A Write-friendly Hashing Scheme for Non-volatile Memory Systems

Pengfei Zuo and Yu Hua

Huazhong University of Science and Technology, China
## Non-volatile Memory

- NVMs are expected to replace DRAM and SRAM

<table>
<thead>
<tr>
<th></th>
<th>SRAM</th>
<th>DRAM</th>
<th>PCM</th>
<th>RRAM</th>
<th>STT-RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-volatile</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Read (ns)</td>
<td>1</td>
<td>10</td>
<td>20~70</td>
<td>10</td>
<td>2~20</td>
</tr>
<tr>
<td>Write (ns)</td>
<td>1</td>
<td>10</td>
<td>150~220</td>
<td>50</td>
<td>5~35</td>
</tr>
<tr>
<td>Standby Power</td>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Scalability (nm)</td>
<td>20</td>
<td>20</td>
<td>5</td>
<td>11</td>
<td>32</td>
</tr>
<tr>
<td>Endurance (10^N)</td>
<td>&gt; 15</td>
<td>&gt; 15</td>
<td>7~8</td>
<td>8~10</td>
<td>12~15</td>
</tr>
</tbody>
</table>

- NVMs vs. DRAM & SRAM
  - ✔️ No-volatile, high scalability, and low standby power
  - ✖️ Limited endurance and asymmetric properties
Rethinking Data Structures on NVMs

- How could in-memory and in-cache data structures be modified to efficiently adapt to NVMs?

- Previous work mainly focuses on tree-based structures
  - CDDS-tree (FAST 2011)
  - NV-tree (FAST 2015)
  - wB+-tree (VLDB 2015)
  - FP-tree (SIGMOD 2016)
  - Write Optical Radix Tree (FAST 2017)

- Hash tables are also widely used in main memory and caches
  - Main memory database
  - In-memory key-value store, e.g., Memcached, Redis
  - In-cache index (ICS 2014, MICRO 2015)
Existing Hashing Schemes on NVMs

(a) Chained Hashing

(b) Linear Probing

(c) 2-choice Hashing

(d) Cuckoo Hashing

- Insertion Deletion → Extra Writes
- Deletion → Extra Writes
- Low Space Utilization: ~35%

Our Design Goals

✔ Minimize NVM writes while ensuring high performance
Our Scheme: Path Hashing

✓ Position Sharing  ✓ Double-path Hashing  ✓ Path Shortening

A novel hash-collision resolution method without extra NVM writes

Deliver high performance on space utilization and request latency
Our Scheme: Path Hashing

✓ Position Sharing ✓ Double-path Hashing ✓ Path Shortening

A novel hash-collision resolution method resulting in no extra NVM writes

Deliver high performance on space utilization and request latency

Addressable cells by hash functions

Un-addressable, shared standby cells
Our Scheme: Path Hashing

✓ Position Sharing
✓ Double-path Hashing
✓ Path Shortening

A novel hash-collision resolution method resulting in no extra NVM writes

Deliver high performance on space utilization and request latency

Insertion and deletion without extra modifications and data movements

Problem: One path can only deal with at most $L$ hash collisions
Our Scheme: Path Hashing

- **Position Sharing**
- **Double-path Hashing**
- **Path Shortening**

A novel hash-collision resolution method resulting in no extra NVM writes

Deliver high performance on space utilization and request latency

Using two different hash functions to compute two paths → high space utilization
Our Scheme: Path Hashing

- Position Sharing
- Double-path Hashing
- Path Shortening

A novel hash-collision resolution method resulting in no extra NVM writes

Deliver high performance on space utilization and request latency

Problem: Each query may probe many nodes in a high tree
Our Scheme: Path Hashing

- Position Sharing
- Double-path Hashing
- Path Shortening

A novel hash-collision resolution method resulting in no extra NVM writes

Deliver high performance on space utilization and request latency

Observation: The bottom levels provide a few standby positions while increasing the length of the read path.

Path Shortening: Removing multiple levels in the bottom.
Our Scheme: Path Hashing

- Position Sharing
- Double-path Hashing
- Path Shortening

A novel hash-collision resolution method resulting in no extra NVM writes.

Deliver high performance on space utilization and request latency.

Observation: The bottom levels provide a few standby positions while increasing the length of the read path.

Evaluation: Reserving a small part of levels can also achieve a high space utilization.

Path Shortening: Removing multiple levels in the bottom.
Physical Storage Structure of Path Hashing

An array:

Level 4
Level 3
Level 2
Physical Storage Structure of Path Hashing

- No pointers
- The nodes in a path can be accessed in parallel for insertion, query and deletion
Experimental Configurations

- Gem5: a full system simulator
- NVMain: a main memory simulator for NVMs

<table>
<thead>
<tr>
<th>Processor and Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
</tr>
<tr>
<td>Private L1 cache</td>
</tr>
<tr>
<td>Shared L2 cache</td>
</tr>
<tr>
<td>Shared L3 cache</td>
</tr>
<tr>
<td>Memory Controller</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Main Memory using PCM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity</td>
</tr>
<tr>
<td>Read latency</td>
</tr>
<tr>
<td>Write latency</td>
</tr>
</tbody>
</table>

- Datasets: Random Number, Document Word, Fingerprint
NVM Writes

No extra writes

No extra writes
Path hashing achieves up to 95% space utilization ratio
Reserving a small part of levels can also achieve a high space utilization ratio.
Request Latency

Insertion Latency (us)

- Chained
- Linear
- 2-choice
- Cuckoo
- Path

Deletion Latency (us)

- Chained
- Linear
- P-2-choice
- P-Cuckoo
- Path

Query Latency (us)

- Chained
- Linear
- P-2-choice
- P-Cuckoo
- Path

Load Factor

0.6 0.8
Conclusion

- Existing main hashing schemes usually cause many extra writes to NVMs
- We propose a write-friendly hashing scheme, path hashing, without extra writes while having high performance
  - Position sharing
  - Double-path hashing
  - Path shortening
- Experimental results on gem5 with NVMMain
  - No extra writes
  - Up to 95% space utilization ratio
  - Low request latency
Thanks! Q&A

Open-source Code: https://github.com/Pfzuo/Path-Hashing
E-mail: pfzuo@hust.edu.cn
Homepage: http://pfzuo.github.io/about/