도커 환경에서 I/O 성능 비율 향상을 위한 페이지 캐시 관리 기법

Page Cache Management Scheme for Improving I/O Proportionality in the Docker Environment

요 약

도커 환경에서는 시스템 리소스를 관리하기 위해 Cgroups를 사용한다. Cgroups는 blkio 서브시스템을 활용하여 컨테이너별 I/O 성능 비율을 조절하지만 해당 서브시스템은 페이지 캐시를 고려하지 않아 컨테이너의 I/O 성능 비율이 보장되지 않는다. 본 논문에서는 이를 해결하기 위해 blkio 서브시스템의 blkio_weight를 고려한 새로운 페이지 캐시 관리 기법을 제안한다. 제안 기법은 blkio_weight를 고려하여 페이지를 회수함으로써, 각 컨테이너에서 사용하는 페이지 개수를 blkio_weight의 비율에 맞게 유지한다. 실험 결과, 제안 기법은 기존 기법보다 22.0% 향상된 I/O 성능 비율을 보이며 제안 기법에서 발생하는 오버헤드는 최대 5.9% 이내 인 것을 확인하였다.

1. 서 론

최근 클라우드 서비스를 제공하는 다양한 서버와 데이터 센터에서는 도커를 가상화 프레임워크로 채용하고 있다. 도커는 리눅스 컨테이너 가상화 기술을 활용하여 어플리케이션을 격리된 공간에서 실행시키는 오픈소스 플랫폼이다. 하드웨어부터 OS까지 추가적인 가상화 계층을 생성하는 하이퍼바이저 방식의 가상화와는 달리, 도커는 어플리케이션 계층만 가상화하여 호스트의 리소스를 공유하는 독립된 공간을 제공한다. 도커는 컨테이너마다 CPU, 메모리, 블록 I/O와 같은 공유되는 호스트의 시스템 리소스 사용을 Cgroups으로 제어한다. 이 중, 블록 I/O의 경우, blkio라는 Cgroups 서브시스템에서 관리하며, 파라미터인 blkio_weight를 통해 컨테이너별 블록 I/O의 성능 비율을 지정할 수 있다. 하지만 blkio는 블록 I/O를 처리하는 과정에서 페이지 캐시의 존재를 고려하지 않는다. 그렇기에 데이터가 블록 계층을 거치지 않고 페이지 캐시로부터 바로 참조가 되는 경우, blkio에서 설정한 각 컨테이너의 blkio_weight를 적용하지 못하고, 결국 정해진 I/O 성능 비율을 보장하지 못한다.

이에 본 논문에서는 Cgroups을 통해 설정된 blkio_weight를 고려하는 페이지 캐시 관리 기법을 제안한다. 페이지가 할당 될 때마다 각 컨테이너별 페이지 할당 수를 기록하고 회수 작업이 진행될 때 각 컨테이너별 페이지 개수와 blkio_weight의 비율을 고려하여 페이지를 회수함으로써 컨테이너마다 설정된 blkio_weight들의 비율과 유사한 수치만큼 페이지 캐시를 각 컨테이너별로 할당된다. 그러므로 상대적으로 높은 blkio_weight의 컨테이너는 더 많은 페이지 캐시를 할당받게 되고 그 결과 blkio_weight가 낮은 컨테이너 대비 더 높은 처리량을 가진다. 실험 결과, 제안 기법은 기존 페이지 캐시 관리 기법보다 최대 6%이내의 오버헤드로 22.0% 향상된 I/O 성능 비율을 제공한다.

2. 배 경

2.1. 도커 컨테이너와 Cgroups

도커는 컨테이너를 활용한 OS 레벨 가상화 기술을 이용하여 사용자들에게 가상 컴퓨팅 환경을 제공한다. 각 컨테이너는 서로 격리된 환경에서 구동 되지만 모두 같은 호스트의 OS와 커널 리소스를 공유한다. 이 리소스를 효과적으로 제어 및 관리하기 위해 도커는 Cgroups라는 리소스 관리 기법을 활용한다[6].

Cgroups는 CPU, 블록 I/O 등의 리소스들을 cpu 서브시스템, blkio 서브시스템 등으로 구분하여 관리 한다[3]. 이 중 blkio 서브시스템은 blkio_weight를 지정하여 블록 I/O 성능의 상대적인 비율을 설정할 수 있다. 이 blkio_weight 값은 리눅스의 블록 계층에서 CFQ I/O 스케줄러가 I/O 요청을 처리할 때 사용된다. 하지만 blkio 서브시스템은 페이지 캐시의 존재를 고려하지 않기에 실제 I/O 처리 과정에서 페이지 캐시를 사용하는 경우, Cgroups을 통해 설정된 blkio_weight의 비율이 반영되지 않는다. 만약 I/O 요청이 접근하려는 데이터가 페이지 캐시에 저장되어 있는 경우엔 디스크로부터 데이터를 읽어오는 과정을 건너뛰고 페이지 캐시에서 바로 데이터를 읽어온다. 이 때 I/O 요청이 I/O 스케줄러를 거치지 않아 blkio_weight가 반영되지 않는다. 따라서 페이지 캐시로의 접근은 결국 Cgroups로 설정된 I/O 성능의 비율을 보장하지 못하게 한다.

그림 1은 페이지 캐시로 인한 비율 저하를 확인하기 위해 FIO 벤치마크를 통해 10MB 파일 100개를 blkio_weight가 다른 네 개의 컨테이너에서 Buffered I/O와 Direct I/O

로 파일을 생성한 후 순차적 읽기를 했을 때의 처리량을 blkio_weight 100에 표준화 하여 나타낸 실험 결과이다. Buffered I/O는 페이지 캐시를 활용하는 반면 Direct I/O는 페이지 캐시를 건너뛰고 블록 계층을 거치며 I/O를 발생시킨다. 그러므로 Direct I/O는 설정된 blkio_weight의 상대적인 비율과 유사한 성능 결과를 보인다. 그에 반해, Buffered I/O는 페이지 캐시로부터 데이터를 바로 참조하기에 처리량의 비율 차이가 없다.

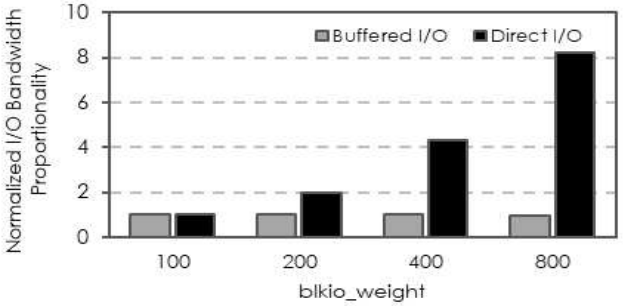

그림 1 Buffered I/O와 Direct I/O의 비율 비교

2.2. 기존 페이지 회수 기법

페이지 캐시는 자주 참조되는 페이지들을 DRAM에 할당 시켜 상대적으로 접근 속도가 느린 디스크로의 I/O를 줄여주는 장점이 있다[4]. 현재 리눅스 커널은 두 개의 LRU 리스트로 페이지 캐시의 페이지들을 관리한다. 새로 할당된 페이지는 inactive list의 헤드에 삽입하며 자주 접근되는 페이지는 inactive list에서 active list의 헤드(head)로 삽입된다. 따라서 자주 접근 되는 페이지일수록 active list의 헤드에 가깝게 위치하며 상대적으로 접근 횟수가 적거나 한번만 접근된 페이지들은 inactive list의 테일(tail)에 위치하게 된다. 리눅스에서는 메모리 공간이 부족해질 때 여유 페이지 확보를 위해 페이지 회수 작업을 진행된다. active list의 크기가 inactive list의 크기보다 큰 경우, active list의 테일에서부터 페이지들을 active list의 헤드로 옮긴다. inactive list에서의 페이지 회수는 맵핑이 사라진 페이지와 클린 페이지를 우선적으로 회수한다[5]. 이렇듯, 페이지 캐시는 자신만의 기준으로 페이지 회수를 진행하여 다른 계층의 I/O 파라미터 값인 blkio_weight는 고려되지 않는다.

3. 제안 기법

현재 리눅스 페이지 캐시 관리 기법은 LRU 정책에 따른 두 개의 리스트로만 페이지를 관리하여 프로세스별로 설정된 blkio_weight의 I/O 비율이 반영되지 않는다. 따라서 위 문제들을 해결하기 위해 본 논문에서는 blkio_weight를 고려하는 페이지 캐시 관리기법을 제안한다. 본 기법을 효과적으로 설명하기 위해 두 개의 LRU 리스트를 한 개의 LRU 리스트로 표현하며, 본 논문에서는 다중의 컨테이너가 하나의 페이지 캐시를 공유하는 경우는 배제한다. 또한, 본 기법의 목적은 컨테이너를 기준으로 페이지 캐시를 관리하기 위함이지만 컨테이너마다 blkio_weight가 다르다 가정하여 본 논문에서는 blkio_weight별로 페이지 캐시를 관리한다.

3.1. 페이지 할당

본 기법에서는 페이지 구조체에 'blkweight' 변수를 추가로 생성하여 페이지를 할당할 때, 해당 페이지를 요청한 컨테이너의 blkio_weight를 저장한다. 추가적으로 컨테이너 blkio_weight별 페이지 캐시 개수를 관리하기 위해 각 blkio_weight별 nr_pages_#라는 변수를 활용한다. 페이지 캐시에서 새로운 페이지를 할당할 때 해당 컨테이너의 blkio_weight를 확인하여 그 blkio_weight에 맞는 nr_page_# 변수를 새로 할당된 페이지의 개수만큼 증가시킨다. 이를 통해 제안 기법에서는 지속적으로 blkio_weight별 페이지 개수를 측정한다. 예를 들어, 그림 2(a)와 같이 blkio_weight 400의 컨테이너가 페이지 할당을 요청하게 되면 nr_pages_400을 atomic하게 증가 시킨다.

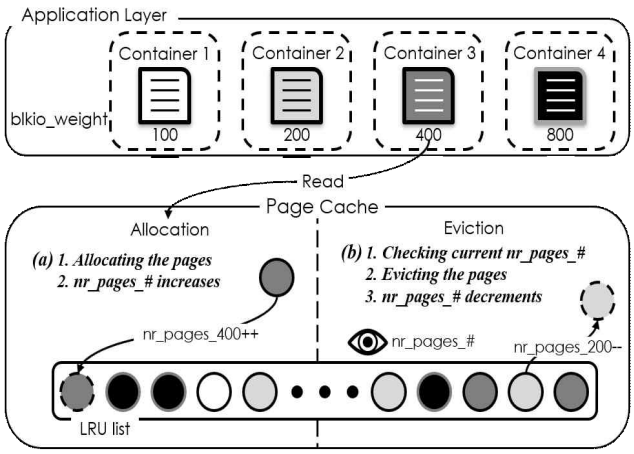

그림 2 제안 기법의 페이지 할당 및 회수 과정

3.2. 페이지 회수

본 기법에서는 각 컨테이너가 사용하는 페이지 캐시의 비율이 컨테이너의 blkio_weight의 비율만큼 유지되는 것에 중점을 둔다. nr_pages_# 변수를 이용하여 blkio_weight별로 페이지가 얼마나 할당 되었는지를 판별한 후, 회수 작업 중에 해당 blkio_weight의 비율보다 상대적으로 많이 할당되어 있는 blkio_weight의 페이지들을 회수한다. 예를 들어 그림 2와 같이 blkio_weight가 각각 100, 200, 400, 800으로 설정된 컨테이너들이 구동되는 환경에서 각 컨테이너가 2, 9, 7, 14 개수만큼 페이지를 캐싱 중이라 가정한다. 이때 각 blkio_weight의 비율은 1:2:4:8 이지만 blkio_weight 200의 컨테이너가 비율 대비 많은 페이지를 가지고 있다. 그렇기 때문에 페이지 회수 과정에서 사용빈도가 낮은 페이지들 중 blkio_weight 200의 페이지를 우선적으로 회수하고, nr_pages_200 변수를 atomic하게 낮춰준다. 이를 통해 본 제안 기법은 각 blkio_weight별 페이지 캐시 개수를 blkio_weight의 비율에 맞게 유지할 수 있다. 다만, 새로운 페이지 할당에 사용하는 경우에 페이지 할당이 늦어지는 경우, OOM(Out of memory)문제가 발생할 수 있기 때문에 기존 기법을 따른다.

4. 성능 평가 및 분석

4.1 실험 환경

본 논문에서는 표 1과 같은 환경에서 실험을 진행하였다. 컨테이너에서 사용한 베이스 이미지는 모두 Ubuntu 14.04이고, 컨테이너 계층에서 FIO 2.2.10과 Filebench 1.5-alpha3를 구동하여 실험을 수행하였다. Docker의 파일시스템은 기본 설정 값인 Overaly2를 사용하였으며 페이지 캐시 적중률 측정은 perf-tools를 통해 측정하였다.

4.2 실험 결과

4.2.1 비율 측정

기존 리눅스 페이지 캐쉬 관리 기법과 제안 기법의 I/O 성능 비율을 비교하기 위해 다음과 같은 실험을 수행하

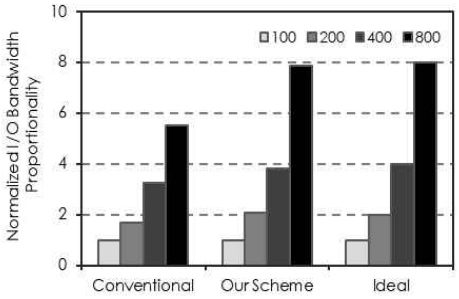

그림 3 I/O 성능 비율 비교

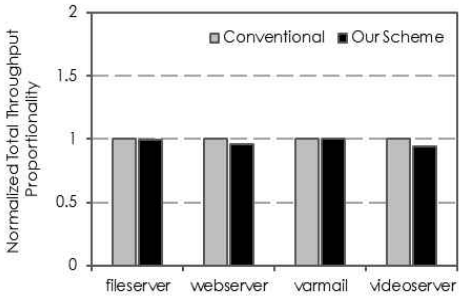

그림 4 오버헤드 비교

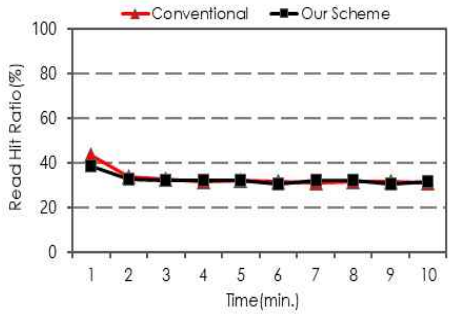

그림 5 webserver 워크로드에서의 캐시 적중률 비교

표 1 실험 환경

CPU	Intel® Core TM I 7-6700
Memory	8G
Hard Disk	Samsung 850 EVO 120G
Kernel Version	4.17.3
Docker Base Image	Ubuntu 14.04 LTS

였다. 먼저, blkio_weight가 서로 다른 총 네 개의 컨테이너에서 각각 한 개의 FIO 벤치마크를 구동하였다. 각 FIO 벤치마크는 2GB 파일 한 개를 생성한다. 이때, 전체 쓰기량이 메모리의 크기보다 크기 때문에 파일 생성 과정 중 페이지 캐시 회수 작업이 발생한다. 그 후, 같은 파일을 1GB 크기만큼 순차적 읽었을 때의 I/O 처리량을 blkio_weight 100의 결과 값에 정규화 하였다. 그림 3에서 볼 수 있듯이 기존 기법에서는 페이지 캐시에서 blkio_weight를 반영하지 않아 이상적인 I/O 처리량의 비율과 24.0% 정도의 차이를 보인다. 반면, 본 기법에서는 페이지 캐시가 회수되는 과정에서 각 컨테이너의 blkio_weight의 비율에 맞게 컨테이너별 페이지 개수를 유지하기 때문에 이상적인 비율의 수치와 2.2%이내의 차이를 보이는 것을 확인했다. 해당 실험에서 총 처리량을 비교해보면 기존 기법 대비 제안 기법이 약 5.8% 감소함을 알 수 있다.

4.2.2 오버헤드 측정

제안 기법에서 추가적인 atomic 연산 작업과 정책 변화로 발생하는 오버헤드를 측정하기 위해 Real-world 벤치마크를 구동하여 총 처리량을 비교해 보았다. 그림 4는 네 개의 컨테이너에서 실행된 워크로드의 총 처리량을 기존 기법 결과에 정규화하여 나타낸 그래프이다. 워크로드로는 Filebench 벤치마크의 fileserver, webserver, varmail, videoserver를 각 컨테이너에서 실행하였다. 제안된 기법의 총 처리량은 기존 기법대비 최대 5.9% 감소된 처리량을 보였다. 제안 기법에서 발생하는 오버헤드는 컨테이너별 페이지 비율을 유지하기 위해 blkio_weight에 비해 페이지 개수가 많은 컨테이너의 경우, 해당 컨테이너의 페이지 중 추후 접근가능성이 있는 페이지도 불가피하게 회수되기 때문이다. fileserver와 varmail에서는 워크로드 특성상 쓰기작업이 읽기작업에 비해 많기 때문에 기존 기법과 유사한 처리량을 보인다.

4.2.3 캐시 적중률 측정

기존 기법과 제안 기법의 페이지 캐시 적중률을 비교하기 위해 perf-tools를 통해 10분 동안 webserver 워크로드 구동 중 발생한 캐시 적중률을 그림 5와 같이 측정하였다. 해당 실험에서 그림 4의 webserver 워크로드를 기존 기법이과 제안 기법이 적용된 환경에서 구동한 결과이다. 제안 기법의 페이지 캐시 적중률은 기존 기법 대비 평균 0.6% 하락함을 확인할 수 있다. 따라서 제안된 페이지 캐시 관리 기법과 기존의 기법은 유사한 페이지 캐시 적중률을 보인다.

5. 결 론

본 논문에서는 컨테이너 환경에서 사용되는 Cgroups의 blkio_weight가 페이지 캐시에서 반영이 되지 않는다는 걸 알아보았으며, 이를 해결하기 위한 페이지 캐시 관리 기법을 제안했다. 본 제안 기법은 페이지 회수 과정 중 blkio_weight를 고려하여 회수함으로써, 각 컨테이너에서 사용하는 페이지 개수를 blkio_weight의 비율에 맞게 유지한다. 실험을 통해 제안 기법의 효용성을 검증해본 결과, 제안된 기법은 기존 기법보다 22.0% 향상된 I/O 성능 비율 을 보이며 제안된 기법에서 발생하는 오버헤드는 최대 5.9% 이내 인 것을 확인하였다. 마지막으로 제안 기법의 페이지 캐시 적중률 또한 평균 0.6% 하락한 것을 확인하였다. 추후연구에서는 blkio_weight가 동일한 컨테이너가 존재하며, 여러 컨테이너가 동일한 페이지 캐시를 공유하는 상황을 가정하여 실험 및 검증할 예정이다.

참 고 문 헌

[1] S. Hykes, Docker Overview [Online]. Available: https://docs.docker.com/engine/docker-overview/#the-underlying-technology (downloaded 2018, Oct. 17)

[2] S. Kim, H. Kim, J. Lee, and J. Jeong, "Enlightening the I/O Path: A Holistic Approach for Application Performance," *Proc. of USENIX Conference on File and Storage Technologies*, pp. 345-358, 2017.

[3] P. Menage, Cgroups [Online]. Available: https://www.kernel.org/doc/Documentation/Cgroups-v1/Cgroups.txt (downloaded 2018, Oct. 17)

[4] S. Yang *et al.*, "Split-level I/O Scheduling," *Proc. of ACM Symposium on Operating Systems Principles*, pp. 474-489, 2015.

[5] S. S. Hahn, S. Lee, I. Yee, D. Ryu, and J. Kim, "FastTrack: Foreground App-Aware I/O Management for Improving User Experience of Android Smartphones," *Proc. of the USENIX Conference on Annual Technical Conference*, pp. 15-28, 2018

[6] Q. Xu *et al.*, "Performance Analysis of Containerized Applications on Local and Remote Storage," *Proc. of International Conference on Massive Storage Systems and Technology*, pp. 1-12, 2017.