Storage Systems (StoSys) XM_0092

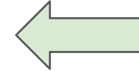
Lecture 3: FTL and GC

Animesh Trivedi

Autumn 2023, Period 1

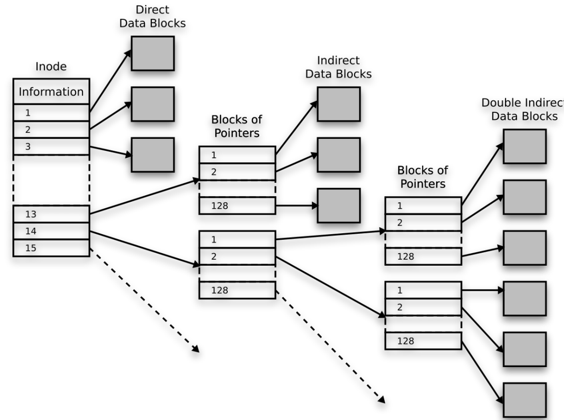
VU VRIJE UNIVERSITEIT AMSTERDAM

Syllabus outline

1. ~~Welcome and introduction to NVM (today)~~
2. ~~Host interfacing and software implications~~
3. Flash Translation Layer (FTL) and Garbage Collection (GC) 
4. NVM Block Storage File systems
5. NVM Block Storage Key-Value Stores
6. Emerging Byte-addressable Storage
7. Networked NVM Storage
8. Trends: Specialization and Programmability
9. Distributed Storage / Systems - I
10. Distributed Storage / Systems - II
11. Emerging topics

For the next lecture: File System

Refresh your idea of a basic file system
files, directories, inodes, etc.



40

File System Implementation

In this chapter, we introduce a simple file system implementation, known as **vsfs** (the **Very Simple File System**). This file system is a simplified version of a typical UNIX file system and thus serves to introduce some of the basic on-disk structures, access methods, and various policies that you will find in many file systems today.

The file system is pure software; unlike our development of CPU and memory virtualization, we will not be adding hardware features to make some aspect of the file system work better (though we will want to pay attention to device characteristics to make sure the file system works well). Because of the great flexibility we have in building a file system, many different ones have been built, literally from AFS (the Andrew File System) [H+88] to ZFS (Sun's Zettabyte File System) [B07]. All of these file systems have different data structures and do some things better or worse than their peers. Thus, the way we will be learning about file systems is through case studies: first, a simple file system (vsfs) in this chapter to introduce most concepts, and then a series of studies of real file systems to understand how they can differ in practice.

THE CRUX: HOW TO IMPLEMENT A SIMPLE FILE SYSTEM
How can we build a simple file system? What structures are needed on the disk? What do they need to track? How are they accessed?

40.1 The Way To Think

To think about file systems, we usually suggest thinking about two different aspects of them; if you understand both of these aspects, you probably understand how the file system basically works.

The first is the **data structures** of the file system. In other words, what types of on-disk structures are utilized by the file system to organize its data and metadata? The first file systems we'll see (including vsfs below) employ simple structures, like arrays of blocks or other objects, whereas

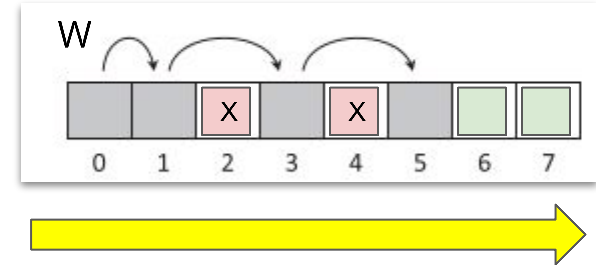
1

Checkout **the background reading** section on Canvas:
https://pages.cs.wisc.edu/~remzi/OSTEP/file-implementation.pdf

NAND Flash constraints

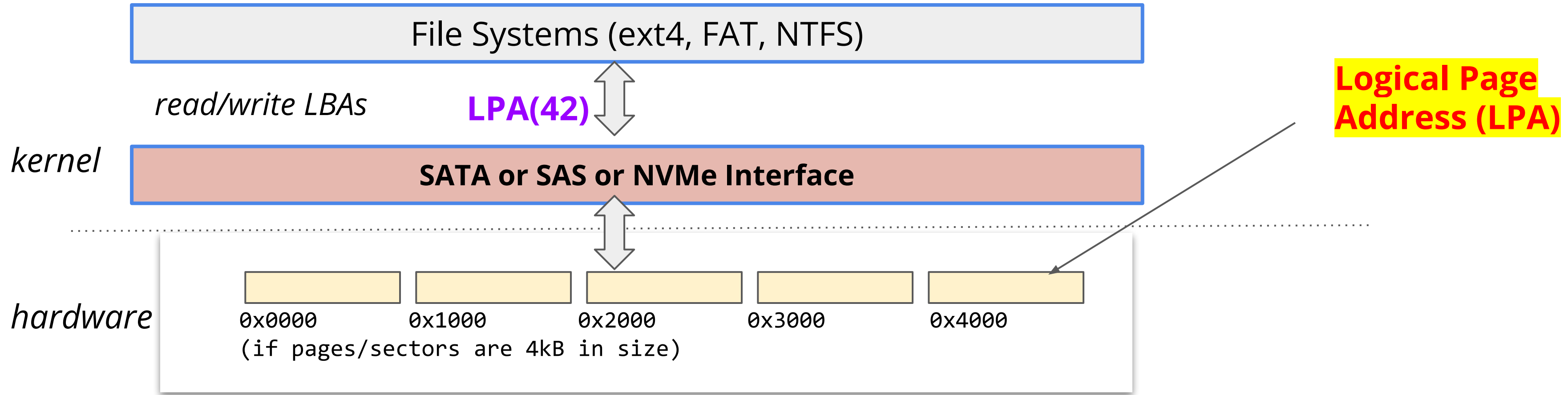
- Complex internal geometry
- No in-place updates
  - Any in-place writes must do a full block read-erase-write cycle
- Limited Program/Erase (P/E) cycles
- No mechanical latency
- Asymmetric read-write performance
  - Reads from flash chips is typically faster than writes
- Sequential page writes/programming within a block
- Anything else?

	Block Size	Page Size	OOB Size	# Pages/Block
Small-block SLC	16KB	512 bytes	16 bytes	32
Large-block SLC	128KB	2KB	64 bytes	64
Large-block MLC	512KB	4KB	128 bytes	128



We continue to use terms page and blocks in the lectures

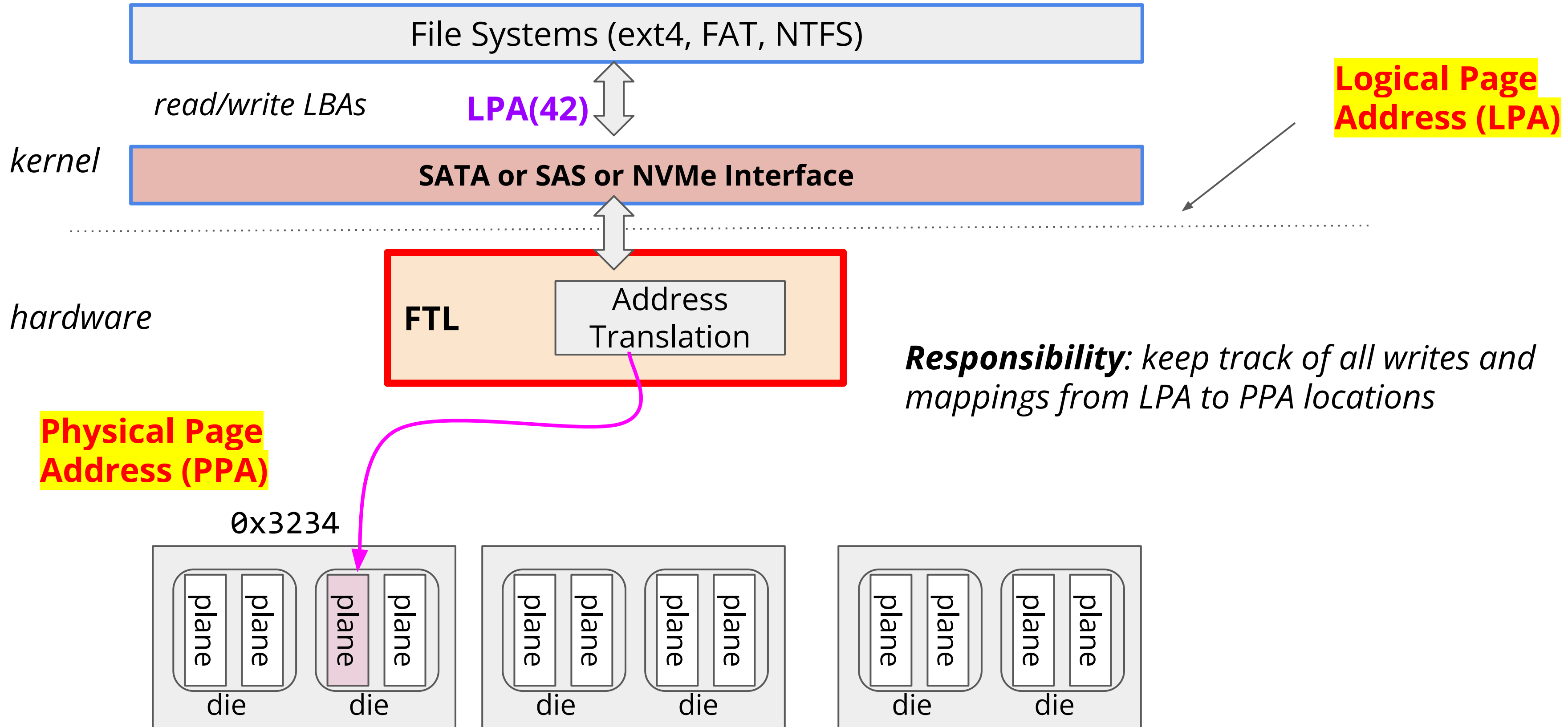
Recall - What is a Flash Translation Layer (FTL)



Logical Page Addresses (LPA) is what a host software sees

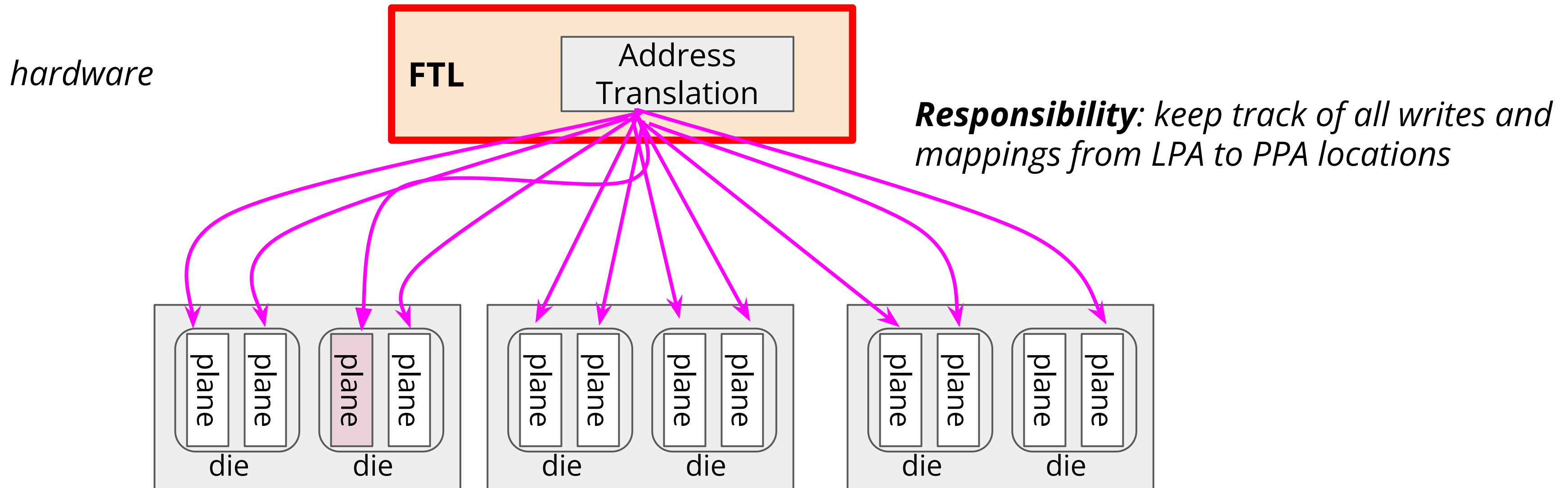
- A host can issue a read/write operation on a particular LBA
- Based on the device, it can be 512 bytes (backward compatible - BIOS might not know about 4kB pages) or some kB (to match the actual flash page size, 4 or 8 kB)

Recall - What is a Flash Translation Layer (FTL)



Recall - What is a Flash Translation Layer (FTL)

What happens when the whole device is written once? Without Erase?

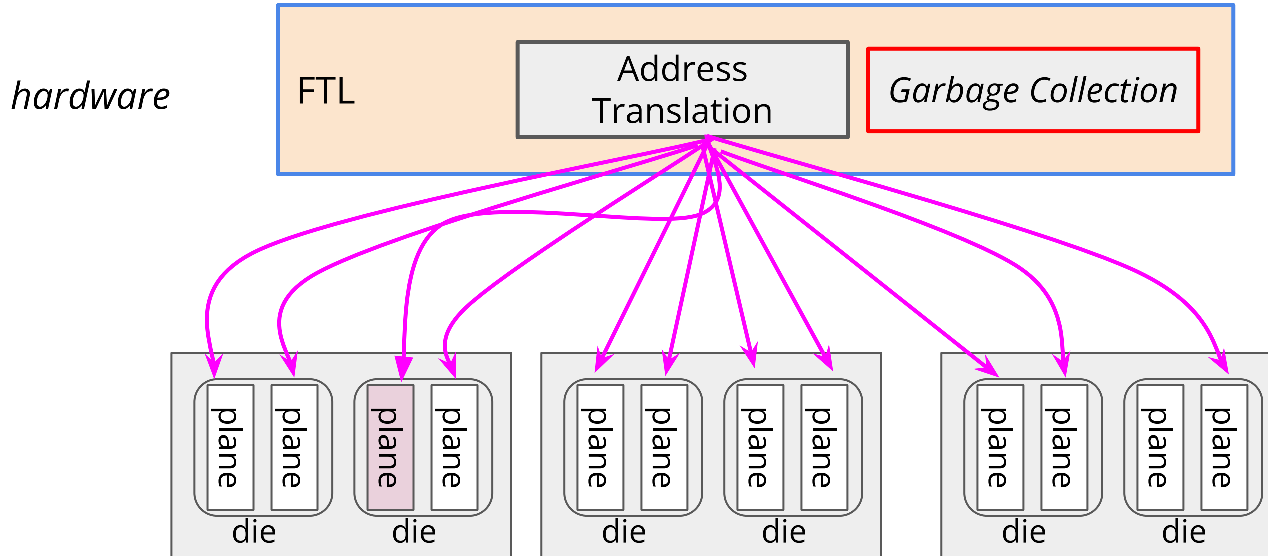


***Responsibility**: keep track of all writes and mappings from LPA to PPA locations*

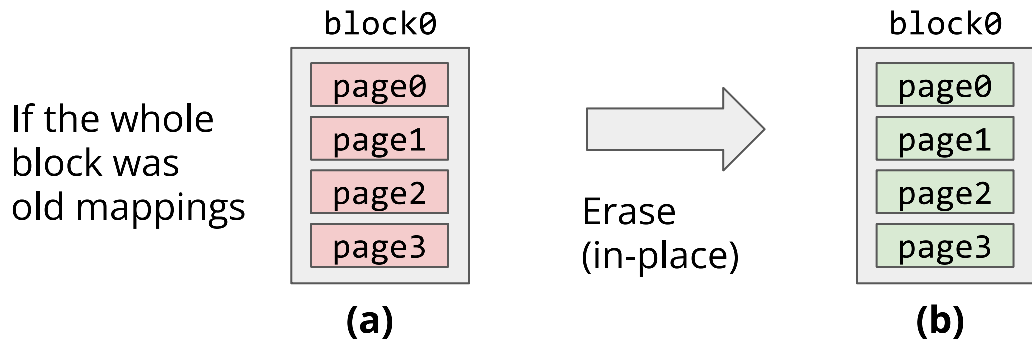
Recall - What is a Flash Translation Layer (FTL)

What happens when the whole device is written once? Without Erase?

Garbage collection - *move data around to prepare blocks for erase*



What does GC do?

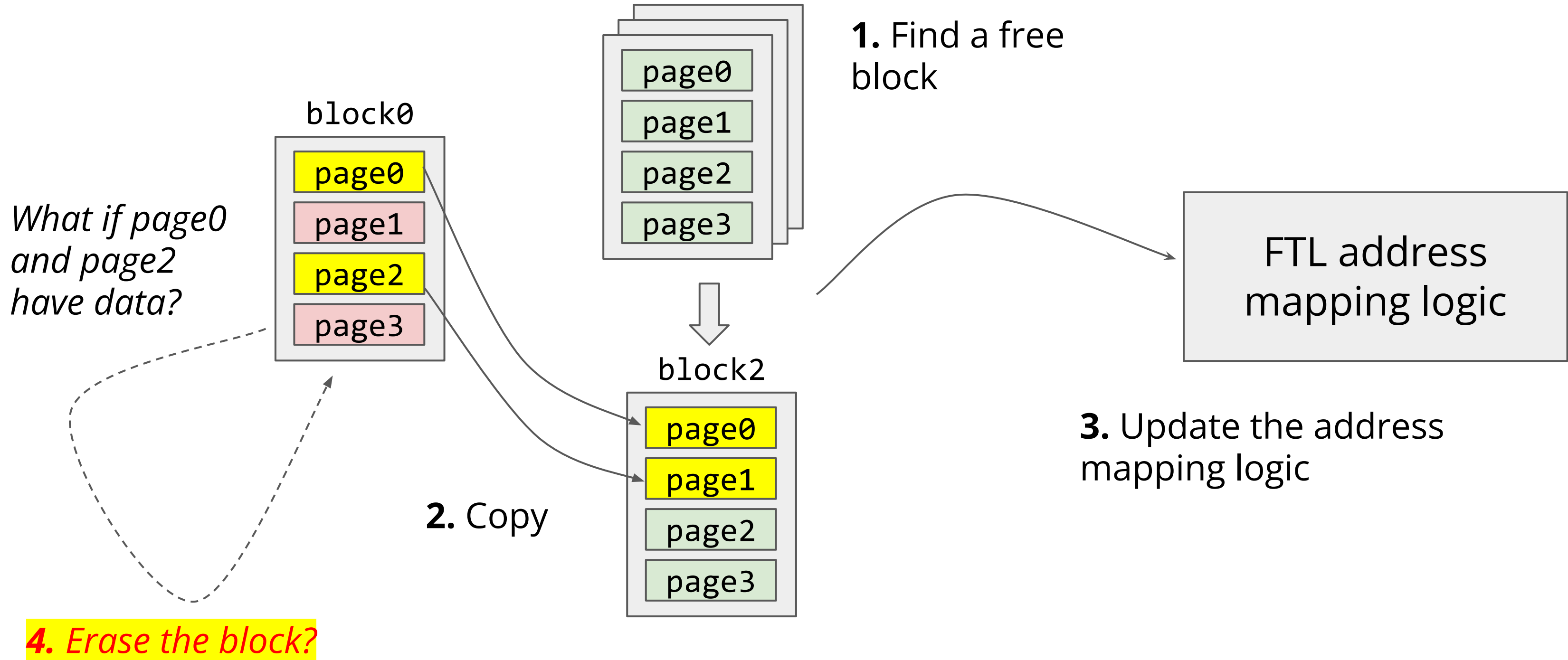


IF the whole block (all pages) contains old data (shown as red) then the FTL can issue the erase command on the block

- Block will be erased, and ready for re-programming
- Put in the pool of free blocks

Easy, right?

What does GC do?



What else should we consider?

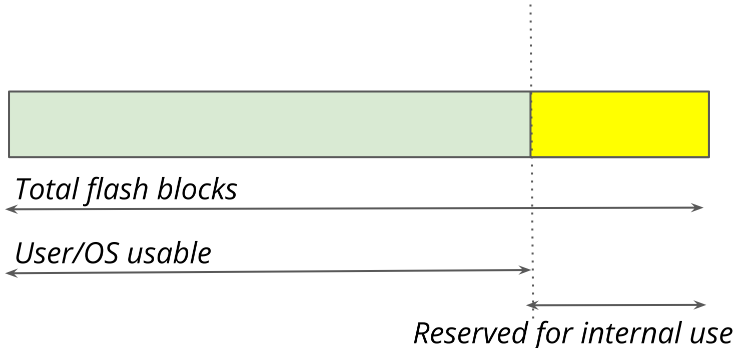
How does GC find free blocks?

If a device is 8GB, and we wrote 8GB of data then where is the free space?

Concept: Over Provisioning (OP)

If a device is 8GB, and we wrote 8GB of data then where is the free space?

Answer: Over Provisioning, have more than what you report (or report less to user/OS)



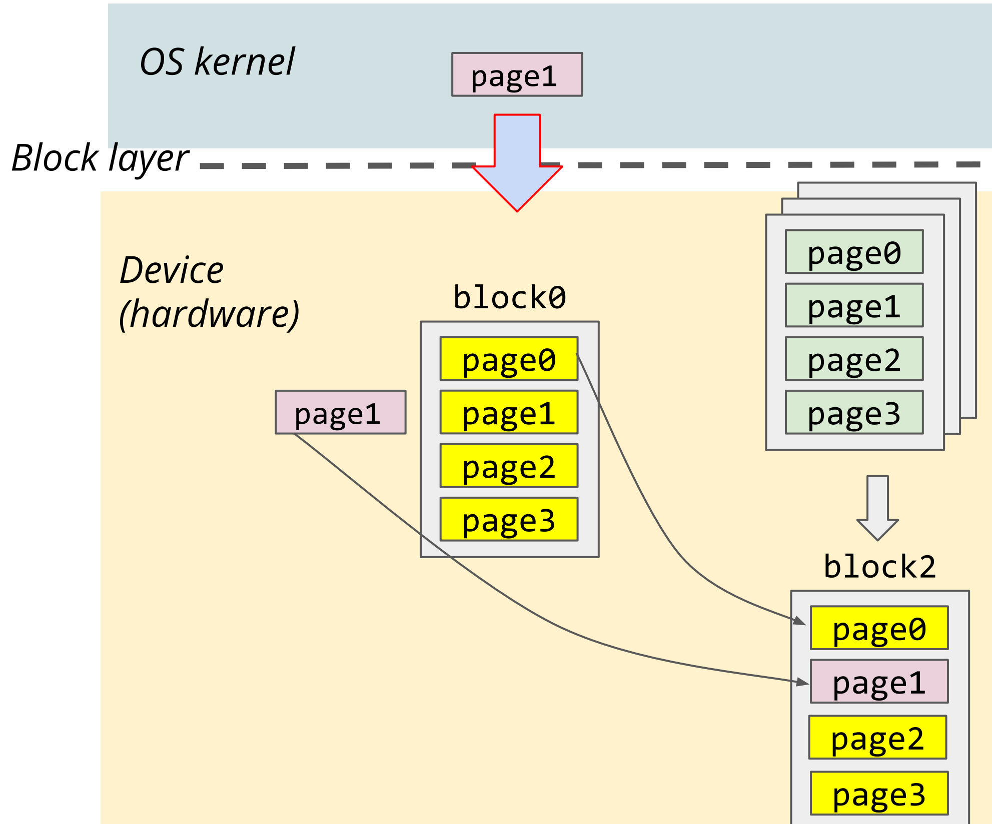
Same over provisioned blocks are also used to manage bad blocks

So in essence, your 16 GB flash drive internally might be 17, 18, or 20 GB flash drive. Sometimes, OP is configurable, other times it is a trade-secret

- **More**: better performance, more space for GC, expensive
- **Low**: cheaper, but less margins for errors

$$\text{Over Provision} = \frac{\text{Total capacity - user capacity}}{\text{User capacity}}$$

Concept: Write Amplification (WA)



What is happening here : **copy-merge-write** cycle

1. The OS/kernel writes 4kB page
2. Internally device has to read 16kB and then write out again 16kB after the modified page

$$\text{Write amplification} = \frac{\text{data written to device}}{\text{data written by user}}$$

*Here amplification is **4x**. The exact value depends upon the page/block size, how many free blocks, FTL design, and many more…*

A device can also expose smaller than the page I/O units like 512 bytes sectors

Concept: Wear Leveling

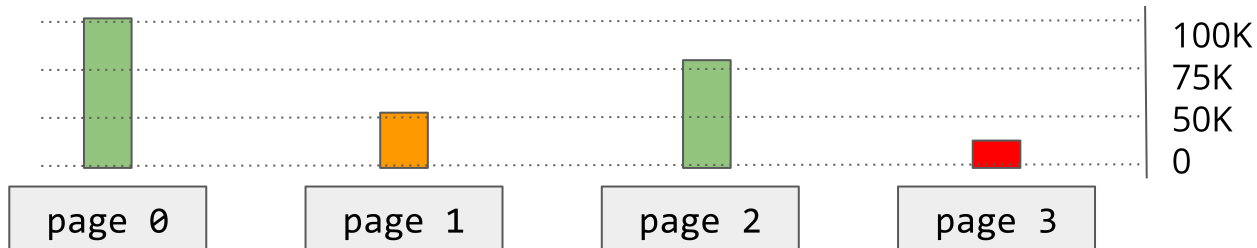
Different flash pages get written and programmed at a different rate

- How frequently they are written
- How frequently they are erased

As a result - their remaining age is different. For SLC chips, 100K is a typical number of cycles available

- Page 0 and 2 are really healthy and not much written
- Page 1 has only half of its PE cycle remaining
- Page 3 has close to death → error rate will increase significantly, and block will be marked bad

An ideal flash drive will try to spread the load evenly across all pages - wear leveling



Concepts: Steady State

When you have a fresh SSD (Fresh out of the Box, of **FoB**):

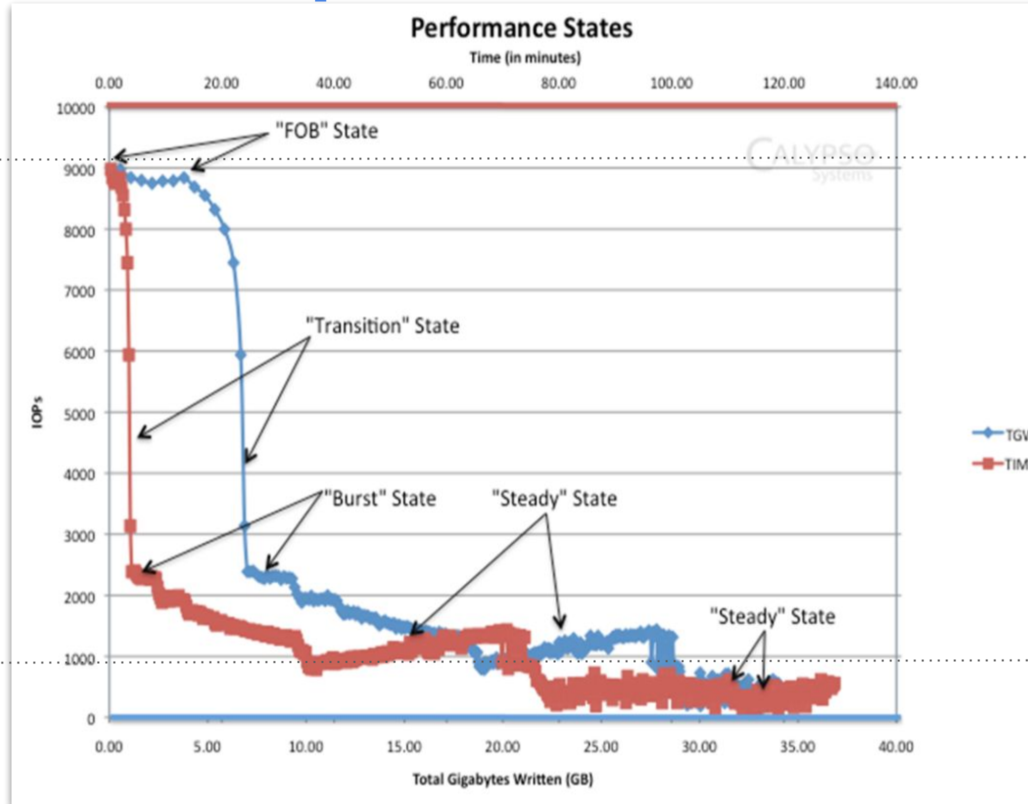
- All pages/blocks are erased
- All pages have full life time
- GC has not done much work
- FTL mappings all empty

If you benchmark your SDD in this state you get a very different performance results, because

- No erasure has been done so far
- No running out of the blocks, need to look for new blocks
- No FTL lookups to update entries

In fact, of you get a FoB flash SSD - it might be the case that you get < *1 useconds* for reading pages and blocks - ***can you guess why?***

Steady State: Impact



Almost A Order Of Magnitude Gap

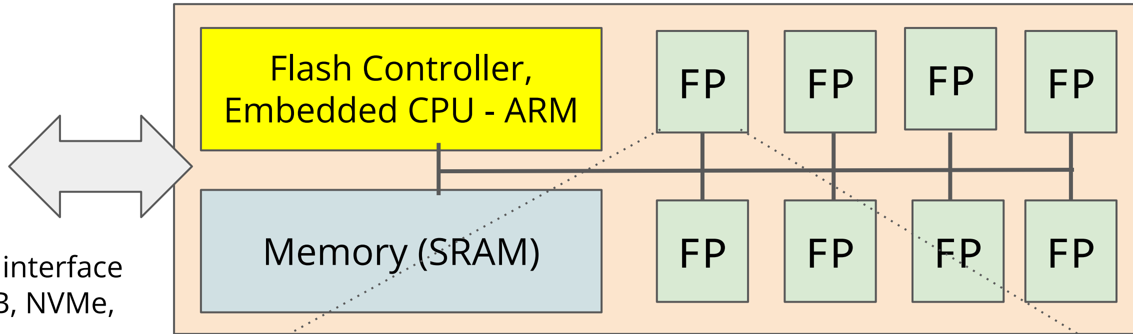
https://www.micron.com/-/media/client/global/documents/products/technical-marketing-brief/ssd_performance_states_tech_brief.pdf

FTL Responsibilities

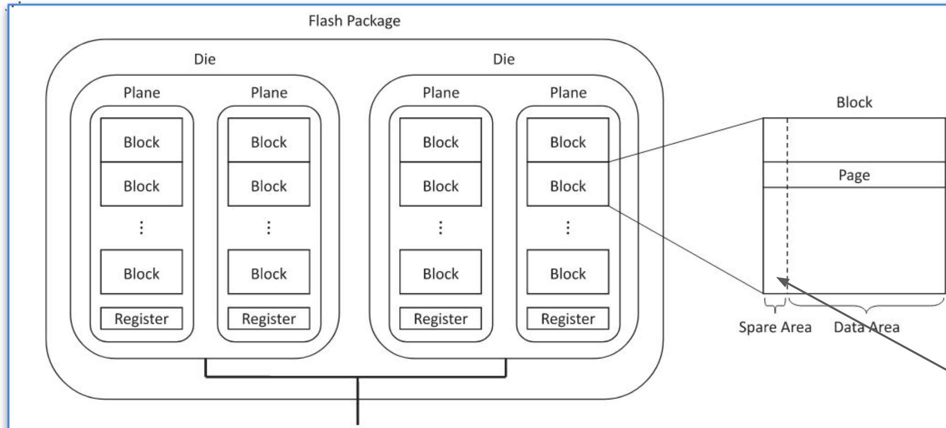
1. Address Translation
   a. Map user visible pages to internal flash locations
2. Garbage Collection
   a. Clear old blocks which do not have “live” data
3. Wear Leveling
   a. Make sure not to “burn” one block and all blocks “age” uniformly
      i. *BTW - is this uniform aging a good idea?*
4. Parallelism and Load Balancing
   a. Use all parallel units for performance
5. Bad Block Management
   a. Error corrections
   b. When blocks are bad (they develop a high read/write error rate, mark them and don’t use)

Also, work in a resource constrained environment with limited CPU power and DRAM

Flash device setup



Systems interface (e.g., USB, NVMe, SCSI)



The embedded CPU runs the FTL logic

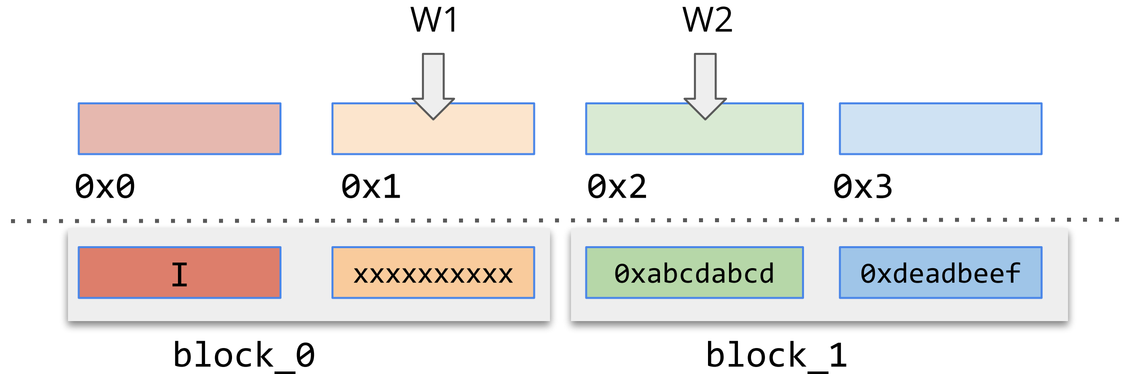
- ***Embedded FTL***

Can use a bit of local memory for staging data

What could be the simplest design you can think of?

Little bit out-of-band data

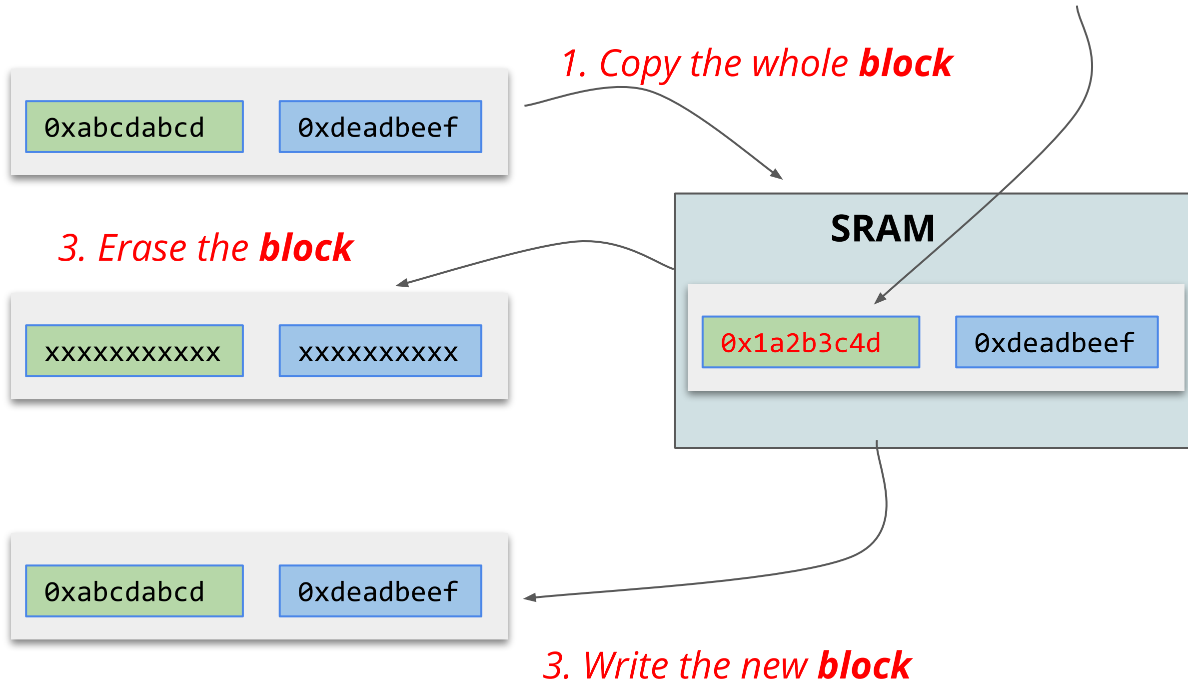
Design 1: A directly mapped FTL



A simple directly mapped FTL

- Trivial look up - fast and simple
- When read comes up for a LPA(x), read the PPA(x)
- When write comes up
    - If the page is in "E" (erased) state then simple, write it out (write1, W1)
    - If the page is in "I" (invalid) or "W" (written) state then the FTL needs to erase the whole block, not just a single page

Design 1: A directly mapped FTL



So, what is the write amplification factor here?

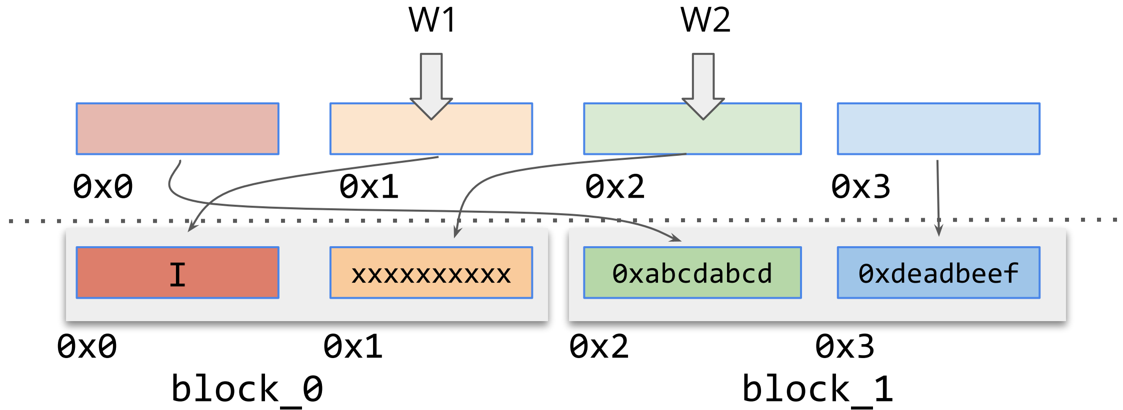
What are the challenges with such directly mapped design?

Issues with the directly mapped FTL

1. **Performance**
   a. Every over-write to a page will result in a full copy-erase-write cycle
   b. Very high **write amplification factor**
      i. Directly proportional to the number of pages in a block
2. **Wear-leveling**
   a. Not all pages are equally popular
   b. Physical pages containing frequently written data (called **HOT data**) will be erased many times, thus **wearing them out**
      i. Every flash page has a finite number of P/E cycle (e.g., 100K for SLC)
      ii. After exhausting their quotas, they will develop high error rates
   c. One way to fix would be to tell the application to write evenly across the whole device, but applications rarely write pages, file systems do
3. **In-place updates**
   a. During a failure, the in SRAM content might be lost

Any advantage of such trivial-design?

Let’s do a Bit Better



LPA	PPA	Valid/Flags
0	0x3	I
1	0x0	E
2	0x1	W
3	0x3	W

Invalid, Written, Erased

Lets keep track of “mappings” in a table, and we can rearrange them when they needs to updated

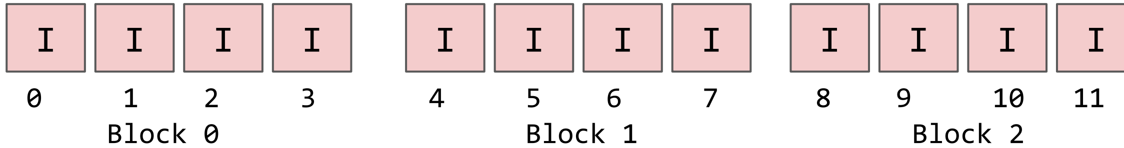
Two types (same idea as **the page tables in the CPU**):

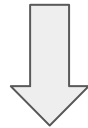
- Directly mapped - from logical to physical mappings - fast lookup (but failure prone)
- Inversely mapped - from physical to logical mappings, easy to reverse lookup, and construct a mappings after failure

A device can use one or mix of these two.

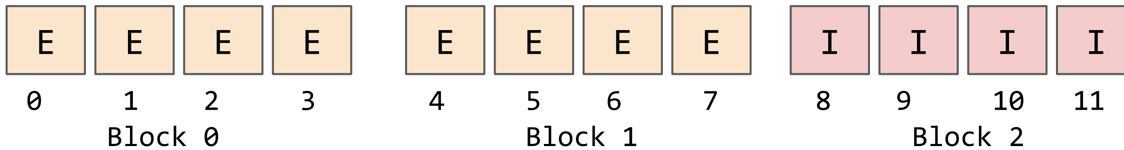
Page-Mapped FTLs

Does tracking of LPA (arbitrary) → PPA (arbitrary)





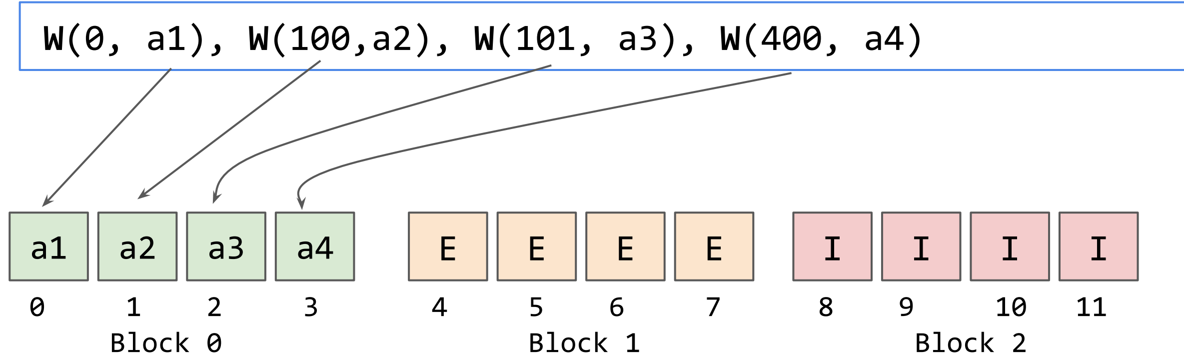
Erase the first two blocks



LPA	PPA	Valid/Flags

Invalid, **W**ritten, **E**rased

Page-Mapped FTLs



LPA	PPA	Valid/Flags
0	*0*	E -> V
100	*1*	E -> V
101	*2*	E -> V
400	*3*	E -> V

Invalid, Written, Erased

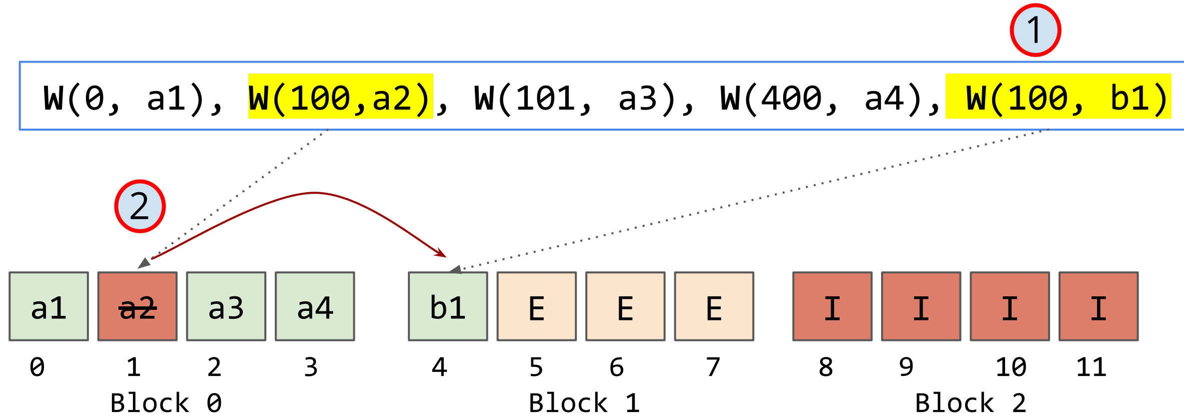
For Write

- As the new content comes in, we fill in the next pages we have around.
- Simple strategy (multiple policies possible regarding the selection of the next free page - not all free pages are created equal …*any guesses*?)

For read

- Lookup in the table the location of PPA and then issue a read to the flash chip

Page-Mapped FTLs : Overwriting



Invalid, Written, Erased

LPA	PPA	Valid/Flags
0	0	V
~~100~~	~~1~~	“I”
101	2	V
400	3	V
100	4	V

When a new content comes in, FTL marks the old location invalid, and choose a new page to write the new content

- If the write was for a full page, then just write to the new location and update the table
- If the writes was less than the page then
  - Read the old content, merge, and then write the merged paged to page 4

This is called **out-of-place writes** - also helps with the failure (as the old content is kept)

The good and bad about Page-level FTLs

The Good

- Most flexible, the best performance, least amount of WA as only a single page is merged

The Bad

- How much memory 1TB SSD need with 4kB pages?
  - Let's assume at least 8 bytes entry per page
  - 1TB / 4kB = 256 million entries
  - In total the size of the FTL table: ~2 GB (this much SRAM in your SSD)
  - We are not even talking about space for other items yet
- So why cannot we put 2 GB of memory in SSDs?
  - Complexity - form factor, electricity, circuitry, power
  - Price, will get super expensive. Memory is expensive

Any idea what can be done here?

Block-Level Mapping

Instead of keeping track of per-page mapping, keep track of **per-block mapping**

Recall: A block is a collect of pages (10-100s of pages), it is the unit of erase

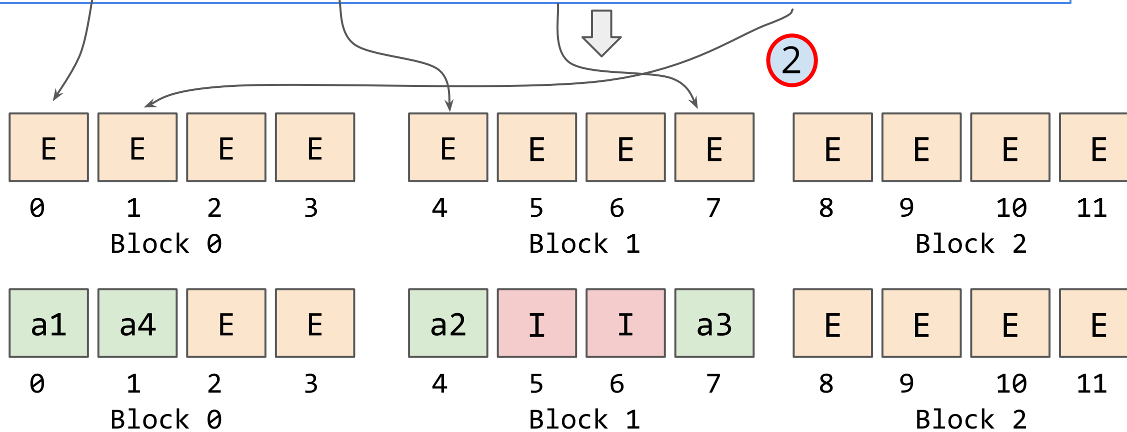
	Block Size	Page Size	OOB Size	# Pages/Block
Small-block SLC	16KB	512 bytes	16 bytes	32
Large-block SLC	128KB	2KB	64 bytes	64
Large-block MLC	512KB	4KB	128 bytes	128

So by what factor, the size of the table will be decreased? Number of pages in a block. So, let's do the calculation again:

- 1 TB / 128 KB x 8 bytes = 64 MB (still large, but manageable)
- SSDs can have 256-512kB blocks.

So this is it then?

Basic working: Reads and Writes



2

LBA	PBA	Valid/Flags
8000	0	V
400	1	V

Invalid, Written, Erased

Nice, small FTL table

Read : find the block level mapping,

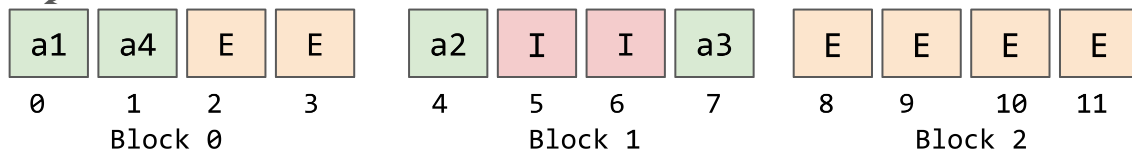
- R(8100) comes in : block address (8000) + page offset (1)
- Lookup PBA of 8000, this is 0
- read(0 + 1) for the content of LPA(8100)

Offset based address calculation ← **Important!**

Challenges with the Block-Level Mappings

Assume a size of 100 bytes for each page, block size 400 bytes

W(8000, a1), W(400,a2), W(700 a3), W(8100, a4), W(8000, b1), W(500, b2)



LBA	PBA	Valid/Flags
8000	0	V
400	1	V

Invalid, Written, Erased

Now what?

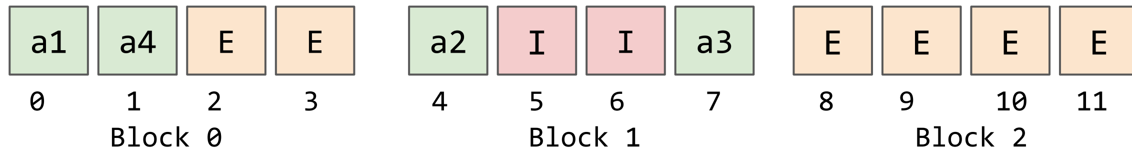
- What to do with W(8000)? This is an **overwrite**
- What to do with W(500)? This is a **write in the middle of a block**

The only thing we can do is to copy, merge, and write out the content of blocks 0 and 1 to a new blocks (block 2, if no more available then erase blocks) → **High Write Amplification factor**

Challenges with the Block-Level Mappings

Assume a size of 100 bytes for each page, block size 400 bytes

`W(8000, a1), W(400,a2), W(700 a3), W(8100, a4), W(8000, b1), W(500, b2)`



LBA	PBA	Valid/Flags
8000	0	V
400	1	V

Invalid, Written, Erased

The key problem is that due to the **offset calculation** the location of a page is fixed inside a block. ***We cannot just choose the next free page inside a block***

- Expensive read-merge-erase-write cycle that we saw with the directly-mapped design

The second problem is dependency on the write pattern → **Very important problem!**

FTL Design Options so far

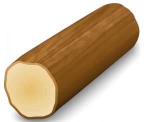
1. Page-level FTLs
   a. Good mapping granularity
   b. Any page can be mapped anywhere, less sensitive to workload patterns
   c. But, large FTL size and need large SRAM in flash
2. Block-level FTLs
   a. Small FTL size
   b. High write amplification factor
   c. Sensitive to workload patterns

There are variants to both page-, and block-level FTLs that helps to mitigate these issues to a certain extend (we are skipping them here)

As usual, the question here is that if we can we do better?

Question: **What is the most important data structure you know?**

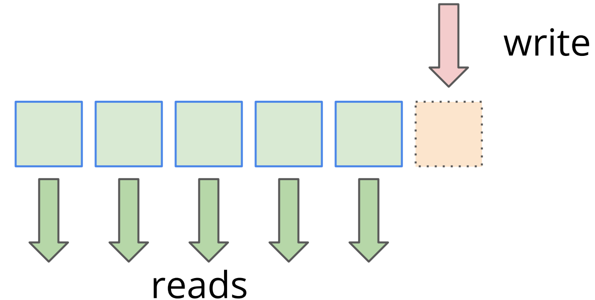
(obviously, it can not be a factual answer, but what do we think?)

hint 

Log: A very powerful data structure

A sequential appending data structure

- Write only at the end (tail)
- Read from anywhere



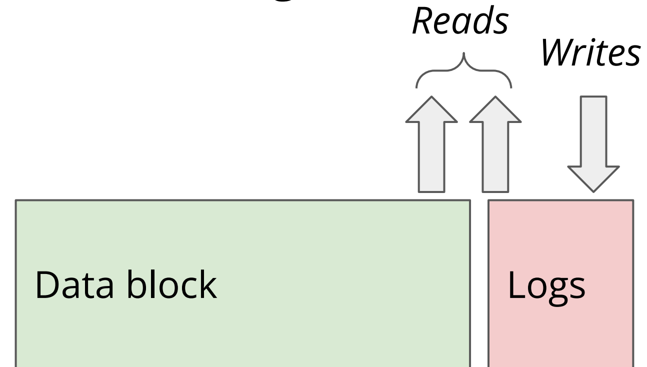
Many unique properties

- No in-place updates, once written, the data becomes immutable
- Serialized writing, one point of writing, the tail - either the write succeeds or not (atomicity)
- Converts a random write to a sequential one ← **very important**
- Parallel reading
- Ordering of events (writes, transactions, or whatever)
- Logging events (used in DBs, file systems) for failure recovery

We will see use of logs in FTLs, flash file system designs, distributed systems. One of the most important data structures around, and it matches NAND flash properties!

Hybrid-Log FTLs

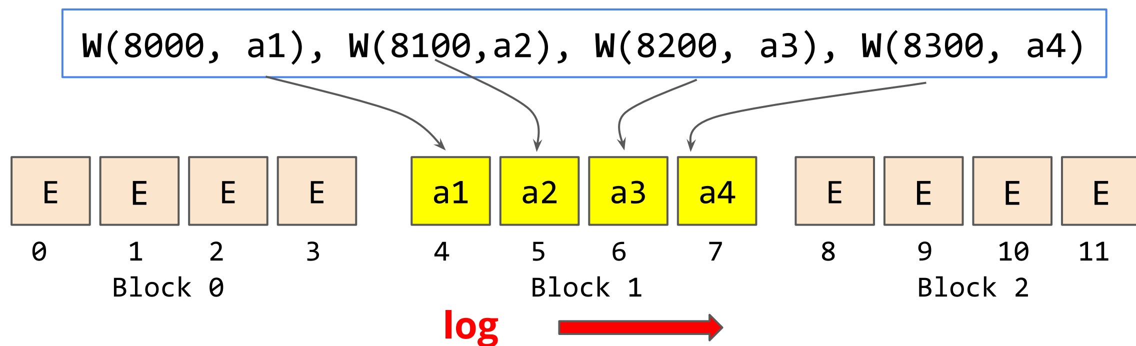
Divide the NAND flash device into two parts: Data and Log



- **Data blocks**
  - Contains the stable data
  - Mapped per-block (or zone) basis
- **Log blocks**
  - All fresh writes go to log pages
  - All filled in a strictly sequential pattern, from low page to higher pages
    Mapped per-page basis

At some point in time data is moved from the log to the data blocks

Working of a Hybrid-Log FTL - Simple Case



Data Block Mappings

LBA	PBA	Valid/Flags

Log Page Mappings

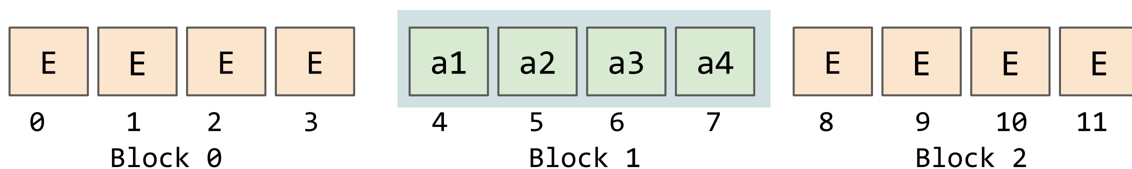
LPA	PPA	Valid/Flags
8000	4	V
8001	5	V
8002	6	V
8003	7	V

A nice sequential pattern

- Block 1 is currently used as a log block which absorbs incoming writes
- Once filled, then we can convert it to a “data” block and map it as a “block-level” entry

Working of a Hybrid-Log FTL - Simple Case

`W(8000, a1), W(8100,a2), W(8200, a3), W(8300, a4)`



Data Block Mappings

LBA	PBA	Valid/Flags
8000	1	v

Log Page Mappings

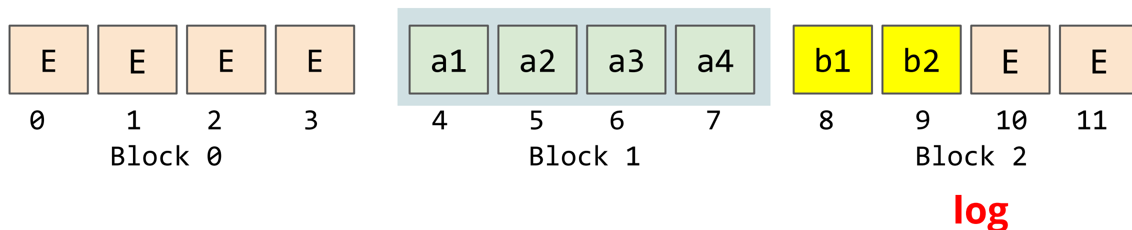
LPA	PPA	Valid/Flags
~~8000~~	~~4~~	~~v~~
~~8001~~	~~5~~	~~v~~
~~8002~~	~~6~~	~~v~~
~~8003~~	~~7~~	~~v~~

A nice sequential pattern

- Block 1 is currently treated as a log block which absorbs incoming writes
- Once filled, then we can convert it to a “data” block and map it as a “block-level” entry

This is called “**Switch Merge**”

Working of a Hybrid-Log FTL - Updates to the Block



Data Block Mappings

LBA	PBA	Valid/Flags
8000	1	v

Log Page Mappings

LPA	PPA	Valid/Flags
8000	8	v
8100	9	v

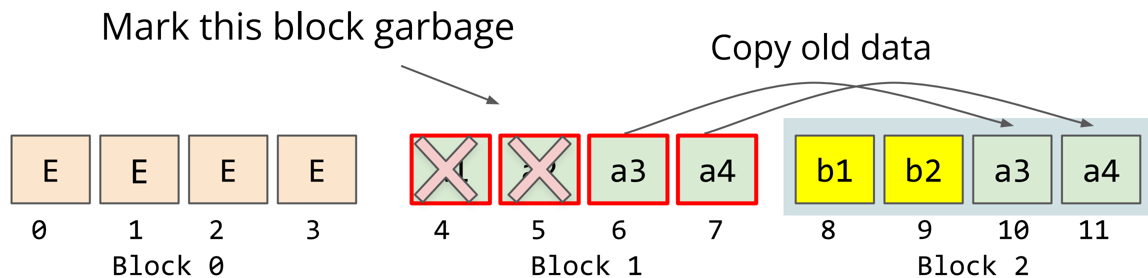
New updates to already written blocks (but still in order)

- Pick up a new "log" block, say block 2
- Use page 8 and 9 to write new values

Read order: first check the Log, and then Data block. (**fresher data is always in the Log**)

Then at some point we can merge the data block 1 and log block 2

Working of a Hybrid-Log FTL - Updates to the Block



Data Block Mappings

LBA	PBA	Valid/Flags
8000	1 → 2	v

Log Page Mappings

LPA	PPA	Valid/Flags
~~8000~~	~~8~~	~~v~~
~~8100~~	~~9~~	~~v~~

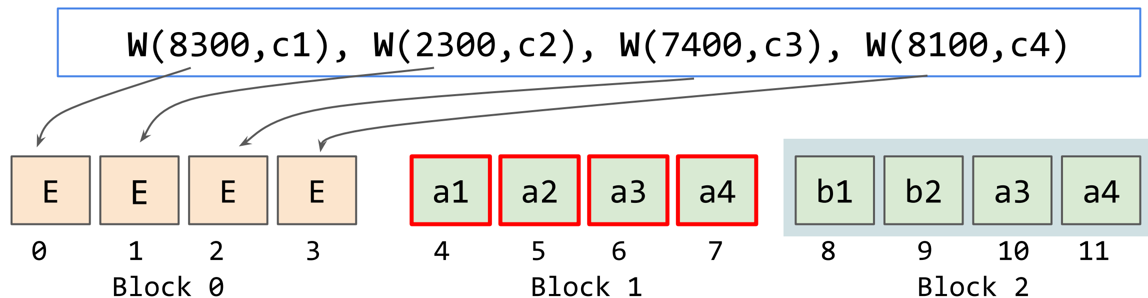
New updates to already written blocks (but still in order)

- Pick up a new "log" block, say block 2
- Use page 8 and 9 to write new values

Then at **some point** we can merge the data block 1 and log block 2

This is called "**Partial Merge**"

Working of a Hybrid-Log FTL - Updates to the Block



Data Block Mappings

LBA	PBA	Valid/Flags
8000	2	v

Log Page Mappings

LPA	PPA	Valid/Flags
8300	0	V
2300	1	v
7400	2	V
8100	3	V

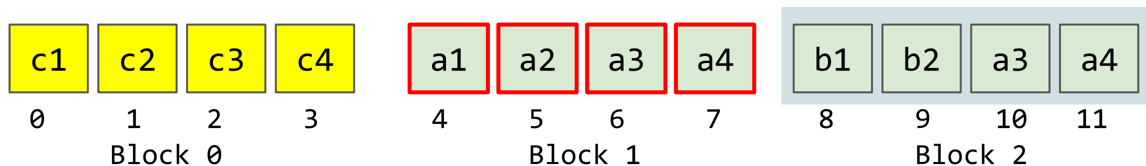
Now completely random writes, take block 0 as the log block

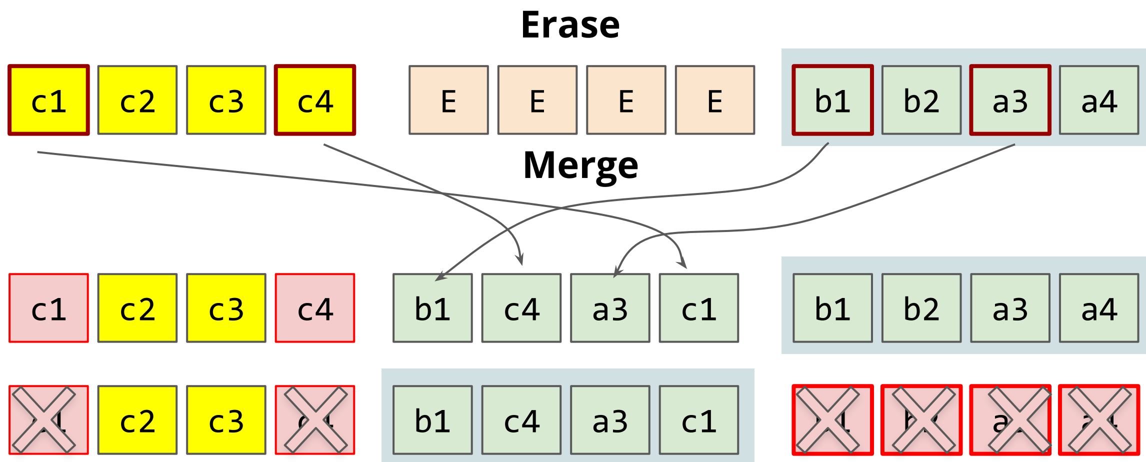
- Multiple writes might have come here
  - Some for already valid entries (like 8000-8400 range)
  - Some for a new writes, previously unmapped

What happens now?

Working of a Hybrid-Log FTL - Random Writes

W(8300,c1), W(2300,c2), W(7400,c3), W(8100,c4)





Data Block Mappings

LBA	PBA	Valid/Flags
8000	2 → 1	v

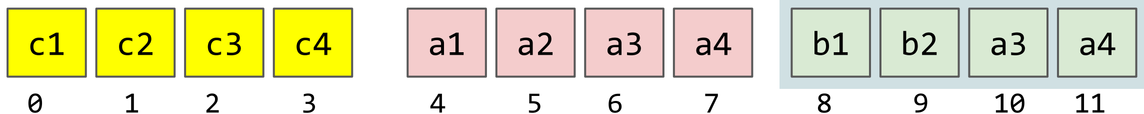
Log Page Mappings

LPA	PPA	Valid/Flags
~~8300~~	~~0~~	~~V~~
2300	1	V
7400	2	V
~~8100~~	~~3~~	~~V~~

Convert from a log to data block

Working of a Hybrid-Log FTL - Random Writes

`W(8300,c1), W(2300,c2), W(7400,c3), W(8100,c4)`

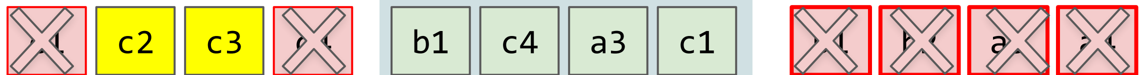


This is called "**Full Merge**"

- Quite expensive
- FTLs try to avoid as much as possible this state

However, it is very difficult to predict

- When is the right time to merge
- What is the upcoming read/write pattern



Convert from a log to data block

Data Block Mappings

LBA	PBA	Valid/Flags
8000	1	v

Log Page Mappings

LPA	PPA	Valid/Flags
~~8300~~	~~0~~	~~V~~
2300	1	V
7400	2	V
~~8100~~	~~3~~	~~V~~

Choices between Data and Log Blocks

Each device can have multiple log blocks…

Which log blocks should track updates from which data blocks?

1. **Block associative:** each data block has its own private log block **(1:1)**
   a. trivial updates and merging (only switch and partial, never full), but 2x capacity waste

2. **Set associative:** “n” log blocks serve writes for a set of consecutive “m” data blocks **(n:m)**
   a. Balance between the merge cost and flexibility

3. **Fully associative:** any log block can track updates from any data blocks **(any:any)**
   a. most flexible, but need full merge
   b. You will be implementing this for project milestones M2/M3

A device can implement a mix of any of these for various workload patterns

So far we have seen

- Read and write access patterns can affect the performance
- Random writes in the middle of a block are bad
  - Leads to partial/full merge (cannot do just switch merge)
- Sequential, large writes are good - more opportunities for the switch merge

The FTL design and data caching (in SRAM) in devices are managed together

- Whenever a data is ready to be flushed out to NAND chips, corresponding FTL entries are also written out (there is a FLUSH command in NVMe)
- One keeps a “working set” of hot data pages and FTL mappings in SRAM memory (like CPU TLBs)

Hence, locality matters with SSDs (*did not we say random and sequential performance is the same for SSDs?)*

Also to consider

Are sequential reads also better than the random ones?

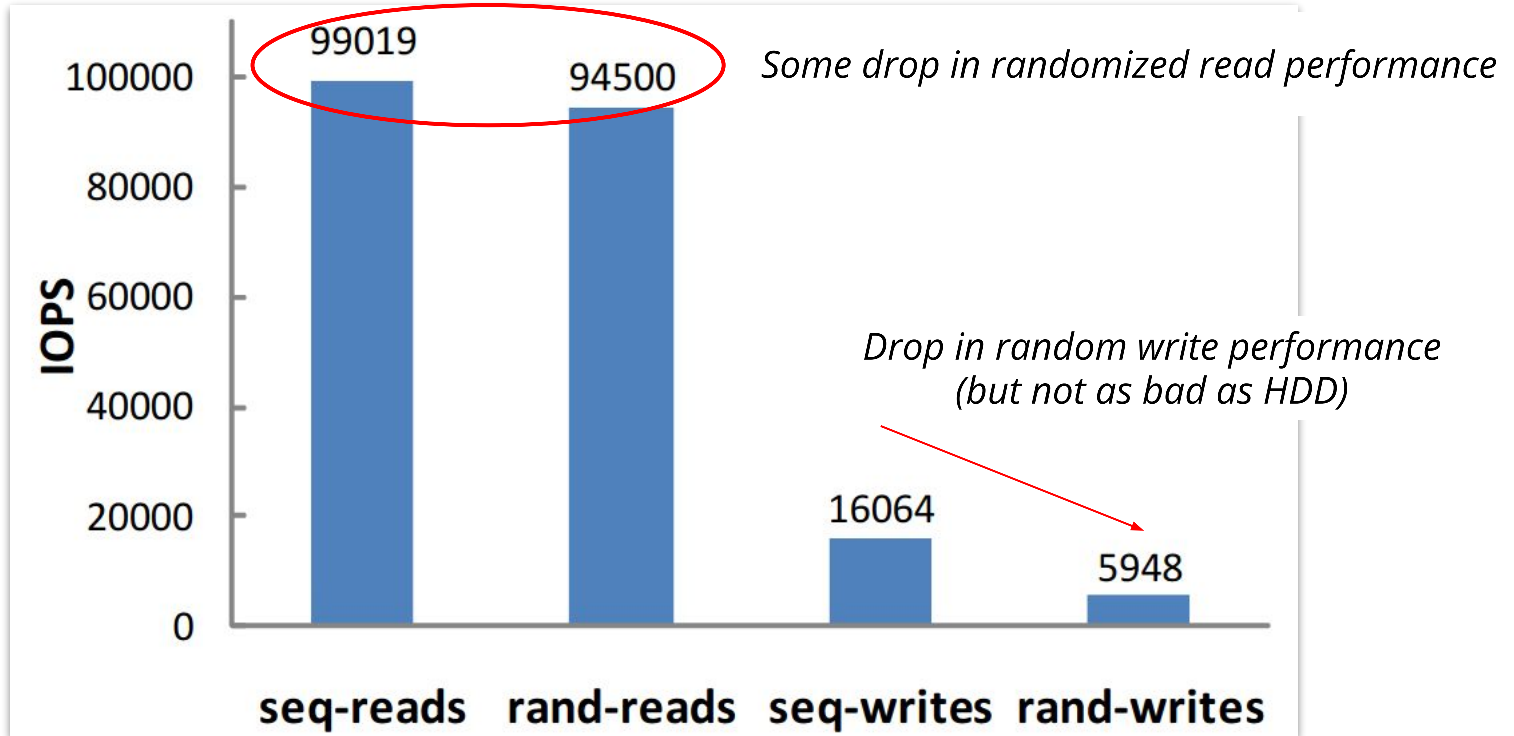
- Because even with the hybrid strategy the size of block-level data table, and the page-level log table can be large
  - They are stored on the flash and brought in demand to SRAM *(caching)*
  - A single block-level entry covers “x” numbers of pages
  - Hence, sequential/close LPAs have faster performance than the random ones

Freshly formatted device (no data) have a very low write/read latency (< 10 usec)

- The device knows there is no old data, write to the best location
- The device knows no data has been written so far, read with zeros
- Even with writes, most writes can be absorbed with on-device SRAM (needs flush)

To benchmark a device, always bring the device first in a steady state

A Typical Example (2016)



https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/VLDB2010-camera.pdf

Garbage Collection

So far we have implicitly discussed that

- A new free block is always available
- Old marked blocks are somehow erased in the due time

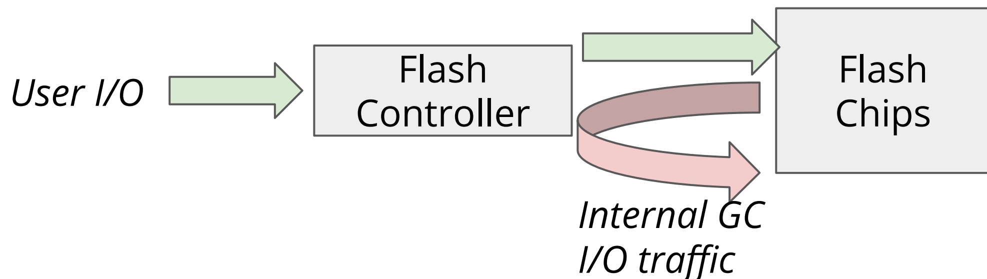
How does this happen? The process of erasing blocks with old data to make them suitable for new data is called **Garbage Collection (GC)**

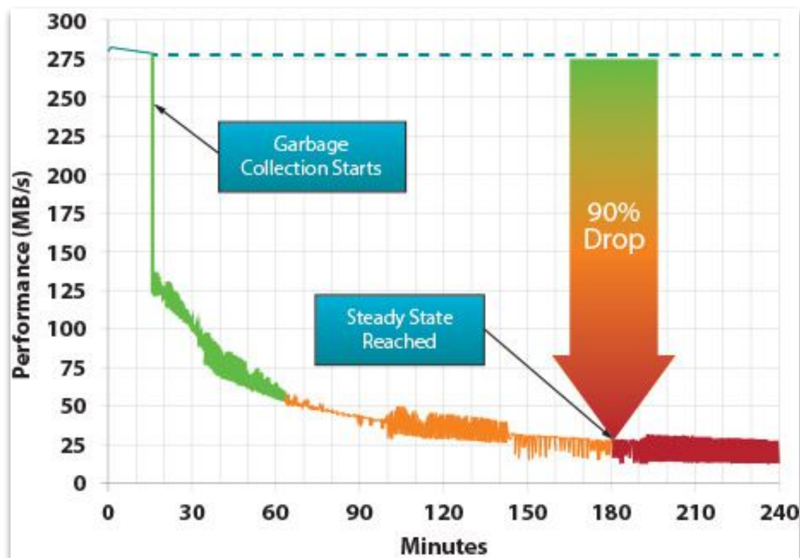
- A block to be erased might contain some live and some expired/dead data

Terminology:

- **Live data** - which is the correct, freshest copy of the data
- **Dead/expired** - old, overwritten or deleted data, which is no longer needed
- **Age of a block/page** - representing the number of P/E cycle a page/block has gone through (recall: there is a finite number of cycles)

Why bother about the GC?



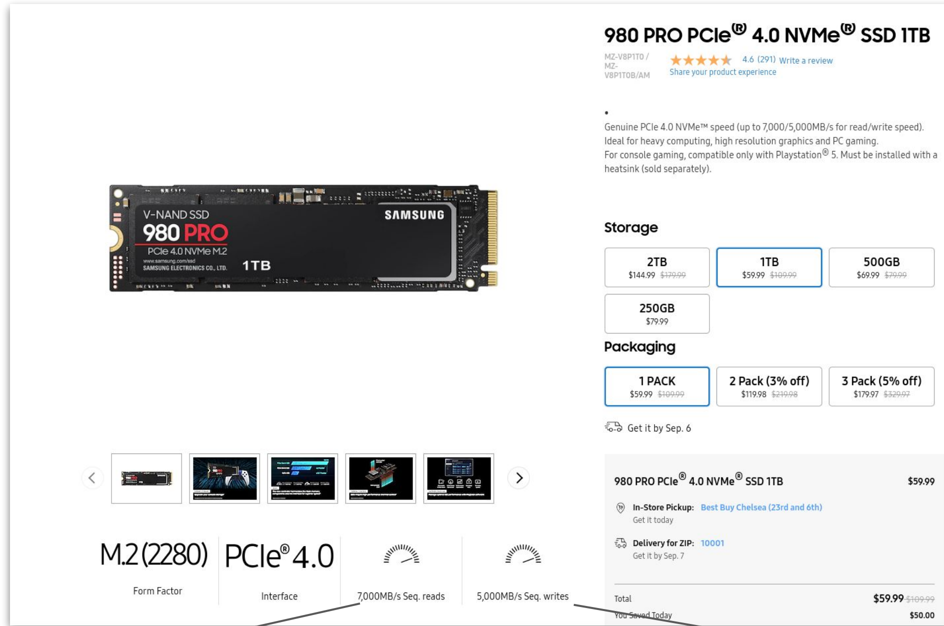


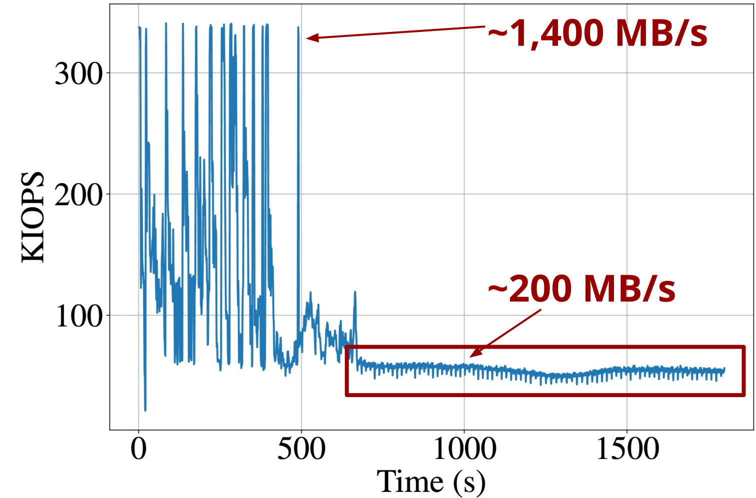
*A flash device has limited resources. If the embedded CPU is occupied running the GC, how is it going to server a user request **(interference)**? Bad impact on **worst case latencies** (95 and 99 percentiles)*

Hence, it is important to understand GC internals and do "**SSD-friendly**" data management on flash devices (see later, **Unwritten Contracts**)

(2017) https://vpsmate.net/lies-damn-lies-ssd-benchmark-test-result1.html

On our cluster



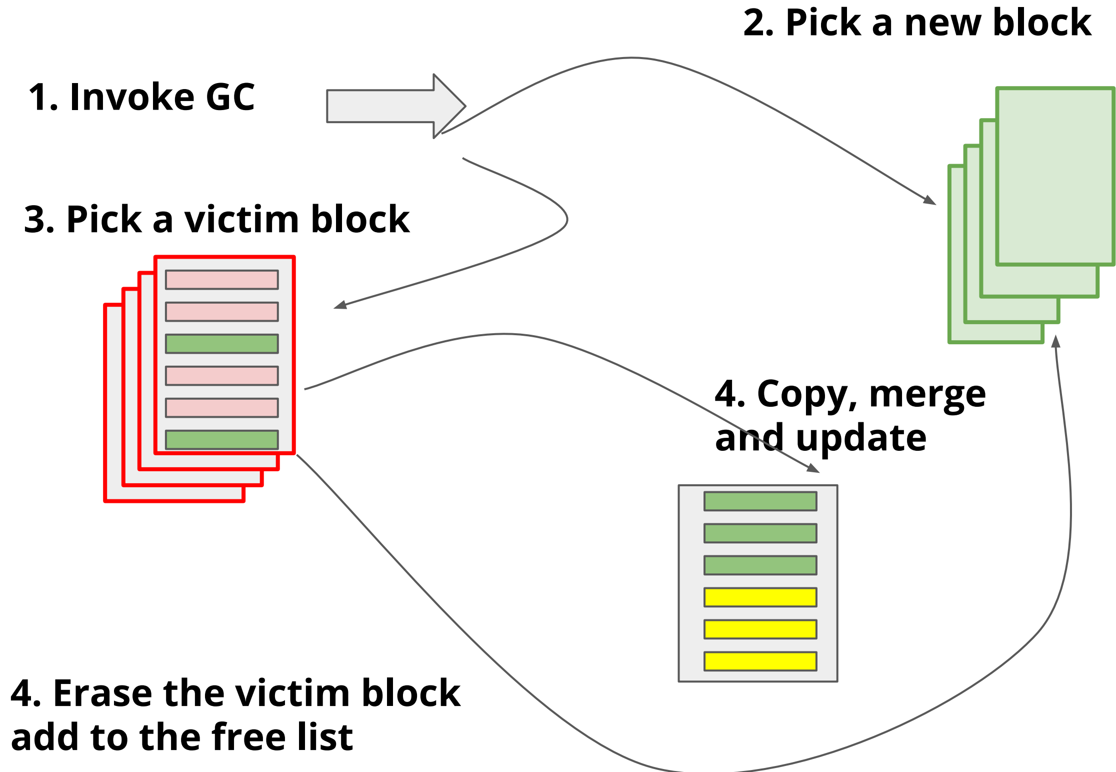


[Seq] **7,000 MB/s** Reads and **5,000 MB/s** Writes

https://www.samsung.com/us/computing/memory-storage/solid-state-drives/980-pro-pcie-4-0-nvme-ssd-1tb-mz-v8p1t0b-am/

(2020) Diego Didona, Nikolas Ioannou, Radu Stoica, and Kornilios Kourtis. 2020. Toward a better understanding and evaluation of tree structures on flash SSDs. Proc. VLDB Endow. 14, 3 (November 2020), 364–377. https://doi.org/10.14778/3430915.3430926

A Typical GC Cycle



Multiple design choices

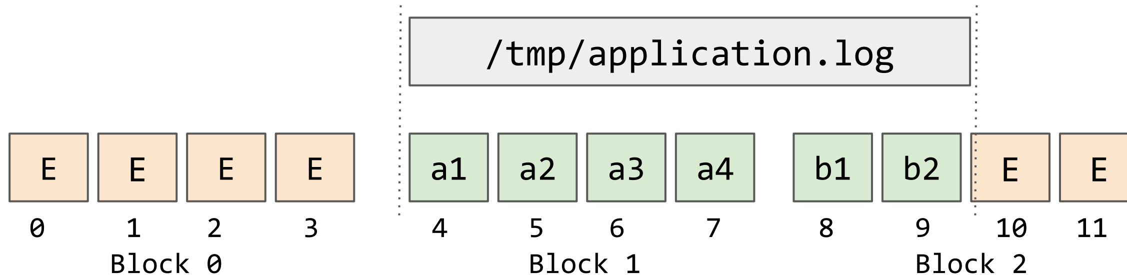
1. How to find live-dead data?
2. When to invoke GC?
3. How to select the victim block(s)?
4. How to minimize interference with the user/system I/O?
5. How to stage data for better GC?
6. How to do high-performance GC?

Identifying “live” vs “dead” data pages

Internally FTL keep track of LPA to PPA mappings

- Every time a new version of the page is written, the previous one can be marked as “dead”

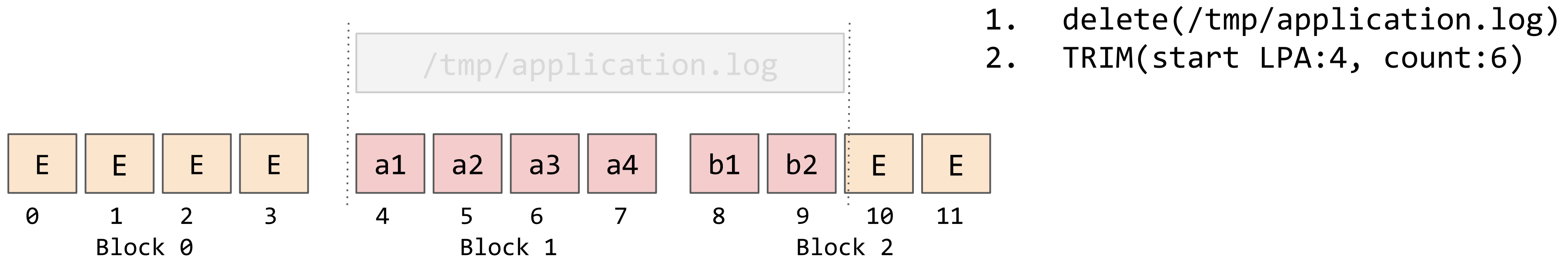
Consider the case below ….



In this case, various files which were created at the systems / file system and application level were created, written to, and then deleted. How does the FTL know about blocks mapped to these files? Technically they contain “dead” data.

Ideas?

TRIM Command (or Deallocate in NVMe)



TRIM: a new command (apart from read/write) for SSDs

- tells an SSD if a range of page addresses are no longer needed
- remove the associated FTL entries internally within the SSD
- mark those pages dead (to be garbage collected)
- can trigger GC to reclaim those pages

When to invoke GC?

As quickly as possible

- interferences with the user I/O request
- contention on shared data and control channels
- useless work if the data written was temporary

As late as possible

- might run out of free blocks
- difficult to handle bursty data (if not free block available)

Typically, during the **idle periods**

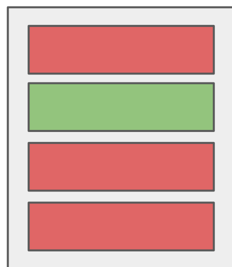
- check the command queue and model when will be the next free window for GC
- when run out of block, then the FTL MUST run the GC (no other option)
- we cannot predict the future write-patterns - *or can we?* (ML, anyone?)

How to Select a Target/Victim Block

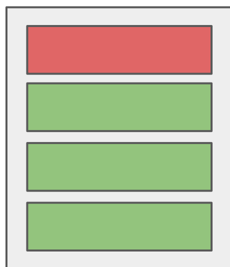
- **Random:** a good and simple strategy if writing load is uniform across the whole device (how often is this the case?)
- **FIFO, RR:** pick the last block that was erased and use it
  - Advantage: uniform wear leveling across all blocks
  - **THINK: do you have a choice when using a LOG?**

Is this good enough? Any other ideas? Or what other criterias should we consider?

Which Block is Preferred?



(a)



(b)

Block (a) contains ¾ dead pages
Block (b) contains ¼ dead pages

Block (a) gives maximum free space for minimum amount of data copies, also reduces write amplification. Pick the one with least amount of live data. **Greedy approach.**

What if the Block (a) was an old block, close to 90K PE cycles (out of 100K) and Block (b) was at 50K? What is a better option now?

Some sort of **cost-benefit Trade-off (CBT)**

- Cost is a function of (amount of work, age, SRAM needed?)
- Benefit is amount of free space reclaimed (and the aging, etc. etc.)

How to Select a Target/Victim Block

- **Random:** a good and simple strategy if writing load is uniform across the whole device (how often is this the case?)
- **FIFO, RR:** pick the last block that was erased and use it
  - Advantage: uniform wear leveling across all blocks
  - Problem: what if the last block contained all “live” data?
- **Greedy:** Pick the block which has the least amount of “live” data
  - Advantage: minimum amount of live data copy, maximum free space reclamation
  - Challenge: what about wear leveling?
- **Cost-Benefit**: various strategies that mix (i) live data amount; (ii) age of the block; (iii) amount of memory required for state keeping;

As you can see there is not strategy that wins all. Modern device FTLs use a combinations of these strategies

At this point

1. FTL seems complicated, and there is no single win strategy for all
    a. Hidden from the systems and applications ***(no SSD API to talk to it)***
2. GC seems complicated and there is no single win strategy for all
    a. Hidden from the systems and applications (no API to talk to it)
3. Both of them are resource heavy
    a. Need CPU power
    b. Need DRAM/SRAM on board
4. A large design space with multiple objectives
    a. Performance, very much workload dependent
    b. Wear-leveling, minimize interference, binning and grouping of data

Is there is a better way to tackle these challenges ...

Host-Based FTLs

Implement all the complex logic on host, why?

- Host has a powerful CPUs (multicore Intel CPUs)
- Host has a lot of DRAM (10-100s of GB)
- All GC and user traffic is visible to the host FTL, no hidden state
- Application-specific customization possible, no need to restrict yourself on the Block interface

Examples: OpenChannel SSDs and LightNVM infrastructure, Baidu's Software-defined Flash, and now NVMe Zone namespace (ZNS) devices

- ZNS devices do not have complete FTL, but the control over the GC at the host
- Another emerging standard is NVMe Flexible Data Placement **(FDP)**
  - https://nvmexpress.org/nvmeflexible-data-placement-fdp-blog/

On device FTLs are called "**Embedded FTLs**" vs "**Host-based FTLs**"

Baidu’s Software Defined Flash

SDF: Software-Defined Flash for Web-Scale Internet Storage Systems

Jian Ouyang Shiding Lin
Baidu, Inc.
{ouyangjian, linshiding}@baidu.com

Song Jiang *
Peking University and
Wayne State University
sjiang@wayne.edu

Zhenyu Hou Yong Wang
Yuanzheng Wang
Baidu, Inc.
{houzhenyu, wangyong03, wangyuanzheng}@baidu.com

Abstract

In the last several years hundreds of thousands of SSDs have been deployed in the data centers of Baidu, China’s largest Internet search company. Currently only 40% or less of the raw bandwidth of the flash memory in the SSDs is delivered by the storage system to the applications. Moreover, because of space over-provisioning in the SSD to accommodate non-sequential or random writes, and additionally, parity coding across flash channels, typically only 50-70% of the raw capacity of a commodity SSD can be used for user data. Given the large scale of Baidu’s data center, making the most effective use of its SSDs is of great importance. Specifically, we seek to maximize both bandwidth and usable capacity.

To achieve this goal we propose *software-defined flash* (SDF), a hardware/software co-designed storage system to maximally exploit the performance characteristics of flash memory in the context of our workloads. SDF exposes individual flash channels to the host software and eliminates space over-provisioning. The host software, given direct access to the raw flash channels of the SSD, can effectively organize its data and schedule its data access to better realize the SSD’s raw performance potential.

Currently more than 3000 SDFs have been deployed in Baidu’s storage system that supports its web page and image repository services. Our measurements show that SDF can deliver approximately 95% of the raw flash bandwidth and provide 99% of the flash capacity for user data. SDF increases I/O bandwidth by 300% and reduces per-GB hardware cost by 50% on average compared with the commodity-SSD-based system used at Baidu.

* This work was performed during his visiting professorship at Peking University.

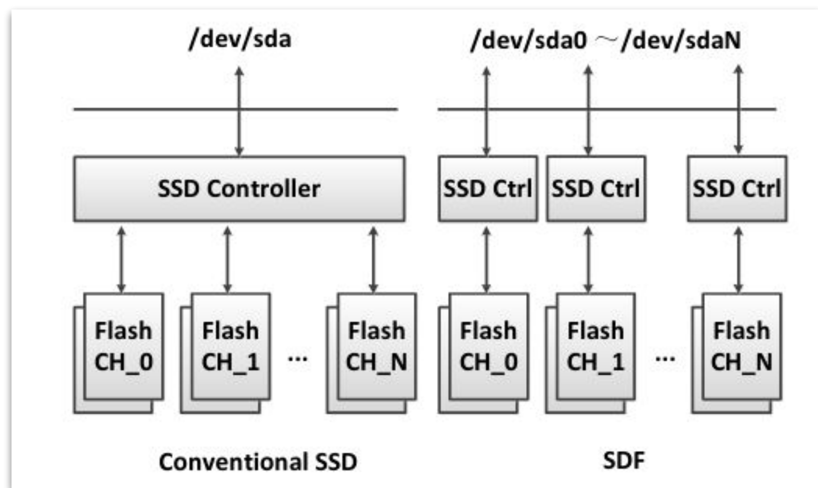
Categories and Subject Descriptors B.3.2 [*Memory Structures*]: Design Styles - mass storage (e.g., magnetic, optical, RAID)

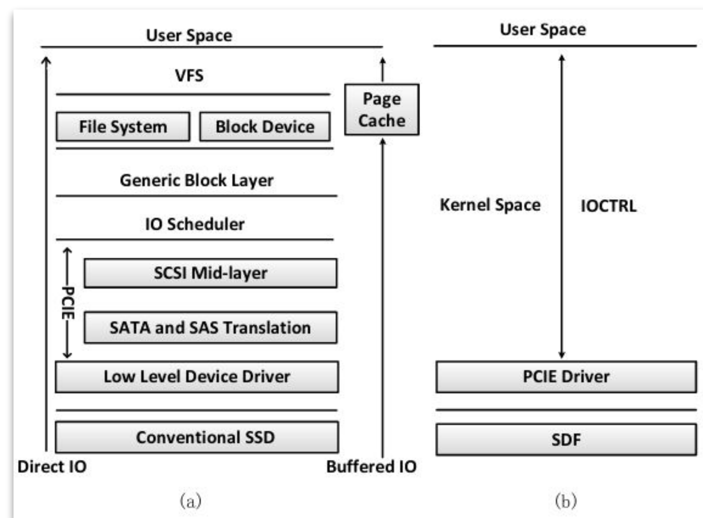
Keywords Solid-State Drive (SSD), Flash Memory, Data Center.

1. Introduction

To accommodate ever-increasing demand on I/O performance in Internet data centers, flash-memory-based solid-state drives (SSDs) have been widely deployed for their high throughput and low latency. Baidu was one of the first large-scale Internet companies to widely adopt SSDs in their storage infrastructures and has installed more than 300,000 SSDs in its production system over the last seven years to support I/O requests from various sources including indexing services, online/offline key-value storage, table storage, an advertisement system, mySQL databases, and a content delivery network. Today SSDs are widely used in data centers, delivering one order of magnitude greater throughput, and two orders of magnitude greater input/output operations per second (IOPS), than conventional hard disks. Given SSD’s much higher acquisition cost per unit capacity, achieving its full performance and storage potential is of particular importance, but we have determined that both raw bandwidth and raw storage capacity of commodity SSDs, in a range of performance categories, are substantially under-

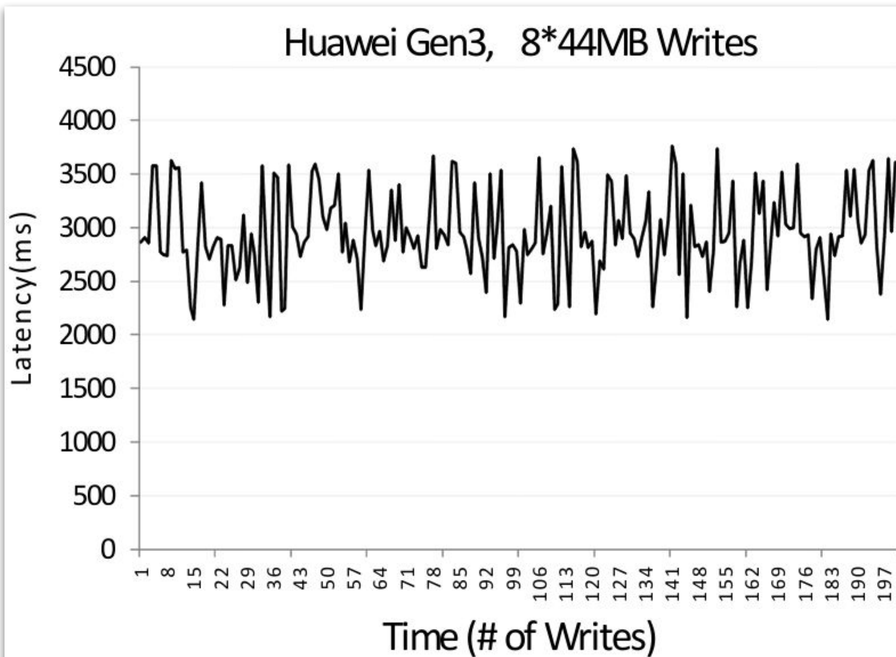
Software-Defined Flash Architecture

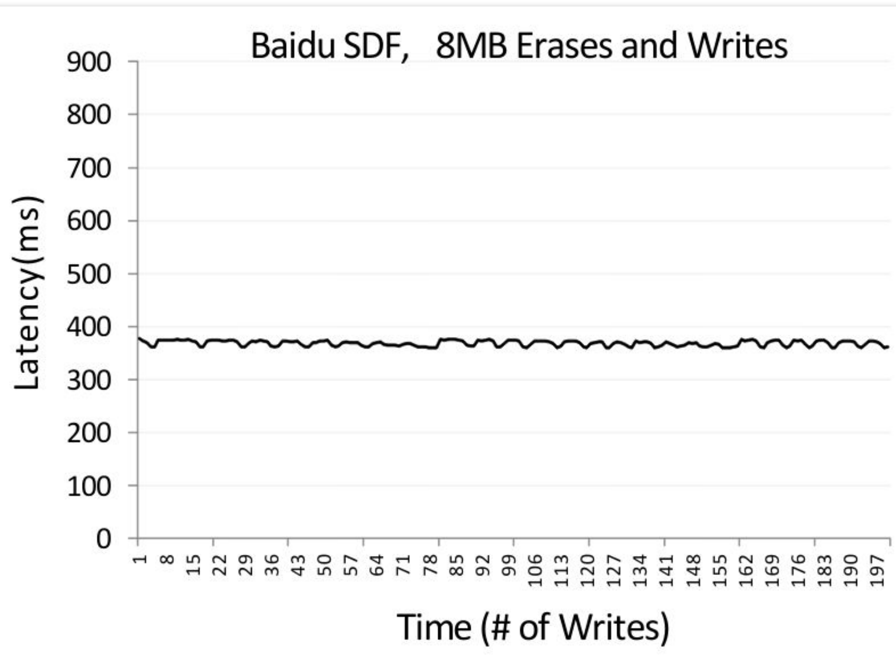




- Full control over I/O Stack (exposed parallelism)
- Full control over GC (reduce interference with user I/O)
- Full control over device provisioning, can be customized for workloads
- Predictable Performance when to do read/write/erase

SDF: Performance Example





The Idea itself is Not New

Fusion-IO (and other companies, Violin Memory) did it back in 2007, run FTL on the host-CPU

Multiple embedded systems do the same, FTL runs on their embedded CPU which is running the application

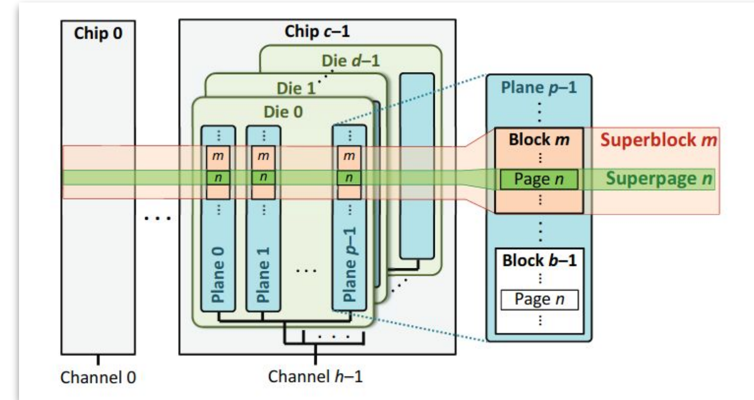
Scale and deployment of execution is unique in the Baidu's case, but many cloud-scale vendors do that now

Pretty cool and influential work

- The idea of opening up flash is used in multiple projects, Open-Channel SSDs, streaming API, Application-managed flash, and now Zone Namespace (ZNS) devices

Design of FTL is a very large field

What we are not covering yet



1. How to extract parallelism… (allocation)
   a. For example, if reading 1MB from a device, it would be nice if its blocks and pages were distributed across parallel dies for performance
      i. **Striping:** horizontal, vertical, n-dimensionals
2. Wear-leveling
   a. Static vs Dynamic wear leveling
   b. Keeping track of block ages
   c. Keeping track of hot and cold data and separate them into various logical “age bins”
3. Failure and bad/corrupted block management
   a. ECC checks and corrections
   b. Maintenance of bad block maps
   c. See, Errors in Flash-Memory-Based Solid-State Drives: Analysis, Mitigation, and Recovery, https://arxiv.org/abs/1711.11427

Further reading

Fantastic SSD Internals and How to Learn and Use Them

Nanqinqin Li, University of Chicago and Princeton University
Mingzhe Hao, University of Chicago
Huaicheng Li, University of Chicago and Carnegie Mellon University
Xing Lin, NetApp
Tim Emami, NetApp
Haryadi S. Gunawi, University of Chicago

ABSTRACT

This work presents (a) Queenie, an application-level tool that can automatically learn 10 internal properties of block-level SSDs, (b) Kelpie, the learning and analysis results of running Queenie on 21 different SSD models from 7 major SSD vendors, and (c) Newt, a set of storage performance optimization examples that use the learned properties. By bringing numerous observations and unique findings, this work exposes substantial improvement spaces for both SSD users and vendors, enlightening possibilities of unleashing more SSD performance potential and highlighting the necessity of further exploring SSD internals.

CCS CONCEPTS

• **General and reference → Empirical studies**; *Measurement*; • **Information systems → Flash memory**.

KEYWORDS

Solid-State Drive, Performance Characterization

ACM Reference Format:
Nanqinqin Li, Mingzhe Hao, Huaicheng Li, Xing Lin, Tim Emami, and Haryadi S. Gunawi. 2022. Fantastic SSD Internals and How to Learn and Use Them. In *The 15th ACM International Systems and Storage Conference (SYSTOR '22), June 13–15, 2022, Haifa, Israel.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3534056.3534940

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SYSTOR '22, June 13–15, 2022, Haifa, Israel
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-9380-5/22/06…$15.00
https://doi.org/10.1145/3534056.3534940

1 INTRODUCTION

Solid-State Drives (SSDs) are a cornerstone of modern storage systems because of their competitive performance, reliability, capacity, and cost [2, 13, 19, 33, 41, 46]. However, while fulfilling user's increasing demands on storage, modern SSDs also bring their own challenges: it is difficult to optimally utilize them as most of them show up as black-box devices, with internal complexities such as FTL mapping, write buffer management, and garbage collection mechanisms, hidden and intangible from their users [21, 23, 26, 29, 49, 50].

These complexities, unfortunately, can bring non-negligible side-effects, with performance inconsistency as one of the notorious ramifications. For example, write buffer flush can contend with reads on NAND resources and bring long latency tails [10, 19, 39]; reads with inappropriate alignment can take extra overhead to process as SSDs apply minimal unit of access [31, 32]; some SSDs are designed for certain purposes, and when used inappropriately, can dramatically downgrade the overall system performance [5, 30].

Motivated to resolve these negative impacts, multiple pieces of prior work [12, 28, 30–32] try to extract crucial SSD properties and propose coherent designs based on the observations. They have reasonably argued that probing SSDs can help build more effective solutions and bring significant performance improvement.

Based on these gains, we further ask: is there more knowledge hidden in modern SSDs, waiting to be learned and utilized, especially as modern SSDs have evolved rapidly over the past decade? We found that there are many questions unanswered in prior work. Do modern SSDs have favorable sizes on reads and punish those that do not comply (§4.1, §5.1)? Do components that were prevalent in SSDs previously, such as read buffer, still exist in recent SSD models (§4.5)? Do large-capacity SSDs, which are very common nowadays, have write buffers of appropriate sizes (§4.2) and the capability to handle highly-parallel writes (§4.4)? Do SSDs apply hybrid (externally and internally triggered) buffer flush policies (§4.3) that can be exploited for less contention and better performance (§5.2)? Do SSDs really perform better when they face less "stress" (§4.6)?

<table>
<tr><th></th><th>P_1 PgSz</th><th>P_2 PgType</th><th>P_3 ChukSz</th><th>P_4 StripeW</th><th>P_5 Layout</th><th>P_6 ReadC</th><th>P_7 RBuf</th><th>P_8 WBuf</th><th>P_9 WrPra</th><th>P_{10} FluWin</th></tr>
<tr><td>$N_{1.6T}S$</td><td>4K</td><td>$MLC_{4L\,2H}$</td><td>4K</td><td>124</td><td>16×8</td><td>✓</td><td>—</td><td>$40.25M_p$</td><td>8</td><td>50ms</td></tr>
<tr><td>$N_{500G}S$</td><td>4K</td><td>$MLC_{4L\,2H}$</td><td>4K</td><td>64</td><td>4×16</td><td>✓</td><td>—</td><td>2M</td><td>1</td><td>4ms</td></tr>
<tr><td>$N_{128G}S$</td><td>4K</td><td>$MLC_{4L\,2H}$</td><td>4K</td><td>32</td><td>16×2</td><td>✓</td><td>—</td><td>1M</td><td>1</td><td>2ms</td></tr>
<tr><td>$N_{2T}I$</td><td>4K</td><td>TLC</td><td>64K</td><td>186</td><td>12×16</td><td>✗</td><td>—</td><td>11.5M</td><td>4</td><td>200ms</td></tr>
<tr><td>$N_{1.6T}I$</td><td>4K</td><td>$MLC_{1L\,1H}$</td><td>128K</td><td>122</td><td>32×4</td><td>✓</td><td>—</td><td>11.5M</td><td>2</td><td>10ms</td></tr>
<tr><td>$N_{1T}I$</td><td>4K</td><td>TLC</td><td>256K</td><td>?</td><td>?</td><td>✗</td><td>—</td><td>$11M_p$</td><td>2</td><td>3ms</td></tr>
<tr><td>$N_{1.6T}W$</td><td>4K</td><td>$MLC_{1L\,1H}$</td><td>64K</td><td>124</td><td>16×8</td><td>✗</td><td>—</td><td>NB+P</td><td>4✗</td><td>—</td></tr>
<tr><td>$N_{1.6T}M$</td><td>4K</td><td>$MLC_{1L\,1H}$</td><td>128K</td><td>128</td><td>16×8</td><td>✓</td><td>—</td><td>15M</td><td>4</td><td>∞</td></tr>
<tr><td>$A_{1.6T}P$</td><td>4K</td><td>TLC</td><td>16K</td><td>247</td><td>16×16</td><td>✓</td><td>—</td><td>NB+P</td><td>4✗</td><td>—</td></tr>
<tr><td>$A_{960G}P_S$</td><td>4K</td><td>$MLC_{4L\,4H}$</td><td>4K</td><td>64</td><td>32×2</td><td>✓</td><td>—</td><td>NB+P</td><td>4✗</td><td>—</td></tr>
<tr><td>$A_{960G}P_T$</td><td>4K</td><td>$MLC_{1L\,1H}$</td><td>16K</td><td>200</td><td>20×10</td><td>✓</td><td>—</td><td>$11.5M_p$ | 406M</td><td>4✗</td><td>200ms | 2.5s</td></tr>
<tr><td>$A_{800G}P$</td><td>8K</td><td>$MLC_{1L\,1H}$</td><td>32K</td><td>128</td><td>16×8</td><td>✓</td><td>16M</td><td>2M | 512M</td><td>4</td><td>∞</td></tr>
<tr><td>$A_{800G}G$</td><td>4K</td><td>$MLC_{1L\,1H}$</td><td>64K</td><td>30</td><td>8×4</td><td>✓</td><td>—</td><td>3.75M</td><td>2</td><td>30ms</td></tr>
<tr><td>$A_{200G}H$</td><td>8K</td><td>SLC</td><td>8K</td><td>253</td><td>16×16</td><td>✓</td><td>—</td><td>20M | 126.5M†</td><td>1</td><td>∞</td></tr>
<tr><td>$T_{480G}S$</td><td>4K</td><td>$MLC_{4L\,4H}$</td><td>4K</td><td>16</td><td>8×2</td><td>✓</td><td>—</td><td>2M | 256M</td><td>1</td><td>10ms | 5s</td></tr>
<tr><td>$T_{200G}S$</td><td>8K</td><td>SLC</td><td>8K</td><td>64</td><td>8×8</td><td>✓</td><td>—</td><td>1M</td><td>1</td><td>35ms</td></tr>
<tr><td>$T_{128G}S$</td><td>4K</td><td>$MLC_{4L\,4H}$</td><td>4K</td><td>32</td><td>16×2</td><td>✓</td><td>—</td><td>1M</td><td>1</td><td>2ms</td></tr>
<tr><td>$T_{100G}S$</td><td>8K</td><td>SLC</td><td>8K</td><td>8</td><td>8×1</td><td>✓</td><td>—</td><td>512K</td><td>1</td><td>40ms</td></tr>
<tr><td>$T_{64G}S$</td><td>16K</td><td>SLC</td><td>—</td><td>1</td><td>1×1</td><td>✓</td><td>—</td><td>4M</td><td>2</td><td>∞</td></tr>
<tr><td>$T_{64G}I$</td><td>4K</td><td>SLC</td><td>4K</td><td>20</td><td>10×2</td><td>✓</td><td>—</td><td>$10M_p$ | 810M</td><td>1</td><td>300ms | ∞</td></tr>
<tr><td>$T_{200G}M$</td><td>4K</td><td>SLC</td><td>4K</td><td>128</td><td>8×16</td><td>✓</td><td>—</td><td>64M</td><td>1</td><td>1s</td></tr>
</table>

Nanqinqin Li, Mingzhe Hao, Huaicheng Li, Xing Lin, Tim Emami, and Haryadi S. Gunawi. 2022. **Fantastic SSD internals and how to learn and use them**. In Proceedings of the 15th ACM International Conference on Systems and Storage (SYSTOR '22). Association for Computing Machinery, New York, NY, USA, 72–84. https://doi.org/10.1145/3534056.3534940

Based on all these details : Unwritten Contract

The Unwritten Contract of Solid State Drives

Jun He Sudarsun Kannan Andrea C. Arpaci-Dusseau Remzi H. Arpaci-Dusseau

Department of Computer Sciences, University of Wisconsin–Madison

Abstract

We perform a detailed vertical analysis of application performance atop a range of modern file systems and SSD FTLs. We formalize the "unwritten contract" that clients of SSDs should follow to obtain high performance, and conduct our analysis to uncover application and file system designs that violate the contract. Our analysis, which utilizes a highly detailed SSD simulation underneath traces taken from real workloads and file systems, provides insight into how to better construct applications, file systems, and FTLs to realize robust and sustainable performance.

as hard drives, how higher layers utilize said interface can greatly affect overall throughput and latency.

Our first contribution is to formalize the "unwritten contract" between file systems and SSDs, detailing how upper layers must treat SSDs to extract the highest instantaneous and long-term performance. Our work here is inspired by Schlosser and Ganger's unwritten contract for hard drives [82], which includes three rules that must be tacitly followed in order to achieve high performance on Hard Disk Drives (HDDs); similar rules have been suggested for SMR (Shingled Magnetic Recording) drives [46].

What are Unwritten Contract

The Written contract is the API from the device, essential to get the basic function working

- Read, Write, Flush, Trim ← The essentials

The Unwritten contract is for performance

- Fluctuation in the performance due to the internal SSD complexity that you have seen so far
- Multi-level details to take into consideration

How can you develop your application for the best performance?

Unwritten Contract

1. **Request Scale**
   a. Leverage parallelism
2. **Locality**
   a. Respect locality with the FTL mappings
3. **Align Sequentially**
   a. Access aligned data to help the FTL
4. **Group by Death Time**
   a. Gives the best chance to GC for clean up
5. **Uniform Data Lifetime**
   a. Group data with the same lifetime to uniformly wear out flash pages

Rule	Impact	Metric
Request Scale	7.2×, 18×	Read bandwidth
	10×, 4×	Write bandwidth
Locality	1.6×	Average response time
	2.2×	Average response time
Aligned Sequentiality	2.5×	Execution time
	2.4×	Erasure count
Grouping by Death Time	4.8×	Write bandwidth
	1.6×	Throughput (ops/sec)
	1.8×	Erasure count
Uniform Data Lifetime	1.6×	Write latency

What You Should Know from this Lecture

1. Concepts: wear-leveling, over provisioning, write amplification, steady state
2. Three basic FTL designs: page-level, block-level, and hybrid
    a. Their advantages and disadvantages
3. GC design choices
    a. Trim command
    b. Various victim block selection algorithms
4. Embedded vs host FTL design options
    a. Advantages of host-based FTL designs
5. How do these design choices influence the performance of an SSD

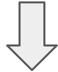
Further Reading

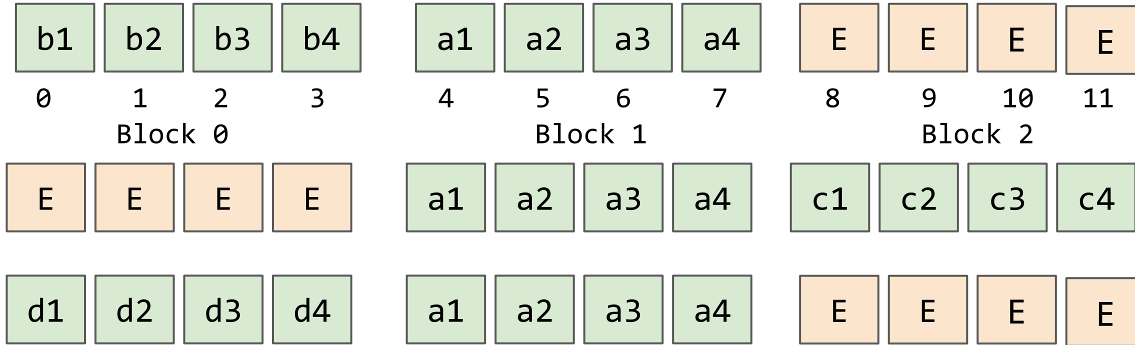
- **[Important]** Chapter 44, Flash-based SSDs, http://pages.cs.wisc.edu/~remzi/OSTEP/file-ssd.pdf
- An Evaluation of Different Page Allocation Strategies on High-Speed SSDs, https://www.usenix.org/system/files/conference/hotstorage12/hotstorage12-final55.pdf
- Wear Unleveling: Improving NAND Flash Lifetime by Balancing Page Endurance, https://www.usenix.org/conference/fast14/technical-sessions/presentation/jimenez
- Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. *The Unwritten Contract of Solid State Drives.* In Proceedings of the 12th ACM EuroSys, 2017.
- LightNVM: The Linux Open-Channel SSD Subsystem, https://www.usenix.org/system/files/conference/fast17/fast17-bjorling.pdf
- Eran Gal and Sivan Toledo. 2005. Algorithms and data structures for flash memories. ACM Comput. Surv. 37, 2 (June 2005), 138–163.
- Dongzhe Ma, Jianhua Feng, and Guoliang Li. 2014. A survey of address translation technologies for flash memories. ACM Computing Surveys 46, 3, Article 36 (January 2014), 39 pages.
- Luc Bouganim, Björn Þór Jónsson, Philippe Bonnet: uFLIP: Understanding Flash IO Patterns. CIDR 2009.
- Andrew Birrell, Michael Isard, Chuck Thacker, and Ted Wobber. 2007. A design for high-performance flash disks. *SIGOPS Operating Systems Reviews.* 41, 2 (April 2007), 88–93.
- Feng Chen, Tian Luo, and Xiaodong Zhang. 2011. CAFTL: a content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In Proceedings of the 9th USENIX FAST, 2011.
- Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In Proceedings of the 14th ACM ASPLOS, 2009.
- Ji-Yong Shin, Zeng-Lin Xia, Ning-Yi Xu, Rui Gao, Xiong-Fei Cai, Seungryoul Maeng, and Feng-Hsiung Hsu. 2009. FTL design exploration in reconfigurable high-performance SSD for server applications. In Proceedings of the 23rd ACM ICS 2009.

Backup Slides

Static vs. Dynamic Wear Leveling

W(8000, c1-c4), W(8000, d1-d4), W(8000, e1-e4)





LBA	PBA	Valid/Flags
8000	0	v
400	1	v

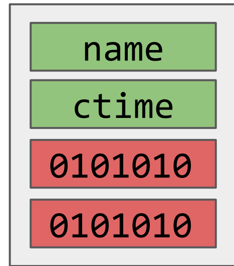
LBA	PBA	Valid/Flags
8000	2	v
400	1	v

Dynamic wear-leveling: In this example, block 0 and 2 are written continuously, thus, also aged continuously

But what about block 1? The FTL did not change the block location 1 because it was never updated.

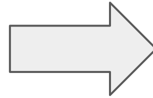
Static Wear-leveling: Cycle around all the blocks (even, the cold, static blocks) too. Time to time, FTL will read the old blocks and just move them around for even wear-leveling

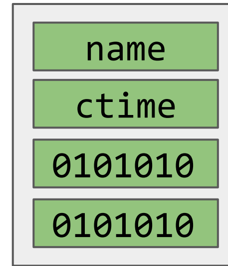
Impact of Workload Patterns



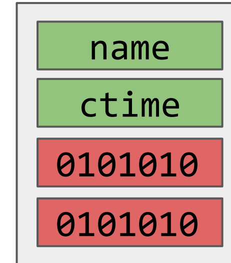
Metadata of the file

Content of the file





A new write, that is buffered in SRAM and in the log page and then merged in this block. Copy 50% of pages for GC



Hot data - frequently updated - is mixed with the cold data - rarely updated in a single block

Everytime the block is updated, we need to copy the cold data from the target block to the new one

If the whole block was hot data (“all validated” at the same time”), easy - just erase

Grouping Data Together

Why group together? To group various write/update patterns together to minimize the effort required to “prepare” a block for GC

- We cannot avoid not doing GC - but we can minimize the “prep” time

How to group data?

- **Based on age:** all pages with the “similar” creation and deletion time should be group together
- **Based on temperate:** how (in)frequently a page is updated. Frequently updated data together will expire together quickly, hence, easy to discard the whole block and just erase (no live data)
- Mix of various other policies -- stream/namespace specific policies

FTL Design Exploration in Reconfigurable High-Performance SSD for Server Applications, ICS 2009

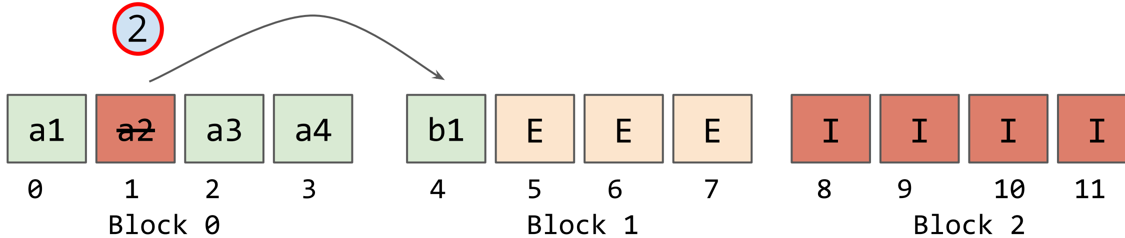
FTL design choices	Best conditions to use	Pros	Cons
Static allocation (b, c, d)	Dominant sequential IO requests	High parallelism, Even distribution of requests	Less benefit for random IO, uncontrollable wear level among flash modules
Static allocation (a, e, f, g)	No benefit for most cases	N.A.	Bad figures for most of evaluation metrics
Page striping unit	High IO rate	Utilizes many flash modules in parallel	Small advantage from data locality increasing cleaning operation
Block striping unit	Low IO rate	Small number of cleaning operations	Uneven request distribution increasing response time
Dynamic allocation with chip allocation pool	Similar ratio of random and sequential IO	Parallelism for both sequential and random IOs, moderate performance for most cases	Potential for data and request skew
Dynamic allocation with SSD allocation pool	Dominant random IO requests	High parallelism for random IO, Even distribution of random requests	Potential for data and request skew, relatively worse performance for sequential IOs
Load balancing	Skewed IO request to few flash modules	Even utilization of flash modules	Increased page migration
Hot/Cold Separation	Evenly distributed IO request and hot/cold data	Less erase, and page copying	Misclassification of data degrades performance
Large wear leveling cluster	Unevenly distributed IO and erase requests, requirements for even wear level throughout SSD	Even wear level throughout large cluster	Larger overhead and response time
Small wear leveling cluster	Evenly distributed IO and erase requests, Requirements for small IO response time	Small overhead for wear leveling	Potential for uneven wear level among flash modules

Table 6: Summary of FTL exploration and tradeoffs

Page-Mapped FTLs : Failure Analysis

①

W(0, a1), W(100,a2), W(101, a3), W(400, a4), W(100, b1)



Invalid, Written, Erased

	LPA	PPA	Valid/Flags
	0	0	V
④	~~100~~	~~1~~	“I”
	101	2	V
	400	3	V
③	100	4	V

Let’s consider when a failure happens

- Before 1 : no write has happened then
- Between 1 and 2 : (while writing p4) then no state has been changed, the last state remains
- Between 2 and 3 : new content has been written on the page 4, but no FTL entry, the last state remains
- Between 3 and 4 : new content written, new FTL entry, but the old is not invalidated, device can find out which is the last written state with timestamp and that wins
- After 4 : everything committed, failure will have no effect

In any case - either you get the old content or the new one, the date is not lost