

졸업자격실험보고서

Ext4 파일시스템에서 자연할당의 효율성

The Effectiveness of Delayed Allocation
On Ext4 Filesystem

지도교수 강동현

동국대학교 과학기술대학 컴퓨터공학과

한준호

2020

졸업자격실험보고서

Ext4 파일시스템에서 자연할당의 효율성

The Effectiveness of Delayed Allocation
On Ext4 Filesystem

한 준 호

지도교수 강 동 현

본 보고서를 졸업자격 실험 보고서로 제출함.

2020년 06월 10일

한 준 호의 졸업자격 실험 보고 통과를 인준함.

2020년 06월 10일

주심 도 재 수

부심 박 기 석

부심 강 동 현

동국대학교 과학기술대학 컴퓨터공학과

목 차

제 1 장 서 론	1
1.1 연구 배경 및 목표	1
제 2 장 이론적 배경	2
2.1 파일 시스템	2
2.2 Ext4 파일 시스템	4
2.3 자연 할당 (Delayed Allocation)	7
제 3 장 기본 설정	9
3.1 실험 환경	9
3.2 기본 실험 방식	10
3.2.1 SSD 장착	10
3.2.2 터미널 환경	10
3.3 SSD 기본 설정	11
3.3.1 파티션	11
3.3.2 파티션 과정	12
3.3.3 포맷	15
3.3.4 마운트	16
3.3.5 자연 할당 옵션	19

제 4 장 실험 및 결과	20
4.1 Fio	20
4.1.1 Fio 설치	20
4.1.2 실행 방법	21
4.1.3 실험 결과	24
4.1.4 fsync 옵션	25
4.2 Filebench	27
4.2.1 Filebench 설치	27
4.2.2 실행 방법	30
4.2.3 실험 진행	32
4.2.4 실험 결과	33
4.3 YCSB	34
4.3.1 YCSB 설치	35
4.3.2 실행 방법	38
4.3.3 실험 결과	40
4.4 Blktrace	45
4.4.1 Blktrace 설치	45
4.4.2 실행 방법	46
4.4.3 실험 결과	48
제 5 장 결 론	50
참 고 문 헌	51

그 림 목 차

[그림 1] 파일 시스템의 기본 구조	3
[그림 2] Ext3 파일 시스템의 블록 매팅	6
[그림 3] Ext4 파일 시스템의 Extent	6
[그림 4] 자연 할당 유/무에 따른 디스크 할당	8
[그림 5] 터미널 아이콘	10
[그림 6] 터미널 기본 화면	10
[그림 7] fdisk -l로 확인한 SSD	12
[그림 8] fdisk 테이블 명령어 command 창	13
[그림 9] 새로운 파티션 생성 과정	14
[그림 10] 생성된 파티션 확인	14
[그림 11] Ext4 파일 시스템으로 포맷	15
[그림 12] 마운트 후 폴더 상태 변화	18
[그림 13] 마운트 후 마운트 확인	18
[그림 14] 마운트 상태 확인	18
[그림 15] SSD 폴더 잡금 상태	18
[그림 16] 마운트 옵션 유/무 확인	19
[그림 17] fio 설치	20
[그림 18] fio 실험 결과	24
[그림 19] fsync 실험 결과	26
[그림 20] filebench 다운로드	27
[그림 21] filebench 정상 설치 확인	29

[그림 22] filebench 실험 도중 멈춤	31
[그림 23] filebench 정상 작동	31
[그림 24] filebench 실험 결과	33
[그림 25] git 설치	35
[그림 26] git clone으로 YCSB 다운로드	35
[그림 27] maven 설치 및 버전 확인	37
[그림 28] mvn clean package	37
[그림 29] Build Success	37
[그림 30] ./bin/ycsb	39
[그림 31] rocksdb에 워크로드 불러오기	39
[그림 32] rocksdb에 워크로드를 실행하기	39
[그림 33] Workload A 실험 결과	40
[그림 34] Workload B 실험 결과	41
[그림 35] Workload C 실험 결과	41
[그림 36] Workload D 실험 결과	42
[그림 37] Workload E 실험 결과	42
[그림 38] Workload F 실험 결과	43
[그림 39] Workload A 추가 실험 결과	44
[그림 40] Blktrace 설치	45
[그림 41] Blktrace 실행	47
[그림 42] Blkparse 실행	47
[그림 43] fio의 blktrace 실험 결과	48
[그림 44] Workload A의 blktrace 실험 결과	49

표 목 차

[표 1] 실험 환경	9
[표 2] fio 파일 예시	22
[표 3] blktrace 옵션	47
[표 4] blkparse 옵션	47

제1장. 서론

1.1 연구 배경 및 목표

요즘 IT가 발달하면서 서로 정보를 주고받는 것이 일상이 되었다. 서로 주고받은 정보들을 저장하고 관리를 해주는 저장 장치들이 그만큼 중요한 역할을 하게 되었다. 컴퓨터를 사용한다면 누구나 “하드”라는 부품을 들어 봤을 것이다. “하드”라고 불리는 HDD는 오랜 시간동안 PC의 저장 장치로 많이 보급되었다. 하지만 HDD는 자기 디스크를 물리적으로 회전시켜 데이터를 관리하기 때문에 소음, 전력 소모, 그리고 충격에 약하다는 단점이 있었는데 이러한 단점들은 보완하고 더 빠른 속도의 정보를 주고 받기 위해 SSD가 개발되었다 [1]. SSD는 Solid State Drive의 약자로 내부의 자기 디스크를 사용하는 HDD와 다르게 내부의 반도체 메모리를 사용하는 저장 장치이다 [2]. 반도체를 사용하기 때문에 HDD처럼 물리적으로 움직이는 부품이 없어 소음이 없고 전력 소모가 줄어들었으며 충격에 의한 손상이 크게 줄었다 [2]. 그리고 무엇보다 HDD보다 더 빠른 속도를 지원하여 최근에는 SSD를 사용하는 기기들이 늘어났다 [2]. SSD를 사용하면서 더 빠르고 효율적인 방법으로 SSD를 사용하는 방법을 찾게 되었고, Linux의 Ext4 파일 시스템에서 디스크의 성능을 향상시키는 여러 가지 기술들이 있는데 그중에서 Delayed Allocation이라는 자연 할당 기능을 사용하여 실험을 진행하여 보았다.

제2장. 이론적 배경

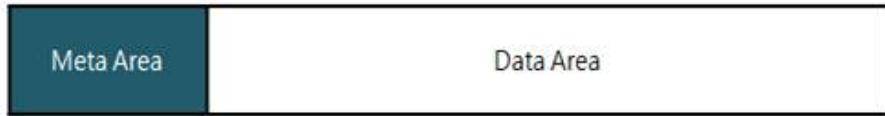
2.1 파일 시스템

파일 시스템이란 컴퓨터에서 데이터들을 쉽게 저장하고 찾고 유지 및 관리를 해주는 체계를 말한다 [3]. 파일 시스템은 하드디스크, CD-ROM과 같은 물리적인 저장 장치만 해당되지 않고 서버, 클라이언트와 같이 가상 형태의 접근 방식을 가진 저장 장치도 그 범위에 포함된다 [3]. 예전에는 데이터의 양도 적고 많은 데이터 처리가 필요가 없어 파일 시스템이 필요가 없었지만 요즘 대용량 저장매체들이 보편화되고 그만큼 많은 양의 데이터를 다루게 되면서 운영체제에서 이러한 많은 양의 데이터들을 관리하기 위해 많은 자원들을 사용하게 되었다. 이로 인해 다른 작업들을 하는데 어려움이 생기게 되었고 효율적인 데이터 관리를 위해 파일 시스템이 필요하게 되었다.

파일 시스템은 기본적으로 [그림 1]과 같이 메타 데이터 영역과 데이터 영역의 두 가지 영역을 가진 구조로 표현이 된다 [3]. 이러한 구조를 활용하여 사용자는 빠르게 데이터 정보를 가져올 수 있다. 메타 데이터 영역은 데이터 영역에 기록된 파일의 이름, 위치, 크기, 시간정보 등등 데이터의 정보들을 저장하는 곳인데 데이터 영역에 기록된 파일 데이터를 직접 필요한 경우가 아니라면 메타 데이터 영역을 활용하여 빠르게 접근하고 해당 파일 정보들을 확인할 수 있다 [3]. 이러한 파일 시스템의 구조를 활용하면 파일을 실행할 때에 윈도우에서 탐색기를 통해 메타 데이터 영역을 탐색하고 해당 파일의 확장자에 맞는 응용 프로그램으로 파일을 실행한다.

그리고 그 응용 프로그램은 해당 파일이 위치한 데이터 영역에서 파일을 읽어서 처리를 한다. 데이터 영역만 존재하지 않고 메타 데이터 영역을 따로 구분하여 사용하는 만큼 데이터 영역이 줄어 들긴 하지만 이러한 구조의 장점으로 큰 효율을 볼 수 있다.

현재 OS 계열에 따라 각각 서로 다른 파일 시스템을 사용하는데 대표적으로 Window에서는 FAT와 NTFS, Linux에서는 Ext, Apple Mac에서는 HFS, Unix에서는 UFS, CD/DVD에서는 CDFS를 사용한다 [3]. 본 실험에서는 Linux 환경에서 진행하여 Ext 파일 시스템을 사용하였다.



[그림 1] 파일 시스템의 기본 구조

2.2 Ext4 파일 시스템

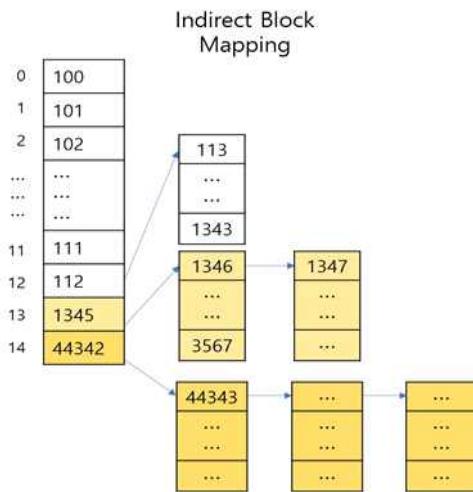
Ext4 파일 시스템은 현재 리눅스 커널의 기본 파일 시스템이다. 최대 1EB의 블루프과 16TB의 파일을 지원하며 64비트의 기억 공간 제한을 없애고 이전의 Ext3 파일 시스템의 단점을 보완하여 하위 호환성이 있는 확장 버전으로 Ext3 사용자에게는 영향을 주지 않으면서 Ext3에서 fork하여 Ext4로 이름을 변경하여 제안되어 개발되었다 [4]. Ext2와 Ext3 파일 시스템에 대한 하위 호환성으로 Ext2와 Ext3 파일 시스템을 Ext4로 마운트하는 것이 가능하고 Ext4의 블록 할당 알고리즘을 기존의 파일 시스템에서 사용할 수 있다. 하지만 Ext4의 파티션이 Extent를 사용한다면 Ext3로 마운트가 불가능하다.

Ext4 파일 시스템은 디스크의 블록들의 읽기 쓰기 성능을 향상시키기 위해 Extent, Multiblock Allocation, Delayed Allocation과 같은 정책들을 추가했다 [4]. 기존의 Ext3 파일 시스템에 경우 요청된 데이터를 처리할 때 블록 단위로 하는 블록 매핑 정책을 사용했으나 Ext4 파일 시스템에서는 기존의 블록 매핑 기법을 대체하기 위해 Extent 정책을 사용했다 [4]. [그림 2]와 [그림 3]를 보면 Ext2와 Ext3 파일 시스템에서는 각각의 블록들의 트랙을 유지하기 위해 간접 블록 매핑을 사용했는데 연속적으로 블록을 할당하였다고 해도 각 데이터 블록을 가리키기 위해 inode에 존재하는 12개의 직접 블록과 3개의 간접 블록을 이용하여 하나씩 지정해야 했다. 하지만 extent 기반 블록 유지방법은 간단한 정보만을 저장하여 메타 데이터의 크기를 절약하였고 데이터 블록의 접근에 걸리는 시간을 단축했다.

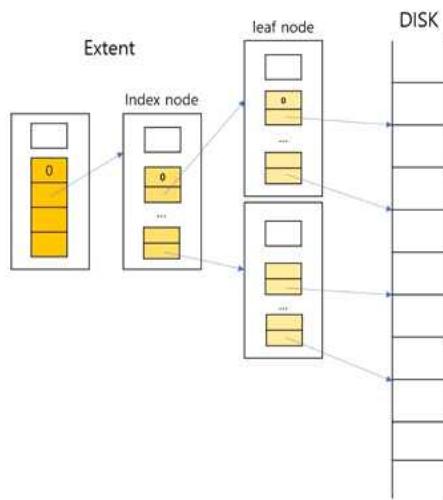
Extent는 연속한 물리적 블록의 묶음을 의미하는데 초기에 어느정도 연속된 공간을 할당하고 그 공간이 부족할 경우 Extent의 단위로 할당을 하는데 최대 128MiB만큼의 연속된 블록에 할당할 수 있어서 한번에 처리할 수 있는 블록의 범위가 늘게 되었고 블록 매핑에서 생긴 단편화를 방지하고 대용량 파일처리시 좋은 I/O 성능을 나타낸다 [4].

블록 매핑 기법에 경우 Ext3 파일 시스템에서는 한번에 한 개의 블록(4KB)을 할당할 수 있는데 이러한 경우 용량에 비해 매핑 되는 블록의 단위가 작기때문에 여러 번 매핑을 해야 했고 그로 인해 성능이 크게 저하되었다. 예를 들어 100MB의 작은 용량이 필요한 경우에도 25,600번 ($100\text{MB} = 102,400\text{KB}$, $102,400\text{KB}/4\text{KB} = 25,600\text{번}$) 블록 할당을 하게 되는데 작은 용량도 25600번의 매핑이 필요하다면 대용량의 데이터의 경우 더 많은 횟수의 블록 할당 호출이 발생하게 되고 연속된 블록으로 할당되지 않아 단편화가 발생하게 된다. 단편화는 하나의 데이터를 한번에 연속적인 블록에 저장하지 못하고 여러 개의 불연속적으로 저장되는 상태를 말하는데 이런 경우 데이터를 처리할 때 큰 성능 저하를 발생시킨다.

Ext4 파일 시스템에서는 단일 블록 할당으로 인한 오버헤드를 막기위해 다중 블록 할당자를 사용했다 [4]. 매 호출마다 한번에 한 개의 블록을 할당하는 것이 아닌 한번에 여러 개의 블록을 할당하여 블록 할당의 횟수가 줄어들게 되었고 연속적인 블록에 할당하게 되어 오버헤드를 줄일 수 있게 되었다.



[그림 2] Ext3 파일 시스템의 블록 맵핑

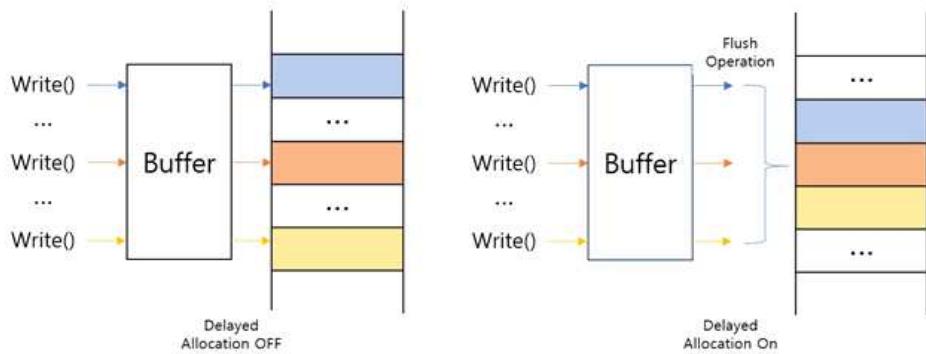


[그림 3] Ext4 파일 시스템의 Extent

2.3 지연 할당 (Delayed Allocation)

Ex4 파일 시스템에 delayed allocation 또는 allocate on flush라는 새로운 블록 할당 기법을 사용하게 되었다. 기존의 Ext3 파일 시스템에서는 write() 함수가 호출되면 실제로 디스크에 데이터가 쓰여 지고 있지 않더라도 디스크에 그 데이터에 대한 블록을 할당했다 [4]. 하지만 만약 파일 크기가 고정되어 있지 않고 용량이 커지는 파일에 대해서는 지속적으로 write() 함수가 호출이 되고 이 경우 CPU의 사용량을 늘리게 되고 연속되지 않은 블록에 할당이 되어 큰 파일에 대해 단편화가 발생하게 되었다. 이러한 문제로 Ext4 파일 시스템에서 지연 할당 기법을 사용하였다. Delayed allocation의 경우 버퍼 캐시에 연속된 데이터들이 저장되면 Flush operation이라는 캐시를 flush 하는 과정을 통해 디스크에 블록을 할당하는데 파일이 캐시에 유지 되어있는 동안에는 그 파일이 실제로 디스크에 쓰일 때까지 블록 할당을 최대한 지연시킨다 [5]. 이로 인해 캐시에 있는 파일 중 같은 파일에 대해서는 다중 블록 할당을 통해 인접한 공간에 할당하게 되었고 연속된 블록에 할당을 하게 되어 내부 단편화를 방지할 수 있게 되었다 [21].

연속된 파일을 할당한다는 가정하에 [그림 4]을 보면 3번의 write() 함수의 호출이 있었는데 지연 할당이 없는 경우 write() 호출할 때마다 디스크에 할당을 하게 되어 저장 장치에 불연속적으로 할당을 하게 되는데 지연 할당의 경우 write() 호출에도 디스크에 할당을 지연하다가 캐시를 flush 할 때에 디스크에 한번에 할당하게 되고 연속된 블록에 할당할 수 있게 된다.



[그림 4] 자연 할당 유/무에 따른 디스크 할당

제3장. 기본 설정

3.1 실험 환경

[표 1] 실험 환경

CPU	Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
Memory	Samsung DDR4 8GB x 4 (32GB)
Storage	Samsung SSD 860 PRO (256GB)
OS	Ubuntu 20.04 LTS
Kernel Version	5.4.0 - 29
File System	Ext4
Benchmark	Fio 3.16 / Filebench 1.4.9 , 1.5 / YCSB 0.18.0 / Blktrace, Blkparse

본 실험에서 Ext4 파일시스템에서 Delayed Allocation On/Off 옵션을 적용시켜 SSD의 I/O 성능을 분석하기 위해 하드웨어로는 intel i7-8700, dram 32GB, Samsung 860 PRO 256GB를 사용하였고 소프트웨어로는 운영체제는 Ubuntu 20.04 LTS, 커널은 5.4.0-29를 설치하였고 운영체제에 맞게 Ext4 파일시스템을 사용했다. SSD의 I/O 성능을 측정하기 위해 Fio, Filebench, YCSB, Blktrace/Blkparse를 사용하였다.

3.2 기본 실험 방식

3.2.1 SSD 장착

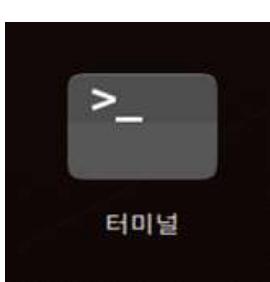
부팅되어 있는 상태에서 장착시 SSD 인식이 안되기 때문에 실험에 사용할 SSD를 장착 후 컴퓨터를 부팅 또는 재부팅을 한다.

3.2.2 터미널 환경

a. 터미널 실행

- Linux에서는 명령어 입력을 통해 응용 프로그램 실행과 세부 설정을 할 수 있기에 터미널 실행이 필요하다.
- [그림 5]와 같은 아이콘을 클릭하여 실행 할 수 있다.
- **Ctrl + Alt + T** 단축키를 통해 아이콘 클릭없이 바로 실행이 가능하다.

※ 기존 터미널보다 가독성을 위해 배경과 글씨색을 변경하여 진행하였다.



[그림 5] 터미널 아이콘



[그림 6] 터미널 기본 화면

3.3 SSD 기본 설정

3.3.1 파티션

파티션은 하나의 디스크를 논리적으로 구분하기 위해 나누는 하나의 구획을 의미한다. OS는 이렇게 나누어진 파티션을 별도의 드라이브로 인식을 한다. 파티션은 기본적으로 백업과 같은 데이터 유지 및 보안상의 이유, 한 대의 PC에 여러 OS를 설치하기 위한 시스템의 안정성 확보, 개인적으로 파일의 구분을 하기 위한 관리의 편리성, 사용하는 OS의 디렉토리 특성 등 다양한 목적을 가지고 있다 [6].

본 실험에서 SSD를 Linux 환경에서 사용하는 Ext4 파일 시스템을 이용하기 위해 파티션을 진행하였고 별도의 구분없이 SSD 전체를 사용하기 위해 한 개의 파티션을 생성하여 실험을 진행하였다.

3.3.2 파티션 과정

사용할 SSD의 파티션 지정을 위해 fdisk 명령어를 사용하였다.

- a. **sudo fdisk -l** : 현재 PC에 연결된 스토리지들과 파티션을 리스트로 출력
 - 현재 연결된 디스크 중 해당 SSD 이름, 용량을 통해 찾기
- b. 장착한 SSD의 경로를 확인
 - [그림 7]처럼 현재 연결된 SSD와 경로, 현재 파티션 상태를 확인
 - ex) **/dev/sda**

```
Disk /dev/sda: 238.49 GiB, 256060514304 bytes, 500118192 sectors
Disk model: Samsung SSD 860
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x8a30afdd
```

[그림 7] fdisk -l로 확인한 SSD

c. **sudo fdisk (경로)** : 연결된 SSD의 파티션 설정 커맨드 창 출력

- fdisk 명령어를 통해 파티션 테이블 명령어 입력창 출력 [그림 8]
- ex) **sudo fdisk /dev/sda**

d. **p**를 입력하여 현재 파티션이 존재하는지 확인

- 불필요한 파티션이 존재할 경우 d를 입력하여 파티션을 제거 후 진행

```
hjh2020@hjh2020:~$ sudo fdisk /dev/sda
Welcome to fdisk (util-linux 2.34).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.

Command (m for help): p
Disk /dev/sda: 238.49 GiB, 256060514304 bytes, 500118192 sectors
Disk model: Samsung SSD 860
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x8a30afdd

Command (m for help):
```

[그림 8] fdisk 테이블 명령어 command 창

e. **n**을 입력하여 새로운 파티션 생성

- enter를 누르며 넘어갈 경우 각 항목별 정해진 default 값이 자동으로 입력되어 파티션 생성 진행
- default 값을 입력하여 파티션 생성의 경우 SSD 전체를 사용하는 1개의 파티션을 생성
- **Command (m for help)** 언급 시까지 Enter를 반복 [그림 9]

```

Command (m for help): n
Partition type
  p  primary (0 primary, 0 extended, 4 free)
  e  extended (container for logical partitions)
Select (default p): p
Partition number (1-4, default 1):
First sector (2048-500118191, default 2048):
Last sector, +/sectors or +/-size{K,M,G,T,P} (2048-500118191, default 500118191
):
Created a new partition 1 of type 'Linux' and of size 238.5 GiB.
Partition #1 contains a ext4 signature.

Do you want to remove the signature? [Y]es/[N]o: y
The signature will be removed by a write command.

```

[그림 9] 새로운 파티션 생성 과정

f. **p**를 입력하여 생성된 파티션을 확인

- ex) **/dev/sda1** (sda의 첫 번째 파티션을 의미)

g. **w**를 입력하여 저장하고 커맨드 창 종료

h. **f**단계에서 **p** 명령어를 통해 파티션이 생성되지 않았을 경우

- c단계부터 다시 실행
- 만약 파티션이 존재할 경우 **d** 명령어를 통해 파티션을 제거한 후
파티션 생성 과정을 실행

```

Disk /dev/sda: 238.49 GiB, 256060514304 bytes, 500118192 sectors
Disk model: Samsung SSD 860
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x8a30afdd

Device      Boot Start       End   Sectors  Size Id Type
/dev/sda1        2048 500118191 500116144 238.5G 83 Linux

```

[그림 10] 생성된 파티션 확인

3.3.3 포맷

포맷을 하게 되면 그 저장 장치에 있는 데이터들이 모두 삭제가 되어 흔히 저장 장치 초기화라는 의미로 알려져 있지만 원래 포맷은 처음 사용하는 디스크 또는 디스크의 재사용을 위해 자료를 저장할 수 있도록 형식을 잡아주는 것을 의미한다 [7]. 각각 사용하는 환경에 맞는 파일 시스템의 형식으로 포맷을 하여 사용하는데 Windows에서 사용할 경우 FAT이나 NTFS, Linux 환경에서는 Ext 파일 시스템으로 포맷을 하여 사용한다 [7]. 본 실험에서는 생성된 파티션을 Linux 환경에 맞게 Ext4 파일 시스템으로 포맷하여 진행하였다.

a. **sudo mkfs.(포맷 형식) (경로)** - 해당 경로의 파티션을 포맷

- 현재 Linux 환경에 맞는 Ext4 파일 시스템으로 포맷
- ex) **sudo mkfs.ext4 /dev/sda1** - sda의 첫 번째 파티션을 포맷
- [그림 11]과 같은 화면이 나오면 정상 포맷 완료

```
h1h2020@h1h2020:~$ sudo mkfs.ext4 /dev/sda1
mke2fs 1.45.5 (07-Jan-2020)
Discarding device blocks: done
Creating filesystem with 62514518 4k blocks and 15630336 inodes
Filesystem UUID: 538c7513-483e-43e7-a608-14f37001c70a
Superblock backups stored on blocks:
            32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632, 2654208,
        4096000, 7962624, 11239424, 20480000, 23887872

Allocating group tables: done
Writing inode tables: done
Creating journal (262144 blocks): done
Writing superblocks and filesystem accounting information: done
```

[그림 11] Ext4 파일 시스템으로 포맷

3.3.4 마운트

마운트는 디스크와 같은 물리적인 장치를 특정 위치(디렉토리)에 연결시켜 주는 것을 의미한다 [8]. OS에 따라 마운트 방법이 다른데 현재 쓰는 PC나 노트북에서 주로 사용하는 운영체제인 Windows에서는 USB, 외장 하드디스크, SSD 등등 저장 장치를 장착하는 것 만으로 인식이 되고 사용이 가능하다. PnP(Plug and Play)라는 기능 덕분에 내부에서 자동으로 마운트 작업을 하여 사용자가 디스크를 편리하고 빠르게 사용할 수 있게 해 주는데 Linux의 경우 PnP 기능이 존재 하지만 시스템 부팅 후 다시 마운트 하여 관리자가 직접 특정 디렉토리에 연결을 시켜줘야 사용이 가능하다. 지속적으로 장착하여 사용하는 저장 장치의 경우 매번 마운트 하는 것 이 번거로울 수가 있는데 명령어를 통해 자동 마운트 하는 방법도 있다 [8].

테스트에서 Ext4 파일 시스템으로 포맷한 SSD를 사용하기 위해 바탕화면(자신이 사용하기 편한 위치)에 마운트 할 폴더를 만들고 그곳에 SSD를 마운트하여 사용했다.

a. 마운트를 위해 마운트를 할 특정 위치(폴더 or 디렉토리)를 생성

- 바탕화면에 SSD라는 폴더를 생성 - 경로 **/home/(계정 ID)/Desktop/SSD**
- 앞의 경로를 생략하여 ~/Desktop/SSD 로 사용이 가능

b. **sudo mount (SSD 경로) (마운트 할 경로)** - 마운트 명령어

- 바탕화면에 생성된 SSD라는 폴더와 포맷이 완료된 파티션을 연결
- ex) **sudo mount /dev/sda1 ~/Desktop/SSD**
- 마운트 해제 시 umount 명령어를 사용
- ex) **sudo umount /dev/sda1** - umount의 경우 마운트 한 경로는 필요없음

c. **sudo mount** 명령어를 통해 마운트가 되었는지 확인 가능

- 마운트 성공시 폴더의 상태가 바탕화면의 폴더에서 SSD로 변경
[그림 12]
- [그림 13], [그림 14]과 같이 마운트 상태 확인 가능

d. 마운트 후 폴더 아이콘에 잠금 그림이 있다면 권한 변경 명령어로
잠금 풀기

- **sudo chmod 777 (폴더 이름)** - 모든 권한 부여 [그림 15]
- 예) **sudo chmod 777 SSD**



[그림 12] 마운트 후 폴더 상태 변화

```
jh2020@jh2020:~$ sudo mount /dev/sda1 ~/Desktop/SSD
jh2020@jh2020:~$ sudo mount
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
udev on /dev type devtmpfs (rw,nosuid,noexec,relatime,size=16381624k,nr_inodes=4095406,mode=755)
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=000)
tmpfs on /run type tmpfs (rw,nosuid,noexec,relatime,size=3281952k,mode=755)
/dev/sdb1 on / type ext4 (rw,relatime,errors=remount-ro)
securityfs on /sys/kernel/security type securityfs (rw,nosuid,nodev,noexec,relatime)
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev)
tmpfs on /run/lock type tmpfs (rw,nosuid,nodev,noexec,relatime,size=5120k)
tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noexec,mode=755)
cgroup2 on /sys/fs/cgroup/unified type cgroup2 (rw,nosuid,nodev,noexec,relatime,nsdelegate)
```

[그림 13] 마운트 후 마운트 확인

```
tmpfs on /run/user/1000 type tmpfs (rw,nosuid,nodev,relatime,size=3281948k,mode=700,uid=1000,gid=1000)
gvfsd-fuse on /run/user/1000/gvfs type fuse.gvfsd-fuse (rw,nosuid,nodev,relatime,user_id=1000,group_id=1000)
/dev/fuse on /run/user/1000/doc type fuse (rw,nosuid,nodev,relatime,user_id=1000,group_id=1000)
/dev/sda1 on /home/jh2020/Desktop/SSD type ext4 (rw,relatime)
```

[그림 14] 마운트 상태 확인



[그림 15] SSD 폴더 잠금 상태

3.3.5 지연 할당 옵션

본 실험에서 Delayed Allocation On/Off (지연 할당 유/무) 상태에 따라 SSD의 I/O 성능을 측정하여 비교하기 위해 SSD를 마운트 할 때 nodelalloc 옵션을 사용하여 Delayed Allocation On/Off 상태를 설정했다.

a. **sudo mount -o (옵션) (SSD 경로) (마운트 할 경로)** – 마운트 옵션 적용

- ex) **sudo mount -o nodelalloc /dev/sda1 ~/Desktop/SSD**

b. **sudo mount** 명령어를 통해 옵션이 적용된 마운트 상태 확인

- 옵션 유/무에 따라 상태 확인 변화 가능 [그림 16]
- mount 옵션을 통해 현재 파티션의 Ext4 파일 시스템의 포맷 상태도 확인 가능

```
/dev/sda1 on /home/hjh2020/Desktop/SSD type ext4 (rw,relatime)
/dev/sda1 on /home/hjh2020/Desktop/SSD type ext4 (rw,relatime,nodelalloc)
```

[그림 16] 마운트 옵션 유/무 확인

제4장. 실험 및 결과

4.1 Fio

본 실험에서 Delayed Allocation On/Off 상태에서 SSD의 I/O 성능을 측정하기 위해 fio 벤치마크를 사용했다.

4.1.1 Fio 설치

- a. **sudo apt-get install fio** - fio 설치 [그림 17]

```
hjh2020@jh2020:~$ sudo apt-get install fio
前辈님! 속도를 높이는 방법입니다... 진료
의존성 트리를 만드는 중입니다
상태 정보를 읽는 중입니다... 완료
다음의 주가 패키지가 설치될 것입니다 :
  libverbs-providers libaio1 libgfapi0 libgfrpc0 libgfxdr0 libglusterfs0
  liblverbs1 libpython2.7-minimal libpython2.7-stdlib librados2 librbd1
  librdmacm1 libtirpc-common libtirpc3 python2.7 python2.7-minimal
제안하는 패키지 :
  gnuplot_gfio python-scipy python2.7-doc binfmt-support
다음 새 패키지를 설치할 것입니다:
  fio libverbs-providers libaio1 libgfapi0 libgfrpc0 libgfxdr0 libglusterfs0
  liblverbs1 libpython2.7-minimal libpython2.7-stdlib librados2 librbd1
  librdmacm1 libtirpc-common libtirpc3 python2.7 python2.7-minimal
8개 업그레이드, 17개 새로 설치, 0개 제거 및 24개 업그레이드 안 함.
9,997 kBайт 아카이브를 받아야 합니다.
이 작업 후 41.8 kBайт의 디스크 공간을 더 사용하게 됩니다.
계속 하시겠습니까? [Y/n] y
```

[그림 17] fio 설치

- b. **apt install**의 경우 대부분 [그림 17]처럼 디스크 공간 사용을 위해 허가 필요

- y를 눌러 설치를 계속 진행

4.1.2 실행 방법

명령어를 통해 실험하는 방식과 fio 파일을 만들어 실행하는 2가지 방법

o) 존재한다 [9],[10].

a. 명령어 입력 방식

- **fio --(옵션)=(설정 값)** - 실험에 사용할 모든 옵션을 입력 [9]
- ex) fio --directory=(마운트 한 폴더) --name=(파일 이름)
--direct=(int) --rw=(진행 방식) --bs=(블록 단위)
--size=(파일 용량) --runtime(실행시간(초단위))
--group_reporting -norandommap

b. fio 파일 생성 및 실행 방식

- fio 파일을 생성하여 필요한 옵션들을 입력하고 파일을 실행
- **sudo fio (파일 이름).fio**

[표 2] fio 파일 예시

[global] ioengine=libaio direct=1 ramp_time=60 size=40G iodepth=1 runtime=3600 bs =4k time_based group_reporting norandommap stonewall	[read-4k-para] filename=/home/hjh2020/Desktop/ SSD/rtest rw=read [write-4k-para] filename=/home/hjh2020/Desktop/ SSD/wtest rw=write
--	--

c. 실험에서 진행된 fio 옵션 [11]

- [global] 부분 - fio 실험에서 공통으로 사용되는 파라미터
- []으로 구성된 부분 - 각각의 옵션이 부여된 개별의 파라미터

[global] - 공통 파라미터

- **ioengine** - 작업 할 I/O 방식을 지정하는 방법
- **direct** - 1의 경우 Direct I/O를 0의 경우 Bufferd I/O를 진행
- **ramp_time** - 각 파라미터의 실험 사이의 여유 시간
- **size** - I/O를 진행할 파일의 사이즈
- **iodepth** - 큐의 깊이
- **runtime** - 진행 할 실험의 시간 (초단위)
- **bs** - 블록 사이즈

- **time_based** - 설정한 runtime 시간만큼 I/O를 진행
- **group_reporting** - 그룹 별로 결과값 출력(전체 출력)
- **norandommap** - 랜덤 워크로드시 과거의 I/O 위치를 고려하지 않음
- **stonewall** - 각 파라미터가 종료된 후 다음 파라미터를 진행

[개별 파라미터]

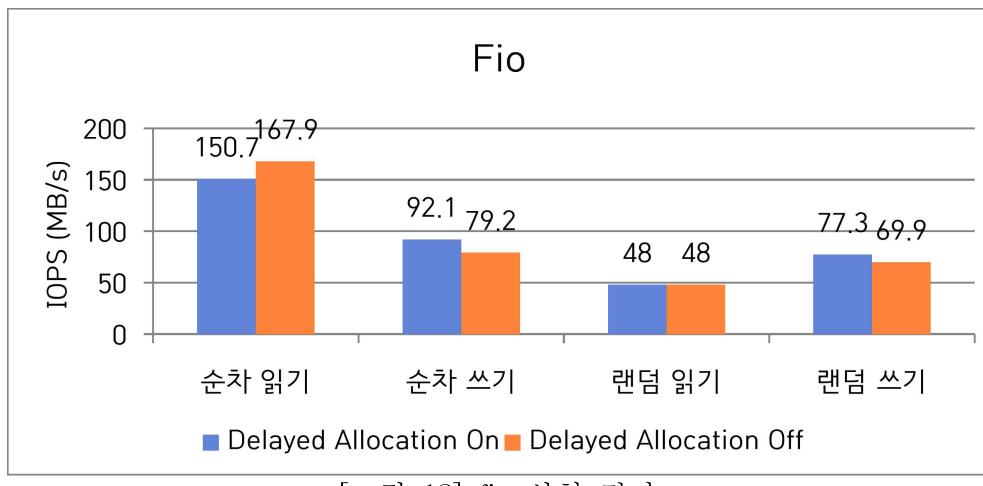
- **filename** - 생성될 파일의 경로와 이름을 지정
- **rw** - I/O의 종류 (write ,read, randwrite, randread, rw ,randrw)

d. 실험 진행

SSD에 순차 읽기/쓰기, 랜덤 읽기/쓰기를 진행하고 마운트시 각각 Delayed Allocation On/Off 상태를 적용하여 실험을 진행하였다. 실험 결과를 text 파일로 저장하고 저장된 text 파일에서 나온 수치들을 그래프로 표현하였다.

- iops * 설정한 블록 단위 = 초당 데이터 전송량
- 초당 데이터 전송량의 단위가 KB/s이므로 1024를 나누어 MB/s로 환산

4.1.3 실험 결과



[그림 18] fio 실험 결과

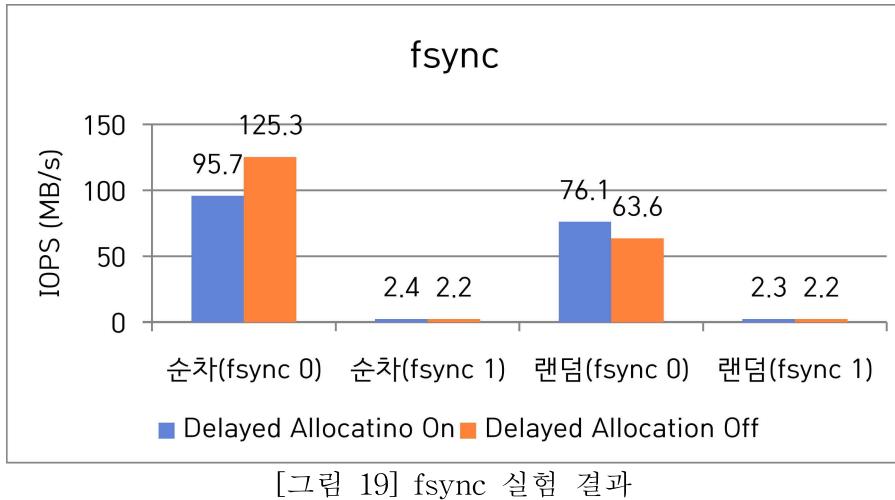
본 실험에서 SSD에 사용 중인 메모리보다 큰 40GB의 파일을 순차 읽기/쓰기, 랜덤 읽기/쓰기를 1시간 동안 실현을 5번 진행하여 그 평균 값을 나타냈다. 그 결과 [그림 18]과 같이 순차 읽기의 경우 Delayed Allocation Off시 약 11% 더 성능이 좋은 결과가 나왔고, 순차 쓰기의 경우 Delayed Allocation On시 약 15% 더 성능이 좋은 결과가 나왔다. 랜덤 읽기의 경우 옵션 On/Off에 따른 차이가 없었으며 랜덤 쓰기의 경우 Delayed Allocation On시 약 10% 더 좋은 성능의 결과가 나왔다. 순차 읽기, 랜덤 읽기의 경우 Delayed Allocation On시 성능이 더 안좋거나 큰 변화가 없었고 순차 쓰기, 랜덤 쓰기의 경우 Delayed Allocation On시 더 좋은 결과 값이 나왔다.

4.1.4 fsync 옵션

저장 장치의 I/O 연산은 커널 안의 버퍼 캐시나 페이지 캐시를 거쳐서 이루어진다. 캐시(Cache)란 저장 장치와 CPU 사이에 속도 차이를 줄여 주기 위한 작은 메모리 공간이다 [12]. 지연 할당에 경우 데이터가 쓰일 때까지 최대한 지연시켜 flush operation이 실행될 때 블록을 할당하게 되는데 fsync는 그 flush operation의 함수 중 하나로 파일에 대한 데이터의 변경이 발생하였을 경우 그 데이터와 그 데이터의 메타데이터 모두를 디스크에 저장을 한다 [21].

실험 통해 결과 값이 나오지 않아 새로 fio 파일에 fsync 옵션을 부여하여 실험을 진행하였다. fsync의 경우 write() 함수가 실행이 되고 캐시에서 바로 저장장치로 반영한다. fsync옵션에 따라 Delayed Allocation On/Off를 하여 순차 쓰기 / 랜덤 쓰기 테스트를 진행하였다.

a. fsync의 기본 default 값 = 0



[그림 19] fsync 실험 결과

처음 실험에서 Delayed Allocation On/Off 옵션을 부여하여 40GB의 파일을 1시간 동안 I/O 시킨 결과 순차 쓰기와 랜덤 쓰기의 경우 Delayed Allocation On시 IOPS의 성능이 각각 15% 10% 더 좋게나왔다. 이번 실험에서는 fsync 옵션을 적용하여 순차 쓰기, 랜덤 쓰기만 실험을 진행하였는데 실험 결과 fsync를 0으로로 설정하여 실험을 한 결과 Delayed Allocation On시 순차 쓰기의 경우 성능이 약 24% 저하되었고 랜덤 쓰기의 경우 약 17% 성능이 더 좋게 나왔다. fsync를 1로 설정하여 실험을 한 결과 전체적인 I/O 성능이 크게 저하되었다. 순차 쓰기, 랜덤 쓰기 둘다 Delayed Allocation On/Off에 따른 성능의 차이는 없었다. fsync를 1로 설정하면 쓰기에 대한 베퍼링이 완전히 사라지므로 성능이 저하되는 것을 알 수 있었다.

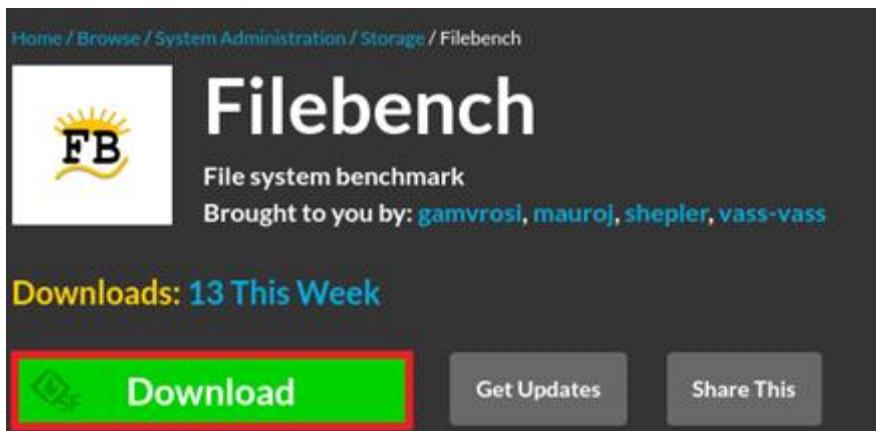
4.2 File bench

본 실험에서 Delayed Allocation On/Off 상태에서 SSD의 I/O 성능을 측정하기 위해 filebench 벤치마크를 사용했다 [12].

4.2.1 filebench 설치

a. filebench 홈페이지를 통해 다운로드 가능 (압축 파일) [그림 20]

- <https://sourceforge.net/projects/filebench>



[그림 20] filebench 다운로드

b. 각 버전마다 압축 파일의 종류가 다르고 각각의 압축 해제 방법이 다름

- 터미널에서 명령어를 통해 압축 해제를 하는 방법
- 지정 폴더에 직접 들어가서 압축 해제를 하는 방법

c. 터미널을 통해 압축 해제를 하는 방법 (filebench 기준 3가지 종류의 압축파일)

- **cd** 명령어를 사용 (change directory 명령 - 해당 폴더 진입)
- ex) **cd Download** - 다운로드 폴더 진입
- **tar** 파일 => **tar -xvf (파일명).tar**
- **tar.gz** 파일 => **tar -zxvf (파일명).tar**
- **zip** => **unzip (파일명).zip**

d. 압축이 해제된 폴더로 진입 (**cd** 명령어)

- ex) **cd filebench-1.5-alpha3**

e. filebench 설치 명령어 입력 전 사전 필수 프로그램 존재

- **sudo apt-get install make**
- **sudo apt-get install gcc**
- **sudo apt-get install g++**
- gcc와 g++의 경우 fio 설치시 자동 설치가 되지만 미설치의 경우 no acceptable C compiler found in \$PATH 오류 발생
- **sudo apt-get install bison**
미설치시 line 176:yacc:command not found, Error 127
오류 발생
- **sudo apt-get install flex**
미설치시 parser.lex.c, no such file or directory 오류 발생

f. **./configure** 명령어로 폴더 내에 configure 파일 실행

g. **sudo make**

- **make** 명령어로 **./configure** 이후 저장된 설치 명령어를 컴파일

h. **sudo make install**

- **make install**의 경우 반드시 **sudo**(관리자 권한으로 실행)가 필요

i. **filebench**를 입력하여 정상 설치 확인

```
hjh2020@hjh2020:~/Desktop/filebench-1.5-alpha3$ filebench
Usage error: No runtime options specified

Usage: filebench {-f <wmlscript> | -h | -c [cvartype]}

Filebench version 1.5-alpha3

Filebench is a file system and storage benchmark that interprets a script
written in its Workload Model Language (WML), and proceeds to generate the
specified workload. Refer to the README for more details.

Visit github.com/filebench/filebench for WML definition and tutorials.

Options:
  -f <wmlscript> generate workload from the specified file
  -h          display this help message
  -c          display supported cvar types
  -c [cvartype] display options of the specific cvar type
```

[그림 21] filebench 정상 설치 확인

4.2.2 실행 방법

a. 설치된 filebench/workload 폴더에서 사용할 워크로드 파일을 다른 곳으로 복사

- **fileserver.f** , **varmail.f** , **webserver.f** 파일을 Desktop 폴더로 복사

b. 메모장을 통해 각 파일들을 열어서 설정 값 입력

- 기본 설정 값에서 filebench를 실행시킬 폴더를 입력
- ex) /home/(계정이름)/Desktop/SSD
- 1.5 버전부터는 run 60이라는 실행에 대한 기본 설정이 되어있지만 그 이하 버전에서는 설정이 안되어있기 때문에 추가 입력이 필요 (초단위 설정)

c. **filebench -f (파일이름).f** 명령어로 해당 워크로드 실행

- ex) **filebench -f fileserver.f**
- 1.5 버전 이하의 경우 실행 시 filebench라는 명령어를 입력 후 filebench> 문구가 출력되고 나서 워크로드를 실행해야 했지만 1.5 버전부터 filebench> 출력 없이 바로 워크로드 실행이 가능

d. 실험 중 오류는 대부분의 경우 명령어 입력 후 실험이 진행되는 것처럼 보이지만 진행되지 않고 Starting 1 filereader instances 출력후 멈춤 [그림 22]

- 예외 발생시 명령어를 입력하여 해결
- 1.5 버전 이하 입력 명령어

`sudo echo 0 to /proc/sys/kernel/randomize_va_space`

- 1.5 버전 이후 입력 명령어

`echo 0 | sudo tee /proc/sys/kernel/randomize_va_space`

- [그림 23]처럼 결과 값이 출력되면 filebench 정상 작동

```
njh2020@njh2020:~/Desktop$ filebench -f fileserver.f
Filebench Version 1.5-alpha3
WARNING: Could not open /proc/sys/kernel/shmmmax file!
It means that you probably ran Filebench not as a root. Filebench will not increase shared
region limits in this case, which can lead to the failures on certain workloads.
0.000: Allocated 173MB of shared memory
0.001: File-server Version 3.0 personality successfully loaded
0.001: Populating and pre-allocating filesets
0.005: bigfileset populated: 10000 files, avg. dir. width = 20, avg. dir. depth
= 3.1, 0 leafdirs, 156.250MB total size
0.005: Removing bigfileset tree (if exists)
0.006: Pre-allocating directories in bigfileset tree
0.014: Pre-allocating files in bigfileset tree
0.194: Waiting for pre-allocation to finish (in case of a parallel pre-allocation)
0.194: Population and pre-allocation of filesets completed
0.194: Starting 1 filereader instances
```

[그림 22] filebench 실험 도중 멈춤

```
appendfilerand1    3361983ops    56028ops/s 437.6mb/s    0.0ms/op [0.00ms - 11.56ms]
openfile1         3361989ops    56029ops/s  0.0mb/s    0.0ms/op [0.00ms - 19.57ms]
closefile1        3361992ops    56029ops/s  0.0mb/s    0.0ms/op [0.00ms - 7.65ms]
wrtfile1          3361995ops    56029ops/s 875.4mb/s    0.0ms/op [0.01ms - 18.89ms]
createfile1       3361998ops    56029ops/s  0.0mb/s    0.1ms/op [0.01ms - 19.64ms]
61.205: IO Summary: 36981777 ops 616311.981 ops/s 56028/112057 rd/wr 2625.4mb/s
0.1ms/op
61.205: Shutting down processes
```

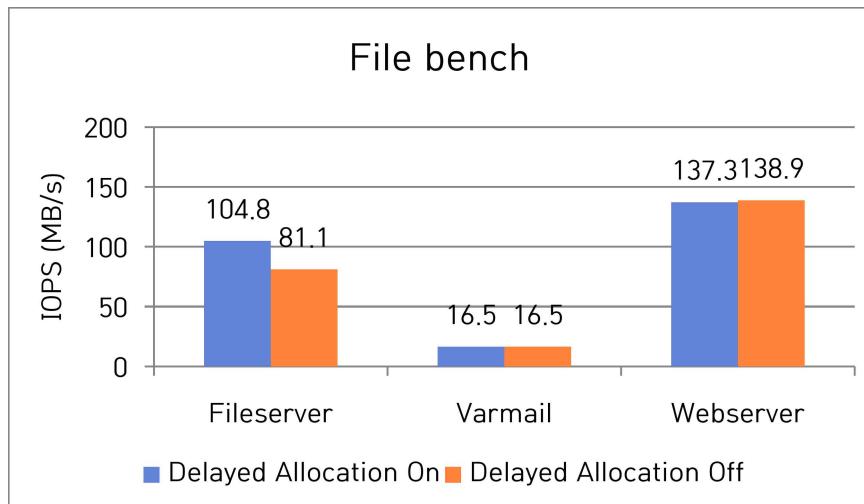
[그림 23] filebench 정상 작동

4.2.3 실험 진행

SSD에 순차 읽기/쓰기, 랜덤 읽기/쓰기를 진행하고 마운트시 각각 Delayed Allocation On/Off 상태를 적용하여 실험을 진행하였다. fio처럼 따로 text 파일로 생성 없이 명령어 입력후 나온 수치들을 그래프로 표현하였다.

- iops * 설정한 블록 단위 = 초당 데이터 전송량
- 초당 데이터 전송량의 단위가 KB/s이므로 MB/s로 환산

4.2.4 실험 결과



[그림 24] filebench 실험 결과

본 실험에서 SSD에 사용 중인 메모리보다 큰 40GB의 파일을 1시간 동안 실험 5번 진행하여 그 평균 값을 나타냈다. 시간과 파일 크기를 제외하고 옵션의 변화 없이 기본 옵션으로 실험을 진행한 결과 [그림 24]에서 볼 수 있듯이 Fileserver의 경우 Delayed Allocation On시 약 23% 더 성능이 좋은 결과가 나왔고, Varmail의 경우 Delayed Allocation On/Off에 따른 차이가 없었으며 Webserver의 경우 Delayed Allocation On시 약 2% 성능의 저하라는 결과가 나왔다. 기본 옵션에서 실행을 했는데 Delayed Allocation On시 Fileserver의 워크로드에서는 더 좋은 결과가 나왔다.

4.3 YCSB

야후에서 만든 클라우드 서비스의 성능을 측정하기 위한 벤치마크 툴로 특정 상황에서 NoSQL을 도입할 때 해당 NoSQL이 사용에 적합한지 또는 여러 개의 NoSQL중 어떤 것이 가장 적합한지 상황에 맞는 벤치마킹 환경 및 결과를 제공한다 [13]. Java를 기반으로 작성된 테스트로, 패키지 관리는 maven으로 작성되어 있다. 지원하는 NoSQL이 많고 Workload를 사용하는 것이 간단하다 [14].

기본 Workload가 대부분의 기능을 지원하기 때문에 특정 상황에 맞게 쉽게 시뮬레이션 할 수 있다. 각 Workload에서 지원하는 인터페이스는 5개로 init(초기화), insert(삽입), update(갱신), read(읽기), delete(삭제)가 있다 [15]. 5개를 활용한 Workload는 총 6개로 A부터 F로 나뉘어져 있는데 A는 업데이트 중심의 작업으로 읽기50%, 업데이트 50%를 수행하고 B는 읽기 중심의 작업으로 읽기 95%, 업데이트 5%로 한번의 태그 작성을 하고 주로 읽기 작업을 수행한다. C는 읽기 전용 작업으로 읽기 100%를 수행하고 D는 최근 레코드 중심의 읽기 작업으로 읽기 95%, 쓰기 5%로 최근 저장된 레코드를 중심으로 읽기를 수행한다. E는 영역 스캔으로 읽기 95%, 쓰기 5%로 각 작업마다 100개 내외의 레코드 영역을 한 번에 쿼리를 하고 F는 읽기-쓰기-수정으로 읽기, 수정, 쓰기 작업을 순서대로 수행한다.

4.3.1 YCSB 설치

a. Github을 통해 파일을 다운 받기 [16]

- **git clone** 명령어를 사용하여 다운
- git 설치 필요 - **sudo apt-get install git** [그림 25]

```
hjh2020@jh2020:~/Desktop$ sudo apt-get install git
选取时，将显示此包的摘要信息
正在读取状态信息... 完成
正在分析软件包...
正在准备安装...
正在安装 git (1:2.27.0-1~deb10u1) ...
正在安装 liberror-perl (0.30-1)
正在安装 git-daemon-run | git-daemon-sysvinit git-doc git-el git-email git-gui gitk
正在安装 gitweb git-cvs git-mediawiki git-svn
正在安装 git-man liberror-perl
正在安装 git-man liberror-perl
0个升级, 3个新安装, 0个卸载以及 24个升级未安装。
5,464 kB的档案将被下载。
此操作后 38.4 MB的磁盘空间将被使用。
继续吗? [Y/n] y
```

[그림 25] git 설치

b. **git clone** 명령어를 통해 YCSB 다운 [그림 26]

- ex) **git clone <https://github.com/brianfrankcooper/YCSB>**

```
hjh2020@jh2020:~/Desktop$ git clone https://github.com/brianfrankcooper/YCSB
'YCSB'에 액세스합니다...
remote: Enumerating objects: 20365, done.
remote: Total 20365 (delta 0), reused 0 (delta 0), pack-reused 20365
正在接收对象: 100% (20365/20365), 31.51 MiB | 8.07 MiB/s, 完成.
正在解压缩对象: 100% (7950/7950), 完成.
```

[그림 26] git clone으로 YCSB 다운로드

c. 터미널 창에서 cd 명령어를 통해 설치된 YCSB 폴더 진입

- cd YCSB

d. mvn 명령어를 통해 패키지 설치

- `sudo apt-get install maven` - mvn 사용을 위해 maven 설치
- `mvn clean package` - 전체 패키지를 설치 [그림 28]
- `mvn -pl site.ycsb:[패키지 이름] -binding -am clean package` - 해당 패키지 한 개만 설치
- Build Success 출력시 설치 완료 [그림 29]

※ 주의사항

- maven 2 버전 사용 시 [issue/406](#) 에러 발생
- `maven 3` 버전 설치 필수 [그림 27]

```
hjh2020@jh2020:~/Desktop$ sudo apt-get install maven
```

매기시 죽죽 끊는 경입니다... 진도
의존성 트리를 만드는 중입니다.
상태 정보를 읽는 중입니다. 와蒌
패키지 maven은 이미 최신 버전입니다 (3.6.3-1).
0개 업그레이드, 0개 새로 설치, 0개 세기 및 24개 업그레이드 안 함.

[그림 27] maven 설치 및 버전 확인

```
hjh2020@jh2020:~/Desktop/YCSBS mvn clean package
```

```
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by com.google.inject.internal.cglib.core.$ReflectUtils$1
(file:/usr/share/maven/lib/guice.jar) to method java.lang.ClassLoader.defineClass(java.lang.String,byte[],int,int,java.security.ProtectionDomain)
WARNING: Please consider reporting this to the maintainers of com.google.inject.internal.cglib.core.$ReflectUtils$1
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
[INFO] Scanning for projects...
[INFO] -----
[INFO] Reactor Build Order:
[INFO] YCSB Root [pom]
```

[그림 28] mvn clean package (전체 패키지 설치)

```
[INFO] S3 Storage Binding ..... SUCCESS [ 3.191 s]
[INFO] Solr Binding ..... SUCCESS [ 41.088 s]
[INFO] Solr 6 Binding ..... SUCCESS [ 33.727 s]
[INFO] Solr 7 Binding ..... SUCCESS [ 52.833 s]
[INFO] Tarantool DB Binding ..... SUCCESS [ 0.879 s]
[INFO] TableStore DB Binding ..... SUCCESS [ 7.456 s]
[INFO] VoltDB Binding ..... SUCCESS [ 16.131 s]
[INFO] YCSB Release Distribution Builder ..... SUCCESS [ 16.355 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 18:59 min
[INFO] Finished at: 2020-05-19T03:58:13+09:00
[INFO] -----
```

[그림 29] Build Success 출력시 설치 완료

4.3.2 실행 방법

a. 진행하려 하는 rocksdb에 워크로드 불러오기, 실행 순으로 진행

[그림 31],[그림 32] [17]

- `./ycsb load rocksdb -s -P workloads/[워크로드 종류]
-p rocksdb.dir=[마운트 위치]`
- `./ycsb run rocksdb -s -P workloads/[워크로드 종류]
-p rocksdb.dir=[마운트 위치]`
- YCSB 실행시 python 설치 필수
`sudo apt-get install python`

b. 해당 디렉토리 내에 생성된 폴더에서 log 파일에서 값을 확인

```
hjh2020@jhj2020:~/Desktop/YCSB$ ./bin/ycsb  
usage: ./bin/ycsb command database [options]  
  
Commands:  
  load      Execute the load phase  
  run       Execute the transaction phase  
  shell     Interactive mode
```

[그림 30] ./bin/ycsb 명령어로 기본 실행 및 설치 확인

```
hjh2020@jhj2020:~/Desktop/YCSB$ ./bin/ycsb load rocksdb -s -P workloads/workload  
a -p rocksdb.dir=/home/hjh2020/Desktop/SSD/test  
[WARN] Running against a source checkout. In order to get our runtime dependencies we'll have to invoke Maven. Depending on the state of your system, this may take ~30-45 seconds  
[DEBUG] Running 'mvn -pl site.ycsb:rocksdb-binding -am package -DskipTests dependency:build-classpath -DincludeScope=compile -Dmdep.outputFilterFile=true'  
WARNING: An illegal reflective access operation has occurred  
WARNING: Illegal reflective access by com.google.inject.internal.cglib.core.$ReflectUtils$1 (file:/usr/share/maven/lib/guice.jar) to method java.lang.ClassLoader.defineClass(java.lang.String,byte[],int,int,java.security.ProtectionDomain)  
WARNING: Please consider reporting this to the maintainers of com.google.inject.internal.cglib.core.$ReflectUtils$1
```

[그림 31] rocksdb에 워크로드를 불러오기

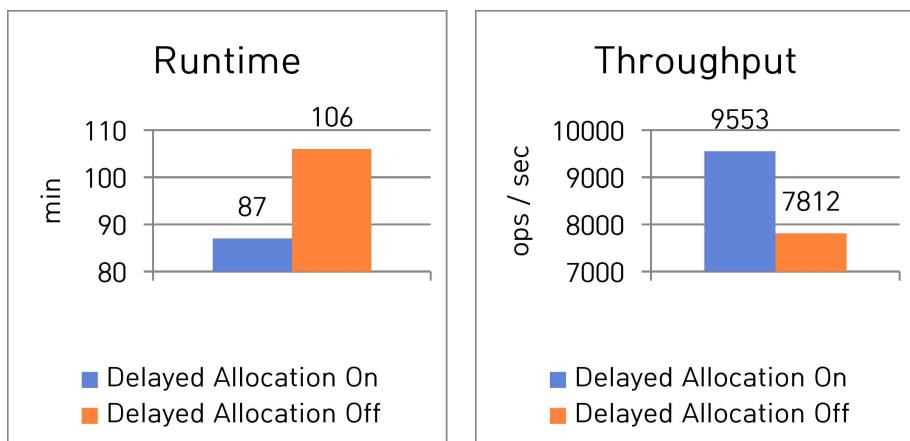
```
hjh2020@jhj2020:~/Desktop/YCSB$ ./bin/ycsb run rocksdb -s -P workloads/workloada  
-p rocksdb.dir=/home/hjh2020/Desktop/SSD/test  
[WARN] Running against a source checkout. In order to get our runtime dependencies we'll have to invoke Maven. Depending on the state of your system, this may take ~30-45 seconds  
[DEBUG] Running 'mvn -pl site.ycsb:rocksdb-binding -am package -DskipTests dependency:build-classpath -DincludeScope=compile -Dmdep.outputFilterFile=true'  
WARNING: An illegal reflective access operation has occurred  
WARNING: Illegal reflective access by com.google.inject.internal.cglib.core.$ReflectUtils$1 (file:/usr/share/maven/lib/guice.jar) to method java.lang.ClassLoader.defineClass(java.lang.String,byte[],int,int,java.security.ProtectionDomain)  
WARNING: Please consider reporting this to the maintainers of com.google.inject.internal.cglib.core.$ReflectUtils$1
```

[그림 32] rocksdb에 워크로드를 실행하기

4.3.3 실험 결과

총 A부터 F까지 6개의 Workload가 존재하는데 각 Workload마다 Record Count와 Operation Count를 50,000,000으로 설정하고 Delayed Allocation On/Off 옵션을 적용하여 비교하며 실험을 진행하였다. 본 실험에서 처리 시간인 runtime과 초당 처리량(ops)를 측정하여 비교하였다.

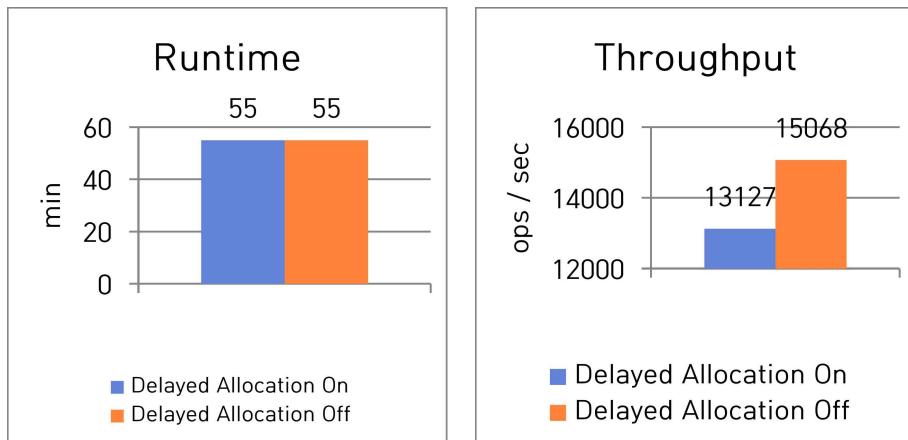
Workload A



[그림 33] Workload A 실험 결과

Workload A의 실험 결과 Runtime의 경우 Delayed Allocation On시 약 18% 더 빠르게 끝났으며 처리량의 경우 Delayed Allocation On시 약 19% 더 많았다. Workload A의 경우 Delayed Allocation On시 더 많은 양을 더 빠르게 처리하였기 때문에 성능이 좋게 나오는 것을 확인 할 수 있었다.

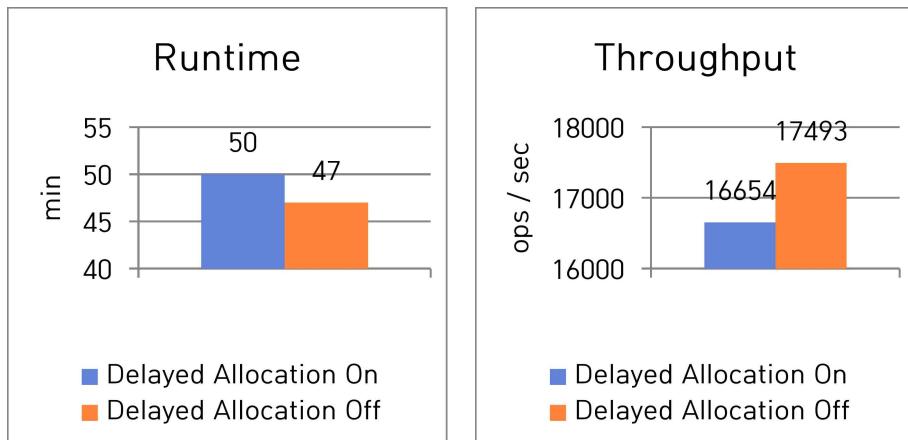
Workload B



[그림 34] Workload B 실험 결과

Workload B의 실험 결과 Runtime의 경우 Delayed Allocation On/Off 차이가 없었으며 처리량의 경우 Delayed Allocation On시 약 13% 더 났다.

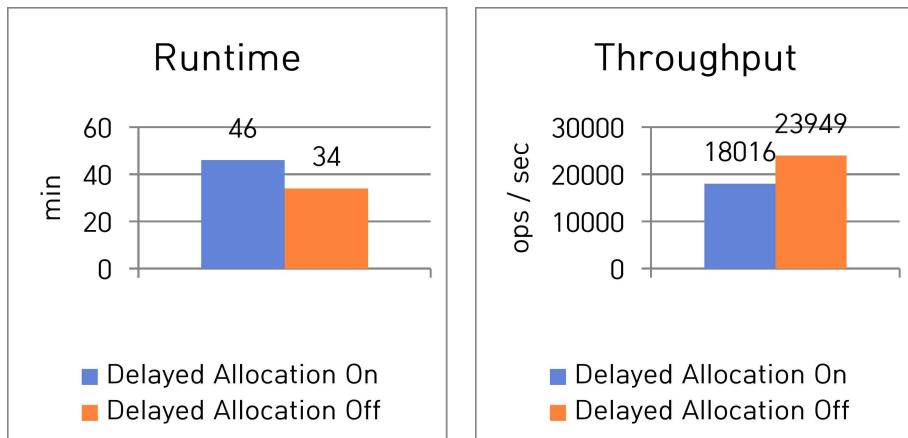
Workload C



[그림 35] Workload C 실험 결과

Workload C의 실험 결과 Runtime의 경우 Delayed Allocation On시 약 6% 더 느리게 끝났고 처리량의 경우 Delayed Allocation On시 약 5% 더 났다.

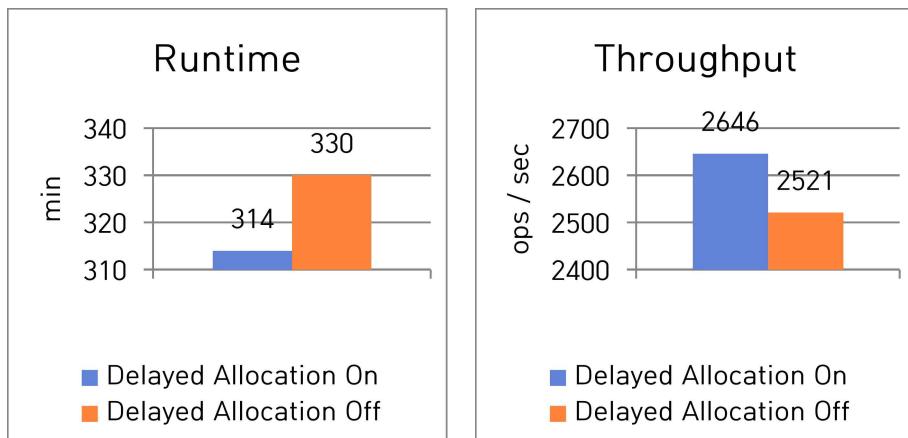
Workload D



[그림 36] Workload D 실험 결과

Workload D의 실험 결과 Runtime의 경우 Delayed Allocation On시 약 27% 더 느리게 끝났고 처리량의 경우 Delayed Allocation On시 약 25% 더 낮았다.

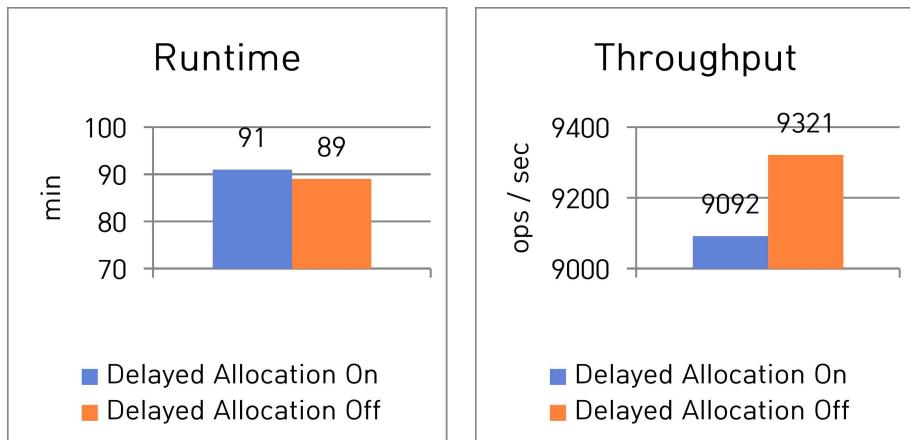
Workload E



[그림 37] Workload E 실험 결과

Workload E의 실험 결과 Runtime의 경우 Delayed Allocation On시 약 5% 더 빠르게 끝났으며 처리량의 경우 Delayed Allocation On시 약 5% 더 많았다.

Workload F

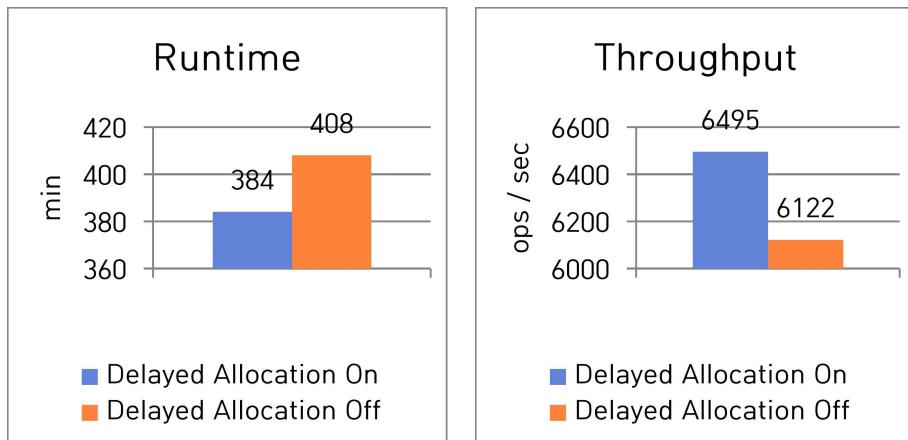


[그림 38] Workload F 실험 결과

Workload F의 실험 결과 Runtime의 경우 Delayed Allocation On 시
약 3% 더 느리게 끝났고 처리량의 경우 Delayed Allocation On 시 약 3%
더 낮았다.

YCSB의 6개의 Workload를 Delayed Allocation On/Off 옵션 상태에 따라
실험을 진행한 결과 Workload A와 Workload E에서 더 좋은 성능이
측정이 되었다.

Workload A (추가 실험)



[그림 39] Workload A 추가실험 결과

추가 실험으로 성능이 좋았던 Workload A의 Record Count와 Operation Count를 150,000,000으로 설정하여 Runtime을 늘려서 실험을 진행하였다. 실험 결과 Runtime의 경우 Delayed Allocation On시 약 6% 더 빠르게 끝났으며 처리량의 경우 Delayed Allocation On시 약 6% 더 많았다.

4.4 Blktrace

blktrace는 linux 환경에 사용하는 block trace로 Block I/O Layer에서 스토리지로 입출력이 일어나는 과정을 분석해주는 추적 메커니즘으로 DB 접근에서 주로 DB recovery나 join시 입출력 패턴을 알아내는데 유용하다 [18]. blktrace가 분석한 내용들을 blkparse를 통해 parsing하여 확인할 수 있다 [19].

4.4.1 blktrace 설치

- a. **sudo apt-get install blktrace** - blktrace 설치 [그림 40]

```
hjh2020@hjh2020:~/Desktop$ sudo apt-get install blktrace
패키지 목록을 엽는 중입니다... 완료
의존성 트리를 만드는 중입니다
상태 정보를 읽는 중입니다... 완료
다음의 추가 패키지가 설치될 것입니다 :
ffmpeg:i386-v-a-driver intel-media-v-a-driver libaacso libaom libass9
libavcodec58 libavdevice58 libavfilter7 libavformat58 libavresample4
libavutil56 libbdplus0 libbluray2 libbs2b0 libchromaprint1 libcodec2-0.9
[그림 40] blktrace 설치
```

4.4.2 실행 방법

- a. 벤치마크를 실행하면서 blktrace를 실행하여 처리량을 측정하고 blkparse를 통해 측정한 결과 값을 분석 [20]
- b. **sudo blktrace -d [마운트 경로] -a [방식] -w [시간(초단위)]**
 - ex) **sudo blktrace -d /dev/sda1 -a issue -w 3600** [그림 41]
 - -o 옵션을 추가하여 trace 결과 값 파일 생성 가능
 - -o 옵션 미입력시 스토리지 이름으로 시작하는 파일 자동 생성
 - ex) sda1.blktrace.0
- c. **sudo blkparse -i [만들어진 파일 이름] -o [출력 파일]**
 - blktrace로 만들어진 파일들을 parsing하여 출력값 생성
 - ex) **blkparse -i sda1.blktrace.0 -o result** [그림 42]
 - -o 옵션 미사용시 결과값 파일이 생성없이 화면에 결과값 출력

```

hjh2020@hjh2020:~/Desktop/YCSB$ sudo blktrace -d /dev/sda1 -a issue -w 3000
[sudo] password for hjh2020: 남모:
== sda1 ==
CPU 0:          447804 events,   20991 KiB data
CPU 1:          573816 events,   26898 KiB data
CPU 2:          524258 events,   24575 KiB data
CPU 3:          567946 events,   26623 KiB data
CPU 4:          477693 events,   22392 KiB data
CPU 5:          480871 events,   22541 KiB data
CPU 6:          394009 events,   18470 KiB data
CPU 7:          4341641 events,  203515 KiB data
CPU 8:          4059003 events,  190266 KiB data
CPU 9:          1172379 events,   54956 KiB data
CPU 10:         725534 events,   34010 KiB data
CPU 11:         633477 events,   29695 KiB data
Total:          14398431 events (dropped 0),   674927 KiB data

```

[그림 41] blktrace 실행

```

Total (sda1):
Reads Queued:    0,      0KiB Writes Queued:    0,      0KiB
Read Dispatches: 13940K, 165451MB Write Dispatches: 457973, 284031MB
Reads Requeued:  0,      0KiB Writes Requeued:  0,      0KiB
Reads Completed: 0,      0KiB Writes Completed: 0,      0KiB
Read Merges:     0,      0KiB Write Merges:     0,      0KiB
PC Reads Queued: 0,      0KiB PC Writes Queued: 0,      0KiB
PC Read Disp.:   217,    59KiB PC Write Disp.: 0,      0KiB
PC Reads Req.:   0,      0KiB PC Writes Req.: 0,      0KiB
PC Reads Compl.: 0,      0KiB PC Writes Compl.: 0,      0KiB
IO unplugs:      0,      0KiB Timer unplugs: 0,      0KiB

```

[그림 42] blkparse 실행

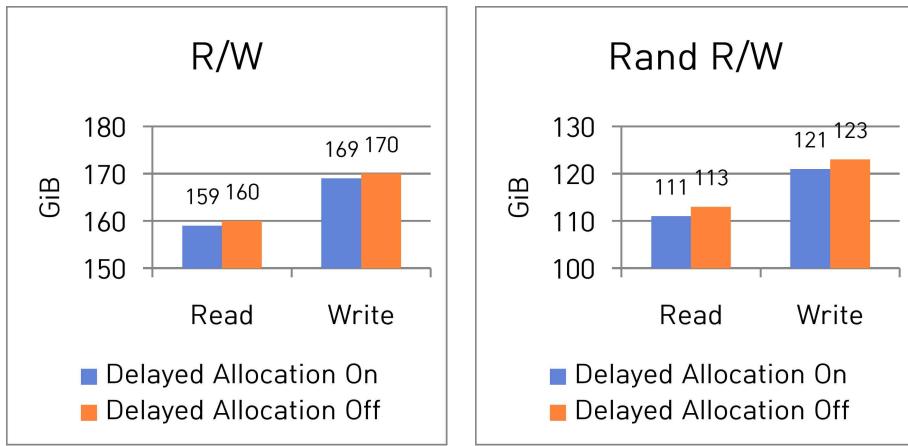
[표 3] blktrace 옵션

[표 4] blkparse 옵션

-b [size]	버퍼 사이즈 설정	-i [input file]	입력 파일
-d [storage]	실험 스토리지 설정	-o [name]	결과 파일 이름
-i [input-dev]	추적 데이터 입력	-O	텍스트 출력 X
-o [name]	출력 결과 파일 이름	-d	이진 출력 파일
-w [time]	실행 시간 설정(sec)	-f	출력 형식 설정
-a [mask]	필터 마스크 선택		

4.4.3 실험 결과

a. Fio

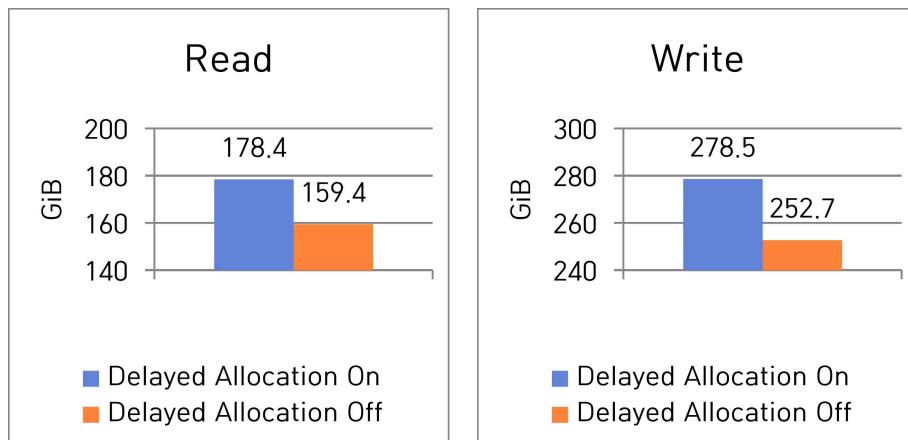


[그림 43] fio의 blktrace 실험 결과

fio 벤치마크를 사용하여 Delayed Allocation On/Off 옵션이 적용된 상태에서 1시간 동안 실험을 진행하였다. 방식은 rw와 randrw로 설정하여 읽기,쓰기를 한번에 진행하였다. Fio 실험을 하는 동시에 blktrace와 blkprase를 1시간동안 실행하여 Read와 Write 양을 측정하였고 나온 결과값으로 그래프를 그렸다.

실험 결과 fio 순차 읽기,쓰기 랜덤 읽기,쓰기 모두 Delayed Allocation On/Off시 성능에 대한 변화가 미비하였다. blktrace와 parse의 경우에도 Delayed Allocation On/Off시 성능에 대한 변화가 미비하였다.

b. YCSB Workload A



[그림 44] Workload A의 blktrace 실험 결과

YCSB의 Workload A의 Record Count와 Operation Count를 50,000,000으로 설정하여 Runtime을 변경하였고 Delayed Allocation On/Off 옵션을 적용하여 비교 실험을 진행하였다. YCSB 실험을 하는 동시에 blktrace와 blkparse를 1시간동안 실행하여 Read와 Write 양을 측정하였고 나온 결과값으로 그래프를 그렸다.

YCSB의 Workload A의 실험 결과 Runtime의 경우 Delayed Allocation On의 경우 약 12% 더 빠르게 끝났으며 처리량의 경우 약 11% 더 많았다. 본 실험에서도 역시 Workload A의 경우 Delayed Allocation On시 성능이 더 좋았다.

Workload A와 동시에 blktrace 및 blkparse의 결과 Delayed Allocation On시 Read의 경우 약 11%, Write의 경우 약 10% 더 좋은 성능의 결과가 나왔다.

제5장. 결론

본 논문에서 Linux의 Ext4 파일시스템에서 Delayed Allocation On/Off 옵션에 대한 효율성을 확인하기 위해 여러 실험을 진행하였다. Fio, Filbench, YCSB, Blktrace/Blkparse를 사용하여 I/O 성능을 측정하였다. Fio 벤치마크의 경우 Delayed Allocation On시 순차 쓰기, 랜덤 쓰기를 할 때 더 좋은 성능을 보이는 결과를 얻었고, Filebench의 경우 Fileserver Workload를 실행하였을 때 더 좋은 결과를 얻었다. YCSB의 경우 Workload A와 Workload E를 실행하였을 때 더 좋은 성능의 결과가 나왔고 그 중에서도 Workload A에 대한 실험에서 가장 좋은 성능의 결과를 확인할 수 있었다. 각각의 벤치마크로 실험을 진행하면서 Blktrace를 통해 Read, Write의 양을 측정한 결과 처리속도뿐 아니라 처리량에서도 더 좋은 성능이 측정이 된다는 것을 확인할 수 있었다.

향후 앞으로의 발전에서 SSD뿐 아니라 다양한 저장 장치들이 개발이 될텐데 더 빠르고 더 효율적인 활용을 위해 Delayed Allocation과 같이 디스크의 효율을 높일 수 있는 다양한 방법들로 I/O 성능을 비교하고 새로운 방법들을 찾아내며 분석할 예정이다.

참 고 문 헌

[1] HDD

https://en.wikipedia.org/wiki/Hard_disk_drive

[2] SSD

https://en.wikipedia.org/wiki/Solid-state_drive

[3] File System

https://en.wikipedia.org/wiki/File_system

[4] Ext4

<https://en.wikipedia.org/wiki/Ext4>

[5] Allocate-on-flush

<https://en.wikipedia.org/wiki/Allocate-on-flush>

[6] Disk partitioning

https://en.wikipedia.org/wiki/Disk_partitioning

[7] format

<https://en.wikipedia.org/wiki/Format>

[8] mount

[https://en.wikipedia.org/wiki/Mount_\(computing\)](https://en.wikipedia.org/wiki/Mount_(computing))

[9] fio – Flexible I/O tester rev. 3.20

https://fio.readthedocs.io/en/latest/fio_doc.html

[10] fio 기본 실행법

<https://harryp.tistory.com/664>

[11] fio(1) – Linux man page

<https://linux.die.net/man/1/fio>

[12] buffer cache

https://en.wikipedia.org/wiki/Disk_buffer

[13] Filebench 설치

<https://sourceforge.net/projects/filebench>

[14] YCSB

<https://en.wikipedia.org/wiki/YCSB>

[15] YCSB 워크로드

<https://sonhyunwoong.tistory.com/15>

[16] YCSB 설치

<https://github.com/brianfrankcooper/YCSB>

[17] YCSB 기본 실행법

<https://github.com/brianfrankcooper/YCSB/wiki/Running-a-Workload>

[18] blktrace(8) – Linux man page

<https://linux.die.net/man/8/blktrace>

[19] blkparse(1) – Linux man page

<https://linux.die.net/man/1/blkparse>

[20] blktrace, blkparse 기본 실행법

<https://ji007.tistory.com/entry/Block-IO-Layer-Tracing-blktrace>

[21] 이혜지, 이민호, 엄영익, “Ext4 파일시스템에서의 지연할당 성능 분석” 한국정보과학회 학술발표논문집, 2017

[22] 송용주, 이민호, 강동현, 엄영익, “Filebench 벤치마크 I/O 특성 분석”, 한국정보과학회 학술발표논문집, 2015