RAID

Redundant Array of Inexpensive* Disks

* In industry, “inexpensive” has been replaced by “independent” :-)

Implement the abstraction of a faster, bigger and more reliable disk using a collection of slower, smaller, and more likely to fail disks. Different configurations offer different tradeoffs.

Key feature: transparency

The Power of Abstraction™ to the OS looks like a single, large, highly performant and highly reliable single disk (a SLED, hopefully with lower-case “e”!)
- A linear array of blocks
- Mapping needed to get to actual disk
- Cost: one logical I/O may translate into multiple physical I/Os

In the box:
- Microcontroller, DRAM (to buffer blocks) [sometimes non-volatile memory, parity logic]
Failure Model

- RAID adopts the strong, somewhat unrealistic Fail-Stop failure model (electronic failure, wear out, head damage)
  - component works correctly until it crashes, permanently
    - disk is either working: all sectors can be read and written
    - or has failed: it is permanently lost
  - failure of the component is immediately detected
    - RAID controller can immediately observe a disk has failed and accesses return error codes

- In reality, disks can also suffer from isolated sector failures
  - Permanent: physical malfunction (magnetic coating, scratches, contaminants)
  - Transient: data is corrupted, but new data can be successfully read from/written to sector
How to Evaluate a RAID

- **Capacity**
  - what fraction of the sum of the storage of its constituent disks does the RAID make available?

- **Reliability**
  - How many disk faults can a specific RAID configuration tolerate?

- **Performance**
  - Workload dependent
RAID-0: Striping

Spread blocks across disks using round robin

<table>
<thead>
<tr>
<th>Stripe</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

+ Excellent parallelism
  can read/write from multiple disks

- Worst-case positioning time
  wait for largest across all disks
**RAID-0: Striping (Big Chunk Edition)**

Spread blocks across disks using round robin

<table>
<thead>
<tr>
<th>Stripe</th>
<th>0</th>
<th>2</th>
<th>4</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>11</td>
<td>13</td>
<td>15</td>
</tr>
</tbody>
</table>

+ improve positioning time  
— decrease parallelism
RAID-0: Evaluation

- **Capacity**
  - Excellent: N disks, each holding B blocks support the abstraction of a single disk with NxB blocks

- **Reliability**
  - Poor: Striping reduces reliability
    - Any disk failure causes data loss

- **Performance**
  - Workload dependent, of course
  - We’ll consider two workloads
    - Sequential: single disk transfers S MB/s
    - Random: single disk transfer R MB/s
    - \( S \gg R \)
RAID-0: Performance

- Single-block read/write throughput
  - about the same as accessing a single disk

- Latency
  - Read: T ms (latency of one I/O op to disk)
  - Write: T ms

- Steady-state read/write throughput
  - Sequential: N x S MB/s
  - Random: N x R MB/s
RAID-1: Mirroring

Each block is replicated twice

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Read from any          Write to both
RAID-1: Evaluation

- **Capacity**
  - Poor: $N$ disks of $B$ blocks yield $(N \times B)/2$ blocks

- **Reliability**
  - Good: Can tolerate the loss (not corruption!) of any one disk

- **Performance**
  - Fine for reads: can choose any disk
  - Poor for writes: every logical write requires writing to both disks
    - suffers worst seek+rotational delay of the two writes
RAID-1: Performance

- Steady-state throughput
  - Sequential Writes: \( \frac{N}{2} \times S \) MB/s
    - Each logical Write involves two physical Writes
  - Sequential Reads: as low as \( \frac{N}{2} \times S \) MB/s

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

Suppose we want to read 0, 1, 2, 3, 4, 5, 6, 7
RAID-1: Performance

- Steady-state throughput
  - Sequential Writes: \( \frac{N}{2} \times S \) MB/s
    - Each logical Write involves two physical Writes
  - Sequential Reads: as low as \( \frac{N}{2} \times S \) MB/s
    - Reads can be distributed across all disks
  - Latency for Reads and Writes: \( T \) ms

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

Suppose we want to read 0, 1, 2, 3, 4, 5, 6, 7
Each disk only delivers half of his bandwidth:
  half of its blocks are skipped!

- Random Writes: \( \frac{N}{2} \times R \) MB/s
  - Each logical Write involves two physical Writes
- Random Reads: \( N \times R \) MB/s
  - Reads can be distributed across all disks
RAID-4: Block Striped, with Parity

Data disks

Parity disk

<table>
<thead>
<tr>
<th>Stripe</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>P0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>P1</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>P2</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>P3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0001</th>
<th>0110</th>
<th>0</th>
<th>11</th>
<th>0000</th>
<th>0001</th>
<th>110</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>0001</td>
<td>100</td>
<td>110</td>
<td>010</td>
<td>101</td>
<td>011</td>
</tr>
</tbody>
</table>
RAID-4: Block Striped, with Parity

Disk controller can identify faulty disk
single parity disk can detect and correct errors
RAID-4: Evaluation

- Capacity
  - N disks of B blocks yield \((N-1) \times B\) blocks

- Reliability
  - Tolerates the failure of any one disk

- Performance
  - Fine for sequential read/write accesses and random reads
  - Random writes are a problem!
RAID-4: Performance

- Sequential Reads: \((N-1) \times S\) MB/s
- Sequential Writes: \((N-1) \times S\) MB/s
- Random Read: \((N-1) \times R\) MB/s
- Random Writes: \(R/2\) MB/s (N is gone! Yikes!)

need to read block from disk and parity block

Compute \(P_{\text{new}} = (B_{\text{old}} \text{ XOR } B_{\text{new}}) \text{ XOR } P_{\text{old}}\)

Write back \(B_{\text{new}}\) and \(P_{\text{new}}\)

Every write must go through parity disk, eliminating any chance of parallelism

Every logical I/O requires two physical I/Os at parity disk:
can at most achieve 1/2 of its random transfer rate (i.e. \(R/2\))

Latency:
- Reads: \(T\) ms; Writes: \(2T\) ms
RAID-5: Rotating Parity (avoids the bottleneck)

Parity and Data distributed across all disks

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>P0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>P1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>P2</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>P3</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>
RAID-5: Evaluation

Capacity & Reliability
- As in Raid-4

Performance
- Sequential read/write accesses as in RAID-4
  - \((N-1) \times S \text{ MB/s}\)
- Random Reads are slightly better
  - \(N \times R \text{ MB/s} \) (instead of \((N-1) \times R \text{ MB/s}\))
- Random Writes much better than RAID-4: \(R/2 \times N/2\)
  - as in RAID-4 writes involve two operations at every disk:
    - each disk can achieve at most \(R/2\)
  - but, without a bottleneck parity disk, we can issue up to \(N/2\) writes in parallel (each involving 2 disks)
SSDs
## Why care?

<table>
<thead>
<tr>
<th>HDD</th>
<th>SSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Require seek, rotate, transfer on each I/O</td>
<td>- No seeks</td>
</tr>
<tr>
<td>- Not parallel (one active head)</td>
<td>- Parallel</td>
</tr>
<tr>
<td>- Brittle (moving parts)</td>
<td>- No moving parts</td>
</tr>
<tr>
<td>- Slow (mechanical)</td>
<td>- Random reads take 10s of µs</td>
</tr>
<tr>
<td>- Poor random I/O (10s of ms)</td>
<td>- Wears out!</td>
</tr>
</tbody>
</table>
**Flash Storage**

**To write 0**
- apply positive voltage to drain
- apply even stronger positive voltage to control gate
- some electrons are tunneled into floating gate

**Diagram Description**
- **N source** and **N drain** connected to **P-Type substrate**
- **Control gate** and **Floating gate** with **Oxide/Nitride/Oxide ONO inter-poly dielectric (insulator)**
- **Oxide tunnel** and **Oxide sidewall**
- **Fowler-Nordheim tunneling**
- Bit stored here, surrounded by an insulator
  - No charge = 1
  - Charge = 0
Flash Storage

**To write 0**
- apply positive voltage to drain
- apply even stronger positive voltage to control gate
- some electrons are tunneled into floating gate

**To write 1**
- apply positive voltage to drain
- apply negative voltage to control gate
- electrons are forced out of floating gate into source

- Oxide/Nitride/Oxide ONO inter-poly dielectric (insulator)
- Control gate
- Floating gate
- Oxide sidewall
- Oxide tunnel
- Fowler–Nordheim tunneling
- Bit stored here, surrounded by an insulator
- No charge = 1
- Charge = 0

- N source
- P-Type substrate
- N drain
Flash Storage

To write 0
- apply positive voltage to drain
- apply even stronger positive voltage to control gate
- some electrons are tunneled into floating gate

To write 1
- apply positive voltage to drain
- apply negative voltage to control gate
- electrons are forced out of floating gate into source

To read
- apply voltage to control gate
- apply voltage across source and drain
- measure current between source and drain to determine whether electrons in gate
  - if electrons in floating gate, must apply higher voltage ato control gate to have current measured current can encode more than a single bit
The Cell

- **Single-level cells**
  - faster, more lasting (50K to 100K program/erase cycles), more stable
  - 0 means charge; 1 means no charge

- **Multi-level cells**
  - can store 2, 3, even 4 bits
  - cheaper to manufacture
  - wear out faster (1k to 10K program/erase cycles)
  - more fragile (stored value can be disturbed by accesses to nearby cells)
The SSD Storage Hierarchy

- **Cell**: 1 to 4 bits
- **Page**: 2 to 8 KB, not to be confused with a VM page
- **Block**: 64 to 256 pages, not to be confused with a disk block
- **Plane/Bank**: Many blocks (Several Ks)
- **Flash Chip**: Several banks that can be accessed in parallel
Basic Flash Operations

- **Read (a page)**
  - 10s of μs, independent of the previously read page
  - great for random access!

- **Erase (a block)**
  - sets the entire block (with all its pages) to 1 (!)
  - very coarse way to write 1s...
  - 1.5 to 2 ms (on a fast SLC)

- **Program (a page)**
  - can change some bits in a page of an erased block to 0
  - 100s of μs
  - changing a 0 bit back to 1 requires erasing the entire block!
Banks

Bank 0

Bank 1

Bank 2

Bank 3
Banks

Each bank contains many blocks
After an Erase, all cells are discharged (i.e., store 1s)

Program

one page
If now we want to set this bit to 1, we need to erase the entire block!

Modified pages must be copied elsewhere, or lost!
Erase

Every erase/program cycle adds some charge to a block; over time, hard to distinguish 1 from 0!
### APIs

<table>
<thead>
<tr>
<th></th>
<th>HDD</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>write sector</td>
<td>program page (0's)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>erase block (1's)</td>
</tr>
</tbody>
</table>

### Performance

<table>
<thead>
<tr>
<th></th>
<th>HDD</th>
<th>Flash</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Throughput</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>≈ 130MB/s (sequential)</td>
<td>≈ 200MB/s (random or sequential)</td>
</tr>
<tr>
<td><strong>Latency</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>≈ 10ms</td>
<td>read 25µs program 200-300µs erase 1.5-2 ms</td>
</tr>
</tbody>
</table>
Using Flash Memory

- Need to map reads and writes to logical blocks to read, program, and erase operations on flash.

Flash Translation Layer (FTL)
Flash Translation Layer

- tries to minimize
  - write amplification: \[
  \frac{\text{write traffic (bytes) to flash chips}}{\text{write traffic (bytes) from client to SSD}}
  \]
  - wear out: practices wear leveling
  - disturbance: writes pages in a block in order, low to high

From Flash to SSD
FTL through Direct Mapping

- Just map logical disk block to physical page
  - reads are fine (yahoo!)
  - write to logical block, however, involves
    - reading the (physical) block where physical page lives
    - erasing the block
    - (re)programming old pages as well as new page

- Severe write amplification
  - writes are slow!

- Poor wear leveling
  - pages corresponding to “hot” logical block experience disproportionate number of erase/program cycles
FTL through Direct Mapping

- Just map logical disk block to physical page
  - reads are fine
  - write to logical block involves
    - reading the (physical) block where physical page lives
    - erasing the block
    - programming old pages as well as new page

- Severe write amplification
  - writes are slow!

- Poor wear leveling
  - page corresponding to “hot” logical block experiences disproportionate number of erase/program cycles
Log Structured FTL

Think of flash storage as implementing a log
Log Structured FTL

1. Think of flash storage as implementing a log

<table>
<thead>
<tr>
<th>Block 0</th>
<th>Block 1</th>
<th>Block 2</th>
<th>Block 3</th>
<th>Block 4</th>
<th>Block 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page 0</td>
<td>Page 1</td>
<td>Page 2</td>
<td>Page 3</td>
<td>Page 4</td>
<td>Page 5</td>
</tr>
</tbody>
</table>

2. On a write, program next available page of physical block being currently written
   - i.e., “append” the write to your log

3. On a read, find in the log the page storing the logical block
   - don’t want to scan the whole log...
   - keep an in-memory map from logical blocks to pages!
SSD’s clients read/write 4KB logical blocks

Many physical SSD blocks; each holds 4 pages, each 4KB

A logical block maps to a physical page

Client operations

1) Erase(00)
Example

- SSD’s clients read/write 4KB logical blocks
- Many physical SSD blocks; each holds 4 pages, each 4KB

A logical block maps to a physical page

<table>
<thead>
<tr>
<th>Block</th>
<th>00</th>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
<th>05</th>
<th>06</th>
<th>07</th>
<th>08</th>
<th>09</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<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>
<td>08</td>
<td>09</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Content</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td>State</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
</tr>
</tbody>
</table>

Flash Chip

Client operations

Write (a1, 100) 2) Program(00)
**Example**

- SSD’s clients read/write 4KB logical blocks
- Many physical SSD blocks; each holds 4 pages, each 4KB

A logical block maps to a physical page

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

Write (a1, 100)

**Client operations**
SSD's clients read/write 4KB logical blocks

Many physical SSD blocks; each holds 4 pages, each 4KB

A logical block maps to a physical page

Table 100 → 00

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

Flash Chip

3) Program(01)

Client operations

Write (a1, 100)
Write (a2, 101)
SSD’s clients read/write 4KB logical blocks

Many physical SSD blocks; each holds 4 pages, each 4KB

A logical block maps to a physical page

<table>
<thead>
<tr>
<th>Table</th>
<th>100 ↦ 00</th>
<th>101 ↦ 01</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</td>
<td>00</td>
<td>01</td>
<td>02</td>
</tr>
<tr>
<td>Page</td>
<td>00</td>
<td>01</td>
<td>02</td>
</tr>
<tr>
<td>Content</td>
<td>a1</td>
<td>a2</td>
<td></td>
</tr>
<tr>
<td>State</td>
<td>V</td>
<td>V</td>
<td>E</td>
</tr>
</tbody>
</table>

Client operations

Write (a1, 100)
Write (a2, 101)
Example

- SSD’s clients read/write 4KB logical blocks
- Many physical SSD blocks; each holds 4 pages, each 4KB

A logical block maps to a physical page

<table>
<thead>
<tr>
<th>Block</th>
<th>Page</th>
<th>Content</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></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>
</tbody>
</table>

| Write (a1, 100) |
| Write (a2, 101) |
| Write (b1, 2000) |
| Write (b2, 2001) |

<table>
<thead>
<tr>
<th>Table</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>00</td>
</tr>
<tr>
<td>101</td>
<td>01</td>
</tr>
<tr>
<td>2000</td>
<td>02</td>
</tr>
<tr>
<td>2001</td>
<td>03</td>
</tr>
<tr>
<td>04</td>
<td>05</td>
</tr>
<tr>
<td>06</td>
<td>07</td>
</tr>
<tr>
<td>08</td>
<td>09</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

| Flash Chip |
| 00          |
| 01          |
| 02          |
| 03          |
| 04          |
| 05          |
| 06          |
| 07          |
| 08          |
| 09          |
| 10          |
| 11          |
SSD's clients read/write 4KB logical blocks

Many physical SSD blocks; each holds 4 pages, each 4KB

A logical block maps to a physical page

<table>
<thead>
<tr>
<th>Table</th>
<th>100 → 00</th>
<th>101 → 01</th>
<th>2000 → 02</th>
<th>2001 → 03</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</td>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
</tr>
<tr>
<td>Page</td>
<td>00 01 02 03</td>
<td>04 05 06 07</td>
<td>08 09 10 11</td>
<td></td>
</tr>
<tr>
<td>Content</td>
<td>a1 a2 b1 b2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>State</td>
<td>V V V V</td>
<td>i i i i</td>
<td>i i i i</td>
<td>i i i i</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
<td></td>
<td>Flash Chip</td>
<td></td>
</tr>
</tbody>
</table>

Client operations

Write (c1, 100)

Erase(01)
SSD’s clients read/write 4KB logical blocks.

Many physical SSD blocks; each holds 4 pages, each 4KB.

A logical block maps to a physical page.

Client operations

Program(04)

Write (c1, 100)
SSD’s clients read/write 4KB logical blocks

Many physical SSD blocks; each holds 4 pages, each 4KB

A logical block maps to a physical page

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

Flash Chip

Write (c1, 100)

Client operations
SSD's clients read/write 4KB logical blocks

Many physical SSD blocks; each holds 4 pages, each 4KB

A logical block maps to a physical page

Table

<table>
<thead>
<tr>
<th>Block</th>
<th>Page</th>
<th>Content</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00</td>
<td>a1 a2</td>
<td>V V V</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>b1 b2 c1</td>
<td>V E E E</td>
</tr>
<tr>
<td></td>
<td>02</td>
<td></td>
<td>i i i i</td>
</tr>
<tr>
<td></td>
<td>03</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Write (c1, 100)
SSD’s clients read/write 4KB logical blocks

Many physical SSD blocks; each holds 4 pages, each 4KB

A logical block maps to a physical page

<table>
<thead>
<tr>
<th>Table</th>
<th>Block</th>
<th>Page 00</th>
<th>Page 01</th>
<th>Page 02</th>
<th>Page 03</th>
<th>2000</th>
<th>2001</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td></td>
</tr>
<tr>
<td>Content</td>
<td>a1</td>
<td>a2</td>
<td>b1</td>
<td>b2</td>
<td>c1</td>
<td>c2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>State</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>E</td>
<td>i</td>
</tr>
</tbody>
</table>

- Client operations
  - Write (c1, 100)
  - Write (c2, 101)
SSD’s clients read/write 4KB logical blocks

Many physical SSD blocks; each holds 4 pages, each 4KB

A logical block maps to a physical page

<table>
<thead>
<tr>
<th>Table</th>
<th>100</th>
<th>04</th>
<th>101</th>
<th>05</th>
<th>2000</th>
<th>02</th>
<th>2001</th>
<th>03</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</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>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>c1</td>
<td>c2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>State</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>E</td>
<td>E</td>
</tr>
</tbody>
</table>

Flash Chip

Write (c1, 100)
Write (c2, 101)
Garbage Collection

- Reclaim dead blocks
  - find a block with garbage pages
  - copy elsewhere the block’s live pages
  - use Mapping Table to distinguish live pages from dead
  - make block available for writing again
Shrinking the Mapping Table

Per-page mapping is memory hungry

- 1TB SSD, 4KB pages, 4B/MTE: 1GB Mapping Table!
Shrinking the Mapping Table

Per-page mapping is memory hungry

- 1TB SSD, 4KB pages, 4B/MTE: 1GB Mapping Table!

Per-block mapping? Decreases MT size by

The Idea: Divide logical block address space in chunks of the size of a physical block

think of logical block address as

E.g., logical block 41

Map all logical blocks within a chunk C to the same physical block B

unlike direct mapping, C can over time map to different Bs (better wear leveling!)
Shrinking the Mapping Table

Assume every chunk is 4 logical blocks, mapped to some physical block.

Then, to find the location of a logical block L:

1. Use the high order bits of L’s address to determine the chunk C that L belongs to.
2. Find the physical block B that chunk C is mapped to.
3. Use least significant bits in L’s address to identify the page within B that stores L.

To find logical block 2001:

- $2001 \div 4$ identifies the chunk that holds logical block 2001.
- $2001 \mod 4$ identifies the page within that chunk that holds logical block 2001.
Per-block Mapping

Reading is easy...

<table>
<thead>
<tr>
<th>Table</th>
<th>500 ➔ 04</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</td>
<td>00 02 03</td>
</tr>
<tr>
<td>Page</td>
<td>00 01 04</td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
<tr>
<td>State</td>
<td>i i i i i</td>
</tr>
</tbody>
</table>

... but writing a page c′ requires reading in the whole block and writing it elsewhere

<table>
<thead>
<tr>
<th>Table</th>
<th>500 ➔ 08</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</td>
<td>00 02 03</td>
</tr>
<tr>
<td>Page</td>
<td>00 01 04</td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
<tr>
<td>State</td>
<td>i i i i i</td>
</tr>
</tbody>
</table>
Hybrid Mapping

- Set aside a few physical blocks to implement log mapped per-page
- Use per-block mapping for the other blocks
- On read
  - search for logical block in Log Table; then go to Data Table (which keeps per-block mapping)
- Periodically, pay the price to copy out content from the log blocks so it can be mapped per block
  - storing contiguous logical blocks in the same physical block may cause write amplification
- For wear leveling, rotate the blocks used for logging
Performance (Throughput)

- Huge difference between SSD and HDD for random I/O
- Not so much for sequential I/O

On SSDs
- Sequential still better than random
- FS design tradeoffs for HDD still apply
- Sequential reads perform better than writes
  - Sometimes you have to erase
- Random writes perform much better than random reads
  - Log transform random accesses into sequential accesses!

<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 840Pro 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>