Flash-based SSDs

CMPU 334 – Operating Systems
Jason Waterman
New form of persistent storage

• Solid-state storage
  • No mechanical or moving parts
  • Built out of transistors
  • Retains information despite power loss

• NAND-based flash
  • Created in the 1980s
  • Before writing a flash page (small chunk of data):
    • First must erase the flash block (large chunk of data) where the page lives
    • This takes a relatively long time
  • Writing a page too often will cause it to wear out
    • 10,000 to 100,000 writes to a page
    • Page is no longer usable
Storing a single bit

• Single-level cell (SLC) flash
  • Single bit stored within a transistor
  • Floating gate stores charge
  • Best performing, more expensive

• Multi-level cell (MLC) flash
  • Two bits are encoded into 4 levels of charge

• Triple-level cell (TLC) flash
  • Encodes 3 bits per cell
  • Cheaper but not as good performance
Flash organization

• Flash chips are organized into banks
  • Each bank is accessed as **erase blocks** or **pages**

• Erase blocks
  • Typically, 128 KB or 256 KB
  • Contains many pages
  • When a single page needs to be overwritten, the entire block must be erased first!

• Pages
  • Fundamental unit for a flash
  • Typical size: 4 KB
Flash Operations

• Read a page
  • Can read any page by specifying the read command and a page number
  • Fast operation (10s of microseconds)
  • Regardless of the location of previous request (random access device)

• Erase a block
  • Before writing to a page within a block, you need to erase the entire block
  • Destroys the contents of the block by setting all bits to the value ‘1’
  • Slow operation (a few milliseconds)

• Program a page
  • Writes data to an erased page by changing some of the ones within a page to zeros
  • Less expensive than erasing a block, but more expensive than reading a page
  • 100s of microseconds
Flash example

• Four 8-bit pages within a 4-page block (unrealistically small)

<table>
<thead>
<tr>
<th>Page 0</th>
<th>Page 1</th>
<th>Page 2</th>
<th>Page 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>00011000</td>
<td>11001110</td>
<td>00000001</td>
<td>00111111</td>
</tr>
<tr>
<td>VALID</td>
<td>VALID</td>
<td>VALID</td>
<td>VALID</td>
</tr>
</tbody>
</table>

• Like to write Page 0 – must move other pages before erasing entire block

<table>
<thead>
<tr>
<th>Page 0</th>
<th>Page 1</th>
<th>Page 2</th>
<th>Page 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>11111111</td>
<td>11111111</td>
<td>11111111</td>
<td>11111111</td>
</tr>
<tr>
<td>ERASED</td>
<td>ERASED</td>
<td>ERASED</td>
<td>ERASED</td>
</tr>
</tbody>
</table>

• Now can write Page 0

<table>
<thead>
<tr>
<th>Page 0</th>
<th>Page 1</th>
<th>Page 2</th>
<th>Page 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000011</td>
<td>11111111</td>
<td>11111111</td>
<td>11111111</td>
</tr>
<tr>
<td>VALID</td>
<td>ERASED</td>
<td>ERASED</td>
<td>ERASED</td>
</tr>
</tbody>
</table>
Flash translation layer (FTL)

- Turns system reads and writes into internal flash operations
  - Logical blocks → low-level read, erase, and program the physical blocks and pages
- Flash chips (persistent storage)
- SRAM (caching and buffering data, mapping tables)
- Control logic for device operation
Performance goals

• Speed
  • Use multiple flash chips in parallel to obtain higher performance

• Reduce write amplification
  • Write traffic in bytes issued by the FTL divided by write traffic issued to the flash

• Wear leveling
  • Spread out writes across blocks of the flash as evenly as possible

• Program disturbance
  • Writing a page can flip bits of neighboring pages
  • Write pages from low page to high page to minimize this
Log-structured FTL

• For reliability and performance FTLs are log structured

• Given a write to logical block N:
  • Device appends the write to the next free spot in the currently being written block

• To find logical block N:
  • Device keeps a mapping table (both in memory and persistent storage)
  • Keeps the physical address of each logical block in the system
Log-structured FTL example

• Write logical block 100

<table>
<thead>
<tr>
<th>Table:</th>
<th>100</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block:</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Page:</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>Content:</td>
<td>a1</td>
<td></td>
</tr>
<tr>
<td>State:</td>
<td>V</td>
<td>E</td>
</tr>
</tbody>
</table>

• Logical write of 101, 2000, 2001

<table>
<thead>
<tr>
<th>Table:</th>
<th>100</th>
<th>0</th>
<th>101</th>
<th>1</th>
<th>2000</th>
<th>2</th>
<th>2001</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block:</td>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page:</td>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
</tr>
<tr>
<td>Content:</td>
<td>a1</td>
<td>a2</td>
<td>b1</td>
<td>b2</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td>State:</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
</tr>
</tbody>
</table>
Persisting the FTL mapping

• Map is stored in memory on the device for performance

• How does the mapping survive a power loss?

• Record some mapping information with each page
  • Out-of-band (OOB) area
  • Mapping can be reconstructed from this information
  • Scanning a large SSD to find mappings is slow

• Higher-end devices use logging and checkpointing
Garbage Collection

- Assume blocks 100 and 101 are written again with contents c1 and c2

<table>
<thead>
<tr>
<th>Block</th>
<th>Page</th>
<th>Content</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>a1</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>a2</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td>02</td>
<td>b1</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td>03</td>
<td>b2</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td>04</td>
<td>c1</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>05</td>
<td>c2</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>06</td>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>07</td>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>08</td>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>09</td>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>i</td>
<td>i</td>
</tr>
</tbody>
</table>

- Garbage collection (reclaiming dead blocks)
  - Find a block with one or more garbage pages
  - Read the live (non-garbage) pages from the block
  - Write out live pages to the log
  - Reclaim block for use in writing
Mapping table size

- 1-TB SSD, 4-KB page size, 4-byte map entry
  - 1 GB of memory needed for just the mappings
  - Page-level FTL mapping is impractical

- Block-Based Mapping
  - One pointer per block of the device instead of per page
  - Logical address divided into block sized chunks
    - Logical addresses consists of two portions: chunk number and offset
    - Poor performance for “small writes” (less than a block) – must copy all pages in the block

- Example: logical blocks 2000, 2001, 2002, and 2003 all have the same chunk number (500) and have offsets (0, 1, 2, 3)

<table>
<thead>
<tr>
<th>Table: 500</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block: 0</td>
<td>0</td>
</tr>
<tr>
<td>Page: 00</td>
<td>01</td>
</tr>
<tr>
<td>Content:</td>
<td>a</td>
</tr>
<tr>
<td>State:</td>
<td>i</td>
</tr>
</tbody>
</table>
Block-based Mapping Writes

- Writing to logical block 2002 (with contents $c'$)
  - Read in 2000, 2001, 2003 and write out all four logical blocks in a new location
  - Update mapping table
  - Small writes (less than a physical block) hurt performance
  - Increase write amplification
  - With block sizes of 256KB or larger, small writes can happen often

<table>
<thead>
<tr>
<th>Table: 500 ➔ 8</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Block:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Page:</strong></td>
<td>00 01 02 03 04 05 06 07 08 09 10 11</td>
</tr>
<tr>
<td><strong>Content:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>State:</strong></td>
<td>i i i i E E E E a b c' d V V V V</td>
</tr>
</tbody>
</table>

Flash Chip
Hybrid Mapping

• FTL keeps a few blocks erased and directs all writes to them
  • Called log blocks
  • Keep per-page mappings for these log blocks

• Keeps two types of tables in memory
  • Log table (per-page mappings)
  • Data table (per-block mappings)

• When looking for a logical block
  • First look in log table
  • Then check data table

• Must keep number of log blocks small
  • Periodically examine log blocks and switch them to data blocks when possible
  • Done with three main techniques, based on the contents of the block
Switch Merge

- Logical pages 1000, 1001, 1002, and 1003 were written and placed in block 2
- Each of these blocks are overwritten in the exact same order (a’, b’, c’, d’)
- FTL can perform a switch merge
  - Log block 0 becomes the storage location
  - Block 2 is erased and used as a log block
- Best case for hybrid FTL
Partial Merge

• What happens in the case of a partial write?

| Log Table: | 1000 → 0 | 1001 → 1 |
| Data Table: | 250 → 8 |

FTL performs a **partial merge**

• Logical blocks 1002 and 1003 are read from physical block 2 and are appended to the log (in pages 2 and 3)

• Then we can do a switch merge like before
Full Merge

• FTL must pull together pages from many other blocks to perform cleaning

• Example: logical blocks 0, 4, 8, and 12 are written to a log block
  • To switch to a block-mapped page the FTL must:
    • Create a data block containing logical blocks 0, 1, 2, and 3
    • Read 1, 2, and 3 from elsewhere and write out 0-4 together
    • Must do the same for logical blocks 4, 8, and 12 as well
  • Then log block can be freed

• Frequent full merges can harm performance and should be avoided when possible
Page Mapping Plus Caching

• Given the complexity of the hybrid approach a simpler solution would be to cache only the active parts of the page mappings in memory
  • Reduces the memory needed

• With a workload that accesses a small set of pages, this approach works well

• With a working set of pages larger than cache memory
  • Each access will require an extra flash read to bring in the missing mapping
  • FTL will have to evict an old mapping
    • If that mapping is dirty (has been changed from the copy in flash) it will have to be written out to flash

• Many workloads will have locality, so caching can reduce memory overheads and keep performance high
Wear leveling

• Multiple erase/program cycles will wear out a flash block
  • Try to spread that work across all the blocks of the device evenly

• Log-structured approach does a good job of spreading out write log
  • Garbage collection helps as well

• What about long-lived data that does not get over-written?
  • Garbage collection will never reclaim the block

• Periodically read all the live data out of those blocks and re-write it elsewhere
  • Helps with wear-leveling
  • Increases write amplification of the SSD
  • Decreases performance
SSD performance and cost

• Performance
  • Great random access compared to HDD

<table>
<thead>
<tr>
<th>Device</th>
<th>Random Reads (MB/s)</th>
<th>Random Writes (MB/s)</th>
<th>Sequential Reads (MB/s)</th>
<th>Sequential Writes (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samsung 840 Pro SSD</td>
<td>103</td>
<td>287</td>
<td>421</td>
<td>384</td>
</tr>
<tr>
<td>Seagate 600 SSD</td>
<td>84</td>
<td>252</td>
<td>424</td>
<td>374</td>
</tr>
<tr>
<td>Intel SSD 335 SSD</td>
<td>39</td>
<td>222</td>
<td>344</td>
<td>354</td>
</tr>
<tr>
<td>Seagate Savvio 15K.3 HDD</td>
<td>2</td>
<td>2</td>
<td>223</td>
<td>223</td>
</tr>
</tbody>
</table>

• Cost
  • About 10x more expensive than HDD
End of the Semester Logistics

• Due date for last assignment: May 7th\textsuperscript{th} 11:59pm
  • Last day to turn in for credit: May 9\textsuperscript{th}

• **Review**: Friday May 5\textsuperscript{th} 3pm – 5pm

• **Final**: Wednesday May 10\textsuperscript{th} 1:00pm - 3:00pm SP 309
  • Closed books and closed notes, no electronic devices
  • Comprehensive

• Practice Problems
  • Posted to the course website