μTree: a Persistent B$^+$-Tree with Low Tail Latency

Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, Jiwu Shu

Tsinghua University

http://storage.cs.tsinghua.edu.cn

Contributions

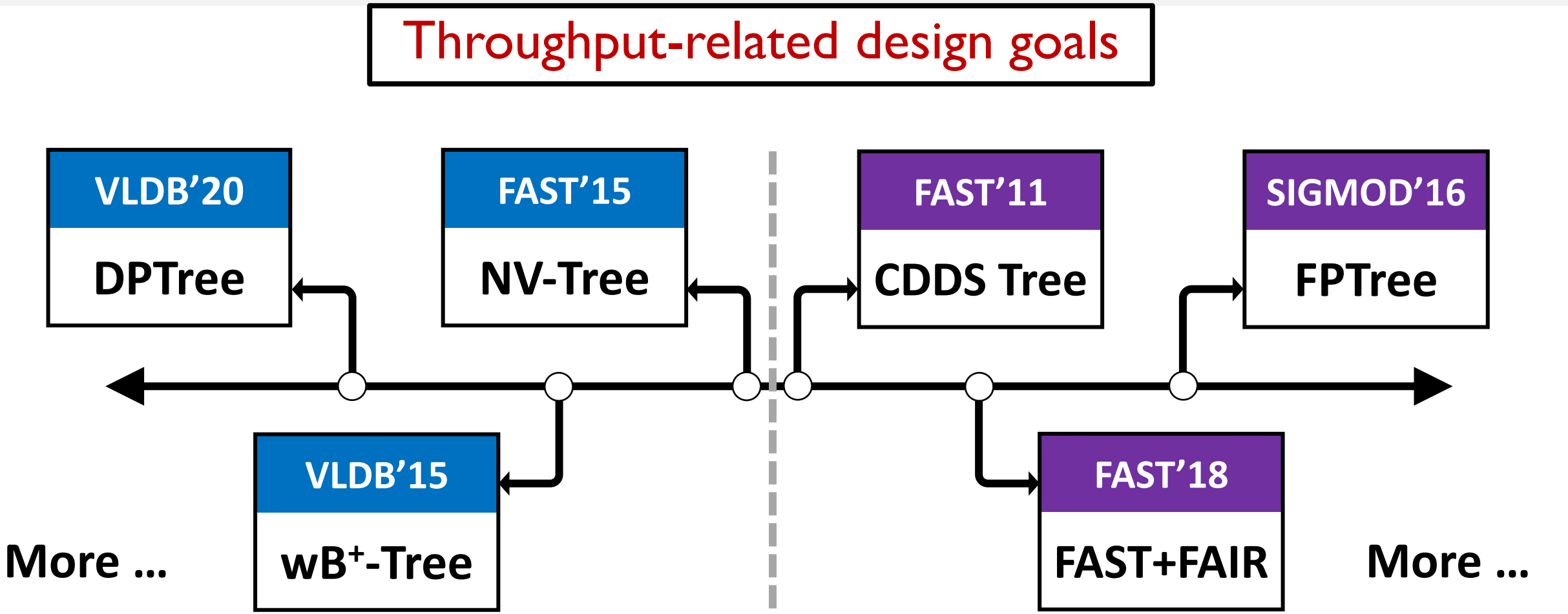
(I) An empirical study: high latency spikes of index structures, especially on persistent memory.

- FAST+FAIR exhibits a 99p-ile latency of 60 µs, 600× higher than PM latency.
- Internal structural refinement operations (SROs)
- Interference overhead between concurrent threads

(II) µTree: improve tail latencies of persistent b+-trees

- Incorporates a ***shadow list layer*** underneath the b+-tree;
- Proposes the ***Coordinated concurrency control***
- Achieves a 99p-ile latency that is one-order of magnitude lower, and 2.8 - 4.7x higher throughput.

Persistent memory data structures



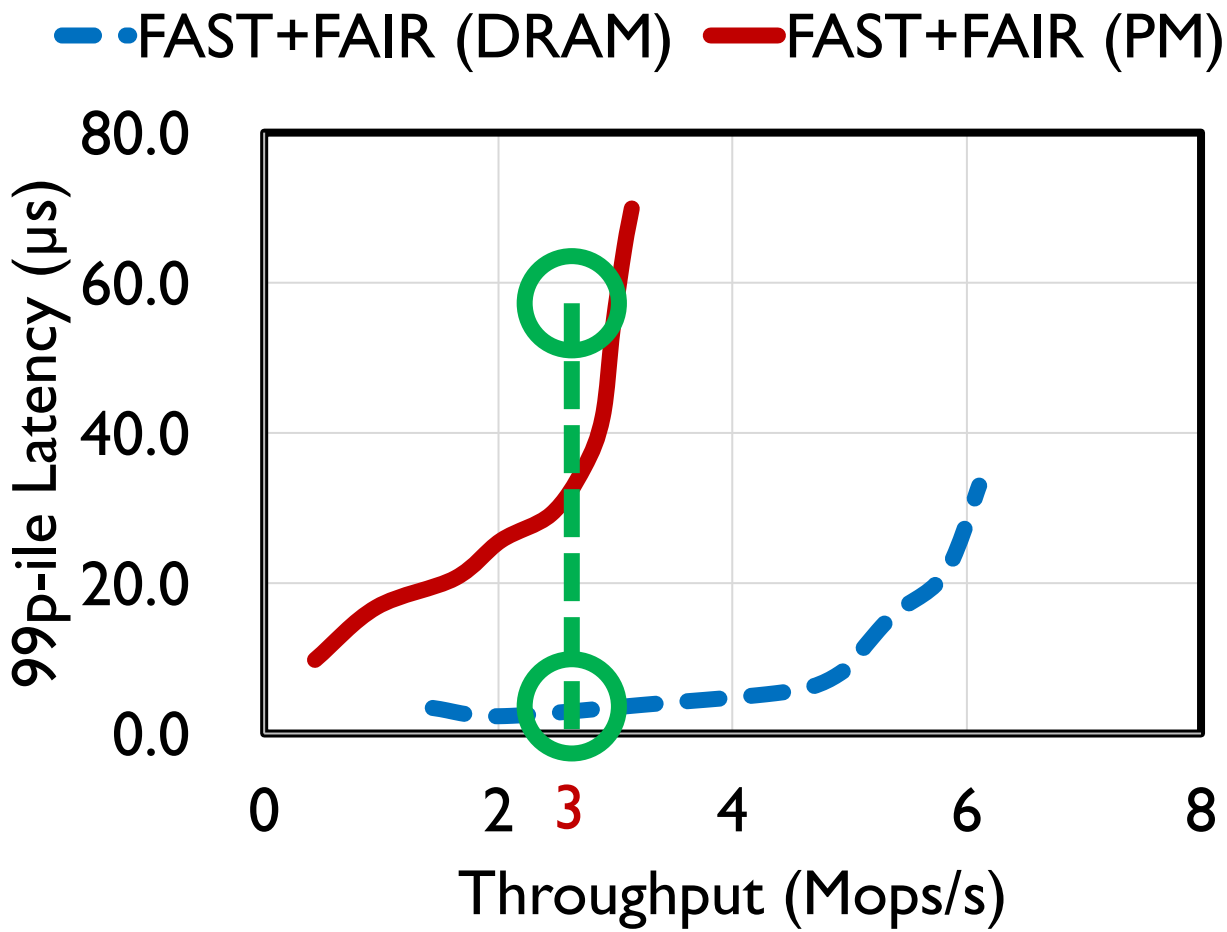
Tail latency matters for datacenter workloads

1% slow sheep

99% fast sheep

- User-perceived latency: determined by the slowest sheep (i.e., back-end node)
- Optimize tail latency from different layer of OS: queue mgmt., core scheduling, etc.

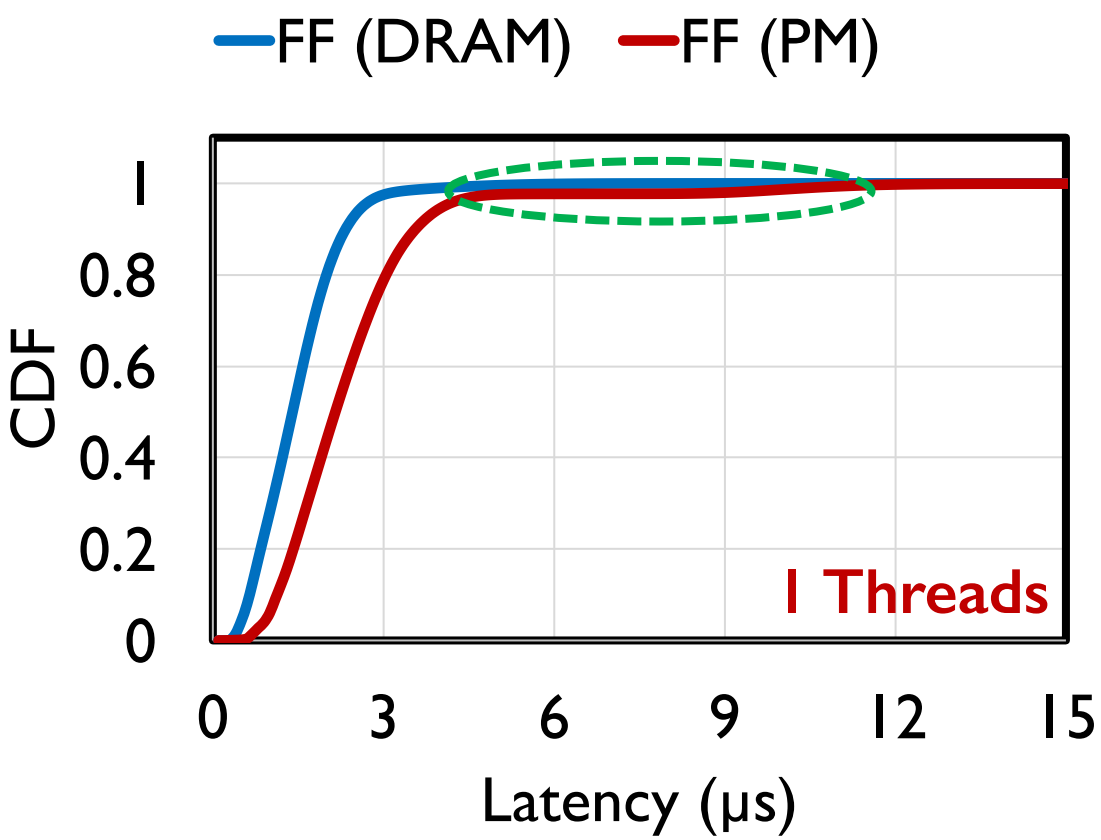
Tail latency problem in persistent b+-tree (I)



FAST+FAIR

- Highly optimized with lock-free designs and avoids the logging overhead.
- FF (DRAM): places data in DRAM and removes all flush ops.
- For a target load running at 3 Mops/s: FF(PM)'s 99p latency is almost 60µs --- **20x** higher than that of FF(DRAM), **600x** higher than PM latency.

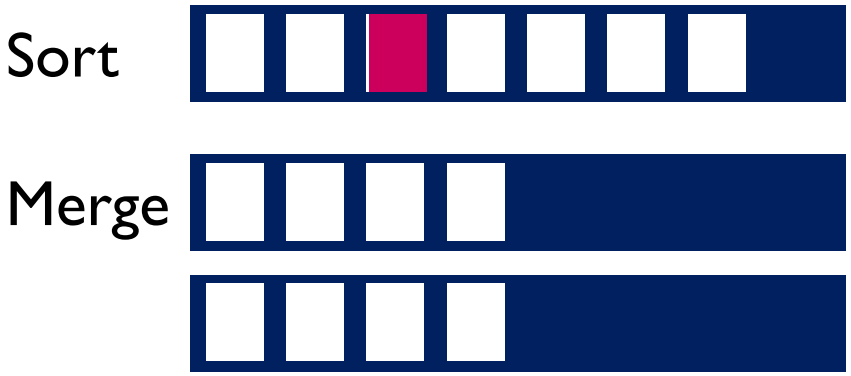
Tail latency problem in persistent b+-tree (II)



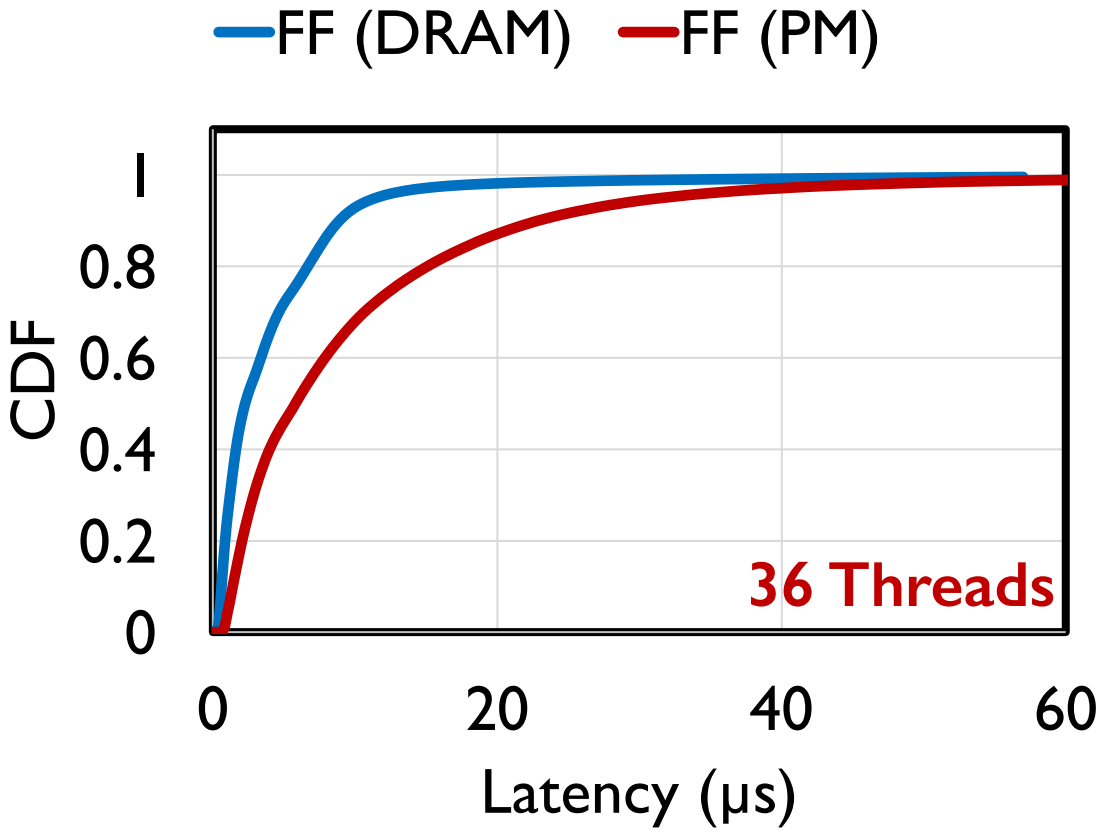
(µs)	Median	90p	99p
DRAM	1.4	2.4	4
PM	2.2	3.6	10.5

Structural Refinement Operations (SRO)

- Sort, split, merge operations
- SROs incur higher data movement overhead (i.e., higher latencies)
- SROs only occur in some of PUT/DEL operations
- PUT/DEL operations that contain SROs typically appear at the tail of the latency distribution



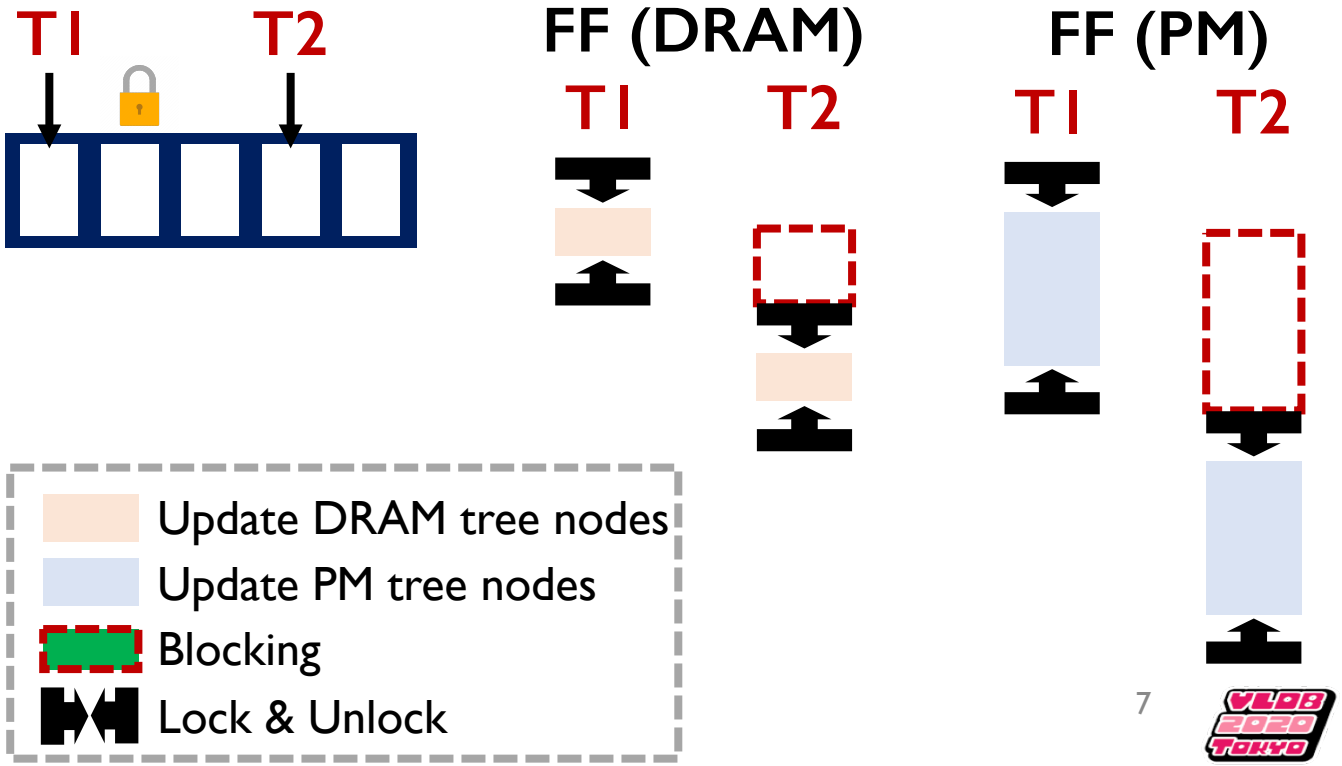
Tail latency problem in persistent b+-tree (III)



(µs)	Median	90p	99p
DRAM	2.3	9	34.2
PM	5.9	22.9	63.1

Interference between concurrent threads

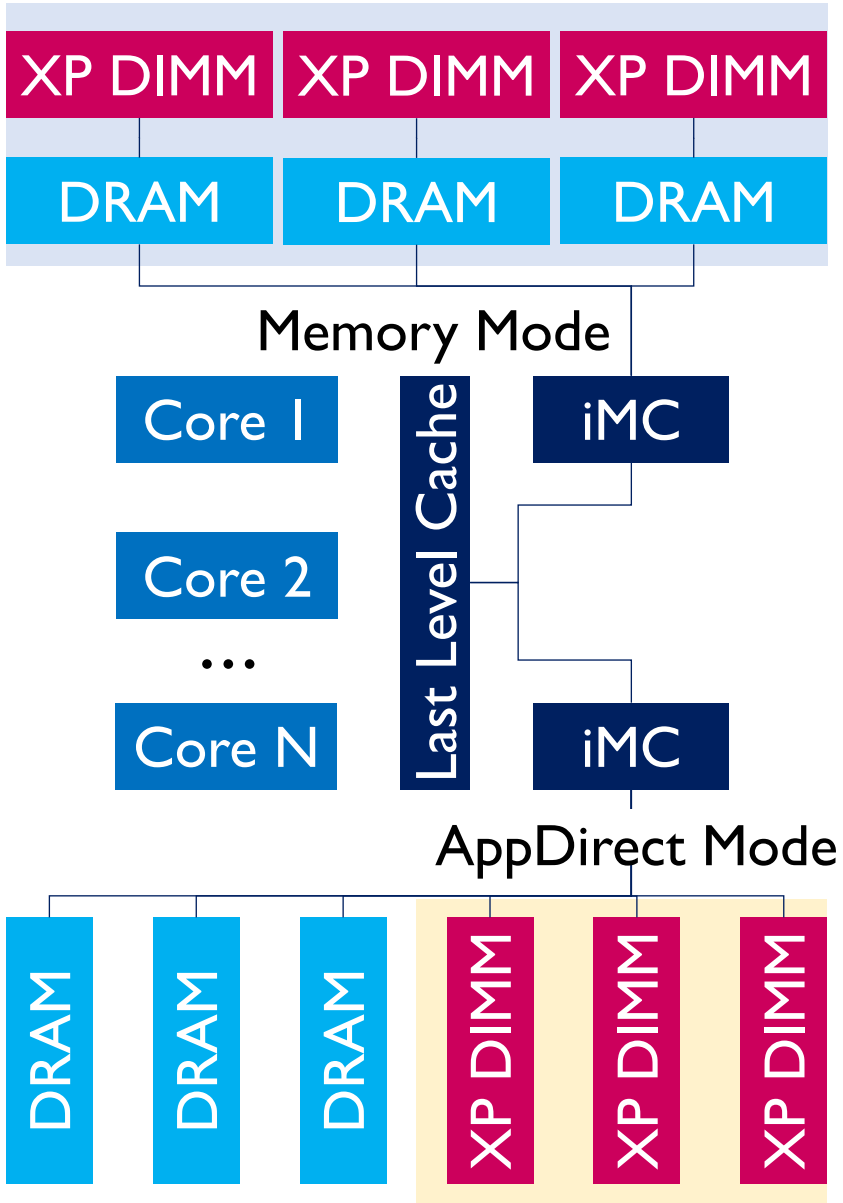
- per-node lock leads to higher delaying in PM when two locking ops conflict.

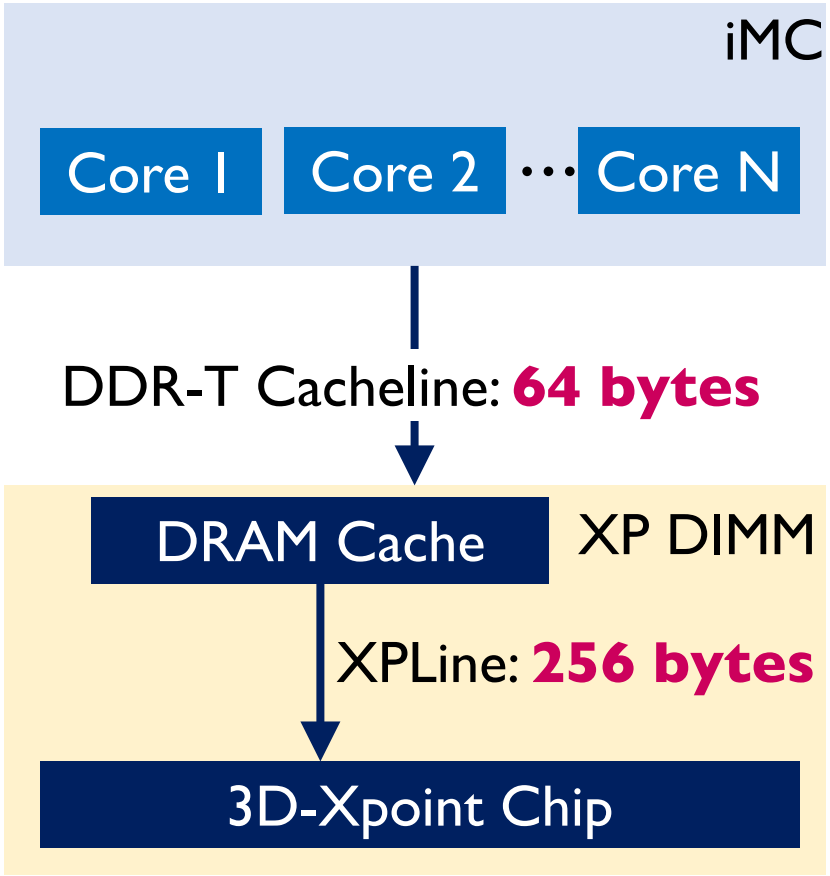


Outline

- Introduction
- Optane DC Persistent Memory Module
- µTree: a Persistent B^+-Tree with Low Tail Latency
- Results
- Summary & Conclusion

Optane DC Persistent Memory Module



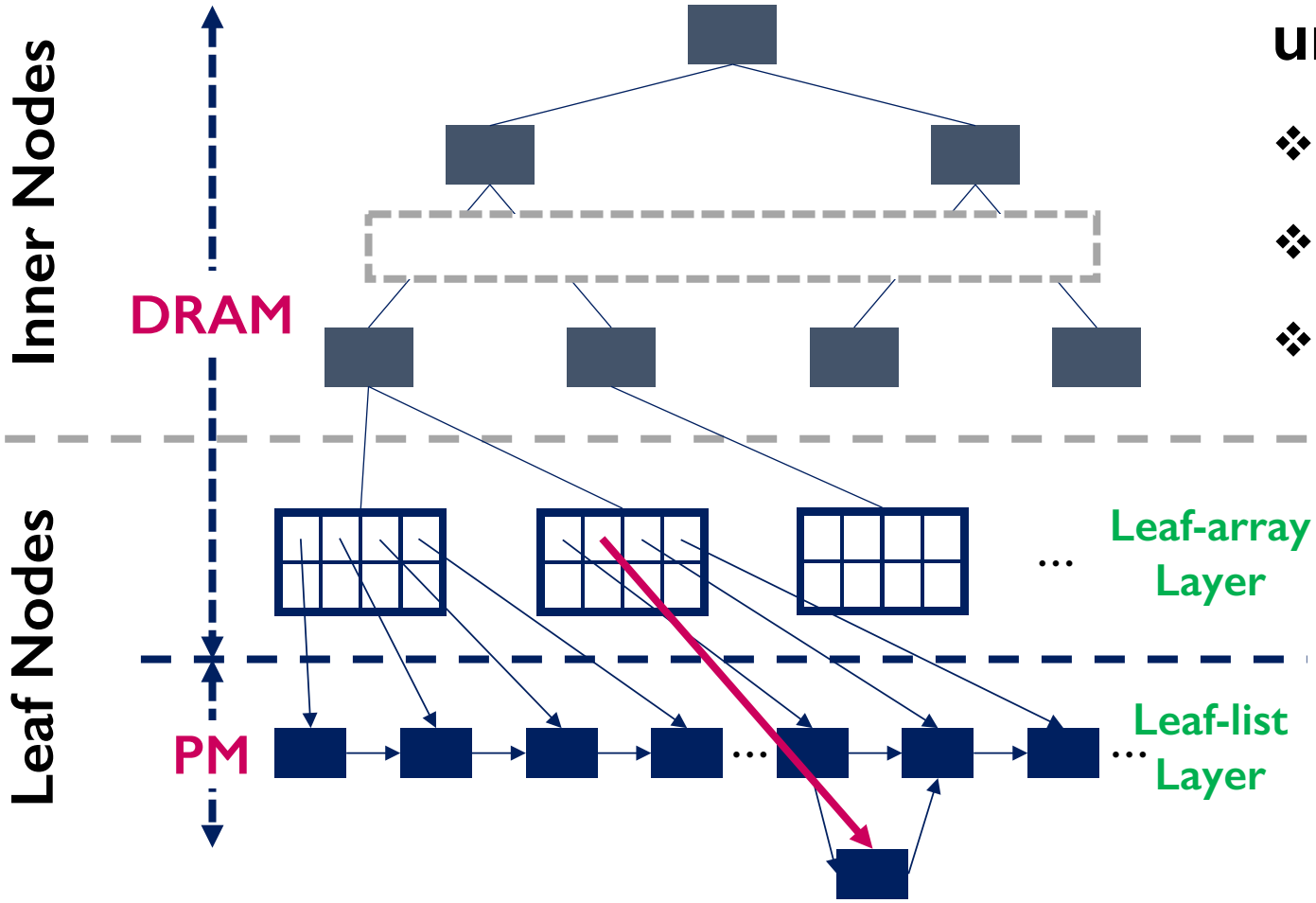


Images are reshaped from "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory", FAST'20

Outline

- Introduction
- Optane DC Persistent Memory Module
- µTree: a Persistent B^+-Tree with Low Tail Latency
- Results
- Summary & Conclusion

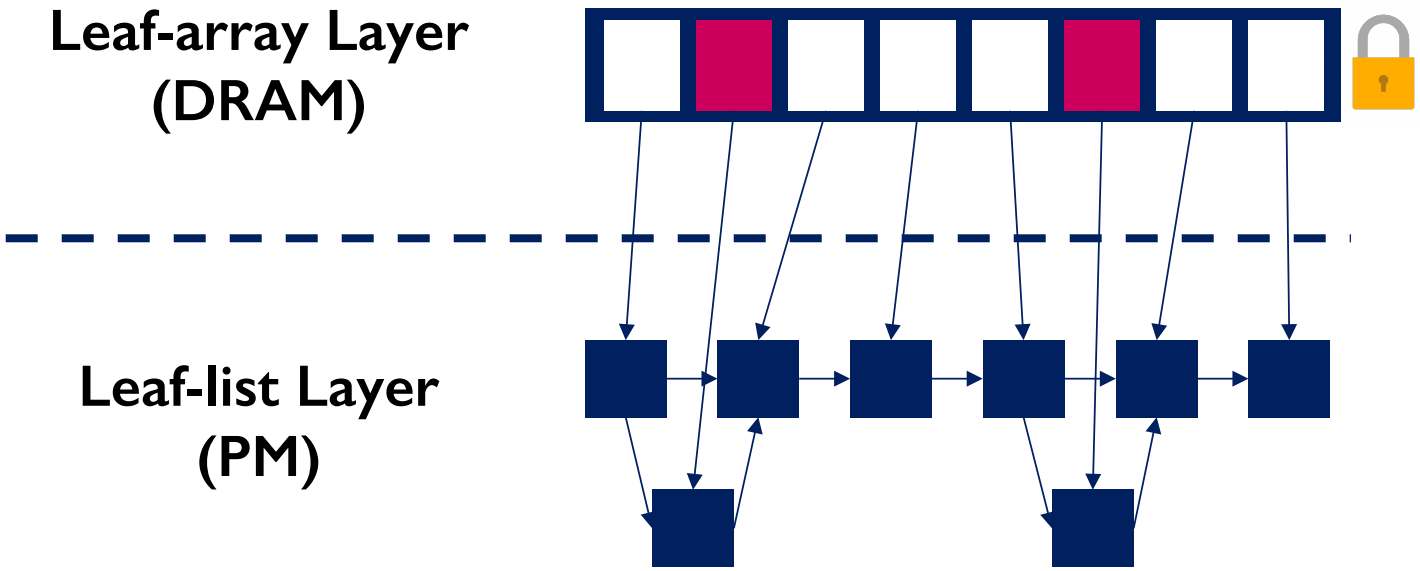
Architecture of µTree

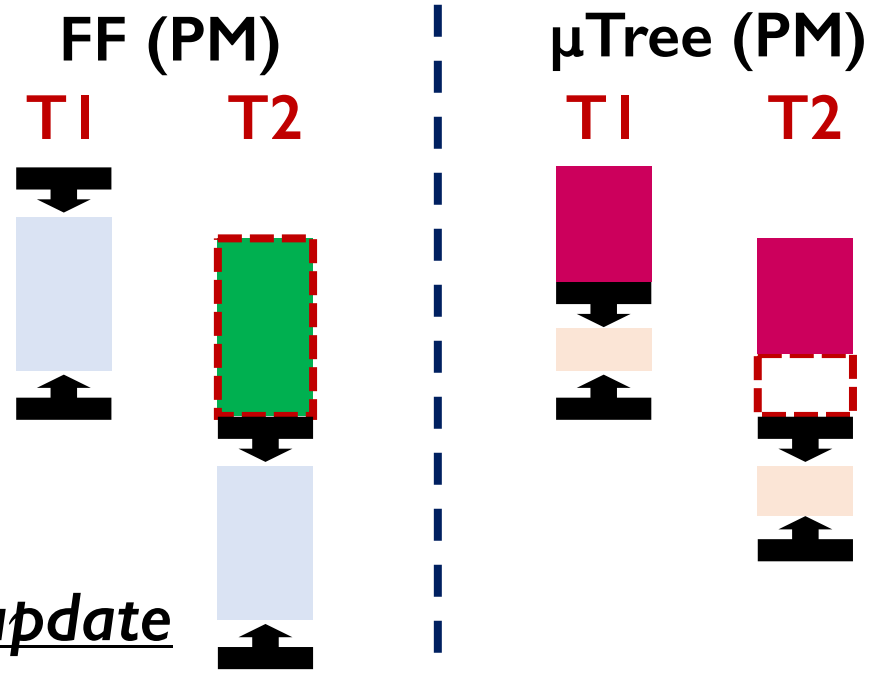


Core idea: add a *shadow list layer* underneath the tree leaf nodes

- Leaf node: **array layer** + **list layer**
- DRAM: Tree inner nodes & leaf-array nodes
- PM: Leaf-list nodes
  - Examples:
    - **PUT**: list layer ⇒ array layer
    - **GET**: array layer ⇒ list layer
- Insights are two-fold:
  - Fast query with the volatile b+-tree: O(logn) & good cache locality
  - Never perform SROs in PM: leaf-list layer does not require SROs

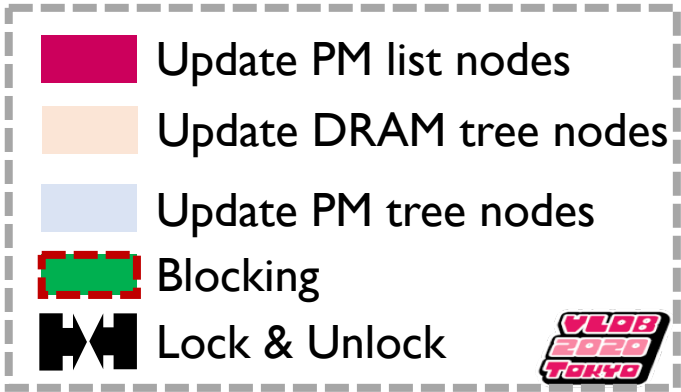
Coordinated Concurrency Control



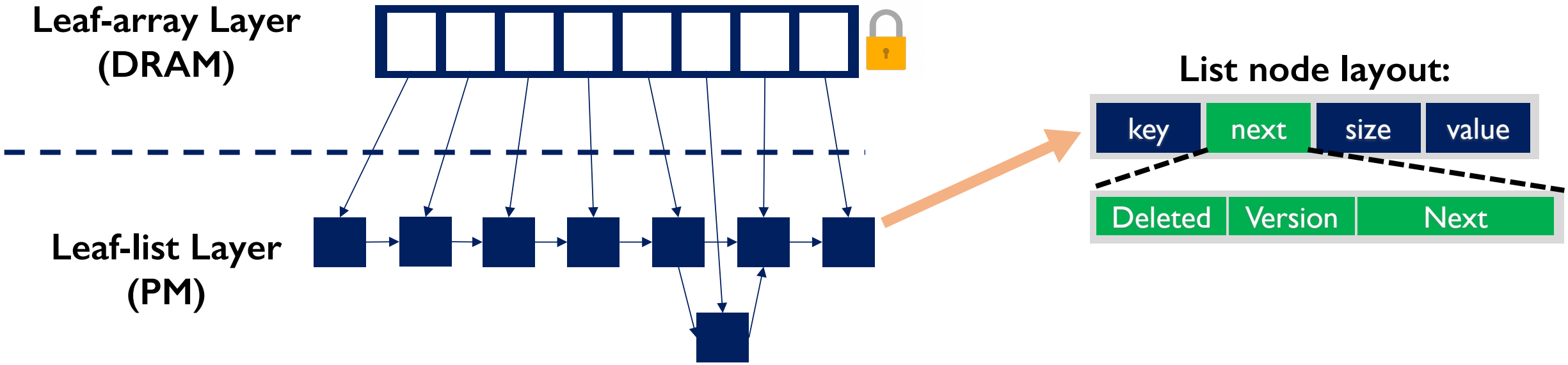


Put-Put Conflicts: ***atomic instructions to the list layer*** **+** ***update array layer via locks***

- I. Link a list node in the list layer via atomic instructions (fine-grained CC);
- II. Acquire the lock & insert a slot in the array layer to point to the list node.
- Insight:
  - PM update operations are moved out of the locking path
  - Reduce interference overhead



Coordinated Concurrency Control



Put-Get Conflicts: ***embed a version bit in the next pointer of the list layer.***

- I. Put operation: toggle the *Version* bit before actually updating an item;
- II. Get operations are executed in the opposite direction:
  - Locate a key-value pair by first find the slot in the array layer, and get the target list node with the pointer
  - Get: array layer ➔ list layer ⟺ Put: list layer ➔ array layer
- Guarantee: Visible items are always persisted (avoid dirty reads)

More design details: Check our paper

Anomalies in *coordinated concurrency control*:

- CAS failures & Put-Del conflicts.

Recovery of the volatile tree layer

- A multi-threaded recovery mechanism is used for fast recovery.

Range queries

- Probe in the list layer directly.

Memory allocation consistency

- µTree adopts an epoch-based approach.

Outline

- Introduction
- Optane DC Persistent Memory Module
- µTree: a Persistent B+-Tree with Low Tail Latency
- Results
- Summary & Conclusion

Experimental Setup

Hardware Platform

CPU	2 Xeon Gold 6240m CPUs (**36 physical cores**)
DRAM	192 GB (32GB/DIMM)
PM	6 Optane DCPMMs (**1.5 TB, 256 GB/DIMM**),
Operating System	Ubuntu 18.04.3 LTS, Linux 4.15.0

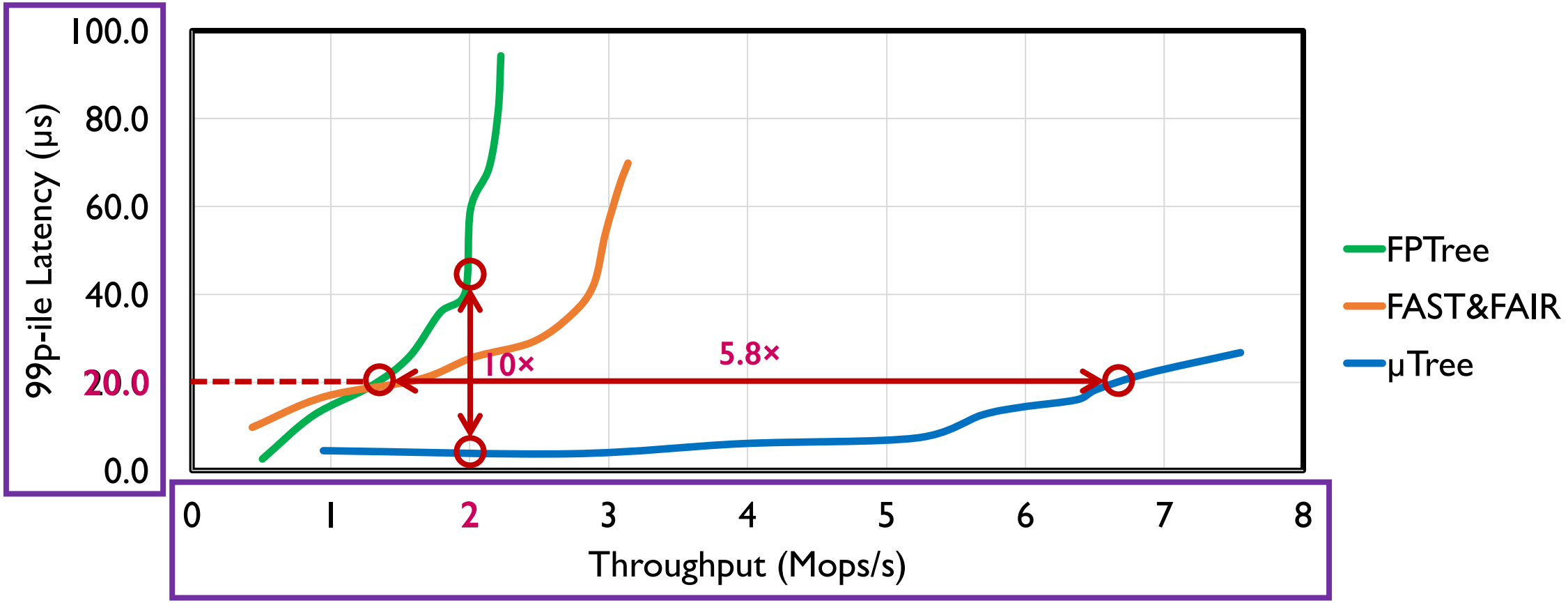
Compared Systems

FPTree	Non-leaf nodes are placed in DRAM; HTM + Locking for CC
FAST&FAIR	All nodes are placed in PM; lock-free reads; no logging overhead.

Workloads

- YCSB (varying r/w ratio, item size, skewness, etc.)
- Redis (a multi-threaded version)

Micro-benchmark: YCSB



- For a target load running at 2 Mops/s, µTree delivers one order magnitude lower 99th percentile latency;
- For a target tail latency of 20 µs, µTree achieves 5.8x higher throughput.

Summary & Conclusion

- Recent work implement PM-aware data structures by improving their throughput-related performance. Scant attention has been paid to tail latency.
  - Overhead of structural refinement operations (SROs);
  - Overhead of cross-thread interference.
- We propose µTree that takes **tail latency** into consideration.
- Key insight: **a shadow list layer** to (1) avoid SRO overheads in PM, and (2) support fine-grained concurrency control
- µTree achieves a 99p-ile latency that is one order magnitude lower, and improves throughput by **1.8 – 3.7 times**.

Thanks & QA

Tsinghua University

http://storage.cs.tsinghua.edu.cn