Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree

Deukyeon Hwang
UNIST

Wook-Hee Kim
UNIST

Youjip Won
Hanyang Univ.

Beomseok Nam
UNIST/SKKU
Background – Persistent Memory

- Fast but Asymmetric Access Latency
- Non-Volatility
- Byte-Addressability
- Large Capacity
Inserting 25 into a node

(0)  10  20  30  40

(1)  10  20  30  40  40

(2)  10  20  30  30  40

(3)  10  20  25  30  40

Partially updated tree node is inconsistent

↓

Append-Only Update
Background – B+-Tree for Persistent Memory

Node Split

Node A

10  20  30  10  20  30

Node A

P1  P2  P3  P1  P2  P3

Node B

40  40  60  60

P4  P5  P6

Logging → Selective Persistence (Internal node in DRAM)
### Background – B+-Tree for Persistent Memory

- **Append-Only**
  - Unsorted keys

- **Selective Persistence**
  - Internal node → DRAM
  - Internal nodes have to be reconstructed from leaf nodes after failures
  - Logging for leaf nodes

- **Previous solutions**

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NV-Tree [FAST’15]</td>
<td>Append-Only leaf update + Selective Persistence</td>
</tr>
<tr>
<td>wB+-Tree [VLDB’15]</td>
<td>Append-Only node update + bitmap/slot array metadata</td>
</tr>
<tr>
<td>FP-Tree [SIGMOD’16]</td>
<td>Append-Only leaf update + fingerprints + Selective Persistence</td>
</tr>
</tbody>
</table>
Contributions

- Append-Only (Unsorted keys)
- Selective Persistence (DRAM + PM)
- Failure-Atomic ShifT (FAST)
- Failure-Atomic In-place Rebalancing (FAIR)
- Lock-Free Search
Modern processors reorder instructions to utilize the memory bandwidth

Memory ordering in x86 and ARM

<table>
<thead>
<tr>
<th></th>
<th>x86</th>
<th>ARM</th>
</tr>
</thead>
<tbody>
<tr>
<td>stores-after-stores</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>stores-after-loads</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>loads-after-stores</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>loads-after-loads</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Inst. w/ dependency</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

- x86 guarantees **Total Store Ordering (TSO)**
- Dependent instructions are not reordered
Pointers in B+-Tree store *unique memory addresses*
8-byte pointer can be *atomically updated*

Read transactions detect *transient inconsistency* between duplicate pointers

*transient inconsistency*
- In-flight state partially updated by a write transaction
Failure-Atomic Shift (FAST)

10  20  30  40  TSO
P1  P2  P3  P4  P5

mfence();

10  20  30  40  40
P1  P2  P3  P4  P5

TSO
Insert (25, P6) into a node using FAST

Read transactions can succeed in finding a key even if a system crashes in any step
Insert (25, P6) into a node using FAST
Insert (25, P6) into a node using FAST
Failure-Atomic ShifT (FAST)

Insert (25, P6) into a node using FAST
Insert (25, P6) into a node using FAST

Key 40 between duplicate pointers is ignored!
Insert (25, P6) into a node using FAST

Shifting P4 invalidates the left 40
Insert (25, P6) into a node using FAST
Insert (25, P6) into a node using FAST
Failure-Atomic ShifT (FAST)

Insert (25, P6) into a node using FAST
Insert (25, P6) into a node using FAST

Storing P6 validates 25
It is necessary to call clflush at the boundary of cache line...
Let’s avoid expensive logging by making read transactions be aware of rebalancing operations

- B^{link}-Tree
FAIR split a node

A read transaction can detect transient inconsistency if keys are out of order
FAIR split a node

Node A

10
20
30

P1 P2 P3 ∧

Node B

40 60

P4 P6 ∧

Setting NULL pointer validates Node B. Node A and Node B are virtually a single node.
FAIR split a node

Migrated keys can be accessed via sibling pointer
FAIR split a node

Node A

| 10 | 20 | 30 | ∧ |

P1  P2  P3

Node B

| 40 | 50 | 60 | ∧ |

P4  P5  P6
Insert a key into the parent node using FAST after FAIR split
Insert a key into the parent node using FAST after FAIR split

Node B can be accessed from Node A
Insert a key into the parent node using FAST after FAIR split

- Searching the key 50 from the root after a system crash

Node B can be accessed from Node A
Insert a key into the parent node using FAST after FAIR split.

FAST inserting makes Node B visible atomically.
Read transactions can tolerate any inconsistency caused by write transactions

↓

Read transactions can access the transient inconsistent tree node being modified by a write transaction

↓

Lock-Free Search
Lock-Free Search

[Example 1] Searching 30 while inserting (15, P6)

\[
\begin{array}{c|c|c|c|c|c|c}
P1 & P2 & P3 & P4 & P5 & \wedge & \wedge \\
\hline
10 & 20 & 30 & 40 & g & g & \\
\end{array}
\]

read →

Read transaction

Write transaction

shift →
[Example 1] Searching 30 while inserting (15, P6)

Read transaction
Write transaction

read →

shift →
Lock-Free Search

[Example 1] Searching 30 while inserting (15, P6)

```
<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P5</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>40</td>
<td>g</td>
</tr>
</tbody>
</table>
```

- Read transaction
- Write transaction

read →

shift →
[Example 1] Searching 30 while inserting (15, P6)
[Example 1] Searching 30 while inserting (15, P6)

Lock-Free Search

Read transaction
Write transaction

read →

shift →
[Example 1] Searching 30 while inserting (15, P6)

10 20 30 30 40 g

P1 P2 P3 P4 P5

Read transaction
Write transaction
[Example 1] Searching 30 while inserting (15, P6)
[Example 1] Searching 30 while inserting (15, P6)

Locked-Free Search
Lock-Free Search

[Example 1] Searching 30 while inserting (15, P6)

read $\rightarrow$

FOUND!

Write transaction

Read transaction

shift $\rightarrow$

10 20 20 30 40 g

P1 P2 P2 P3 P4 P5
[Example 2] Searching 30 while deleting (20, P2)

Read transaction
Write transaction

read →

10 20 30 40 g g
P1 P2 P3 P4 P5 ▲ ▲

← shift
[Example 2] Searching 30 while deleting (20, P2)
[Example 2] Searching 30 while deleting (20, P2)

Read transaction
Write transaction

read →

10  30  30  40  g  g
P1  P3  P3  P4  P5  ∧  ∧

← shift
[Example 2] Searching 30 while deleting (20, P2)
Lock-Free Search

[Example 2] Searching 30 while deleting (20, P2)

- Read transaction
- Write transaction

```
[Example 2] Searching 30 while deleting (20, P2)

read ->

10 30 40 40 g g
P1 P3 P4 P5

Read transaction
Write transaction
```
[Example 2] Searching 30 while deleting (20, P2)
[Example 2] Searching 30 while deleting (20, P2)

The read transaction cannot find the key 30 due to shift operation
**Direction flag:**

- **Even Number**
  - Insertion shifts to the right.
  - Search must scan from Left to Right

- **Odd Number**
  - Deletion shifts to the left.
  - Search must scan from Right to Left

---

**Lock-Free Search**

- **Insert 25**
- **Search 40**

**Diagram:**

- **Counter:** 2
- **P1:** 10
- **P2:** 20
- **P3:** 30
- **P4:** 40
- **P5:** g
- **g**

**Shift (→):**

**Read (→):**

**Notes:**

- Insertion shifts to the right for even numbers.
- Deletion shifts to the left for odd numbers.
- Search direction changes accordingly.
Direction flag:
- Even Number
  - Insertion shifts to the right.
  - Search must scan from Left to Right

Odd Number
- Deletion shifts to the left.
- Search must scan from Right to Left
**Direction flag:**
- **Even Number**
  - Insertion shifts to the right.
  - Search must scan from Left to Right

• **Odd Number**
  - Deletion shifts to the left.
  - Search must scan from Right to Left

---

The read transaction has to check the counter once again to make sure the counter has not changed. Otherwise, search the node again.
The ordering of Transaction A and Transaction B cannot be determined

Dirty reads problem
Our Lock-Free Search supports low isolation level
For higher isolation level, read lock is necessary for leaf nodes.
Experimental Environments

- Xeon Haswell-Ex E7-4809 v3 processors
  - 2.0 GHz, 16 vCPUs with hyper-threading enabled, and 20 MB L3 cache
  - Total Store Ordering (TSO) is guaranteed

- g++ 4.8.2 with -O3

- PM latency
  - Read latency
    - A DRAM-based PM latency emulator, Quartz
  - Write latency
    - Injecting delay
Sorted keys, cache locality, and memory level parallelism → up to 20X speed up
Exact Match Query Performance with varying PM Latencies

FAST+FAIR → FP-Tree → wB+-Tree → WORT → Skiplist
FAST+Logging uses logging instead of FAIR when splitting a node.

- clflush: I/O time
- Search: Tree traversal time
- Node Update: Computation time

WORT, FAST+FAIR, FP-Tree → FAST+Logging → wB+-Tree → Skiplist

• FAST+Logging uses logging instead of FAIR when splitting a node
• FAST+FAIR consistently outperforms other indexes because of its good insertion performance and superior range query performance
Lock-Free Search with concurrent threads

(a) 50M Search
(b) 50M Insertion
(c) 200M Search / 50M Insertion / 12.5M Deletion

- Lock-free search with FAST+FAIR shows high scalability and performance
- FAST+FAIR+LeafLock shows comparable scalability and provides high concurrency level
We designed a byte addressable persistent B+-Tree that
  • stores keys in order
  • avoids expensive logging

FAST and FAIR always transform B+-Trees into consistent/transient inconsistent B+-Trees

Lock-Free search
  • By tolerating transient inconsistency
Thank you

Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree

Deukyeon Hwang  Wook-Hee Kim  Youjip Won  Beomseok Nam  
UNIST  UNIST  Hanyang Univ.  UNIST
• To guarantee the order of instructions, the \textit{dmb} instruction is used for FAST+FAIR
• Although there is an overhead by \textit{dmb}, FAST+FAIR is less affected by latency