SPFS: On Stacking a Persistent Memory File System on Legacy File Systems

Hobin Woo¹, Daegyu Han², Seungjoon Ha¹, Sam H. Noh³,⁴, Beomseok Nam²
Samsung Electronics¹, SungKyunKwan University², UNIST³, Virginia Tech⁴
Background

- **Persistent Memory (PMEM)**
  - Low access latency
  - Byte-addressability
  - Persistency

- **Intel® Optane™ DCPMM**
  - First commercialized PMEM product
  - High-capacity and low latency
  - Intermediate layer between DRAM and SSD
  - Killed in 2022

- **Alternatives Products:** NVDIMM, MRAM, CXL, …
Motivation

- File systems for tiered storage – PMEM and conventional block storage
  - High performance
  - Low-cost capacity

- **Ziggurat, Strata**: Monolithic file system
  - Limitations of managing all types of storage in a single file system
    - Reinventing features of mature file systems
      (VFS cache, I/O scheduling, LFS, ⋯)
    - Complexity of handling multiple types of storage
    - Impractical deployment
Motivation

- File systems for tiered storage – PMEM and conventional block storage
  - High performance
  - Low-cost capacity

- Ziggurat, Strata: Monolithic file system

  Limitations of managing all types of storage in a single file system

Q: Can we reuse file systems that have been improved for decades?

- Reinventing features of mature file systems (VFS cache, I/O scheduling,

A: Stackable File System: modular, practical

- Complexity of handling multiple types of storage
- Impractical deployment
**SPFS**

- **SPFS** (Stackable Persistent Memory File System*)
  - Modular approach to storage tiering
  - Provide PMEM as persistent write-cache to PMEM-oblivious file systems

- **SPFS+x**: SPFS can be placed on top of any file system $x$ (EXT4, F2FS, XFS,⋯)

- **Goal:**
  - Absorb small synchronous writes

---

*: Available at https://github.com/DICL/spfs
SPFS Challenges

- As a stackable file system, SPFS must be lightweight
- SPFS must be effective in classifying and absorbing synchronous writes
Novel Designs of SPFS

- As a stackable file system, SPFS must be **lightweight**
  → Manages all metadata in lightweight and efficient hash tables
  → Novel *Extent Hashing* algorithm

- SPFS must be **effective in classifying and absorbing synchronous writes**
  → Lazy *Sync Point Profiler*

- SPFS+x improves performance of x by ~9.9x for synchronous workloads
Contribution #1
Sync Point Profiler
How is the profiler of SPFS different from the state-of-the-art?
### Profiler comparison

<table>
<thead>
<tr>
<th>Metric</th>
<th>Ziggurat</th>
<th>SPFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profiling Focus</td>
<td>Write size</td>
<td>Synchronicity</td>
</tr>
<tr>
<td>Metric</td>
<td>Individual write size</td>
<td>Bytes written between consecutive <code>fsync()</code> calls on the same file</td>
</tr>
<tr>
<td>Tendency</td>
<td>Eager (fast-first)</td>
<td>Lazy</td>
</tr>
<tr>
<td>Workload</td>
<td>Sync.</td>
<td>Async.</td>
</tr>
<tr>
<td>Large writes</td>
<td>PMEM</td>
<td>Disk</td>
</tr>
<tr>
<td>Small writes</td>
<td>PMEM</td>
<td>PMEM</td>
</tr>
</tbody>
</table>

- **VFS cache is better for asynchronous writes**
  - Can not resolve **synchronous small writes**
- **SPFS focuses on synchronicity**
Example: Eager Write Point Profiling vs. Lazy Sync Point Profiling

- Ziggurat

(a) Write Point Profiler
Example: Eager Write Point Profiling vs. Lazy Sync Point Profiling

- **Ziggurat**
  - Small IO detected
  - W(A) → W(B) → F() → W(C) → F() → W(C')
  - VFS Cache: Fails to leverage faster VFS cache
  - PMEM: A → B → C → C'
  - Disk

- **SPFS**
  - Large cumulated IO detected
  - W(A) → W(B) → F() → W(C) → F() → W(C')
  - VFS Cache: Benefit from VFS Cache
  - PMEM: A → B → C → C'
  - Disk

(a) Write Point Profiler

(b) Sync Point Profiler
Promotion may degrade performance if access pattern changes to
   • read-intensive
   • large non-transactional writes

Sync Factor (SF)
   • Small value if I/O pattern does not meet the criteria for small sync. writes

\[ SF_t = \alpha \cdot weight(IO\_type) + (1 - \alpha) \cdot SF_{t-1} \]

- Attenuation factor
  \( 0 < \alpha < 1 \)
  \( 0: \text{read-intensive, large update} \)
  \( C: \text{otherwise} \)

SPFS benefits from VFS cache by demoting read-intensive files to \( x \)
Contribution #2

Extent Hashing

SPFS is the first file system that manages all metadata including extents using a hash table.
### Challenge: How to hash extent (i.e., range data)
- Suppose extent $E(\text{start } 3, \text{length } 7)$, hash function $H(x) = x$

### Too many hash entries
- There are too many hash entries to fit into the available space.

### Extent Hashing
- Bound the number of entries to $O(\log_2 B)$
  - $O(1)$ for best case
Extent Hashing: Insert

- **Insert**
  - Store pointers according to the binary representation of keys in the range
- **Stride length**
  - $2^{TNZ(key)}$: Maximum distance to next pointer from the current key
  - $TNZ(x)$
    - Trailing Number of Zero bits of $x$
    - e.g., $TNZ(11_2) = 0$, $TNZ(100_2) = 2$, $TNZ(1010_2) = 1$

- **Example**

```plaintext
0000_0001_0010_0011_0100_0101_0110_0111_1000
```

# pointers: 8 → 4

Key Range: 1-8
Extent Hashing: Search

- Search
  - while (search(query key))
    - Flip the rightmost non-zero bit of query key

- **Example:** Find an extent \(7(0111_2)\) belongs to

- **Search Cost:** \(O(\log_2 B)\)
Evaluation
Experimental Setup

- **Evaluation machines**

<table>
<thead>
<tr>
<th>Server</th>
<th>Virtual</th>
<th>Processor</th>
<th>DRAM</th>
<th>PMEM</th>
<th>SSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCPMM</td>
<td>-</td>
<td>Dual Intel Xeon Gold 5215 (10 cores, 2.50 GHz)</td>
<td>128 GB DDR4</td>
<td>256 (128×2) GB Intel Optane DCPMM</td>
<td>2 TB Samsung 860 EVO mSATA SSD</td>
</tr>
<tr>
<td>NVDIMM-N</td>
<td>QEMU</td>
<td>Dual Intel Xeon Gold 5218 (16 cores, 2.30 GHz)</td>
<td>32 GB DDR4</td>
<td>16 GB Dell EMC NVDIMM-N</td>
<td>512 GB Samsung 970 NVMe SSD</td>
</tr>
</tbody>
</table>

- **File System Setup**

  - Default mount option
  - EXT4, F2FS, XFS: SSD
  - NOVA: PMEM (DCPMM or NVDIMM-N)
  - Ziggurat: PMEM + SSD
  - SPFS+*: PMEM + x (EXT4, F2FS, XFS)

※ Note: Evaluations performed on the NVDIMM-N server are marked with (NVDIMM-N) in the title
Extent Hashing: Comparison of file mapping structures

- 8000 256 MB files (KMEM-DAX)

![Graphs showing latency comparisons between different hashing techniques.](image)

- ExtentTree: Per-file, FAST and FAIR B+tree
- ExtentHash: Global, based on CCEH
- BlockHash: Global, ExtentHash stride=1, no log-scale search (same as HashFS)
Performance Effect of Stacking SPFS on $\chi$

- Mix of buffered I/O (BIO) and direct I/O (DIO) - *filesystem* workload

**Graph**

- **SPFS+$\chi$:** Benefit from steering BIO and DIO to the right device
- **NOVA & Ziggurat:** Insensitive to BIO/DIO
- **$\chi$:** Suffer from DIO

**SPFS+$\chi$ benefits from the device-aware stackable design**

- Steers BIOs to the lower file system while absorbing the DIOs in DCPMM
Experiments with FIU Trace (small NVDIMM-N)

- Replay transactional traces of FIU: Moodle, Usr1, Usr2
Conclusion

- **We designed and implemented SPFS**
  - A stackable file system for PMEM
    - Absorb order-preserving small synchronous writes
  - Take advantage of the legacy block device file systems
    - Provide faster DRAM cache and large capacity
  - Manage all file system metadata in dynamic hash tables
    - Novel Extent Hashing

- **SPFS+_x Improves performance of lower file system x by up to 9.9 ×**
Thank You :) 

Questions?

hobin.woo@samsung.com
hdg9400@skku.edu