Optimizing Tree-structure Indexes for CXL-based Heterogeneous Memory with SINLK

Haoru Zhao, Mingkai Dong✉, Fangnuo Wu, Haibo Chen
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

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 SINLK, a tree-structure aware, node-grained data placement scheme for tree-structure indexes on CXL-HM. With SINLK, developers can easily adapt existing tree-structure indexes to CXL-HM. We have integrated the B+tree and radix tree with SINLK to demonstrate its effectiveness. Evaluations show that SINLK 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 in YCSB and real-world workloads.

1 Introduction

In-memory tree-structure indexes [1–11] (e.g., B+tree) are fundamental in various domains, such as databases [12–18], storage engines [19, 20] and big data analytics [21–25]. Their efficiency is crucial for overall system performance [3, 4, 13, 26]. With the increasing data scale in datacenters, the memory demand of tree-structure indexes is growing rapidly [3, 27–32]. In order to expand memory capacity and enhance memory elasticity, heterogeneous memory (HM) has emerged. Many efforts [33–48] have been made to build tree-structure indexes on HM to leverage the large memory capacity while mitigating the slow memory's impact on performance.

✉Corresponding author: Mingkai Dong (mingkaidong@sjtu.edu.cn).

Nowadays, Compute Express Link (CXL) [49] becomes a promising HM solution to expand memory capacity. For example, Samsung CMM-B [50] scales memory capacity up to 24 TB with CXL memory pool. Due to CXL's cache-coherent memory semantics (i.e., CXL provides byte-addressable memory accessed in a cache-coherent way in the same address space as CPU-attached DRAM, i.e., fast memory), large tree-structure indexes designed for fast memory can easily adapt to CXL-HM, where fast 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 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 [33–35] are not suitable for CXL-HM because CXL-attached memory lacks local processors like NUMA nodes. Indexes optimized for RDMA [36–40] 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 [42–48], 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, applying existing CXL-HM data placement schemes [51–55] to tree-structure indexes is also inefficient. As these schemes manage data at page granularity while tree nodes are typically smaller, this granularity mismatch leads to inaccurate hotness detection, increased migration overhead, and suboptimal data placement.

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 to adapt to 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.

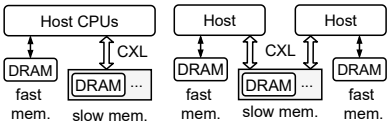
We design SINLK, a node-grained, tree-structure aware data placement scheme to optimize tree-structure indexes on CXL-HM. First, SINLK introduces *leaf-centric access tracking*, capturing the hotness of all access paths while minimizing interference with tree operations by only tracking leaf nodes. Second, SINLK incorporates *structure-aware migration* which considers node relationships instead of 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, SINLK 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 indexes.

To demonstrate SINLK's effectiveness and generality, we integrate SINLK with two widely used tree-structure indexes—B+ tree [1] and radix tree [10, 11]—with less than 3% trees' internal code modification. Evaluation on a real CXL platform with synthetic workloads shows that SINLK achieves up to 133% higher throughput and 67% lower P99 latency than the SOTA data placement scheme MEMTIS [52]. For YCSB [56], SINLK achieves up to 71% and 60% higher throughput than MEMTIS and optimized HM index (optimized PACTree [57]), respectively. For real-world workloads [58], SINLK achieves up to 66% higher throughput and 81% lower P99 latency than MEMTIS, with 44% higher throughput and 63% lower P99 latency than 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 SINLK, an efficient and generic data placement scheme for tree-structure indexes on CXL-HM.
- **Evaluation.** We integrate SINLK with B+ tree and radix tree. Evaluation on a real CXL platform with various workloads demonstrates the advantages of SINLK.

[1]We mainly focus on tree-structure indexes that store values in leaves, the most common form of such indexes [10].



(a) CXL 1.0/1.1 **(b)** CXL 2.0/3.0

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

(c) Performance

Figure 1. CXL-based heterogeneous memory system. *(a,b) System architecture. (c) Comparison of fast and slow memory latency and bandwidth tested by Intel MLC [63] on our CXL1.1 platform.*

2 Background and Motivation

2.1 CXL-based Heterogeneous Memory

CXL [49, 59] 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 [60] allows host CPUs to access CXL-attached memory with cache-coherent load/store. CXL-HM with CXL 2.0/3.0 [61, 62] (Figure 1(b)) further allows the hosts to expand the memory by CXL memory pool, enhancing memory expansion's flexibility. As CXL-HM provides a unified physical address space of fast memory and CXL-attached memory, existing tree-structure indexes for fast memory can be effortlessly migrated to CXL-HM.

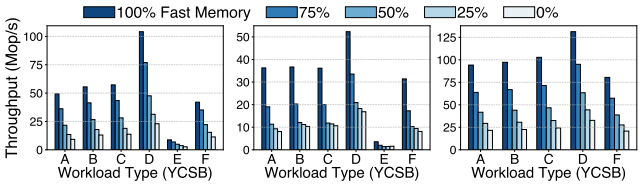
2.2 Tree-structure Indexes on CXL-HM

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. Moreover, this gap will be widened with the introduction of CXL-switch since CXL2.0 [64]. Thus, directly migrating tree-structure indexes designed for fast memory to CXL-HM leads to poor performance. We illustrate this via Masstree [1], Wormhole [2], and adaptive radix tree (ART) [10, 11], three typical tree-structure indexes. The setup of the following experiments is the same as our evaluation (§6.1). As shown in Figure 2[2], when employing the default weighted memory allocation policy [65], the performance of all three indexes drops by about 70% with 25% fast memory usage compared to 100% fast memory across all workloads. Therefore, *optimizing the placement of tree-structure indexes on CXL-HM is essential to achieve better performance.*

2.3 Issues with Optimized HM-Indexes on CXL-HM

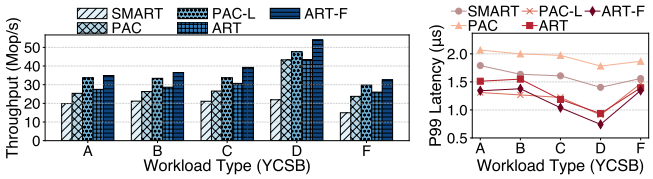
In this section, we study existing tree-structure indexes optimized for other types of HM (e.g., NUMA, RDMA-based HM, PM-based HM) and data placement schemes for CXL-HM to demonstrate that:

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



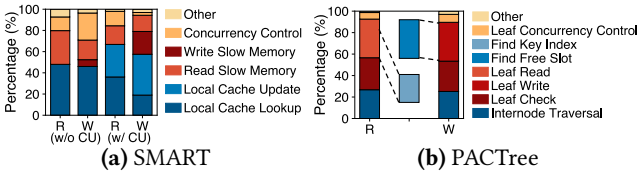
(a) Masstree (b) Wormhole (c) ART

Figure 2. Throughput (tput.) of indexes with different fast memory usage ratios in YCSB [56] benchmark.



(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

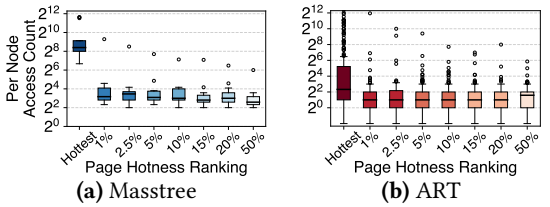
Figure 4. Performance breakdown per operation type. *(w/o CU)&(w/ CU) means without/with local cache update.*

- 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 and semantics. We compare the original ART, the RDMA-optimized ART (SMART [38]), and the PM-optimized ART (PACTree [57]) 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 granularity mismatch between tree nodes and pages.

NUMA-Index. Indexes optimized for NUMA reduce cross-node communication and minimize accesses to remote memory through replication [35, 66, 67], partitioning [33], or delegation [34, 68]. These methods are not applicable to CXL-HM because *CXL-attached memory lacks local processors for data processing, unlike the memory in the NUMA node.*

RDMA-Index Case Study (SMART). SMART optimizes ART's data placement by maintaining a fast memory cache for accessed internal nodes (a.k.a., local cache), which is a common optimization in RDMA-Indexes [37–39, 41]. Figure 3(a) shows that SMART's throughput is only 50%–74% of ART[3]. Our profiling (Figure 4(a)) reveals that the primary overhead (~50%) in SMART comes from the local cache.

[3]The congestion control design of SMART (RDWC) is disabled because it causes about a 15% performance drop when there is no congestion.



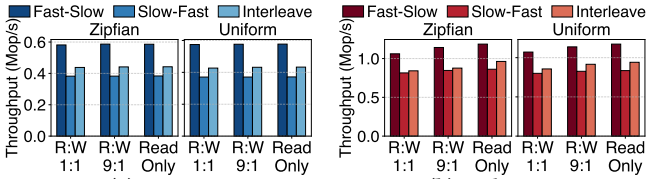
(a) Masstree **(b)** ART

Figure 5. Node access count distribution across pages of varying hotness. *We analyze distribution of node access counts in the page with the highest access frequency (Hottest) and those ranked top 1%–50% by access frequency.*

Since the latency gap between CXL-attached memory and fast memory is much smaller than that of RDMA-connected memory [69], the costs of cache maintenance and querying outweigh the benefits of reducing slow memory accesses. Specifically, while the local cache significantly reduces access to 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 eviction management.

PM-Index Case Study (PACTree). PACTree optimizes ART's data placement by using fat (64 entries), unordered leaf nodes, which is a common optimization [42, 43, 46–48, 70–72] in PM-indexes, along with asynchronous tree structural modification (a.k.a., 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 [31, 73], and the high allocation costs associated with persistence [57, 70].* Although fat, unordered leaf nodes 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) [42, 43, 45–48]. We apply this technique to PACTree (PAC-L in Figure 3). Although it can improve PACTree's performance on CXL-HM, this static placement strategy is insufficient because: (1) It neglects fast memory capacity, which can result in fast memory underutilization or exhaustion. For example, ART outperforms PAC-L when fast memory capacity is larger (ART-F in Figure 3), and PAC-L's performance drops by ~17% when fast memory is full. (2) It ignores workload characteristics, requiring access to the leaf in slow memory for each request.



(a) B+ tree **(b)** Radix tree

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

Applying CXL-HM Data Placement Schemes. Existing data placement schemes [51–55] 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 [52]. 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

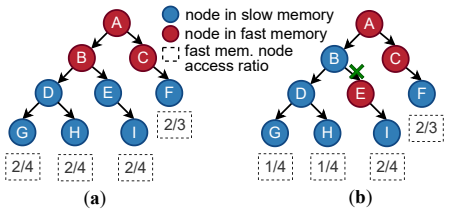
Our key insight is that the efficient placement of tree-structured indexes on CXL-HM should match the tree's inherent characteristics with the features of CXL-HM, which involves: (1) matching the tree's node-granularity data usage with CXL-HM's cache-coherent memory semantics to achieve node-grained 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

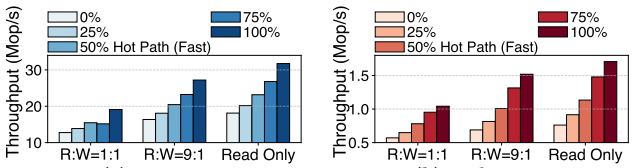
3.1.1 Structural Characteristics. Due to the tree's branching structure and the downward access from its root, nodes at upper levels (i.e., upper nodes) are fewer but have higher access frequency than lower nodes. For instance, in a B+ tree [74] and radix tree [75] with 50 million keys, nodes in the 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 using the 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.



(a) **(b)**

Figure 7. *Single-boundary structure.* *(a) The tree follows this structure. (b) When 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 $\frac{1}{4}$.*



(a) B+ tree **(b)** Radix tree

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

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) illustrates 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 nodes G and H.

3.1.2 Access Characteristics. Trees are accessed from the root downwards, forming access paths from the root to target nodes (e.g., leaf nodes in B+ trees and radix trees). In real-world workloads, where accesses are often skewed [29, 76–79], *certain paths are accessed much more frequently.* We identify these paths as *hot paths.*

To investigate how hot paths can affect the performance of trees on CXL-HM, we evaluate a multi-threaded B+ tree [80] and the radix tree under a skewed workload, where 90% of requests targeted 10% data. We vary the proportion of hot paths in fast memory and measure throughput. As illustrated in Figure 8, tree performance improves across all workloads with the higher proportion of 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. Therefore, although 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 [3, 4, 13, 26]. 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 single-boundary structure and may break the locality of hot paths. When using the path-grained tracking, existing migration mechanisms [31, 51–54, 81] 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. Since 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.

Allocation and migration require complicated control to meet the complexity of tree structure. Given the complexity of trees' hierarchical structure, more parameters are needed 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 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 [52, 53]) or lack coordination between allocation and migration [51], we need proactive and holistic strategies to meet the performance stability requirements of the trees.

4 Design of SINLK

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

The frontend module is invoked in the critical path of the tree operations so that 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.

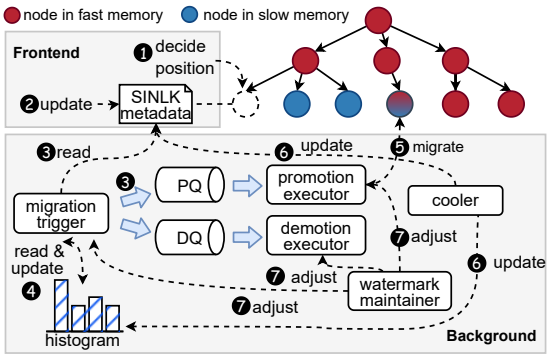


Figure 9. Architecture and interactions of SINLK. *PQ stands for promotion queue, DQ stands for demotion queue.*

The background module consists of several background workers to conduct node migration (§4.2) and system adjustment (§4.3) asynchronously to optimize fast memory usage. The *migration trigger* decides whether migration is needed and picks the appropriate nodes for migration based on their access frequency (❸) and the distribution (i.e., the histogram) of all leaf nodes' access frequency (❹). The *promotion executor* and *demotion executor* get nodes from the corresponding queues and carry out 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 and adjusts 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 introduce 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).

4.1 Lightweight Node Allocation and Tracking

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

Layer-aware Allocation. Guided by the *layer principle*, SINLK 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, SINLK maintains the highest level of fast memory node allocation, L_{fast}, which is adjusted based on current fast memory usage.

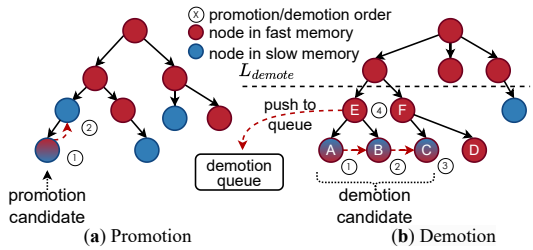


Figure 10. Promotion and demotion procedures.

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.

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

4.2 Structure-Aware Migration

SINLK 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, SINLK employs a histogram-based approach to precisely identify hot paths and cold nodes (§4.2.3). To achieve efficient migration, SINLK 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 memory. As shown in Figure 10(a), SINLK'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 node in fast memory is identified as a cold node, it is essential to demote not only this leaf node but also some of its ancestors to better utilize the limited fast memory resources. The specific number of ancestor nodes to be migrated 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 illustrated in Figure 10(b), nodes A, B, and C are current demotion candidates. Migrating node C and then its parent node F breaches the *single-boundary structure* because node D is still in fast memory. Therefore, to demote parent node F, node D must also be demoted simultaneously. However, if node D, which is not a demotion candidate at that time, is demoted, it may inadvertently migrate a node in a hot path to slow memory, potentially degrading system performance.

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
```

To address this, SINLK adopts the demotion procedure as depicted in algorithm 1: once a node is demoted, its parent node is queued for demotion (line 13). The decision to demote the parent node is deferred until all its cold children have been processed by the demotion executor. If all of the parent node'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 the *single-boundary structure* (line 9). Additionally, the lowest level for demotion is denoted by L_{demote}, nodes below L_{demote} will not participate in the demotion process (line 5). Therefore, for the tree in Figure 10(b), the demotion process follows the labels ①~④.

4.2.3 Histogram-based Hotness Identification. Like prior studies [52, 82, 83], SINLK employs a logarithmic histogram to represent access frequency distribution, thereby identifying hot paths and cold nodes for migration.

The migration trigger and the cooler collaborate to maintain the histogram. The migration trigger updates the histogram with 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

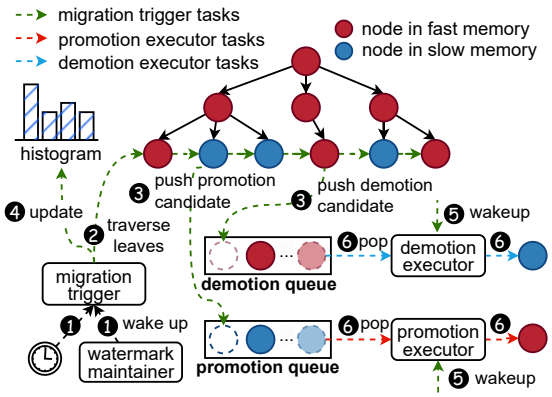


Figure 11. Migration workflow.

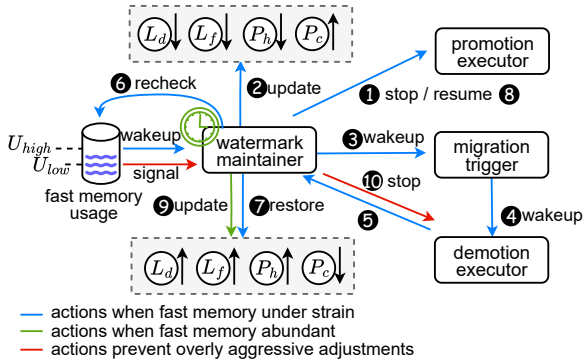


Figure 12. System parameter adjustments and the collaboration of workers in the hyper watermark mechanism. *L_d is L_{demote}, L_f is L_{fast}, P_c is P_{cold}, and P_h is P_{hot}.*

logarithmic scale, the reduction can be efficiently conducted by shifting all bins to the left by one position.

Based on the histogram, SINLK uses a hot threshold T_{hot} and a cold threshold T_{cold} to identify hot paths and cold nodes for migration. Leaf nodes with access frequency higher than T_{hot} are leaves of hot paths; while leaf nodes with access frequency lower than T_{cold} are deemed cold nodes.

As static thresholds cannot adapt to dynamic workloads, SINLK periodically updates the hot and cold thresholds using two parameters, P_{cold} and P_{hot}. These two parameters[4] determine the percentage of cold and hot leaves, respectively. SINLK scans the histogram to set T_{hot} to a value where nodes with frequency higher than T_{hot} just exceeds P_{hot}. Similarly, T_{cold} is set to a value where nodes with frequency lower than T_{cold} just fall below P_{cold}. Note that the interval between T_{hot} and T_{cold} prevents premature classification of less accessed nodes as cold, avoiding aggressive node 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 leaf nodes (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 respective demotion/promotion queue (❸). During scanning, the migration trigger also updates the histogram recording access frequency distribution (❹). When the queue is too long or the migration trigger has checked all leaf nodes, the corresponding executor is awakened (❺) to carry out the migration according to the procedures described in §4.2.1 and §4.2.2 (❻). It is worth noting 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

To achieve high performance, SINLK selects appropriate initial positions for nodes based on their level (§4.1) and periodically migrates cold nodes and hot paths (§4.2). These procedures involve a series of tree-related parameters, including L_{fast} (§4.1), P_{cold}/P_{hot} (§4.2), and L_{demote} (§4.2). Manually adjusting these parameters is challenging, as it is difficult to adapt to workload changes and may cause conflicting fast memory usage intentions between node migration and allocation, causing performance instability.

Therefore, SINLK introduces a hyper watermark mechanism that synthetically adjusts these parameters and coordinates the behavior of background workers based on the current fast memory usage. It ensures consistent fast memory usage trends for node allocation and migration under various conditions. By trying to maximize the utilization of fast memory without exhausting it, the hyper watermark mechanism effectively prevents performance degradation from burst migrations and other extreme behaviors.

4.3.1 Asymmetric Adjustment. The impacts of fast memory strain and abundance are different. Depletion of fast memory can force the new upper-level nodes to be allocated to slow memory, obstruct hot path promotions, or trigger burst demotions, resulting in severe performance degradation that is unacceptable for high-performance tree-structure indexes. Conversely, when fast memory is abundant, although it leaves some fast memory underutilized, it does not have such a significant impact on performance stability and is advantageous for handling burst insertions of new nodes and promoting hot paths in the future.

Therefore, SINLK employs asymmetric adjustments. When the fast memory usage ratio exceeds the high watermark U_{high}, SINLK adopts an aggressive adjustment strategy to promptly bring the fast memory usage back to a reasonable range. Conversely, when the fast memory usage ratio falls below the low watermark U_{low}, a conservative adjustment is taken to optimize fast memory usage while reserving space for new nodes and hot path promotions.

4.3.2 Fast Memory Under Strain. When the fast memory usage ratio exceeds U_{high}, the watermark maintainer is immediately awakened to regulate the system parameters and background workers. The blue lines in Figure 12 illustrate

[4] P_{cold} and P_{hot} are initialized based on the maximum fast memory usage and further dynamically adjust through the hyper watermark mechanism (§4.3) based on the current fast memory usage.

this process. First, the watermark maintainer temporarily halts all ongoing promotion tasks (❶). Subsequently, it adjusts system parameters to increase demotion likelihood by increasing P_{cold} and decreasing P_{hot}, intensify demotion by decreasing L_{demote}, and reduce the number of nodes initialized in fast memory by decreasing L_{fast} (❷). The watermark maintainer then awakens the migration trigger to select cold nodes for demotion (❸), which in turn activates the demotion executor to migrate these cold nodes (❹). Following this, the watermark maintainer waits for the demotion executor to complete the demotion tasks (❺), then rechecks the fast memory usage (❻), and repeats steps ❷–❻ until the fast memory usage ratio falls below U_{high}. Upon achieving this, the watermark maintainer resets these system parameters to their pre-adjustment states (❼) and re-enables the execution of the promotion executor (❽).

4.3.3 Fast Memory Abundant.

As depicted by the green line and clock icon of Figure 12, the watermark maintainer periodically checks if the fast memory usage ratio drops below U_{low}. If so, it increases P_{hot}, L_{demote}, L_{fast}; and decreases P_{cold} (❾). This raises chances of promotion and fast memory allocation while reducing demotion likelihood. This approach is deemed conservative as it only adjusts parameters without instantly triggering migration executions.

4.3.4 Optimizations.

When fast memory is under strain, SINLK uses an aggressive adjustment strategy that sometimes over-identifies cold nodes, leading to excessive demotions and potentially dropping the fast memory usage ratio below U_{low}. To mitigate this and protect performance, SINLK acts promptly when the fast memory usage ratio falls below U_{low} and adjustments for prior memory pressure are still in progress. The watermark maintainer is signaled to halt current demotion tasks (❿). Subsequently, SINLK restores system parameters to their original states (❼) and reactivates the promotion executor (❽), safeguarding system performance through efficient fast memory utilization.

When fast memory is under pressure, the watermark maintainer may need multiple attempts, involving several adjustments to system parameters. If the adjustments to L_{demote} and L_{fast} are too aggressive, they can lead to excessive demotion and an overly pessimistic fast memory usage trend, hurting performance. To prevent this, we limit changes to L_{demote} and L_{fast} based on the maximum fast memory usage, avoiding excessively aggressive adjustments.

The demotion procedure in §4.2.2 only considers demoting the ancestors of leaf nodes in fast memory. To prevent cold ancestors of slow memory leaf nodes from being stuck in fast memory and to demote more cold internal nodes under fast memory strain, the migration trigger adds some cold leaf nodes in slow memory to the demotion queue, creating the opportunity to demote their ancestors later.

Table 1. Workload composition. *SP is Skewed Partition.*

Workload	Request Composition
Update Heavy (SP, YCSB-A)	50% Read, 50% Update
Read Mostly (SP, YCSB-B)	95% Read, 5% Update
Read Only (SP, YCSB-C)	100% Read
With Insert (SP), 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

5 Implementation

SINLK Framework. We provide a lightweight and easy-to-use framework for integrating SINLK into various existing tree-structure indexes. For SINLK 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 the access frequency. This memory overhead is acceptable because tree nodes are typically equal to or larger than the cacheline [84], and sometimes the node size is unaffected due to memory alignment. For the frontend module, SINLK framework provides helper functions that implement layer-aware allocation and leaf-centric access tracking for developers to use. Additionally, developers need to implement tree-specific logic in the interfaces defined by SINLK framework for the background module to interact with the tree. Notably, while SINLK requires some user-implemented functions, many of them can leverage the existing logic of the tree, such as node splitting for migration and the iterator for leaf traversal.

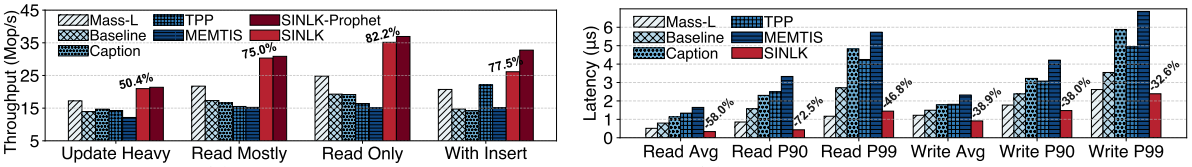
Integration. We have integrated ART and Masstree (a B+ tree variant optimized for multicore architecture) with SINLK framework, termed as S-ART and S-Masstree, respectively. In contrast to Masstree's 25 k LoC and ART's 1.7 k LoC, SINLK only modifies ~150 and ~50 LOC respectively. Moreover, we use ~1.8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

Concurrency Control. To prevent concurrent writes to the migrating node, we lock the node and its parent until the node's migration completes. For concurrent reads, we use an unused version bit to indicate the node's migration status. Read correctness is ensured by the optimistic read, which checks version numbers before/after each read and retries if changed. Note that there is no lock contention between background workers, which use atomic operations (e.g., CAS) to access shared state (e.g., parameters, histogram).

Configuration. In SINLK's runtime, 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.

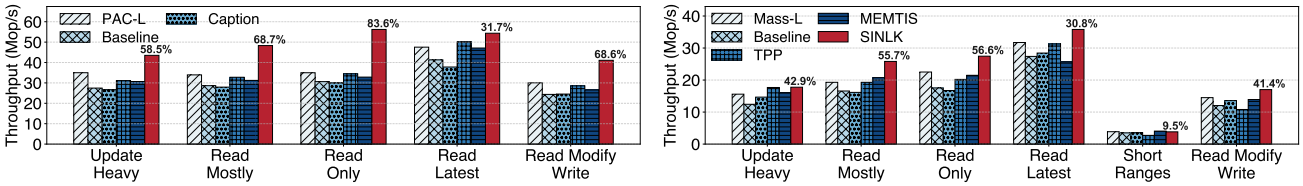
6 Evaluation

In this section, we evaluate SINLK from multiple perspectives to answer the following questions:



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

Figure 13. Performance of SINLK (S-Masstree) and compared systems in the microbenchmark.



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

Figure 14. Throughput of SINLK and compared systems in the macrobenchmark.

- Does SINLK perform well in various scenarios? (§6.2)
- Can SINLK maintain good performance in dynamic changing workloads and different initial system settings? (§6.3)
- How scalable is SINLK when varying the number of threads and dataset sizes? (§6.4)
- How do SINLK's techniques contribute to the final performance, and how does SINLK perform across various key/value sizes? (§6.5)
- How significant is SINLK's overhead? (§6.6)
- How does SINLK perform in real-world workload? (§6.7)

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 memory expansion card [85]). We use a single socket to avoid NUMA effects. All experiments except the concurrency scalability are conducted with 28 threads.

Workloads. We use the Skewed Partition (SP) synthetic workload as the microbenchmark, where 90% of requests target 5% of contiguous keys (termed *hot region*). This pattern closely matches the strong locality of hot keys in production environments [86]. We use YCSB[5] [56] with the default Zipfian distribution as the macrobenchmark. Details of both benchmarks are shown in Table 1. Additionally, we evaluate SINLK using Alibaba Block Traces [58, 87] which is a real-world workload collected from production.

Compared Systems. The baseline uses the default weighted allocation policy [65] to allocate a specified ratio of memory to CXL-attached DRAM. Moreover, we compare SINLK with three SOTA data placement schemes for CXL-HM: TPP [51], MEMTIS [52], and Caption [88]. For S-ART, we compare it with PAC-L, the optimized PACTree [57] with small leaves and fast memory internode (§2.3). For S-Masstree, we compare it with Mass-L (i.e., Masstree with fast memory internode[6]). As Caption does not specify the maximum fast memory usage, we set its limit to be the same as the baseline.

6.2 Overall Performance

6.2.1 Microbenchmark. SINLK-Prophet represents an ideal scenario where all hot paths are allocated in fast memory during initialization. To ensure fair comparison, we measure SINLK-Prophet's fast memory usage (~10% of total memory) and use this fast memory amount as the limit for other systems. Figure 13(a) shows that S-Masstree outperforms the baseline by 50%–82%[7]. Furthermore, SINLK performs similarly to SINLK-Prophet in the first three workloads, showing its ability to effectively identify and promote hot paths at runtime. For *With Insert*, SINLK-Prophet outperforms SINLK because it allocates most new nodes in fast memory, exceeding SINLK's maximum fast memory usage by 1.3–1.5×.

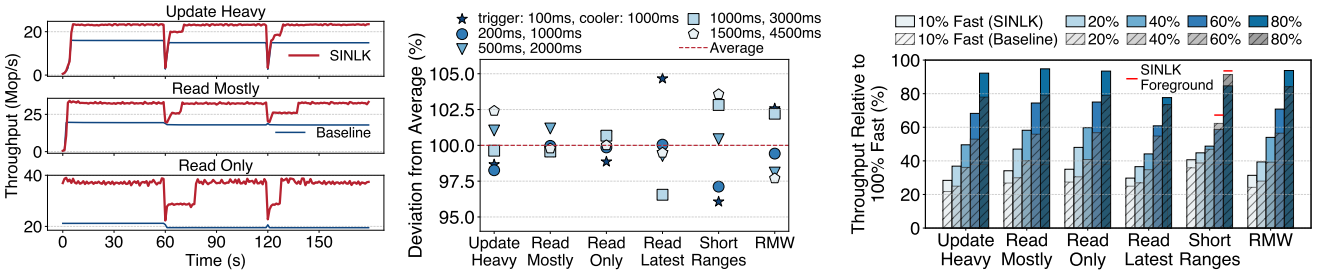
For other systems, Caption performs similarly to the baseline due to its interleave allocation method, which is similar to the baseline. TPP performs well in *With Insert* because it allocates all nodes in fast memory at the beginning when fast memory is abundant. However, TPP struggles to identify cold nodes and demote them in read-heavy workloads, hurting performance. MEMTIS performs comparatively poorly across all workloads because its default usage of huge pages does not adapt well to workload characteristics. Mass-L outperforms other baselines but still lags behind SINLK as static placement cannot adapt to dynamic workloads.

6.2.2 Macrobenchmark In the macrobenchmark, we set the baseline maximum fast memory usage at 20% and use its corresponding memory consumption as the limit for other systems, based on Zipfian access patterns and the dataset size. Note that this setting with low fast memory usage is consistent with previous work [31, 52, 54]. Besides this set-

[5]We don't evaluate *Short Range* for S-ART as its source code lacks scan.

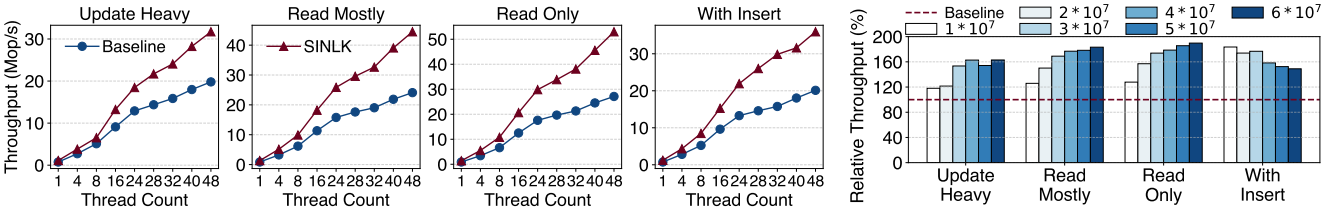
[6]Internodes are allocated to slow memory when fast memory runs out.

[7]In Figure 13, Figure 14, Figure 21(a) and Figure 22(a), the marked improvements are compared with the baseline.



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

Figure 15. Sensitivity analysis of SINLK (S-Masstree).



(a) Scalability with different thread counts **(b)** Scalability with different dataset sizes

Figure 16. Scalability of SINLK (S-Masstree) in microbenchmark.

ting, we further evaluate SINLK's performance across various data scales and fast memory usage later to demonstrate its performance stability (§6.3.3, §6.4 part1).

As shown in Figure 14, S-ART outperforms the baseline by 32%–84%, while S-Masstree outperforms the baseline by 31%–57% (except for *Short Ranges*). *Short Ranges*'s performance depends on the number of leaves in fast memory. Due to the limited fast memory and SINLK prioritizing upper nodes in fast memory, the performance gain in this workload is not as significant. SINLK performs better in read-dominant workloads because migration interferes less with read operations.

PAC-L performs worse than S-ART because its static memory layout is unable to adjust to the dynamic workload. Other data placement systems fail to perform well across all workloads. For instance, while TPP performs similarly to S-Masstree in *Update Heavy*, it fails to maintain comparable performance for *Read Only* and *Read Modify Write*.

6.2.3 Latency Analysis We measure SINLK'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 13(b), SINLK's average read/write latencies are both under 1 μs. Moreover, SINLK sharply reduces read latency, with P90 latency 73% lower than the baseline. SINLK significantly reduces P90/P99 latency versus TPP and MEMTIS by continuously detecting and demoting cold nodes, along with the watermark mechanism, thereby preventing burst migrations.

6.3 Sensitivity Analysis

An ideal data placement scheme should be able to adapt to changing workloads and perform well across various configurations. In SINLK, most parameters are dynamically adjusted during runtime, except for workers' wake-up intervals and maximum fast memory usage. To evaluate SINLK's performance stability, we conduct a sensitivity analysis that shows its performance under dynamic changing workloads (§6.3.1) and different system parameter settings (§6.3.2, §6.3.3).

6.3.1 Dynamic Workload Changes. To simulate dynamic workload changes, we shift the entire hot region in the microbenchmark every 60 s. As shown in Figure 15(a), although SINLK 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 migrates them to fast memory. SINLK can fully recover its performance in up to 17 s, showing its ability to quickly adapt to changes in hot paths for stable high performance in dynamic workloads.

6.3.2 Different Worker Wake-up Intervals. In SINLK, 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. Therefore, we only evaluate the impact of wake-up intervals using five sets of different intervals for the other two workers. Figure 15(b) shows that SINLK's performance is relatively stable to wake-up interval choices, with less than 5% deviations from the average. In real-world scenarios, wake-up intervals can be adjusted based on system conditions, such as extending the intervals when CPU resources are tight to reduce overhead.

6.3.3 Different Maximum Fast Memory Usage To evaluate SINLK's performance with different maximum fast memory usage, Figure 15(c) compares SINLK with the baseline

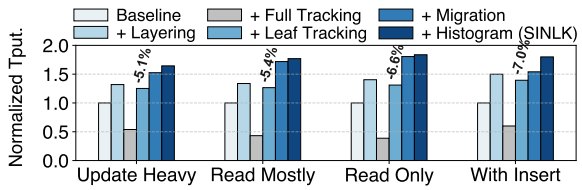


Figure 17. The factor analysis for techniques in SINLK.

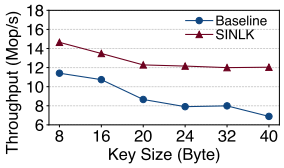


Figure 18. *Impact of key sizes (Micro Update Heavy).*

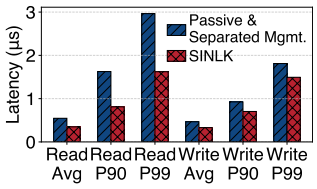


Figure 19. *Impact of passive and separated management (YCSB-D).*

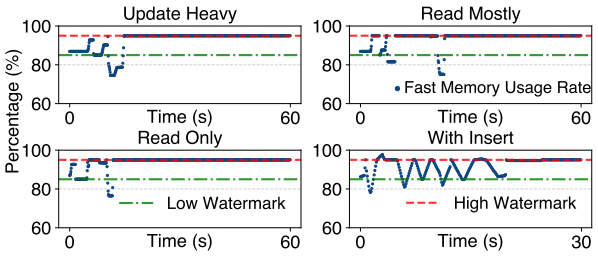


Figure 20. Fast memory usage ratio variation over time.

in the macrobenchmark, varying maximum fast memory usage from 10% to 80% of the baseline's total memory. As the fast memory usage increases, SINLK's performance improves steadily. When 80% data is in fast memory, SINLK's performance is close to all data in fast memory for most workloads. When the maximum fast memory usage exceeds 60%, SINLK slightly lags the baseline in *Short Ranges* because the higher fast memory usage intensifies competition for CPU cache between background workers and the scan operation. Nevertheless, when only the frontend module is enabled (red line in Figure 15(c)), SINLK still outperforms the baseline.

6.4 Scalability Performance

In this section, we evaluate the scalability of SINLK by varying the number of threads and dataset sizes.

Concurrency Scalability. Concurrency scalability is a key metric for assessing index performance [1, 2, 8]. Comparing SINLK with the baseline (Figure 16(a)) reveals that SINLK not only guarantees the index's scalability but also derives amplified performance gains over the baseline with more threads. The gains is amplified because SINLK benefits each thread, accumulating as the thread count grows. At 48 threads, SINLK's throughput is 28.6–38.1 × that of a single thread.

Dataset Size Scalability. An important aspect in evaluating data placement schemes is their ability to maintain high performance across varying data scales [31, 52, 89]. Figure 16(b) shows the performance gains of SINLK compared to the baseline when loading data varying from 10 to 60 million records. In the first three workloads, SINLK's improvement grows with the dataset size as the longer data access paths in larger datasets emphasize the benefits of promoting hot paths. However, for *With Insert*, new nodes creates new hot paths, requiring time for SINLK to identify and promote them, slightly reducing gains at larger scales. Nonetheless, SINLK outperforms the baseline by over 50% even at larger scales.

6.5 Performance Breakdown

This section evaluates the impact of each technique and key/value size in SINLK. We first present the factor analysis (§6.5.1), then evaluate the holistic management (§6.5.2).

6.5.1 Factor Analysis for SINLK

Contributions of techniques. We begin with the naive Masstree and gradually apply each proposed technique to conduct S-Masstree. Figure 17 shows how each technique contributes to SINLK's performance.

+ Layering. This step implements layer-aware allocation. With this design, SINLK exhibits performance gains ranging from 31.9% to 49.8% compared to the baseline.

+ Access Tracking. In this step, we evaluate the overhead of access tracking. Tracking only leaf nodes incurs a minimal performance loss of 5.1% to 7.0% compared to the last step, while tracking all nodes incurs about 60% performance degradation. This indicates that the overhead of updating only the leaf's information for access tracking is acceptable.

+ Migration. In this step, we begin to migrate hot paths and cold nodes, improving SINLK's performance by 2.8% to 28.7% over only using layer-aware allocation. However, the fixed hot/cold thresholds are not suitable for the *With Insert* workload, resulting in less noticeable performance gains.

+ Histogram. By introducing a histogram to dynamically adjust the hot/cold thresholds, SINLK achieves performance gains ranging from 1.7% to 16.8% over the static thresholds.

Impact of key and value size. We evaluate S-Masstree with various key and value sizes. Figure 18 shows that SINLK achieves stable performance improvement (up to 75%) across different key sizes. SINLK also stably outperforms the baseline (30%–40%) with value sizes from 8 to 256 bytes (we do not present the details here due to the limited space).

6.5.2 Effectiveness of Holistic Management.

Figure 19 shows that the passive (demotion only when fast memory is tight) and separated (the watermark only cares about migration) management significantly worsens the tail latency due to burst migrations when fast memory is tight.

To assess the watermark mechanism's effectiveness, we monitor the S-Masstree's fast memory usage ratio over time. Figure 20 shows that as hot paths are promoted, fast memory usage gradually increases. When exceeding the high watermark, it declines due to increased demotion. The watermark mechanism adjusts allocation and migration to maintain fast

Table 2. 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

memory usage within the watermark bounds. This ensures that once hot paths are promoted, fast memory usage ratio stabilizes near the high watermark across all workloads.

6.6 Overhead Analysis

CPU Time Overhead. We assess the overhead of each background worker in SINLK by measuring the proportion of its execution time to the total CPU time. Table 2 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 leaf nodes to identify hot paths and cold nodes. Next is the watermark maintainer, which is frequently activated to monitor fast memory usage.
Memory Overhead. SINLK only increase memory usage of Masstree's internal/leaf nodes by only 0.36%/0.93%. For ART's internal/leaf nodes, the increase is 0.04%–1.3%/2.0%.

6.7 Real-world Workload Evaluation

Alibaba Block Traces, collected from Alibaba Cloud elastic block service [90], reflect representative user behaviors in the cloud [87]. We choose two traces with the largest working set (i.e., those with the greatest need for memory expansion) for evaluation (Trace 7 as Trace A, and Trace 40 as Trace B). Figure 21 shows that S-ART outperforms the baseline by up to 90% in throughput while reducing P99 latency by up to 79% compared to MEMTIS and 63% compared to PAC-L. Figure 22 reveals that S-Masstree increases throughput by up to 39% compared to the baseline and reduces P99 latency by up to 54% compared to MEMTIS. This result demonstrates that SINLK is also effective in real-world scenarios.

7 Discussion and Future Work

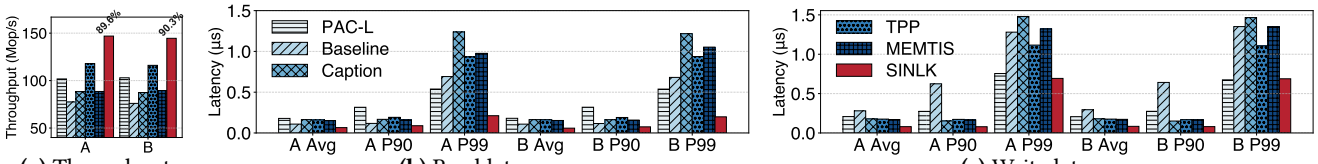
False Negative Internal Nodes. SINLK selects promotion candidates starting from hot leaf nodes. In this case, a large group of infrequently accessed leaf nodes may not trigger promotion individually, but their aggregated accesses could trigger promotion of the ancestors (i.e., false negative internal nodes). However, we find this situation is rare in practice. We break down the accesses to a B+ tree with 20 million keys following the standard Zipfian distribution and find that false negative internal nodes only appear in the last level internal nodes with a probability of less than 0.9%.
Hardware-based Access Tracking. SINLK updates the access frequency on each leaf access and traverses all leaves to select migration candidates. The former inevitably affects tree operation performance, and the latter is the main overhead in SINLK's background processing (§6.6). In the future, we will explore hardware-based sampling, like Intel Precise Event Based Sampling (PEBS) [91] and AMD Instruction-Based Sampling (IBS) [92], to further reduce this overhead.
Multi-layer Memory Hierarchy. Currently, SINLK targets two-layer memory hierarchies with fast and slow memory. To extend SINLK to multi-layer memory hierarchies, we can attempt to rank various memory types by their latency and throughput, and then apply SINLK's mechanisms and strategies between each pair of adjacent layers.

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 [51–54, 81–83, 88, 93–97]. These schemes typically involve three major tasks: access tracking [89, 98, 99], hotness detection [52, 93, 100], data migration [94, 101, 102]. SINLK 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 SINLK does not consider memory contention, Colloid [54] may further enhance SINLK when memory contention occurs.
Object-granularity Data Placement. The first type of user-space object-granularity data placement includes libraries [103–107] like CacheLib [103, 104] and X-Mem [105], which require specialized APIs or offline profiling for statistics. HeMem [31] avoids offline profiling and new APIs but does not manage the small allocation and migrates data at page granularity. The second type is hardware-based cache-line granularity placement [108–110], such as Intel Flat Memory Mode and Johnny Cache. Additionally, many databases are exploring data placement for CXL-HM [111–115]. Unlike these works, SINLK leverages the tree's characteristics to achieve fine-grained management and get more semantics than hardware-based solutions without offline profiling.
Far Memory Management. Far memory systems [116–120] utilize remote memory connected via network as the memory expansion. Unlike CXL-HM, far memory requires different interfaces and protocols to access remote memory, such as RDMA and RPCs. They usually focus on reducing network overhead [121, 122], read/write amplification [123, 124], and synchronization costs [125]. Far memory systems and SINLK are both related and orthogonal, as far memory indexes can benefit from SINLK's insights and SINLK can leverage far memory as a new slow memory layer to extend memory.

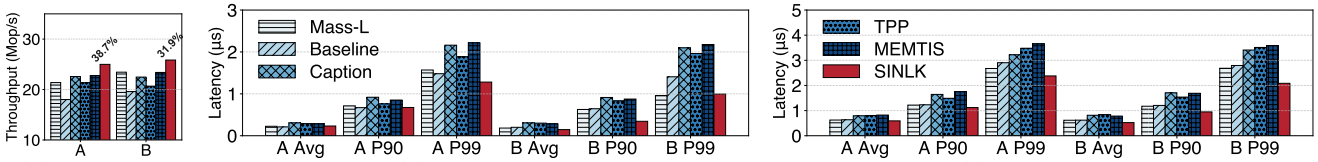
9 Conclusion

This paper proposes SINLK, a fine-grained, structure-aware data placement scheme for tree-structure indexes that utilizes the tree's inherent characteristics to achieve high stable performance.



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

Figure 21. Performance comparison between S-ART and compared systems with Alibaba Block Traces.



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

Figure 22. Performance comparison between S-Masstree and compared systems with Alibaba Block Traces.

References

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

[2] Xingbo Wu, Fan Ni, and Song Jiang. Wormhole: A fast ordered index for in-memory data management. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

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

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

[5] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D Nguyen, Tim Kaldewey, Victor W Lee, Scott A Brandt, and Pradeep Dubey. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 339–350, 2010.

[6] Jun Rao and Kenneth A Ross. Making B+-trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 475–486, 2000.

[7] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 302–313. IEEE, 2013.

[8] Tomer Shanny and Adam Morrison. Occualizer: Optimistic concurrent search trees from sequential code. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 321–337, 2022.

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

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

[11] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, DaMoN '16, New York, NY, USA, 2016. Association for Computing Machinery.

[12] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 18–32, New York, NY, USA, 2013. Association for Computing Machinery.

[13] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1567–1581, New York, NY, USA, 2016. Association for Computing Machinery.

[14] HyPer – a hybrid OLTP&OLAP high performance DBMS. https://hyper-db.de/, 2024.

[15] MemSQL. https://www.memsql.com/, 2024.

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

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

[18] VoltDB. https://www.voltdb.com/, 2024.

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

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

[21] 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. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1383–1394, New York, NY, USA, 2015. Association for Computing Machinery.

[22] Michael Körber, Nikolaus Glombiewski, and Bernhard Seeger. Index-Accelerated Pattern Matching in Event Stores. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1023–1036, New York, NY, USA, 2021. Association for Computing Machinery.

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

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

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

[26] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim,

and Parthasarathy Ranganathan. Meet the walkers: accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, page 468–479, New York, NY, USA, 2013. Association for Computing Machinery.

[27] Christoph Anneser, Andreas Kipf, Huanchen Zhang, Thomas Neumann, and Alfons Kemper. Adaptive Hybrid Indexes. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, pages 1626–1639, New York, NY, USA, 2022. Association for Computing Machinery.

[28] Huanchen Zhang, Xiaoxuan Liu, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. Order-Preserving Key Compression for In-Memory Search Trees. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 1601–1615, New York, NY, USA, 2020. Association for Computing Machinery.

[29] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 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*, pages 53–64, 2012.

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

[31] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 392–407, New York, NY, USA, 2021. Association for Computing Machinery.

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

[33] Zviad Metreveli, Nickolai Zeldovich, and M. Frans Kaashoek. CPHASH: a cache-partitioned hash table. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, page 319–320, New York, NY, USA, 2012. Association for Computing Machinery.

[34] Irina Calciu, Justin Gottschlich, and Maurice Herlihy. Using Elimination and Delegation to Implement a Scalable NUMA-Friendly Stack. In *5th USENIX Workshop on Hot Topics in Parallelism (HotPar 13)*, San Jose, CA, June 2013. USENIX Association.

[35] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 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*, ASPLOS '17, page 207–221, New York, NY, USA, 2017. Association for Computing Machinery.

[36] Ziegler, Tobias and Tumkur Vani, Sumukha and Binnig, Carsten and Fonseca, Rodrigo and Kraska, Tim. Designing distributed tree-based index structures for fast rdma-capable networks. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 741–758, New York, NY, USA, 2019. Association for Computing Machinery.

[37] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1033–1048, New York, NY, USA, 2022. Association for Computing Machinery.

[38] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R. Lyu, and Yangfan Zhou. SMART: A high-performance adaptive radix tree for disaggregated memory. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 553–571, Boston, MA, July 2023. USENIX Association.

[39] Xuchuan Luo, Jiacheng Shen, Pengfei Zuo, Xin Wang, Michael R. Lyu, and Yangfan Zhou. CHIME: A cache-efficient and high-performance hybrid index on disaggregated memory. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 110–126, New York, NY, USA, 2024. Association for Computing Machinery.

[40] Baotong Lu, Kaisong Huang, Chieh-Jan Mike Liang, Tianzheng Wang, and Eric Lo. DEX: Scalable Range Indexing on Disaggregated Memory. *Proc. VLDB Endow.*, 17(10):2603–2616, August 2024.

[41] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 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*, FAST'23, USA, 2023. USENIX Association.

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

[43] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 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*, SIGMOD '16, pages 371–386, New York, NY, USA, 2016. Association for Computing Machinery.

[44] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 257–270, Santa Clara, CA, February 2017. USENIX Association.

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

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

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

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

[49] Compute Express Link. Compute express link (CXL). https://computeexpresslink.org/, February 2024.

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

[51] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 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*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

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

[53] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. Nomad: Non-Exclusive Memory Tiering via Transactional Page Migration. In *18th USENIX Symposium on Operating*

Systems Design and Implementation (OSDI 24), pages 19–35, Santa Clara, CA, July 2024. USENIX Association.

[54] Midhul Vuppalapati and Rachit Agarwal. Tiered Memory Management: Access Latency is the Key! In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 79–94, New York, NY, USA, 2024. Association for Computing Machinery.

[55] Zhe Zhou, Yiqi Chen, Tao Zhang, Yang Wang, Ran Shu, Shuotao Xu, Peng Cheng, Lei Qu, Yongqiang Xiong, Jie Zhang, and Guangyu Sun. NeoMem: Hardware/Software Co-Design for CXL-Native Memory Tiering, 2024.

[56] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154. Association for Computing Machinery, 2010.

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

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

[59] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. An introduction to the compute express link (CXL) interconnect. *ACM Comput. Surv.*, 56(11), July 2024.

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

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

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

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

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

[65] Johannes Weiner. [PATCH] mm: mempolicy: N:M interleave policy for tiered memory nodes. https://lore.kernel.org/linux-mm/YqD0%2FtzFwXvJ1gK6@cmpxchg.org/T/, 2022.

[66] Hongliang Qu and Zhibin Yu. 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*, ASPLOS '24, page 1233–1249, New York, NY, USA, 2024. Association for Computing Machinery.

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

[68] Foteini Strati, Christina Giannoula, Dimitrios Siakavaras, Georgios Goumas, and Nectarios Koziris. An adaptive concurrent priority queue for NUMA architectures. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, CF '19, page 135–144, New York, NY, USA, 2019. Association for Computing Machinery.

[69] Kai Lu, Siqi Zhao, Haikang Shan, Qiang Wei, Guokuan Li, Jiguang Wan, Ting Yao, Huatao Wu, and Daohui Wang. Scythe: A Low-latency RDMA-enabled Distributed Transaction System for Disaggregated Memory. *ACM Trans. Archit. Code Optim.*, 21(3), September 2024.

[70] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. ROART: Range-query Optimized Persistent ART. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 1–16. USENIX Association, February 2021.

[71] 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. PLIN: a persistent learned index for non-volatile memory with high performance and instant recovery. *Proc. VLDB Endow.*, 16(2):243–255, October 2022.

[72] Shimin Chen and Qin Jin. Persistent B+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.

[73] Yongju Song, Wook-Hee Kim, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 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*, ASPLOS 2023, page 588–602, New York, NY, USA, 2023. Association for Computing Machinery.

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

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

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

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

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

[79] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. pages 191–208, November 2020.

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

[81] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, et al. 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*, pages 727–741, 2023.

[82] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, et al. 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*, pages 317–330, 2019.

[83] Dong Xu, Junhee Ryu, Kwangsik Shin, Pengfei Su, and Dong Li. FlexMem: Adaptive Page Profiling and Migration for Tiered Memory. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 817–833, Santa Clara, CA, July 2024. USENIX Association.

[84] Richard A. Hankins and Jignesh M. Patel. 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*, SIGMETRICS '03, page 283–294, New York, NY, USA, 2003. Association for Computing Machinery.

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

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

[87] Qiuping Wang, Jinhong Li, Patrick P. C. Lee, Tao Ouyang, Chao Shi, and Lilong Huang. Separating Data via Block Invalidation Time In-

ference for Write Amplification Reduction in Log-Structured Storage. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 429–444, Santa Clara, CA, February 2022. USENIX Association.

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

[89] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 331–345, 2019.

[90] 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. 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*, FAST '24, USA, 2024. USENIX Association.

[91] Performance monitoring unit sharing guide. https://www.intel.com/content/www/us/en/content-details/727001/performance-monitoring-unit-sharing-guide.html, 2022.

[92] Advanced Micro Devices. Amd uprof v4.0 user guide. https://www.amd.com/content/dam/amd/en/documents/developer/uprof-v4.0-gaGA-user-guide.pdf, 2022. Accessed: 2024-03-26.

[93] Neha Agarwal and Thomas F. Wenisch. 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*, ASPLOS '17, pages 631–644, New York, NY, USA, 2017. Association for Computing Machinery.

[94] 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, 2022.

[95] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 715–728. USENIX Association, July 2021.

[96] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 521–534, New York, NY, USA, 2017. Association for Computing Machinery.

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

[98] Vamsee Reddy Kommareddy, Simon David Hammond, Clayton Hughes, Ahmad Samih, and Amro Awad. Page migration support for disaggregated non-volatile memories. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, pages 417–427, New York, NY, USA, 2019. Association for Computing Machinery.

[99] Jee Ho Ryoo, Lizy K. John, and Arkaprava Basu. A case for granularity aware page migration. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, pages 352–362, New York, NY, USA, 2018. Association for Computing Machinery.

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

[101] Rik van Riel and Vinod Chegu. Automatic numa balancing. In *Red Hat Summit*, 2014.

[102] Adnan Maruf, Ashikee Ghosh, Janki Bhimani, Daniel Campello, Andy Rudoff, and Raju Rangaswami. Multi-clock: Dynamic tiering for hybrid memory systems. In *HPCA*, pages 925–937, 2022.

[103] Don Moon, Daniel Byrne, and Sounak Gupta. 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, 2023. Presented by Don Moon (SK hynix), Daniel Byrne (Intel), and Sounak Gupta (Intel).

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

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

[106] Kai Wu, Yingchao Huang, and Dong Li. 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*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.

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

[108] Yan Sun, Jongyul Kim, Dougas 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. M5: Mastering Page Migration and Memory Management for CXL-based Tiered Memory Systems. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2025.

[109] 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. Managing Memory Tiers with CXL in Virtualized Environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 37–56, Santa Clara, CA, July 2024. USENIX Association.

[110] Baptiste Lepers and Willy Zwaenepoel. 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)*, pages 519–534, Boston, MA, July 2023. USENIX Association.

[111] Alberto Lerner and Gustavo Alonso. CXL and the Return of Scale-Up Database Engines. *Proc. VLDB Endow.*, 17(10):2568–2575, August 2024.

[112] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. Enabling CXL Memory Expansion for In-Memory Database Management Systems. In *Proceedings of the 18th International Workshop on Data Management on New Hardware*, DaMoN '22, New York, NY, USA, 2022. Association for Computing Machinery.

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

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

[115] Yibo Huang, Newton Ni, Vijay Chidambaram, Emmett Witchel, and Dixin Tang. Pasha: An Efficient, Scalable Database Architecture for CXL Pods.

[116] Bin Yan, Youyou Lu, Qing Wang, Minhui Xie, and Jiwu Shu. Patronus: high-performance and protective remote memory. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies*, FAST'23, USA, 2023. USENIX Association.

[117] 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. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 775–787, Boston, MA, July 2018. USENIX Association.

[118] Ankit Bhardwaj, Chinmay Kulkarni, and Ryan Stutsman. Adaptive Placement for In-memory Storage Functions. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 127–141. USENIX Association, July 2020.

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

[120] Yijie Zhong, Minqiang Zhou, Zhirong Shen, and Jiwu Shu. UniMem: Redesigning Disaggregated Memory within A Unified Local-Remote Memory Hierarchy. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 463–477, Santa Clara, CA, July 2024. USENIX Association.

[121] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Canvas: Isolated and Adaptive Swapping for Multi-Applications on Remote Memory. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 161–179, Boston, MA, April 2023. USENIX Association.

[122] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. 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)*, pages 181–198, Boston, MA, April 2023. USENIX Association.

[123] Zhiyuan Guo, Zijian He, and Yiying Zhang. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 692–708, 2023.

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

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