Persistent B+Trees in NonVolatile Main Memory

CS 561 2021 spring

Chen-Wei Weng

U58151415

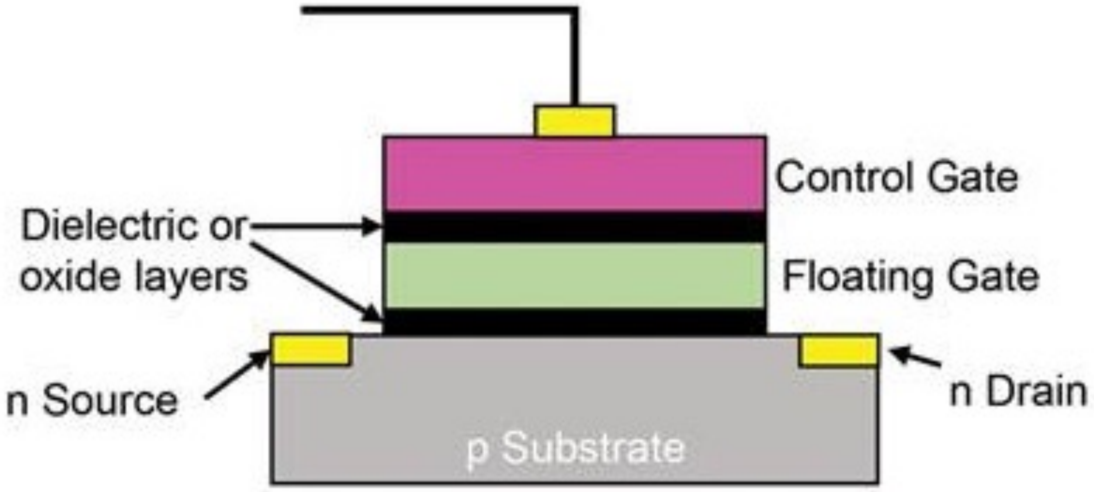
BOSTON UNIVERSITY

Outline

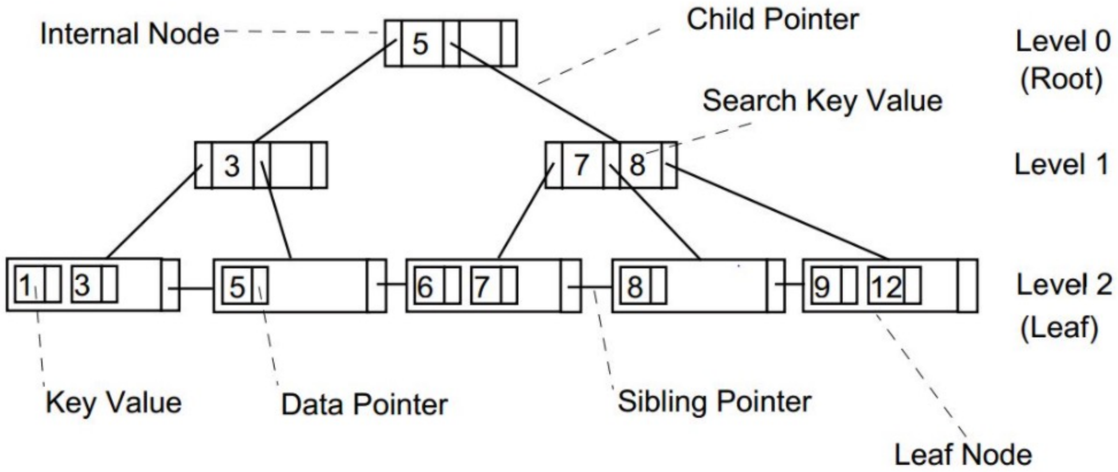
- Introduction
- Motivation
- Challenges
- Performance analysis
- wB+ tree
- Comparison
- Conclusion

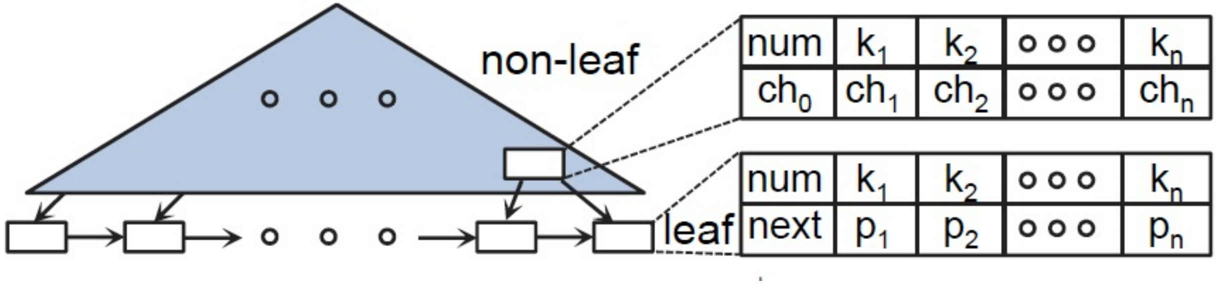
NVMM Non-Volatile Main Memory

- Data can be retained after power off
- Data is trapped in the floating gate



B+ tree





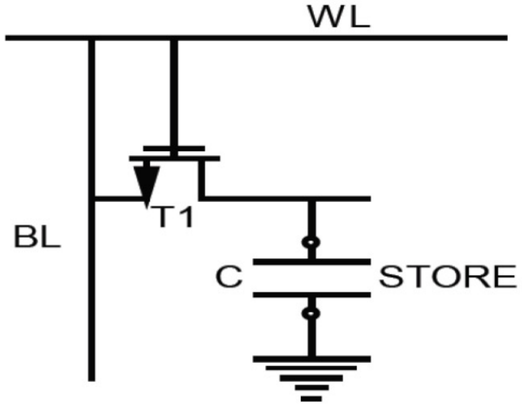
num	k_1	k_2	o o o	k_n
ch_0	ch_1	ch_2	o o o	ch_n

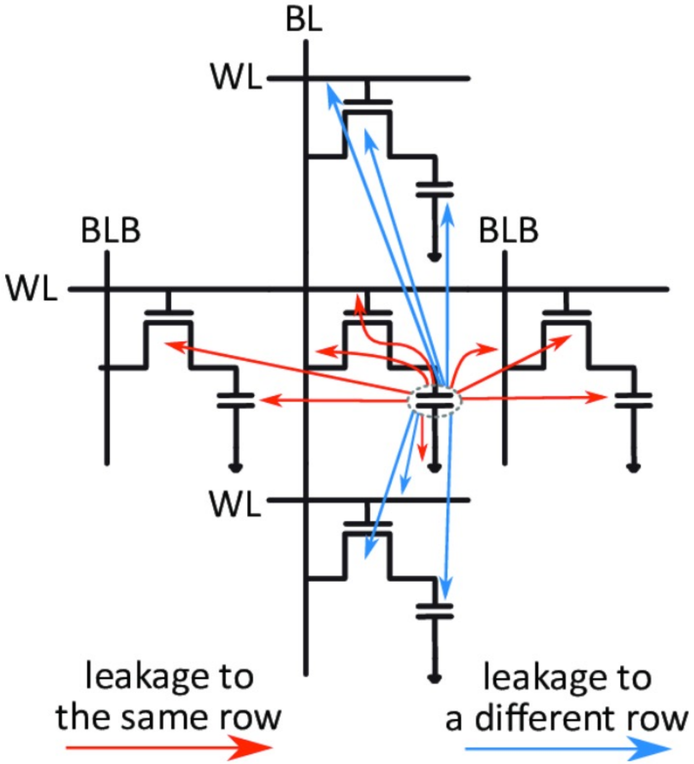
num	k_1	k_2	o o o	k_n
next	p_1	p_2	o o o	p_n

BOSTON
UNIVERSITY

Motivation

- Limitations of DRAM technology
- Increasing capacity of main memory





Why NVMM?

- physical mechanisms are amenable to much smaller feature sizes
- support byte-addressable reads and writes with performance close to that of DRAM
- lower power than DRAM due to non-volatility

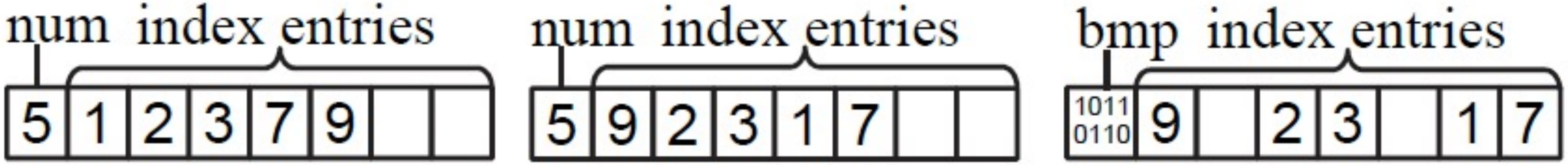
Challenges

- B+ tree in NVMM
- Data structure inconsistency

Solutions

- PCM-friendly B+ tree
- clflush & mfence
- Logging & shadowing

PCM-friendly B+ tree

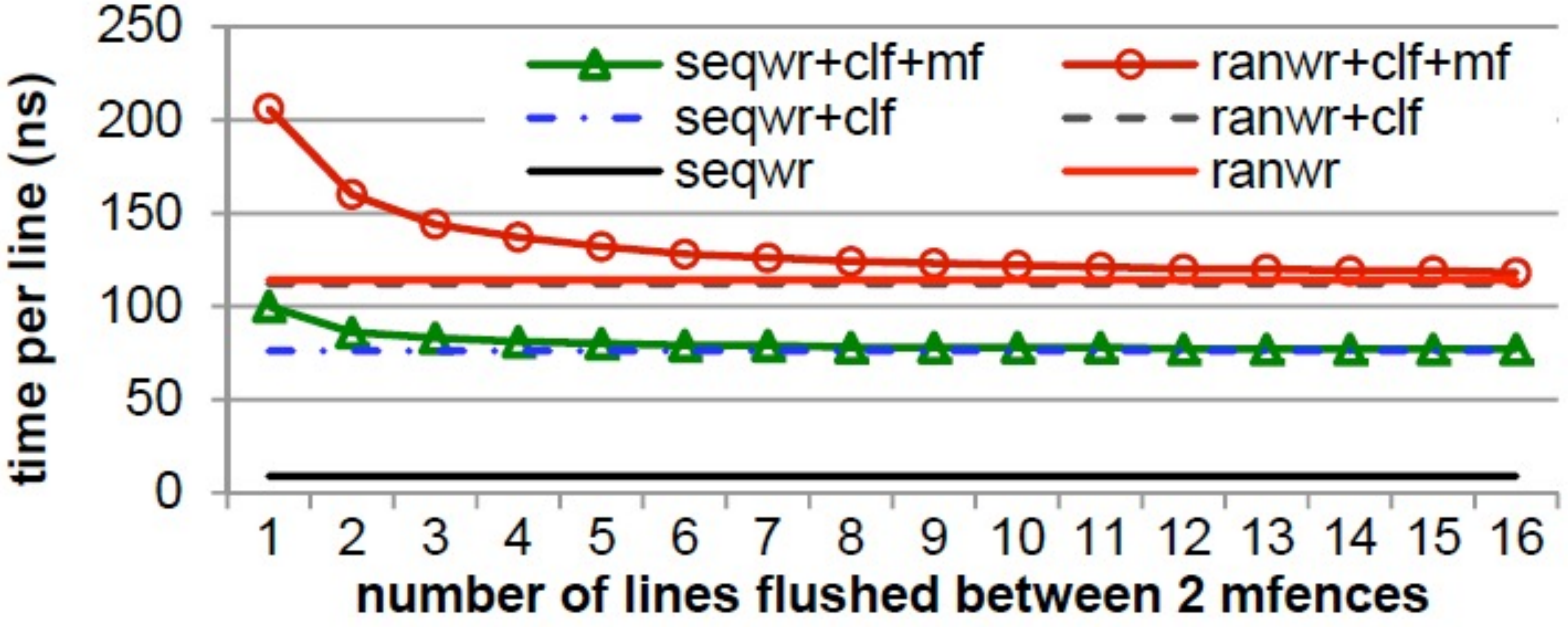


(a) Sorted entries (b) Unsorted leaf (c) Unsorted leaf w/ bitmap

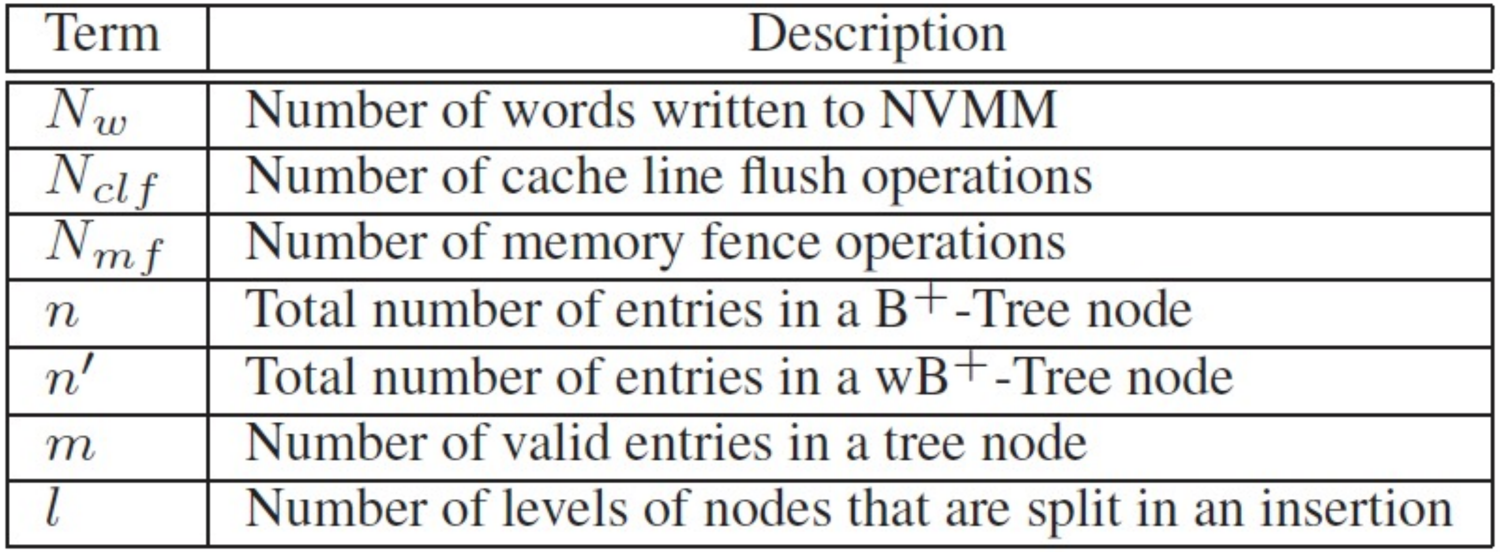
Clflush & Mfence

- X86 processor operations to control cache lines
- clflush
  - clflush invalidates the cache line that contains the address from all levels of caches, and broadcasts the invalidation to all CPU cores in the system
- mfence
  - mfence guarantees that all memory reads and memory writes issued before the mfence in program order

Clflush & Mfence



Performance Analysis Metrices



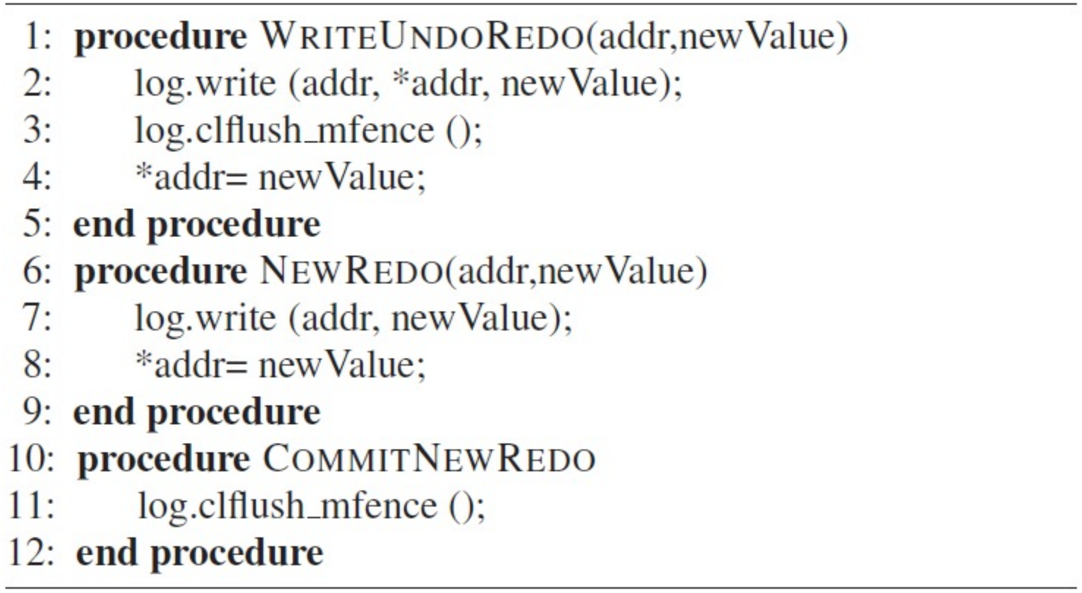
Term	Description
N_w	Number of words written to NVMM
N_{clf}	Number of cache line flush operations
N_{mf}	Number of memory fence operations
n	Total number of entries in a B$^+$-Tree node
n'	Total number of entries in a wB$^+$-Tree node
m	Number of valid entries in a tree node
l	Number of levels of nodes that are split in an insertion

Undo Redo Logging

- Record REDO and UNDO information for every update in a log
  - Sequential write to a log (put it on a separate disk)
  - Minimal information written to log, multiple updates fit in a single log page

- Log : An ordered list of REDO/UNDO actions
  - Log record contains <XID, pageID, offset, length, old data, new data>
  - Additional control information

Undo Redo Logging



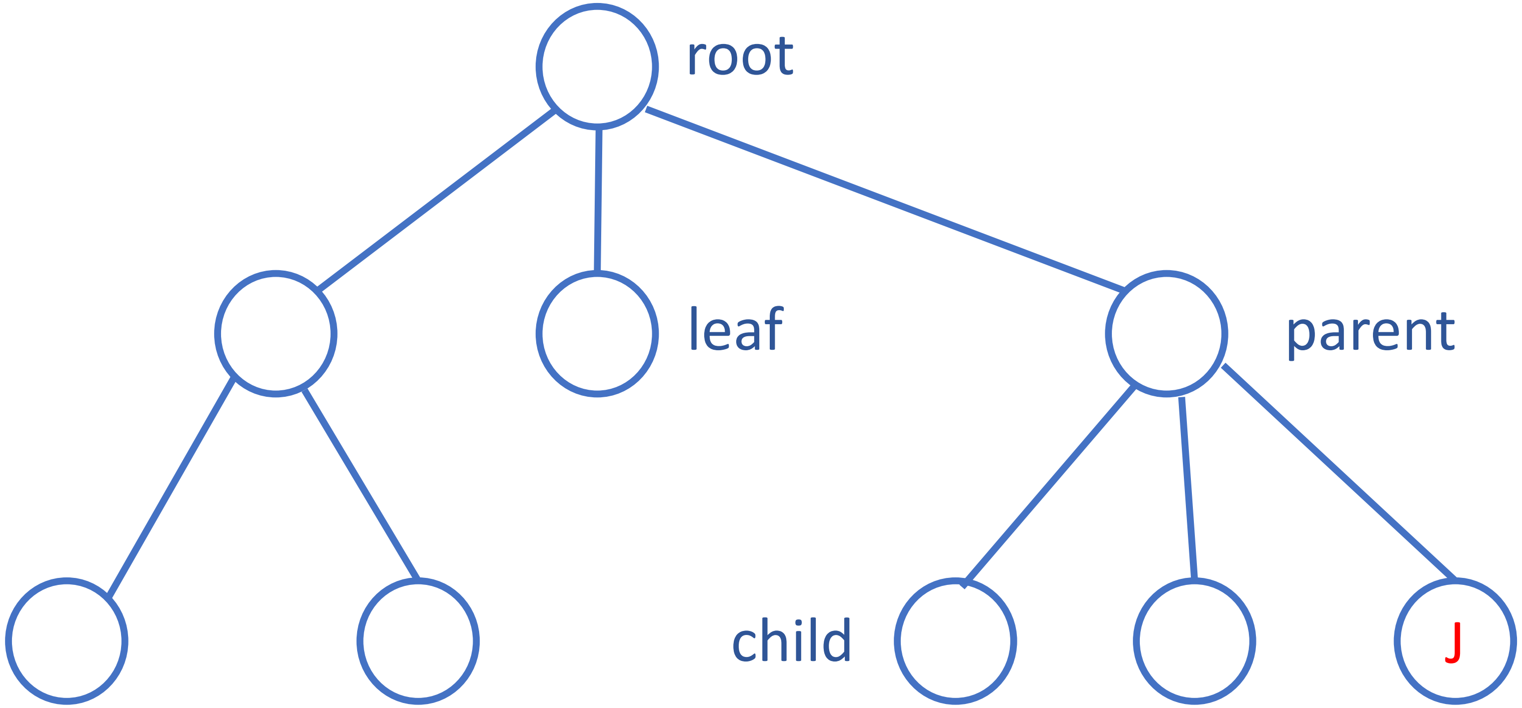
```
procedure WriteUndoRedo(addr,newValue)
    log.write (addr, *addr, newValue);
    log.clflush_mfence ();
    *addr= newValue;
end procedure
procedure NewRedo(addr,newValue)
    log.write (addr, newValue);
    *addr= newValue;
end procedure
procedure CommitNewRedo
    log.clflush_mfence ();
end procedure
```

$Nw = 4m + 12$

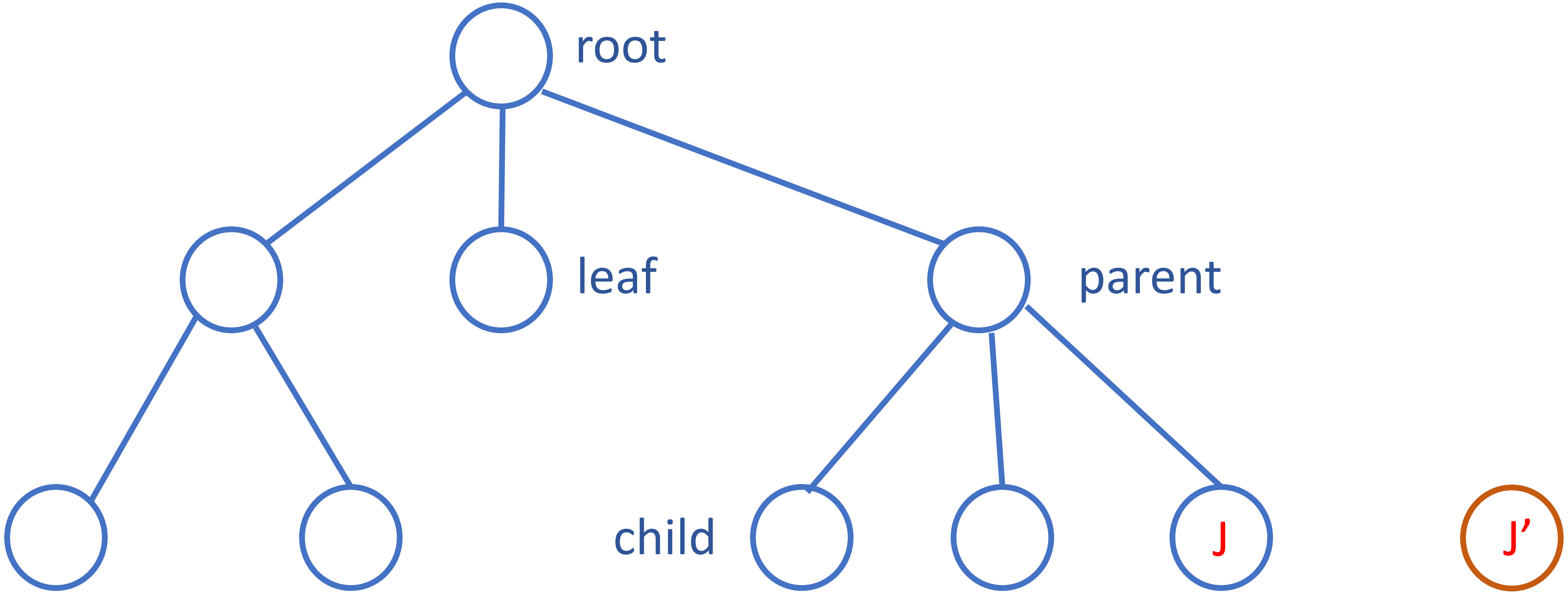
$Nclf = m + 3$

$Nmf = m + 3$

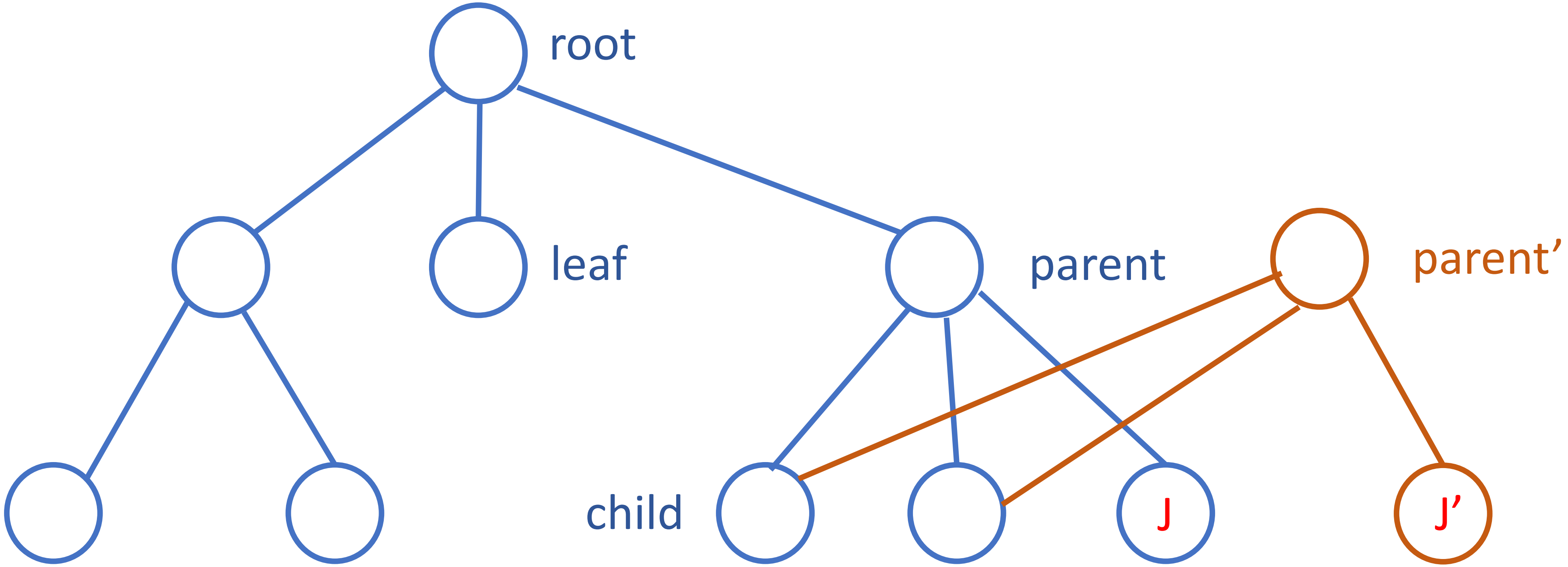
Shadowing



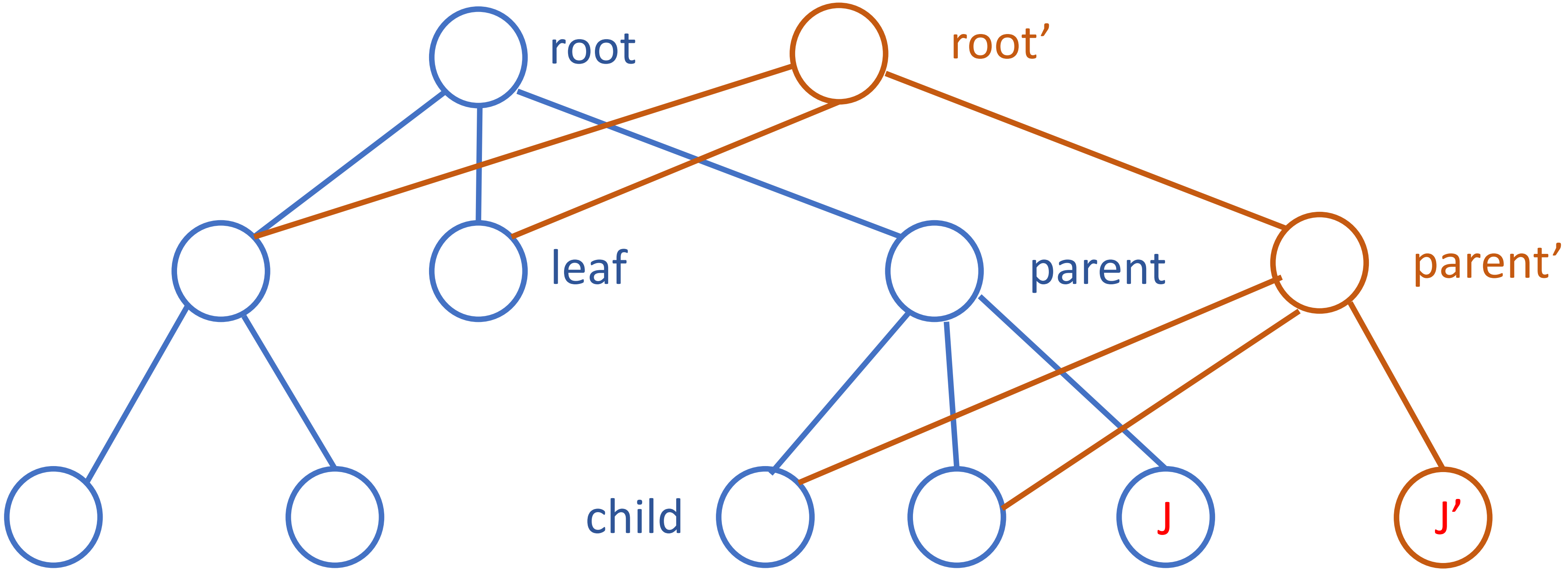
Shadowing



Shadowing



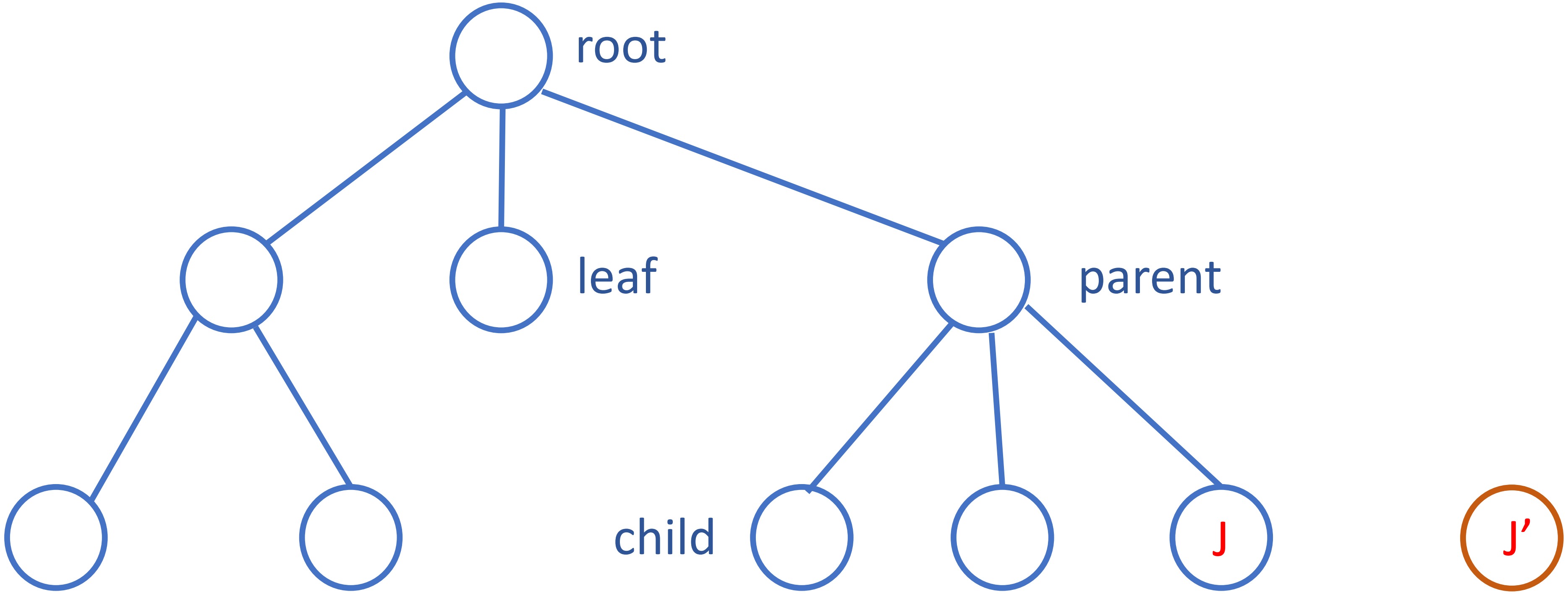
Shadowing



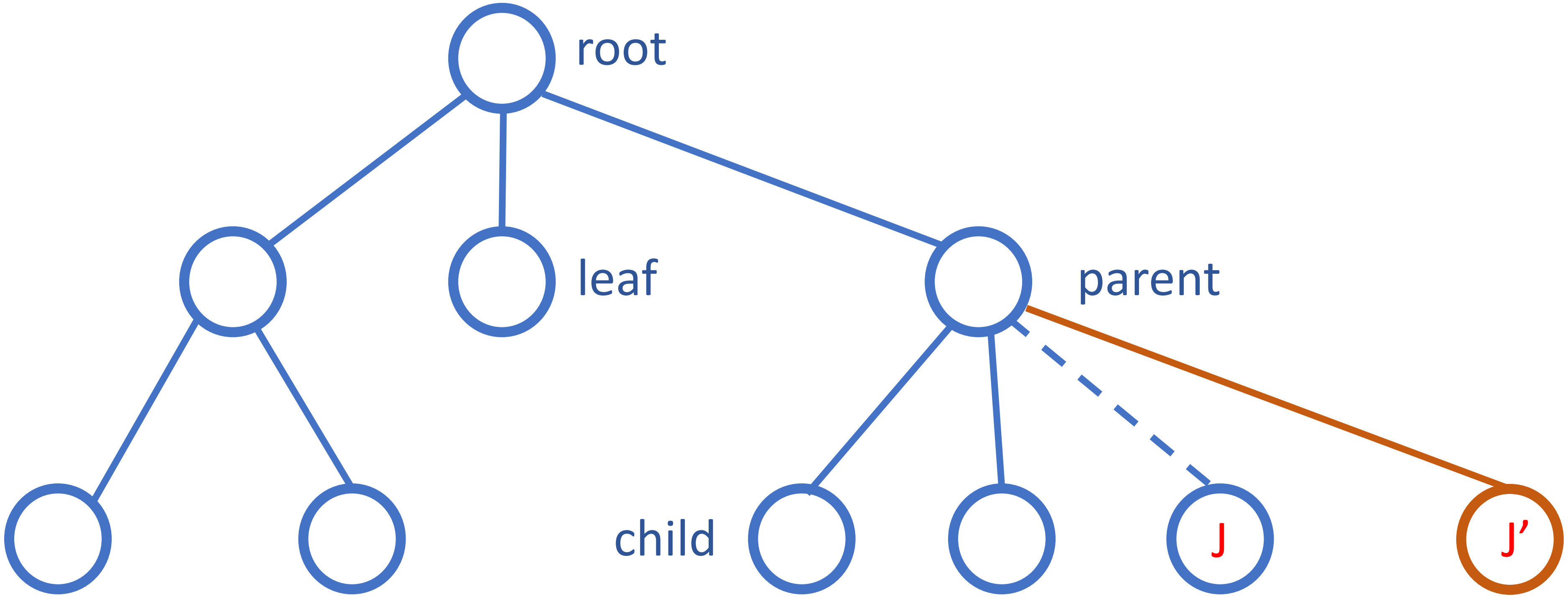
Shadowing

- Short-Circuit Shadowing
- NVM supports 8-byte atomic write
- Proposed by Condit et al.

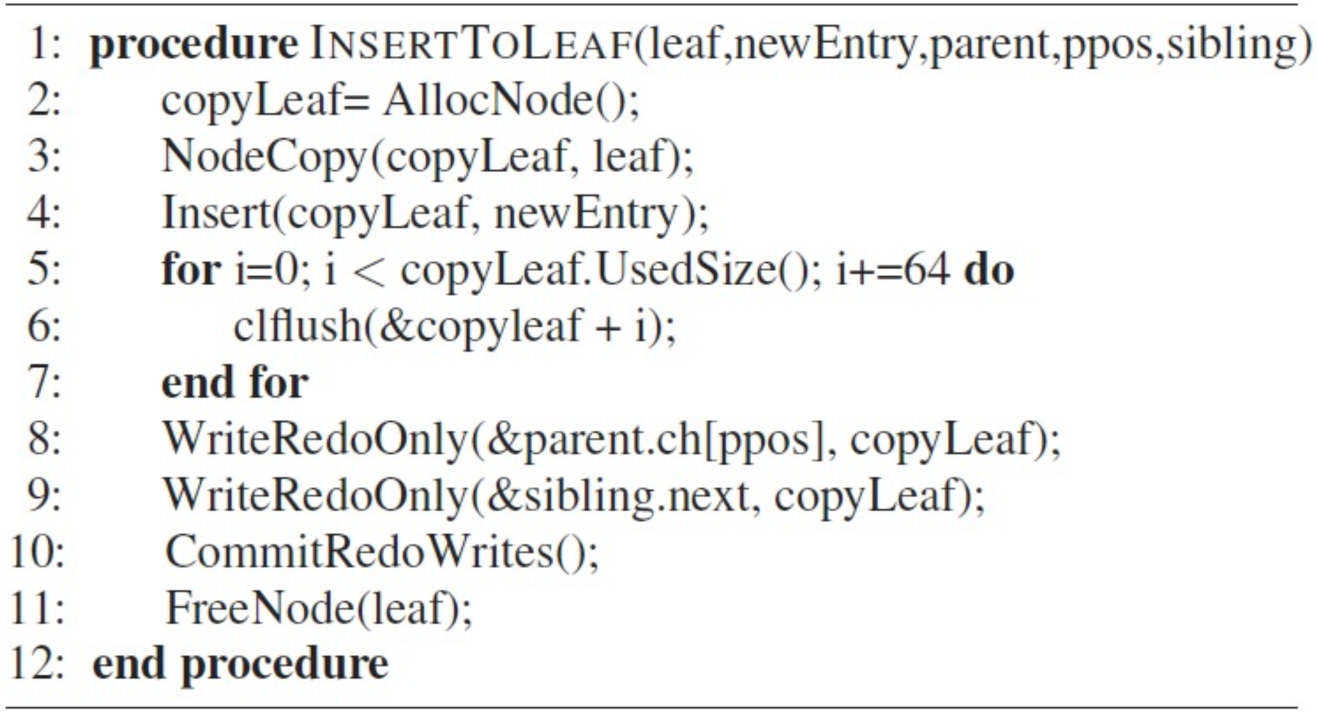
Short-Circuit Shadowing



Short-Circuit Shadowing



Shadowing



```
procedure InsertToLeaf(leaf,newEntry,parent,ppos,sibling)
    copyLeaf= AllocNode();
    NodeCopy(copyLeaf, leaf);
    Insert(copyLeaf, newEntry);
    for i=0; i < copyLeaf.UsedSize(); i+=64 do
        clflush(&copyleaf + i);
    end for
    WriteRedoOnly(&parent.ch[ppos], copyLeaf);
    WriteRedoOnly(&sibling.next, copyLeaf);
    CommitRedoWrites();
    FreeNode(leaf);
end procedure
```

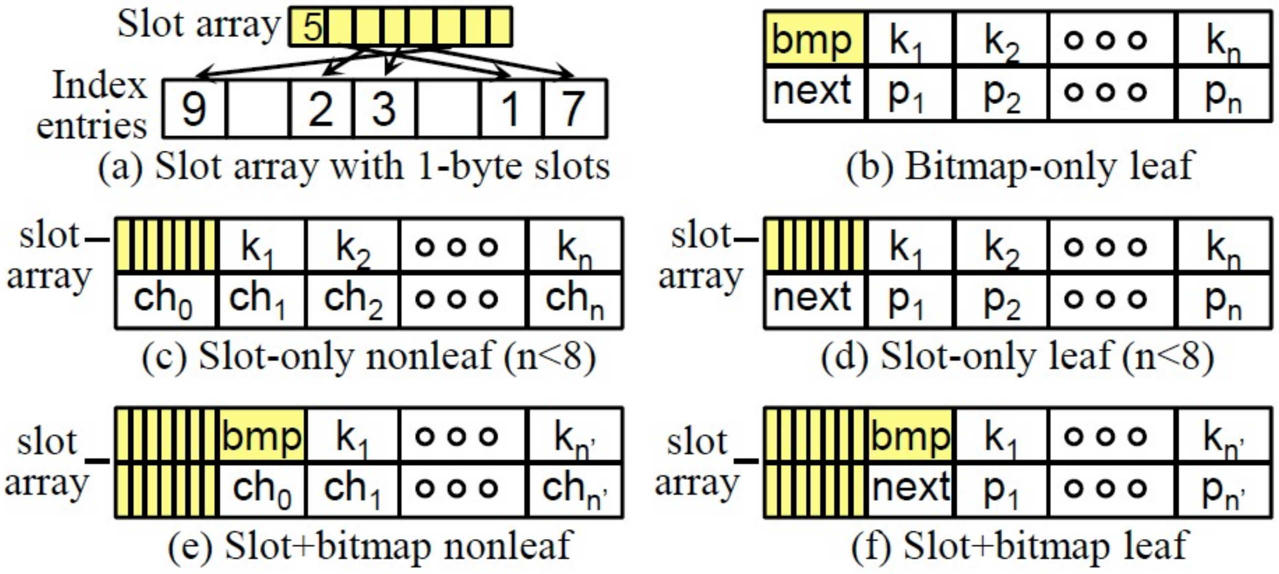
Nw = 2m + 11

Nclf = 0.25m + 2.5

Nmf = 2

Write-Atomic B+ tree

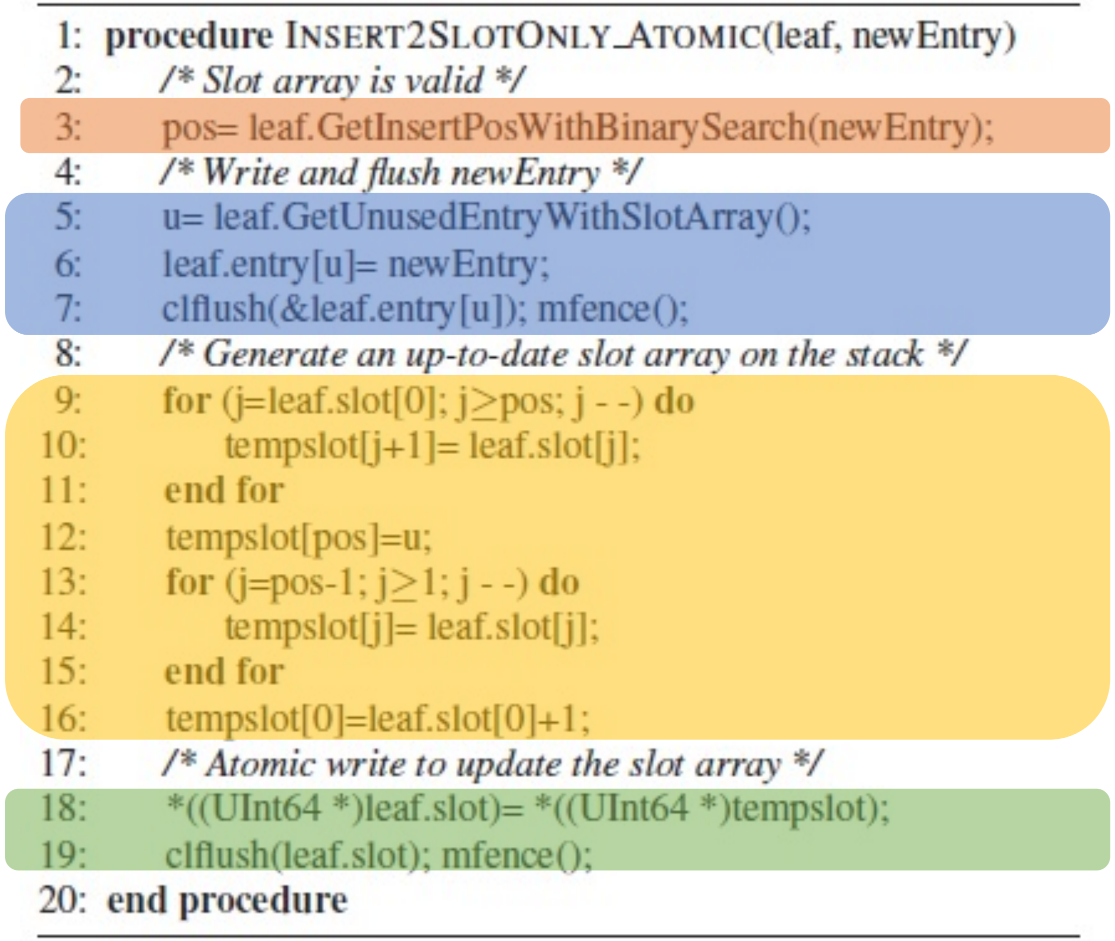
- Atomic write to commit all changes
- Minimize the movement of index entries
- Slot array + bitmap



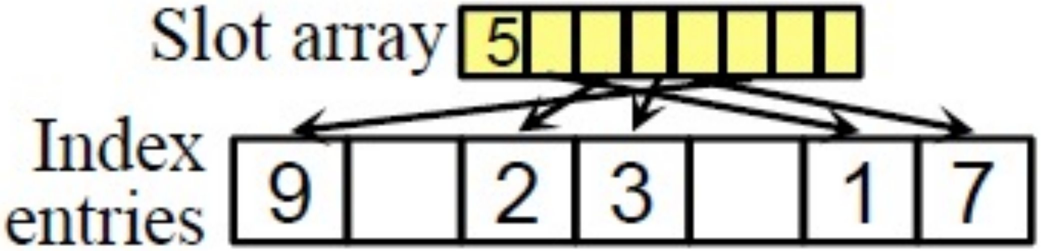
Write atomic B+ tree

- Insertion
- Deletion
- Search

Write atomic B+ tree Insertion

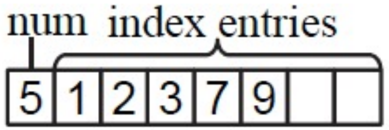


```
procedure INSERT2SLOTONLY_ATOMIC(leaf, newEntry)
    /* Slot array is valid */
    pos= leaf.GetInsertPosWithBinarySearch(newEntry);
    /* Write and flush newEntry */
    u= leaf.GetUnusedEntryWithSlotArray();
    leaf.entry[u]= newEntry;
    clflush(&leaf.entry[u]); mfence();
    /* Generate an up-to-date slot array on the stack */
    for (j=leaf.slot[0]; j≥pos; j - -) do
        tempslot[j+1]= leaf.slot[j];
    end for
    tempslot[pos]=u;
    for (j=pos-1; j≥1; j - -) do
        tempslot[j]= leaf.slot[j];
    end for
    tempslot[0]=leaf.slot[0]+1;
    /* Atomic write to update the slot array */
    *((UInt64 *)leaf.slot)= *((UInt64 *)tempslot);
    clflush(leaf.slot); mfence();
end procedure
```



Comparison Insertion

Logging



Nw = 4m + 12

Nclf = m + 3

Nmf = m + 3

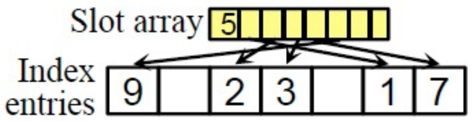
Shadowing

Nw = 2m + 11

Nclf = 0.25m + 2.5

Nmf = 2

wB+-Tree

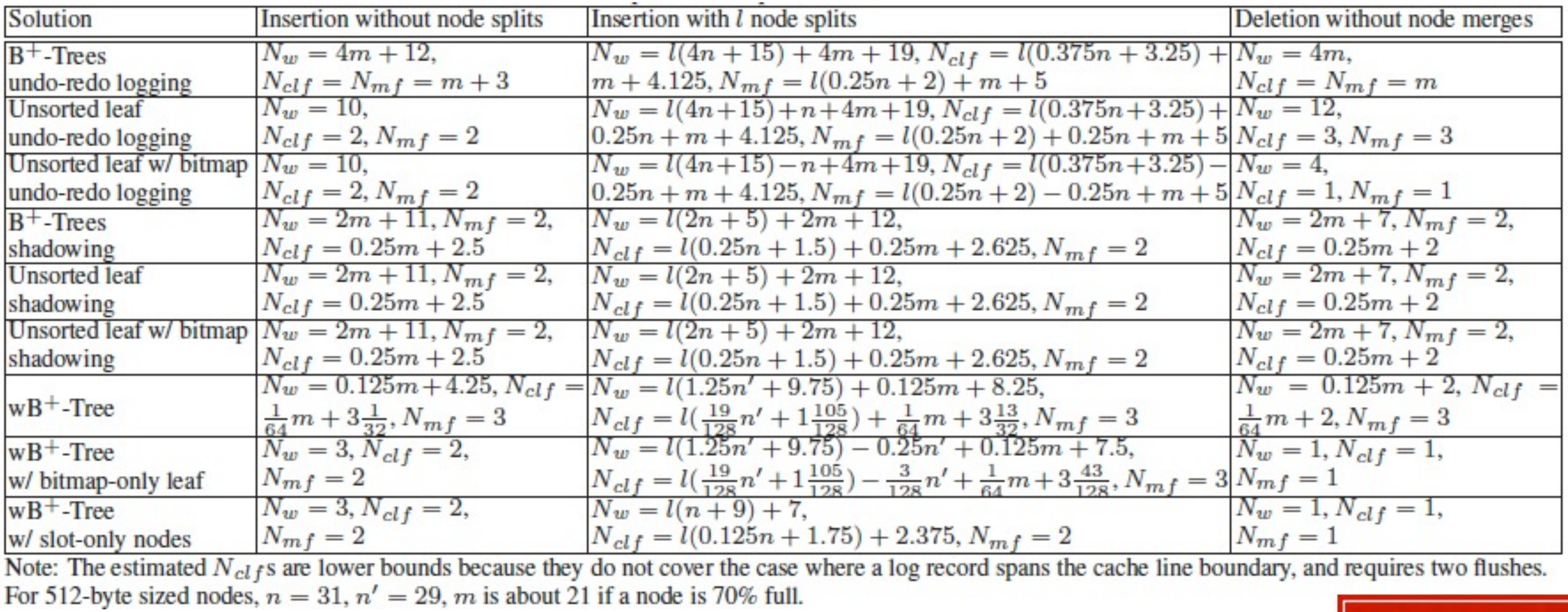


Nw = 3

Nclf = 2

Nmf = 2

Comparison

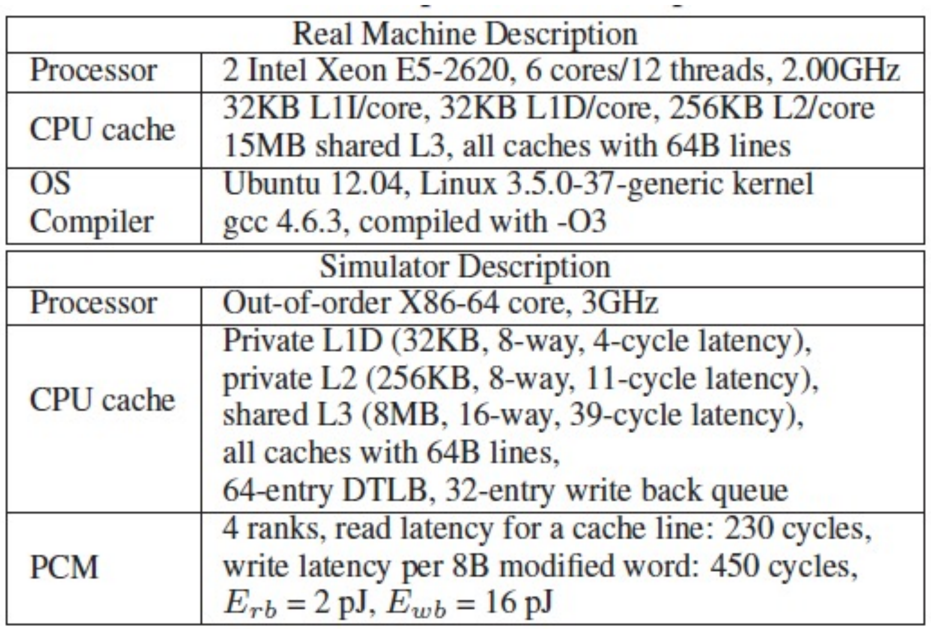


Solution	Insertion without node splits	Insertion with l node splits	Deletion without node merges
B$^+$-Trees undo-redo logging	$N_w = 4m + 12$, $N_{clf} = N_{mf} = m + 3$	$N_w = l(4n + 15) + 4m + 19$, $N_{clf} = l(0.375n + 3.25) + m + 4.125$, $N_{mf} = l(0.25n + 2) + m + 5$	$N_w = 4m$, $N_{clf} = N_{mf} = m$
Unsorted leaf undo-redo logging	$N_w = 10$, $N_{clf} = 2$, $N_{mf} = 2$	$N_w = l(4n+15)+n+4m+19$, $N_{clf} = l(0.375n+3.25)+0.25n + m + 4.125$, $N_{mf} = l(0.25n + 2) + 0.25n + m + 5$	$N_w = 12$, $N_{clf} = 3$, $N_{mf} = 3$
Unsorted leaf w/ bitmap undo-redo logging	$N_w = 10$, $N_{clf} = 2$, $N_{mf} = 2$	$N_w = l(4n+15)-n+4m+19$, $N_{clf} = l(0.375n+3.25)-0.25n + m + 4.125$, $N_{mf} = l(0.25n + 2) - 0.25n + m + 5$	$N_w = 4$, $N_{clf} = 1$, $N_{mf} = 1$
B$^+$-Trees shadowing	$N_w = 2m + 11$, $N_{mf} = 2$, $N_{clf} = 0.25m + 2.5$	$N_w = l(2n + 5) + 2m + 12$, $N_{clf} = l(0.25n + 1.5) + 0.25m + 2.625$, $N_{mf} = 2$	$N_w = 2m + 7$, $N_{mf} = 2$, $N_{clf} = 0.25m + 2$
Unsorted leaf shadowing	$N_w = 2m + 11$, $N_{mf} = 2$, $N_{clf} = 0.25m + 2.5$	$N_w = l(2n + 5) + 2m + 12$, $N_{clf} = l(0.25n + 1.5) + 0.25m + 2.625$, $N_{mf} = 2$	$N_w = 2m + 7$, $N_{mf} = 2$, $N_{clf} = 0.25m + 2$
Unsorted leaf w/ bitmap shadowing	$N_w = 2m + 11$, $N_{mf} = 2$, $N_{clf} = 0.25m + 2.5$	$N_w = l(2n + 5) + 2m + 12$, $N_{clf} = l(0.25n + 1.5) + 0.25m + 2.625$, $N_{mf} = 2$	$N_w = 2m + 7$, $N_{mf} = 2$, $N_{clf} = 0.25m + 2$
wB$^+$-Tree	$N_w = 0.125m+4.25$, $N_{clf} = \frac{1}{64}m + 3\frac{1}{32}$, $N_{mf} = 3$	$N_w = l(1.25n' + 9.75) + 0.125m + 8.25$, $N_{clf} = l(\frac{19}{128}n' + 1\frac{105}{128}) + \frac{1}{64}m + 3\frac{13}{32}$, $N_{mf} = 3$	$N_w = 0.125m + 2$, $N_{clf} = \frac{1}{64}m + 2$, $N_{mf} = 3$
wB$^+$-Tree w/ bitmap-only leaf	$N_w = 3$, $N_{clf} = 2$, $N_{mf} = 2$	$N_w = l(1.25n' + 9.75) - 0.25n' + 0.125m + 7.5$, $N_{clf} = l(\frac{19}{128}n' + 1\frac{105}{128}) - \frac{3}{128}n' + \frac{1}{64}m + 3\frac{43}{128}$, $N_{mf} = 3$	$N_w = 1$, $N_{clf} = 1$, $N_{mf} = 1$
wB$^+$-Tree w/ slot-only nodes	$N_w = 3$, $N_{clf} = 2$, $N_{mf} = 2$	$N_w = l(n + 9) + 7$, $N_{clf} = l(0.125n + 1.75) + 2.375$, $N_{mf} = 2$	$N_w = 1$, $N_{clf} = 1$, $N_{mf} = 1$

Note: The estimated N_{clf}s are lower bounds because they do not cover the case where a log record spans the cache line boundary, and requires two flushes. For 512-byte sized nodes, $n = 31$, $n' = 29$, m is about 21 if a node is 70% full.

Experiment

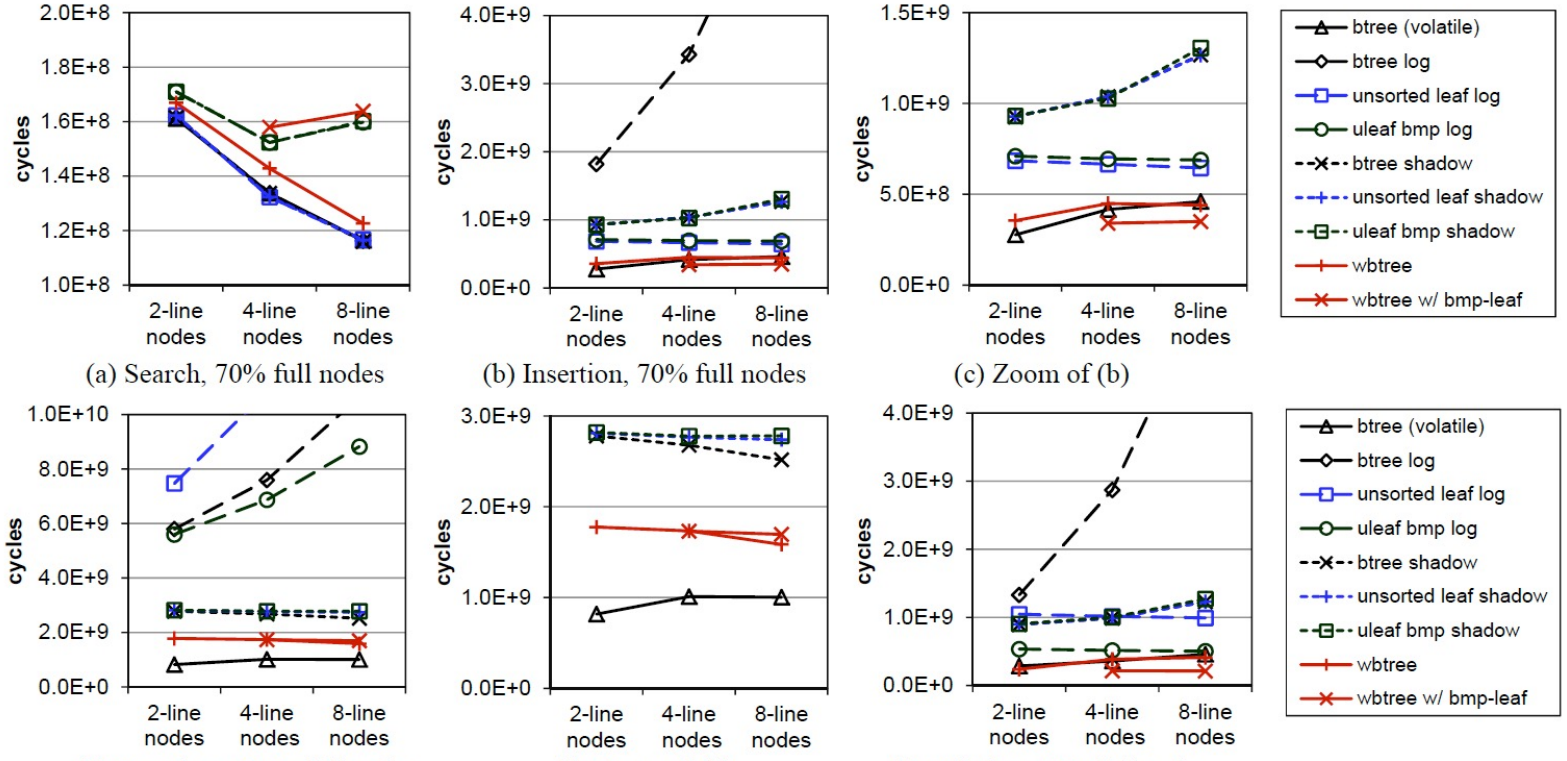
- Setup
  - Real machine modeling DRAM-like fast NVMM
  - Simulation modeling PCM-based NVMM



Real Machine Description	
Processor	2 Intel Xeon E5-2620, 6 cores/12 threads, 2.00GHz
CPU cache	32KB L1I/core, 32KB L1D/core, 256KB L2/core 15MB shared L3, all caches with 64B lines
OS Compiler	Ubuntu 12.04, Linux 3.5.0-37-generic kernel gcc 4.6.3, compiled with -O3

Simulator Description	
Processor	Out-of-order X86-64 core, 3GHz
CPU cache	Private L1D (32KB, 8-way, 4-cycle latency), private L2 (256KB, 8-way, 11-cycle latency), shared L3 (8MB, 16-way, 39-cycle latency), all caches with 64B lines, 64-entry DTLB, 32-entry write back queue
PCM	4 ranks, read latency for a cache line: 230 cycles, write latency per 8B modified word: 450 cycles, E_{rb} = 2 pJ, E_{wb} = 16 pJ

Experiment Simulation



(a) Search, 70% full nodes

(b) Insertion, 70% full nodes

(c) Zoom of (b)

(d) Insertion, 100% full nodes

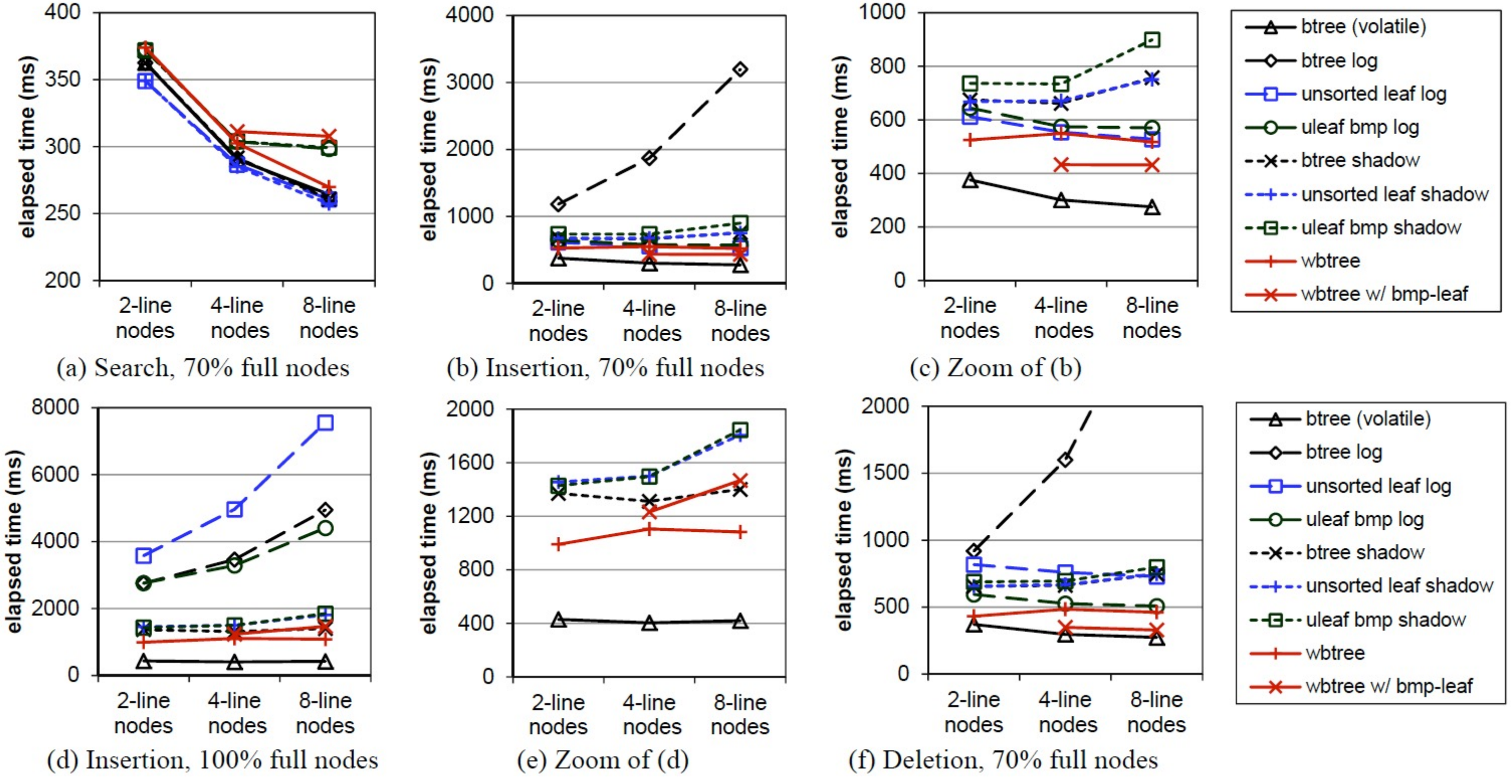
(e) Zoom of (d)

(f) Deletion, 70% full nodes

- Undo-Redo logging incurs drastic 6.6–13.7x slowdowns for B+-Trees and 2.7–12.6x slowdowns for PCM-friendly B+-Trees.
- Shadowing incurs 2.1–7.8x slowdowns
- wB+-Trees achieve a factor of 4.2–27.1x improvement over the slowest previous persistent solution
- The best wB+-Tree result is 1.5–2.4x better than the fastest previous persistent solution
- wB+-Tree w/ bmp-leaf achieves slightly better insertion and deletion performance than wB+-Tree, but sees worse search performance.

BOSTON UNIVERSITY

Experiment Real Machine



(a) Search, 70% full nodes

(b) Insertion, 70% full nodes

(c) Zoom of (b)

(d) Insertion, 100% full nodes

(e) Zoom of (d)

(f) Deletion, 70% full nodes

- wB+-Tree achieves similar search performance compared to the baseline main-memory non-persistent B+-Trees
- undo-redo logging incurs 1.6–11.8x slowdowns
- Shadowing incurs 1.7–3.3x slowdowns
- The wB+-Trees achieve 2.1–8.8x improvement over the slowest previous persistent solution,and the best wB+-Tree result is 1.2–1.6x better than the best previous persistent solution in each insertion or deletion

BOSTON UNIVERSITY

Conclusion

- Traditional approaches(logging, shadowing) incur drastic writes and cache line flush
- NVM write plays a major role in PCM based NVMM
- Cache line flush is the major part for DRAM-like NVMM
- Write atomic B-trees has better insertion and deletion performance, while achieving good search performance

Reference

- Persistent B+-Trees in Non-Volatile Main Memory-Shimen Chen et al