Building Scalable Dynamic Hash Tables on Persistent Memory

Baotong Lu
Chinese University of Hong Kong

Xiangpeng Hao
Simon Fraser University

Tianzheng Wang
Simon Fraser University

Eric Lo
Chinese University of Hong Kong
Persistent Memory Primer

- Persistent Memory (PM)
  - Bigger capacity/lower price (vs. DRAM)
  - Close-to-DRAM speed
  - Byte-addressable
  - Persistent

- Intel Optane DCPMM
  - Distinct properties: latency and bandwidth

PM-based hash table: DRAM-based emulation vs. Optane DCPMM

B. Lu, X. Hao, T. Wang, E. Lo
Building Scalable Dynamic Hash Tables on Persistent Memory
Dynamic Hashing

- **Extendible hashing** and **Linear hashing**
  - Directory + buckets

**Partial Expansion**

**Scalable Structure!**
## Goals - High Throughput

<table>
<thead>
<tr>
<th>Goals</th>
<th>What’s wrong on emulation-based PM hashing?</th>
</tr>
</thead>
<tbody>
<tr>
<td>High (single-thread) throughput</td>
<td>• <strong>Optane</strong>: End-to-end random write latency &lt; read latency [1]</td>
</tr>
<tr>
<td></td>
<td>• Wrong assumption before:</td>
</tr>
<tr>
<td></td>
<td>• Write latency &gt; read latency in emulation</td>
</tr>
<tr>
<td></td>
<td>• Previous write-limited hash [2, 3, 4] ==&gt; excessive PM reads</td>
</tr>
</tbody>
</table>

[1] An empirical guide to the behavior and use of scalable persistent memory *FAST’20*
[2] Revisit hash table design or phase change memory, *INFLOW’15*
[4] Write-optimized and high-performance hashing index for persistent memory *OSDI’18*
Goals - Scalable on Multi-core

<table>
<thead>
<tr>
<th>Goals</th>
<th>What’s wrong on emulation-based PM hashing?</th>
</tr>
</thead>
<tbody>
<tr>
<td>High (single-thread) throughput</td>
<td>• Excessive PM reads</td>
</tr>
<tr>
<td>Scalable on multi-core</td>
<td>• <strong>Optane</strong>: PM bandwidth is much lower than DRAM, especially for small random writes (~14X) [1]!</td>
</tr>
</tbody>
</table>

[1] An empirical guide to the behavior and use of scalable persistent memory FAST’20
[2] Write-optimized and high-performance hashing index for persistent memory OSDI’18
[3] Write-optimized dynamic hashing for persistent memory FAST’19
## Goals - Scalable on Multi-core

<table>
<thead>
<tr>
<th>Goals</th>
<th>What’s wrong on emulation-based PM hashing?</th>
</tr>
</thead>
<tbody>
<tr>
<td>High (single-thread) throughput</td>
<td>• Excessive PM reads</td>
</tr>
<tr>
<td>Scalable on multi-core</td>
<td>• <strong>Optane</strong>: PM bandwidth is much lower than DRAM, especially for small random writes (~14X) [1]!</td>
</tr>
<tr>
<td></td>
<td>• Previous designs [2, 3] ==&gt; heavy-weight read-write lock!</td>
</tr>
</tbody>
</table>

![Lightweight concurrency control!](image)

[1] An empirical guide to the behavior and use of scalable persistent memory *FAST’20*
[2] Write-optimized and high-performance hashing index for persistent memory *OSDI’18*
[3] Write-optimized dynamic hashing for persistent memory *FAST’19*

---

B. Lu, X. Hao, T. Wang, E. Lo

Building Scalable Dynamic Hash Tables on Persistent Memory
Goals - Full Functionality

<table>
<thead>
<tr>
<th>Goals</th>
<th>What’s wrong on emulation-based PM hashing?</th>
</tr>
</thead>
<tbody>
<tr>
<td>High throughput</td>
<td>• Excessive PM reads</td>
</tr>
<tr>
<td>Scalable on multi-core</td>
<td>• Heavyweight concurrency control</td>
</tr>
<tr>
<td>Instant recovery</td>
<td>• Recovery time = $O(\text{data size})$ [4]</td>
</tr>
<tr>
<td>High space utilization</td>
<td>• Low load factor [4]</td>
</tr>
<tr>
<td>Variable-length key</td>
<td>• Inefficient support [1, 2, 3, 4]</td>
</tr>
</tbody>
</table>

Without sacrificing any desirable properties!

[1] Revisit hash table design or phase change memory. *INFLOW’15*
[2] A write-friendly hashing scheme or non-volatile memory. *MSST’17*
[3] Write-optimized and high-performance hashing index for persistent memory *OSDI’18*
[4] Write-optimized dynamic hashing for persistent memory *FAST’19*
Dash = Fast + Scalable + Full Functionality

<table>
<thead>
<tr>
<th>Goals</th>
<th>What’s wrong on emulation-based PM hashing?</th>
</tr>
</thead>
<tbody>
<tr>
<td>High (single-thread) throughput</td>
<td>• Excessive PM reads</td>
</tr>
<tr>
<td>Scalable on multi-core</td>
<td>• Heavyweight concurrency control</td>
</tr>
<tr>
<td>Instant recovery</td>
<td>• Recovery time = O(data size)</td>
</tr>
<tr>
<td>High space utilization</td>
<td>• Low load factor</td>
</tr>
<tr>
<td>Efficient support of variable-length key</td>
<td>• Inefficient support</td>
</tr>
</tbody>
</table>

Reduce both PM reads and writes + With full functionality

Extendible hashing and linear hashing
Problem 1: Excessive PM reads

• Intel Optane DCPMM
  • End-to-end read latency > write

• Fingerprint [1, 2]: one-byte hash
  • Filtering PM reads

• Goals achieved
  • High (single-thread) throughput
  • Efficient variable-length key support

---

[1] FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-tree for Storage Class Memory. SIGMOD’16
Effectiveness of Fingerprint

• With-Fingerprint vs. Without-Fingerprint
  • The effect of reducing PM accesses
  • The effect of variable-length key support
Problem 2: Heavyweight Concurrency Control

Read-write lock on PM

- RW-lock incurs writes for search
- Even the search operations could easily exhaust the PM bandwidth!

Dash: optimistic lock

- No writes for search operations
- Goals achieved: scalable on multicore for search operations
Problem 3: Low Space Utilization

Pre-mature splits

- Low space efficiency
- Excessive PM writes

Dash: bucket load balancing

1. Balanced insert
2. Displace
3. Stash

Pre-mature split
Load Factor

• Goals achieved: high load factor (> 90%).

Max. Load factor after adding different techniques

Overall Load factor comparison
Problem 4: Crash Consistency

Persistency on PM
- Volatile CPU cache
  - Cacheline flush + fence
- Failure atomicity of PM is 8B
  - Writes on hash table > 8B

Dash: crash consistency design
- **SMO (segment split)**
- Basic operations (Insert/Delete)

Directory:
```
  4 10 33
  12 11 15
  32 31
  40
```

Segment:
```
  4 12 10 33
  32 40 11 15
  40
```

Dash: Scalable hashing on persistent memory
Problem 4: Crash Consistency - SMO

• Fully relying on logging is heavyweight
• Recoverable segment split without logging
  • State variable: indicate which step the SMO is in
  • Side link: find the new segment

(a) Initial state.
(b) Allocate a new segment and do the rehashing.
(c) Update the directory entry and local depth.
Problem 5: Non-instant Recovery

Slow recovery process
• Heavyweight recovery work
  • Locked locks
  • Ongoing SMO
  • ...

Dash: instant recovery
• Lazy recovery
  • Amortize real recovery work to runtime
• Accept requests instantly upon system restart
  • Global version and local version
Evaluation: Scalability

- Setting: 24-core CPU, 128GB X 6 Optane DCPMM (on all six channels)
- Near-linear scalability for search operation
- Better insert performance than state-of-the-arts

![Graphs showing scalability for search, insert, and mixed operations]
Evaluation: Recovery

• Dash: Instant recovery regardless of indexed data size
• Multi-threading helps throughput to return to normal very fast!

Recovery time (ms)

<table>
<thead>
<tr>
<th>Hash Table</th>
<th>Number of indexed records (million)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>40</td>
</tr>
<tr>
<td>Dash-EH</td>
<td>57</td>
</tr>
<tr>
<td>Dash-LH</td>
<td>57</td>
</tr>
<tr>
<td>CCEH</td>
<td>113</td>
</tr>
<tr>
<td>Level hashing</td>
<td>53</td>
</tr>
</tbody>
</table>
Conclusions

• Emulation-based PM hash tables considered harmful on real PM
• **Dash**: fast, scalable and full functionality hash index on real PM

Applicability

• **Extendible hashing/linear hashing**, also applicable to other hashing schemes
• Ready to be integrated with KV store/DB systems

• Full-paper: *Dash: Scalable Hashing on Persistent Memory, PVLDB 2020*

Open source at: [https://github.com/baotonglu/dash](https://github.com/baotonglu/dash)