F2FS: A New File System for Flash Storage

Changman Lee, Dongho Shim, Joo-Young Hwang, Sang-yeun Cho
Memory Business Unit,
Samsung Electronics Co., Ltd.

Presented by Joo-Young Hwang
at USENIX FAST 2015
Contents

- Introduction
- Design
  - Flash Friendly On-disk Layout
  - Cost-effective Index Structure
  - Multi-head logging
  - Cleaning
  - Adaptive Logging
  - Recovery
- Evaluation
  - Experimental Setup
  - Mobile Benchmark
  - Server Benchmark
  - Multi-head Logging Effect
  - Cleaning Cost Analysis
  - Adaptive Logging Performance (under aged condition)
- Conclusion
Random writes is bad to flash storage device.
  - Free space fragmentation
    → Sustained random write performance degrades
    → Lifetime reduced

Sequential write oriented file systems
  - log-structured file systems, copy-on-write file systems

Our Contribution
  - Design and implementation of a new file system to fully leverage and optimize the usage of NAND flash solutions (with block interface).
  - Performance comparison with Linux file systems (Ext4, Btrfs, Nilfs2).
    - Mobile system and server system
Key Design Considerations

- Flash-friendly on-disk layout
- Cost-effective index structure
- Multi-head logging
- Adaptive logging
- Fsync acceleration with roll-forward recovery
Flash-friendly On-Disk Layout

- **Flash Awareness**
  - All the FS metadata are located together for locality,
  - Start address of main area is aligned to the zone size,
  - File system cleaning is done in a unit of section (FTL’s GC unit).

- **Cleaning Cost Reduction**
  - Multi-head logging for hot/cold data separation.
Update propagation issue: wandering tree
One big log
F2FS Index Structure

- Restrained update propagation: node address translation method
- Multi-head log

* NAT: Node Address Table

Segment Info. Table (SIT)
Segment Summary (SSA)

Multiple logs
Multi-head Logging

- Data temperature classification
  - Node > data
  - Direct node > indirect node
  - Directory > user file

- Separation of multi-head logs in NAND flash.
  - Zone-aware log allocation for set-associative mapping FTL mapping.
  - Multi-stream interface
Cleaning

Cleaning is done in section unit.
- Section to be aligned with FTL’s GC unit.

Cleaning procedure
1. Victim selection: get a victim section through referencing Segment Info. Table (SIT).
   - Greedy algorithm for foreground cleaning job
   - Cost-benefit algorithm for background cleaning job
2. Valid block check: load parent index structures of there-in data identified from Segment Summary Area (SSA).
3. Migration: move valid blocks by checking their cross-reference
4. Mark victim section as “pre-free”.
   1. Pre-free sections are freed after the next checkpoint is made.
To reduce cleaning cost at highly aged conditions, F2FS changes write policy dynamically.

- Append logging (logging to clean segments)
  - Need cleaning operations if there is no free segment.
  - Cleaning causes mostly random read and sequential writes.
- Threaded logging\(^1\) (logging to dirty segments)
  - Reuse invalid blocks in dirty segments
  - No need cleaning
  - Cause random writes

* Node is always written with append logging policy.

Checkpoint and rollback
- Maintains shadow copy of checkpoint, NAT, SIT blocks
- Recovers the latest checkpoint
- Keeps NAT/SIT journal in checkpoint to avoid NAT, SIT writes

Create dir1 and file1,
Create checkpoint #0,
File2 update and fsync done,
SPO
Recovery
- Roll-back to the latest stable checkpoint (#0)
  ; recover dir1, file1
- Roll-forward recovery of file2’s fsync’ed data
Fsync handling
- On fsync, checkpoint is not necessary.
- Direct node blocks are written with fsync mark.

Roll-forward recovery procedure
1. Search marked direct node blocks
2. Per marked node block, identify old and new data blocks by checking the difference between the current and previous node block.
3. Update SIT; invalidate old data blocks
4. Replay new data block writes; update NAT, SIT accordingly
5. Create checkpoint
**Experimental Setup**

- Mobile and server systems
- Performance comparison with ext4, btrfs, nilfs2

<table>
<thead>
<tr>
<th>Target</th>
<th>System</th>
<th>Storage Devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mobile</td>
<td>CPU: Exynos 5410</td>
<td>eMMC 16GB:</td>
</tr>
<tr>
<td></td>
<td>Memory: 2GB</td>
<td>2GB partition: (114, 72, 12, 12)*</td>
</tr>
<tr>
<td></td>
<td>OS: Linux 3.4.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Android: JB 4.2.2</td>
<td></td>
</tr>
<tr>
<td>Server</td>
<td>CPU: Intel i7-3770</td>
<td>SATA SSD 250GB: (486, 471, 40, 140)*</td>
</tr>
<tr>
<td></td>
<td>Memory: 4GB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OS: Linux 3.14</td>
<td>PCIe (NVMe) SSD 960GB:</td>
</tr>
<tr>
<td></td>
<td>Ubuntu 12.10 server</td>
<td>(1,295, 922, 41, 254)*</td>
</tr>
</tbody>
</table>

* (Seq-Rd, Seq-Wr, Rand-Rd, Rand-Wr) in MB/s

<table>
<thead>
<tr>
<th>Target</th>
<th>Name</th>
<th>Workload</th>
<th>Files</th>
<th>File size</th>
<th>Threads</th>
<th>R/W</th>
<th>fsync</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mobile</td>
<td>iozone</td>
<td>Sequential and random read/write</td>
<td>1</td>
<td>1G</td>
<td>1</td>
<td>50/50</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>SQLite</td>
<td>Random writes with frequent fsync</td>
<td>2</td>
<td>3.3MB</td>
<td>1</td>
<td>0/100</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>Facebook-app</td>
<td>Random writes with frequent fsync</td>
<td>579</td>
<td>852KB</td>
<td>1</td>
<td>1/99</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>Twitter-app</td>
<td>generated by the given system call traces</td>
<td>177</td>
<td>3.3KB</td>
<td>1</td>
<td>1/99</td>
<td>Y</td>
</tr>
<tr>
<td>Server</td>
<td>videoserver</td>
<td>Mostly sequential reads and writes</td>
<td>64</td>
<td>1GB</td>
<td>48</td>
<td>20/80</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>fileserver</td>
<td>Many large files with random writes</td>
<td>80,000</td>
<td>128KB</td>
<td>50</td>
<td>70/30</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>varmail</td>
<td>Many small files with frequent fsync</td>
<td>8,000</td>
<td>16KB</td>
<td>16</td>
<td>50/50</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>oltp</td>
<td>Large files with random writes and fsync</td>
<td>10</td>
<td>800MB</td>
<td>211</td>
<td>1/99</td>
<td>Y</td>
</tr>
</tbody>
</table>
Mobile Benchmark

- In F2FS, more than 90% of writes are sequential.
- F2FS reduces write amount per fsync by using roll-forward recovery.
  - If checkpoint is done per fsync, write amount in SQLite insert test is 37% more than Ext4, and normalized performance is 0.88.
- Btrfs and nilfs2 performed poor than ext4.
  - Btrfs: heavy indexing overheads, Nilfs2: periodic data flush
  - For iozone-RW, Btrfs, Nilfs2 write 15%, 41% more I/Os than Ext4, respectively.
  - For iozone-RR, Btrfs has 50% more I/Os than other file systems.
Performance gain of F2FS over Ext4 is more on SATA SSD than on PCIe SSD.
- Varmail: 2.5x on the SATA SSD and 1.8x on the PCIe SSD
- Oltp: 16% on the SATA SSD and 13% on the PCIe SSD

Discard size matters in SATA SSD due to interface overhead.
- When using small discard (256KB) for F2FS, fileserver performance is degraded by 18%.
Using more logs gives better hot and cold data separation.

- 2 logs: node, data
- 4 logs: hot node, warm/cold node, hot data, warm/cold data

Test condition:
Run two workloads simultaneously:
1. Varmail (10,000 files in 100 dirs, total writes 6.5 GB)
2. Copy jpg files (500KB, 5,000 files, 2.5GB) \(\Rightarrow\) considered as cold

<table>
<thead>
<tr>
<th>Type</th>
<th>Temp.</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>Hot</td>
<td>Direct node blocks for directories</td>
</tr>
<tr>
<td></td>
<td>Warm</td>
<td>Direct node blocks for regular files</td>
</tr>
<tr>
<td></td>
<td>Cold</td>
<td>Indirect node blocks</td>
</tr>
<tr>
<td></td>
<td>Hot</td>
<td>Directory entry blocks</td>
</tr>
<tr>
<td></td>
<td>Warm</td>
<td>Data blocks made by users</td>
</tr>
<tr>
<td>Data</td>
<td>Cold</td>
<td>Data blocks moved by cleaning; Cold data blocks specified by users; Multimedia file data</td>
</tr>
</tbody>
</table>

![Graph showing number of valid blocks vs. percentage of segments]
Under high utilization, F2FS uses adaptive logging to restrain FS cleaning cost.

- Only node segment cleaning is done.
- Even in 97.5% util., WAF is less than 1.025.

Without adaptive logging, WAF\(^1\) goes up to more than 3.

Test condition:
120GB of 250GB (SATA SSD)
Util (Cold : Hot) = 80%(60:20), 90%(60:30), 95%(60:35), 97.5%(60:37.5)
Workload: 20GB 4KB random writes, 10 iterations

1. WAF: Write Amplification Factor
Adaptive logging gives graceful performance degradation under highly aged volume conditions.

- Fileserver test on SATA SSD (94% util.)
  - Sustained performance improvement: 2x/3x over ext4/btrfs.
- lozone test on eMMC (100% util.)
  - Sustained performance is similar to ext4.
Conclusion

- **F2FS features**
  - Flash friendly on-disk layout -> align FS GC unit with FTL GC unit,
  - Cost effective index structure -> restrain write propagation,
  - Multi-head logging -> cleaning cost reduction,
  - Adaptive logging -> graceful performance degradation in aged condition,
  - Roll-forward recovery -> fsync acceleration.

- **F2FS shows performance gain over other Linux file systems.**
  - 3.1x (iozone) and 2x (SQLite) speedup over Ext4,
  - 2.5x (SATA SSD) and 1.8x (PCIe SSD) speedup over Ext4 (varmail)

- **F2FS is publicly available, included in Linux mainline kernel since Linux 3.8.**
Thank You!