The Transactions of the Korea Information Processing Society
TKIPS Vol.13, No.3, pp.91~101
eISSN: 3022-7011
https://doi.org/10.3745/TKIPS.2024.13.3.91

Technique to Reduce Container Restart for Improving Execution Time of Container Workflow in Kubernetes Environments

Taeshin Kang† · Heonchang Yu††

ABSTRACT

The utilization of container virtualization technology ensures the consistency and portability of data-intensive and memory volatile workflows. Kubernetes serves as the de facto standard for orchestrating these container applications. Cloud users often overprovision container applications to avoid container restarts caused by resource shortages. However, overprovisioning results in decreased CPU and memory resource utilization. To address this issue, oversubscription of container resources is commonly employed, although excessive oversubscription of memory resources can lead to a cascade of container restarts due to node memory scarcity. Container restarts can reset operations and impose substantial overhead on containers with high memory volatility that include numerous stateful applications. This paper proposes a technique to mitigate container restarts in a memory oversubscription environment based on Kubernetes. The proposed technique involves identifying containers that are likely to request memory allocation on nodes experiencing high memory usage and temporarily pausing these containers. By significantly reducing the CPU usage of containers, an effect similar to a paused state is achieved. The suspension of the identified containers is released once it is determined that the corresponding node's memory usage has been reduced. The average number of container restarts was reduced by an average of 40% and a maximum of 58% when executing a high memory volatile workflow in a Kubernetes environment with the proposed method compared to its absence. Furthermore, the total execution time of a container workflow is decreased by an average of 7% and a maximum of 13% due to the reduced frequency of container restarts.

Keywords : Resource Management, Kubernetes, Container Workflow, Memory Oversubscription

쿠버네티스 환경에서 컨테이너 워크플로의 실행 시간 개선을 위한 컨테이너 재시작 감소 기법

강 태 신† · 유 헌 창††

요 약

데이터 집약적이고 메모리 변동성이 높은 워크플로의 이식성 보장을 위해 컨테이너 가상화 기술이 사용되고 있다. 그리고 쿠버네티스는 이러한 컨테이너 애플리케이션들을 관리하기 위한 오케스트레이션 도구로써 사실상 표준으로 사용되고 있다. 클라우드 사용자는 리소스 부족으로 인한 컨테이너 재시작을 방지하기 위해 컨테이너 애플리케이션을 오버프로비저닝하는 경향이 있다. 그러나 과도한 오버프로비저닝은 CPU, 메모리 등 시스템 리소스의 사용량을 낮아지게 만든다. 이 문제를 해결하기 위해 컨테이너 리소스를 초과 사용하는 방식이 널리 사용되고 있으나, 지나친 메모리 리소스 초과 사용은 노드의 메모리 부족으로 인해 연쇄적인 컨테이너 재시작을 유발할 수 있다. 컨테이너 재시작 발생 시 작업을 처음부터 다시 시작해야 하므로 많은 상태저장 애플리케이션이 포함된 메모리 변동성이 높은 컨테이너에 큰 오버헤드를 유발할 수 있다. 본 논문은 쿠버네티스 환경에서 메모리 초과 사용 시 컨테이너 재시작을 완화하는 기법을 제안한다. 메모리 사용량이 많은 노드에서 메모리 할당을 요청할 가능성이 큰 컨테이너를 식별하고 이러한 컨테이너를 일시정지한다. 컨테이너의 CPU 사용량을 크게 줄이면 컨테이너가 일시정지하는 상태와 유사한 효과를 얻을 수 있다. 해당 노드의 메모리 사용량이 개선된 것으로 판단되면 컨테이너의 일시정지를 해제한다. 제안기법을 적용하여 쿠버네티스 환경에서 메모리 변동성이 높은 워크플로를 구동한 경우 제안기법을 사용하지 않았을 때에 비해 컨테이너의 재시작 횟수가 평균 40%, 최대 58% 감소하였다. 그리고 컨테이너 재시작 횟수 감소로 인해 컨테이너 워크플로의 총 실행 시간이 평균 7%, 최대 13% 단축되었다.

키워드 : 자원관리, 쿠버네티스, 컨테이너 워크플로, 메모리 초과 사용

※ 이 논문은 2023년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임(No.2022-0-01198, 융합보안대학원(고려대학교)).
※ 본 연구는 과학기술정보통신부 및 정보통신기획평가원의 ICT혁신인재4.0 사업의 연구결과로 수행되었음(IITP-2023-RS-2022-00156439).

† 준 회 원 : 고려대학교 컴퓨터학과 석박사통합과정
†† 종신회원 : 고려대학교 컴퓨터학과 교수
Manuscript Received : August 1, 2023
Accepted : September 26, 2023
* Corresponding Author : Heonchang Yu(yuhc@korea.ac.kr)

1. 서 론

최근 과학적 워크플로, 기계학습, 빅데이터 등 대규모의 데이터 세트를 활용하는 분야가 많아지고 있으며, 이에 따라 데이터 집약적이고 메모리 변동성이 높은 워크플로 구동에 대한 수요가 증가하고 있다. 이러한 작업을 비용 효율적으로 수행하고, 작업에 대한 관리 편의성 증진을 위해 VM(가상 머신 기술) 기반 클라우드 환경이 널리 채택되었다[1-3]. 그러나 VM 기술은 리소스 활용도가 낮고, 탄력적 확장 및 클러스터 확장에 많은 시간이 소요되며 여러 노드를 동일하게 구성하는 과정이 복잡하고 반복적인 경향이 있다[4]. 컨테이너 가상화 기술은 이러한 문제를 해결하기 위한 더 나은 대안을 제시한다. 컨테이너 가상화 기술은 이식성이 보장되면서도 VM 가상화 기술에 비해 오버헤드가 적어 최근 많이 사용되고 있으며 관련 연구가 활발하게 진행되고 있다[5-7].

컨테이너 기술을 사용하여 데이터 집약적이고 메모리 변동성이 높은 워크플로를 효율적으로 구동하고 관리하려면 각 컨테이너에 대해 리소스 제한, 로드 밸런싱 및 오토 스케일링 등을 효과적으로 수행할 수 있어야 한다[8]. 이러한 기능을 지원하는 컨테이너 오케스트레이션 도구로 Apache Yarn[9], SwarmKit[10](Docker-Swarm[11]), Kubernetes[12] 등이 사용되고 있다. 특히 쿠버네티스는 컨테이너 오케스트레이션 영역에 있어서 사실상 표준이다.

클라우드 사용자는 리소스 부족으로 인한 컨테이너 재시작을 예방하기 위해 컨테이너 애플리케이션을 오버프로비저닝하는 경향이 있다. 그러나 이러한 관행은 노드 당 할당되는 노드의 시스템 리소스(CPU, 메모리 등) 활용률을 낮아지게 만든다[13]. 이 문제를 해결하기 위해 클라우드 환경에서 컨테이너 리소스 초과 사용이 널리 사용되고 있다[14]. 리소스 초과 사용은 많은 컨테이너가 동시에 많은 리소스를 요구하지 않을 것이라고 가정한다. 컨테이너 리소스를 초과 사용하면 더 많은 컨테이너 애플리케이션을 노드에서 실행할 수 있어 시스템 리소스 활용률이 증진되어 전력 및 비용 효율성이 증가하며, 제한된 리소스 내에서 더 많은 컨테이너를 실행할 수 있기 때문에 워크플로의 실행 시간이 짧아진다. 그러나 과도한 메모리 리소스 초과 사용은 노드의 메모리 부족으로 인해 연쇄적인 컨테이너 재시작을 유발할 수 있다. 쿠버네티스는 기본적으로 스왑 메모리를 지원하지 않기 때문에 시스템의 물리적 메모리를 초과하여 메모리 할당을 요청한 컨테이너는 재시작하게 된다. 컨테이너 재시작은 진행하고 있던 작업을 처음부터 다시 시작해야 하므로, 특히 상태저장 애플리케이션이 많은 메모리 변동성이 높은 컨테이너에 큰 오버헤드를 유발할 수 있다.

본 논문은 메모리 초과 사용을 사용하여 쿠버네티스에서 컨테이너 재시작을 줄이는 기법을 제안한다. 제안기법을 사용하여 쿠버네티스 환경에서 메모리 변동성이 높은 워크플로를 구동했을 때와 비교 실험한 결과 제안기법을 사용하지 않았을 때에 비해 컨테이너의 재시작 횟수를 평균 40%, 최대 58% 줄였으며, 컨테이너 재시작 횟수 감소로 인해 컨테이너 워크플로의 총 실행 시간이 평균 7%, 최대 13% 감소하였다.

본 논문은 다음과 같이 구성된다. 2장에서 쿠버네티스 환경에서 기존 리소스 관리 방식과 해당 방식의 한계점에 관해 설명하고, 3장에서는 컨테이너 워크플로의 실행 시간을 개선하기 위한 관련 연구를 조사하고 각 연구의 한계점에 관해 기술한다. 4장에서는 메모리 초과 사용률에 따른 실행 시간 지연율을 분석하고, 컨테이너의 CPU 리소스 제한을 통해 컨테이너를 충분히 일시정지 할 수 있음을 실험을 통해 보여준다. 5장에서는 제안하는 컨테이너 재시작 감소 기법을 모델링하고 이를 구현한다. 6장에서는 제안한 기법의 효용성을 평가한다. 마지막으로, 7장에서는 본 연구의 성과를 분석하고, 향후 연구 방향을 기술한다.

2. 쿠버네티스의 자원관리 방식 및 한계점

2.1 쿠버네티스 구조

쿠버네티스는 컨테이너 단위가 아닌 파드 단위로 관리를 수행한다. 파드는 쿠버네티스에서 생성하고 관리할 수 있는 배포 가능한 최소 컴퓨팅 단위이며 컨테이너 애플리케이션 워크로드들의 구성요소로 정의된다. 파드에는 여러 개의 컨테이너가 포함될 수 있다. Fig. 1은 쿠버네티스의 전체적인 구조를 나타낸 것이다[16].

쿠버네티스 클러스터는 컨테이너 애플리케이션을 실행하는 워커 머신 집합(워커 노드)으로 구성된다. 모든 클러스터에는 애플리케이션 워크로드의 구성요소인 파드의 호스팅을 담당하는 하나 이상의 워커 노드가 존재한다. 마스터 노드는 클러스터의 워커 노드와 파드를 관리한다. 실제 환경에서 쿠버네티스 클러스터는 여러 마스터 노드에서 control plane이 실행되고, 여러 워커 노드에서 컨테이너 애플리케이션이 실행되어 내결함성과 고가용성을 제공한다.

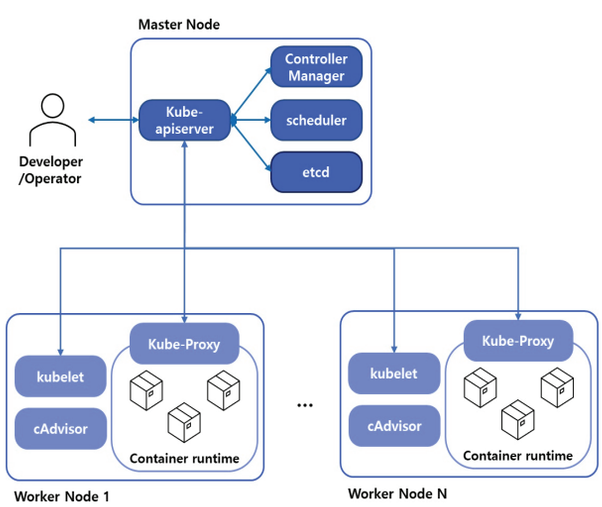

Fig. 1. The Architecture of Kubernetes

Fig. 2. Structure of Communication Between Kubelet and Containerd

스케줄러는 마스터 노드에 있는 구성요소로 아직 노드가 할당되지 않은 새로 생성된 파드를 감시하고 각 파드의 실행에 적합한 노드를 선택하는 역할을 한다. 즉, 스케줄러는 컨테이너 애플리케이션 배포가 가능한 워커 노드를 선택하고 선택한 노드에 컨테이너를 배치할 수 있도록 한다.

kubelet은 워커 노드에 있는 구성요소로 클러스터의 각 노드에서 실행되는 에이전트이다. kubelet은 쿠버네티스 API 서버와 통신하고 노드에서 실행되고 있는 컨테이너를 관리한다. Fig. 2는 쿠버네티스와 containerd 간의 통신 구조를 나타낸 것이다. kubelet이 파드와 컨테이너를 실행하고 관리하기 위해서는 각 노드에 컨테이너 런타임과 원활한 통신이 필요하다. 컨테이너 런타임은 컨테이너의 실행을 담당하는 소프트웨어로 docker[17], containerd[18] 등이 있다. CRI는 kubelet과 컨테이너 런타임 간의 통신을 위한 기본 gRPC 프로토콜을 정의한다[19]. kubelet은 gRPC를 통해 컨테이너 런타임(서버)과 연결할 때 클라이언트의 역할을 수행한다. 즉, kubelet은 containerd를 목적지로 하는 CRI-API를 호출하여 컨테이너의 생성 및 삭제와 같은 관리 작업을 수행한다.

2.2 Request와 Limit 값

쿠버네티스에서 파드를 생성할 때, 각 컨테이너에 대해 리소스(CPU, 메모리 등) 요구량(request)과 최대 사용량(limit)을 지정할 수 있다. 파드의 request 및 limit 값은 파드에 정의된 모든 컨테이너의 리소스 request와 limit 값의 합이다.

request를 지정하면, kube-scheduler는 파드가 배치될 노드를 결정하며, kubelet은 컨테이너가 리소스를 사용할 수 있도록 설정된 request 값만큼 리소스를 스케줄한다. kube-scheduler는 파드가 배치될 노드를 결정할 때 각 리소스의 실제 사용량이 아닌 노드에 배포된 파드들의 리소스 request 값의 합을 기준으로 스케줄링한다. 리소스 request 값을 지정한다는 것은 파드에 필요한 리소스의 최소량을 지정하는 것과 같다. 따라서 파드의 request 값보다 낮은 리소스를 가진 노드에는 해당 파드가 배치되지 않는다.

limit 값을 지정하면 kubelet은 실행 중인 컨테이너가 설정된 limit 값보다 많은 리소스를 사용할 수 없도록 제한한다. 컨테이너가 실행 중인 노드에 사용할 수 있는 리소스가 충분하면, 컨테이너에 지정된 request 값보다 더 많은 리소스를 사용할 수 있도록 허용된다. 그러나, 컨테이너는 설정된 limit 값보다 더 많은 리소스를 사용할 수 없다. 컨테이너가 limit 값보다 많은 메모리 할당을 시도하면 OOMKilled 메시지와 함께 해당 컨테이너가 종료된다.

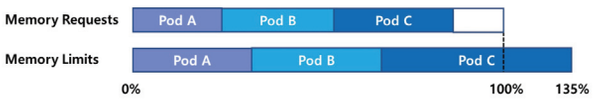

Fig. 3. Memory Resourse Oversubscription in Kubernetes

2.3 리소스 초과 사용

리소스 limit 값은 노드의 할당 가능한 리소스 양으로 제한되지 않는다. 즉, 노드에 할당된 파드의 리소스 총합은 실제 노드의 리소스 용량의 100%를 초과할 수 있다. 이러한 설정은 쿠버네티스 사용자가 모든 컨테이너는 정의된 limit 값에 근접하는 리소스를 요구하지 않을 것이라고 가정한다는 것을 의미한다.

만약 노드의 CPU 리소스 사용량이 컨테이너의 limit 값의 합보다 많이 사용한다면, 컨테이너의 실제 사용량을 request에 정의된 값까지 낮춘다. 예를 들어 limit 값이 500m (millicores), request 값이 100m인 경우, 현재 500m으로 가동되고 있는 컨테이너의 CPU 할당량을 100m로 낮춘다. 그런데도 상태가 개선되지 않으면 우선순위에 따라 운영 중인 컨테이너가 강제 종료된다.

Fig. 3은 메모리 리소스가 초과 사용된 상황을 나타낸 것이다. 파드 A, B, C의 request 값이 전체 노드 메모리의 20%, 30%, 40%로 정의되어 있고, limit 값이 25%, 45%, 65%로 정의되어 있다. 즉, 파드 A, B, C는 각각 25%, 50%, 63% 초과 사용되었다. 해당 노드에는 request 값이 노드 메모리의 10% 이하인 파드만 할당할 수 있다. 현재 파드의 limit 값의 합은 물리적 메모리에 할당할 수 있는 크기보다 35% 초과 할당되었다. 메모리의 경우 할당되어 사용 중인 메모리의 크기를 줄일 수 없으므로 노드의 물리적인 메모리보다 초과하여 메모리 할당을 요구하는 컨테이너가 있으면 우선순위에 따라 컨테이너를 강제 종료한다.

2.4 파드 재시작을 위한 CrashLoopBackOff 루틴

쿠버네티스는 파드 재시작 시 지수 백오프(Backoff) 지연시간을 적용한다. 컨테이너 재시작은 최대 300초로 제한된 지수 백오프(exponential back-off) 지연(10초, 20초, 40초, 80초...) 시간을 순차적으로 가지며 재시작을 수행한다. 만약 10분 동안 이상 없이 동작하면 지수 백오프 지연시간은 초기화된다. 이러한 루틴은 컨테이너 재시작 사유와 관계없이 동등하게 적용된다.

2.5 쿠버네티스 리소스 관리 정책의 한계점

쿠버네티스는 2.3절에 설명한 바와 같이 컨테이너의 request와 limit 값을 다르게 설정함으로써 컨테이너의 리소스 초과 사용을 지원한다. 그러나 리소스 초과 사용은 노드의 메모리 가용량 부족 시 해당 노드에 있는 일부 파드는 노드의 메모리 대역폭 확보를 위해 반드시 재시작되어야 한다는 한계점이 있다. 수평적으로 확장 가능한 상태 비저장 애플리케이

션은 애플리케이션 재시작이 발생하더라도 애플리케이션을 서비스하기 위한 여러 복사본 중 일부만 중지되기 때문에 큰 문제가 발생하지 않는다.

그러나 과학적 워크플로, 기계학습, 빅데이터 등을 활용하는 데이터 집약적이고 메모리 변동성이 큰 상태저장 애플리케이션은 재시작 발생 시 처음부터 작업을 다시 시작해야 하므로 재시작으로 인한 오버헤드가 매우 크다. 특히 임의의 컨테이너 애플리케이션에서 순간적으로(예: 0.1초) 메모리 버스트가 발생했음에도 불구하고 해당 노드에서 실행되고 있는 컨테이너 일부가 재시작되어야 한다. 이를 해결하기 위한 방법으로 스왑 메모리 사용을 고려해볼 수 있다. 스왑 메모리를 사용하면 노드의 메모리가 고갈되었을 때 일부 메모리 페이지를 디스크로 스왑하여 컨테이너가 재시작하는 것을 막을 수 있다[20]. 그러나, 쿠버네티스는 기본적으로 스왑 메모리를 지원하지 않는다. 쿠버네티스가 스왑을 지원하지 않는 이유는 스왑 지원 시 파드의 메모리 사용을 보장하는 것이 어렵기 때문이다. 이에 따라 노드의 메모리 가용량 부족 시 해당 노드에 있는 일부 파드는 노드의 메모리 대역폭 확보를 위해 반드시 재시작되어야 한다.

3. 관련 연구

컨테이너 기반 상태저장 애플리케이션의 실행 시간 개선을 위한 연구는 아직 활발하게 진행되고 있지 않다. 특히 스왑 메모리가 비활성화된 쿠버네티스 환경에서 컨테이너 워크플로에 대한 실행 시간 개선에 관한 연구는 더욱 부족한 편이다. 이는 많은 연구가 상태 비저장 애플리케이션의 성능 개선에 중점을 두었기 때문이다[21-26].

그러나 최근 쿠버네티스 환경에서 메모리 변동성이 높은 워크플로 구동에 대한 수요가 증가하고 있어 상태저장 애플리케이션의 성능 향상에 관한 연구가 점점 늘어나고 있다. Dhuraibi는 애플리케이션의 워크로드에 따라 각 컨테이너에 할당된 CPU와 메모리를 모두 확장 및 축소하는 ELASTICDOCKER를 제안하였다[27]. 이는 컨테이너의 수직 탄력성을 최초로 고려한 논문으로 알려져 있다. Rattihalli는 컨테이너 마이그레이션을 통합하여 쿠버네티스 수직 파드 확장 시스템을 중단 없이 개선하는 RUBAS를 제안하였다[28]. 그러나, 두 연구 모두 구동 중인 애플리케이션의 리소스 요구량이 시스템 리소스의 양보다 많아질 경우 어떻게 대응할 것인지에 대해서는 명확하게 정의하고 있지 않다.

한편, 성능 개선을 위해 본 연구와 유사하게 컨테이너 환경에서 리소스 초과 사용을 고려한 논문들이 있다. Nakazawa[14]는 스왑 메모리가 활성화된 Docker 환경에서 로드가 많은 컨테이너의 swappiness를 조정하여 스왑의 발생 횟수를 줄임으로써 기존 방식에 비해 약 3배의 오버커밋을 달성하였다. Chen[15]는 컨테이너 기반 탄력적 메모리 관리자인 Pufferfish를 통해 애플리케이션의 OOM 오류를 방지하고 스왑 메모리로의 빈번한 교체 작업을 수행하는 컨테이너를 일시정지함으로써 Apache Yarn 환경에서 클러스터 메모리 사용률을 2.7배, 평균 작업 런타임을 5.5배 향상시켰다. 그러나, 두 논문 모두 스왑 메모리가 비활성화된 환경에서 리소스 초과 사용을 활용하는 경우는 고려하지 않았다. 본 연구에서 제안하는 컨테이너 재시작 감소 기법은 스왑 메모리가 비활성화된 쿠버네티스 환경에서 메모리 초과 사용을 활용한다는 점, 그리고 시스템 메모리 부족으로 인해 컨테이너 재시작이 발생하기 이전에 기법이 작동한다는 점에서 기존 연구들과 차별화된다.

4. 연구 동기

4장에서는 높은 메모리 변동성을 가진 컨테이너 워크플로의 메모리 초과 사용률에 따른 실행 시간 지연 정도를 분석하고, 컨테이너의 CPU 사용량 제한을 통해 컨테이너를 충분히 일시정지할 수 있음을 보인다. 동기 부여 실험에서 사용한 컨테이너 워크플로(Fig. 4, 5)는 6장의 실험 평가에서 사용한 높은 메모리 변동성을 가진 컨테이너 워크플로와 동일하다.

4.1 메모리 초과 사용률에 따른 컨테이너 실행 시간

2장에서 설명한 메모리 리소스 초과 사용이 합리적으로 설정되면 컨테이너 워크플로의 수행시간이 리소스를 초과 사용하지 않았을 때에 비해 단축된다. 그러나 리소스 초과 사용으로 인해 컨테이너 재시작이 발생할 수 있다. 재시작이 발생한 컨테이너는 작업 과정이 저장되지 않기 때문에 처음부터 작업을 재수행해야 한다. 이는 컨테이너 워크플로의 총 실행 시간 지연을 초래한다. 워크플로의 총 실행 시간은 컨테이너 워크플로에서 실행된 첫 번째 컨테이너의 시작 시간부터 마지막으로 완료된 컨테이너의 완료 시간까지 걸린 시간을 의미한다. 그리고 컨테이너 재시작 비율은 컨테이너 워크플로의 재시작 발생 횟수를 워크플로의 컨테이너 수로 나눈 값으로 정의한다. 예를 들어 임의 워크플로의 컨테이너 수가 100개이고, 컨테이너 재시작이 20번 발생했다면 컨테이너 재시작 비율은 0.2이다.

Fig. 4는 컨테이너 메모리 초과 사용률에 따른 워크플로의 총 실행 시간과 컨테이너 재시작 비율의 변화를 나타낸 것이다. 단위 컨테이너의 limit 크기(2GB, 4GB, 8GB)와 관계없이 모두 리소스 초과 사용이 증가할수록 실행 시간이 감소하다가 다시 증가하는 것을 보여준다. 이는 메모리 초과 사용률이 증가하면서 컨테이너 재시작 발생 빈도가 높아졌기 때문이다. 리소스 초과 사용이 증가하면 더 많은 컨테이너를 동시에 처리할 수 있기 때문에 컨테이너 워크플로의 총 실행 시간은 감소한다. 그러나, 과도한 초과 사용으로 인해 컨테이너 재시작 발생률이 높아지는 시점부터 컨테이너의 전체 실행 시간이 크게 증가하는 것을 확인할 수 있다.

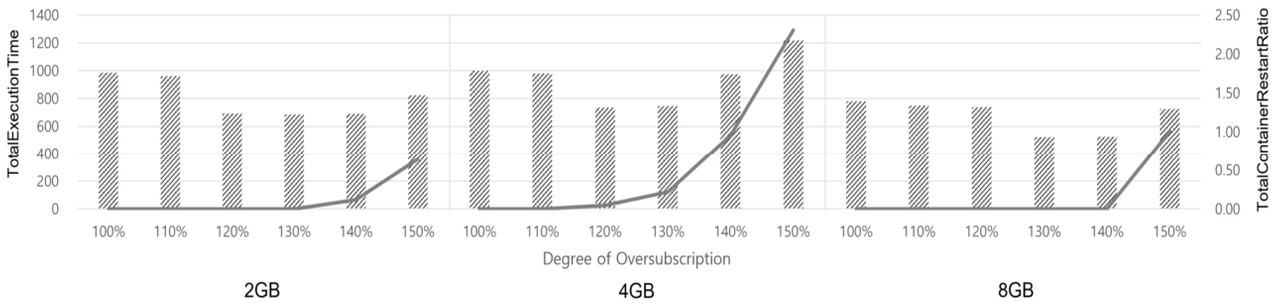

Fig. 4. Total Execution Time and Container Restart Ratio of the Entire Workflow According to the Degree of Container Memory Oversubcription

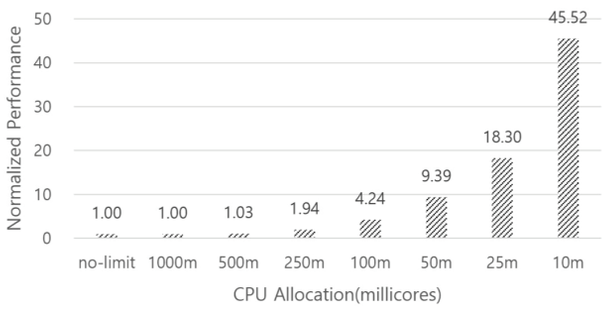

Fig. 5. Container Execution Time based on CPU Quota

4.2 CPU 사용량 제한 기반 컨테이너 일시정지 메커니즘

본 논문에서는 쿠버네티스 환경, 즉 스왑 메모리가 비활성화 되어 있는 환경에서 메모리 초과 사용 시 컨테이너의 재시작을 줄일 수 있는 기법에 대해 제안한다. 문제를 해결하기 위한 핵심 접근 방식은 메모리 할당을 요청할 가능성이 큰 컨테이너를 일시정지 하는 것이다. 컨테이너의 CPU 사용량을 크게 제한하면 컨테이너 애플리케이션이 일시정지하는 것과 유사한 효과를 낼 수 있다. Fig. 5는 CPU 할당량에 따른 컨테이너 애플리케이션 실행 시간을 정규화하여 나타낸 것이다. 1000m은 단위시간 동안 CPU 1개를 온전히 점유할 수 있음을 나타낸다. 즉, 10m은 단위시간 동안 컨테이너 1/100개만을 점유하도록 컨테이너의 CPU 사용량을 제한한 것이다. 컨테이너에 아무런 CPU 제한을 걸지 않았을 때와 실행 시간 비교 결과 CPU 할당량이 25m일 때에 약 18배, 10m일 때 약 45배 증가하였다. 이러한 결과를 통해 메모리 할당을 요청할 가능성이 큰 컨테이너의 CPU 사용량을 제한함으로써 컨테이너 재시작을 줄일 수 있는 가능성을 보여준다.

5. 모델링 및 구현

본 장은 크게 두 부분으로 구성된다. 첫 번째 부분은 제안기법의 핵심 아이디어를 설명하고, 이를 알고리즘으로 제시한다. 두 번째 부분은 모델링한 제안기법을 스왑이 비활성화 되어 있는 쿠버네티스 환경 내에서 제안된 방법의 구현을 설명하는 데 중점을 둔다.

5.1 제안된 메커니즘의 모델링

본 연구의 목적은 스왑 메모리가 비활성화 되어 있는 환경에서도 컨테이너의 재시작 감소를 통한 전체 컨테이너 워크플로의 실행 시간을 줄이는 것이다. 제안기법의 동작 과정을 요약하면 다음과 같다.

1) 메모리 사용량이 높은 노드를 선정한다. 메모리 사용량이 높은 노드를 선정하는 기준은 워크플로의 특성에 따라 달라질 수 있다.

2) 메모리 할당을 요청할 가능성이 큰 컨테이너들을 선정하고, 선정한 컨테이너들의 CPU 사용량을 크게 줄여 컨테이너를 일시정지한다. 일시정지하는 컨테이너의 수는 워크플로의 특성에 따라 조정할 수 있다.

3) 노드의 메모리 사용량이 개선되지 않으면 컨테이너를 추가로 선정하고 해당 컨테이너를 일시정지한다. 이 과정은 노드의 메모리 사용량이 개선되었다고 판단될 때까지 반복된다. 반면에 해당 노드에서 메모리 사용량이 개선되었으면 모든 컨테이너의 일시정지를 해제한다.

본 논문에서는 메모리 할당을 요청할 가능성이 큰 컨테이너를 메모리 사용량이 낮은 컨테이너로 정의한다. 메모리 사용량이 높은 컨테이너는 이미 컨테이너의 상한에 근접하는 메모리를 사용하고 있어서 추가적인 메모리 할당을 요청할 가능성이 작다. 그리고 해당 컨테이너에서 높은 메모리를 사용하는 절차가 완료된 이후 메모리 할당 해제를 요청하여 메모리 사용량이 낮아질 것으로 기대된다. 반면 메모리 사용량이 낮은 컨테이너는 이미 낮은 수준의 메모리를 사용하고 있어서 추가로 메모리 할당 해제를 요청할 가능성이 작다.

Table 1은 알고리즘에서 언급된 용어와 약어를 설명한 것이며, Algorithm 1은 메모리 초과 사용 환경에서 컨테이너의 재시작을 줄이기 위한 제안기법을 정의한다. Algorithm 1은 컨테이너가 실행되는 각 노드에 대하여 각각 실행된다. 먼저, 7번 줄에서 컨테이너 리소스 정보 수집 및 관리 수행을 위해 컨테이너 런타임과 통신하기 위한 클라이언트를 생성한다. 클라이언트를 생성하는 함수는 1-5번 줄에 정의되어 있다. 대상 컨테이너 런타임인 containerd에 unix 소켓 프로토콜을 통해 연결을 시도하고, 연결 성공 시 생성한 클라이언트를 반환한다.

Table 1. Notation Index

Notation	Explanation
M_{usage}	Current memory usage
M_{upper}	Upper Memory limit for container restriction enforcement
M_{lower}	Lower Memory limit for container de-restriction enforcement
T	Timeout_interval
C	The set of containers
C_{list}	The list of restricted containers
R	Number of containers to restrict
CPU_r	CPU Quotas to allocate to restricted containers
CPU_d	Default CPU Quotas for containers

Algorithm 1. Reducing Container Restart Technique

Algorithm 1 Reducing Container Restart Technique

```
1:  Function ConnectToContainerRuntimeService()
2:     endpoint = containerd.sock
3:     connect to endpoint via unix socket protocol
4:     client = getruntimeClient()
5:     return client
6:
7:  Function Main()
8:     client = ConnectToContainerRuntimeService()
9:     while M_usage < M_upper do
10:       wait()
11:    end while
12:    RestrictingContainerResources(client)
13:    for i = 0, |C_list| -1 do
14:       call updateContainerResources() through the client.
15:       change the CPU usage of container C[i] to CPU_d
16:       remove C[i] to C_list
17:    end for
```

그다음, 9-11번 줄에서 노드의 메모리 사용량을 일정 주기로 수집하고 이를 설정한 메모리 상한값과 비교한다. 메모리 사용량이 설정한 메모리 상한값보다 커지기 전까지 컨테이너 재시작 감소를 위한 제안기법이 실행되지 않는다.

12번 줄은 메모리 할당을 요청할 가능성이 큰 컨테이너를 선정하고, 해당 컨테이너의 CPU 사용량을 제한하기 위해 Algorithm 2에 정의된 *RestrictingContainerResources()* 함수를 수행한다.

Algorithm 2에서는 먼저, 메모리 할당을 요청할 가능성이 큰 컨테이너 선정을 위해 노드에서 실행되고 있는 컨테이너의 정보를 받아오고, 컨테이너의 메모리 사용량을 오름차순으로 정렬한다(2번 줄). 이는 정렬된 C에서 낮은 인덱스 값을 가질수록 메모리 사용량이 적음을 의미한다. 그리고 이미 CPU 사용량을 제한한 컨테이너를 다시 제한하는 것을 막기 위해 C_{list}에 등록된 컨테이너들을 C에서 삭제한다(3번 줄). 10-13줄은 메모리 사용량이 낮은 컨테이너 R개의 CPU 사용량을 제한하

Algorithm 2. Restricting Container Resources

Algorithm 2 Restricting Container Resources

```
1:  Function RestrictingContainerResources(client)
2:     Sort C by container memory usage in ascending order
3:     remove containers of C_list from C
4:     if C is empty then
5:        for i = 1, R do
6:           targetContainer = C_list[|C_list| - 1]
7:           call removeContainer() CRI-API through the client.
8:           delete targetContainer from this node.
9:           remove targetContainer to C_list
10:       end for
11:    end if
12:    for i = 0, R - 1 do
13:       call updateContainerResources() through the client.
14:       change the CPU usage of container C[i] to CPU_r.
15:       append C[i] to C_list
16:    end for
17:    round = 0
18:    while M_usage > M_lower do
19:       if round == T then
20:          RestrictingContainerResources()
21:          return
22:       end if
23:       wait()
24:       round = round + 1
25:    end while
```

고, 제한한 컨테이너를 C_{list}에 등록한다. 예를 들어 R이 2이면 메모리 사용량이 가장 낮은 2개의 컨테이너(C의 index 0, 1번)를 제한한다. CPU 사용량을 변경하기 위해서 생성한 클라이언트를 통해 컨테이너 런타임으로부터 *updateContainerResources()* CRI-API를 호출한다.

이러한 작업을 통해 노드의 메모리 사용량이 설정한 메모리 하한값보다 작아진다면, 즉, 메모리 사용량이 개선되었다고 판단되면 Algorithm 2는 종료된다. 노드의 메모리 사용량이 개선되었으므로 Algorithm 1의 13-17번 줄에서 C_{list}에 등록된 모든 컨테이너의 CPU 제한을 해제한다. 그리고, C_{list}에 등록된 모든 컨테이너를 C_{list}로부터 제거한다.

반면에 노드의 메모리 사용량이 개선되지 않았다고 판단되면 Algorithm 2의 18-25번 줄을 반복 실행하면서 노드의 메모리 사용량이 개선되기까지 기다린다. 그런데도 Timeout interval에 도달할 때까지 개선되지 않았다면 Algorithm 2를 재귀적으로 실행하여 다른 컨테이너들을 추가로 제한한다(20번 줄).

그러나 모든 컨테이너를 제한하여 노드에서 동작하는 컨테이너가 모두 일시정지 되었을 수 있다. 이를 해결하기 위해 제안기법에서는 일부 컨테이너를 희생하고 재시작하는 방식을 채택한다. 가장 마지막에 제한한 컨테이너 R개를 *removeContainer()* CRI-API를 호출하여 중지하고 재시작한다. Algorithm 2의 4-11번 줄).

5.2 제안 메커니즘 구현

본 연구는 스왑이 비활성화되어 있는 쿠버네티스 환경에서 컨테이너의 리소스를 모니터링하고, 특정 컨테이너의 리소스를 변경하는 과정을 수반한다. 이를 구현하기 위해 쿠버네티스에서 제공하는 공식 API인 client-go[21] 사용을 고려해볼 수 있다. 그러나, client-go는 쿠버네티스 API 서버와 통신하여 값을 가져오는 방식을 사용하는데, 각 노드에 존재하는 파드 또는 컨테이너의 리소스 변경을 제공하지 않는다.

따라서 본 연구는 제안기법을 구현하기 위해 컨테이너 런타임 인터페이스(CRI) API를 통해 컨테이너 리소스를 제어한다. Fig. 6은 쿠버네티스 기반 컨테이너 통신 구조를 제안기법을 구현한 모듈을 포함하여 나타낸 것이다. Fig. 2와의 차이점은 우리가 구현한 모듈들이 새롭게 추가되었다는 것이다. 우리는 기존 쿠버네티스 리소스 관리자인 kubelet과 구현 모듈들을 논리적으로 결합하여 확장 파드 및 컨테이너 리소스 관리자로 정의한다. 구현 모듈들이 kubelet과 물리적으로 통신하지는 않지만, CRI-API를 통해 각 노드의 컨테이너 리소스 관리를 수행하므로 노드의 관점에서 kubelet과 구현 모듈들은 확장된 파드 및 컨테이너 리소스 관리자로 볼 수 있다.

구현 모듈에 대하여 각 모듈은 파드 및 컨테이너 리소스 수집기, 컨테이너 리소스 변경자, 노드 메모리 정보 수집기로 정의된다. 세 모듈은 모두 go를 통해 구현되었다.

1) 파드 및 컨테이너 리소스 수집기

CRI-API를 이용하여 containerd로부터 노드에서 실행되고 있는 파드 및 컨테이너 정보를 가져온다.

2) 컨테이너 리소스 변경자

CRI-API를 통해 리눅스 cgroup에 간접 접근함으로써 기존에 설정되어 있는 컨테이너 리소스로부터(예: cpu limit, memory limit 등) 값을 변경한다.

3) 노드 메모리 정보 수집기

리눅스 /proc로부터 노드의 메모리 값을 주기적으로 가져온다.

Fig. 7은 컨테이너 재시작 감소 알고리즘의 동작 과정을 구현 모듈 호출 기반으로 나타낸 것이다. 먼저 노드 메모리 정보 수집기를 통해 메모리 사용량을 가져온 다음 설정된 메모리 상한값과 비교한다. 그런 다음 메모리 할당 요청 가능성이 큰 컨테이너 선정을 위해 파드 및 컨테이너 리소스 수집기에서 노드에서 실행되는 모든 파드 및 컨테이너 정보를 가져온다. 모든 컨테이너가 제한되었다면, 컨테이너 리소스 변경자를 실행하여 일부 컨테이너를 재시작한다. 반면에 아직 제한이 가능한 컨테이너가 남아있다면 제한할 컨테이너 선정 및 선정 컨테이너들을 제한한다. 마지막으로 메모리 사용량이 설정된 하한값보다 작아진다면 모든 컨테이너의 제한을 컨테이너 리소스 변경자를 통해 해제한다.

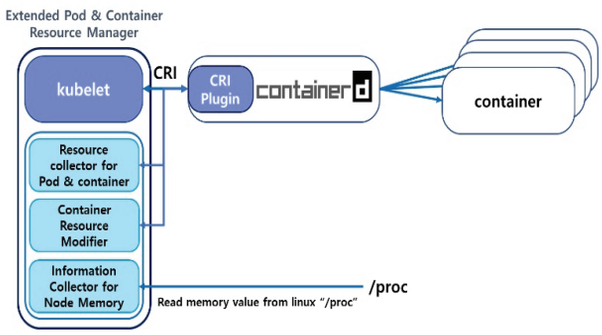

Fig. 6. Structure of Extended Pod and Container Resource Manager

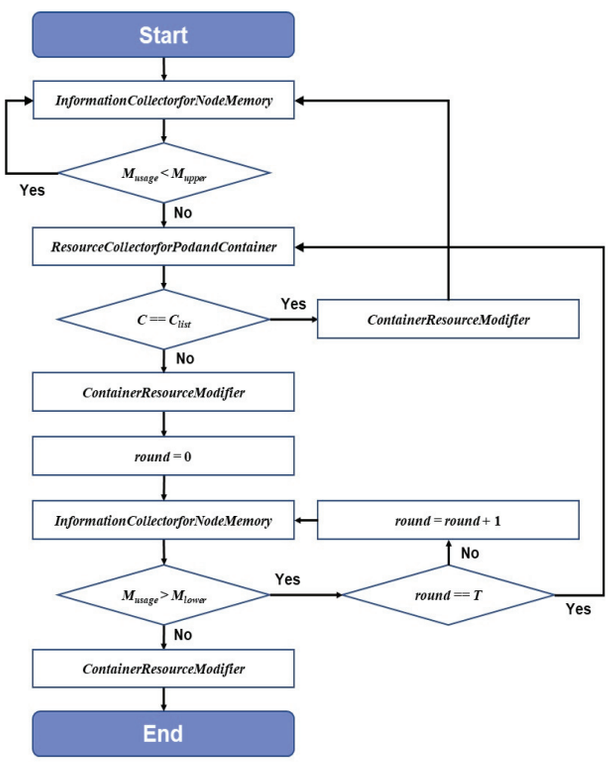

Fig. 7. The Operation Process of the Container Restart Reduction Algorithm based on the Module Call

6. 평 가

6.1 실험환경 구성

본 실험에서는 총 4개의 노드로 클러스터가 구성되며, 그 중 1개는 클러스터 제어를 수행하는 마스터 노드 역할을 한다. 나머지 3개의 노드는 워커 노드로 구성한다. 모든 노드는 Intel i9-10900K 프로세서로 구성되어 있고, 운영체제로 Ubuntu 18.04.06이 설치되며, 1Gbps 이더넷으로 상호 연결된다. 워커 노드의 메모리는 32GB, 마스터 노드의 메모리는 16GB로 구성된다.

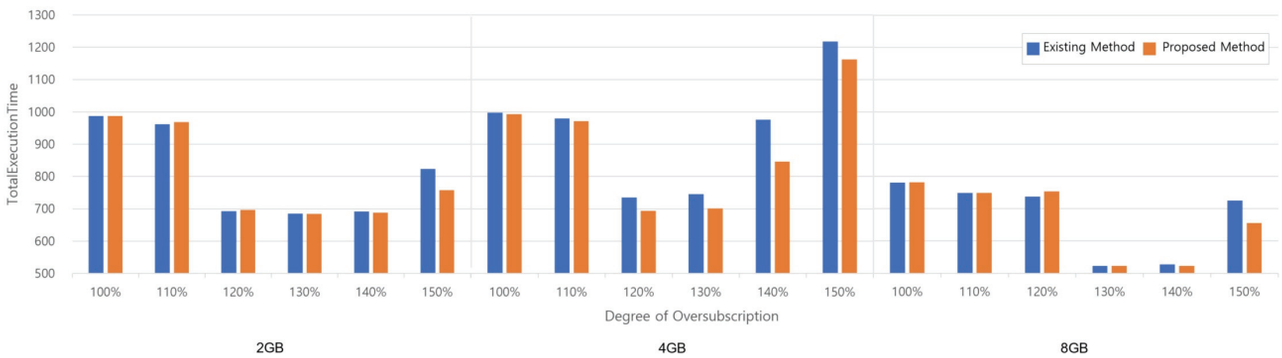

Fig. 8. Comparison of the Total Execution Time of the Entire Workflow Between the Existing Method and the Proposed Method based on the Degree of Container Memory Oversubscription

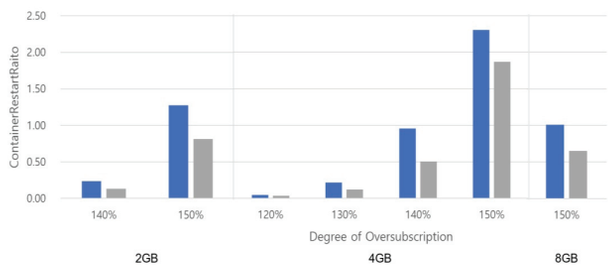

Fig. 9. Comparison of the Container Restart Ratio of the Entire Workflow Between the Existing and the Proposed Method based on the Degree of Container Memory Oversubscription

6.2 실험 수행 방법

본 연구에서는 메모리 변동성이 높은 컨테이너 애플리케이션의 메모리가 유동적으로 변화하는 상황을 시뮬레이션하기 위해 다음과 같은 조건을 가진 컨테이너 워크로드를 여러 개 생성하여 제안하는 방법을 평가한다. 메모리 변동성이 높은 컨테이너 애플리케이션은 c언어를 통해 구현된다. 이러한 컨테이너 워크로드는 다음과 같은 조건을 가진다.

1) Limit 값보다 많은 메모리가 컨테이너에 할당되지 않는다. 이는 컨테이너의 크기보다 큰 메모리가 할당되어 컨테이너가 재시작하는 것을 막기 위함이다.

2) Limit 값의 절반에 해당하는 메모리는 컨테이너 실행 중에 항상 할당되어 있다.

3) Limit/2와 limit 사이의 범위에서 메모리가 무작위로 할당되고, 할당된 메모리는 반복적으로 할당 해제된다. 메모리 할당 정책은 메모리를 순차적으로 할당하고 메모리 할당이 완료되면 할당된 메모리를 모두 할당 해제한다.

컨테이너 워크플로는 kubernetes job을 통해 배포된다[30]. 실험에 사용한 컨테이너 워크플로의 구성은 Table 2와 같다. 2GB 컨테이너 100개와 4GB 컨테이너 50개, 8GB 컨테이너 25개를 하나의 컨테이너 워크플로로 구성한다. 컨테이너 메모리를 순차적으로 할당될 때 단위 작업당 256MB, 512MB, 1024MB씩 할당된다.

Table 2. The Composition of the Container Set

Limit container Size	2GB	4GB	8GB
Number of Containers	100	50	25
Minimum Memory Usage	1GB	2GB	4GB
Memory Allocation per Unit	256MB	512MB	1024MB

Table 3. Parameter Configuration

Limit container Size	2GB	4GB	8GB
M_{upper}	94	100-140%: 89 150%: 91	88
M_{lower}	91	100-140%: 86 150%: 89	86
T	3	3	5
R	2	100~140%: 1 150%: 2	1
CPU_r	1000 (0.01CPU)		
CPU_d	20000 (2 CPU)		

Table 3은 실험에 사용된 파라미터들을 나타낸다. 해당 파라미터는 여러 개의 파라미터를 조합하여 실험한 것 중 가장 좋은 결과를 보여준 것이다. 4GB에서는 컨테이너 워크플로의 유형이 동일함에도 불구하고 초과 사용률에 따라(100-140%일 때와 150%일 때) 파라미터들이 다르게 설정되었다. 각 파라미터 집합은 실험에 대해 완전히 최적화되지 않았다. 즉, 컨테이너 재시작을 줄이기 위해 더 최적화된 파라미터 집합이 존재할 수 있다.

6.3 실험 결과 분석 평가

실험은 영역별로 동일한 횟수가 수행되었으며, 전체 실험 결과는 각 실험에 대한 평균값을 나타낸 것이다. Fig. 8과 Fig. 9는 제안기법을 적용하지 않았을 때와 제안기법을 적용했을 때 컨테이너 워크플로의 실행 시간과 컨테이너의 재시작 비율을 나타낸 것이다.

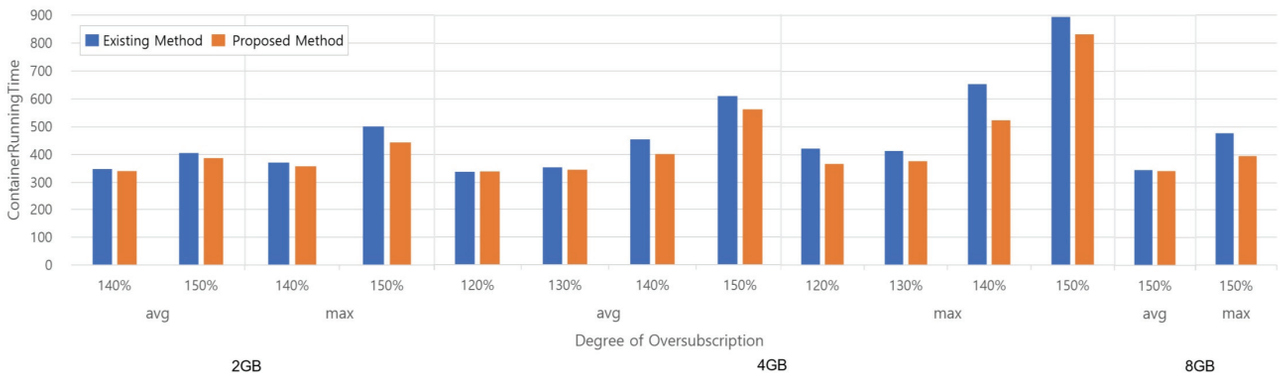

Fig. 10. Comparison of the Container Running Time Between the Existing Method and the Proposed Method based on the Degree of Container Memory Oversubscription

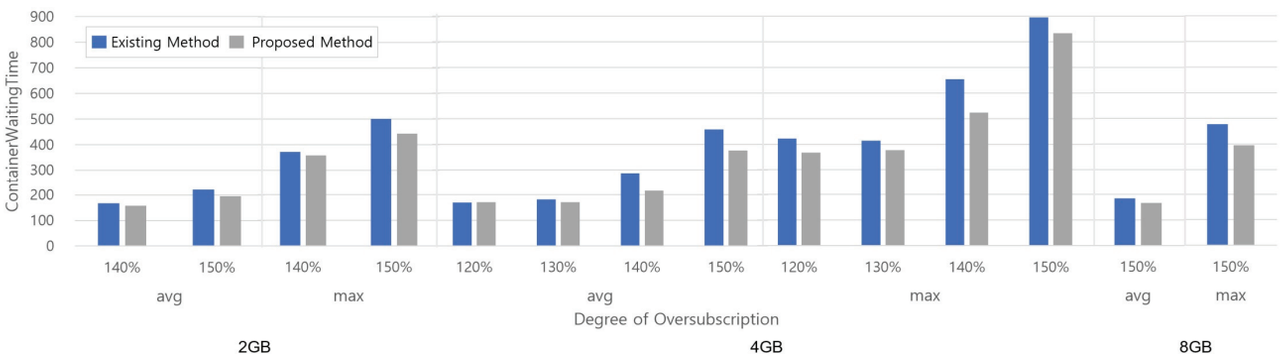

Fig. 11. Comparison of the Container Waiting Time Between the Existing Method and the Proposed Method based on the Degree of Container Memory Oversubscription

Fig. 9를 통해 메모리 초과 사용으로 인해 컨테이너 재시작이 발생한 모든 경우에 대해 제안기법을 사용했을 때가 사용하지 않았을 때와 비교하여 컨테이너 재시작 빈도가 낮아졌음을 확인할 수 있다. 2GB와 4GB에서 컨테이너 재시작 비율이 약 41% 낮아졌으며, 특히 4-140%에서는 약 58%로 크게 줄었다. 8GB에서는 약 36% 감소했다.

4GB에서 140%일 때가 150%일 때에 비해 감소 폭이 더 컸는데, 이는 150%에서 과도한 메모리 초과 사용으로 인해 제안기법의 영향이 제한적으로 나타났기 때문으로 보인다. 한편 같은 초과 사용률임에도 불구하고 컨테이너의 크기에 따라 재시작 비율이 감소한 정도가 다르게 나타나고 있다. 예를 들어 2GB-150%와 4GB-150%에 비해 8GB-150에서 감소폭이 더 낮은 모습을 보였다. 이는 컨테이너의 크기에 따라 노드의 리소스 밀집도 및 컨테이너의 메모리 할당에 대한 특성이 다르기 때문으로 추정된다.

제안기법으로 인해 컨테이너의 재시작 비율이 낮아짐으로써 컨테이너 워크플로의 실행 시간이 단축되었음을 Fig. 8에서 확인할 수 있다. 2GB-150%일 때, 컨테이너 워크플로의 실행 시간이 약 8% 감소하였고, 4GB-140%일 때 14%, 8GB-150%에서 약 10% 감소하였다.

전체적으로 Fig. 9의 컨테이너 재시작 감소율이 높을수록 컨테이너 워크플로의 실행 시간이 단축되는 정도도 같이 커지는 것을 확인할 수 있다. 예를 들어 재시작 감소율이 약 58%인4GB-140%에 비해 40% 정도인 4GB-150%의 실행 시간 감소율이 더 낮다. 반면 컨테이너 재시작이 발생하지 않은 경우 두 실행 시간의 차이가 거의 없었는데, 이는 제안기법으로 인해 컨테이너의 실행 시간에 영향을 미치는 오버헤드가 크지 않음을 보여준다.

Fig. 10과 Fig. 11은 제안기법을 적용하지 않았을 때와 제안기법을 적용했을 때 컨테이너의 실행 시간과 컨테이너 대기 시간을 비교한 것이며, 컨테이너 재시작이 발생하지 않은 사례는 비교 대상에서 제외하였다.

컨테이너 실행 시간은 컨테이너가 재시작하기 이전까지의 컨테이너 실행 시간, CrashLoopBackOff에 소요된 시간과 컨테이너가 완료되기까지 걸린 시간을 합한 것이다. 4GB-140%에서 컨테이너의 평균 실행 시간이 454초에서 402초로 약 12% 감소했는데, 이는 컨테이너의 재시작 횟수가 감소하면서 컨테이너가 재시작하기 이전까지의 컨테이너 실행 시간, CrashLoopBackOff에 소요된 시간이 기존에 비해 감소하였기 때문이다. 다른 사례에서도 같은 이유로 컨테이너들의 평균 실행 시간이 감소하였음을 확인할 수 있다. 가장 오래 실행된 컨테이너의 실행 시간은 4GB-140%일 때 653초에서 523초로

약 20% 감소하였는데, 이는 여러 번 재시작이 발생한 컨테이너의 수가 기존에 비해 감소하였기 때문으로 추정된다. 특히 8GB-150%에서는 가장 오래 실행된 컨테이너의 실행 시간이 약 18% 감소하였는데, 해당 실행 시간의 감소가 전체 컨테이너 워크플로의 실행 시간 감소로 이어졌다.

모든 컨테이너가 동시에 실행될 수 없으므로 일부 컨테이너는 실행 중인 컨테이너가 완료되기까지 실행을 대기해야 한다. Fig. 11의 컨테이너 대기시간은 컨테이너가 실행되기까지 걸리는 시간으로 정의된다. 컨테이너 대기시간도 컨테이너 실행 시간과 유사한 패턴을 보인다. 4GB-140%일 때, 컨테이너의 평균대기 시간이 약 286초에서 215초로 약 25% 감소하였으며, 가장 오래 대기한 컨테이너의 대기시간도 약 20% 감소함을 보였는데, 이는 컨테이너의 평균 실행 시간이 감소하면서 대기 중인 컨테이너가 더 빠르게 실행될 수 있기 때문으로 보인다.

논문에서 제안하는 기법을 통해 노드의 메모리 부족으로 컨테이너를 일시정지할 경우 재시작이 발생하지 않은 컨테이너의 실행 시간은 일시정지한 시간만큼 늘어날 수 있다. 그러나 컨테이너 재시작이 발생하면 컨테이너는 처음부터 다시 실행되어야 하며, 이 오버헤드는 컨테이너 일시정지에 필요한 오버헤드보다 훨씬 크다. 그러므로 제안기법 사용 시 재시작이 발생한 컨테이너의 실행 시간은 기존에 비해 크게 줄어들게 된다. 따라서 재시작이 발생하지 않은 컨테이너의 실행 시간이 증가하더라도 컨테이너의 평균 실행 시간은 Fig. 10과 같이 감소하며, 이에 따라 컨테이너 워크플로의 전체 실행 시간도 Fig. 8과 같이 감소한다.

7. 결 론

본 논문은 쿠버네티스 환경에서 메모리 초과 사용 시 컨테이너 재시작을 줄이기 위해 메모리 할당을 요청할 가능성이 큰 컨테이너를 선정하고 해당 컨테이너를 일시정지하는 기법을 제안했다. 이를 통해 메모리 초과 사용 환경에서 컨테이너의 크기와 관계없이 제안기법을 사용하지 않았을 때와 비교하여 컨테이너의 재시작 횟수를 평균 40%, 최대 58%를 줄였으며, 컨테이너 재시작 횟수 감소로 인해 컨테이너 워크플로의 총 실행 시간이 평균 7%, 최대 13% 감소하는 성과를 보였다. 실험 결과를 통해 본 논문은 스왑 메모리를 사용하지 않는 환경에서 컨테이너 재시작 발생 빈도 감소 및 컨테이너 워크플로의 전체 실행 시간을 줄일 수 있는 기회를 제시하였다.

그럼에도 불구하고 컨테이너 워크플로에서 컨테이너 간 크기의 이질성과 컨테이너 실행의 우선순위는 고려하지 못하였다. 그리고 메모리 상한, 제한할 컨테이너 수 등의 임계값이 워크로드의 특성에 따라 동적으로 변경되는 기법을 설계하고 구현하는 것이 과제로 남아있다. 향후 연구에서는 워크플로의 특성에 적응적인 파라미터 도출 기법을 모델링하고, 이를 제안 모듈과 통합하여 메모리 변동성이 높은 실제 워크플로에서 성능을 검증하고자 한다.

References

[1] A. R. Setlur, S. J. Nirmala, H. S. Singh, and S. Khoriya, “An efficient fault tolerant workflow scheduling approach using replication heuristics and checkpointing in the cloud,” *Journal of Parallel and Distributed Computing*, Vol.136, pp.14-28, 2020.

[2] C. Yang, Q. Huang, Z. Li, K. Liu, and F.Hu, “Big Data and cloud computing: innovation opportunities and challenges,” *International Journal of Digital Earth*, Vol.10, No.1, pp.13-53, 2017.

[3] A. Das, P Rad, K. K. R. Choo, B. Nouhi, J. Nish and J. Martel, “Distributed machine learning cloud teleophthalmology IoT for predicting AMD disease progression,” *Future Generation Computer Systems*, Vol.93, pp.486-498, 2019.

[4] B. Liu, J. Li, W. Lin, W. Bai, P. Li, and Q. Gao, “K-PSO: An improved PSO-based container scheduling algorithm for big data applications,” *International Journal of Network Management*, Vol.31, No.2, pp.e2092, 2021.

[5] M. Niu, B. Cheng, Y. Feng, and J. Chen, “GMTA: A geo-aware multi-agent task allocation approach for scientific workflows in container-based cloud,” *IEEE Transactions on Network and Service Management*, Vol.17, No.3, pp.1568-1581, 2020.

[6] I. Altintas et al., “Workflow-driven distributed machine learning in CHASE-CI: A cognitive hardware and software ecosystem community infrastructure.” *IEEE international parallel and distributed processing symposium workshops,* pp.865-873, 2019.

[7] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu, and W. Zhou, “A comparative study of containers and virtual machines in big data environment.” *IEEE 11th International Conference on Cloud Computing*, pp.178-185, 2018.

[8] M. Malinverno, J. Mangues-Bafalluy, C. E. Casetti, C. F. Chiasserini, M. Requena-Esteso, and J. Baranda, “An Edge-Based Framework for Enhanced Road Safety of Connected Cars,” *IEEE Access*, Vol.8, pp.58018-58031, 2020.

[9] V. K. Vavilapalli et al., “Apache hadoop yarn: Yet another resource negotiator,” *Proceedings of the 4th annual Symposium on Cloud Computing*. pp.1-16, 2013.

[10] Swarmkit [Internet], https://github.com/moby/swarmkit.

[11] Docker-swarm [Internet], https://github.com/docker-archive/classicswarm.

[12] kubernetes [Internet], https://github.com/kubernetes/kubernetes.

[13] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. Das, "Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms," *Proceedings of the ACM Symposium on Cloud Computing*. pp.153-167, 2021.

[14] R. Nakazawa, K. Ogata, S. Seelam, and T. Onodera, "Taming performance degradation of containers in the case of extreme memory overcommitment," *IEEE 10th International Conference on Cloud Computing*, pp.196-204, 2017.

[15] W. Chen, A. Pi, S. Wang, and X. Zhou, "Pufferfish: Container-driven elastic memory management for data-intensive applications," *Proceedings of the ACM Symposium on Cloud Computing*, pp.259-271, 2019.

[16] Kubernetes Components [Internet], https://kubernetes.io/docs/concepts/overview/components.

[17] Docker [Internet], https://www.docker.com.

[18] Containerd [Internet], https://containerd.io.

[19] Container Runtime Interface [Internet], https://kubernetes.io/docs/concepts/architecture/cri.

[20] D. Williams, H. Jamjoom, Y. H. Liu, H. Weatherspoon, "Overdriver: Handling memory overload in an oversubscribed cloud," *ACM SIGPLAN Notices*, Vol.46, No.7 pp.205-216, 2011.

[21] J. Dogani, R. Namvar, F. Khunjush, "Auto-scaling techniques in container-based cloud and edge/fog computing: Taxonomy and survey," *Computer Communications*, Vol.209, pp.120-150, 2023.

[22] C Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," *ACM Computing Surveys*, Vol.55, No.7, pp.1-37, 2022.

[23] L. M. Ruíz, P. P. Pueyo, J. Mateo-Fornés, J. V. Mayoral, and F. S. Tehàs, "Autoscaling pods on an on-premise kubernetes infrastructure qos-aware," *IEEE Access*, Vol.10, pp.33083-33094, 2022.

[24] F. Zhang, X. Tang, X. Li, S. U. Khan, and Z. Li, "Quantifying cloud elasticity with container-based autoscaling," *Future Generation Computer Systems*, Vol.98, pp.672-681, 2019.

[25] Y. Sfakianakis, M. Marazakis, and A. Bilas, "Skynet: Performance-driven resource management for dynamic workloads, " *IEEE 14th International Conference on Cloud Computing*, pp.527-539, 2021.

[26] N. D. Nguyen, L. A, Phan, D. H. Park, S. Kim, and T. Kim, "ElasticFog: Elastic resource provisioning in container-based fog computing," *IEEE Access*, Vol.8 pp.183879-183890, 2020.

[27] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of docker containers with elasticdocker," *IEEE 10th International Conference on Cloud Computing*, pp.472-479, 2017.

[28] G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari, "Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes," *IEEE 12th International Conference on Cloud Computing*, 2019.

[29] "clinet-go", [Internet], https://github.com/kubernetes/client-go.

[30] Kubernetes Jobs, [Internet], https://kubernetes.io/docs/concepts/workloads/controllers/job.

강 태 신

https://orcid.org/0009-0002-4833-1039
e-mail : teri98@korea.ac.kr
2022년 평택대학교 ICT융합학부(학사)
2022년 ~ 현 재 고려대학교 컴퓨터학과 석박사통합과정
관심분야 : 클라우드컴퓨팅, 시스템 자원관리, 오토스케일링, 스케줄링

유 헌 창

https://orcid.org/0000-0003-2216-595X
e-mail : yuhc@korea.ac.kr
1989년 고려대학교 컴퓨터학과(학사)
1991년 고려대학교 컴퓨터학과(석사)
1994년 고려대학교 컴퓨터학과(박사)
1998년 ~ 현 재 고려대학교 컴퓨터학과 교수
관심분야 : 클라우드컴퓨팅, 가상화, 분산시스템