WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems

Se Kwon Lee
K. Hyun Lim¹, Hyunsub Song, Beomseok Nam, Sam H. Noh

UNIST
¹Hongik University
### Persistent Memory (PM)

- Persistent memory is expected to replace both DRAM & NAND

<table>
<thead>
<tr>
<th></th>
<th>NAND</th>
<th>STT-MRAM</th>
<th>PCM</th>
<th>DRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-volatility</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>Read (ns)</td>
<td>$2.5 \times 10^4$</td>
<td>5 - 30</td>
<td>20 – 70</td>
<td>10</td>
</tr>
<tr>
<td>Write (ns)</td>
<td>$2 \times 10^5$</td>
<td>10 - 100</td>
<td>150 - 220</td>
<td>10</td>
</tr>
<tr>
<td>Byte-addressable</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Density</td>
<td>185.8 Gbit/cm$^2$</td>
<td>0.36 Gbit/cm$^2$</td>
<td>13.5 Gbit/cm$^2$</td>
<td>9.1 Gbit/cm$^2$</td>
</tr>
</tbody>
</table>

Indexing Structure for PM Storage Systems

B+Tree
Consistency Issue of B+tree in PM

- **B+tree is a block-based index**
  - Key sorting ➔ Block granularity write
  - Rebalancing ➔ Multi-blocks granularity write

- **Persistent memory**
  - Byte-addressable ➔ Byte granularity write
  - Write reordering

  Can result in consistency problem
Consistency Issue of B+tree in PM

- Traditional case

Volatile

CPU Caches

Write reordering

Not persistent data

DRAM

Block based storage

Non-volatile

Block granularity update

Write reordering

Not persistent data
Consistency Issue of B+tree in PM

- **PM case**

  Byte granularity update
  Write reordering
  Persistent data

---

Garbage data persistently stored
### Durability
- **CLFLUSH** (Flush cache line)
  - Can be reordered

### Ordering
- **MFENCE** (Load and Store fence)
  - Order CPU cache line flush instructions
Primitives for Data Consistency in PM

- Durability
  - CLFLUSH (Flush cache line)
    - Can be reordered

- Ordering
  - MFENCE (Load and Store fence)
    - Order CPU cache line flush instructions

Serialization of CLFLUSH and MFENCE is known to cause large overhead
Primitives for Data Consistency in PM

- **Atomicity**
  - 8-byte failure atomicity
    - Need only CLFLUSH
  - Logging or CoW based atomicity (more than 8 bytes)
    - Requires duplicate copies
Primitives for Data Consistency in PM

- Non-volatile Data area
- Log area

Primitives for Data Consistency in PM

- Atomicity
  - 8-byte failure atomicity
    - Need only CLFLUSH
  - Logging or CoW-based atomicity (more than 8 bytes)
    - Requires duplicate copies

Logging increases cache line flush overhead
B+tree Variants for Persistent Memory

How can we ensure consistency using failure-atomic writes without logging?

Unsorted keys → Append-only with metadata
Failure-atomic update of metadata

Unsorted key → Decreases search performance
B+tree Variants for Persistent Memory

- Logging still necessary
  - Multi-block granularity updates due to node splits and merges
    - Cannot update atomically
  
- Logging-based solution
  - wB+Tree, FPTree

- Tree reconstruction based solution
  - NVTree

large overhead
B+tree Variants for Persistent Memory

Key sorting

Rebalancing

Fundamental characteristics of B+tree cause problems
B+ tree Variants for Persistent Memory

Why use B+ trees in the first place?

Perhaps there is a better tree data structure more suited for PM?
Our Contributions

- Show Radix Tree is a suitable data structure for PM
- Propose optimal radix tree variants WORT and WOART
  - WORT: Write Optimal Radix Tree
  - WOART: Write Optimal redesigned Adaptive Radix Tree (ART)
    - Optimal: maintain consistency only with single failure-atomic write without any duplicate copies
§ Deterministic structure
Radix Tree

- Deterministic structure
  - No key comparison

---

ACA  ACC  ACZ

A  C  A

A  C  Z

ACA  ACC  ACZ

C  CAC
### Deterministic structure
- No key comparison
  - Only 8-byte pointer entries
  - Implicitly stored keys
Radix Tree

- **Deterministic structure**
  - No key comparison
    - Only 8-byte pointer entries
    - Implicitly stored keys
    - No problem caused by key sorting
**Radix Tree**

- **Deterministic structure**
  - No key comparison
    - Only 8-byte pointer entries
    - Implicitly stored keys
    - No problem caused by key sorting
  - No modification of other keys
    - Single 8-byte pointer write per node
    - Easy to use failure-atomic write
Problem of Deterministic Structure

- For sparse key distribution
  - Waste excessive memory space $\rightarrow$ Optimized through path compression
Path Compression in Radix Tree

- **Path compression**
  - Search paths that do not need to be distinguished can be removed

Unnecessary search path

```
ACA
ACC
ACZ
...
A
C
C
...
A
C
...

Unnecessary search path can be removed by path compression.
```
**Path Compression in Radix Tree**

- **Path compression**
  - Common search path is compressed in header
  - Improve memory utilization & indexing performance

![Diagram of Path Compression in Radix Tree]

**Diagram Notes**
- A, C, Z: Nodes in the Radix Tree
- ACA, ACC, ACZ: Paths with compressed headers
Path compression split

Prefix keys are not equal
AZ != AC
Node Split with Path Compression

- Path compression split

\[ \text{Split} \quad AC \]

① New parent allocation

\[ \text{AZA} \]

\[ \text{A} \quad \text{C} \quad \text{Z} \]

\[ \text{ACA} \quad \text{ACC} \quad \text{ACZ} \]
Node Split with Path Compression

- Path compression split

② Decompression of old common prefix
Node Split with Path Compression

- Path compression split

However, this split process causes consistency problem in PM.
Path compression
Problem
in PM
Path compression split

- cause updates of multiple nodes
- have to employ expensive logging methods
Path compression

Solution
- **Failure-atomic path compression**
  - Add *node depth field* to compression header

```c
struct Header {
    unsigned char depth;
    unsigned char PrefixArr[7];
};
```

Compression header (8 bytes)

- ACA
- ACC
- ACZ
- **Failure-atomic path compression**
  - Add **node depth field** to compression header

AZA to be inserted

Compression header (8 bytes)

A
AC
ACA

C
ACC

Z
AC
ACZ

Our solution
### Failure-atomic path compression

- Add **node depth field** to compression header

---

**Consistent state**

- **Add node depth field to compression header**

**Inconsistent state**

- **Decompression of old common prefix**

- **Crash**
Failure-atomic path compression

- Failure detection in WORT
  - Depth in a header $\neq$ Counted depth $\rightarrow$ Crashed header
### Failure-atomic path compression

- **Failure recovery in WORT**
  - Compression header can be reconstructed → Atomically overwrite

**Diagram:**
- Consistent state
- Inconsistent state
- Compression header (8 bytes)
- ACA, ACC, ACZ
Our proposed radix tree variant is optimal for PM

- Consistency is always guaranteed with a single 8-byte failure-atomic write without any additional copies for logging or CoW

WORT (Write Optimal Radix Tree)

WOART (Write Optimal Adaptive Radix Tree)

1. Failure-atomic path compression

2. Redesigned adaptive node

```
struct Header {
  unsigned char depth;
  unsigned char PrefixArr[7];
}
```
### Experimental environment

#### System configuration

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
</tr>
<tr>
<td>Intel Xeon E5-2620V3 X 2</td>
</tr>
<tr>
<td><strong>OS</strong></td>
</tr>
<tr>
<td>Linux CentOS 6.6 (64bit) kernel v4.7.0</td>
</tr>
<tr>
<td><strong>PM</strong></td>
</tr>
<tr>
<td>Emulated with 256GB DRAM</td>
</tr>
<tr>
<td>Write latency: Injecting additional stall cycles</td>
</tr>
</tbody>
</table>
### Evaluation

- **Experimental environment**

**Comparison group**

<table>
<thead>
<tr>
<th>Radix tree variants</th>
<th>B+tree variants</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORT</td>
<td>wB+Tree (VLDB' 15)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PM</th>
<th>PM</th>
<th>DRAM</th>
<th>DRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Radix tree variant" /></td>
<td><img src="image2.png" alt="B+tree variant" /></td>
<td><img src="image3.png" alt="DRAM tree" /></td>
<td><img src="image4.png" alt="DRAM tree" /></td>
</tr>
</tbody>
</table>

---

**Note:** The images above illustrate the comparison of different data structure variants for experimental evaluation.
Evaluation

- Experimental environment

Synthetic Workload Characteristics

- Dense \([1 \ldots N]\)

- Sparse \([1 \ldots 2^{64}]\)

- Clustered \([1 \ldots 2^{64}]\)
Insertion performance

- WORT outperform the B+tree variants in general
### Insertion performance

- WORT outperform the B+tree variants in general
  - DRAM-based internal node $\rightarrow$ more favorable performance for FPTree
### Insertion performance

- WORT vs wB+Tree
  - Performance differences increase in proportion to write latency

![Graphs showing performance comparison](image-url)
- **CLFLUSH count per operation**
  - B-tree variants incur more cache flush instructions
**Search performance**

- WORT always perform better than B+Tree variants
Range query performance

- Performance gap for range query decreases for PM indexes compared with it between WORT and original B+Tree
  - B+Tree variants do not keep the keys sorted → Rearrangement overhead
- **MC-benchmark performance on Memcached**
  - WORT outperform B+Tree variants in both SET and GET
    - Additional indirection & flush overhead in B-tree variants
Conclusion

- Showed suitability of radix tree as PM indexing structure

- Proposed optimal radix tree variants WORT and WOART
  - Optimal: maintain consistency only with single failure-atomic write without any duplicate copies
Any question?