저작자표시-비영리-변경금지 2.0 대한민국

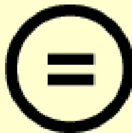
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 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

공학석사학위논문

LSM-tree 기반 키-밸류 스토어에 최적화된 사용자 수준 파일 시스템

User-level File System Optimized for
LSM-tree Based Key-Value Store

2022년 8월

서울대학교 대학원
컴퓨터공학부
정 선 민

공학석사학위논문

LSM-tree 기반 키-밸류 스토어에 최적화된 사용자 수준 파일 시스템

User-level File System Optimized for LSM-tree Based Key-Value Store

2022년 8월

서울대학교 대학원

컴퓨터공학부

정 선 민

LSM-tree 기반 키-밸류 스토어에 최적화된 사용자 수준 파일 시스템

User-level File System Optimized for
LSM-tree Based Key-Value Store

지도교수 김 진 수

이 논문을 공학석사 학위논문으로 제출함

2022년 5월

서울대학교 대학원

컴퓨터공학부

정 선 민

정선민의 공학석사 학위 논문을 인준함

2022년 7월

위 원 장:	이 재 진	(인)
부위원장:	김 진 수	(인)
위 원:	이 재 욱	(인)

초 록

최근 스토리지 기술이 발전함에 따라 I/O 처리 시간이 수 us에 달하는 고성능 저장 장치가 등장하고 있다. 이로 인해 기존에는 무시할 만한 수준에 불과했던 커널 I/O 스택으로 인한 오버헤드의 비중이 커져서 병목 현상의 원인으로 자리잡는 추세이다. 이러한 문제를 해결하기 위해 사용자 수준에서 직접 저장 장치에 접근함으로써 커널의 I/O 스택으로 인한 오버헤드를 제거하기 위한 프레임워크들이 등장하였다. 그러나 프레임워크를 사용하기 위해서는 응용과 I/O 인터페이스를 맞추기 위한 파일 시스템과 저장 장치에 대한 접근을 줄이기 위한 페이지 캐시를 별도로 구현할 필요가 있다. 이를 위해 본 논문에서는 사용자 수준 파일 시스템 및 페이지 캐시를 디자인하고 LSM-tree 기반 키-밸류 스토어를 목표 응용으로 하여 최적화하였다.

주요어: 사용자 수준 파일 시스템, 사용자 수준 페이지 캐시, 키-밸류 스토어
학 번: 2020-20121

차 례

제 1 장 서론 **1**

제 2 장 배경 지식 **2**

2.1 LSM-tree 기반 키-밸류 스토어 2

2.2 SPDK 3

2.3 ARC (Adaptive Replacement Policy) 4

제 3 장 관련 연구 **6**

3.1 사용자 수준 파일 시스템 6

3.2 LSM-tree 기반 키-밸류 스토어 7

제 4 장 설계 및 구현 **8**

4.1 사용자 수준 파일 시스템 (UserFS) 8

4.2 사용자 수준 페이지 캐시 10

4.2.1 LSM-tree 기반 키-밸류 스토어를 위한 페이지 캐시 정책 . . 10

4.2.2 레벨을 고려한 LRU 기반 정책의 한계 11

4.2.3 LSM-tree 기반 키-밸류 스토어를 위한 ARC 기반 정책 . . . 12

제 5 장 실험 결과 및 분석 **16**

5.1 실험 환경 16

5.2 YCSB 벤치마크 17

5.2.1 키-밸류 쌍을 순차적으로 삽입한 경우 17

5.2.2 키-밸류 쌍을 임의의 순서로 삽입한 경우 19

5.3 Mixgraph 벤치마크 21

5.4 캐시의 비율 변화에 따른 성능 차이 22
5.5 일반 NVMe SSD에 대한 성능 차이 24

제 6 장 결론 **25**

ABSTRACT **28**

표 차례

표 4.1 UserFS에서 지원하는 API의 종류 9

표 5.1 실험 환경 16

표 5.2 YCSB 워크로드의 특성 17

표 5.3 Prefix_dist 워크로드 주요 특성 21

그림 차례

그림 2.1 LSM-tree 기반 키-밸류 스토어의 구조 3
그림 2.2 ARC의 구조 4

그림 4.1 UserFS의 구조 8
그림 4.2 LRULevel의 구조 11
그림 4.3 LRULevel이 잘 동작하는 경우와 잘 동작하지 않는 경우의 MRC 13
그림 4.4 LSM-tree 기반 키-밸류 스토어의 특성을 고려한 ARC의 구조 . 14

그림 5.1 키-밸류 쌍을 순차적으로 삽입한 경우의 MRC 및 처리량 . . . 18
그림 5.2 키-밸류 쌍을 임의의 순서로 삽입한 경우의 MRC 및 처리량 . . 20
그림 5.3 Mixgraph 벤치마크의 처리량 22
그림 5.4 블럭 캐시와 페이지 캐시의 비율에 따른 YCSB 워크로드의 처리량 23
그림 5.5 일반 NVMe SSD(Samsung 970 PRO)에서의 YCSB 워크로드의 처리량 24

제 1 장 서론

최근 스토리지 기술의 발전으로 인해 NVMe SSD나 Persistent Memory가 등장하면서 저장 장치에 접근하는 데 걸리는 시간이 수 us 단위로 매우 낮아졌다. 이로 인해 기존에는 무시할 만한 수준에 불과했던 커널 I/O 스택으로 인한 오버헤드가 I/O 처리 시간의 35% 정도를 차지할 정도로 커져서 병목 현상의 원인이 되고 있다[7]. 커널을 거치지 않고 사용자 수준에서 직접 저장 장치에 접근하게 되면 이러한 오버헤드를 제거할 수 있다. 이를 위해 NVMeDirect[6]나 SPDK[11]와 같이 사용자 수준에서 직접 저장 장치에 접근할 수 있도록 하는 프레임워크가 등장하였다.

그러나 이러한 프레임워크를 응용에 직접 적용하는 것은 두 가지의 이유로 적절하지 않다. 첫 번째는 인터페이스의 불일치이다. 대부분의 응용이 파일 인터페이스를 사용하는 반면, SPDK와 같은 프레임워크들은 블럭 인터페이스를 제공하고 있다. 따라서 응용과 프레임워크 사이에서 파일 인터페이스와 블럭 인터페이스를 연결하는 사용자 수준 파일 시스템이 필요하다.

두 번째는 페이지 캐시의 부재이다. 커널을 경유하지 않으면 커널에서 제공하는 페이지 캐시를 사용하지 못하므로 모든 I/O 요청에 대해서 저장 장치에 접근해야 한다. 이로 인해 저장 장치에 접근하는 시간 자체는 줄어들더라도 I/O 요청들을 처리하는 데 걸리는 전체적인 시간은 오히려 크게 늘어날 수 있다. 대부분의 접근 요청이 소수의 데이터에 대해서 이루어진다는 점을 고려하면[10], 사용자 수준 페이지 캐시를 활용함으로써 저장 장치에 접근하는 횟수를 크게 줄일 수 있다.

커널에서는 어떤 워크로드를 수행할 지 모르기 때문에 모든 경우를 고려해야 하지만, 파일 시스템과 페이지 캐시를 사용자 수준에서 구현하게 되면 목표 응용의 워크로드에 대해서 최적화할 수 있는 장점을 가진다. 본 논문에서는 LSM-tree 기반 키-밸류 스토어를 목표 응용으로 하는 사용자 수준 파일 시스템 및 페이지 캐시를 디자인하였다.

제 2 장 배경 지식

2.1 LSM-tree 기반 키-밸류 스토어

LSM-tree(Log Structured Merge Tree)는 임의의 위치에 대한 쓰기 연산들을 모아서 한꺼번에 크고 순차적인 쓰기를 하는 특징을 가지고 있는 인덱스 구조이다. LSM-tree 기반 키-밸류 스토어의 구성 요소는 크게 세 가지가 있다. 첫 번째는 메모리에 있는 메모리 테이블(memtable)로, 크고 순차적인 쓰기를 하기 전에 수정 사항들을 모아두는 역할을 한다. 두 번째는 WAL(Write Ahead Log)로, 메모리 테이블에만 기록되고 저장 장치에는 반영되지 않은 수정 사항들에 대해서 크래시가 발생하더라도 내용을 복구할 수 있도록 저장 장치에 기록하는 로그 파일이다. 마지막은 SST(Sorted String Table)로, 키-밸류 쌍들을 정렬된 형태로 저장하는 파일이다. 그림 2.1에서와 같이 SST들은 여러 개의 레벨로 구성되는데, 레벨이 커질 수록 SST의 수가 10배 정도씩 증가하는 형태이다. 각 레벨의 SST들은 서로 겹치지 않는 범위의 키-밸류 쌍을 가지고 있지만, 예외적으로 레벨 0의 경우에는 키의 범위가 겹칠 수 있다.

LSM-tree 기반 키-밸류 스토어에서 키-밸류 쌍을 저장하는 과정은 그림 2.1과 같다. 우선 키-밸류 쌍을 WAL에 적은 뒤 메모리 테이블에 기록한다. 메모리 테이블이 가득 차면 불변의 메모리 테이블(immutable memtable)로 바뀐다. 불변의 메모리 테이블은 SST로 변환되어 레벨 0에 기록되는데, 이 과정을 플러시(flush)라고 한다. 특정 레벨 n의 SST 수가 정해진 한계를 넘게 되면 키의 범위가 겹치는 레벨 n+1의 SST와 합쳐지게 되는데, 이 과정을 컴팩션(compaction)이라고 한다.

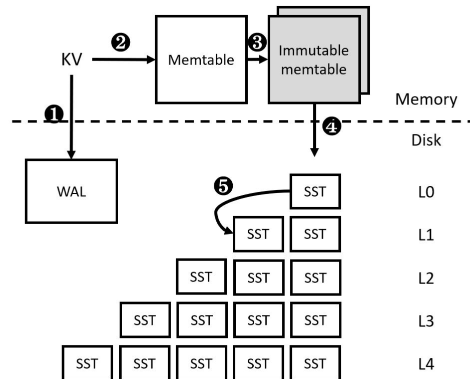


그림 2.1: LSM-tree 기반 키-밸류 스토어의 구조

2.2 SPDK

SPDK는 커널을 거치지 않고 사용자 수준에서 직접 저장 장치에 접근할 수 있도록 함으로써 커널 I/O 스택 오버헤드를 없애고 고성능 저장 장치의 이점을 최대한 활용할 수 있도록 하는 라이브러리이다. SPDK는 인터럽트 대신 폴링 기반의 비동기 I/O를 제공함으로써 컨택스트 스위치로 인한 오버헤드를 제거하였다. 또, I/O를 처리하는 과정에서 락을 사용하지 않기 때문에 확장성이 뛰어나다는 장점을 가진다.

SPDK는 BlobFS라는 사용자 수준 파일 시스템을 제공하고 있지만, 아직까지는 간단한 수준의 파일 I/O만 지원하고 있다. POSIX 인터페이스를 제공하지 않으며, 파일을 순차적으로 쓰는 것만 가능하고, 디렉토리 구조를 지원하지 않는 등 제한 사항이 많다.

BlobFS도 저장 장치에 대한 접근 횟수를 줄이기 위해 페이지 캐시를 활용한다. 그러나 페이지 캐시를 관리하는 방법이 커널에서의 것과 큰 차이를 보인다. 4KB가 아닌 256KB의 큰 단위로 페이지를 관리하고, 임의의 위치에 대한 읽기 요청에 대해서는 페이지를 캐시에 넣지 않는다. 즉, 쓰기 요청이 들어왔을 때나 파일에 대한 순차적인 읽기 패턴이 인식되었을 때 미리 읽어두기 위한 용도로만 페이지를 캐시

에 넣는다. 페이지가 가득 찬 경우에는 페이지 캐시를 사용중인 파일들의 리스트를 순회하며 깨끗한 페이지를 우선적으로 쫓아낸다.

2.3 ARC (Adaptive Replacement Policy)

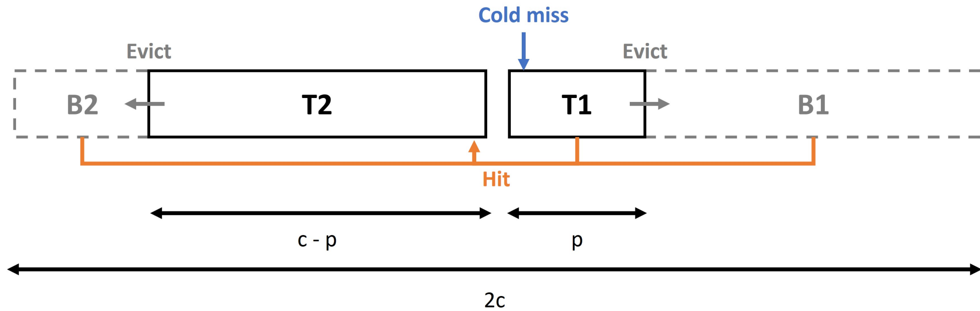


그림 2.2: ARC의 구조

ARC는 워크로드의 변화에 반응하여 유동적으로 여러 캐시 리스트의 크기를 조절하는 캐시 관리 정책이다. ARC의 구조는 그림 2.2과 같다. ARC는 실제 데이터를 가지고 있는 데이터 캐시(T1, T2)와 페이지에 대한 메타데이터만 가지고 있는 메타데이터 캐시(B1, B2)를 가진다. 그리고 각각은 한 번만 접근된 콜드 페이지(T1, B1)와 두 번 이상 접근된 핫 페이지(T2, B2)로도 구분할 수 있다. 각각의 캐시 리스트는 LRU 정책으로 관리되며, T1의 길이는 p라는 변수의 크기 조절로 정해진다.

메타데이터 캐시는 막 쫓겨난 페이지에 대한 접근 기록을 제공한다. 따라서, 어떤 페이지가 메타데이터 캐시에서 히트가 발생하면 해당 페이지는 핫 페이지로 간주되고 바로 T2에 들어간다. 메타데이터 캐시의 또 다른 용도는 데이터 캐시의 크기를 조절하는 정보를 제공하는 것이다. 어떤 페이지가 B1에서 메타데이터 캐시 히트가 발생한 경우, T1의 길이가 길었다면 해당 페이지에서 데이터 캐시 히트가 발생했을 것이기 때문에 T1의 크기를 늘린다. 반대로 B2에서 히트가 발생한 경우에는 T2의

길이를 늘리게 된다.

ARC는 접근 빈도를 고려하는 다른 캐시 정책과는 달리 시간 복잡도가 상수이고 튜닝 파라미터 없이 리스트의 길이를 관리하는 변수가 자동으로 조절되며 워크로드의 변화에 잘 반응한다는 장점을 가진다.

제 3 장 관련 연구

3.1 사용자 수준 파일 시스템

SPDK를 활용한 사용자 수준 파일 시스템으로는 EvFS[12], uFS[8]가 있다. EvFS는 POSIX 인터페이스를 제공함으로써 기존의 응용을 수정할 필요 없이 고성능 저장 장치의 이점을 잘 활용할 수 있도록 한 사용자 수준 파일 시스템이다. SPDK에서 제공하는 블럭 할당자인 Blobstore를 기반으로 만들어졌으며, SPDK에서 제공하는 이벤트 기반 구조를 활용하여 파일 I/O를 비동기적으로 처리할 수 있도록 하였다.

uFS는 프로세스의 형태로 파일 시스템 서비스를 제공한다. uFS는 동적 라이브러리 형태로 응용에 연결되는 uLib와 프로세스 형태의 파일 시스템인 uServer로 구성되며, 파일 시스템 서비스는 uLib와 uServer 간의 IPC를 통해서 이루어진다. uServer는 SPDK의 사용자 수준 NVMe 드라이버를 사용한다. I/O를 처리하는 쓰레드들은 각자의 I/O queue를 가지고 있으며, 락을 사용하지 않기 위해 I/O 쓰레드들이 자원을 공유하지 않는 구조로 디자인되었다. 또한 I/O의 양에 따라 I/O 쓰레드의 수를 유동적으로 조절함으로써 불필요한 자원 소모를 줄였다.

EvFS와 uFS 모두 사용자 수준 페이지 캐시를 가지고 있지만 최적화에 대해서는 별다른 관심을 두지 않았다. EvFS는 페이지 캐시를 통해 읽기와 쓰기를 병합한다는 언급은 하고 있지만 구체적인 페이지 캐시 관리 기법에 대해서는 언급하지 않았다. uFS는 각 I/O 쓰레드가 페이지 캐시를 똑같이 나누어 가지며, I/O 쓰레드는 LRU로 각자의 페이지 캐시를 관리한다. 한 번에 하나의 프로세스만이 페이지에 접근할 수 있도록 하였으며, 읽기 용도로만 페이지 캐시를 사용할 수 있다.

3.2 LSM-tree 기반 키-밸류 스토어

SPDK를 활용한 LSM-tree 기반 키-밸류 스토어로는 SpanDB[3]와 TridentKV[9]가 있다. SpanDB는 성능이 다른 저장 장치들을 같이 사용할 때 고성능 저장 장치의 성능을 잘 활용하고자 한 LSM-tree 기반 키-밸류 스토어이다. 고성능 저장 장치에 WAL과 상위 레벨의 SST를 저장하고, SPDK의 폴링 기반 비동기 I/O를 통해 성능을 크게 향상시켰다. 고성능 저장 장치에 SST와 WAL 파일을 읽고 쓰기 위해 매우 간단한 수준의 사용자 수준 파일 시스템 및 페이지 캐시를 구현하였으나, 디자인이나 최적화에는 초점을 두지 않았다.

TridentKV는 읽기에 최적화하는 것을 목표로 한 LSM-tree 기반 키-밸류 스토어로, 읽기의 성능을 끌어올리기 위한 방법 중 하나로 폴링 기반의 비동기적인 읽기를 지원하기 위해 SPDK를 활용하였다. 그러나 사용자 수준 파일 시스템이나 페이지 캐시에 대해서는 별다른 언급을 하지 않았다.

제 4 장 설계 및 구현

4.1 사용자 수준 파일 시스템 (UserFS)

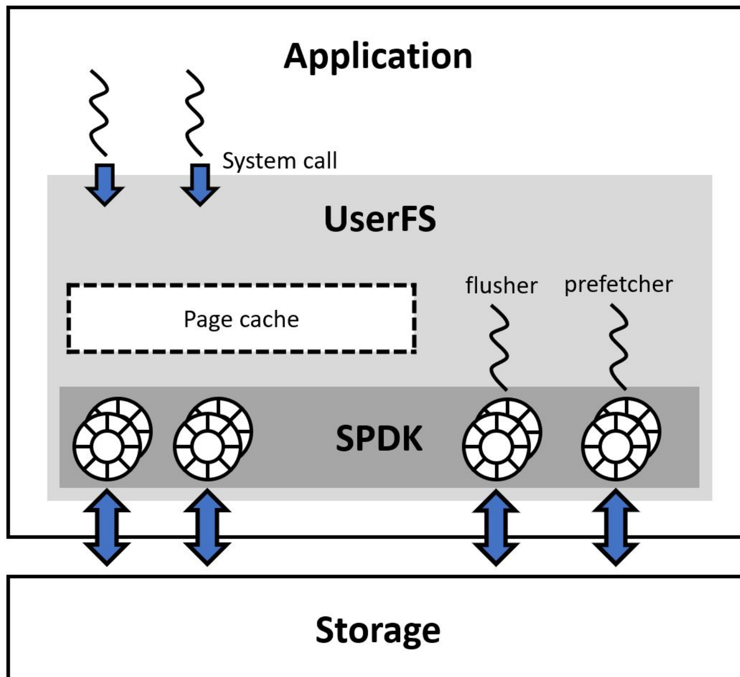


그림 4.1: UserFS의 구조

사용자 수준 파일 시스템인 UserFS의 대략적인 구조는 그림 4.1과 같다. UserFS는 LIBC와 같은 POSIX 파일 API를 사용하는 동적 라이브러리 형태로 제공되며, 응용이 사용하는 시스템 콜을 가로채서 커널 대신 직접 처리함으로써 시스템 콜로 인한 컨택스트 스위치 오버헤드를 제거한다. 따라서 기존의 응용을 수정할 필요없이 라이브러리를 로드하기만 하면 바로 사용할 수 있다는 장점을 가진다. 현재

UserFS에서 지원하는 API의 종류는 표 4.1과 같다.

표 4.1: UserFS에서 지원하는 API의 종류

Type	APIs
File	open, read, write, pread, pwrite, lseek, close, rename, fallocate, stat, fstat, statfs, fstatfs, truncate, ftruncate, unlink, access, fsync, fdatasync, fadvise
Directory	mkdir, rmdir, getdents

UserFS는 SPDK에서 제공하는 사용자 수준 NVMe 드라이버 라이브러리를 활용해서 폴링 기반의 비동기 I/O를 통해 저장 장치에 직접 접근한다. I/O를 처리하는 과정에서 락을 사용하지 않는 SPDK의 구조를 잘 활용하기 위해 응용 쓰레드와 UserFS 내부 쓰레드는 각자 I/O queue를 할당받고 I/O 요청을 직접 처리한다.

저장 장치에 대한 접근 횟수를 줄이기 위해 UserFS는 페이지 캐시를 활용하며, 커널에서와 마찬가지로 4KB 단위로 페이지들을 관리한다. 이와 더불어 I/O를 백그라운드에서 처리함으로써 I/O 요청을 처리하는 데 걸리는 시간을 감추기 위해서 파일 시스템 내부에 페이지 캐시를 관리하기 위한 추가적인 쓰레드들을 가진다. Flusher 쓰레드는 주기적으로 새로 쓰인 페이지들을 저장 장치에 기록하며 prefetcher 쓰레드는 순차적인 읽기를 진행하는 파일을 미리 페이지 캐시에 읽어들인다.

UserFS는 커널의 ext4 파일 시스템과 동일한 레이아웃을 사용하며 ext4와 호환 가능하다. 따라서 ext4로 마운트하여 커널 파일 시스템을 통해서도 응용을 실행하거나 파일에 접근할 수 있으며 ext4 파일 시스템을 관리하기 위해 제공되는 e2fsck와 같은 프로그램들을 사용할 수 있다는 장점을 가진다.

4.2 사용자 수준 페이지 캐시

페이지 캐시를 통해 저장 장치에 접근하는 횟수를 줄이고자 할 때, 어떤 페이지들을 가지고 있을지에 대한 정책에 따라서 성능이 크게 좌우된다. 모든 워크로드에 대해서 준수한 성능을 보장할 필요가 있는 커널과는 달리, 사용자 수준에서는 목표 응용의 워크로드에 초점을 맞춰서 페이지 캐시 정책을 디자인할 수 있다는 장점이 있다.

4.2.1 LSM-tree 기반 키-밸류 스토어를 위한 페이지 캐시 정책

페이지 캐시 정책의 목표는 캐시 미스율을 낮추는 것이다. LSM-tree 기반 키-밸류 스토어에서는 임의의 위치에 대한 쓰기가 발생하지 않고 항상 순차적인 쓰기만이 발생한다. 따라서 쓰기에 대해서는 항상 콜드 미스만이 발생하게 되므로, 읽기 미스율을 낮추는 것에 초점을 맞출 필요가 있다.

LSM-tree 기반 키-밸류 스토어의 레벨 구조 및 컴팩션을 고려하여 페이지 캐시 정책을 디자인하면 보편적인 기존의 정책보다 읽기 미스율을 낮출 수 있다. 레벨은 LSM-tree 구조와 관련된 다른 연구에서도 주목한 바 있다. 오래된 데이터가 컴팩션 과정에서 하위 레벨로 내려가는 반면, 상위 레벨의 SST는 최근에 새로 쓰이거나 값이 수정된 데이터를 담고 있다. 따라서 상위 레벨의 SST들을 메모리에 고정하거나[5], 성능이 더 좋은 저장 장치에 기록하는[3] 것과 같이 상위 레벨의 SST를 우선적으로 캐싱하는 것을 고려할 수 있다.

기존의 SST 파일들을 읽어들여서 새로운 SST 파일을 쓰는 과정인 컴팩션은 순차적인 읽기와 쓰기를 대량으로 발생시킨다. 따라서 컴팩션이 발생하는 시점에 기존의 SST 파일들을 미리 읽어들일 필요가 있다. 컴팩션의 결과로 쓰인 SST들은 생성된 즉시 사용되는 것이 아니라 컴팩션이 끝나고 기존의 SST 파일들을 삭제한 뒤에 사용하게 된다. 따라서 파일이 사용되지 않는 기간에 해당 파일의 페이지들을 콜드 페이지로 여기고 페이지 캐시에서 내보내는 것은 적절하지 않다.

4.2.2 레벨을 고려한 LRU 기반 정책의 한계

LSM-tree 기반 키-밸류 스토어의 I/O 특성을 고려하여 상위 레벨을 우선적으로 캐싱하는 아이디어를 널리 사용되는 캐시 정책인 LRU(Least Recently Used)에 적용하면 그림 4.2의 LRULevel과 같다. 레벨에 따라 별도의 LRU 리스트를 관리하고, 캐시가 가득 차면 하위 레벨의 LRU 리스트에서부터 페이지를 내쫓는 방식이다.

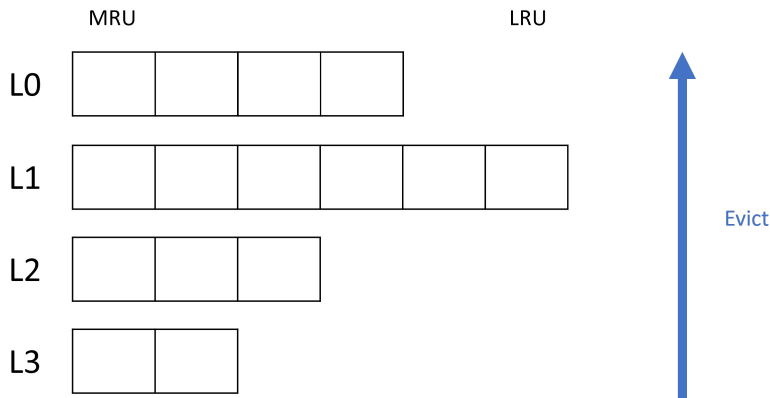


그림 4.2: LRULevel의 구조

이러한 정책이 효과를 보기 위해서는 우선적으로 캐싱하는 상위 레벨에 핫 데이터가 있어서 대부분의 I/O 요청을 처리할 수 있어야 한다. 그림 4.3a는 이러한 경우의 읽기 미스율 그래프를 나타내고 있는데, LRULevel은 LRU와 OPT(optimal) 사이에서 중간 정도의 읽기 미스율을 보여주고 있다. 그러나 우선순위가 밀리는 아랫 레벨에 핫 데이터가 분포해 있는 경우에는 그림 4.3b와 같이 LRU보다 더 높은 읽기 미스율을 보이게 된다. 따라서 LRULevel 정책을 사용하기 위해서는 아랫 레벨에 핫 데이터가 분포한 경우에 대해서도 효과적으로 동작할 수 있도록 레벨 별 캐시 리스트의 길이를 조절하는 추가적인 메커니즘이 필요하다.

그 전에 우선 레벨 별 캐시 리스트의 길이를 이상적으로 조절했을 때 읽기 미스율을 LRU에 비해 크게 줄일 수 있는지 확인해 볼 필요가 있다. 가장 이상적인 레벨

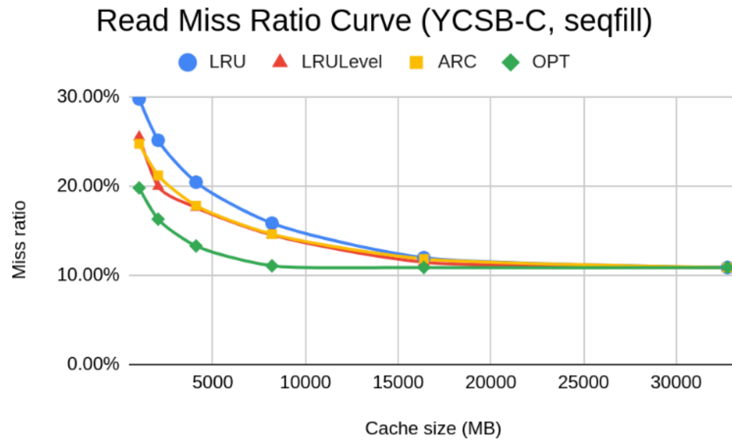
별 캐시 리스트의 길이는 미래의 접근 패턴을 기반으로 캐시를 관리하는 OPT에서의 레벨 별 캐시 리스트의 길이이다. OPT의 레벨 별 리스트의 길이를 LRULevel에 적용했을 때의 읽기 미스율 그래프는 그림 4.3b의 LRULevelOPT와 같다. LRU에 비해 읽기 미스율이 소폭 낮아지긴 했지만, LRULevel이 잘 동작하는 그림 4.3a에서처럼 LRU와 OPT의 중간 정도까지는 개선되지 않았다. 이를 통해 LRU 기반 페이지 캐시 관리 정책은 레벨 정보를 고려한다 해도 읽기 미스율을 개선하는 데 있어서 한계가 있음을 알 수 있다.

그에 비해 ARC의 경우에는 그림 4.3a와 그림 4.3b에서 살펴볼 수 있듯이 핫 데이터가 어느 레벨에 분포하는지와 관계 없이 항상 LRU와 OPT의 중간 정도의 읽기 미스율을 보이는 것을 알 수 있다. 따라서 LRU보다는 ARC를 기반으로 LSM-tree 기반 키-밸류 스토어에 최적화된 페이지 캐시 정책을 디자인하는 것이 적절하다.

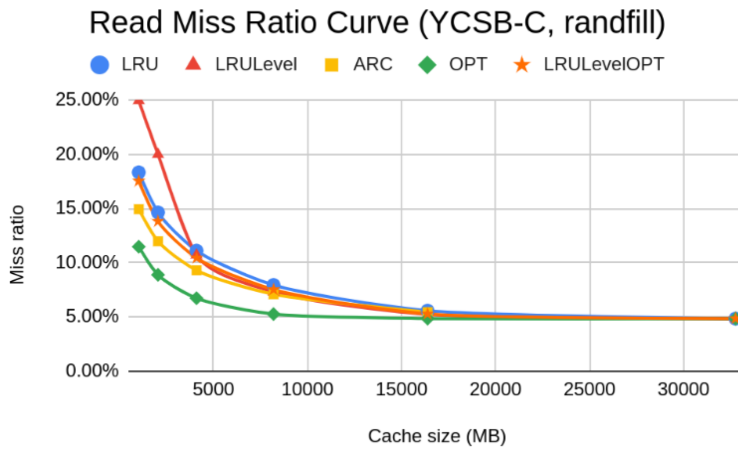
4.2.3 LSM-tree 기반 키-밸류 스토어를 위한 ARC 기반 정책

ARC를 그대로 LSM-tree 기반 키-밸류 스토어를 목표 응용으로 하는 페이지 캐시 관리 정책으로 사용하기에는 컴팩션과 WAL로 인해 적절하지 않다. 컴팩션 과정 동안 대상이 되는 SST 파일들을 작은 단위로 순차적으로 읽게 되는데, 요청이 올 때 저장 장치에서 읽어들이는 것은 비효율적이다. 따라서 컴팩션 대상이 되는 SST 파일들을 큰 단위로 미리 읽어들일 수 있도록 프리패치(prefetch)를 구현할 필요가 있다.

컴팩션의 결과 SST 파일이 쓰이는 즉시 바로 사용되는 것이 아니라 컴팩션이 다 끝난 이후에 사용된다는 점도 ARC와는 상충된다. 컴팩션의 결과는 T1에 쓰이게 되고, 컴팩션이 끝날 때까지 사용되지 않는 시간 동안 B1으로 쫓겨나게 된다. 컴팩션이 끝나고 결과 파일이 쓰이기 시작하면 B1에서 캐시 히트가 발생하게 되고, 이로 인해 T1의 길이가 늘어나게 된다. 읽기 미스율을 낮추기 위해서는 사용 빈도가 있는 T2의 페이지들을 많이 들고 있는 것이 효율적이므로, 컴팩션이 다 끝난 이후에 T1의 길이를 늘리는 것은 적절하지 않다.



(a) LRULevel이 잘 동작하는 경우



(b) LRULevel이 잘 동작하지 않는 경우

그림 4.3: LRULevel이 잘 동작하는 경우와 잘 동작하지 않는 경우의 MRC

순차적으로 쓰고 읽을 일이 없다시피 한 WAL의 경우 T1에 쓰였다가 빠르게 쫓겨나는 것이 적절해 보인다. 하지만, 수정된 페이지라 쫓아내기 전에 저장 장치에 쓸 필요가 있다는 점이 문제가 된다. WAL을 저장 장치에 쓸 때까지 기다렸다가 페이지를 얻는 것도, 깨끗한 페이지를 찾을 때까지 리스트를 순회하는 것도 큰 오버헤드로 작용하게 되므로 부적절하다.

이러한 점들을 고려하여 ARC를 개선한 구조는 그림 4.4과 같다.

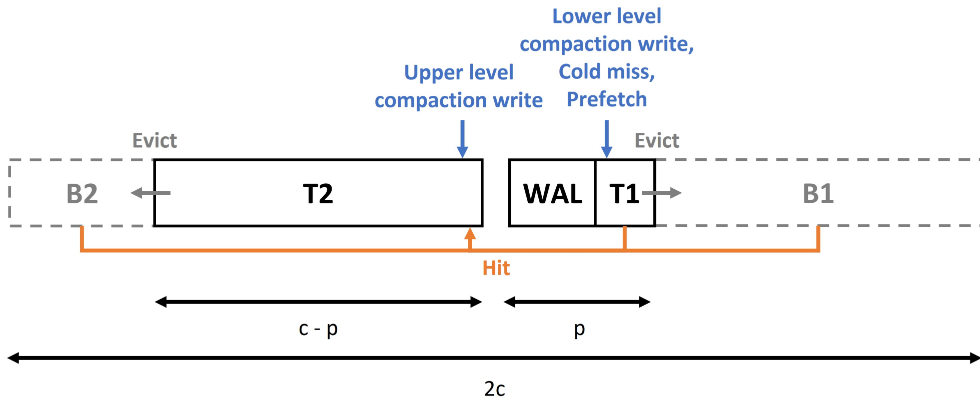


그림 4.4: LSM-tree 기반 키-밸류 스토어의 특성을 고려한 ARC의 구조

우선, 컴팩션을 빠르게 처리할 수 있도록 프리패치를 구현하였다. 컴팩션 쓰레드가 대상 파일을 순차적으로 읽는 패턴을 파악하여 1MB 단위로 페이지를 미리 읽어들이고, 그 중 절반을 사용하면 다음 부분을 읽어들이도록 하였다. 프리패치된 페이지는 T1에 쓰이며, 컴팩션 쓰레드에 의해서는 캐시 히트가 나더라도 T2로 옮겨지지 않는다. 따라서 컴팩션 대상이 되는 파일은 읽히고 나서 빠르게 쫓겨나게 되는데, 컴팩션이 끝나고 나서는 제거되어 쓰이지 않게 될 파일인 만큼 적절한 조치라고 할 수 있다. 컴팩션 대상 파일이지만 이미 캐시에 있던 페이지들은 기존에 사용되던 페이지이므로 별다른 조치를 취하지 않았다.

다음으로는 컴팩션이 끝나고 사용될 확률이 높은 상위 레벨의 결과 파일들을 T1

이 아닌 T2에 쓰도록 하여 컴팩션이 끝날 때까지 쫓겨나지 않을 수 있도록 하였다. 상위 레벨과 하위 레벨을 구별하는 기준은 전체 레벨 수의 절반으로 정하였다. 또한 별도의 WAL 리스트를 두어 T1에서 페이지를 쫓아내야 할 때 수정된 페이지로 인해 느리게 처리되지 않도록 보장하였다. WAL 리스트는 flush 쓰레드에 의해 주기적으로 저장 장치에 쓰이고 페이지를 반환하도록 하였다.

T1에서 WAL을 분리함으로써 T1에 속한 페이지는 그림 4.4에서와 같이 콜드 미스인 페이지와 아랫쪽 레벨의 컴팩션 결과 파일들, 프리패치된 컴팩션 대상 페이지들로 좁혀지게 된다. 그중에서 콜드 미스인 페이지를 제외한 두 가지 종류가 LRU에 위치하면 쫓아낼 페이지를 선정하기가 쉽지 않다. 컴팩션 결과 파일들의 경우에는 쫓아내기 전에 저장 장치에 쓸 필요가 있어서 번거로울 뿐 아니라 수 MB 단위로 한꺼번에 캐싱된다. 프리패치된 파일들도 곧 쓰일 예정인 페이지들을 미리 읽어들인 것이기 때문에 쫓아내는 것이 비효율적이고 1MB 단위로 한꺼번에 캐싱된다. 따라서, 두 종류의 페이지가 LRU에 있으면 쫓아낼 페이지를 정하는 것이 상당한 오버헤드로 작용한다. 이로 인한 성능 저하를 피하기 위해, T1의 길이가 p보다 커서 T1에서 페이지를 쫓아내야 하더라도 컴팩션 결과 파일이거나 프리패치된 파일일 경우 T2에서 페이지를 쫓아내도록 ARC에서의 리스트 길이 관리 조건을 완화하였다.

제 5 장 실험 결과 및 분석

5.1 실험 환경

실험은 널리 사용되는 LSM-tree 기반 키-밸류 스토어인 RocksDB[1]를 목표 응용으로 하여 진행하였으며, 시스템의 사양은 표 5.1와 같다. 커널의 ext4 파일 시스템과 SPDK에서 제공하는 BlobFS, UserFS의 세 가지 페이지 캐시 관리 정책인 LRU, LRULevel, ARC에 대해서 성능을 비교하였다. 페이지 캐시의 양을 커널과 사용자 수준에서 비슷한 수준으로 맞추기 위해 cgroup을 통해 사용 가능한 메모리의 양을 제한했으며, 모든 실험은 세 번씩 수행한 뒤 중간값으로 표기했다. RocksDB의 옵션들은 기존의 설정을 따르되, RocksDB 내부에서 사용하는 블럭 캐시의 양은 매뉴얼에 따라 가용 메모리의 1/3 가량이라고 할 수 있는 페이지 캐시 크기의 1/2로 설정하였다.

실험을 진행한 워크로드는 총 두 가지이다. 첫 번째로는 널리 사용되는 키-밸류 워크로드인 YCSB 벤치마크[4]를 사용하였다. 다음으로는 페이스북에서 실제 워크로드를 분석한 결과를 바탕으로 YCSB보다 실제 워크로드와 비슷하도록 디자인한 mixgraph 벤치마크[2]를 사용하였다.

표 5.1: 실험 환경

Processor	Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz
Memory	256GB
OS	Ubuntu 20.04.2 LTS (kernel v5.4.0)
Storage	Optane P4800X SSD, Samsung 970 Pro (5.5)

5.2 YCSB 벤치마크

YCSB 벤치마크는 총 여섯 개의 워크로드를 제공하고 있다. 실험에서는 그 중에서 세 가지를 사용하였는데, 사용한 벤치마크의 종류 및 특성은 표 5.2와 같다. 키는 8B의 숫자를 사용하고 밸류는 1KB로 고정하였으며, 50M개의 키-밸류 쌍을 삽입한 뒤 각 쓰레드마다 30M개의 요청을 처리하도록 하였다. Zipfian 분포를 사용하되, 작은 값의 키일수록 요청이 빈번히 발생하도록 하였다. 페이지 캐시는 2GB로, RocksDB 블럭 캐시는 1GB로 설정하였다. 키-밸류 쌍을 순차적으로 삽입하여 상위 레벨에 핫 데이터가 몰려 있는 경우와 임의의 순서로 삽입하여 핫 데이터가 여러 레벨에 흩어져 있는 경우에 대해서 각각 실험을 진행하였다.

표 5.2: YCSB 워크로드의 특성

Workload	**Properties**
YCSB-A	Write-intensive (Get:Put = 50:50)
YCSB-B	Read-intensive (Get:Put = 95:5)
YCSB-C	Read-only (Get:Put = 100:0)

5.2.1 키-밸류 쌍을 순차적으로 삽입한 경우

키-밸류 쌍을 순차적으로 삽입한 경우에 대해서 실험을 진행한 결과는 그림 5.1과 같다. 왼쪽은 캐시 시뮬레이터 상에서 UserFS의 세 가지 페이지 캐시 관리 정책에 대한 읽기 MRC(Miss Ratio Curve)를 나타낸 그래프이고, 오른쪽은 쓰레드의 수를 늘려 가면서 실험을 진행하였을 때의 처리량을 나타낸 그래프이다.

키-밸류를 순차적으로 삽입하게 되면 컴팩션이 일어나지 않아 모든 SST 파일들이 겹치지 않는 범위의 키-밸류 쌍을 가지고 있게 된다. 또한 상위 레벨에 핫 데이터가 몰려 있기 때문에 LRULevel 정책이 잘 동작하는 경우이다. 따라서 MRC

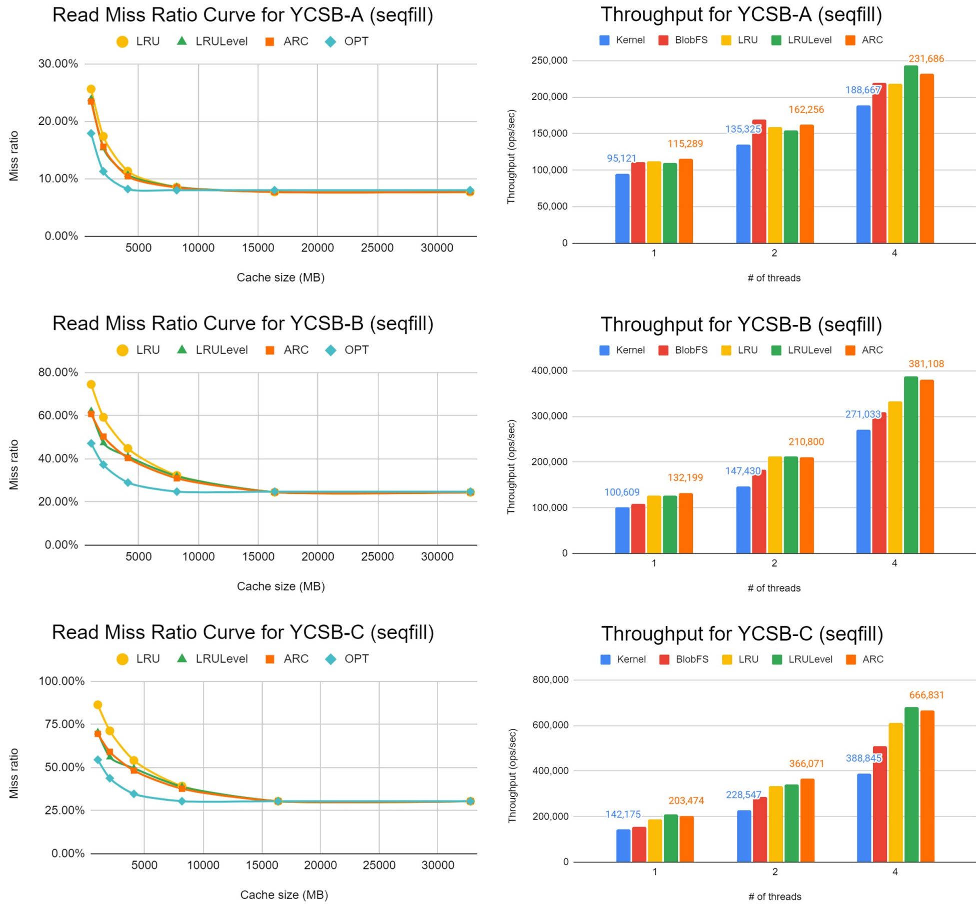


그림 5.1: 키-밸류 쌍을 순차적으로 삽입한 경우의 MRC 및 처리량

와 처리량에서 모두 LRULevel이 대체적으로 ARC보다 비슷하거나 뛰어난 성능을 보인다. 특히, 읽기 미스율의 측면에서는 두 가지 정책이 비슷한 성능을 보이지만 처리량의 경우 쓰레드가 많아지면 LRULevel보다 ARC가 성능이 5% 가량 떨어지게 된다. 이는 보다 복잡한 페이지 캐시 관리 정책으로 인한 오버헤드가 성능에 영향을 미친 것으로 보인다.

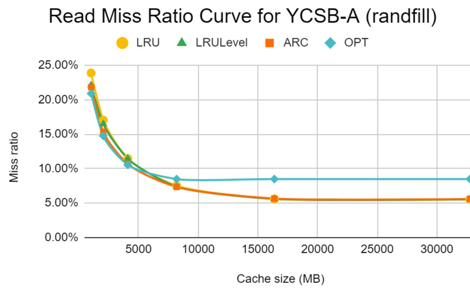
커널 파일 시스템의 경우 I/O 스택의 오버헤드로 인해 모든 워크로드에서 UserFS보다 낮은 성능을 보인다. 순차적인 쓰기와 프리패치에 대해서만 페이지 캐시를 사용하는 BlobFS의 경우, 쓰기의 비중이 큰 YCSB-A에서는 UserFS보다 성능이 더 좋게 나오기도 한다. 하지만 YCSB-B와 YCSB-C에서는 UserFS보다 성능이 30%까지 떨어지게 된다. 실제 워크로드에서 읽기의 비중이 크다는 점을 고려하면[2][10], 임의의 읽기에 대한 페이지 캐시의 역할을 간과할 수 없다.

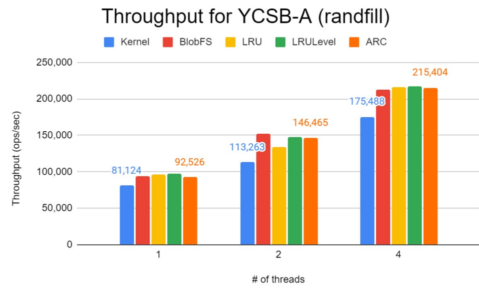
5.2.2 키-밸류 쌍을 임의의 순서로 삽입한 경우

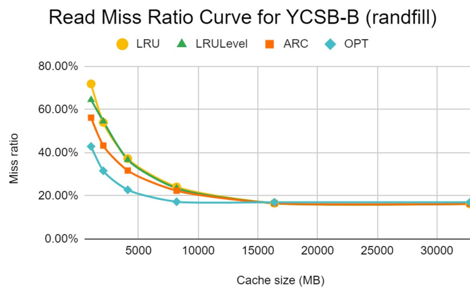
키-밸류 쌍을 임의의 순서로 삽입한 경우에 대해서 실험을 진행한 결과는 그림 5.2과 같다. 왼쪽은 캐시 시뮬레이터 상에서 UserFS의 세 가지 페이지 캐시 관리 정책에 대한 읽기 MRC(Miss Ratio Curve)를 나타낸 그래프이고, 오른쪽은 쓰레드의 수를 늘려 가면서 실험을 진행하였을 때의 처리량을 나타낸 그래프이다.

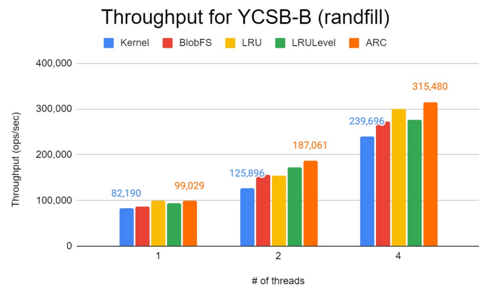
키-밸류를 임의의 순서로 삽입하게 되면 핫 데이터인 작은 값의 키들이 여러 레벨에 분산되어 위치하게 된다. 하위 레벨로 갈수록 더 많은 SST를 가진다는 점을 고려하면, 하위 레벨에 핫 데이터가 많이 위치하기 때문에 LRULevel 정책이 잘 동작하지 않는 경우이다. 이는 읽기 미스율 그래프에서 확인할 수 있는데, 세 가지 워크로드에서 모두 LRU와 비슷한 수준의 높은 읽기 미스율을 보이는 것을 알 수 있다. 처리량의 측면에서도 쓰기의 비중이 커서 핫 데이터가 금방 상위 레벨에 쓰이게 되는 YCSB-A를 제외하면 대체적으로 LRU와 ARC보다 낮은 성능을 보인다.

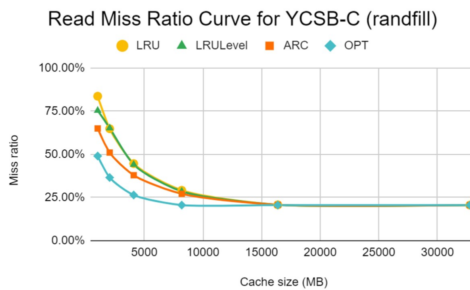
ARC의 경우 키-밸류 쌍을 순차적으로 삽입했을 때와 같이 읽기 미스율이 LRU와 OPT의 중간 정도에 해당한다. YCSB-A의 경우 다른 캐시 관리 정책들과 비슷한











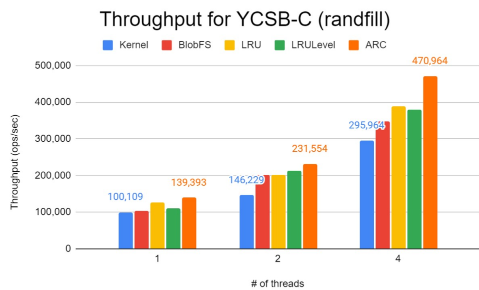


그림 5.2: 키-밸류 쌍을 임의의 순서로 삽입한 경우의 MRC 및 처리량

읽기 미스율을 보이지만 처리량의 측면에서는 캐시 관리 오버헤드로 인해 성능이 1~5% 정도 낮게 나온다. YCSB-B와 YCSB-C에서는 LRU나 LRULevel보다 최대 20% 정도까지 높은 성능을 보인다. 커널 파일 시스템 및 BlobFS와 비교하면 키-밸류 쌍을 순차적으로 삽입했을 때와 비슷한 성능 차이를 보인다.

5.3 Mixgraph 벤치마크

페이스북에서 제공하는 Mixgraph 벤치마크[2]는 실제 키-밸류 스토어 워크로드들에 대한 분석을 바탕으로 쿼리의 구성, 키나 밸류의 크기, 핫 데이터의 지역성 등을 고려함으로써 YCSB 벤치마크보다 실제 워크로드에 가깝도록 만들어졌다. 워크로드는 논문에서 예시로 제공하는 prefix_dist를 따르되, 시간에 따른 qps(query per second)의 변화를 없애고 데이터베이스의 크기를 키우기 위해 키-밸류 쌍의 수와 블럭 캐시 사이즈를 늘렸다. Prefix_dist 워크로드의 주요 특성은 표 5.3와 같으며, 임의의 순서로 키-밸류 쌍을 삽입한 뒤 워크로드를 실행하였다.

표 5.3: Prefix_dist 워크로드 주요 특성

Key size	48 byte
Value size	43 ± α byte (variable)
# of KV pairs	200M
# of operations	100M
# of hot key ranges	30
Composition	Get 83% / Put 14% / Scan 3%
Block cache	512MB
Page cache	1GB

쓰레드의 수를 늘려 가면서 실험을 진행하였을 때의 처리량을 나타낸 그래프는

그림 5.3와 같다. 전체적으로 성능의 차이가 YCSB 벤치마크에 비해 줄어들었는데, 이는 키-밸류 쌍의 크기가 작기 때문에 쿼리를 처리하는 데 걸리는 시간에서 I/O가 차지하는 비중이 줄어들기 때문인 것으로 보인다. ARC는 다섯 가지 중 가장 좋은 성능을 보였으며, 커널의 ext4 파일 시스템이나 BlobFS와 비교해서는 20∼30% 정도, LRU나 LRULevel과 비교해서는 5∼15% 정도의 성능 향상을 보였다.

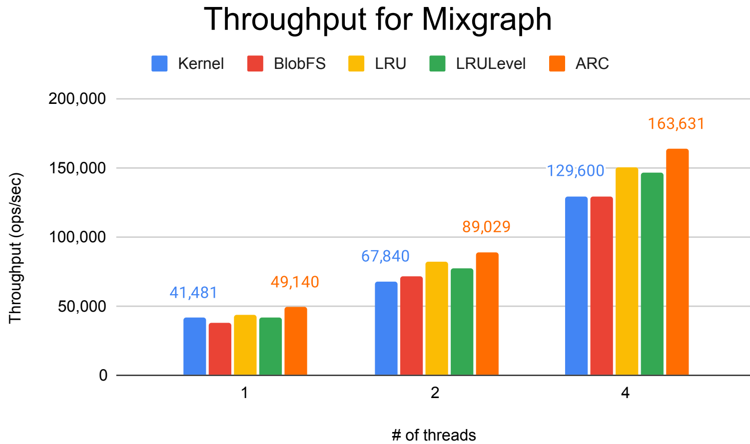


그림 5.3: Mixgraph 벤치마크의 처리량

5.4 캐시의 비율 변화에 따른 성능 차이

LSM-tree 기반 키-밸류 스토어는 읽기 요청의 성능을 높이기 위해 응용 내부에서 별도로 읽기 전용 캐시를 사용한다. 특히 RocksDB에서는 LRU 기반의 블럭 캐시를 통해 압축되지 않은 키-밸류 쌍들이 담긴 블럭을 관리한다. 블럭 캐시는 응용 내부에서 바로 읽기 요청을 처리할 수 있다는 점에서 읽기 성능을 크게 향상시킬 수 있다. 그러나 압축되지 않은 블럭을 관리하기 때문에 SST를 그대로 관리하는 페이지 캐시보다 보관할 수 있는 데이터의 양이 적고, 읽기 요청 전용이기 때문에

컴팩션 과정에서는 사용되지 않는다. 반대로 페이지 캐시는 읽기 요청을 처리하는 데 있어 블럭 캐시보다는 느리지만, 보다 많은 양을 관리할 수 있고 컴팩션 과정에서 큰 역할을 담당한다. RocksDB의 경우, 기본적으로 설정된 블럭 캐시는 8MB이지만 권장되는 크기는 전체 가용 메모리의 1/3이다.

키-밸류 쌍을 임의의 순서로 삽입한 데이터베이스에 대하여 두 가지 캐시의 비율을 조절하며 YCSB 워크로드의 성능을 비교한 결과는 그림 5.4과 같다. 총 3GB 정도의 메모리 공간을 캐시로 사용한다고 할 때 기본 설정인 8MB, 권장 설정인 1GB, 권장 설정보다 많은 2GB를 블럭 캐시로 사용할 때의 처리량을 측정하였다. 세 가지 워크로드에서 모두 권장 설정일 때 성능이 가장 높았으며, 특히 YCSB-C일 때 성능 차이가 두드러지는 모습을 볼 수 있다. 이는 YCSB-A와 YCSB-B의 경우 핫 데이터들이 수정되어 위쪽 레벨에 모이는 반면, YCSB-C의 경우에는 읽기만으로 구성되어 핫 데이터들이 여러 레벨에 흩어져 있기 때문이다. 이를 통해 성능 향상을 위해서는 블럭 캐시와 페이지 캐시의 비율을 적절히 조절할 필요가 있다는 사실을 알 수 있다.

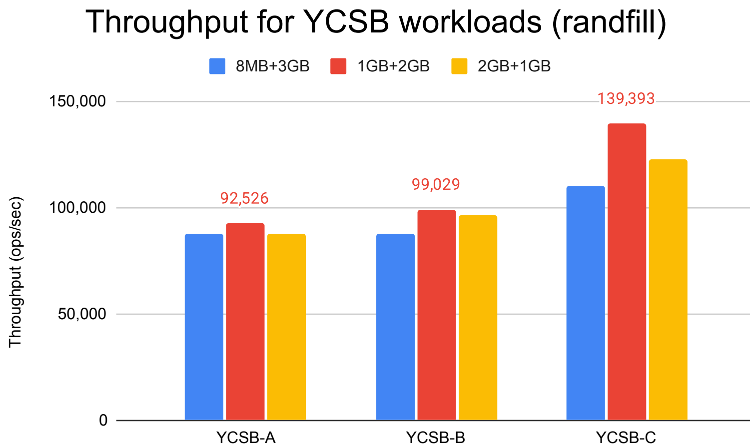


그림 5.4: 블럭 캐시와 페이지 캐시의 비율에 따른 YCSB 워크로드의 처리량

5.5 일반 NVMe SSD에 대한 성능 차이

키-밸류 쌍을 임의의 순서로 삽입했을 때 일반 NVMe SSD인 Samsung 970 PRO에서 YCSB 워크로드의 성능을 비교한 결과는 그림 5.4과 같다. 이전까지의 실험에 사용한 고성능 NVMe SSD인 Intel Optane SSD의 경우 커널 I/O 스택으로 인한 오버헤드가 I/O를 처리하는 데 걸리는 시간의 1/3을 차지한다[7]. 그러나 아직까지 보편적으로 사용되는 일반 NVMe SSD의 경우에는 I/O 처리 시간이 상대적으로 길기 때문에 커널 I/O 스택 오버헤드가 크게 두드러지지 않는다. 따라서 사용자 수준에서 저장 장치에 접근함으로써 얻을 수 있는 이점이 없다. 이로 인해 UserFS의 LRU나 LRULevel은 커널의 ext4와 비슷하거나 떨어지는 성능을 보이게 되며, 읽기 요청을 대부분 저장 장치로부터 직접 처리하는 BlobFS의 경우 성능이 급격히 떨어진다.

반면, I/O를 처리하는데 걸리는 시간이 길면 캐시 미스로 인한 페널티가 크기 때문에 캐시 미스율을 줄이는 것이 성능에 보다 큰 영향을 미치게 된다. 이로 인해 ARC와 LRU 기반 정책의 성능 차가 고성능 NVMe SSD에서보다 큰 것을 확인할 수 있다.

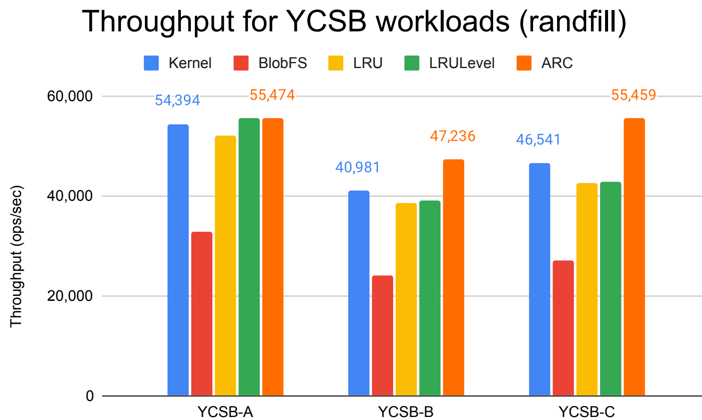


그림 5.5: 일반 NVMe SSD(Samsung 970 PRO)에서의 YCSB 워크로드의 처리량

제 6 장 결론

본 연구에서는 LSM-tree 기반 키-밸류 스토어를 목표 응용으로 하는 사용자 수준 파일 시스템 및 페이지 캐시를 디자인하였다. 파일 시스템의 경우 SPDK를 활용하여 사용자 수준에서 직접 저장 장치에 접근하게 함으로써 커널 I/O 스택으로 인한 오버헤드를 제거하였고, 커널의 ext4 파일 시스템과 동일한 레이아웃을 사용하여 커널과 사용자 수준에서 모두 파일 시스템을 사용할 수 있도록 하였다. 페이지 캐시의 경우 기존의 LRU 기반 페이지 캐시 관리 정책으로는 읽기 미스율을 낮추는 데 한계가 있음을 확인하고, ARC를 기반으로 LSM-tree 기반 키-밸류 스토어의 I/O 패턴에 적합하도록 개선하였다. 이를 통해 커널의 ext4 파일 시스템보다 성능을 향상시킬 수 있었다.

참고 문헌

[1] Rocksdb. https://github.com/facebook/rocksdb.

[2] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.

[3] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. SpanDB: A fast, Cost-Effective LSM-tree based KV store on hybrid storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 17–32. USENIX Association, February 2021.

[4] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. pages 143–154, 09 2010.

[5] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. PinK: High-speed in-storage key-value store with bounded tails. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 173–187. USENIX Association, July 2020.

[6] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, June 2016. USENIX Association.

[7] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. Asynchronous I/O stack: A low-latency kernel I/O stack for Ultra-Low

latency SSDs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 603–616, Renton, WA, July 2019. USENIX Association.

[8] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 819–835, New York, NY, USA, 2021. Association for Computing Machinery.

[9] Kai Lu, Nannan Zhao, Jiguang Wan, Changhong Fei, Wei Zhao, and Tongliang Deng. Tridentkv: A read-optimized lsm-tree based kv store via adaptive indexing and space-efficient partitioning. *IEEE Transactions on Parallel and Distributed Systems*, 33(8):1953–1966, 2022.

[10] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, November 2020.

[11] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. Spdk: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161, 2017.

[12] Takeshi Yoshimura, Tatsuhiro Chiba, and Hiroshi Horii. EvFS: User-level, Event-Driven file system for Non-Volatile memory. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

ABSTRACT

User-level File System Optimized for LSM-tree Based Key-Value Store

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

With the recent development of storage technology, high-performance storage devices with the latency of sub-ten microseconds are emerging. As a result, the overhead due to the kernel I/O stack, which were previously only negligible, is becoming a bottleneck. To address this problem, some user-level frameworks have emerged to eliminate the overhead from the kernel I/O stack by accessing storage devices directly at the user-level. However, to use the framework, it is necessary to implement a separate file system and page cache to match I/O interface with applications and reduce access to storage devices. To this end, this paper designs the user-level file system and page cache and optimizes them for LSM-tree-based key-value stores.

Keywords: User-level file system, User-level page cache, Key-value store
Student Number: 2020-20121