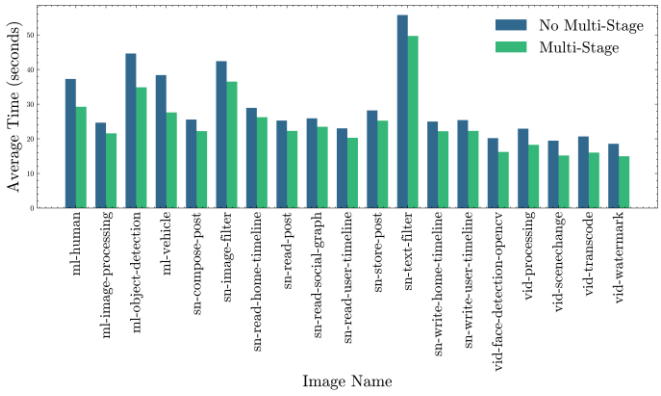
(그림 2) sn-compose-post 와 ml-object-detection 이미지 사이즈 구성

멀티 스테이지 빌드 기법을 적용하면, 전체 이미지 크기가 352.64MB 로 줄어든다. 이는 필요한 파이썬 패키지만 복사하고 불필요한 Linux 빌드 패키지는 제외하며 최소 용량의 OS 이미지를 사용했기 때문이다. (그림 4)을 통해 멀티 스테이지 빌드의 효과가 이미지 풀링 시간에 직접적인 영향을 끼친다는 것을 확인할 수 있으며, 유사한 구성을 가진 'social-network' 그리고 'video-processing' 함수들도 약 4 초 가량의 풀링 시간 감소를 보였다.

주목할 점은 더 큰 이미지 크기를 가진 'machine-learning pipeline' 모듈에서 멀티스테이지 빌드를 적용했을 때 이미지 크기가 더 적게 감소했다는 것이다. 그러나 이미지 풀링 시간 자체는 더 크게 감소했다. 'ml-object-detection' 함수에는 총 487MB 의 용량의 build-essential, python-dev, python3-distutils, libgomp1 과 같은 Linux 패키지가 포함되어 있다. 또한, 약 247MB 가량의 머신 러닝 관련 모델을 추가로 다운로드하는 'caching_models.py' 라는 추가 python 소스 코드도 포함되어 있어 총 이미지 크기는 2.39GB 로 나타났다.



(그림 3) 모든 함수 이미지 사이즈



(그림 4) 모든 함수 이미지 풀링 시간

해당 컨테이너에 멀티 스테이지 빌드 기법을 적용하면 이미지 크기를 1.4GB 로 줄일 수 있다. 이미지 크기 감소 폭은 'social-network-compose-post' 함수보다 작지만, 풀링 타임은 원본에 비해 약 9.75 초 감소했다. 비슷한 구성을 가진 'ml-human' 및 'ml-vehicle' 함수도 약 9~10 초의 풀링 타임 감소를 보였다. 'social-network-image-filter' 및 'social-network-text-filter' 함수는 이미지 크기와 구성이 모두 머신 러닝 이미지와 유사했지만 이미지 풀링 타임 감소 폭이 약 6 초로 더 작게 나타났다. 이러한 결과는 Dockerfile 에 추가 파일을 다운로드하는 'RUN python -m textblob.download_corpora' 명령어가 포함되어 있기 때문이다.

전반적으로, 멀티 스테이지 빌드 기법을 사용하면 이미지 크기를 성공적으로 줄일 수 있으며, 특히 릴리즈 스테이지에서 빌드 단계 종속성을 많이 제외할 수 있는 이미지의 경우 이미지 감소 폭이 크게 나타났다. 'ml-object-detection'과 같이 크기가 큰 함수의 경우 이미지 크기 감소 폭은 낮았지만 이미지 풀링 시간은 다른 이미지보다 더 큰 폭으로 감소했다.

4. 결론

멀티 스테이지 빌드 기법은 컨테이너의 이미지 크기를 줄이는 데 성공적인 효과를 보였다. 그러나 애플리케이션의 유형과 사용된 종속성에 따라 이미지 크기 감소 폭이 다르게 나타나고, 이미지 크기 감소가 이미지 풀링 시간 감소와 항상 비례하지는 않았다. 멀티 스테이지 빌드를 적용하면 이미지 풀링 시간을 감소시킬 수 있지만, 이미지 풀링 시간에 영향을 미치는 다른 요인도 있으므로 그 또한 고려해야 한다.

향후 이미지 풀링 시간에 영향을 미치는 요인들을 분리하고, 다양한 애플리케이션 유형 및 언어 또는 프레임워크를 대상으로 실험을 진행할 예정이다. 또한, 다양한 서버리스 프레임워크에서 부하 테스트를 수행하여 각 서버리스 환경에 특화된 콜드 스타트 문제를 완화하는 방법을 연구하고자 한다.

Acknowledgement

본 연구는 산림청(한국임업진흥원) '산림분야 재난·재해의 현안해결형 연구개발사업 (RS - 2022-KF002134)'의 지원에 의하여 이루어진 것입니다.

참고문헌

[1] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, & Yue Cheng (2021). FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute. In 2021 USENIX Annual Technical Conference (USENIX ATC 21) (pp. 443–457). USENIX Association.

[2] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In 2020 USENIX Annual Technical Conference (USENIX ATC 20), pages 205–218. USENIX Association, July 2020.

[3] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit S Warke, Mohamed Mohamed, and Ali R Butt. Largescale analysis of the docker hub dataset. In 2019 IEEE International Conference on Cluster Computing (CLUSTER), pages 1–10. IEEE, 2019.

[4] J. Manner (2023). A Structured Literature Review Approach to Define Serverless Computing and Function as a Service. In 2023 IEEE 16th International Conference on Cloud Computing (CLOUD) (pp. 516-522).

[5] Docker, Docker [internet], https://www.docker.com/

[6] Docker, Multi-stage build [internet], https://docs.docker.com/build/building/multi-stage/

[7] Zhuangzhuang Zhou, Yanqi Zhang, & Christina Delimitrou. (2022). QoS-Aware Resource Management for Multi-phase Serverless Workflows with Aquatope.