

ZNS 를 이용한 리눅스 컨테이너 I/O isolation 기법



201724419 김동욱

201724596 채기중

201924445 김지원

지도교수: 안 성 용

목 차

1. 서론.....	1
1.1. 연구 배경.....	1
1.2. 기존 문제점	1
1.3. 연구 목표.....	3
2. 연구 배경.....	4
2.1. Block Interface SSD.....	4
2.2. ZNS SSD.....	5
2.3. Linux Cgroup.....	6
2.4. Device Mapper(Dm-zoned)	6
3. 연구 내용.....	8
3.1. 기존 I/O Routine.....	8
3.2. Dm-zoned 컨테이너별 존 분리.....	10
3.3. 알고리즘 구현.....	11
3.3.1. 동작 과정.....	11
3.3.2. 구조체 구현.....	12
3.3.3. 함수 구현.....	14
4. 연구 결과 분석 및 평가.....	23
4.1. 실험 환경.....	23
4.2. 실험 결과.....	24
5. 결론 및 향후 연구 방향.....	25
5.1. 활용 방안.....	25

5.2. 향후 과제	25
6. 참고 문헌	26

1. 서론

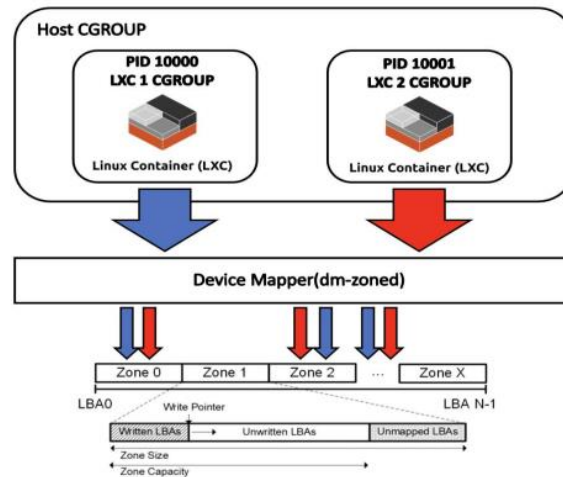
1.1. 연구 배경

Linux 컨테이너는 런타임 환경에서 애플리케이션을 패키지와 분리하는 기술이다. 실행에 필요한 모든 파일(라이브러리와 그에 대한 종속 항목)이 호스트 시스템과 분리되어 동작하기 때문에 컨테이너 환경에서 동작하는 프로세스는 이식과 관리가 쉽다. 하지만 컨테이너의 리소스가 호스트와 분리되었다고 해도 실제 SSD에 저장되어 있는 데이터들은 격리되어 있지 않다. 만약 이 데이터들 사이의 격리가 이루어진다면 각 컨테이너들이 저장 장치에 접근할 때 I/O 간섭이 발생하지 않기에 빠르고 효율적인 처리가 가능해진다.

1.2. 기존 문제점

현재 리눅스 컨테이너가 사용하는 block Interface SSD는 page 단위로 write를 실행한다. Page size보다 작은 write가 요청되어도 무조건 하나의 page를 통째로 사용하는 write amplification, 즉 부가적 쓰기가 일어나게 된다. 또한, SSD에 데이터를 저장할 때는 접근하려는 page가 free 상태일 때만 write가 가능하다. 이미 쓰여진 데이터가 변경되면 기존 page의 내용은 내부 레지스터로 복사 후 변경되어 새로운 free 상태의 page에 기록되는 out-place update 과정을 거친다. 이러한 write amplification과 out-place update 제약으로 인해 유효한 데이터와 garbage 영역의 분리가 일어나지 않아 효율적인 저장 공간의 사용이 어렵다는 단점을 갖는다. 컨테이너별로 사용되는 데이터들 역시 무작위로 섞여서 관리되고, 하나의 서버를 공유해서 사용하는 리눅스 컨테이너들은 이러한 데이터의 혼재로 인해 발생하는 I/O 간섭 때문에 성능 저하를 겪게 된다.

ZNS SSD는 전체 저장 공간을 작고 일정한 용량의 구역인 zone으로 나누어 별도의 영역에 순차적으로 write가 가능하게 한다. 하나의 zone에는 유사한 데이터들이 모여 있으며 zone 단위로 삭제가 이루어지기 때문에 garbage가 발생하지 않는다. 따라서 데이터들을 zone 단위로 묶어 관리하는 ZNS SSD 환경 위에서 리눅스 컨테이너를 실행하게 되면 데이터의 사용 주기와 빈도에 따라 효율적인 접근이 가능해진다.

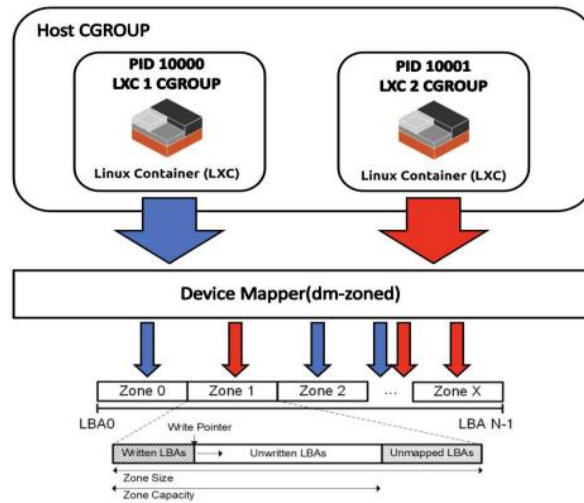


[그림 1] 현재 LXC의 Dm-zoned 매핑 방식

저장 방식을 zoned storage system으로 바꾸기 위해 기존 LXC 환경에서 ZNS SSD를 device mapper로 매핑한 후 이를 컨테이너의 storage pool로 사용해 컨테이너를 생성하게 되면 device mapper를 거쳐 여러 개의 zone에 컨테이너의 I/O가 발생한다. 컨테이너를 추가로 생성하는 경우 또다시 여러 개의 zone을 할당하게 된다. 이때 비어 있는 zone을 할당하지 않고 그림 1과 같이 기존에 생성된 컨테이너가 사용하고 있는 zone에 append하는 형식으로 I/O가 발생한다.

이 경우 컨테이너마다 독립된 zone을 할당받지 못하고 이미 다른 컨테이너가 사용하고 있는 zone을 공유하므로 ZNS SSD의 장점을 제대로 활용할 수 없다. ZNS SSD는 기존의 regular device와 달리 애플리케이션당 독립된 zone을 사용함으로써 sequential write를 제공한다는 장점이 있는데, 현재 시스템의 구조에서는 각각의 zone에 여러 개의 컨테이너가 할당되어 있으므로 I/O 명령의 분리가 일어나지 않는다는 문제가 존재한다.

1.3. 연구 목표

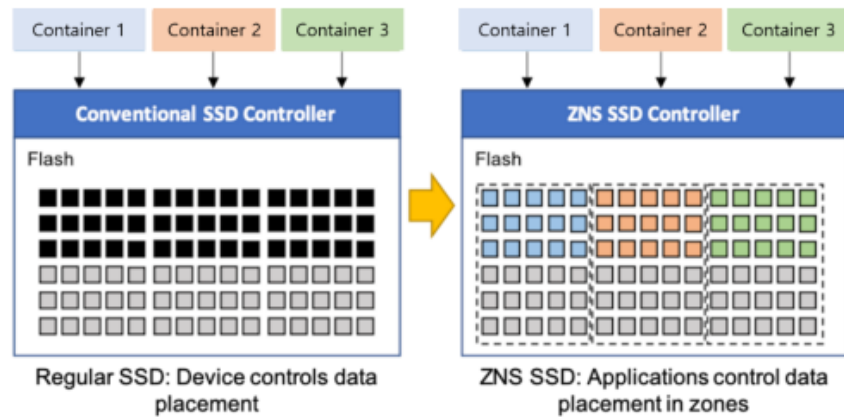


[그림 2] 컨테이너별로 각각 다른 Zone을 사용하는 구조

ZNS SSD는 용도와 사용 주기가 동일한 데이터를 zone에 순차적으로 저장하고 지운다. 하지만 현재 리눅스 컨테이너는 ZNS SSD를 지원하지 않기 때문에 단순히 저장 장치의 종류를 변경하는 것만으로는 이점을 활용할 수 없다. 또한 여러 개의 컨테이너가 하나의 zone에 접근하여 컨테이너별 zone 분리가 일어나지 않는다. 이를 개선하여 각각의 컨테이너가 서로 다른 zone을 사용하도록 한다면 I/O stream 간의 간섭이 제거되어 성능을 향상시킬 수 있다.

현재의 device Mapper는 Logical ZNS SSD를 emulate할 수 있지만 단순히 데이터를 zone 단위로 관리하는 것 외에는 부가적인 기능이 없다는 한계가 있다. 여러 개의 컨테이너에서 들어오는 데이터들이 출처와 관계없이 단순히 사용 주기와 용도에 따라 분리되기 때문에 같은 zone에 다수의 I/O stream이 접근한다는 단점이 있다. 이는 곧 I/O 간섭을 야기시켜 성능 저하가 일어나기 때문에 기존의 zone 관리 방식에 컨테이너별 데이터 분리 기준을 추가하여 격리가 가능하게 해야 한다.

따라서 본 팀은 각 컨테이너에서 I/O가 발생할 시 사용되는 데이터들을 zone 단위로 격리시키는 알고리즘을 개발하여 리눅스 컨테이너가 ZNS SSD의 특성을 이용할 수 있도록 하는 것을 목표로 한다.

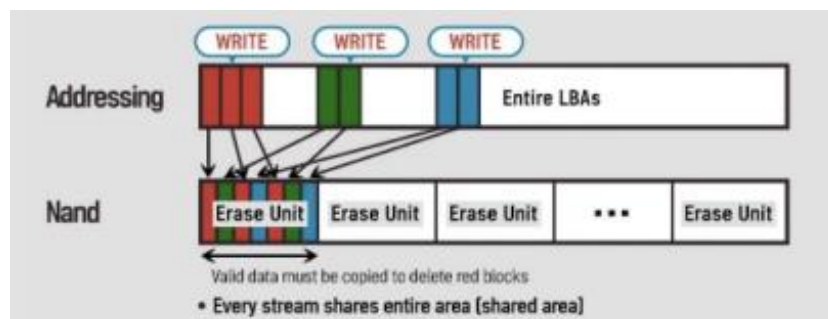


[그림 3] 기존 SSD의 저장 방식과 본 팀이 목표하는 ZNS SSD를 이용한 분리 할당 방식. 단순히 ZNS SSD를 리눅스 시스템에 적용하는 것만으로는 컨테이너별로 독립된 zone을 할당할 수 없다.

2. 연구 배경

2.1. Block Interface SSD

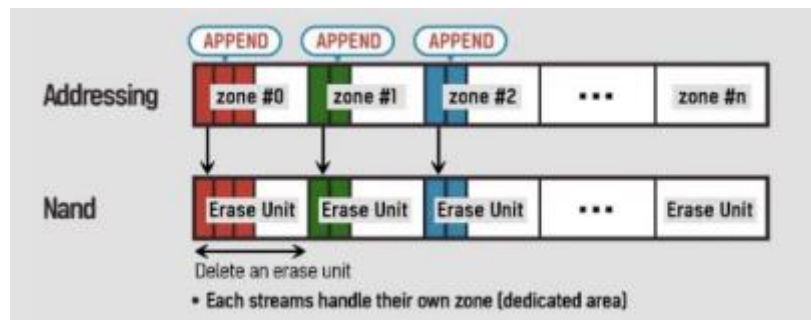
현재 사용되고 있는 block Interface SSD는 random write 방식을 이용해 데이터를 기록한다. 여러 개의 소프트웨어에서 생성되는 데이터들은 논리적으로는 원하는 영역에 저장되는 것처럼 보일지 몰라도 물리적으로는 nand block에 순차적으로 저장된다. Nand flash의 특성상 이 영역에는 유효한 데이터와 garbage 데이터가 혼재하기 때문에 저장 공간을 효율적으로 사용하기 어렵다. 따라서 garbage collection 작업을 통해 공간을 확보해줘야 하는데, 이는 유효한 자원들을 모아 다른 nand block에 다시 write하는 overhead가 큰 방식이다. Garbage collection이 일어나는 동안은 현재 실행 중이던 read/write 작업이 중단되기 때문에 성능 저하가 발생한다.



[그림 4] 기존 SSD의 write 방식

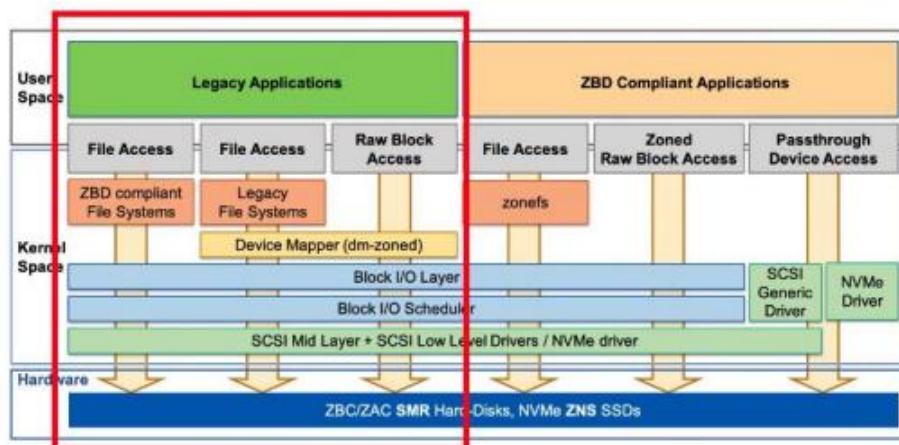
2.2. ZNS SSD

ZNS SSD는 zoned namespace 기법을 이용한 차세대 저장장치이다. 용도와 사용 주기가 비슷한 데이터를 zone 단위로 순차적으로 저장하고 지우기 때문에 garbage collection으로 인한 추가적인 I/O가 발생하지 않아 overhead가 적어진다. 각 zone은 write pointer를 가지며, 해당 zone으로 내려온 I/O 명령은 write pointer로부터 sequential write 된다.



[그림 5] ZNS SSD의 write 방식

이와 같은 방식 덕분에 ZNS SSD는 성능 저하가 없는 효율적인 저장 공간 운용이 가능하다. 따라서 서버 상에서 여러 운영체제와 프로세스들을 처리하는 현대의 데이터 센터에 적합하다.



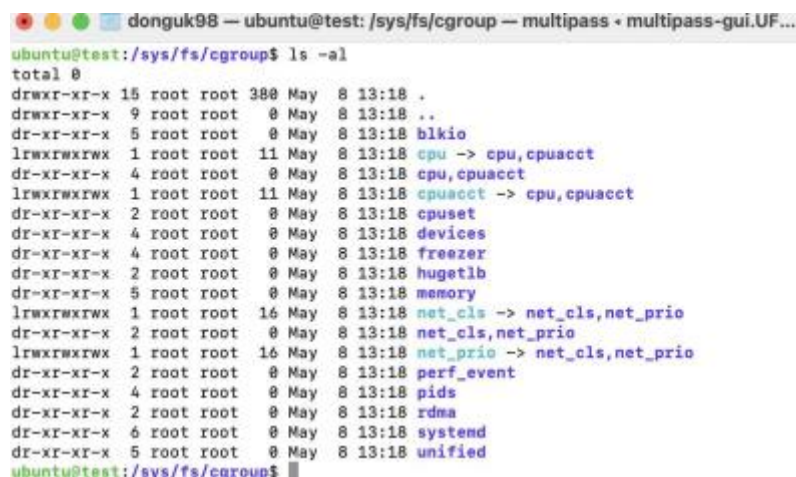
[그림 6] Legacy Application의 ZNS SSD I/O 과정

ZNS SSD를 리눅스 시스템에 연동하기 위해서는 F2FS나 EXT4와 같은 기존의 filesystem을 사용하는 legacy application이 zoned storage의 개념을 이해할 수 있도록 해야 한다. ZBD를 지원하는 filesystem을 이용하거나 후술할 device mapper를 이용하는 방법이 있는데, 본 팀은 dm-zoned device mapper를 이용하였다.

2.3. Linux Cgroup

Linux Kernel System은 cgroup을 이용하여 단일 또는 태스크 단위의 프로세스 그룹에 대한 자원 할당을 제어한다. Cgroup은 프로세스와 마찬가지로 계층적으로 구성되어 있다. 하위 cgroup은 부모 cgroup 속성의 일부를 상속하도록 되어 있으며 여러 개의 서로 다른 cgroup 계층이 동시에 존재할 수 있다. 각 계층은 하나 이상의 subsystem에 연결되는데, subsystem이란 cpu, memory, blkio등과 같은 자원을 조절하는 시스템을 말한다.

Cgroup은 file system의 형태로 관리되며, /sys/fs/cgroup 경로 내에서 cgroup의 계층구조가 directory와 file로 구현되어 있는 것을 확인할 수 있다. 이는 Linux 상에서 cgroup 제어를 위해 탄생한 특수 filesystem으로 cgroupfs라 불린다. 특정 경로 아래에 있는 directory 또는 file을 조작하여 cgroup을 제어할 수 있다.



```
ubuntu@test:/sys/fs/cgroup$ ls -al
total 0
drwxr-xr-x 15 root root 380 May  8 13:18 .
drwxr-xr-x  9 root root  80 May  8 13:18 ..
dr-xr-xr-x  5 root root  80 May  8 13:18 blkio
lrwxrwxrwx  1 root root 11 May  8 13:18 cpu -> cpu,cpuacct
dr-xr-xr-x  4 root root  80 May  8 13:18 cpu,cpuacct
lrwxrwxrwx  1 root root 11 May  8 13:18 cpuacct -> cpu,cpuacct
dr-xr-xr-x  2 root root  80 May  8 13:18 cpuset
dr-xr-xr-x  4 root root  80 May  8 13:18 devices
dr-xr-xr-x  4 root root  80 May  8 13:18 freezer
dr-xr-xr-x  2 root root  80 May  8 13:18 hugetlb
dr-xr-xr-x  5 root root  80 May  8 13:18 memory
lrwxrwxrwx  1 root root 16 May  8 13:18 net_cls -> net_cls,net_prio
dr-xr-xr-x  2 root root  80 May  8 13:18 net_cls,net_prio
lrwxrwxrwx  1 root root 16 May  8 13:18 net_prio -> net_cls,net_prio
dr-xr-xr-x  2 root root  80 May  8 13:18 perf_event
dr-xr-xr-x  4 root root  80 May  8 13:18 pids
dr-xr-xr-x  2 root root  80 May  8 13:18 rdma
dr-xr-xr-x  6 root root  80 May  8 13:18 systemd
dr-xr-xr-x  5 root root  80 May  8 13:18 unified
```

[그림 7] Linux 시스템에서 cgroupfs의 모습

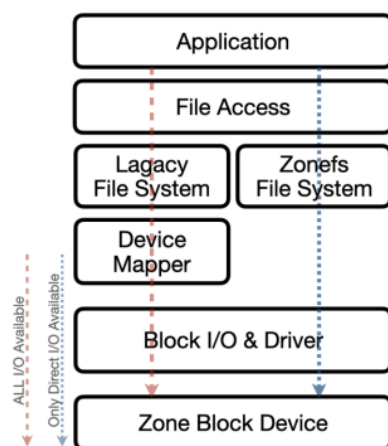
원하는 subsystem directory 아래에 새로운 directory를 생성하면 부모 directory의 요소들이 상속되어 다수의 file들이 생성되고, 이렇게 새로운 cgroup을 생성할 수 있다. File들 중 task 파일 안에는 process PID들이 기록되는데, 이를 수정해 process들의 시스템 자원을 제한할 수 있다.

2.4. Device Mapper(Dm-zoned)

Linux는 5.6 버전부터 zonefs라는 ZBD에 특화된 filesystem을 제공한다. Zonefs는 raw block에 direct I/O를 쉽게 할 수 있도록 추상화된 메서드를 제공하고 zone을 파일 형태로 시각화한다는 장점이 있다. 일반적인 POSIX Filesystem과 다르게 zonefs는 ZBD의 sequential write 제약을 사용자에게 숨기지 않는다. 따라서 보다 원시적인 block device에

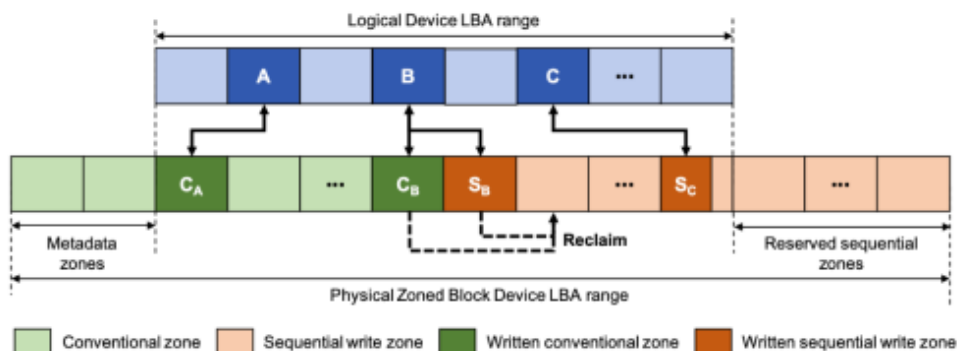
대한 접근을 허용한다는 장점이 있다. 그러나 ZBD compliant application을 위해 만들어진 시스템이기 때문에 기존의 POSIX filesystem과 호환되지 않고 direct I/O만 지원한다는 단점이 있어 linux 컨테이너와 같은 legacy application에는 적용할 수 없다.

Zonefs를 사용하기 위해 mkzonefs라는 tool을 이용해 FEMU 상에 emulate 했던 ZNS SSD(nvme0n1)을 포맷하고 zonefs에 마운트 했으나, Linux 컨테이너는 ext4, btrfs와 같은 file system을 지원하므로 device mapper를 이용하는 방법을 채택하였다.



[그림 8] Zone Block Device에서의 Application I/O Flow

Linux 컨테이너별로 독립된 zone에 sequential write를 가능하게 하기 위해서는 device mapper라는 중간자를 이용해 filesystem에게 ZBD의 sequential write 제약을 숨겨 POSIX filesystem에서 ZBD를 사용할 수 있도록 해야 한다.



[그림 9] Dm-zoned target device에서의 zone 매핑

Dm-zoned device mapper는 그림 9와 같이 conventional zone과 sequential write zone을 결합시킨 physical ZBD에 target device를 매핑한다. Conventional zone은 Metadata / Data / Buffered zone으로 나뉘고 sequential write zone은 Data zone으로만 이루어져 있다.

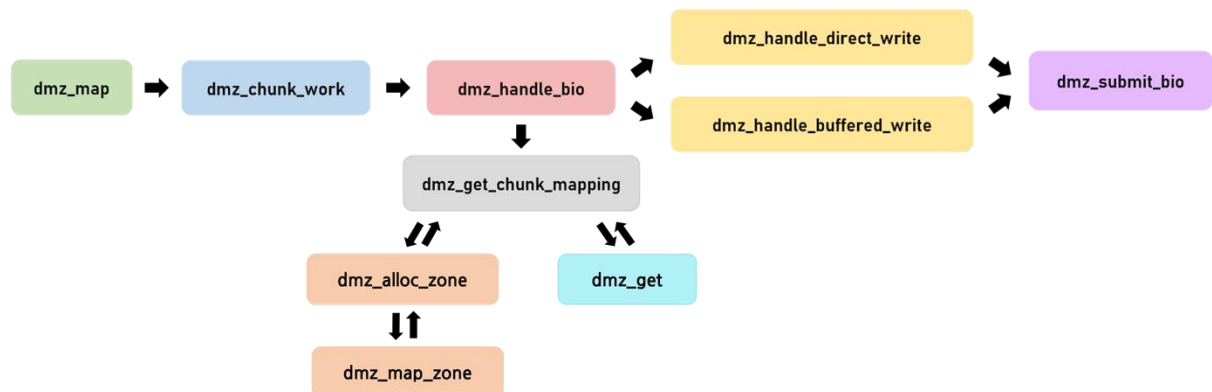
Logical zone이 conventional zone에 이미 매핑되어 있는 상태에서 write 요청이 들어 올 경우에는 바로 random data zone에 write가 가능하다. 만약 sequential zone에 매핑되어 있을 경우 chunk의 write pointer와 sequential zone의 write pointer를 비교 동일 정렬된 경우에만 바로 write한다. 동일하게 정렬되어 있지 않은 경우 Dual Conventional-Sequential Zone 매핑이 이루어지는데, 임시로 conventional zone을 매핑하여 write를 수행한다. 이는 sequential data zone과 buffer zone을 동시에 사용하는 경우이다.

Conventional zone의 여유 공간이 부족한 경우 저장 공간을 회수하는 reclaim이 수행된다. Reclaim이란 conventional zone의 데이터를 비어 있는 sequential zone으로 복사하는 일종의 garbage collection 과정이다. 복사가 완료되면 chunk 매핑을 갱신해야 한다.

하지만 dm-zoned 환경에서 여러 개의 컨테이너에 I/O가 발생하는 경우 동일한 chunk에 접근하는 현상이 발생한다. 즉 하나의 zone에 다수의 컨테이너가 입출력을 수행하는 문제점이 있어 I/O 과정에서 컨테이너 간 간섭이 발생한다. 따라서 추가적인 알고리즘을 구현해 각 컨테이너에서 들어오는 데이터를 분리해서 저장할 수 있도록 해야 한다.

3. 연구 내용

3.1. 기존 I/O Routine



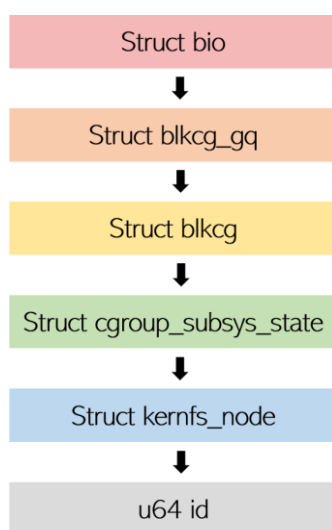
[그림 10] I/O Routine

Device Mapper로 입력된 Bio 요청은 처음에 `dmz_map` 함수로 전달된다. `Dmz_map` 함수는 target device 정보와 bio 구조체의 sector 정보를 이용해 chunk work 구조체에 sector 정보를 할당하여 target device가 유효한지 확인하고 bio 요청을 Flush list와 Bio list(Work queue)에 추가한다. 이렇게 Bio list에 추가된 Bio는 `dmz_chunk_work`에서 처리된다. `Dmz_chunk_work`는 bio list에서 bio를 가져와 `dmz_handle_bio`에 전달한다.

Dmz_handle_bio는 bio를 처리하는 함수이다. 인자로 전달받은 bio를 처리하기 위해서 dmz_get_chunk_mapping 함수를 이용해 zone을 할당받는데, metadata 구조체와 bio가 매핑된 chunk id, bio의 operation flag, bio 구조체가 인자로 전달된다. Dmz_get_chunk_mapping 함수는 전달받은 chunk id를 이용해 해당 chunk에 zone이 매핑되었는지 확인한다. 만약 zone이 이미 zone이 매핑되어 있다면 그 zone을 반환하고, 그렇지 않다면 새로운 zone을 할당받아 매핑 시킨 후 반환한다. 이후 dmz_handle_bio는 인자로 받은 bio operation flag에 따라 dmz_handle_read, dmz_handle_write, dmz_handle_discard 중 해당하는 함수를 호출한다.

만약 bio operation 타입이 REQ_OP_WRITE, 즉 쓰기 요청이라면 dmz_handle_write가 호출된다. 이 함수는 전달받은 zone의 타입에 따라 direct / buffered write의 여부를 결정한다. Zone이 random zone이거나, sequential zone이고 bio의 chunk block이 zone write pointer block과 일치한다면 dmz_handle_direct_write 함수를 호출하고 그렇지 않다면 dmz_handle_buffered_write 함수를 호출한다.

Dmz_handle_direct_write와 dmz_handle_buffered_write는 dmz_submit_bio에 submit 요청을 전달하는 공통 작업을 하지만, dmz_handle_buffered_write는 sequential 한 write를 위해 buffer zone이 필요하기 때문에 buffer zone을 할당하고 buffer zone에 bio를 submit 한다. Dmz_submit_bio 함수는 받아온 bio의 clone을 만든다. 이는 device mapper에 의해 하나의 storage로 보였던 target device로 들어온 요청을 실제 random block device와 zoned block device로 나눠서 io를 수행하기 위해 bio의 sector를 수정하기 위함이다. 생성한 clone bio는 submit_bio_noacct 함수를 통해 실제 device에 쓰인다.



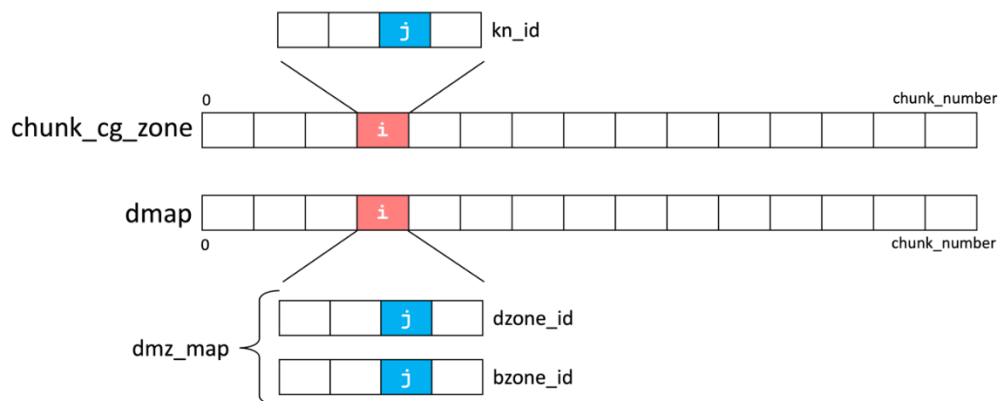
[그림 11] Bio Struct 내부 구조도

Bio 요청에 대한 정보는 구조체를 통해 전달된다. Bio 구조체는 Waiting list(Work queue)에 대한 정보 등을 담고 있는데 과제를 수행하며 주로 접근했던 부분은 blkcg_gg 구조체인 bi_blkcg이다. 해당 구조체에는 blkcg 구조체가 있고, blkcg 구조체에 cgroup_subsys_state 구조체가 존재한다. cgroup_subsys_state의 cgroup 구조체 내부 kernfs_node 구조체의 멤버 변수인 id값이 cgroup 별 고유 id가 된다.

3.2. Dm-zoned 컨테이너별 존 분리

Device mapper인 dm-zoned는 metadata에 관련된 dm-zoned-metadata.c와 dm-zoned-target.c, dm-zoned-reclaim.c 그리고 세 파일에서 사용되는 구조체와 함수 선언을 담은 dm-zoned.h 파일들이 있다. 해당 파일들 내에는 metadata를 초기화하는 함수들과 I/O 명령에 대해 zone들을 할당해 주는 함수들이 존재한다. 이 함수들과 구조체에 cgroup 정보를 추가하여 컨테이너별로 분리된 zone을 할당받아 사용할 수 있도록 한다.

기존의 device mapper는 bio의 섹터를 이용하여 하나의 chunk를 가리키고, 각 chunk는 하나의 zone을 매핑하는 1:1 방식을 사용한다. 그러나 하나의 chunk에는 여러 개의 bio, 즉 섹터는 같지만 다른 cgroup인 bio가 접근할 수 있기 때문에 여러 개의 컨테이너가 하나의 zone을 공유해서 쓰는 문제가 발생한다. 따라서 우리는 하나의 chunk에 여러 개의 zone이 매핑되는 1:N 방식을 이용하여 chunk 내부에서 cgroup이 다른 bio들이 분리된 zone을 할당받도록 하였다.



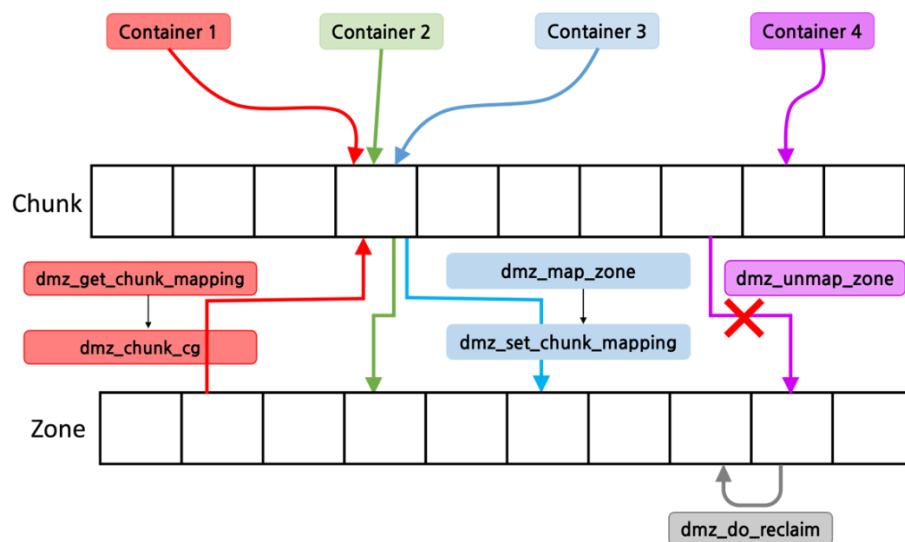
[그림 12] dmap과 chunk_cg_zone에 대한 접근 방식

Dmz_map 구조체는 chunk id를 인덱스로 하는 dmap 배열의 엔트리이다. 해당 chunk에 매핑된 dzone_id(Data zone)와 bzone_id(Buffered zone)를 저장한다. 이를 크기 4의 배열 형식으로 수정하여 한 chunk 당 4개의 zone id를 저장할 수 있도록 한다. 이후

chunk_cg_zone이라는 128x4 크기의 2차원 배열을 선언한다. 이 배열은 chunk id를 인덱스로 하며 해당 chunk를 사용하는 cgroup id를 4개 저장한다. chunk_cg_zone과 dmap에 같은 chunk id를 이용해서 접근한 후 0~3 범위 내 동일한 index 값을 사용해 cgroup_id와 dzone_id/bzone_id를 저장하면 컨테이너가 사용하는 zone의 id를 계속해서 추적할 수 있다.

3.3. 알고리즘 구현

3.3.1. 동작 과정



[그림 13] 알고리즘 동작 과정

Dmz_handle_bio를 통해 chunk에 bio 요청이 들어오면 `dmz_get_chunk_mapping` 함수는 `dmz_chunk_cg`를 호출해 cgroup에 할당된 zone id를 찾을 수 있는 index를 얻는다. 만약 매핑된 zone이 없다면 `dmz_map_zone` 함수를 호출하고 `dmz_set_chunk_mapping`을 이용해 비어 있는 zone을 할당한다. 사용이 끝난 zone은 `dmz_unmap_zone` 함수로 unmap된다. Reclaim 요청이 들어올 경우 `dmz_do_reclaim` 함수가 해당 zone을 unmapping하고 새로운 zone을 매핑한다.

3.3.2. 구조체 구현

◇ dmz_map

```
struct dmz_map {  
    // 기존의 단일 변수를 배열로 변경, 최대 4 개까지 저장 가능하다  
    __le32          dzone_id[4];  
    __le32          bzone_id[4];  
};
```

Chunk id를 index 값으로 하는 dmap이라는 이름을 가진 구조체 배열의 chunk number에 해당하는 인덱스의 원소인 dmz_map 구조체는 해당 chunk의 dzone id, bzone id를 저장한다. Dzone_id는 sequential zone id를, bzone_id는 buffered zone id를 저장한다. 기존에는 하나의 chunk 당 하나의 zone id밖에 저장하지 못했으나 dzone id와 bzone id를 배열 형식으로 변경하여 4개의 zone id를 저장할 수 있다.

◇ dmz_metadata

```
struct dmz_metadata {  
    struct dmz_dev      *dev;  
    unsigned int        nr_devs;  
    char                devname[BDEVNAME_SIZE];  
    char                label[BDEVNAME_SIZE];  
    uuid_t              uuid;  
  
    // 일부분 생략  
    . . .  
  
    /* Zone allocation management */  
    struct mutex        map_lock;  
    struct dmz_mblock   **map_mblk;  
    unsigned int        nr_cache;  
    atomic_t            unmap_nr_cache;  
    struct list_head    unmap_cache_list;  
    struct list_head    map_cache_list;  
    atomic_t            nr_reserved_seq_zones;  
    struct list_head    reserved_seq_zones_list;  
    wait_queue_head_t   free_wq;  
  
    // 추가한 배열. [chunk id][Dmz_map 의 dzone_id, bzone_id 와 동기화된  
index] 형식이다.  
    unsigned int        chunk_cg_zone[128][4];  
};
```

Zone에 대한 정보를 저장하고 있는 구조체이다. Device 정보인 dmz_dev 구조체를 포함해서 zone의 개수, metadata zone, data zone, cache zone 정보 등을 담고 있다.

◇ dm_zone

```
struct dm_zone {
    struct list_head    link;
    struct dmz_dev      *dev;
    unsigned long       flags;
    atomic_t            refcount;

    unsigned int         id;
    unsigned int         wp_block;
    unsigned int         weight;
    unsigned int         chunk;
    struct dm_zone       *bzone;

    // 추가한 멤버변수. 각 zone 을 사용하는 cgroup 의 구조체 주소를 저장한다.
    struct cgroup        *cg;
};
```

각 zone에 대한 정보를 담고 있는 구조체이다. Device 구조체, zone의 flag 상태, reference counter 등을 저장한다. 이 구조체 내에 zone id와 write pointer, 매핑된 chunk id, 연결된 buffered zone에 대한 포인터가 저장되어 있어 해당 정보들을 이용해 알고리즘 개발을 진행하였다. 각 zone에 매핑되는 cgroup 구조체의 포인터를 저장하여 하나의 cgroup만 zone을 사용하도록 했다.

3.3.3. 함수 구현

◇ dmz_chunk_cg

Dmz_metadata 구조체에 정의한 chunk_cg_zone 매핑 테이블에서 chunk와 cgroup_id에 해당하는 인덱스를 반환해 주는 함수이다. Chunk에 해당하는 배열에서 cgroup_id와 동일한 값이 존재하면 해당 인덱스를 반환하고, cgroup id와 동일한 값이 존재하지 않으면 CHUNK_PER_ZONE(index: 4) 즉, 매핑된 값을 찾지 못한 인덱스 값이 반환된다.

```
/*chunk's cgroup dzone index */
unsigned int dmz_chunk_cg(struct dmz_metadata *zmd, unsigned int chunk,
unsigned int cgroup_id){

    unsigned int i;
    for(i=0; i<CHUNK_PER_ZONE; i++)
        if(zmd->chunk_cg_zone[chunk][i] == cgroup_id)
            break;

    return i;
}
```

◇ dmz_ctr_metadata

Dm-zoned의 metadata를 초기화하는 함수이다. Metadata 구조체에 선언된 멤버 변수들을 초기화하며 dmz_load_mapping 함수를 호출하는데, 해당 시점 이전에 chunk와 dmap, zone을 매핑하는 매핑 테이블 값을 CHUNK_CG_UNMAPPED 값으로 초기화한다.

```
int dmz_ctr_metadata(struct dmz_dev *dev, int num_dev,
                    struct dmz_metadata **metadata,
                    const char *devname)
{
    struct dmz_metadata *zmd;
    unsigned int i;
    struct dm_zone *zone;
    int ret;
    zmd = kzalloc(sizeof(struct dmz_metadata), GFP_KERNEL);
    if (!zmd)
        return -ENOMEM;

    ...
    /* Initialize zone descriptors */
    ret = dmz_init_zones(zmd);
    if (ret)
        goto err;
    /* Get super block */
    ret = dmz_load_sb(zmd);
    if (ret)
        goto err;
```

```

/* Set metadata zones starting from sb_zone */
for (i = 0; i < zmd->nr_meta_zones << 1; i++) {
    zone = dmz_get(zmd, zmd->sb[0].zone->id + i);

    ...

}
/** Initialize chunk_cg_zone array */
unsigned int v;
unsigned int w;
for(v=0; v<CHUNK_SIZE; v++)
    for(w=0; w<CHUNK_PER_ZONE; w++)
        zmd->chunk_cg_zone[v][w] = CHUNK_CG_UNMAPPED;

/* Load mapping table */
ret = dmz_load_mapping(zmd);
if (ret)
    goto err;
*metadata = zmd;
return 0;
}

```

◇ dmz_load_mapping

Device mapper가 연결되는 시점에 모든 chunk를 순회하며 chunk와 chunk에 해당하는 zone을 매핑하는 함수이다. 기존에 저장되어 있는 chunk, zone 값을 매핑하고 chunk와 zone이 매핑되지 않은 경우에는 chunk 값만 증가시킨다.

Dmz_chunk_cg(zmd, chunk, CHUNK_CG_UNMAPPED)를 통해 chunk에 해당하는 인덱스를 가져온 후 dzone_id가 매핑되어있는지 체크한다. Emulated zns ssd는 초기 데이터가 존재하지 않으므로 모든 chunk와 zone은 unmapped된 상태이다.

```

static int dmz_load_mapping(struct dmz_metadata *zmd)
{
    struct dm_zone *dzone, *bzone;
    struct dmz_mblock *dmap_mblk = NULL;
    struct dmz_map *dmap;
    unsigned int i = 0, e = 0, chunk = 0;
    unsigned int dzone_id;
    unsigned int bzone_id;

    /* Metadata block array for the chunk mapping table */
    zmd->map_mblk = kcalloc(zmd->nr_map_blocks,
                           sizeof(struct dmz_mblk *), GFP_KERNEL);
    if (!zmd->map_mblk)
        return -ENOMEM;

    /* Get index with chunk cg zone mapping table */
}

```

```

    unsigned int idx;
    idx = dmz_chunk_cg(zmd, chunk, CHUNK_CG_UNMAPPED);

    /* Check data zone */
    dzone_id = le32_to_cpu(dmap[e].dzone_id[idx]);

    /* Get chunk mapping table blocks and initialize zone mapping */
    while (chunk < zmd->nr_chunks) {
        if (!dmap_mblk) {
            /* Get mapping block */
            dmap_mblk = dmz_get_mblock(zmd, i + 1);
            if (IS_ERR(dmap_mblk))
                return PTR_ERR(dmap_mblk);
            zmd->map_mblk[i] = dmap_mblk;
            dmap = (struct dmz_map *) dmap_mblk->data;
            i++;
            e = 0;
        }

        /* Check data zone */
        dzone_id = le32_to_cpu(dmap[e].dzone_id[idx]);
        if (dzone_id == DMZ_MAP_UNMAPPED)
            goto next;

        ...

        set_bit(DMZ_DATA, &dzone->flags);
        dzone->chunk = chunk;
        dmz_get_zone_weight(zmd, dzone);

        ...

next:
        chunk++;
        e++;
        if (e >= DMZ_MAP_ENTRIES)
            dmap_mblk = NULL;
    }
    ...
    return 0;
}

```

◇ dmz_get_chunk_mapping

I/O 루틴 중 dmz_handle_bio에서 I/O가 처리되기 전 chunk에 해당하는 zone을 가져오는 함수이다. 현재 bio의 cgroup id는 bio->blkcg->css.cgroup->kn->id 값에 할당되어 있다. dmz_chunk_cg 함수를 통해 chunk에 해당하는 chunk_cg_zone 배열에서 bio cgroup id와 일치하는 인덱스를 가져온다.

일치하는 인덱스가 존재하지 않는 경우에는 cgroup 값을 CHUNK_CG_UNMAPPED로 dmz_chunk_cg에 인자로 담아 cgroup이 할당되지 않은 인덱스를 가져온다. 할당된 zone 이 activate 상태가 되면, 해당 zone에 cgroup과 chunk_cg_zone에 인덱스 값을 대입한다.

```

struct dm_zone *dmz_get_chunk_mapping(struct dmz_metadata *zmd, unsigned int
chunk, int op, struct bio *bio)
{
    struct dmz_mblock *dmap_mblk = zmd->map_mblk[chunk >>
DMZ_MAP_ENTRIES_SHIFT];
    struct dmz_map *dmap = (struct dmz_map *) dmap_mblk->data;
    int dmap_idx = chunk & DMZ_MAP_ENTRIES_MASK;
    unsigned int dzone_id;
    struct dm_zone *dzone = NULL;
    int ret = 0;
    int alloc_flags = zmd->nr_cache ? DMZ_ALLOC_CACHE : DMZ_ALLOC_RND;

    unsigned int cgroup_id = bio->bi_blkcg->blkcg->css.cgroup->kn->id;
    unsigned int idx;

    dmz_lock_map(zmd);
again:
    // In chunk_cg_zone, get index according to cgroup
    idx = dmz_chunk_cg(zmd, chunk, cgroup_id);

    // if there is no index according to cgroup, get index which not
    allocated cgroup
    if(idx == CHUNK_PER_ZONE){
        idx = dmz_chunk_cg(zmd, chunk, CHUNK_CG_UNMAPPED);
    }

    /* Get the chunk mapping */
    dzone_id = le32_to_cpu(dmap[dmap_idx].dzone_id[idx]);
    if (dzone_id == DMZ_MAP_UNMAPPED) {
        /*
         * Read or discard in unmapped chunks are fine. But for
         * writes, we need a mapping, so get one.
         */
        if (op != REQ_OP_WRITE)
            goto out;

        /* Allocate a random zone */
        dzone = dmz_alloc_zone(zmd, 0, alloc_flags);
        ...

        dmz_map_zone(zmd, dzone, chunk, idx);
    } else {
        /* The chunk is already mapped: get the mapping zone */
        dzone = dmz_get(zmd, dzone_id);
        ...
    }
}

```

```

    }

    ...

    dmz_activate_zone(dzone);
    dmz_lru_zone(zmd, dzone);

    dzone->cg = bio->bi_bkg->blkcg->css.cgroup;
    if(op == REQ_OP_WRITE){
        zmd->chunk_cg_zone[chunk][idx] = kn_id;
    }

out:
    dmz_unlock_map(zmd);

    return dzone;
}

```

◇ dmz_map_zone

Zone에 chunk가 매핑되지 않은 경우 zone과 chunk의 매핑을 위해 호출되는 함수이다. dmz_get_chunk_mapping에서 계산한 인덱스를 인자로 받아 dmz_set_chunk_mapping 함수를 호출한다.

```

void dmz_map_zone(struct dmz_metadata *zmd, struct dm_zone *dzone,
                  unsigned int chunk, unsigned int idx)
{
    /* Set the chunk mapping */
    dmz_set_chunk_mapping(zmd, chunk, dzone->id,
                          DMZ_MAP_UNMAPPED, idx);
    dzone->chunk = chunk;
    if (dmz_is_cache(dzone))
        list_add_tail(&dzone->link, &zmd->map_cache_list);
    else if (dmz_is_rnd(dzone))
        list_add_tail(&dzone->link, &dzone->dev->map_rnd_list);
    else
        list_add_tail(&dzone->link, &dzone->dev->map_seq_list);
}

```

◇ dmz_unmap_zone

Zone과 chunk, zone(data zone)과 bzone(buffer zone)의 매핑 관계를 끊기 위해 호출되는 함수이다. 인자로 받은 zone의 cgroup과 일치하는 chunk_cg_zone 인덱스를 가져온다. 인자로 넘어온 zone이 bzone인 경우, dzone과 bzone의 관계만 끊는 경우이기 때문에 dzone은 여전히 chunk와 매핑을 유지시켜주고, dzone인 경우 chunk와 chunk_cg_zone에 매핑된 cgroup id까지 매핑 관계를 끊어준다.

```

void dmz_unmap_zone(struct dmz_metadata *zmd, struct dm_zone *zone)
{
    unsigned int chunk = zone->chunk;
    unsigned int dzone_id;

    if (chunk == DMZ_MAP_UNMAPPED) {
        /* Already unmapped */
        return;
    }

    unsigned int idx;

    idx = dmz_chunk_cg(zmd, chunk, zone->cg->kn->id);

    if (test_and_clear_bit(DMZ_BUF, &zone->flags)) {
        /*
         * Unmapping the chunk buffer zone: clear only
         * the chunk buffer mapping
         */
        dzone_id = zone->bzone->id;
        zone->bzone->bzone = NULL;
        zone->bzone = NULL;
    } else {
        /*
         * Unmapping the chunk data zone: the zone must
         * not be buffered.
         */
        if (WARN_ON(zone->bzone)) {
            zone->bzone->bzone = NULL;
            zone->bzone = NULL;
        }
        dzone_id = DMZ_MAP_UNMAPPED;
        zmd->chunk_cg_zone[chunk][idx] = CHUNK_CG_UNMAPPED;
    }

    dmz_set_chunk_mapping(zmd, chunk, dzone_id, DMZ_MAP_UNMAPPED);

    zone->chunk = DMZ_MAP_UNMAPPED;
    list_del_init(&zone->link);
}

```

◇ dmz_set_chunk_mapping

Dmz_map_zone과 dmz_unmap_zone에서 인자로 받은 idx를 이용하여 chunk에 해당하는 dzone_id와 bzone_id를 설정해 주는 함수이다.

```

static void dmz_set_chunk_mapping(struct dmz_metadata *zmd, unsigned int
chunk, unsigned int dzone_id, unsigned int bzone_id, unsigned int idx)
{
    struct dmz_mblock *dmap_mblk = zmd->map_mblk[chunk >>
DMZ_MAP_ENTRIES_SHIFT];
    struct dmz_map *dmap = (struct dmz_map *) dmap_mblk->data;
    int map_idx = chunk & DMZ_MAP_ENTRIES_MASK;

    dmap[map_idx].dzone_id[idx] = cpu_to_le32(dzone_id);
    dmap[map_idx].bzone_id[idx] = cpu_to_le32(bzone_id);
    dmz_dirty_mblock(zmd, dmap_mblk);
}

```

◇ dmz_do_reclaim

Dm-zoned의 reclaim은 dmz_do_reclaim 함수에서 reclaim하려는 dzone의 종류에 따라 random data zone의 경우 dmz_reclaim_rnd_data, buffer zone의 경우 dmz_reclaime_buf, sequential data zone의 경우 dmz_reclaim_seq_data로 분기된다.

```

static int dmz_do_reclaim(struct dmz_reclaim *zrc)
{
    ...

    if (dmz_is_cache(dzone) || dmz_is_rnd(dzone)) {
        if (!dmz_weight(dzone)) {
            /* Empty zone */
            dmz_reclaim_empty(zrc, dzone);
            ret = 0;
        } else {
            ret = dmz_reclaim_rnd_data(zrc, dzone);
        }
    } else {
        ret = dmz_first_valid_block(zmd, bzone, &chunk_block);
        if (ret == 0 || chunk_block >= dzone->wp_block) {
            ret = dmz_reclaim_buf(zrc, dzone);
        } else {
            ret = dmz_reclaim_seq_data(zrc, dzone);
        }
    }
    ...
}

```

◇ dmz_reclaim_rnd_data

Chunk별 chunk_cg_zone 매핑 테이블에서 chunk의 cgroup 인덱스를 가져오고 random data zone의 block 데이터를 sequential data zone으로 복사한다. 복사를 정상적

으로 수행했을 경우 random data zone의 매핑을 해제하고 비워준 뒤 sequential data zone에 cgroup을 대입하고, chunk별 chunk_cg_zone 매핑 테이블을 업데이트한다.

```
static int dmz_reclaim_rnd_data(struct dmz_reclaim *zrc, struct dm_zone
*dzone)
{
    ...
    cg = dzone->cg;
    kn_id = dzone->cg->kn->id;
    idx = dmz_chunk_cg(zmd, chunk, dzone->cg->kn->id);
    ...
    szone = dmz_alloc_zone(zmd, zrc->dev_idx,
                          alloc_flags | DMZ_ALLOC_RECLAIM);
    ...
    ret = dmz_reclaim_copy(zrc, dzone, szone);
    if (ret == 0) {
        ret = dmz_copy_valid_blocks(zmd, dzone, szone);
    }
    if (ret) {
        ...
    } else {
        ...
        dmz_unmap_zone(zmd, dzone);
        dmz_free_zone(zmd, dzone);
        dmz_map_zone(zmd, szone, chunk, idx);
        szone->cg = cg;
        dmz_set_szone(zmd, idx, chunk, kn_id);
        ...
    }
    ...
}
```

◇ dmz_reclaim_buf

Buffer zone의 block 데이터를 data zone에 복사한다. 복사를 정상적으로 수행하여 병합했을 경우 buffer zone의 매핑을 해제하고 비워준다.

```
static int dmz_reclaim_buf(struct dmz_reclaim *zrc, struct dm_zone *dzone)
{
    ...
    ret = dmz_reclaim_copy(zrc, bzone, dzone);
    if (ret < 0)
        return ret;
    ...
    ret = dmz_merge_valid_blocks(zmd, bzone, dzone, chunk_block);
    if (ret == 0) {
        ...
        dmz_unmap_zone(zmd, bzone);
    }
}
```



```

        dmz_free_zone(zmd, bzone);
        ...
    }
    ...
}

```

◇ dmz_reclaim_seq_data

Chunk_cg_zone 매핑 테이블에서 chunk의 cgroup 인덱스를 가져오고 sequential data zone의 block 데이터를 buffer zone으로 복사한다. 복사를 정상적으로 수행하고 병합했을 경우 random data zone의 매핑을 해제하고 비워준다. Buffer zone은 dmz_map_zone(bzone, idx, chunk) 함수의 매핑을 통하여 random data zone이 되는데 이는 추후에 수행할 sequential data zone에 reclaim하기 위한 과정이다.

```

static int dmz_reclaim_seq_data(struct dmz_reclaim *zrc, struct dm_zone
*dzone)
{
    ...
    idx = dmz_chunk_cg(zmd, chunk, dzone->cg->kn->id);
    cg = dzone->cg;
    kn_id = dzone->cg->kn->id;
    ...
    ret = dmz_reclaim_copy(zrc, dzone, bzone);
    ...
    ret = dmz_merge_valid_blocks(zmd, dzone, bzone, 0);
    if (ret == 0) {
        ...
        dmz_unmap_zone(zmd, bzone);
        dmz_unmap_zone(zmd, dzone);
        dmz_free_zone(zmd, dzone);
        dmz_map_zone(zmd, bzone, chunk, idx);
        bzone->cg = cg;
        dmz_set_szone(zmd, idx, chunk, kn_id);
        ...
    }
    ...
}

```

4. 연구 결과 분석 및 평가

4.1. 실험 환경

CPU	Intel® Xeon® CPU E5-2620 v4 @ 2.10GHz 32Core
Memory	122G
OS	Ubuntu 20.04.5 LTS, Linux 5.10
Virtual Environment	Femu

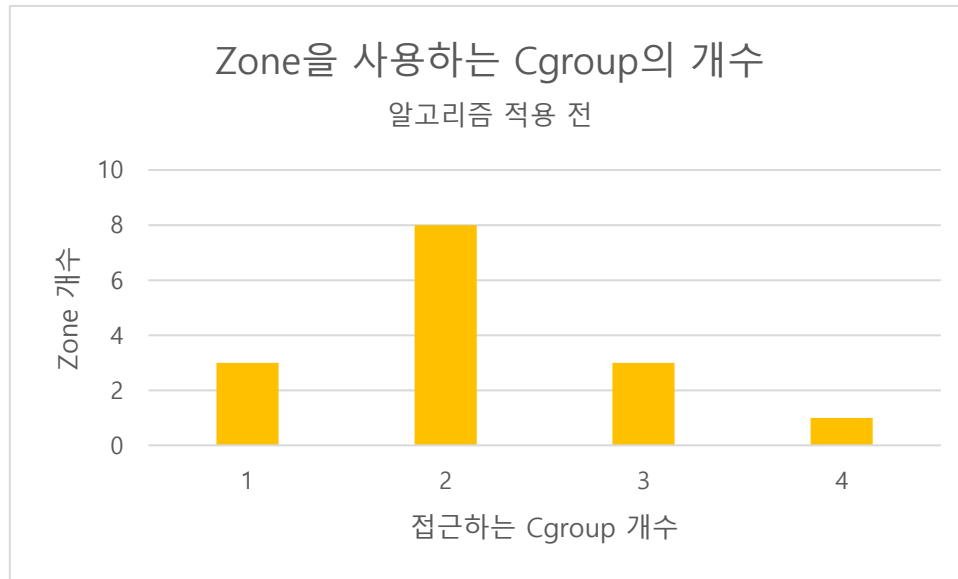
[표 1] 시스템 사양

Ioengine	libaio
Iodepth	1
Direct	1
Block size	4kb
File size	100mb
Read/Write	Random write
Number of jobs	1
Time_based	
Group_reporting	

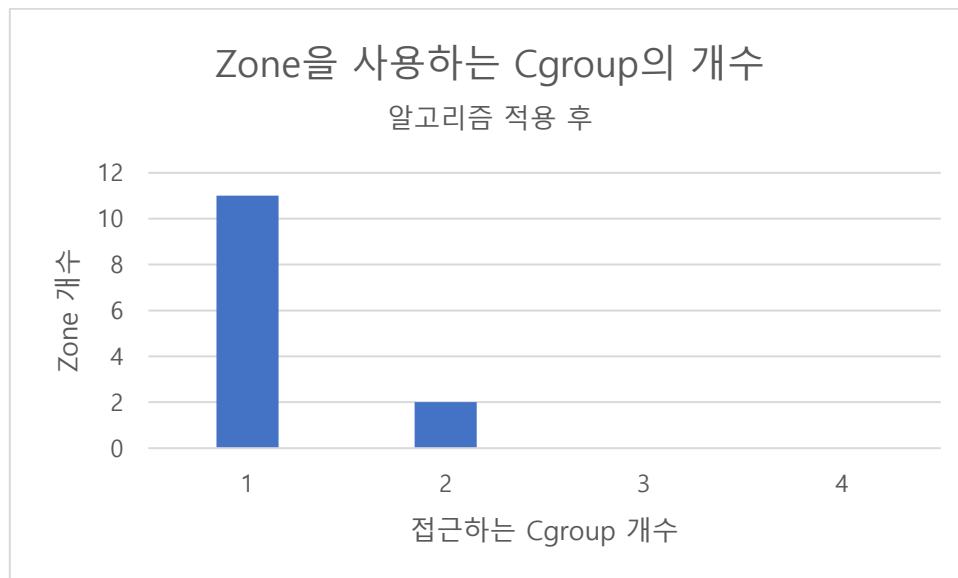
[표 2] Fio Global 설정

알고리즘을 적용하기 전, 단순히 device mapper를 이용해 ZNS SSD를 장착한 환경과 알고리즘 적용 후의 환경에서 컨테이너 내부 입출력을 시행하였다. 양쪽 환경에는 컨테이너가 3개씩 생성되어 있으며, 각 컨테이너에서 I/O를 수행해 Zone에 접근하는 Cgroup의 개수를 카운트하였다.

4.2. 실험 결과



[그래프 1] 알고리즘 적용 전 - 접근하는 Cgroup의 수에 따른 Zone의 개수



[그래프 2] 알고리즘 적용 후 - 접근하는 Cgroup의 수에 따른 Zone의 개수

[그래프 1]을 통해 알고리즘 적용 전에는 여러 개의 cgroup이 하나의 zone에 접근하는 경우가 많았음을 알 수 있다. 적게는 2개부터 최대 4개의 cgroup이 하나의 zone에 접근하여 I/O 간섭이 발생한다.

알고리즘을 적용한 후 cgroup 주소가 다를 경우 별개의 zone을 사용하도록 구현하였기 때문에 [그래프 2]에서는 대부분 zone당 하나의 cgroup만 접근하는 것을 알 수 있다. 일부 zone의 경우 2개 또는 그 이상의 cgroup이 접근하는 것처럼 관찰되는 경우가 있는데, 이는 기존에 사용되던 zone이 unmap되었다가 다른 cgroup에 mapping되어 두 개의 cgroup이 하나의 zone을 공유하는 것처럼 보이기 때문이라 추측한다.

5. 결론 및 향후 연구 방향

5.1. 활용 방안

수많은 데이터들이 쏟아져 나오는 현대 사회에서 데이터 센터의 스토리지 용량 요구는 지속해서 증가하고 있다. 대용량 데이터 인프라에서는 수만 개의 SSD 및 HDD를 사용하고 있으며, 엄청난 규모의 데이터를 비용 효율적으로 관리하는 방법에 대한 요구가 늘고 있다. Zoned Storage는 호스트와 장치 간의 논리/물리적 매핑에 필요한 여러 수준의 간접 참조를 제거하여 소프트웨어와 하드웨어가 효율적인 작동을 하도록 한다.

본 팀의 연구 과제는 단순한 호스트 수준에서의 간접 저하뿐만 아니라 리눅스 컨테이너 사이에서의 I/O 성능 간섭을 제거함으로써 보다 확실한 프로세스 격리 기능을 제공할 수 있도록 한다. 리눅스 컨테이너를 기반으로 하는 클라우드 서버에 대한 수요가 증가하고 보다 빠른 데이터 처리 능력이 요구되는 현재, 클라우드 시스템에 ZNS SSD를 접목시켜 간섭을 제거하면 하나의 서버를 공유하여 사용하는 사용자들의 성능 경험을 최대화시킬 수 있다.

5.2. 향후 과제

본 팀이 개발한 알고리즘은 배열을 기반으로 static 하게 cgroup 정보를 저장하도록 설계했다. Chunk 당 4개의 zone을 할당받을 수 있도록 설계하였는데 4개보다 더 많은 컨테이너가 I/O를 실행하는 상황에서는 컨테이너가 더 이상 zone을 할당받지 못한다. 또한 chunk에 zone이 한 번 매핑된 경우 bio sector 정보를 기준으로 zone을 나누기 때문에 zone 용량을 효율적으로 사용하지 못한다. 따라서 bio sector 정보를 기준으로 zone을 나누는 것이 아닌, segment 단위로 분리하여 cgroup 마다 zone을 할당하도록 개발한다면 보다 효율적으로 zone 용량을 사용할 수 있을 것이다.

6. 참고 문헌

- [1] Zoned Storage. <https://zonedstorage.io/>
- [2] The Linux Kernel. ZoneFS-Zone filesystem for Zoned block devices. <https://www.kernel.org/doc/html/latest/filesystems/zonefs.html>
- [3] Western digital corporation Github. Zonefs-tools. <https://github.com/westerndigitalcorporation/zonefs-tools>
- [4] Linux 컨테이너 s. Introduction of LXD. <https://linux 컨테이너 s.org/lxd/introduction/>
- [5] Ubuntu Manuals. <https://manpages.ubuntu.com/>
- [6] Red Hat Customer Portal. <https://access.redhat.com/>
- [7] Blackinkgj Blog. <https://blackinkgj.github.io/>
- [8] Kyuhwa Han, Sungkyunkwan University and Samsung Electronics, “ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction”, July, 2021. Available: <https://www.usenix.org/conference/osdi21/presentation/han>
- [9] Matias Bjørling, Western Digital, “ZNS: Avoiding the Block Interface Tax for Flash-based SSDs”, July, 2021. Available: <https://www.usenix.org/conference/atc21/presentation/bjorling>
- [10] Bootlin, Linux v5.10. <https://elixir.bootlin.com/linux/v5.10/source>
- [11] The Linux Kernel Documentation. <https://docs.kernel.org/index.html>
- [12] Western Digital <https://www.westerndigital.com/ko-kr>
- [13] Kakao Tech <https://tech.kakao.com/>