RECIPE : Converting Concurrent DRAM Indexes to Persistent-Memory Indexes

Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap*, Taesoo Kim, Vijay Chidambaram

*On the job market
Persistent Memory (PM)

- New storage class memory technology
- Performance similar to DRAM
- Non-volatile & high-capacity
  - Up-to 6TB on a single machine
Indexing on PM

- PM has high capacity and low latency
  - 6TB on a single machine → 100 billion 64-byte key-value pairs
- Indexing data on PM is crucial for efficient data access
PM Indexes

PM Indexes need to achieve three goals simultaneously:

- Cache Efficiency
- Concurrency
- Crash Consistency
PM Indexes

• Cache Efficiency
  • Persistent memory is attached to the memory bus
  • 3x higher latency than DRAM $\rightarrow$ More cache-sensitive
PM Indexes

- Concurrency
  - High concurrency is necessary for scalability on any modern multicore platform
PM Indexes

- Crash Consistency
  - CPU cache is still volatile
  - Arbitrarily-evicted cache lines $\rightarrow$ Persistence reordering
PM Indexes

• Crash Consistency
  • CPU cache is still volatile
  • Arbitrarily-evicted cache lines $\rightarrow$ Persistence reordering

Program order
write (log);
write (commit);
PM Indexes

• Crash Consistency
  • CPU cache is still volatile
  • Arbitrarily-evicted cache lines $\rightarrow$ Persistence reordering

Persistence reordering
write (log);
write (commit);

Reordered
PM Indexes

• Crash Consistency
  • CPU cache is still volatile
  • Arbitrarily-evicted cache lines → Persistence reordering

Persistence reordering
write (log);
write (commit);

Crash

Inconsistency
PM Indexes

• Crash Consistency
  • CPU cache is still volatile
  • Arbitrarily-evicted cache lines → Persistence reordering
    • **Flush**: persist writes to PM
    • **Fence**: ensure one write prior another to be persisted first

### Consistent persistence ordering

write (log)
flush (log)
fence ()
write (commit)
flush (commit)
fence ()
Challenge in building PM indexes

Correctness condition: return previously inserted data without data loss or corruption
Challenge in building PM indexes

Concurrency and crash consistency interact with each other, a bug in either can lead to data loss.
Bug in Concurrent PM Index

• We found **bugs** in FAST&FAIR [FAST’18] and CCEH [FAST’19]
• FAST&FAIR: Concurrent PM-based B+Tree
  • One bug in concurrency mechanism
  • Two bugs in recovery mechanism
• CCEH: Concurrent PM-based dynamic hash table
  • One bug in concurrency mechanism
  • One bug in recovery mechanism
How can we reduce the effort involved in building concurrent, crash-consistent PM indexes?

Answer: We can convert concurrent DRAM indexes to PM indexes with low effort.

Insight: Isolation and Crash Consistency are similar.
How can we reduce the effort involved in building concurrent, crash-consistent PM indexes?

Approach: Convert concurrent DRAM indexes to PM indexes with low effort

Insight: Isolation and Crash Consistency are similar
DRAM Index

- Already designed for cache efficiency and concurrency
DRAM Index

CRASH VULNERABLE

CONCURRENCY

CACHE EFFICIENCY

VOLATILE
Challenge in Conversion

- Require minimal changes to DRAM index
  - Without modifying the original design principles of DRAM index
Insight for Conversion

• Similar semantics between isolation and consistency\(^1\)

• Isolation
  • Return consistent data while multiple active threads are running

• Crash consistency
  • Return consistent data even after a crash happens at any point

1. Steven Pelley et al., Memory Persistency, ISCA’14
Insight for Conversion

• Similar semantics between isolation and consistency\(^1\)

Approach: reuse mechanisms for isolation in DRAM indexes to obtain crash consistency

1. Steven Pelley et al., Memory Persistency, ISCA’14
• Principled approach to convert DRAM indexes into PM indexes

• Case study of changing five popular DRAM indexes

• Conversion involves different data structures such as Hash Tables, B+ Trees, and Radix Trees

• Conversion required modifying \( \leq 200 \) LOC

• Up-to 5.2x better performance in multi-threaded evaluation

https://github.com/utsaslab/RECIPE
Outline

• Overall Intuition
• Conversion Conditions
• Conversion Example: Masstree
• Assumptions & Limitations
• Evaluation
Outline

- **Overall Intuition**
- Conversion Conditions
- Conversion Example: Masstree
- Assumptions & Limitations
- Evaluation
Overall Intuition for Conversion

• Blocking algorithms
  • Use explicit locks to prevent the conflicts of threads to shared data

• Non-blocking algorithms
  • Use well-defined invariants and ordering constraints without locks
  • Employed by most high-performance DRAM indexes
Overall Intuition for Conversion

• Non-blocking algorithms
  • Readers **Detect** and **Tolerate** inconsistencies
    • E.g., Ignore duplicated keys
Overall Intuition for Conversion

• Non-blocking algorithms
  • Readers Detect and Tolerate inconsistencies
    • E.g., Ignore duplicated keys
  • Writers also **Detect**, but **Fix** inconsistencies
    • E.g., Eliminate duplicated keys
Overall Intuition for Conversion

- Non-blocking algorithms
  - Readers Detect and Tolerate inconsistencies
  - Writers also Detect, but Fix inconsistencies
  - Helping mechanism \(1 \approx\) Crash Recovery \(2\)
  - Such indexes are *inherently* crash consistent

1. Keren Censor-Hillel et al., Help!, PODC’15
2. Ryan Berryhill et al., Robust shared objects for non-volatile main memory, OPODIS’15
• Not all DRAM indexes can be converted with **low effort**

• Exploit **inherent crash recovery** in the index

• Provide **specific conditions** that must hold for a DRAM index to be converted

• Provide a matching **conversion actions** for each condition
Outline

• Overall Intuition
• **Conversion Conditions**
• Conversion Example: Masstree
• Assumptions & Limitations
• Evaluation
Three Conversion Conditions

• Condition 1: Updates via Single Atomic Store
• Condition 2: Writers fix inconsistencies
• Condition 3: Writers don’t fix inconsistencies
• Conditions are not exhaustive!
Three Conversion Conditions

• **Condition 1: Updates via Single Atomic Store**
• **Condition 2: Writers fix inconsistencies**
• **Condition 3: Writers don’t fix inconsistencies**
Condition 1: Updates via Single Atomic Store

- Non-blocking readers, (Non-blocking or Blocking) writers
- Updates become visible to other threads via single atomic commit store
Condition 1: Updates via Single Atomic Store

- Updates become visible to other threads via single atomic commit store
- Conversion: Add flushes after each store and bind final atomic store using fences
Three Conversion Conditions

- Condition 1: Updates via Single Atomic Store
- **Condition 2: Writers fix inconsistencies**
- Condition 3: Writers don’t fix inconsistencies
Condition 2: Writers fix inconsistencies

- Non-blocking readers and writers (don’t hold locks)
- Readers & Writers → Detect (✔), Tolerate (✔), Fix (✔)

Update

A sequence of ordered deterministic steps

Writer 1 ➔ Step 1 ➔ Step 2 ➔ Step 3 ➔ Commit Step
Condition 2: Writers fix inconsistencies

- Non-blocking readers and writers (don’t hold locks)
- Readers & Writers → Detect (✔), Tolerate (✔), Fix (✔)
Condition 2: Writers fix inconsistencies

- Non-blocking readers and writers (don’t hold locks)
- Readers & Writers → Detect (✔), Tolerate (✔), Fix (✔)
Condition 2: Writers fix inconsistencies

- Readers & Writers $\rightarrow$ Detect (✓), Tolerate (✓), Fix (✓)
  - Inherently crash recoverable
Condition 2: Writers fix inconsistencies

- Readers & Writers → Detect (✔), Tolerate (✔), Fix (✔)
  - Inherently crash recoverable

```
Step 1  Step 2  Step 3
Writer 1  Crash  PM
Commit Step
```
Condition 2: Writers fix inconsistencies

- Readers & Writers $\rightarrow$ Detect (✔), Tolerate (✔), Fix (✔)
- Inherently crash recoverable
Condition 2: Writers fix inconsistencies

- Readers & Writers $\rightarrow$ Detect (✓), Tolerate (✓), Fix (✓)
  - Inherently crash recoverable
  - Conversion: Adding **flushes** and **fences** after each store and specific loads

![Diagram showing steps of a thread process]

- Step 1: Commit
- Step 2: PM
- Step 3: Fence
- Commit Step: Flushes
Three Conversion Conditions

- Condition 1: Updates via Single Atomic Store
- Condition 2: Writers fix inconsistencies
- **Condition 3: Writers don’t fix inconsistencies**
Condition 3: Writers don’t fix inconsistencies

- Non-blocking readers, Blocking writers (hold locks)
- Readers & Writers → Detect (✔), Tolerate (✔), Fix (✗)
Condition 3: Writers don’t fix inconsistencies

- Non-blocking readers, Blocking writers (hold locks)
- Readers & Writers → Detect (✔), Tolerate (✔), Fix (X)
Condition 3: Writers don’t fix inconsistencies

- Non-blocking readers, Blocking writers (hold locks)
- Readers & Writers $\rightarrow$ Detect (✔), Tolerate (✔), Fix (X)
Condition 3: Writers don’t fix inconsistencies

- Readers & Writers → Detect (✓), Tolerate (✓), Fix (✓)
- Conversion: **Add helping mechanism**
  - **Reuse** existing algorithm handling each step
Outline

• Overall Intuition
• Conversion Conditions
• Conversion Example: Masstree
• Assumptions & Limitations
• Evaluation
Conversion of Masstree

• Example: B-link Tree (Masstree)
Conversion of Masstree

• Example: B-link Tree (Masstree)

1. installing new sibling

2. insert middle key to parent
Conversion of Masstree

• Example: B-link Tree (Masstree)

Insert 26

Intermediate state
Conversion of Masstree

• Example: B-link Tree (Masstree)

Lookup 25

Insert 26

10 30

1 10

15 25
Conversion of Masstree

• Example: B-link Tree (Masstree)

Lookup 25

Insert 26

Detect (25 > 15)

Tolerate
Conversion of Masstree

• Example: B-link Tree (Masstree)
Conversion of Masstree

• Example: B-link Tree (Masstree)
  • Add helping mechanism to resume split

```
<table>
<thead>
<tr>
<th>Insert 30</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
```
# Conversion Results of Five DRAM Indexes

<table>
<thead>
<tr>
<th>DRAM Index</th>
<th>DS Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLHT (Cache-Line Hash Table)</td>
<td>Hash table</td>
</tr>
<tr>
<td>HOT (Height Optimized Trie)</td>
<td>Trie</td>
</tr>
<tr>
<td>BwTree [ICDE’13]</td>
<td>B+Tree</td>
</tr>
<tr>
<td>ART (Adaptive Radix Tree) [ICDE’13]</td>
<td>Radix Tree</td>
</tr>
<tr>
<td>Masstree [Eurosys’12]</td>
<td>Hybrid (B+Tree &amp; Trie)</td>
</tr>
</tbody>
</table>


Conversion Results of Five DRAM Indexes

- We produce the P-* family of PM indexes

<table>
<thead>
<tr>
<th>DRAM Index</th>
<th>PM Index</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLHT</td>
<td>P-CLHT</td>
<td>#1</td>
</tr>
<tr>
<td>HOT</td>
<td>P-HOT</td>
<td>#1</td>
</tr>
<tr>
<td>BwTree</td>
<td>P-BwTree</td>
<td>#1, #2</td>
</tr>
<tr>
<td>ART</td>
<td>P-ART</td>
<td>#1, #3</td>
</tr>
<tr>
<td>Masstree</td>
<td>P-Masstree</td>
<td>#1, #3</td>
</tr>
</tbody>
</table>
Outline

• Overall Intuition
• Conversion Conditions
• Conversion Example: Masstree
• Assumptions & Limitations
• Evaluation
Assumptions & Limitations

- Assume garbage collection in memory allocator
- Assume locks are volatile or re-initialized after a crash
- Provide low level of isolation: Read Uncommitted
- RECIPE applies only to individual data structures
Outline

• Overall Intuition
• Conversion Conditions
• Conversion Example: Masstree
• Assumptions & Limitations
• **Evaluation**
Evaluation

• How much effort is involved in converting indexes?
• What is the performance of converted indexes?
• Are the converted indexes crash consistent?
Evaluation

• How much effort is involved in converting indexes?
• What is the performance of converted indexes?
• Are the converted indexes crash consistent?
Evaluation

• How much effort is involved in converting indexes?
• What is the performance of converted indexes?
## Modified Lines of Code

- Conversion for all indexes $\rightarrow \leq 200$ LoC changes

<table>
<thead>
<tr>
<th>RECIPE-converted Indexes</th>
<th>Lines of Code</th>
</tr>
</thead>
</table>
|                          | Index Core    | Modified |%
| P-CLHT                   | 2.8K          | 30 (1%)  |
| P-HOT                    | 2K            | 38 (2%)  |
| P-BwTree                 | 5.2K          | 85 (1.6%)|
| P-ART                    | 1.5K          | 52 (3.4%)|
| P-Masstree               | 2.2K          | 200 (9%) |
Modified Lines of Code

- Conversion for all indexes $\rightarrow \leq 200$ LoC changes

Conversion for all indexes: $\leq 200$ LoC changes
$\leq 9\%$ from core code base

<table>
<thead>
<tr>
<th>Index</th>
<th>Original Size</th>
<th>LoC Changes</th>
<th>LoC Change Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-HOT</td>
<td>2K</td>
<td>38 (2%)</td>
<td></td>
</tr>
<tr>
<td>P-BwTree</td>
<td>5.2K</td>
<td>85 (1.6%)</td>
<td></td>
</tr>
<tr>
<td>P-ART</td>
<td>1.5K</td>
<td>52 (3.4%)</td>
<td></td>
</tr>
<tr>
<td>P-Masstree</td>
<td>2.2K</td>
<td>200 (9%)</td>
<td></td>
</tr>
</tbody>
</table>
Evaluation

• How much effort is involved in converting indexes?
• What is the performance of converted indexes?
Performance Evaluation

- 2-socket 96-core machine with 32MB LLC
- 768 GB Intel Optane DC PMM, 378 GB DRAM
- YCSB with 16 threads
- Ordered/Unordered indexes, Integer/String keys

<table>
<thead>
<tr>
<th>Load</th>
<th>Workload A</th>
<th>Workload B</th>
<th>Workload C</th>
<th>Workload E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion 100%</td>
<td>Insertion 50%</td>
<td>Insertion 5%</td>
<td>Point Lookup 100%</td>
<td>Insertion 5%</td>
</tr>
<tr>
<td></td>
<td>Point Lookup 50%</td>
<td>Point Lookup 95%</td>
<td></td>
<td>Range Scan 95%</td>
</tr>
</tbody>
</table>
Ordered Index

- Support both point and range operations
- P-HOT
  - Persistent Height-Optimized Trie converted by RECIPE
- FAST & FAIR [FAST’18]
  - Hand-crafted PM-based concurrent B+Tree
Ordered Index

• P-HOT produced by RECIPE conversion
• P-HOT performs up-to 5.2x better in point operations
• Cache-efficient designs of P-HOT $\rightarrow$ Low cache misses
RECIPE

• Principled approach to convert concurrent DRAM indexes into PM indexes
• Case study of changing five DRAM indexes
• Evaluations with YCSB show RECIPE indexes have better performance than hand-crafted PM indexes
• Try our indexes: https://github.com/utsaslab/RECIPE