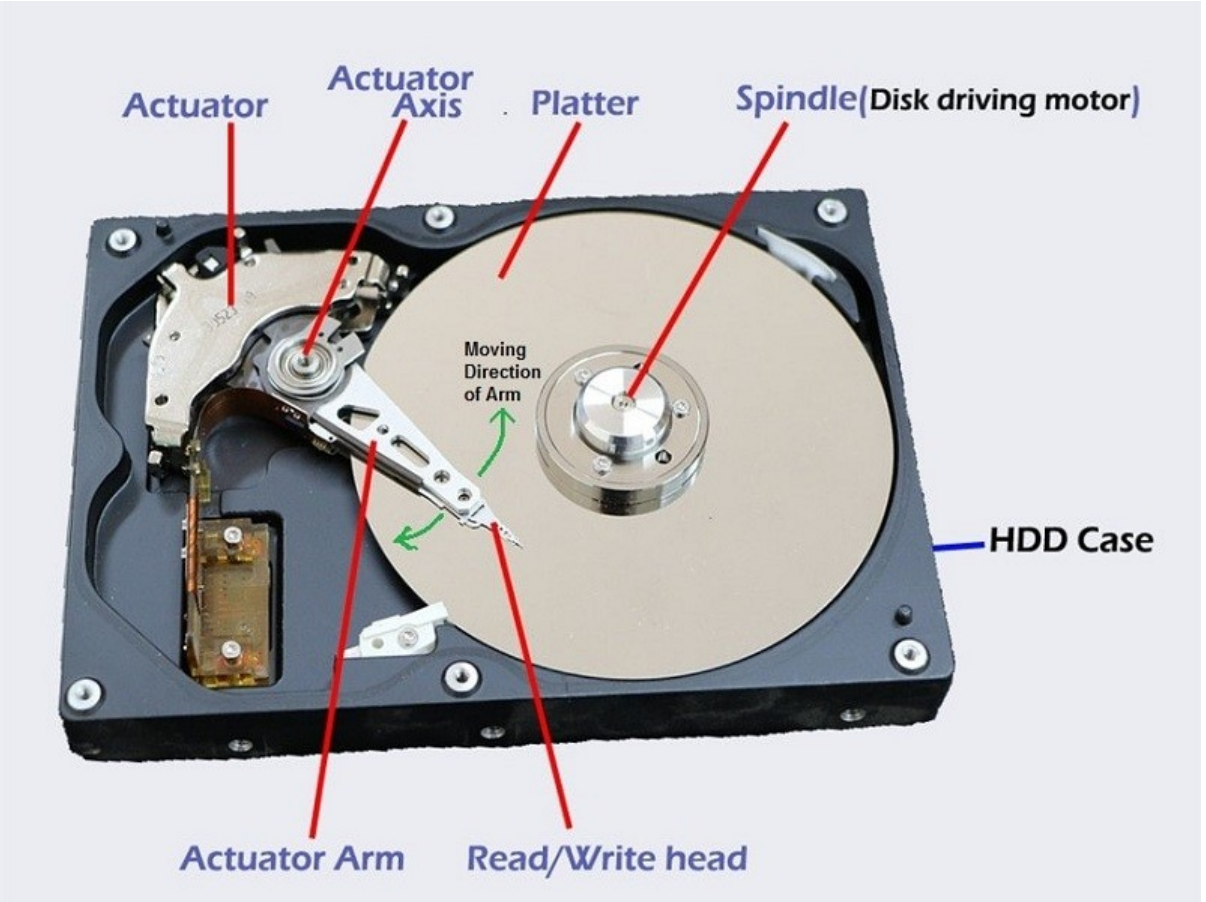
Hot Data Identification with Multiple Bloom Filters: Block-Level Decision vs I/O Request-Level Decision

1. SMR(Shingled Magnetic Recording) 드라이브의 원리

2. SMR 을 어떻게 하면 잘 사용할 수 있을까 ?
   (e.g. drive managed caching mechanism for hot data identification:
   Indirection Systems for Shingled-Recording Disk Drive)

3. 본 논문 소개
   (e.g. splitting E-region into two parts – H-region and C-region)

HDD (Hard Disk Drive)



HDD (Hard Disk Drive) : why so slow?

=> Access Time 분석

1) Seek Time

헤드를 움직여서 데이터가 존재하는 트랙 위치로 움직이기 (10ms)

2) Rotational Latency

플래터를 회전시켜서 헤드 아래에 데이터가 존재하는 섹터를 가져다놓기 (6ms)

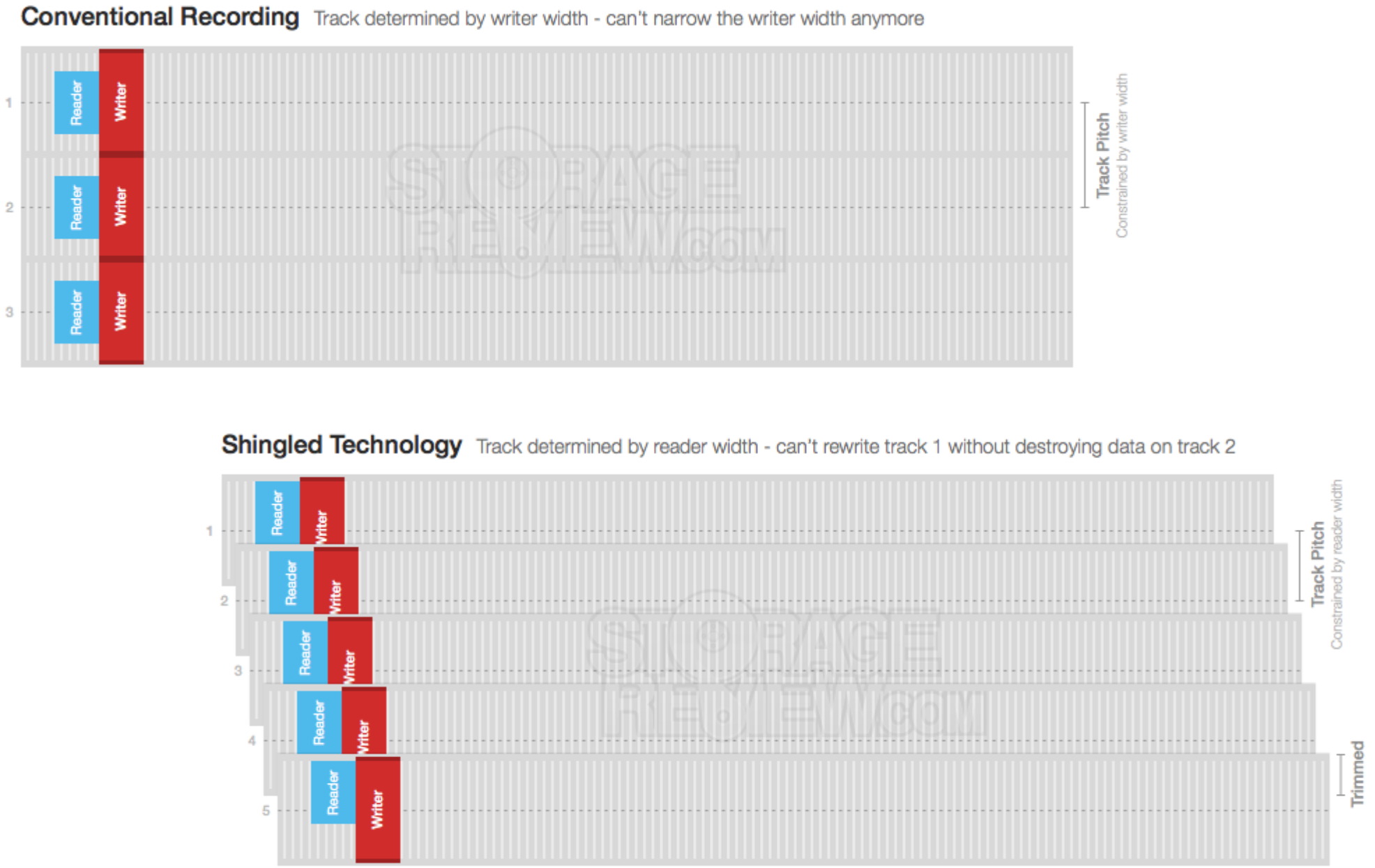
3) Command Processing Time

데이터 처리 시간 (0.0003ms)

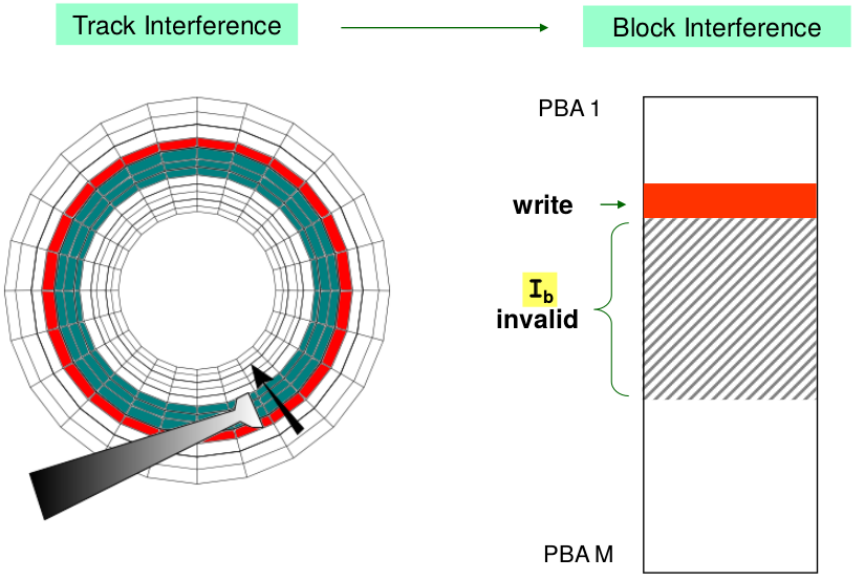
4) Settle Time

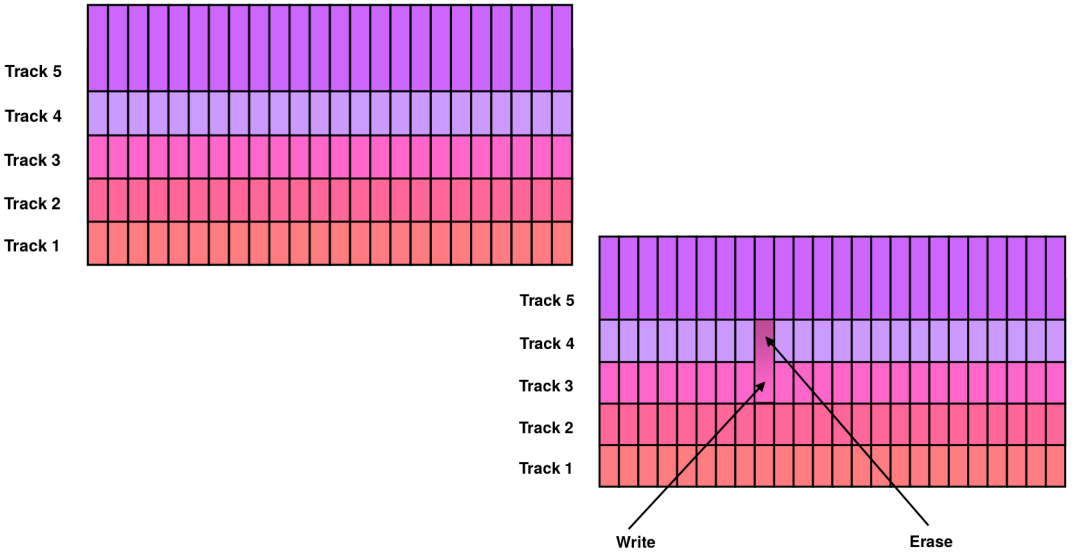
헤드를 플래터에 안착시키는데 걸리는 시간 (0.01ms)

SMR (Shingled Magnetic Recording) Drive



SMR (Shingled Magnetic Recording) Drive





SMR (Shingled Magnetic Recording) Drive

=> SMR 구현 방식

1) Drive Managed:

호스트 시스템은 디바이스의 내부 구조를 알지 못한다

Backward Compatible

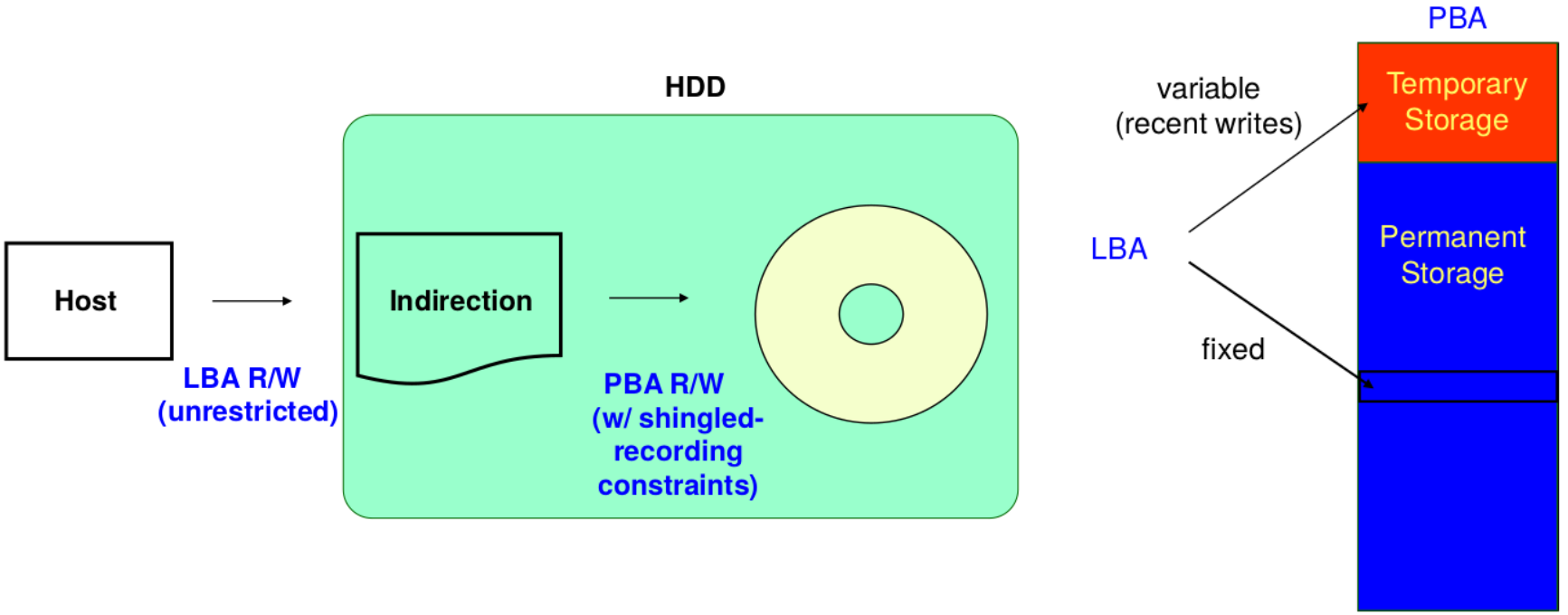
드라이브를 제어하는 기능이 드라이브 내부에 포함되어 있다 .

2) Host Based:

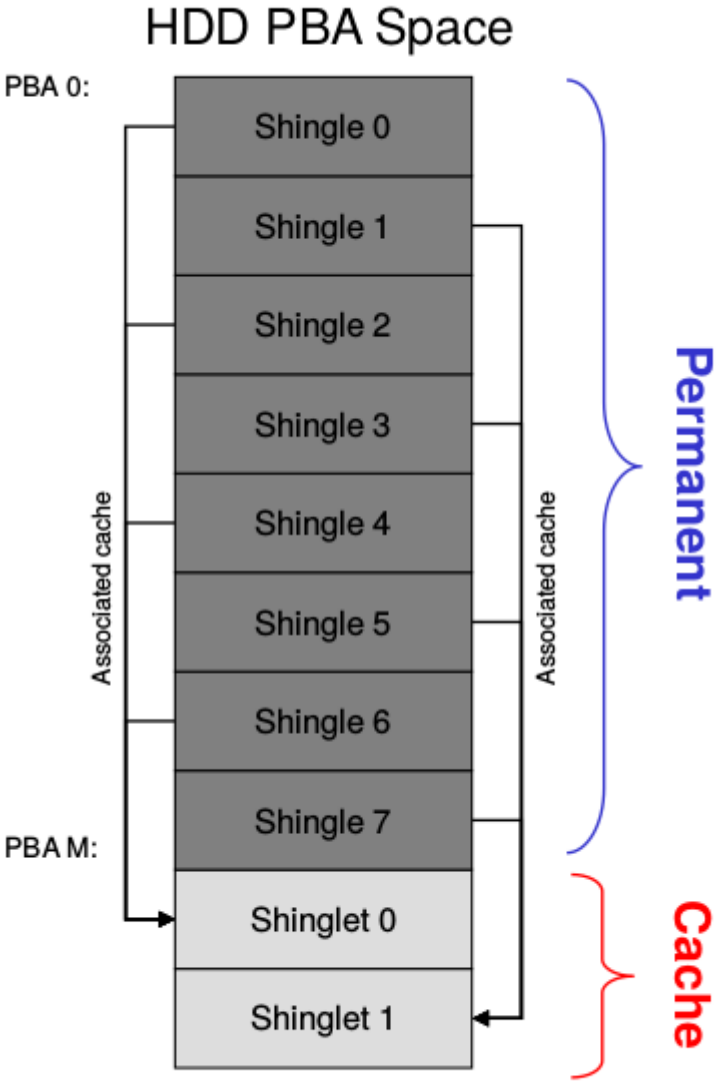
디바이스를 제어하기 위한 명령어가 호스트 시스템 커널에 추가되어야 한다

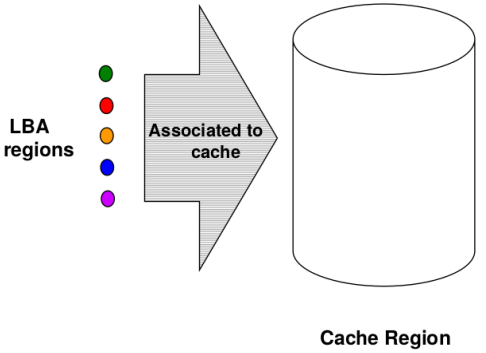
Not Backward Compatible

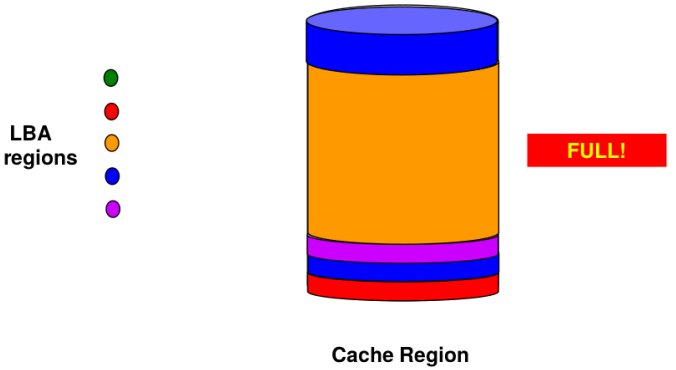
How to deploy SMR Disk Drive

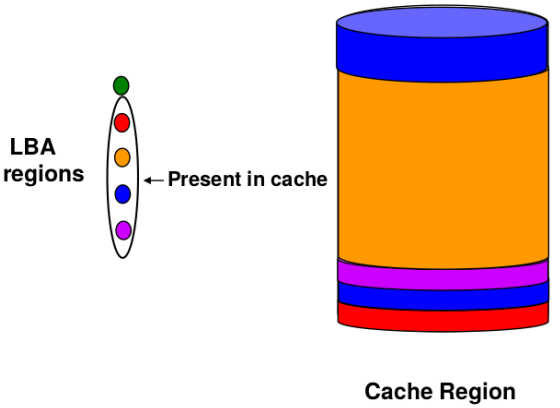


Set-Associative Cache Architecture

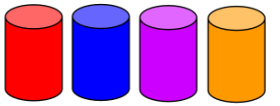








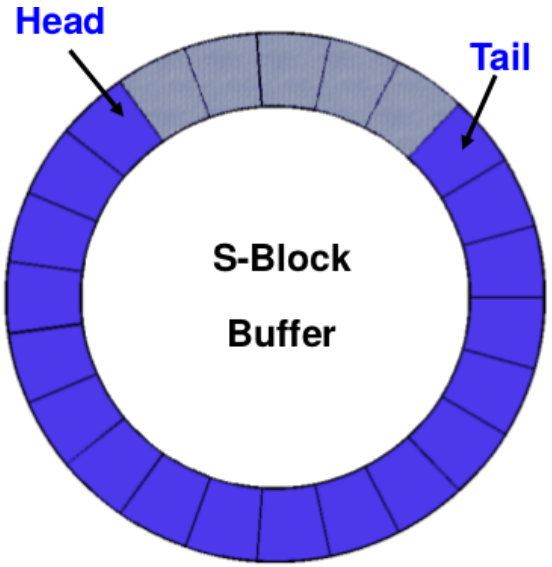
- Read cache
- For each LBA region present in cache: RMW* full region

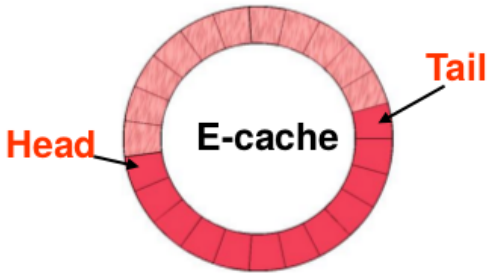


(Read+Write) x #Present

- Full cache available again

*** RMW = Read-Modify-Write**





* defragmentation

S-Blocks

- Temporary (red) and permanent (blue) storage managed as ring buffers
- S-Block: intermediate unit between sector and region

Algorithm V.1: write_block

```
Input: LBA
i = section(LBA)
if is_full(C_buff_i) then
    cache_buffer_defrag(C_buff_i)
end
Add(LBA,C_buff_i)
```

Algorithm V.2: cache_buffer_defrag

```
Input: buff
if num_invalid(buff)==0 then
    S_Blk = choose_S_block_to_destage()
    group_destage(S_Blk)
end
while is_valid(tail) do
    blk = read(tail)
    write(head,blk)
    tail = tail + 1
    head = head + 1
end
tail = tail + 1
```

Algorithm V.3: group_destage

```
Input: S_Blk
read(S_Blk)
foreach block of S_Blk present in cache buffer do
    read(block)
    invalidate(block)
end
modify_with_cached_blocks(S_Blk)
write_S_block(S_Blk)
```

Algorithm V.4: write_s_block

```
Input: S_Blk
i = section(S_Blk)
if is_full(S_buff_i) then
    S_block_buffer_defrag(S_buff_i)
end
Add(S_Blk,S_buff_i)
```

Algorithm V.5: S_block_buffer_defrag

```
Input: S_buff
while is_valid(tail) do
    S_blk = read(tail)
    write(head,S_blk)
    tail = tail + 1
    head = head + 1
end
tail = tail + 1
```

Algorithm V.6: read_block

```
Input: LBA
i = section(LBA)
PBA_c = cache_lookup(LBA,C_buff_i)
if PBA_c ≠ NOT_IN_CACHE then
    read(PBA_c)
end
else
    read_from_S_block(LBA)
end
```

Hot Data Identification with Multiple Bloom Filters: Block-Level Decision vs I/O Request-Level Decision

위에서 언급한 Hot Data Identification 방법은 다음과 같다 .
새로운 I/O Request 발생 → 해당 블록을 E-region 에 기록 → S-Blocks 로 복사
→ Native 로 복사

본 논문의 새로운 Hot Data Identification 방법은 다음과 같다 .
새로운 I/O Request 발생 → MBF 를 사용해 해당 블록의 Hot/Cold 판별
→ if (isHot) H-region 에 집어넣기 elsif (isCold) C-region 에 집어넣기
→ WDAC 를 사용해 valid 블록까지 Hot/Cold 판별
→ S-Blocks 로 복사
→ WDAC 를 사용해 valid 블록까지 Hot/Cold 판별
→ Native 로 복사

Hot Data Identification with Multiple Bloom Filters: Block-Level Decision vs I/O Request-Level Decision

원래 Block Level Decision 이었지만
I/O Request Level Decision 으로 SMR 디자인을 바꿨다 .

HDD 는 sequential read/write 속도에 비해 random read/write 속도가 현저히 낮다 .
그러므로 Hot Data Identification 실행한다는 점에도 동일할 지라도 NVM 을 사용할 때와 달리 sequential read/write 상황을 최대한 유도하는 것이 성능 확보에 유리하다 .

Block Level Decision 이라면 Block 단위로 처리하기 때문에 destage 단계에서 RANDOM ACCESS 가 발생한다 .
그러나 I/O Request Level Decision 이라면 해당 I/O Request 에서 접근한 continuous series of blocks 를 다루게 되므로 응답시간을 대폭 줄일 수 있다 .

Hot Data Identification with Multiple Bloom Filters: Block-Level Decision vs I/O Request-Level Decision

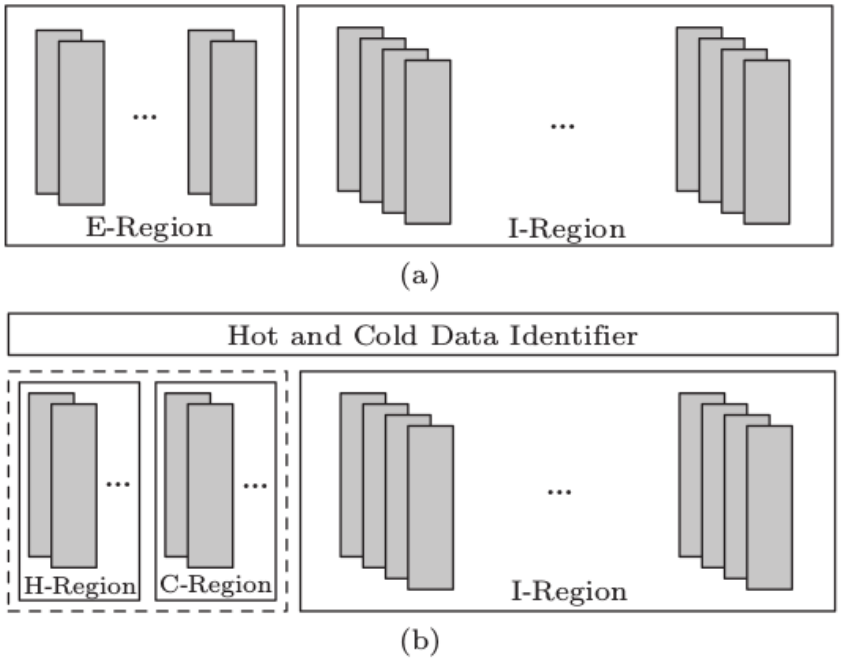


Fig.10. SMR drive layout comparison between (a) a basic SMR drive and (b) our proposed SMR drive.

SMR 드라이브 디자인을 살펴보면 GC 프로세스에서 valid 블록을 복사하는 횟수가 아주 많다 . 이 부분이 가장 큰 오버헤드로 작용한다 .

그러나 E-region 에 있는 hot / cold 데이터를 구분해서 저장할 수 있다면 H-region 에는 invalid 블록만 가득할 것이므로 GC Overhead 를 대폭 감소시킬 수 있다 .