NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems

Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong and Bingsheng He

Data Storage Institute, A-STAR, Singapore
Nanyang Technological University
Fast 15
OUTLINE

- Motivation
- NV-Tree
- Evaluation
- Related Work
- Conclusion
Motivation

- Next generation of non-volatile memory (NVM)
  - Provides DRAM-like performance and disk-like persistency
  - Can replace both DRAM and disk to build a single level system

- In-NVM data consistency is required
  - Ordering memory writes
  - Fundamental for keeping data consistency
  - Non-trivial in NVM due to CPU design
    - E.g, w1, (MFENCE,CLFLUSH,MFENCE), w2, (MFENCE,CLFLUSH,MFENCE)
Motivation

- Making B+tree or its variants consistent is expensive

- Reason: CPU cache line invalidation is amplified due to CLFLUSH

B+ tree: 16X slower
CDDS-tree: 20X slower
Motivation

CFLUSH What?

Sorting entries in LN produces up to >90% of total CLFLUSH

Sort LN
LN
Sort LN
IN

Percentage

0%
20%
40%
60%
80%
100%

Node Size

LCB CDDS

512B

1 Million Insertion

LCB CDDS

1024B

LCB CDDS

2048B

LCB CDDS

4096B
NV-Tree

- Design decisions
  - Selectively Enforce Data Consistency
    - Only enforces consistency on LNs (Leaf Nodes)
  - Keep Entries in LN Unsorted
  - Organizing IN in Cache-optimized Format
    - All INs are stored in a consecutive memory space and located by offset instead of pointers
    - All nodes are aligned to CPU cacheline
NV-Tree

- Overview
NV-Tree

- **IN Design**
  - All INs are stored in a continuous memory space
    - Memory address of node id
      \[ = \text{addr} + \text{id} \times \text{size\_IN} \]
    - \( \text{addr} \): memory address of node 0
    - \( \text{size\_IN} \): size of a IN
  - Can be located without pointers
  - No consistency required
  - Locating the next IN during tree traverse
    - E.g. one IN have \( m \) children
      \[ \text{Memory address of the } k\text{-th} (k = 1 \ldots m) \text{ child of node id} = \text{addr} + (\text{id} \times m + k) \times \text{size\_IN} \]
NV-Tree

- **LN Design**
  - Dynamically allocated and aligned to CPU cacheline
    - Every LN has a pointer from PLN
  - Data is encapsulated in **LN_Elements**
  - **LN_Elements** are unsorted and append-only
  - In-node search is bounded by **nElements**
  - **No partial modification**
    - Append **LN_Element**,(MFENCE,CLFLUSH,MFENCE)
    - Atomically increase **nElements** by 1 (8-byte), (MFENCE,CLFLUSH, MFENCE)
NV-Tree

- Insert/Update/Delete

Step 1: Append LN_Elements

Step 2: Atomically increase nElement
NV-Tree

- Insert Algorithm

```
Algorithm 2: NV-Tree Insertion

Input: k: key, v: value, r: root
Output: SUCCESS/FAILURE

begin
  if r = NULL then /* Create new tree
      with the given KV-pair. */
      r ← create_new_tree(k, v);
  return SUCCESS

  leaf ← find_leaf(k, r);
  if LN has space for new KV-pair then
    newElement ← CreateElement(k, v);
    flush(newElement);
    AtomicInc(leaf.number);
    flush(leaf.number);
  else
    leaf_split_and_insert(leaf, k, v)
  return SUCCESS
```
**NV-Tree**

- **Split**
  - All data modified by unfinished split is *invisible* upon system failure
  - Those data become *visible after* a 8-byte atomic update

![Diagram](Image)

*Figure 4: Example of LN Split*
## NV-Tree

- Split / Replace / Merge
  - *Minimal Fill Factor (MFF)*

<table>
<thead>
<tr>
<th>Percentage of <strong>Valid</strong> Elements in Full Node</th>
<th>Percentage of <strong>Total</strong> Elements in Right Sibling</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; MFF</td>
<td>-</td>
<td>Split</td>
</tr>
<tr>
<td>&lt; MFF</td>
<td>&gt; MFF</td>
<td>Replace</td>
</tr>
<tr>
<td>&lt; MFF</td>
<td>&lt; MFF</td>
<td>Merge</td>
</tr>
</tbody>
</table>
NV-Tree

- **Rebuilding**
  - Triggered when a PLN is full
    - Due to the fixed position of each IN

- **Strategy**
  - Rebuild-from-PLN
    - Reuse the existing `<key, LN_pointer>` array in PLNs
  - Rebuild-from-LN
NV-Tree

- Recovery

<table>
<thead>
<tr>
<th>Shutdown Type</th>
<th>Shutdown Action</th>
<th>Recovery Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>Store all INs to NVM</td>
<td>Retrieve the root</td>
</tr>
<tr>
<td>System Failure</td>
<td>N/A</td>
<td>Rebuild-from-LN</td>
</tr>
</tbody>
</table>

- Instant recovery
  - Normal shut down and NVM has enough space
    - Keep all INs in NVM
  - Otherwise
    - Rebuilding-from-LN
Evaluation

- Experiment Setup

- NVDIMM server
  - IntelXeonE5-2650
  - 2.4GHz, 512KB/2MB/20MB L1/L2/L3 Cache
  - 16GBDRAM,16GBNVDIMM
  - NVDIMM has the same performance as DRAM
Evaluation-Insertion Performance

- LCB+Tree (Log-based Consistent B+Tree)
- CDDS-Tree
- NV-Tree

<table>
<thead>
<tr>
<th></th>
<th>LCB+Tree</th>
<th>CDDS-Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1M</td>
<td>15.2X</td>
<td>8X</td>
</tr>
<tr>
<td>10M</td>
<td>6.3X</td>
<td>9.7X</td>
</tr>
<tr>
<td>100M</td>
<td>5.3X</td>
<td>8.2X</td>
</tr>
</tbody>
</table>
Evaluation-Update/Delete/Search

Update

Delete

Search

Comparable to CDDS-Tree with larger nodes

Comparable to both competitors

<table>
<thead>
<tr>
<th>LCB+Tree</th>
<th>CDDS-Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6X</td>
<td>8.5X</td>
</tr>
</tbody>
</table>
Evaluation-Mixed Workloads

1 million operations (insertion/search)
- On an existing NV-Tree with 1 million entries

![Bar chart showing execution time comparison between LCB+Tree, CDDS-Tree, and NV-Tree for different write/read ratios.]

<table>
<thead>
<tr>
<th>w/r</th>
<th>LCB+Tree</th>
<th>CDDS-Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>90%/10%</td>
<td>6.6X</td>
<td>10X</td>
</tr>
<tr>
<td>10%/90%</td>
<td>2X</td>
<td>2.8X</td>
</tr>
</tbody>
</table>
Evaluation-CPU Cache Efficiency

Intel vTune Amplifier
- Number of LOADs

![Graph showing the comparison of LOADs for different node sizes between LCB+Tree and CDDS-Tree. The graph indicates that LCB+Tree has up to 90% reduced LOADs compared to CDDS-Tree.]

- Number of L3 Misses

![Graph showing the comparison of L3 Misses for different node sizes between LCB+Tree and CDDS-Tree. The graph indicates that LCB+Tree has up to 83% reduced L3 Misses compared to CDDS-Tree.]

- Comparison Table
<table>
<thead>
<tr>
<th>LCB+Tree</th>
<th>CDDS-Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up to 90%</td>
<td>Up to 92%</td>
</tr>
<tr>
<td>reduced</td>
<td>reduced</td>
</tr>
</tbody>
</table>

- Comparison Table
<table>
<thead>
<tr>
<th>LCB+Tree</th>
<th>CDDS-Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up to 83%</td>
<td>Up to 90%</td>
</tr>
<tr>
<td>reduced</td>
<td>reduced</td>
</tr>
</tbody>
</table>
Evaluation-Rebuilding

1/10/100 Million Insertion, 512B/1KB/2KB/4KB Node Size
- Rebuilding time is neglectable
  - 0.01% - 2.77%

Rebuilding strategy
- Rebuild-from-PLN is 22% - 47% faster
Evaluation-End To End Performance

KV-Store
- NV-Store
- Redis
  - Volatile / Consistent

Workloads
- YCSB
  - StatusUpdate (read-latest)
  - SessionStore (update-heavy)

5% Insertion
Up to $3.2X$ speedup

50%/50% Search/Update
Up to $4.8X$ speedup
Related Work

- **CDDS-Tree**
  - Uses flush to enforce consistency on all the tree nodes.
  - In order to keep entries sorted, when an entry is inserted to a node, all the entries on the right side of the insertion position need to be shifted.

- **Others**
  - MFENCE and CLFLUSH
    - E.g. Mnemosyne
  - Epoch
    - E.g. BPFS NV-Heaps
Conclusion

Providing data consistency for B⁺-tree or its variants in Non-volatile Memory is **costly**
- *Ordering memory writes* is non-trivial and expensive in NVM
- Logs are needed because the size of atomic writes is limited
- Keeping in-node data *sorted* produces *unnecessary* ordered writes

**NV-Tree**
- Consistent, log-free and cache-optimized
- Decouple leaf nodes (LN) and internal nodes (IN)
  - LN
    - Enforce consistency
    - Unsorted keys with append-only scheme
  - IN
    - Reconstructable, no consistency guaranteed
    - Sorted keys and cache-optimized layout