Serverless 기반 워크플로 실행에서 Cold Start 완화를 위한 런타임 인터페이스에 관한 연구†,‡

황승현, 강지훈, 유헌창
고려대학교 대학원 컴퓨터학과
e-mail : {somnus, k2j23h, yuhc}@korea.ac.kr

Study on Runtime Interface for Mitigating Cold Start in Serverless based Workflow

Seunghyun Hwang, Jihun Kang, Heonchang Yu
Dept. of Computer Science and Engineering, Korea University

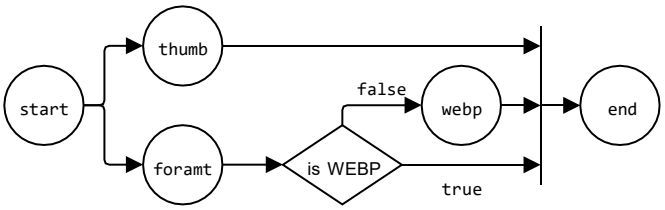
요 약

Serverless 플랫폼은 작업이 요청되었을 때 인스턴스를 생성하였다가 작업이 끝난 후 제거되기 때문에 실제 작업을 처리할 때만 요금이 부과되어 경제적이다. 하지만 작업이 요청될 경우 인스턴스가 생성되고 작업에 필요한 리소스를 가져오는 지연시간 즉, Cold Start 가 발생한다. 특히 워크플로 실행에서 각 작업이 개별적으로 처리되기 때문에 워크플로의 각 단계마다 Cold Start 가 발생하는데, 워크플로는 사전에 정의되기 때문에 이러한 Cold Start 의 누적은 회피 될 수 있다. 본 논문은 Serverless 환경에서 워크플로를 실행할 때 누적되는 Cold Start 를 완화하기 위해 앞의 단계에서 다음 단계를 미리 준비시키는 런타임 인터페이스들에 대하여 제안한다. 이미지 프로세싱 워크플로에 제안한 런타임 인터페이스를 적용시켜 Cold Start 를 제외한 워크플로의 실행시간을 최대 21%까지 줄였다.

1. 서론

Serverless 환경에서 작업을 처리하기 위한 인스턴스는 작업이 할당되었을 때만 존재하며 작업이 종료되면 인스턴스도 제거된다. Serverless 플랫폼은 사용되지 않는 자원을 특정 사용자를 위해 예약해 놓을 필요가 없어 가용자원을 확보할 수 있으며 사용자는 작업 요청을 대기하기 위해 자원을 미리 점유하는 동안의 요금이 발생하지 않아 플랫폼과 사용자 모두 경제적인 이득을 볼 수 있다. 하지만 Serverless 플랫폼의 이러한 특성상 작업 요청이 발생할 경우, 작업을 처리하기 위한 샌드박스가 생성되고 작업에 필요한 라이브러리가 설치되는 지연시간, Cold Start 를 겪게 된다.[1] 게다가 기존의 Serverless 플랫폼에서 워크플로 실행은 워크플로의 작업을 차례대로 실행하는 독립적인 작업으로 처리되기 때문에 각 작업들의 실행은 개별적으로 처리되어 워크플로의 각 단계마다 Cold Start 가 발생된다.

Serverless 환경에서 워크플로를 실행할 때 Cold Start 로 인한 오버헤드를 보이기 위해 (그림 **1**)과 같은 이미지 프로세싱 워크플로를 AWS Lambda Node.js 런타임으로 구현하여 AWS Step Functions 로 실행하였다. (그림 **1**)의 워크플로는 요청된 이미지의 Thumbnail 을 만들면서(`thumb`) 이미지의 포맷이 WEBP 인지 확인하고(`format`) WEBP 가 아닐 경우

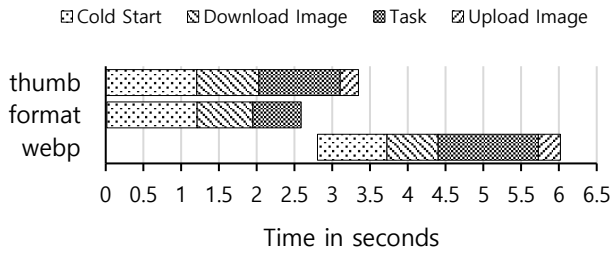


(그림 1) Thumbnail 을 생성하는 이미지 프로세싱 워크플로

WEBP 로 변환(`webp`)한다. 작업 `thumb` 와 `format` 은 병렬로 실행되고, 작업 `webp` 는 작업 `format` 이 끝난 후 실행되며 작업에 사용되는 이미지는 AWS S3 에 저장된다. 워크플로는 작업 `thumb` 와 `webp` 가 완료되면 종료된다. (그림 **2**)는 (그림 **1**)의 워크플로를 실행한 결과를 보여준다. 워크플로의 총 실행시간은 6094ms 으로, 만약 작업 `webp` 를 처리하기 위한 인스턴스를 작업 `format` 이 실행중인 동안 준비했다면 워크플로의 총 실행시간은 5038ms 로 줄어들 수 있으며, 작업 `webp` 가 처리해야할 이미지를 미리 다운로드 할 수 있다면 총 실행시간은 4216ms 까지 줄어들 수 있다.

따라서 본 논문에서는 워크플로가 사전에 정의되어 있다는 것을 고려하여 워크플로가 시작되면 앞에서 실행되는 작업이 처리되는 동안 나중에 실행되는 작업들을 미리 준비시켜 누적되는 Cold Start 를 완화시키기 위한 런타임 인터페이스를 제안한다. 본 논문의 기본 아이디어는 다음에 실행될 작업을 사전에 알고 있으므로 미리 준비시킨다는 것이다. 이는 런타임 인터페이스가 아니라 플랫폼이 자체적으로 실행될 작업을 미리 준비시킴으로써 구현될 수 있지만, 이러한 접근은 두가지 한계점을 가진다. 첫번째로, 워크플

†본 연구는 과학기술정보통신부 및 정보통신기술진흥센터의 대학 ICT 연구센터 지원사업의 연구결과로 수행되었음 (IITP-2018-0-001405)
‡이 논문은 2018 년도 정부(과학기술정보통신부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임 (No.2018-0-00480)



(그림 2) Thumbnail 을 생성하는 이미지 프로세싱 워크플로를 AWS Serverless 플랫폼에서 실행한 결과

로는 분기점을 가질 수 있다. 워크플로 전체를 미리 준비시킨다면 분기문에 의해 실행되지 않는 작업 인스턴스들에 대해서 자원이 낭비된다. 분기문에 의한 자원 낭비는 미리 준비시킬 작업 인스턴스의 수를 제한함으로써 해결될 수 있지만, 이는 두번째 한계점을 야기한다. 주요 Serverless 플랫폼은 Scalability 를 위해서 작업이 Stateless 할 것을 권장하며 작업의 최대 실행시간을 제한한다.[2] 따라서 사용자는 작업이 단일 책임을 가지도록 설계하기 때문에 큰 작업을 여러 단계의 작은 작업들로 분해되어 각 작업들의 실행시간은 단축된다. 만약 어떤 작업의 실행시간이 다음 작업의 Cold Start 로 인한 지연시간보다 짧으며, 이러한 상황이 연속된다면 작업 인스턴스를 미리 준비시킴으로써 얻을 수 있는 이득은 사실상 없을 것이다. 따라서 다음 작업을 언제 준비시킬지에 대한 결정은 플랫폼이 아니라 사용자에 있어야 하며, 결정은 런타임에 이루어 져야할 필요성이 있다.

Serverless 컴퓨팅에서 Cold Start 는 알려진 문제 중 하나로,[3] 2 장에서 이 문제와 관련된 연구에 대해서 소개한다. 3 장에서 본 논문이 제안하는 런타임 인터페이스에 대해서 설명하고, 4 장에서 제안한 런타임 인터페이스를 적용하였을 때의 성능 향상을 실험을 통해 보이고 한계점에 대해서 설명한다. 마지막으로 5 장에서 본 논문의 결론과 향후 연구에 대해서 설명한다.

2. 관련 연구

Oakes, Edward, et al.[4]은 Serverless 컴퓨팅에 최적화된 컨테이너 SOCK 을 개발했다. Docker 컨테이너에서 Serverless 작업을 처리하는데 불필요한 기능들을 제거하고 Zygote 기법을 이용하여 라이브러리가 미리 로드되어 있는 컨테이너를 fork 함으로써 라이브러리 설치와 로드되는 비용을 조건적으로 무시할 수 있다. 결과적으로 Docker 보다 약 18 배 빠르게 컨테이너가 생성되었으며 Zygote 가 가능할 경우 21 배까지 빠르게 컨테이너를 생성할 수 있었다. Serverless 인스턴스 생성의 가장 큰 병목은 라이브러리 설치인데, SOCK 의 경우 가능한 라이브러리를 미리 로컬 스토리지에 저장해서 설치하는 시간을 절약하는 것으로, 적지 않은 스토리지 자원을 차지한다. 이는 다양한 런타임을 필요로 할 수 있는 워크플로 실행의 경우에는 제약으로 작용하게 된다.

Akkus, Istemi Ekin, et al.[5]은 워크플로를 고려한 Serverless 시스템인 SAND 를 제안했다. 워크플로의 각 단계가 차례대로 실행되는 것을 고려하여 워크플로 실행에 필요한 라이브러리를 하나의 컨테이너에 모두 설치하고 워크플로 인스턴스가 단 하나의 컨테이너에서만 처리되도록 하였다. 결과적으로 워크플로가 실행되는 동안 최초 한번을 제외하고 Cold Start 가 발생하지 않게 되었다. 하지만 병렬처리되는 작업의 경우 성능격리를 시키지 못하며, SOCK 과 마찬가지로 여러 개의 런타임을 요구하는 워크플로는 처리하지 못한다.

3. 런타임 인터페이스

본 장에서는 워크플로를 실행할 때 앞의 단계에서 뒤의 단계를 미리 준비시키는 것과 관련된 세가지 런타임 인터페이스 `Prepare`, `Wait`, `IsAvailable` 을 제안한다. 인터페이스 `Prepare` 는 작업 인스턴스로 사용될 컨테이너를 미리 점유하고 사용될 라이브러리를 미리 로드시키며, 필요한 경우 작업을 미리 준비시킬 수 있는 수단을 제공한다. 인터페이스 `WaitTurn` 은 직전 작업이 끝나기까지 기다리는 인터페이스이다. 인터페이스 `IsAvailable` 은 이용가능한 대기중인 작업 인스턴스가 있는지 확인하는 인터페이스로, Prepare 와 달리 작업 인스턴스를 점유하지 않는다.

3.1. Prepare

```
void Prepare(Descriptor d, Array<String> params)
```

인터페이스 `Prepare` 는 위와 같이 정의된다. 타입 `Descriptor` 는 플랫폼에 등록된 사용자의 작업을 특정하는 설명자이다. 플랫폼은 인자 `d` 로 특정되는 작업을 인자 `params` 를 매개변수로 실행한다.

3.2. WaitTurn

```
Array<String> WaitTurn()
```

인터페이스 `WaitTurn` 은 위와 같이 정의된다. 인터페이스 `Prepare` 로 실행된 작업은 미리 실행된 작업이기 때문에 아직 워크플로가 해당 단계까지 진행되지 않았음을 의미한다. 따라서 해당 작업이 실행되어야 할 단계까지 인터페이스 `WaitTurn` 은 호출된 시점에서 현재 작업을 중지시킨다. 워크플로가 `WaitTurn` 을 호출한 작업의 단계에 도달한 경우, `WaitTurn` 은 중지상태가 끝나고 이전 작업의 결과를 반환한다.

3.3. IsAvailable

```
bool IsAvailable(Descriptor d)
```

인터페이스 `IsAvailable` 은 위와 같이 정의된다. 기존의 Serverless 플랫폼에는 최근에 요청된 작업이 가까운 시간내에 다시 요청될 경우를 대비하여 이전에 사용되었던 인스턴스의 삭제를 보류하고 다시 요청되었을 때 이를 재사용하는 정책이 있다.[6] 인터페이스 `IsAvailable` 은 인자 `d` 로 특정되는 작업을 실행할 요청수가 해당 작업을 처리할 수 있는 인스턴스의 수보다 작을 경우 `true`, 그렇지 않을 경우 `false` 를 반환한다.

인터페이스 `Prepare` 만으로도 Cold Start 를 완화할 수 있지만, 인터페이스 `IsAvailable` 이 필요한 이유는 요금과 관련이 있다. 인터페이스 `Prepare` 로 인해 실행된 인스턴스가 인터페이스 `WaitTurn` 으로 작업이 중단된 상태일지라도 해당 인스턴스는 미리 점유된 상태이기 때문에 다른 작업을 처리하는데 사용될 수 없다. 따라서 작업이 중단된 시간 동안에도 플랫폼은 사용자에게 요금을 청구하게 된다.

인터페이스 `Prepare` 가 실행될 인스턴스를 미리 점유하지 않게 되면 다음과 같은 상황이 발생한다. 작업 F를 두번째 단계에서 실행하는 워크플로 W_1과 작업 F를 첫번째 단계에서 실행하는 워크플로 W_2가 있다고 하자. 워크플로 W_1은 첫번째 단계에서 인터페이스 `Prepare` 를 호출하여 작업 F를 준비시키고, 플랫폼은 작업 F를 처리하기 위한 인스턴스 S_F를 생성한다. 이 때 워크플로 W_1의 첫번째 단계가 끝나기 전에 워크플로 W_2의 실행요청이 발생한다면, 플랫폼

```
1. Descriptor next;
2.
3. function f(conf){
4.     init(conf);
5.
6.     if(!IsAvailable(next)){
7.         Prepare(next);
8.     }
9.
10.    arg := WaitTurn();
11.    rst := doTask(arg);
12.    return rst;
13. }
```

(그림 3) 제안하는 인터페이스의 용례

은 워크플로 W_2의 요청을 처리하기 위해 S_F를 할당할 것이다. 따라서 워크플로 W_1의 첫번째 단계가 끝나고 두번째 단계인 작업 F 를 시작할 때 워크플로 W_1은 Cold Start 를 겪게 된다.

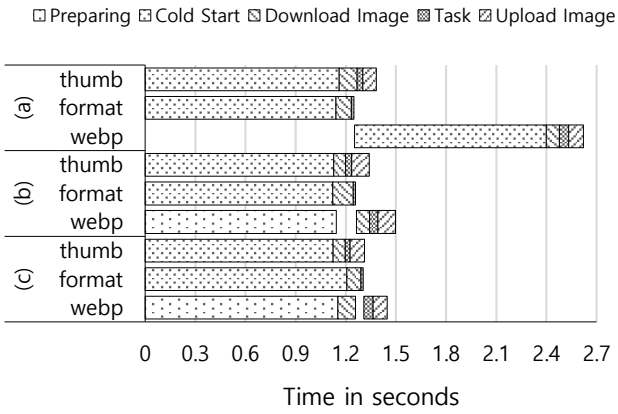
3.4. 용례

(그림 3)은 본 장에서 설명한 인터페이스를 적용한 작업 `f` 의 예를 보여준다. 1 번 줄의 변수 `next` 는 다음에 실행되어야 할 작업을 특정하는 설명자이다. 4 번 줄에서 작업 `f` 는 이전 작업이 인터페이스 `Prepare` 로 넘겨준 초기화 정보를 받아 작업을 준비한다. 작업 `f` 는 `doTask` 의 실행시간이 Cold Start 시간 보다 짧은 것을 고려하여 6 번 줄에서 `next` 가 재사용 가능한지 확인하고, 재사용 불가능 하다면 7 번 줄에서 미리 준비시킨다. 그리고 10 번 줄에서 차례를 기다렸다가 작업을 수행한다. 만약 `doTask` 의 실행시간이 Cold Start 보다 길 경우 `doTask` 의 남은 실행시간이 Cold Start 에 필요한 시간과 가까워지는 시점에서 인터페이스 `Prepare` 를 호출한다면 사용자는 지불 비용을 최적화할 수 있다.

4. 실험

자체개발한 Serverless 플랫폼인 Metro 의 Node.js 런타임에 4 장에서 제안한 런타임 인터페이스를 구현하여 (그림 **1**)의 이미지 프로세싱 워크플로를 다음 세가지 경우에 대해서 실행하였다: (a) 제안한 런타임 인터페이스를 사용하지 않을 경우. (b) 런타임 인터페이스 `Prepare` 로 인스턴스를 점유하고 점유된 인스턴스가 바로 인터페이스 `WaitTurn` 을 호출할 경우. (c) 런타임 인터페이스 `Prepare` 로 인스턴스를 점유하고 점유된 인스턴스가 작업에 필요한 이미지를 미리 다운로드 받은 후 인터페이스 `WaitTurn` 을 호출할 경우.

자체개발한 Serverless 플랫폼은 컨테이너풀을 관리하여 컨테이너 생성 시간은 발생하지 않지만 작업에 필요한 라이브러리를 설치하는 시간으로 인해 약 1 초 정도의 Cold Start 가 발생한다. 작업에 사용되는 이미지는 AWS S3 에 저장된다. (그림 **4**)는 실험의 결과를 보여준다. 제안한 런타임 인터페이스를 적용하지 않은 (a)의 경우 AWS 에서 실행한 결과와 마찬가지로 작업 `webp` 가 작업 `format` 이 끝난 후 Cold Start 를 겪게 되어 워크플로가 완료되는데 2621ms 가 걸렸다. 인터페이스 `Prepare` 를 이용한 (b)의 경우 작업 `webp` 가 작업 `format` 이 끝나기 전에 실행이 준비가 된 상태이므로, 작업 `format` 이 끝난 후 바로 실행이 시작되어 워크플로가 완료되는데 1497ms 가 걸렸다. 인터페이스 `Prepare` 로 작업에 필요한 리소스 정보를 받아 미리 작업 준비를 한 (c)의 경우는 작업할 이미지를 미리 다운로드 받아 실행이 시작되었을 때 이미지를 다운 받는 시간을 절약하여 워크플로가 완료되는데 1311ms 가 걸렸다.



(그림 4) Thumbnail 을 생성하는 이미지 프로세싱 워크플로를 제안한 런타임 인터페이스를 적용하여 실행한 결과

결과적으로 본 논문이 제안하는 런타임 인터페이스를 적용함으로써 Serverless 환경에서 워크플로를 실행할 때 누적되는 Cold Start 를 완화할 수 있으며, 워크로드에 따라 워크플로 실행시간을 최적화할 수 있다. Cold Start 의 평균적인 시간이 t_c라고 했을 때, N개의 작업을 연속적으로 실행하는 워크플로의 경우 제안한 인터페이스로 얻을 수 있는 최대의 시간 이득은 $(N-1)\times t_c$ 이다. 또한 인터페이스 `WaitTurn` 을 호출하기 전에 준비에 필요한 정보를 미리 받아 준비할 경우 시간 이득은 더 커진다.

본 논문에서 제안한 런타임 인터페이스를 활용할 경우 실행시간의 이득은 볼 수 있지만 요금은 더 많이 부과될 수 있다. 예를 들어 5 장의 실험에서, (a)의 경우 사용자가 지불해야 할 작업의 총 실행시간이 549ms 이지만, (b)의 경우에는 인터페이스 Prepare 로 인해 Preparing 단계 직후부터 요금이 부과되어 지불해야 할 총 시간은 702ms 이다. (c)의 경우에는 Preparing 단계이후 바로 인터페이스 `WaitTurn` 을 호출하지 않고 필요한 리소스를 미리 다운받는데 활용하여 지불해야할 총 시간이 578ms 이다. 따라서 제안한 런타임 인터페이스를 효율적으로 활용하기 위해서는 적절한 시점에 인터페이스 `Prepare` 를 호출해야 하며, 다음 작업에 필요한 리소스를 준비하는데 필요한 정보를 미리 주어 인터페이스 `WaitTurn` 을 가능한 늦게 호출해야 한다. 그리고 별도의 필요한 리소스가 없는 작업의 경우에는 인터페이스 `IsAvailable` 을 호출하여 재사용 가능한 인스턴스를 확인함으로써 비용을 최적화할 수 있다. 시간에 민감한 작업의 경우에는 인터페이스 Prepare 로 인스턴스를 미리 점유해야 하지만, 이것은 성능과 비용의 트레이드-오프이다.

5. 결론

본 논문에서는 Serverless 환경에서 워크플로를 실행할 때, 플랫폼이 워크플로 실행을 고려하지 못하여 Cold Start 로 인한 지연시간이 누적되는 문제를 AWS Serverless 플랫폼에서 이미지 프로세싱 워크플로를 실행하여 보였다. 그리고 이러한 문제점을 완화하기 위한 런타임 인터페이스의 필요성에 대해서 제기하고 필요한 런타임 인터페이스를 제안하고 직접 구현하여 실험함으로써 그 효용성을 보였다.

제안한 런타임 인터페이스는 다음 작업에 필요한 인스턴스를 미리 준비시키고 가능한 경우 워크플로 실행시간을 최적화할 수 있는 수단을 제공한다. 하지만 사용자가 지불해야할 요금이 더 많아 질 수 있다. 이는 성능과 비용의 트레이드-오프이며, 사용자가 실행하는 워크로드에 대한 이해도가 높다면 비용 또한 최적화 가능한 문제이다.

향후 연구로는, 워크플로 실행을 고려하여 각 작업의 상태를 워크플로 실행 시간 동안 유지할 수 있는 방법에 대한 연구가 필요하다고 생각된다. 본 논문에서 보인 이미지 프로세싱 워크플로의 경우, Serverless 플랫폼의 각 작업이 Stateless 를 고려해야하기 때문에 각 작업에서 같은 이미지를 반복해서 다운로드하고 있다. 만약 각 작업이 호스트의 같은 스토리지 공간을 공유할 수 있다면 플랫폼은 작업이 상태를 저장하고 다른 작업에서 그 상태를 되찾는데 필요한 네트워크 트래픽 비용을 절감할 수 있으며, 사용자의 경우에는 워크플로 실행시간이 줄어들어 지불해야할 비용을 절감할 수 있다.

참고문헌

[1] Wang, Liang, et al. "Peeking behind the curtains of serverless platforms." 2018 {USENIX} Annual Technical Conference (USENIX ATC 18). 2018.

[2] AWS Lambda Limits. [online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/limits.html

[3] Baldini, Ioana, et al. "Serverless computing: Current trends and open problems." Research Advances in Cloud Computing. Springer, Singapore, 2017. 1-20.

[4] Oakes, Edward, et al. "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers." *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018.

[5] Akkus, Istemi Ekin, et al. "SAND: Towards High-Performance Serverless Computing." *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018

[6] Understanding Container Reuse in AWS Labmda. 2014. [online]. Available: https://aws.amazon.com/ko/blogs/compute/container-reuse-in-lambda/