저작자표시-비영리-변경금지 2.0 대한민국

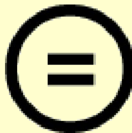
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.


변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사 학위논문

BIUS: 사용자 수준의 블록 장치 드라이버

2021년 8월

서울대학교 대학원

컴퓨터공학부

정 연 규

BIUS: 사용자 수준의 블록 장치 드라이버

지도 교수 김 진 수

이 논문을 공학석사 학위논문으로 제출함
2021년 6월

서울대학교 대학원
컴퓨터공학부
정 연 규

정연규의 공학석사 학위논문을 인준함
2021년 7월

위 원 장 이 재 진

부위원장 김 진 수

위 원 Bernhard Egger

초 록

블록 저장 장치는 운영체제에서 데이터를 저장하기 위해 사용하는 핵심 요소로, 최근 SMR 하드디스크나 ZNS SSD와 같이 기존의 블록 저장 장치와는 다른 특징을 갖는 장치들이 출시되고 있는 한편, 리눅스의 device mapper와 같이 다양한 기능을 수행하는 블록 장치에 대한 수요도 늘어나고 있다. 실제 블록 장치에 대한 에뮬레이션이나 가상의 블록 장치 구현을 사용자 수준에서 할 수 있게 되면 커널 수준에서 개발할 때에 비해 더욱 빠르고 안전하게 개발할 수 있게 된다. BIUS는 사용자 수준에서 블록 장치를 구현할 수 있는 단순하지만 효율적인 환경을 제공한다.

주요어 : 블록 장치, 사용자 수준, 구역화된 블록 장치
학　번 : 2019-21955

목 차

제1장 서론 1
제2장 배경 지식 2
2.1 구역화된 블록 저장 장치 2
2.1.1 구역의 상태 2
2.1.2 구역에 대한 요청 3
제3장 관련 연구 4
3.1 NBD[9] 4
3.1.1 NBD 의 동작 방식 4
3.1.2 NBD 기반 사용자 수준 블록 장치 4
제4장 BIUS 구현 5
4.1 BIUS 개요 5
4.2 블록 장치 등록 및 연결 6
4.3 데이터 전달 기법 6
4.3.1 데이터 복사 6
4.3.2 데이터 매핑 7
제5장 실험 결과 9
5.1 실험 환경 9
5.2 입출력 블록 크기에 따른 오버헤드 비교 9
5.3 fio 벤치마크 10
5.4 램 디스크 성능 비교 11
제6장 활용 사례 13
6.1 F2FS 위의 RocksDB 13
6.2 ZenFS 위의 RocksDB 14
제7장 결론 16
참고 문헌 17
Abstract 18

표 목차

표 1. BIUS 요청의 유형 6
표 2. 실험 환경 9

그림 목차

그림 1. 구역의 상태 3
그림 2. BIUS의 구조 5
그림 3. 입출력 블록 크기에 따른 순차 읽기 속도 9
그림 4. fio 벤치마크 결과 10
그림 5. 램 디스크 성능 비교 결과 11
그림 6. 구역의 크기에 따른 쓰기 양의 변화 13
그림 7. 구역의 크기에 따른 F2FS에 의한 쓰기 증폭량 13
그림 8. 공간 사용량에 따른 구역의 수 15
그림 9. 구역의 크기에 따른 공간 증폭량 15

제1장 서론

파일 시스템은 전통적으로 커널 수준으로 구현되어 왔으나, 사용자 수준으로 구현된 파일 시스템 또한 점차 늘어나고 있다. 사용자 수준에서 파일 시스템을 구현하게 되면 다음과 같은 장점을 얻을 수 있다. (1) 커널 공간에 구현할 때와 달리 다양한 프로그래밍 언어와 라이브러리를 사용할 수 있으며, 이 덕분에 빠르게 프로토타입을 만들 수 있다. (2) 코드에 버그가 있을 때 커널 코드는 전체 시스템을 불안정하게 만들 수 있지만 유저 코드는 제한된 범위의 피해만을 입힌다. (3) 더 높은 이식성을 가진다. 이러한 특성으로 인해 사용자 수준의 파일시스템 프레임워크인 libfuse가 널리 사용되고 있으며, 이에 대한 연구도 진행되어 왔다[1, 2].

최근에는 수십 MiB/s의 전송속도를 갖는 하드디스크부터 수 GiB/s의 전송속도를 갖는 NVMe SSD가 상용화되어 있으며, SMR(Shingled Magnetic Recording) 하드디스크나 ZNS(Zoned NameSpace) SSD와 같이 기존의 블록 저장 장치와는 다른 특성을 갖는 다양한 블록 저장 장치도 사용되고 있다. 뿐만 아니라, 소프트웨어 레이드, 블록 스냅샷, 실시간(on-the-fly) 암호화 등의 다양한 기능을 지원하는 가상의 블록 장치가 리눅스의 커널에 구현되어 사용되고 있다[3].

파일 시스템 요청과 마찬가지로 블록 장치에 대한 요청도 사용자 수준에서 받아 처리할 수 있게 되면 커널 수준에서 개발할 때에 비해 새로운 종류의 블록 장치를 에뮬레이션 하거나 새로운 기능을 갖는 가상의 블록 장치를 프로토타이핑 할 때 더 빠르고 안전하게 개발할 수 있게 된다는 이점이 생긴다. 또한 아직 ZNS SSD는 상용화가 되어 있지 않기 때문에 실물 장치가 없더라도 다양한 환경에서 소프트웨어 스택을 테스트해보고 최적화할 수 있게 되는 효과를 거둘 수 있다. 본 논문에서는 사용자 수준에서 기존의(conventional) 블록 장치뿐만 아니라 구역화된(zoned) 블록 장치를 구현할 수 있는 프레임워크인 BIUS(Block device In UserSpace)를 제안하고 그 효용성을 보일 것이다.

제2장 배경 지식

2.1 구역화된 블록 저장 장치

기존의 블록 저장 장치는 블록들의 거대한 배열로 표현되며, 임의의 위치에서 블록 단위로 읽기와 쓰기가 가능하다. 반면에 구역화된 블록 저장 장치는 여러 구역의 배열로 표현되며 하나의 구역이 여러 블록의 배열로 표현된다. 구역화된 블록 저장 장치에서 읽기는 임의의 위치에서 가능하나, 쓰기는 각 구역 내에서 순차적으로만 할 수 있다. 만일 한 구역 내에 이미 기록된 정보를 수정하거나 삭제하고 싶다면 구역 전체를 초기화해야 한다[5]. 하나의 구역은 장치에 따라 수백 MiB에서 1GiB가량의 크기를 가지며 한 장치 내에서 모든 구역은 동일한 크기를 갖는다.

ZNS SSD의 경우 기존 SSD에 비해서 GC(Garbage Collection)를 위한 예비 플래시의 수를 줄일 수 있고, FTL 매핑을 관리하기 위한 DRAM의 크기를 줄일 수 있기 때문에 비용 면에서 효율적이다. 또한 호스트가 수명이 비슷할 것으로 예상되는 데이터들을 같은 구역에 저장할 수 있어, GC로 인한 WAF(Write Amplification Factor)를 줄이고 지연 시간을 줄일 수 있어 업계의 관심을 받고 있다[6, 7].

2.1.1 구역의 상태

하나의 구역이 가질 수 있는 상태는 비어 있음(Empty), 암시적으로 열림(Implicitly Opened), 명시적으로 열림(Explicitly Opened), 닫힘(Closed), 가득 참(Full) 및 읽기 전용(Read only), 오프라인이 있다[8]. 각 상태에서 가능한 전이를 나타내면 그림 1과 같다. 한 번에 열려 있을 수 있는 최대 구역의 수는 장치 내 자원의 제약으로 인해 제한될 수 있으며 구역 내에서 오류가 발생할 경우 읽기 전용이나 오프라인으로 상태가 전환될 수 있다.

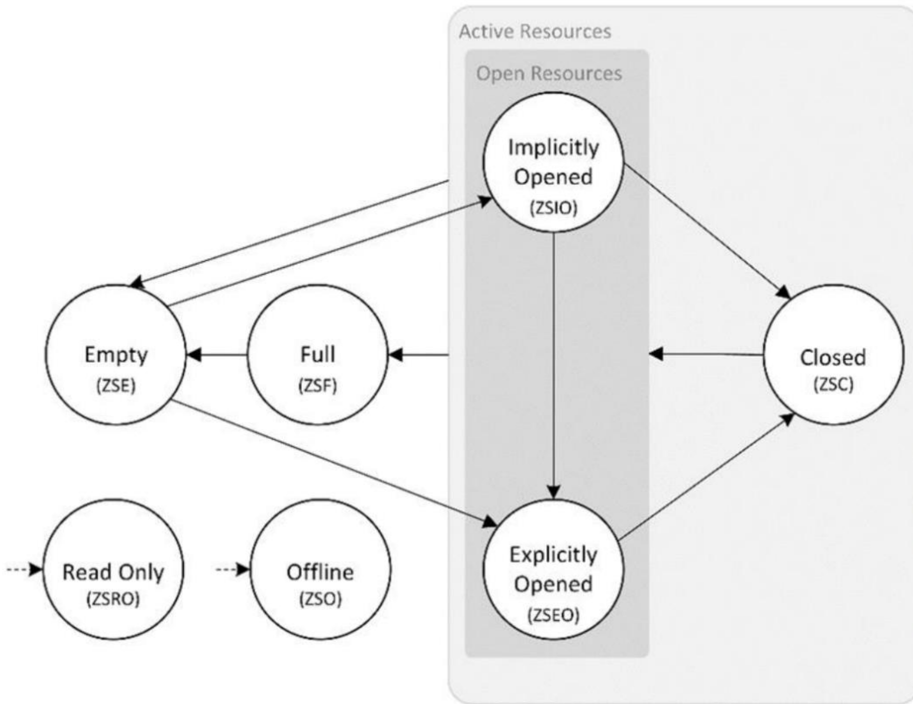


그림 1. 구역의 상태

2.1.2 구역에 대한 요청

사용자 수준의 어플리케이션은 기존 블록 장치에게 하는 것과 동일하게 구역화된 블록 장치에게 읽기나 쓰기 요청을 보낼 수 있다. 다만 쓰기 요청의 경우 요청한 위치가 그 구역의 쓰기 포인터의 위치와 맞지 않으면 오류가 발생할 수 있다.

구역 관리를 위한 요청으로는 구역 열기(zone open), 구역 닫기(zone close)가 있으며, 아직 가득 차지 않은 구역을 가득 찬 것과 같은 상태로 만드는 구역 끝내기(zone finish), 구역의 모든 데이터를 지우고 빈 상태로 만드는 구역 초기화(zone reset), 모든 구역의 데이터를 지우고 빈 상태로 만드는 전체 구역 초기화(zone reset all), 구역의 정보를 읽어오는 구역 보고(report zone)가 있다. 어플리케이션은 ioctl 시스템 콜을 통해 커널에게 이러한 요청들을 보낼 수 있다.

또한 대상 구역의 쓰기 포인터의 정확한 위치를 알아야 하는 쓰기 요청과는 달리 구역만을 지정해주면 해당 구역의 다음에 쓰일 공간에 데이터를 기록하고 그 위치를 반환해주는 구역 덧붙이기(zone append) 명령이 있다.

제3장 관련 연구

3.1 NBD[9]

기존의 리눅스 환경의 사용자 수준에서 블록 요청을 처리할 필요가 있을 때 주로 NBD(Network Block Device)가 사용되어 왔다. NBD는 서버, 클라이언트, 커널 모듈로 구성되는데, 네트워크를 통해 공유할 데이터를 가지고 있는 측에서 서버 프로그램을 실행하며, 서버가 공유한 데이터를 블록 장치로써 사용하는 측에서 클라이언트와 커널 모듈을 사용한다.

3.1.1 NBD의 동작 방식

NBD 드라이버는 로드 될 때 미리 지정된 개수만큼의 블록 장치를 생성한다. NBD 클라이언트는 서버와의 소켓 연결을 수립한 후 ioctl 시스템 콜을 통해 블록 장치에게 서버와 연결된 소켓을 전달한다. NBD 커널 모듈은 블록 장치에 요청이 들어오면 ioctl을 통해 건네 받은 소켓으로 요청을 전달한다. 이 경우 소켓이 NBD 서버와 직접 연결되어 있으므로 클라이언트는 연결을 수립한 후 별다른 역할을 하지 않는다.

만일 TLS 암호화를 적용하는 등 서버와 주고 받은 데이터를 가공하여 커널 모듈에게 건네 주어야 할 경우 NBD 클라이언트는 서버와 통신할 때 사용하는 소켓과는 별개의 유닉스 소켓을 만들어 그 소켓을 커널에게 전달한다. 이후 블록 요청은 커널과 연결된 유닉스 소켓을 통해 클라이언트에게 전달되며 클라이언트는 이 요청을 적절하게 가공하여 서버에게 전달한다.

3.1.2 NBD 기반 사용자 수준 블록 장치

이러한 NBD의 동작 방식을 이용하여 커널 모듈이 보내는 블록 요청에 적절하게 응답하는 NBD 서버 어플리케이션을 만들어 사용자 수준에서 블록 장치로 들어온 요청을 처리하도록 할 수 있다. 가상 디스크 파일을 실제 블록 장치처럼 사용할 수 있도록 해주는 qemu-nbd가 이를 이용하여 만들어졌으며[10], BUSE[11], nbdcpp[12]와 같은 범용 라이브러리도 만들어져 있다.

제4장 BIUS 구현

4.1 BIUS 개요

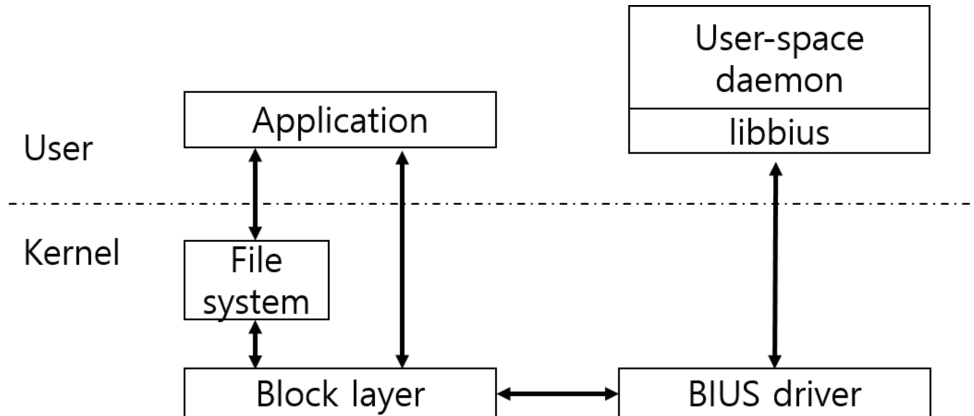


그림 2. BIUS의 구조

기존의 사용자 수준 블록 장치들은 단일 쓰레드만을 사용하고 구역화된 블록 장치를 지원하지 못한다. BIUS 커널 모듈은 mq 인터페이스를 사용하여 커널 내에서도 여러 쓰레드를 사용하여 요청을 처리할 수 있으며 구역화된 블록 장치에 대한 인터페이스 또한 지원하여 사용자 수준에서 보다 다양하고 고성능을 낼 수 있는 블록 장치를 구현할 수 있도록 해준다.

BIUS의 전체적인 구조는 그림 2와 같다. BIUS 커널 모듈이 로드되면 문자 장치(character device) `/dev/bius`를 등록한다. 유저 어플리케이션이 devtmpfs를 통해 BIUS 블록 장치에 직접 요청을 보내거나, BIUS 장치 위의 파일시스템에 접근하여 간접적으로 요청을 보내면 커널의 블록 계층을 거쳐 BIUS 드라이버에 요청이 전달된다. BIUS 드라이버는 큐에 요청을 저장하였다가 유저 데몬이 `/dev/bius` 장치에 read 시스템 콜을 보내면 큐에 저장된 블록 요청을 반환한다. 유저 데몬은 각 요청에 대해 등록된 콜 백 함수를 호출하여 요청을 처리한 후에 `/dev/bius` 장치에 write 시스템 콜을 보내 커널에게 요청의 결과를 알린다. 각 요청에는 64비트 정수로 고유한 ID가 부여되어 있어 커널은 어느 요청에 대한 응답인지 구분할 수 있다.

그룹	요청 명
관리	CREATE_DEVICE, CONNECT_DEVICE
블록 요청	READ, WRITE, DISCARD, FLUSH
구역 요청	REPORT_ZONES, ZONE_OPEN, ZONE_CLOSE, ZONE_FINISH, ZONE_APPEND, ZONE_RESET, ZONE_RESET_ALL

표 1. BIUS 요청의 유형

BIUS에서 사용하는 요청을 분류하면 표 1과 같다. 이 중 관리를 위해 사용되는 2개의 요청은 유저 데몬으로부터 커널로 보내는 요청이고, 블록 명령을 위한 4개의 요청과 구역 명령을 위한 7개의 요청은 커널에서 유저 데몬에게 보내는 요청이다.

4.2 블록 장치 등록 및 연결

사용자 수준에서 BIUS 데몬이 실행되면 주 쓰레드는 커널에게 CREATE_DEVICE 요청을 보낸다. 이 때 새로 만들 블록 장치가 기존(conventional) 블록 장치인지 구역화된(zoned) 블록 장치인지 지정할 수 있으며, 블록 장치의 이름, 크기 등을 지정할 수 있다. 이후 유저 데몬은 작업 쓰레드를 생성하며 각 작업 쓰레드는 커널에게 CONNECT_DEVICE 요청을 보내 생성된 블록 장치에 연결한다.

4.3 데이터 전달 기법

BIUS는 유저 데몬과 커널 드라이버 간의 데이터 전송을 위해 복사와 매핑의 2가지 방법을 지원한다.

4.3.1 데이터 복사

데이터 복사를 사용할 경우, 유저 데몬이 `/dev/bius` 장치에 대해 read 시스템 콜을 보내면 커널은 요청에 대한 정보가 담긴 헤더를 read 시스템 콜을 통해 전달된 버퍼에 쓴다. 쓰기 요청의 경우 버퍼에 남은 공간이 있으면 커널은 헤더 뒤에 데이터도 함께 복사한 후 시스템 콜을 반환한다. 유저 데몬은 read 시스템 콜의 반환 값과 헤더의 length 필드를 확인하여 얼마만큼의 데이터를 더 받아와야 하는지 확인할 수

있으며 데이터의 복사가 끝나지 않은 경우 다시 한 번 read 시스템 콜을 보내 나머지 데이터를 받아와야 한다. 유저 데몬은 요청을 처리한 후에 그 결과가 담긴 반환 헤더를 버퍼에 쓰고 write 시스템 콜을 보내어 커널에게 요청의 결과를 전달한다. 쓰기 요청의 경우 헤더 뒤에 데이터를 함께 보내거나, 헤더를 보낸 후에 다시 한 번 write 시스템 콜을 보내 커널이 데이터를 복사해갈 수 있게 해야 한다.

4.3.2 데이터 매핑

위의 데이터 복사 기법을 사용할 경우, 커널 내의 버퍼와 유저 데몬 내의 버퍼 간에 메모리 복사가 지속적으로 발생하며 이로 인한 오버헤드가 발생하게 된다. 이러한 오버헤드를 줄이고자 BIUS는 커널 내의 버퍼를 유저 데몬의 가상 주소 공간에 직접 매핑해주어 메모리 복사 횟수를 줄이는 기법을 지원한다.

데이터 매핑을 사용하고자 할 경우 유저 데몬은 `/dev/bius` 장치를 연 후에 mmap 시스템 콜을 통해 데이터가 매핑 될 가상 주소 공간을 예약해 두어야 한다. BIUS 드라이버는 mmap 시스템 콜을 받으면 해당 가상 공간 영역을 하나의 zero 페이지에 읽기 전용으로 매핑하며, 이 영역의 페이지 테이블 엔트리에 대한 포인터를 보관하여 둔다. 이후 유저 데몬에게 읽기나 쓰기 요청을 전달할 때 BIUS 드라이버는 유저 데몬의 가상 주소 공간에 `bio_vec`을 통해 전달받은 버퍼를 매핑한다. 매핑에 대한 정보는 요청 헤더를 통해 유저 데몬에게 전달한다.

불연속한 세그먼트

리눅스 커널은 하나의 블록 요청에 여러 개의 메모리 세그먼트를 포함시킬 수 있다. 한 세그먼트 안의 페이지들은 물리적으로 연속하며, 하나의 요청 안에 포함된 서로 다른 세그먼트들은 물리적으로 연속하지 않을 수 있다. 하나의 요청에 물리적으로 연속하지 않은 여러 세그먼트가 포함되어 있더라도 각 세그먼트의 시작 주소와 크기가 메모리 페이지 크기에 정렬되어 있으면 유저 데몬의 가상 메모리 공간에서 연속한 하나의 버퍼를 만들어 줄 수 있다. 하지만 그 시작이나 끝이 페이지 크기에 정렬되어 있지 않은 세그먼트가 있다면 이 페이지를 유저 데몬에게 매핑해줄 때 다음과 같은 2가지 문제가 발생한다. (1) 유저 데몬에게 전달되어야 할 버퍼가 가상 주소 공간에서 연속하지 않게

된다. (2) 페이지 내에 데이터 버퍼의 일부가 아닌 부분이 있다면 원래는 유저 데몬이 접근할 수 없어야 할 부분에 접근이 가능해지는 문제가 발생한다. 이 두 문제를 해결하기 위하여 페이지 크기로 정렬되지 않은 세그먼트가 있다면 정렬되지 않은 첫 페이지나 마지막 페이지를 직접 유저 데몬에게 매핑하는 대신 BIUS 드라이버가 보유한 예비 페이지를 매핑한다. 또한 정렬되지 않은 세그먼트가 2개 이상 존재할 경우, 각 세그먼트의 유저 가상 메모리 공간에서의 시작 주소와 크기가 담긴 리스트를 만들어 유저 데몬에게 전달한다. BIUS 드라이버는 이를 위해 각 연결이 수립될 때 연결 별로 몇 개의 페이지를 할당 받아 보관해 둔다.

PFN 매핑

페이지를 매핑할 때는 빠른 동작을 위해 page 구조체를 건들지 않고 페이지 테이블 엔트리만을 수정하는 방법을 사용한다. 이를 리눅스 커널에서는 PFN 매핑이라 부르며 이렇게 매핑 된 주소 공간을 구분하기 위해 VM_PFNMAP 플래그를 사용한다. 페이지 테이블 엔트리를 수정하기 전에는 캐시를 무효화해야 하며 엔트리를 수정한 후에는 TLB에 캐시 되어 있는 정보를 무효화해야 한다[4]. TLB를 무효화하는 작업은 상당한 오버헤드를 발생시키므로 mmap으로 할당된 가상 메모리 공간 전체가 아닌 매핑이 수정된 최소한의 영역에 대해서만 이루어져야 한다.

매핑은 블록 요청이 처리되는 도중에만 유지되며, 요청의 처리가 끝나는 즉시 zero 페이지로 매핑을 바꾸어 더 이상 유저 데몬이 접근할 수 없도록 한다.

제5장 실험 결과

5.1 실험 환경

실험에 사용된 환경은 아래 표 2와 같다.

CPU	AMD Ryzen 7 3800X 8-Core Processor @ 3.9GHz
Memory	DDR4 64GiB
Kernel	Linux 5.10.41
SSD	WD SN-750 1TB NVMe SSD @ PCIe Gen3 x4

표 2. 실험 환경

5.2 입출력 블록 크기에 따른 오버헤드 비교

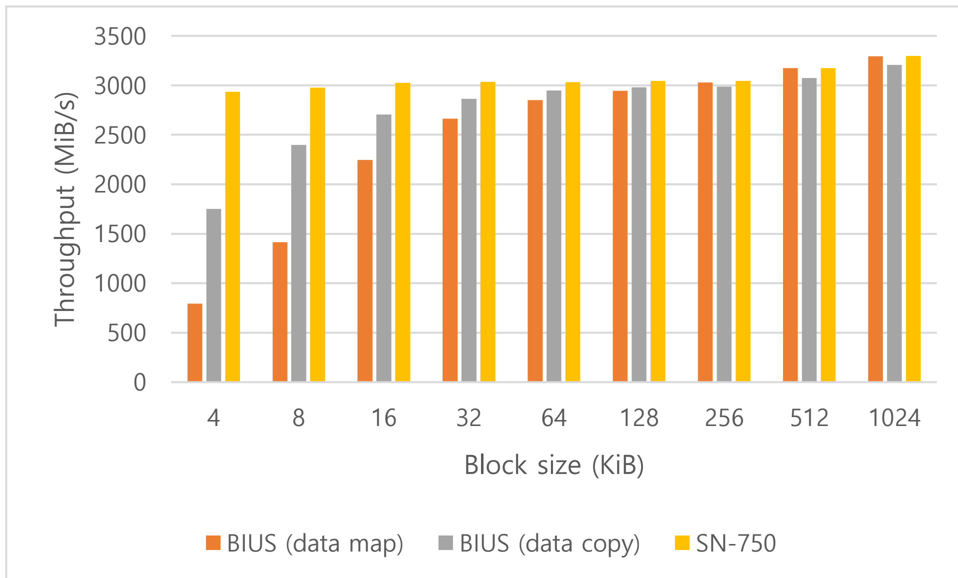


그림 3. 입출력 블록 크기에 따른 순차 읽기 속도

그림 3은 입출력 블록 크기를 바꿔가며 순차 읽기 실험을 한 결과이다. SN-750은 SSD에 직접 fio를 수행한 결과이며, BIUS는 각각 데이터 매핑과 데이터 복사만을 사용하도록 하였고, `pread`와 `pwrite`를 사용하여 실제 SSD에게 요청을 전달하였다. 유저 데몬과 fio가 SSD에 접근할 때는 버퍼를 사용하도록 하였으며, fio가 BIUS의 사용자 수준 블록 장치에 접근할 때는 원하는 크기의 요청을 BIUS 커널 모듈로부터 유저 데몬에 보낼 수 있도록 Direct IO를 사용하였다.

SSD에 직접 접근하여 순차 읽기는 수행했을 때는 모든 블록 크기에 대해 3000MiB/s 이상의 성능을 보였는데 이는 커널이 순차 읽기 패턴을 감지하고 readahead를 수행하였기 때문이다. 데이터 복사와 데이터 매핑 두 경우 모두 입출력 블록 크기가 작을 때 성능이 하락하는 경향을 보였는데 데이터 매핑에서 더욱 크게 오버헤드가 발생하는 모습을 보였다. 이는 매핑을 변경할 때 TLB flush를 수행하는 오버헤드가 크게 작용하기 때문이다. 다만 입출력 블록 크기가 256KiB보다 커졌을 때는 데이터 매핑이 데이터 복사보다 높은 성능을 보였다.

5.3 fio 벤치마크

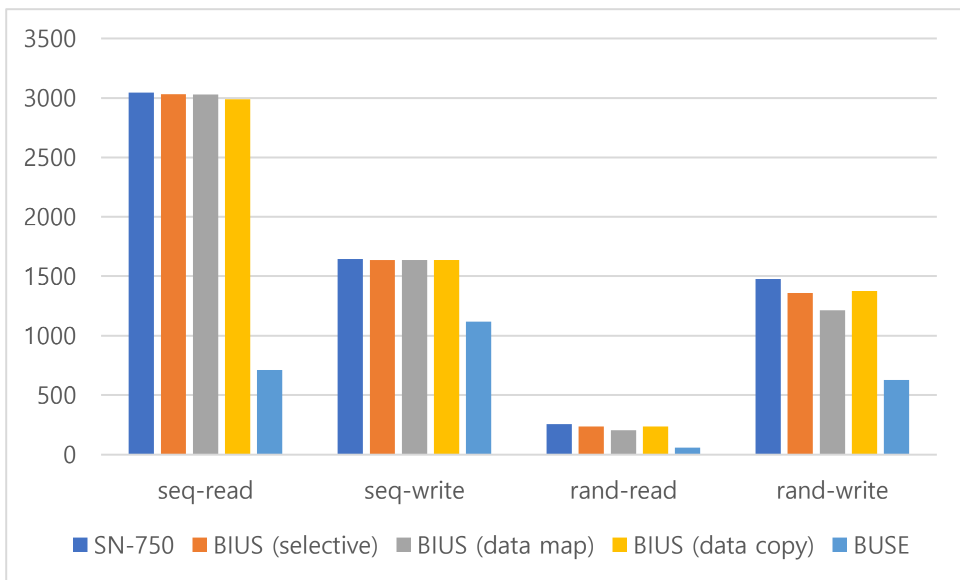


그림 4. fio 벤치마크 결과

그림 4는 fio를 사용해 순차 읽기/쓰기와 임의 읽기/쓰기의 성능을 비교한 결과이다. 좌측의 SN-750은 직접 SSD에 fio를 수행한 결과이며, 중간의 BIUS는 유저 데몬으로 들어온 요청을 `pread`와 `pwrite`를 사용해 실제 SSD로 전달하도록 한 결과이다. Selective는 128KiB를 초과하는 블록은 데이터 매핑을, 그 이하는 데이터 복사를 사용하도록 한 버전이고, data map과 data copy는 각각 데이터 매핑과 데이터 복사만을 사용하도록 한 버전이다. 제일 우측의 BUSE[11]는 NBD 커널 모듈이 보낸 요청을 `pread`와 `pwrite`를 통해 실제 SSD로 보내도록 한 결과이다. 모든 워크로드에서 fio의 입출력 엔진은 psync를

사용하였고, 4개의 입출력 쓰레드를 사용하도록 하였다. 순차접근은 256KiB 단위로 실험을 진행하였고 임의 접근은 4KiB 단위로 실험을 진행하였다. 모든 입출력은 버퍼를 사용하여 진행하도록 하였다.

순차 읽기에 대해서는 데이터 복사 방식을 사용했을 때 −1.8%p의 오버헤드를 보였으며, 데이터 매핑 방식을 사용했을 때 각각 −0.5%p의 오버헤드를 보여, 입출력 블록 크기가 큰 경우 사용자 수준과 커널 수준을 오가는 오버헤드가 크지 않은 것으로 나타났다. 임의 접근의 경우에는 데이터 복사 방식을 사용했을 때 읽기와 쓰기 각각 −7.0%p, −6.9%p의 오버헤드를 보였으며, 데이터 매핑을 사용했을 때 −20.0%p, −17.8%p의 오버헤드를 보였다. 임의 접근의 경우 작은 입출력 크기로 인해 빈번하게 사용자 수준과 커널 수준을 오가게 되어 비교적 큰 성능 저하가 발생했다.

BUSE의 경우는 모든 워크로드에서 SSD에 직접 접근할 때에 비해 낮은 성능을 보였는데, 이는 하나의 소켓을 통해 단일 쓰레드로 모든 블록 요청을 처리하는 구현이 주된 원인으로 보인다.

5.4 램 디스크 성능 비교

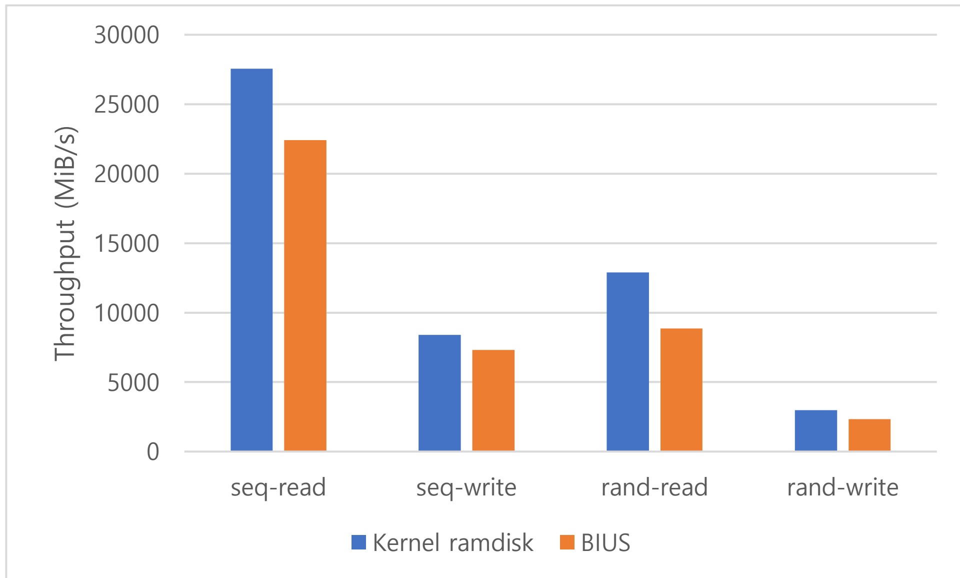


그림 5. 램 디스크 성능 비교 결과

그림 5는 커널 수준에서 구현한 램 디스크와 사용자 수준에서 구현한 램 디스크의 성능을 비교한 결과이다. 커널 램 디스크는 별도의 커널 모듈로 구현하여 실험하였고, 각각 커널 모듈과 유저 데몬에서

32GiB의 메모리를 할당 받은 후 블록 입출력을 처리하도록 하였다. 각 실험은 fio를 사용하여 진행했으며 순차 입출력의 경우 256KiB의 블록 크기를, 임의 입출력의 경우 4KiB의 블록 크기를 사용하였다. BIUS는 selective 모드로 동작하여 입출력 블록 크기가 큰 순차 접근 시에는 데이터 매핑 모드로 동작하고 임의 접근 시에는 데이터 복사 모드로 동작하게 하였다.

실험 결과 커널 수준에 구현된 램 디스크와 비교하여 BIUS를 사용하여 사용자 수준에 구현한 램 디스크가 순차 읽기에서 −18.6%p, 순차 쓰기에서 −12.9%p, 임의 읽기에서 −31.3%p, 임의 쓰기에서 −21.9%p의 성능 하락을 보여 순차 접근보다 임의 접근에서, 쓰기보다 읽기에서 보다 큰 오버헤드가 발생하는 모습을 확인할 수 있었다. 다만 모든 워크로드에서 커널 수준으로 구현된 램 디스크보다는 느리더라도 고성능의 SSD를 에뮬레이션하기에 충분한 성능을 보이는 것을 확인할 수 있었다.

제6장 활용 사례

6.1 F2FS 위의 RocksDB

아래는 BIUS로 구역의 크기를 바꿔가며 32GiB 크기의 램 디스크를 만들고 그 위에 F2FS 파일 시스템을 올린 후 YCSB A(읽기 50%, 업데이트 50%) 워크로드를 실행했을 때의 실험 결과를 나타낸 것이다. 초기에 800만 개의 레코드를 로드하였고, 값은 1000B의 크기를 갖도록 하였으며 6400만 회의 업데이트를 수행하도록 하였다.

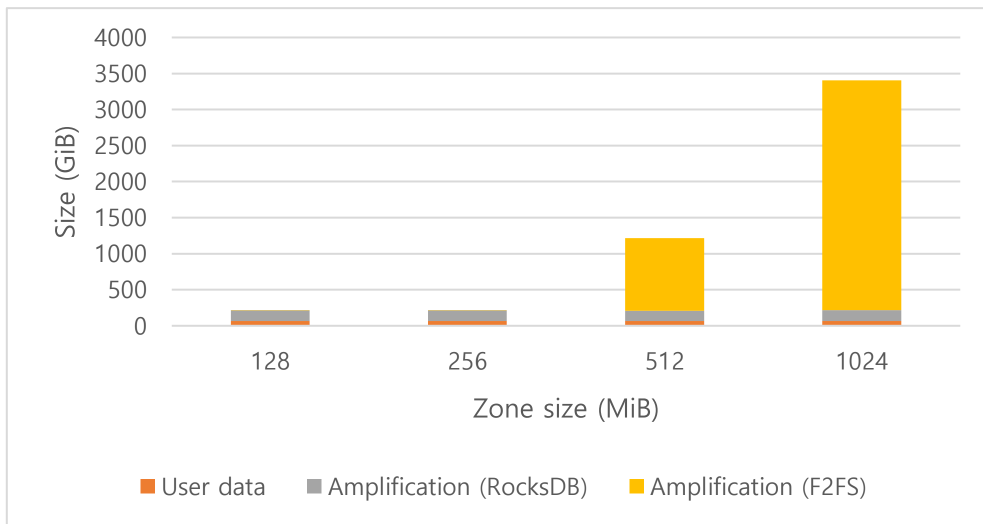


그림 6. 구역의 크기에 따른 쓰기 양의 변화

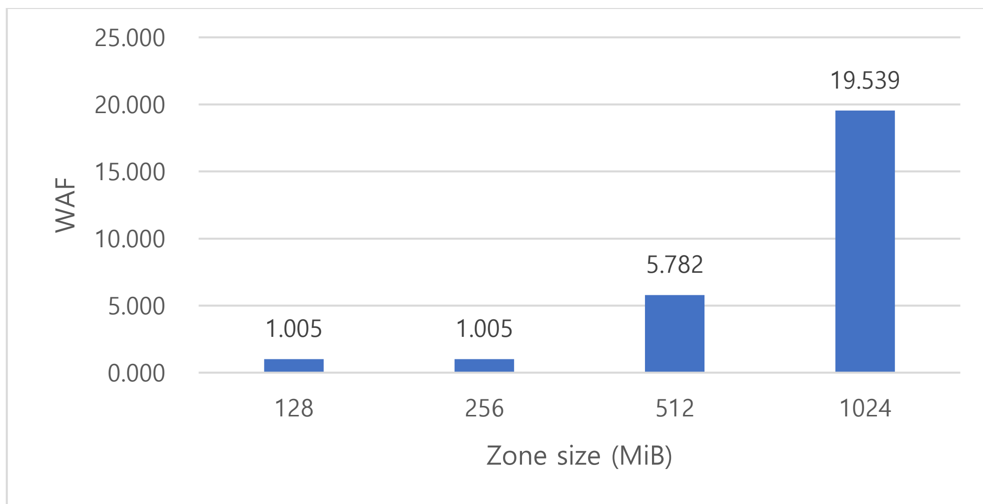


그림 7. 구역의 크기에 따른 F2FS에 의한 쓰기 증폭량

그림 6은 구역의 크기를 변화시켜가며 발생한 총 쓰기의 양을 나타낸 것이다. 막대 그래프 내에서 주황색으로 표시된 부분은 사용자 데이터에 해당하는 부분이고, 회색으로 표시된 부분은 RocksDB의 LSM 트리 구조로 인해 발생하는 쓰기 증폭량(write amplification), 노란색으로 표시된 부분은 F2FS 파일 시스템으로 인한 쓰기 증폭량이다.

레코드가 지속적으로 업데이트 되어가며 하나의 구역 내에 유효한 데이터와 삭제되어야 할 데이터가 뒤섞이게 되고 남은 공간이 부족해지면 유효한 데이터만을 다른 구역으로 이동시키는 GC(Garbage Collection) 작업이 필요해지게 된다. 구역의 크기가 작을 때는 사용 가능한 구역의 수가 많고 하나의 구역에 들어가는 데이터의 양이 적으므로 이러한 오버헤드가 적게 발생하지만, 구역의 크기가 커질수록 유효한 데이터를 옮기는 작업을 빈번하게 수행하게 되어 오버헤드가 커지는 것을 확인할 수 있었다. 이 중 F2FS 파일 시스템에 의한 쓰기 증폭량만을 그래프로 나타내면 그림 7과 같다.

6.2 ZenFS 위의 RocksDB

ZenFS는 Western Digital 사에서 개발한 사용자 수준의 파일 시스템으로 구역화된 블록 장치에 특화된 파일 시스템이다[13]. FUSE와 같은 다른 사용자 수준의 파일 시스템과는 다르게 시스템 콜을 통한 파일 입출력을 지원하지 않는 대신 RocksDB의 플러그인으로서 동작하여 RocksDB가 수행하는 파일 입출력을 처리하는 역할을 수행한다. ZenFS에는 아직 쓰레기 수집 기능이 구현되어 있지 않아 한 구역 내의 데이터가 모두 삭제되어야 그 구역을 초기화하고 다시 사용하는 방식으로 동작하고 있다.

아래의 실험은 BIUS로 구역의 크기를 바꿔가며 32GiB 크기의 램 디스크를 만들고 그 위에 ZenFS를 올린 후 YCSB A(읽기 50%, 업데이트 50%) 워크로드를 실행했을 때의 실험 결과를 나타낸 것이다. 초기에 800만 개의 레코드를 로드하였고, 값은 1000B의 크기를 갖도록 하였으며 3200만 회의 업데이트를 수행하도록 하였다. 다만 구역의 크기가 256MiB일 때는 디스크 공간이 부족하다는 오류가 발생하며 YCSB 워크로드가 중단되어 중단된 직후의 측정 결과를 기입하였다.

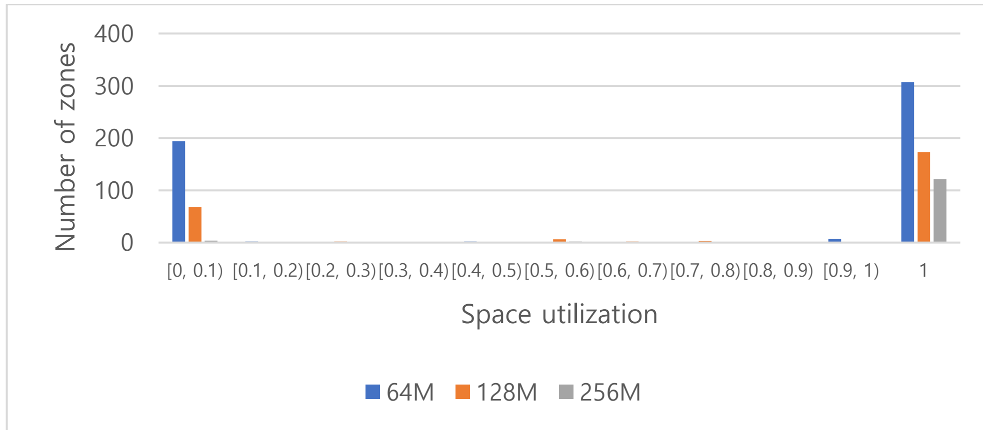


그림 8. 공간 사용량에 따른 구역의 수

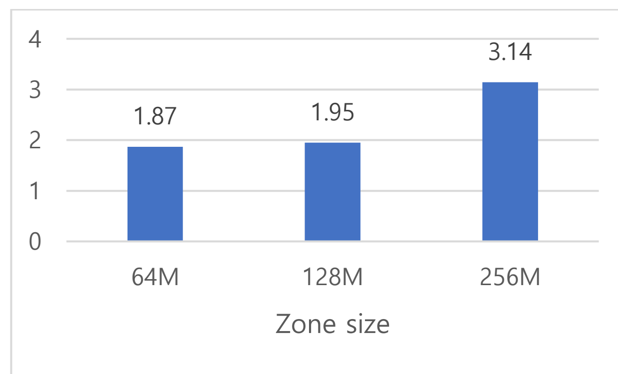


그림 9. 구역의 크기에 따른 공간 증폭량

그림 8은 구역의 크기를 바꿔가며 워크로드를 돌린 후 공간의 사용량에 따른 구역의 수를 나타낸 것이다. 대부분의 구역이 거의 비어 있거나(0%이상~10%미만) 가득 차 있는 것을 확인할 수 있었으며, 그 중간의 완전히 기록되지 않은 구역의 수는 64MiB와 128MiB에서 각각 11개와 15개인 것을 확인할 수 있었다. 또한 그림 9와 같이, 블록 장치에 기록된 총 용량을 유효한 데이터 양으로 나눈 값인 공간의 증폭량(space amplification)은 구역의 크기가 64MiB와 128MiB일 때 2에 가까운 값을 보였으며 256MiB일 때 3을 초과하여, 역시 구역의 크기가 커질수록 오버헤드가 증가하는 경향을 확인할 수 있었다.

제7장 결론

본 연구에서는 기존의 사용자 수준의 블록 장치들이 최신의 빠른 SSD를 에뮬레이션 하거나 구역화된 블록 장치를 구현하기에는 부족함을 확인하고, 이러한 요구 사항을 만족시키는 사용자 수준의 블록 장치 드라이버 BIUS를 제안하였다. BIUS를 사용하면 블록 요청에 대한 간단한 콜 백 함수를 정의해주는 것만으로 커널 수준으로 구현한 장치의 69%~87% 이상의 성능을 발휘할 수 있다. BIUS를 이용하면 향후 ZNS SSD를 위한 파일 시스템이나 키-밸류 스토어의 특성 파악과 최적화가 더 쉽게 가능해질 것으로 기대된다.

참고 문헌

[1] Vangoor, Bharath Kumar Reddy, Vasily Tarasov, and Erez Zadok. "To FUSE or not to FUSE: Performance of user-space file systems." *15th USENIX Conference on File and Storage Technologies (FAST '17).* 2017.

[2] libfuse. https://github.com/libfuse/libfuse/

[3] Device Mapper. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/index.html

[4] Cache and TLB Flushing Under Linux. https://www.kernel.org/doc/html/latest/core-api/cachetlb.html

[5] Zoned Storage Overview. https://zonedstorage.io/introduction/zoned-storage/

[6] Choi, Gunhee, et al. "A New LSM-style Garbage Collection Scheme for ZNS SSDs." *12th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. 2020.

[7] Stavrinos, Theano, et al. "Don't be a blockhead: zoned namespaces make work on conventional SSDs obsolete." *Proceedings of the Workshop on Hot Topics in Operating Systems*. 2021.

[8] Bjørling, Matias. "Zone append: A new way of writing to zoned storage." *Santa Clara, CA, February. USENIX Association*. 2020.

[9] Network Block Device. https://sourceforge.net/projects/nbd/

[10] QEMU Disk Network Block Device Server. https://qemu.readthedocs.io/en/latest/tools/qemu-nbd.html

[11] BUSE: A block device in user space for Linux. https://github.com/acozzette/BUSE

[12] nbdcpp: Network Block Device drivers in userspace C++. https://github.com/dsroche/nbdcpp

[13] ZenFS. https://github.com/westerndigitalcorporation/zenfs

Abstract

BIUS: A Block device driver In User Space

YeonGyu Jeong
Department of Computer Science & Engineering
The Graduate School
Seoul National University

Block device is primary part in operating systems. Nowadays zoned block devices, which have different characteristic than conventional block device, are emerging. On the other hand, demand for virtual block device with various functions, like Device Mapper in Linux, is also arising. For those who want to emulate characteristics of new block devices and for those who want to develop new virtual block devices quickly and safely, BIUS, a block device driver in user space will provide simple but efficient environment for developing block device in user space.

Keywords: Block device, User space, Zoned block device
Student Number: 2019-21955