SIDLE: Tree-structure Aware Indexes for CXL-based Heterogeneous Memory

Haoru Zhao
Shanghai Jiao Tong University
zhaohaoru@sjtu.edu.cn

Mingkai Dong
Shanghai Jiao Tong University
mingkaidong@sjtu.edu.cn

Fangnuo Wu
Shanghai Jiao Tong University
wufangnuo@sjtu.edu.cn

Haibo Chen
Shanghai Jiao Tong University
haibochen@sjtu.edu.cn

ABSTRACT

On heterogeneous memory (HM) where fast memory (i.e., CPU-attached DRAM) and slow memory (e.g., remote NUMA memory, RDMA-connected memory, Persistent Memory (PM)) coexist, optimizing the placement of tree-structure indexes (e.g., B+ tree) is crucial to achieving high performance while enjoying memory expansion. Nowadays, CXL-based heterogeneous memory (CXL-HM) is emerging due to its high efficiency and memory semantics. Prior tree-structure index placement schemes for HM cannot effectively boost performance on CXL-HM, as they fail to adapt to the changes in hardware characteristics and semantics. Additionally, existing CXL-HM page-level data placement schemes are not efficient for tree-structure indexes due to the granularity mismatch between the tree nodes and the page.

In this paper, we argue for a *CXL native, tree-structure aware* data placement scheme to optimize tree-structure indexes on CXL-HM. Our key insight is that *the placement of tree-structure indexes on CXL-HM should match the tree's inherent characteristics with CXL-HM features*. We present SIDLE, a tree-structure aware, node-grained data placement scheme for tree-structure indexes on CXL-HM. With SIDLE, developers can easily adapt existing tree-structure indexes to CXL-HM. We have integrated the B+ tree and radix tree with SIDLE to demonstrate its effectiveness. Evaluations show that SIDLE improves throughput by up to 71% and reduces P99 latency by up to 81% compared with state-of-the-art data placement schemes (e.g., MEMTIS) and HM-optimized tree-structure indexes (e.g., PACTree) in YCSB and real-world workloads.

PVLDB Reference Format:
Haoru Zhao, Mingkai Dong, Fangnuo Wu, and Haibo Chen. Tree-structure Aware Indexes for CXL-based Heterogeneous Memory. PVLDB, 19(7): 1499 - 1515, 2026.
doi:10.14778/3801059.3801065

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/sidle-project/sidle.

1 INTRODUCTION

In-memory tree-structure indexes [37, 68, 81, 82, 87, 101, 113, 118, 135, 139, 148] (e.g., B+ tree) are the cornerstone of in-memory databases [15–19, 127, 150], storage engines [83, 90] and big data analytics [20–22, 31, 73]. Their efficiency is crucial for overall system performance [37, 71, 148, 150]. With the growing data scale in datacenters, the memory demand of tree-structure indexes increases rapidly [30, 32, 91, 114, 148, 151, 152]. To expand memory capacity and enhance memory elasticity, heterogeneous memory (HM) has emerged. Many efforts [39, 40, 45, 78, 89, 93, 96, 98, 99, 104, 108, 134, 144, 149, 158, 160] have focused on building tree-structure indexes on HM to leverage the large memory capacity while mitigating the slow memory's impact on performance.

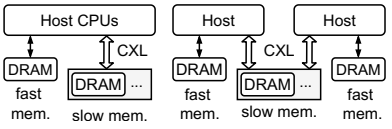
Nowadays, Compute Express Link (CXL) [92] becomes a promising HM solution to expand memory capacity. For example, Samsung CMM-B [13] scales memory capacity up to 24 TB with CXL memory pool. Due to CXL's cache-coherent memory semantics (i.e., CXL provides cache-coherent memory in the same address space as fast memory), large tree-structure indexes designed for fast memory can easily adapt to CXL-HM, where fast memory (i.e., CPU-attached DRAM) and CXL-attached memory coexist.

However, placing these tree-structure indexes designed for fast memory directly onto CXL-HM results in poor performance, as there is still a performance gap between CXL-attached memory and fast memory (latency ~2×, bandwidth ~60%, as shown in Figure 1(c)). For instance, directly placing 75% of the tree's memory usage on CXL-attached memory can lead to a performance drop of ~70% (Figure 2). Therefore, it is vital to optimize the placement of tree-structure indexes on CXL-HM to reduce performance degradation while taking advantage of the increased memory capacity.

Previous optimizations for tree-structure index placement on HM (e.g., NUMA, RDMA, PM) are inefficient for CXL-HM. Replication, partitioning, and delegation methods used in NUMA-optimized indexes [39, 40, 104] are not suitable for CXL-HM because CXL-attached memory lacks local processors like NUMA nodes. Indexes optimized for RDMA [96, 98, 99, 132, 134, 160] maintain a cache for accessed data in fast memory. However, the narrowed performance gap between slow and fast memory in CXL-HM makes cache maintenance overhead outweigh the benefits of caching. In indexes optimized for PM [45, 78, 93, 108, 144, 149, 158], placing the internal nodes of trees in fast memory is commonly used to mitigate slow memory's limited write bandwidth and higher latency. For CXL-HM, this static approach is insufficient because it neglects the capacity limit of fast memory, leading to fast memory exhaustion or underutilization, and fails to adapt to dynamic workloads.

Moreover, existing CXL-HM data placement schemes [79, 103, 130, 140] are also inefficient for tree-structure indexes. As these schemes manage data at page level while tree nodes are typically smaller, this granularity mismatch leads to inaccurate hotness detection, increased migration overhead, and suboptimal data placement.

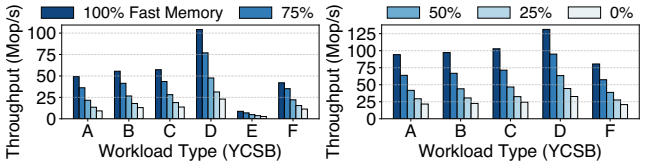
Proceedings of the VLDB Endowment, Vol. 19, No. 7 ISSN 2150-8097.
doi:10.14778/3801059.3801065



	B/W(MB/s)	Lat. (ns)
Fast	33095.3	111.2
Slow (CXL)	21724.3	276.9

(a) CXL 1.0/1.1 (b) CXL 2.0/3.0 (c) Performance

Figure 1: CXL-based heterogeneous memory. ***(a,b) System architecture. (c) Fast and slow memory latency and bandwidth tested by Intel MLC [5] on our CXL1.1 platform.***



(a) Masstree (b) ART

Figure 2: Throughput (tput.) of indexes with different fast memory usage ratios in YCSB [47] benchmark.

Our key insight is that *the placement of tree-structure indexes on CXL-HM should match the tree's inherent characteristics with the features of CXL-HM.* We identify three key characteristics of trees: (1) *node granularity*: node is the unit of data usage and management in trees; (2) *layered hotness*: nodes in upper levels are accessed more frequently (i.e., hot) than those in lower levels; (3) *path-style access*: the access granularity of trees is the path (i.e., all nodes from the root to a leaf[1]), and nodes along paths to hot leaf nodes (i.e., hot path) contribute to considerable memory access.

Therefore, on CXL-HM we should focus on: (1) Match the node granularity with CXL-HM's cache-coherent memory semantics. Instead of caching, we should store hot nodes in fast memory and less frequently accessed (i.e., cold) nodes in slow memory, track node hotness, and migrate nodes between fast and slow memory based on the hotness for dynamic workloads. (2) Match the layered hotness and path-style access with the performance differences between fast and slow memory by prioritizing upper-level nodes (i.e., *layer principle*) and hot paths (i.e., *path principle*) in fast memory.

We design SIDLE, a node-grained, tree-structure aware data placement scheme to optimize tree-structure indexes on CXL-HM. First, SIDLE introduces *leaf-centric access tracking*, capturing the hotness of all access paths while minimizing interference with tree operations by only tracking leaf nodes. Second, SIDLE incorporates *structure-aware migration*, which considers node relationships—rather than migrating each node independently—to keep upper-level nodes in fast memory and achieve the quick promotion (i.e., moving to fast memory) of the entire hot path. Finally, SIDLE proposes *hyper watermark mechanism* that holistically coordinates system behaviors (i.e., node allocation and migration) based on real-time fast memory usage rather than static thresholds, thereby ensuring stable high performance for dynamic workloads.

To demonstrate SIDLE's effectiveness and generality, we integrate SIDLE with two widely used tree-structure indexes—B+ tree [101] and radix tree [81, 82]—with less than 3% trees' internal code modification. Evaluation on a real CXL platform with synthetic workloads shows that SIDLE achieves up to 133% higher throughput and 67% lower P99 latency than the SOTA data placement scheme MEMTIS [79]. For YCSB [47], SIDLE achieves up to 71% and 60% higher throughput than MEMTIS and optimized HM index (optimized PACTree [70]). For real-world workloads [4], SIDLE achieves up to 66% higher throughput and 81% lower P99 latency than MEMTIS, with 44% higher throughput and 63% lower P99 latency than the optimized PACTree.

In summary, we make the following contributions:

- **Analyses.** We thoroughly analyze existing solutions for the placement of tree-structure index on CXL-HM and propose that the key to efficient placement lies in matching the tree's inherent characteristics with CXL-HM features.
- **Design.** We design SIDLE, an efficient and generic node-grained data placement scheme for tree-structure indexes on CXL-HM.
- **Evaluation.** We integrate SIDLE with B+ tree and radix tree. Evaluation on a real CXL platform with various workloads demonstrates the advantages of SIDLE.

2 BACKGROUND AND MOTIVATION

2.1 CXL-based Heterogeneous Memory

CXL [49, 92] is an emerging interconnect technology notable for its memory expansion capability, cache-coherent memory semantics, and high performance. As illustrated in Figure 1(a), CXL-HM with CXL 1.0/1.1 [9] allows host CPUs to access CXL-attached memory with cache-coherent load/store. CXL-HM with CXL 2.0/3.0 [10, 11] (Figure 1(b)) further allows the hosts to expand the memory by CXL memory pool, enhancing the flexibility of memory expansion.

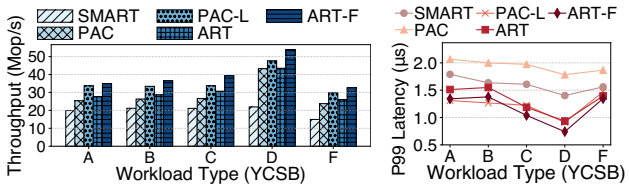
2.2 Tree-structure Indexes on CXL-HM

Tree-structure indexes are the main memory consumers of various in-memory database [37, 148, 150]. The growing demand for in-memory processing is pushing memory consumption by these indexes into terabytes and more [30, 32, 91], necessitating HM adoption. CXL-HM is a promising solution because it provides a unified address space of fast and CXL memory, allowing existing tree-structure indexes for fast memory to be effortlessly migrated to CXL-HM, avoiding the high cost of redesigning new indexes.

Figure 1(c) shows that there is still a performance gap between fast and slow memory in CXL-HM: slow memory has higher latency (~2×) and lower bandwidth (~60%) than fast memory on our CXL1.1 platform. This gap will be further widened with the CXL-switch introduction in CXL 2.0 [66]. Thus, directly porting tree-structure indexes designed for fast memory to CXL-HM leads to poor performance. We illustrate this via Masstree [101] and adaptive radix tree (ART) [81, 82], two typical tree-structure indexes. The setup of the following experiments is the same as our evaluation (§6.1). As shown in Figure 2[2], with the default weighted memory allocation policy [136] (i.e., allocating a specified percentage of memory to CXL in N : M interleave way), the performance of both indexes drops by ~70% with 25% fast memory usage compared to 100% fast memory across all workloads. Thus, *optimizing the placement of tree-structure indexes on CXL-HM is essential for better performance.*

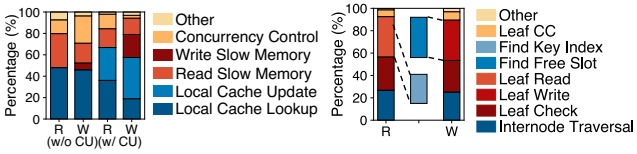
[1]We mainly focus on tree-structure indexes that store values in leaves, the most common form of tree-structure indexes [81].

[2]We don't test workload E for ART because its source code does not implement the scan operation.



(a) Throughput (b) P99 Latency

Figure 3: Comparison of SMART, PACTree, and ART on CXL-HM. *PAC stands for PACTree. All indexes have a fast memory usage ratio of 20%, except ART-F at 40%. PAC, ART, and ART-F use the same memory allocation policy as Figure 2.*



(a) SMART (b) PACTree

Figure 4: Performance breakdown per operation type. *(w/o CU) & (w/ CU) means without/with local cache update. Leaf CC is short for Leaf Concurrency Control.*

2.3 Issues of Optimized HM-Indexes on CXL-HM

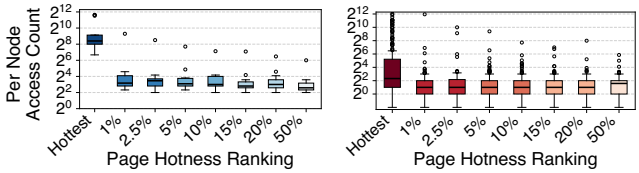
In this section, we study existing tree-structure indexes optimized for other types of HM (e.g., NUMA, RDMA-based HM, PM-based HM) and data placement schemes for CXL-HM to demonstrate that:

- Prior optimizations for tree-structure index placement on HM are either impractical or inefficient on CXL-HM due to CXL-HM's unique hardware characteristics. We compare the original ART, the RDMA-optimized ART (SMART [99]), and the PM-optimized ART (PACTree [70]) using YCSB. Figure 3 shows that *the original ART outperforms SMART and PACTree on CXL-HM.*
- Applying existing CXL-HM data placement schemes to tree-structure indexes is also inefficient due to the granularity mismatch between tree nodes and pages.

NUMA-Index. Indexes optimized for NUMA reduce cross-node communication and minimize accesses to remote memory via replication [40, 48, 111], partitioning [104], or delegation [39, 122]. These methods are not applicable to CXL-HM since *CXL-attached memory lacks local processors for data processing, unlike the NUMA node.*

RDMA-Index Case Study (SMART). SMART optimizes ART's data placement by maintaining a fast memory cache for accessed internal nodes (i.e., local cache), which is a common optimization in RDMA-Indexes [89, 98, 99, 134]. Figure 3(a) shows that SMART's throughput is only 50%–74% of ART. Our profiling (Figure 4(a)) reveals that the primary overhead (~50%) comes from the local cache. *Since the latency gap between CXL memory and fast memory is much smaller than that of RDMA-connected memory [97], the costs of cache maintenance and querying outweigh the benefits of reducing slow memory accesses.* Specifically, while the local cache significantly reduces access to the slow memory (nearly all internal node accesses hit the cache in the Zipfian read/update-only workloads), querying the local cache incurs more overhead than direct tree traversal due to the additional cache miss checks and cache eviction management.

PM-Index Case Study (PACTree). PACTree optimizes ART's data placement by using fat (64 entries), unordered leaf nodes, which



(a) Masstree (b) ART

Figure 5: Node access count distribution across pages of varying hotness, covering the page with highest access frequency (Hottest) and those ranked top 1%–50% by access frequency.

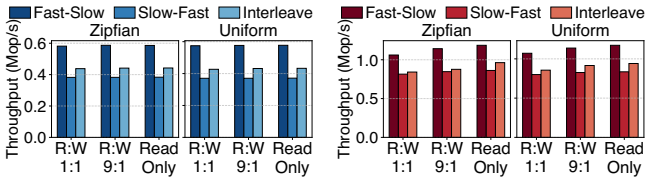
is a common optimization [44, 45, 93, 100, 108, 144, 149, 155] in PM-indexes, along with asynchronous tree structural modification (i.e., async SMO). Figure 3 shows that PACTree's throughput is only 90% of ART, and its tail latency is 1.3~1.9× of ART. Our breakdown (Figure 4(b)) indicates that the main source of performance overhead (~61%) is the fat, unordered leaf and async SMO. *These optimizations incur additional overhead in CXL-HM because CXL-attached DRAM does not suffer from the read-write asymmetry present in PM [114, 120], and the high allocation costs associated with persistence [70, 100].* While fat, unordered leaves reduce write amplification and allocation overhead, they impose significant search overhead, with nearly 99% of the time in the leaf write operation spent finding a free position. Additionally, while async SMO decouples the write overhead of slow memory from the critical path, it introduces substantial overhead—particularly harming tail latency—through leaf checking, which ensures the correctness of async SMO.

Another common placement optimization in PM-indexes is to place only leaves in slow memory and keep internal nodes in fast memory (a.k.a., *fast memory internode*) [45, 93, 108, 144, 149, 158]. We apply this technique to PACTree (PAC-L in Figure 3). While it can improve PACTree's performance on CXL-HM, this static placement strategy is insufficient as: (1) It neglects fast memory capacity, which can result in fast memory underutilization or exhaustion. For example, ART outperforms PAC-L when more fast memory exists (ART-F in Figure 3), and PAC-L's performance drops by ~17% when fast memory is full. (2) It ignores workload patterns, forcing every request to access slow-memory leaves.

Applying CXL-HM Data Placement Schemes. Existing data placement schemes [79, 103, 130, 140, 159] for dynamic data placement (i.e., migrating data based on workload demands) on CXL-HM manage data at page granularity (e.g., 4 KiB), such as MEMTIS [79]. However, since tree nodes are typically smaller than pages, this granularity mismatch can lead to inaccurate hotness detection. As shown in Figure 5, even for the top 1% of frequently accessed pages, the majority of nodes remain cold—the upper quartile of nodes' access count is just 16 and 4 for Masstree and ART, respectively. Additionally, the hottest node on each page contributes 82% and 85% of total accesses, respectively. Therefore, page-level migration may misplace cold nodes in fast memory, causing suboptimal data placement and increasing migration overhead.

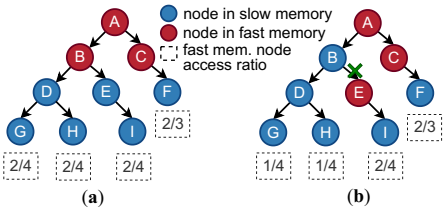
3 DESIGN PRINCIPLES AND CHALLENGES

The above analysis indicates that compared to RDMA-connected memory and PM, CXL-attached memory behaves more like a *slow DRAM*: Its characteristics are largely consistent with CPU-attached DRAM except for slightly lower performance. Therefore, the key



(a) B+ tree (b) Radix tree

Figure 6: Throughput of different node placement strategies. ***The fast memory usage ratio is ~50% in all configurations.***



(a) (b)

Figure 7: Single-boundary structure. ***(a) The tree follows this structure. (b) When the tree violates this, putting E in fast memory and B in slow memory, the fast memory access ratio of the paths from A to G and A to H decreases from*** $\frac{2}{4}$ ***to*** $\frac{1}{4}$**.**

requirement for tree-structure indexes on CXL-HM is a *fine-grained, lightweight, and adaptive data placement scheme that can efficiently place frequently accessed data on the fast memory.*

Our key insight is that the placement of tree-structure indexes on CXL-HM should match the tree's inherent characteristics with the features of CXL-HM, which involves: (1) matching the tree's node-granularity data usage with CXL-HM's cache-coherent memory semantics to achieve node-grained flexible data placement; (2) matching node hotness indicated by the tree's hierarchical structure and access pattern with the performance gap between fast and slow memory to efficiently utilize fast and slow memory.

In this section, we present the principles of data placement derived from tree characteristics and the challenges in designing an efficient node-grained data placement scheme.

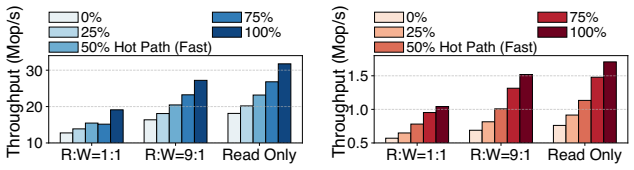
3.1 Principles from Tree Characteristics

3.1.1 Structural Characteristics. Due to the tree's branching structure and the downward access from its root, nodes at upper levels (i.e., upper nodes) are fewer but have higher access frequency than lower nodes. For instance, in a B+ tree [3] and radix tree [2] with 50 million keys, nodes in the upper half levels comprise less than 1% of total nodes but have average access frequency three orders of magnitude higher than those in the lower half levels. Thus, *upper nodes are naturally hot and well-suited for fast memory.*

We validate this observation with B+ tree and radix tree in three settings: *Fast-Slow* (upper nodes in fast memory), *Slow-Fast* (the reverse of *Fast-Slow*), and *Interleave* (levels interleaved in two memory types). Figure 6 shows *Fast-Slow* performs best, as fewer upper nodes allow more nodes along the path to be placed in fast memory. Notably, *Interleave* outperforms *Slow-Fast* but still trails *Fast-Slow*, highlighting the benefits of placing upper nodes in fast memory.

Layer Principle. It is beneficial to store the tree's upper nodes in the fast memory as many as possible.

Building on the *layer principle*, we propose a guarantee to improve the expected ratio of fast memory access in each path to target nodes: *all ancestors of a fast memory node should be placed in fast memory.* As shown in Figure 7(a), this guarantee ensures only one boundary between fast and slow memory nodes along all paths, so we term it as *single-boundary structure.* Figure 7(b) shows that when the placement of nodes B and E violates this guarantee, the paths from A to G and H lose a fast memory node, potentially degrading the performance when accessing the data of G and H.



(a) B+ tree (b) Radix tree

Figure 8: Tput. at different hot path ratios in fast memory. ***Percentages of Hot Path (Fast) indicate the proportion of hot paths in fast memory. For nodes not on Hot Path (Fast), their placement follows the Fast-Slow configuration in Figure 6.***

3.1.2 Access Characteristics. Trees are accessed from the root downwards, forming access paths from the root to target nodes (e.g., leaves in B+ trees and radix trees). In real-world workloads, where accesses are often skewed [32, 50, 54, 125, 145], *certain paths are accessed more frequently.* We identify these paths as *hot paths.*

To evaluate how hot paths can affect tree performance on CXL-HM, we test a multi-threaded B+ tree [1] and the radix tree under a skewed workload, where 90% of requests targeted 10% of the data. We vary the proportion of hot paths in fast memory and measure throughput. As shown in Figure 8, tree performance improves across all workloads with more hot paths in fast memory.

Path Principle. It is crucial to identify the hot paths and endeavor to place them in fast memory as many as possible.

3.2 Challenges

To adapt to dynamic workloads, an efficient data placement scheme should be able to (1) track node hotness and migrate nodes based on their hotness; and (2) adjust the allocation and migration strategy according to current memory usage. Thus, while these principles offer valuable guidance, several challenges still need to be addressed.

Fine-grained access tracking incurs high overhead for tree operations. Tree-structure indexes have high requirements for both latency and throughput [37, 71, 148, 150]. Prior work [42]'s per-object (i.e., per-node) access tracking introduces high overhead (over a 50% performance drop) on the critical path. This is because the tree's access pattern requires updating the access information of multiple nodes along the path for each data access. Thus, we should relax the tracking granularity from nodes to paths.

Migration cannot guarantee the single-boundary structure and may break the locality of hot paths. When using the path-grained tracking, existing migration mechanisms [53, 79, 103, 114, 130, 140] that treat each object (e.g, page, node) independently are no longer suitable due to missing per-node hotness data. Moreover, simply migrating by path is insufficient. As an internal node belongs to multiple paths, demoting it (i.e., moving to slow memory) may (1) break the *single-boundary structure* if it has fast memory children; (2) disrupt hot path locality, as it may be part of other hot paths.

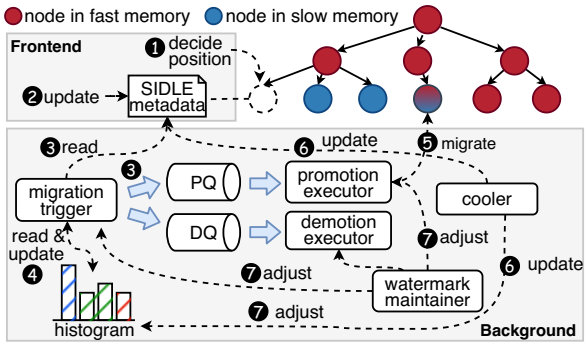


Figure 9: Architecture and interactions of SIDLE.*PQ stands for promotion queue, DQ stands for demotion queue.*

Allocation and migration require complicated control to meet the complexity of the tree structure. The complexity of trees' hierarchical structure necessitates more parameters to control allocation and migration than page-level data placement schemes. For example, migration requires two metrics for deciding the path and the number of nodes in each path that need to be migrated. Additionally, allocation needs a layer threshold to determine whether a node should be placed in fast memory based on the *layer principle*. For these parameters, using static values cannot adapt to the dynamic workloads, and manual adjustments are error-prone and labor-intensive. Moreover, compared to existing allocation and migration strategies that are either passive (e.g., demotion only when fast memory is scarce [79, 140]) or lack coordination between allocation and migration [103], we need proactive and holistic strategies to meet the performance stability requirements of the trees.

4 DESIGN OF SIDLE

To address the challenges in §3.2, we design SIDLE, an efficient and generic data placement scheme for tree-structure indexes on CXL-HM. As depicted in Figure 9, SIDLE consists of a lightweight frontend module and a background module.

The frontend module is invoked in the critical path of the tree operations, thus it is designed to be lightweight to minimize its impact on performance. It determines the initial position of a new node (❶) based on the *layer principle* while maintaining the *single-boundary structure*. It also updates the access frequency of leaf nodes (❷), laying the groundwork for hot path and cold node identification.

The background module consists of several background workers to conduct node migration (§4.2) and system adjustment (§4.3) asynchronously to optimize the fast memory usage. The *migration trigger* picks appropriate nodes for migration based on their access frequency (❸) and the distribution (i.e., the histogram) of all leaves' access frequency (❹). The *promotion executor* and *demotion executor* get nodes from the corresponding queues and execute the migration tasks (❺). The *cooler* periodically halves the nodes' access frequency (❻) to ensure the freshness of access information. To coordinate all workers, the *watermark maintainer* monitors the fast memory usage as well as adjusting system parameters and the execution of other workers to ensure efficient utilization of fast memory (❼).

To reduce the impact of the data placement scheme on tree performance (Challenge 1), we propose a lightweight node allocation and tracking scheme (§4.1). To maintain the *single-boundary structure* and localize hot paths during node migration (Challenge 2), we design the structure-aware migration procedures (§4.2). Additionally, we holistically adjust allocation and migration using the hyper watermark mechanism (§4.3) to ensure stable high performance while optimizing fast memory utilization (Challenge 3).

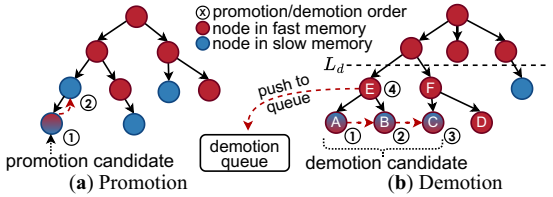


Figure 10: Promotion and demotion procedures.

4.1 Lightweight Node Allocation and Tracking

SIDLE only leaves two essential and lightweight tasks on the critical path of tree operations: Determining initial node positions based on the *layer principle* while maintaining the *single-boundary structure*, and tracking access frequency for each path via leaf nodes as guided by the *path principle*. All other tasks are executed in the background asynchronously, minimizing the impact on the tree's performance.

Layer-aware Allocation. Guided by the *layer principle*, SIDLE allocates upper nodes in fast memory while ensuring that the relationship between the new node and its ancestors conforms to the *single-boundary structure*. Specifically, SIDLE maintains the highest level of fast memory node allocation, L_{fast}, which is adjusted based on current fast memory usage. When allocating new nodes or changing the node's position by auto-balancing, the decision on the node's placement is based on its current level l and its parent's type. If $l < L_{fast}$ and its parent is in fast memory, the node is allocated in fast memory; otherwise, it is placed in slow memory. Such a layer-aware allocation incurs little overhead on the critical path and effectively maintains the *single-boundary structure*.

Leaf-centric Access Tracking. The *path principle* highlights the necessity of placing hot paths in fast memory. Since data access ultimately reaches the leaf node, we can identify the path leading to the frequently accessed leaf as the hot path. Essentially, determining hot paths is based on leaf nodes' access frequency.

Thus, SIDLE maintains access frequency information only for leaves. For each data access, instead of updating the access information of all nodes along the path, SIDLE only updates the access information of the leaf. This leaf-centric approach significantly reduces access tracking's overhead while providing sufficient information to identify hot paths and cold nodes for effective migration.

4.2 Structure-Aware Migration

SIDLE designs the migration mechanism tailored for tree structures, ensuring that the hot paths can be quickly migrated to fast memory (§4.2.1) and that node demotion does not disrupt the *single-boundary* structure (§4.2.2). Additionally, SIDLE employs a histogram-based approach to precisely identify hot paths and cold nodes (§4.2.3). To achieve efficient migration, SIDLE decouples the preparation and execution of migration processes, assigning the migration trigger and promotion/demotion executors to work collaboratively (§4.2.4).

4.2.1 Promotion Procedure. When a leaf node in slow memory is identified as a hot node that needs promotion, following the *path principle*, it is necessary to quickly migrate the entire hot path to fast

Algorithm 1: Demotion Algorithm

```
1  Initialization: The migration trigger traverses all leaf nodes, finds
     the leaf nodes that need to be demoted, and adds them to the
     demotion queue;
2  while demotion queue is not empty do
3      pop node cur from demotion queue;
5      if cur.level < L_demote then
6          break;
7      end
9      if cur is internal node and at least one of its children is in fast
         memory then
10         continue;
11     end
12     demote cur to slow memory;
13     add cur's parent to the demotion queue if it is not in the queue;
14 end
```

memory. As shown in Figure 10(a), SIDLE's promotion procedure starts with the leaf node (①), recursively migrating all of its ancestor nodes in slow memory (②) upwards to fast memory.

4.2.2 Demotion Procedure. When a leaf in fast memory is identified as cold, it is essential to demote not only this leaf but also some of its ancestors to better utilize the limited fast memory resources. The specific number of ancestor nodes to be demoted depends on two key factors. The first is the system parameter L_{demote}, which reflects the intensity of demotion required under the current fast memory usage. The second factor is to ensure that the demotion does not disrupt the *single-boundary structure*. As Figure 10(b) shows, nodes A, B, and C are current demotion candidates. Migrating node C and then its parent F breaches the *single-boundary structure*, as node D is still in fast memory. Thus, to demote parent node F, node D must also be demoted together. However, if node D, which is not a demotion candidate, is demoted, it may inadvertently migrate a node in a hot path to slow memory, potentially degrading performance.

To address this, SIDLE adopts the demotion procedure as algorithm 1: once a node is demoted, its parent is queued for demotion (line 13). The decision to demote the parent is deferred until all its cold children have been processed by the demotion executor. If all of the parent's children have been in slow memory by then, it indicates that the parent node is not part of any hot path, making it safe to demote. Conversely, if not all children are in slow memory, the parent node is kept in fast memory to prevent the demotion of hot paths while maintaining *single-boundary structure* (line 9). Moreover, the lowest level for demotion is denoted by L_{demote}, nodes below L_{demote} will not participate in the demotion process (line 5). E.g., for the tree in Figure 10(b), its demotion follows labels ①~④.

4.2.3 Histogram-based Hotness Identification. Like prior studies [74, 79, 141], SIDLE uses a histogram to represent the access frequency distribution, thereby identifying hot paths and cold nodes for migration. Specifically, the histogram's x-axis denotes access frequency ranges in logarithmic scale (e.g., the 2nd bin represents [2, 4)), while the y-axis represents the leaf count within each range.

The migration trigger and the cooler collaborate to maintain the histogram. The migration trigger updates the histogram with the leaf nodes' latest access frequency. The cooler halves all leaf nodes' access frequency and reflects the reduction in the histogram. Note that, as the histogram is on a logarithmic scale, the reduction can be efficiently conducted by shifting all bins to the left by one position.

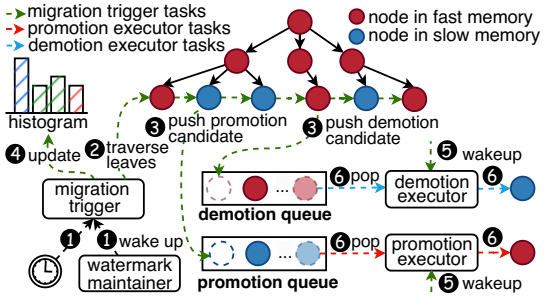


Figure 11: Migration workflow.

To identify candidates for migration, SIDLE derives hot and cold thresholds (T_{hot}, T_{cold}) from the histogram. Leaves with access frequency higher than T_{hot} are identified as hot; while those with access frequency lower than T_{cold} are classified as cold.

As static thresholds cannot adapt to dynamic workloads, SIDLE periodically updates the hot and cold thresholds based on the target percentage of cold and hot leaves, P_{cold} and P_{hot}. For updating T_{hot} and T_{cold}, SIDLE traverses the histogram from the right (hottest) side, accumulating each bin's leaf count leftward. Once the cumulative proportion of leaves exceeds P_{hot}, the corresponding bin's upper bound is set as T_{hot}. Similarly, T_{cold} is set to a value where leaves with frequency lower than T_{cold} just fall below P_{cold}. The interval between T_{hot} and T_{cold} prevents premature classification of less accessed nodes as cold, avoiding aggressive migration.

4.2.4 Migration Workflow. As shown in Figure 11, the migration trigger is activated when fast memory usage is under pressure or on a schedule (❶). It scans all leaves (❷) and selects cold ones in fast memory as demotion candidates, and hot ones in slow memory as promotion candidates. These nodes are pushed into the respective queues (❸). Concurrently, the migration trigger updates the access frequency histogram (❹). Then, the corresponding executor is awakened (❺) to perform the migration following §4.2.1 and §4.2.2 (❻). Note that migration is executed by independent workers, enabling the trigger to continue selecting migration candidates without being blocked by the migration process.

4.3 Hyper Watermark Mechanism

SIDLE selects appropriate initial positions for nodes by their level (§4.1) and periodically migrates hot and cold nodes (§4.2). This involves a series of tree-related parameters, i.e., L_{fast} (§4.1), L_{demote} (§4.2), and P_{cold}/P_{hot} (§4.2). Manually tuning these parameters is challenging, as it is difficult to adapt to workload changes and coordinate allocation with migration, causing unstable performance.

Asymmetric Adjustment. In SIDLE, the user only needs to specify the high and low watermarks of fast memory usage ratio (i.e., U_{high} and U_{low}). Other parameters (L_{fast}, L_{demote}, P_{cold}, P_{hot}) are automatically adjusted by the *hyper watermark mechanism* based on real-time fast memory usage. Specifically, the watermark maintainer monitors the fast memory usage ratio (U_f) and performs asymmetric adjustments to ensure U_f remains between the user-defined high and low watermarks.

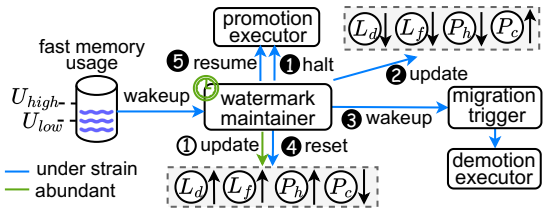


Figure 12: Workflow of hyper watermark mechanism. L_d: L_{demote}, L_f: L_{fast}, P_c: P_{cold}, P_h: P_{hot}.

Case1: Fast Memory Under Strain ($U_f > U_{high}$). Fast memory strain forces new upper nodes to be allocated to slow memory, obstructs hot path promotions, or triggers burst demotions, causing a severe performance drop. In such cases, SIDLE triggers *aggressive* adjustments to promptly bring fast memory usage back to a reasonable range. Specifically, as shown by the blue lines in Figure 12, The watermark maintainer first halts all ongoing promotions (❶). It then recalibrates parameters to intensify demotion and restrict new fast-memory allocations by increasing P_{cold}, decreasing L_{demote} and L_{fast} (❷). Watermark maintainer then activates the migration trigger and demotion executor to select and demote cold nodes (❸), iteratively repeating this process (❷–❸) until U_f falls below U_{high}. Once pressure is relieved, to adapt to current workload and avoid over-adjustment, it resets parameters to the average of their current and pre-adjustment values (❹) and resumes promotions (❺).

Case2: Fast Memory Abundant ($U_f < U_{low}$). While fast memory abundance leaves some fast memory underutilized, it does not have such a significant impact on performance stability. In such cases, we adjust parameters *conservatively* without immediately triggering migration. Specifically, as shown by the green line and clock icon in Figure 12, the watermark maintainer periodically checks if the fast memory usage drops below U_{low}. If so, it increases L_{demote}, L_{fast}, P_{hot}; and decreases P_{cold} (①). This raises chances of promotion and fast memory allocation while reducing demotion likelihood.

Benefits and Trade-offs. The hyper watermark mechanism simplifies deployment by eliminating manual tuning. Moreover, by treating real-time fast memory usage as a direct indicator of parameter quality, it automatically recalibrates to suitable configurations that ensure stable performance under varying workloads. Specifically, the aggressive adjustment under fast memory strain effectively prevents fast memory exhaustion, which could cause severe performance degradation. Meanwhile, the conservative adjustment during fast memory abundance optimizes utilization while reserving space for future new allocations and promotions.

While the aggressive adjustment may lead to temporarily underutilized fast memory when recovering from high load, and the conservative adjustment may inadvertently affect warm data, we believe these trade-offs are necessary to avoid severe performance cliffs caused by fast memory exhaustion and benefit overall stability.

Optimizations. To mitigate over-correction caused by aggressive adjustments, SIDLE employs two optimizations: (1) promptly halting ongoing adjustments for memory strain when detecting U_f already below U_{low}, and (2) limiting decreases of L_{fast} and L_{demote} to at least L_{min} (in our implementation, half of the max layers fittable in fast memory), avoiding over-pessimistic allocations and demotions.

The migration workflow in §4.2.4 only demotes the ancestors of fast-memory leaves. To prevent cold ancestors of slow memory leaves from being stuck in fast memory, the migration trigger also adds some cold leaves in slow memory to the demotion queue, creating the opportunity to demote their ancestors later.

```
►Invoke: periodic/under strain
func watermarkMaintainer():
 // U_f: current fast memory usage
 if U_f > U_high: adjustStrain()
 elif U_f < U_low: adjustAbundant()
►Fast memory abundant
func adjustAbundant():
 if L_d < L_max: L_d += 1 ①
 if L_f < L_max: L_f += 1
 if P_h < 100: P_h += 1
 if P_c > 0: P_c -= 1
```

```
►Fast memory under strain
func adjustStrain():
 stop(promote) ❶
 while (U_f > U_high):
  if L_d > L_min: L_d -= 1
  if L_f > L_min: L_f -= 1
  if P_h > 0: P_h -= 1 ❷
  if P_c < 100: P_c += 1
  invoke(demote) ❸
 reset(L_d, L_f, P_c, P_h) ❹
 start(promote) ❺
```

Figure 13: Pseudocode of watermark. L_{max}: *tree height. The steps ❶–❺ and ① correspond directly to those in Figure 12.*

Pseudocode. To clearly illustrate the hyper watermark mechanism, we provide pseudocode in Figure 13. watermarkMaintainer() monitors real-time fast memory usage U_f to autonomously recalibrate system behaviors. This approach transforms parameters into adaptive variables responsive to fast memory usage, driving the system toward suitable configurations through continuous self-correction.

5 FRAMEWORK AND CONCURRENCY

SIDLE Framework. We provide a lightweight and easy-to-use framework for integrating SIDLE into various existing tree-structure indexes. For SIDLE metadata, the framework requires developers to add one byte in internal nodes to record the node's level and memory type. Leaf nodes need two additional bytes for access frequency.[3] This memory overhead is acceptable as tree nodes are typically equal to or larger than the cacheline [61], and sometimes the node size is unaffected due to memory alignment.

For the frontend module, SIDLE framework provides helper functions for layer-aware allocation and leaf-centric access tracking. Developers can directly invoke these functions.

Additionally, to interact with diverse trees, developers need to implement *leaf iteration* and *node migration* interfaces. When background workers need to interact with the tree, they invoke these interfaces. For example, the promotion executor uses the migration interface to move hot-path nodes to fast memory. Notably, developers can usually leverage existing tree logic to implement these interfaces, such as the iterator for the leaf iteration and node splitting for the migration. We provide more details about the framework and a user guide in supplementary material.

Framework Integration. We have integrated ART and Masstree (a multicore-optimized B+ tree variant) with SIDLE framework, termed as S-ART and S-Masstree, respectively. In contrast to Masstree's 25 k LoC and ART's 1.7 k LoC, SIDLE only modifies ~150 and ~50 LOC, respectively. We further use ~1.8 k LoC to implement SIDLE framework, along with ~450 and ~320 LoC to implement its interfaces for Masstree and ART. This lightweight integration indicates that SIDLE can be easily applied to existing tree-structure indexes.

Database Integration. We integrate SIDLE into the in-memory OLTP database Silo [127], replacing its original Masstree index. We incorporate SIDLE's background workers (five threads) into Silo's

[3]For migration, each node also needs to maintain a parent pointer. Notably, most tree implementations (such as B+trees [1, 34], tries [128], and binary search trees [29, 105]) already incorporate parent pointers for performance or concurrency purposes.

Table 1: Workload composition. *HS is Hotspot.*

Workload	Request Composition
Update Heavy (HS, YCSB-A)	50% Read, 50% Update
Read Mostly (HS, YCSB-B)	95% Read, 5% Update
Read Only (HS, YCSB-C)	100% Read
With Insert (HS), Read Latest (YCSB-D)	95% Read, 5% Insert
Short Ranges (YCSB-E)	95% Scan, 5% Insert
Read Modify Write (RMW) (YCSB-F)	50% Read, 50% RMW

existing background module, launching these threads upon the database engine startup. Since databases typically already have a background module (for logging, GC, etc.), adding these workers introduces minimal complexity. While each worker runs a dedicated thread (five in total), their resource footprint is very lightweight and will not affect foreground tasks' performance (Figure 18(a)).

Concurrency Correctness. SIDLE introduces a total of four shared data that involve concurrent interations: (1) *histogram* shared between migration trigger and cooler, (2) *system parameters* updated by watermark maintainer and read by other background workers, (3) *leaf access frequency* read/written by background workers and foreground tree operations, and (4) *tree structure* concurrently modified by background migrations and foreground tree operations. The first three cases only involve reads/writes within 8 bytes, thus we use atomic operations (e.g., CAS) to ensure concurrency safety and correctness. In contrast, the fourth case involves tree structural modifications, for which we adopt optimistic concurrency control.

Specifically, to prevent concurrent writes, we lock the node and its parent until the migration completes. For trees with the leaf-level linked list (e.g., B+ tree), we also lock the leaf's siblings when migrating it. The locking order is consistent with other tree operations to prevent deadlocks. To prevent concurrent reads, the node is flagged as migrating before migration, and its version is updated afterward. Reads retry upon detecting this flag or version changes.

This can ensure the correctness of migration and foreground operations. The migration and foreground writes are mutually exclusive; thus, migration will not corrupt the node data and tree structure, and foreground writes cannot access inconsistent data during migration. Meanwhile, the version check ensures that reads cannot access inconsistent data during migration and makes migration atomic and transparent to reads. We further formally verify the correctness of concurrent migrations and foreground operations using TLA+ [75] in the supplementary material.

Configuration. In SIDLE's runtime, by default, the migration trigger, the cooler, and the watermark maintainer wake up every 500 ms, 2000 ms, and 100 ms, respectively. The high watermark and low watermark are set to 95% and 85%, respectively.

6 EVALUATION

6.1 Environment Setup

Testbed. We run all experiments on Intel® Xeon® Platinum 8468V CPUs (48 cores), equipped with 395 MiB CPU cache, 32 GiB DRAM, and 32 GiB CXL-attached DRAM (connected through a CXL 1.1 memory expansion card [14]), except the large scale dataset scalability test (§6.4 part2). Since the remote NUMA-node is another tier of slow memory besides CXL, all experiments use a single socket, which is a common evaluation practice in both industry [103, 156] and academia [79, 114, 130, 141]. The thread count is 28 (except Figure 18(a)), key and value sizes are both 8 bytes (except Figure 17(b)).

Table 2: Memory footprint of S-ART (GB) vs. dataset sizes.

Dataset	10M	20M	50M	100M	200M	400M	0.5B	1B	2B	4B
Mem. (GB)	2.6	4.9	14.9	26.8	48.5	63.7	81.6	188.2	432.3	996.1

Workloads. For microbenchmarks, we use a Hotspot (HS) access distribution, where 90% of requests concentrate on 5% of contiguous keys (termed *hot region*). This pattern closely matches the strong locality of hot keys in production environments [41]. We use YCSB[4] [47] with the default Zipfian distribution as the macrobenchmark. Details of both benchmarks are shown in Table 1. We use TPC-C [126] to evaluate SIDLE's performance on database tasks. Additionally, we evaluate SIDLE using Alibaba Block Traces [4, 133], which is a real-world workload collected from production.

Memory Configurations. Unless otherwise specified, all experiments use 20M records, with maximum fast memory usage limited to 20% of the total memory footprint. This fast memory limit follows the typical practice of using CXL for memory expansion and is consistent with prior work [79, 114, 130]. We also further evaluate SIDLE across various fast memory usage and data scales to show its stability (§6.3, §6.4), with memory footprints shown in Table 2.

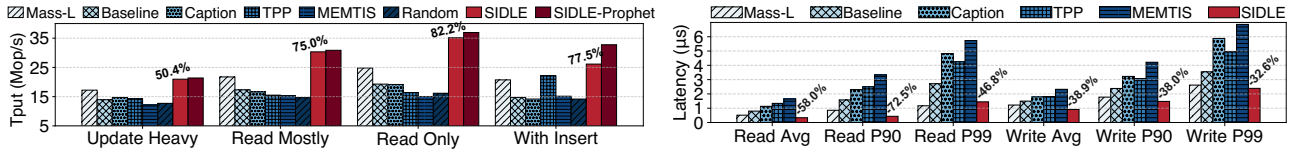
Compared Systems. The **baseline** uses the default weighted allocation policy [136] to allocate a specified ratio of memory to CXL. Moreover, we compare SIDLE with three SOTA data placement schemes for CXL-HM: **TPP** [103], **MEMTIS** [79], and **Caption** [124]. We also compare S-ART with **PAC-L** and S-Masstree with **Mass-L**. PAC-L and Mass-L are optimized PACTree [70] and Masstree [101], respectively, both utilizing *fast memory internode* techniques (§2.3). To evaluate the effectiveness of SIDLE's migration, we implement a **Random** baseline: on each node access, promote the current node with a 1% probability and demote a random one.

6.2 Overall Performance

Microbenchmark. SIDLE-Prophet represents an ideal scenario where all hot paths are pre-allocated in fast memory. For fairness, we use SIDLE-Prophet's fast memory usage (~10% of total memory) as the limit for other systems. Figure 14(a) shows that S-Masstree outperforms the baseline by 50.4%–82.2%[5] by placing hot nodes in fast memory using layer-aware allocation and structure-aware migration. SIDLE performs similarly to SIDLE-Prophet, proving it can effectively promote hot paths. SIDLE-Prophet only beats SIDLE in *With Insert* by allocating most new nodes in fast memory, using 30–50% more fast memory than SIDLE. SIDLE outperforms page-level data placement schemes (MEMTIS, TPP, Caption) by 42.6%–133.5% through fine-grained, dynamic node-level placement. SIDLE outperforms the Mass-L by 21.8%–41.7%, which statically places some nodes in fast memory. SIDLE also outperforms Random by 65.1%–117.5%, as its chance-based promotion and random demotion cannot effectively identify and retain hot nodes in fast memory.

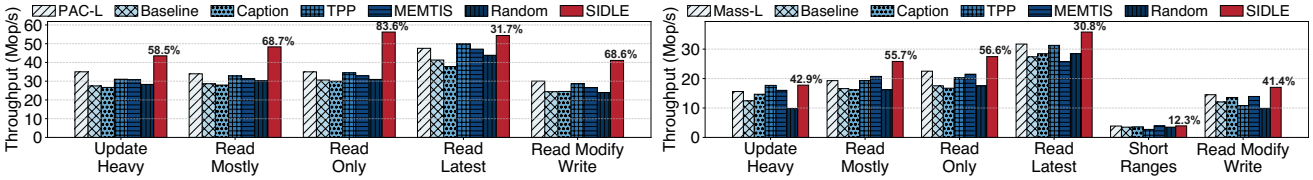
For other systems, Caption matches the baseline due to its similar interleaved allocation method. TPP performs well in *With Insert* by initially allocating all nodes to fast memory, but suffers in *Read Only*

[4]We don't evaluate *Short Range* for S-ART as its source code lacks scan.
[5]In Figure 14, Figure 15, Figure 23(a), and Figure 24(a), the marked improvements are compared with the baseline.



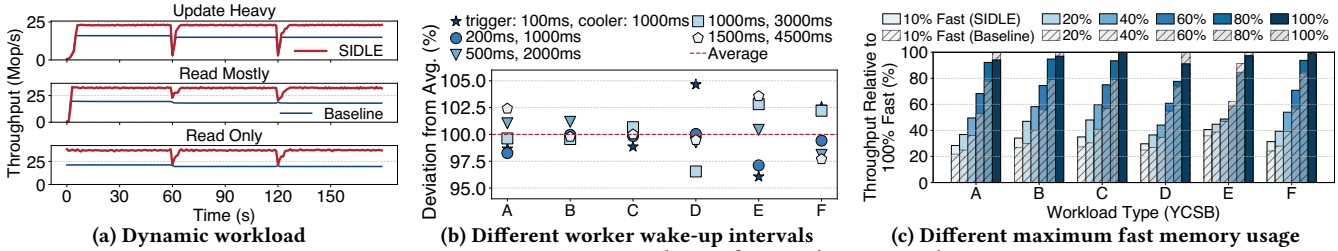
(a) Throughput (b) Latency of *Update Heavy* workload

Figure 14: Performance of SIDLE (S-Masstree) and compared systems in the microbenchmark.



(a) S-ART (b) S-Masstree

Figure 15: Throughput of SIDLE and compared systems in the macrobenchmark.



(a) Dynamic workload (b) Different worker wake-up intervals (c) Different maximum fast memory usage

Figure 16: Sensitivity analysis of SIDLE (S-Masstree).

as it struggles to identify and demote cold nodes. MEMTIS performs poorly across all workloads because its default usage of huge pages does not adapt well to tree characteristics. Mass-L outperforms other baselines by placing upper nodes in fast memory, but it still lags behind SIDLE due to its static placement.

Macrobenchmark. Figure 15 shows that S-ART and S-Masstree outperform the baseline by 31.7%–83.6% and 30.8%–56.6% (except for *Short Ranges*), respectively. *Short Ranges*' performance depends on fast memory leaf count. Due to the limited fast memory and SIDLE prioritizing upper nodes in fast memory, the performance gain in this workload is not as significant (12.3%). SIDLE performs better in read-dominant workloads, as migration interferes less with reads. PAC-L performs worse than S-ART since its static memory layout cannot adapt to dynamic workloads. Other data placement systems fail to perform well across all workloads. E.g., while TPP performs similarly to S-Masstree in *Update Heavy*, it fails to maintain comparable performance for *Read Only* and *Read Modify Write*.

Latency Analysis. We measure SIDLE's latency under both micro and macro benchmarks. Due to the limited space, we only present the result of S-Masstree under the microbenchmark *Update Heavy* workload, though similar improvements are observed across other workloads. In Figure 14(b), SIDLE's average read/write latencies are both under 1 μs. Moreover, SIDLE greatly reduces read latency, with P90 latency 72.5% lower than the baseline. SIDLE significantly reduces P90/P99 latency versus TPP and MEMTIS by continuously detecting and demoting cold nodes, along with the hyper watermark mechanism, thereby preventing burst migrations.

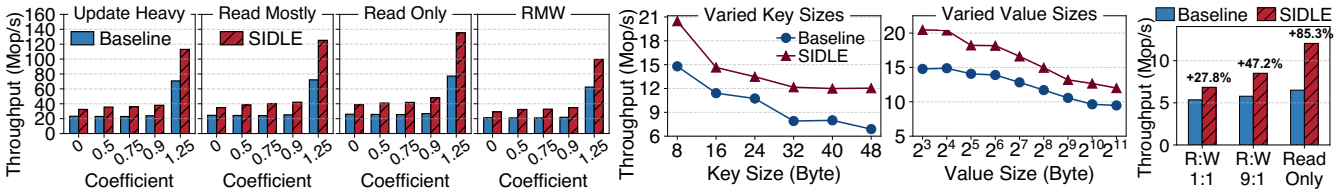
6.3 Sensitivity Analysis

In SIDLE, most parameters are dynamically determined, except for workers' wake-up intervals and maximum fast memory usage. To evaluate SIDLE's stability, we conduct a sensitivity analysis of its performance under different workloads and parameter settings.

Dynamic Workload Changes. To simulate dynamic workload changes, we shift the entire hot region in the microbenchmark every 60 s. As shown in Figure 16(a), although SIDLE suffers a temporary performance decline when the hot region changes—due to factors like the invalidation of hot paths and CPU cache misses—it quickly detects the new hot paths and promotes them. SIDLE can fully recover its performance in up to 6 s and regain 84% of it within 2 s, showing its ability to quickly adapt to changes in hot paths for stable high performance in dynamic workloads.

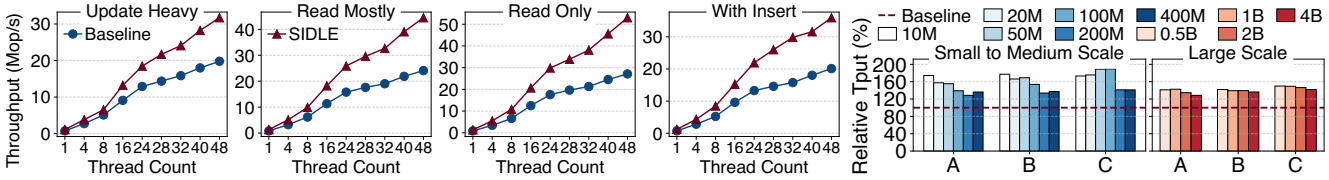
Different Worker Wake-up Intervals. In SIDLE, the migration worker, cooler, and watermark maintainer are regularly awakened. The watermark maintainer's wake-up interval has minimal impact because it only adjusts some parameters during regular awakenings. Thus, we only evaluate the impact of wake-up intervals using five sets of different intervals for the other two workers. Figure 16(b) shows that SIDLE's performance is stable across wake-up interval choices, with < 5% deviations from the average. In practice, wake-up intervals can be adjusted based on system conditions, such as extending the intervals under high CPU load to reduce overhead.

Different Maximum Fast Memory Usage. Figure 16(c) shows SIDLE's throughput scaling steadily as fast memory allocation increases from 10% to 80%. At 80%, SIDLE's throughput is close to all data in fast memory for most workloads. Above 60% allocation, SIDLE slightly lags the baseline in *Short Ranges* because the higher fast memory usage intensifies competition for CPU cache between background workers and the scan. Nevertheless, SIDLE still outperforms the baseline when only the frontend module is enabled. At 100%, SIDLE's performance is comparable (91.2–99.9%) to statically placing all data in fast memory, demonstrating its low overhead.



(a) Throughput of SIDLE (S-ART) under varied skewness levels (b) Varied key/value sizes (*Micro Update Heavy*) (c) Long URL keys

Figure 17: Performance of SIDLE under different workload characteristics.



(a) Scalability with different thread counts (S-Masstree) (b) Scalability with different data scales (S-ART)

Figure 18: Scalability of SIDLE in microbenchmark.

Different Skewness Level. We evaluate SIDLE under varying levels of skewness using YCSB with different Zipfian coefficients. A higher coefficient indicates greater skewness, while a coefficient of 0 represents a uniform distribution. As Figure 17(a) shows, SIDLE performs well across all skewness levels. Under uniform and Zipfian (coefficients of 0.5, 0.75, 0.9, 1.25) distributions, SIDLE outperforms the baseline by 37.4–48.7%, 54.1–59.3%, 56.0–67.0%, 59.4–79.4%, and 59.5-75.2%. SIDLE's advantage is still substantial under the relatively uniform workload because it prioritizes placing upper nodes in fast memory, reducing slow memory access for all requests.

Different Key and Value Sizes. We evaluate S-Masstree with various key and value sizes. Figure 17(b) shows that SIDLE achieves stable performance improvement (up to 74.8%) across different key sizes. SIDLE also stably outperforms the baseline (25.0%–38.5%) with value sizes from 8 to 2048 bytes.

Long URL dataset. To assess SIDLE with long keys, we evaluate S-ART on a real-world URL dataset from MemeTracker.org [86], containing 400 M URLs averaging 82 bytes in length. Figure 17(c) shows that S-ART outperforms the baseline by 27.8–85.3%, as the leaf-centric tracking remains effective for long paths due to the tree's root-to-leaf access pattern and the migration is lightweight.

6.4 Scalability Performance

Concurrency Scalability. Figure 18(a) shows that SIDLE not only guarantees the index scalability but also derives amplified performance gains over the baseline at higher thread counts due to cumulative per-thread benefits. At 48 threads, SIDLE's throughput is 28.6–38.1 × that of a single thread. At this point, SIDLE continues to scale well despite co-locating some foreground threads with background threads, confirming negligible background overhead.

Dataset Size Scalability. Figure 18(b) shows the performance gains of S-ART over the baseline with dataset varying from 10 M to 4 B records, with S-ART's memory usage shown in Table 2. Fast memory usage is set to 50% of total memory in this experiment. Datasets exceeding our CXL testbed's memory limit (~400 M records) are evaluated on a 1 TB dual-socket machine, where we emulate CXL memory using downclocked remote NUMA [79] to achieve comparable performance (Figure 1(c)). Under different data scales, SIDLE robustly outperforms the baseline by 28.3–88.7%. Gains decrease slightly at larger scales due to the increased overhead of detecting and migrating longer, more numerous hot paths. Nevertheless, even with 4 B records (occupying ~1 TB memory), SIDLE still outperforms the baseline by 28.3–41.9%. Furthermore, the consistent improvement across large-scale datasets (0.5 B–4 B) indicates that SIDLE can effectively handle even larger datasets.

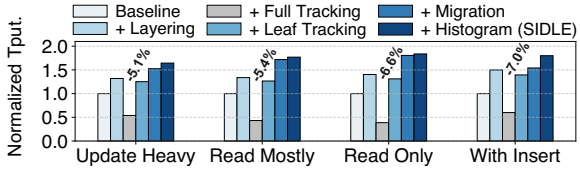


Figure 19: The factor analysis for techniques in SIDLE.

6.5 Performance Breakdown

6.5.1 Contributions of techniques.

In Figure 19, we begin with the original Masstree and gradually apply each proposed technique to conduct S-Masstree.

+ Layering. Layer-aware allocation improves throughput by 31.9–49.8% over the baseline, as it prioritizes placing upper frequently accessed nodes in fast memory, benefiting all requests.

+ Access Tracking. In this step, we evaluate access tracking's overhead. Tracking only leaves incurs a minimal 5.1–7.0% performance loss, while tracking all nodes incurs ~60% performance loss. This confirms that leaf-centric tracking's overhead is acceptable.

+ Migration. Migrating hot paths and cold nodes in this step boosts SIDLE's throughput by 2.8–28.7%, as it ensures that hot and cold nodes generated at runtime are properly placed. However, the fixed hot/cold thresholds are not suitable for all workloads, resulting in less noticeable performance gains for *With Insert*.

+ Histogram. By introducing a histogram to dynamically adjust the hot/cold thresholds according to workload characteristics, SIDLE achieves 1.7–16.8% performance gains over the static thresholds.

6.5.2 Effectiveness of Hyper Watermark. To assess the hyper watermark's effectiveness, we track S-Masstree's fast memory usage ratio over time. As shown in Figure 20, fast memory usage fluctuates due to migration, allocation, and parameter tuning. *With Insert*

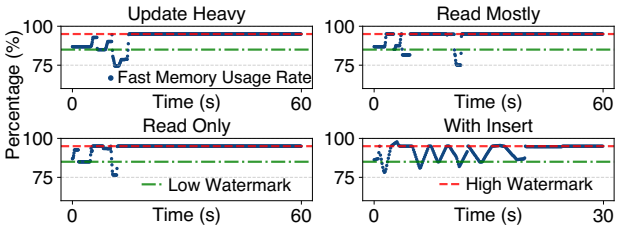


Figure 20: Fast memory usage ratio variation over time.

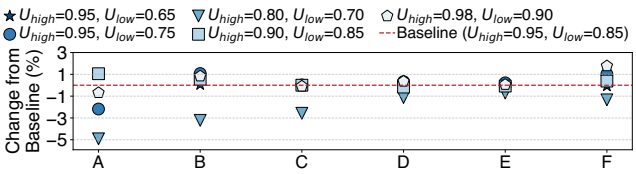


Figure 21: Impact of different watermark ranges.

has more fluctuations due to continuous new node generation. The hyper watermark mechanism maintains the fast memory usage within the watermark bounds and ensures that once hot paths are promoted and parameters are suitable, fast memory usage stabilizes near U_{high} across all workloads. Note that initial variations in fast memory usage benefit performance, as they represent hot-node promotions and parameter optimization.

Figure 21 shows SIDLE's performance under different watermark ranges. As watermark mechanism ensures fast memory usage stabilizes near U_{high} with low overhead, watermark ranges have minimal performance impact (within ±3%), except when U_{high} is too low.

6.6 Overhead Analysis

CPU Time Overhead. We assess the overhead of each background worker by measuring the proportion of its execution time to the total CPU time. Table 3 shows that the background workers only accounts for 1.51‰–2.97‰ of the total runtime. The migration trigger takes the longest time among all workers, as it scans all leaves to identify hot and cold nodes. Next is the watermark maintainer, which is frequently activated to monitor fast memory usage.

Memory Overhead. SIDLE only increases the memory usage of Masstree's internal and leaf nodes by 0.36% and 0.93%. For ART's internal and leaf nodes, the increase is 0.04%–1.3%/2.0%.

6.7 TPC-C Benchmark Evaluation

To assess SIDLE's effectiveness in real database tasks, we integrate S-ART and S-Masstree into an OLTP database according to §5. Then we evaluate its performance using the TPC-C benchmark. Figure 22 shows the performance of S-ART and S-Masstree relative to the baseline. S-ART improves throughput by 26.3% and reduces tail latency by 31.4–39.9% compared to the baseline. Similarly, S-Masstree increases throughput by 29.1% and reduces tail latency by 11.2–36.7% compared to the baseline. These results indicate that SIDLE is effective and practical in database scenarios.

6.8 Real-world Workload Evaluation

Alibaba Block Traces, collected from Alibaba Cloud elastic block service [154], reflect representative user behaviors in the cloud [133]. We choose two traces with the largest working set (i.e., those with

Table 3: Runtime proportion of background worker execution. *MT is migration trigger, PE/DE is promotion/demotion executor, CL is cooler, WM is watermark maintainer.*

Run Time Ratio (‰)	MT	PE	DE	CL	WM	Total
Update Heavy	1.69	1.37×10^{-2}	1.12×10^{-2}	1.04×10^{-2}	0.799	2.52
Read Mostly	1.52	2.36×10^{-2}	8.39×10^{-3}	3.64×10^{-3}	0.529	2.09
Read Only	1.13	6.77×10^{-3}	9.01×10^{-3}	4.29×10^{-3}	0.365	1.51
Read Latest	1.84	1.46×10^{-1}	4.65×10^{-2}	1.03×10^{-1}	0.828	2.97
Short Ranges	1.55	6.91×10^{-6}	5.40×10^{-2}	4.02×10^{-2}	0.636	2.28
RMW	1.76	1.01×10^{-1}	1.44×10^{-2}	1.06×10^{-2}	0.839	2.73

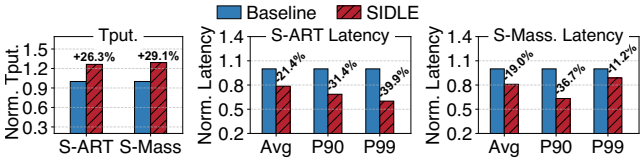


Figure 22: Comparison between SIDLE and the baseline under TPC-C benchmark. S-Mass is short for S-Masstree.

the greatest need for memory expansion) for evaluation (Trace 7 as Trace A, and Trace 40 as Trace B). Figure 23 shows that S-ART outperforms the baseline by up to 90.3% in throughput while reducing P99 latency by up to 81.2% compared to MEMTIS and 63.2% compared to PAC-L. Figure 24 reveals that S-Masstree increases throughput by up to 38.7% compared to the baseline and reduces P99 latency by up to 54.3% compared to MEMTIS. This result demonstrates that SIDLE is also effective in real-world scenarios.

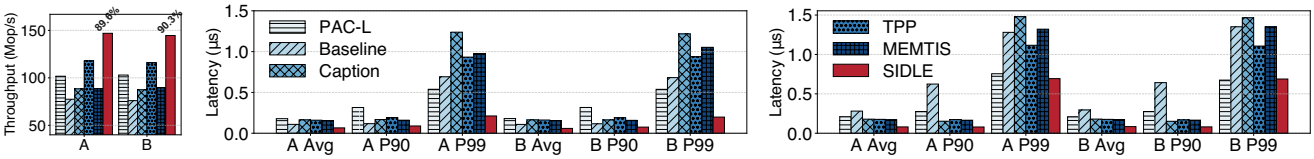
7 DISCUSSION AND FUTURE WORK

False Negative Internal Nodes. SIDLE selects promotion candidates starting from hot leaves. While aggregated accesses from infrequently accessed leaves could theoretically justify promoting an ancestor (i.e., false negative internal nodes), this rarely occurs in practice. We analyze a B+ tree with 20 M keys under a Zipfian distribution and observe such false negative internal nodes only at the last level with a probability below 0.9%.

Scan may Break Single-boundary Structure. For trees with leaf linked lists (e.g., B+ tree), scans may make leaves hotter than parents, potentially breaking single-boundary structure. In this case, SIDLE remains effective since: (1) Single-boundary structure still aids in locating start leaves. (2) Real-world scan workloads typically focus on small hot subsets [41, 51, 133]; SIDLE can promote these hot leaves, with *cold ancestor promotion being rare* (< 6.7% in YCSB-E). Thus, SIDLE still enhances B+tree scan performance (Figure 15(b)).

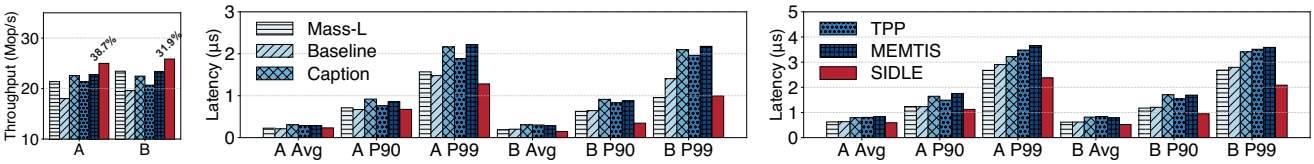
Node Size Impact on Migration Decision. Larger nodes should require higher access frequencies to justify promotion, as they occupy more fast memory. SIDLE naddresses this naturally by migrating upwards along paths. Since larger nodes like Node256 in ART act as routing hubs for more downstream paths, they inherently accumulate higher access frequencies than smaller nodes like Node4. This eliminates the need for explicit migration thresholds based on node size. Experiments confirm that promoted Node256 nodes have access frequencies ~44× higher than Node4 nodes.

Hardware-based Access Tracking. SIDLE updates access frequencies on leaf access and traverses all leaves to select migration candidates. The former inevitably affects the performance, and the latter dominates the background processing overhead (§6.6). In the future,



(a) Throughput (b) Read latency (c) Write latency

Figure 23: Performance comparison between S-ART and compared systems with Alibaba Block Traces.



(a) Throughput (b) Read latency (c) Write latency

Figure 24: Performance comparison between S-Masstree and compared systems with Alibaba Block Traces.

we will explore hardware-based sampling [8, 23], like Intel Precise Event Based Sampling (PEBS), to further reduce this overhead.

Multi-layer Memory Hierarchy. Currently, SIDLE targets two-layer memory hierarchies with fast and slow memory. To extend it to multi-layer memory hierarchies, we can attempt to rank various memory types by their latency and throughput, and then apply SIDLE's mechanisms between each pair of adjacent layers.

SIDLE's Scope and Limitations. SIDLE mainly targets CXL devices with software-visible, cache-coherent (CC) memory whose latency is within the same order of magnitude as local DRAM. SIDLE is ill-suited for CXL devices with hardware-managed caching [156] or lacking CC. Nevertheless, mainstream CXL-attached memory meets these requirements and behaves as "slightly slower DRAM". Across various architectures (e.g., multi-hop introduced by CXL switches), generations (i.e., CXL 1.1–4.0), and vendors, it typically exhibits 1.5–4× higher latency than local DRAM [10–12, 66, 94, 124, 137].

8 RELATED WORK

Page-level Data Placement. Page-level data placement schemes aim to store hot pages in the fast memory and cold pages in the slow memory, while migrating pages to meet the workload demands [7, 24, 46, 53, 67, 69, 72, 74, 95, 102, 109, 115, 117, 129, 141, 143]. SIDLE is inspired by the ideas and methods from these works and designs the data placement scheme for tree-structure indexes on CXL-HM. Note that SIDLE mainly targets scenarios with low memory contention; Colloid [130] may further enhance SIDLE when contention occurs.

Object-granularity Data Placement. The first type of object-granularity data placement includes libraries [6, 52, 106, 138, 153] like CacheLib [6, 106], X-Mem [52] and HeMem [114], which require specialized APIs or offline profiling, or otherwise cannot manage small allocations and migrate data at the page level. The second type is hardware-based cacheline-granularity placement [84, 123] like Intel Flat Memory Mode [156]. Unlike them, SIDLE leverages trees' characteristics to achieve fine-grained management and richer semantics than hardware-based solutions without offline profiling.

CXL for Database. Many studies have explored leveraging CXL to improve database performance [26, 27, 33, 58, 65, 76, 77, 85, 137, 146]. Unlike prior methods in CXL databases that simply cached hot data [63], SIDLE adopts a fine-grained and dynamic data placement that better leverages CXL's features, benefiting databases on CXL.

Far Memory Management. Far memory systems [25, 28, 36, 59, 88, 110, 116, 119, 131, 142, 157] utilize network-connected memory for expansion. Unlike CXL, accessing far memory requires different interfaces and protocols (e.g., RDMA, RPCs), and incurs much higher latency. Far memory systems and SIDLE are both related and orthogonal, as far memory indexes can benefit from our insights, and SIDLE may employ far memory as a new slow memory layer.

IO-optimized Tree Indexes. Unlike CXL-HM, the performance gap between disk and DRAM is much larger than that between CXL and DRAM. Moreover, disks can only be accessed at the page granularity. Thus, previous works optimized for DRAM + disk systems [43, 55, 57, 60, 62, 80, 107, 121, 147] (e.g., LeanStore [80]) mainly rely on caching to reduce disk accesses. However, our analysis in §2.3 shows that caching is ineffective for CXL-HM. Motivated by this, we design SIDLE for tree-structure indexes on CXL-HM.

Cache-optimized Tree Indexes. Cache-optimized indexes [35, 38, 56, 64, 112, 113] primarily improve CPU cache hit rates by optimizing the memory layout of trees and leveraging techniques such as prefetching. However, these techniques are inapplicable to CXL-HM because local DRAM cannot automatically serve as a cache for CXL-attached memory like CPU cache. Instead, the SIDLE framework can help these indexes perform better on CXL-HM.

9 CONCLUSION

We identify that the placement of tree-structure indexes on CXL-HM should match the tree's inherent characteristics with CXL-HM features. We propose SIDLE, a fine-grained, structure-aware data placement scheme for tree-structure indexes on CXL-HM that achieves high stable performance.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and feedback. We are sincerely grateful to Jingkai He for his valuable support throughout this work, and to Dingyan Zhang for his helpful suggestions. This work is supported in part by the National Natural Science Foundation of China (No. 62132014), the Fundamental Research Funds for the Central Universities, the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (JYB2025XDXM113). Corresponding author: Mingkai Dong (mingkaidong@sjtu.edu.cn).

REFERENCES

[1] 2012. kohler/masstree-beta: Beta release of Masstree. https://github.com/kohler/masstree-beta.

[2] 2013. armon/libart: Adaptive Radix Trees implemented in C. https://github.com/armon/libart.

[3] 2014. begeekmyfriend/bplustree: A minimal but extreme fast B+ tree indexing structure demo for billions of key-value storage. https://github.com/begeekmyfriend/bplustree.

[4] 2020. alibaba/block-traces. https://github.com/alibaba/block-traces.

[5] 2021. Intel® Memory Latency Checker V3.11. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

[6] 2021. Introducing new memory types to CacheLib. https://github.com/facebook/CacheLib/discussions/102.

[7] 2022. kernel/git/vishal/tiering.git - Vishal Verma's fork of linux.git. https://git.kernel.org/pub/scm/linux/kernel/git/vishal/tiering.git/log/?h=tiering-0.8.

[8] 2022. Performance Monitoring Unit Sharing Guide. https://www.intel.com/content/www/us/en/content-details/727001/performance-monitoring-unit-sharing-guide.html.

[9] 2024. CXL 1.0 Specification. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-1.0-Specification.pdf.

[10] 2024. CXL 2.0 Specification. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-2.0-Specification.pdf.

[11] 2024. CXL 3.0 Specification. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-3.0-Specification.pdf.

[12] 2024. CXL 4.0 Specification. https://computeexpresslink.org/wp-content/uploads/2025/12/CXL_4.0-Webinar_December-2025_FINAL.pdf.

[13] 2024. CXL Memory Module - Box (CMM-B). https://semiconductor.samsung.com/news-events/tech-blog/cxl-memory-module-box-cmm-b/.

[14] 2024. CXL® Memory eXpander Controller (MXC). https://www.montage-tech.com/MXC.

[15] 2024. HyPer – A Hybrid OLTP&OLAP High Performance DBMS. https://hyper-db.de/.

[16] 2024. MemSQL. https://www.memsql.com/.

[17] 2024. SAP HANA. https://www.sap.com/products/hana.html.

[18] 2024. TimesTen: Fastest OLTP database, ultra high availability, elastic scalability. https://www.oracle.com/database/technologies/related/timesten.html.

[19] 2024. VoltDB. https://www.voltdb.com/.

[20] 2025. Apache Druid. https://druid.apache.org/.

[21] 2025. Apache Flink®: Stateful Computations over Data Streams. https://flink.apache.org/.

[22] 2025. Elasticsearch: The heart of the Elastic Stack. https://www.elastic.co/elasticsearch.

[23] Advanced Micro Devices. 2022. AMD uProf v4.0 User Guide. https://www.amd.com/content/dam/amd/en/documents/developer/uprof-v4.0-gaGA-user-guide.pdf. Accessed: 2024-03-26.

[24] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-transparent Page Management for Two-tiered Main Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 631–644. https://doi.org/10.1145/3037697.3037706

[25] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 775–787. https://www.usenix.org/conference/atc18/presentation/aguilera

[26] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. 2022. Enabling CXL Memory Expansion for In-Memory Database Management Systems. In *Proceedings of the 18th International Workshop on Data Management on New Hardware* (Philadelphia, PA, USA) *(DaMoN '22)*. Association for Computing Machinery, New York, NY, USA, Article 8, 5 pages. https://doi.org/10.1145/3533737.3535090

[27] Minseon Ahn, Thomas Willhalm, Norman May, Donghun Lee, Suprasad Mutalik Desai, Daniel Booss, Jungmin Kim, Navneet Singh, Daniel Ritter, and Oliver Rebholz. 2024. An Examination of CXL Memory Use Cases for In-Memory Database Management Systems Using SAP HANA. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 3827–3840. https://doi.org/10.14778/3685800.3685809

[28] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively prefetching remote memory with leap. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'20)*. USENIX Association, USA, Article 58, 15 pages.

[29] Anandarao. 2017. Red-Black Tree Implementation. https://github.com/anandarao/Red-Black-Tree. Accessed: 2025-08-28.

[30] Christoph Anneser, Andreas Kipf, Huanchen Zhang, Thomas Neumann, and Alfons Kemper. 2022. Adaptive Hybrid Indexes. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1626–1639. https://doi.org/10.1145/3514221.3526121

[31] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1383–1394. https://doi.org/10.1145/2723372.2742797

[32] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*. 53–64.

[33] Alexander Baumstark, Marcus Paradies, Kai-Uwe Sattler, Steffen Kläbe, and Stephan Baumann. 2024. So Far and yet so Near - Accelerating Distributed Joins with CXL. In *Proceedings of the 20th International Workshop on Data Management on New Hardware* (Santiago, AA, Chile) *(DaMoN '24)*. Association for Computing Machinery, New York, NY, USA, Article 7, 9 pages. https://doi.org/10.1145/3662010.3663449

[34] begeekmyfriend. 2022. A minimal but extreme fast B+ tree indexing structure demo for billions of key-value storage. https://github.com/begeekmyfriend/bplustree. Accessed: 2025-08-28.

[35] M.A. Bender, E.D. Demaine, and M. Farach-Colton. 2000. Cache-oblivious B-trees. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*. 399–409. https://doi.org/10.1109/SFCS.2000.892128

[36] Ankit Bhardwaj, Chinmay Kulkarni, and Ryan Stutsman. 2020. Adaptive Placement for In-memory Storage Functions. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 127–141. https://www.usenix.org/conference/atc20/presentation/bhardwaj

[37] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A height optimized trie index for main-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data*. 521–534.

[38] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. 2002. Cache oblivious search trees via binary trees of small height. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (San Francisco, California) *(SODA '02)*. Society for Industrial and Applied Mathematics, USA, 39–48.

[39] Irina Calciu, Justin Gottschlich, and Maurice Herlihy. 2013. Using Elimination and Delegation to Implement a Scalable NUMA-Friendly Stack. In *5th USENIX Workshop on Hot Topics in Parallelism (HotPar 13)*. USENIX Association, San Jose, CA. https://www.usenix.org/conference/hotpar13/workshop-program/presentation/calciu

[40] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2017. Black-box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 207–221. https://doi.org/10.1145/3037697.3037721

[41] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, modeling, and benchmarking RocksDB key-value workloads at facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) *(FAST'20)*. USENIX Association, USA, 209–224.

[42] Lei Chen, Shi Liu, Chenxi Wang, Haoran Ma, Yifan Qiao, Zhe Wang, Chenggang Wu, Youyou Lu, Xiaobing Feng, Huimin Cui, Shan Lu, and Harry Xu. 2024. A Tale of Two Paths: Toward a Hybrid Data Plane for Efficient Far-Memory Applications. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 77–95. https://www.usenix.org/conference/osdi24/presentation/chen-lei

[43] Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, and Gary Valentin. 2002. Fractal prefetching B+-Trees: optimizing both cache and disk performance. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (Madison, Wisconsin) *(SIGMOD '02)*. Association for Computing Machinery, New York, NY, USA, 157–168. https://doi.org/10.1145/564691.564710

[44] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in non-volatile main memory. *Proc. VLDB Endow.* 8, 7 (Feb. 2015), 786–797. https://doi.org/10.14778/2752939.2752947

[45] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. uTree: a persistent B+-tree with low tail latency. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2634–2648. https://doi.org/10.14778/3407790.3407850

[46] Jinyoung Choi, Sergey Blagodurov, and Hung-Wei Tseng. 2021. Dancing in the dark: Profiling for tiered memory. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 13–22.

[47] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana,

USA) *(SoCC '10)*. Association for Computing Machinery, 143–154.

[48] Henry Daly, Ahmed Hassan, Michael F Spear, and Roberto Palmieri. 2018. NUMASK: high performance scalable skip list for NUMA. In *32nd International Symposium on Distributed Computing (DISC 2018)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

[49] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. 2024. An Introduction to the Compute Express Link (CXL) Interconnect. https://doi.org/10.1145/3669900. *ACM Comput. Surv.* 56, 11, Article 290 (July 2024), 37 pages. https://doi.org/10.1145/3669900

[50] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. 2013. Anti-caching: A new approach to database management system architecture. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1942–1953.

[51] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: an extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.* 7, 4 (Dec. 2013), 277–288. https://doi.org/10.14778/2732240.2732246

[52] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.

[53] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, et al. 2023. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 727–741.

[54] Ahmed Eldawy, Justin Levandoski, and Per-Åke Larson. 2014. Trekking through siberia: Managing cold data in a memory-optimized database. *Proceedings of the VLDB Endowment* 7, 11 (2014), 931–942.

[55] Goetz Graefe and Harumi Kuno. 2011. Modern B-tree techniques. In *2011 IEEE 27th International Conference on Data Engineering*. 1370–1373. https://doi.org/10.1109/ICDE.2011.5767956

[56] G. Graefe and P.-A. Larson. 2001. B-tree indexes and CPU caches. In *Proceedings 17th International Conference on Data Engineering*. 349–358. https://doi.org/10.1109/ICDE.2001.914847

[57] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi Kuno, Joseph Tucek, Mark Lillibridge, and Alistair Veitch. 2014. In-memory performance for big data. *Proc. VLDB Endow.* 8, 1 (Sept. 2014), 37–48. https://doi.org/10.14778/2735461.2735465

[58] Yunyan Guo and Guoliang Li. 2024. A CXL- Powered Database System: Opportunities and Challenges. *2024 IEEE 40th International Conference on Data Engineering (ICDE)* (2024), 5593–5604. https://api.semanticscholar.org/CorpusID:269491865

[59] Zhiyuan Guo, Zijian He, and Yiying Zhang. 2023. Mira: A Program-Behavior-Guided Far Memory System. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 692–708. https://doi.org/10.1145/3600006.3613157

[60] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, and How to Exploit it: High-Performance I/O for High-Performance Storage Engines. *Proc. VLDB Endow.* 16, 9 (May 2023), 2090–2102. https://doi.org/10.14778/3598581.3598584

[61] Richard A. Hankins and Jignesh M. Patel. 2003. Effect of node size on the performance of cache-conscious B+-trees. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (San Diego, CA, USA) *(SIGMETRICS '03)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/781027.781063

[62] Xiangpeng Hao and Badrish Chandramouli. 2024. Bf-Tree: A Modern Read-Write-Optimized Concurrent Larger-Than-Memory Range Index. *Proc. VLDB Endow.* 17, 11 (Aug. 2024), 3442–3455. https://doi.org/10.14778/3681954.3682012

[63] Xiangpeng Hao, Xinjing Zhou, Xiangyao Yu, and Michael Stonebraker. 2024. Towards Buffer Management with Tiered Main Memory. *Proc. ACM Manag. Data* 2, 1, Article 31 (March 2024), 26 pages. https://doi.org/10.1145/3639286

[64] Yongjun He, Jiacheng Lu, and Tianzheng Wang. 2020. CoroBase: coroutine-oriented main-memory database engine. *Proc. VLDB Endow.* 14, 3 (Nov. 2020), 431–444. https://doi.org/10.14778/3430915.3430932

[65] Yibo Huang, Haowei Chen, Newton Ni, Yan Sun, Vijay Chidambaram, Dixin Tang, and Emmett Witchel. 2025. Tigon: A Distributed Database for a CXL Pod. (July 2025). https://www.usenix.org/conference/osdi25/presentation/huang-yibo

[66] JP Jiang. 2024. CXL Switch for Scalable & Composable Memory Pooling/Sharing. https://files.futurememorystorage.com/proceedings/2024/20240807_CXLT-202-1_Jiang.pdf.

[67] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 521–534. https://doi.org/10.1145/3079856.3080245

[68] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 339–350. https://doi.org/10.1145/1807167.1807206

[69] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. 2021. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 715–728. https://www.usenix.org/conference/atc21/presentation/kim-jonghyeon

[70] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 424–439. https://doi.org/10.1145/3477132.3483589

[71] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the walkers: accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (Davis, California) *(MICRO-46)*. Association for Computing Machinery, New York, NY, USA, 468–479. https://doi.org/10.1145/2540708.2540748

[72] Vamsee Reddy Kommareddy, Simon David Hammond, Clayton Hughes, Ahmad Samih, and Amro Awad. 2019. Page migration support for disaggregated non-volatile memories. In *Proceedings of the International Symposium on Memory Systems* (Washington, District of Columbia, USA) *(MEMSYS '19)*. Association for Computing Machinery, New York, NY, USA, 417–427. https://doi.org/10.1145/3357526.3357543

[73] Michael Körber, Nikolaus Glombiewski, and Bernhard Seeger. 2021. Index-Accelerated Pattern Matching in Event Stores. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1023–1036. https://doi.org/10.1145/3448016.3457245

[74] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, et al. 2019. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 317–330.

[75] Leslie Lamport. [n.d.]. The TLA Home Page. https://lamport.azurewebsites.net/tla/tla.html. Accessed: 2025-12-19.

[76] Donghun Lee, Thomas Willhalm, Minseon Ahn, Suprasad Mutalik Desai, Daniel Booss, Navneet Singh, Daniel Ritter, Jungmin Kim, and Oliver Rebholz. 2023. Elastic Use of Far Memory for In-Memory Database Management Systems. In *Proceedings of the 19th International Workshop on Data Management on New Hardware* (Seattle, WA, USA) *(DaMoN '23)*. Association for Computing Machinery, New York, NY, USA, 35–43. https://doi.org/10.1145/3592980.3595311

[77] Sangjin Lee, Alberto Lerner, Philippe Bonnet, and Philippe Cudré-Mauroux. 2024. Database Kernels: Seamless Integration of Database Systems and Fast Storage via CXL.. In *CIDR*.

[78] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 257–270. https://www.usenix.org/conference/fast17/technical-sessions/presentation/lee-se-kwon

[79] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 17–34.

[80] Viktor Leis. 2024. LeanStore: A High-Performance Storage Engine for NVMe SSDs. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 4536–4545. https://doi.org/10.14778/3685800.3685915

[81] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 38–49.

[82] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware* (San Francisco, California) *(DaMoN '16)*. Association for Computing Machinery, New York, NY, USA, Article 3, 8 pages. https://doi.org/10.1145/2933349.2933352

[83] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 447–461. https://doi.org/10.1145/3341301.3359628

[84] Baptiste Lepers and Willy Zwaenepoel. 2023. Johnny Cache: the End of DRAM Cache Conflicts (in Tiered Main Memory Systems). In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 519–534. https://www.usenix.org/conference/osdi23/

presentation/lepers

[85] Alberto Lerner and Gustavo Alonso. 2024. CXL and the Return of Scale-Up Database Engines. *Proc. VLDB Endow.* 17, 10 (Aug. 2024), 2568–2575. https://doi.org/10.14778/3675034.3675047

[86] Jure Leskovec, Lars Backstrom, and Jon Kleinberg. 2009. Meme-tracking and the dynamics of the news cycle. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Paris, France) *(KDD '09)*. Association for Computing Machinery, New York, NY, USA, 497–506. https://doi.org/10.1145/1557019.1557077

[87] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, USA, 302–313. https://doi.org/10.1109/ICDE.2013.6544834

[88] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. 2016. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 355–370. https://doi.org/10.1145/2882903.2882949

[89] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. ROLEX: a scalable RDMA-oriented learned key-value store for disaggregated memory systems. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) *(FAST'23)*. USENIX Association, USA, Article 7, 15 pages.

[90] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: a memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/2043556.2043558

[91] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: a holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Seattle, WA) *(NSDI'14)*. USENIX Association, USA, 429–444.

[92] Compute Express Link. 2024. Compute Express Link (CXL). https://computeexpresslink.org/.

[93] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+Trees: optimizing persistent index performance on 3DXPoint memory. *Proc. VLDB Endow.* 13, 7 (mar 2020), 1078–1090. https://doi.org/10.14778/3384345.3384355

[94] Jinshu Liu, Hamid Hadian, Hanchen Xu, Daniel S. Berger, and Huaicheng Li. 2024. Dissecting CXL Memory Performance at Scale: Analysis, Modeling, and Optimization. arXiv:2409.14317 [cs.OS] https://arxiv.org/abs/2409.14317

[95] Jinshu Liu, Hamid Hadian, Hanchen Xu, and Huaicheng Li. 2025. Tiered Memory Management Beyond Hotness. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. Introduces a new performance metric, amortized offcore latency (AOL), and proposes two tiering mechanisms, SOAR and ALTO, that leverage this metric to improve data placement and page migration in tiered memory systems.

[96] Baotong Lu, Kaisong Huang, Chieh-Jan Mike Liang, Tianzheng Wang, and Eric Lo. 2024. DEX: Scalable Range Indexing on Disaggregated Memory. *Proc. VLDB Endow.* 17, 10 (Aug. 2024), 2603–2616. https://doi.org/10.14778/3675034.3675050

[97] Kai Lu, Siqi Zhao, Haikang Shan, Qiang Wei, Guokuan Li, Jiguang Wan, Ting Yao, Huatao Wu, and Daohui Wang. 2024. Scythe: A Low-latency RDMA-enabled Distributed Transaction System for Disaggregated Memory. *ACM Trans. Archit. Code Optim.* 21, 3, Article 57 (Sept. 2024), 26 pages. https://doi.org/10.1145/3666004

[98] Xuchuan Luo, Jiacheng Shen, Pengfei Zuo, Xin Wang, Michael R. Lyu, and Yangfan Zhou. 2024. CHIME: A Cache-Efficient and High-Performance Hybrid Index on Disaggregated Memory. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 110–126. https://doi.org/10.1145/3694715.3695959

[99] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R. Lyu, and Yangfan Zhou. 2023. SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 553–571. https://www.usenix.org/conference/osdi23/presentation/luo

[100] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. 2021. ROART: Range-query Optimized Persistent ART. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 1–16. https://www.usenix.org/conference/fast21/presentation/ma

[101] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, Bern, Switzerland, 183–196.

[102] Adnan Maruf, Ashikee Ghosh, Janki Bhimani, Daniel Campello, Andy Rudoff, and Raju Rangaswami. 2022. MULTI-CLOCK: Dynamic Tiering for Hybrid Memory Systems.. In *HPCA*. 925–937.

[103] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 742–755. https://doi.org/10.1145/3582016.3582063

[104] Zviad Metreveli, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. CPHASH: a cache-partitioned hash table. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New Orleans, Louisiana, USA) *(PPoPP '12)*. Association for Computing Machinery, New York, NY, USA, 319–320. https://doi.org/10.1145/2145816.2145874

[105] Alexander Milevski. 2025. avl: Fast AVL tree for Node and browser. https://github.com/w8r/avl. Accessed: 2025-08-28.

[106] Don Moon, Daniel Byrne, and Sounak Gupta. 2023. More cache for less cash: CXL Memory Bandwidth and Capacity Expansion in Software Caches. In *2023 OCP Global Summit - Server: Composable Memory System (CMS)*. OCP. Presented by Don Moon (SK hynix), Daniel Byrne (Intel), and Sounak Gupta (Intel).

[107] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* (Washington, D.C., USA) *(SIGMOD '93)*. Association for Computing Machinery, New York, NY, USA, 297–306. https://doi.org/10.1145/170035.170081

[108] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 371–386. https://doi.org/10.1145/2882903.2915251

[109] Zhenlin Qi, Shengan Zheng, Ying Huang, Yifeng Hui, Bowen Zhang, Linpeng Huang, and Hong Mei. 2025. Chrono: Meticulous Hotness Measurement and Flexible Page Migration for Memory Tiering. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 835–853. https://doi.org/10.1145/3689031.3717462

[110] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. 2023. Hermit: Low-Latency, High-Throughput, and Transparent Remote Memory via Feedback-Directed Asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 181–198. https://www.usenix.org/conference/nsdi23/presentation/qiao

[111] Hongliang Qu and Zhibin Yu. 2024. WASP: Workload-Aware Self-Replicating Page-Tables for NUMA Servers. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 1233–1249. https://doi.org/10.1145/3620665.3640369

[112] Jun Rao and Kenneth A. Ross. 1999. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 78–89.

[113] Jun Rao and Kenneth A. Ross. 2000. Making B+- trees cache conscious in main memory. *SIGMOD Rec.* 29, 2 (May 2000), 475–486. https://doi.org/10.1145/335191.335449

[114] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 392–407. https://doi.org/10.1145/3477132.3483550

[115] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. 2024. MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 803–817. https://doi.org/10.1145/3627703.3650075

[116] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: high-performance, application-integrated far memory. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 18, 18 pages.

[117] Jee Ho Ryoo, Lizy K. John, and Arkaprava Basu. 2018. A Case for Granularity Aware Page Migration. In *Proceedings of the 2018 International Conference on Supercomputing* (Beijing, China) *(ICS '18)*. Association for Computing Machinery, New York, NY, USA, 352–362. https://doi.org/10.1145/3205289.3208064

[118] Tomer Shanny and Adam Morrison. 2022. Occualizer: Optimistic Concurrent Search Trees From Sequential Code. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 321–337. https://www.usenix.org/conference/osdi22/presentation/shanny

[119] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Yuxin Su, Jiazhen Gu, Hao Feng, Yangfan Zhou, and Michael R. Lyu. 2023. Ditto: An Elastic and Adaptive Memory-Disaggregated Caching System. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 675–691. https://doi.org/10.1145/3600006.3613144

[120] Yongju Song, Wook-Hee Kim, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. Prism: Optimizing Key-Value Store for Modern Heterogeneous Storage Devices. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 588–602. https://doi.org/10.1145/3575693.3575722

[121] Jeffrey R. Spirn and Shalom Tsur. 1985. Memory management for B-trees. *Perform. Eval.* 5, 3 (Aug. 1985), 159–174. https://doi.org/10.1016/0166-5316(85)90010-0

[122] Foteini Strati, Christina Giannoula, Dimitrios Siakavaras, Georgios Goumas, and Nectarios Koziris. 2019. An adaptive concurrent priority queue for NUMA architectures. In *Proceedings of the 16th ACM International Conference on Computing Frontiers* (Alghero, Italy) *(CF '19)*. Association for Computing Machinery, New York, NY, USA, 135–144. https://doi.org/10.1145/3310273.3323164

[123] Yan Sun, Jongyul Kim, Zeduo Yu, Jiyuan Zhang, Siyuan Chai, Michael Jaemin Kim, Hwayong Nam, Jaehyun Park, Eojin Na, Yifan Yuan, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2025. M5: Mastering Page Migration and Memory Management for CXL-based Tiered Memory Systems. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 604–621. https://doi.org/10.1145/3676641.3711999

[124] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, et al. 2023. Demystifying CXL memory with genuine CXL-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 105–121.

[125] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: a scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN symposium on principles and practice of parallel programming*. 308–320.

[126] Transaction Processing Performance Council. 2010. TPC-C Benchmark Specification, Version 5.11.0. https://www.tpc.org/tpcc/. 2025-08-28.

[127] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713

[128] UncP. 2019. aili: the fastest in-memory index in the East. https://github.com/UncP/aili. Accessed: 2025-08-28.

[129] Rik van Riel and Vinod Chegu. 2014. Automatic NUMA balancing. In *Red Hat Summit*.

[130] Midhul Vuppalapati and Rachit Agarwal. 2024. Tiered Memory Management: Access Latency is the Key!. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 79–94. https://doi.org/10.1145/3694715.3695968

[131] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2023. Canvas: Isolated and Adaptive Swapping for Multi-Applications on Remote Memory. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 161–179. https://www.usenix.org/conference/nsdi23/presentation/wang-chenxi

[132] Jing Wang, Qing Wang, Yuhao Zhang, and Jiwu Shu. 2025. Deft: A Scalable Tree Index for Disaggregated Memory. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 886–901. https://doi.org/10.1145/3689031.3696062

[133] Qiuping Wang, Jinhong Li, Patrick P. C. Lee, Tao Ouyang, Chao Shi, and Lilong Huang. 2022. Separating Data via Block Invalidation Time Inference for Write Amplification Reduction in Log-Structured Storage. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 429–444. https://www.usenix.org/conference/fast22/presentation/wang

[134] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1033–1048. https://doi.org/10.1145/3514221.3517824

[135] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 473–488. https://doi.org/10.1145/3183713.3196895

[136] Johannes Weiner. 2022. [PATCH] mm: mempolicy: N:M interleave policy for tiered memory nodes. https://lore.kernel.org/linux-mm/20220607171949.85796-1-hannes@cmpxchg.org/.

[137] Marcel Weisgut, Daniel Ritter, Pınar Tözün, Lawrence Benson, and Tilmann Rabl. 2025. CXL Memory Performance for In-Memory Data Processing. *Proceedings of the VLDB Endowment* 18, 9 (2025), 3119–3133. https://doi.org/10.14778/3746405.3746432 PVLDB Artifact Availability: The source code have been made available at https://github.com/hpides/cxlbench/tree/paper/vldb25 (microbenchmarks) and https://github.com/hyrise/hyrise/tree/paper/vldb25 (Hyrise).

[138] Kai Wu, Yingchao Huang, and Dong Li. 2017. Unimem: runtime data managementon non-volatile memory-based heterogeneous main memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 58, 14 pages. https://doi.org/10.1145/3126908.3126923

[139] Xingbo Wu, Fan Ni, and Song Jiang. 2019. Wormhole: A Fast Ordered Index for In-memory Data Management. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 18, 16 pages. https://doi.org/10.1145/3302424.3303955

[140] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. 2024. Nomad: Non-Exclusive Memory Tiering via Transactional Page Migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 19–35. https://www.usenix.org/conference/osdi24/presentation/xiang

[141] Dong Xu, Junhee Ryu, Kwangsik Shin, Pengfei Su, and Dong Li. 2024. FlexMem: Adaptive Page Profiling and Migration for Tiered Memory. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 817–833. https://www.usenix.org/conference/atc24/presentation/xu-dong

[142] Bin Yan, Youyou Lu, Qing Wang, Minhui Xie, and Jiwu Shu. 2023. Patronus: high-performance and protective remote memory. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) *(FAST'23)*. USENIX Association, USA, Article 20, 16 pages.

[143] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 331–345. https://doi.org/10.1145/3297858.3304024

[144] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: reducing consistency cost for NVM-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Santa Clara, CA) *(FAST'15)*. USENIX Association, USA, 167–181.

[145] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. (Nov. 2020), 191–208. https://www.usenix.org/conference/osdi20/presentation/yang

[146] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Gerry Fan, Yang Kong, Bo Wang, Jing Fang, Yuhui Wang, Tao Huang, Wenpu Hu, Jim Kao, and Jianping Jiang. 2025. Unlocking the Potential of CXL for Disaggregated Memory in Cloud-Native Databases. In *Companion of the 2025 International Conference on Management of Data* (Berlin, Germany) *(SIGMOD/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 689–702. https://doi.org/10.1145/3722212.3724460

[147] Geoffrey X. Yu, Markos Markakis, Andreas Kipf, Per-Åke Larson, Umar Farooq Minhas, and Tim Kraska. 2022. TreeLine: an update-in-place key-value store for modern storage. *Proc. VLDB Endow.* 16, 1 (Sept. 2022), 99–112. https://doi.org/10.14778/3561261.3561270

[148] Adar Zeitak and Adam Morrison. 2021. Cuckoo Trie: exploiting memory-level parallelism for efficient DRAM indexing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 147–162.

[149] Bowen Zhang, Shengan Zheng, Zhenlin Qi, and Linpeng Huang. 2022. NBTree: a lock-free PM-friendly persistent B+-tree for eADR-enabled PM systems. *Proc. VLDB Endow.* 15, 6 (feb 2022), 1187–1200. https://doi.org/10.14778/3514061.3514066

[150] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1567–1581. https://doi.org/10.1145/2882903.2915222

[151] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. 2015. In-Memory Big Data Management and Processing: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (2015), 1920–1948. https://doi.org/10.1109/TKDE.2015.2427795

[152] Huanchen Zhang, Xiaoxuan Liu, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Order-Preserving Key Compression

for In-Memory Search Trees. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1601–1615. https://doi.org/10.1145/3318464.3380583

[153] Lei Zhang, Reza Karimi, Irfan Ahmad, and Ymir Vigfusson. 2020. Optimal data placement for heterogeneous cache, memory, and storage systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 1 (2020), 1–27.

[154] Weidong Zhang, Erci Xu, Qiuping Wang, Xiaolu Zhang, Yuesheng Gu, Zhenwei Lu, Tao Ouyang, Guanqun Dai, Wenwen Peng, Zhe Xu, Shuo Zhang, Dong Wu, Yilei Peng, Tianyun Wang, Haoran Zhang, Jiasheng Wang, Wenyuan Yan, Yuanyuan Dong, Wenhui Yao, Zhongjie Wu, Lingjun Zhu, Chao Shi, Yinhu Wang, Rong Liu, Junping Wu, Jiaji Zhu, and Jiesheng Wu. 2024. What's the story in EBS glory: evolutions and lessons in building cloud block store. In *Proceedings of the 22nd USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) *(FAST '24)*. USENIX Association, USA, Article 17, 16 pages.

[155] Zhou Zhang, Zhaole Chu, Peiquan Jin, Yongping Luo, Xike Xie, Shouhong Wan, Yun Luo, Xufei Wu, Peng Zou, Chunyang Zheng, Guoan Wu, and Andy Rudoff. 2022. PLIN: a persistent learned index for non-volatile memory with high performance and instant recovery. *Proc. VLDB Endow.* 16, 2 (Oct. 2022), 243–255. https://doi.org/10.14778/3565816.3565826

[156] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. 2024. Managing Memory Tiers with CXL in Virtualized Environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 37–56. https://www.usenix.org/conference/osdi24/presentation/zhong-yuhong

[157] Yijie Zhong, Minqiang Zhou, Zhirong Shen, and Jiwu Shu. 2024. UniMem: Redesigning Disaggregated Memory within A Unified Local-Remote Memory Hierarchy. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 463–477. https://www.usenix.org/conference/atc24/presentation/zhong

[158] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: differential indexing for persistent memory. *Proc. VLDB Endow.* 13, 4 (dec 2019), 421–434. https://doi.org/10.14778/3372716.3372717

[159] Zhe Zhou, Yiqi Chen, Tao Zhang, Yang Wang, Ran Shu, Shuotao Xu, Peng Cheng, Lei Qu, Yongqiang Xiong, Jie Zhang, and Guangyu Sun. 2024. NeoMem: Hardware/Software Co-Design for CXL-Native Memory Tiering. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1518–1531. https://doi.org/10.1109/MICRO61859.2024.00111

[160] Ziegler, Tobias and Tumkur Vani, Sumukha and Binnig, Carsten and Fonseca, Rodrigo and Kraska, Tim. 2019. Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 741–758. https://doi.org/10.1145/3299869.3300081