Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory

Pengfei Zuo, Yu Hua, Jie Wu
Huazhong University of Science and Technology, China

OSDI 2018
## Persistent Memory (PM)

- Non-volatile memory as PM is expected to replace or complement DRAM as main memory
  - Non-volatility, low power, large capacity

<table>
<thead>
<tr>
<th></th>
<th>PCM</th>
<th>ReRAM</th>
<th>DRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read (ns)</td>
<td>20-70</td>
<td>20-50</td>
<td>10</td>
</tr>
<tr>
<td>Write (ns)</td>
<td>150-220</td>
<td>70-140</td>
<td>10</td>
</tr>
<tr>
<td>Non-volatility</td>
<td>√</td>
<td>√</td>
<td>×</td>
</tr>
<tr>
<td>Standby Power</td>
<td>~0</td>
<td>~0</td>
<td>High</td>
</tr>
<tr>
<td>Density (Gb/cm²)</td>
<td>13.5</td>
<td>24.5</td>
<td>9.1</td>
</tr>
</tbody>
</table>

Index Structures in DRAM vs PM

➢ Index structures are critical for memory & storage systems
➢ Traditional indexing techniques originally designed for DRAM become inefficient in PM
  – Hardware limitations of NVM
    • Limited cell endurance
    • Asymmetric read/write latency and energy
    • Write optimization matters
  – The requirement of data consistency
    • Data are persistently stored in PM
    • Crash consistency on system failures
Tree-based vs Hashing Index Structures

- **Pros:** good for range query
- **Cons:** $O(\log(n))$ time complexity for point query
- Ones for PM have been widely studied
  - CDDS B-tree [FAST’11]
  - NV-Tree [FAST’15]
  - wB+-Tree [VLDB’15]
  - FP-Tree [SIGMOD’16]
  - WORT [FAST’17]
  - FAST&FAIR [FAST’18]
Tree-based vs Hashing Index Structures

Tree-based index structures

- **Pros**: good for range query
- **Cons**: $O(\log(n))$ time complexity for point query
- Ones for PM have been widely studied
  - CDDS B-tree [FAST’11]
  - NV-Tree [FAST’15]
  - wB+-Tree [VLDB’15]
  - FP-Tree [SIGMOD’16]
  - WORT [FAST’17]
  - FAST&FAIR [FAST’18]

Hashing index structures

- **Pros**: constant time complexity for point query
- **Cons**: do not support range query
- Widely used in main memory
  - Main memory databases
  - In-memory key-value stores, e.g., Memcached and Redis
  - When maintained in PM, multiple non-trivial challenges exist
    - Rarely touched by existing work
Challenges of Hashing Indexes for PM

① High overhead for **consistency guarantee**

- Ordering memory writes
  - Cache line flush and memory fence instructions
- Avoiding partial updates for non-atomic writes
  - Logging or copy-on-write (CoW) mechanisms

Volatile caches

Non-volatile memory

8-byte width
Challenges of Hashing Indexes for PM

1. High overhead for consistency guarantee
2. Performance degradation for reducing writes

- Hashing schemes for DRAM usually cause many extra writes for dealing with hash collisions [INFLOW’15, MSST’17]
- Write-friendly hashing schemes reduce writes but at the cost of decreasing access performance
  - PCM-friendly hash table (PFHT) [INFLOW’15]
  - Path hashing [MSST’17]
Challenges of Hashing Indexes for PM

1. High overhead for consistency guarantee
2. Performance degradation for reducing writes
3. Cost inefficiency for resizing hash table
   - Double the table size and iteratively rehash all items
   - Take $O(N)$ time to complete
   - $N$ insertions with cache line flushes & memory fences
## Existing Hashing Index Schemes for PM

(“×”: bad, “✓”: good, “–”: moderate)

<table>
<thead>
<tr>
<th></th>
<th>Bucketized Cuckoo (BCH)</th>
<th>PFHT(^1)</th>
<th>Path Hashing(^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory efficiency</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Search</td>
<td>✓</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Deletion</td>
<td>✓</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Insertion</td>
<td>×</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>NVM writes</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Resizing</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Consistency</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>


# Existing Hashing Index Schemes for PM

("\x": bad, "\✓": good, "\−": moderate)

<table>
<thead>
<tr>
<th></th>
<th>Bucketized Cuckoo (BCH)</th>
<th>PFHT(^1)</th>
<th>Path Hashing(^2)</th>
<th>Level Hashing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory efficiency</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Search</td>
<td>✓</td>
<td>−</td>
<td>−</td>
<td>✓</td>
</tr>
<tr>
<td>Deletion</td>
<td>✓</td>
<td>−</td>
<td>−</td>
<td>✓</td>
</tr>
<tr>
<td>Insertion</td>
<td>×</td>
<td>−</td>
<td>−</td>
<td>✓</td>
</tr>
<tr>
<td>NVM writes</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Resizing</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Consistency</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>


Level Hashing

Write-optimized & High-performance Hash Table Structure

TL:

BL:

One movement

Resizing support

Consistency support

Cost-efficient In-place Resizing Scheme

Low-overhead Consistency Guarantee Scheme

Inplace Resizing Scheme
Write-optimized Hash Table Structure

1. Multiple slots per bucket
2. Two hash locations for each key
3. Sharing-based two-level structure
4. At most one movement for each successful insertion
Write-optimized Hash Table Structure

① Multiple slots per bucket
② Two hash locations for each key
③ Sharing-based two-level structure
④ At most one movement for each successful insertion

![Diagram showing hash table structure]
Write-optimized Hash Table Structure

1. Multiple slots per bucket
2. Two hash locations for each key
3. Sharing-based two-level structure
4. At most one movement for each successful insertion
Write-optimized Hash Table Structure

1. Multiple slots per bucket
2. Two hash locations for each key
3. Sharing-based two-level structure
4. At most one movement for each successful insertion
Write-optimized Hash Table Structure

① Multiple slots per bucket
② Two hash locations for each key
③ Sharing-based two-level structure
④ At most one movement for each successful insertion
Write-optimized Hash Table Structure

- **Write-optimized**: only 1.2% of insertions incur one movement
- **High-performance**: constant-scale time complexity for all operations
- **Memory-efficient**: achieve high load factor by evenly distributing items

![Diagram of hash table structure](image)
Cost-efficient In-place Resizing

➢ Put a new level on top of the old hash table and only rehash items in the old bottom level
Cost-efficient In-place Resizing

- Put a new level on top of the old hash table and only rehash items in the old bottom level
Cost-efficient In-place Resizing

- Put a new level on top of the old hash table and only rehash items in the old bottom level

TL: 0 1 2 3 4 5 6 7
BL: 2N-2 2N-3 2N-4
IL: (the interim level)

BL: 2N-2 2N-3 2N-4
Cost-efficient In-place Resizing

- Put a new level on top of the old hash table and only rehash items in the old bottom level

---

TL: 0 1 2 3 4 5 6 7
BL: 2N-2 2N-3 2N-4 2N-1
IL: (the interim level)

Rehashing
Cost-efficient In-place Resizing

- Put a new level on top of the old hash table and only rehash items in the old bottom level
Cost-efficient In-place Resizing

- Put a new level on top of the old hash table and only rehash items in the old bottom level
  - The new hash table is exactly double size of the old one
  - Only 1/3 buckets (i.e., the old bottom level) are rehashed
Low-overhead Consistency Guarantee

- A token associated with each slot in the open-addressing hash tables
  - Indicate whether the slot is empty
  - A token is 1 bit, e.g., “1” for non-empty, “0” for empty

A bucket:

```
1 1 0 0 KV₀ KV₁
```

Tokens

Slots
**Low-overhead Consistency Guarantee**

- A token associated with each slot in the open-addressing hash tables
  - Indicate whether the slot is empty
  - A token is 1 bit, e.g., “1” for non-empty, “0” for empty
- Modifying the token area only needs an atomic write
  - Leveraging the token to perform log-free operations
Log-free Deletion

➢ Delete an existing item

Delete

1 1 0 0 KV₀ KV₁
Log-free Deletion

Delete an existing item

1 1 0 0 KV₀ KV₁

Delete

Modify the token in an atomic write

1 0 0 0 KV₀ KV₁
Log-free Deletion

Delete an existing item

Log-free insertion and log-free resizing

– Please find them in our paper
Consistency Guarantee for Update

➢ If directly update an existing key-value item in place
  – Inconsistency on system failures
Consistency Guarantee for Update

- If directly update an existing key-value item in place
  - Inconsistency on system failures
- A straightforward solution is to use logging

Expensive!
Our scheme: check whether there is an empty slot in the bucket storing the old item
- Yes: log-free update
- No: using logging

**Update**

1. Write KV1’ in an empty slot

2. Modify the two tokens in an atomic write
Our scheme: check whether there is an empty slot in the bucket storing the old item
- Yes: log-free update
- No: using logging

Update

① Write KV1’ in an empty slot

② Modify the two tokens in an atomic write
Performance Evaluation

➢ Both in DRAM and simulated PM platforms
  – Quartz (Hewlett Packard)
    • A DRAM-based performance emulator for PM

➢ Comparisons
  – Bucketized cuckoo hashing (BCH) [NSDI’13]
  – PCM-friendly hash table (PFHT) [INFLOW’15]
  – Path hashing [MSST’17]
  – In PM, implement their persistent versions using our proposed log-free consistency guarantee schemes
Level hashing has the best insertion performance in both DRAM and NVM.
Opportunistic log-free update scheme reduces the update latency by 15%~52%, i.e., speeding up the updates by $1.2 \times - 2.1 \times$
The search latency of level hashing is close to that of BCH, which is much lower than PFHT and path hashing.
Level hashing reduces the resizing time by about 76%, i.e., speeding up the resizing by $4.3 \times$. 
Concurrent Throughput

Concurrent level hashing:
Support multiple-reader multiple-writer concurrency via simply using fine-grained locking

Concurrent level hashing has $1.6 \times - 2.1 \times$ higher throughput than libcuckoo\(^1\), due to locking fewer slots for insertions

---

\[1\] X. Li et al.. “Algorithmic improvements for fast concurrent cuckoo hashing”, Eurosys, 2014.
Conclusion

➢ Traditional indexing techniques originally designed for DRAM become inefficient in PM
➢ We propose level hashing, a write-optimized and high-performance hashing index scheme for PM
  – Write-optimized hash table structure
  – Cost-efficient in-place resizing
  – Log-free consistency guarantee
➢ $1.4 \times -3.0 \times$ speedup for insertion, $1.2 \times -2.1 \times$ speedup for update, and over $4.3 \times$ speedup for resizing
Thanks! Q&A
(Poster #10)

Open-source code: https://github.com/Pfzuo/Level-Hashing