Disk-Based LSMs: An Unexpectedly Good Index for Partly Coherent CXL Memory

Kiran Hombal
University of Illinois Urbana-Champaign

Jiyu Hu
University of Illinois Urbana-Champaign

Marcos K. Aguilera
NVIDIA

Ramnatthan Alagappan
University of Illinois Urbana-Champaign

Aishwarya Ganesan
University of Illinois Urbana-Champaign

Abstract. We consider the design of an in-memory range index for systems with CXL shared memory in which only a small part of the memory is cache coherent. While existing index data structures can be ported to such a setting using a software-coherence approach, the result performs poorly. We find that the key reason is that such data structures have a large *updatable surface area*—the part of the data structure that can be modified in-place. We propose a new design that, perhaps surprisingly, is based on log-structured merge trees (LSMs)—data structures originally designed for disks rather than memory. We further enhance LSMs by allowing in-place updates to the upper level of the LSM. The resulting data structure outperforms state-of-the-art schemes based on Adaptive Radix Trees (ARTs) and B-trees ported to our setting, achieving up to 9.4× higher peak throughput over in-memory indexes and up to 15.1× over existing CXL systems.

CCS Concepts: • **Computer systems organization → Processors and memory architectures**.

Keywords: CXL, Indexing, Software coherence

ACM Reference Format:
Kiran Hombal, Jiyu Hu, Marcos K. Aguilera, Ramnatthan Alagappan, and Aishwarya Ganesan. 2026. Disk-Based LSMs: An Unexpectedly Good Index for Partly Coherent CXL Memory. In *Symposium on Operating Systems Principles (SOSP '26), September 29–October 2, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3830418.3843886

1 Introduction

Compute Express Link (CXL) is a new technology that enables multiple physical hosts to share a memory device and access it using CPU load and store instructions [25, 26]. The shared memory can be used to build more efficient applications, such as database systems, key-value stores, and file systems, where hosts share data via CXL memory [51, 52].

An important foundation for all of these applications is an index structure that supports efficient point queries, range queries, and updates. However, building an index for CXL shared memory faces a challenge: while the CXL memory will span several TBs, only a small portion of the memory (e.g., a few hundred MB) is expected to be kept coherent across hosts by the hardware [12, 51–53], with the rest non-coherent. This model is called the *partly coherent CXL* memory model, where there is a small hardware coherent region (SCR) and a large non-coherent region (LNR) [46]. In this model, if hosts access data in the LNR, the host-side processor caches may have stale data, causing correctness problems.

Recent systems [46, 51] address the issue of a small SCR by providing coherence in software (for the data in the LNR). Specifically, they provide coherence for objects in the LNR by tracking some per-object coherence metadata in the SCR. The metadata helps hosts detect stale processor caches so that they can flush them before accessing data in the LNR. For example, Tigon [51] uses this approach to provide coherence for database records that reside in the LNR.

One could apply this software-coherence approach to index structures (§3.1). We port two state-of-the-art in-memory indexes to this model, the adaptive radix tree [64] (ART) and the B+tree [22], both of which use optimistic lock coupling (OLC) [65, 102]. We call these two ported indexes SC-ART and SC-BTree, where SC stands for software-coherent. OLC already maintains a version number per tree node to detect concurrent modifications, and we piggyback on these version numbers, using them as the coherence metadata for the index. Each index keeps its version numbers in the SCR and consults them to detect when a host's cached copy of the corresponding node in the LNR may be stale. Unfortunately, we find that this approach leads to poor performance. This is because any node in the index data structure can be updated, and thus a version number is required for all nodes. However, with large datasets, the version numbers for all index structure nodes do not fit within SCR, and thus version numbers for some nodes must themselves be kept in LNR, requiring cache flushes to access them, leading to poor performance. We can alleviate this problem by increasing the size of each node in the data structure, which reduces the total number of nodes so that the version numbers for all the nodes fit in SCR. However, increasing the node size causes disruptive false invalidations, where an update to a single key in a node results in cache invalidations for all the keys in the node, including keys that were not modified.

SOSP '26, Prague, Czech Republic

ACM ISBN 979-8-4007-2585-2/26/09
https://doi.org/10.1145/3830418.3843886

The core issue with these in-memory indexes is that they have a large *updatable surface area*, the part of the data structure that can be modified in-place (§3.4). As a result, we are faced with a dilemma. On the one hand, with small nodes, the required version numbers do not fit in the SCR (§3.2). On the other hand, with larger nodes, the index suffers from too many false invalidations on updates (§3.3). Both cases lead to excessive cache flushes and thus poor performance.

This paper's central thesis is that, perhaps surprisingly, *log-structured merge trees [17, 33, 37, 79] (LSMs) that were built for disk-based storage fit the partly coherent model much better than existing in-memory indexes* (§4.1). The crux of why LSMs suit the partly coherent model is that they force all in-place modifications to a small portion of the data structure, which fits in the SCR, while large portions of the data structure remain immutable, eschewing the coherence concerns of in-place updates. Specifically, in an LSM, a small buffer called the *memtable* captures all in-place updates. When full, the memtable is written out as an immutable sorted file called an *SSTable*; such immutable SSTables form a large portion of the data structure. Since the memtable is small, it can be stored in SCR and there is no need to track coherence for it. SSTables are never updated in place during their lifecycle; they are only deleted during a background compaction process. Thus, they can be maintained in the LNR without any coherence concerns during their lifetime. To ensure correctness when SSTables are recycled, we keep in SCR the identifiers of the active SSTables and their LNR locations. These SSTable identifiers are strictly increasing. If a host sees an identifier it has not seen before, it flushes the SSTable's memory region before accessing it. Because SSTables are large, there are few of them, and so the active SSTable list in SCR is small.

Based on this thesis, our first key idea is to use LSMs as the base index for partly coherent CXL memory. We port LSMs to the partly coherent memory model using the software-coherence approach, resulting in what we call SC-LSM. Compared to in-memory data structures ported using software coherence, SC-LSM performs better because it avoids the cache flushes incurred by the alternatives.

While SC-LSM is a significant improvement over other indexes, it suffers from a canonical problem that LSM implementations typically suffer from: with more writes, compactions cannot catch up. In such cases, either writes must be stalled, which reduces write performance, or writes can go through, which reduces read performance as reads have to access a large number of SSTables. This is a well-known issue with LSMs, not unique to SC-LSM.

The in-memory setting, however, offers a unique opportunity to solve this problem. The key insight is that compactions can be significantly reduced if the LSM's upper layers can be updated in place; this reduces the number of SSTables created. This idea does not work in disk-based LSMs as in-place updates require random writes, which squander the performance benefits of LSMs. However, in the in-memory setting, such in-place updates are feasible because random writes are not prohibitively expensive. Based on this insight, our next idea is to make the upper level of the LSM in-place updatable (§4.2). Doing so increases the updatable surface area, but we do so carefully: we limit the updatable surface area such that the version numbers for that area, even if tracked in a fine-grained manner, can still fit in the SCR.

Prism. Based on these ideas, we build Prism, an LSM-based index for partly coherent CXL. Prism uses a three-tier design, where each tier uses a different coherence mechanism (§5). A memtable in the SCR benefits from hardware coherence, while a bounded updatable layer (BUL) in the LNR absorbs repeated writes in place using fine-grained version numbers in the SCR, reducing compaction pressure. The remaining data resides in immutable SSTables in the LNR, where coherence is tracked at SSTable granularity. Prism further inverts the memtable's role from a write buffer into a cache for hot keys, ensuring only frequently written keys consume scarce SCR space while one-hit wonders flow directly to BUL.

Results. Our experiments (§7) show that Prism's small updatable surface area translates to significant performance gains over in-memory indexes ported to the partly coherent model. For a key-value store running YCSB, Prism achieves up to 9.4× higher throughput than SC-BTree on write-heavy workloads, where SC-BTree's performance collapses due to false invalidations. Even on read-heavy workloads, where the multi-level LSM read path is traditionally a disadvantage, Prism improves over SC-ART by up to 5.8× and over SC-BTree by up to 1.4×. On range scans, although Prism performs slightly worse than SC-BTree at very low write ratios, it outperforms SC-BTree with moderate and high write ratios. We also compare against a state-of-the-art RDMA-based disaggregated memory index and existing CXL shared-memory systems, and find that Prism outperforms them significantly. Finally, with only 128 MB of SCR, Prism matches the throughput of SC-ART given an unlimited SCR on write-heavy workloads, confirming that an LSM-based design can close the gap with in-memory indexes despite the SCR constraint.

Contributions. This paper makes four contributions.

- We identify *updatable surface area* as a fundamental challenge for indexes in partly coherent CXL memory (§3).
- We show that LSMs, designed for disk, surprisingly fit the partly coherent CXL model well (§4).
- We design Prism, a novel LSM-based index that reduces compactions via a bounded updatable layer and improves performance via an SCR-aware memtable (§5).
- We show that Prism significantly outperforms in-memory indexes and state-of-the-art CXL and RDMA systems (§7).

2 Background

We describe the CXL shared memory model (§ 2.1) and provide an overview of in-memory range indexes (§ 2.2). We

then provide a brief background on log-structured merge trees (§ 2.3), which our design builds on.

2.1 CXL Shared Memory

CXL is a cache-coherent interconnect layered on top of PCIe that enables processors to access memory attached to external devices using CPU load and store instructions [25, 26]. Early specifications (CXL 1.1) target single-host memory expansion, but the CXL 3.0 and CXL 3.2 standards extend this capability to *multi-host* settings where several machines can map the same CXL-attached memory into their respective address spaces and access it concurrently [52, 67, 111]. As with conventional DRAM, accesses to CXL memory happen via the usual hierarchy of CPU caches at each host.

Partly Coherent CXL Memory. Although the CXL 3.0 specification defines a hardware coherence protocol, vendors including AMD, Micron, and Samsung report that the underlying coherence mechanisms such as snoop filters and back-invalidation become prohibitively expensive when shared memory exceeds a few hundred megabytes [12, 45, 51–53]. Thus, although CXL devices will have terabytes of memory, hardware will keep only a small fraction coherent, with the rest non-coherent without support for speculative prefetching [12, 45, 51]. This divides CXL memory into two regions: a small hardware coherent region (SCR) of a few hundred MB, for which hardware keeps caches synchronized across hosts; and a much larger non-coherent region (LNR), spanning terabytes, for which no such guarantee holds [46].

Software Coherence. Without hardware coherence for LNR, hosts may hold stale copies of shared data in their caches, causing correctness problems [51, 103]. Recent CXL systems like Tigon [51], Megalon [46], and CxlAlloc [78] address this by providing coherence in software. They store per-object coherence metadata (like version numbers or validity bitmaps) in SCR, where hardware ensures coherence. Before accessing shared data in LNR, a host looks up the corresponding metadata in SCR; if the metadata indicates that CPU caches may be stale, the host flushes the relevant cache lines and re-reads from CXL memory. For software coherence to work, we assume that a CPU does not spontaneously cache data in the LNR that the program does not access.

2.2 In-Memory Range Indexes

Our goal is to build range indexes for partly coherent CXL memory. We now provide an overview of state-of-the-art in-memory indexes. While many in-memory indexes exist, the most performant and popular indexes fall into two categories: radix-tree-based indexes, such as the Adaptive Radix Tree (ART) [7, 14, 44, 61, 64, 65, 74]; and B-tree-based indexes, such as B+trees [22, 59, 63, 82, 89, 102].

Adaptive Radix Tree (ART). ART is a radix-tree-based index that achieves high point-query throughput using adaptive node sizes to reduce memory consumption while maintaining $O(k)$ lookups, where k is the key length [64]. ART defines four node types (Node4, Node16, Node48, and Node256) that hold up to 4, 16, 48, and 256 children, respectively. Nodes start as the smallest type (Node4, ~40 bytes) and grow adaptively as keys are inserted. The tree node at which a key terminates stores the pointer to the data record. ART with optimistic lock coupling (ART-OLC) [65] is the state-of-the-art concurrent in-memory index. ART-OLC uses per-node version numbers to enable lock-free reads; writers increment the version number after modifying a node, while readers check the version number to detect concurrent modifications and retry if necessary. An update to the data record increments the version number of the node that points to it.

B-trees. B-trees and their variants (*e.g.* B+trees [22, 63]) are widely used in database systems and file systems. They organize keys in large, fixed-size nodes, with each leaf node containing multiple data records (either inline or as pointers). The number of keys per leaf node, called the *leaf capacity*, is determined by the node size and the data-record size. Insertions add keys to leaf nodes, and node splits propagate changes up through internal nodes. Concurrency control in B-trees typically uses either lock coupling [102] or optimistic techniques with per-node version numbers like ART-OLC.

2.3 Log-Structured Merge Trees

Log-structured merge trees (LSMs) [79] are data structures used to organize on-disk data. Unlike B-trees, which require random writes upon updates, LSMs convert random updates into sequential I/O, improving write throughput. Thus, LSMs are widely adopted by Bigtable [17], LevelDB [37], Cassandra [60], RocksDB [33], and others [27, 71, 81, 88].

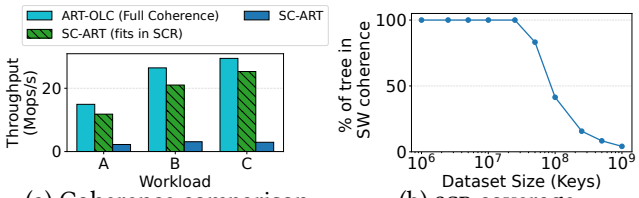
An LSM organizes data across an in-memory write buffer, called the *memtable*, and multiple levels of immutable sorted files called *SSTables*. The memtable absorbs incoming writes, and, once it reaches capacity, it is serialized as a new SSTable at level 0 (L0). Each level L_i is roughly 10× larger than the previous level L_{i-1}. When a level fills, a background *compaction* process merge-sorts SSTables from that level into the next. From L1 onward, SSTables within a level have non-overlapping key ranges, whereas L0 may contain overlapping files because each one comes from an independently flushed memtable. Point and range queries probe the memtable first and then scan levels from newest to oldest.

3 Motivation

To share data on CXL memory, applications need a range index that operates correctly and efficiently under the partly coherent model. No existing index is designed for this model. We thus port two state-of-the-art in-memory indexes, ART-OLC and B-tree-OLC, to the partly coherent model and study the challenges they face.

3.1 Indexes on Partly Coherent CXL

To port these indexes, we use the software coherence approach described in § 2.1: we assign version numbers to track coherence and store them in the SCR. Conveniently, these indexes use a mechanism for concurrency control (optimistic



(a) Coherence comparison (b) SCR coverage

Figure 1. SC-ART SCR Overflow.

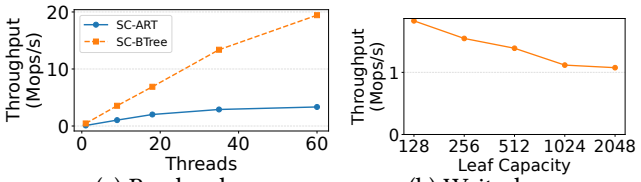
lock coupling or OLC) that already maintains per-node version numbers for readers to detect concurrent modifications. We piggyback on this existing mechanism to provide software coherence. Specifically, we separate the version numbers from the tree nodes and store them in SCR. The actual tree nodes and the key-value pairs reside in LNR to save SCR space. Before reading a node from LNR, a host checks the node's version number in SCR; if it has changed since the host last read the node, the host flushes the corresponding cache lines and re-reads from CXL memory. On a write, hosts update the corresponding version numbers and flush the cache lines to push data to CXL. As any node in the tree can be modified, we maintain a version number for each tree node in SCR. We call these ported indexes *SC-ART* and *SC-BTree*, respectively.

3.2 Version Numbers Overflow the SCR

Software coherence enables one to build indexes larger than SCR, and the approach works well as long as all the version numbers fit in SCR. However, as the dataset grows, so does the number of tree nodes and version numbers. Eventually, the version numbers themselves do not fit in the SCR and must spill into LNR. For a version number that is in LNR, a host has no way of knowing whether its cached copy is fresh or stale, so it must perform a cache flush on *every* version number access, regardless of whether the underlying data has changed. This leads to excessive cache flushes for larger datasets and thus poor performance.

To understand how severe this problem is, we evaluate SC-ART under two SCR configurations. In the first, we cap the SCR at a realistic 128 MB, matching the expected SCR capacity [12, 52, 53]. In the second, we unrealistically assume an unlimited SCR so that all version numbers fit. We compare the throughput of the two SC-ART configurations with the traditional ART-OLC that assumes fully coherent memory, using YCSB workloads (A: 50% writes and 50% reads, B: 5% writes and 95% reads, and C: 100% reads). We use 100M keys and describe the detailed setup in §7.

Figure 1(a) shows the result. In the unrealistic configuration with unlimited SCR, where all version numbers fit in SCR, SC-ART performs well, adding only a small overhead compared to the fully coherent ART-OLC. The overhead is due to the inherent cost of software coherence. Specifically, each node access requires another memory access to check the version number in SCR. Furthermore, hosts also flush the caches upon writes or when version changes are detected



(a) Read-only (b) Write-heavy

Figure 2. SC-BTree False Invalidation.

upon reads. However, with a realistic SCR of 128 MB, SC-ART performs poorly. This is because, with 100M keys, less than half of the version numbers fit in SCR; others spill to LNR. Thus, when accessing such version numbers in LNR, hosts must perform a cache flush on every version-number access, even if nothing has changed. This leads to excessive cache flushes, which significantly degrades performance.

Figure 1(b) shows how a realistic SCR cannot fit version numbers for large datasets. In the YCSB dataset, which exhibits sparse branching at most radix positions, the majority of nodes in the tree are the smallest Node4 type, which holds only 4 children. This means the tree contains a large number of nodes, each requiring one version entry in SCR. With 10M keys, the tree has ~4M nodes, which fit comfortably in 128 MB. However, with 100M keys, the tree has ~40M inner nodes and SCR coverage falls below 50%. For dataset sizes larger than 1B keys, only a small fraction of the version numbers fit in SCR. This in turn causes significant performance degradation for SC-ART. With small node sizes, SC-BTree faces the same degradation from version numbers overflowing the SCR.

3.3 Effects of Increasing Fan-Out

To solve the problem of version numbers not fitting in the SCR, we can increase the fan-out (i.e., size) of each node. Doing so reduces the number of nodes and thus version numbers. Since the node sizes are adaptively determined by the key distribution in ART, it is not possible to statically configure ART to use larger nodes. However, we can use SC-BTree to evaluate this idea. We fix the fan-out of SC-BTree to a large value (128), so that many keys are grouped into a single large node, and a single version number tracks coherence for all keys in that node. With such a large fan-out, version numbers easily fit in SCR.

Figure 2(a) compares the performance of SC-BTree (with a fan-out of 128) to SC-ART for a read-only workload with 128 MB SCR capacity. As shown, SC-BTree outperforms SC-ART despite ART typically being the faster index on single-machine fully coherent platforms [65]. This is because SC-BTree maintains far fewer version numbers, allowing software coherence to cover a much larger portion of the tree within a 128 MB SCR. SC-ART, in contrast, performs poorly due to flushes when accessing version numbers not in SCR.

However, larger fan-out introduces a new problem: *false invalidations*. When any single key within a tree node is updated, the version number for the entire node changes. This forces every host that has cached any key from that

node to flush and re-read the entire node, even if the key it wants to access has not been modified. Thus, in the presence of writes, SC-BTree suffers from excessive cache flushes due to false invalidations, leading to overall poor performance.

Figure 2(b) shows this problem under the write-heavy YCSB-A workload. As shown, with a leaf node capacity (fan-out) of 128 (the left-most point), SC-BTree achieves only 1.8M ops/s; in contrast, the configuration of SC-ART where the version numbers fit in SCR achieves much higher performance of 12M ops/s (see Figure 1(a)). Furthermore, the performance of SC-BTree worsens with increasing leaf node fan-out. This is because as each node covers more keys, a single-key update falsely invalidates an increasingly large number of unmodified keys, degrading throughput. Thus, although larger fan-out helps fit version numbers in SCR, it also leads to excessive cache flushes due to false invalidation.

3.4 The Core Problem

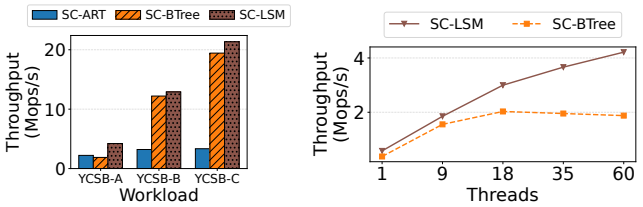
These in-memory indexes perform poorly in the partly coherent model because they have a large *updatable surface area*. We define the updatable surface area as the portion of the data structure that can receive *in-place* modifications. In both ART and B-trees, any node from root to leaf can be modified in place, so the updatable surface area spans the entire index. Thus, every node requires a version number. If the index uses small nodes, it can track coherence precisely and avoid false invalidations, but it needs many version numbers, which cannot fit in SCR. In contrast, version numbers for larger nodes can fit in SCR, but the index suffers from false invalidations as coherence is tracked coarsely. As long as the updatable surface area is large, one faces a fundamental trade-off between version numbers not fitting in SCR and false invalidations, both of which lead to excessive cache flushes and poor performance.

4 Rationale and Key Ideas

As we discussed, state-of-the-art memory indexes, when ported to use software coherence, suffer from poor performance. This paper thus asks, *what is the right way to build an index for partly coherent memory?* Perhaps surprisingly, we find the answer lies not in in-memory indexes, but rather in a class of data structures originally intended for a different medium, namely *disk-based LSMs*. This section first argues why LSMs are a good fit for partly coherent CXL because of their small updatable surface area (§4.1). We then propose a modification to LSMs to better fit it for the in-memory model while keeping their updatable surface area to be small (§4.2).

4.1 LSMs: A Surprisingly Good Index for CXL?

The core thesis of this paper is that widely used log-structured merge trees (LSMs) [3, 17, 27, 30, 31, 33, 36, 37, 48, 49, 60, 71, 72, 76, 79, 87, 97], data structures designed for disk-based storage, fit the partly coherent CXL model far better than any in-memory index. The crux of this fit lies in how LSMs organize data. An LSM splits its data into two components. The



(a) Across YCSB workloads (b) YCSB-A, varying threads

Figure 3. LSM vs. In-Memory Indexes on CXL.

first is a small, in-memory write buffer called the *memtable* that absorbs incoming writes in place. The second is a collection of immutable sorted files called *SSTables* (sorted string tables), organized across multiple levels. When the memtable fills up, it is frozen and written out as a new SSTable. Once created, an SSTable is immutable and cannot be modified. Periodically, SSTables are merged to create new ones in a process called *compaction*. After the merge is completed, the old SSTable is discarded.

The key feature that makes LSMs well suited as indexes for partly coherent memory is their small updatable surface area. Unlike in-memory indexes, where updates may occur throughout the entire data structure, LSM trees confine in-place updates to the memtable, while the SSTables are immutable. Because the memtable is small, it fits in the SCR, where hardware guarantees coherence. In contrast, SSTables, which store the vast majority of the data, can be placed in the LNR. Since SSTables are never updated in place and are only deleted during background compaction, there is no need to track coherence during their lifetime.

Compactions replace SSTables, and we want to garbage collect old SSTables and reuse their allocation in the LNR. To avoid coherence issues, we use the software-coherence approach again. Each SSTable has an increasing id, and the system keeps in the SCR some metadata about SSTables: a list of active SSTable ids, their location in the LNR, and their size. If a host sees a new id in the active list, it flushes the entire SSTable region in the LNR to clear it from its cache. This ensures that subsequent reads of the SSTable return fresh data. Given that SSTables are large (on the order of a few MBs), their total number is relatively small, especially compared to the number of tree nodes in an ART or a B-tree, allowing the SSTable metadata to easily fit in the SCR.

A question that naturally arises is whether out-of-place update data structures, such as copy-on-write (CoW) B-trees [85], also solve the updatable surface area problem like LSMs. Specifically, with CoW, updates create new versions of B-tree nodes rather than modifying them in place. However, CoW structures like B-trees suffer from severe write amplification as every update to a small portion of a node requires copying the entire node to a new location, wasting significant memory bandwidth. For example, in CoW B-trees with leaf capacity B, updating one key copies the entire leaf, yielding a write amplification of $\Theta(B)$. This cost is further exacerbated by the fact that all nodes along the path up to the root must

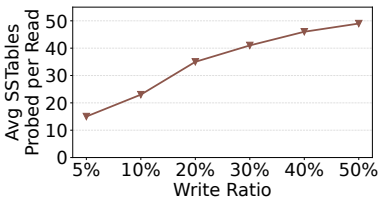


Figure 4. Compaction Bottleneck in SC-LSM.

be updated. In contrast, LSMs have lower write amplification than B-trees. A leveled LSM's amortized write amplification is $\Theta(T \cdot \log_T \frac{N}{B})$, where T is the size ratio between consecutive levels (for typical configurations, T is relatively small), and B is the SSTable size [11, 98]. Prior work measured roughly 4× lower write amplification for RocksDB (LSM) compared to WiredTiger (B-tree) [80]. This is because an LSM batches updates in every level starting from the memtable, and as a result, when an SSTable is overwritten, larger portions of it have received new updates [11, 79] compared to CoW B-trees, which lack such a batching opportunity. Thus, LSMs reduce memory bandwidth overhead greatly over CoW B-trees.

Based on our core thesis, we port the LSM to the in-memory partly coherent model, which we call SC-LSM. SC-LSM places the memtable entirely in the SCR, as it is small (typically a few megabytes). It then allocates all SSTables in the LNR. Figure 3 (a) compares the performance of SC-LSM against SC-ART and SC-BTree under three YCSB workloads. We use a dataset of 100M keys and an SCR capacity of 128 MB. As shown, SC-LSM significantly outperforms SC-ART because it does not suffer from excessive cache flushes. SC-LSM and SC-BTree perform closely for read-heavy and read-only workloads (B and C). However, SC-LSM achieves up to 2.2× higher throughput than SC-BTree under the write-heavy (A) workload (Figure 3 (b)). This is because with more writes, SC-BTree suffers from excessive false invalidations, leading to poor performance.

4.2 Updatable Upper Layers

While SC-LSM is a significant improvement over ported in-memory indexes, it suffers from a well-known LSM limitation. With high write rates, compactions cannot keep up, as we now explain. As described in § 2.3, L0 SSTables may overlap, so a point query must search through every L0 SSTable in the worst case. At high write rates, the memtable fills and flushes quickly, producing L0 SSTables faster than compaction can drain them to lower levels. As L0 grows, reads must probe more SSTables, increasing read latency. As shown in Figure 4, at 50% writes, SC-LSM probes 49 SSTables per read on average, compared to 15 at 5% writes. Traditional LSMs handle this by stalling writes to slow L0 growth, but this degrades write performance. Another concern is that although LSMs have lower write amplification than B-trees [98], it increases with more compactions. These are well-known LSM issues, not unique to SC-LSM [72].

However, the in-memory setting presents a unique opportunity to solve these problems. The key insight is that

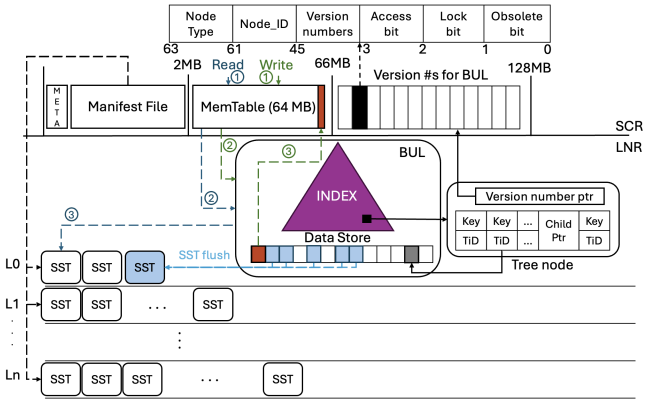


Figure 5. Prism Architecture. *The SCR (128 MB) holds the manifest (2 MB), memtable (64 MB), and version numbers for the BUL (62 MB). The BUL data and all SSTables (L0 through Ln) reside in the LNR.*

compaction pressure can be fundamentally reduced *when the upper layers of the LSM can be updated in place*, as this reduces the number of SSTables created. This solution is a non-starter for disk-based LSMs, where in-place updates require random writes that squander the sequential I/O benefits of the LSM design. In memory, however, random in-place writes are not as expensive. When a key is updated multiple times, each update can overwrite the existing entry in the upper layer, rather than producing a new entry that becomes part of a new SSTable. As a result, we reduce compaction frequency, keep L0 small, and reduce write amplification.

Does this mean we are increasing the updatable surface area? Yes, but we do so carefully. We *bound* the size of the mutable upper layer such that its version numbers, even if tracked at fine granularity, fit in the SCR. We call this layer the *bounded updatable layer* or BUL. The end result is a design that combines a bounded, fine-grained mutable layer that absorbs writes in place with a coarse-grained immutable SSTable layer that handles the bulk of the data with no false invalidations. We realize these ideas to build Prism.

5 Design

We first provide an overview of Prism (§5.1). We discuss BUL that absorbs writes and reduces compaction pressure (§5.2) and SCR-aware memtable design that treats memtable as a cache for hot keys (§5.3). We then describe the SSTable and manifest structure (§5.4), how coherence is maintained across tiers (§5.5), and the read and write paths (§5.6).

5.1 Architectural Overview

Figure 5 shows the high-level architecture of Prism. Prism divides data across three tiers, all backed by CXL shared memory. (1) At the top is SCR (128 MB), which in turn is divided into three regions. The *manifest* (2 MB) records all active SSTables, their level assignments, key range boundaries, and SSTable IDs. The *memtable* (64 MB) is an ART-backed structure, maintained entirely in SCR, where hardware coherence keeps it consistent across hosts with no software

overhead. The remaining 62 MB stores *version numbers* for BUL, one per node in its tree, enabling fine-grained software coherence. (2) Below SCR, BUL resides in LNR. It is a bounded, in-place updatable ART that absorbs writes and reduces compaction pressure. Each tree node in BUL contains keys, child pointers, and a pointer back to its version number in SCR. (3) Below BUL, immutable SSTables are organized into multiple levels (L0 through Ln) following a leveled compaction policy [79], where SSTables at each level are merged into the next when the level exceeds its size target. Each SSTable contains a sorted data region and an internal ART index built at creation time. SSTables are never modified after creation. To handle reuse, coherence is tracked at SSTable granularity. A single designated host runs all background operations, including BUL flushes and SSTable compactions. Together, the memtable and BUL form the updatable surface area, while the rest of the data lives in immutable SSTables in LNR.

5.2 Bounded Updatable Layer (BUL)

As shown in Figure 4, the biggest bottleneck in SC-LSM is L0 accumulation. To reduce the number of SSTables produced and alleviate L0 size growth, we dedicate a portion of L0 to be in-place updatable in the form of an ART, depicted in Figure 5 as BUL. Instead of flushing the memtable directly to L0 as a new SSTable, BUL absorbs entries flushed from the memtable in-place. Thus, when the same keys are repeatedly flushed from memtable, instead of flushing them as part of separate SSTables, they get updated in place in BUL. Because repeated writes are absorbed in place, only unique, cold keys are eventually flushed from BUL to L0. This directly reduces the total number of SSTables produced, keeping L0 small.

BUL can accommodate significantly more entries in place than the memtable because the data is stored in LNR. BUL uses version numbers stored in SCR to ensure its coherence, similar to SC-ART. Unlike SC-ART, BUL is explicitly bounded in size so that all its version numbers fit within the remaining SCR capacity. Eventually, BUL fills with unique keys, so we need to flush it to SSTables. Since BUL is significantly larger than an SSTable, we cannot flush the entire BUL at once, as it would increase the data written to L0. Instead, BUL flushes entries to L0 one SSTable at a time using a clock sweep policy (§6). The clock sweep flushes cold keys as a sorted SSTable to L0, while recently accessed keys remain in BUL.

Internals of BUL. Internally, BUL uses a modified ART-OLC [65], where each node's version entry is a 64-bit word stored in SCR (shown at the top of Figure 5). The version word packs both concurrency control and coherence metadata. Bit 0 is the *obsolete* flag, set when a node is logically deleted (replaced by a grown or shrunk node). Any thread that reads this flag restarts its operation because the node is queued for epoch-based reclamation [35], since OLC readers traverse nodes without holding locks. Bit 1 is the *lock* bit, which serializes concurrent writers to the same node. Bit 2 is the *access* bit, used by the clock sweep to track node hotness for eviction. Bits 3–44 store the 42-bit *version counter*, incremented on every lock and unlock cycle. Readers snapshot this counter before reading node data and compare it afterward; if it changed, the reader must restart (after flushing the relevant cachelines). This version counter also helps identify stale caches (§5.5). Bits 45–61 store a 16-bit *node ID*, a strictly increasing ID assigned at creation time to handle node reuse. Bits 62–63 encode the *node type* (Node4, Node16, etc.).

Each tree node in BUL contains keys, child pointers, and a per-key *TiD* (tuple identifier). TiD is an offset into the *data store*, a separate region within BUL in LNR that holds the actual data records. Coherence for a data record is tied to the version number of the node that points to it (§5.5).

5.3 SCR-Aware Memtable

SCR is both the highest-performing and the most constrained resource in the system. Data in SCR benefits from hardware coherence with zero software overhead, but only 64 MB is available for the memtable. This scarce capacity should be reserved for frequently written keys that would otherwise incur repeated software coherence checks in BUL.

In the design described in §5.2, all writes enter through the memtable regardless of their access frequency. The memtable then flushes its entire contents to BUL when full. This design has two problems. First, keys that are written only once (one-hit wonders) occupy precious SCR space without benefiting from hardware coherence, only to be flushed shortly after. Second, when the memtable fills, if the entire memtable is flushed to BUL, then it requires inserting each key from the memtable into BUL, which requires a per-key ART traversal and insertion, making this flush expensive.

We address both problems by inverting the role of the memtable from a write buffer into a *cache for hot keys*. Writes check the memtable and update in place if the key already resides there, but otherwise go directly to BUL. New keys can only enter the memtable through promotion from BUL, which occurs when a key in BUL receives a subsequent write. This ensures that every key in the memtable has received frequent writes and can benefit from hardware coherence.

A key that is written only once goes to BUL, where it is eventually marked cold by the clock sweep and evicted to an SSTable in L0. This one-hit wonder filtering [105] ensures that infrequently written keys never consume SCR capacity. When the memtable reaches capacity, the least recently used entries are evicted back to BUL in small batches, freeing SCR space for hotter keys.

This inversion also eliminates the expensive bulk flush from the memtable to BUL. Evictions happen incrementally through LRU, spreading the cost over time.

5.4 SSTables and Manifest

Each SSTable is an immutable, sorted file in LNR containing a data region and an internal ART index. The data region stores key-value pairs. The ART index is built at SSTable

creation time and maps keys to offsets within the data region. Unlike disk-based LSMs, Prism does not use Bloom filters. Bloom filters are designed to avoid expensive disk I/O on negative lookups, but in the CXL memory setting, probing the SSTable's ART index is already a memory access with comparable cost to checking a Bloom filter, making the additional memory overhead of Bloom filters unjustified.

The manifest is a small structure in SCR that records all active SSTables, their level assignments, key range boundaries, and SSTable IDs. Hosts consult the manifest to determine which SSTables to probe during reads and to detect when compaction has replaced an SSTable. Because the manifest resides in SCR, hardware provides coherence for it.

5.5 Coherence and Correctness Across Tiers

With all three tiers in place, we now describe how Prism maintains coherence across hosts.

The property Prism maintains is that a read of a key returns the value most recently written to that key by any host. The difficulty is, a host may hold a cached copy of a location in LNR that another host has since overwritten, and the hardware does not inform it. A host may therefore serve a read from its own CPU cache only when the protocol can establish that the cached data still matches CXL memory. The memtable sidesteps this by residing entirely in SCR. The other two tiers keep their data in LNR and pair it with state in SCR, which the hardware does keep coherent, and a host consults that state before trusting its cache. Below is the protocol used for each tier, and then for the transitions between tiers, since a key does not stay in one tier for its lifetime.

Memtable. The memtable resides in SCR. Thus, all hosts can access the memtable without worrying about coherence.

BUL. BUL stores its data nodes in LNR with per-node version numbers in SCR. The root node resides in SCR. Each host locally maintains the latest version number for each node it has read and the latest node ID it has seen. Starting from the root, the host traverses the tree following the ART-OLC protocol. Before reading each child node from LNR, the host checks the corresponding version number in SCR. Three cases arise. First, if a host encounters a node ID it has not seen before, the node's memory region may contain stale cached data from a previously recycled node, so it flushes and reads the node from CXL memory. Second, if the version number is unchanged since the host last accessed the node, the host reads from its CPU cache without any flush. Third, if the version number has changed since the last access, the host updates its local copy of the version number, flushes the stale cache lines, and restarts the traversal from the BUL root. The traversal continues until the host reaches a leaf node, which contains a pointer to the data record. Coherence for the data record is tied to the leaf node's version number. When a writer updates a data record, it also updates the node's version number, so readers detect the change as described above. On writes, the host acquires the node's lock bit and increments the version counter. After modifying the node or its data record in the LNR, the host flushes modified cache lines to push data to CXL memory, then releases the lock and increments the version counter again. Inserting a new child node changes the parent's version counter, making all hosts aware of it on their next traversal. This provides fine-grained coherence using one SCR entry per ART node. Because BUL is bounded (§5.2), all version numbers fit within SCR.

SSTables. Cache coherence is not a concern during the lifetime of an SSTable as it is immutable. However, when compaction replaces an SSTable, the underlying memory region may be recycled for a new SSTable while hosts still hold stale content in their CPU caches. To detect this, each SSTable is assigned a strictly increasing SSTable ID. When a host encounters an SSTable ID it has not seen before, it flushes the CPU cache lines for that memory region and reads from CXL memory. If the host has already seen the ID, no flush is needed. The manifest resides in SCR. When creating a new SSTable, the designated host flushes its cache lines before updating the manifest.

Correctness during Transitions. As a key moves between tiers, its coherence mechanism changes. Each transition is ordered so that the key is visible under the new mechanism before it is removed from the old one.

On promotion from BUL to the memtable, the writer holds the BUL leaf lock throughout the transition, so no other writer can modify the key in BUL. The writer inserts the key into the memtable in SCR, and only then deletes the entry from BUL and releases the lock. During this window, a concurrent reader may find the key in both the memtable and BUL. This is safe because the read path always checks the memtable before BUL. A reader mid-traversal in BUL will detect the version change from the deletion and restart from the top of the read path, where it will find the key in the memtable.

On eviction from the memtable back to BUL, the eviction thread holds the BUL leaf lock for the target node. It first inserts the key into BUL with a new version entry in SCR, and only then removes the key from the memtable. A brief window exists where the key is visible in both locations, but both copies hold the same value.

On eviction from BUL to an SSTable, the flush thread holds the leaf locks for all entries being flushed. It collects cold entries, writes them into a new immutable SSTable in LNR, updates the manifest in SCR with the new SSTable ID, and only then removes the entries from BUL and reclaims their version entries from SCR. Until BUL entries are removed, readers may find the key in both BUL and the new SSTable, but the values are identical.

If a concurrent request is already traversing BUL and the target node becomes unavailable due to a promotion or eviction, the traversal restarts from the memtable, since the key may now reside in the memtable or an SSTable.

Correctness Argument Summary. At any instant, a key is in the memtable, in BUL, in an SSTable, or in transition between two of them, so the four cases are exhaustive. Hardware coherence covers the memtable. The per-node version check covers BUL, flushing whenever a version moves or a node ID is unfamiliar. The manifest covers SSTables, publishing a new ID only after the region is flushed (§5.4). The ordering rule covers transitions, making a key readable under its new mechanism before it stops being readable under the old one, while the read path visits the tiers newest first and returns the newer of two briefly visible copies (§5.6). In each case, a host trusts a cached copy only once the protocol has established that it still matches CXL memory.

Model Checking. To validate the correctness, we also built a model (in Python) of partly coherent memory and Prism's software coherence protocol. To emulate non-coherent memory, we model hosts with CPU caches that are not kept coherent with the underlying memory region. Specifically, a host that reads from LNR keeps the cached bytes until the protocol tells it to flush, so a missing flush surfaces as a stale read. The model also captures how Prism works, i.e., its components like memtable, BUL, SSTables, its software-coherence protocol, and also the movement of keys between tiers.

We then built a checker to check this model. We checked the model with a trace of 16M index operations across varying host counts, dataset sizes, and tier capacities. We confirmed that the checker finds no bugs and that the correctness invariant holds. To validate that the model and checker are effective in catching bugs in the software-coherence protocol, we purposefully tweaked the coherence protocol to introduce bugs (e.g., dropping a writer's flush after its version bump, skipping a reader's revalidation of the version number, and publishing a recycled SSTable's new ID before its contents reach LNR). When these incorrect modifications are introduced, our checker flags stale reads immediately.

5.6 Putting it All Together

We now describe how the different pieces work together. Figure 5 shows the write (green) and read (blue) paths.

Write path. Inserts or updates to a key happen through the memtable and BUL, never accessing SSTables directly.

① *Memtable.* A host first checks if a key exists in the memtable. If so, it updates the value in place and promotes the entry to the LRU head. If not, the host proceeds to BUL.

② *BUL insert.* The host traverses BUL from the root (§5.5). If the key does not exist, the host inserts it. If BUL occupancy exceeds 90%, the host performs an eviction to L0 before proceeding. If BUL is full, the write stalls until space is available.

③ *BUL promotion.* If the key already exists in BUL, it has been written before and is thus promoted to the memtable and deleted from BUL.

Delete path. Prism handles deletes using tombstones, like standard LSMs. Deletes follow the same write path but insert a tombstone marker instead of a value. The tombstone propagates downward through eviction and compaction like any other entry. During compaction, tombstones suppress older versions of the same key. A tombstone is permanently removed only when compaction reaches the lowest level.

Read path. The read path checks the three tiers in order, returning immediately on the first hit, performing the coherence checks described in §5.5.

① *Memtable.* The host looks up the key in the memtable, returning the value on hit. On miss, it goes to BUL.

② *BUL.* Starting from the root, the host traverses BUL. On hit, the host returns the value. On miss, the read falls through to the SSTables.

③ *L0 SSTables.* If the key is not found in BUL, the host consults the SSTable manifest stored in SCR. L0 SSTables may have overlapping key ranges because each one corresponds to an independently flushed batch from BUL, so the host must check every L0 SSTable. The host iterates from newest to oldest, probing each SSTable's internal ART index, and returns the value on the first hit.

L1+ SSTables (coarse-grained coherence). From L1 onward, SSTables within each level have non-overlapping key ranges, so at most one SSTable per level can contain the target key. The host iterates over the manifest entries in SCR and uses the min and max key boundaries to identify the single candidate SSTable at each level without leaving SCR. Once identified, the host probes the candidate's ART index. If no level contains the key, the host returns not found.

Range queries. The host performs an ordered scan across all three tiers and merges the results. The memtable and BUL both support ordered iteration through the ART structure. For SSTables, the host identifies all overlapping SSTables at L0 and at most one per level from L1 onward using the manifest's key-range boundaries. The host merges the streams from all tiers in key order, returning the most recent version of each key. Coherence during the scan follows the same per-tier mechanisms as point lookups.

5.7 Failure Model

Prism assumes a fail-stop model, the same assumption made by prior CXL systems such as Tigon [51]. A failure of any host or the CXL memory device causes the entire system to halt. All state resides entirely in memory, so there is no recovery. Durability is outside the scope of this work.

6 Implementation

Prism is implemented as a C++ library in ~11K LOC. Each host links against the Prism library and accesses the shared index through its API.

Cache Line Management. To invalidate stale data on reads, we use `clflushopt` with an `sfence` before and an `mfence` after to enforce ordering. For write-back operations, we use `clwb` with the same fence ordering.

Background Tasks. Prism runs three background tasks on a single designated host. First, when the memtable reaches 90%

occupancy, an eviction thread removes 10% of the memtable entries and re-inserts them into the BUL in a non-blocking manner. Second, the BUL periodically flushes cold entries to L0 using a clock sweep policy. Each ART node in the BUL has an access bit. The clock hand sweeps through the ART nodes, clearing set access bits. If a node's access bit is already clear, it has not been accessed since the last sweep and is considered cold. Cold entries are collected and flushed as SSTables to L0. Each periodic flush produces one to two SSTables. When the number of BUL entries exceeds a high watermark, an immediate eviction runs until the entry count drops below a low watermark, after which periodic flushing resumes. Third, a single compaction thread performs N-way merges in the background, where N is configurable per level. The compaction thread reads and merges entries from the source SSTables in sorted order. Once the merged output is ready, 8 worker threads construct the output SSTables in parallel by splitting the output into SSTable-sized chunks, building ART indexes, and writing the final SSTables to the LNR. Level file count targets are 4 for L0, 4 for L1, 10 for L2, 100 for L3, 1000 for L4, and 10000 for L5. Running these tasks on one host is an implementation decision, not a requirement of the design; we see no fundamental obstacle to distributing them. The manifest already resides in SCR, giving hosts a coherent place to agree on which SSTables are live, so the parallel compaction strategies used in existing LSM engines apply [9, 34]. We leave a distributed implementation of compactions to future work.

Memory management. BUL allocates ART nodes individually from a free list in LNR. When a node is logically deleted (e.g., during a shrink or promotion), it is marked obsolete and reclaimed via epoch-based reclamation after concurrent readers finish. Reclaimed nodes return to the free list, and their version entries in SCR are also reclaimed. SSTable memory regions are recycled after compaction replaces an SSTable. The SSTable ID mechanism ensures safe recycling.

7 Evaluation

Our evaluation answers the following questions.

- How does Prism perform in read-only workloads? (§7.2)
- How does Prism perform in read-write workloads? (§7.3)
- How does Prism perform under range queries? (§7.4)
- How does an application perform using Prism? (§7.5)
- How does Prism perform with workloads that have little or no skew? (§7.6)
- How does the idea of BUL help improve performance of the base SC-LSM by reducing compactions? How do the other design decisions in Prism help improve performance further? (§7.7)
- How close does Prism get to the SC-ART configuration that is given unlimited SCR? (§7.8)
- How does Prism perform under real-world traces? (§7.9)
- How sensitive is Prism to SCR size? (§7.10)
- How does Prism perform against a state-of-the-art RDMA-based disaggregated memory index? (§7.11)
- How does Prism perform against state-of-the-art software coherence approaches for partly coherent CXL? (§7.12)

7.1 Experimental Setup

Multi-host CXL 3.0 devices are not yet commercially available, so we emulate CXL sharing across hosts using a 4-socket server with Intel Xeon Gold 6418H processors. One NUMA node (node 0) with 64 GB of DRAM serves as the shared CXL memory device, while the remaining three nodes act as distinct hosts, each with 24 physical cores (hyperthreading disabled) and 64 GB of local DRAM. We emulate CXL access latencies by throttling the uncore frequency of node 0 from 2.4 GHz to 0.8 GHz, following prior work [5]. Threads are distributed evenly across the three host nodes. SCR is sized at 128 MB, and the rest of node 0 serves as the LNR. In our testbed, the entire memory is hardware coherent and we emulate the partly coherent model atop this hardware like recent work [51]. Specifically, Prism and the baselines perform coherence checks and issue cache flushes as needed before accessing the LNR memory. Thus, the performance overhead of software coherence we report is faithful. However, since the underlying memory is itself coherent, it would not expose correctness issues in the coherence protocols. Prism's correctness is verified by the model checker (§ 5.5).

Baselines. We compare Prism against the in-memory indexes that we port to the partly coherent model using software coherence (§3.1), namely SC-ART and SC-BTree. SC-ART uses adaptive node sizes and its version numbers do not fit in SCR and overflow into LNR. SC-BTree uses a fan-out of 128 and its version numbers fit in SCR.

We extensively tuned the baselines' performance, especially how they utilize SCR. SC-ART stores only version numbers in SCR rather than whole tree nodes. SC-ART further optimizes version-number placement by adopting a level-based placement policy, where version numbers of the nodes closer to the roots are preferred in SCR because more operations traverse them. Compared to a variant that maintains version numbers in SCR on a first-come, first-served basis, this SC-ART variant raises throughput by 50%. For SC-BTree, we experimented with different fanouts and chose the best setting (a smaller fan-out overflows the SCR with version numbers while a larger one incurs higher false invalidation).

Because SC-ART spills version numbers into LNR, it must increment them there without racing. SC-ART does so with a lock bit embedded in the version number and atomic operations on LNR memory that SC-ART assumes the hardware provides. Note that in Prism, version numbers never reside in LNR (because it bounds the updatable surface area) and thus Prism does not have to rely on the assumption that LNR provides atomics.

Default parameters. Unless stated otherwise, we pre-load the index with 100M key-value pairs with 24-byte keys and

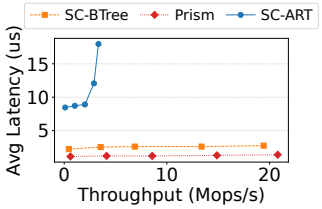
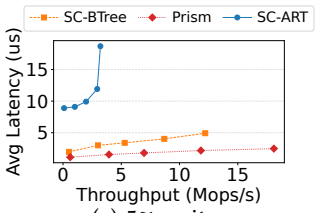
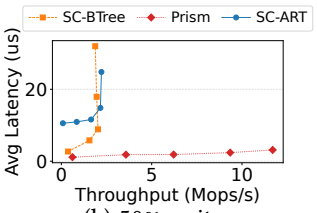


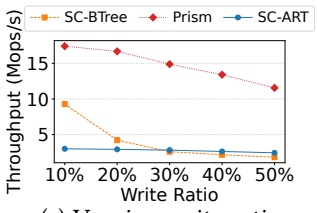
Figure 6. Read Performance.



(a) 5% writes



(b) 50% writes



(c) Varying write ratio

Figure 7. Mixed Read-Write Workloads.

Figure 8. Range Queries Under Write Pressure.

100-byte values. We use a Zipfian access distribution with α=0.99, following the YCSB specification [24].

7.2 Read-Only Performance

Figure 6 plots latency against throughput under a read-only workload. SC-ART achieves the lowest peak throughput because its fine-grained per-node version numbers overflow the 128 MB SCR at this dataset size (~40% coverage), forcing cache flushes when accessing them in LNR (§ 3.2). SC-BTree achieves up to 6× higher peak throughput than SC-ART because it fits all version numbers within SCR. Without writes, SC-BTree incurs no cache flushes from false invalidations, and no penalty from its coarse granularity. Prism also avoids cache flushes in this workload and improves peak throughput notably (up to 6.3×) over SC-ART. Prism also slightly improves over SC-BTree as the ART-OLC indexes used in SSTables provide faster point lookups than B-tree traversals.

7.3 Mixed Read-Write Workloads

Figures 7(a) and 7(b) plot latency against throughput for 5% and 50% writes, respectively. SC-ART suffers from poor performance at both write ratios due to version-number accesses requiring cache flushes. SC-BTree performs well at 5% writes (peak 12.2M ops/s) due to fewer false invalidations. However, at 50% writes, its throughput collapses to 1.9M ops/s due to excessive false invalidations, performing even worse than SC-ART. While coarse-grained coherence helped SC-BTree in read-heavy cases, with more writes, as each write falsely invalidates an entire node, its performance collapses. In contrast, Prism's performance scales when adding more threads. At 50% writes, it significantly outperforms SC-BTree by up to 6.2× and SC-ART by up to 5.6× in peak throughput. This is because Prism incurs far fewer cache flushes due to its three-tier coherence design. Most writes are absorbed within the first two layers with efficient fine-grained coherence. The memtable efficiently captures repeated writes via hardware coherence, requiring no cache flushes. The bounded BUL then tracks coherence precisely without overflowing SCR, thereby avoiding the false invalidations typical of coarse-grained approaches. Finally, although the third layer receives writes not absorbed in the first two layers, even with coarse-grained out-of-place updates for SSTables, it achieves lower false invalidations due to the batching of writes in LSMs.

Figure 7(c) sweeps the write ratio from 10% to 50%. SC-BTree's peak throughput drops sharply because larger write ratios amplify the false invalidation across all hosts. SC-ART suffers from low throughput throughout. Although Prism's performance drops with increasing writes, it still significantly outperforms SC-ART and SC-BTree at all write ratios. Prism's performance drops with more writes for two reasons. First, it incurs cache flushes in BUL for all writes and upon detecting stale reads, which is the cost required for software coherence. Second, more writes increase the number of SSTables accessed by reads in the third layer. A higher write ratio fills the BUL sooner and flushes it more often. Each flush adds one or two SSTables to L0, and because L0 SSTables have overlapping key ranges, a read must examine all of them until compaction drains L0 into L1 (§7.7).

7.4 Range Queries

We run a mixed workload with short-range queries and writes and vary the write ratio. We follow YCSB-E, with scan lengths drawn uniformly between 1 and 100 keys. Figure 8 shows the results. At 5% writes, SC-BTree outperforms Prism. This is because B-tree nodes store keys in sorted order within large contiguous nodes, making sequential scans efficient. Prism must merge results across multiple SSTables and tiers, incurring higher per-scan overhead. While this is a known trade-off of LSM-based designs, Prism's advantages show up with more writes.

As the write ratio increases, SC-BTree's performance drops sharply, declining 8× from 5% to 50% writes as false invalidations force repeated flushes. Prism, in contrast, outperforms SC-BTree from 10% writes onward, achieving 6.0× higher throughput at 50% writes. Prism's throughput first rises with more writes because updates are cheaper than scans. As writes increase further, cache flushes in BUL and larger L0 SSTables reduce performance.

7.5 KV Store Application under YCSB

We now evaluate the performance of a key-value store application atop Prism using YCSB macrobenchmark workloads [24]: A (50% w, 50% r), B (5% w, 95% r), C (read-only), D (5% insert, 95% r; read latest), E (5% w, 95% short-range scans) and F (50% read-modify-write, 50% r). Figure 9 shows the results. SC-ART has low performance in all workloads due

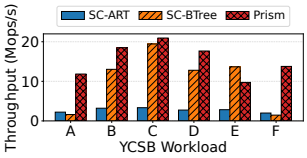


Figure 9. YCSB Workloads.

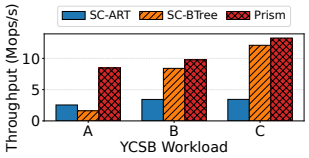


Figure 10. Uniform Access Distribution.

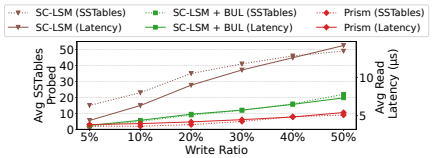


Figure 11. Compaction Impact on Reads.

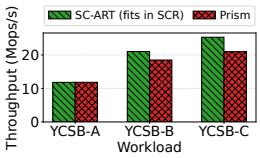


Figure 12. Prism vs. Unlimited SCR Size.

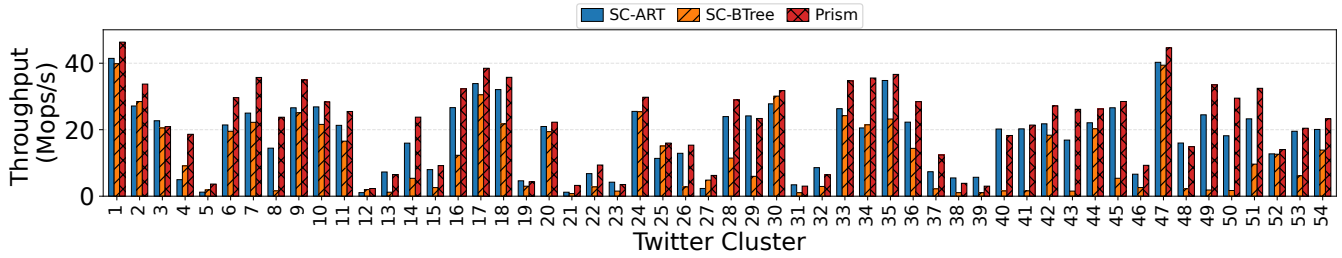


Figure 13. Twitter Production Traces.

to version number overflows, achieving the lowest throughput on read-dominant workloads (B, C, D). On write-heavy workloads (A and F), SC-BTree drops below SC-ART due to false invalidation under coarse-grained coherence (§ 7.3).

On workload C, Prism improves slightly over SC-BTree (1.1×). On workload B, Prism improves by 1.4× because even 5% writes begin triggering false invalidations in SC-BTree. As writes increase, with write-heavy workloads (A and F), Prism outperforms SC-BTree by 7.4× and 9.4×, respectively. On workload D, Prism achieves 1.4× over SC-BTree because recently inserted keys reside in the memtable or BUL, from which they are served with low latency. On workload E (range scans), with only 5% writes, SC-BTree does not suffer significantly from false invalidation and since B-trees are generally efficient with range queries, SC-BTree outperforms Prism. However, as we showed in §7.4, Prism starts to outperform SC-BTree, even with range queries, when the percentage of writes is 10% or higher.

7.6 Uniform Access Distribution

So far, workloads had a Zipfian distribution, where a few keys are more popular than others. Prism benefits from that skew because the memtable caches hot keys in SCR and repeatedly updated keys are absorbed in the BUL. Uniform workloads remove both these advantages for Prism. Figure 10 shows the results for YCSB workloads A, B, and C using a uniform key distribution. While Prism still outperforms the baselines, the improvements are narrower than the ones reported under a Zipfian distribution (§7.5). The margins narrow because without skew the memtable and BUL absorb fewer repeated writes. However, Prism outperforms the baselines, by 2.8× to 3.8× over SC-ART and by 1.1× to 5.2× over SC-BTree. This is because SC-ART still has more version numbers than SCR can hold and SC-BTree pays a false invalidation per write.

7.7 Ideas and Techniques Ablation

As write load increases in LSMs, L0 fills quickly with SSTables, affecting read performance as reads must access many overlapping L0 files. We evaluate how BUL and Prism's design choices address this concern. We perform an ablation study of three systems: SC-LSM (the base LSM of §4.1), SC-LSM + BUL (SC-LSM augmented with the bounded updatable layer), and Prism (the full system). Figure 11 shows the average number of SSTables probed per read and the average read latency as writes grow from 5% to 50%.

As shown, SC-LSM suffers from severe L0 accumulation, with reads probing 15 SSTables at 5% writes and 49 at 50% writes, sharply increasing read latency. Adding the bounded updatable layer (SC-LSM + BUL) reduces the number of SSTables probed and read latencies. With BUL, only 2 SSTables are probed at 5% writes as repeated writes are absorbed in place within BUL rather than being flushed to L0. At 50% writes, read latency increases for SC-LSM + BUL, but it still probes significantly fewer SSTables than SC-LSM. This shows the benefit of adding an updatable upper layer to an LSM.

Prism, due to its design choices, reduces the number of SSTables created. This is because Prism uses the memtable as a cache instead of flushing it to BUL. Furthermore, by writing first to BUL and only promoting to the memtable on repeated access, one-hit wonders do not displace popular keys to BUL. Reducing writes to BUL, in turn, reduces the writes flushed to L0. Thus, Prism reduces read latency significantly across all write ratios compared to SC-LSM + BUL, demonstrating the benefit of the design choices in Prism beyond BUL.

7.8 Comparison with Unlimited SCR Size

We compare Prism (with 128 MB of SCR) against an ideal system that runs SC-ART with an unlimited SCR, where all version numbers fit and coherence is tracked precisely without false invalidations. This system is unrealistic, but it represents the best performance achievable with software coherence for an in-memory index.

Figure 12 shows the results. On workload A (50% writes), Prism matches the ideal SC-ART. With read-heavy workloads (B and C), Prism reaches 83–88% of the ideal SC-ART throughput, which is good performance given that Prism has

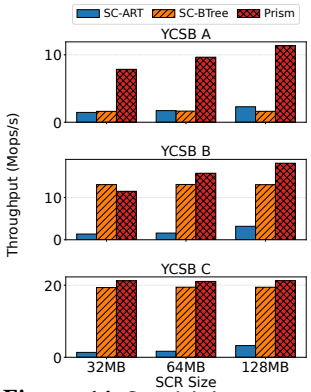


Figure 14. Sensitivity to SCR Size.

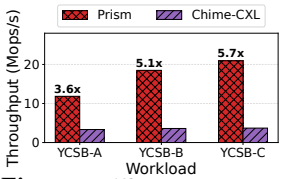


Figure 15. Prism vs. Chime-CXL.

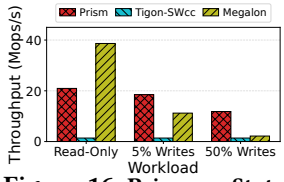


Figure 16. Prism vs. State of the art.

a much smaller SCR. The gap is mainly due to the overhead of the LSM read path, where reads must check the memtable, BUL, and potentially multiple SSTable levels, compared to a single ART traversal in SC-ART.

7.9 Real-World Traces

We now evaluate Prism on production traces from Twitter's in-memory caching clusters [99]. Most of these clusters follow a skewed Zipfian popularity distribution, and several are more skewed than YCSB's default access distribution [104].

Figure 13 shows throughput for each cluster. Prism outperforms SC-BTree on all 54 clusters, by 3.9× on average and up to 18×. Its margin grows with the write ratio, from 1.2× below 2% writes to 5.9× at or above 50% writes. This is because every write can cause false invalidation in SC-BTree regardless of the access distribution, while Prism absorbs writes in its fine-grained upper layers.

Prism outperforms SC-ART on 43 of the 54 clusters, by 1.3× on average and up to 3.7×. Of the remaining 11 traces in which Prism underperforms, six of them (13, 23, 31, 32, 38, 39) are dominated by writes with an average write ratio of 84% and are also less skewed, with an average Zipf α of 0.12. On these traces, Prism's performance drops for two main reasons. First, a higher fraction of writes causes more cache flushes in BUL (§ 7.3). Second, less skew leaves few repeated writes for its upper layers to absorb, resulting in more SSTables (§ 7.6). The remaining five traces (3, 19, 29, 40, 48) hold fewer than 40M keys, so SC-ART's version numbers fit within SCR and it runs at its best. Even then, SC-ART wins only by a small margin of 7% on average.

7.10 Sensitivity to SCR Size

We now evaluate Prism under SCR sizes smaller than the 128 MB used in previous experiments. We do so by varying SCR size from 128 MB to 32 MB and running YCSB A, B, and C workloads. Figure 14 shows the result.

First, SC-ART uses fine-grained version numbers. As a result, with smaller SCR, more of its version numbers spill to LNR, resulting in performance drops by up to 2.4× compared to the 128 MB SCR case. Second, SC-BTree's throughput does not significantly vary because its version numbers fit within SCR even at 32 MB. However, its throughput drops by approximately 12× from YCSB C to YCSB A because writes cause false invalidations. Third, Prism's throughput is also affected by SCR size. Its performance drops by 31% on YCSB A and 37% on YCSB B as the SCR shrinks from 128 MB to 32 MB. This is because a smaller SCR reduces both the memtable cache and BUL's write-absorption capacity, causing more operations to reach SSTables. YCSB C is unaffected because BUL and memtable are not affected in a read-only workload. However, Prism still achieves the highest throughput across almost all workloads and SCR sizes. The only exception is YCSB B with a 32 MB SCR, where SC-BTree is 1.14× faster than Prism. This is because at 32 MB and YCSB-B, SC-BTree's version numbers fully fit within SCR and the small write fraction does not result in too many invalidations.

We also note that a small non-empty SCR is required to achieve reasonable performance in any system that implements a software coherence approach. This is because without an SCR, version numbers must reside in LNR, and every access to them requires cache flushes, degrading performance. In a separate experiment, where we ran SC-ART without any SCR (not shown), SC-ART's throughput degrades by roughly 20× versus running with a 128 MB SCR. These results motivate future hardware vendors to provide at least a small hardware-coherent region. We believe an SCR of 32–128 MB is a reasonable choice since it is a small fraction of a terabyte-scale memory device, yet it is sufficient to realize efficient coherence in software, as our experiments show.

7.11 Comparison with Distributed Memory Systems

We compare Prism against CHIME [73], a state-of-the-art hybrid index for RDMA-based disaggregated memory, ported to CXL shared memory (Chime-CXL). CHIME uses a B+ tree with hopscotch-hashing-based leaf nodes and a three-level optimistic synchronization scheme that embeds per-node and per-entry version numbers into each cache line. Figure 15 shows throughput across three YCSB workloads.

Prism outperforms Chime-CXL by 3.6× on workload A, 5.1× on workload B, and 5.7× on workload C. The gap stems from cache flush overhead in Chime-CXL's mutable B+ tree. Since nodes reside in the LNR, every leaf access requires at least two cache flushes, one to fetch the version number and one to verify it after reading the data. If verification fails, the reader must flush and re-traverse the entire path from the root. In contrast, Prism flushes the cache only if needed (upon detecting a concurrent write or stale caches). Thus, Prism's improvements increase from 3.6× to 5.7× as the read ratio increases.

7.12 Comparison with Existing CXL Systems

We compare Prism against two state-of-the-art systems that share data over partly coherent CXL memory, Tigon [51] and Megalon [46]. Tigon is a transactional database that stores its entire B+tree index and per-tuple coherence metadata in

SCR and the tuple data in LNR. When the SCR fills up, Tigon unshares tuples and migrates them to host-local memory. For a fair comparison, we implement Tigon-SWcc, a Tigon-like index without the transaction layer, keeping only the index and the coherence mechanism. Megalon similarly stores the coherence metadata in SCR. It improves over Tigon by replicating the index into the host's local memory, which allows it to allocate more per-object coherence metadata. It also dynamically allocates coherence metadata only for objects that receive writes. Therefore, unlike Tigon, read-only data does not require coherence metadata.

Figure 16 shows the performance for YCSB A, B, and C. Tigon-SWcc achieves only about 1.4M ops/s across all three workloads, which is 8.6× to 15.1× lower than Prism. This is because the 100M-key index does not fit in the 128 MB SCR, so Tigon-SWcc constantly migrates data in and out of CXL, and this data movement hurts its performance. For the read-only YCSB-C, Megalon outperforms Prism. This is because Megalon serves index lookups directly from the replica's local DRAM and accesses CXL only for the actual data record. In contrast, Prism traverses the index that resides in CXL memory. Megalon's local index, however, comes at a significant memory cost: roughly 32 bytes per key at each host, which is 25% of the key-value size for small object sizes like 100 bytes. This overhead grows with the number of hosts. Prism instead keeps a single copy of its index in CXL memory that all hosts share. Moreover, because Megalon needs coherence metadata for every written key, its performance drops with writes. With 5% writes, Megalon's coherence records overflow SCR with a large dataset, forcing it to constantly reclaim coherence records to make room for new ones. In contrast, Prism's SCR usage is bounded regardless of the dataset size. As a result, Prism outperforms Megalon by 5.5× on the write-heavy YCSB-A and 1.6× on YCSB-B.

8 Related Work

CXL Memory Systems. Prior CXL work has largely focused on memory tiering [41, 57, 66, 70, 75, 93, 94, 106, 110], pooling [40, 67], OS memory management [96], hardware characterization [53, 95], programming models [8], and coherence at scale [45], none of which consider sharing CXL memory across hosts. A separate line of work explores CXL shared memory for database design [51, 52], memory allocation and resilience [78, 109], partial failure tolerance [111], remote fork [4], and data processing [6, 10, 21]. These efforts are complementary to Prism, which focuses on index design under partial cache coherence. Among these, Tigon [51] and Megalon [46] are closest, as both share data across hosts by tracking per-object coherence metadata in SCR. However, their SCR usage grows with the number of shared objects, so with large datasets Tigon must migrate data in and out of CXL and Megalon must constantly reclaim metadata. Prism, in contrast, bounds its SCR usage regardless of dataset size and outperforms both under read-write workloads (§7.12).

In-Memory Indexes. Concurrent in-memory indexes are well studied, with designs based on radix-trees [7, 14, 44, 61, 64, 65, 74] and B-trees [22, 59, 63, 82, 89, 102]. These indexes target fully cache-coherent single-host systems and, as we show (§3), adapt poorly to partly coherent CXL.

Write-Optimized Data Structures. Log-structured merge trees (LSMs) [72, 79] are widely adopted [17, 33, 37, 49, 60, 76], with extensive research on key-value separation [71], compaction [28, 29, 81, 87], concurrency [39, 56], learned indexes [27], tiered storage [18, 107], and distributed deployments [13, 50]. These optimizations primarily target SSDs.

A few efforts port LSMs to persistent memory [20, 58, 100]. Unlike Prism, however, prior persistent-memory LSMs did not have to worry about coherence. Of this prior work, NoveLSM [58] is the closest, since it also makes an upper LSM layer updatable in place to reduce compaction. However, the details underlying this idea are different. In NoveLSM, only the size of persistent memory limits how large the in-place-updatable layer can grow. Prism, in contrast, must carefully handle coherence for the updatable portion in LNR. Furthermore, Prism bounds the size of BUL so that the coherence metadata fits within SCR.

Bϵ-trees [11, 15, 23, 54] are an alternative write-optimized design that avoids random IOs with benefits similar to LSMs. While other write-optimized structures could have suited Prism, we choose LSMs for their popularity.

Disaggregated Memory Indexes. A body of work on RDMA-based disaggregated memory [1] has built KV stores [32, 55, 77], one-sided RDMA-based indexes [68, 69, 73, 84, 101, 113], elastic systems [47, 91, 92, 112], and new abstractions for disaggregated memory [2, 16, 19, 38, 42, 43, 62, 83, 86, 90, 108]. These systems operate on a different hardware model, accessing remote memory through network IO rather than CPU loads and stores. Prism is the first index designed for partly coherent CXL shared memory.

9 Conclusion

Prism is an LSM-based range index designed for partly coherent CXL shared memory. Prism is based on the observation that the large updatable surface area of in-memory indexes is the root cause of poor performance under partial coherence, and that LSMs, originally designed for disk, naturally confine in-place updates to a small region that fits within the SCR. Prism further introduces a bounded updatable layer to reduce LSM compactions. Our experiments show that Prism outperforms in-memory indexes ported to the partly coherent memory model using software coherence.

Acknowledgments. We thank our shepherd and the SOSP'26 reviewers for their insightful comments. We thank the other DASSL members for their discussions. This material was supported by NSF grants CNS-2340218 and CNS-2339784, IIDAI grants, as well as gifts from NetApp and TigerBeetle.

References

[1] Marcos K. Aguilera, Emmanuel Amaro, Nadav Amit, Erika Hunhoff, Anil Yelam, and Gerd Zellweger. 2023. Memory Disaggregation: Why Now and What are the Challenges. *ACM SIGOPS Operating Systems Review* 57, 1 (2023), 38–46.

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote Regions: A Simple Abstraction for Remote Memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA.

[3] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J. Tsotras, Rares Vernica, Jian Wen, and Till Westmann. 2014. AsterixDB: A Scalable, Open Source BDMS. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1905–1916.

[4] Chloe Alverti, Stratos Psomadakis, Burak Ocalan, Shashwat Jaiswal, Tianyin Xu, and Josep Torrellas. 2025. CXLfork: Fast Remote Fork over CXL Fabrics. In *Proceedings of the 30th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '25)*. Rotterdam, Netherlands.

[5] Emmanuel Amaro, Stephanie Wang, Aurojit Panda, and Marcos K Aguilera. 2023. Logical Memory Pools: Flexible and Local Disaggregated Memory. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks (HotNets '23)*. Cambridge, MA, 25–32.

[6] Christoph Anneser, Lukas Vogel, Ferdinand Gruber, Maximilian Bandle, and Jana Giceva. 2023. Programming Fully Disaggregated Systems. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS '23)*. Providence, RI.

[7] Nikolas Askitis and Ranjan Sinha. 2007. HAT-Trie: A Cache-Conscious Trie-Based Data Structure For Strings. In *Proceedings of the 30th Australasian Conference on Computer Science (ACSC '07)*.

[8] Gal Assa, Moritz Lumme, Lucas Burgi, Michal Friedman, and Ori Lahav. 2026. A Programming Model for Disaggregated Memory over CXL. In *Proceedings of the 31st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '26)*. Pittsburgh, USA.

[9] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA.

[10] Alexander Baumstark, Marcus Paradies, Kai-Uwe Sattler, Steffen Klabe, and Stephan Baumann. 2024. So Far and yet so Near - Accelerating Distributed Joins with CXL. In *Proceedings of the 20th International Workshop on Data Management on New Hardware (DaMoN '24)*. Santiago, Chile.

[11] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. 2015. An Introduction to Bε-Trees and Write-Optimization. *;login: The USENIX Magazine* 40, 5 (2015), 22–28.

[12] Daniel S. Berger. 2024. Realistic Expectations for CXL Memory Pools. Talk presented at the 2nd Workshop on Disruptive Memory Systems (DIMES '24).

[13] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. 2020. Hailstorm: Disaggregated Compute and Storage for Distributed LSM-Based Databases. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Lausanne, Switzerland.

[14] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data (SIGMOD '18)*. Houston, TX.

[15] Gerth Stølting Brodal and Rolf Fagerberg. 2003. Lower Bounds for External Memory Dictionaries. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '03)*. Baltimore, MD, 546–554.

[16] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient Distributed Memory Management with RDMA and Caching. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1604–1617.

[17] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. Seattle, WA, 205–218.

[18] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)*. Virtual Event.

[19] Lei Chen, Shi Liu, Chenxi Wang, Haoran Ma, Yifan Qiao, Zhe Wang, Chenggang Wu, Youyou Lu, Xiaobing Feng, Huimin Cui, Shan Lu, and Harry Xu. 2024. A Tale of Two Paths: Toward a Hybrid Data Plane for Efficient Far-Memory Applications. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '24)*. Santa Clara, CA.

[20] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Lausanne, Switzerland.

[21] Yannis Chronis, Anastasia Ailamaki, Lawrence Benson, Helena Caminal, Jana Giceva, David A. Patterson, Eric Sedlar, and Lisa Wu Wills. 2025. Databases in the Era of Memory-Centric Computing. In *Proceedings of the 15th Conference on Innovative Data Systems Research (CIDR '25)*. Amsterdam, Netherlands.

[22] Douglas Comer. 1979. The Ubiquitous B-Tree. *Comput. Surveys* 11, 2 (1979), 121–137.

[23] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard P. Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. Online, 49–63.

[24] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '10)*. Indianapolis, IN.

[25] CXL Consortium. 2022. Compute Express Link (CXL) Specification, Revision 3.0.

[26] CXL Consortium. 2023. Compute Express Link (CXL) Specification, Revision 3.1.

[27] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI '20)*. Banff, Canada.

[28] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-value Store. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data (SIGMOD '17)*. Chicago, IL.

[29] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-time Trade-offs for LSM-tree Based Key-value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data (SIGMOD '18)*. Houston, TX, 505–520.

[30] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*. Stevenson, WA, 205–220.

[31] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *Proceedings of the 8th Conference on Innovative Data Systems Research (CIDR '17)*. Chaminade, CA.

[32] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*. Seattle, WA, 401–414.

[33] Facebook. 2024. RocksDB. https://rocksdb.org/.

[34] Facebook. 2024. RocksDB Subcompaction. https://github.com/facebook/rocksdb/wiki/Subcompaction.

[35] Keir Fraser. 2004. *Practical Lock-Freedom.* Technical Report UCAM-CL-TR-579. University of Cambridge, Computer Laboratory.

[36] Lars George. 2011. *HBase – The Definitive Guide: Random Access to Your Planet-Size Data.* O'Reilly Media.

[37] Sanjay Ghemawat, Jeff Dean, Chris Mumford, David Grogan, and Victor Costan. 2011. LevelDB. https://github.com/google/leveldb.

[38] Christina Giannoula, Kailong Huang, Jonathan Tang, Nectarios Koziris, Georgios I. Goumas, Zeshan Chishti, and Nandita Vijaykumar. 2023. DaeMon: Architectural Support for Efficient Data Movement in Fully Disaggregated Systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 7, 1 (2023).

[39] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2015. Scaling Concurrent Log-Structured Data Stores. In *Proceedings of the EuroSys Conference (EuroSys '15)*. Bordeaux, France, 1–14.

[40] Donghyun Gouk, Miryeong Kwon, Hanyeoreum Bae, Sangwon Lee, and Myoungsoo Jung. 2023. Memory Pooling With CXL. *IEEE Micro* 43, 2 (2023), 48–57.

[41] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA.

[42] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI '17)*. Boston, MA.

[43] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. Clio: A Hardware-Software Co-Designed Disaggregated Memory System. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. Lausanne, Switzerland.

[44] Steffen Heinz, Justin Zobel, and Hugh E. Williams. 2002. Burst Tries: A Fast, Efficient Data Structure for String Keys. *ACM Transactions on Information Systems* 20, 2 (2002), 192–223.

[45] Jaewan Hong, Marcos K. Aguilera, Emmanuel Amaro, Vincent Liu, Aurojit Panda, and Ion Stoica. 2025. The Dawn of Disaggregation and the Coherence Conundrum: A Call for Federated Coherence. *arXiv preprint arXiv:2504.16324* (2025).

[46] Jiyu Hu, Seokjoo Cho, Landon Johnson, Kiran Hombal, Shreesha G Bhat, Marcos K Aguilera, Ramnatthan Alagappan, and Aishwarya Ganesan. 2026. MEGALON: Efficient Data Sharing for Partly Coherent CXL Memory. In *Proceedings of the 20th USENIX Conference on Operating Systems Design and Implementation (OSDI '26)*. Seattle, WA.

[47] Zhisheng Hu, Pengfei Zuo, Yizou Chen, Chao Wang, Junliang Hu, and Ming-Chang Yang. 2024. Aceso: Achieving Efficient Fault Tolerance in Memory-Disaggregated Key-Value Stores. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP '24)*. Austin, Texas.

[48] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-based HTAP Database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[49] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An Optimized Storage Engine for Large-scale E-commerce Transaction Processing. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data (SIGMOD '19)*. Amsterdam, Netherlands, 651–665.

[50] Haoyu Huang and Shahram Ghandeharizadeh. 2021. Nova-LSM: A Distributed, Component-Based LSM-Tree Key-Value Store. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data (SIGMOD '21)*. Virtual.

[51] Yibo Huang, Haowei Chen, Newton Ni, Yan Sun, Vijay Chidambaram, Dixin Tang, and Emmett Witchel. 2025. Tigon: A Distributed Database for a CXL Pod. In *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI '25)*. Boston, MA.

[52] Yibo Huang, Newton Ni, Vijay Chidambaram, Emmett Witchel, and Dixin Tang. 2025. Pasha: An Efficient, Scalable Database Architecture for CXL Pods. In *Proceedings of the 15th Conference on Innovative Data Systems Research (CIDR '25)*. Amsterdam, Netherlands.

[53] Sunita Jain, Nagaradhesh Yeleswarapu, Hasan Al Maruf, and Rita Gupta. 2024. Memory Sharing with CXL: Hardware and Software Design Approaches. *arXiv preprint arXiv:2404.03245* (2024).

[54] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2015. BetrFS: A Right-Optimized Write-Optimized File System. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*. Santa Clara, CA.

[55] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-Value Services. In *Proceedings of the 2014 ACM SIGCOMM Conference*. Chicago, IL.

[56] Konstantinos Kanellis, Badrish Chandramouli, Ted Hart, and Shivaram Venkataraman. 2025. From FASTER to F2: Evolving Concurrent Key-Value Store Designs for Large Skewed Workloads. *Proceedings of the VLDB Endowment* 18, 12 (2025), 4910–4923.

[57] Konstantinos Kanellis, Sujay Yadalam, Hayden Coffey, Shivaram Venkataraman, and Michael Swift. 2026. From Good to Great: Parameter Tuning in Memory Tiering Systems. *IEEE Trans. Comput.* (2026).

[58] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA.

[59] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. Indianapolis, IN.

[60] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems*

Review 44, 2 (2010), 35–40.

[61] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*. Santa Clara, CA.

[62] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. MIND: In-Network Memory Management for Disaggregated Data Centers. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP '21)*. Virtual.

[63] Philip L. Lehman and S. Bing Yao. 1981. Efficient Locking for Concurrent Operations on B-Trees. *ACM Transactions on Database Systems* 6, 4 (1981), 650–670.

[64] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Proceedings of the 29th International Conference on Data Engineering (ICDE '13)*. Brisbane, Australia.

[65] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of Practical Synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN '16)*. San Francisco, CA.

[66] Baptiste Lepers and Willy Zwaenepoel. 2023. Johnny Cache: the End of DRAM Cache Conflicts (in Tiered Main Memory Systems). In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI '23)*. Boston, MA.

[67] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '23)*. Vancouver, BC, Canada.

[68] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. ROLEX: A Scalable RDMA-oriented Learned Key-Value Store for Disaggregated Memory Systems. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST '23)*. Santa Clara, CA.

[69] Yi Liu, Minghao Xie, Shouqian Shi, Yuanchao Xu, Heiner Litz, and Chen Qian. 2024. Outback: Fast and Communication-efficient Index for Key-Value Store on Disaggregated Memory. *Proceedings of the VLDB Endowment* 18, 2 (2024), 335–348.

[70] Jiaheng Lu, Yiwen Zhang, Hasan Al Maruf, Minseo Park, Yunxuan Tang, Fan Lai, and Mosharaf Chowdhury. 2024. Mercury: QoS-Aware Tiered Memory System. *arXiv preprint arXiv:2412.08938* (2024).

[71] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*. Santa Clara, CA.

[72] Chen Luo and Michael J. Carey. 2020. LSM-based Storage Techniques: A Survey. *The VLDB Journal* 29, 1 (2020), 393–418.

[73] Xuchuan Luo, Jiacheng Shen, Pengfei Zuo, Xin Wang, Michael R. Lyu, and Yangfan Zhou. 2024. CHIME: A Cache-Efficient and High-Performance Hybrid Index on Disaggregated Memory. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP '24)*. Austin, Texas.

[74] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. 2021. ROART: Range-query Optimized Persistent ART. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)*. Virtual Event.

[75] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '23)*. Vancouver, BC, Canada, 742–755.

[76] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3217–3230.

[77] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC '13)*. San Jose, CA.

[78] Newton Ni, Yan Sun, Zhiting Zhu, and Emmett Witchel. 2026. Cxlalloc: Safe and Efficient Memory Allocation for a CXL Pod. In *Proceedings of the 31st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '26)*. Pittsburgh, USA.

[79] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385.

[80] Yifan Qiao, Xubin Chen, Ning Zheng, Jiangpeng Li, Yang Liu, and Tong Zhang. 2022. Closing the B+-tree vs. LSM-tree Write Amplification Gap on Modern Storage Hardware with Built-in Transparent Compression. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST '22)*. Santa Clara, CA.

[81] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*. Shanghai, China.

[82] Jun Rao and Kenneth A. Ross. 2000. Making B+-Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*. Dallas, TX, 475–486.

[83] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP '21)*. Virtual.

[84] Feng Ren, Mingxing Zhang, Kang Chen, Huaxia Xia, Zuoning Chen, and Yongwei Wu. 2024. Scaling Up Memory Disaggregated Applications with SMART. In *Proceedings of the 29th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)*. San Diego, CA.

[85] Ohad Rodeh. 2008. B-trees, Shadowing, and Clones. *ACM Transactions on Storage* 3, 4, Article 2 (2008).

[86] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI '20)*. Banff, Canada, 315–332.

[87] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2021. Constructing and Analyzing the LSM Compaction Design Space. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2216–2229.

[88] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. Scottsdale, AZ.

[89] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. 2011. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors. *Proceedings of the VLDB Endowment* 4, 11 (2011), 795–806.

[90] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*. Carlsbad,

CA, 69–87.

[91] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Yuxin Su, Jiazhen Gu, Hao Feng, Yangfan Zhou, and Michael R. Lyu. 2023. Ditto: An Elastic and Adaptive Memory-Disaggregated Caching System. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP '23)*. Koblenz, Germany, 675–691.

[92] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. 2023. FUSEE: A Fully Memory-Disaggregated Key-Value Store. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST '23)*. Santa Clara, CA, 81–98.

[93] Kevin Song, Jiacheng Yang, Zixuan Wang, Jishen Zhao, Sihang Liu, and Gennady Pekhimenko. 2025. HybridTier: An Adaptive and Lightweight CXL-Memory Tiering System. In *Proceedings of the 30th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '25)*. Rotterdam, Netherlands.

[94] Yan Sun, Jongyul Kim, Zeduo Yu, Jiyuan Zhang, Siyuan Chai, Michael Jaemin Kim, Hwayong Nam, Jaehyun Park, Eojin Na, Yifan Yuan, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2025. M5: Mastering Page Migration and Memory Management for CXL-based Tiered Memory Systems. In *Proceedings of the 30th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '25)*. Rotterdam, Netherlands.

[95] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23)*. Toronto, Canada.

[96] Bijan Tabatabai, James Christopher Sorenson III, and Michael M. Swift. 2024. FBMM: Making Memory Management Extensible With Filesystems. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC '24)*. Santa Clara, CA.

[97] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Portland, OR, 1493–1509.

[98] TiKV Project. 2020. B-Tree vs LSM-Tree. https://tikv.org/deep-dive/key-value-engine/b-tree-vs-lsm/.

[99] Twitter. 2020. Twitter Cache Trace. https://github.com/twitter/cache-trace.

[100] Jing Wang, Youyou Lu, Qing Wang, Yuhao Zhang, and Jiwu Shu. 2023. Revisiting Secondary Indexing in LSM-based Storage Systems with Persistent Memory. In *Proceedings of the 2023 USENIX Annual Technical Conference (USENIX ATC '23)*. Boston, MA.

[101] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data (SIGMOD '22)*. Philadelphia, PA.

[102] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data (SIGMOD '18)*. Houston, TX.

[103] Fangnuo Wu, Mingkai Dong, Wenjun Cai, Jingsheng Yan, and Haibo Chen. 2025. Guidelines for Building Indexes on Partially Cache-Coherent CXL Shared Memory. In *arXiv preprint arXiv:2511.06460*.

[104] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A Large Scale Analysis of Hundreds of In-memory Cache Clusters at Twitter. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI '20)*. Banff, Canada.

[105] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. 2023. FIFO Queues Are All You Need for Cache Eviction. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP '23)*. Koblenz, Germany, 130–149.

[106] Xiran Yang, Yifei Yu, Chuandong Li, Jianqiang Zeng, Ke Zhou, Diyu Zhou, Xiaolin Wang, Zhenlin Wang, and Yingwei Luo. 2025. Ginkgo: A Learned-Index Enhanced Tiered Memory System. *IEEE Micro* (2025).

[107] Hobin Yoon, Juncheng Yang, Sveinn Fannar Kristjansson, Steinn E. Sigurðarson, Ymir Vigfusson, and Ada Gavrilovska. 2018. Mutant: Balancing Storage Cost and Latency in LSM-Tree Data Stores. In *Proceedings of the 9th ACM Symposium on Cloud Computing (SoCC '18)*. Carlsbad, CA.

[108] Wonsup Yoon, Jisu Ok, Sue Moon, and Youngjin Kwon. 2025. Adios to Busy-Waiting for Microsecond-scale Memory Disaggregation. In *Proceedings of the 20th European Conference on Computer Systems (EuroSys '25)*. Rotterdam, Netherlands.

[109] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. 2023. Partial Failure Resilient Memory Management System for (CXL-based) Distributed Shared Memory. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP '23)*. Koblenz, Germany, 658–674.

[110] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. 2024. Managing Memory Tiers with CXL in Virtualized Environments. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '24)*. Santa Clara, CA.

[111] Zhiting Zhu, Newton Ni, Yibo Huang, Yan Sun, Zhipeng Jia, Nam Sung Kim, and Emmett Witchel. 2024. Lupin: Tolerating Partial Failures in a CXL Pod. In *Proceedings of the 2nd Workshop on Disruptive Memory Systems (DIMES '24)*. Austin, TX, 41–50.

[112] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A Fast and Cost-Efficient Storage Engine Using DRAM, NVMe, and RDMA. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data (SIGMOD '22)*. Philadelphia, PA.

[113] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. 2021. One-Sided RDMA-Conscious Extendible Hashing for Disaggregated Memory. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC '21)*. Virtual, 15–29.