Custom DFTL Simulator : Multi-Stream SSD

반도체시스템공학과 2017313016 윤정섭

1. Introduction

기존에 구현했던 DFTL Simulator에 입력 데이터의 Workload에 따라 분리하여 저장, 관리하도록 구현하여 성능을 향상시킨다. 그 후, 기존의 것과 성능을 비교해본다.

2. Theoretical background

2.1 Data Buffer

Data Buffer 는 적은 양에 데이터를 NAND 대신에 RAM 에 저장해 두어 데이터의 읽고 쓰거나 변경할 때의 효율성을 증가시킨다. RAM 은 NAND 와 달리 수정 가능하고, 접근 속도가 훨씬 빠르다. 따라서 자주 접근해야 하는 특정 데이터에 대해, RAM 에 저장하여 접근 명령을 처리하는 속도를 향상시킨다. 다만 Data Buffer 는 설명한 것처럼 RAM 의 용량을 차지하기 때문에, data buffer 의 크기를 너무 크게 할 경우 그만큼의 overhead 를 갖게 되므로 주의해야 한다.

2.2 Multi-streamed SSD

Multi-streamed SSD 는 Figure 1 과 같이 stream 인터페이스 지원을 통해 stream 별로 flash block 에 데이터를 저장할 수 있는 SSD 를 말한다. 각 stream 별로 flash block 에 데이터를 저장하게 되면 Garbage Collection (GC) overhead 를 줄일 수 있어 성능이 향상되는 효과를 가져온다.

좀 더 자세히 살펴보면, GC 를 할 때 victim block 안에 valid page 가 적을수록 효율적이다. GC 라는 작업 자체가 invalid page 의 수를 줄여 NAND 의 공간을 확보하는 목적이기 때문에, 일차적으로 victim block 안에 valid page 가 많다는 것은 overhead 가 크다는 것을 의미하게 된다. 따라서 우리는 데이터를 저장할 때, 빠르게 invalid 가 될 확률이 높은 데이터끼리, 그렇지 않은 데이터끼리 모아서 같은 block 내에 저장하는 것이 유리하다. 빠르게 invalid 가 될 데이터, 즉 lifetime 이 짧은 데이터끼리 모아두면, 그 block 의 데이터는 다른 block 에 비해 많은 invalid page 를 보유하게 될 것이고, GC 의 victim 으로 선정되었을 때, overhead 가 적어진다. 마찬가지로 lifetime 이 긴 데이터끼리 모아두면, 그 block 에는 invalid page 가 적을 것이고, victim block 으로 선정될 확률 자체가 낮아지게 되는 것이다. 결론적으로 우리는 각 stream 별로 data 의 lifetime 이 유사하기 때문에, 이를 통해 GC 의 overhead 를 줄이도록 SSD 를 설계할 수 있다는 것이다.

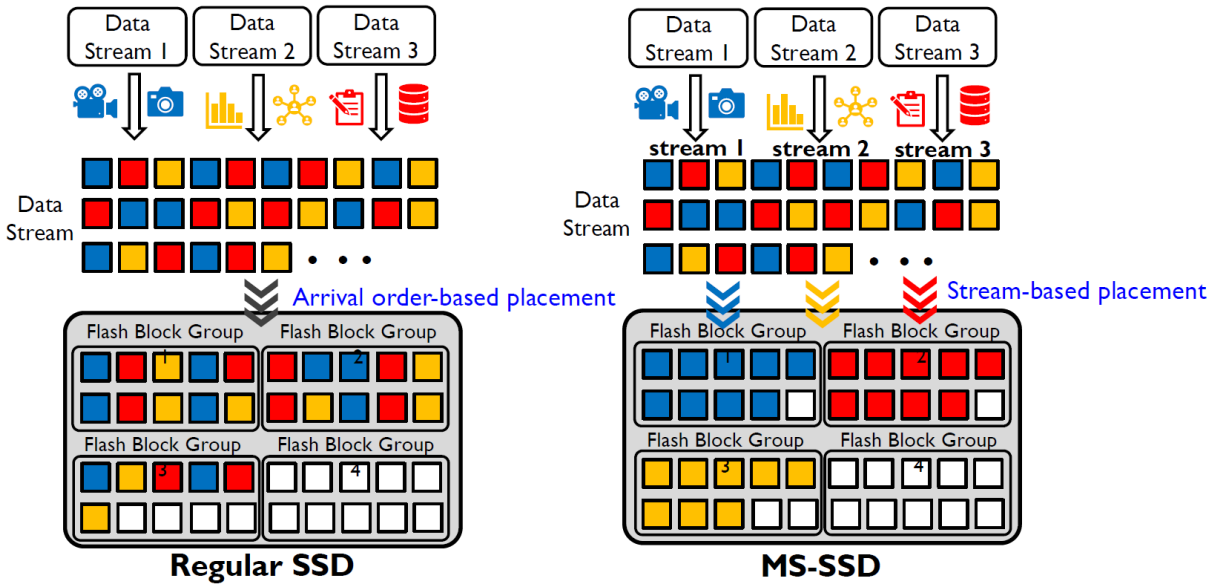

Figure 1. Comparison between Regular SSD and Multi-streamed SSD

2.3 Disk Defragmenter

디스크 조각 모음 (Disk Defragmenter)은 SSD 이전 HDD 에서 자주 사용되던 기술로, 조각나 있는 데이터를 한 곳으로 모아주는 기능이다. 데이터의 동시 입출력, 저장공간 부족, 데이터의 삭제 등 여러 가지 이유로 인해 같은 파일의 데이터가 sequential 하게 쓰이지 않을 수 있다. 이는 HDD 에서 파일을 읽어오는데 걸리는 시간이 길어지는 요인으로 작용할 수 있다. HDD 의 구조가 Figure 2 와 같이 나타난다.

HDD 는 데이터를 물리적인 방법으로 읽어온다. Figure 2 에 나타난 Head 가 Flatter 에 저장된 데이터를 직접 읽어오는 방식으로 작동하는데, 따라서 읽어야 하는 데이터가 Flatter 상에서 멀리 떨어져 있으면, 읽는데 시간이 오래 걸릴 수 밖에 없게 되는 것이다. 따라서 이를 해결하기 위해 디스크 조각 모음이 필요하다. 현재 사용되는 SSD 에서는 데이터를 물리적인 방법을 통해 읽어오지 않기 때문에, 디스크 조각 모음을 통해 성능 향상이 거의 나타나지 않아 그 기능의 중요성이 많이 떨어졌다

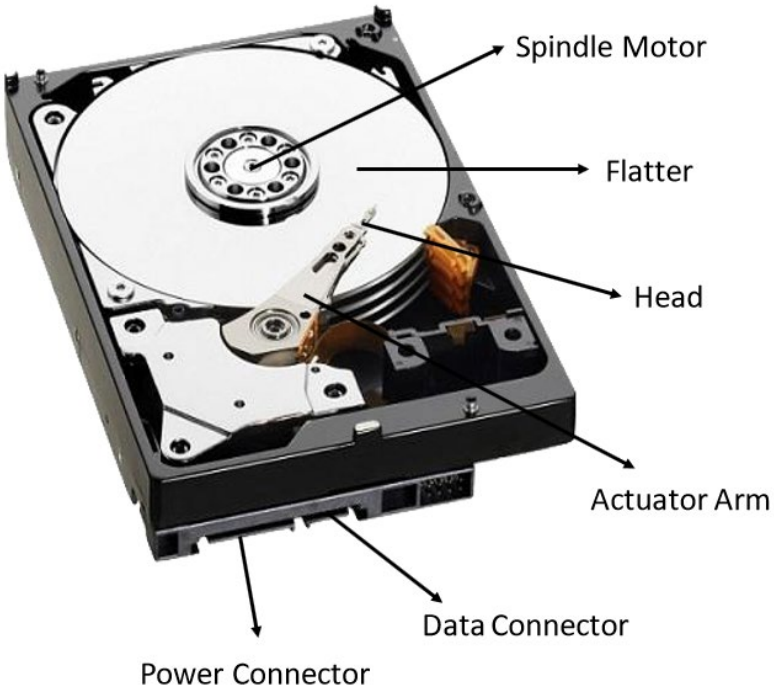

Figure 2. Structure of HDD

.

3. Main Concept

Multi-stream SSD 와 유사하게 동작하기 위하여, Host 로부터 write command 를 받을 때 그 데이터의 workload type (Hot, Cold) 을 함께 입력 받아 그에 따라 처리 가능하도록 구현하였다. 추가적으로, Disk Defragmenter 이 내부적으로 가능하도록 하여 더 효율적으로 데이터를 저장할 수 있도록 하였다.

3.1 Workload Separation

먼저 workload separation 은 physically 되어 있지 않고, logically 되어 있다. 좀 더 자세히 설명하면, SSD 는 각 workload type 의 데이터가 얼마나 들어올지 모르기 때문에, 이를 미리 NAND 상에 구역을 나누어 두는 것은 매우 비효율적이다. 실제로 Host 에서 Hot 데이터만 들어올 수도 있고, 이러한 경우에 미리 구역을 나누어 놓았다면 Cold 구역은 전혀 사용하지 못하는 구역이 되어 버리기 때문이다. 이러한 문제를 해결하기 위해, 우리는 각 workload type 이 block 을 필요로 할 때마다 할당해주는 형식으로 진행된다. 각 block 의 상태는 DRAM 에 저장된다. 기존의 block 의 상태는 비어 있다는 상태의 Free, L2P 정보가 들어있다는 의미의 Trans, 데이터 정보가 들어 있다는 의미의 Data 의 3 가지 상태였지만, 이제는 Trans 가 Trans_hot, Trans_cold 로 Data 또한 Data_hot, Data_cold 로 나누어져 총 5 가지의 상태가 될 수 있다.

이렇게 각 데이터가 분리되어 저장, 관리되기 위하여 GC 또한 workload 에 따라 다르게 이루어진다. GC 가 trigger 되는 시점은 이전과 동일하다. 그러나 workload 별 데이터의 확실한 분리를 위해 GC 의 victim block 선정에서 변화가 있다. GC 에서, victim block 은 GC 가 'trigger 되는 시점의 write 되는 데이터의 workload type'과 동일한 상태의 block 중에서 선정한다. 만약 victim block 의 상태와 write 데이터의 workload type 이 다르다면 결과적으로 workload 에 따른 separation 이 망가지게 되기 때문이다. 이는 GC, MAP_GC 에서 동일하게 적용된다. 하지만 이러한 방식은 특정 상황에서 문제를 일으킨다.

3.2 Problems in GC

아래 Figure 3 와 같은 경우를 생각해보자. GC 에서 victim block 를 선정해야 하는데, 우연히 해당 workload 의 모든 block 에 invalid page 가 존재하지 않는 경우이다. 사실 이는 꽤 자주 일어날 수 있는 경우이다. 그림과 같이 NAND 가 꽉 찬 상태에서 Cold 데이터가 들어오는 상황을 가정하자. 보통 Cold block 에는 invalid page 수가 적고 Hot block 에는 많기 때문에, Host 가 보기에는 Hot block 의 invalid page 때문에 NAND 내부의 공간이 많이 남은 것으로 판단된다. 따라서 Cold data 를 계속해서 write 하라는 명령을 보낼 수 있고, 반면 실제로 NAND 안에서는 앞서 설명한 정책에 따라 cold block 에서만 GC 가 일어나는데, invalid page 가 없어 더 이상 쓸 자리가 없어지는 것이다. 이를 해결하기 위한 방법을 다음 절에서 설명하겠다.

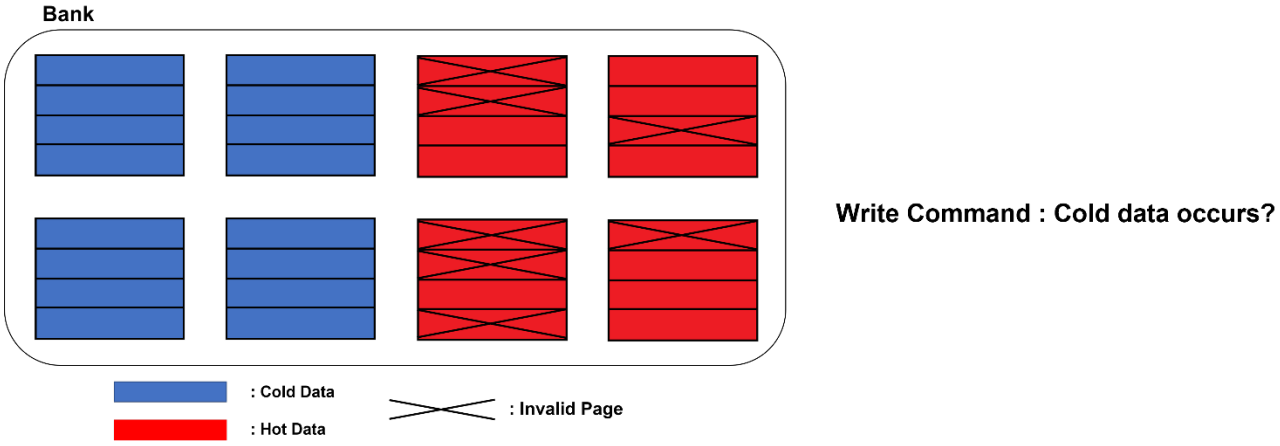

Figure 3. Corner case of Multi-Stream SSD

3.3 Solution : Disk Defragmenter

위에서 설명한 문제점을 해결하기 위해 디스크 조각 모음 (Disk Defragmenter)의 방식을 도입했다. 바로 NAND 에서 각 block 의 valid page 만 모아주는 방식으로, page copy 가 많이 일어나 시간이 오래 걸리지만 한번에 모든 block 의 invalid page 를 없애고, free block 을 생성할 수 있다는 장점이 있다.

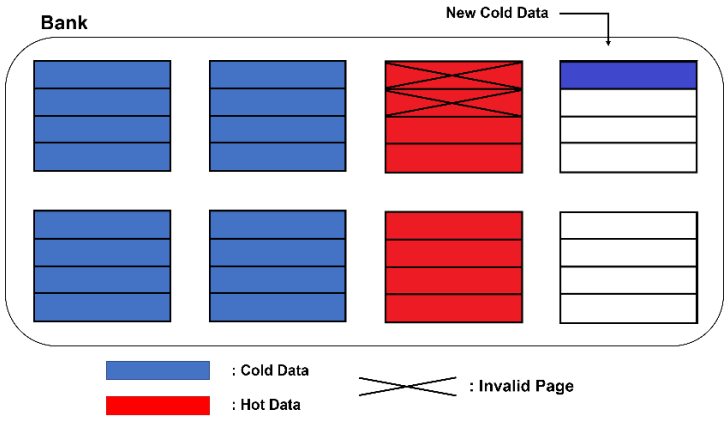

Figure 4. Solved the problem of corner case

위 Figure 3 의 예시를 다시 살펴보면, 저 상황에서 Hot block 들에 대해 disk defragmenter 를 진행하게 된다. Hot block 은 invalid page 가 많기 때문에 disk defragmenter 을 하게 되면 그만큼 free block 이 생성되기 때문에, 이 free block 에 새로 들어온 Cold data 를 할당할 수 있다. 이와 같은 방법으로 위에서 제시한 문제점을 해결하였다.

4. Code Implementation

Code Implementation 에서는 코드 전체를 설명하기보단 기존의 DFTL Simulator 와 비교했을 때 달라진 점을 위주로 설명하도록 하겠다. 코드의 캡쳐 사진 또한 제출하는 파일인 ftl.c 가 아닌 다른 파일들의 코드를 위주로 첨부하도록 하겠다. 모든 코드는 https://github.com/hissubi/DFTL_Simulator 에서 확인할 수 있다.

4.1 DRAM Usage

```
int n_page_invalid[N_BANKS][BLKS_PER_BANK];
p_data next_page[N_BANKS][5];

int n_data_block[N_BANKS];
int n_trans_block[N_BANKS];
int block_state[N_BANKS][BLKS_PER_BANK];

cmt_data CMT[N_BANKS][N_CACHED_MAP_PAGE_PB];
p_data GTD[N_BANKS][N_MAP_PAGES_PB];
int cache_count[N_BANKS][N_CACHED_MAP_PAGE_PB];

buff_data buffer[N_BUFFERS];
int buffer_sector_bitmap[N_BUFFERS][SECTORS_PER_PAGE];
int buffer_empty_page = N_BUFFERS;
```

Figure 5. DRAM usage

```
typedef struct _p_data{
    u32 PBN, PPN;
} p_data;
p_data * PMT;
typedef struct _cmt_data{
    int map_page;
    u32 data[N_MAP_ENTRIES_PER_PAGE];
    int dirty;
    int state;
} cmt_data;
typedef struct _buff_data{
    int lpn; char state;
    u32 data[SECTORS_PER_PAGE];
} buff_data;
```

Figure 6. Structure used in DRAM usage

위 두 Figure 5 에서 Simulator 구현에 있어서 사용한 metadata 를 볼 수 있다. 이 데이터들은 결국 SSD 안의 DRAM 에 저장되는 데이터라고 생각할 수 있다. Bank 수, bank 당 block 수, block 당 page 수,

page 당 sector 수에 따라 그 크기가 달라진다. DFTL 을 기초로 한 SSD 이니 만큼, DRAM 에 저장하는 데이터 중 스케일이 L2P table 만큼 큰 metadata 가 없어 상대적으로 DRAM 을 적게 소모함을 알 수 있다. 다만, data buffer, workload separation, disk defragmenter 등 추가적인 기능 구현으로 인해 기존의 DFTL Simulator 에 비해서는 DRAM 의 사용량이 늘어났다.

4.2 Data buffer

Data buffer 구현을 위해 metadata 에 데이터와 해당하는 lpn 을 저장해주는 buffer 배열, 각 buffer page 의 sector 별 empty/full 여부를 표시하는 buffer_sector_bitmap 배열, buffer 에 빈 페이지 개수를 저장하는 buffer_empty_page 까지가 추가되었다.

함수에서는 data 와 관련된 ftl_read, ftl_write 에 많은 변화가 생겼다. 먼저 ftl_read 에서, NAND 에 저장된 data 를 읽어 오기 전, buffer_read 함수에서 buffer 에 해당 data 가 있는지 검사한다. buffer_read 의 역할을 좀 더 정확히 살펴보자. buffer 에 해당 lpn, offset 에 대한 정보가 있는지를 확인하여, 있는 경우 read_buffer 에 저장하여 ftl_read 에 넘겨준다. 이때, buffer 는 sector 단위로 데이터를 관리하기 때문에, 넘겨주는 read_buffer 에 어떤 sector 가 valid 한 sector 인지에 대한 정보를 sector_bitmap 에 담아 함께 ftl_read 에 넘겨준다. buffer_read 의 return value 는 입력된 모든 범위의 데이터가 buffer 에 존재하여 모두 read_buffer 를 통해 넘겨줬을 경우 1 을, 그렇지 않은 경우 0 을 return 한다. 이렇게 넘겨 받은 정보를 토대로 ftl_read 는 host 에게 데이터를 넘겨준다.

ftl_write 는 ftl_write, ftl_write_direct, ftl_flush 의 세 함수로 나누어졌다. 먼저, ftl_write 에서는 입력받은 data 를 buffer 에 쓴다. 이 과정에서, 이미 buffer 에 동일한 lpn 의 데이터가 저장되어 있는 경우 해당 데이터를 수정하고, 그렇지 않은 경우 새로운 buffer 공간에 데이터를 써준다. 이때, buffer 가 꽉 찼을 경우 ftl_flush 를 통해 buffer 를 비워주고, 해당 작업을 이어나간다. ftl_flush 는 buffer 에 쓰여 있는 data 들을 NAND 에 옮겨 쓰기 위해 데이터를 가공하여 ftl_write_direct 에 넘겨준다. buffer 는 sector 단위로 관리를 하기 때문에, page 단위로 관리하는 NAND 에 옮겨 쓰기 위해서는 비어 있는 sector 에 대한 처리가 필요하다. 비어 있는 sector 에 해당하는 데이터가 이미 NAND 에 저장되어 있는 경우에는 읽어와서 자리를 채우고, 그렇지 않은 경우에는 0xFFFFFFFF 로 빈자리를 채워 데이터를 page 단위로 가공하여 ftl_write_direct 로 넘겨준다. ftl_write_direct 는 기존 DFTL Simulator 와 동일하게 NAND 에 쓰던 동작을 하는 함수이다.

4.3 Modification for workload

Workload data 를 받아서 처리하기 위해서 여러 파일을 수정하였다. 가장 먼저 input file 에 write command 에서 입력 마지막에 H 또는 C 를 추가하여 입력 받는 데이터의 workload 데이터를 표기하였다.

이에 대한 자세한 내용은 뒤에 input generator 에서 다시 설명하겠다. 이렇게 입력 받는 데이터를 ftl 에 넘겨주기 위해 ftl_test.c 에서도 아래 figure 와 같이 수정이 이루어졌다.

```
case 'W':
    scanf("%d %d %c", &lba, &nsect, &freq);
    assert(lba >= 0 && lba + nsect <= N_LPNS * SECTORS_PER_PAGE);
    buf = malloc(SECTOR_SIZE * nsect);
    for (i = 0; i < nsect; i++)
        buf[i] = get_data();
    ftl_write(lba, nsect, freq, buf);
```

Figure 7. Modification of ftl_test.c

ftl.c 에서는 거의 모든 함수에서 수정이 이루어졌다. 각 함수에 인자로 workload_type, 다시 말해 어떠한 종류의 workload 인 데이터가 들어왔는지를 입력 받는다. 특히 주목해야 하는 부분은 get_next_page 함수와 next_page 배열이다. get_next_page 함수는 다음에 write 할 PBN, PPN 을 정해주는 함수로, 쉽게 생각하면 write pointer 를 움직이는 함수이고, next_page 배열은 그 write pointer 라고 생각할 수 있다. 각 workload 마다 write 이 일어나야 하는 위치가 다르기 때문에, next_page 가 workload 마다 따로 관리가 되는 것을 확인할 수 있다.

추가적으로 확인해 주어야 하는 부분은 GC, MAP_GC 함수이다. 기존에도 같은 상태의 block 안에서 victim block 을 선정했다. 다만 이때는 DATA, TRANS, FREE 의 세가지 block state 만 존재했다면, 지금은 위에서 언급한 것처럼 5 가지의 state 를 갖고 있기 때문에 주의해야한다. 또한, 위 3.2 절에서 언급한 Problem 의 상황이 일어날 수 있기 때문에, victim block 이 greedy method 로 선정되지 않았을 때, disk defragmenter 를 호출하는 것을 볼 수 있다.

4.4 Disk Defragmenter

추가 기능을 위해 새로 등장한 함수이다. 입력된 bank 의 workload_type 과 반대되는 type 에 대해 진행되는 것을 볼 수 있는데, 이는 3.3 절의 해결 방법에서 언급했던 것처럼, 특정 workload 가 valid page 로 꽉 차게 되면 나머지 workload 에서 free block 을 만들어내야 하기 때문에 이와 같은 방식으로 작동한다. bank 내의 invalid page 를 포함한 모든 동일한 workload 의 block 들에 대해, valid page 를 free block 으로 옮기고 block 을 erase 한다.

4.5 Input generator and others

workload 가 분리된 input 을 만들기 위해서, 아래 figure 와 같은 함수를 통해 새로운 input file 을 생성하였고, 보다 더 정확한 결과 분석을 위해 stat 으로 저장되는 값, 실행이 끝나고 보여지는 stat 의 값이 계산되는 과정에서 수정이 이루어졌다.

4.5.1 Input 1

```
void make_input1(FILE* file)
{
    int K = 100; int N = 600; int M = 10;
    fprintf(file, "S 321\n");
    for(int i = 0; i < N; i++)
    {
        fprintf(file, "W %d 8 C\n", i*16);
        fprintf(file, "W %d 8 H\n", i*16+8);
    }
    for(int k = 0; k < K; k++)
    {
        for(int j = 0; j < M; j++)
        {
            for(int i = 0; i < N; i++)
            {
                if(i%2) fprintf(file, "W %d 8 H\n", i*32+8);
                fprintf(file, "W %d 8 H\n", 16*N + j*32*N + i*32);
                fprintf(file, "W %d 24 C\n", 16*N + j*32*N + i*32+8);
            }
        }
    }
    for(int i = 0; i < M*N; i++)
    {
        fprintf(file, "R %d 32\n", i*32);
    }
}
```

Figure 8. Test input 1

첫번째 input 은 두 종류에 stream 을 동시에 처리하는 상황을 가정하였다. 코드에서 볼 수 있듯이, Hot data 가 들어오는 동시에 Cold data 도 들어오고 있음을 알 수 있다. 원래 이 Simulator 의 목적인 Multi-stream 상황에서의 효율성을 확인하기 위해 다음과 같이 첫번째 test input 을 만들었다.

4.5.2 Input 2

```
void make_input2(FILE* file)
{
    int N = 2400; int M = 50; int ratio = 10;
    fprintf(file, "S 111\n");
    for(int i = 0; i < N; i++)
    {
    ¦   fprintf(file, "W %d 16 C\n", i*32);
    ¦   fprintf(file, "W %d 16 H\n", i*32+16);
    ¦   fprintf(file, "R %d 32\n", i*32);
    }

    for(int j = 0; j < M; j++)
    {
    ¦   for(int i = 0; i < N/2; i++)
    ¦   {
    ¦   ¦   fprintf(file, "W %d 16 H\n", N*32 + i*32);
    ¦   ¦   fprintf(file, "W %d 16 H\n", N*32 + i*32+16);
    ¦   }
    ¦
    ¦   if(j%ratio == ratio-1)
    ¦   {
    ¦   ¦   for(int i = 0; i < N; i++)
    ¦   ¦   {
    ¦   ¦   ¦   fprintf(file, "W %d 24 C\n", i*32);
    ¦   ¦   }
    ¦   }

    ¦   for(int i = 0; i < N; i++)
    ¦   {
    ¦   ¦   fprintf(file, "W %d 16 H\n", i*32+16);
    ¦   }
    }
    for(int i = 0; i < N; i++)
    {
    ¦   fprintf(file, "R %d 32\n", i*32);
    }
}
```

Figure 9. Test input 2

두 번째 input은 workload가 다른 데이터가 동시에 들어오지 않고 구분되어 들어오는 경우이다. 코드에서 Hot data가 쓰이는 반복문과 Cold data가 쓰이는 반복문이 구분되어 있는 것을 볼 수 있다. 이는 Hot data의 전체적인 입력이 끝나고, Cold data의 전체적인 입력이 진행되는 방식으로 생각할 수 있고, 결국 두 stream의 data가 따로 들어오는 경우를 의미한다.

4.5.3 Input 3

```
void make_input3(FILE* file) // H, C ratio 1 : 1
{
    int K = 100; int N = 4000;
    fprintf(file, "S 123\n");

    for(int j = 0; j < K; j++)
    {
        for(int i = 0; i < N; i++)
        {
            fprintf(file, "W %d 16 C\n", i*32);
            fprintf(file, "W %d 16 H\n", i*32+16);
        }
        for(int i = 0; i < N; i++)
        {
            if(j%2) fprintf(file, "W %d 8 C\n", i*32);
            else fprintf(file, "W %d 8 H\n", i*32+8);

            if(j%2) fprintf(file, "W %d 8 H\n", i*32+16);
            else fprintf(file, "W %d 8 C\n", i*32+24);
        }
    }
    for(int j = 0; j < N; j++)
    {
        fprintf(file, "R %d 32\n", 32*j);
    }
}
```

Figure 10. Test Input 3

세 번째 input은 workload와 관계없이 데이터가 들어오는 경우이다. Host가 workload를 제대로 구분하지 못하였을 때, 혹은 미세한 차이로 하나의 stream은 Hot, 다른 stream은 Cold로 판정되었을 때로 생각할 수 있을 것이다. 실제로 코드에서 들어오는 Hot data와 Cold data의 비율이 1:1임을 확인할 수 있다.

4.5.4 Others

추가적으로 Disk Defragmenter 에 대한 stat 을 따로 보여주기 위해 다음 figure 와 같이 stat structure 에 몇가지 항목을 추가하고, 실행화면 마지막에 띄우는 값을 조금 변경하였다. 물론 Disk Defragmenter 에서 일어나는 NAND write 는 WAF 계산할 때 고려하도록 하였다.

```c
struct ftl_stats {
    int gc_cnt;
    int map_gc_cnt;
    long host_write;
    long nand_write;
    long gc_write;
    long map_write;
    long map_gc_write;
    long cache_hit;
    long cache_miss;
    long fragmenter_cnt;
    long fragmenter_write;
};
```

Figure 11. Modified ftl_stats structure in ftl.h

```c
static void show_stat(void)
{
    printf("\nResults ------\n");
    printf("Host writes: %ld\n", stats.host_write);
    printf("GC writes: %ld\n", stats.gc_write);
    printf("Number of GCs: %d\n", stats.gc_cnt);
    printf("MAP writes : %ld\n", stats.map_write);
    printf("Number of MAP GCs : %d\n", stats.map_gc_cnt);
    printf("Number of MAP GC write : %ld\n", stats.map_gc_write);
    printf("Number of fragmenters : %ld\n", stats.fragmenter_cnt);
    printf("Number of fragmenters write : %ld\n", stats.fragmenter_write);

    printf("Valid pages per GC: %.2f pages\n", (double)stats.gc_write / stats.gc_cnt);
    printf("Valid pages per Map GC: %.2f pages\n", (double)stats.map_gc_write / stats.map_gc_cnt);

    printf("Cache hit rate : %.2f %%\n", (double)(stats.cache_hit*100. / (stats.cache_hit + stats.cache_miss)));
    printf("WAF: %.2f\n", (double)((stats.nand_write + stats.gc_write+stats.map_write + stats.map_gc_write +
stats.fragmenter_write) * 8.0 / stats.host_write));
}
```

Figure 12. Modified show_stat in ftl_test.c

5. Comparison & Analysis

모든 input 은 N_Bank 4, BLKS_PER_BANK 128, PAEGS_PER_BLK 64 인 조건에서 실행하였다. 기타 OP ratio, Data block / Map block 개수 등의 조건은 기존에 ftl.h 에 명시되어 있던 것과 동일하게 진행하였다.

5.1 Input 1

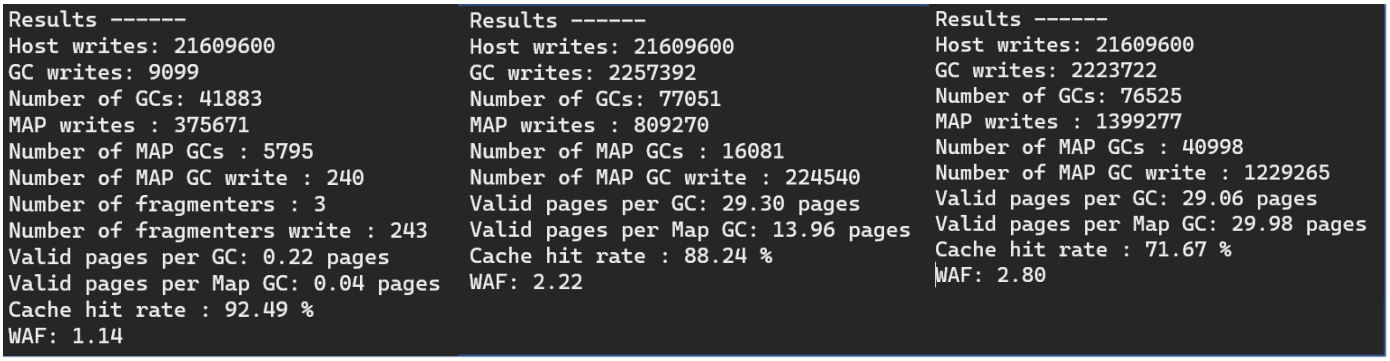

Figure 13. Stats comparison of test input 1

좌측부터 차례대로 Data buffer, Workload Separation 모두 구현된 경우, Data buffer 기능만 구현된 경우, 기존의 Lab4 DFTL (data buffer, Workload Separation 없음) 이다. WAF 를 보면 확연하게 차이가 나타나는 것을 확인할 수 있다. 동일한 input 에 대해 GC write, MAP GC write 및 defragmenter write 이 적게 나타나기 때문에, 더 향상된 성능을 보여준다. 특히 주목해야 할 점은 GC, MAP GC 시의 평균 valid page 개수인데, workload separation 을 적용한 경우에는 확실히 낮은 수치를 보여준다. 이를 통해, 동시에 다른 종류의 workload 작업이 일어나는 상황에서 뛰어난 성능을 가졌다고 말할 수 있을 것이다.

5.2 Input 2, 3

좌측부터 차례대로 Data buffer, Workload Separation 모두 구현된 경우, Data buffer 기능만 구현된 경우, 기존의 Lab4 DFTL (data buffer, Workload Separation 없음) 이다. Input 2, Input 3 에서는 오히려 Workload Separation 이 적용되었을 때 미세하지만 낮은 WAF 를 갖는 것을 알 수 있다. 결국 workload separation 된 환경에서는 GC 를 진행할 때 같은 workload 인 block 들 중 victim block 을 선정하기 때문에, 특정 경우에서는 선정된 victim block 이 최선이 아닌 경우가 존재한다. 즉, GC policy 가 greedy 일때를 기준으로, workload separation 을 위해서 가장 invalid page 가 많은 block 이 아닌 다른 block 을 victim 으로 선정한다는 것이다. 물론 이런 경우 실제 copy 수가 더 많아지는 overhead 를 갖고 있다. 이러한 경우가 Input 2, 3 에서 나타나, WAF 가 기존보다 더 커지는 현상이 발생한다.

또한, 현재 구현된 Workload Separation 에서는 특정 workload 를 저장할 공간이 부족할 경우 (3.2 절의 상황), 자동으로 Disk Defragmenter 을 호출하게 되어 있다. 앞서 설명했던 것처럼 , Disk Defragmenter 은 자체적인 overhead 를 갖고 있기 때문에, copy 수가 많아져 WAF 를 증가시키는 요인이 될 수 있다.

물론, 각 상황 별 input 한 개씩만 가지고 성능 비교를 확실하게 하기는 어렵다고 생각한다. 제작된 input 자체가 특수한 경우라 특정 Simulator 에서만 성능이 좋거나, 나쁘게 나올 수도 있기 때문이다. 실제로 input 2, 3 의 경우 GC write 자체가 0 인 경우도 존재하여 일반적인 경우라고 보기 힘들 수도 있을 것이다. 다만, Workload Separation 이 적용된 Simulator 가 항상 좋은 성능을 보여주는 것은 아니라는 것을 보여주기 위하여 이러한 input 을 선정하였다.

```
Results ------
Host writes: 4204800
GC writes: 23936
Number of GCs: 8112
MAP writes : 65722
Number of MAP GCs : 1081
Number of MAP GC write : 8722
Number of fragmenters : 2
Number of fragmenters write : 102
Valid pages per GC: 2.95 pages
Valid pages per Map GC: 8.07 pages
Cache hit rate : 93.36 %
WAF: 1.19
```

```
Results ------
Host writes: 4204800
GC writes: 0
Number of GCs: 7786
MAP writes : 65700
Number of MAP GCs : 953
Number of MAP GC write : 0
Valid pages per GC: 0.00 pages
Valid pages per Map GC: 0.00 pages
Cache hit rate : 93.39 %
WAF: 1.12
```

```
Results ------
Host writes: 4204800
GC writes: 0
Number of GCs: 7786
MAP writes : 65700
Number of MAP GCs : 953
Number of MAP GC write : 0
Valid pages per GC: 0.00 pages
Valid pages per Map GC: 0.00 pages
Cache hit rate : 87.72 %
WAF: 1.12
```

Figure 14. Stats comparison of test input 2

```
Results ------
Host writes: 19200000
GC writes: 3136
Number of GCs: 28574
MAP writes : 300712
Number of MAP GCs : 4638
Number of MAP GC write : 864
Number of fragmenters : 200
Number of fragmenters write : 3136
Valid pages per GC: 0.11 pages
Valid pages per Map GC: 0.19 pages
Cache hit rate : 90.62 %
WAF: 1.13
```

```
Results ------
Host writes: 19200000
GC writes: 0
Number of GCs: 37072
MAP writes : 300000
Number of MAP GCs : 4612
Number of MAP GC write : 0
Valid pages per GC: 0.00 pages
Valid pages per Map GC: 0.00 pages
Cache hit rate : 93.65 %
WAF: 1.12
```

```
Results ------
Host writes: 19200000
GC writes: 0
Number of GCs: 37072
MAP writes : 300000
Number of MAP GCs : 4612
Number of MAP GC write : 0
Valid pages per GC: 0.00 pages
Valid pages per Map GC: 0.00 pages
Cache hit rate : 87.50 %
WAF: 1.12
```

Figure 15. Stats comparison of test input 3

6. Conclusion & Things to improve

6.1 Conclusion

Workload type 에 따라서 분리되어 저장, 관리하는 Simulator 를 구현해보았고, 기존의 DFTL Simulator 와 비교해보았다. 기존 목적이었던 동시에 서로 다른 Workload type 의 데이터 write 이 이루어질 때, 즉 Multi Streaming 이 일어날 때에 성능은 확연히 향상된 것을 볼 수 있었다.

6.2 Things to improve

6.2.1 Input file 의 일반성, 다양성

앞부분에서도 언급했지만, input file 이 각 case 별로 1 개씩 밖에 없어서 Simulator 간 비교가 정확히 이루어지지 않았다고 생각한다. 또한 Input file 이 인위적으로 만들어진 경우이다 보니, 실제로 SSD 에서 file R/W 가 이루어질 때와 차이가 있을 것이라고 생각한다. 따라서 이를 개선하기 위해서는 실제 SSD 에서 ftl 이 받는 신호를 가져와서 input data 로 가공해서 쓸 수 있다면, input file 을 더욱 다양하고, 일반적으로 만들 수 있을 것이고, 분석에서의 신뢰성을 높일 수 있을 것이다.

6.2.2 Disk Defragmenter 의 overhead

Disk Defragmenter 방식은 언급한 바와 같이 overhead 가 크다. SSD 의 용량이 더욱 커지게 된다면 overhead 도 그만큼 커질 것이고, 이 작업이 실제 File R/W 도중 일어난다면 User 입장에서 속도가 갑자기 느려지는 것으로 인식될 것이다. 이는 우리가 comparison 과정에서 보았던 overhead 보다는 더욱 큰 문제로 다가올 수 있다.

이를 개선하기 위해서는 Disk Defragmenter 가 trigger 되는 시점을 바꾸는 것으로 어느 정도 해결할 수 있다. 현재는 꼭 필요할 때만 Disk Defragmenter 를 진행하지만, 이를 NAND 가 쉬고 있을 때 진행하도록 추가해 준다면, File R/W 가 진행되는 도중 Disk Defragmenter 가 호출되는 경우는 적어질 것이므로 이러한 문제점이 해결되는 효과를 가져올 수 있을 것이다. 사실 이런 방식이 구현된다면, Workload separation 으로 인해 victim block 을 선정하는 데 있어서 생기는 overhead 또한 일부 해결 가능하기 때문에 큰 이점으로 작용할 수 있다. 이번 Simulator 에서는 Host 로부터 입력이 들어오는 timing 에 관련된 정보를 받지 않아, 이러한 구현이 불가능 했지만 실제 이를 적용한다면 Disk Defragmenter 로 인한 overhead 를 줄일 수 있을 것이다.

7. Reference

- 강의자료 Solid State Storage Technologies, p.16
- Multi-streamed SSD Research : https://dhkoo.github.io/2019/02/26/multistream/
- Hard Disk Structure Picture : https://gsk121.tistory.com/260