FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory

Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang*, Jiwu Shu

Tsinghua University
*The Ohio State University

http://storage.cs.tsinghua.edu.cn
PM-aware Systems in the past decade …

BPFS [SOSP’09]  Mnemosyne [ASPLOS’11]
PMFS [Eurosys’14]  CDDS-Tree [FAST’11]
Aerie [Eurosys’14]  Heapo [Eurosys’16]
HiNFS [Eurosys’16]  wB^+Tree [VLDB’15]
NOVA [FAST’16]  NV-Tree [FAST’15]
NOVA-Fortis [SOSP’17]  LSNVMM [ATC’17]
Strata [SOSP’17]  FPTree [SIGMOD’16]
ZoFS [SOSP’19]  Hotpot [SoCC’17]
SplitFS [SOSP’19]  FAST&FAIR [FAST’18]

Before 2019: The Emulation Era

RECIPE [SOSP’19]  Level-Hashing [OSDI’18]
Octopus [ATC’17]  Wisper [ASPLOS’17]
CCEH [FAST’19]  Pisces [ATC’19]
Hardware Emulation Assumptions

Assumptions:

- Byte-addressability
- Close-to-DRAM Bandwidth
- High Write Latency
- Low Read Latency

- Cacheline & XPLLine
  - 64 bytes / 256 bytes

- Slow Write Bandwidth
  - 2.2 GB/s per DIMM
  - (1/3-1/6 of DRAM)

- Comparable Write Latency
  - ~100 ns

- High Read Latency
  - Rnd: 300 ns (3.7x of DRAM)
Hardware Emulation Assumptions

- **Byte-addressability**
  - **Fine-grained** Journaling
    - PMFS [Eurosys’16], NOVA [FAST’16], etc.
  - **Fine-grained** Caching
    - HiNFS [Eurosys’16], Tinca [SC’17], etc.
  - **Fine-grained** Index Structures
    - Level-Hashing [OSDI’18], etc.
  - **8-byte** Atomic Operations
    - FPTree [SIGMOD’16], FAST&FAIR [FAST’18]

- **Generate a large number of synchronized & small-sized I/Os.**

  - **XPLIne**: 256 bytes
  - 1/8 DRAM Bandwidth
Using a log structure: An intuitive approach

Buffer, then commit
Using a **log structure**: An intuitive approach

The idea of log structure is very successful for SSD/HDD

- SSD/HDDs prefer **sequential access pattern**
- The overhead of multiple storage accesses can be amortized via **batching**
  - Buffer up to tens of MBs of data before persist them

**Q: Can a log structure still retain its benefits with Optane DCPMMs?**

- Optane shows very close performance for random/sequential accesses
- 256-byte I/O units are enough to saturate the Optane bandwidth
  - It’s not beneficial to batch data larger than this I/O size
- Log cleaning overhead
FlatStore: An Efficient Log-Structured Key-Value Storage Engine

Simple insight: **Selective batch** to maximize the potential performance.

- Small updates are appended to the per-core log structure
- Large updates are stored separately via a persistent allocator

Techniques:

- **Compacted Log Format**: Improve the batching opportunity
- **Pipelined Horizontal batching**: Without increasing the latency

Results:

- Support both hash- and tree-based index structures
- Achieves up to **35 Mops/s** with a single server node
- **2.5 – 6.3 times** faster than existing systems
Outline

- Introduction
- Optane DC Persistent Memory Module
- FlatStore: An Efficient Log-structured Storage Engine
- Results
- Summary & Conclusion
Optane DC Persistent Memory Module

Images are reshaped from “An Empirical Guide to the Behavior and Use of Scalable Persistent Memory”, FAST'20
**FAST&FAIR [FAST’18]:** State-of-the-art Persistent B$^+$-Tree data structure

- Avoids logging and doesn’t block reads by using **synchronized 8-B atomic operations**
- Sort & balance overhead

![Graph showing Throughput (Mops/s) vs. # of Threads](image)

- **Throughput (Mops/s)**: 0, 20, 40, 60, 80, 100, 120
- **# of Threads**: 2, 4, 6, 8, 10, 12, 14, 16, 18
- **Optane-64B-Writes**
  - Throughput: ➤ 20 ➤ 40 ➤ 60 ➤ 80 ➤ 100 ➤ 120
  - # of Threads: ➤ 2 ➤ 4 ➤ 6 ➤ 8 ➤ 10 ➤ 12 ➤ 14 ➤ 16 ➤ 18
- **FAST&FAIR**
  - Throughput: ➤ 0 ➤ 0 ➤ 0 ➤ 0 ➤ 0 ➤ 0 ➤ 0 ➤ 0 ➤ 0

- **Optane-64B-Writes vs. FAST&FAIR**
  - At 2 threads: 17x
  - At 18 threads: 28x
When Log Structure Meets Optane DCPMM

- Random and sequential accesses achieve the same peak performance
- Minimal IO units to saturate bandwidth: **256-byte blocks**
  - It is not beneficial to batch more data than a single I/O unit (i.e., 256 B)
- Batching increases latency inevitably
Outline

- Introduction
- Optane DC Persistent Memory Module
  - FlatStore: An Efficient Log-structured Storage Engine
- Results
- Summary & Conclusion
Overall Architecture of FlatStore

- **B⁺-Tree**
- **Hash Index**
- **Volatile Index**
- **DRAM**
- **PM**
- **Per-core Log Structure**
- **Lazy-Persist Allocator**
- **Bitmap**
- **Free Lists**
- **Small KVs**
- **Large KVs**
- **Clients**
- **RDMA RPC**

Diagram shows the overall architecture with components like B⁺-Tree, Hash Index, Volatile Index, DRAM, PM, Per-core Log Structure, Lazy-Persist Allocator, Bitmap, Free Lists, Small KVs, Large KVs, Clients, and RDMA RPC.
Compacted Log Format

Log entries are formatted via the operation log technique

- Describe each operation, instead of recording the value

<table>
<thead>
<tr>
<th>Op</th>
<th>Emd</th>
<th>Version</th>
<th>Key</th>
<th>Ptr</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Op</th>
<th>Emd</th>
<th>Version</th>
<th>Key</th>
<th>Size</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>24</td>
<td>88</td>
<td>96</td>
</tr>
</tbody>
</table>

- **16 log entries** (256-byte) can be flushed to Optane DC altogether
Pipelined Horizontal Batching

Common wisdom: Batching increases both throughput and latency
Putting it all together

B⁺-Tree

Hash Index

Volatile Index

DRAM

PM

Per-core Log Structure

Core 1

Core 2

Core N

Lazy-Persist Allocator

Bitmap 01011100

Free Lists

Clients

RDMA RPC

- New requests
- Write KV pairs
- Prepare log entries
- Grab the lock (core 1)
- Collect log entries
- Release the lock
- Write & persist log
- Update volatile index
Lazy-persist allocator are used to store large KV pairs
  - Bitmaps describing the allocation states don’t need to be persisted synchronously, since the log entries has already record such information

Grouping the cores to conduct pipelined horizontal batching
  - The size of each group balances the contention level and batching opportunity

Non-blocking parallel log cleaning
  - Obsolete log entries are reclaimed concurrently without blocking the front-end operations

Recovery of the volatile index
  - Volatile index are kept in DRAM and is vulnerable to system/power failures
Outline

- Introduction
- Optane DC Persistent Memory Module
- FlatStore: An Efficient Log-structured Storage Engine
- Results
- Summary & Conclusion
Experimental Setup

Hardware Platform

<table>
<thead>
<tr>
<th>Server Node</th>
<th>4 Optane DCPMMs (1TB), 2 Xeon Gold 6240m CPUs (36 cores), 128 GB DRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client Nodes x11</td>
<td>2 Xeon E5-2650 v4 CPUs (24 cores), 128 GB DRAM</td>
</tr>
<tr>
<td>Switch</td>
<td>Mellanox MSB7790-ES2F Switch (100 Gbps)</td>
</tr>
</tbody>
</table>

Compared Systems

<table>
<thead>
<tr>
<th></th>
<th>CCEH</th>
<th>Tree-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash-based</td>
<td>Three level (directory, segments, buckets), 4 slots in a bucket</td>
<td>Inner nodes are placed in DRAM.</td>
</tr>
<tr>
<td>Level-Hashing</td>
<td>Two-level (top/bottom level), 4 slots in a bucket</td>
<td>FAST&amp;FAIR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All nodes are placed in PM.</td>
</tr>
</tbody>
</table>

Workloads

- Facebook ETC Pool: Mixture of small & large KV pairs
- YCSB (varying r/w ratio, item size, skewness, etc)
FlatStore’s performance is $3.9 \times$ higher than FPTree ($2^{nd}$ best) for 8-byte values

- Multiple small values can be persisted together
- FlatStore doesn’t introduce structural modification overhead

For large values (e.g., 1024-byte), FlatStore still shows $1.7 \times$ higher throughput
Macro-benchmark: Facebook ETC Pool

Facebook ETC Pool: mixture of small & large KV pairs.

- **Tiny** (1-13 bytes, 40%), zipfan distribution
- **Small** (14-300 bytes, 55%), zipfan distribution
- **Large** (> 300 bytes, 5%), uniform distribution
By introducing pipelined horizontal batching, FlatStore uses less time to collect a batch, thus achieving lower latency.

Pipelined HB contributes to improving the performance, since it dynamically collect a batch, instead of using a predefined threshold (e.g., minimal batch size).
Real PM device — Optane DCPMMs — exhibit much different hardware properties from what we assumed, which make many existing optimizations inapplicable.

We propose FlatStore to revitalize the log-structured design on Optane Memory. Key insight: Selective batch to maximize the potential performance.

- Compacted Log Format
- Pipelined Horizontal Batching

FlatStore supports hash- and tree-based index structure, which is 2.5 – 6.3 times faster than existing systems.
Thanks & QA

Tsinghua University & The Ohio State University

http://storage.cs.tsinghua.edu.cn
Log Cleaning Overhead

Workload: YCSB (64B values)

- Background cleaner reclaims the blocks without blocking the normal requests
- Log-structure only contains small-sized metadata or KV items
- Multiple GC groups (check our paper for details)

Only 10% reduction
Basic Performance of Optane DCPMMs

with 4 Optane DCPMMs

Bandwidth (GB/s) vs. # of Threads
Value size distribution in real-world workloads

Figure comes from “Workload Analysis of a Large-Scale Key-Value Store”, SIGMETRICS’12

Table comes from “Scaling Memcache at Facebook”, NSDI’13
Micro-benchmark: YCSB

Throughput (Mops/s)

Value Size (Bytes)

Put (Skew)

FPTree  FAST-FAIR  FlatStore-M
Using a **log structure**: An intuitive approach

Q: Can a log structure still retain its benefits with Optane DCPMMs?

<table>
<thead>
<tr>
<th></th>
<th><img src="image" alt="SSD" /></th>
<th><img src="image" alt="HDD" /></th>
<th><img src="image" alt="CMS" /></th>
</tr>
</thead>
<tbody>
<tr>
<td>w/o data journaling</td>
<td>🙆‍♂️</td>
<td>🙆‍♂️</td>
<td>🙆‍♂️</td>
</tr>
<tr>
<td>Sequential access</td>
<td>🙆‍♂️</td>
<td>🙆‍♂️</td>
<td>🙆‍♂️</td>
</tr>
<tr>
<td>Batching</td>
<td>🙆‍♂️</td>
<td>🙆‍♂️</td>
<td>🙆‍♂️</td>
</tr>
</tbody>
</table>

GC: 🙆‍♂️ or 🙆‍♀️ ?