Flash-based SSDs

[ Material based on slides from Tyler Caraza-Harter ]
www: https://tyler.caraza-harter.com
SSDs vs. HDDs

Dimension 1: Cost
Cost: HDD vs. SSD

Average HDD and SSD prices in USD per gigabyte

Data sources: Mkomo.com, Gartner, and Pingdom (December 2011)

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.

SSDs vs. HDDs

Dimension 1: Cost
Dimension 2: Physical Media
Disk Overview

I/O cost: **setup** (seek + rotate), **transfer**

Implications:
- cannot parallelize operations (only one head)
- slow (mechanical parts must move through space)
- poor random I/O (locality around disk head)
  
  Random I/Os take 10ms+!
Flash

No moving parts! Instead, SSDs:

- Hold charge in **cells**
  - No seeks in I/O setup!
- Hardware organization supports **internal parallelism**.
SLC: Single-Level Cell

NAND Cell
SLC: Single-Level Cell

NAND Cell

charge
SLC: Single-Level Cell

NAND Cell

charge

0
MLC: Multi-Level Cell

NAND Cell

charge

00
MLC: Multi-Level Cell

NAND Cell
MLC: Multi-Level Cell

NAND Cell
MLC: Multi-Level Cell
Single- vs. Multi-Level Cell

SLC

charge

MLC

charge
Single- vs. Multi-Level Cell

- SLC: expensive, robust
- MLC: cheap, sensitive
Single- vs. Multi-Level Cell

SLC
expensive robust

MLC
cheap sensitive

TLC (3 bits/cell) and QLC (4 bits/cell) also exist, and are even cheaper and more sensitive than MLC.
SSDs vs. HDDs

Dimension 1: Cost
Dimension 2: Physical Media
Dimension 3: Lifetime
Wearout

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

MLC: ~10K writes
SLC: ~100K writes
Wearout

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

MLC: ~10K writes
SLC: ~100K writes

Cell management strategy: wear leveling.
- Distribute writes across cells to more evenly spread the wear
  - Prevents some cells from wearing out while others still fresh.
SSDs vs. HDDs

Dimension 1: Cost
Dimension 2: Physical Media
Dimension 3: Lifetime
Dimension 4: Internal Organization
Banks

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

Banks can be accessed in parallel.
Banks

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

Banks can be accessed in parallel.
Banks

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

Banks can be accessed in parallel.
Banks

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

Banks can be accessed in parallel.
Banks

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

Banks can be accessed in parallel.

Bank 0  Bank 1  Bank 2  Bank 3
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 [unit: \textit{page}]
- called “\textit{program}”

Writing 1’s:
- slow, course-grained [unit: \textit{block}]
- called “\textit{erase}”
A Bank Consists of Blocks
A Bank Consists of Blocks

Bank 0
Bank 2
Bank 3

each bank contains many "blocks"
A Block Consists of Pages
A Block Consists of Pages

one block
Block

one page
The Heirarchy of SSD components:

One NAND flash Chip

Is made of up several Banks

Is made up of several blocks

Is made up of several pages
Block

program

1111 1111 1111 1001 1111
1111 1111 1111 1111
1111 1111 1111 1111
Block

1111 1111 1111 1001
1111 1111 1111 1100
1111 1111 1111 1111
1111 1111 1111 1111

program
Block

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

erase
Block

1111 1111 1111 1111
1111 1111 1111 1111
1111 1111 1111 1111
1111 1111 1111 1111

erase
Block
APIs

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

**Channel**: The number of chips that the controller can talk to simultaneously
- Low end SSDs: 2-4 channels
- High end SSDs: 8+ channels
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 \text{-} 550 \text{ 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
   - Example: JFFS, YAFFS
   - **Problem**: this takes a lot of work!

2. Translate traditional API onto flash API.
   - then we can use FFS, LFS, etc. without any additional work!
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:

block 0

block 1

block 2
Flash:

Memory:

read all other pages in block
Modify target page in memory.
<table>
<thead>
<tr>
<th>Block 0</th>
<th>Block 1</th>
<th>Block 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

Memory:

- 00 00
- 01 11
- 11 11
- 00 11

Flash:

- 00 00
- 00 11
- 11 11
- 11 11
- 00 11
- 01 11
- 11 11
- 11 11
program all pages in block

Memory:

Flash:

block 0

block 1

block 2
Flash:

Memory:

Flash:
block 0
00 00 11 11 00 11
block 1
11 11 11 11 11 11
block 2
01 11 11 11 11 11
Write Amplification

Problem: Random writes are extremely expensive!

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

Problem: 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.
File Systems over Flash

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

What about *wear leveling*?
Copy-On-Write FS *may* prevent some expensive random writes.

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

(Perhaps some other FS has features or APIs our applications rely on, so we must use it)
Better Solution

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

Translate logical device addr to physical addr.

**FTL**: Flash Translation Layer.

**Question**: How should translations be managed?
Flash Translation Layer

logical: 0 1 2 3 4 5 6 7

physical: block 0: 00 00 01 01 00 00 block 1: 10 11 11 11 11 11
Flash Translation Layer

write 1101

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
Flash Translation Layer

logical:

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array}
\]

write 1101

physical:

block 0:
\[
\begin{array}{cccc}
00 & 00 & 00 & 00 \\
01 & 10 & 11 & 00 \\
\end{array}
\]

block 1:
\[
\begin{array}{cccc}
10 & 11 & 11 & 11 \\
01 & 01 & 11 & 11 \\
\end{array}
\]
Flash Translation Layer

write 1101

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 01 11 11
11 11 11 11
Flash Translation Layer

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 (OS) or in firmware (code running on SSD).
  - usually done in firmware

Where to store LBA->PBA mappings? SRAM.

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

SSD: looks like a traditional disk

(Traditional block API)

FTL

SRAM: mapping tbl
Problem: Big Mapping Table

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

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

That table would be too big, SRAM is expensive!
Page Translations

logical:

0 1 2 3 4 5 6 7

physical:

block 0
00 01 10 11 00

block 1
10 01 11 11 11
2-Page Translations

**logical:**

0 1 2 3 4 5 6 7

**physical:**

<table>
<thead>
<tr>
<th>block 0</th>
<th>block 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 01 10 11 00</td>
<td>10 01 11 11 11</td>
</tr>
<tr>
<td>00 00 00 01 11</td>
<td>01 01 11 11 11</td>
</tr>
</tbody>
</table>
Larger Mappings

Advantage: larger mappings decrease table size.

Disadvantage?
2-Page Translations

**logical:**

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

**physical:**

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

block 0  block 1
2-Page Translations

write 1011

logical:

physical:

block 0

block 1
2-Page Translations

write 1011

copy

physical:

block 0

00 00 00 00
01 10 11 00

block 1

10 01 10 01
01 01 11 01

logical:

0 1 2 3 4 5 6 7
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

logical:

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
00 & 00 & 00 & 00 & 10 & 01 & 10 & 01 \\
01 & 10 & 11 & 00 & 01 & 01 & 11 & 01 \\
\end{array}
\]

physical:

write 1011
Larger Mappings

Advantage: larger mappings decrease table size.

Disadvantages?
- Increased write amplification
  - 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.
  - always search for page in these mappings first

After blocks become full, merge changes with old data.

Eventually garbage collect old pages.
Merging

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
write D2

logical: 0 1 2 3 ...

physical:

block 0

block 1 (log)

block 2
write D2

logical:

physical:

block 0

block 1 (log)

block 2
logical: 0 1 2 3 ...

physical:

block 0: A B C D
block 1 (log): D2 11 11 11 11
block 2: 11 11 11 11 11
eventually, we need to get rid of red arrows, as these represent expensive mappings
logical: 0 1 2 3 ...

physical:

block 0: A B C D
block 1 (log): D2 11 11 11 11
block 2: A B C D2
logical: 0 1 2 3 ...

physical:

block 0
A B C D

D2

block 1 (log)
11 11 11 11

garbage

block 2
A B C D2
Merging

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

11 11 11 11 11

block 1 (log)

11 11 11 11 11

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 11

block 2
Merging

Merging technique depends on I/O pattern.

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

logical: 0 1 2 3 ...

physical: A B C D

block 0

block 1 (log)

block 2
write A2
write B2

logical: 0 1 2 3 ...

physical:

block 0 A B C D
block 1 (log) A2 B2 11 11
block 2 11 11 11 11 11
write C2

logical: 0 1 2 3 ...


block 0 block 1 (log) block 2
write D2

logical: 0 1 2 3 ...

physical:

block 0: A B C D
block 1 (log): A2 B2 C2 D2
block 2: 11 11 11 11 11
physical: A B C D
block 0

logical: [0 1 2 3] ...

garbage

physical: A2 B2 C2 D2
block 1 (log)

block 2
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.