1. HDD 는 무엇이고 HDD 가 느린 이유는?? SMR Drive 는 무엇인가??

=> 사진 +

=> Access Time 분석 + Data Transfer Rate

=> 측정: IOPS

=> SMR(Shingled Magnetic Recording) 드라이브의 원리

하드디스크의 Write Head 에 비해서 Read Head 가 훨씬 크다는 점을 이용해서 READ_HEAD_SIZE 만 남기고 나머지 데이터는 겹쳐지도록 데이터를 저장하는 기술

이를 통해서 HDD 용량을 할 수 있다

반드시 맨처름 위치에서부터 log-structured data structure 처럼 순차적으로 기록해 나가야 한다

중간 위치를 수정하고 싶으면 주변 트랙도 전부 다 다시 써야 한다 (Read-Modiy-Write)

=> SMR 구현 방식

Drive Managed:

호스트 시스템은 디바이스의 내부 구조를 알지 못한다

Backward Compatible

Host Based:

디바이스를 제어하기 위한 명령어가 호스트 시스템 커널에 추가되어야 한다

Not Backward Compatible

2. Indirection System for Shingled-Recording Disk Drive

=> 디바이스 전체를 블록 단위로 나누어 어떤 블록이 hot 해서 자주 접근되고 수정되는 지를 확인해서 따로 관리하면 성능을 높일 수 있다는 생각에서 출발한다

기존의 host - device 레이어 사이에 hot 블록을 검출하는 알고리즘을 도입하고 "Indirection System" 이라고 명명

순차적으로 계속 뒤에다 적어나가는 것도 하나의 방법이겠지만 그것보다 효율적으로 데이터를 관리해야 한다

이는 Drive-Managed 방식을 채택한 것이다.

PBA 영역 전체를 Temporary(cache) vs Permanent(native) 로 나눠서 매핑한다.

[Set-Associative Disk Cache Architecture]

example mapping figure, algorithm pseudo code figure

[Shingled Access with S-Blocks]

캐시로 사용하는 E-region 이 워낙에 적어서 금세 full 에 이르기 때문에 디바이스에 Latency 가 증가한다. 이를 극복하기 위해 cache 와 native 사이에 S-Blocks 이라는 중간 단계를 추가한다

circular buffer

1) E-region 이 일정한 값을 넘어서 꽉 찼을 때 -> E-region 에 invalid 블록 제거

-> invalid 블록이 하나도 없으면 S-Blocks 에서 invalid 블록 제거

-> 간헐적으로 S-Blocks 에 E-region 데이터를 동기화

2) S-Blocks 이 일정한 값을 넘어서 꽉 찼을 때 -> S-Blocks 에서 invalid 블록 제거

-> invalid 블록이 하나도 없으면 native 에 S-Blocks 의 데이터를 동기화

=> 여기서 고민해봐야할 문제

1. How Many S-Blocks to destage before cache-buffer defrag?

S-Blocks 를 자주 destage 하면 캐시 버퍼에서 앞쪽에 접근된 블록을 invalid 로 만들기 때문에 GC 프로세스가 짧아져서 좋다

2. How to choose the S-Blocks for group destage?

여러 가지 방식이 가능하다

(현재 버퍼에 가장 많은 블록, 캐시 버퍼의 tail 에 가장 가까운 블록)

3. How many invalid blocks/S-Blocks to reclaim during buffer defrag?

1 번과 다르게 defrag 를 미룰수록 중간에 invalid 되는 블록이 많아질 것이다.

3. H-SWD: Hot Data Identification into Shingled Write Disks
=> 이 논문은 "Indirection System"에 적용되는 GC 알고리즘의 성능을 높이는 것이 주된 목표이다.
=> 아키텍쳐는 (FIg3) 디바이스를 여러 개의 섹션으로 나눴다. 각 섹션마다 내부적으로 소량의 Hot Band 와 대량의 Cold Band 로 나누고 각각 Tail 과 Head 를 지정했다.
=> 데이터 블록에 접근이 일어나면 이전에는 곧바로 기록하고 나중에 평가하는 방식이었다면 여기서는 접근이 일어나자마자 데이터 블록이 hot / cold 를 판별하고 거기에 맞춰서 밴드에 enqueue 한다.
=> GC Algorithm 4 figures (Intra, Normal, Forced, only_for_Hot_Band)

4. Hot Data Identification with Multiple Bloom Filters: Block-Level Decision vs IO Request-Level Decision
=> 기존의 "Indirection System" 이 적용된 디바이스에 대해서 I/O Request 수준에서 면밀히 분석해보니 hot 데이터와 cold 데이터에 동시에 접근하는 경우가 많다는 것을 알게 되었다. Seek Time 이 길어지고 이에 따라 response time 이 길어지는 부작용이 있음을 발견했다. 이를 극복하기 위해서 E-region 을 다시 H-region / C-region 으로 세분화하는 방식으로 성능을 향상시키는 것이 이 논문의 핵심이다.
=> I/O Request 가 발생하면 그 즉시 Hot Data Identification Algorithm(Multiple Bloom Filter & Window-Based Direct Address Counting) 을 통해서 데이터를 판별하고, 맞춰서 H-region / C-region 에 데이터를 집어넣는다. 시간이 경과하면서 Indirection System GC 가 실행되고 cache -> S-Blocks -> native 순서를 거치며 데이터는 shingled native track 에 저장된다.