8. Host-managed Flash

Special Topics in Computer Systems:
Modern Storage Systems
(SE820-01)

Instructor:
Prof. Sungjin Lee (sungjin.lee@dgist.ac.kr)

Today

- **Why Host-Managed Flash?**
- SDF: Software-Defined Flash
- AMF: Application-Managed Flash
- LightNVM: The Linux Open-Channel SSD Subsystem
- Reference

Why Host-Managed Flash?

- **An emerging approach to managing NAND flash, especially in data-center environments**

datacenter은 다른 일반적인 environments과 다르게
많은 부분을 전용으로 design한다.

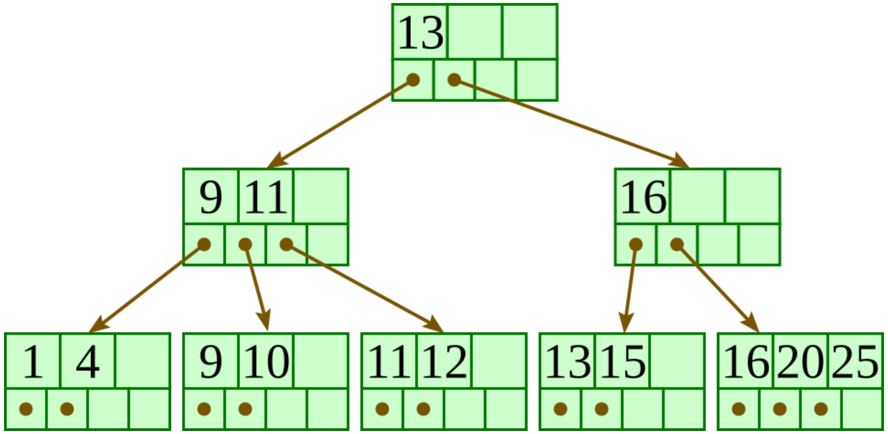
- **Why?**

1. ***Changes in data management algorithms***
   - Log-structured/COW file systems and LSM-Tree become popular
2. ***Changes in I/O access patterns***
   - I/O access patterns become mostly sequential
3. ***Changes in storage media***
   - Low latency storage media (e.g., Flash and 3D-XPoint)

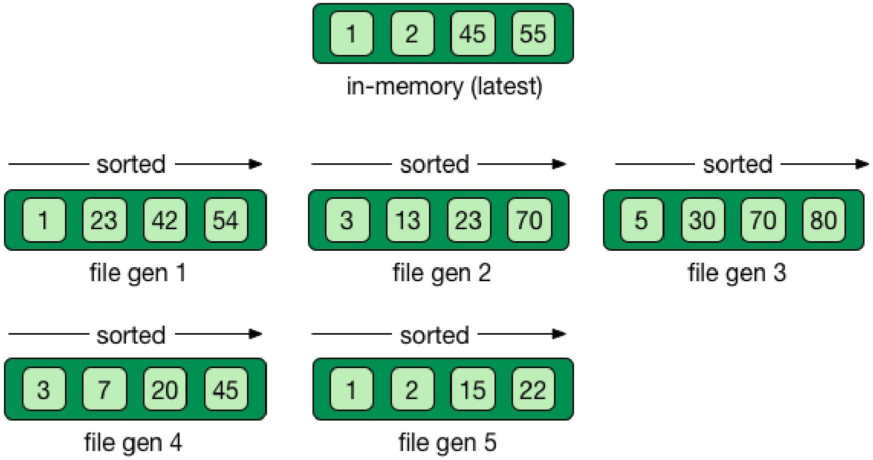
Changes in Data Mgmt. Algorithms

- **Modern web services rely upon two types of workloads for small objects**
  - *Interactive workloads*: "in-place update" storage optimized for random reads
    - B-Tree or B+ Tree algorithms
  - *Analytical workloads*: "out-of-place update" storage optimized for high write throughput and sequential reads
    - LSM-tree algorithms

많은 데이터를 저장하고 워크로드의 detail을 분석하고자 하는 경우(서치엔진 등)



B+ Tree



LSM-Tree

Changes in Data Mgmt. Algorithms (Cont.)

- **Often use two different storage management policies**
  - Fast-path processing tasks with "in-place update" storage;
  - Asynchronous analytical tasks with "out-of-place update" storage
  - Limitations
    - Take several hours for ML models to react to users' behavior
    - Force operators to manage redundant infrastructure

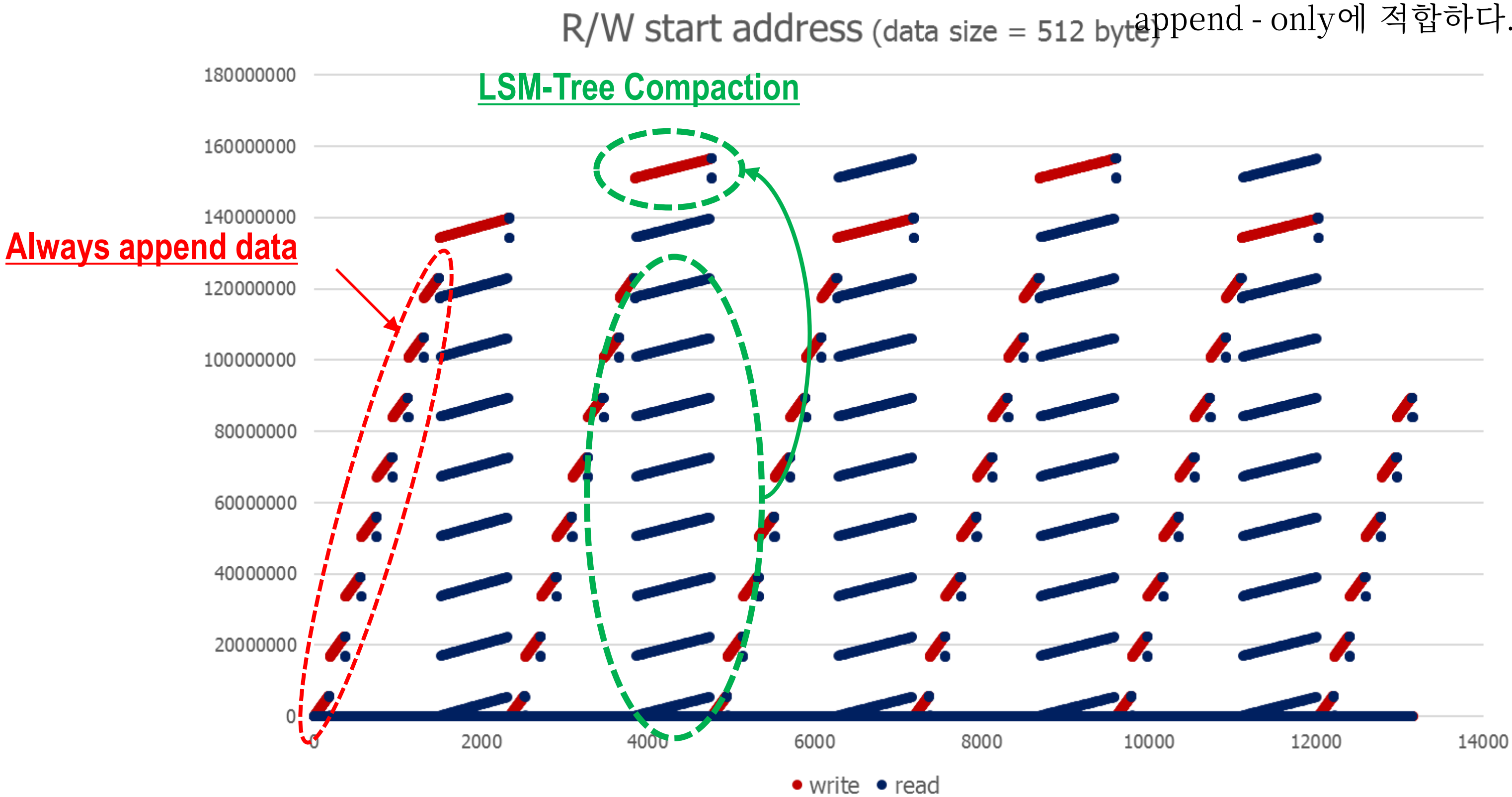
      두 가지를 다 잘 처리할 수 있는 하나의 알고리즘으로 사용해야 더 좋을 것이며
      최근에는 out - of - place update policy를 더 선호한다.
- **Hyperscale companies (e.g., Google, Yahoo, Facebook, …) *prefer* the "out-of-place update" policy**

  read가 좋은 B+ tree 대신
  write가 좋은 LSM - tree
  - Writes account for a larger portion of the total I/Os
    - For interactive workloads, 10-20% in 2010 → 50% in 2012
  - Better algorithms for LSM-Tree
    - e.g., Bloom filter, key sorting, better merge scheduling, …

Changes in I/O Access Patterns

- LSM-tree algorithm is based on an append-only write policy
- Writes sent to storage are almost sequential

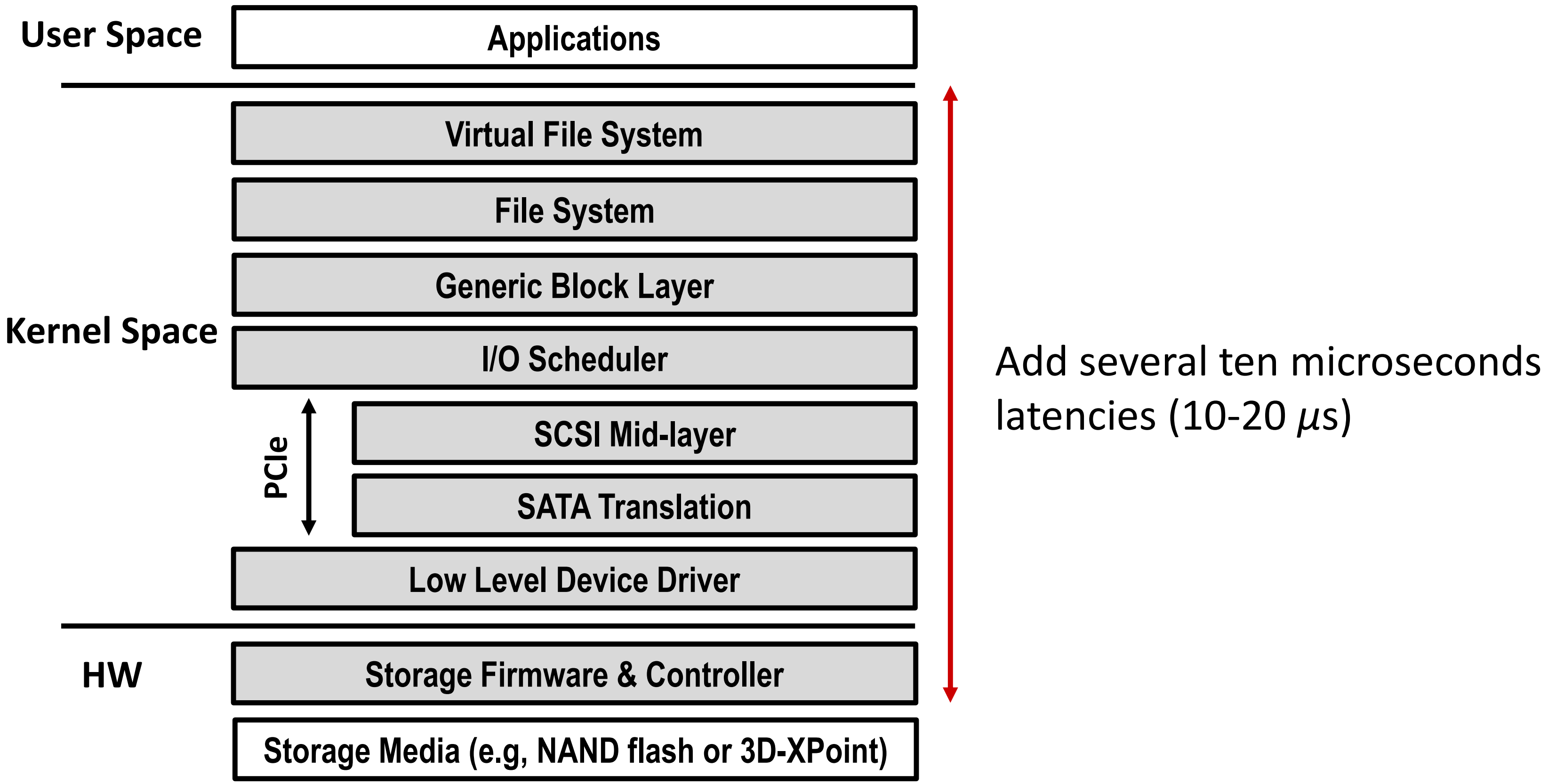
I/O Access pattern이
대부분 sequential이기에
append - only에 적합하다.



- **Suitable for append-only storage like Flash and SMR**

Changes in Storage Media

- Existing storage stacks are so complicated!!!



- Okay for slow HDDs with *millisecond* latencies
- Not okay for fast non-volatile memories with *microsecond* latencies (e.g., flash and PRAM) → Software overheads are becoming a new bottleneck

Today

- Write-optimized Storage Systems
- **SDF: Software-Defined Flash**
- AMF: Application-Managed Flash
- LightNVM: The Linux Open-Channel SSD Subsystem
- Reference

I/O Characteristics in Data Center

i/o 특성이 바뀌는 것에 주목함

- In data canters, small-write traffic to a storage system is managed by a log-structured merge (LSM) tree
  - e.g., Google's BigTable, Baidu's CCDB, Yahoo's PNUTS, HBase, Cassandra, LevelDB, SQLite, MongoDB 3.0, …

- With LSM, almost all of the write requests become sequential only with few random writes

(block SSD)

- Commodity SSDs are not designed to effectively support or to take advantage of such new workloads
  - In terms of I/O performance, capacity, and costs

Problems with Commodity SSDs

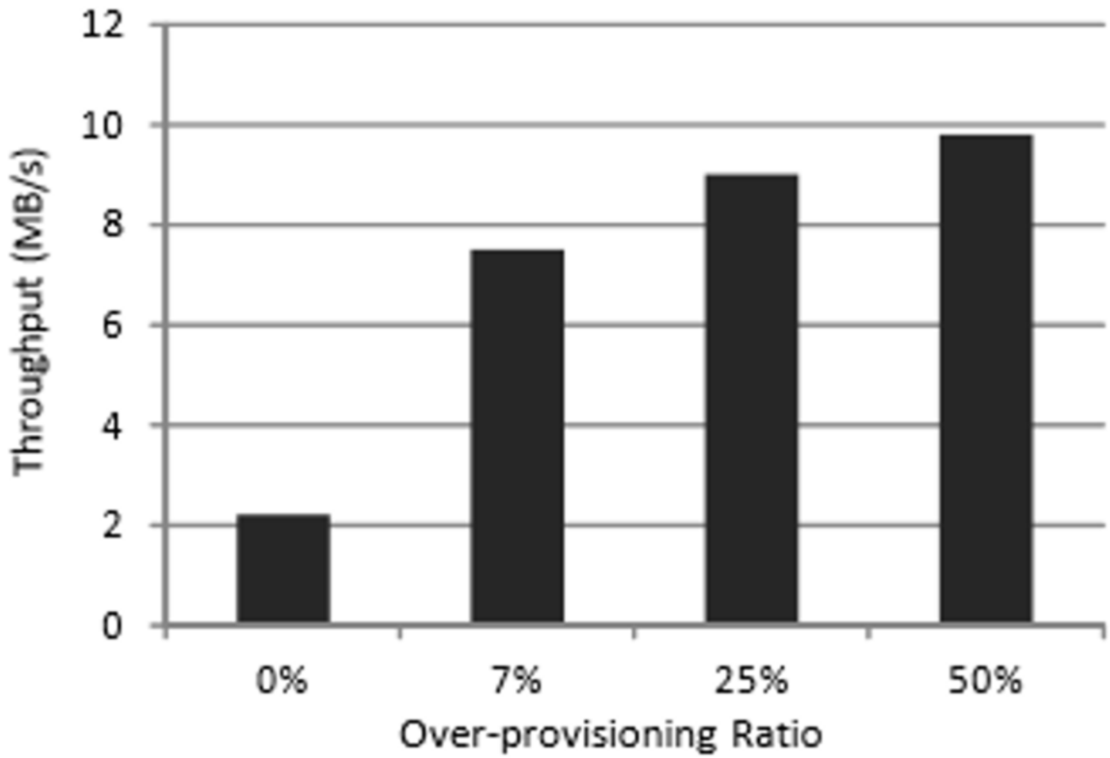
- **I/O performance**
  - The effective bandwidth received at the storage system serving real world workloads can be much lower
  - I/O requests from upper-level software (e.g., file systems and databases) are not optimized to generate SSD-friendly access patterns
- **Space efficiency**
  - A large amount of flash space is reserved for overprovisioning and is not available for storing user data

GC 성능을 위해서는 OP가 많아야 한다.
그런데 공간을 많이 소비

- **Costs**
  - With hundreds of thousands of servers, commodity SSDs have severe implications, in terms of both initial and recurring costs
  - Increase hardware costs, energy costs, physical space requirement, and maintenance costs

Over-provisioning Ratio

- Throughput of SSDs (Intel 320) when various fractions of its raw capacity are allocated as over-provisioning space



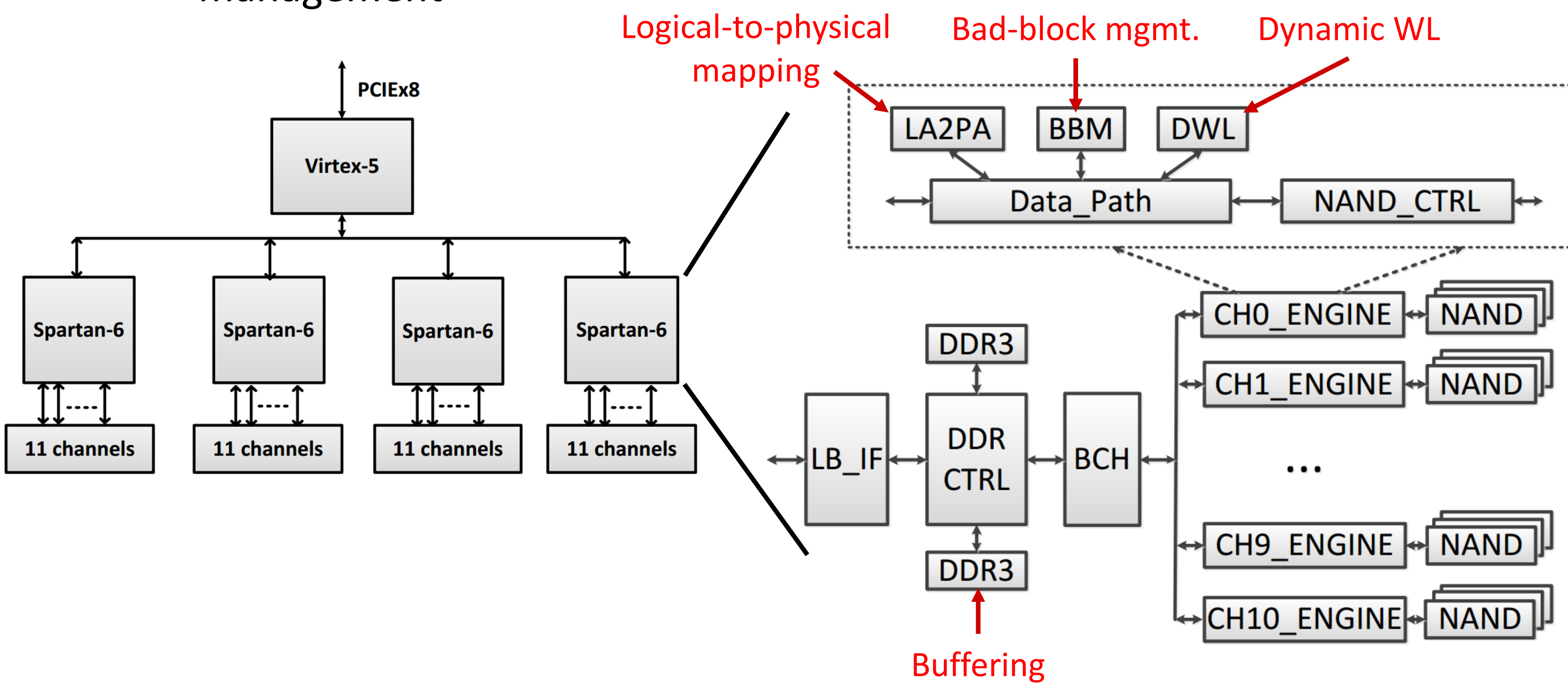
SDF: Software-Defined Flash

- Motivated by a significant workload characteristic in data centers supporting Internet-wide services
- Redesign SSDs and I/O stacks in a data center with its particular workload characteristics

- **SDF's approaches**
  - (1) Exposing the channels/ways/blocks in SSD hardware to software
    - Get rid of upper-level software
  - (2) Host software to explicitly manage SSD hardware
    - Better understand host workloads
    - Make a device simpler
  - (3) No reserved space for internal GC No OP
    - All raw flash space available for storing user data

SDF Hardware

- **Virtex-5**: PCIe DMA and chip-to-chip bridging
  - Expose all the channels to the host transparently
- **Spartan-6**: Independent FTL for each of 11 channels
  - Perform block-level mapping, dynamic wear-leveling, and bad block management



SDF Hardware (Cont.)

SDF의 하드웨어는 web - scale 스토리지 환경에 꼭 필요하다고 증명된 기능들을 포함하고 있다.

- SDF's hardware includes features and functionalities that are proven truly necessary in web-scale storage environments

- **(1) No static wear leveling**

(No swap). swap은 잘 접근되지 않았을 때 이루어지는데
HDD의 cache로서 쓰이는 만큼 잘 안 접근되는 데이터는 cache(SSD)에 있을 일이 별로 없다.

  - Mainly used as a cache for HDDs – data that is rarely accessed is not expected to reside in the cache for a long time

  → Simplify HW design and reduce I/O variations

  애초에 이 환경에서는 SSD가 HDD의 캐시로서 사용되므로, 어차피 cold data가 없다.
  따라서 static wear leveling이 필요하지 않으므로 아예 그 기능을 없애서 HW design을 단순화하고 I/O variation을 줄인다.

- **(2) No DRAM for caching**
  - Data is cached in a host DRAM – a storage device is not expected to have strong temporal locality
  - Write requests are acknowledged only after the data is stored on flash

  → No or less amount of DRAM and no battery/capacitor

어차피 host DRAM에도 데이터가 캐시되므로, SSD에는 어차피 temporal locality가 엄청 강한 데이터는 들어오지 않을 것이다. 따라서 SSD 안에서는 캐싱용 DRAM과 배터리를 뺀다.

SDF Hardware (Cont.)

parity 기반 데이터 보호 기법도 없다. host software에서 replication으로 보호한다.
따라서 하드웨어에서는 flash 자체의 error correction을 위한 BCH ECC만 한다.

- **(3) No parity-based data protection**
  - Use software-managed replication which can be done by the host software
  - Only BCH ECC for flash error correction

  → Simplify HW design and reduce I/O variations

- **(4) No over-provisioning area**
  - Since software manages SSD directly (Baidu's CCDB), there is no need for GC inside SSD

  → 20% to 50% cost reduction

OP 영역을 없앤다.
소프트웨어가 SSD를 직접 관리하니까,
굳이 GC를 위해 OP가 따로 필요하지 않다.

SDF Interface

Flash FS와 비슷

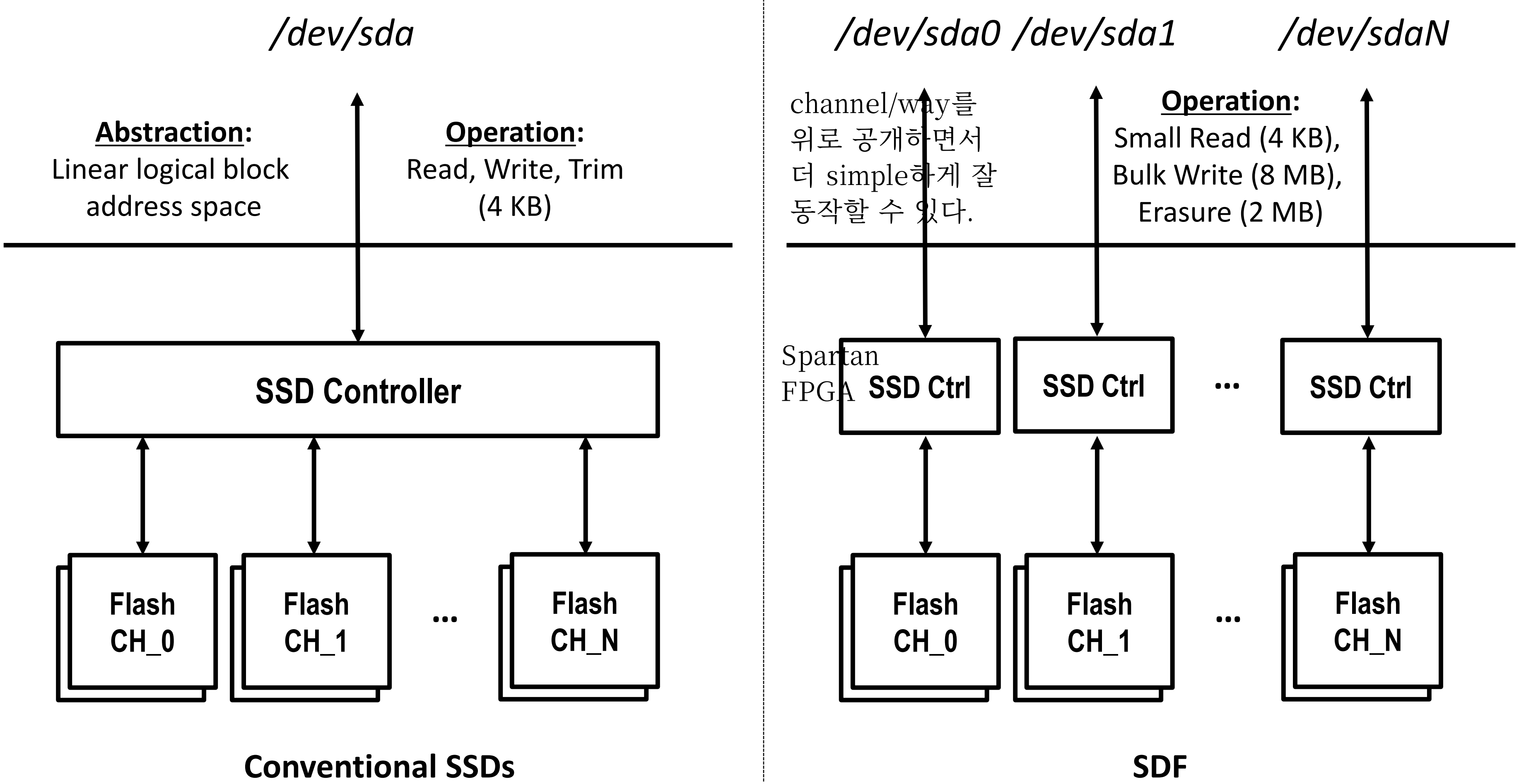
- **Expose the asymmetry in the read/write/erase operations to the software** 블록 제거도 software에 의해 명시적으로 행해진다.
  - The block erasure is explicitly done by the software
  - read unit은 작게
  - The read unit is kept small (e.g., one or a few flash pages)
  - The write unit size is set a multiple of the flash erase block size, and write addresses are required to be block-aligned
    write unit은 flash block의 몇배로 크게.
    write address 또한 block 단위 align.
    - Valid and invalid pages do not coexist in the same block
      이를 통해 한 block 안에서는 valid/invalid page가 공존할 수 없다.

- **Expose the device's internal parallelism to the software**
  - Each channel is exposed to the applications as an independent device
    - e.g., */dev/sda0* through */dev/sda43*

디바이스의 내부적인 병렬성을 소프트웨어에게 공개한다.
각 채널은 '독립적인 디바이스'로서 application에게 노출된다.

SDF Interface (Cont.)

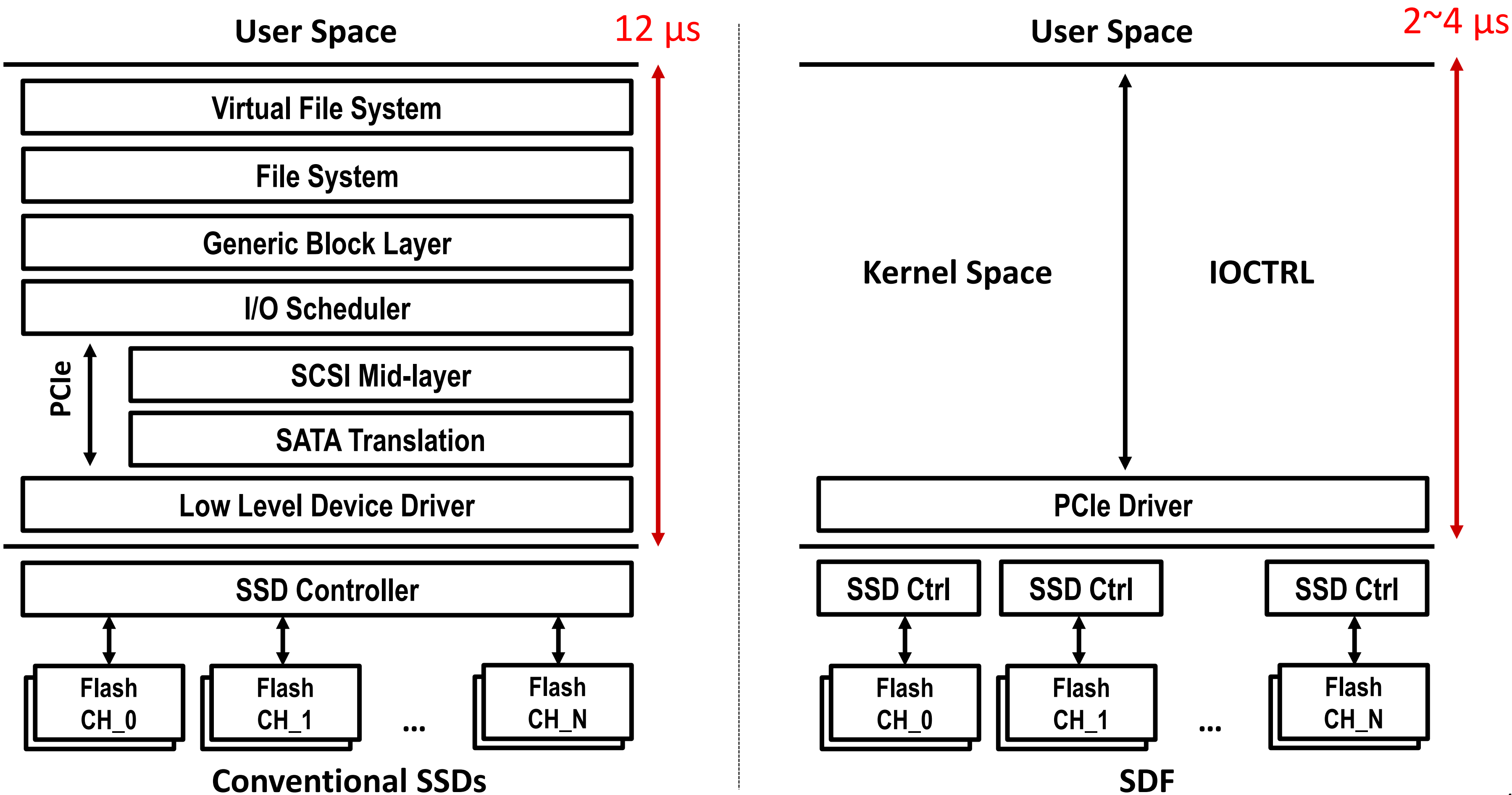
host에게는 N+1개의 SSD가
있는 것처럼 보인다.
datacenter에서는 어차피 여러 multi - threads application을
같이 수행하므로 동작 가능



SDF I/O Stack

통합된 user - space block layer를 application에게 제공하므로, 대부분의 커널 모듈을 우회할 수 있다.

- Provide a unified user-space block layer to the applications, bypassing almost all of kernel modules (e.g., file system, block layer, and so on)



SDF Data Management Systems

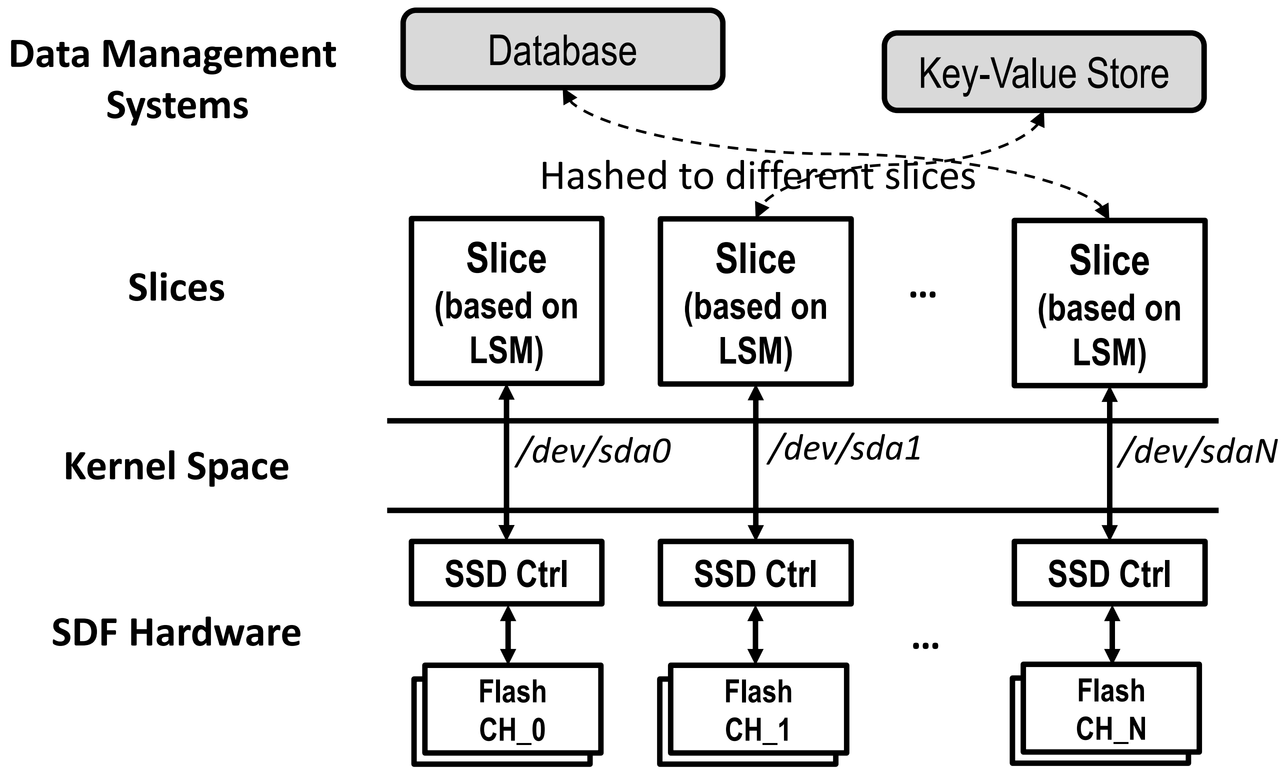
2013년 논문. 혁신적임.

다양한 데이터 관리 애플리케이션, 파일 시스템, 데이터베이스, 키 - 밸류 스토어들은 여러 slice 위에서 실행된다.

- Various data management applications, file system, database, and key-value store, run atop multiple *slices*
- Slices are based on LSM-tree algorithms and talk with SDF hardware directly

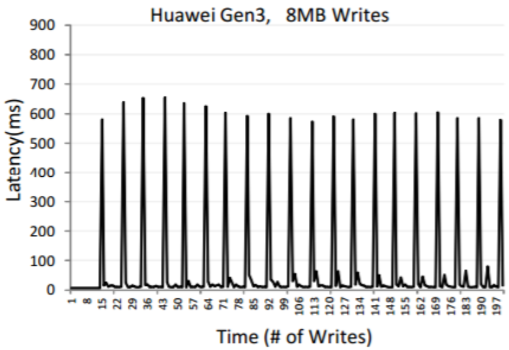
각 channel에 부착된

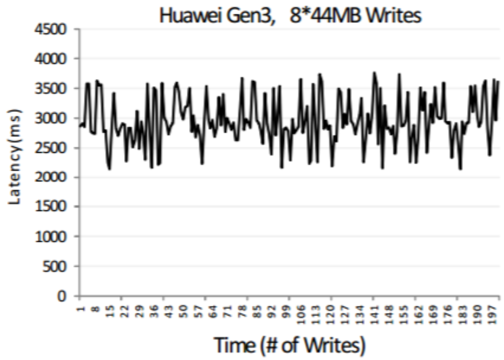
Database는 쉽게 key - value store로 변환될 수 있으니 그에 적합한 LSM 기반으로

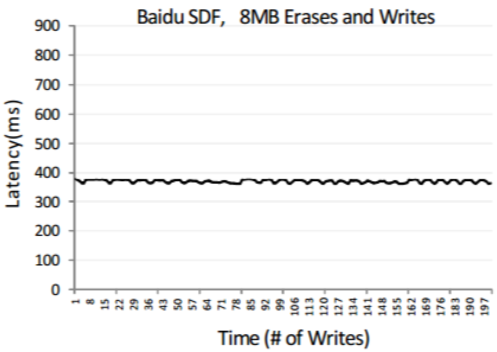


Experimental Results

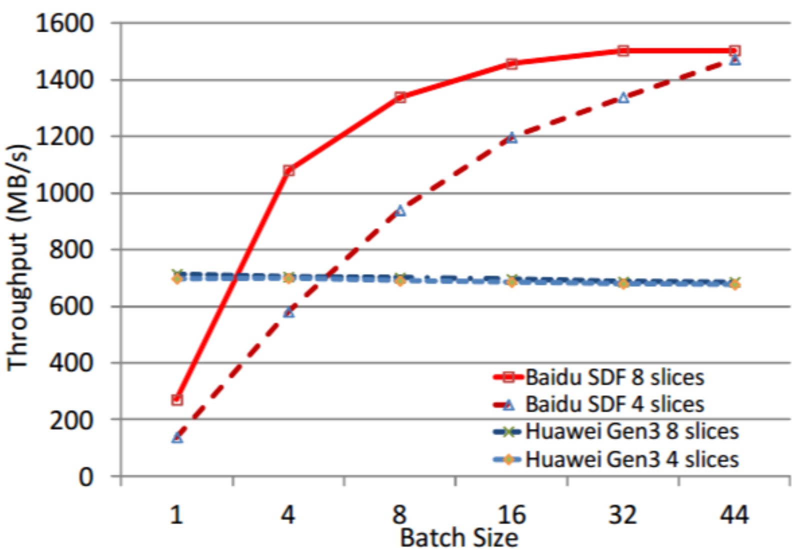
- Latencies of write requests







- I/O Throughputs

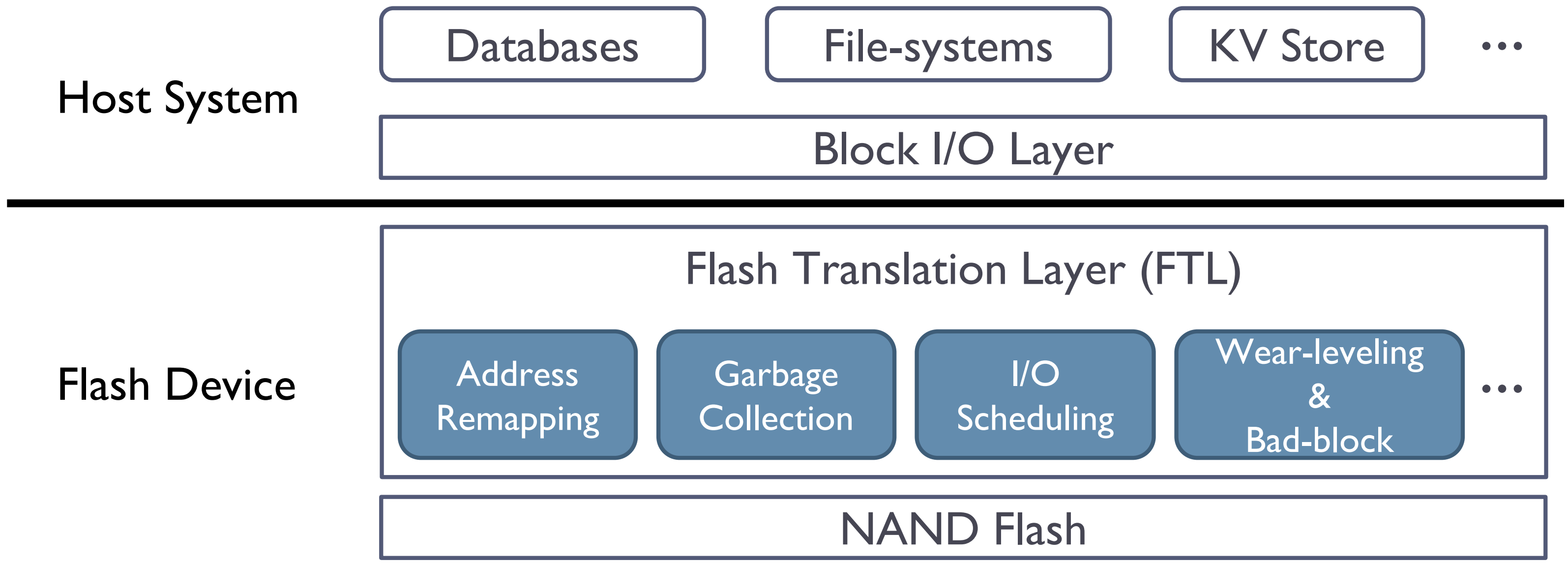


Today

- Write-optimized Storage Systems
- SDF: Software-Defined Flash
- **AMF: Application-Managed Flash**
- LightNVM: The Linux Open-Channel SSD Subsystem
- Reference

FTL is a Complex Piece of Software

- FTL runs complicated firmware algorithms to avoid in-place updates and manages unreliable NAND substrates

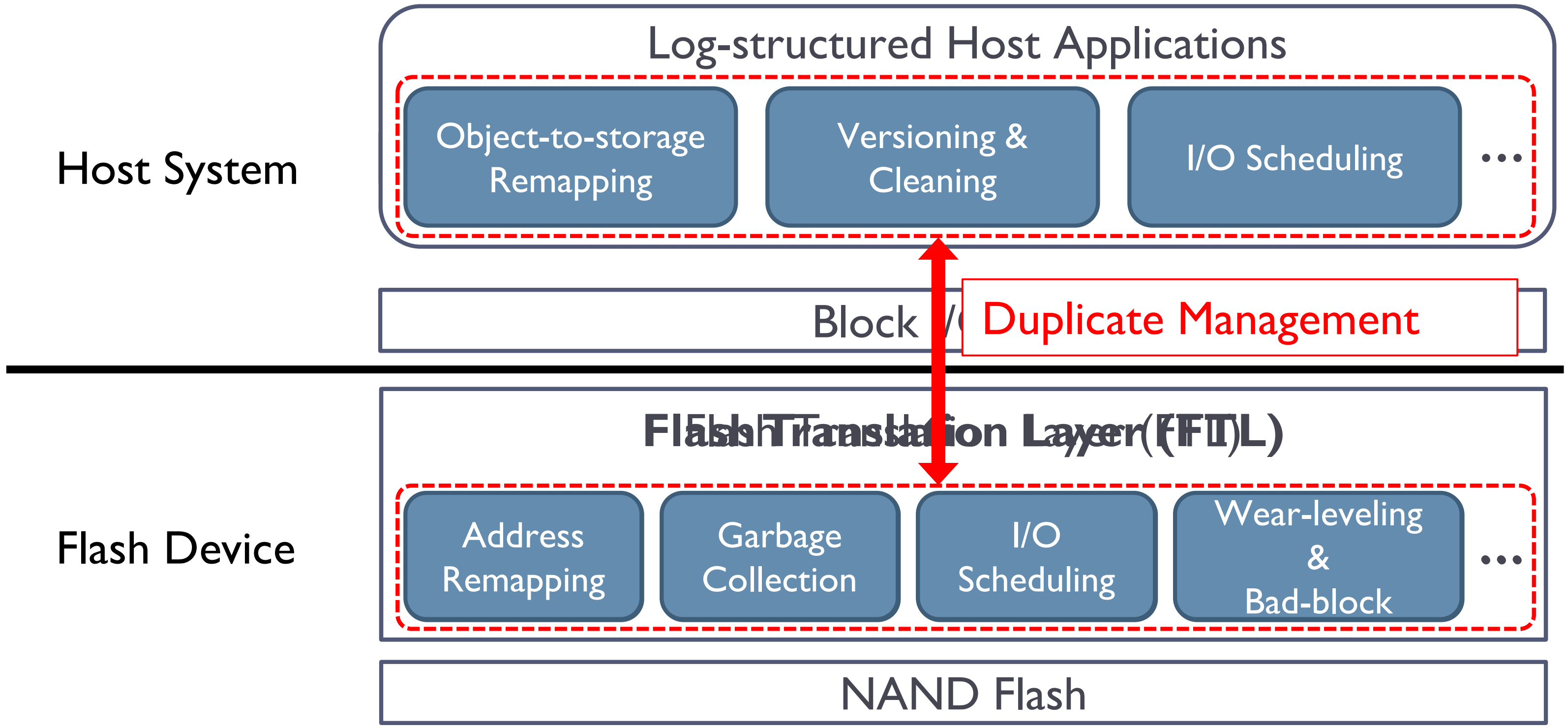


- But, FTL is ***a root of evil*** in terms of HW resources and performance
  - Requires significant hardware resources (e.g., 4 CPUs / 1-4 GB DRAM)
  - Incurs extra I/Os for flash management (e.g., GC)
  - Badly affects the behaviors of host applications

However,

Functionality of FTL is Mostly Useless

- Many host applications manage underlying storage in *a log-like manner*, mostly avoiding in-place updates



- This duplicate management not only (1) incurs serious performance penalties but also (2) wastes hardware resources

Which Applications???

File Systems

SpriteLFS

WAFL

BlueSky

F2FS

NILFS

Btrfs

HDFS

Key-value Stores

LevelDB

RocksDB

LSM-Tree

Which Applications ???

Databases

RethinkDB

Cassandra

BigTable

LogBase

MongoDB

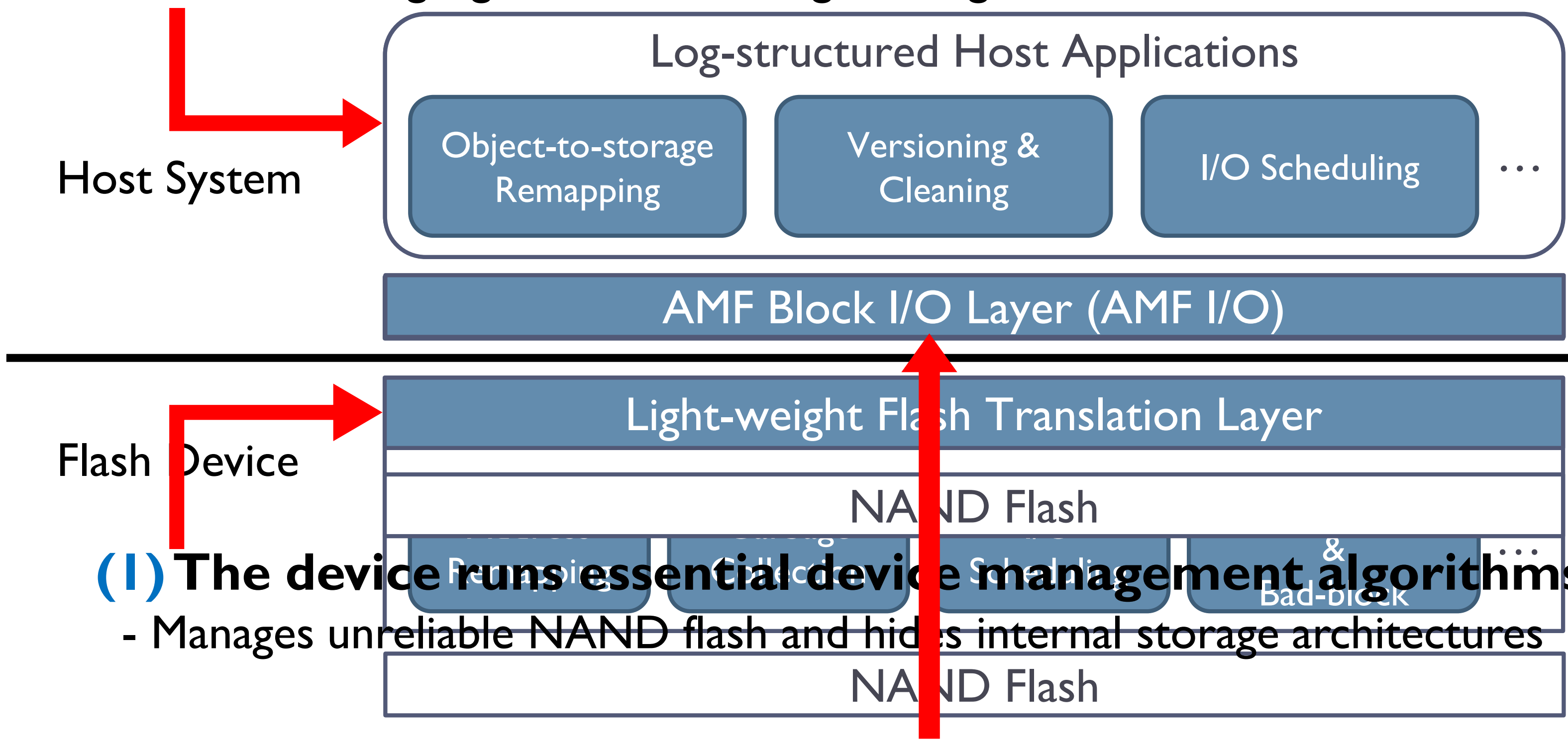
Hyder

What if we removed FTL from storage devices and allowed applications to directly manage NAND flash?

Application-Managed Flash (AMF)

(2) The host runs almost all of the complicated algorithms

- Reuse existing algorithms to manage storage devices



(1) The device runs essential device management algorithms

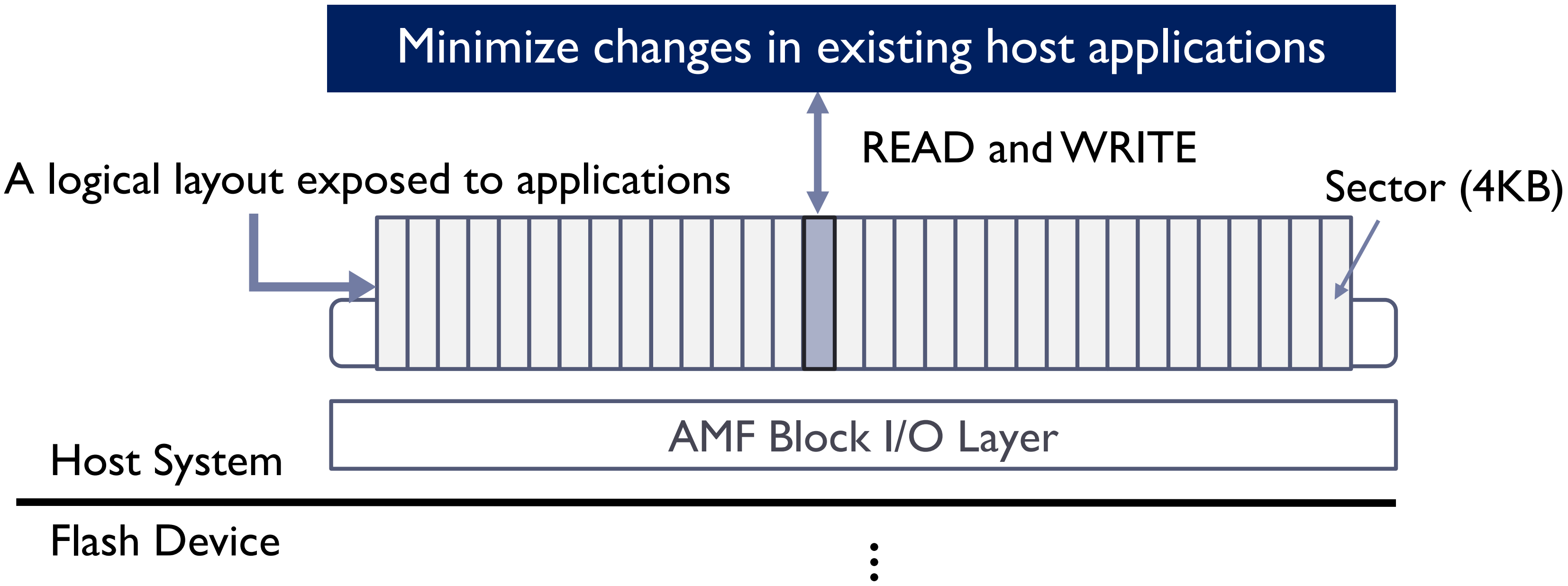
- Manages unreliable NAND flash and hides internal storage architectures

(3) A new AMF block I/O abstraction enables us to separate the roles of the host and the device

AMF Block I/O Abstraction (AMF I/O)

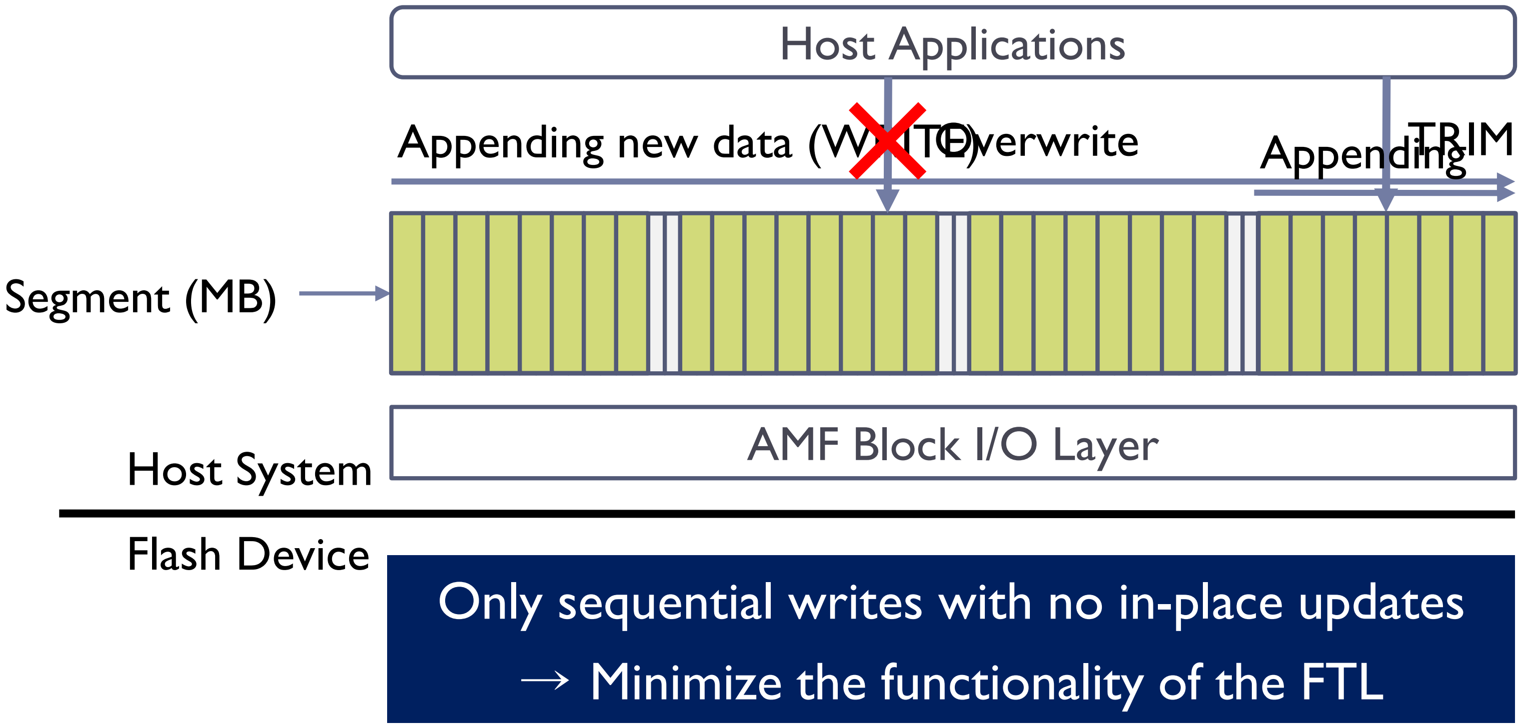
- **AMF I/O is similar to a conventional block I/O interface**
  - A linear array of fixed-size sectors (e.g., 4 KB) with existing I/O primitives (e.g., READ and WRITE)

    기존 I/O primitives를 통한 fixed - size sector의 linear array

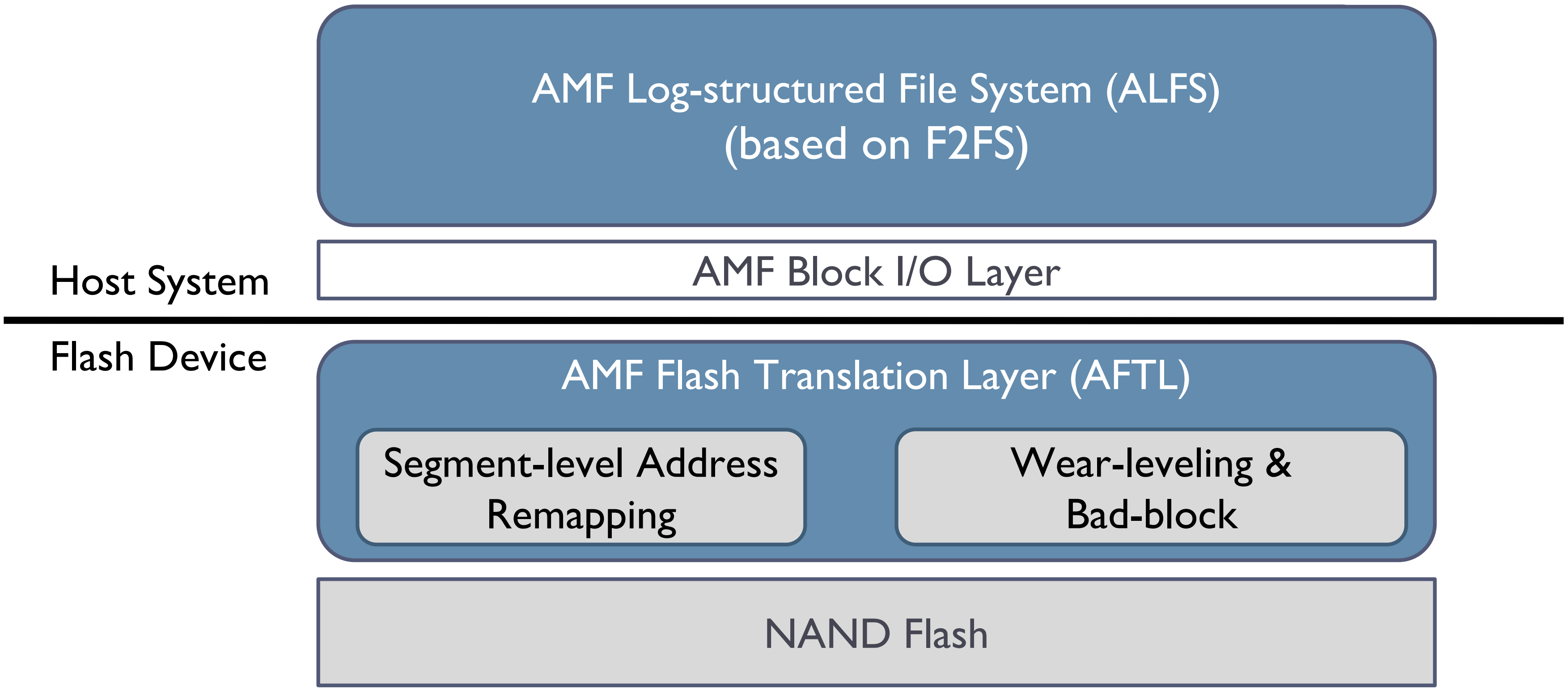


Append-only Segment

- **Segment: a group of 4 KB sectors (e.g., several MB)**
  - A unit of free-space *allocation* and free-space *reclamation*
- **Append-only: overwrite of data is prohibited**



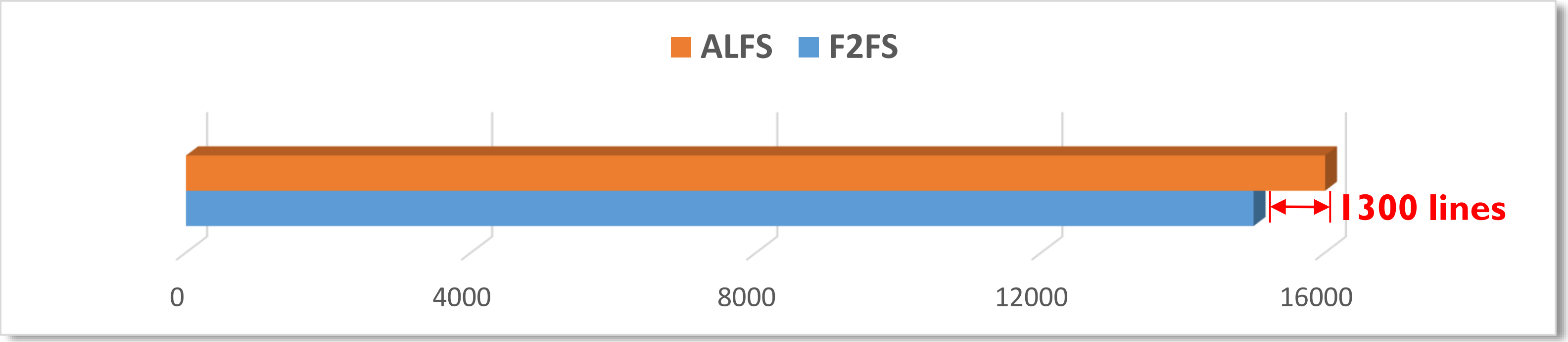
Case Study with File System



AMF Log-structured File System (ALFS)

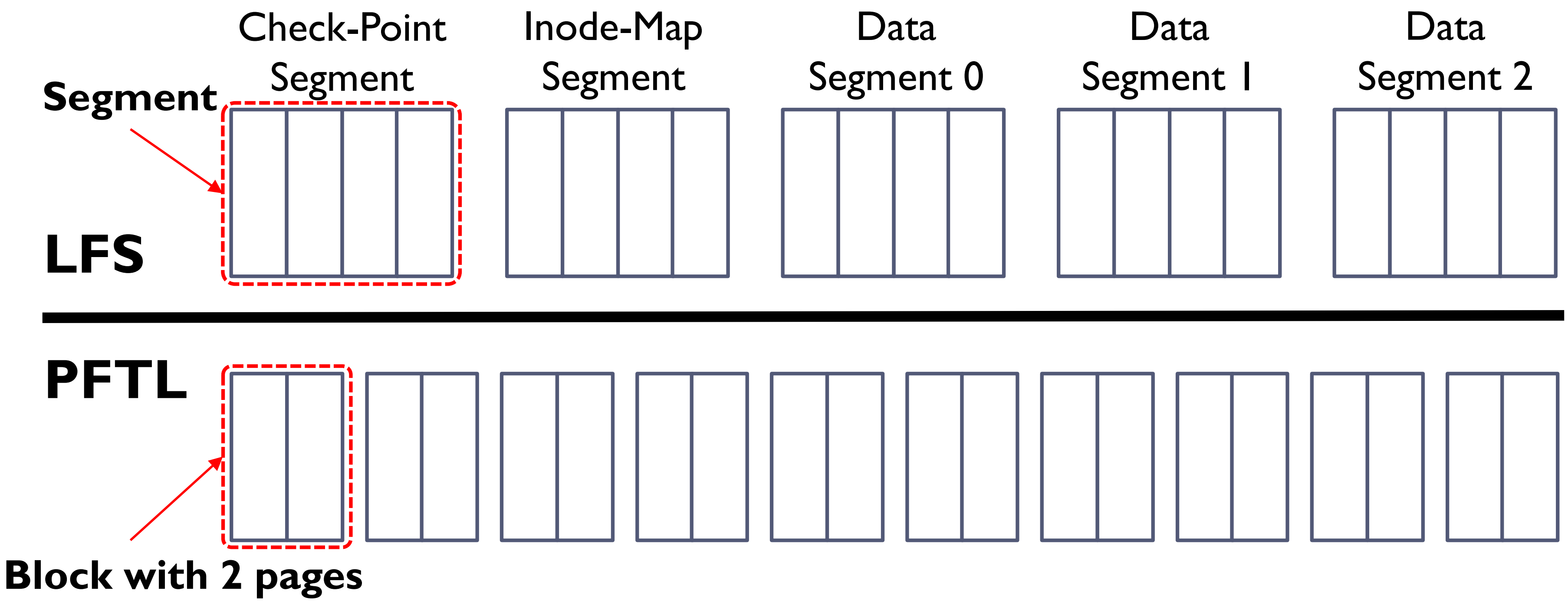
- **ALFS is based on the F2FS file system**
- **How did we modify F2FS for ALFS?**
  - Eliminate in-place updates
    - F2FS overwrites check-points and inode-map blocks
  - Change the TRIM policy
    - TRIM is issued to individual sectors

- **How many new codes were added?**



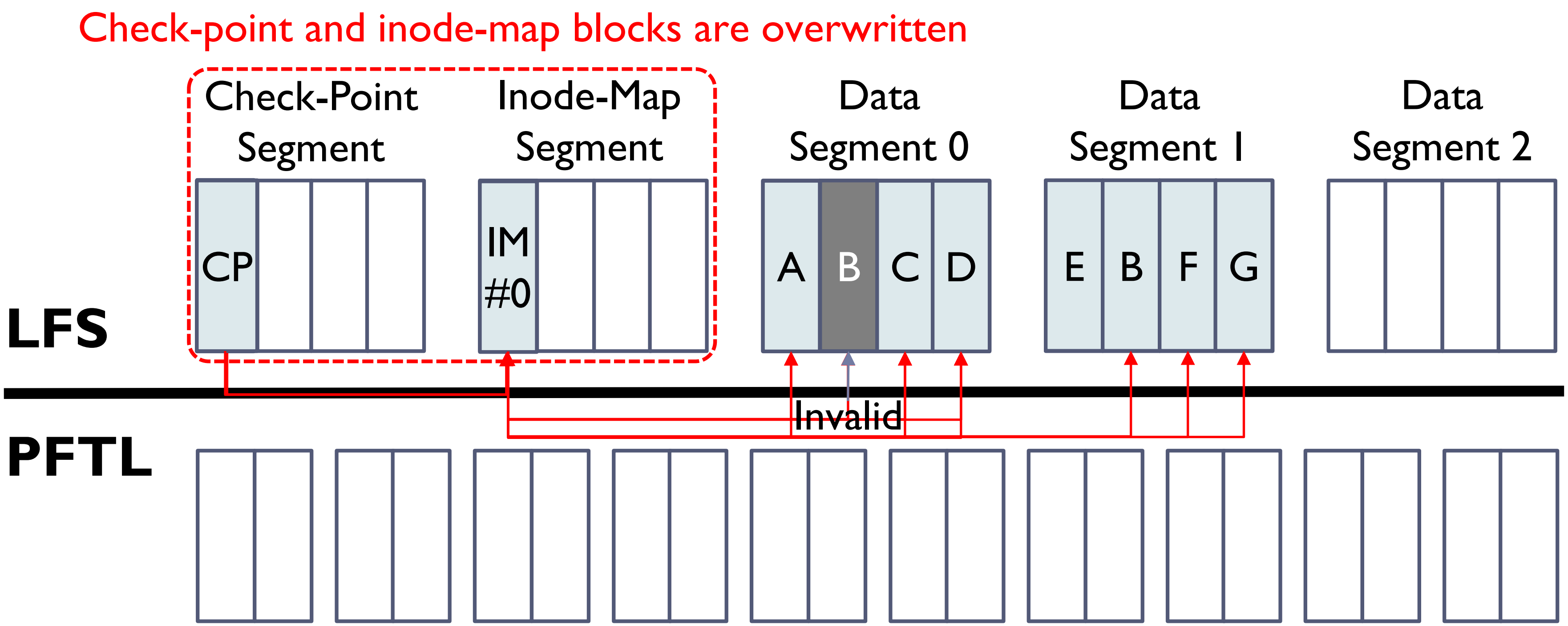
<A comparison of source-code lines of F2FS and ALFS>

How Conventional LFS (F2FS) Works



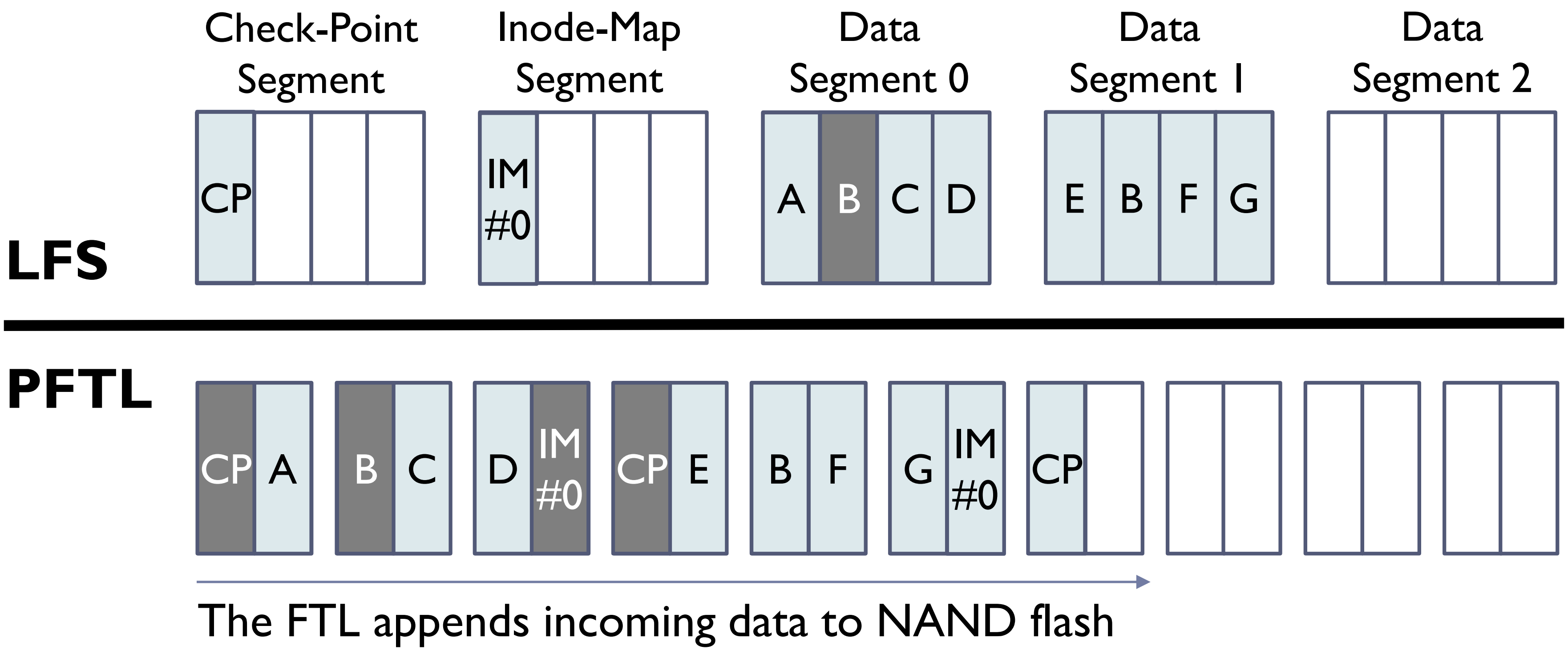
* PFTL: page-level FTL

How Conventional LFS (F2FS) Works



* PFTL: page-level FTL

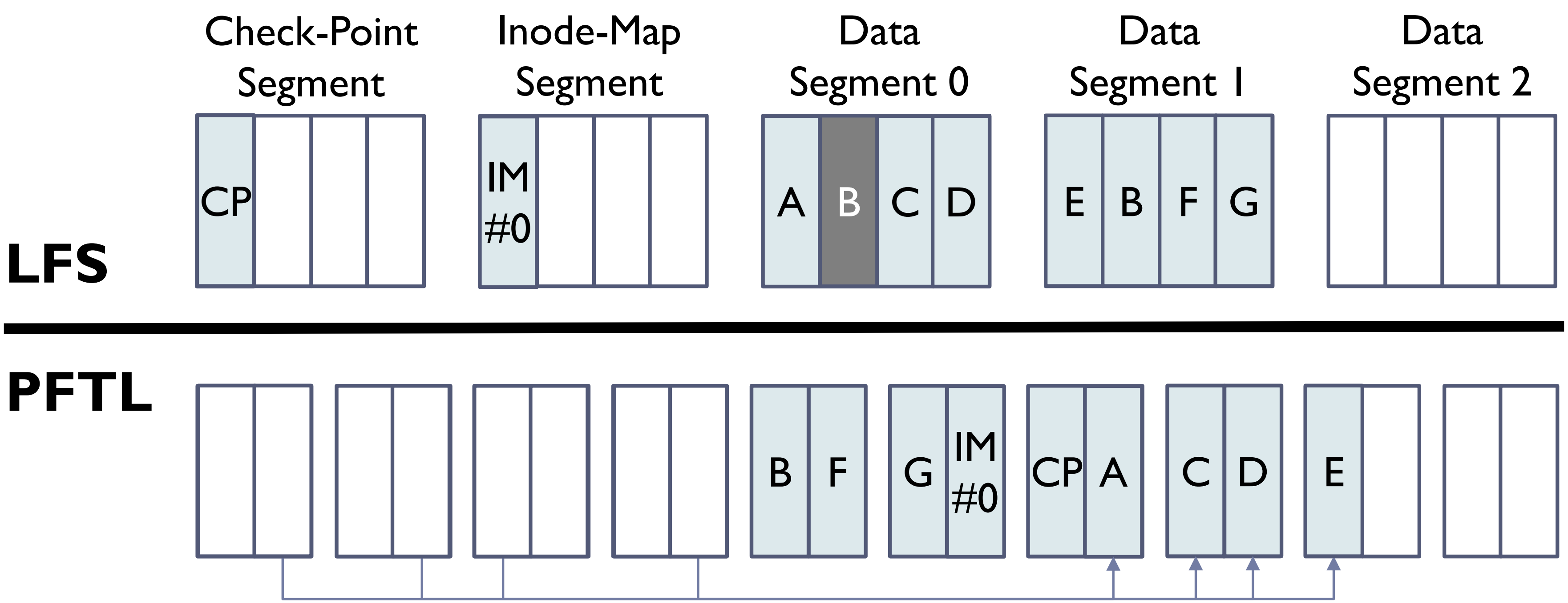
How Conventional LFS (F2FS) Works



invalid data가 FTL 안에 흩뿌려진다.

*** PFTL: page-level FTL**

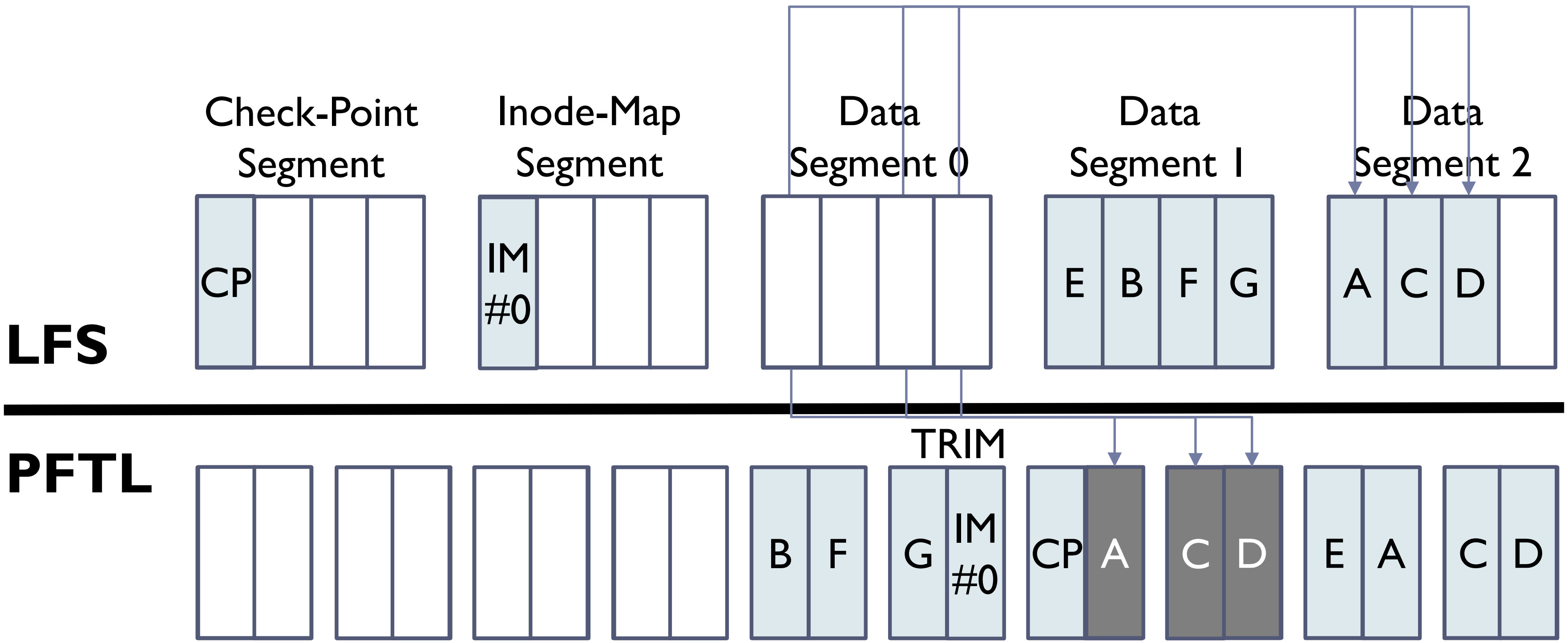
How Conventional LFS (F2FS) Works



The FTL triggers garbage collection: **4** page copies and **4** block erasures

*** PFTL: page-level FTL**

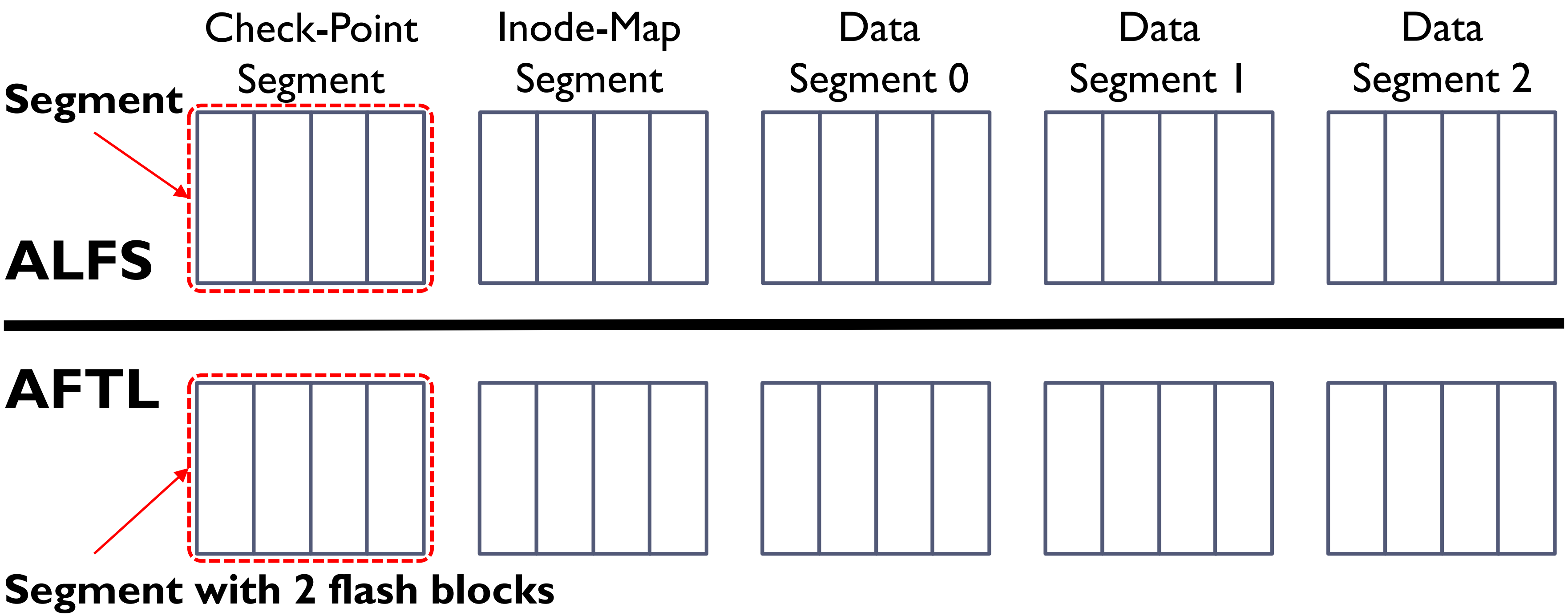
How Conventional LFS (F2FS) Works



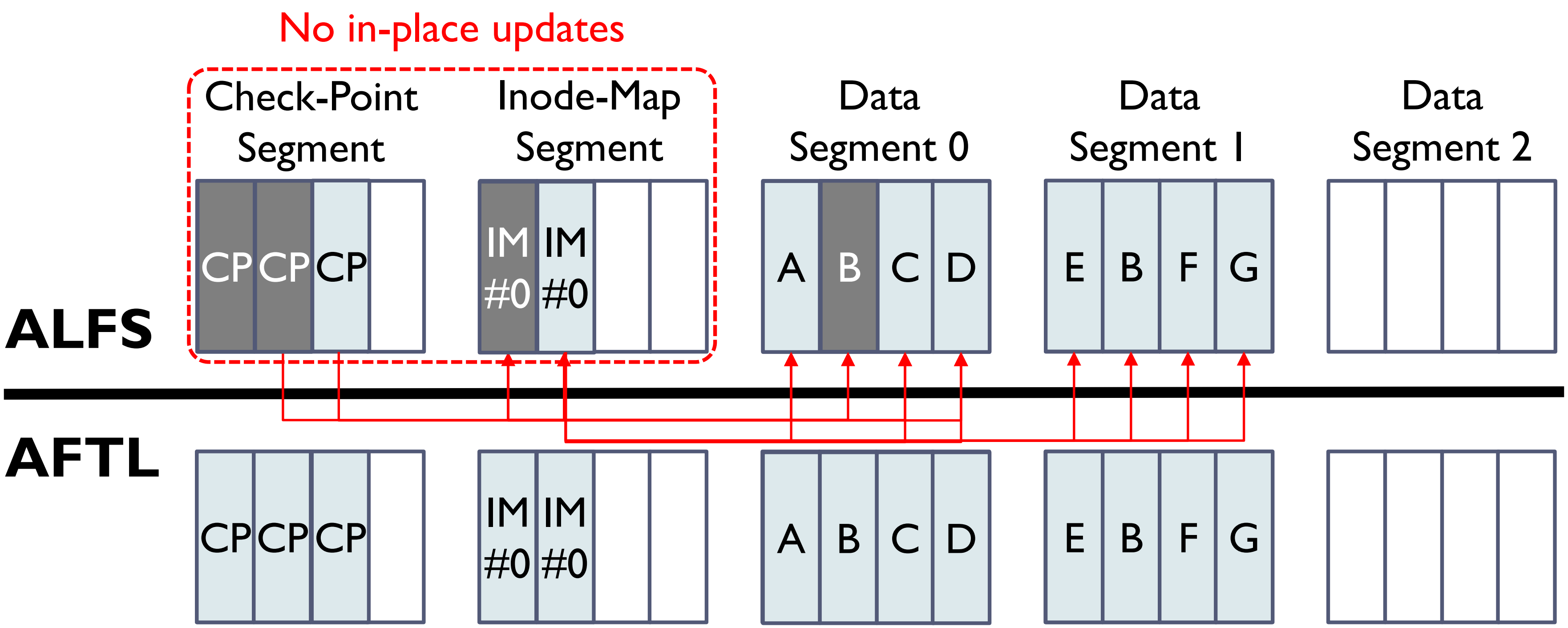
The LFS triggers garbage collection: **3** page copies

* **PFTL: page-level FTL**

How ALFS Works

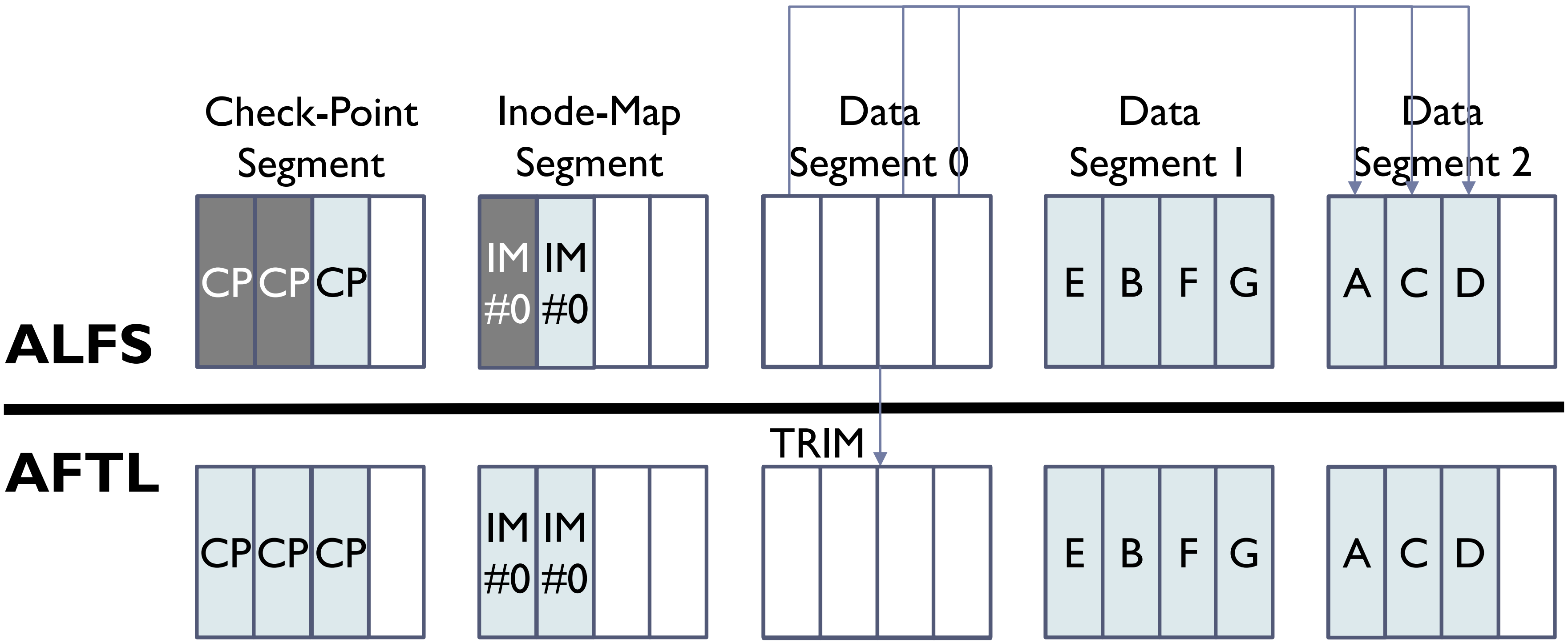


How ALFS Works



No obsolete pages – GC is not necessary

How ALFS Works



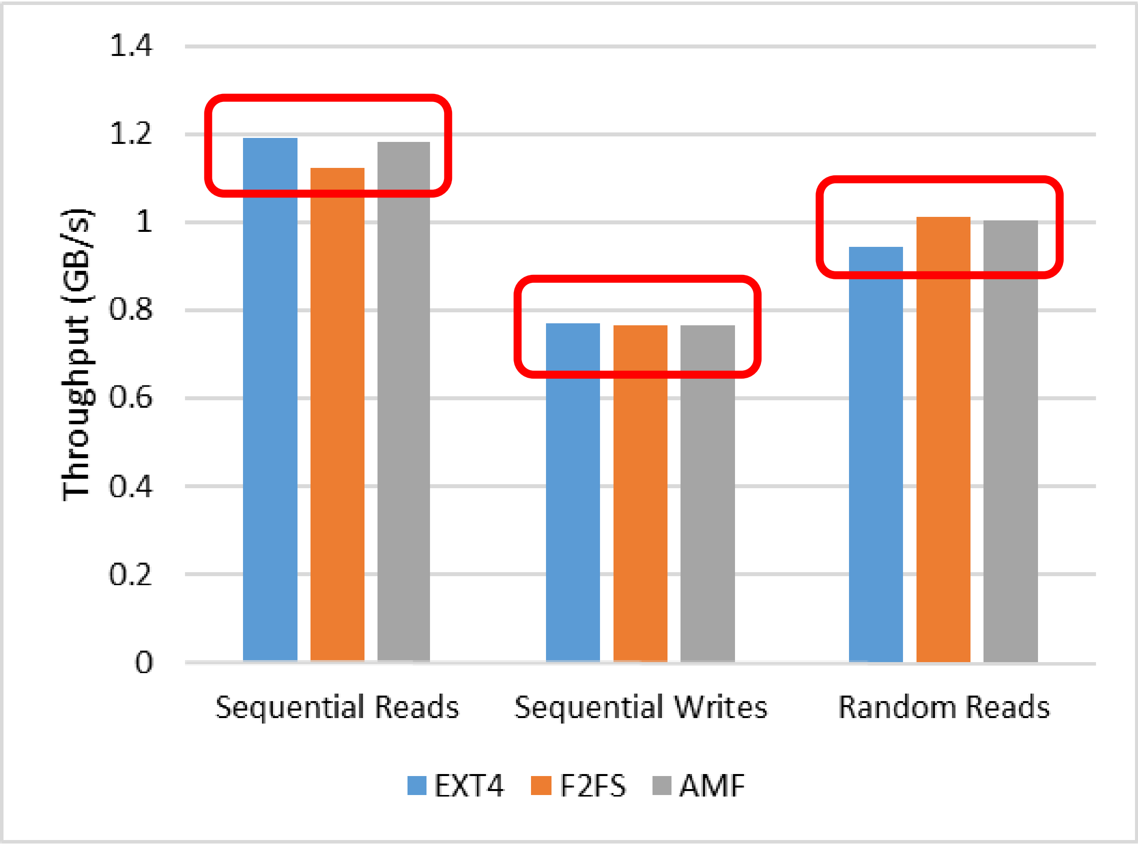
The ALFS triggers garbage collection: **3** page copies and **2** block erasures

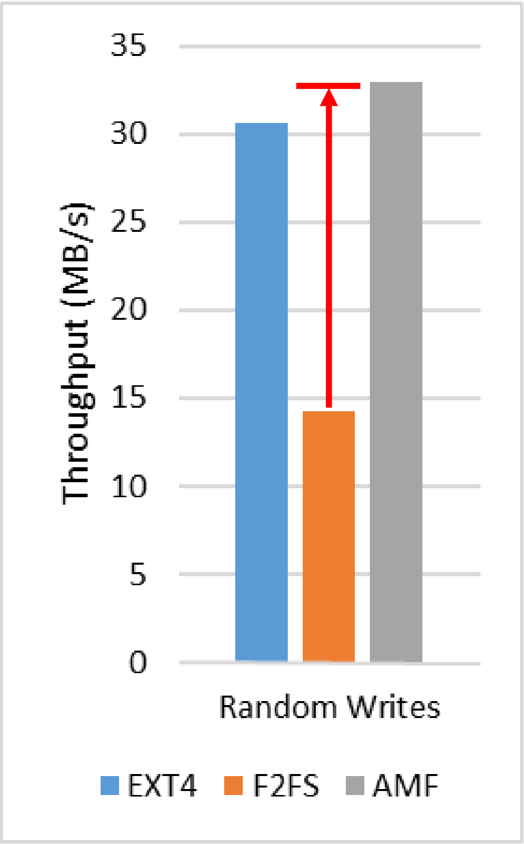
Comparison of F2FS and AMF

Duplicate Management

F2FS		AMF
File System	PFTL	File System
3 page copies	4 copies + 4 erasures	3 copies + 2 erasures
7 copies + 4 erasures		**3 copies + 2 erasures**

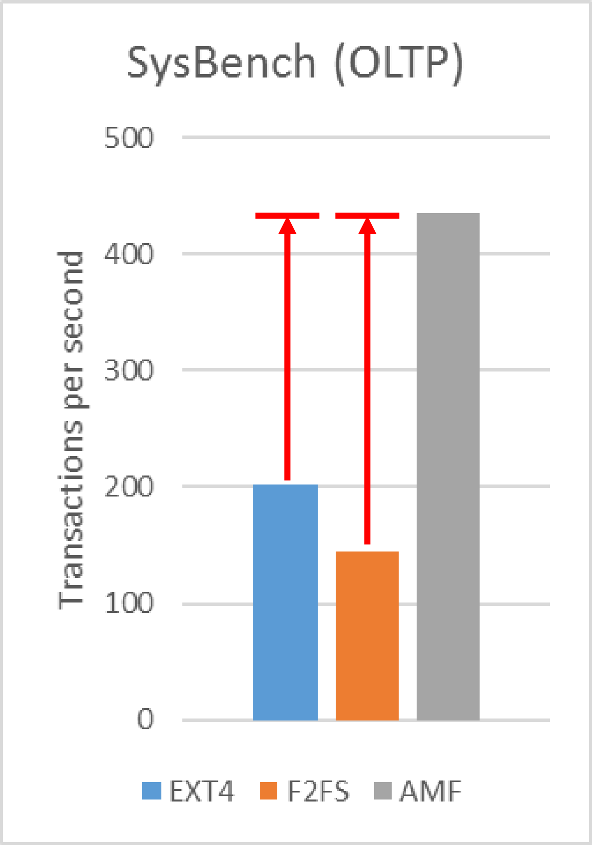
Performance with FIO

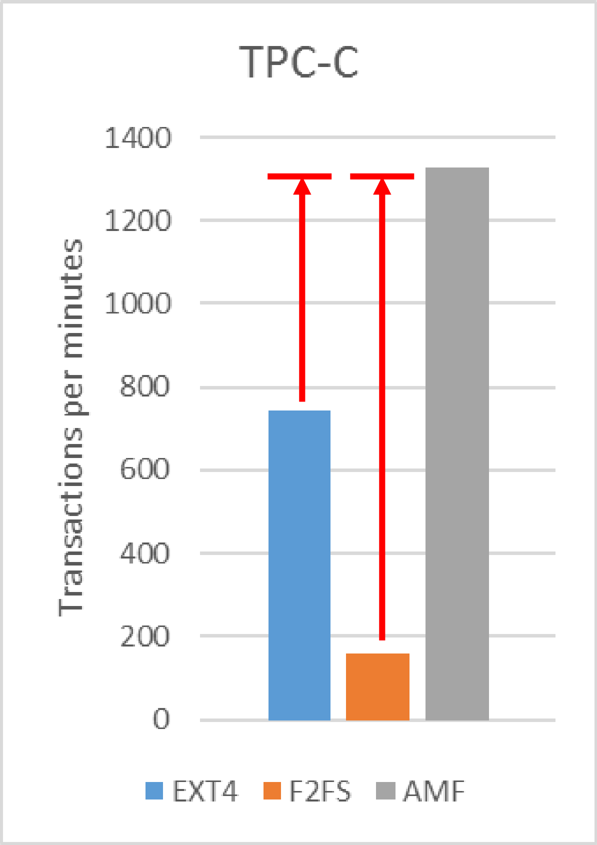




- **For random writes, AMF shows better throughput**
- **F2FS is badly affected by the duplicate management problem**

Performance with Databases





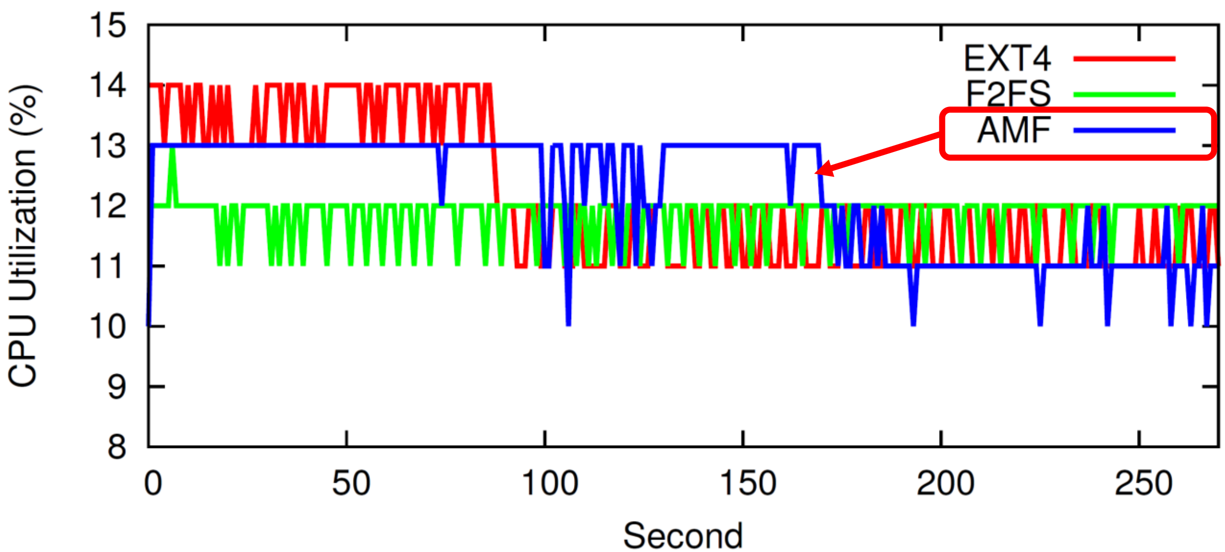
- **AMF outperforms EXT4 with more advanced GC policies**
- **F2FS shows the worst performance**

Resource (DRAM & CPU)

- **FTL mapping table size**

SSD Capacity	Block-level FTL	Hybrid FTL	Page-level FTL	AMF
512 GB	4 MB	96 MB	512 MB	4 MB
1 TB	8 MB	186 MB	1 GB	8 MB

- **Host CPU usage**



Today

- Write-optimized Storage Systems
- SDF: Software-Defined Flash
- AMF: Application-Managed Flash
- **LightNVM: The Linux Open-Channel SSD Subsystem**
- Reference

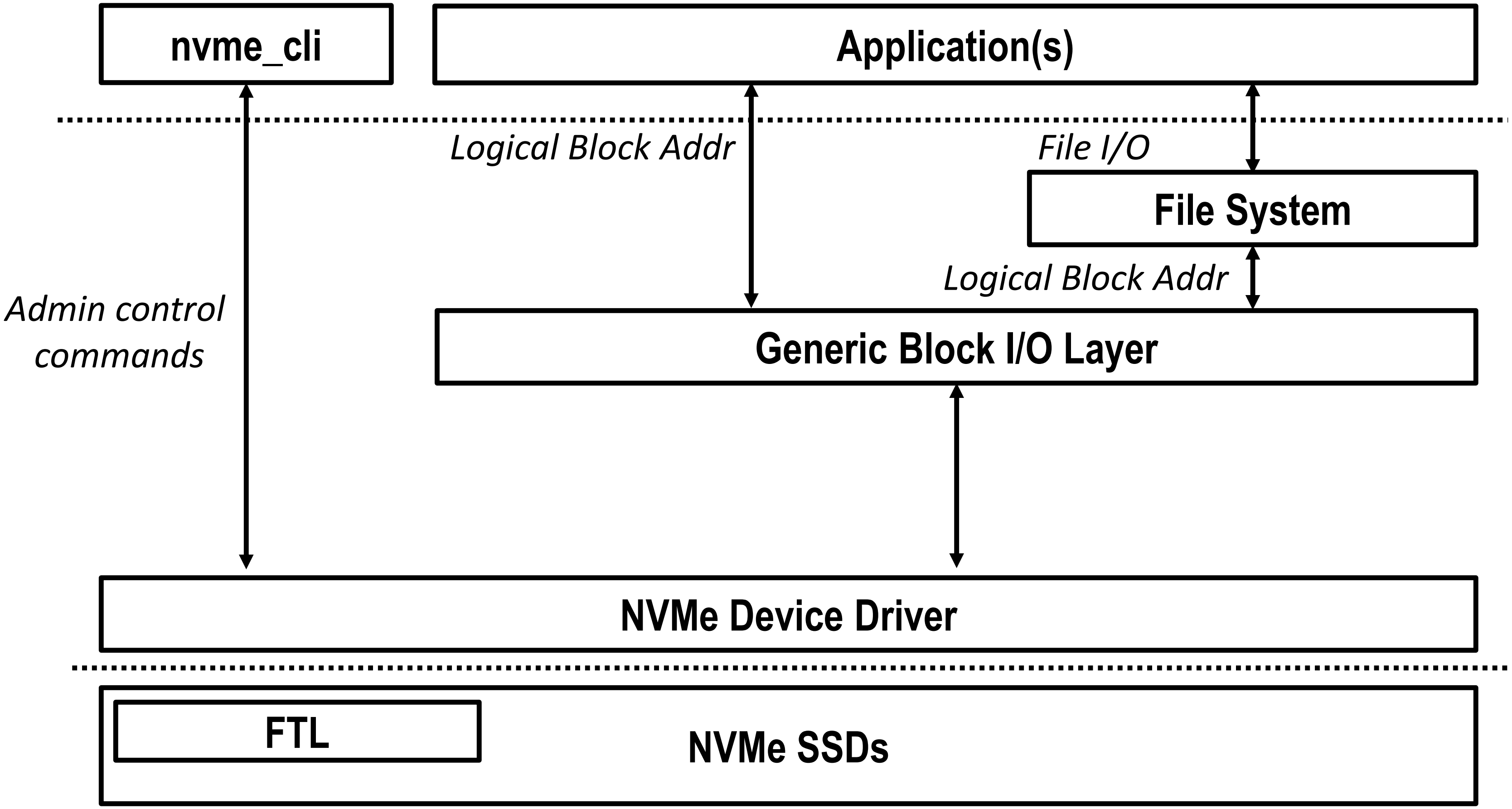
Open-Channel SSDs

- **Open-channel SSDs are emerging on the market**
  - Excellent platform for addressing SSD shortcomings and managing trade-offs related to throughput, latency, power, and capacity
  - Examples:
    - *Fusion-IO and Violin Memory*: a storage stack that manages NAND media and provides a block I/O interface
    - *Baidu's SDF*: a key-value store integrated with underlying storage media
    - *AMF*: a new SSD interface, compatible with the legacy block device interface, exposing error-free and append-only segments

- **The integration of open-channel SSDs into the storage infrastructure has been limited**
  - A single point in the design space is explored with a fixed collection of trade-offs

Design of LightNVM

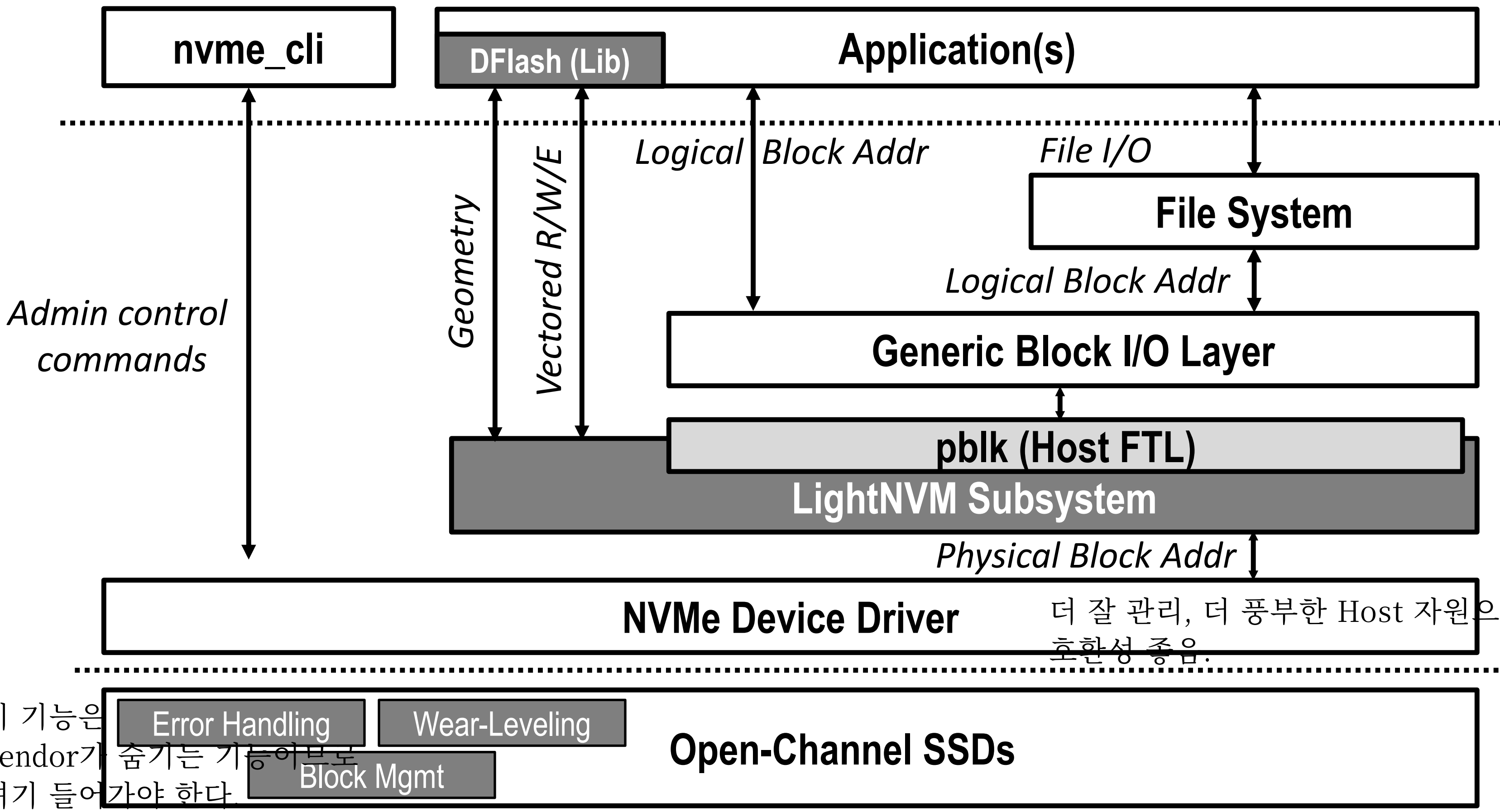
- LightNVM is the first open, generic subsystem for Open-Channel SSDs and host-based SSD management



Design of LightNVM

- LightNVM is the first open, generic subsystem for Open-Channel SSDs and host-based SSD management

Lib로 인해, App이 kernel 상관없이 직접 관리할 수 있다. i/o latency 줄어든다.



더 잘 관리, 더 풍부한 Host 자원으로 관리, 호환성 좋음.

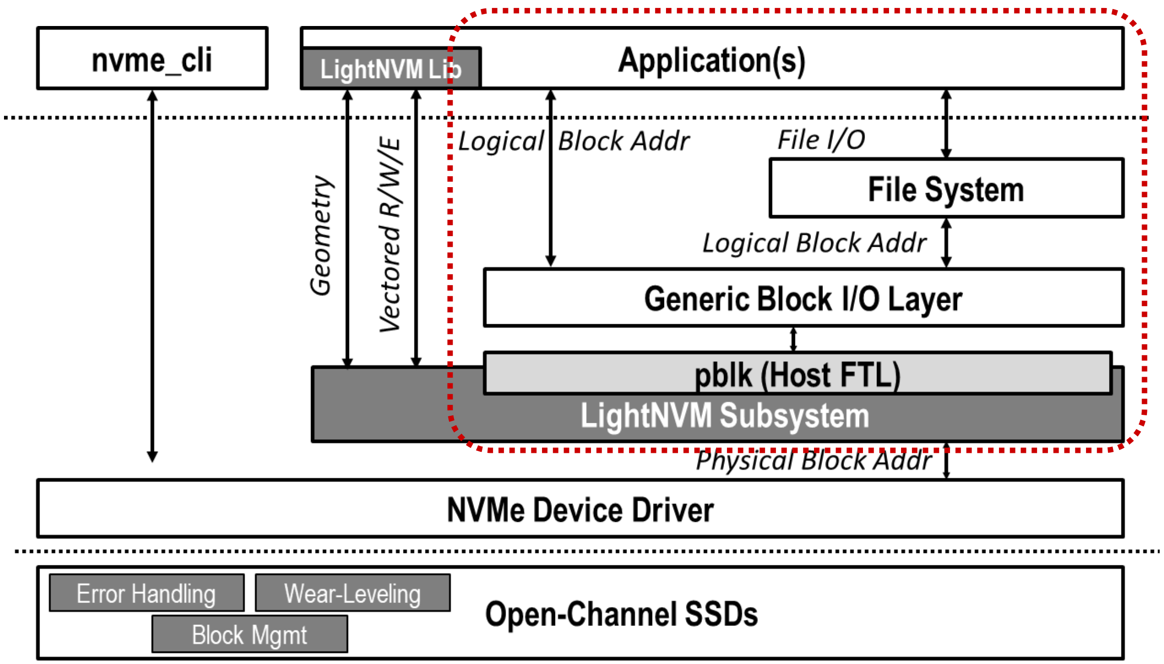
이 기능은 vendor가 숨기는 기능이므로 여기 들어가야 한다.

LightNVM의 use case 1: Block I/O Device

Use Case #1: Block I/O Device

pblk는 block I/O interface를 위에 expose하는 fully associative, host - based FTL이다.

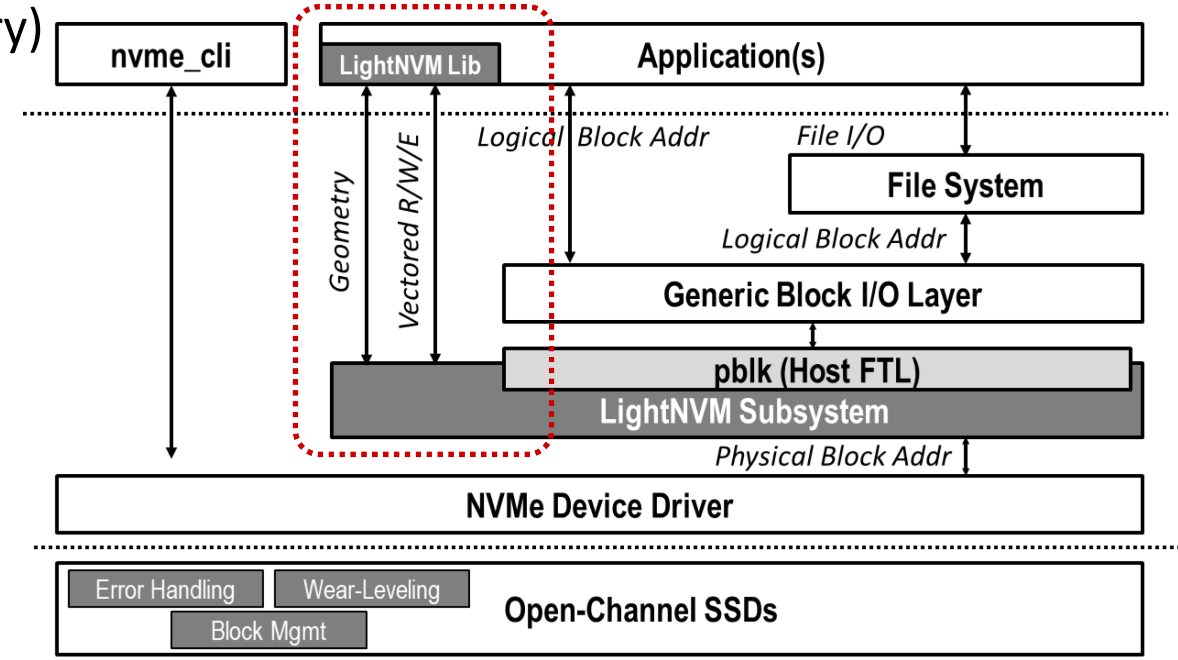
- **The Physical Block Device (*pblk*) is a fully associative, host-based FTL that exposes a block I/O interface**
  - Dealing with controller- and media-specific constraints (e.g., write buffering)
  - Flushes – forces *pblk*'s in-flight data to the device
  - Map logical addresses onto physical addresses
  - Garbage collection
  - Handling error
- ***pblk* allows us to run existing apps and file systems on top of Open-channel SSDs**



use case 2: SSD에 최적화된 application 만들기 가능

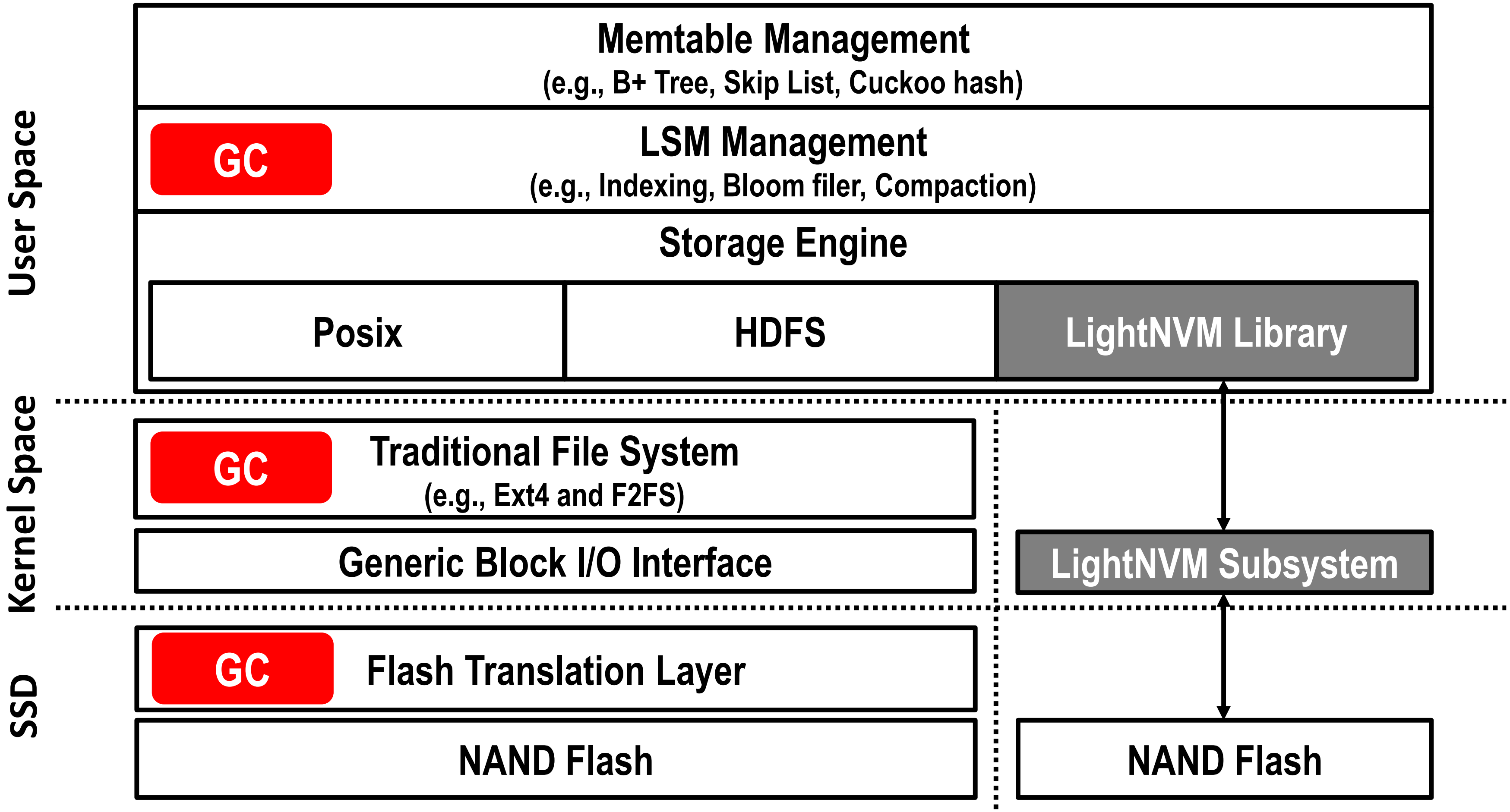
Use Case #2: Application-specific SSD

- **Exposes flash geometry to the applications, which allows us to implement SSD-optimized applications**
- **Target applications**
  - Key-value store: RocksDB optimized for LightNVM is under development by CNEXLab
  - Application-specific FTL implementation (as a user-level library)



Use Case #2: Application-specific SSD (Cont.)

- RocksDB with LightNVM

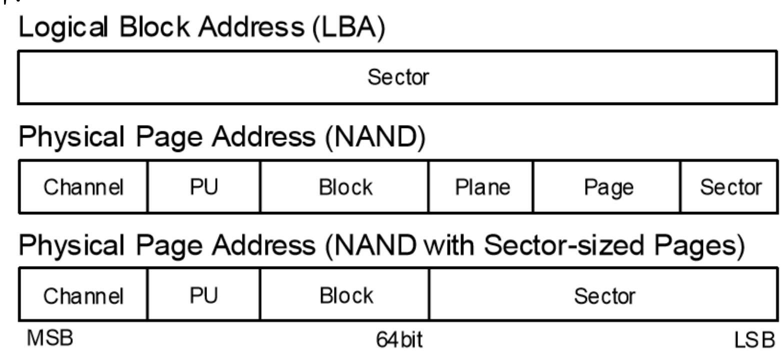


기존의 NVMe protocol 그대로 사용

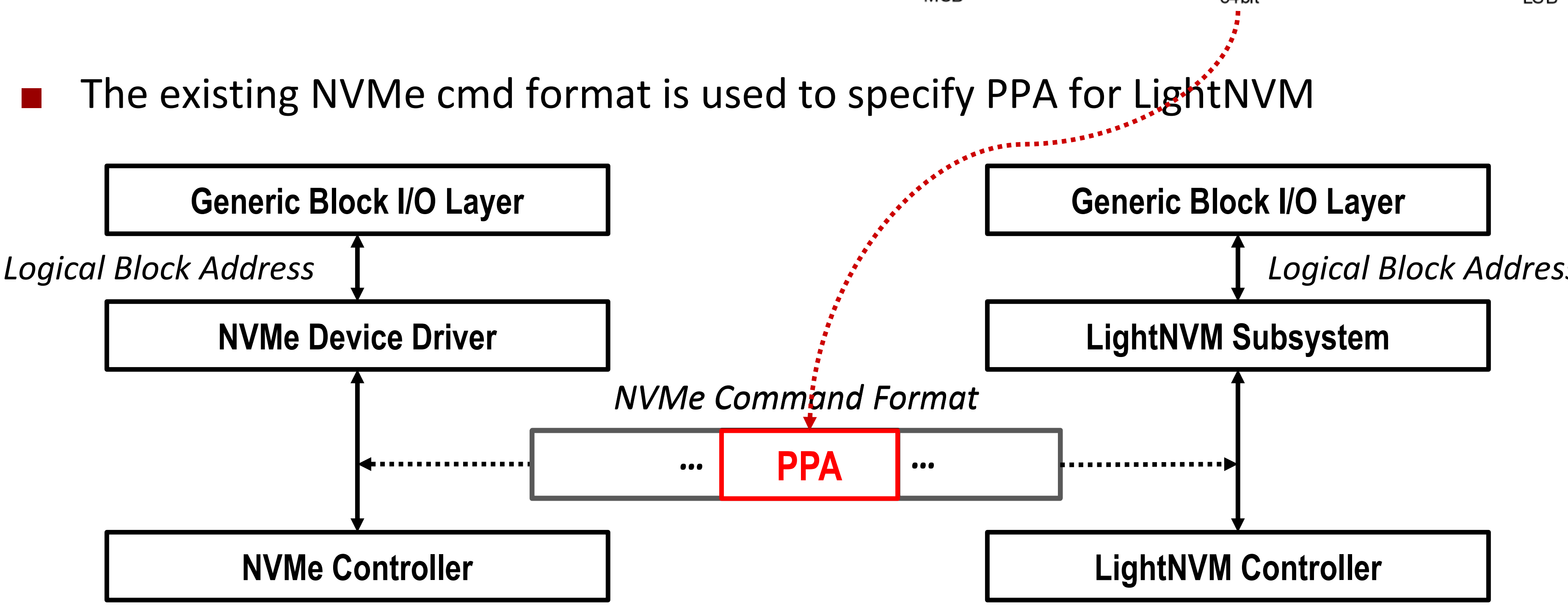
LightNVM Address Space

host에게 노출된 채널의 모음은, 각각이 한 set의 PU를 포함한다.

- Expose to the host a collection of channels, each containing a set of *Parallel Units* (PUs) (a NAND die)
- Each PU is decomposed into either (*blocks, plane, page, sector*) or (*block, sector*)



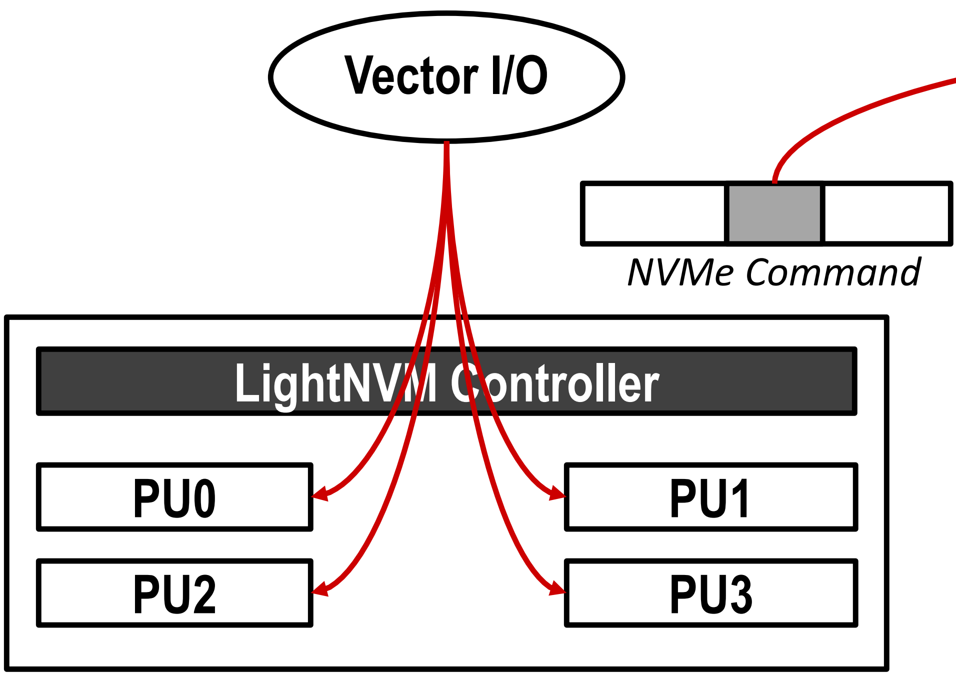
- The existing NVMe cmd format is used to specify PPA for LightNVM



Vector I/O

LBA는 연속적이지만 PPA는 연속적이지 못해 i/o overhead.
vector i/o로 보내면 i/o traffic을 줄일 수 있다.

- Obtain higher throughput through parallel units
- Large overhead if I/Os are separately issued
- Introduce vector I/O interface to enable host to submit I/Os to multiple PUs using one command
- Vector Read/Write/Erase using scatter/gather address list



PPA List

LBA#	
0	CH0, PU0, Sector 120
1	CH0, PU3, Sector 64
2	CH0, PU2, Sector 212
3	CH0, PU1, Sector 513

Write Buffering – Host-side

- **(1) Host-side write buffering**

그냥 host에서 실제 flash page만큼 모아서 flush하기.
host에서 모으다가 failure나면 날아간다. 현재 구현.

  - Sector writes (4 KB) are buffered until enough data is gathered to fill a flash page (e.g., 16-32 KB); paired pages must be handled similarly
  - In case of a flush, add padding in the command
  - (+) Avoid interference between the host and devices
  - (+) Acknowledged as reads hit the cache
  - (–) The contents loss in case of a power failure

host에 있으니
배터리도 못쓰고
날라간다.

→ **The current implementation of LightNVM**

Write Buffering – Device-side

failure로부터 write buffer를 보호하는 방법: Device - side

- **(2) Device-side write buffering**
  - Sector writes are buffered on the device side
  - (+) High durability with on-device battery or capacitor
  - (–) Unpredictability (interfere with host reads)

host가 write buffering을 볼 수 없으니 unpredictability.
언제 flush되는 지 모른다.

Future direction

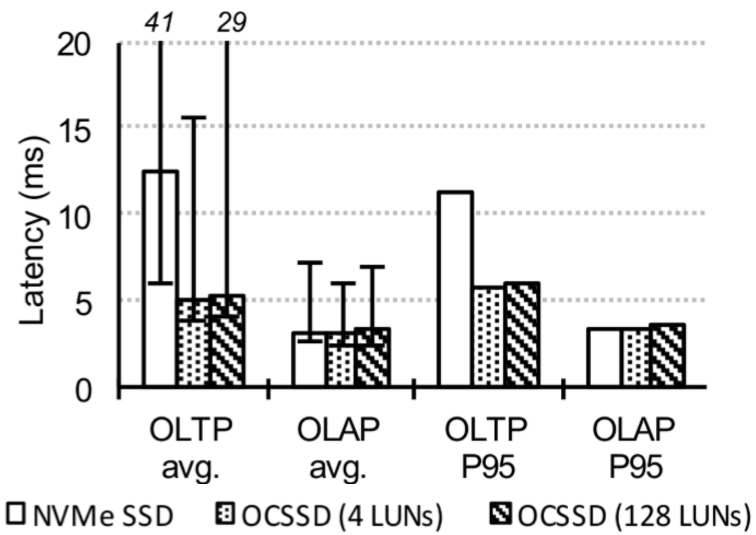
- **(3) Device-side write buffering with *Controller Memory Buffer* (CMB)**
  - Sector writes are buffered on the device, but *explicitly* flushed by the host
  - (+) High durability and predictability
  - (–) Complicated implementation

그러나 이렇게 하면 host가 device write buffer를 flush하니 볼 수 있다.

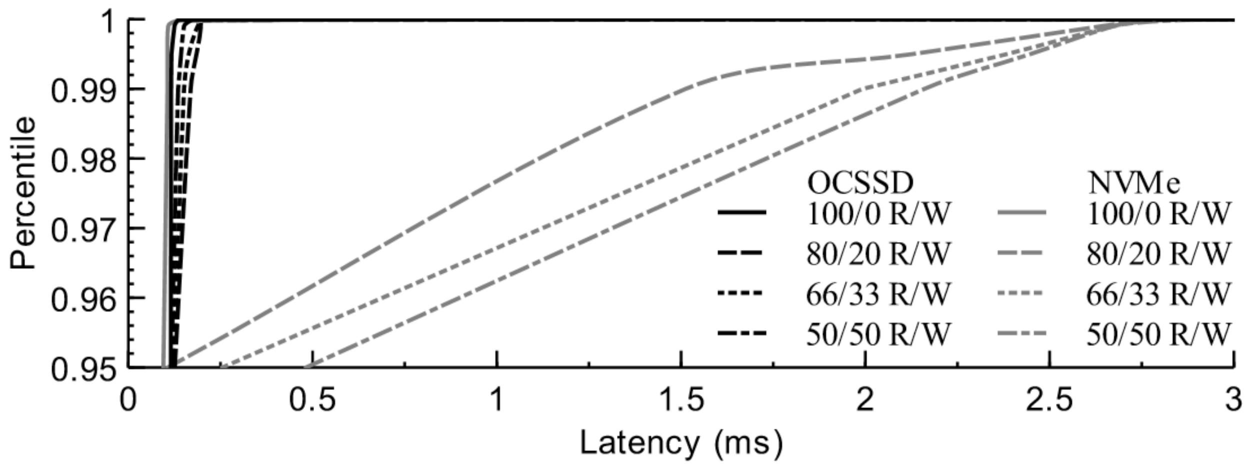
buffer는 battery backup 되는 device 안에 넣지만
관리는 host에서 하자는 것.

Performance

- Latencies for OLTP and OLAP on NVMe SSD and OCSSD



- Read latency comparison (4KB random reads / 64 KB writes)



End of Chapter 8