Pusan National University
Computer Science and Engineering
Technical Report 2023-10

# Zoned Namespace 를 이용한 SSD 성능 Isolation 기법 연구

저자 1 : 윤건우

저자 2 : 황인욱

저자 3 : 조준서

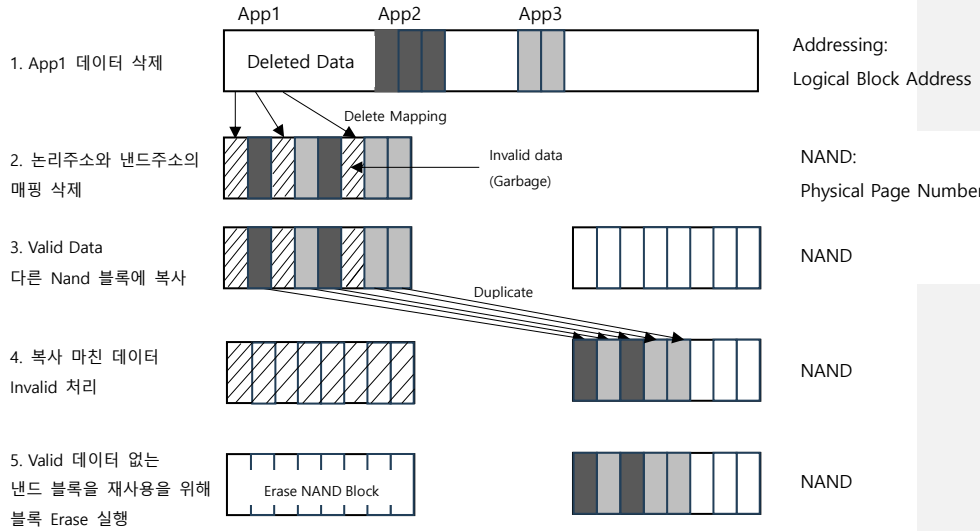
지도교수 : 안성용

# 목 차

# 1. 서론

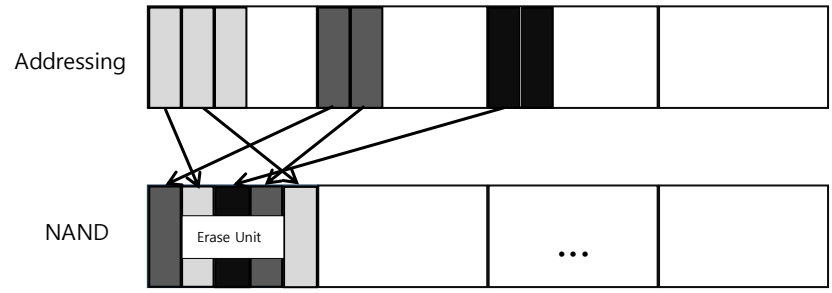
## 1.1. 연구 배경

SSD(Solid State Drive)[1]란 반도체를 이용하여 정보를 저장하는 장치이다. SSD는 HDD (Hard Disk Drive)에 비해 빠른 속도를 자랑하는데 SSD 역시 가비지 콜렉션(Garbage Collection)이 발생하는 단점이 있다. 가비지 콜렉션은 [그림 1]과 같이 동작한다. 이때 진행 중이던 읽기, 쓰기 동작이 잠시 멈추기 때문에 서비스 품질이 떨어지게 된다. 이러한 이유로 응용프로그램 사이의 성능 간섭과 가비지 콜렉션 발생 문제를 해결할 기술로 등장한 것이 차세대 기업용 SSD 표준 솔루션인 ZNS 솔루션이다.




[그림 1] 가비지 콜렉션 과정

ZNS(Zoned Namespace)[2]는 네임스페이스(Namespace)를 존(Zone) 단위로 나눠 사용하는 기술을 말한다. 여기서 네임스페이스는 논리 블록으로 포맷할 수 있는 비 휘발성 메모리의 양을 뜻한다. 컴퓨터 탐색기에 보이는 C:₩, D:₩가 각각 하나의 네임스페이스라고 이해하면 쉽다. ZNS는 기존 SSD와 저장 방법이 다른데 기존 SSD에서는 여러 응용프로그램이 논리적으로 각자 원하는 영역에 데이터를 저장하더라도, 그림[2]와 같이 실제 하나의 낸드(NAND) 블록에 순차적으로 데이터가 저장된다. 이렇게 되면 하나의 낸드블록에 여러 데이터가 섞여 저장되므로 데이터 변경 시 Garbage 발생을 피할 수 없다.




[그림 2] Conventional SSD 저장 방식

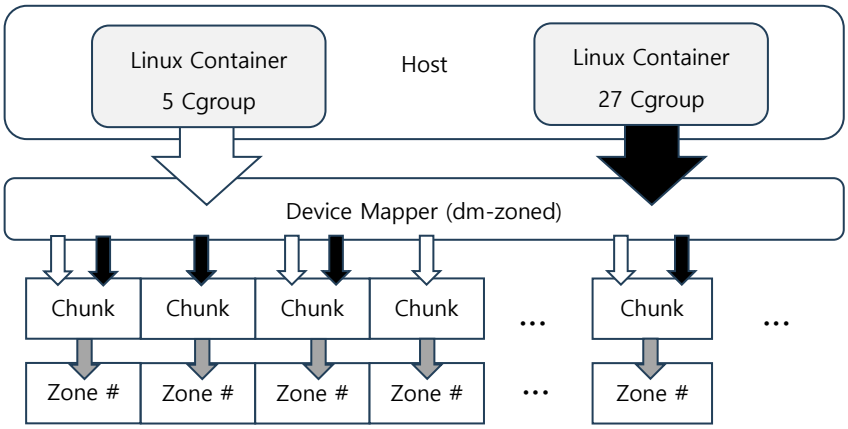
반면, ZNS SSD에서는 여러 응용프로그램이 각자 정해진 존에 순차적으로 데이터를 저장한다. 존은 논리적인 공간과 더불어 실제 낸드블록에서도 나누어져 있다. 하나의 존 안에는 비슷한 데이터끼리 모여 있으며, 순차적으로 저장했다가 존 단위로 지우기 때문에 Garbage가 발생하지 않는다. 따라서 가비지 콜렉션이 불필요하고 이로 인한 오버헤드를 제거할 수 있다.

현재 리눅스 컨테이너는 ZNS SSD를 지원하지 않기 때문에 단순히 저장 장치의 종류 변경(SSD를 ZNS SSD로 교체)하는 것만으로는 ZNS SSD에 직접적으로 입출력을 할 수 없다.. 그래서 디바이스 매퍼(Device Mapper)[3]라는 가상 블록 장치를 물리적 블록 장치에 매핑하기 위한 리눅스 커널에서 제공하는 프레임워크를 사용해야 한다. 우리는 실험을 위해 dm-zoned라는 디바이스 매퍼를 사용한다.

### 1.2. 기존 문제점

현재 Linux 커널에서는 ZNS SSD를 저장소로 사용하기 위해 dm-zoned라는 Device Mapper를 사용한다. 컨테이너들을 생성한 후, 입출력을 요청하면 [그림 3]과 같이 dm-zoned에 의해 요청들이 청크(Chunk)에 모이게 되고 청크의 요청들은 매핑된 각 존에 쓰여지게 된다. 하지만 현재 dm-zoned의 입출력 요청들은 어떤 컨테이너로부터 생성되었는지 구분되지 않은 채 청크에 모인다. 따라서 결국 여러 컨테이너에서 생성된 입출력 요청들이 구분되지 않은 채 하나의 청크에 같이 쓰여지고 존에 매핑되고 저장된다. 결과

적으로 컨테이너 간의 데이터 격리가 이루어지지 않게 된다.




[그림 3] 기존 dm-zoned 매핑 방식

[그림 4]는 기존 dm-zoned의 매핑 방식을 프린트로 찍어서 나온 결과 값들 중 일부이다. 그림에서 알 수 있듯 Cgroup 27과 Cgroup 5가 동일한 청크인 109번을 공유하는 것을 알 수 있다. 이렇게 하나의 Zone에 여러 컨테이너의 I/O 요청들이 섞일 경우 한 컨테이너의 과도한 I/O활동이 다른 컨테이너의 성능에 부정적인 영향을 미칠 수 있다. 예를 들어 컨테이너 간의 I/O 요청들의 우선순위 관리에 어려움이 생기고 특정 컨테이너가 높은 지연율을 가지는 성능 격차가 생길 수 있다. 또한, 각 컨테이너의 특정 자원에 대한 독자적인 운용이 불가능 해지며 특정 cgroup으로서 자원이 격리가 일어나지 않아 병렬성의 효율 또한 떨어지게 된다.

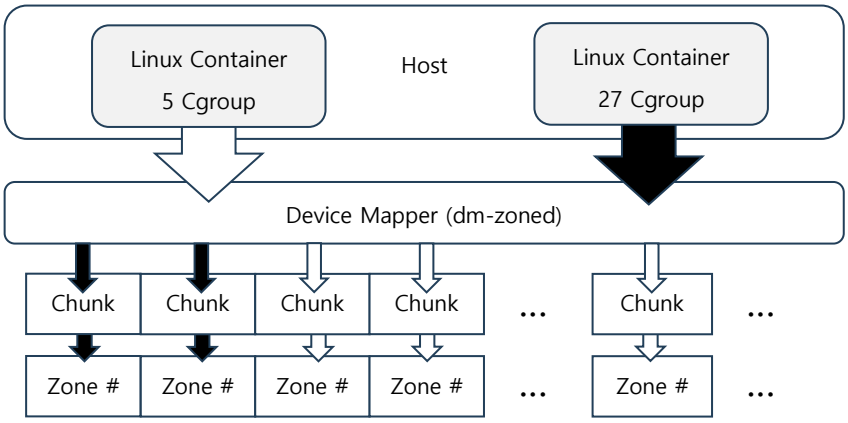
```
[    48.261675] Cgroup 27's request is going to Chunk 109
[    48.261687] bio list add!!!
[    48.261697] Cgroup 27's request is going to Chunk 109
[    48.261710] bio list add!!!
[    48.261732] Cgroup 27's request is going to Chunk 109
[    54.764518] Cgroup 5's request is going to Chunk 109
[    54.764530] bio list add!!!
[    54.764541] Cgroup 5's request is going to Chunk 109
[    54.764557] bio list add!!!
[    54.764591] Cgroup 5's request is going to Chunk 109
```

[그림 4] 기존 청크 매핑 결과

## 1.3. 연구 목표

### 1.3.1. 연구 계획

기존 문제점인 청크 수준에서부터 컨테이너 간의 격리가 이루어지지 않아 컨테이너들이 존을 공유한다는 문제점을 해결하기 위한 이상적인 격리 방안은 디바이스 매퍼인 dm-zoned에서 청크로 매핑되는 과정이 Cgroup별로 나뉘어야 하는 것이다. 그러면 결과적으로 각 존에 한 컨테이너의 데이터만 쓰여지게 된다. 이를 위해서 dm-zoned의 동작에서 하나의 청크에는 한 컨테이너의 입출력만이 들어가도록 매핑되어야 한다. 이를 그림으로 나타내면 [그림 5]와 같다.




[그림 5] dm-zoned 매핑 개선 방향 예

Linux에서 입출력은 Block 단위로 처리된다. 이를 BIO(Block I/O)라고 부르고 기존 리눅스 컨테이너에서 실행되는 프로세스들은 cgroup으로 묶여 격리된다. 우리는 BIO로부터 각 cgroup의 정보를 얻은 후, 이를 활용하여 서로 다른 BIO들이 동일한 청크를 공유해서 사용하지 않게 하기 위해 dm-zoned의 매핑 방식을 수정할 계획이다.

### 1.3.2. 연구 기대 효과

각 컨테이너의 I/O 요청이 별도의 ZNS SSD의 Zone 영역에 분리되면 컨테이너 간의 입출력 격리가 강화된다. 이로써 한 컨테이너에서 발생한 I/O 요청이 다른 컨테이너에 미치는 영향을 줄일 수 있다. 또한, 사용주기와 목적이 비슷한 데이터를 하나의 존에서

다루는 ZNS SSD의 성능에서 향상 효과를 기대할 수 있다.

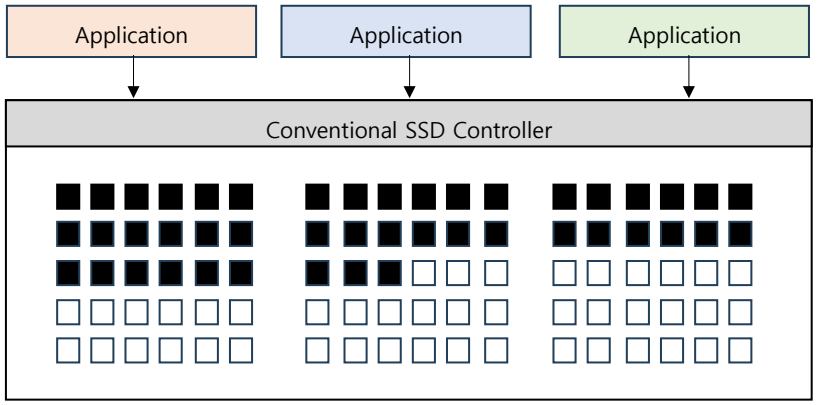
각 컨테이너는 자체 Zone에 대한 입출력을 수행하므로 병렬성이 향상되고, 다수의 컨테이너가 동시에 쓰기 요청을 수행할 때 성능 향상이 나타날 수 있다. 또한, 각 컨테이너의 데이터가 별도의 Zone에 할당되므로 데이터 관리가 더 효율적이고 특정 컨테이너의 데이터에 문제가 발생했을 때 해당 Zone의 문제만 해결함으로써 장애 복구에서도 용이성이 생길 것으로 기대된다.

# 2. 연구 배경

## 2.1. 배경 지식

### 2.1.1. Conventional SSD

일반적인 SSD는 내부 저장 공간을 나누지 않고 여러 개의 프로세스에서 생성되는 데이터를 임의로 저장한다. 각 프로세스가 원하는 영역에 Logical Block Addressing을 하더라도, 실제로는 하나의 NAND Block에 순차적으로 데이터가 저장된다. 또한 덮어쓰기가 불가한 NAND Flash 특성 상 유효한 데이터와 불필요한 데이터가 혼재되어 저장 공간을 효율적으로 사용할 수 없다. 따라서 기존 SSD는 이를 해결하기 위해 유효한 데이터를 다른 빈 공간으로 옮겨 쓰고, 불필요한 데이터만 남은 영역을 지우는 Garbage Collection 작업이 필요하고 이 과정에서 추가적인 입출력과 비용이 발생한다.

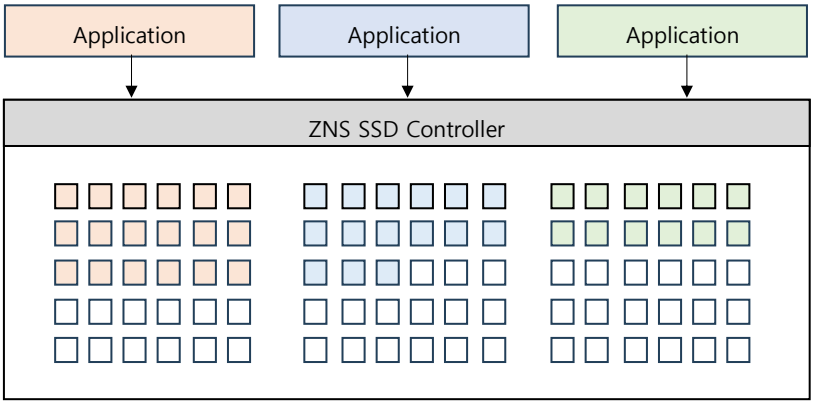



[그림 6] 기존 SSD의 데이터 배치 방식

### 2.1.2. Zoned Namespace SSD

Zoned Namespace SSD, 즉 ZNS SSD는 용도와 사용주기가 비슷한 데이터를 Zone 단위로 나누어 순차적으로 저장하고 지우기 때문에 기존 SSD의 Garbage Collection과 같은 작업이 불필요하고 추가적인 입출력과 비용이 발생하지 않는다. 또한 데이터를 Zone 단위로 관리하기 때문에 Zone 사이의 데이터와 성능 간섭을 최소화할 수 있다.

하나의 하드웨어 위에 다수의 가상 운영체제를 구동한다면 다양한 프로그램이 하나의 저장 장치를 동시에 읽고 쓴다. 이러한 다중 사용 환경에서는 서로 간섭 없이 일정한 입출력 성능을 제공하는 것이 요구된다. 즉, 여러 사용자들이 SSD에 자신만의 Zone을 가질 수 있다면 성능 간섭을 최소화할 수 있을 것이다.

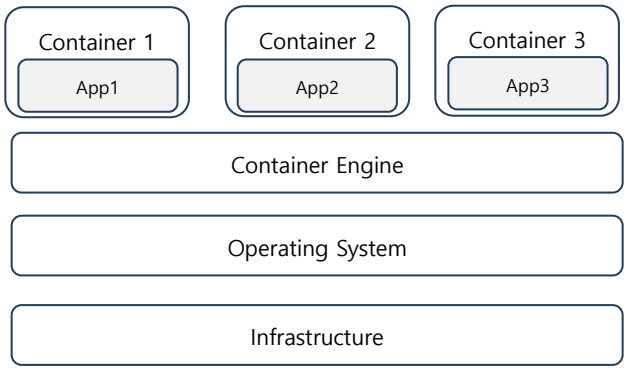



[그림 7] ZNS SSD의 데이터 배치 방식

### 2.1.3. 리눅스 컨테이너(Linux Container)

리눅스 컨테이너란 운영체제 수준의 가상화 기술로 단일 리눅스 커널에서 동작하고 있는 각 프로세스를 격리시켜 독자적인 리눅스 시스템 환경을 구축하는 것이다. 커널을 공유하는 방식이기 때문에 실행 속도가 빠르고 성능상의 손실이 거의 없다. Cgroup, namespace와 같은 격리 기술을 활용하며, 호스트 머신에게는 프로세스로 인식되지만 컨테이너 관점에서는 독립적인 환경을 가진 것처럼 보인다는 것이 장점이다.

독립적인 환경을 제공하는 리눅스 컨테이너에게 SSD 상의 독립적인 공간을 제공한다면, 입출력 성능에서 좋은 효과를 보일 것이다. 하지만 현재 Linux에서 ZNS SSD를 사용하기 위한 device mapper에서는 각 컨테이너의 입출력을 격리하지 못하고 있다.




[그림 8] Linux Container

#### 2.1.4. 청크(chunk)

dm-zoned는 기존의 블록 디바이스도 순차 쓰기를 통해 ZNS SSD를 사용할 수 있도록 지원해준다. 따라서 dm-zoned는 기존 디바이스의 공간을 ZNS SSD의 zone과 같은 크기를 가지는 청크라는 단위로 나누고, 실제 zone에 1대1로 매핑된다.

우리는 빈 청크에 대해서 초기 쓰기 명령이 들어오면 해당 청크를 빈 conventional 존에 매핑시키는 Conventional or cache zone mapping 방식을 이용할 것이다.

#### 2.1.5. Cgroup

cgroup (control groups)은 프로세스들을 격리시켜 시스템의 자원 사용 정보를 수집하고, 제한 시킬 수 있도록 하는 기술이다. cgroup을 통해 리눅스는 컨테이너 간의 격리 환경을 만들 수 있다. 우리는 여러 cgroup이 격리된 zone을 사용할 수 있도록 dm-zoned의 코드를 분석하고 수정할 것이다.

### 2.2. 개발 도구/소스/라이브러리

#### 2.2.1. 리눅스 코드 분석 도구 : ctags, cscope

- ctags

프로그래밍 소스코드의 태그 (전역변수 선언, 함수 정의, 매크로 선언)들의 Database(tags file)를 생성하는 Unix 명령어로 Linux Vim Editor에서 소스 코드 분석 및 수정할 때 유용

한 도구이다. 설치 및 사용 방법은 참고 문헌[6]을 참조하였다.

- cscope

ctags와 마찬가지로 Vim에서 활용하는 도구로 변수, 함수, 매크로, 구조체 등을 검색하기 위해서 사용된다. 주로 ctags의 부족한 부분을 채워주기 위해서 많이 사용된다. 설치 및 사용 방법은 참고 문헌[7]을 참조하였다.

#### 2.2.2. 성능 측정 도구

- FIO

: FIO (Flexible I/O Tester)는 리눅스에서 디스크 및 파일 시스템 성능을 측정하기 위한 도구이다. 임의로 디스크에 입출력을 할 수 있으며 랜덤/순차, 읽기/쓰기 등 다양한 패턴들을 모두 사용할 수 있다.

FIO를 수행하는 방법 중 해당 연구에서 jobfile을 만들어서 사용하였고 jobfile의 조건들로 블록의 크기를 4KB, 읽기/쓰기 패턴은 randwrite, 즉 랜덤 쓰기 방식으로 설정하였다. 이를 통해 랜덤 쓰기 방식을 적용한 워크로드로 ZNS SSD의 dm-zoned I/O Isolation이 잘 이루어졌는가를 평가할 것이다.

- IOzone

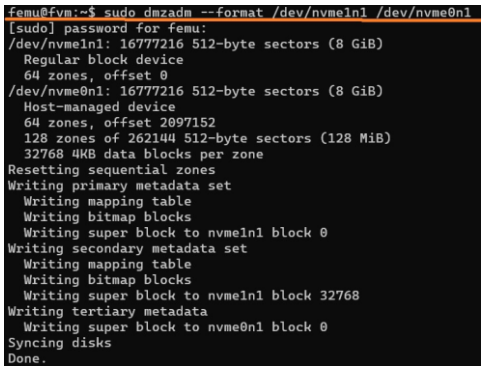
: IOzone[8]은 파일시스템 벤치마크 도구로서 벤치마크는 다양한 파일 작업을 생성하고 측정한다. IOzone은 컴퓨터 플랫폼에 대한 광범위한 파일 시스템 분석을 결정하는데 유용한데 벤치마크는 다음과 같은 작업 대해 파일 I/O 성능을 테스트 한다. (read, write, re-read, re-write, read backwards, read strided, fread, fwrite, random, 등) 여기서 우리가 사용할 IOzone 명령어와 옵션에 대한 설명은 아래와 같다.

```
~# ./iozone   -a -I 0 -g 3g -f /mnt/zns/파일이름
-a 옵션 : 자동화 모드 실행
-I 옵션 : 실행할 프로세스 또는 쓰레드 개수의 하한선을 설정
-g 옵션 : 테스트를 위한 최대 파일의 크기를 설정 옵션 (G : GB)
-f 옵션 : 테스트를 위해 사용할 파일을 설정하는 옵션
```

[명령어 1] IOzone 명령어

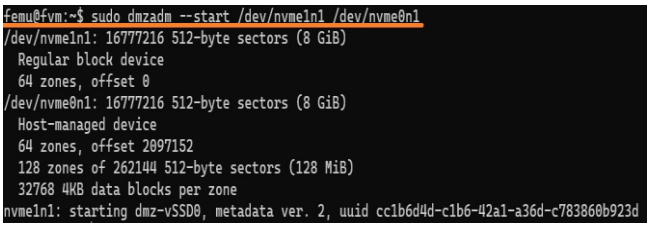
### 2.2.3. dm-zoned-tools

ZNS SSD는 기존의 블록 저장장치와 달리 순차 쓰기제약(Sequential write constraint)이 있기 때문에 기존 파일 시스템을 그대로 사용할 수 없다. 이때 리눅스 커널에서는 기존 파일 시스템이나 애플리케이션의 수정 없이 ZNS SSD를 사용할 수 있도록 dm-zoned라고 하는 영역(zone) 기반 저장장치를 위한 디바이스 매퍼를 제공하고 있다.
dm-zoned는 사용자의 요청과 하위 저장장치 사이에서 중개자 역할을 하여 요청의 유형과 저장장치의 기능에 따라 요청을 적절한 장치로 재설정한다. 또한, 사용자가 기존 소프트웨어를 업데이트하지 않고도 ZNS SSD의 성능 및 효율성 이점을 활용할 수 있으며, 순차 쓰기제약과 같은 영역 기반 저장장치 관리의 복잡성을 처리하여 사용자가 하위 저장 기술이 아닌 애플리케이션에 집중할 수 있도록 한다.




[그림 9] target device 생성

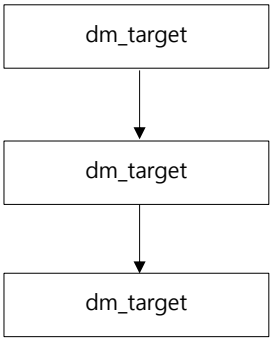
[그림 7]의 dmzadm --format 명령어는 dm-zoned target device를 생성하는 명령어로, zoned block device는 우선 dmzadm tool을 통해 포맷되어야 한다.




[그림 10] target device 실행

[그림 8]의 dmzadm –start 명령어는 포맷된 dm-zoned target device를 실행하도록 하는 명령어이다.

### 2.2.4. 리눅스 소스 코드 (구조체)

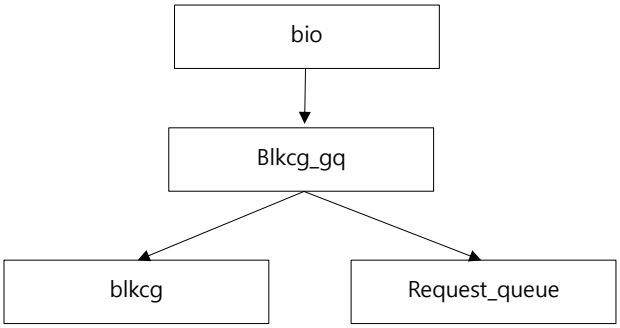



[그림 11] dm_target 및 관계도

- dm_target 구조체
  : 특정 데이터를 대상으로 삼는 필드(void *private)를 가지고 있다. 해당 필드는 본 실험에서 dmz_target 구조체 객체를 참조하고 있다.

- dmz_target 구조체
  : 대상에 대한 내용을 담고 있는 구조체로 dmz_metadata 구조체를 필드로 가지고 있다.

- dmz_metadata 구조체
  : 내부 메모리에 대한 정보를 가지고 있으며 대표적으로 zone의 총 개수, 사용 가능한 개수 등 zone에 대한 정보 및 할당 관리에 대한 정보를 필드로 가지고 있다.




[그림 12] bio 구조체 및 관계도

- bio 구조체

  : block 계층과 더 낮은 계층 (i.e., drivers, stacking drivers)의 input/Output 의 주요 요소로 주요 필드로 blkcg_gq 구조체를 가지고 있다

- blkcg_gq 구조체

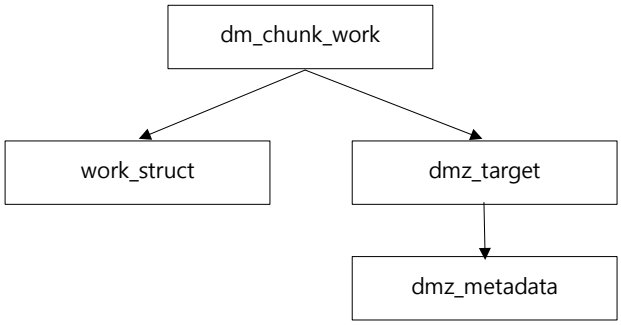
  : bio에 대한 blkcg(block cgroup) 구조체와 request_queue 구조체를 연관 짓고 있는 구조체이다.

- request_queue 구조체

  : Zoned block device에 대한 정보를 가지고 있으며, 필드로 디바이스의 zone의 총 개수 및 zone이 conventional인지 sequential인지에 대한 정보를 담고 있다.

- blkcg 구조체

  : block cgroup의 줄임말로 cgroup_subsys_state 구조체를 필드로 가지며 해당 구조체는 cgroup의 id를 가지고 있다.




[그림 13] dm_chunk_work 구조체 및 관계도

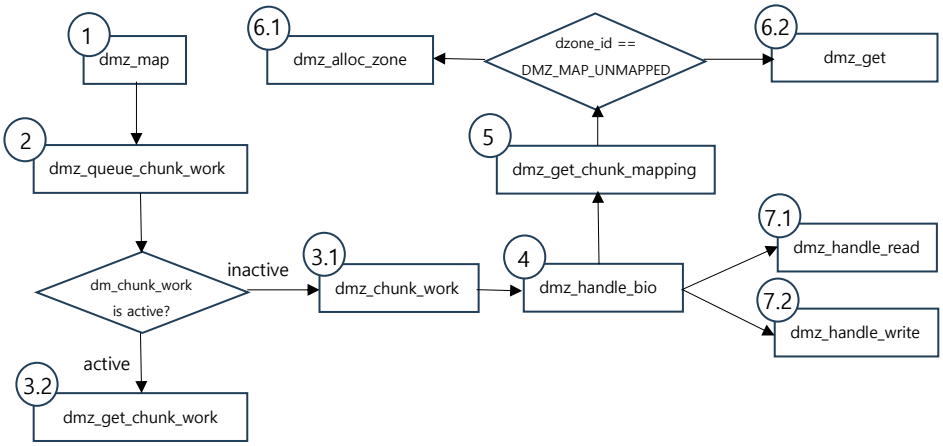
- dm_chunk_work 구조체

  : 청크 단위 작업에 대한 내용을 담고 있는 구조체로 해당 청크에 할당될 bio들에 대한 linked list인 bio_list를 가지고 있으며 work_struct에 대한 필드를 가지고 있다.

- work_struct 구조체

  : dm_chunk_work가 활성화가 안 되어 있지 않다면 활용하는 구조체로 이때 활성화는 해당 청크에 bio 단위의 작업이 이루어졌는가이다.

### 2.2.5. 리눅스 소스 코드 (BIO 처리 과정)




[그림 14] bio 처리 함수 호출 순서도

[그림 14]는 BIO단위 입출력을 처리할 때 호출되는 함수들로 숫자에 따라서 호출된다. 다음 함수들을 파악함으로써 dm-zoned의 문제점인 하나의 청크에 매핑되는 여러 chunk가 매핑되는 것을 해결할 수 있다.

1) dmz_map (struct dm_target *ti, struct bio *bio)
: 새로운 bio를 처리할 때 처음 호출되는 함수로 인자로 받은 dm_target과 bio를 이용해 dmz_target, dmz_metadata 등을 초기화하고 해당 bio를 존단위로 저장하기 위해서 해당 존의 크기에 맞게 bio를 나눈다. 그 후 bio를 처리하기 위해 dmz_queue_chunk_work()를 호출한다.

2) dmz_queue_chunk_work (struct dmz_target *dmz, struct bio *bio)
: cgroup과 bio에 해당하는 청크 번호를 가지고 해당 번호에 대한 dm_chunk_work 구조체가 정의되어 있다면(활성화 유무) 호출될 시 dmz_get_chunk_work()를 호출한다. 정의되어 있지 않다면 해당 번호에 대한 dm_chunk_work를 할당한 후 dmz_chunk_work() 함수가 추후 처리되도록 INIT_WORK()함수를 통해 예약을 한다.

3.1) dmz_get_chunk_work(struct dm_chunk_work *cw)
: bio에 해당 하는 청크가 활성화 되어 있다면 호출되는 함수로 dm_chunk_work 구조

체의 필드인 refcount(참조 횟수)를 늘리는 함수이다.

3.2) dmz_chunk_work (struct work_struct *work)

: dmz_queue_chunk_work()에서 할당한 dm_chunk_work의 필드인 work_struct를 매개변수로 받고 해당 work_struct를 가지는 dm_chunk_work를 가져온 후 dm_chunk_work의 필드인 bio_list에 포함된 bio들을 하나씩 처리하도록 dmz_handle_bio() 함수를 호출한다.

4) dmz_handle_bio (struct dmz_target *dmz, struct dm_chunk_work *cw, struct bio *bio)

: bio를 처리하는 함수로 bio에 해당하는 청크와 매핑된 존을 구하기 위해 dmz_get_chunk_mapping() 함수를 호출한다. 이를 통해 구한 zone을 활성화 시킨 후 bio에 맞는 입출력을 수행한다. (dmz_handle_read, dmz_handle_write 등)

5) dmz_get_chunk_mapping (struct dmz_metadata *zmd, unsigned int chunk, int op)

: 청크와 매핑된 존을 구하는 함수로 이미 청크와 존이 매핑되어 있다면 dmz_get을 호출해서 존을 구하고 매핑된 존이 없고 해당 bio가 쓰기 작업을 하는 요청이라면 존을 할당하기 위해서 dmz_alloc_zone()을 호출한다. 이후 할당된 존과 청크를 매핑시키기 위해서 dmz_map_zone()을 호출한다.

6.1) dmz_alloc_zone (struct dmz_metadata *zmd, unsinged int dev_idx, unsigned long flags)

: 청크와 매핑된 존이 없을 때 호출되는 함수로 사용 가능한 free 존을 구해서 해당 존을 반환한다.

6.2) dmz_get (struct dmz_metadata *zmd, unsigned int zone_id)

: 인자로 받은 zone_id에 해당하는 존인 dm_zone 구조체를 반환한다.

7.1) dmz_handle_read (struct dmz_target *dmz, struct dm_zone *zone, struct bio *bio)

: 읽기 요청을 하는 BIO를 처리하는 함수.

7.2) dmz_handle_write (struct dmz_target *dmz, struct dm_zone *zone, struct bio *bio)

: 쓰기 요청을 하는 BIO를 처리하는 함수.

## 3. 연구 내용

### 3.1. 청크 매핑 구조체 정의

기존 dm-zoned 방식인 청크에서 존으로 매핑되는 구조의 문제는 하나의 청크를 여러 cgroup이 공유한다는 점이다. 이를 해결하기 위해 청크에 할당된 bio들을 cgroup별로 나눠 줄 필요가 있다. 이를 위해 [구조체 1]을 선언해 주었다. 이 구조체를 통해 기존 방식인 청크에서 존으로 매핑되는 방식을 chunk → chunk_group → zone으로 매핑되는 방식으로 바꾸어 주도록 설계했다.
해당 구조체의 필드로는 해당 chunk_group이 사용되었는지 확인을 위해 isUsed, 사용하고 있는 cgroup의 id를 담을 group, 매핑되는 청크 번호인 originChunk가 있다.

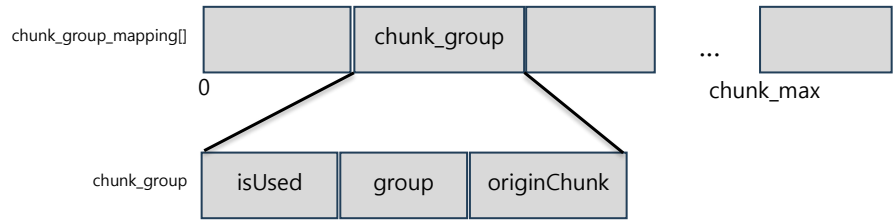
```
struct chunk_group {
        bool isUsed
        int group
        int originChunk
}
```

[구조체 1] chunk_group 구조체

chunk_group 구조체가 청크의 개수만큼 필요하기 때문에 dmz_target 구조체에 청크의 개수만큼 chunk_group 타입의 배열을 [구조체 2]와 같이 할당하도록 하였다. 해당 배열을 그림으로 나타내면 [그림 13]과 같다

```
struct dmz_target {
        ...
        struct chunk_group chunk_group_mapping[chunk_Max]
        ...
}
```

[구조체 2] chunk_group 배열 선언




[그림 15] chunk_group_mapping

### 3.2. 청크 매핑 알고리즘

dm_zoned에서 bio를 처리할 때 처음 호출하는 함수가 [그림 12]에 나와 있는 dmz_map함수이다. 이때 매개변수로 dm_target, bio 구조체를 받는다. 청크 번호는 dm_zoned에서 정의한 [코드 1]과 같이 dm_target과 bio의 필드를 통해서 구한다. 그래서 기존 연구 계획인 [그림 6]처럼 공유하던 청크를 각 cgroup 별로 사용하기 위해서는 bio와 dm_target이 생성되는 부분의 코드를 수정을 해주어야 한다. 하지만 이는 dm_zoned 코드 외의 영역이므로 [그림 14]와 같이 매핑 계획을 수정하였다.

```
#define dmz_bio_chunk(struct dm_target, struct bio)
                ((bio) -> bi_iter.bi_sector >> dmz_zone_nr_sectors_shift(zmd))
```

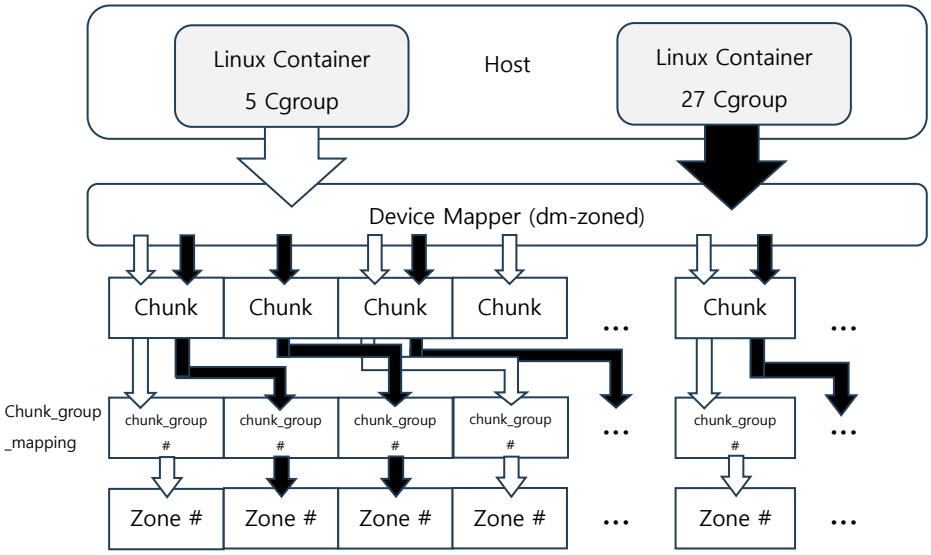
[코드 1] dmz_bio_chunk 함수

기존 dm-zoned의 청크 매핑 방식의 문제점인 하나의 청크에 여러 cgroup의 bio가 포함되는 것을 막기 위해 위에서 만들어 준 구조체를 사용한다. [알고리즘 1]은 dmz_queue_chunk_work에 추가한 알고리즘으로 청크가 이미 할당된 상황에서 위에서 언급한 chunk_group_mapping 배열에 cgroup과 청크 번호에 따라 다시 배열에 할당하는 과정이다. 이를 통해 기존 청크 번호 및 cgroup에 따라 하나의 chunk_group을 할당 받고 있다. [그림 15]는 해당 알고리즘의 작동 과정을 나타내고 있다.

```
cgroup_id = bio->bi_blkg->blkcg->css.id
index = 0
loop_start :
if ( index < chunk_group_mapping_size ) then
        chunk_group = dmz -> chunk_group_mapping[index]
```

```
        if chunk_group.isUsed then
                if ( chunk_group == cgroup_id
                        && chunk_group.originChunk == chunk ) then
                        goto out
                else
                        index++
                        goto loop_start
                endif
        else
                chunk_group.originChunk = chunk
                chunk_group.isUsed = true
                chunk_group.group = cgroup_id
                goto out:
        endif
endif
out:
```

[알고리즘 1] 청크 매핑 알고리즘




[그림 16] 개선한 dm-zoned 방식

## 4. 연구 결과 분석 및 평가

파일 시스템의 입출력을 분석하기 위해 dmesg(diagnostric message) 리눅스 명령어를 사용한다. 이는 시스템 부팅 메시지를 확인하는 명령어로 커널에서 출력되는 메시지를 기록하는 역할을 수행하기 때문에 이 명령어를 사용해서 파일 시스템 입출력을 통한 로그를 확인하였다.

### 4.1. 성능 평가 툴 : FIO

[그림 17], [그림 18]은 FIO를 통해 파일 입출력을 실행한 결과로 제안한 적용 방식으로 I/O Isolation이 잘 이루어졌는지 보여주는 그림이다. 해당 그림은 모든 Bio들 중 일부를 나태내고 있다. [그림 17]를 보면 0번 청크를 사용하는 cgroup 1, 5번이 각각 chunk_group_mapping 배열의 122번 3번 인덱스의 chunk_group에 매핑된 것을 확인할 수 있다. 또한 동일한 청크를 공유하는 Cgroup의 Bio들이 서로 다른 chunk_group으로 매핑되고 각각의 존으로 매핑된 것을 확인할 수 있다.

이를 통해 청크에서 존으로 할당되는 방식에서 chunk → chunk_group → zone 으로 매핑되는 방식이 cgroup 별로 I/O Isolation이 잘 이루어진 것을 확인할 수 있다.

```
[   97.504852] Chunk 112 work start
[   97.504860] Bio Write : Cgroup? 1, chunk? 0, chunk_group? 112, zone? 20
[   97.504939] Bio Write : Cgroup? 1, chunk? 0, chunk_group? 112, zone? 20
[   97.504969] Chunk 112 work finish
[   97.552897] Chunk 112 work start
[   97.552905] Bio Write : Cgroup? 1, chunk? 0, chunk_group? 112, zone? 20
[   97.552987] Bio Write : Cgroup? 1, chunk? 0, chunk_group? 112, zone? 20
[   97.553015] Chunk 112 work finish
```

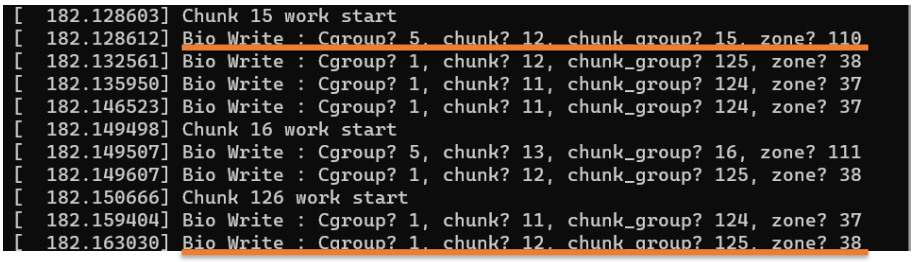
[그림 17] FIO : Cgroup1의 0번 청크 112번 chunk_group, 20번 존으로의 매핑 결과

```
[   86.059513] Bio Write : Cgroup? 5, chunk? 0, chunk_group? 3, zone? 3
[   86.059560] Bio Write : Cgroup? 5, chunk? 0, chunk_group? 3, zone? 3
[   86.059580] Bio Write : Cgroup? 5, chunk? 0, chunk_group? 3, zone? 3
[   86.059599] Bio Write : Cgroup? 5, chunk? 0, chunk_group? 3, zone? 3
[   86.059618] Bio Write : Cgroup? 5, chunk? 0, chunk_group? 3, zone? 3
[   86.059637] Bio Write : Cgroup? 5, chunk? 0, chunk_group? 3, zone? 3
[   86.059656] Bio Write : Cgroup? 5, chunk? 0, chunk_group? 3, zone? 3
[   86.059674] Bio Write : Cgroup? 5, chunk? 0, chunk_group? 3, zone? 3
[   86.059693] Bio Write : Cgroup? 5, chunk? 0, chunk_group? 3, zone? 3
[   86.059712] Bio Write : Cgroup? 5, chunk? 0, chunk_group? 3, zone? 3
```

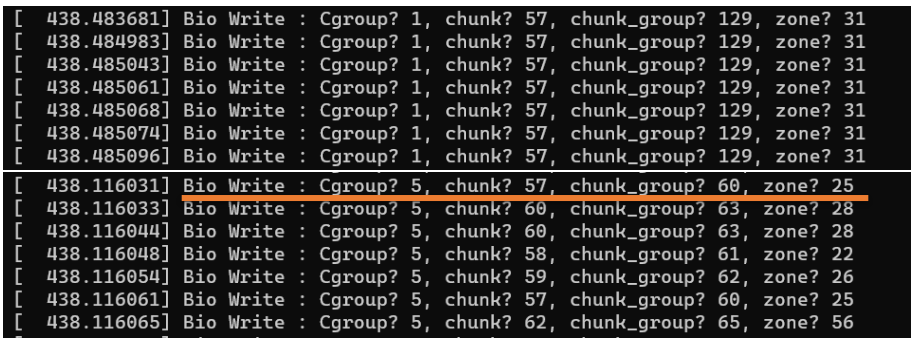
[그림 18] FIO : Cgroup5의 0번 청크에서 3번 chunk_group, 3번 존으로의 매핑 결과

### 4.2. 성능 평가 툴 : IOzone

[그림 19], [그림 20]은 IOzone을 통한 파일 입출력을 실행한 결과이다. 동일하게 I/O Isolation이 잘 이루어졌는지 보여주는 그림이다. [그림 19]은 동일한 12번 청크에 Bio를 매핑한 cgroup 1, 5번이 서로 다른 chunk_group와 존에 매핑되는 결과이다.
[그림 20]은 동일한 57번 청크에 Bio를 매핑한 cgroup 1,5번이 서로 다른 chunk_group과 존을 매핑한 결과이다. 이렇게 IOzone을 통해서도 cgroup별로 I/O Isolation이 잘 이루어진 것을 확인할 수 있다.




[그림 19] IOzone : Cgroup1, 5의 청크 12번 매핑 결과




[그림 20] IOzone : Cgroup 1,5의 청크 57번 매핑 결과

# 5. 결론 및 향후 연구 방향

## 5.1. 결론

본 연구에서는 chunk_group이라는 새로운 구조체를 활용해 하나의 Cgroup이 독립된 청크를 사용할 수 있도록 dm-zoned의 매핑 방식을 수정하였다. chunk_group 구조체를 통해 각 Cgroup이 할당받은 originChunk의 번호에 따라 새로운 Chunk를 할당하였고 사용 중인 청크에는 다른 Cgroup이 연결되지 못하는 방식을 사용하였다. 결과적으로 하나의 Cgroup은 독립된 청크를 사용하는 것을 확인하였다.

## 5.2. 향후 연구 방향

현재 연구에서는 기존에 할당받고 난 후 할당된 결과를 가지고 새로운 할당을 수행하는 방식을 사용하고 있다. 이로 인해 불필요한 추가 할당이 발생했다. 향후 연구에서는 DM-ZONE 코드 수정뿐만 아니라, BIO를 생성하는 부분부터 개입하여 추가 할당을 최소화하는 방안을 적용하면 불필요한 추가 할당을 방지할 수 있을 것이다.

현재 사용 중인 Cgroup과 기존에 할당받은 originChunk를 찾기 위해 chunk_group_mapping 테이블을 완전 탐색을 사용하기 때문에 탐색 시간이 비효율적이다. 해당 테이블의 탐색 알고리즘을 수정한다면 탐색 시간을 줄일 수 있을 것이다.

Cgroup이 기존에 할당받은 청크 개수만큼 각각 새로운 청크에 할당해 주었기에 청크가 과도하게 할당되는 문제가 있었다. 기존의 originChunk가 아닌 Cgroup을 기준으로 Chunk를 새로 할당하는 방식을 사용하면 Cgroup에 할당되는 청크의 개수를 줄일 수 있을 것이다.

동일한 청크를 할당받은 Cgroup에 대해 단순하게 한 Cgroup의 bio를 새로운 청크에 할당하였기에 청크들의 저장 공간 활용이 매우 비효율적이다. 이러한 청크가 제대로 채워지지 않은 상태에서의 파일 입출력을 막기 위한 새로운 할당 방식이 필요하다.

# 6. 개발 일정 및 역할 분담

## 6.1. 개발 일정

| 5월 | | | 6월 | | | | | 7월 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3주 | 4주 | 5주 | 1주 | 2주 | 3주 | 4주 | 5주 | 1주 | 2주 | 3주 | 4주 | 5주 |
| 착수보고서 | | | | | | | | | | | | |
| | ZNS SSD 관련 내용분석 | | | | | | | | | | | |
| | | 개발 환경 이해 및 구축 (cscope, ctags) | | | | | | | | | | |
| | | | | 리눅스 커널 코드 분석 ( 구조체 ) | | | | | | | | |
| | | | | | | | | 리눅스 커널 코드 분석 ( bio의 처리 ) | | | | |
| | | | | | | | | | | | 중간 보고서 | |

[표 1] 개발 일정

| 8월 | | | | | 9월 | | | | | 10월 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1주 | 2주 | 3주 | 4주 | 5주 | 1주 | 2주 | 3주 | 4주 | 5주 | 1주 | 2주 | 3주 |
| 리눅스 커널 코드 분석 ( chunk, chunk_work ) | | | | | | | | | | | | |
| | 1차 알고리즘 설계 ( bio chunk 할당 ) | | | | | | | | | | | |
| | | | 2차 알고리즘 설계 ( bio chunk 매핑 ) | | | | | | | | | |
| | | | | 구조체 설계 ( Chunk_group ) | | | | | | | | |
| | | | | Chunk Mapping 알고리즘 구현 | | | | | | | | |
| | | | | | | | | 테스트 및 디버깅 | | | | |
| | | | | | | | | | 1차 성능 분석 | | | |
| | | | | | | | | | 알고리즘 개선 | | | |
| | | | | | | | | | | 2차 성능 분석 | | |
| | | | | | | | | | | | 최종 보고서 | |

[표 2] 개발 일정

### 6.2. 역할 분담

| 이름 | 역할 |
|---|---|
| 윤건우 | - ZNS SSD 관련 조사<br>- 리눅스 커널 빌드 설정<br>- dm-zoned 코드 분석 : bio 관련, chunk 관련<br>- Chunk – Group 매핑 알고리즘 설계<br>- 성능 평가 툴 조사 및 I/O Isolation 성능 평가 |
| 황인욱 | - ZNS SSD 관련 조사<br>- 리눅스 환경 설정<br>- dm-zoned 코드 분석 : 구조체, bio 관련<br>- 구조체 및 알고리즘 테스트 및 디버깅 |
| 조준서 | - 리눅스 컨테이너 관련 조사<br>- 커널 분석 환경 설정<br>- dm-zoned 코드 분석 : 구조체, chunk 관련<br>- Chunk – Group 매핑 구조체 설계 |

[표 3] 역할 분담

## 7. 산학협력 프로젝트 멘토 의견 반영 내용

| 멘토 의견 | 반영 내용 | 반영 위치 |
|---|---|---|
| 과제 목적 및 필요성 | 1) 본 연구 과제를 통해 달성하고자 하는 목표에 대한 설명 부족<br>(I/O Isolation에 의한 성능 향상 근거)<br>2) 해결하고자 하는 문제에 설명 부족 | 1) 서론에 서술<br>2) 1.2 기존 문제점에 서술 |
| 평가 방식 다양화 | 1) I/O Isolation이 잘 이루어지는가를 보여줄 방법 제시<br>2) 성능 평가 툴(trace-replay, filebench 등) 다양화 | 1) 연구 결과에서 모든 출력문을 보여주지는 못 하였지만 이를 설명해 줄 매핑 방식을 제시<br>2) 2.2.2에 툴 제시(iozone) 및 4에서 사용 후 결과 제시 |

[표 4] 산학협력 프로젝트 멘토 의견 반영 내용

## 8. 참고 문헌


[1] 위키피디아. 솔리드 스테이트 드라이브 [Online]. Available: https://ko.wikipedia.org/wiki/솔리드_스테이트_드라이브 (accessed 2023, Oct. 17)

[2] 손한근. (2019, Apr 18). ZNS SSD, 기존 SSD 와 무엇이 다를까? [Online]. Available: https://news.skhynix.co.kr/post/zns-ssd-existing-ssd-and (accessed 2023, Oct. 17)

[3] 위키피디아. Device mapper [Online]. Available: https://en.wikipedia.org/wiki/Device_mapper (accessed 2023, Oct. 17)

[4] Zone Storage [Oline]. Available: https://zonedstorage.io/docs/introduction (accessed 2023, Oct 16)

[5] westterndigitalcoporation. (2022, Jun 6). dm-zoned-tools [Online]. Available: https://github.com/westerndigitalcorporation/dm-zoned-tools/tree/master (accessed 2023, Oct, 16)

[6] Bowbowbow, (2016, Mar 20) [리눅스] ctags 사용법 [Online]. Available: https://bowbowbow.tistory.com/15 (accessed 2023, Oct. 16)

[7] Chanhyun Park. (2015, Apr 28). [Linux] 소스 분석 툴 cscope 사용법 [Online]. Available: https://harryp.tistory.com/131 (accessed 2023, Oct. 16)

[8] IOzone. IOzone FileSystem Benchmark [Online]. Available: https://www.iozone.org/ (accessed 2023, Oct. 18)


변경된 필드 코드