creative commons

COMMONS DEED

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

ZNS SSD를 위한 블록 계층 플래시 캐시

A Block-layer Flash Cache for ZNS SSDs

2023년 2월

서울대학교 대학원

컴퓨터공학부

이 준 석

공학석사학위논문

ZNS SSD를 위한 블록 계층 플래시 캐시

A Block-layer Flash Cache for ZNS SSDs

2023년 2월

서울대학교 대학원

컴퓨터공학부

이 준 석

ZNS SSD를 위한 블록 계층 플래시 캐시

A Block-layer Flash Cache for ZNS SSDs

지도교수 김 진 수

이 논문을 공학석사 학위논문으로 제출함

2022년 11월

서울대학교 대학원

컴퓨터공학부

이 준 석

이준석의 공학석사 학위 논문을 인준함

2023년 1월

위 원 장: 이 재 진 (인)

부위원장: 김 진 수 (인)

위 원: Bernhard Egger (인)

초 록

플래시 메모리는 하드 디스크에 비해 높은 성능을 보이지만 상대적으로 비싸다는 특징이 있다. 플래시 메모리와 하드 디스크가 각각 갖는 장점을 모두 활용하기 위해서 플래시 메모리를 하드 디스크의 캐시로 활용하는 연구가 많이 수행되고 있다. 하지만, 플래시 메모리의 특성 때문에, 플래시 메모리가 캐시로 활용되었을 때 쓰기 증폭 문제가 심화한다. Bcache는 리눅스 커널의 모듈로, 블록 디바이스를 캐시로 사용하는 기능을 제공한다. 하지만, Bcache가 동작하는 방식은 기존의 플래시 캐시가 갖는 쓰기 증폭 문제를 더욱 악화시킨다. 새로운 스토리지 인터페이스인 ZNS를 지원하는 ZNS SSD를 캐싱 다바이스로 활용하면 Bcache가 발생시키는 쓰기 증폭 문제를 해결할 수 있다. 그러나, ZNS SSD를 Bcache의 캐싱 디바이스로 사용하기 위해서는 반출 단위가 zone 크기에 맞춰 커지게 되고 적중률의 손해가 유발된다. 따라서, 적중률 손해를 크게 제한하는 배치, 반출 두 가지 캐싱 정책을 제안한다. 제안된 기법을 통해 약간의 적중률 희생하는 것으로 쓰기 증폭 문제를 해결할 수 있다.

주요어: Bcache, ZNS, 블록 레벨 캐시, 플래시 캐시, 캐싱 정책, Hot/Cold Separation
학 번: 2021-24096

차 례

제 1 장 서론 **1**

제 2 장 배경 지식 **3**

2.1 플래시 메모리 3

2.2 GC(Garbage Collection) 4

2.3 FTL(Flash Translation Layer) 4

2.4 캐싱 정책 5

2.4.1 반출 정책 5

2.4.2 배치 정책 6

2.5 플래시 캐시 6

2.5.1 메모리 사용량 6

2.5.2 쓰기 증폭 7

2.6 Bcache 7

2.7 ZNS(Zoned NameSpace) 11

제 3 장 관련 연구 **13**

3.1 캐싱 알고리즘 13

3.1.1 LRU 기반 정책 13

3.1.2 다계층 LRU 14

3.1.3 Bloom Filter 14

3.1.4 기타 정책들 17

3.2 데이터의 온도 판별 17

제 4 장 설계 및 구현 **19**
4.1 배치 정책 19
4.2 반출 정책 23

제 5 장 실험 결과 및 분석 **25**
5.1 실험 환경 25
5.2 YCSB 벤치마크 25
5.3 캐싱 정책 별 성능 26
5.4 Chunk 크기 별 성능 28
5.5 LFU 반출 정책 성능 29

제 6 장 결론 **31**

ABSTRACT **37**

표 차례

표 5.1 실험 환경 25

그림 차례

그림 2.1 컨벤셔널 디바이스 내부 구조 3
그림 2.2 Bucket 단위로 동작하는 Bcache의 형태 8
그림 2.3 컨벤셔널 디바이스를 사용하는 Bcache의 문제점 8
그림 2.4 Bcache의 캐싱 디바이스로 활용되는 컨벤셔널 디바이스 내부 에서의 WAF 변화 10
그림 2.5 ZNS SSD를 캐싱 디바이스로 활용하는 Bcache 11

그림 3.1 ARC의 형태 14
그림 3.2 DAM의 형태 15
그림 3.3 MBF의 형태 16

그림 4.1 컨벤셔널 디바이스를 사용하는 Bcache와 단순히 Bucket을 ZNS 디바이스의 Zone에 대응시켜 사용하는 Bcache의 적중률 비교 19
그림 4.2 ZNS 다바이스를 위한 캐싱 정책을 사용하는 Bcache의 구조 . . 21

그림 5.1 캐싱 정책에 따른 적중률 비교 26
그림 5.2 Chunk 크기에 따른 적중률 비교 29
그림 5.3 LRU와 LFU 반출 정책 별 성능 비교 30

제 1 장 서론

캐싱 기법은 상대적으로 적은 비용으로 높은 성능의 메모리를 사용할 수 있기 때문에 많은 분야에서 활용되고 있다. 플래시 메모리는 하드 디스크가 갖는 물리적인 제한에서 벗어나 더욱 높은 성능을 보이고, 특히 하드 디스크가 취약한 작은 무작위 읽기에서 높은 성능을 보인다. 그럼에도, 하드 디스크가 갖는 가격의 이점 때문에 많은 양의 데이터를 저장해야 하는 곳에서는 여전히 하드 디스크가 활용되고 있다. 이때, 플래시 메모리가 갖는 성능의 이점을 활용하기 위해 플래시 메모리를 하드 디스크의 캐싱 디바이스로 활용하는 아이디어가 다양하게 연구되고 있다[10, 25, 5, 1, 9, 21, 12, 22, 13, 18].

하지만, 메모리를 캐시로 활용하는 기존의 연구들과 달리, 플래시 메모리를 캐시로 활용할 때는 플래시 메모리가 갖는 고유한 특성으로 인해 쓰기 증폭 문제가 심화한다. 기존의 연구들은 쓰기 증폭 문제를 해결하기 위해 플래시 메모리에 기록되는 데이터를 메모리에 버퍼링하여 모은 후 한 번에 쓰는 방법과[1], 적중률의 하락을 감수하는 FIFO(First In First Out) 기반 반출 정책을 사용하여 큰 연속 쓰기가 발생할 수 있도록 하는 방법을 활용한다[13, 22]. 하지만, 기존의 블록 인터페이스 SSD의 FTL이 제공하는 추상화로 인해 쓰기 증폭 문제가 완전히 해결될 수 없다.

Bcache는 리눅스 커널의 블록 계층에 위치한 모듈로, 블록 디바이스를 캐시로 활용하는 기능을 제공한다. Bcache는 캐싱 다바이스의 공간을 고정 길이의 논리적인 단위인 bucket으로 나누어 사용한다. 하지만 Bcache는 동시에 다수의 bucket을 열어두고 데이터를 병렬적으로 채워가는 형태로 동작하는데, 이러한 형태는 컨벤셔널 디바이스 내부에서 극심한 무작위 쓰기를 발생시켜 높은 쓰기 증폭을 유발한다.

따라서, 새로운 스토리지 인터페이스인 ZNS를 지원하는 ZNS SSD를 활용하여 Bcache의 심각한 쓰기 증폭 문제를 해결하는 방법을 제안한다. Bcache가 bucket이라는 단위로 디바이스를 활용하는 형태는 ZNS 디바이스가 동작하는 방식에 매우

적합하다. 또한, Bcache가 bucket의 단위로 분류한 데이터들은 ZNS의 zone 개념을 통해 디바이스 내부에서도 동일하게 분류될 수 있기 때문에, Bcache가 여러 bucket에 병렬적으로 데이터를 채워나가는 방식으로 인해 겪는 쓰기 증폭 문제에서 해방될 수 있다. 그러나 bucket의 크기가 ZNS SSD의 zone 크기에 맞춰지면서 반출 단위의 크기가 증가하고, 이에 따라 Bcache는 반출 시점마다 높은 적중률 손실을 보인다. 이러한 문제점을 해결하기 위해, 기존의 연구들과 함께 적합한 캐시 정책을 제안한다. 적합한 정책은 다음의 조건들을 고려했다.

1. 메모리 사용량이 디바이스의 크기에 영향을 받는 것을 최소화해야 한다. 플래시 메모리를 캐싱 디바이스로 사용하기 때문에 메모리 캐시에 비해 캐싱에 활용되는 공간의 크기가 크고, 백킹 디바이스로 활용되는 하드 디스크의 크기는 더욱 크다. 따라서, 디바이스의 크기에 메모리 사용량이 깊은 연관이 있는 정책은 심각한 메모리 사용량 문제를 야기한다.

2. 적중률이 희생되는 것을 최소화해야 한다. 반출 시에 증가한 데이터의 손실량으로 인해 적중률이 영향을 받는 것을 효과적으로 관리할 수 있는 정책이 필요하다.

큰 반출 단위는 반출의 대상을 결정하는 캐싱 정책에도 영향을 미친다. 반출 정책으로 가장 일반적으로 사용되는 LRU(Least Recently Used) 기반의 정책은 큰 반출 단위에선 적합하지 않다. 따라서 LRU 기반 정책이 적합하지 않은 이유에 대해서 탐색하고, LFU(Least Frequently Used) 기반 정책을 적합한 반출 정책으로 제안한다. 또한 실험을 통해 두 정책의 성능을 비교하여 보인다.

제 2 장 배경 지식

본 장에서는 논문에서 제시하는 문제점과 제안하는 기법에 대한 이해를 위해 플래시 메모리, 캐싱 정책, Bcache, ZNS에 대해서 설명한다.

2.1 플래시 메모리

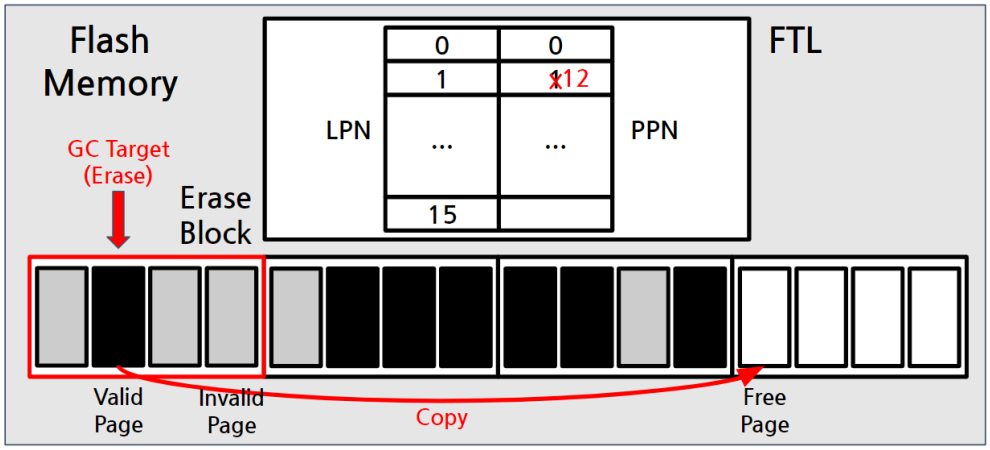

그림 2.1: 컨벤셔널 디바이스 내부 구조

플래시 메모리는 하드 디스크가 가진 물리적인 제약에서 벗어나 더욱 높은 성능을 보인다. 플래시 메모리는 하드 디스크와는 다른 구조로 되어 있고, 동작하는 방식 또한 큰 차이를 갖고 있어 고유한 특징을 보인다. 그림2.1은 플래시 메모리의 내부 구조를 도식화하여 나타낸 그림이다. 플래시 메모리는 페이지의 단위로 데이터가 작성되고, 한번 쓰인 페이지를 재사용하기 위해서는 블록 단위로 비워지는 과정이 선행되어야 하며, 블록 단위는 페이지보다 크다. 이렇게 페이지에 데이터가 쓰이는

것을 작성(program)이라고 하고, 재사용을 위해서 비우는 과정을 삭제(erase)라고 하며, 한 번 쓰이고 지워지는 과정을 Program / Erase Cycle (P/E Cycle)이라고 한다. 하지만, 플래시 메모리는 제한된 수명을 가지고 있기 때문에, 블록마다 견딜 수 있는 P/E Cycle이 제한되어 있다.

2.2 GC(Garbage Collection)

플래시 메모리를 사용하는 과정에서 디바이스 내부가 유효한 또는 유효하지 않은 데이터를 담고 있는 페이지들로 채워지게 된다(그림2.1). 추가적인 호스트의 쓰기 요청을 수행하기 위해선 유효하지 않은 페이지들을 비워서 새로운 자유 공간을 확보해야 한다. 이런 과정을 GC라고 한다. 삭제의 단위는 작성 단위보다 더 크기 때문에 하나의 삭제 블록 단위는 다수의 페이지로 구성되어 있다. 만약 삭제하고자 하는 블록 안에 유효한 페이지가 있다면, 해당 페이지의 데이터를 보존하기 위해, 별도의 페이지에 복사한 후 삭제를 진행한다. 이는 사용자가 발생시킨 쓰기 요청과는 별도로 추가되는 쓰기로, 쓰기 증폭(WA, Write Amplification)이라고 부른다. 특히, 디바이스 내부에서 발생하는 쓰기 증폭은 디바이스 수준의 쓰기 증폭(DLWA, Device Level Write Amplification)이라고 부른다. 쓰기 증폭은 플래시 메모리의 제한된 수명에 악영향을 미친다.

2.3 FTL(Flash Translation Layer)

GC의 과정에서 삭제 대상의 블록 안에 있는 유효한 데이터의 위치가 새롭게 변경될 수 있고, 이 때문에 사용자가 처음 작성했던 데이터의 위치와 실제 데이터의 위치가 달라질 수 있다. 이러한 문제점을 해결하기 위해서 플래시 디바이스 내부의 FTL에서 추상화를 제공한다(그림2.1). 사용자가 사용하는 주소인 논리 페이지 번호(LPN, Logical Page Number)와 데이터가 위치한 실제 주소인 물리 페이지 번호(PPN, Physical Page Number) 사이의 사상 정보를 관리하고, PPN이 바뀌었을 때

해당 사상 정보를 변경해 주는 것으로 앞선 문제를 해결할 수 있다. 하지만, FTL의 추상화는 호스트의 입장에서 데이터가 디바이스 내부에서 어떻게 구성되는지 알 수 없게 만들고, 디바이스의 입장에서 호스트가 의도한 데이터의 특성을 알지 못하게 만든다. 이러한 호스트와 디바이스 사이의 정보 교환의 단절 때문에 호스트가 데이터들을 목적(e.g 수명)에 맞춰 분류하고자 하더라도, 디바이스 내부에서는 다양한 목적을 갖는 데이터들이 같은 삭제 블록 안에 저장될 수 있다. 이는 자연히 DLWA를 증가시키고, 호스트가 발생시키는 작은 무작위 쓰기는 이러한 현상을 심화시킨다.

2.4 캐싱 정책

캐싱 정책은 캐시가 동작하는 방식을 결정하여 캐시 성능에 큰 영향을 미친다. 캐싱 정책은 그 역할에 따라 크게 2가지로 나뉜다.

2.4.1 반출 정책

캐싱 기법은 모든 데이터가 저장된 백킹 디바이스 위에 크기는 작지만, 더 빠른 디바이스를 캐싱 디바이스로 활용하는 기법이다. 이를 통해, 적은 비용으로 높은 성능을 성취해낼 수 있다. 캐싱 다바이스의 크기는 백킹 디바이스의 크기보다 작기 때문에, 동시에 캐싱할 수 있는 데이터의 양에 한계가 있다. 따라서, 캐시의 공간이 가득 찼을 때, 새로운 데이터를 캐싱하기 위한 자유 공간을 확보하는 정책이 필요하다. 반출 정책은 이미 캐시에 반입된 데이터 중 반출할 대상을 결정하는 정책이다. 반출 정책은 캐싱 되어 있는 데이터 중에서 가장 캐싱 효율이 떨어지는 데이터를 반출 대상으로 선정한다. 반출 정책에 대한 다양한 연구들이 있고, LRU 기반의 정책들이 주를 이루고 있다[14, 8, 11, 19, 16, 23]. Bcache는 반출 정책으로 LRU, FIFO, Random을 지원하며, LRU를 기본 정책으로 채택하고 있다.

2.4.2 배치 정책

배치 정책은 반출 정책과는 다른 목적을 갖는다. 반출 정책은 캐싱 되어있는 데이터 중에서 대상을 선정하는 반면, 배치 정책은 아직 캐싱 되지 않은 데이터를 대상으로 동작을 결정한다. 캐시에 반입되지 않은 데이터 중 반입 시 데이터 구성 방법(e.g 데이터의 특성에 따른 그룹화 방법) 등을 결정한다. 기존의 캐싱 정책은 반출 정책에 초점이 맞추어져 있어 적절한 배치 정책에 대한 고려가 필요하다[2].

2.5 플래시 캐시

플래시 캐시는 플래시 메모리를 캐싱 디바이스로 활용하는 기법이다. 일반적으로 두 계층의 디바이스를 이용해서 동작하며, 저장되는 모든 데이터를 관리하는 백킹 다비이스와, 성능을 위해 일부의 데이터만 관리하는 캐싱 디바이스로 구성되어 있다. 백킹 디바이스로는 플래시 메모리보다 성능은 낮지만, 값이 싸고 용량이 큰 하드 디스크를 사용한다. 메모리를 캐시로 사용하는 기존의 캐싱 기법들과 달리, 플래시 메모리를 캐싱 디바이스로 활용했을 때는 플래시 메모리가 갖는 고유한 특성 때문에 다양한 문제들이 발생한다.

2.5.1 메모리 사용량

플래시 메모리는 DRAM에 비해서 커다란 용량을 갖고 있기 때문에 캐싱 되는 데이터의 수가 현저히 늘어난다. 따라서, 캐싱 되어 있는 데이터의 위치를 저장하기 위한 인덱스의 크기도 증가한다. 이러한 문제는 저장되는 데이터의 크기가 매우 작은 키-벨류 캐시에서 두드러지게 나타난다. Bcache의 경우, 블록 계층의 캐시로, 캐싱 되는 데이터가 키-벨류 데이터보다 큰 4KB 블록 단위이고, 이 때문에 인덱스가 발생시키는 메모리 사용량 문제에 영향을 덜 받는다. 또한, 인덱스를 플래시 메모리에 저장하고, 인덱스의 일부를 메모리에 캐싱해서 활용하는 방법으로 메모리 사용량 문제를 대처하였다.

2.5.2 쓰기 증폭

플래시 메모리는 DLWA 이외에도 다양한 원인이 쓰기 증폭을 발생시킬 수 있다. 로그 구조 파일시스템(LFS, Log-structured File System)[17]에서는 자유 공간을 확보하기 위해서 FTL과 유사한 GC 과정을 파일 시스템 계층에서 진행한다. 이때, 이미 저장된 데이터를 새로운 영역에 옮겨적는 과정이 발생하고, 같은 데이터의 쓰기가 중복되어 수행되는 쓰기 증폭을 발생한다. 이러한 쓰기 증폭을 애플리케이션 계층의 쓰기 증폭(ALWA, Application Layer Write Amplification)이라고 부른다. ALWA, DLWA 모두 추가적인 쓰기를 발생시키고, 제한된 플래시 메모리의 수명에 악영향을 미쳐, 디바이스의 소모를 가속한다.

2.6 Bcache

Bcache는 리눅스 커널의 모듈로, 플래시 메모리를 캐싱 디바이스로 사용하는 기능을 제공한다. Bcache는 백킹 디바이스 크기만큼의 용량을 가진 블록 디바이스를 상위 레벨에 노출한다. 상위 레벨에서 Bcache가 노출한 블록 디바이스에 저장하는 모든 데이터는 백킹 디바이스에 반영된다. 이후 접근되는 데이터들은 캐싱 디바이스에 복사되어 반입되고, 백킹 디바이스 대신 성능이 높은 캐싱 디바이스가 상위 레벨의 요청을 대신 처리하는 것으로 성능 향상을 끌어낸다. 캐시 워크로드는 인덱스의 업데이트가 빈번히 발생하고 이는 플래시 메모리에 저장된 인덱스의 in-place update를 발생시킨다. 빈번한 in-place update는 플래시 메모리의 성능과 수명에 악영향을 미치고, Bcache는 이러한 문제를 해결하기 위해서 append only b+tree를 인덱스로 활용한다.

그림2.2는 Bcache가 캐싱 디바이스를 사용하는 방식을 도식화하여 나타낸 그림이다. Bcache는 캐싱 디바이스의 전체 공간을 bucket이라는 1MB의 고정된 논리적인 공간으로 나누어 사용한다. Bcache의 모든 동작은 bucket 단위로 이루어진다. 캐싱 될 데이터의 위치를 할당하는 단위와, 자유 공간 확보를 위해 데이터를 반출

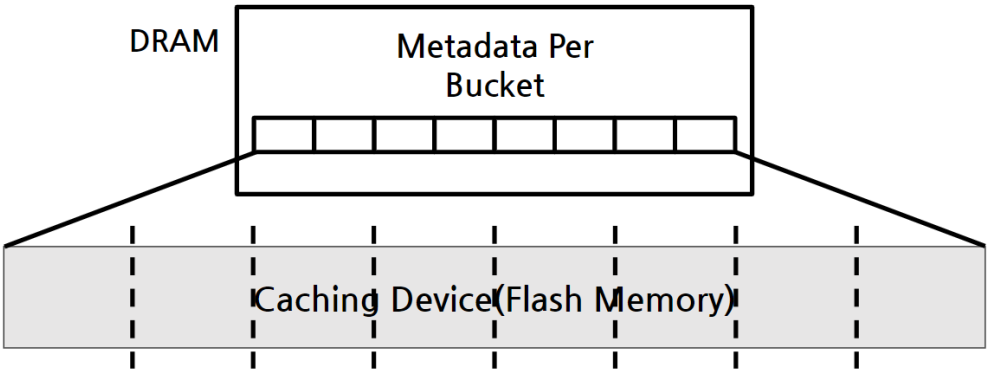

그림 2.2: Bucket 단위로 동작하는 Bcache의 형태

시키는 단위도 bucket이다. Bucket이 재사용되면, 해당 bucket 안에 캐싱 되어 있던 데이터들을 가리키는 인덱스의 키들이 전부 무효화 되어야 한다. 이러한 과정을 단순화하기 위해서 bcache에서는 bucket 마다 세대 번호를 관리하고 있으며, bucket이 재사용될 때마다 해당 번호를 증가시킨다. 유효하지 않은 키를 통한 데이터 접근은, 해당 키가 가진 세대 번호와 키가 가리키는 bucket의 현재 세대 번호를 비교하는 것으로 판별해낼 수 있다.

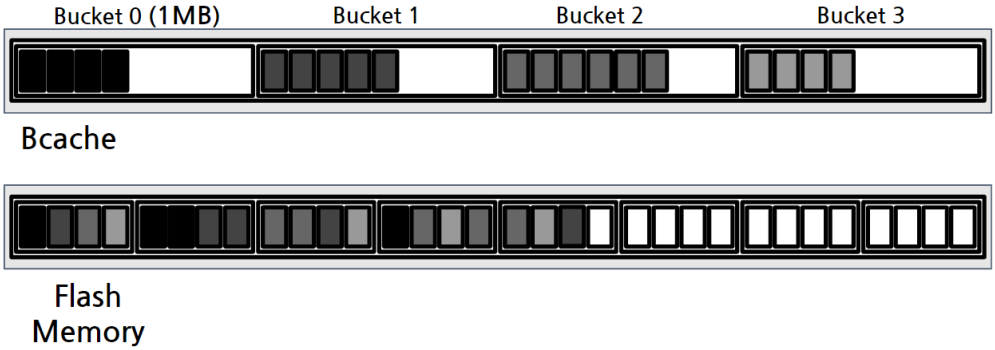

그림 2.3: 컨벤셔널 디바이스를 사용하는 Bcache의 문제점

Bcache는 캐싱하고자 하는 데이터의 논리 블록 주소(LBA, Logical Block Address)가 연속되어 있거나, 같은 테스크에서 발생한 요청의 데이터인 경우, 동일한 bucket에 캐싱 될 수 있도록 하는 간단한 배치 정책을 사용한다. 이를 위해서 open bucket이라는 개념으로 동시에 여러 개의 bucket을 열어두고 병렬적으로 데이터를 채워나간다. 그림2.3은 Bcache가 컨벤셔널 디바이스를 캐싱 디바이스로 사용하여 데이터를 채웠을 때, Bcache가 논리적으로 구성한 데이터의 형태와 실제 컨벤셔널 디바이스 내부에 저장되는 데이터의 형태를 표현한 그림이다. 그림에서 표현된 것처럼, Bcache가 bucket이라는 논리적인 단위로 데이터를 분류하여 데이터를 채우더라도 디바이스 내부에서는 다양한 bucket의 데이터가 하나의 삭제 블록 안에 섞여 들어가게 된다. 이러한 현상이 발생하는 이유는, FTL에서의 추상화로 인해 발생한 정보의 단절 때문이다. FTL의 추상화로 인해 디바이스 내부에서는 요청된 데이터가 어떤 bucket에 속해있는지 알 수 없다. 따라서, Bcache가 여러 bucket들에 병렬적으로 채워나가는 데이터들은 요청되는 순서대로 디바이스 내부에 기록되고, 이는 작은 무작위 쓰기와 동일한 효과를 발생한다. Bcache가 데이터를 반출시키는 단위는 bucket 단위이기 때문에, 동일한 삭제 블록에 섞여 들어간 데이터들은 각각이 속해있는 bucket이 반출되는 시점까지의 수명을 갖는다. 따라서, 하나의 삭제 블록 안에서 다양한 수명을 갖는 데이터들이 존재하게 되고, GC 과정에서 복사해야 하는 유효한 데이터의 양이 증가하여 DLWA가 커진다.

그림2.4는 Bcache가 컨벤셔널 디바이스를 캐싱 디바이스로 사용할 때, 디바이스 내부에서 GC 연산이 수행될 때마다 WAF(Write Amplification Factor)를 측정한 결과를 그래프로 나타낸 것이다. 실험에 사용된 환경은 5장에서 언급하는 실험의 환경과 동일하다. WAF는 호스트가 실제 쓰고자 하는 데이터 양 대비 실제 디바이스에서 발생한 쓰기의 양을 나타낸다. 만약 호스트가 요청한 1MB의 데이터를 작성하기 위한 공간 확보를 위해 디바이스 내부의 GC 과정이 1MB의 유효한 데이터 복사를 수행했다면, WAF는 실제 쓰고자 하는 양(1MB) 대비 실제 발생한 쓰기의 양(실제 쓰인 데이터 1MB + 유효 데이터 복사 1MB = 2MB)이기 때문에 2(2MB/1MB)가 된다.

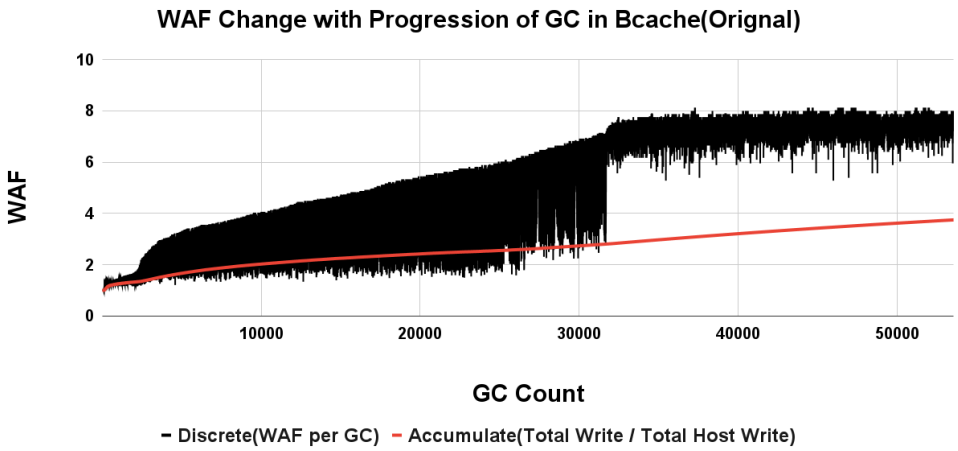

그림 2.4: Bcache의 캐싱 디바이스로 활용되는 컨벤셔널 디바이스 내부에서의 WAF 변화

따라서, WAF가 1이라는 것은 쓰기 증폭이 전혀 일어나지 않았다는 것을 의미한다. 실험 결과에서 볼 수 있듯이 실험이 진행됨에 따라서 자유 공간 확보를 위해 GC가 지속해서 발생하고, GC가 진행됨에 따라서 디바이스 내부의 WAF가 지속해 증가하는 것을 확인할 수 있다. 그래프의 검은색은 GC가 발생할 때마다의 WAF를 측정한 결과이고, 빨간색은 실험의 시작부터 누적된 결과다. GC 개별의 WAF는 8에 가까운 수치가 측정되는 반면, 누적 WAF는 4 정도로 비교적 낮게 측정되는 이유는 초기에 캐싱 디바이스가 비워진 상태에서 데이터를 채우는 과정이 포함되기 때문이다. 따라서, 시간이 흐르면서 누적 WAF도 GC 개별의 WAF인 8에 수렴할 것을 예상할 수 있다. 이 실험 결과를 통해서 실제 캐싱하고자 하는 데이터 대비 디바이스 내부에서는 약 8배 많은 쓰기가 수행되고 있고, Bcache의 동작 방식은 제한된 플래시 메모리의 수명을 심각하게 저해하고 있다는 것을 확인할 수 있다.

2.7 ZNS(Zoned NameSpace)

ZNS는 기존의 블록 인터페이스의 FTL에서 제공하는 추상화로 인해 발생하는 호스트와 디바이스 간 정보 교환의 단절을 해결하기 위해서 제안되었다. 디바이스의 저장 공간은 Zone이라는 논리적인 단위로 나뉘고, zone의 개념은 디바이스를 사용하는 호스트에게 공개된다. Zone의 크기는 큰 zone 혹은 작은 zone인지 여부에 따라서 달라진다. ZNS는 같은 zone 안에서 데이터들이 반드시 연속되어야 한다는 제약조건을 갖고 있다. 플래시 메모리의 삭제 블록의 특성과 마찬가지로 zone을 재사용하기 위해서는 zone 초기화(reset)를 해야 하고, zone 안에 있는 모든 데이터가 같이 비워진다. ZNS는 블록 인터페이스와 다르게 디바이스가 GC를 하지 않기 때문에 호스트가 직접 초기화할 대상을 선정하고, 원하는 시점에 GC를 수행해야 한다. 선정한 GC 대상 zone에 대해서 초기화 명령을 디바이스에 내려야 하며, 해당 zone에서 유효한 데이터를 새로운 영역에 복사하여 살리는 과정을 직접 수행해야 한다. 이렇게 zone이라는 개념을 통해서 호스트와 디바이스 사이에서 정보교환이 가능해진다. 이를 통해 데이터에 관한 풍부한 정보를 가진 호스트가 그 특성을 고려하여 데이터들을 구성하고, zone을 통해서 데이터의 구성이 디바이스 내부에서도 반영된다. 호스트와 디바이스 내부의 일치된 데이터의 구성은 DLWA를 효과적으로 관리하는 것을 가능하게 만든다.

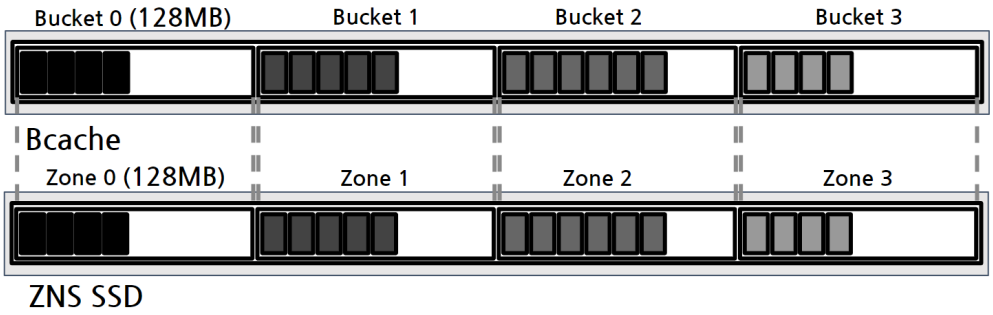

그림 2.5: ZNS SSD를 캐싱 디바이스로 활용하는 Bcache

Bcache가 디바이스를 논리적인 bucket 단위로 나눠서 사용하는 형태는, ZNS의 zone이 동작하는 방식과 매우 유사하다. Bcache가 사용하는 bucket을 ZNS의 zone에 대응시키면 자연스럽고 효과적으로 ZNS를 활용할 수 있다. 그림2.5는 ZNS SSD를 캐싱 디바이스로 활용하는 Bcache의 형태를 나타낸 그림이다. Bcache의 bucket이 ZNS의 zone에 대응되었고, bucket의 크기가 zone 크기에 맞춰서 1MB에서 128MB로 증가하였다. 그림2.3과는 다르게 Bcache가 bucket을 기준으로 분류한 데이터들이 zone을 통해 디바이스 내부에서도 동일하게 분류되어 저장된다. 따라서 Bcache가 여러 open bucket들을 병렬적으로 채워나가더라도, bucket 다수의 데이터가 디바이스 내부에서 하나의 삭제 단위로 섞여 들어가지 않는다. Bcache는 공간 확보를 위한 반출 정책이 bucket 단위로 이루어지고, 반출 대상으로 선정된 bucket은 그에 대응하는 zone을 초기화하는 것으로 간단히 수행될 수 있기 때문에 유효한 데이터를 복사하는 별도의 과정을 제거할 수 있다. 이에 따라 플래시 캐시가 가진 쓰기 증폭 문제가 완전히 해결된다. 또한, Bcache는 하나의 bucket 안에 작성되는 데이터들은 연속된 형태로 채워나가기 때문에, ZNS의 zone 안에서 데이터의 작성이 연속되어야 한다는 조건이 충족된다.

제 3 장 관련 연구

3.1 캐싱 알고리즘

캐싱 알고리즘에 대한 다양한 연구가 존재한다. 하지만, 기존의 연구들은 배치 정책이 아닌 반출 정책에 집중되어 있다. 본 장에서는 다양한 반출 정책을 카테고리별로 소개한다.

3.1.1 LRU 기반 정책

기존의 캐싱 정책들이 초점을 두고 있는 반출 정책의 목적은 이미 캐싱 되어 있는 데이터 중 가장 캐싱 효율이 떨어지는 데이터를 판별해내는 것이다. 하지만 반출 대상을 선정하는 최적 기법은 미래의 데이터 접근 패턴을 알아야 한다는 점 때문에 구현에 어려움이 있다. 이를 해결하기 위해서 과거의 데이터 접근 패턴을 미래 접근 패턴의 근사치로 사용하고자 하는 기법들이 많이 사용된다. LRU는 현재 캐싱 되어 있는 데이터 중에서 가장 오랫동안 접근되지 않은 데이터를 반출 대상으로 선정한다. LRU 구현을 위해서 데이터들의 접근 정보를 관리하는 큐를 두고, 데이터가 접근될 때마다 해당 데이터와 대응되는 정보를 큐의 MRU(Most Recently Used)의 위치로 이동한다. 오랫동안 접근되지 않은 데이터는 LRU 위치로 밀려나게 되고, 반출 시점에 LRU 위치에 있는 데이터가 반출 대상이 된다.

ARC[14]는 LRU를 기반으로 만들어진 정책이다. 그림3.1은 ARC가 동작하는 방식을 나타낸 그림이다. ARC는 실제 캐싱 되어있는 데이터를 관리하는 영역(T1과 T2)과, 가까운 시기에 캐시에서 반출된 데이터의 메타데이터를 관리하는 영역(B1과 B2) 두 가지를 활용하여 동작한다. 각 영역의 크기는 워크로드에 따라서 동적으로 조절된다. T2 영역에 들어오기 위해서는 짧은 시간 안에 2번 이상 접근이 일어나야 하므로, 스캔과 같이 큰 영역을 한 번만 읽는 워크로드로 인해 캐싱 되어 있는 접근이

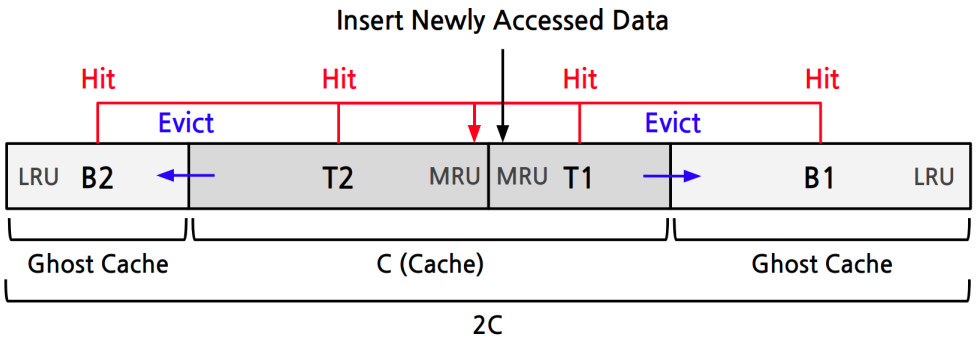

그림 3.1: ARC의 형태

빈번한 데이터들이 반출되는 것을 방지할 수 있다.

3.1.2 다계층 LRU

기본적인 LRU 기반의 정책들은 하나의 LRU 리스트를 활용한다. SLRU[6]는 기존의 LRU들과 다르게 여러 계층의 LRU 리스트를 활용하여 동작한다. 캐시 누락이 발생해서 새로운 데이터가 캐싱 될 때, 해당 데이터는 가장 낮은 레벨 리스트의 MRU 위치에 들어온다. 만약, 캐시 적중이 발생했다면, 해당 데이터가 있던 리스트에서 한 단계 높은 리스트의 MRU로 옮겨지게 된다. 새롭게 올라온 데이터로 인해서 밀려난 상위 레벨의 데이터는 아래 리스트(새롭게 올라온 데이터가 들어있던 리스트)의 MRU로 들어가게 된다. 결과적으로 가장 아래 있는 리스트에서 밀려난 데이터가 캐시에서 반출되게 된다. 가장 높은 레벨에서 적중이 발생하는 경우에는 레벨의 변화 없이 해당 리스트의 MRU로 이동된다.

3.1.3 Bloom Filter

리스트를 관리하고 유지하는 것은 상대적으로 큰 메모리와 계산 자원을 소모한다. 따라서, Bloom Filter를 이용하는 방법이 제안되었다. Bloom Filter 기법은 새롭

게 캐싱할 데이터의 LBA를 다수의 해싱 함수에 대입해서 해시값을 계산한다. 이후 계산된 해시값들에 대응되는 Bloom Filter의 엔트리들에 접근 정보를 기록한다. 기록된 접근 정보는 데이터의 접근 빈도를 파악하는 데 활용된다. Bloom Filter 기법은 적은 메모리와 적은 계산양으로 동작한다는 장점을 갖고 있지만, 거짓 양성 판단을 허용하기 때문에 잘못된 예측 결과가 도출될 수 있다는 단점을 갖고 있다.

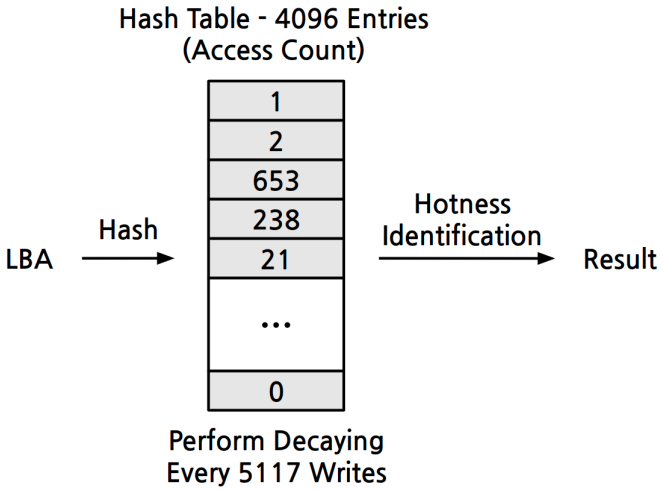

그림 3.2: DAM의 형태

그림3.2는 DAM[7]의 구조를 나타낸 그림이다. DAM은 가장 기본적인 형태의 Bloom Filter 기반의 캐싱 정책으로, 새롭게 요청된 데이터를 바탕으로 다수의 해싱 함수를 이용하여 대응되는 Bloom Filter의 엔트리들을 계산해낸다. 각 엔트리는 접근된 횟수를 저장하고, 하나의 LBA가 다수의 해싱 함수로 결정되는 엔트리들의 횟수를 전부 증가시킨다. 데이터의 접근 빈도 판단 시에는, 데이터의 LBA에 대응되는 여러 엔트리 중 가장 적은 횟수 값을 이용한다. 이러한 형태는 엔트리들의 횟수가 지속해 증가하고, 한번 횟수가 높아진 엔트리는 이후 접근이 되지 않더라도 높은 수치를 유지한다는 문제점이 있다. 이를 해결하기 위해서 일정 접근 주기마다 모든 엔트리의 횟수를 절반으로 줄이는 decaying 기법이 함께 사용된다.

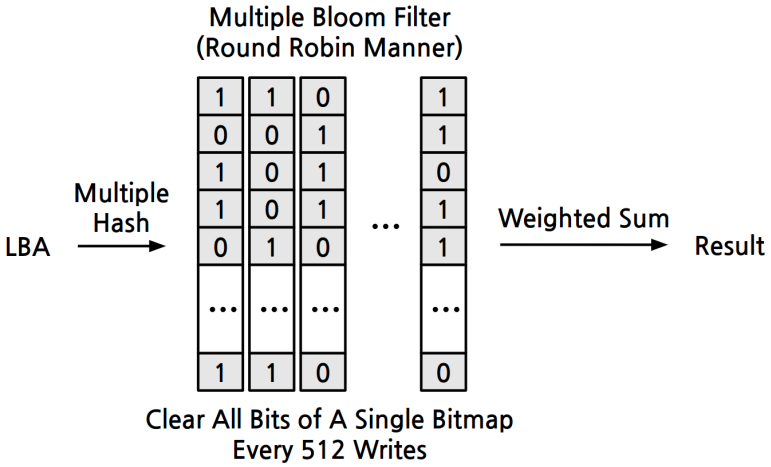

그림 3.3: MBF의 형태

MBF(Multiple Bloom Filter)[15]는 기존의 Bloom Filter 기법들이 최신도(최근에 저장된 정보들일수록 더욱 높은 신뢰도를 부여)를 제대로 반영하지 못한다는 문제를 해결했다. MBF가 갖는 가장 큰 차이점으로는 여러 벌의 Bloom Filter를 이용하고, 접근된 횟수를 세는 것이 아닌, 접근될 때마다 하나의 Bloom Filter의 비트를 1로 설정한다는 것이다. 그림3.3은 MBF가 동작하는 방식을 표현한 그림이다. 새로운 접근이 발생하면, 해당 데이터의 LBA를 다수의 해싱 함수로 계산한다. 각 Bloom Filter들은 비트맵의 형태로, 엔트리 수만큼의 비트를 갖고 있다. 여러 벌의 Bloom Filter들은 Round Robin의 형태로 사용되고, 하나의 접근으로 특정 Bloom Filter의 비트들이 설정되면, 이어지는 접근은 다음 순서의 Bloom Filter에 기록된다. MBF는 decaying 기법으로 일정 접근 주기마다 Round Robin 방식으로 선택되는 Bloom Filter의 모든 비트를 비우는 방식을 사용한다. 이에 따라, 비워진 Bloom Filter의 순번과 가까이 있는 Bloom Filter일수록 최근의 데이터만 담고 있고, 멀어질수록 오래된 기록들이 함께 담겨 있다. 따라서, 데이터의 접근 빈도를 판단할 때는 비워진 Bloom Filter에 가까워질수록 높은 가중치를 두어 더한 가중 합산 방식을 사용한다.

이 방법을 통해서 최신도를 활용할 수 있다.

3.1.4 기타 정책들

LFU(Least Frequently Used) 기반의 정책은 LRU 기반의 정책과는 다르게, 각 데이터의 접근 빈도를 통해서 반출 대상을 선정한다[4]. LRU 정책과 LFU 정책은 각각의 특징이 존재하고, LRFU[11]는 이러한 특징을 융합하여 활용하는 기법을 제안한다. 최근 활발히 연구되고 있는 기계학습 기법을 활용한 연구도 존재한다[20]. 캐싱 워크로드가 수행되면서 발생하는 다양한 정보를 기계학습에 활용하여 캐싱 정책을 구현한다. 기계학습 기반의 캐싱 알고리즘은 메모리 사용량과 연산량이 상대적으로 많다는 특징을 갖고 있다.

3.2 데이터의 온도 판별

Multistream SSD는 사용자에게 stream이라는 인터페이스를 제공한다. 사용자는 사용하는 데이터의 특성에 맞춰 stream이라는 개념으로 분류하여 SSD에 저장할 수 있다. 이는 FLT의 추상화로 인해 발생한 디바이스와 사용자 간의 정보 단절을 해결하는 것으로 디바이스의 성능과 수명 증가를 가져올 수 있다. 이러한 Multistream SSD를 효과적으로 활용하는 방법을 제안한 다양한 연구들이 있다[24, 3].

그중 AutoStream[24]은 디바이스에 저장되는 데이터들의 온도(접근 빈도)를 능동적으로 분석하여 stream을 분류하는 2가지 기법을 제안한다. 하나는 SLRU와 유사하게 MQ(Multi-Queue)를 사용하는 방법이고, 다른 하나는 전체 디바이스 공간을 2M 크기의 고정된 chunk 단위로 나누어 관리하는 SFR(Sequentiality Frequency Recency) 방법이다. SFR의 경우 논리적으로 나눈 chunk마다 발생하는 요청을 기록하고, 기록된 횟수를 이용하여 데이터들의 온도 분류한다. AutoStream의 경우 온도를 다양한 레벨로 분류할 수 있는 특징이 있다. 하지만, 저장에 활용되는 디바이스의 전체 크기에 비례하게 데이터를 관리해야 하므로 디바이스가 커지면 메모리와 계산

자원의 소모가 심해진다.

제 4 장 설계 및 구현

본 논문에서는 Bcache가 ZNS를 캐싱 디바이스로 사용하는 환경에 적합한 배치 정책과 반출 정책을 제안한다.

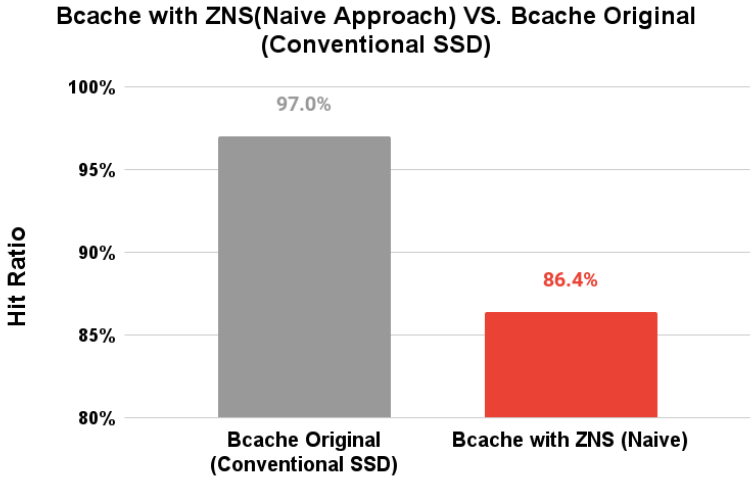

그림 4.1: 컨벤셔널 디바이스를 사용하는 Bcache와 단순히 Bucket을 ZNS 디바이스의 Zone에 대응시켜 사용하는 Bcache의 적중률 비교

4.1 배치 정책

그림4.1은 컨벤셔널 디바이스를 사용하는 기존의 Bcache와 단순히 bucket을 ZNS 디바이스의 zone에 대응시켜 사용하는 Bcache의 적중률을 비교한 결과다. 컨벤셔널 디바이스를 사용할 때와 다르게, ZNS 디바이스를 캐싱 디바이스로 활용하면 bucket의 단위로 분류된 데이터들이 디바이스 내부에서도 zone에 의해 동일하게 분류되어

저장되고, 쓰기 증폭 문제를 해결할 수 있다. 하지만 실험 결과에서 볼 수 있듯이 컨벤셔널 디바이스를 사용할 때와 대비해서 적중률이 크게 저하되는 것을 확인할 수 있다. 이러한 현상이 발생하는 이유는 반출의 단위가 zone 크기에 맞춰 증가했기 때문이다. 컨벤셔널 다비이스를 사용하는 Bcache bucket의 크기는 1MB이지만, ZNS 디바이스를 사용하는 Bcache의 bucket 크기는 128MB이다. Bucket은 Bcache의 반출 단위이고, 커진 반출 단위는 다음의 2가지 이유로 적중률 하락을 야기한다.

1. 반출 시점에 컨벤셔널 디바이스를 사용하는 Bcache에 비해 ZNS 디바이스를 사용하는 Bcache는 128배 많은 데이터를 손실하게 된다.

2. 기존의 Bcache는 데이터를 저장할 bucket을 할당할 때 데이터의 온도를 깊게 반영하지 않는다. 반출 단위가 작을 때는 해당 bucket을 구성하는 데이터들의 수가 많지 않다. 따라서, 반출 정책을 통해 온도가 낮은(접근 빈도가 낮은) bucket을 내보내는 것만으로 데이터들의 온도를 충분히 반영할 수 있다. 하지만, 큰 반출 단위 안에는 다양한 온도를 갖는 데이터들이 함께 저장되게 되고, 반출 정책에 의해서 온도가 낮게 판별된 bucket이라도 높은 온도를 갖은 데이터들이 함께 포함되어 있을 가능성이 커진다. 그 결과 반출 시에 높은 온도의 데이터들이 함께 반출되면서 적중률이 크게 하락하는 결과를 가져온다.

이러한 문제를 해결하는 가장 간단한 방법으로는 반출 대상이 비워질 때, 해당 bucket 안에 있는 데이터 중 온도가 높은 데이터를 살려 캐시에 다시 반입하는 방법이 있다. 하지만 이것은 ALWA를 야기한다. 기존의 컨벤셔널 디바이스를 사용하는 Bcache가 안고 있는 쓰기 증폭 문제를 해결하기 위해서 ZNS를 사용했지만, ZNS를 사용하기 위해서 다시 쓰기 증폭을 발생시키는 모순적인 상황이 발생한다. 따라서 데이터를 내보내는 시점이 아닌, 데이터를 캐시에 반입하는 시점에 데이터들의 온도를 예측하여 분류하는 배치 정책이 필요하다. 온도가 비슷한 데이터들끼리 분류가 되어 있다면, 반출 대상으로 선정된 bucket에 높은 온도의 데이터들이 함께 포함되어 내보내지는 현상을 방지할 수 있다.

하지만, 기존의 캐싱 알고리즘들은 반출 정책에 초점이 맞춰져 있다. 즉, 이미 캐싱 되어 있는 데이터 중 어떤 대상을 내보낼지 결정한다. 그러나 이러한 알고리즘들은 다음과 같은 사고의 전환을 통해서 배치 정책으로 활용될 수 있다.

1. 디바이스에 캐싱 되는 모든 데이터의 정보를 저장하지 않고, 일부만을 관리한다.

2. 현재 디바이스에 저장된 데이터의 정보를 관리하는 것이 아니라, 과거 접근 정보의 일부를 샘플링하여 새롭게 캐싱할 데이터의 온도 예측에 활용한다.

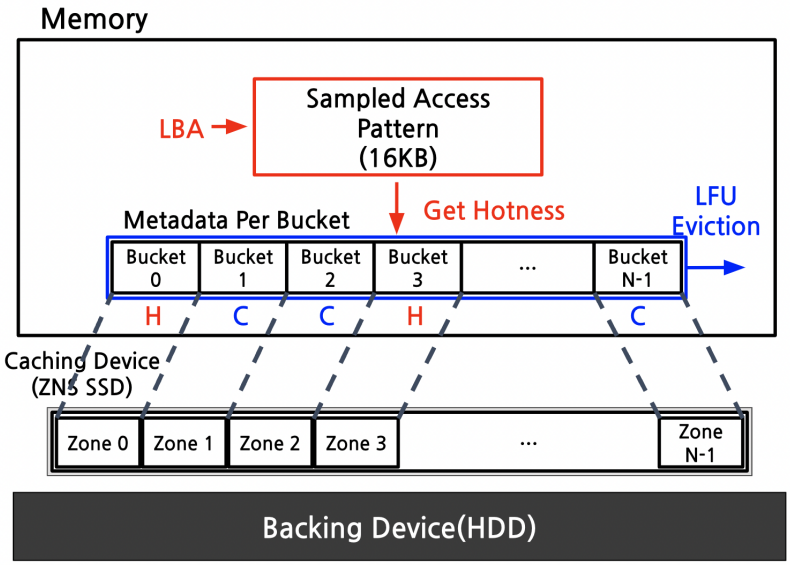

그림 4.2: ZNS 다바이스를 위한 캐싱 정책을 사용하는 Bcache의 구조

그림 4.2는 본 논문에서 제안하는 ZNS 디바이스를 위한 캐싱 정책을 사용하는 Bcache의 전체 구조를 나타낸 그림이다. 기존의 Bcache와 동일하게 하드 디스크로 이루어진 백킹 디바이스 위에 플래시 메모리를 캐시로 사용하는 구조로 되어 있고,

블록 인터페이스를 사용하는 컨벤셔널 디바이스 대신 ZNS 디바이스를 캐싱 디바이스로 활용한다는 차이가 있다. ZNS 디바이스는 zone이라는 개념을 통해서 디바이스 전체 공간을 분할하여 노출하고, Bcache는 이 zone들에 bucket을 대응하여 사용한다. Bcache가 사용하는 논리적인 단위인 bucket들의 정보를 관리하기 위해서 bucket 수만큼의 메타데이터를 메모리에서 관리한다. ZNS 디바이스를 위한 캐싱 정책을 활용하는 Bcache는 기존의 기법과는 다르게 과거 데이터 접근 패턴의 일부를 샘플링하는 구조를 메모리에 추가로 유지한다. 데이터 접근 샘플링 구조는 16KB로 매우 작은 크기만을 차지하고, 사용하는 크기는 설정이 가능하다. 본 구조를 사용하는 초기에는 샘플링된 과거의 접근 패턴이 없기 때문에 접근되는 모든 데이터가 낮은 온도로 분류되어 같은 bucket에 함께 저장된다. 하지만, 워크로드가 진행되면서 반출 정책에 의해서 bucket들이 비워지고, 비워진 bucket에 들어있던 높은 온도의 데이터들의 정보는 데이터 접근 샘플링 구조에 남아있게 된다. 이후 해당 데이터에 대한 새로운 접근은 데이터 접근 샘플링 구조에 남아있는 정보를 바탕으로 높은 온도의 데이터로 판단되어 캐시에 반입된다.

데이터 접근 샘플링 구조는 3.1장에서 언급한 각 캐싱 정책에 따라 데이터들의 온도 판단에 필요한 정보들을 저장한다. 간단한 LRU 정책의 경우 메모리에 4,096개(총 16KB 크기의 데이터 샘플링 구조, 4B의 LBA를 저장)의 엔트리로 이루어진 LRU 리스트를 유지한다. 이는 캐싱되는 모든 데이터의 정보를 저장하기에는 매우 부족한 크기다. 하지만 과거에 접근된 정보 중 상위 4,096개의 정보를 관리하는 역할로 활용된다. 데이터의 접근이 발생하면, 해당 데이터의 정보가 리스트에 존재하는지 확인한다. 만약 존재한다면 데이터의 정보를 MRU 위치로 옮긴다. 리스트에 존재하지 않는다면 기존의 데이터들을 LRU 방향으로 한 칸씩 이동한 후, MRU 위치에 해당 데이터의 정보를 새롭게 삽입한다. 이때, 리스트가 가득 차 있다면 LRU 위치에 저장된 정보를 제거한다. 주목할 점은 데이터 샘플링 구조에 저장되는 정보는 처리하고자 하는 요청의 데이터 자체가 아닌 데이터의 LBA다. 접근된 데이터가 캐싱 되어 있지 않아서 새롭게 들어와야 한다면, LRU 리스트에 해당 데이터의 정

보가 남아있는지 확인한다. 만약 남아있다면 높은 온도를 갖은 데이터로 분류하고, 남아있지 않다면 낮은 온도의 데이터로 분류한다. AutoStream, ARC, SLRU, LRU, DAM, MBF를 배치 정책으로 구현하여 컨벤셔널 디바이스 사용하는 기존의 Bcache와 함께 비교하였다.

또한 제한된 크기의 데이터 접근 샘플링 구조를 효과적으로 활용하기 위해서 AutoStream이 사용하는 chunk 단위 기법을 본 논문의 구조에 적용했다. 따라서, 접근되는 데이터들의 LBA를 2MB 단위로 묶어 접근 샘플링 구조에 저장하는 정보로 활용했다. Chunk 크기에 대한 영향도 실험하여 비교했다.

4.2 반출 정책

Bcache는 LRU를 기본 반출 정책으로 활용한다. 하지만 LRU 정책은 커다란 bucket 단위로 데이터들을 분류해놓은 환경에서는 적합하지 않다. 첫 번째 이유는, 각 bucket이 접근될 확률은 bucket 안에 캐싱 되어 있는 모든 데이터가 접근될 확률의 합이 되기 때문이다. 작은 단위로 데이터들을 구성한 기존의 Bcache와 달리 zone 크기에 맞춰 큰 단위로 데이터들을 구성했기 때문에 온도가 낮은 데이터를 담고 있는 bucket일지라도 접근될 확률은 상대적으로 높아진다. 따라서, 온도가 낮은 데이터만 분류된 bucket도 분류 단위가 커졌을 때는 비교적 빈번히 접근되게 된다. 두 번째는 분류된 크기가 증가한 만큼 bucket의 수가 감소하기 때문이다. 상대적으로 적은 bucket들로 구성되어 있기 때문에 bucket들 간의 경쟁이 심해진다. 그로 인해 온도가 높은 데이터들을 가진 bucket이 짧은 시간 안에 LRU에 도달할 수 있고, 높은 온도의 bucket을 한 번만 반출 하는 것으로도 심각한 적중률의 손해를 입게 된다. 세 번째는 LRU의 특성상 정확한 bucket들의 온도를 반영할 수 없다. 예를 들어 가장 온도가 낮은 bucket(리스트의 LRU에 위치한 bucket)도 단 한 번의 접근으로 가장 높은 온도의 bucket이 된다(리스트의 MRU에 위치한 bucket). 반면에 LFU 정책은 이러한 환경에 적합하다. Bucket들이 직접 접근된 횟수를 반영하기 때문에 온도가 낮은

bucket이 적은 횟수의 접근만으로 높은 온도의 bucket으로 판단되는 것을 방지할 수 있다. Bcache에서는 다양한 반출 정책을 구현하기 용이하도록 우선순위 값을 이용하여 반출 대상을 선택한다. 우선순위 값은 bucket들 마다 관리되며, 기존의 Bcache가 사용하던 LRU 정책 또한 우선순위 값을 활용하여 구현되었다. 이 우선순위 변수를 활용하여 bucket들의 접근 횟수를 관리하면 추가적인 메모리를 사용하지 않고 LFU 정책을 구현할 수 있다.

제 5 장 실험 결과 및 분석

5.1 실험 환경

실험에 사용된 시스템의 사양은 다음의 표5.1과 같다. 정확한 실험 환경을 설정하고, 플래시 메모리 디바이스 내부의 동작을 살펴보기 위해 메모리 위에서 플래시 메모리 디바이스를 에뮬레이션해 주는 Virt를 이용하여 실험했다. Virt는 컨벤셔널 디바이스와 ZNS 디바이스 모두 에뮬레이션이 가능하다. 캐싱 디바이스와 실험에 사용되는 총데이터양의 비율은 1:8로 설정했다. 총 저장되는 데이터의 양은 64GB 고 캐싱 디바이스의 크기는 8GB다. ZNS 디바이스의 경우 작은 zone 환경을 사용했다. 한 zone의 크기는 128MB고, 총 64개의 zone으로 이루어져 있다. ZNS 디바이스를 사용하는 Bcache가 zone 크기에 맞춰 커진 bucket을 사용할 수 있도록 수정하여 활용했다.

표 5.1: 실험 환경

Processor	Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz
Memory	256GB DRAM, 1TB Optane P4800X SSD Memory Mode
OS	Ubuntu 20.04.2 LTS (kernel v5.10.37)
HDD(Backing Device)	Seagate BarraCuda ST2000DM006-2DM1 2TB

5.2 YCSB 벤치마크

본 논문에서는 개선된 Bcache의 읽기 성능에 초점을 두고 실험했다. 따라서 YCSB의 다양한 워크로드 중 YCSB-C(읽기 전용)를 이용하여 실험을 진행했다. 캐

싱 디바이스의 크기와 실제 저장되는 모든 데이터 크기 간의 1:8 비율을 맞추기 위해서 16B 크기의 키와 1KB 크기의 벨류를 갖는 키-벨류 쌍 6,400,000개를 사용하여 실험을 진행했다. 읽기 실험을 진행하기에 앞서서 6,400,000개의 데이터를 Bcache 디바이스에 저장한 후, 8개의 스레드가 각각 800,000개의 데이터를 무작위로 읽는 것으로 실험을 진행했다. 또한, direct 옵션을 주어 애플리케이션 자체의 캐시로 인해 Bcache의 성능이 영향받지 않도록 했다. 실험에 사용된 워크로드는 중복이 허용되기 때문에 같은 데이터가 여러 번 읽힐 수 있고, 각 데이터가 접근되는 빈도는 Zipfian 분포를 갖고 있어 데이터들 사이에서 지역성이 존재하도록 구성했다. Bcache의 경우 4MB보다 큰 연속 접근에 대해서는 플래시 캐시를 거치지 않고 백킹 디바이스인 하드 디스크에서 바로 수행한다. 따라서, 초기에 6,400,000개의 데이터를 채우는 과정은 플래시 캐시에 캐싱 되지 않는다.

5.3 캐싱 정책 별 성능

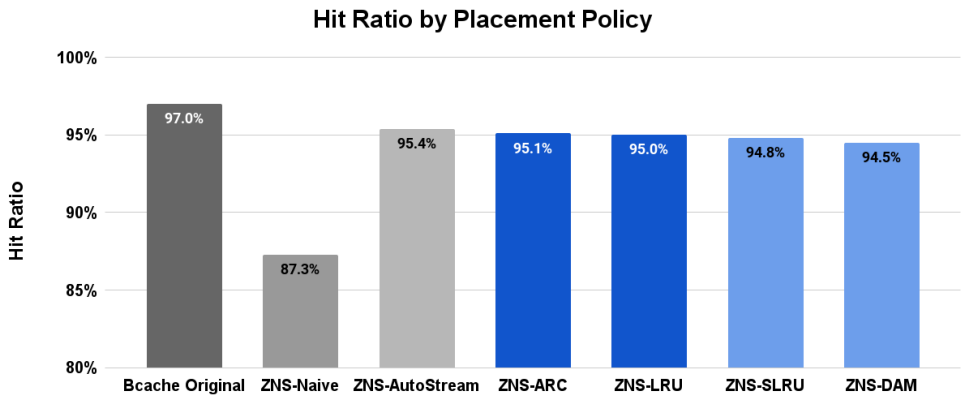

그림 5.1: 캐싱 정책에 따른 적중률 비교

그림5.1은 3.1장에서 언급한 각 캐싱 정책을 배치 정책으로 사용했을 때의 적중

률을 비교한 결과다. 컨벤셔널 디바이스를 캐싱 디바이스로 사용하는 Bcache Original의 경우, Bcache가 기존에 사용하던 LRU 정책을 반출 정책으로 사용하였고, ZNS 디바이스를 캐싱 디바이스로 사용하는 나머지 정책들의 경우 LFU 정책을 반출 정책으로 사용하였다. Bcache Original과 ZNS-AutoStream을 제외한 나머지 정책들은 본 논문에서 주장하는 데이터 온도 예측 기법을 사용하고 있고, 16KB 크기의 데이터 접근 샘플링 구조를 사용한다. ZNS-AutoStream의 경우 알고리즘의 동작 방식이 모든 LBA에 대해서 정보를 관리하기 때문에 메모리 제한을 두지 않았다. ZNS-AutoStream은 실험에 사용되는 모든 데이터에 대해서 접근 횟수를 전부 저장하고, 이를 바탕으로 데이터들의 온도를 판단한다. 따라서 과거의 접근 패턴으로 데이터의 온도를 예측하는 나머지 기법들의 상한선 근사치를 제공하기 위한 기법으로 활용했다.

실험 결과를 살펴보면 컨벤셔널 디바이스를 사용하는 Bcache Original이 97%로 가장 높은 적중률을 보인다. 그 이유는 Bcache Original은 매우 작은 반출 단위를 갖고 있어 반출 시점에 손실되는 데이터의 양이 매우 적기 때문이다. 하지만 다른 정책들과 비교했을 때 매우 근사한 차이를 보이고, 적은 적중률 이점을 얻기 위해서 극심한 쓰기 증폭을 감수해야 한다는 문제점이 있다. 이는 디바이스 수명 단축을 심화시켜 캐싱 시스템의 유지 비용을 증가시킨다.

그에 반해 ZNS 디바이스를 캐싱 디바이스로 사용하는 나머지 정책들의 경우 쓰기 증폭 문제를 완전히 해결한다. ZNS 디바이스를 사용하는 Bcache 정책들은 Bucket을 단순히 zone에 대응시키는 방식인 ZNS-Naive에 대비해서 적중률이 매우 높게 측정된 것을 확인할 수 있다. 이는 데이터의 온도를 예측하여 분류하는 배치 정책의 필요성을 뒷받침한다. 배치 정책을 통해 데이터들이 온도에 따라 bucket에 분류되고, 반출 시점에 높은 온도의 데이터들이 커진 반출 단위에 포함되어 함께 반출되는 것이 방지된다. 결과적으로 반출 시점마다 Bcache Original에 비해서 128배 높은 데이터의 손실이 발생함에도 매우 제한된 적중률 손실만을 겪는다.

ZNS-ARC와 ZNS-LRU는 데이터들의 온도를 두 단계로 단순하게 나누는 정책

이고, ZNS-SLRU와 ZNS-DAM은 데이터들의 온도를 여러 단계(3단계 이상)로 자세하게 나누는 정책이다. 실험 결과에서 볼 수 있듯이 데이터의 온도를 단순하게 나누는 정책이 자세하게 나누는 정책보다 더 높은 성능을 보이는 것을 알 수 있다. 이러한 현상이 발생한 이유는 매우 제한된 메모리 안에서 효과적으로 온도가 높은 데이터들의 정보를 관리해야 하기 때문이다. 데이터의 온도를 여러 단계로 관리하는 정책들의 경우 16KB의 데이터 접근 샘플링 구조 안에서 각 레벨의 데이터 간의 경쟁이 치열해져 정확한 데이터의 온도를 판단하기 어렵다. 또한, 매우 높은 온도의 데이터들은 전체 데이터 중 극히 일부만을 차지하고, 이러한 데이터에게 할당된 bucket의 활용성이 떨어지게 된다. 데이터 접근 샘플링 구조의 크기를 증가시켜 다수의 레벨 간 경쟁을 완화할 수 있지만 자연히 메모리 사용량이 증가하는 문제가 야기된다. 결과적으로 여러 레벨로 데이터를 나누는 정책 대비, 적은 메모리의 공간을 효과적으로 활용하여 정확하게 높은 온도의 데이터들을 판단할 수 있는 두 레벨 정책이 적합하다.

5.4 Chunk 크기 별 성능

캐싱 디바이스를 사용하는 제한된 데이터 접근 샘플링 구조를 효과적으로 활용하기 위해서 접근되는 LBA를 2MB chunk 단위로 묶어 활용했다. 이는 데이터 온도를 판단하기 위해 관리하여야 하는 정보의 양을 현저히 줄일 수 있어 매우 효과적이다. LBA의 양은 Bcache 시스템에 사용되는 백킹 디바이스의 크기에 따라 달라지고, 백킹 디바이스가 커질수록 데이터 접근 샘플링 구조에서 다뤄야 하는 LBA의 양이 증가한다. Chunk 크기를 줄이면 데이터 묶음의 수가 많아지고, LBA의 수가 증가하는 효과가 발생한다. 따라서 chunk 크기를 2MB에서 16KB로 줄여 128배 많아진 LBA에 대한 대응 실험을 진행했다. 그림5.2에서 파란색은 2MB chunk를 사용하는 결과이고, 붉은색은 16KB chunk를 사용하는 결과다. 실험에서 볼 수 있듯이 두 단계 레벨로 나누는 정책이 여러 단계로 나누는 정책에 비해 많아진 LBA에 더

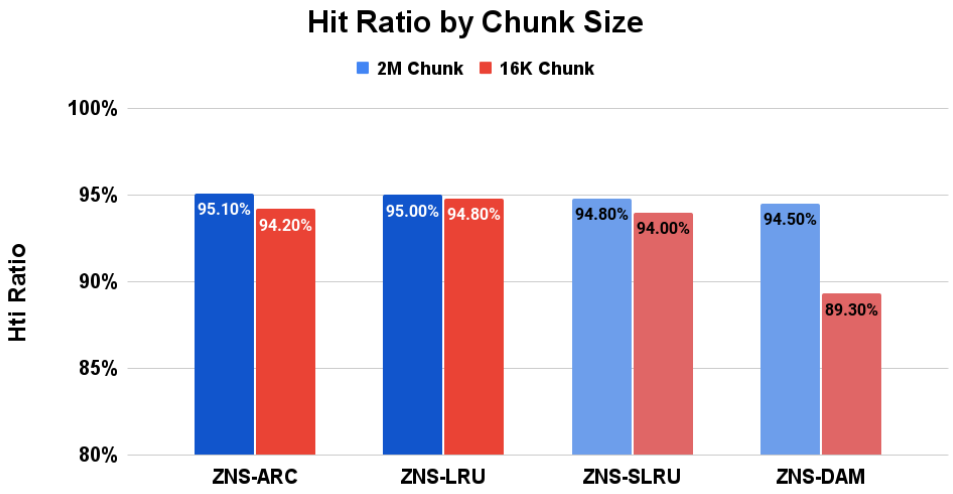

그림 5.2: Chunk 크기에 따른 적중률 비교

효과적으로 대응하는 것을 볼 수 있다. 여러 단계로 나누는 정책의 경우 더욱 많아진 LBA로 인해 각 레벨의 데이터 간 경쟁이 더욱 치열해져 적중률의 하락이 심화한다. 그에 반해 두 단계 레벨로 나누는 정책들은 아주 적은 적중률 하락만을 보이는 것을 확인할 수 있다.

5.5 LFU 반출 정책 성능

그림5.3는 기존의 Bcache가 사용하는 LRU 반출 정책 대비 LFU 반출 정책의 효과를 비교한 결과다. 단순히 Bucket을 zone에 대응시킨 ZNS-naive와 LRU를 배치 정책으로 사용하는 ZNS-LRU에 대해서 반출 정책으로 LRU를 사용했을 때와 LFU를 사용했을 때를 비교했다. 실험 결과에서 볼 수 있듯이 두 경우 모두 LFU 정책을 사용하는 것이 LRU 정책을 사용하는 것에 비해서 성능이 더 향상된 것을 확인할 수 있다. 이는 커진 반출 단위로 인해 LRU 정책은 bucket들의 온도 판단을 정확히

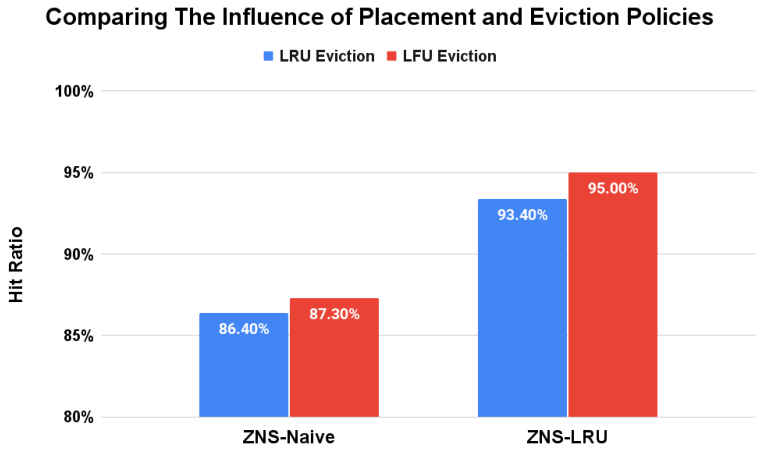

그림 5.3: LRU와 LFU 반출 정책 별 성능 비교

수행할 수 없고 LFU 정책이 적합하다는 것을 보여준다.

제 6 장 결론

본 연구에서는 기존의 Bcache가 가진 쓰기 증폭 문제를 해결하기 위해서 ZNS 디바이스를 사용하는 기법에 대해 살펴보았다. Bcache의 동작 형태가 ZNS와 매우 좋은 궁합을 보이지만, zone의 크기에 맞춰서 반출의 단위 또한 같이 증가하여 다양한 문제가 발생한다. 본 논문에서는 이러한 문제들을 정의하고 발생하는 원인을 분석하여 새로운 캐싱 구조를 제시했다. 데이터들 온도의 예측을 위한 데이터 접근 샘플링 구조를 활용하고, 다양한 캐싱 정책들을 비교하여 가장 효과적인 정책을 제안했다. 또한, 기존의 LRU 반출 정책의 부적합성을 밝히고 LFU 반출 정책의 효과성을 보였다. 이를 통해 약간의 적중률을 희생하는 것으로 쓰기 증폭 문제를 완전히 제거할 수 있었다.

참고 문헌

[1] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. The {CacheLib} caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768, 2020.

[2] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. {AdaptSize}: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 483–498, 2017.

[3] Janki Bhimani, Ningfang Mi, and Bo Sheng. Bloomstream: Data temperature identification for flash based memory storage using bloom filters. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 237–244. IEEE, 2018.

[4] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)*, 13(4):1–31, 2017.

[5] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 65–78, 2019.

[6] Nicolas Gast and Benny Van Houdt. Transient and steady-state regime of a family of list-based cache replacement algorithms. In *Proceedings of the 2015 ACM*

SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, pages 123–136, 2015.

[7] Jen-Wei Hsieh, Tei-Wei Kuo, and Li-Pin Chang. Efficient identification of hot data for flash memory storage systems. *ACM Transactions on Storage (TOS)*, 2(1):22–40, 2006.

[8] Sai Huang, Qingsong Wei, Dan Feng, Jianxi Chen, and Cheng Chen. Improving flash-based disk cache with lazy adaptive replacement. *ACM Transactions on Storage (TOS)*, 12(2):1–24, 2016.

[9] Yichen Jia, Zili Shao, and Feng Chen. Slimcache: An efficient data compression scheme for flash-based key-value caching. *ACM Transactions on Storage (TOS)*, 16(2):1–34, 2020.

[10] Taeho Kgil and Trevor Mudge. Flashcache: A nand flash memory file cache for low power web servers. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, page 103–112, New York, NY, USA, 2006. Association for Computing Machinery.

[11] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers*, 50(12):1352–1361, 2001.

[12] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A {Capacity-Optimized}{SSD} cache for primary storage. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 501–512, 2014.

[13] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S Berger, Nathan Beckmann, and Gregory R Ganger. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 243–262, 2021.

[14] Nimrod Megiddo and Dharmendra S Modha. {ARC}: A {Self-Tuning}, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, 2003.

[15] Dongchul Park and David HC Du. Hot data identification for flash-based storage systems using multiple bloom filters. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11. IEEE, 2011.

[16] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. Cflru: a replacement algorithm for flash memory. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 234–241, 2006.

[17] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, page 1–15, New York, NY, USA, 1991. Association for Computing Machinery.

[18] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. Didacache: an integration of device and application for flash-based key-value caching. *ACM Transactions on Storage (TOS)*, 14(3):1–32, 2018.

[19] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. Eelru: simple and effective adaptive page replacement. *ACM SIGMETRICS Performance Evaluation Review*, 27(1):122–133, 1999.

[20] Zhenyu Song, Daniel S Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation*, 2020.

[21] Kefei Wang and Feng Chen. Cascade mapping: Optimizing memory efficiency for flash-based key-value caching. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 464–476, 2018.

[22] Qiuping Wang, Jinhong Li, Wen Xia, Erik Kruus, Biplob Debnath, and Patrick PC Lee. Austere flash caching with deduplication and compression. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 713–726, 2020.

[23] Suzhen Wu, Yanping Lin, Bo Mao, and Hong Jiang. Gcar: Garbage collection aware cache management with improved performance for flash-based ssds. In *Proceedings of the 2016 International Conference on Supercomputing*, pages 1–12, 2016.

[24] Jingpei Yang, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. Autostream: automatic stream management for multi-streamed ssds. In *Proceedings of the 10th ACM International Systems and Storage Conference*, pages 1–11, 2017.

[25] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. CacheSack: Admission optimization for google datacenter flash

caches. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1021–1036, Carlsbad, CA, July 2022. USENIX Association.

ABSTRACT

A Block-layer Flash Cache for ZNS SSDs

Junseok Lee
Department of Computer Science & Engineering
The Graduate School
Seoul National University

Flash memories outperform HDDs. However, Flash memories are more expensive than HDDs. So, much research tries to use flash memory as caching device of HDD. But the characteristics of flash memory occurs Write Amplification Problems while using Flash memory as caching device. Bcache, the module of the Linux Kernel, provides caching techniques for using block device as cache. In contrast, the workloads that Bcache generates worsening write amplification problem while using flash memory as caching device. Using ZNS SSD which supports new ZNS interface as caching device can solve the write amplification problem. However, in order to use the ZNS SSD as a caching device for Bcache, the erase unit increases to the size of the zone and causes damage to the hit ratio. Therefore, I propose two caching policies, Placement and Eviction, which greatly limit hit ratio damage. The proposed technique can solve the write amplification problem by sacrificing a small hit ratio.

Keywords: Bcache, ZNS, Block Level Cache, Flash Cache, Caching Policy, Hot/Cold Separation

Student Number: 2021-24096