ZNS를 이용한

파일시스템 성능 개선 연구

Team: OS 을~매나 맛있게요?

부산대학교 전기컴퓨터공학부 정보컴퓨터공학전공

School of Electrical and Computer Engineering, Computer Engineering Major

Pusan National University

2022년 5월 16일

지도교수: 안 성 용 (인)

목차

1 과제 개요

1.1 과제 목표

ZNS상에서의 성능을 개선하기 위해 Zone Target Device Mapper에서의 File Data와 Journal Data가 Zone Namespace SSD의 Zone에 섞이는 비율을 분석하고 Journal Data 저장을 위한 Zone을 지정하여 다른 타입의 데이터를 분리한다.

1.2 과제 배경 및 필요성

SSD(Solid-State Drive)는 HDD(Hard Disk Drive)에 비해 굉장히 빠른 성능을 가진 저장 장치이다. 그러나 현재 flash-based SSD는 수 십 년된 block interface를 가지고 있다. 이로 인해 Capacity Over-Provisioning, page mapping tables을 위한 DRAM(Dynamic Random-Access Memory), Garbage Collection Overheads, 그리고 Garbage Collection을 완화하기 위해 쓰이는 호스트 소프트웨어의 복잡성 측면에서 상당한 비용이 발생한다.

ZNS SSD(Zone Namespace SSD)에서는 여러 응용 프로그램이 각자 정해진 Zone에 순차적으로 데이터를 저장한다. 여기서 Zone은 논리적인 공간처럼 NAND 블록에도 나눠져 있다. 하나의 Zone안에는 비슷한 데이터끼리 모여있으며, 순차적으로 저장했다가 Zone단위로 지우기 때문에 Garbage가 발생하지 않는다. 따라서 SSD에서의 Garbage Collection이 없어도 됨으로써 발생했던 여러 오버헤드를 제거할 수 있다.

기존의 SSD에서는 Data를 저장시에 해당 Data가 어떻게 분포하는지를 전혀 알지 못했다. 그러나 ZNS에서는 Zone의 개념을 도입하면서 Data가 어떻게 저장되는지를 확인할 수 있고 그러므로 journal data와 file data를 분류하여 journal을 위한 PBA random zone을 설정해줌으로서 Space Amplification과 Garbage Collection을 방지하여 성능 향상의 효과를 볼 수 있다.

2 배경지식

2.1 ZNS SSD (Zone Namespace SSD)

새로운 NVMe storage interface인 ZNS(Zoned Namespace)는 고정된 크기의 Zone으로 분할된 LBA(Logical Block Address)을 제공한다. ZNS Storage Interface는 SMR HDD용으로 처음 도입되었으며 최근에는 플래시 메모리 SSD에 채택되었다. ZNS는 이러한 저장 매체의 순차적 전용 쓰기 제약 조건을 고려하여 각 영역을 순차적으로 작성해야 하며 reset 작업 후 재사용할 수 있도록 규정하고 있다. ZNS SSD의 경우 일반적으로 다른 플래시 칩의 여러 플래시 지우기 블록에 영역을 매핑하여 플래시 칩 레벨 병렬화를 활용한다. 각 Zone은 순차적으로 작성되므로 ZNS SSD는 내부적으로 Zone level 논리적-물리적 주소 매핑(즉, Zone과 flash block간의 매핑)을 유지할 수 있다. Zone의 매핑된 Flash block은 Zone reset 시에 완전히 무효화되므로 SSD 내부 Garbage Collection이 필요하지 않으며 Garbage Collection에 의한 Write Amplification도 제거할 수 있다.

2.2 dm-zoned device mapper target(이하 dm-zoned)

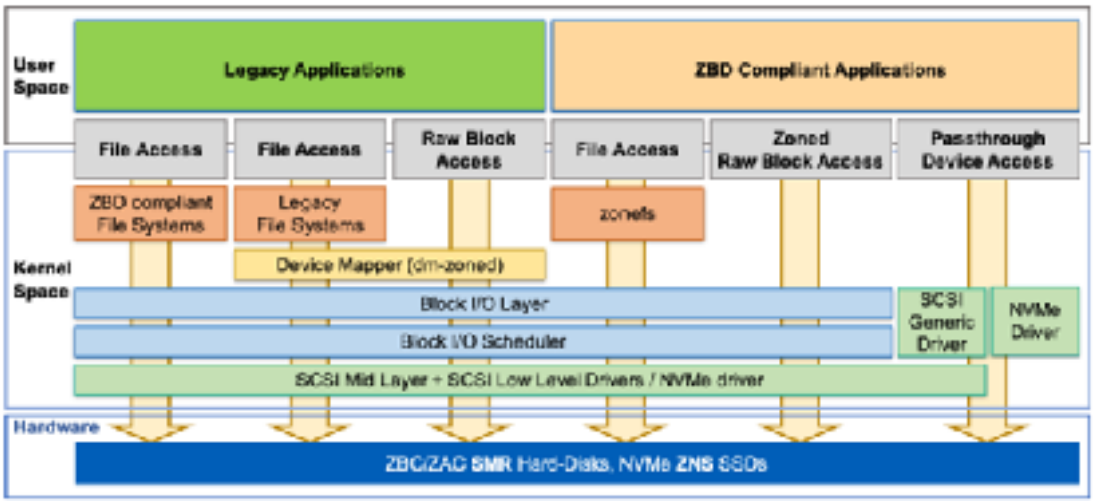

Figure 1. Zone mapping overview of the dm-zoned device mapper target

App 개발자는 다양한 I/O 경로를 통해 Zone Block Device를 사용할 수 있고, 다른 프로그래밍 인터페이스로 제어할 수 있으며, Zone block Device를 다양한 방식으로 노출할 수 있다.

File Acess와 Raw Block Access I/O 경로를 통해 legacy application(ZNS SSD에 맞는 fully-sequential write stream을 구현하도록 수정되지 않은 application)을 수행할 수 있다. 그 중 File Access interface는 응용 프로그램이 데이터를 파일 및 디렉토리로 구성할 수 있도록 파일 시스템이 구현하는 인터페이스이다. File access interface의 두가지 구현중 Legacy File System을 집중한다.

Legacy File system은 수정되지 않은 파일 시스템이 사용되며 장치 순차 쓰기 제약 조건은 zoned block device를 regular block device로 변환하여 device mapper target driver(이하 dm-zoned)에 의해 처리된다. 즉, dm-zoned는 sequential write만 가능한 zoned block devices에 random write access를 제공한다.

2.3 EXT4 file system – journaling

Ext4(extended file system 4)는 linux의 파일 시스템으로, 저장 장치를 논리 블록 단위로 분산하여 오버헤드를 줄이고, 안정성과 전송량을 증가시킨다. Ext4는 시스템 충돌 시 파일 손상을 보호하기 위해 저널링 시스템을 사용하는데, 이를 Ext4 journaling file system이라고 한다.

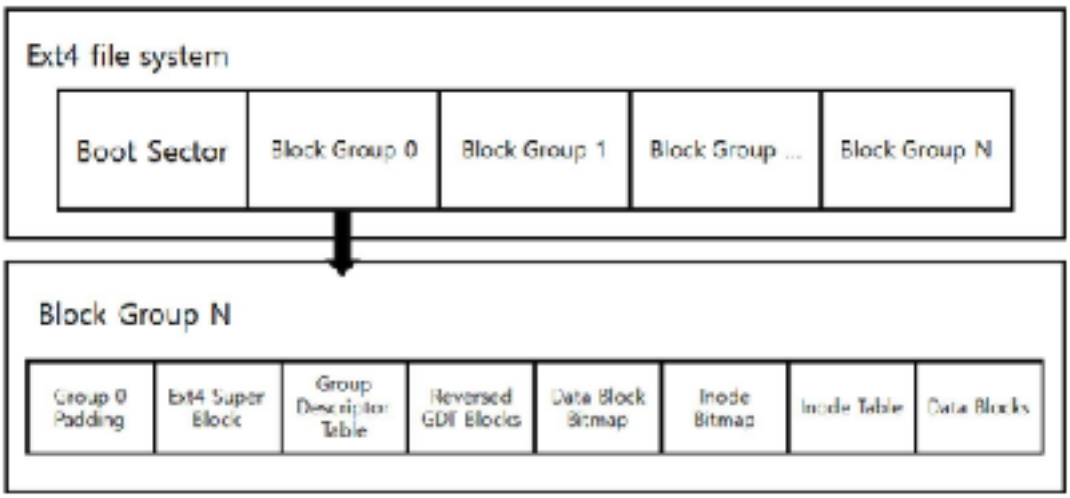

Figure 2. EXT4 Block Group 구조

Ext4 file system은 블록 그룹으로 분할되며, 블록 단위로 저장 공간을 할당한다. 각 블록 그룹은 위 자료와 같은 필드로 나누어진다. 0번째 블록 그룹에는 파일 시스템 전체의 정보를 담은 Super Block과 각 블록 그룹의 정보를 담고 있는 Group Descriptor가 포함된다. 이 두 블록은 오류 발생 시 복구를 위하여 특정 블록 그룹에 사본을 저장하도록 구성되어 있다.

Journaling file system은 디스크에서 변화가 진행 중인 부분에 대한 로그를 저널 안에 기록한다. 이를 통해 인터럽트의 발생으로 write operation이 제대로 끝마치지 못하고 종료되면 저널에 저장된 정보를 이용해 복구할 수 있다.

저널의 모든 블록은 12bytes의 Block Header로 시작한다. 그 다음 내부를 유지하는 주요 데이터가 포함된 Super Block, 저널에서 데이터 블록의 최종 위치를 설명하는 저널 블록 태그를 포함하는 Descriptor Block, 저널을 통해 기록된 데이터를 저장하는 Data Block, 이전에 저널된 데이터의 중복을 막기 위해 사용된 저널 블록에 표시하는 Revocation Block, 저널에 완전히 기록되었음을 표시하는 Commit Block으로 구성되어있다.

Ext4는 파일 시스템의 메타 데이터만을 저널링하는 data=ordered 모드를 기본으로 한다. 사용자에 따라 다른 모드인 data=journal, data=writeback으로 변경할 수 있다. Data=journal 모드는 모든 데이터와 메타 데이터가 저널을 통해 디스크에 기록된다.

3 과제 내용

3.1 dm-zoned write processing 커널 코드 분석

Figure 3. dm-zoned의 sequential write processing 상태 다이어그램

sequential write processing이 어떤 방식으로 chunk BIO를 디스크에 mapping 하는지에 대하여 write buffering scheme을 포함하여 sequential write와 관련된 동작을 수행하는 코드를 분석한다.

Dm-zoned는 write buffering scheme을 구현하여 ZBD(zone block device)의 sequential write가 필요한 data zone에 random write access를 수행한다. Write buffering scheme은 디스크 상에 conventional write zone을 구현하여 write buffering zone으로 사용한다.

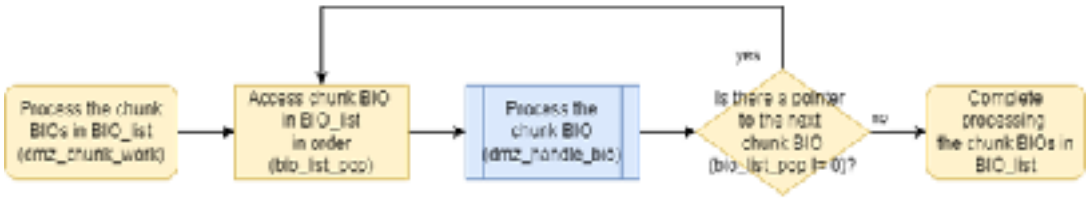

Figure 4. dmz_chunk_work processing routine

dmz_chunk_work -> 여러 개의 chunk bio를 *bi_next를 통해 링크로 생성된 bio_list를 chunk bio 단위로 나눠서 처리하는 함수이다.

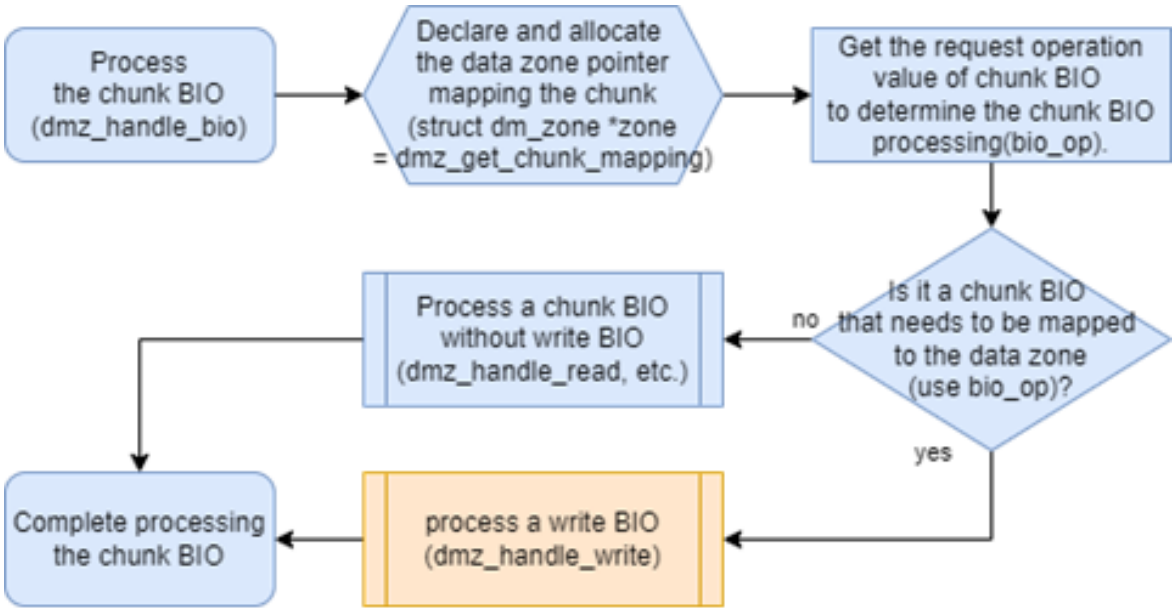

Figure 5. dmz_handle_bio processing routine

Dmz_handle_bio -> dmz_chunk_work에서 접근한 bio_list 내 chunk bio 처리 연산을 결정하는 함수이다. Data zone이 선언 및 할당이 완료된 후, request operation 값에 따라 chunk bio를 data zone에 매핑하는 연산을 수행해야하는 경우, 함수 dmz_handle_write를 호출한다.

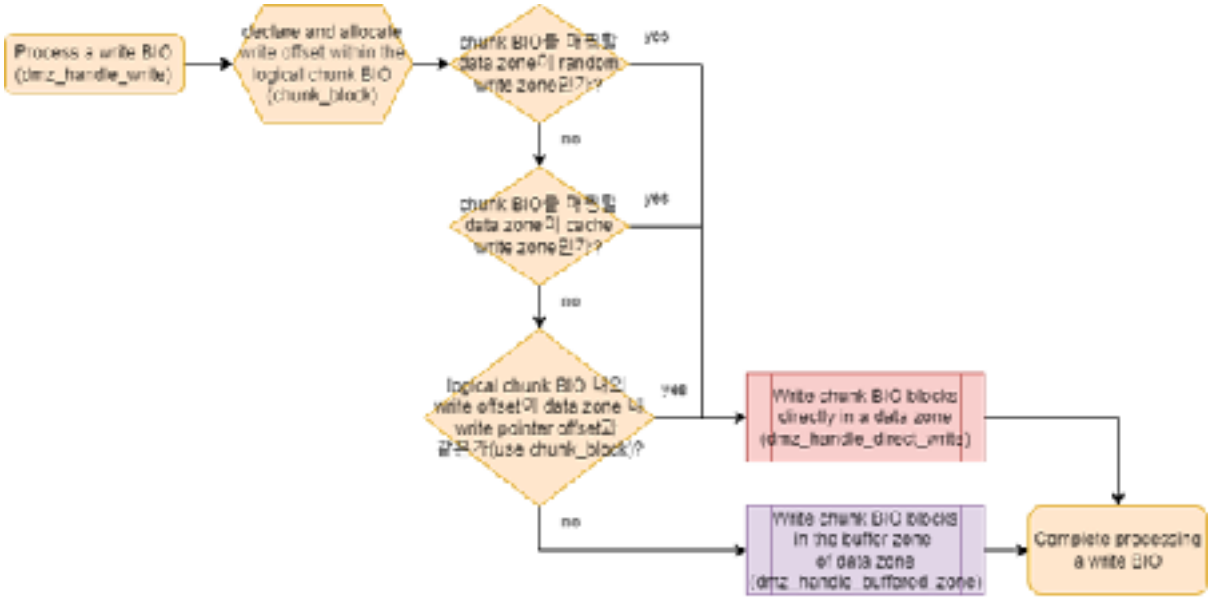

Figure 6. dmz_handle_write processing routine

Dmz_handle_write -> chunk bio를 data zone에 매핑할 방식을 결정하는 함수이다. Dmz_handle_bio에서 할당한 data zone이 random zone이거나 cache zone인 경우, 또는 chunk_block 값과 sequential write zone 내 zone write pointer offset(wp_block) 값이 같은 경우, dmz_handle_direct_write를 호출하고 만약 3가지 조건에 대해 모두 해당하지 않을 경우, dmz_handle_buffered_write를 호출한다.

3.2 dm-zoned reclaim 커널 코드 분석

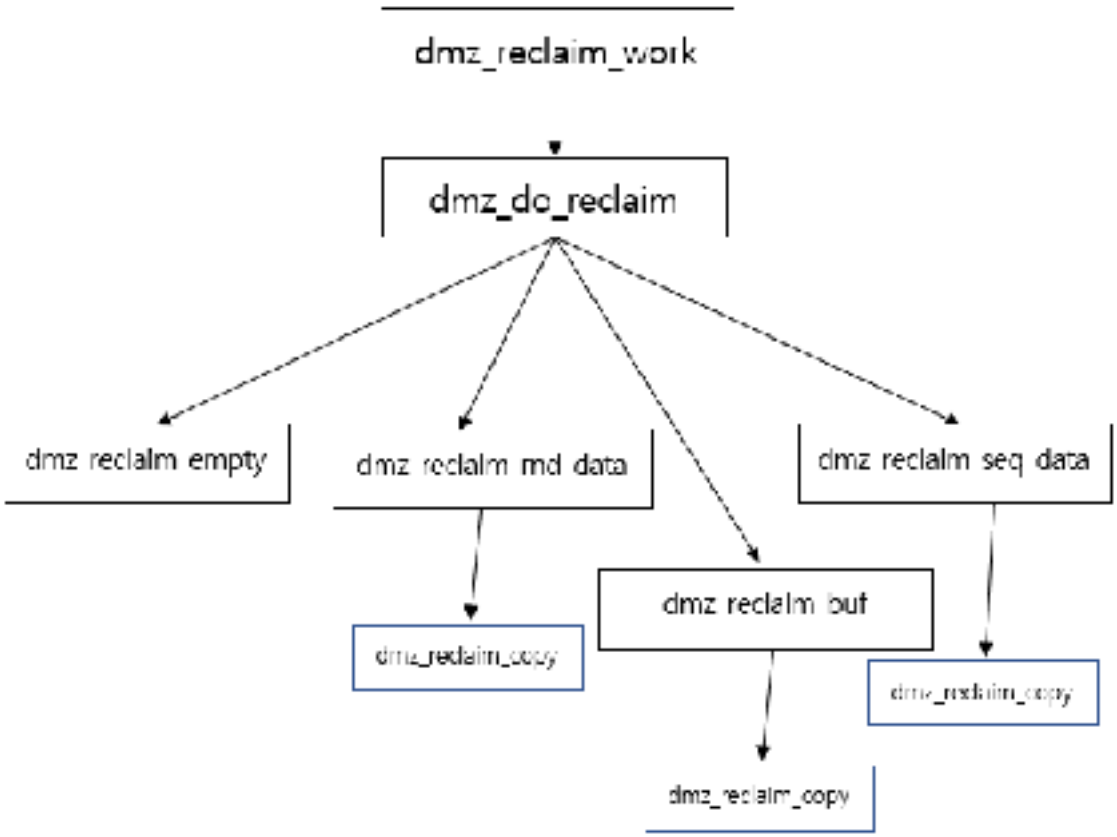

Figure 7. dm-zoned reclaim 동작 call path

dmz_reclaim_work로 reclaim 동작 수행 명령이 들어오면 dmz_do_reclaim 함수를 수행한다.

Dmz_do_reclaim이 실행되면 버퍼의 상황 / data zone의 상황에 따라

1 비어있는 zone을 reclaim할지

2 버퍼에서 data zone으로 block을 옮길지

3 data zone에서 버퍼로 block을 옮길지

4 random data zone에서 sequential zone으로 옮길지를 결정한다.

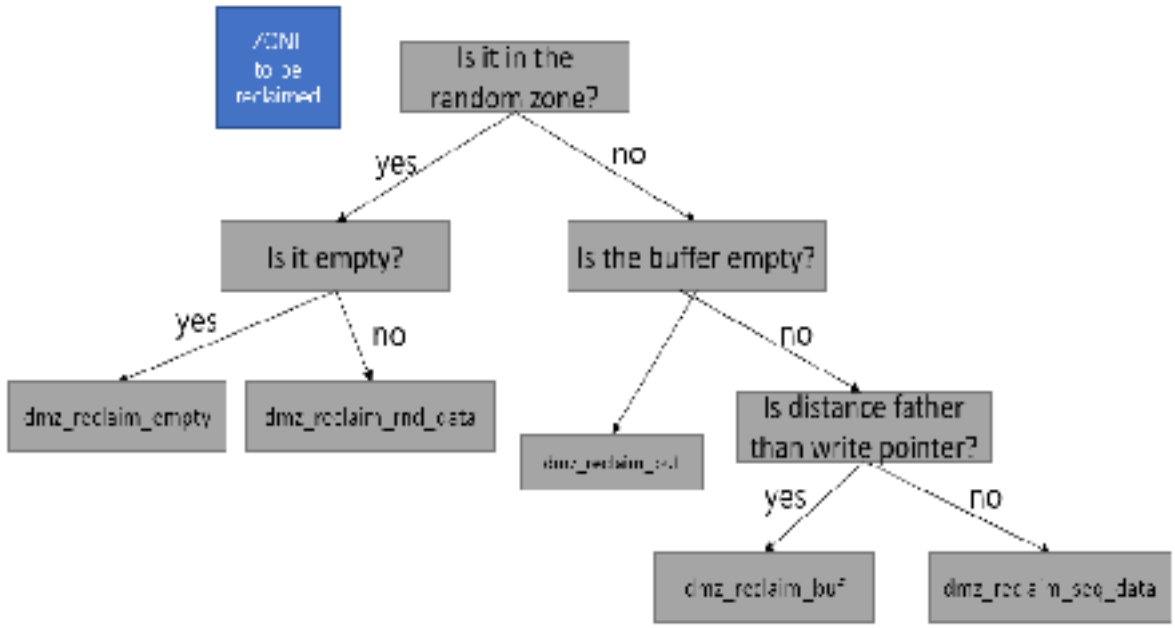

Figure 8. zone reclaim 동작 결정 algorithm

Dmz_do_reclaim은 zone의 여러 조건을 확인 후 그에 상응하는 동작을 수행한다.

먼저, dmz_get_zone_for_reclaim 함수를 통해 reclaim 대상이 될 zone을 가져온다.

Reclaim할 zone을 선택하는 규칙은 이렇다.

1) Free한 sequential zone이 하나도 없는 경우

Random zone을 reclaim하려면 (dmz_reclaim_rnd_data) free sequential zone이 필요하다.

따라서 sequential zone에서 reclaim 대상이 될 zone을 선택한다.

2) 적어도 하나의 free sequential zone이 있는 경우

Access time이 가장 오래된 random write zone을 reclaim 대상으로 선택한다.

받아온 zone이 random zone일 경우 먼저 비어있는지 확인한다.

비어있으면 dmz_reclaim_empty 함수를 호출한다.

유효한 block이 있으면, random data zone을 free sequential zone으로 옮기는 dmz_reclaim_rnd_data를 호출한다.

3) Zone이 sequential인 경우

Buffer로 사용하고 있는 zone이 비어있는지 확인 후 비어있으면 buffer에서 data zone으로 유효한 block을 옮기는 dmz_reclaim_buf 함수를 호출한다.

비어있지 않다면 buffer에서 first valid block의 위치(distance)를 계산한다.

Distance가 write pointer와 같거나 멀리 있으면 (distance >= write pointer) buffer에서 sequential zone으로 바로 이동시키는 것이 가능하므로 dmz_reclaim_buf를 호출한다.

Distance 값이 write pointer보다 작은 경우 buffer에서 sequential zone으로 옮기는 것이 불가능하므로 sequential zone에서 buffer로 옮긴 후 자연스럽게 reclaim되도록 한다.

4 과제 구현

4.1 구현 목표

본 연구는 ext4 file system의 journaling에 관점을 두었다.

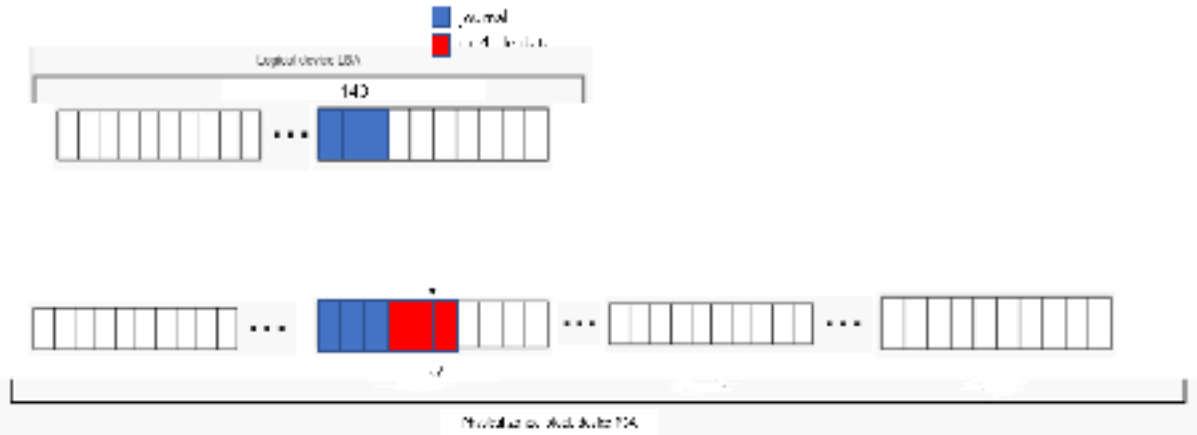

Figure 9. 첫번째 접근시 journaling data가 LBA 140 -> PBA 37

EXT4 file system journaling에서는 LBA(Logical Block Address)에서 PBA(Physical Block Address)로 journal data가 mapping된다. 만약 LBA에서 해당 chunk 140을

다시 접근하였을 때 해당 데이터는 PBA의 전과 같은 위치에 접근이 불가능하다. 그러므로 새로운 PBA 81에 접근 후 이전 PBA 37의 데이터를 invalid화 시킨다.

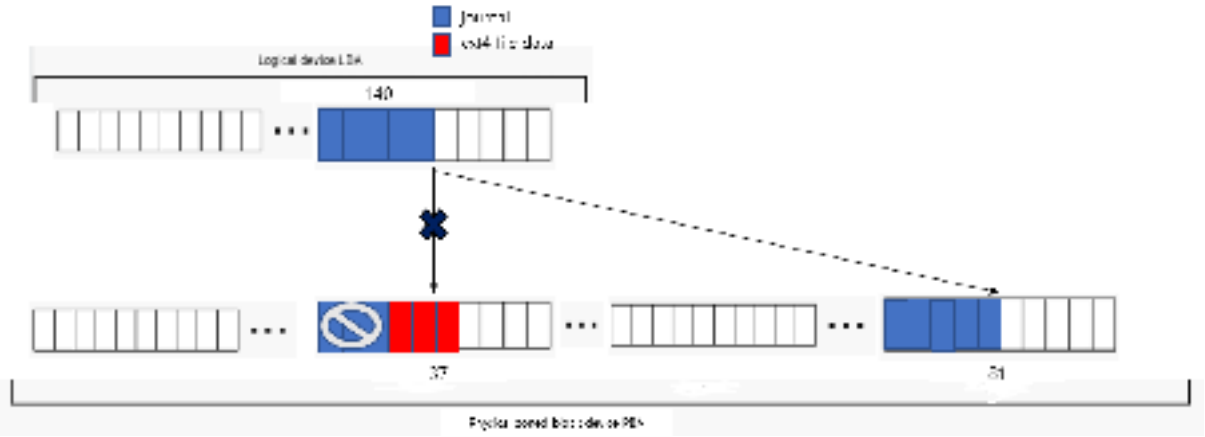

Figure 10. 재접근시에 journaling data가 LBA 140 -> PBA 81

여기서 PBA 37에 journal data가 invalid한 상태에서 일반 file data와 함께 섞여 있을 시에 해당 Zone을 재사용하기 위해서는 Zone reset과정이 필요하고 Zone reset을 위해서는 Garbage Collection이 필요한데 이 동작은 Write Amplification을 발생시켜서 성능 저하를 일으킨다.

여기서 본 연구의 목적으로 journaling을 위한 random zone을 따로 지정해준다면 그 해당 Zone에는 journal data만 존재할 것이고 해당 zone이 invalid한 데이터로 가득 찰 시에 zone reset을 시키면 재사용이 가능하니 Garbage Collection이 필요가 없고 그로 인해 Write Amplification의 발생을 방지함으로서 성능을 개선시킨다.

4.2 구현 내용

Include/linux/blk_types.h

```
enum {
        .....
        BIO_EXT4_FILE,      // bio flags for ext4 file data
        BIO_EXT4_JRNL,      // bio flags for ext4 journal data
```

Enum에 file data인지 journal data인지 구별할 수 있는 flag를 설정한다.

fs/iomap/direct-io.c

```
Struct iomap_dio{
        ....
        Int is_ext4_file;
```

Iomap_dio 구조체에 is_ext4_file이라는 맴버를 넣는다.

```
__iomap_dio_rw() {
        ....
        dio->is_ext4_file = 1;
}
Iomap_dio_bio_actor(){
        ....
        If(dio->is_ext4_file == 1)
                bio_set_flag(bio, BIO_EXT4_FILE)
```

Dio에 is_ext4_file의 값을 1로 set하고 그 값이 if문을 충족할 시에 해당 bio에 BIO_EXT4_FILE 값만큼 shift해준다.

Include/linux/buffer_head.h

```
Struct buffer_head{
        ....
        Int is_ext4_jrnl;
```

};

Buffer_head 구조체에 is_ext4_jrnl 맴버를 넣는다.

fs/jbd2/commit.c

```
jbd2_journal_commit_transaction{
        ....
        bh->is_ext4_jrnl = 1;
        submit_bh(....);
```

Journal을 commit하는 jbd2_journal_commit_transaction함수에서 생성되는 buffer_head bh에 ext4_is_jrnl 값을 1로 set하고 submit_bh 함수로 해당 bh를 submit한다. Submit_bh 함수는 bio를 생성하면서 bh의 is_ext4_jrnl값을 물려준다.

Fs/buffer.c

```
Submit_bh_wbc(){
        ....
        If(bh->is_ext4_jrnl == 1)
                Bio_set_flag(bio, BIO_EXT4_JRNL);
```

Buffer_head에 is_ext4_jrnl = 1이면 bio에 BIO_EXT4_JRNL의 flag만큼 shift해준다.

Drivers/md/dm-zoned-target.c

```
Case REQ_OP_WRITE:
        ....
        If(bio->bi_disk->disk_name[0] == 'd' && bio_flagged(bio, BIO_EXT4_FILE) && !dmz_is_seq(zone))
        If(bio->bi_disk->disk_name[0] == 'd' && bio_flagged(bio, BIO_EXT4_FILE) && dmz_is_seq(zone))
```

Write request가 들어왔을 시에 조건문을 통해 bio가 file data인지 journal인지 그리고 sequential인지 random인지를 알 수 있다.

5 분석 평가

5.1 분석 환경

분석은 가상머신 FEMU상에서 ZNS SSD를 emulate한 환경위에서 실행한다.

```
-device femu,devsz_mb=4096,femu_mode=3 \
-device femu,devsz_mb=1024,femu_mode=1 \
```

Figure 10. ./run-zns.sh disk 설정

Random zone을 위해 blackbox SSD 1GB, Sequential zone을 위한 ZNS SSD 3GB를 설정하였다.

Ext4 file system을 해당 디스크에 마운트 하였고 fio(flexible i/o tester)를 이용하여 분석을 하였다.

5.2 분석 결과

Figure 11. Zone 별로 내려온 File Data의 fio 수

Figure 12. Zone 별로 내려온 Journal Data의 fio 수

40개의 zone 중에 0~7까지 1GB의 random zone에서는 journal data가 찍히지 않는 것을 확인하였고 sequential zone에서 각 zone에서의 file과 journal data의 비율

을 확인할 수 있었다.

그리고 chunk 번호를 추가로 출력하여 분석해본 결과 journal이 chunk 14번을 통해서 zone에 내려오는 것을 확인하였고 chunk 14번을 기준으로 zone에 내려오는 비율을 확인 해본 결과 해당 그래프에서 확인할 수 있다.

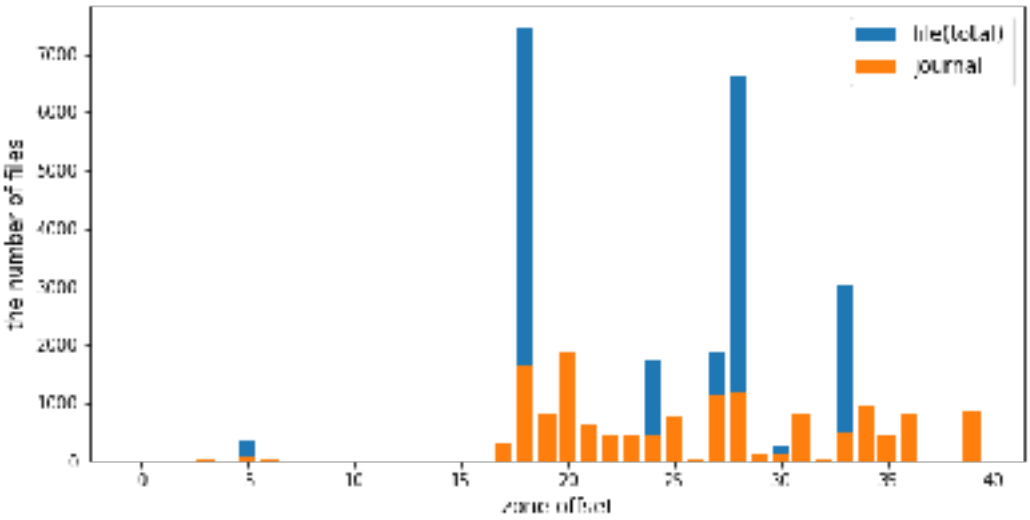

Figure 13. LBA Chunk 14번에서 내려온 Journal/File Data 의 fio 개수

6 추가 개발 예정

현재 각 zone에서의 file, journal data의 정보를 확인하였고 향후 journal과 file data를 zone 별로 구별할 예정에 있다.

7 추진 체계

7.1 구성원 별 역할

이름	역할 분담
	- 보고서 작성 및 자료 수집 - Emulated ZNS SSD 작업환경 구축

임경민	- 커널 코드 수정 - 성능 측정 및 분석 - 논문 분석
이선후	- 보고서 작성 및 자료 수집 - Emulated ZNS SSD 작업환경 구축 - 커널 코드 수정 - 논문 분석 - 성능 측정 및 분석
김상현	- 보고서 작성 및 자료 수집 - Emulated ZNS SSD 작업환경 구축 - 커널 코드 수정 - 논문 분석 - 성능 측정 및 분석

7.2 개발 일정

개발 내용	6월		7월					8월					9월				
논문 분석	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
ZNS SSD 특징 및 구조 이해																	
FEMU 작업 환경 구축																	
중간 보고서																	
Linux 커널 코드 분석																	
커널 소스																	

수정																	
최종 보고서																	
졸업 과제 발표 준비																	

8 참고문헌

[1] Matias Bjørling, Western Digital; Abutalib Aghayev, The Pennsylvania State University; Hans Holmberg, Aravind Ramesh, and Damien Le Moal, Western Digital; Gregory R. Ganger and George Amvrosiadis, Carnegie Mellon University, "ZNS: Avoiding the Block Interface Tax for Flash-based SSDs" – USENIX

[2] Kyuhwa Han, Sungkyunkwan University and Samsung Electronics; Hyunho Gwak and Dongkun Shin, Sungkyunkwan University; Jooyoung Hwang, Samsung Electronics, "**ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction" – USENIX**