Flash

[Slight modifications to slides from Tyler Caraza-Harter
www: https://tyler.caraza-harter.com]
Cost: HDD vs. SSD

Cost: HDD vs. SSD

Note: These are trends, not the most up-to-date data.

There are different classes of HDDs and SSDs which complicate this graph, but the thing to note is that there is a gap, but it is narrowing and all costs are trending downward.

Disk Overview

I/O requires: seek, rotate, transfer

Inherently:
- not parallel (only one head)
- slow (mechanical)
- poor random I/O (locality around disk head)

Random requests take 10ms+
Flash

Hold charge in cells. No moving parts!

Inherently parallel.

No seeks!
SLC: Single-Level Cell

NAND Cell
SLC: Single-Level Cell

NAND Cell
SLC: Single-Level Cell

NAND Cell

charge

0
MLC: Multi-Level Cell

NAND Cell
MLC: Multi-Level Cell

NAND Cell

01
MLC: Multi-Level Cell

NAND Cell

charge

10
MLC: Multi-Level Cell

NAND Cell

charge

11
Single- vs. Multi-Level Cell

SLC

MLC

charge

charge
Single- vs. Multi-Level Cell

SLC
- expensive
- robust

MLC
- cheap
- sensitive
Wearout

Problem: flash cells wear out after being overwritten too many times.

MLC: ~10K times
SLC: ~100K times

Usage strategy:
Wearout

Problem: flash cells wear out after being overwritten too many times.

MLC: \(~10K\) times
SLC: \(~100K\) times

Usage strategy: wear leveling.
- prevents some cells from wearing out while others still fresh.
Banks

Flash devices are divided into banks (aka, planes).

Banks can be accessed in parallel.
Banks

Flash devices are divided into banks (aka, planes).

Banks can be accessed in parallel.

read

Bank 0

Bank 1

Bank 2

Bank 3
Banks

Flash devices are divided into banks (aka, planes).

Banks can be accessed in parallel.

Bank 0  Bank 1  Bank 2  Bank 3
Flash devices are divided into banks (aka, planes).

Banks can be accessed in parallel.
Banks

Flash devices are divided into banks (aka, planes).

Banks can be accessed in parallel.
Flash Writes

Writing 0’s:
- fast, fine-grained

Writing 1’s:
- slow, course-grained
Flash Writes

Writing 0’s:
- fast, fine-grained
- called “program”

Writing 1’s:
- slow, course-grained
- called “erase”
Flash Writes

Writing 0’s:
- fast, fine-grained [pages]
- called “program”

Writing 1’s:
- slow, course-grained [blocks]
- called “erase”
each bank contains many “blocks”
Block

one block
One NAND flash Chip

Is made of up several Banks

Is made up of several blocks

Is made up of several pages
Block
Block

program

1111 1111 1111 1001
1111 1111 1111 1111
1111 1111 1111 1111
1111 1111 1111 1111
```
<table>
<thead>
<tr>
<th>1111</th>
<th>1111</th>
<th>1111</th>
<th>1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>1111</td>
<td>1110</td>
<td>1100</td>
</tr>
<tr>
<td>1111</td>
<td>1111</td>
<td>0001</td>
<td>1111</td>
</tr>
<tr>
<td>1111</td>
<td>1111</td>
<td>1111</td>
<td>1111</td>
</tr>
</tbody>
</table>
```

program
Block

1111 1111 1111 1001
1111 1111 1111 1100
1111 1111 1110 1111
1111 1111 0001 1111

erase
Block

erase
<table>
<thead>
<tr>
<th></th>
<th>disk</th>
<th>flash</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>read</strong></td>
<td>read sector</td>
<td>read page</td>
</tr>
<tr>
<td><strong>write</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>disk</td>
<td>flash</td>
</tr>
<tr>
<td>--------</td>
<td>---------------</td>
<td>--------------</td>
</tr>
<tr>
<td><strong>read</strong></td>
<td>read sector</td>
<td>read page</td>
</tr>
<tr>
<td><strong>write</strong></td>
<td>write sector</td>
<td>program page</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0’s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>erase block</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(1’s)</td>
</tr>
</tbody>
</table>
Flash Hierarchy

**Plane**: 1024 to 4096 blocks  
- planes accessed in parallel

**Block**: 64 to 256 pages  
- unit of erase

**Page**: 2 to 8 KB  
- unit of read and program
Disk vs. Flash Performance

Throughput:
- disk: ~130 MB/s (sequential)
- flash: ~200 MB/s - 550 MB/s
Disk vs. Flash Performance

**Throughput:**
- disk: ~130 MB/s (sequential)
- flash: ~200 MB/s - 550 MB/s

**Latency**
- disk: ~10 ms (one op)
- flash
  - read: 10-50 us
  - program: 200-500 us
  - erase: 2 ms
Traditional File Systems

- File System
- Storage Device

Traditional API:
- read sector
- write sector
Traditional File Systems

- File System
- Storage Device

Traditional API:
- read sector
- write sector

not same as flash
Options

1. Build/use new file systems for flash
   - JFFS, YAFFS
   - lot of work!

2. Build traditional API over flash API.
   - use FFS, LFS, whatever we want
Traditional API with Flash

read(addr):
    return flash_read(addr)

write(addr, data):
    block_copy = flash_read(block of addr)
    modify block_copy with data
    flash_erase(block of addr)
    flash_program(block of addr, block_copy)
FS wants to write 0001

Memory:

Flash:
Flash:

```
00 00  
00 11  
11 11  
00 11  
11 11  
00 11  
```

Memory:

```
00 00  
00 11  
11 11  
00 11  
```

Flash:

```
00 00  
11 11  
11 11  
00 01  
11 11  
11 11  
```

Block 0: 00 00 11 11
Block 1: 11 11 11 11
Block 2: 00 01 11 11

read all other pages in block
modify target page in memory
program all pages in block
Write Amplification

Random writes are extremely expensive!

Writing one 2KB page may cause:
- read, erase, and program of 256KB block.
Write Amplification

Random writes are extremely expensive!

Writing one 2KB page may cause:
- read, erase, and program of 256KB block.

Would FFS or LFS be better with flash?
File Systems over Flash

Copy-On-Write FS may prevent some expensive random writes.

What about wear leveling? LFS won’t do this.

What if we want to use some other FS?
Better Solution

Add *copy-on-write* layer *between* FS and flash. Avoids RMW (read-modify-write) cycle.

Translate logical device addrs to physical addrs.

**FTL:** Flash Translation Layer.

Should translation use math or data structure?
Flash Translation Layer

logical:

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

physical:

block 0:
00 | 00 | 00 | 00
01 | 10 | 11 | 00

block 1:
10 | 11 | 11 | 11
01 | 11 | 11 | 11
11 | 11 | 11 | 11
Flash Translation Layer

write 1101

logical:
0 1 2 3 4 5 6 7

physical:
00 00 00 00 10 11 11 11
01 10 11 00 01 11 11 11

block 0  block 1
Flash Translation Layer

write 1101

logical:

0 1 2 3 4 5 6 7

physical:

00 00 00 00 00 10 11 00
01 10 11 00

block 0

block 1
Flash Translation Layer

write 1101

logical:

physical:

block 0

block 1
Flash Translation Layer

logical:

must eventually be garbage collected

physical:

block 0

block 1
FTL

Could be implemented as device driver or in firmware (usually the latter).

Where to store mapping? SRAM.

Physical pages can be in three states:
- valid, invalid, free
States

- Free
- Valid
- Invalid
States

- free
- valid
- invalid

- Program: free -> valid
- Erase: invalid -> free, invalid -> valid
SSD Architecture

SSD: looks like disk

- FTL
- SRAM: mapping tbl
Problem: Big Mapping Table

Assume 200GB device, 2KB pages, 4-byte entries.

SRAM needed: \((200\text{GB} / 2\text{KB}) \times 4\text{ bytes} = 400\text{ MB}\).

Too big, SRAM is expensive!
Page Translations

logical:

0 1 2 3 4 5 6 7

physical:

block 0
00 00 00 00
01 10 11 00

block 1
10 01 11 11
01 01 11 11

physical:

block 0
00 00 00 00
01 10 11 00

block 1
10 01 11 11
01 01 11 11
2-Page Translations

logical:

0 1 2 3 4 5 6 7

physical:

block 0
00 00 00 00
01 10 11 00

block 1
10 01 11 11
01 01 11 11
Larger Mappings

Advantage: larger mappings decrease table size.

Disadvantage?
2-Page Translations

**Logical:**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
</table>

**Physical:**

Block 0:

<table>
<thead>
<tr>
<th>00</th>
<th>00</th>
<th>00</th>
<th>00</th>
</tr>
</thead>
</table>

| 01 | 10 | 11 | 00 |

Block 1:

| 10 | 01 | 11 | 11 |

| 01 | 01 | 11 | 11 |

| 11 | 11 | 11 | 11 |
2-Page Translations

write 1011

logical:

0 1 2 3 4 5 6 7

physical:

block 0
00 00 00 00
01 10 11 00

block 1
10 01 11 11
01 01 11 11
11 11
2-Page Translations

logical:

write 1011

physical:

00 00 00 00
01 10 11 00

block 0

10 01 10 01
01 01 11 01

block 1

copy
2-Page Translations

write 1011

logical:

0 1 2 3 4 5 6 7

physical:

block 0:
00 00 00 00
01 10 11 00

block 1:
10 01 10 01
01 11 01 01
2-Page Translations

write 1011

logical:

physical:

block 0

block 1
Larger Mappings

Advantage: larger mappings decrease table size.

Disadvantages?
- more read-modify-write updates
- more garbage
- less flexibility for placement
Hybrid FTL

Use course-grained mapping for most (e.g., 95%) of data. Map at *block level*.

Use fine-grained mapping for recent data. Map at *page level*.
Log Blocks

Write changed pages to designated log blocks.

After blocks become full, merge changes with old data.

Eventually garbage collect old pages.
Merging technique depends on I/O pattern.

Three merge types:
- full merge
- partial merge
- switch merge
Merging

Merging technique depends on I/O pattern.

Three merge types:
- full merge
- partial merge
- switch merge
logical:

\[ \begin{array}{c}
\text{0} & \text{1} & \text{2} & \text{3} \\
\end{array} \]

\[ \downarrow \]

physical:

\[ \begin{array}{cccc}
\text{A} & \text{B} & \text{C} & \text{D} \\
\text{11} & \text{11} & \text{11} & \text{11} \\
\text{11} & \text{11} & \text{11} & \text{11} \\
\text{11} & \text{11} & \text{11} & \text{11} \\
\text{11} & \text{11} & \text{11} & \text{11} \\
\end{array} \]

block 0

block 1 (log)

block 2
write D2

logical: 0 1 2 3 ...

physical: A B C D

block 0

block 1 (log)

block 2
write D2
logical: 0 1 2 3 ...

eventually, we need to get rid of red arrows, as these represent expensive mappings
logical: 0 1 2 3 ...
physical: A B C D
block 0

D2 11 11 11 11
block 1 (log)

A B C D2
block 2
Merging technique depends on I/O pattern.

Three merge types:
- full merge
- partial merge
- switch merge
write D2

logical: 0 1 2 3 ...

physical: A B C D
block 0

block 1 (log)

block 2
write D2

logical:

0 1 2 3 ...

physical:

A B C D

block 0

11 11 11 11 D2

block 1 (log)

11 11 11 11

block 2

11 11 11 11 11
logical: 0 1 2 3 ...
physical: A B C D
block 0

block 1 (log) 11 11 11 11 D2
block 2 11 11 11 11 11
physical:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D2</th>
<th>11</th>
<th>11</th>
<th>11</th>
<th>11</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>block 0</td>
<td>block 1</td>
<td>block 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

diagram:

logical:

[0 1 2 3]
Merging

Merging technique depends on I/O pattern.

Three merge types:
- full merge
- partial merge
- switch merge
write A2

logical:

physical:

block 0

block 1 (log)

block 2
write A2

logical: 0 1 2 3 ...

physical:

block 0 A B C D
block 1 (log) A2 11 11 11 11
block 2 11 11 11 11 11
write B2

logical: 0 1 2 3 ...

downarrow


block 0 block 1 (log) block 2
write C2

logical: 0 1 2 3 ...

write D2

logical: 0 1 2 3 ...


block 0 block 1 (log) block 2
logical: [0, 1, 2, 3, ...]
physical: [A, B, C, D, block 0]

block 1 (log): [A2, B2, C2, D2]


garbage
Merging

Merging technique depends on I/O pattern.

Three merge types:
- full merge
- partial merge
- switch merge
Summary

Flash is much faster than disk, but…

It is more expensive.

It’s not a drop-in replacement beneath an FS without a complex layer for emulating hard disk API.