컨테이너 기반 오토스케일링 환경의 성능 분석

허준, 유헌창
고려대학교 컴퓨터정보통신대학원 소프트웨어공학과
e-mail : {jheo, yuhc}@korea.ac.kr

Performance Analysis of Container based Autoscaling System

June Heo, Heonchang Yu
Dept. of Software Engineering, Korea University

요 약

컨테이너 기술은 운영체제 수준 가상화 기술 중 하나로 하드웨어 레벨 가상화 기술에 비해 인스턴스의 빠른 생성 및 종료시킬 수 있는 특성이 있다. 이러한 특성은 작업 부하에 따라 인스턴스의 수량을 동적으로 조정하는 오토스케일링 상황에서 유리하게 작용할 수 있다. 본 논문에서는 다수의 노드를 기반으로 구성된 컨테이너 기반의 오토스케일링 환경과 가상머신 기반의 오토스케일링 환경을 성능 측면에서 비교하고 컨테이너 기반 환경에서 자원 할당의 변화가 성능에 주는 영향을 측정 및 분석한다.

1. 서론

운영체제 수준의 가상화 기술 중 하나인 컨테이너 기술은 하나의 운영체제 커널을 공유하면서도 각각의 인스턴스가 별도의 네임스페이스, 루트, 환경변수, 네트워크, 컨트롤 그룹 등을 가질 수 있도록 구분하여 주는 기술이다. 이때 각각의 컨테이너는 애플리케이션의 실행에 필요한 프로그램, 환경, 라이브러리, 시스템 도구 등을 모아둔 패키지로 다른 애플리케이션 및 운영체제로부터 격리되어 독립적으로 실행되는 특성이 있다. 이러한 특성은 애플리케이션이 항상 같은 환경에서 실행될 것을 보장해주며 인스턴스의 생성, 실행 및 종료가 다른 애플리케이션 및 운영체제와 환경으로부터 격리됨을 보장해준다. 또한 하드웨어 레벨의 가상화 기술과 비교하여 운영체제의 복사 및 구동 과정이 생략되기 때문에 인스턴스의 생성 및 종료가 빠르게 완료되는 특성이 있다[1].

이러한 컨테이너의 특성은 상황에 따라 동적으로 인스턴스의 규모를 확장하거나 축소해야 할 때 유리하게 작용한다. 예를 들어 소프트웨어가 사용하는 CPU 의 점유율이 지속적으로 증가하여 하나의 인스턴스를 추가하여 단일 인스턴스에 주어지는 부하를 감소시켜야 히는 스케일 아웃의 상황이나 CPU 점유율이 감소하여 원 사용의 효율성을 높이기 위해 일부 인스턴스를 종료시키는 스케일 인의 상황이 있을 수 있다. 특히 전체 환경에 제공되는 자원을 여러 개의 애플리케이션 및 시스템이 공유하며 사용하는 클라우드 환경에서는 이러한 동적인 인스턴스의 확장과 축소를 적극적으로 활용하는 구조의 시스템이 효율성을 높여줄 수 있다.

오토스케일링은 이러한 인스턴스의 확장 및 축소를 각각의 인스턴스에서 수집된 자원사용량을 이용하여 자동으로 수행하는 기술을 말한다. 오토스케일링 과정에서 관리 프로그램은 CPU, 메모리 사용량, 네트워크 사용량 등을 측정한 뒤 정의된 기준치를 기반으로 분석하여 적절한 인스턴스의 수량을 결정하고 현재 인스턴스의 수량과 차이가 있는 경우 인스턴스를 확장하거나 축소하는 작업을 반복적으로 수행한다. 이러한 오토스케일링은 시스템의 예상치 못한 부하의 증가나 감소에 유연하게 대응하여 시스템의 가용성 및 자원사용의 효율성을 높여주는 장점이 있다. 특히 컨테이너 기반의 운영체제 수준 가상화 기술은 하드웨어 가상화 기술과 비교하여 자원을 세분화하여 할당할 수 있는 점, 인스턴스의 확장 및 축소에 필요한 비용이 적다는 점 등이 장점으로 작용하여 오토스케일링 전략을 적극적으로 적용할 수 있게 해준다.

본 논문에서는 하드웨어 가상화 및 운영체제 수준 가상화의 성능을 비교하고 컨테이너의 자원 할당 전략에 대해 분석한다. 이를 위해 대표적인 클라우드 기반 가상환경인 아마존 웹 서비스와 분산 환경 컨테이너 관리 도구인 쿠버네티스를 이용하여 오토스케일링 상황에서의 하드웨어 가상화 및 운영체제 수준 가상화의 성능을 비교 측정한다. 또한, 컨테이너 기반 환경의 오토스케일링 상황에서 자원 할당의 변화가 전체 수행 성능에 어떤 영향을 주는지를 측정 및 분석한다.

2. 배경

2.1 컨테이너 자원 관리

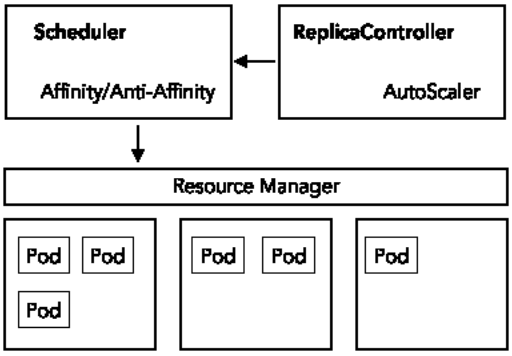
컨테이너는 실행되는 호스트 및 다른 프로세스로부터 네트워크, 사용자, 저장소, 컨트롤 그룹 등을 격리할 수 있으나 CPU, 메모리, 네트워크 대역폭 등의 자

원은 같은 호스트에서 실행되는 다른 프로세스 및 컨테이너와 같이 사용한다. 이때 특정 컨테이너가 호스트의 모든 자원을 점유하는 것을 방지하기 위하여 CPU, 메모리, 디스크, 네트워크 I/O 등을 제한한다[2].

단일 노드가 아닌 분산된 호스트에서 다수의 컨테이너가 실행되는 경우 컨테이너의 배치와 자원 활용에 관하여 노드가 제공하는 자원 및 현재 가용한 자원의 파악, 컨테이너의 배치 위치 결정, 환경 변화에 따른 동적인 계획 수정과 같은 고려사항이 필요하다[3]. 이와 같은 자원 활용의 고려는 컨테이너의 빠른 배치 및 해제를 가능하게 해주며, 컨테이너의 복제 및 축소가 빈번하게 발생하는 스케일-아웃 및 스케일-인 상황의 효율성을 높여준다.

2.2 쿠버네티스의 자원 관리

분산 컨테이너 관리 시스템인 쿠버네티스의 경우 각 컨테이너의 배치를 결정할 때 CPU 및 메모리 자원 사용량만을 고려한다. 각 컨테이너는 기동에 필요한 최소 자원 및 최대 자원을 명시할 수 있으며 이때 자원의 단위는 millicore (싱글 코어가 제공하는 자원의 1/1000) 및 바이트 단위로 명시할 수 있다[4].



(그림 1) 쿠버네티스 스케줄러의 동작

(그림 1)은 쿠버네티스의 스케줄러가 오토스케일러의 결정에 따라 주어진 노드에 컨테이너를 배치하는 구조를 보여준다. 이때 스케줄러는 가용한 노드에 남은 자원, 우선순위, 이미 배치된 컨테이너 등을 고려하여 점수를 매기고 가장 점수가 높은 노드에 컨테이너를 배치한다. 만약 가용한 자원의 부족 등으로 컨테이너가 배치될 수 있는 노드가 하나도 없는 경우 해당 컨테이너는 대기 상태로 자원이 확보될 때까지 대기하게 된다.

3. 실험

3.1 컨테이너 오토스케일링

오토스케일링 기법은 애플리케이션의 자원 사용량, 네트워크 요청 등의 변동에 따라 애플리케이션을 수평적 혹은 수직적으로 확장하거나 축소하는 방법이다. 오토스케일링의 과정을 일반화하면 (1) 모니터링 (Monitor) (2) 분석 (Analyze) (3) 스케줄링 (Plan) (4) 실행 (Execute)의 4 단계로 나눌 수 있다[6].

쿠버네티스는 자원 사용량에 따라 컨테이너를 수평적으로 확장하거나 축소하기 위하여 HPA(Horizontal Pod Autoscaler)라는 컨트롤러를 구현하여 사용한다. HPA 가 쿠버네티스에 배치된 컨테이너를 사용량에 맞게 오토스케일링 하는 과정은 다음과 같다[7].

(1) 모니터링 (Monitor)

컨테이너 내부의 자원 사용량 모니터링을 위한 프로세스를 설치하여 컨테이너 내부의 CPU 및 메모리 사용량을 계속 측정하여 쿠버네티스의 마스터 노드에 보고한다. 또, 컨테이너가 기동 되는 노드 자체의 자원 사용량에 대한 모니터링도 가능한데 이 사용량은 쿠버네티스 노드가 클라우드 환경에서 가상머신으로 구동되는 경우 해당 노드도 오토스케일링 하기 위한 용도로 사용될 수 있다. 다만 본 논문에서는 컨테이너가 실행되는 워커 노드의 오토스케일링 시나리오는 제외하였다.

(2) 분석 (Analyze)

컨테이너의 수는 기존에 배치된 컨테이너들의 CPU 점유율의 합을 사전에 설정된 목표 CPU 점유율로 나눈 값을 구한 뒤 정수가 되도록 올림 하여 결정한다. 예를 들어 기존에 배치된 컨테이너 3 개의 CPU 점유율이 각각 70%, 40%, 80%이고 목표 CPU 점유율이 50% 라면 (70 + 40 + 80) / 50 은 3.8 이 되므로 오토스케일러는 목표 컨테이너 수량을 4 개로 재조정한다. 이러한 분석 과정에서 CPU 점유율의 일시적인 급증이나 급감으로 인한 컨테이너의 생성과 파괴에 따른 비용증가를 막기 위하여 각각의 스케일링 실행 후 일정한 유휴 시간을 두거나 목표 CPU 점유율에 일정한 비율의 허용치를 부여할 수 있다.

(3) 스케줄링 (Plan)

분석 단계에서 결정된 컨테이너 수량에 맞추어 컨테이너를 확장하거나 축소한다. 컨테이너가 확장되는 경우 2.2 절에서 언급된 자원관리 기준에 따라 노드를 선택하여 새로운 컨테이너를 배치한다. 컨테이너가 축소되는 경우 동작 중인 컨테이너 중 하나를 기준 없이 무작위로 선택하여 삭제한다

(4) 실행 (Execute)

컨테이너의 수량 및 배치 / 제거 전략이 확정된 경우 스케줄러는 각 노드와 통신하여 컨테이너 생성 / 제거 명령을 내린다[8]. 이때 컨테이너 생성과 제거가 성공적으로 완료된 경우 오토스케일링 프로세스를 종료하고 측정 단계로 돌아가서 전체 과정을 반복한다. 컨테이너 생성 혹은 제거에 실패한 경우 상태를 유지하고 실행 과정을 성공할 때까지 반복한다.

3.2 쿠버네티스 기반의 오토스케일링 측정

3.2.1 실험 환경

이 실험을 통해 쿠버네티스를 기반으로 한 컨테이너의 오토스케일링과 클라우드를 기반으로 한 가상머신의 오토스케일링의 효율성을 비교해보고, 컨테이너의 자원할당에 따른 오토스케일링 성능의 영향을 판단한다. 해당 실험은 가상머신에서 애플리케이션을

직접 구동하는 경우와 가상머신에 쿠버네티스 클러스터를 구성하여 컨테이너 기반으로 애플리케이션을 구동하는 경우로 나누어서 애플리케이션의 수행 성능을 측정하였다.

실험은 아마존 웹 서비스에서 제공하는 EC2 인스턴스 상에서 이루어졌으며, 모든 노드는 2.40GHz 2 개 코어 및 4GB 메모리로 이루어진 인스턴스로 구성되었다. 실험에 사용된 OS 는 데비안 리눅스 4.4 이며 쿠버네티스 1.9, 도커 18.03 버전을 사용하여 컨테이너를 관리하였다.

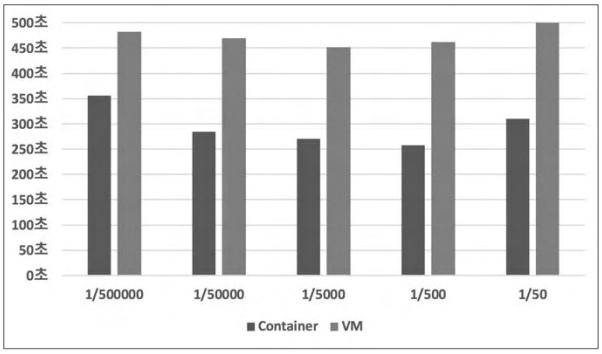
모든 실험은 가상머신 및 컨테이너 환경에 소수 계산을 수행하는 애플리케이션을 배포한 뒤, 별도로 구성된 측정 노드에서 총 5 억 개의 소수 계산을 일정 단위로 분할하여 애플리케이션에 요청하는 환경을 구성하였다. 가상머신 및 컨테이너가 스케일 아웃 되어 다수의 인스턴스가 실행될 경우 해당 계산은 각 인스턴스에 분산하여 수행하도록 구성하였다. 가상머신의 경우 1 개의 인스턴스에서 시작하여 최대 5 개까지 확장되도록 하였으며, 컨테이너의 경우 2 코어 컨테이너 기준으로 1 개의 인스턴스에서 5 개의 인스턴스까지 확장하도록 하였다. 컨테이너에 할당된 자원이 2 코어 미만인 경우 노드의 자원을 최대한 활용할 수 있도록 최대치를 조정하였으며, 컨테이너에 할당된 자원별 오토스케일링의 범위는 <표 1>과 같다.

<표 1> 할당자원 별 오토스케일링 범위

	최소 (시작)	최대
2 Core	1 컨테이너	5 컨테이너
1 Core	1 컨테이너	10 컨테이너
0.5 Core	1 컨테이너	20 컨테이너
0.1 Core	1 컨테이너	100 컨테이너

모든 오토스케일링은 CPU 점유율을 기준으로 사용하였으며 확장 및 축소의 기준은 50% 이상의 평균 CPU 점유율이다.

3.2.2 가상머신과 컨테이너 환경의 성능 비교

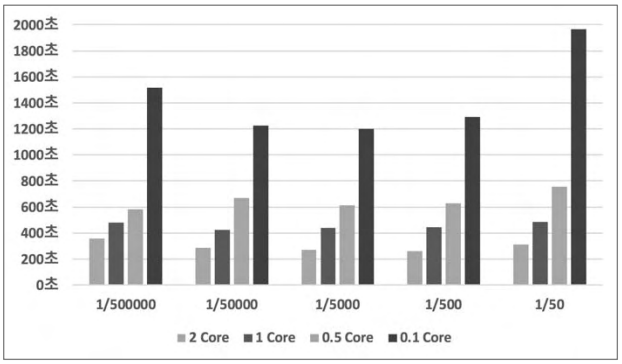


(그림 2) 가상머신과 컨테이너의 연산 수행 시간

(그림 2)는 가상머신과 컨테이너 환경에서의 오토스케일러를 이용하여 인스턴스를 모두 1 대에서 5 대까지 확장할 수 있도록 설정한 뒤 연산을 수행했을 때의 결과를 측정한 것이다. 결과를 살펴보면 가상머신에 비해 컨테이너가 동일한 연산을 빠르게 완료한 것을 확인할 수 있다. 이는 연산이 지속되면서 CPU 사용량의 증가에 따라 인스턴스가 확장될 때, 새로 추가된 인스턴스가 연산작업에 합류하기까지의 시간이 가상머신은 40 초, 컨테이너는 8 초가량으로 컨테이너가 더 빠르게 스케일 아웃을 완료하고 연산작업에 합류한 것을 원인으로 볼 수 있다. 이는 지속적으로 많은 량의 연산을 처리하여 부하를 분산해야 할 때 컨테이너 환경이 가상머신에 비하여 빠르게 인스턴스를 확장할 수 있음을 나타낸다.

총 5 억 개의 소수 계산을 1,000 개 단위 (1/500000)에서 10,000,000 개 단위(1/50)까지 변경하며 작업을 분산했을 때 가상머신과 컨테이너 환경 모두 전체적으로 유의미한 속도 차이를 보이지는 않았으나, 작업이 지나치게 크게 분할될 경우 최대 5 개의 인스턴스로 스케일 아웃이 완료되기 전에 작업이 분배되어 일부 노드가 계산 작업을 수행하지 않는 결과가 발생하였다. 이는 부하가 일시적으로 발생하는 경우 작업을 분할하는 단위가 지나치게 크면 전체 노드가 작업에 합류하지 못할 수 있음을 나타내며 충분히 작은 단위로 분할하여야 효율성을 증가시킬 수 있음을 의미한다. 다만, 부하가 장기간 지속되어 작업이 전체 노드에 할당될 것이 분명한 경우 연산의 분할단위가 전체 성능과 효율성에 미치는 영향이 점점 줄어들 수 있음을 추론할 수 있다.

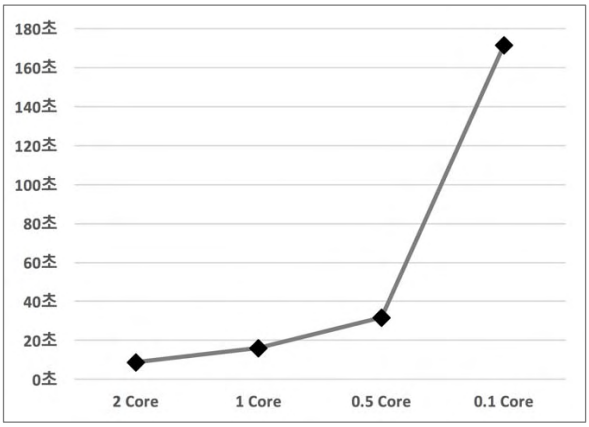
3.2.3 컨테이너 자원 할당에 따른 수행 결과 비교



(그림 3) 컨테이너 할당 자원별 연산 수행 시간

(그림 3)은 컨테이너 환경에서 단일 컨테이너에 자원을 0.1 코어, 0.5 코어, 1 코어, 2 코어 단위로 할당하였을 때 전체 연산 수행이 완료된 시간을 비교한 것이다.

실험 결과에서 컨테이너에 최대한 많은 CPU 자원을 할당할수록 전체 연산 시간이 줄어드는 것을 확인할 수 있다. 이때, 0.1 코어로 자원을 할당한 경우 연산 수행속도가 현저히 느려지는 것을 확인할 수 있다. 이러한 현상의 원인을 분석하자면, 첫 번째로 컨테이너가 생성되어 연산작업에 합류할 때 자원이 충분하지 않은 경우 매우 오랜 시간이 걸리는 문제를 생각할 수 있다.



(그림 4) 컨테이너에 할당된 자원에 따른 기동시간

(그림 4)는 컨테이너에 할당된 자원별로 컨테이너 및 애플리케이션이 기동 되어 연산작업에 합류하기까지의 시간을 보여준다. 2 코어 자원이 할당된 컨테이너의 경우 오토스케일링에 의해 스케일 아웃이 결정된 이후 컨테이너가 생성되어 연산에 합류하기까지 9초 이내의 시간이 걸린 반면 0.1 코어 컨테이너는 172 초가량의 시간에 걸리는 것을 확인할 수 있다. 두 번째 원인으로는 쿠버네티스가 측정 노이즈를 경감시키기 위해 스케일 아웃이 발생한 후, 다음 스케일 아웃까지 일정한 딜레이를 부여하는 점에서 찾을 수 있는데, 총 100 개의 컨테이너가 확장되어야 하는 0.1 코어 컨테이너의 경우 이러한 딜레이로 인하여 실험이 완료될 때까지 모든 컨테이너가 생성되지 않아서 전체 노드의 성능을 활용할 수 없었다. 마지막으로 애플리케이션의 유지를 위한 오버헤드를 저사양 컨테이너가 감당하지 못하는 문제를 생각할 수 있다. 각 컨테이너에서 실행된 애플리케이션은 네트워크 요청의 처리, 로그 기록, 모니터링, 작업 할당 대기, 스레드 관리 등을 위하여 연산 작업 외에도 일정한 자원을 사용하는데 0.1 코어 컨테이너의 경우 해당 작업만으로도 할당된 CPU 자원의 절반 이상을 사용하는 것을 확인할 수 있었다. 이는 컨테이너에 자원이 적게 할당될수록 연산 작업 외의 오버헤드의 비중이 커짐으로써 스케일 아웃이 되더라도 전체 연산 능력이 충분히 향상되지 못함을 알려준다.

4. 결론 및 향후연구

본 논문은 가상머신 기반의 환경과 컨테이너 기반의 환경에서 오토스케일링 될 수 있는 환경을 구성한 뒤 동일한 연산 작업을 수행하여 컨테이너 환경의 스케일 아웃 효율이 전반적으로 더 좋음을 측정하였다. 또, 컨테이너에 할당하는 자원이 충분하지 못할 경우 전체 작업 수행속도가 매우 느려지며 높은 자원을 할당할수록 스케일 아웃 효율성이 좋아져서 연산 속도가 빨라짐을 알 수 있었다.

따라서 충분히 많은 노드를 확보하고 다양한 시스템을 함께 실행할 수 있는 클라우드 환경에서는 컨테이너 기반의 오토스케일링 환경을 구축하여 적용하는 것이 부하의 급격한 증가나 감소에 유연하게 대응할 수 있음을 보여주었고, 각 컨테이너에 할당하는 자원 역시 최대한 높게 할당하는 쪽이 성능 향상에 유리함을 알 수 있었다.

다만 컨테이너에 자원을 높게 할당할 경우 컨테이너의 최소 및 최대 수량의 범위가 낮은 자원이 할당된 경우에 비하여 좁기 때문에 상대적으로 오토스케일링의 유연성은 떨어진다고 볼 수 있다. 따라서 컨테이너 기반의 오토스케일링 환경에서는 작업의 유형, 부하가 발생하는 주기 및 지속 시간, 할당할 수 있는 최대 자원 및 같은 노드를 쓰는 다른 시스템과의 자원 분배 등을 고려하여 적절한 자원 할당량을 결정하는 것이 더 중요하다고 할 수 있다.

측정된 실험 결과를 바탕으로 향후 CPU 점유율 기반이 아닌 네트워크 요청의 숫자 및 크기, 애플리케이션의 내부 상태, 메모리 및 저장소의 점유율 등을 기반으로 더 다양한 측정치를 고려하여 상황에 따라 오토스케일링의 효율성을 높이는 방안을 찾을 수 있다. 또한 작업의 유형에 따라 적절한 컨테이너 자원 할당량을 결정할 수 있도록 시스템의 규모, 대기 상태의 자원 사용량, 전체 노드의 규모 등을 고려하여 가장 효율적으로 자원을 할당할 수 있는 전략을 결정하는 방법을 추가로 도출할 계획이다.

참고문헌

[1] FELTER, Wes, et al. An updated performance comparison of virtual machines and linux containers. In: Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On. IEEE, 2015. p. 171-172.
[2] the series about LXD 2.0 : Resource Control . https://stgraber.org/2016/03/26/lxd-2-0-resource-control-412/
[3] ABDELBAKY, Moustafa, et al. Docker containers across multiple clouds and data centers. In: Utility and Cloud Computing (UCC), 2015 IEEE/ACM 8th International Conference on. IEEE, 2015. p. 368-371.
[4] Kubenetes Concepts : Managing Compute Resources Container. https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/
[5] Scheduler Algorithm in Kubernetes. https://github.com/kubernetes/kubernetes/blob/release-1.1/docs/devel/scheduler_algorithm.md
[6] LORIDO-BOTRAN, Tania; MIGUEL-ALONSO, Jose; LOZANO, Jose A. A review of auto-scaling techniques for elastic applications in cloud environments. Journal of grid computing, 2014, 12.4: 559-592.
[7] Kubernetes Tasks : Horizontal Pod Autoscaler. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
[8] Kubernetes Concepts : Master-Node communication. https://kubernetes.io/docs/concepts/architecture/master-node-communication/