FAST '17
FEB 27-MAR 2, 2017
SANTA CLARA, CA
www.usenix.org/fast17

WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems

Sam H. Noh/노삼혁

UNIST

Se Kwon Lee, K. Hyun Lim[1], Hyunsub Song, Beomseok Nam

[1]Hongik University

UNIST ECE SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING CISSR Center for Innovative System Software Research NECSST Next-generation Embedded / Computer System Software Technology

Persistent Memory (PM)

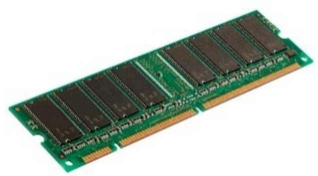
- **Persistent memory is expected to replace both DRAM & NAND**

	NAND	STT-MRAM	PCM	DRAM
Non-volatility	o	o	o	x
Read (ns)	2.5×10^4	5 - 30	20 – 70	10
Write (ns)	2×10^5	10 - 100	150 - 220	10
Byte-addressable	x	o	o	o
Density	185.8 Gbit/cm^2	0.36 Gbit/cm^2	13.5 Gbit/cm^2	9.1 Gbit/cm^2

K. Suzuki and S. Swanson. “A Survey of Trends in Non-Volatile Memory Technologies: 2000-2014”, IMW 2015

Non-volatile

+

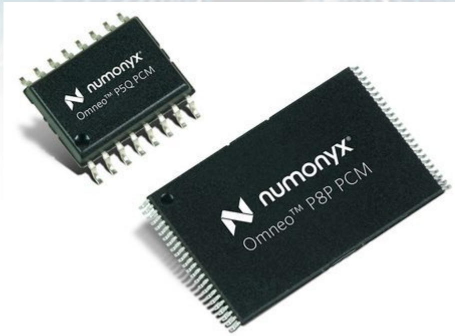


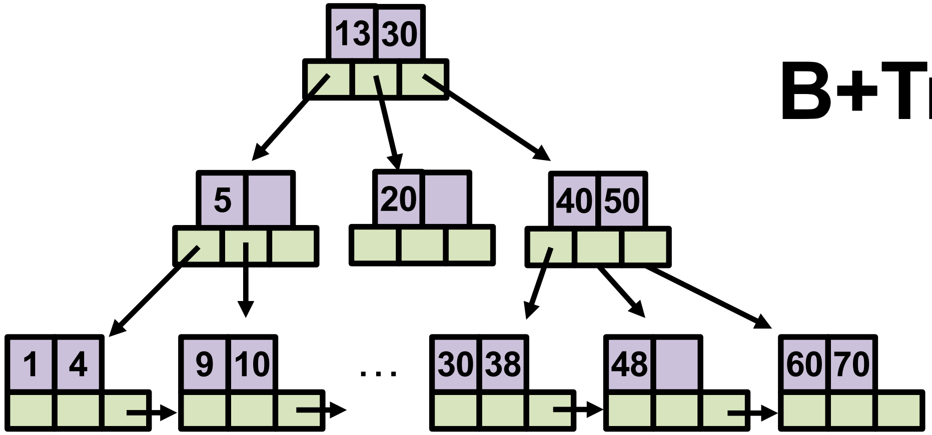
High performance

=

Persistent Memory

Indexing Structure for PM Storage Systems

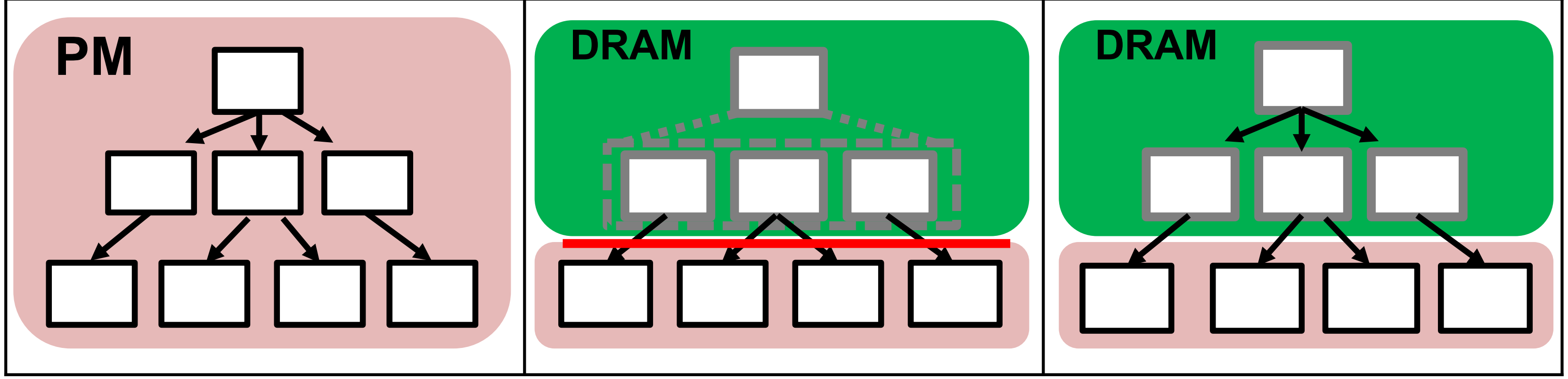




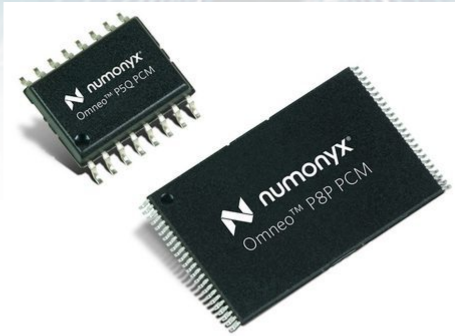
B+Tree

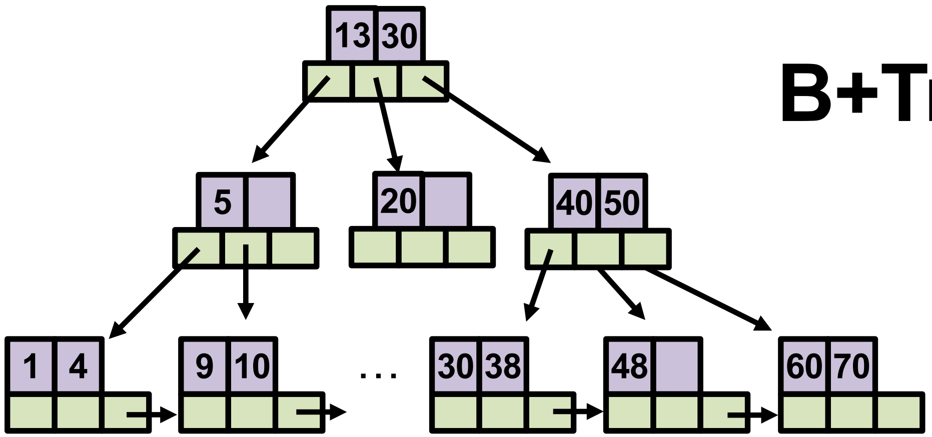
B+tree Variants for Persistent Memory

B+tree variants		
wB+Tree (VLDB' 15)	NVTree (FAST' 15)	FPTree (SIGMOD' 16)



B+Tree for PM Storage Systems?





B+Tree

Consistency Issue of B+tree in PM

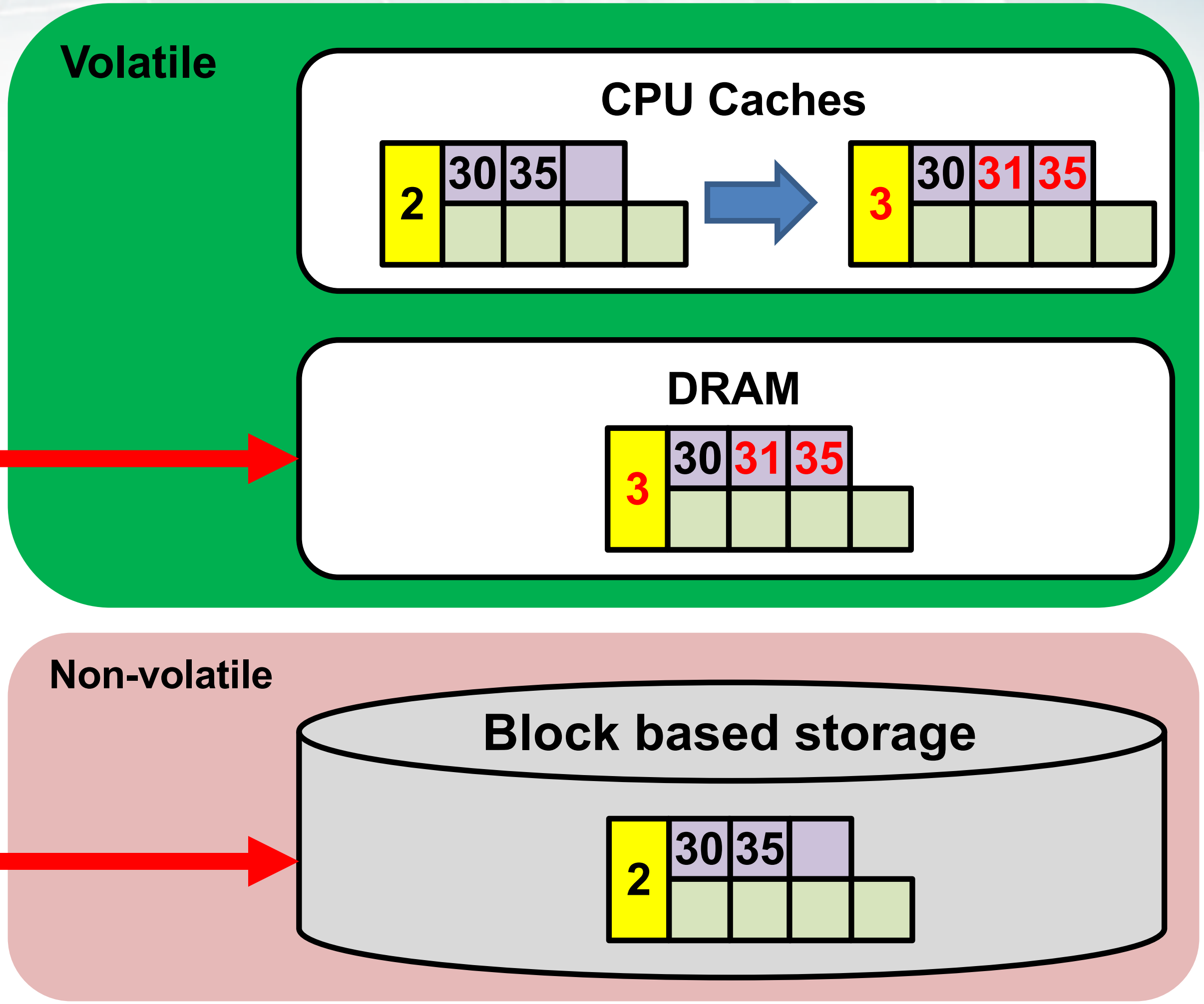
- **B+tree developed for block-based storage**
  - Consistency and atomicity guaranteed in block units
  - Controlled by software (file system)
- **Persistent memory is byte-based storage**
  - Consistency and atomicity guaranteed in 8-byte units
  - Controlled by hardware (CPU cache policy)
- **With B+Tree on PM, discrepancy can lead to consistency problems**

Consistency Issue of B+tree in PM

- **Traditional case**

Write reordering

Not persistent data



Block granularity update

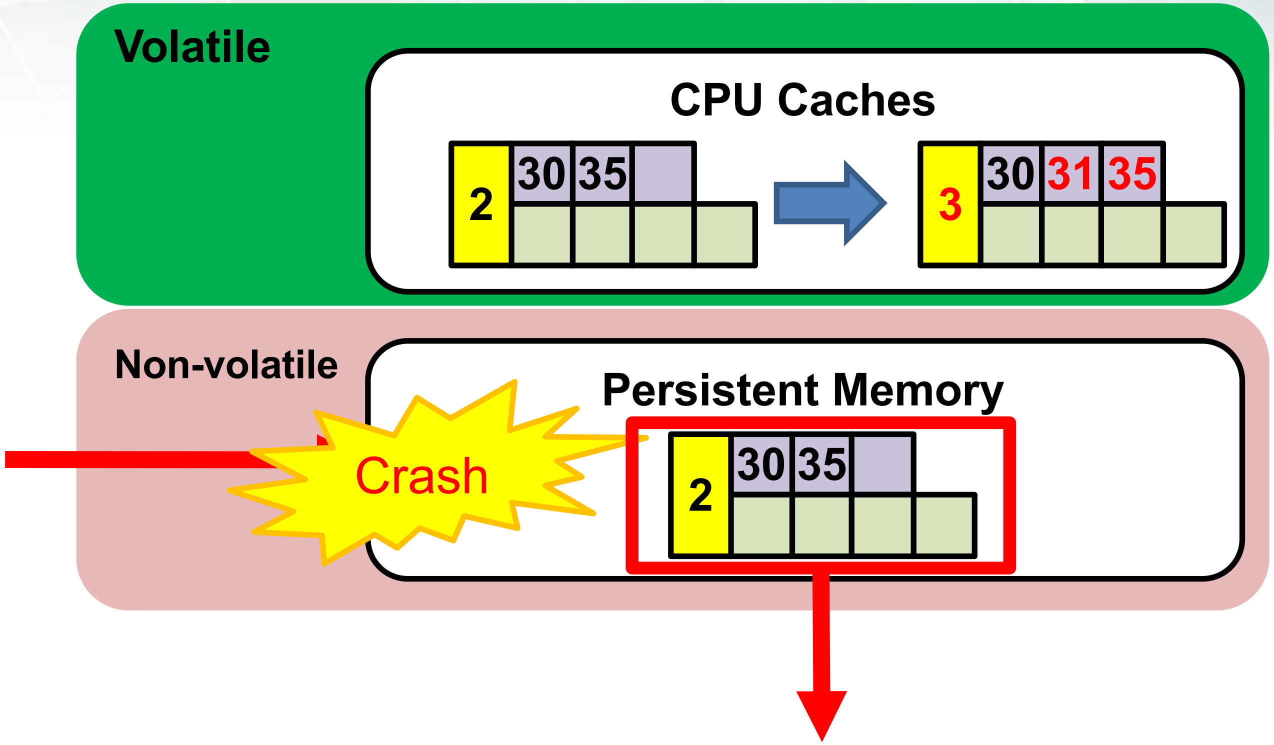
Consistency Issue of B+tree in PM

- **PM case**

Byte granularity update

Write reordering

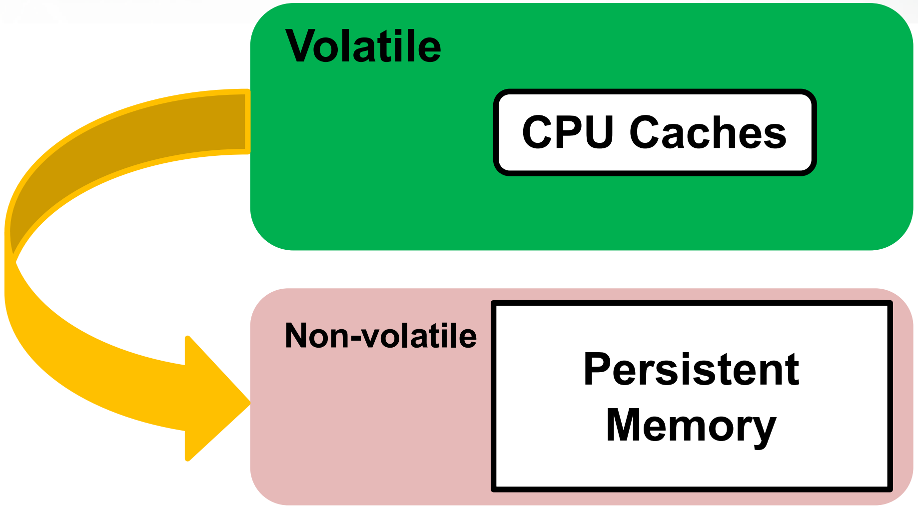
Persistent data



Garbage data persistently stored

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



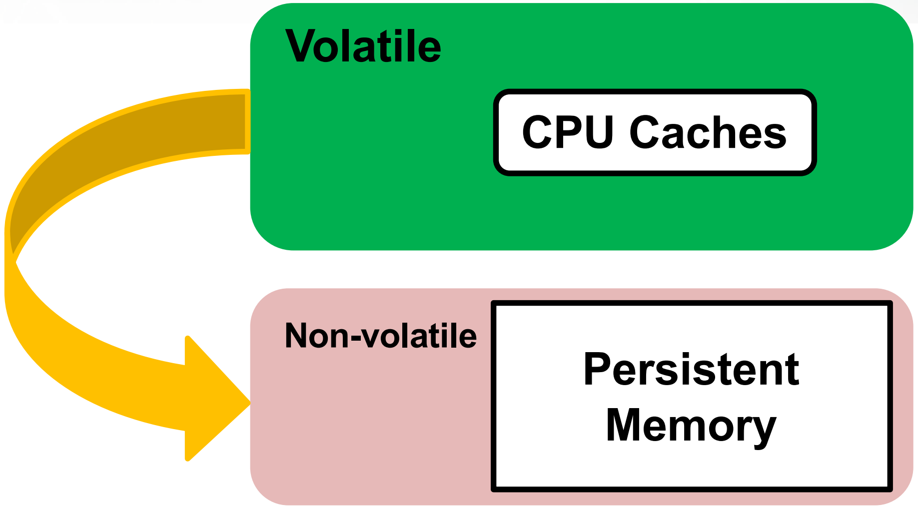
Primitives for Data Consistency in PM

- D
  -
- C
  -
  instructions

Serialization of *CLFLUSH* and *MFENCE* is known to cause **large overhead**

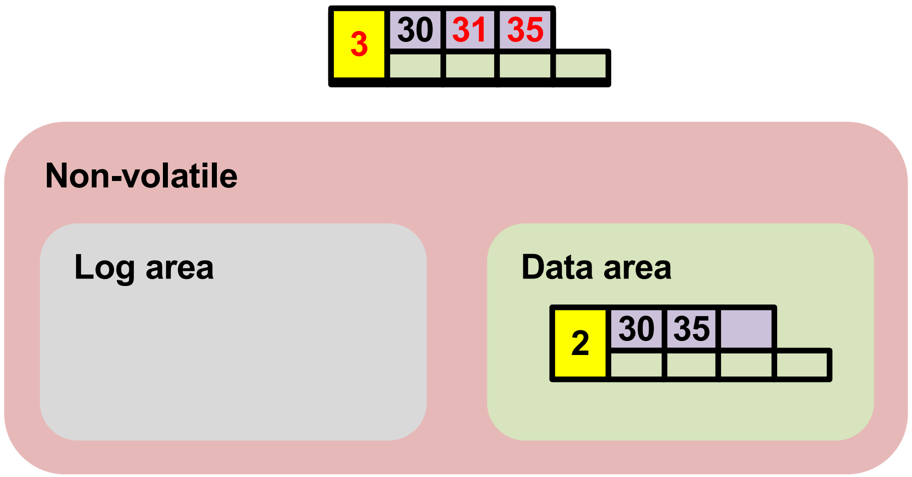
Primitives for Data Consistency in PM

- **Durability**
  - ***CLFLUSH*** (Flush cache line)
    - Can be reordered
- **Ordering**
  - ***MFENCE*** (Load and Store fence)
    - Order CPU cache line flush instructions
- **Atomicity**
  - ***CLFLUSH*** (Flush cache line)
    - 8-byte failure atomicity



Primitives for Data Consistency in PM

- **Atomicity**
  - larger than 8-byte writes?
    - Logging or CoW based atomicity
    - Requires duplicate copies



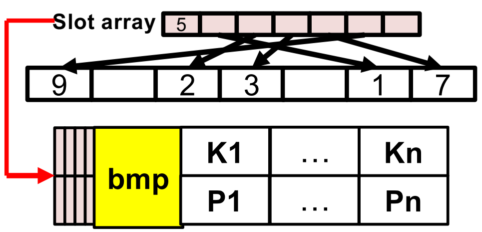
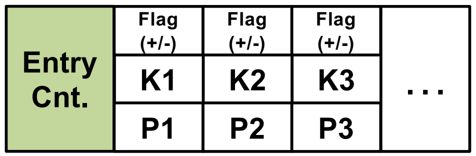
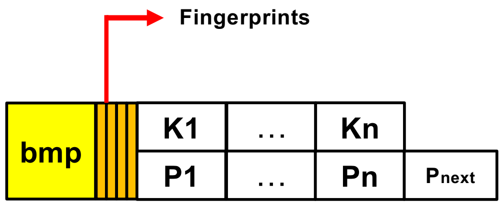
Primitives for Data Consistency in PM

Logging increases cache line flush overhead

B+tree Variants for Persistent Memory

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

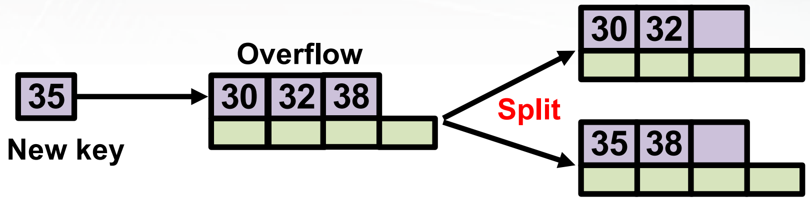
Unsorted keys → Append-only with metadata
Failure-atomic update of metadata

wB+Tree (VLDB' 15)	NVTree (FAST' 15)	FPTree (SIGMOD' 16)
		

Unsorted key → Decreases search performance

B+tree Variants for Persistent Memory

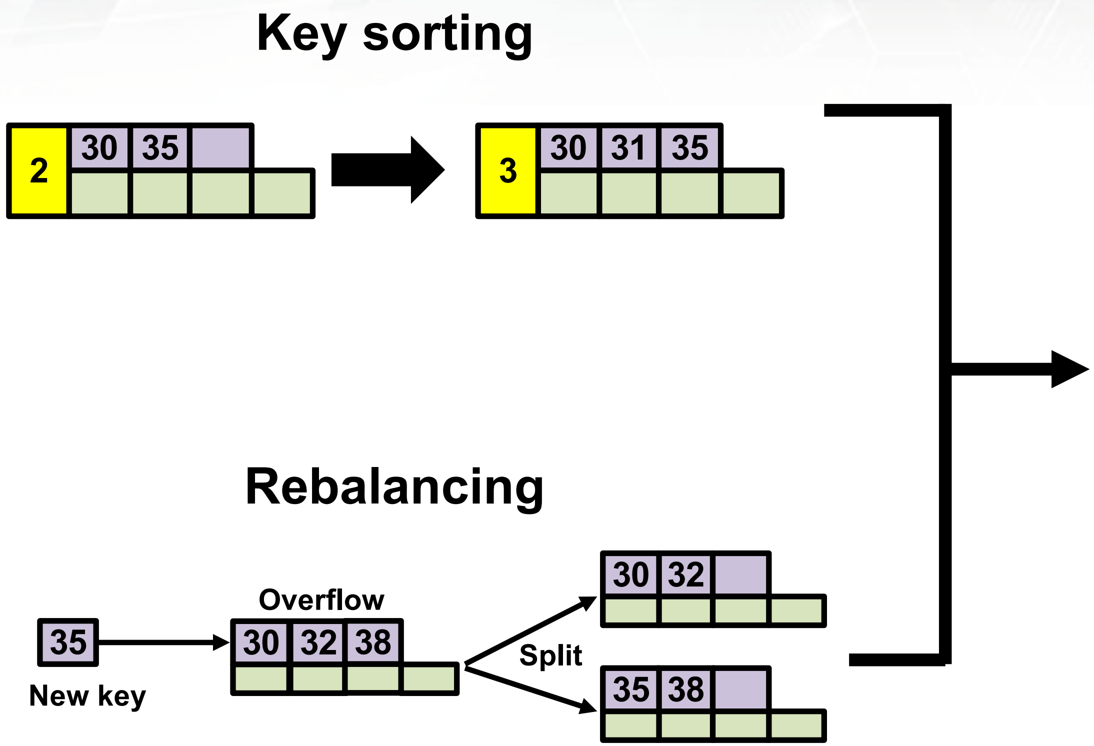
- **Logging still necessary**
  - Multi-block granularity updates due to node splits and merges
    - Cannot update atomically



  - Logging-based solution
    - wB+Tree, FPTree
  - Tree reconstruction based solution
    - NVTree

→ **large overhead**

B+tree Variants for Persistent Memory



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

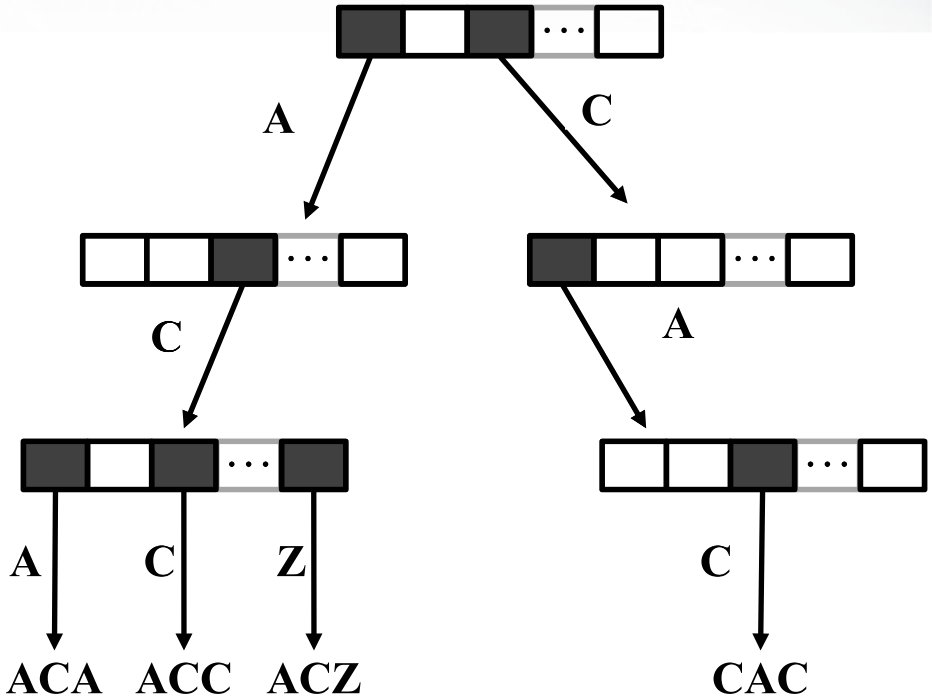
- **Show Radix Tree is a suitable data structure for PM**
- **Propose optimal radix tree variants WORT and WOART**
  - WORT: Write Optimal Radix Tree
  - WOART: Write Optimal redesigned Adaptive Radix Tree (ART)

Optimal: maintain consistency only with single failure-atomic write without any duplicate copies

Radix Tree

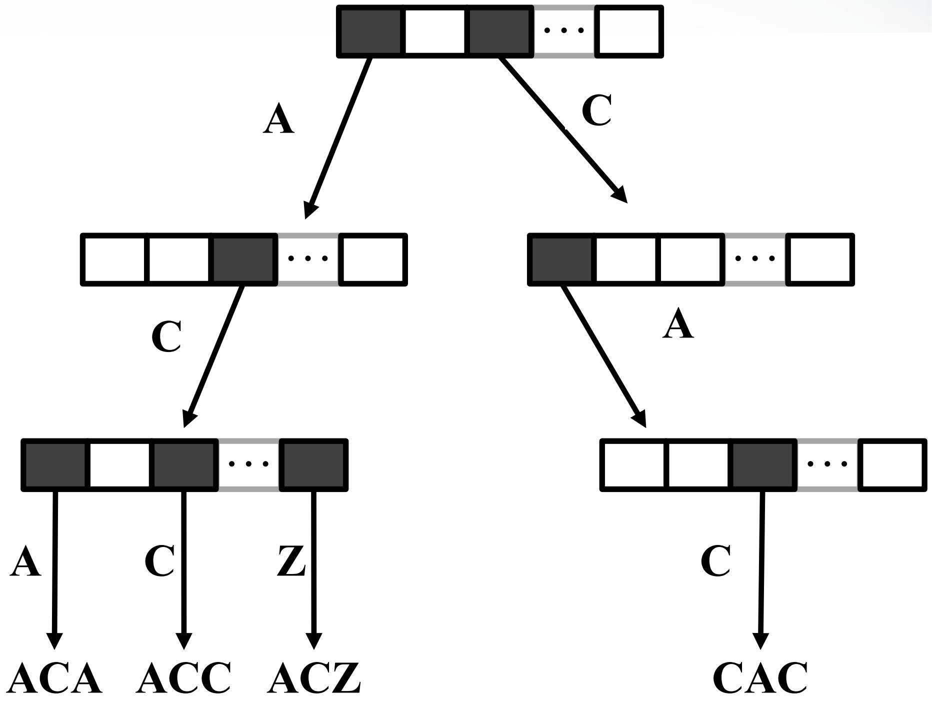
Background

- **Deterministic structure**



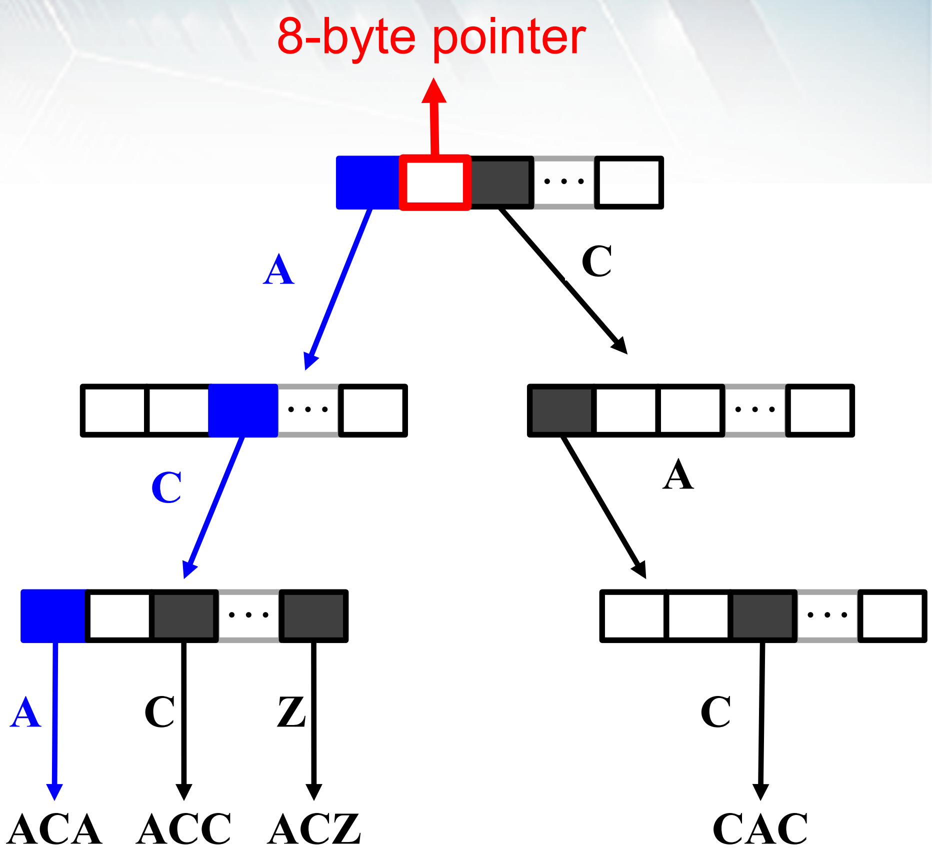
Radix Tree

- **Deterministic structure**
  - No key comparison



Radix Tree

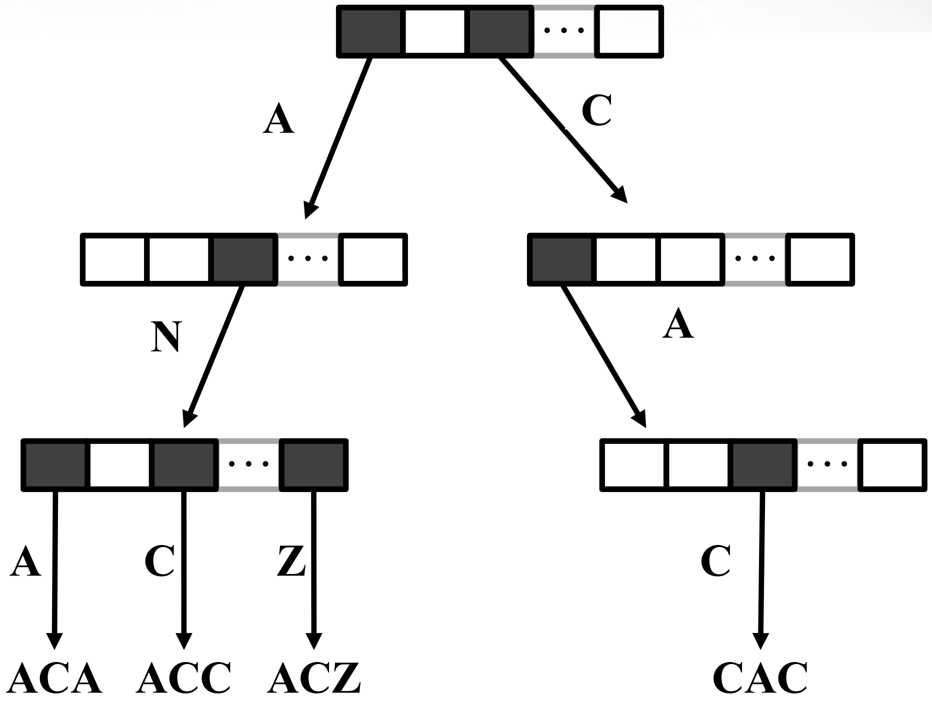
- **Deterministic structure**
  - No key comparison
    - Only 8-byte pointer entries
    - Implicitly stored keys



Radix Tree

Background

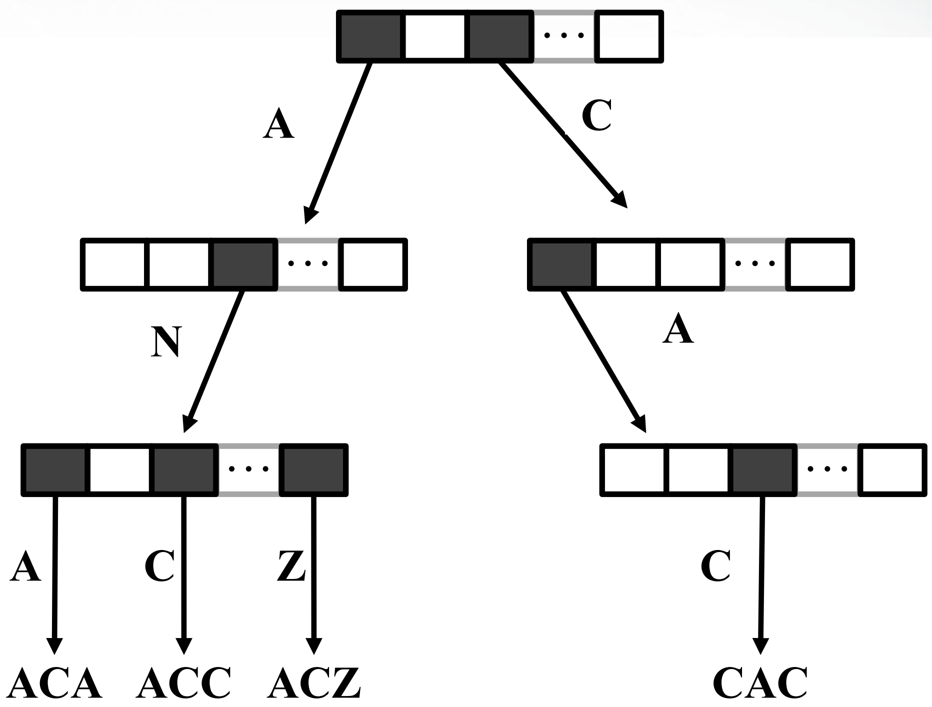
- **Deterministic structure**
  - No key comparison
    - Only 8-byte pointer entries
    - Implicitly stored keys
    - No problem caused by key sorting



Radix Tree

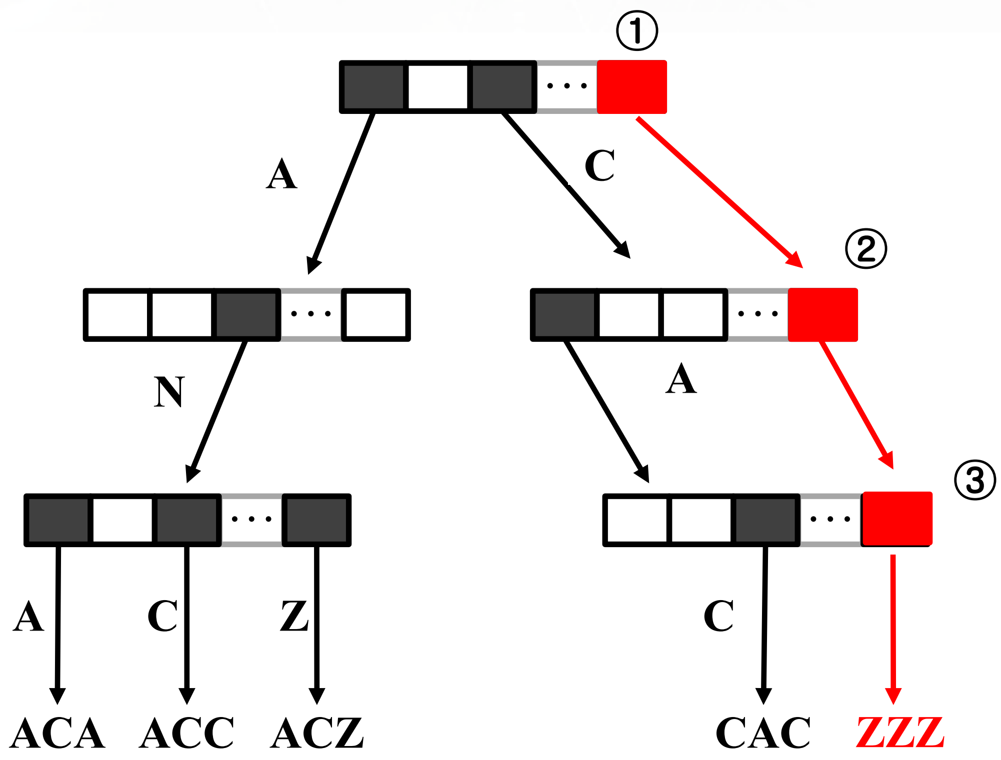
Background

- **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



Radix Tree Insertion Example

- **Insertion**
  - New nodes are sequentially linked

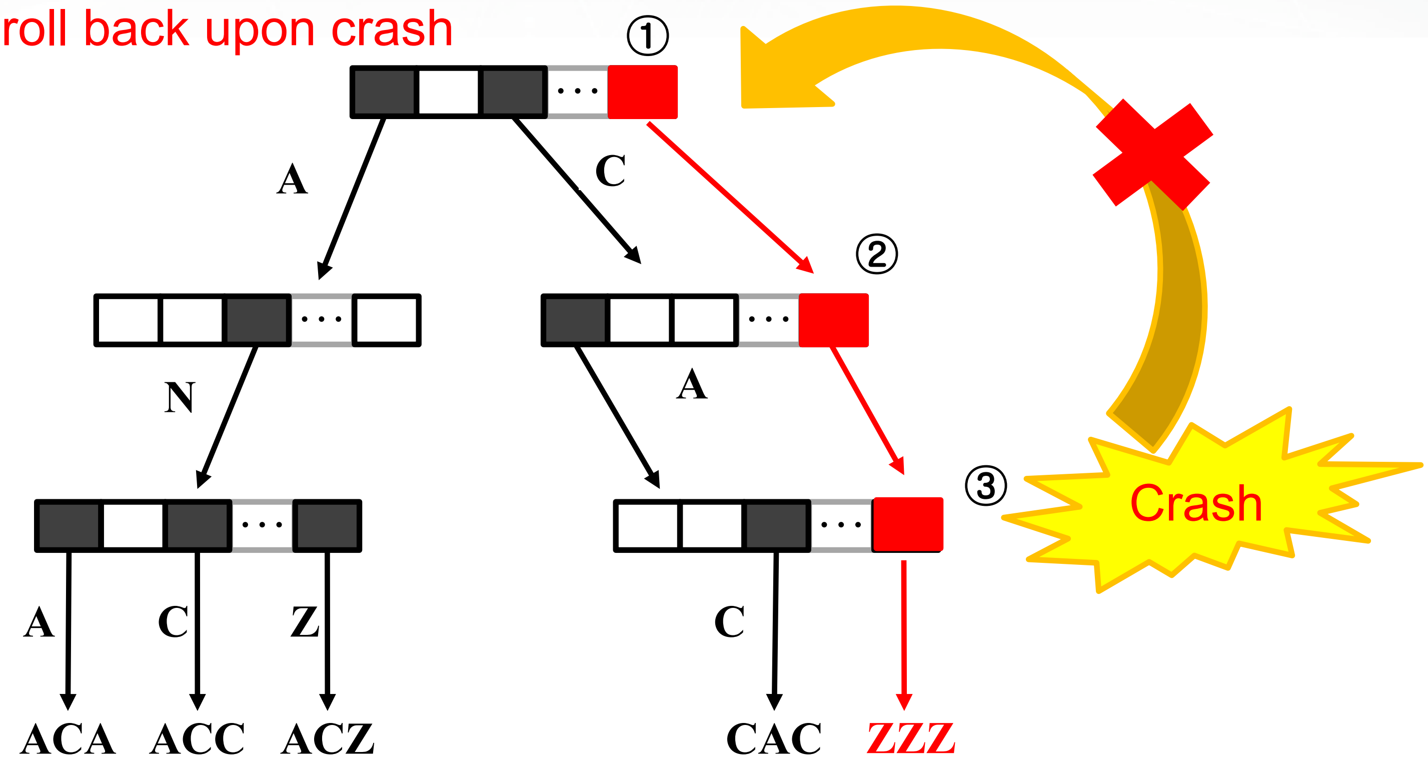


Two Issues with Original Radix Tree on PM

- **Insertion Order**
- **Path Compression**

Insertion process of original radix tree

- **Insertion**
  - New nodes are sequentially linked
    - Cannot roll back upon crash



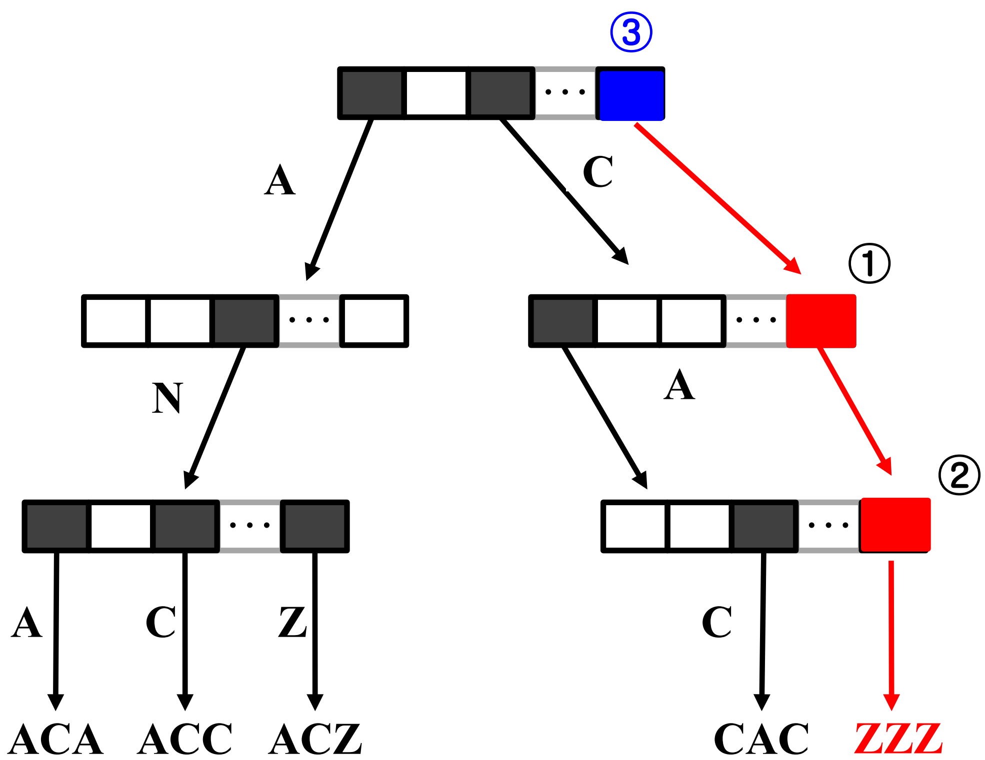
Two Issues with Original Radix Tree on PM

- **Insertion Order ➔ Our Solution**
- Path Compression

WORT (Write Optimal Radix Tree) for PM

Our solution

- **Change order of writes**
  - Operation to replace the very first NULL pointer with the address of the next level node is set as the last operation



Next-generation Embedded / Computer System Software Technology

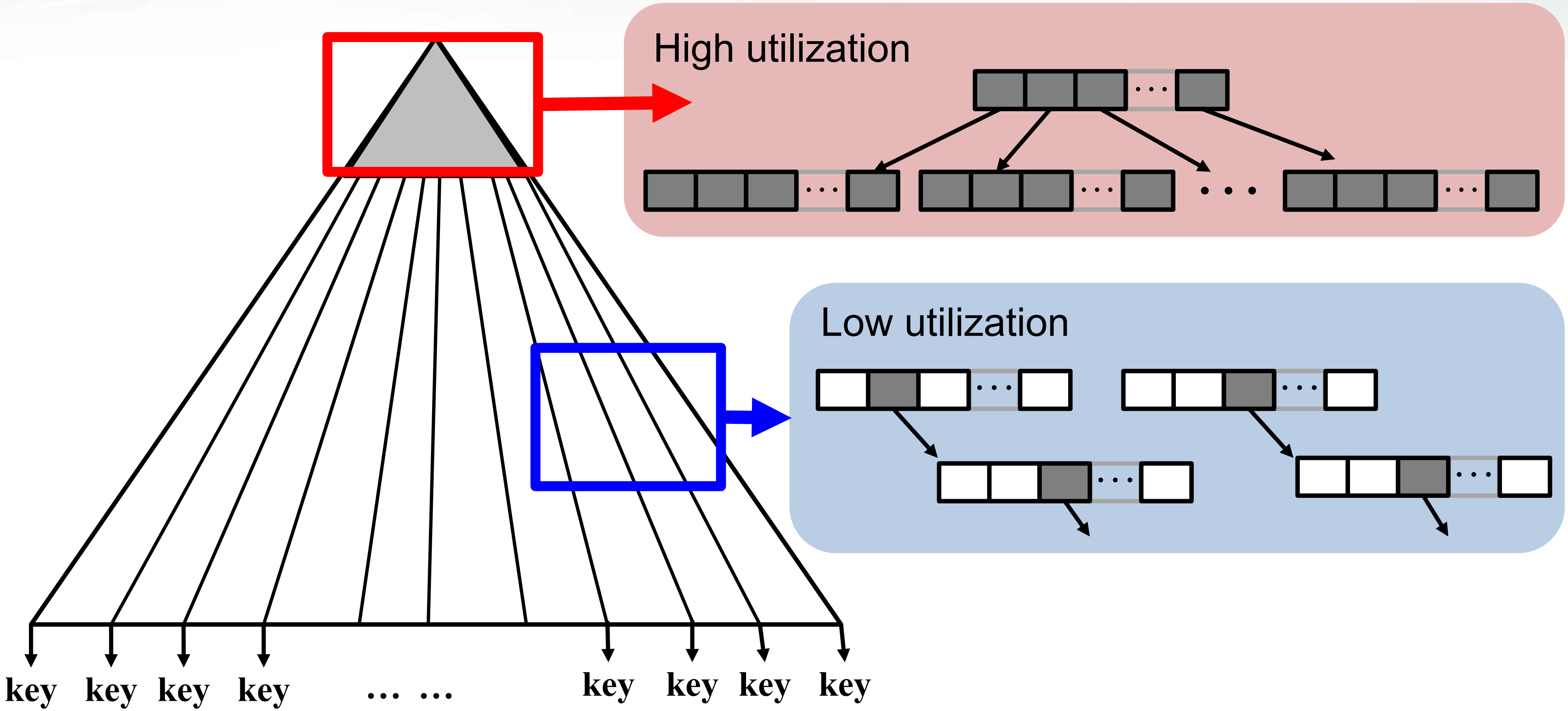
Two Issues with Original Radix Tree on PM

- Insertion Order
- **Path Compression**

Problem of Deterministic Structure

Background

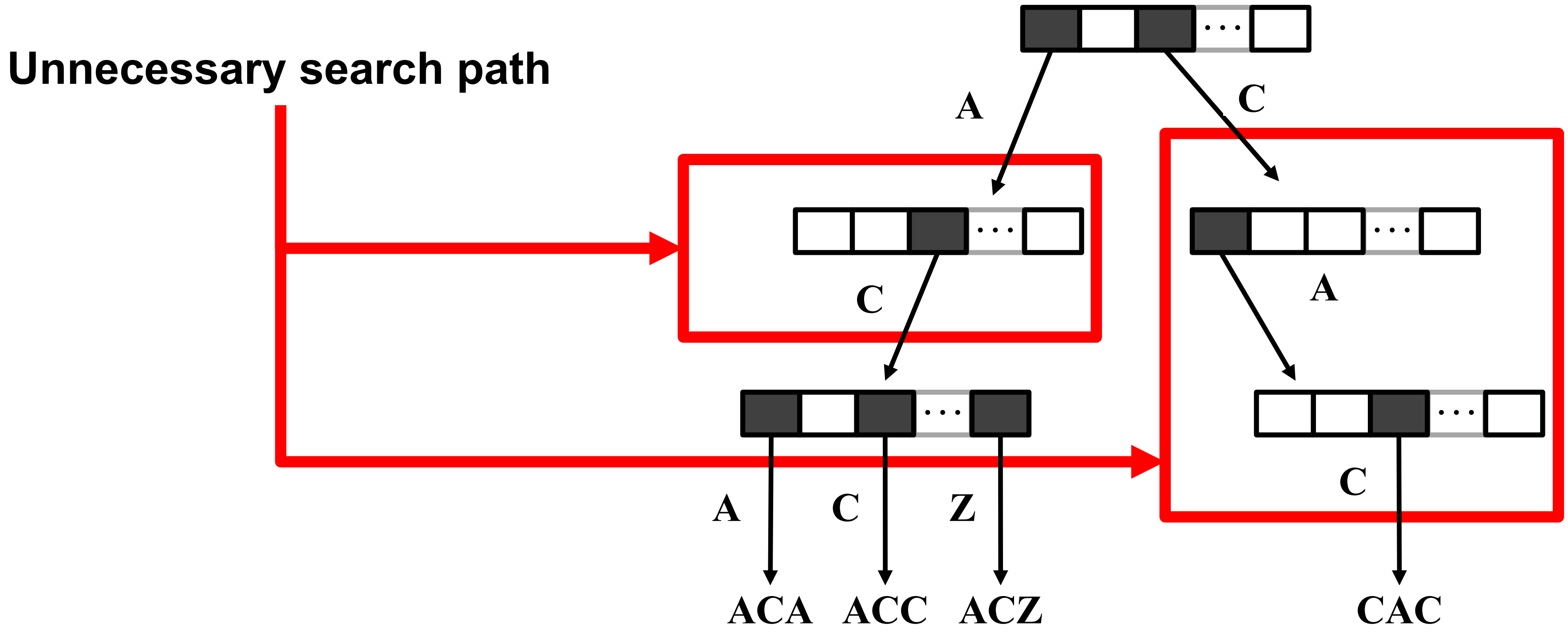
- **For sparse key distribution**
  - Waste excessive memory space → Optimized through path compression



Path Compression in Radix Tree

Background

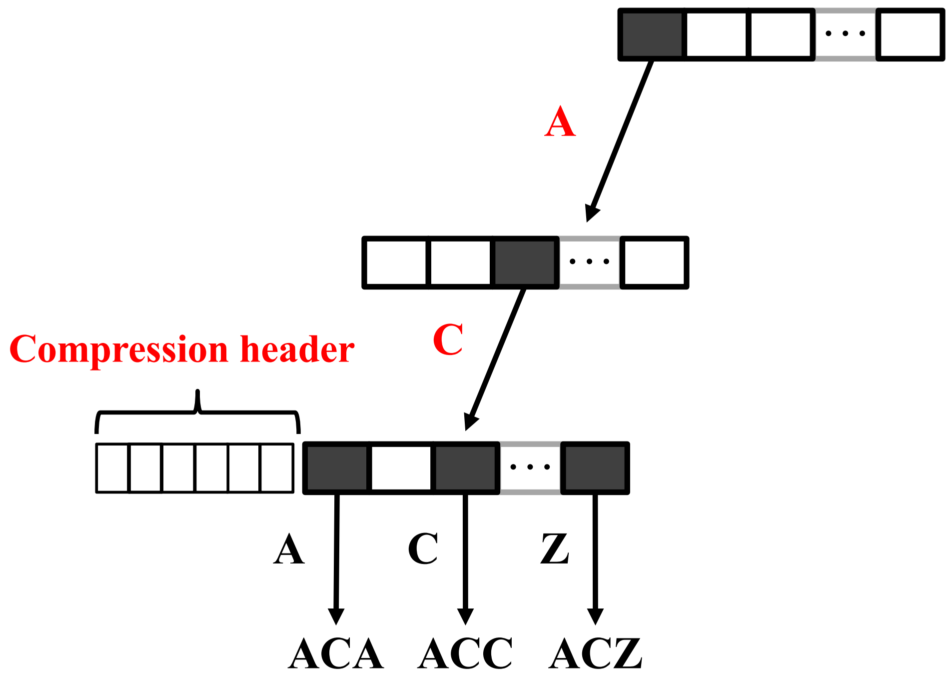
- **Path compression**
  - Search paths that do not need to be distinguished can be removed



Path Compression in Radix Tree

Background

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

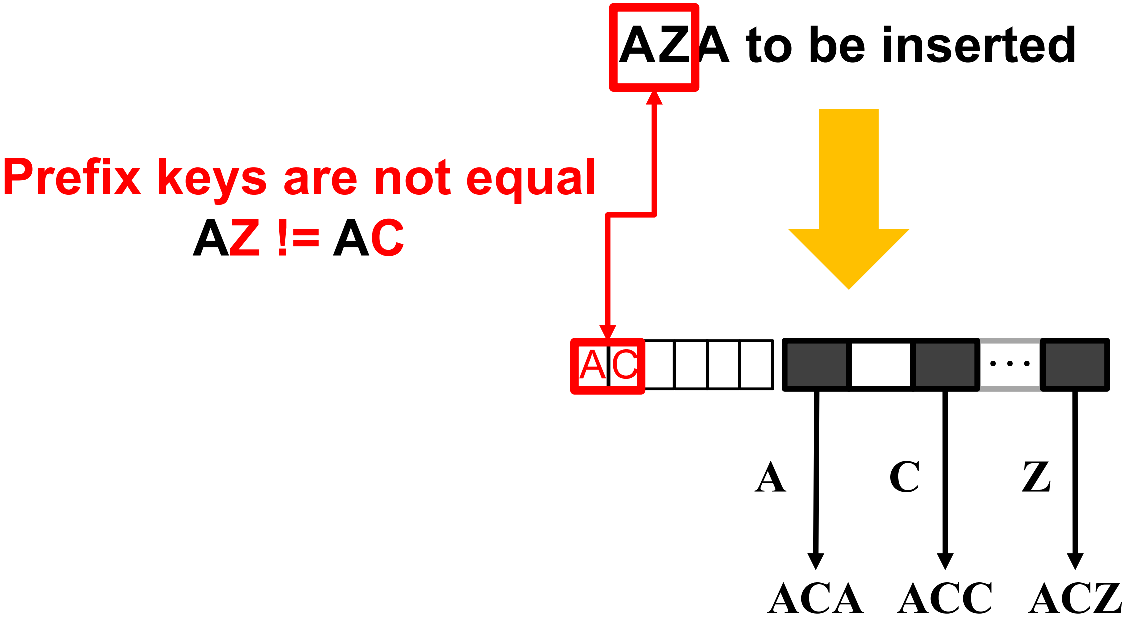


UNIST ECE CISSR NECSST Next-generation Embedded / Computer System Software Technology

Node Split with Path Compression

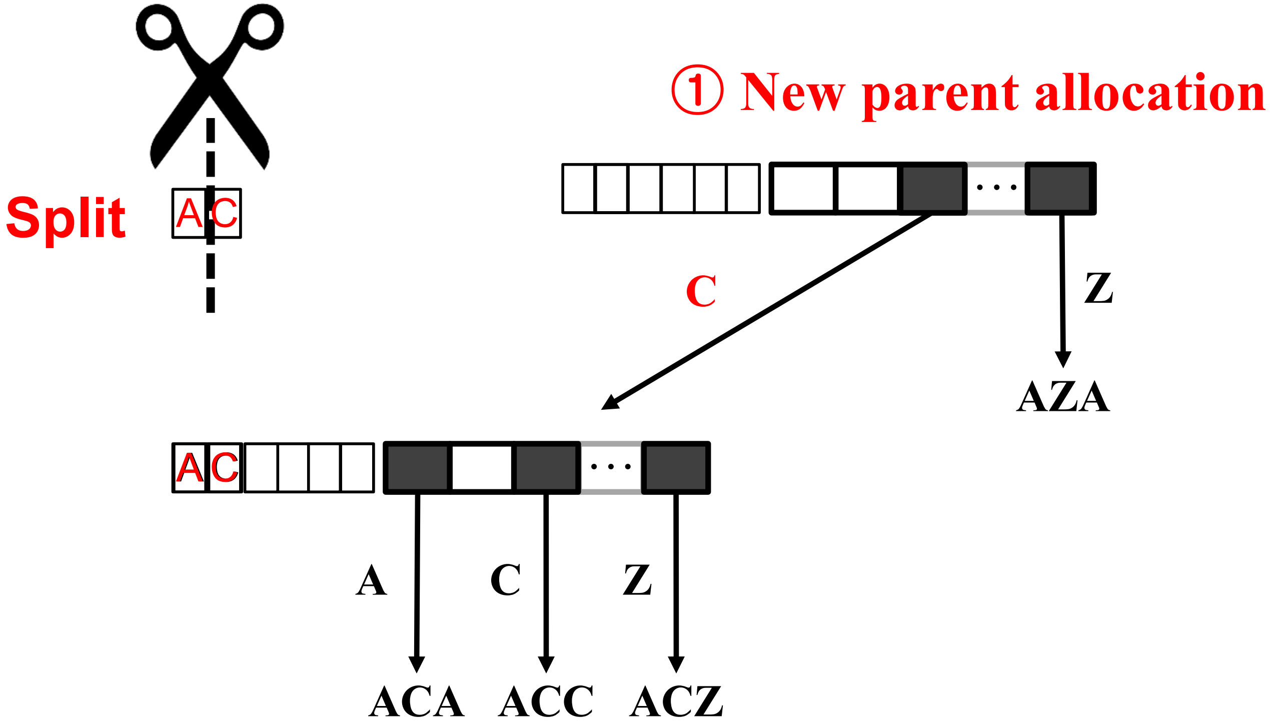
Background

- Path compression split



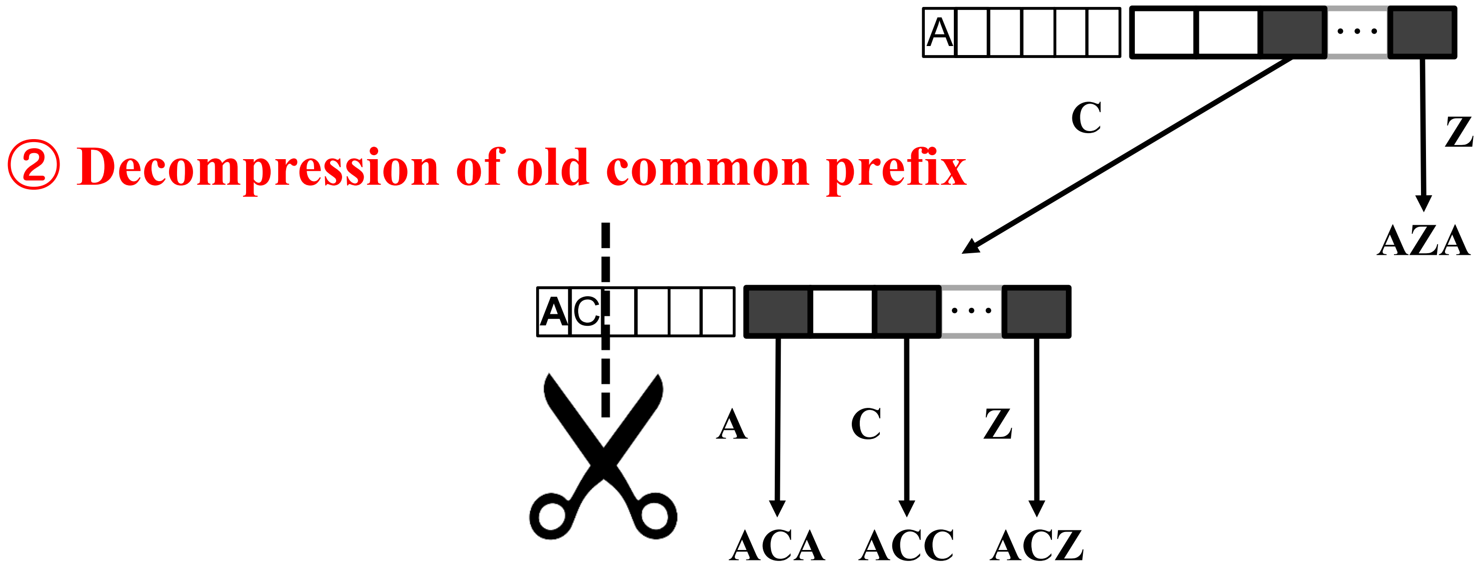
Node Split with Path Compression

- Path compression split



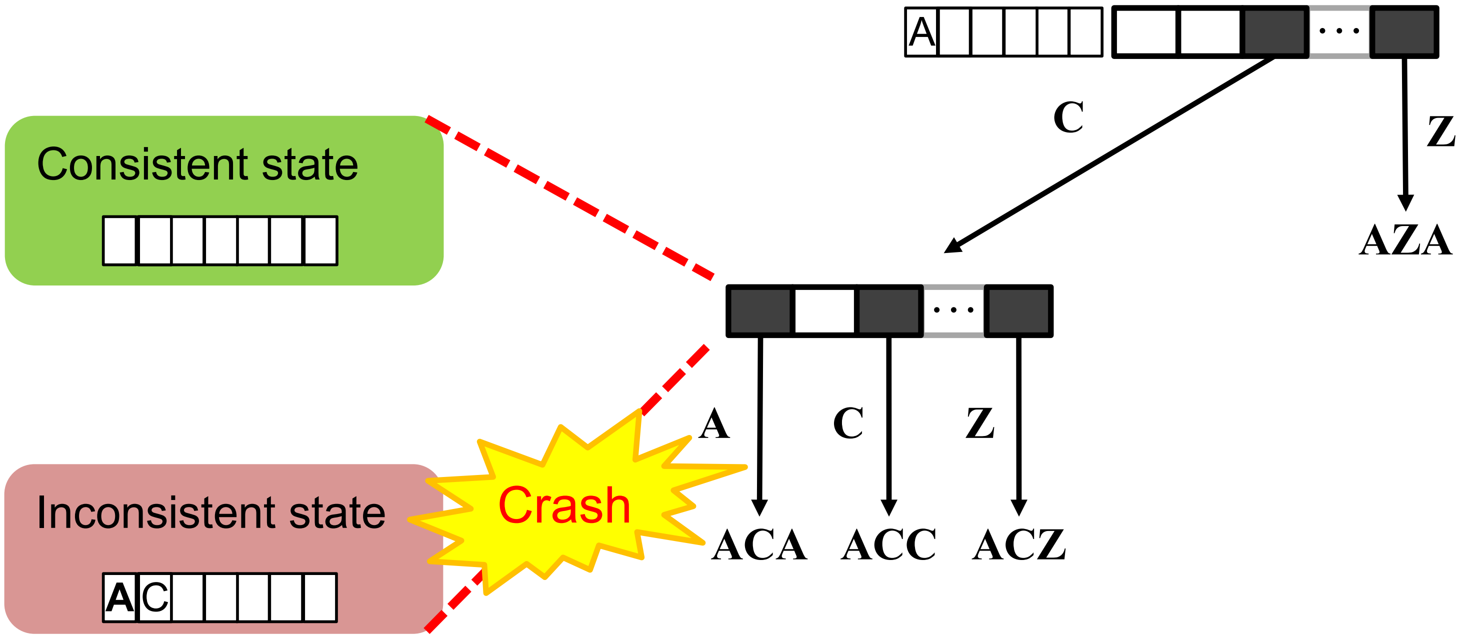
Node Split with Path Compression

- Path compression split



Consistency Issue of Path Compression

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



Two Issues with Original Radix Tree on PM

- Insertion Order
- **Path Compression**
  - Same issue as B+Tree node split
  - Consistency issue on PM

Two Issues with Original Radix Tree on PM

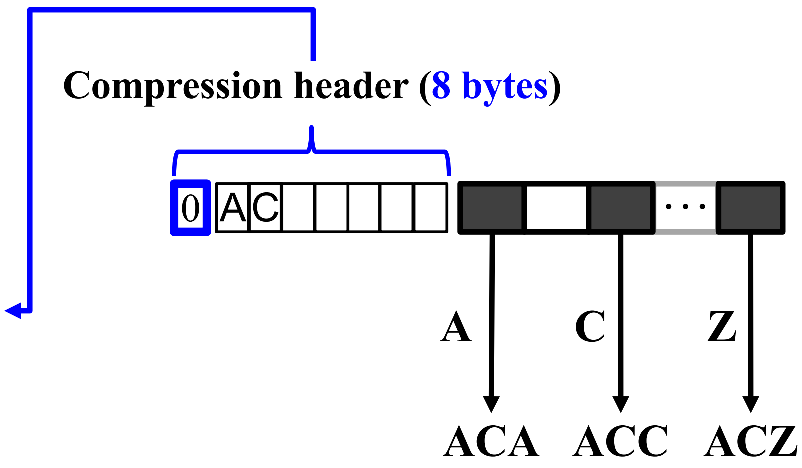
- Insertion Order
- **Path Compression ➔ Our Solution**
  - Same issue as B+Tree node split
  - Consistency issue on PM

WORT (Write-Optimal Radix Tree) for PM

Our solution

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

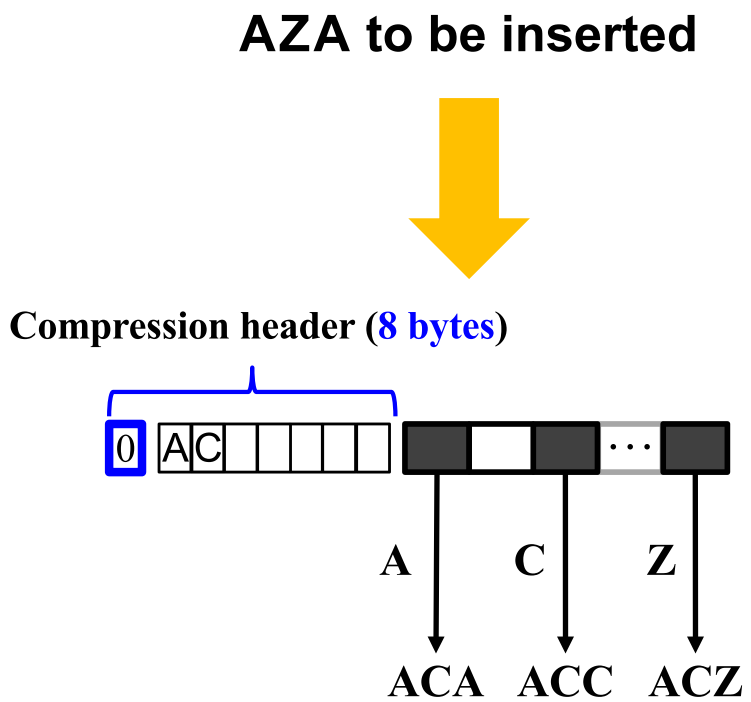
```
struct Header {
    unsigned char depth;
    unsigned char PrefixArr[7];
}
```



WORT (Write-Optimal Radix Tree) for PM

Our solution

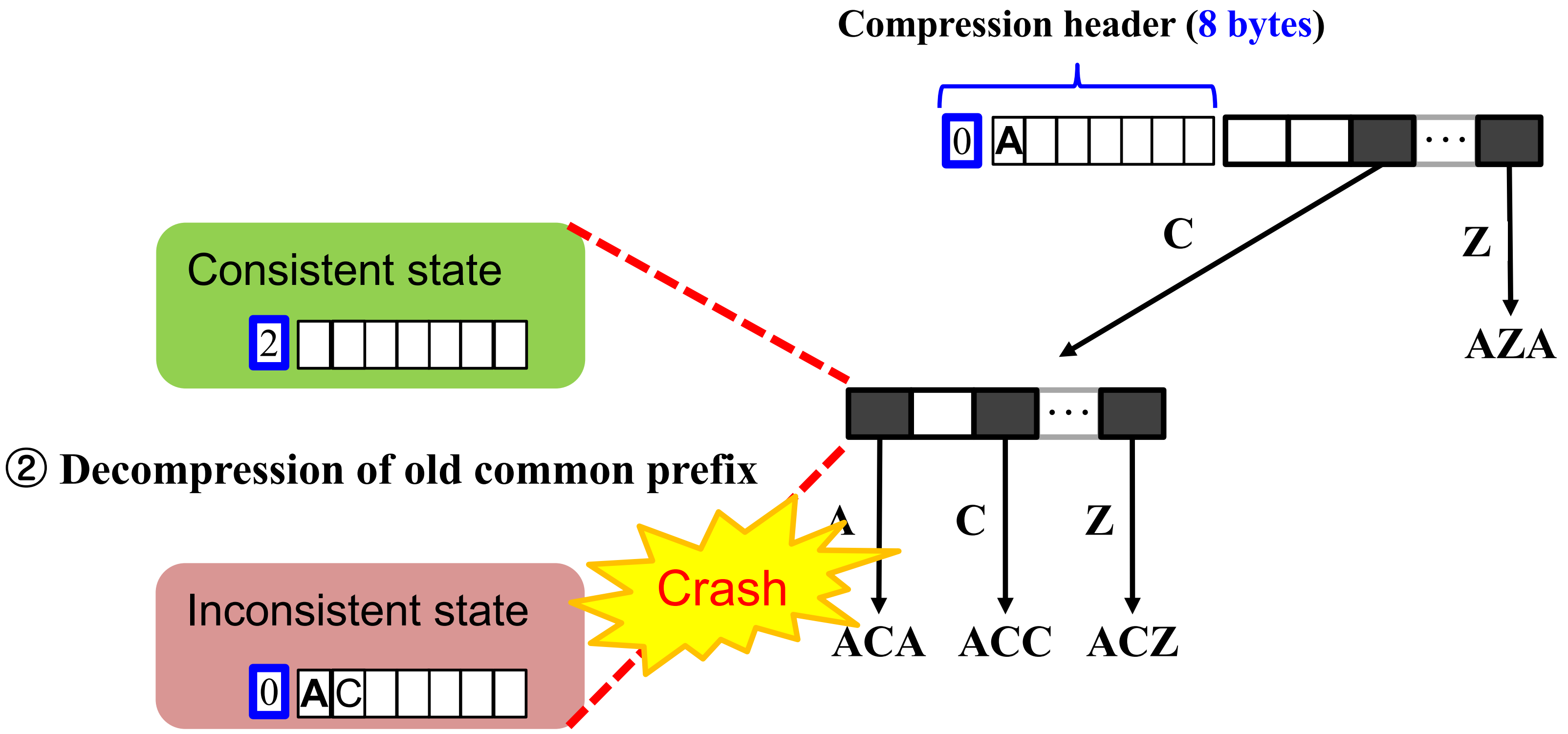
- Failure-atomic path compression
  - Add **node depth field** to compression header



WORT (Write-Optimal Radix Tree) for PM

Our solution

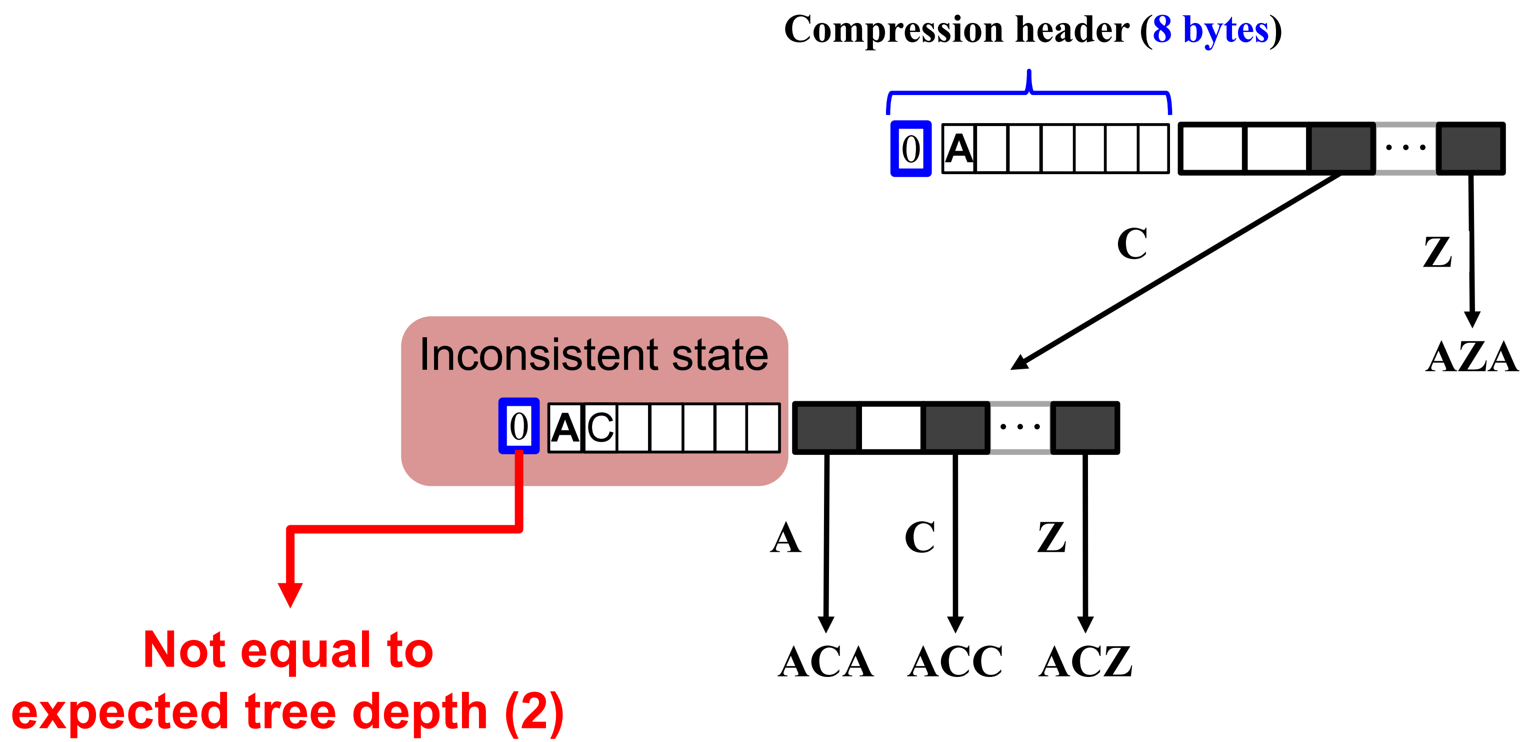
- Failure-atomic path compression
  - Add **node depth field** to compression header



WORT (Write-Optimal Radix Tree) for PM

Our solution

- **Failure-atomic path compression**
  - Failure detection in WORT
    - Depth in a header ≠ Counted depth → Crashed header

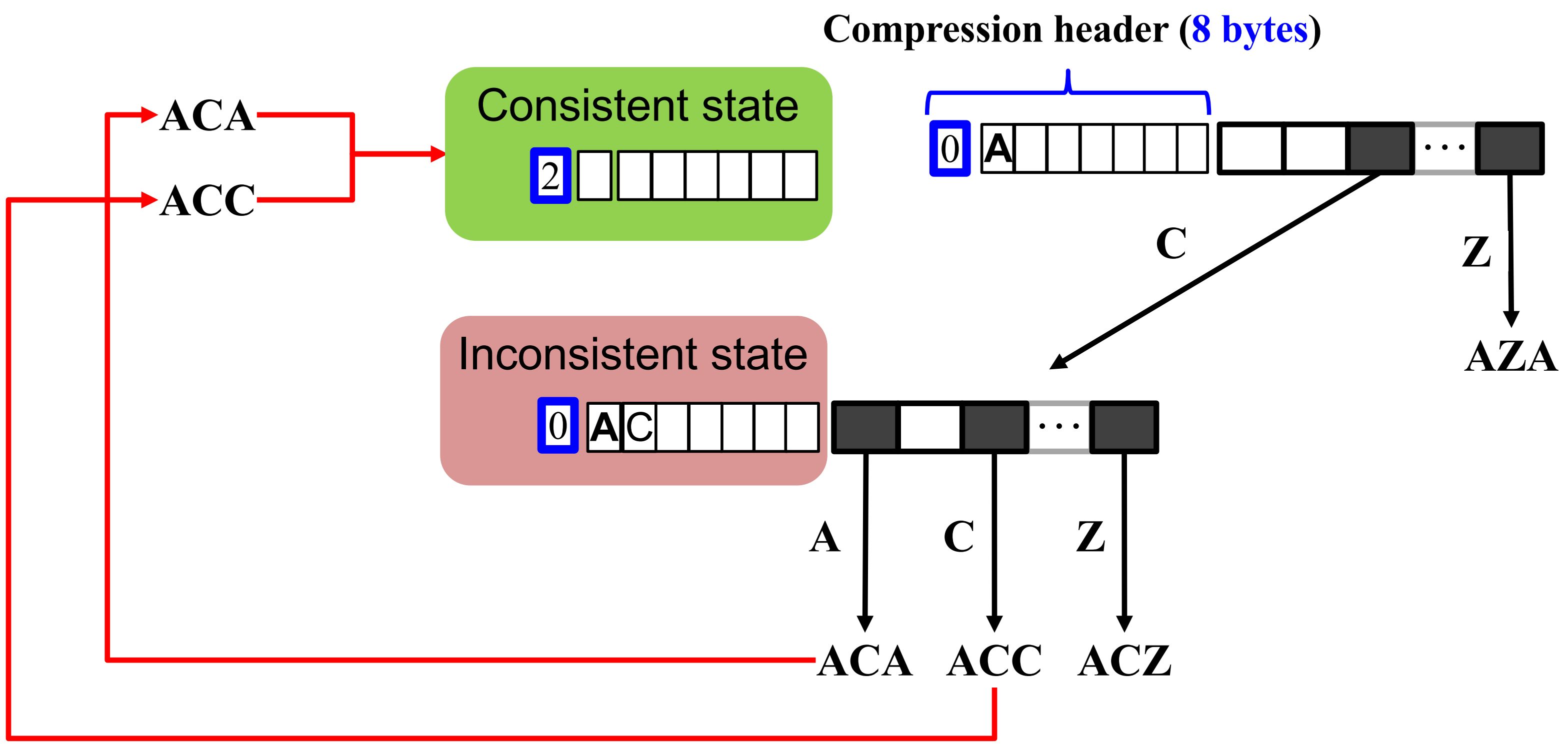


UNIST ECE SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING CISSR Center for Innovative System Software Research NECSST Next-generation Embedded / Computer System Software Technology

WORT (Write-Optimal Radix Tree) for PM

Our solution

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

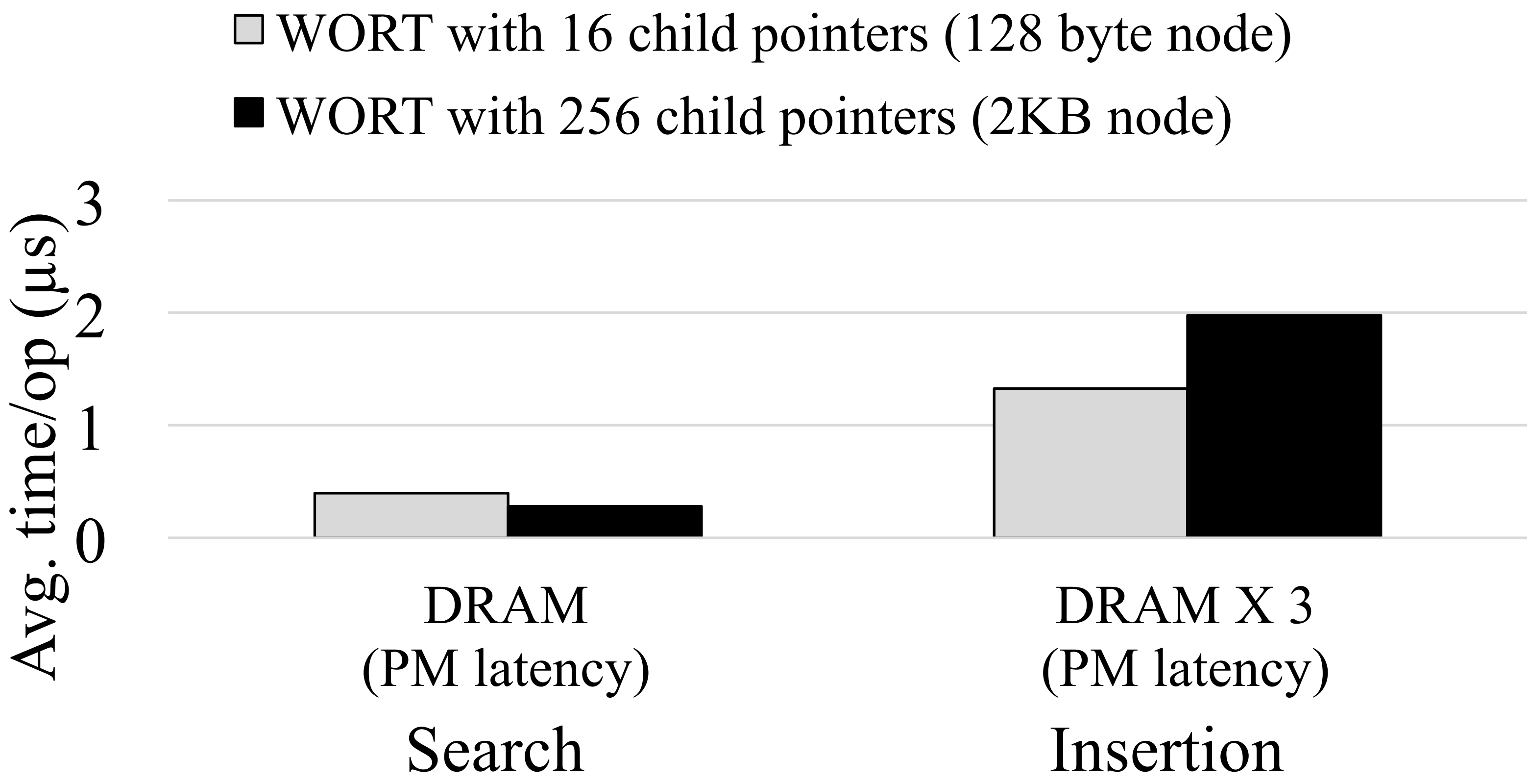


Yet Another Issue with Original Radix Tree

- Insertion Order
- Path Compression
- **Node Size**
  - Exists solutions
  - We focus on ART

Influence of Node Size on WORT

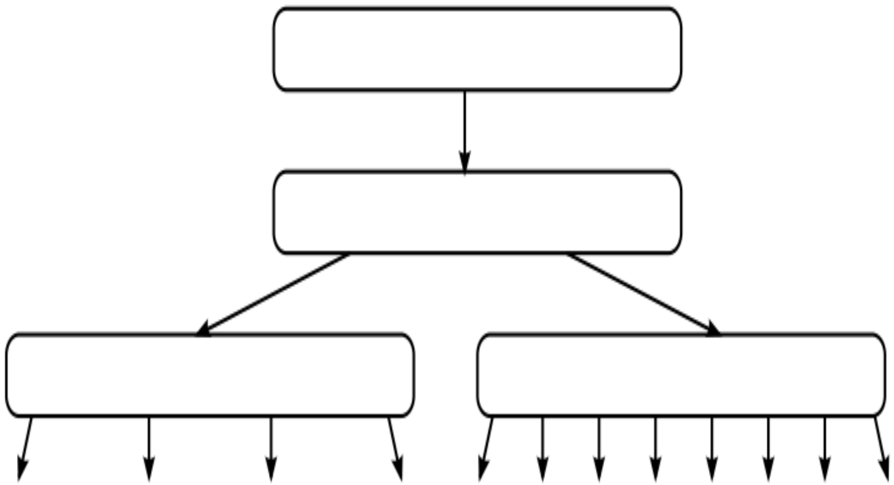
- **Trade off between node size and performance**
  - Node size ↑ → Tree height ↓ → Search performance ↑
  - Node size ↑ → Space overhead ↑ → Insertion performance ↓



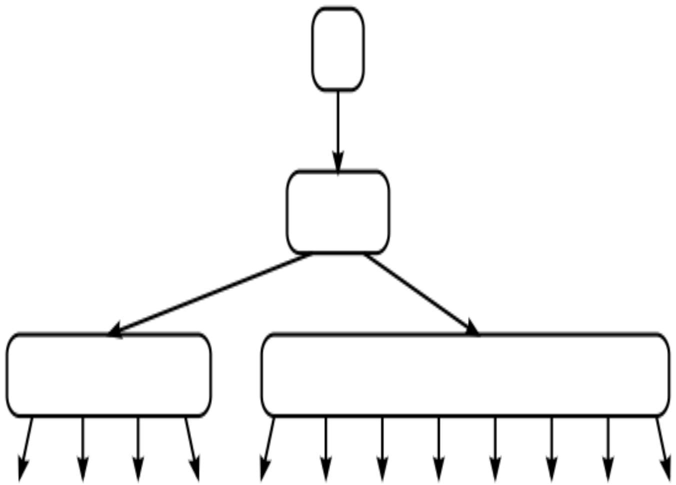
Adaptive Radix Tree (ICDE 2013)

Background

- **Adaptive nodes**
  - Nodes are exchanged with different sizes according to the number of valid entries



<Original radix tree>

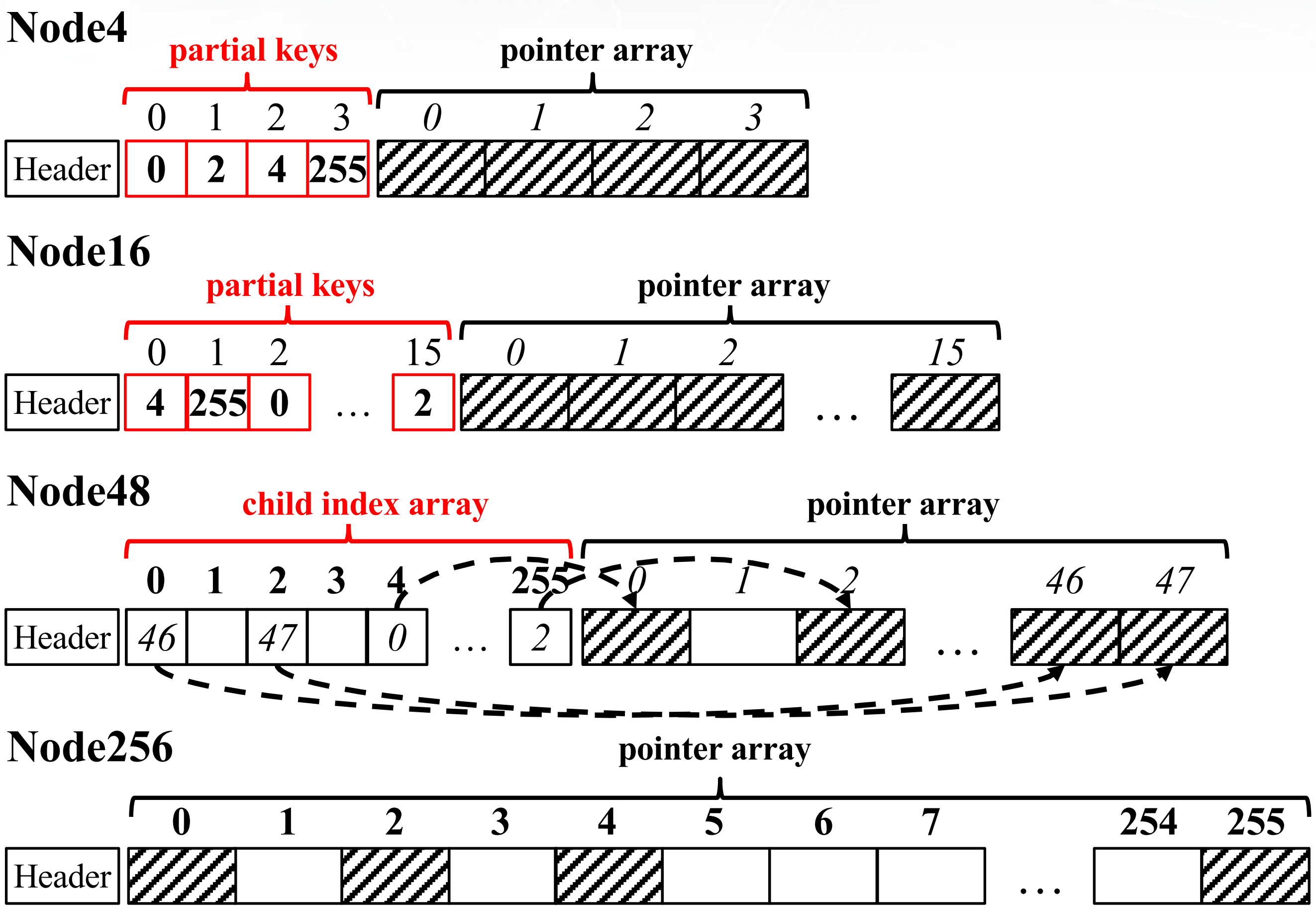


<Adaptive Radix Tree>

Adaptive Radix Tree (ICDE 2013)

Background

- **Designs of adaptive nodes**
  - Additional array (partial keys and child index array) is a 1-byte reference to a key in the pointer array



UNIST ECE School of Electrical and Computer Engineering CISSR NECSST Next-generation Embedded / Computer System Software Technology

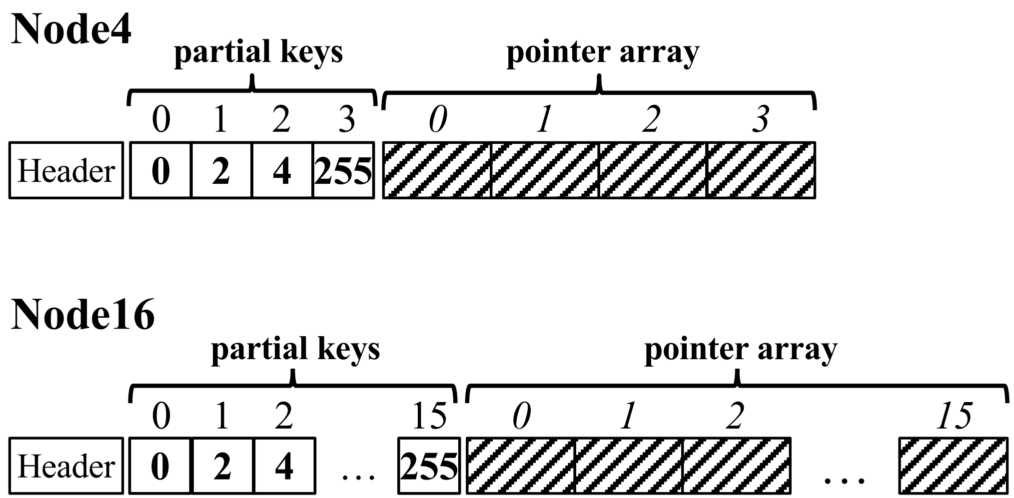
Yet Another Issue with Original Radix Tree

- Insertion Order
- Path Compression
- **Node Size**
  - Exists solutions
  - We focus on ART

  → Issue on PM

Consistency Issue of Adaptive nodes

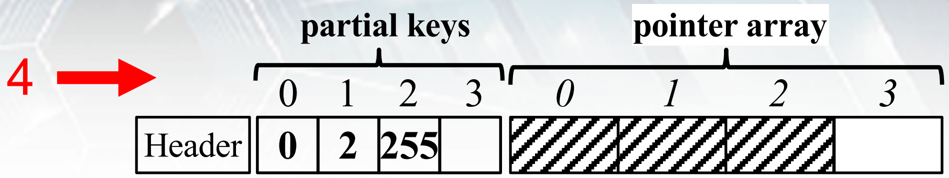
- **Node4 and Node16**
  - Partial keys and pointer array are sorted by order of character value
    - Update size is larger than 8-byte atomic write granularity
      → need logging or CoW to guarantee consistency



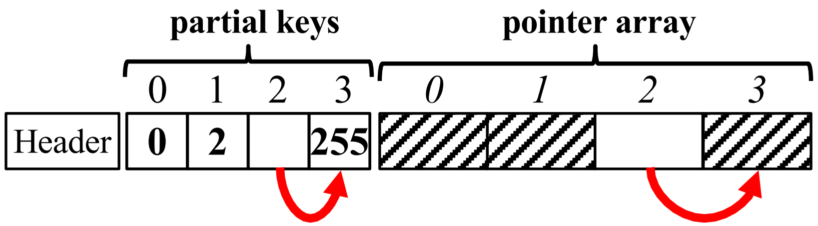
Consistency Issue of Adaptive nodes

- **Insertion process of Node4**

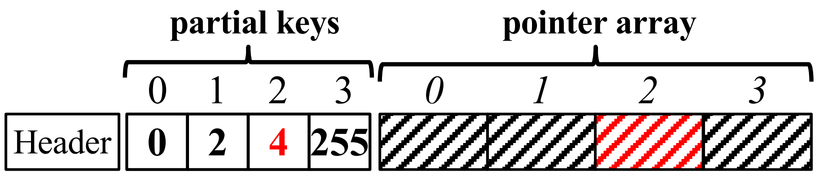
① Insert new key



② Existing entries are moved
To maintain sorted order



③ New key and child are inserted
in empty entries



Consistency of these insertion processes cannot be guaranteed with single 8-byte atomic write

Yet Another Issue with Original Radix Tree

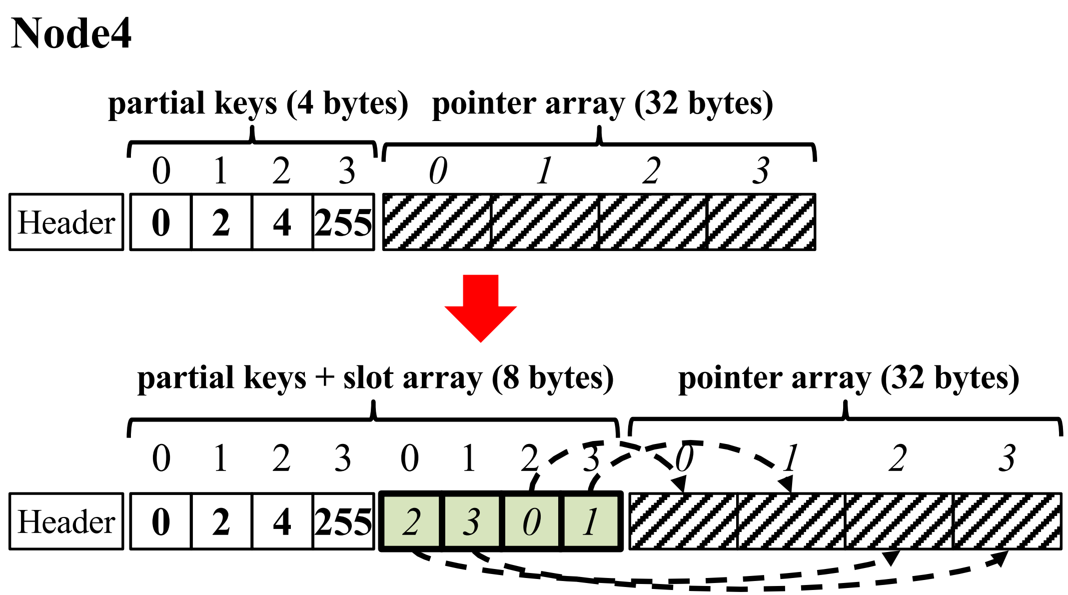
- Insertion Order
- Path Compression
- **Node Size**
  - Exists solutions
  - We focus on ART

  ➔ Issue on PM ➔ Our Solution

WOART (Write Optimal Adaptive Radix Tree)

Our solution

- **Redesigned Node4**
  - Pointer array is maintained in unsorted order
  - Slot array is used to indicate validation of each entry in pointer array and indirectly expresses sorted order of pointer array

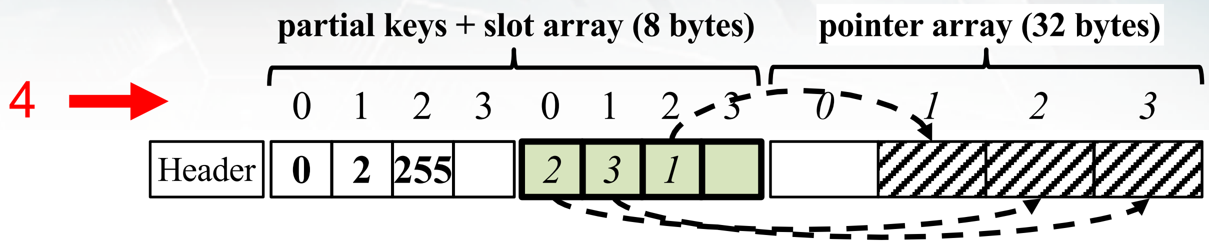


WOART (Write Optimal Adaptive Radix Tree)

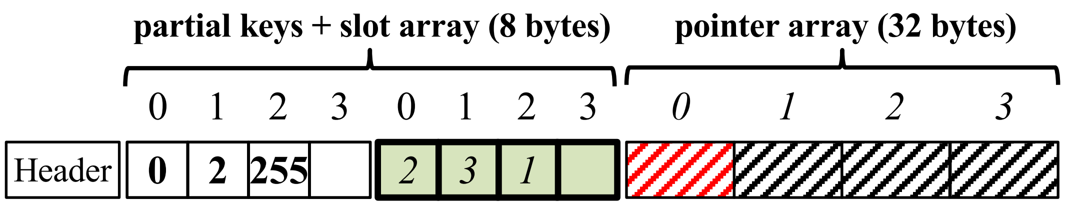
Our solution

- **Insertion process in redesigned Node4**

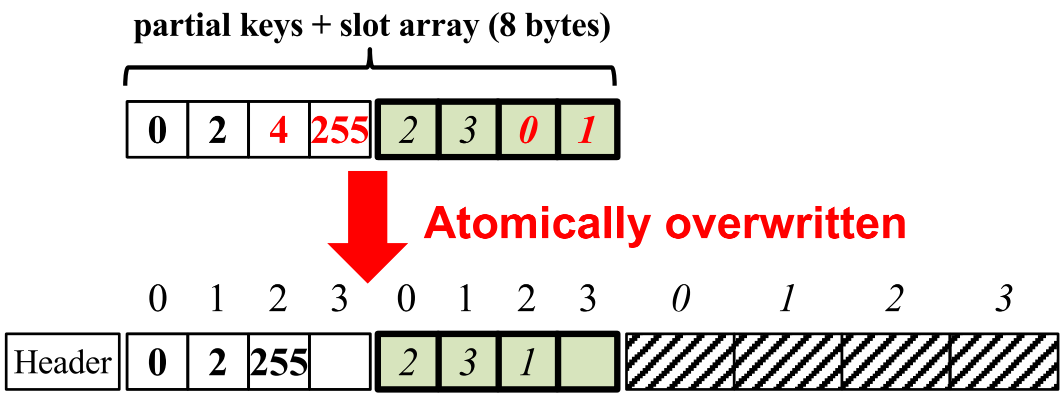
① Insert new key



② Empty entry in pointer array is found by slot array and written to



③ Partial keys and slot array is out-place updated, written atomically overwriting old value

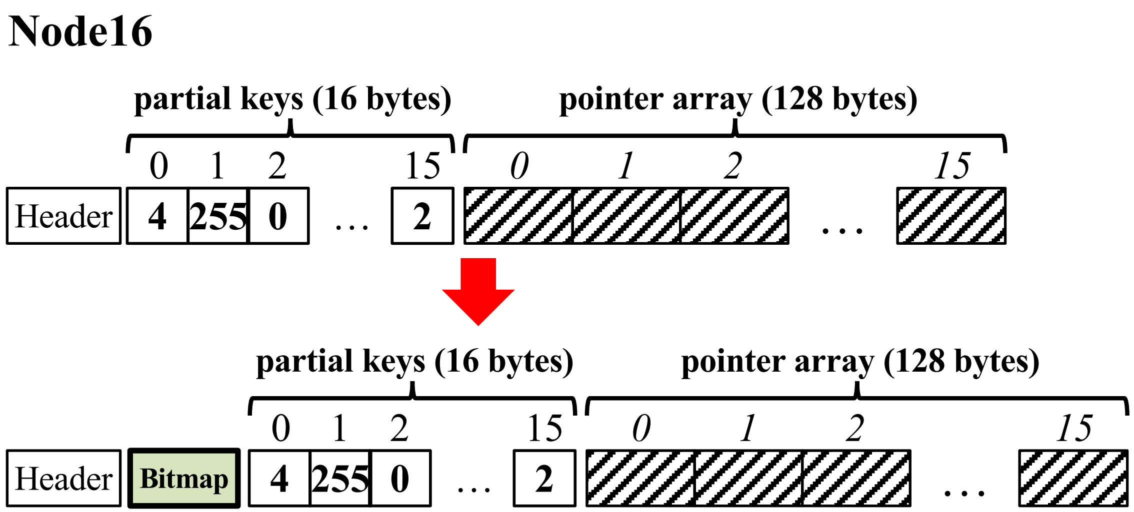


UNIST ECE School of Electrical and Computer Engineering CISSR Center for Innovative System Software Research NECSST Next-generation Embedded / Computer System Software Technology

WOART (Write Optimal Adaptive Radix Tree)

Our solution

- **Redesigned Node16**
  - Partial keys and pointer array are maintained in unsorted order
  - Bitmap is used to represent the validity of each entry in partial keys and pointer array



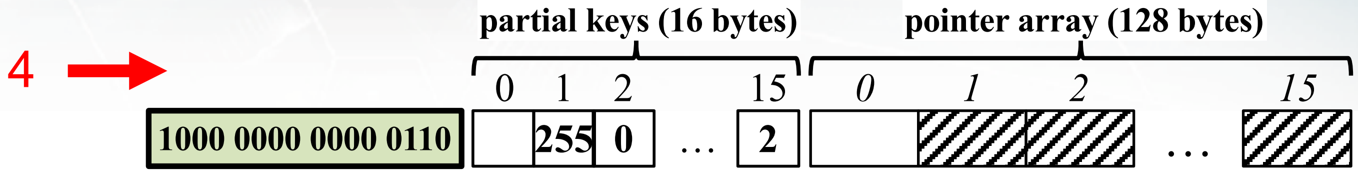
UNIST ECE CISSR NECSST Next-generation Embedded / Computer System Software Technology

WOART (Write Optimal Adaptive Radix Tree)

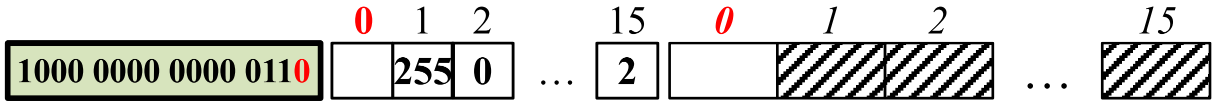
Our solution

- **Insertion process in redesigned Node16**

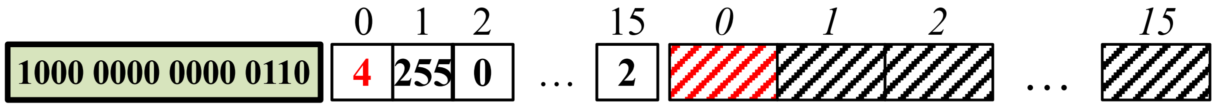
① Insert new key



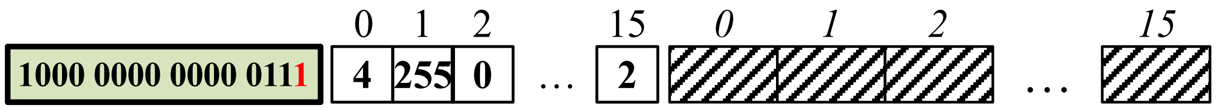
② Empty entries of partial key and pointer array are found by bitmap



③ Partial keys and pointer array are updated



④ Bitmap is atomically updated



Write Optimal Data Structure for PM

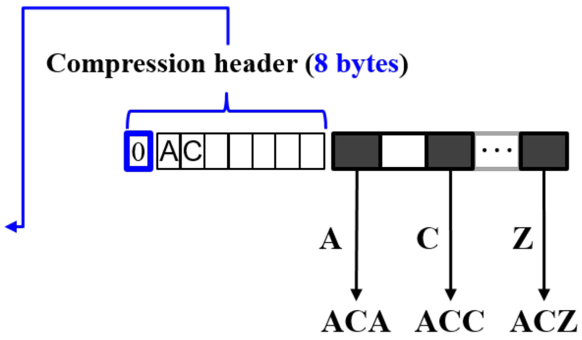
- **Our proposed radix tree variants are optimal for PM**
  - Consistency is always guaranteed with a single 8-byte failure-atomic write without any additional copies for logging or CoW

WORT (Write Optimal Radix Tree)

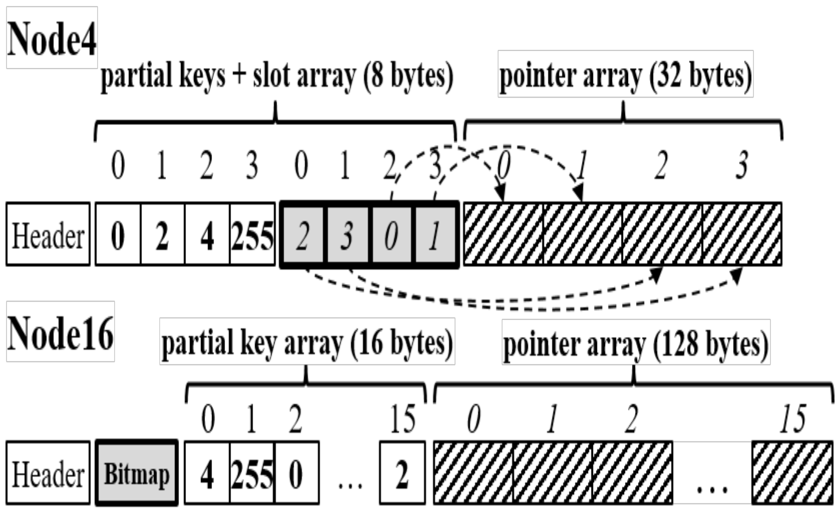
WOART (Write Optimal Adaptive Radix Tree)

1. Failure-atomic path compression

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



2. Redesigned adaptive node



Evaluation

- **Experimental environment**

System configuration

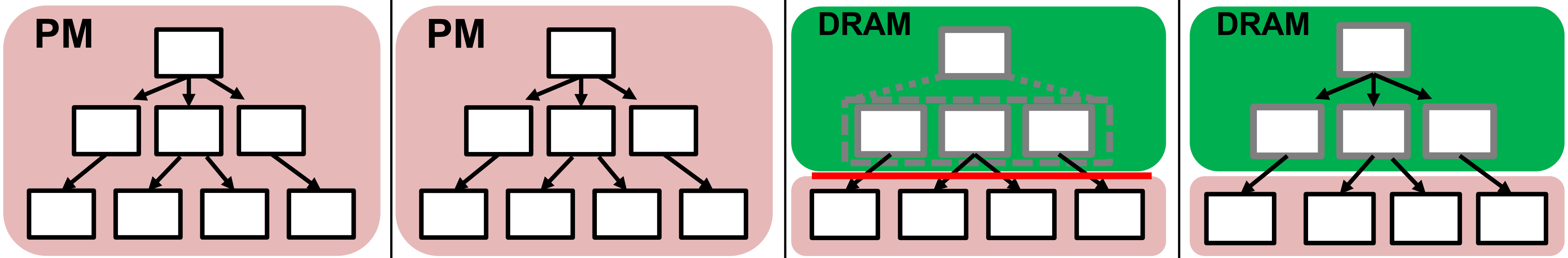
	Description
CPU	Intel Xeon E5-2620V3 X 2
OS	Linux CentOS 6.6 (64bit) kernel v4.7.0
PM	Emulated with 256GB DRAM Write latency: Inject additional stall cycles

Evaluation

- **Experimental environment**

Comparison group

Radix tree variants	B+tree variants		
WORT	wB+Tree (VLDB' 15)	NVTree (FAST' 15)	FPTree (SIGMOD' 16)

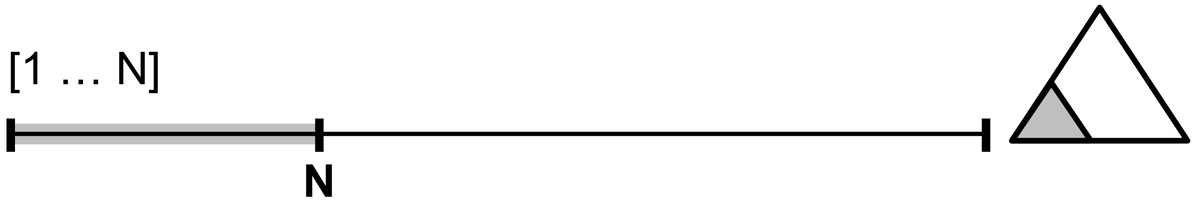


Evaluation

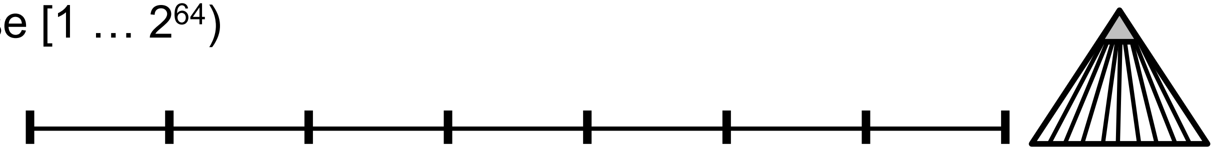
- **Experimental environment**

Synthetic Workload Characteristics

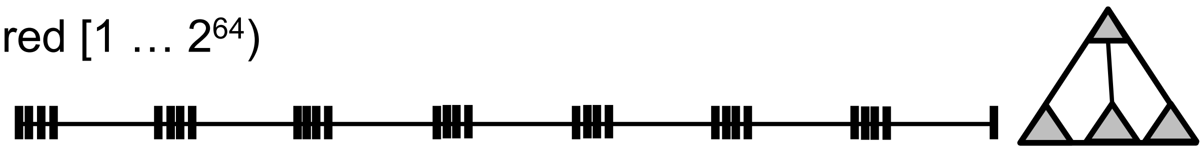
– Dense [1 … N]



– Sparse [1 … 2^{64})

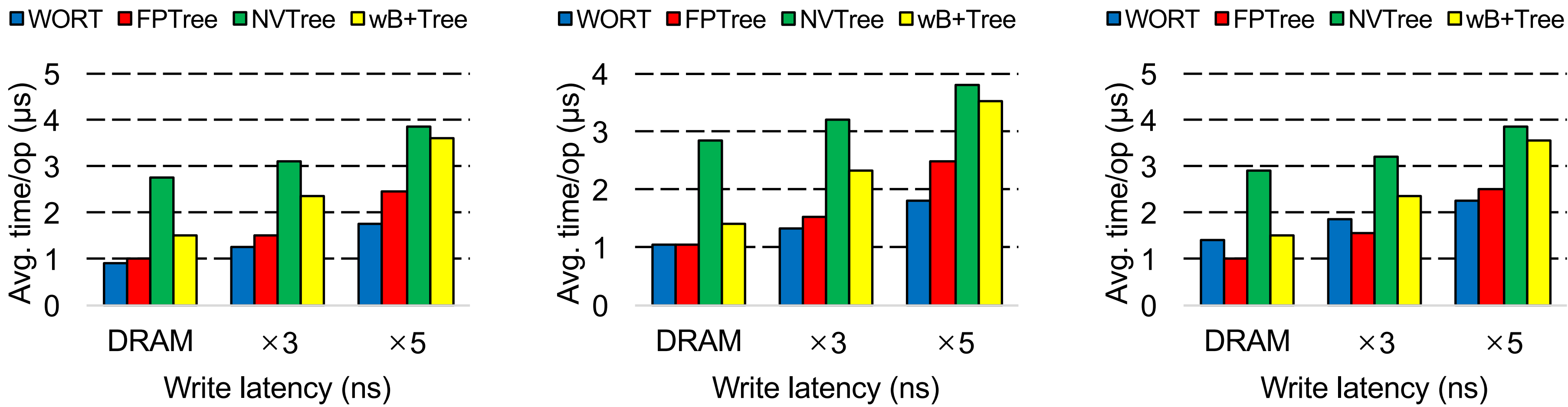


– Clustered [1 … 2^{64})



Evaluation

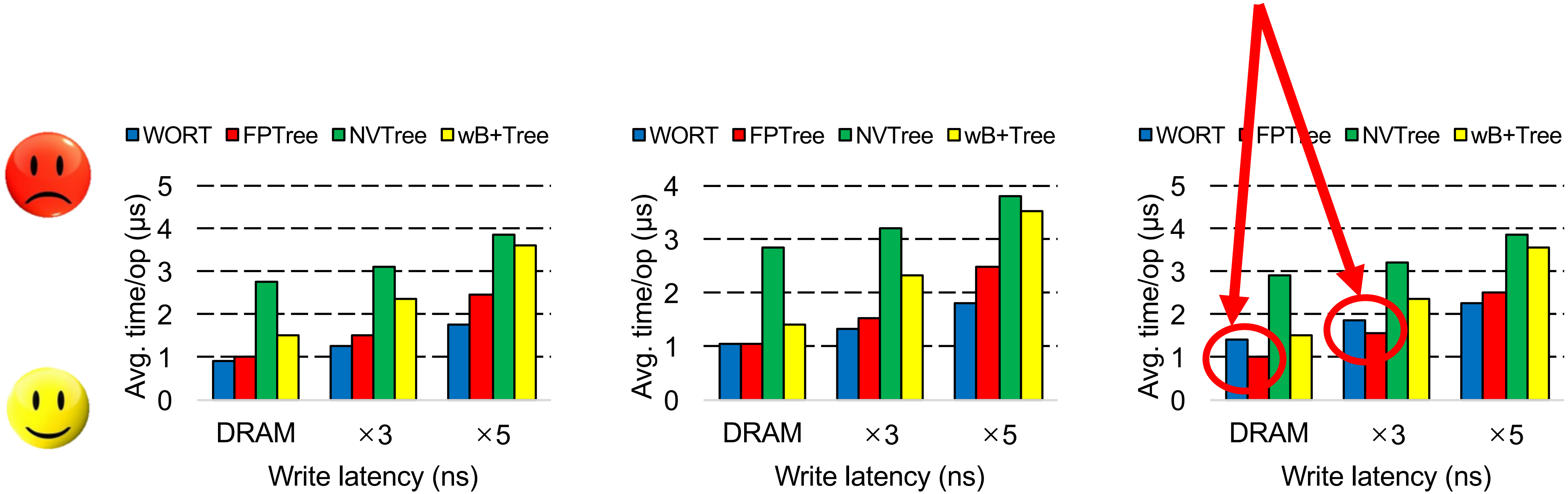
- **Insertion performance**
  - WORT and WOART generally outperform the B+Tree variants



(a) Dense **(b) Sparse** **(c) Clustered**

Evaluation

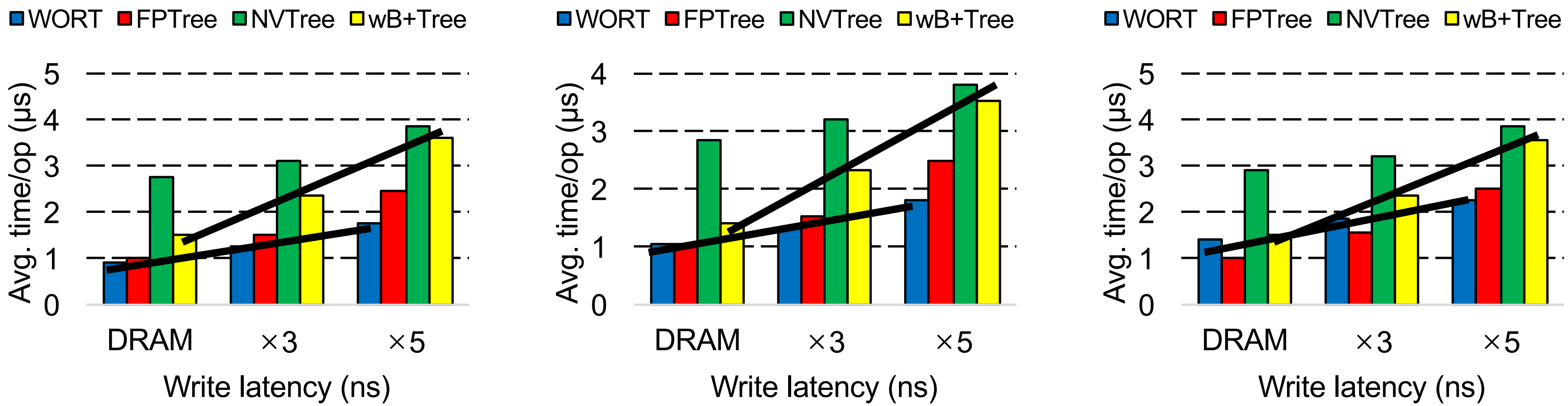
- **Insertion performance**
  - WORT and WOART generally outperform the B+Tree variants
    - DRAM-based internal node → more favorable performance for FPTree



(a) Dense (b) Sparse (c) Clustered

Evaluation

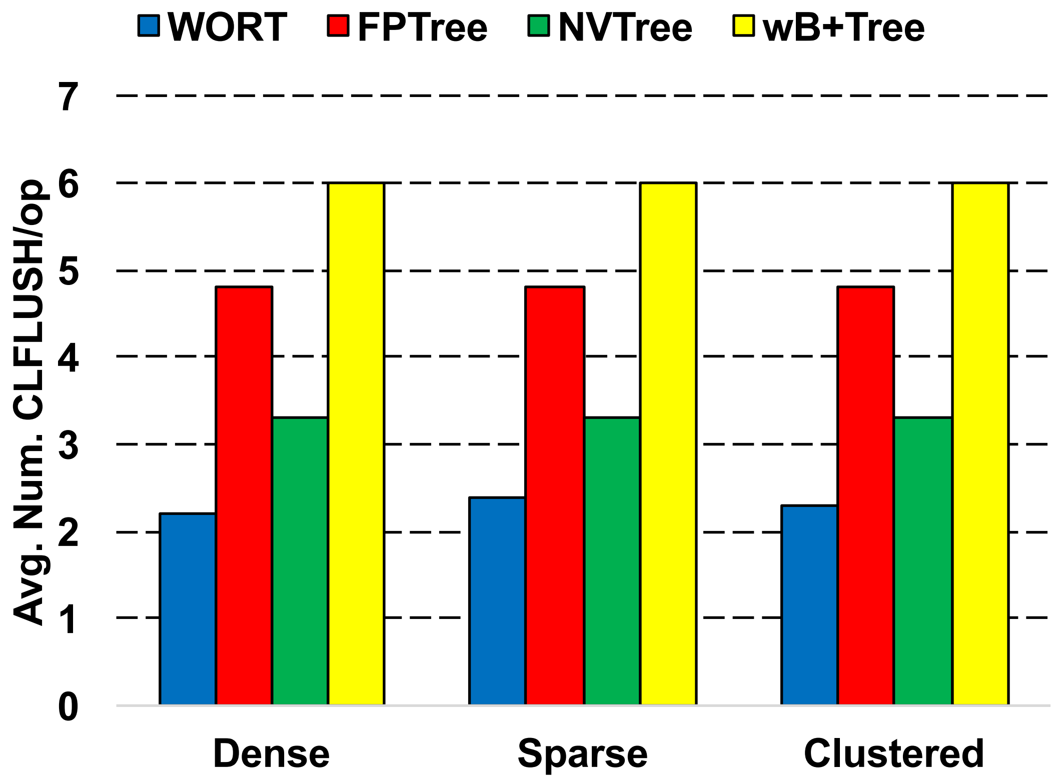
- **Insertion performance**
  - WORT vs wB+Tree
    - – Performance differences increase in proportion to write latency



(a) Dense **(b) Sparse** **(c) Clustered**

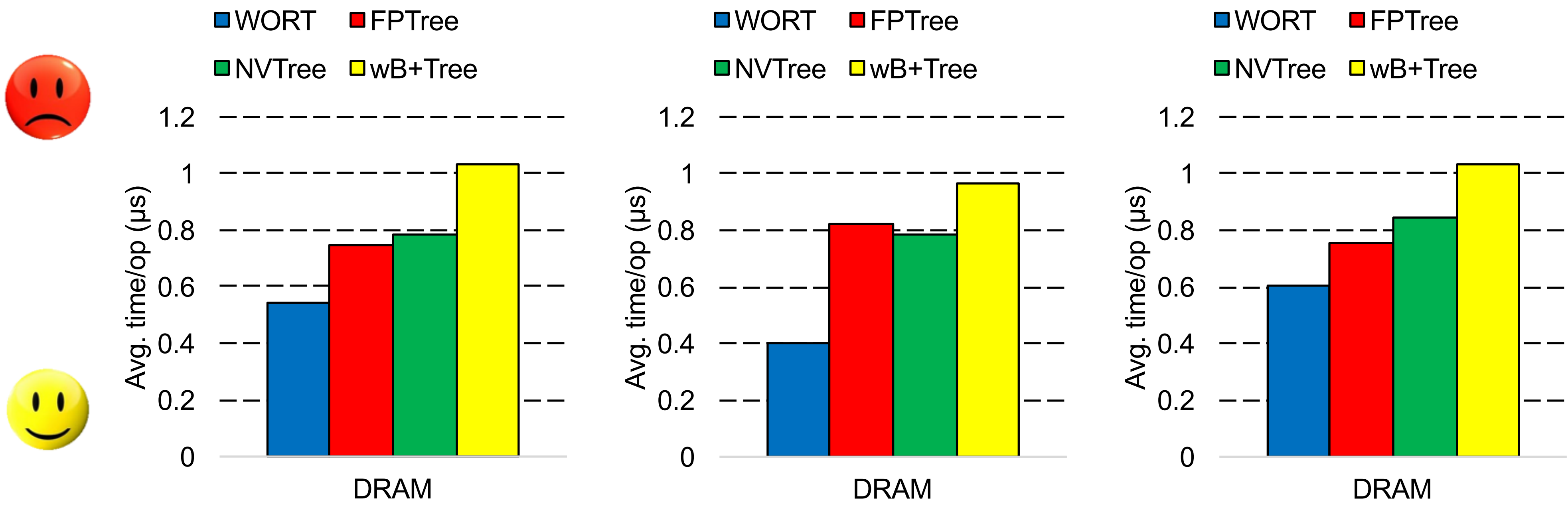
Evaluation

- **CLFLUSH count per operation**
  - B+Tree variants incur more cache flush instructions



Evaluation

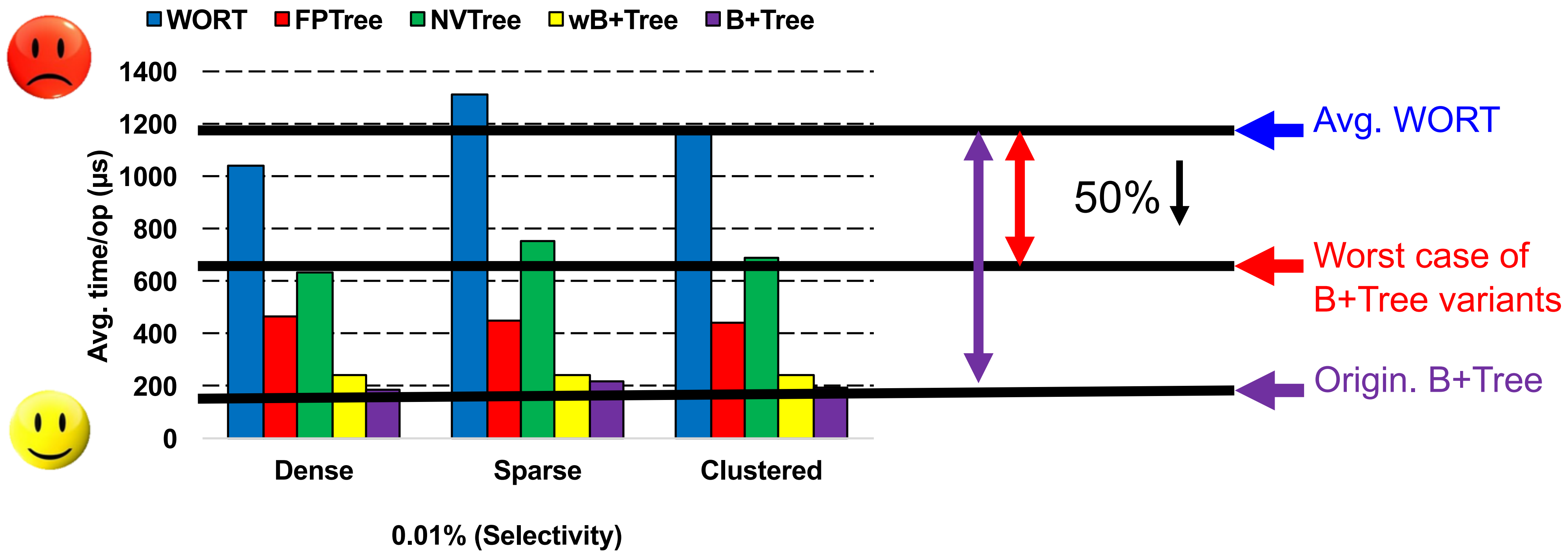
- **Search performance**
  - WORT always perform better than B+Tree variants



(a) Dense **(b) Sparse** **(c) Clustered**

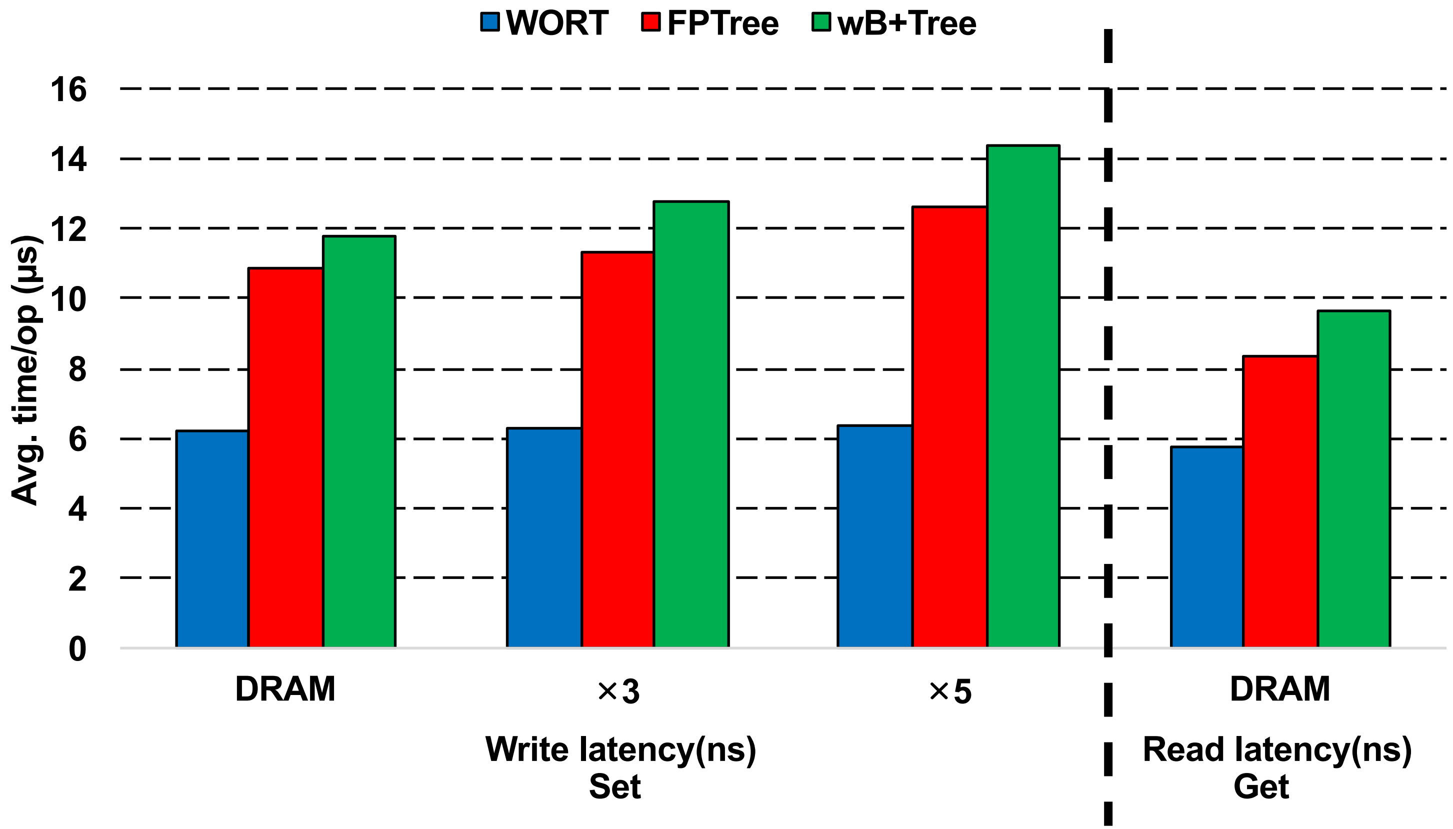
Evaluation

- **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



Evaluation

- **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

Moral of the Story

- **What we take for granted: all algorithms and data structures may be worth a revisit**
  - Failure-Atomic Slotted Paging for Persistent Memory
    - ASPLOS 2017
  - Soft Updates Made Simple and FAST
    - ATC 2017
  - A Write-friendly Hashing Scheme for Non-volatile Memory Systems
    - MSST 2017
- **Change in environment ➔ Change in assumption?**
- **Fault Model?**


Q&A