CCL-BTree: A Crash-Consistent Locality-Aware B+-Tree for Reducing XPBuffer-Induced Write Amplification in Persistent Memory

Zhenxin Li, Shuibing He, Zheng Dang, Peiyi Hong, Xuechen Zhang †, Rui Wang, Fei Wu

EuroSys 2024
Persistent memory (PM)

The first commercially available PM device -- Intel Optane Persistent Memory

- Memory-like speed and byte addressability
  - Low latency (~100 ns for small I/O)
  - High bandwidth (3 GB/s write and 8 GB/s read per DIMM)
  - Byte-addressable using load/store instructions

- Storage-like capacity and persistence
  - Up to 3TB (6 * 512 GB) per socket
  - Durable storage like SSD
Two types of write amplification in PM

Small writes suffer from write amplification in two hardware layers!

Optane DCPMM → XPBuffer → 3DX-Point Media

- CPU Cache
  - User Data: 0-64 B
  - Cacheline: 64 B

- XPLine: 256 B

XPI: \( XBI = \frac{\text{the amount of data written to media}}{\text{the amount of users’ data}} \)

Which one has a greater impact on write performance?
CLI vs. XBI amplification

- One socket with 4 * 128 GB Intel Optane DCPMMs 200 series

(a): Fix the value of XBI and increase the CLI by four times

(b): Fix the value of CLI and increase the XBI by four times

---

XBI-amplification determines the performance when the PM bandwidth is exhausted!
XBI-amplification in persistent B+-trees

Most existing persistent B+-trees focus on reducing CLI amplification, not XBI amplification

- One socket with 4 * 128 GB Intel Optane DCPMMs 200 series
- 48 threads, 8B key, 8B value, warm up 50M KVs & upsert 50M KVs

![Graph showing CLI and XBI amplification for different B+-tree algorithms](image)

(a) Uniform distribution

Avg XBI-amplification: 37.1

(b) Zipfian distribution

Avg XBI-amplification: 12.4

Severe XBI-amplification
Our Solution: CCL-BTree

Main components:

- Buffer node
  
  Cache multiple small KVs in buffer nodes and flush them to PM in batch

- Write ahead log
  
  Maintain the crash consistency of data in buffer nodes

---

Diagram:

- DRAM
  
  Buffer Node

- Inner Node

- PM
  
  Leaf Node

- 256 B

- T1: Leaf-node centric buffering

- T2: Write-conservative logging

- T3: Locality-aware garbage collection

- Write-Ahead Log

- Append
Technique 1: leaf-node centric buffering

Merge contiguous small writes and flush them to leaf nodes in batch

- Directly insert into the leaf node
- Trigger one XPLine flush for each KV
- Severe XBI-amplification

(a) Naïve insert

- Introduce buffer node (# slots = 2)
- Trigger one XPLine flush for 3 KVs
- Lower XBI-amplification

(b) Buffered insert
Technique 2: write-conservative logging

Skip the **unnecessary** log operations while ensuring the crash consistency

- **(a) Naive logging**
  - A naive logging method writes logs for each new KV

- **(b) Write-conservative logging**
  - CCL-BTree skips logging for KVs that trigger the insertion of leaf nodes when the buffer nodes are full
Technique 3: locality-aware garbage collection

Convert random leaf node access to **sequential logging**

(a) Naive GC
- The naïve GC flushes all KVs in buffer nodes to leaf nodes, which incurs many random accesses.

(b) Locality-aware GC
- Our GC only copies a portion of KVs in buffer nodes to new logs in sequential manner.
Theoretical performance analysis

- Insert $K$ key-value pairs with uniform distributions
- $N_{batch}$: the number of slots in a buffer node.
- The log item size: 24 bytes (16-byte KV and 8-byte timestamp)

😊 When $N_{batch} = 2$ (the default value), CCL-BTree reduces 60.4% XPLine writes

<table>
<thead>
<tr>
<th># of 256 B XPLine writes</th>
<th>Traditional B+-tree</th>
<th>CCL-BTree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leaf Node</td>
<td>$K$</td>
<td>$\frac{1}{N_{batch} + 1} \times K$</td>
</tr>
<tr>
<td>Log</td>
<td>0</td>
<td>$\frac{24}{256} \times \frac{N_{batch}}{N_{batch} + 1} \times K$</td>
</tr>
<tr>
<td>Total</td>
<td>$K$</td>
<td>$\frac{256 + 24 \times N_{batch}}{256 \times (N_{batch} + 1)} \times K$</td>
</tr>
</tbody>
</table>
Experimental setup

- **Platform**
  - 2 sockets
  - CPU: Intel Xeon Gold 5318Y (24 physical/48 logical cores)
  - DRAM: 4 * 16 GB 2666 Mhz
  - PM: 4 * 128 GB Intel Optane DCPMMs 200 series

- **Target comparisons**
  - FPTree [SIGMOD’16]
  - FAST&FAIR [FAST’18]
  - DPTree [VLDB’19]
  - uTree [VLDB’20]
  - LBTree [VLDB’20]
  - PACTree [SOSP’21]

- **Other settings**
  - Use the same 256 B tree node size for each index
  - Use pre-allocated PM pools from the local socket for all indexes to minimize the allocation overhead
Overall performance

- 8-byte key and 8-byte value
- Warm up the index with 50 million KVs and then run each test with 50 million operations

For write workloads, CCL-BTree outperforms other B+-Tree variants by **1.4x** at least

For read workloads, CCL-BTree has competitive performance
Other tests

Demonstrate the high efficiency of CCL-Btree under various workloads.


More details

- System optimizations
  - NUMA-friendly PM accesses
  - Concurrency control.
  - Variable-size KVs.

- Evaluation results
  - Comparison with persistent log structures
  - Realistic datasets test
  - Latency test
  - Recovery
  - …

Please check our paper!
Conclusion

- We propose CCL-BTree to address the XPBuffer-induced write amplification issue in persistent B+-trees.
  - Leaf-node centric buffering
  - Write-conservative logging
  - Locality-aware garbage collection

- CCL-BTree improves the insert throughput by 1.97x to 9.35x

- The source code is available at https://github.com/ISCS-ZJU/CCL-BTree
CCL-BTree: A Crash-Consistent Locality-Aware B+-Tree for Reducing XPBuffer-Induced Write Amplification in Persistent Memory

Zhenxin Li, Shuibing He, Zheng Dang, Peiyi Hong, Xuechen Zhang†, Rui Wang, Fei Wu

Thanks & QA

Open source: https://github.com/ISCS-ZJU/CCL-BTree
Contact information: zhenxin@zju.edu.cn