μTree: a Persistent B⁺-Tree with Low Tail Latency

Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, Jiwu Shu

Tsinghua University

http://storage.cs.tsinghua.edu.cn
Contributions

(I) An empirical study: high latency spikes of index structures, especially on persistent memory.

- FAST+FAIR exhibits a 99p-ile latency of 60 µs, 600× higher than PM latency.
- Internal structural refinement operations (SROs)
- Interference overhead between concurrent threads

(II) μTree: improve tail latencies of persistent b+-trees

- Incorporates a shadow list layer underneath the b+-tree;
- Proposes the Coordinated concurrency control
- Achieves a 99p-ile latency that is one-order of magnitude lower, and 2.8 - 4.7x higher throughput.
Persistent memory data structures

Throughput-related design goals

- VLDB’20: DPTree
- FAST’15: NV-Tree
- FAST’11: CDDS Tree
- SIGMOD’16: FPTree
- VLDB’15: wB\textsuperscript{+}-Tree
- FAST’18: FAST+FAIR

More ...

Reduce persistence overhead
Reduce consistency overhead
Tail latency matters for datacenter workloads

- User-perceived latency: determined by the slowest sheep (i.e., back-end node)
- Optimize tail latency from different layer of OS: queue mgmt., core scheduling, etc.
Tail latency problem in persistent b+-tree (I)

**FAST+FAIR**
- Highly optimized with lock-free designs and avoids the logging overhead.
- FF (DRAM): places data in DRAM and removes all flush ops.
- For a target load running at 3 Mops/s: FF(PM)’s 99p latency is almost 60μs, **20x** higher than that of FF(DRAM), **600x** higher than PM latency.
Tail latency problem in persistent b+-tree (II)

Structural Refinement Operations (SRO)

- Sort, split, merge operations
- SROs incur higher data movement overhead (i.e., higher latencies)
- SROs only occur in some of PUT/DEL operations
- PUT/DEL operations that contain SROs typically appear at the tail of the latency distribution

<table>
<thead>
<tr>
<th>(µs)</th>
<th>Median</th>
<th>90p</th>
<th>99p</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM</td>
<td>1.4</td>
<td>2.4</td>
<td>4</td>
</tr>
<tr>
<td>PM</td>
<td>2.2</td>
<td>3.6</td>
<td>10.5</td>
</tr>
</tbody>
</table>
Tail latency problem in persistent b+-tree (III)

Interference between concurrent threads

- per-node lock leads to higher delaying in PM when two locking ops conflict.

<table>
<thead>
<tr>
<th>(µs)</th>
<th>Median</th>
<th>90p</th>
<th>99p</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM</td>
<td>2.3</td>
<td>9</td>
<td>34.2</td>
</tr>
<tr>
<td>PM</td>
<td>5.9</td>
<td>22.9</td>
<td>63.1</td>
</tr>
</tbody>
</table>

- Update DRAM tree nodes
- Update PM tree nodes
- Blocking
- Lock & Unlock
Outline

- Introduction
- Optane DC Persistent Memory Module
- μTree: a Persistent B⁺-Tree with Low Tail Latency
- 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
Outline

- Introduction
- Optane DC Persistent Memory Module
  - μTree: a Persistent B+-Tree with Low Tail Latency
- Results
- Summary & Conclusion
Architecture of µTree

Core idea: add a **shadow list layer** underneath the tree leaf nodes

- Leaf node: array layer + list layer
- DRAM: Tree inner nodes & leaf-array nodes
- PM: Leaf-list nodes

Examples:

- **PUT**: list layer ⇒ array layer
- **GET**: array layer ⇒ list layer

Insights are two-fold:

- Fast query with the volatile b+-tree: O(logn) & good cache locality
- Never perform SROs in PM: leaf-list layer does not require SROs
Put-Put Conflicts: *atomic instructions to the list layer + update array layer via locks*

- I. Link a list node in the list layer via atomic instructions (fine-grained CC);
- II. Acquire the lock & insert a slot in the array layer to point to the list node.

**Insight:**
- PM update operations are moved out of the locking path
- Reduce interference overhead
郭调的Concurrence Control

Leaf-array Layer (DRAM)

Leaf-list Layer (PM)

Put-Get Conflicts: *embed a version bit in the next pointer of the list layer.*

- 1. Put operation: toggle the *Version* bit before actually updating an item;
- 2. Get operations are executed in the opposite direction:
  - Locate a key-value pair by first find the slot in the array layer, and get the target list node with the pointer
  - Get: array layer ➔ list layer  ⇐  Put: list layer ➔ array layer
- Guarantee: Visible items are always persisted (avoid dirty reads)
Anomalies in *coordinated concurrency control*:

- CAS failures & Put-Del conflicts.

Recovery of the volatile tree layer

- A multi-threaded recovery mechanism is used for fast recovery.

Range queries

- Probe in the list layer directly.

Memory allocation consistency

- μTree adopts an epoch-based approach.
Outline

- Introduction
- Optane DC Persistent Memory Module
- \(\mu\text{Tree}:\) a Persistent B+-Tree with Low Tail Latency
- Results
- Summary & Conclusion
Experimental Setup

Hardware Platform

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>2 Xeon Gold 6240m CPUs (36 physical cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM</td>
<td>192 GB (32GB/DIMM)</td>
<td></td>
</tr>
<tr>
<td>PM</td>
<td>6 Optane DCPMMs (1.5 TB, 256 GB/DIMM),</td>
<td></td>
</tr>
<tr>
<td>Operating System</td>
<td>Ubuntu 18.04.3 LTS, Linux 4.15.0</td>
<td></td>
</tr>
</tbody>
</table>

Compared Systems

<table>
<thead>
<tr>
<th>System</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPTree</td>
<td>Non-leaf nodes are placed in DRAM; HTM + Locking for CC</td>
</tr>
<tr>
<td>FAST&amp;FAIR</td>
<td>All nodes are placed in PM; lock-free reads; no logging overhead.</td>
</tr>
</tbody>
</table>

Workloads

- YCSB (varying r/w ratio, item size, skewness, etc.)
- Redis (a multi-threaded version)
Micro-benchmark: YCSB

- For a target load running at 2 Mops/s, μTree delivers one order magnitude lower 99th percentile latency;
- For a target tail latency of 20 µs, μTree achieves 5.8x higher throughput.
Summary & Conclusion

- Recent work implement PM-aware data structures by improving their throughput-related performance. Scant attention has been paid to tail latency.
  - Overhead of structural refinement operations (SROs);
  - Overhead of cross-thread interference.
- We propose µTree that takes tail latency into consideration.
- Key insight: a shadow list layer to (1) avoid SRO overheads in PM, and (2) support fine-grained concurrency control
- µTree achieves a 99p-ile latency that is one order magnitude lower, and improves throughput by 1.8 – 3.7 times.
Thanks & QA

Tsinghua University
http://storage.cs.tsinghua.edu.cn