Borges: A Low-Latency Distributed Shared Log on a CXL Memory/SSD Hybrid

Haowei Chen
The University of Texas at Austin

Yiming Xiang
The University of Texas at Austin

Zhipeng Jia*
Anthropic PBC

Yan Sun
UIUC

Nam Sung Kim
UIUC

Emmett Witchel
The University of Texas at Austin

Abstract

Fault-tolerant, distributed shared logs are a useful substrate for distributed applications, but existing designs coordinate shard servers, sequencers, and replicas over the network, so append and replay latency is dominated by message exchange. We present Borges, to our knowledge the first fault-tolerant shared log whose data and ordering state reside in a DRAM/SSD hybrid device accessed with Compute Express Link (CXL). Because current CXL pods lack inter-host cache coherence, Borges assigns every shared metadata word a single writer, atomically publishes each committed prefix as a matching global cut and stream index pair, and tracks each stream with compact byte offsets, so replay stays cheap even when a stream spans shards. We evaluate Borges on Samsung's CMM-H device against CXL ports of Scalog and Boki, using a linearizable lock, a key-value store, and a fault-tolerant workflow with exactly-once semantics. Borges achieves 2.87× lower append p99 latency and 2.16× higher throughput than the Scalog port, and for the Retwis workflow application it lowers p50 latency by 5.7×/22.5× relative to the Scalog/Boki ports.

CCS Concepts: • **Computer systems organization → Distributed architectures**; *Dependable and fault-tolerant systems and networks.*

Keywords: Compute Express Link (CXL), distributed shared logs, distributed storage, persistent storage

ACM Reference Format:
Haowei Chen, Yiming Xiang, Zhipeng Jia, Yan Sun, Nam Sung Kim, and Emmett Witchel. 2026. Borges: A Low-Latency Distributed Shared Log on a CXL Memory/SSD Hybrid. In *Proceedings of the 32nd ACM Symposium on Operating Systems Principles (SOSP '26), September 29–October 2, 2026, Prague, Czechia.* ACM, New York, NY, USA, 18 pages.

1 Introduction

Shared logs [7, 8, 19, 29, 41, 64] offer a compelling abstraction for distributed systems: a fault-tolerant, totally ordered, append-only history that all participants can read as a prefix. This single abstraction unifies replication, coordination, and recovery, which is why shared logs have become a useful substrate for replicated data structures, metadata services, and workflow systems [7, 9, 29].

However, many scalable shared logs are built around networked sequencing and replication, and therefore optimize for aggregate throughput rather than per-operation latency, by sharding the log, batching appends, and pipelining sequencing and replication across the network. But those same mechanisms add queuing and multiple communication steps to the critical path. Consequently, even systems that sustain more than one million appends/s still report millisecond-class append latency (1–2 ms) [8, 19, 29]. That tradeoff works well for high-fanout ingestion, but it is a poor fit for latency-sensitive operations whose throughput is directly limited by commit latency, such as locks.

The append path is only part of the problem. Many shared-log applications, organize state transitions to logical *streams* which define the histories of individual objects, and replay a stream's committed prefix to reconstruct the state. Networked designs therefore naturally face a difficult placement tradeoff. Binding a stream to one shard keeps replay local but creates hotspots under skew [8, 9, 19, 64]. Distributing a stream across shards balances load, but makes replay expensive because each cross-shard read requires a remote access [29, 41]. This leaves networked logs with no good design point for skewed, latency-sensitive workloads such as locks, coordination objects, and hot-key stateful services.

Compute Express Link (CXL) creates an opportunity to revisit this design space. CXL exposes byte-addressable shared memory across a small pod of hosts with sub-microsecond access latency [21, 37, 38]. Paired with a hybrid device such as Samsung's CMM-H, whose DRAM cache is backed by persistent NAND, this interface can host both shared metadata and durable log state [54]. Our directly attached CMM-H measures 583ns access latency, rather than RDMA's microsecond-scale regime (§2.2). This is attractive to a shared log in two ways. First, coordination can move from message exchanges to shared-memory metadata: instead of exchanging progress information over the network, a sequencer can read it directly. Second, cross-shard replay becomes a shared-storage operation rather than a remote-storage operation: any shard server can directly read another shard's committed records

SOSP '26, Prague, Czechia
2026.

*Work done while at Google LLC.

from shared memory. In principle, CXL could remove server-side message overhead from both ordering and replay.

But a CXL-based shared log is not a straightforward port of a networked design. Current multi-host CXL pod prototypes do not provide inter-host hardware cache coherence [53, 80], so synchronization primitives that assume coherent caches (e.g., POSIX locks and futexes) do not work at all. CAS-based alternatives can be implemented in software, but each contender requires an explicit cache flush before the read, leading to repeated flush-retry cycles under contention. In our Scalog-CXL baseline, acquiring internal metadata locks alone contributes 15.2%, 15.9%, and 15.7% of replication, sequencing, and commit latency, respectively, and 24.1% of end-to-end p99 latency (§4.2).

In addition, shared visibility creates a publication problem. The global commit frontier, called the *global cut*, describes the committed prefix, while the index of log entries identifies which log entries are visible during replay. Readers must observe consistent versions of both. Otherwise, replay may omit committed entries, return them out of order, or include entries beyond the selected prefix. The key challenge is therefore not merely to use a faster interconnect, but to redesign synchronization and publication policies for CXL's shared-memory model.

We present Borges, a distributed shared log designed for a CXL DRAM/SSD hybrid and the first fault-tolerant shared log whose data and ordering state reside in such a device. Figure 1 gives a high-level view. Borges shards the log across independent, replicated CXL devices, while streams may span shards. To reduce synchronization that requires coherence, Borges uses *single-writer metadata*: each shared word has a unique writer and is accessed using simple loads and stores. To publish a consistent prefix, Borges uses a *dual-index* mechanism: the sequencer constructs the next stream index in a writer-only copy not visible to readers, and atomically publishes it with the matching global cut, so readers see a consistent committed version. Finally, Borges exploits CXL's byte addressability to organize each shard-local stream portion as linked, fixed-size segments and track its committed boundary with lightweight byte offsets, making replay efficient even when a stream spans multiple shards.

Borges targets a small, pod-scale, strongly consistent core where latency matters more than scaling to hundreds of machines, Within that scope, its design changes the usual tradeoffs of shared logs. We evaluate Borges on an emulated CXL pod with a real Samsung CMM-H device (§4). For log append, Borges achieves 2.87× lower p99 latency and 2.16× higher throughput than Scalog-CXL. A linearizable lock built on Borges has 3.1× lower p50 latency than vScalog-CXL and 13.7× lower p50 latency than Boki-CXL; under high skew, Borges' throughput drops only 2.1%, compared with nearly 30% for vScalog-CXL. At the application level, Borges reduces Retwis p50 latency by 5.7× relative to Scalog-CXL (22.5× relative to Boki-CXL) and improves throughput by 19.4×

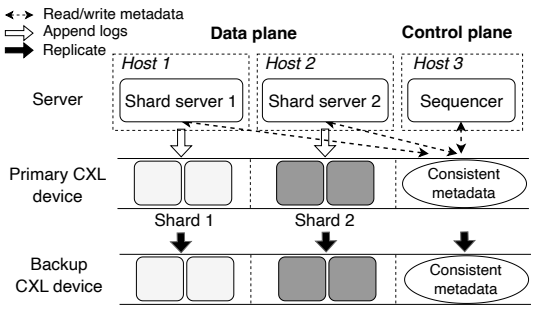

Figure 1. Conceptual deployment of a CXL-based shared log. Within each replica, logical shards 1 and 2 are co-located on one device. The primary and backup replicas occupy independent devices. Clients (not shown) contact shard servers over the network, while shard servers and the sequencer coordinate through CXL-resident metadata.

over Scalog-CXL. These results show that CXL does not merely speed up an existing shared-log design; it opens a new design point for low-latency, skew-tolerant shared logs.

This paper makes three contributions.

- We design and evaluate the first fault-tolerant shared log with data and ordering state resident in CXL DRAM/SSD hybrid storage.
- We introduce a CXL-native log design that replaces coherence-dependent synchronization with lock-free, single-writer metadata and uses dual-index publication to expose only committed, globally ordered state while enabling efficient cross-shard replay.
- We evaluate Borges on an emulated CXL pod with a real Samsung CMM-H device across log append, locks, skewed workloads, and fault-tolerant workflows, showing substantial latency and throughput gains over Scalog and Boki.

2 Background and Motivation

Shared-log systems typically follow one of two architectures: *order-first* or *replication-first*.

In *order-first* logs such as CORFU, Delos, and vCorfu [7, 8, 64], the global ordering path first fixes each record's position in the logical log. The storage layer then places and replicates the record according to its placement and replication protocols while preserving that order.

In *replication-first* logs such as Scalog, Boki, and LazyLog [19, 29, 41], the data plane first makes a record durable before assigning it a global position. Shards periodically summarize their replication progress in *local cuts*. The ordering layer aggregates these local cuts, selects only records known to be durable, and publishes a totally ordered sequence of *global cuts*. All shards then apply the same deterministic merge rule to this sequence, assigning every selected record a consistent global position.

Borges adopts replication-first for the reasons identified by Scalog [19]: ordering only already-replicated records avoids holes when a client fails after reserving a global position,

and removes per-record sequencer interaction from the append path. This structure preserves shard-parallel ingestion while allowing CXL shared memory to reduce the remaining coordination and replay costs.

2.1 Why Existing Shared Logs Struggle with Latency-Sensitive Workloads

The first bottleneck is the write path. Existing shared logs rely on message-based coordination, so an append pays not only for storage and replication, but also for communication among shard replicas and the sequencer. In Scalog [19], a primary shard determines local order, forwards records to a backup, and both replicas report progress to the sequencer. Boki [29] restructures this pipeline but still relies on message-based sequencing and notification. Even in our optimized implementations of Scalog and Boki, server-side communication accounts for 51.0% and 40.1% of end-to-end append latency, respectively, as shown in Figure 2. Both systems use batching to amortize that cost, but batching adds another 21.7% and 21.4% to the end-to-end latency, respectively. This tradeoff is effective for throughput-oriented ingestion, but it is poorly matched to operations whose rate is limited by confirmed commit latency.

The second bottleneck lies on the replay path. Stateful applications reconstruct an object's state by replaying the committed records in its stream, but existing systems face an unpleasant placement tradeoff. *Single-shard placement* [8, 9, 19] keeps replay local to avoid cross-shard retrieval, but hot streams concentrate load on their assigned shards. In our evaluation, vScalog-CXL's throughput drops by nearly 30% under high skew (§ 4.5). *Cross-shard placement* [29, 41] distributes a stream's records across shards to spread load, but replay then requires remote fetches. On our platform, a Boki-CXL remote record fetch costs 673μs at p50 and exceeds 2.5 ms at p99 (§ 4.4). These placement choices therefore offer either replay locality or tolerance to skew, offering no good option for skewed, latency-sensitive workloads.

These bottlenecks matter because many uses of a shared log are latency-sensitive rather than throughput-oriented. Locks, coordination objects, metadata services, and exactly-once workflow steps depend on both fast confirmation of order and fast replay of recently committed state.

2.2 Why CXL Creates an Opportunity

Compute Express Link (CXL) exposes byte-addressable shared memory across a small pod of hosts with sub-microsecond access latency [6, 18, 34, 43]. For shared logs, this changes the design space by making two things possible that network-based architectures cannot offer. First, coordination metadata can live in shared memory rather than travel through message queues, so a sequencer could read shard progress directly instead of waiting for RPCs. Second, every shard's committed records are directly readable by every host, so cross-shard replay need not be a remote operation. These

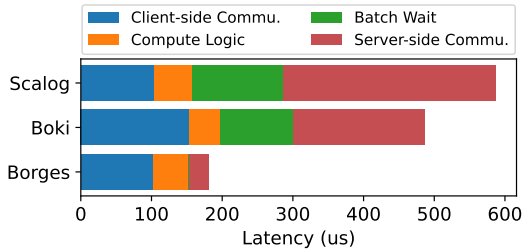

Figure 2. Append latency breakdown for Borges and representative network-based baselines. Server-side coordination dominates even when all systems use a single sequencer and the client path is unchanged.

properties create the *opportunity* for low-latency appends and skew-tolerant replay, but realizing that opportunity requires new software designs, because CXL shared memory introduces its own coordination and correctness challenges.

We evaluate this opportunity using Samsung's CMM-H [54, 72], a CXL Type-3 DRAM/SSD hybrid that exposes a persistent, byte-addressable load/store interface. The device presents a 48GB DRAM cache (8-way associative with LRU replacement) backed by NAND, and appears to software as persistent memory on a CPU-less NUMA node. Durability is provided by a two-stage mechanism: platform eADR first flushes dirty CPU cache lines to the device on power loss, and the device then performs a global persistent flush to move buffered data from DRAM to NAND [32, 71].

We measured the CMM-H using Intel's Memory Latency Checker with a 3:1 read/write ratio on sequential access which matches the dominant data path in Borges. With a 1GB working set, the device delivers 583ns latency, stays below 590ns for working sets up to 32GB, and rises only modestly to 621–638ns for 64–512GB. Its bandwidth reaches 9.3GB/s with 32 cores on a 16GB working set, drops to 7.0GB/s at 32GB, and then stabilizes around 4.9–5.4GB/s for 64–256GB. That flat latency profile, even beyond the 48GB DRAM capacity, combined with durability required for fault tolerance, makes the CMM-H an ideal substrate for a shared log.

To exploit the opportunity, Borges targets a CXL pod of roughly 8–16 backend hosts without involving any CXL switches, serving applications with low latency [34, 53, 78]. The pod-scale setting is practical. Production systems already intentionally keep the strongly consistent core small because every participant increases commit latency. Coordination services such as Chubby [12] and ZooKeeper [5, 27] typically use small replicated cores of 3–5 nodes, while etcd recommends no more than seven members [58]. CXL databases such as Tigon and Pasha [25, 26] also target a pod scale of hosts to provide the best performance.

2.3 Why Naive CXL Designs Still Fail

Simply porting a networked log onto CXL does not capture these benefits. CXL provides shared storage, but without the hardware cache coherence that conventional shared-memory

programming assumes, so both the synchronization strategy and the correctness model must be redesigned.

Challenge 1: No inter-host cache coherence. Current CXL pod prototypes share a device, but not a coherent cache hierarchy. Although CXL 3.2 specifies inter-host cache coherence [16], existing systems such as Octopus and Niagara 2.0 do not implement it ([53, 80]). This means one host can hold a stale cached copy of a shared line until software explicitly invalidates or bypasses it. POSIX locks, futexes, and similar shared-memory synchronization primitives assume hardware coherence and therefore do not work across hosts.

A reasonable naive assumption is to preserve the multi-writer coordination structure of prior shared-memory designs and re-express it in software over shared CXL metadata. This typically means CAS-based locking [25, 46], together with explicit cross-host visibility management. Without coherence, locking creates a performance problem. Each lock acquisition or pointer update first requires a fresh read of the current value from shared memory; each failed CAS repeats the flush-read-modify-retry cycle; and under contention the shared metadata line itself becomes the hotspot. In our Scalog-CXL baseline, lock acquisition alone accounts for 24.1% of end-to-end p99 append latency (§ 4.2).

Takeaway 1. A CXL-native shared log should keep metadata and data ownership simple and avoid repeated read-modify-writes by multiple hosts on the fast path.

Challenge 2: Shared visibility creates a publication problem. CXL makes every shard's storage directly readable by every host. This is precisely what makes cross-shard replay attractive, but it also creates a correctness challenge that network-isolated storage largely avoids. In a replication-first log, physical placement and logical commitment are decoupled in time: a record may already be present in a shard's log region before it has been replicated everywhere, and before the sequencer has fixed its position in the globally committed prefix. In a networked design, this intermediate state is naturally hidden because other hosts cannot directly read that shard's storage. But this changes in a CXL design.

Exposing this intermediate state makes consistency between the reader-visible *global-cut sequence* and *index* essential. The cut sequence identifies the committed prefix and determines its global order, while the global index controls the visibility of committed records. If the index advances beyond its matching cut, a reader may observe uncommitted or not-yet-ordered data. Conversely, if a cut becomes visible with a partially updated index. Replay may then omit records that the cut has already committed, or expose a later record from one shard while missing an earlier record from another, violating the global order. Two invariants must therefore hold: (1) only records covered by the visible global cut are reachable through the index, and (2) every visible cut is paired with a fully constructed index that describes exactly the same committed prefix. The central publication problem is therefore how to expose the global cut and index as a consistent pair.

System	Append/order path	Read/replay path
Borges	Lock-free ordering	Per-stream global dual index
Scalog	Paxos and aggregators	Per-record local index
Boki	Metalog coordination	Per-record global index
LazyLog	Lazy ordering	Readable after ordered
SpecLog	Speculative ordering	Later confirmation

Table 1. Technical comparison with shared-log systems.

Takeaway 2. A CXL-native shared log needs an explicit publication mechanism that exposes each global cut only with a matching index, thereby revealing a consistent, globally ordered committed prefix. And because the whole point of using CXL is to make cross-shard replay cheap, that mechanism must preserve efficient replay even when a stream spans shards.

2.4 Positioning Relative to Prior Work

Borges occupies a design point that prior systems do not reach. Table 1 compares Borges with the most relevant shared logs. Scalog and Boki retain message-based ordering, notification, or remote replay on their critical paths. Simply attaching their storage to CXL would not resolve the synchronization and publication challenges identified above.

Emerging shared logs, e.g., SpecLog [11] and LazyLog [41] reduce **perceived** latency by speculating on order or deferring it, which is valuable for workloads that can tolerate temporary uncertainty. But strictly serialized workloads such as locks and exactly-once workflows cannot proceed until order is actually fixed, and they still require low-latency replay of committed state. Borges instead targets low **actual** commit and replay latency. Its novelty is not merely using CXL as a faster transport, but redesigning the shared log around non-coherent shared persistent memory.

3 Design

Borges addresses the two core challenges from §2.3 with three coupled ideas: a *single-writer metadata discipline*, *dual-index publication*, and *byte-offset indexing*. The first removes coherence-sensitive multi-writer coordination from the fast path by assigning every shared word a unique owner. The second separates private commit construction from public visibility, so readers observe only fully committed, globally ordered state. The third keeps the reader-visible index compact by publishing stream-level byte boundaries, rather than per-record metadata, so cross-shard replay remains efficient.

The rest of this section explains how these three ideas fit together to make Borges safe and efficient on CXL memory.

3.1 Architecture

Overview. Figure 3 shows the overall architecture. Borges follows the replication-first decomposition of systems such

as Scalog and Boki, with **shard servers** in the data plane and a single **sequencer** in the control plane. Servers in the pod communicate only through replicated CXL storage, removing server-side network messages from the critical path. Clients remain outside the pod and contact shard servers over the network. Shared CXL storage substrate is used for both communication and durability.

Streams, shards, and segments. Logically, the log is organized as *streams*, each capturing the history of one logical object such as a key or workflow instance. Replay is via streams rather than scanning the whole log. For parallelism, Borges partitions the log into *shards*, each owned by one shard server, but a stream may span multiple shards. This is a deliberate design point: pinning a stream to one shard would simplify replay, but it would also recreate hotspots under skew (§ 4.5). Borges instead keeps placement flexible and makes cross-shard replay efficient.

Within each shard, a stream is stored as a linked list of fixed-size *segments* (2 KB by default). Rather than publish a reader-visible pointer for every record, Borges tracks only stream boundaries. The global index therefore stores one logical byte offset per stream in each shard, which keeps the published metadata compact and makes publication and replay cheaper than a per-record global index.

Metadata ownership. Borges splits CXL-resident metadata into *local* and *global* regions to enforce single-writer coordination. *Local metadata* is written by one shard and read by the sequencer: a *local cut* reports replicated progress to each device, and *stream deltas* report how many bytes each stream advanced in local order. *Global metadata* is written only by the sequencer and read by all shards: the *global cut* identifies the committed prefix across shards, and the *global index* records each stream's committed byte boundary in each shard. This split gives a clean publication boundary: shards report progress privately, the sequencer aggregates it privately, and then publishes new global state to shards.

Each shard server has two responsibilities. It accepts client appends, assigns local sequence numbers, replicates records to CXL devices, and reports local progress to the sequencer. The sequencer never moves log data directly; it operates only on metadata. This keeps data movement parallel in the shards while centralizing the only global ordering decision. Borges uses a single sequencer because, at pod scale, CXL removes the network bottleneck that motivated sequencer scaling in prior work [19]; we return to this point in §3.5.

3.2 Append Path: Replicate, Sequence, Publish

Figure 4 shows the critical path of a log append. Borges structures that path as a sequence of ownership-preserving stages. Every CXL-resident data record and metadata object is cache-line aligned and padded to a whole number of cache lines. Independently updated objects therefore never share a cache line, eliminating false sharing. More importantly, each shared word has exactly one writer. This replaces coherence-sensitive read-modify-write coordination with lock-free load/store communication, following the discipline motivated in §2.3.

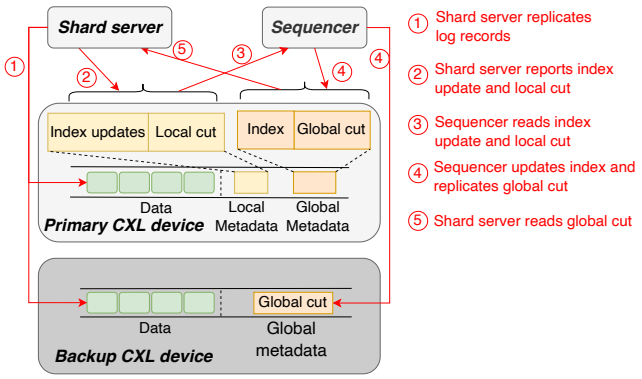

Figure 3. Overview of the Borges architecture (§ 3.1) and the end-to-end append workflow (§ 3.2).

Replication. When a client issues `logAppend()`, the target shard server assigns the next local sequence number and appends the record to the stream's current segment. Two replication threads then copy that record to the primary and backup CXL devices in parallel. The append becomes durable once both copies have advanced past that record.

In parallel, an indexer thread emits a compact stream delta (index update) to a ring buffer shared with the sequencer. Each delta contains a stream identifier and the number of newly appended bytes. This is a key design choice: Borges does *not* track per-record index. Instead, shard-local appends produce private stream deltas that the sequencer later folds into the per-stream global index. This keeps the published shared index compact, and the tiny deltas overlap almost entirely with replication latency.

Sequencing. The sequencer turns shard-local progress into a safe global prefix. Each replication thread writes its own local cut (packed in a 64-bit atomic word), indicating how many records it has copied to its device. The sequencer reads the local cuts for a shard and takes their minimum. That minimum is the number of records that are fully replicated and therefore eligible to enter the committed prefix. This is essential for correctness: a higher local progress value may describe records already present on one device but not the other. Using those records to advance the published index would expose data that could be lost on failure. The sequencer waits for the consumer position of the stream-delta ring to catch up and reads deltas only up to the minimum replicated progress and ignores later deltas for now. In the example of Figure 4, record r3 has reached the primary but not the backup, so its stream delta is not consumed.

Sequencing is also where single-writer ownership pays off: the sequencer reads local cuts and stream deltas from

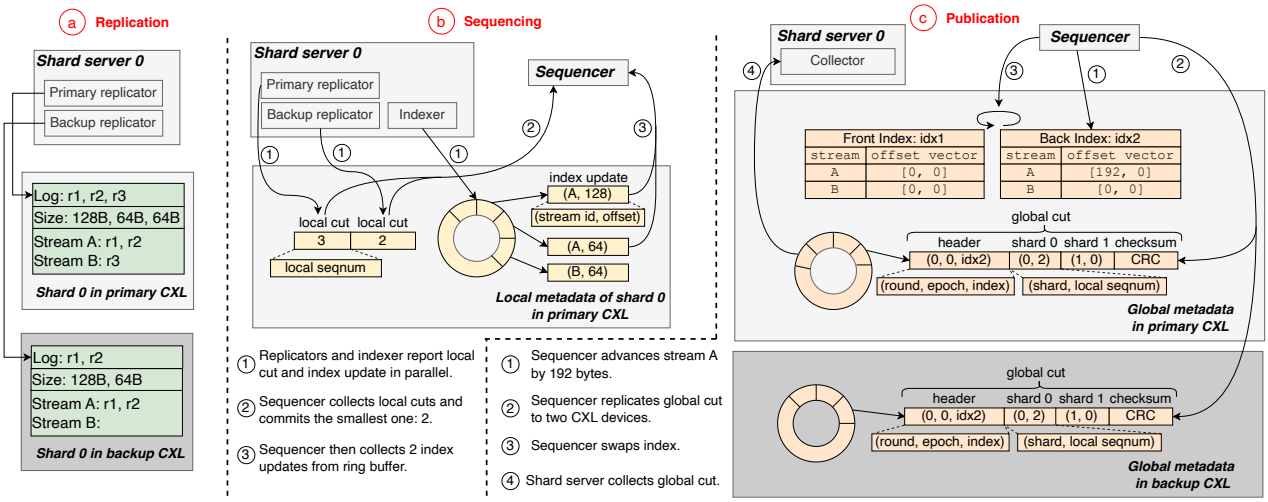

Figure 4. Borges processes a log append in three stages: ⓐ replication, ⓑ sequencing, and ⓒ publication. Each ring buffer records producer and consumer positions (64-bit atomic words). The sequencer does not advance the producer position of the global-cut ring until the corresponding index has been published. Consumer of ring-buffer always processes data faster than producer (table 3).

shard-local metadata, without blocking any writers on the fast path.

Publication. Replication makes data durable, and sequencing identifies the safe prefix, but neither is enough by itself. The remaining problem is publication: how does the system make the newly safe prefix visible to readers without exposing partially updated state?

A natural design would update one shared index in place. In a CXL setting, that is both unsafe, because readers could observe a partially updated index, and expensive, because the sequencer and readers would repeatedly touch the same shared cache lines.

Borges instead uses *dual-index publication*. It maintains two copies of the global index: a *front index*, which is the only index readers may consult, and a *back index*, which the sequencer updates privately. Each index stores the ring-buffer cursors describing which stream deltas have already been consumed. After sequencing identifies the next safe prefix, the sequencer applies those deltas to the back index. Once the back index fully materializes the new committed prefix, the sequencer constructs a global cut. Its header identifies the publication round and reconfiguration epoch (§ 3.6) and names the matching back-index pointer, while its payload records the global progress of each shard. On each CXL device, the sequencer persists the header and payload before atomically installing the checksum as the final validity marker. Borges only replicates the global cut which works as the ground truth for recovery and is sufficient to reconstruct other metadata. After both cut copies are complete, it issues a persistence fence.

This ensures that the index and cut agree before publication. Only then does the sequencer publish the cut/index pair. It first swaps the reader-visible index pointer to the newly prepared index and then advances the producer position of the global-cut ring. This ordering gives Borges a crucial invariant: *every visible global cut names a fully constructed index that materializes exactly the same committed prefix.* Consequently, readers never observe a cut/index mismatch.

Table 2 shows the dual-index publication protocol, which follows an RCU-style publish-and-wait pattern. Its correctness rests on three invariants: readers proceed only if they observe the same `index_ptr` before and after announcing it in `active[i]`; the new index is built privately, so readers see either the old complete index or the new one, never a partial update; and the writer does not retire the old index until no reader still advertises `old_index_ptr`. Unlike CAS-based reader tracking, which would create a multi-writer hotspot and incur repeated cache flushes and fences on non-coherent CXL, this RCU-style protocol uses only non-temporal loads and stores on single-writer slots. Its wait is normally short because readers advertise the old index only while copying compact stream boundaries, not while scanning log data. Borges is forward-compatible with future inter-host hardware coherence: single-writer ownership and dual-index publication remain correct, while an implementation can use coherent loads and stores and omit explicit cache management. If a reader fails while still advertising an old pointer, lease-based failure handling provides the liveness escape hatch (§ 3.7).

Linearizability. The publication protocol also gives Borges a clean linearization point. A write becomes visible only

when the sequencer publishes the matching cut/index pair, and clients are acknowledged only after that publication. A read replays exactly the committed prefix named by the current published front index and global cut sequence, so it includes all previously completed writes and excludes all unpublished ones. Because Borges never exposes a cut without its matching index, or an index without its matching cut, playback reconstructs a consistent linearizable prefix.

3.3 Replay Path: Reconstructing a State via a Stream

Borges deliberately separates *physical placement* from *logical replay*. A stream may span multiple shards to absorb skew, but replay still proceeds against a compact per-stream index rather than per-record global metadata. The key observation is that replay needs only two inputs: (1) the committed byte boundary of the stream in each shard, and (2) the sequence of global cuts that defines global order. Everything else can be reconstructed locally from the log data itself.

Why byte-offset indexing. A natural alternative is a per-record global index that stores a reader-visible metadata item for every committed record. That would simplify replay lookup, but it would also force the sequencer to publish much more shared metadata and force readers to consult shared metadata far more frequently. At the other extreme, binding each stream to one shard keeps the shared index simple, but it recreates the hotspot problem under skew.

Borges chooses a middle point. The global index stores only per-stream byte boundaries for each shard. During append, shard-local stream deltas remain private to the append/sequence pipeline; during publication, the sequencer folds them into a compact reader-visible front index. This keeps publication cheap and replay state-oriented: readers fetch a small amount of shared metadata up front and then do most of their work by scanning log data rather than repeatedly looking up per-record metadata.

On a playback request, the shard server reads the stream's committed byte-boundary vector, with one logical offset per shard, from the front index. Each shard also maintains the head address of that stream's segment chain, and Borges uses Cxlalloc [46] for translation of CXL-resident memory. Replay then scans each shard-local portion from its head to its committed tail. To avoid rebuilding state from scratch, each shard caches a local object view and replays only the suffix beyond the last applied position, similar to prior systems [9, 29].

Deriving global order. Figure 5 shows how replay combines local shard order with the global cuts. Each cut states how many records from each shard belong to the committed prefix at that round. Each shard stores records in local order, with each record carrying a local sequence number and size. Replay therefore becomes a deterministic merge: the shard server walks each shard-local stream up to its committed byte boundary, and uses the cut sequence to decide which records become visible in which global round.

```
# reader i calls beforeRead before reading index
def beforeRead():
    while True:
        index1 = load(index_ptr)
        store index1 INTO active[i]
        index2 = load(index_ptr)
        if index1 == index2:
            return index1
        else:
            store 0 INTO active[i]
            backoff
# reader i calls afterRead after reading index
def afterRead():
    store 0 INTO active[i]
```

```
# Sequencer calls switchIndex to install a new index
def switchIndex(new_index_ptr, old_index_ptr):
    store new_index_ptr INTO index_ptr
    for each reader i:
        while load(active[i]) == old_index_ptr:
            backoff
```

Table 2. Reader-announcement pointer-swap protocol used to publish the front index. Loads and stores are non-temporal.

In the running example, the first cut (0,0), (1,1) says the first committed record comes from shard 1. The next cut (0,2), (1,1) contributes two records from shard 0. The final cut (0,3), (1,2) contributes one additional record from each shard. Within a round, Borges follows the same lexicographic shard ordering used by Scalog and Boki. The result is that any shard can reconstruct the same global order for a cross-shard stream. The segment layout also keeps each shard-local scan mostly sequential, which is a better fit for CXL-attached storage than scattered per-record lookups [9, 29].

3.4 Log Trim in Borges

Trim is expressed in the same stream-oriented vocabulary as the rest of the design. A client issues a trim request that identifies a shard, a stream, and a local sequence number. Each shard marks the corresponding prefix as reclaimable and frees a segment once all of its records have been reclaimed.

Deleting an entire stream is implemented through the publication path rather than through a special side channel. Borges appends a placeholder record whose index delta carries a zero byte offset, an impossible value for a normal append. When the sequencer consumes that delta, it removes the stream from the published index. Actual space reclamation can then proceed lazily on each shard.

3.5 The Single Sequencer

A single sequencer is a deliberate design choice, not a prototype shortcut. Prior systems scaled sequencers mainly to amortize network communication, which made sequencing itself a distributed bottleneck [7, 19]. In Borges, by contrast, the sequencer performs only metadata work over shared CXL storage. Each sequencing round consumes a fixed-size

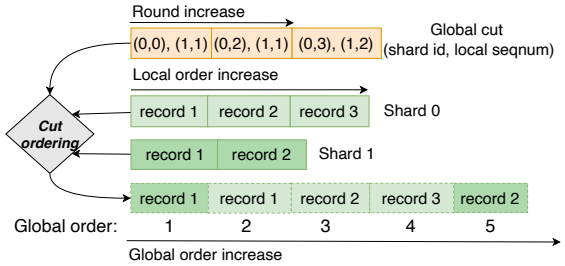

Figure 5. Replay across shards using per-stream byte boundaries and the sequence of global cuts.

metadata summary from every shard, so bursty request arrivals do not increase the sequencer's per-round CPU work. Centralizing that path therefore removes an unnecessary control-plane distribution problem while still preserving pod-scale performance. The sequencer is also effectively stateless because the metadata it consumes and publishes is already materialized in CXL memory. Recovery therefore means reinstating the role, not reconstructing private state or protecting a separate replicated control plane. Our evaluation shows that one sequencer provides ample throughput for a CXL pod with up to 16 hosts (§ 4.8).

Larger deployments can then scale across pods, with one sequencer per pod, following Boki's logBook-style decomposition [29]. Sequencers independently linearize only their pod-local records, avoiding cross-pod coordination and the cost of a deployment-wide total order.

3.6 Reconfiguration

Reconfiguration reuses the same publication discipline as the normal data path. The sequencer maintains a cluster configuration in a shared CXL-resident region and tags each configuration with a monotonically increasing *epoch*. Each versioned configuration record is persisted on both devices before a cut that names its epoch can be published. To add a shard, a joining shard server posts a request for the sequencer to observe before a sequencing round. To remove a shard, the sequencer marks that shard inactive and the shard stops accepting new records. The important point is that a configuration change becomes effective only when it is published with a global cut. After modifying the configuration, the sequencer increments the epoch and embeds it in the next published global cut. This reuses the same atomic publication boundary as ordinary log data, so reconfiguration cannot expose a mixed old/new cluster view.

3.7 Fault Tolerance

Borges assumes fail-stop failures of an individual shard server, sequencer, or CXL device. The global-cut works as the authoritative record of committed state. Every published cut is replicated to both CXL devices and names a fully replicated data prefix. From the cut sequence and surviving log records, Borges can reconstruct its reader-visible index and other soft metadata. This makes fault tolerance simpler than in designs where multiple nodes maintain partially overlapping volatile state. Borges detects shard and sequencer failures using leases [22]. The lease timeout should be configured above the measured high-percentile local scheduling jitter, which depends on the hardware of deployment.

Sequencer failure. Because the sequencer is stateless, recovery does not require replaying a private control log or reconstructing hidden state. A replacement sequencer restarts from the last fully replicated global cut, installs the index named by that cut as the current front index, and overwrites the back index with the same state. Any partially constructed cut is ignored and overwritten by the next published cut. This recovers from crashes even if the old sequencer failed in the middle of publication.

Shard server failure. When a shard server fails, the sequencer removes it from the active configuration and excludes its reader slot from the index-swap protocol. If the failed shard had an outstanding read that still advertises an old index pointer, the system waits only for that reader's lease to expire. The replacement shard server then recovers its local sequence number, delta cursor, and per-stream byte boundaries from the most recent replicated global cut and its associated index. Any uncommitted state beyond that cut is discarded by construction, because it was never published.

CXL device failure. With $f + 1$ device replicas, Borges preserves committed state despite up to f device failures. After a backup-only failure, the service continues in degraded mode while replacements are rebuilt. A primary failure instead pauses commits.

For the two-replica configuration, recovery scans the survivor's global-cut log backward, skips any torn tail, and selects the newest cut by verifying the checksum field. The most recently complete cut is authoritative for recovery: every record it names had reached both devices and was assigned a legal position in the global order. Records beyond that cut were never published, so no client was acknowledged for them and no reader observed them; recovery discards them. The cut itself may name records whose acknowledgments were still in flight at failure, but those operations were merely pending, so retaining them cannot contradict a completed operation or prior observation. The recovered prefix therefore contains every acknowledged record plus, at most, a legally ordered suffix of pending ones.

Shard servers copy the selected records and the sequencer copies the cut history to the replacement. If the primary index was lost, the sequencer rebuilds it from the selected prefix and restores the derived metadata and configuration.

3.8 Log-Native Exactly-Once Workflows

These design choices also motivate a log-native workflow model. Prior systems typically maintain workflow state in an external store and use the log only as WAL [29, 48, 50, 74].

Following prior workflow systems [29, 48, 74], Borges's client library assigns every logical read and write a unique and deterministic operation identifier that remains stable across retries, allowing replay to recognize duplicates. However, Borges instead treats the log itself as the ground truth. All workflow state is represented as an ordered sequence of read and write records in the shared log.

This leads to a different exactly-once protocol. For writes, Borges simply appends the operation; duplicates do not need to be filtered immediately because replay already reconstructs state from the committed prefix and can suppress repeated logical writes with the same identifier. For reads, Borges appends a read record, waits until that record is committed, and then replays the prefix up to that point. During replay, the workflow runtime tracks operation identifiers, applies each logical write at most once, and returns the result associated with the earliest committed read record for that operation identifier.

The key point is that this protocol is not bolted on top of Borges's design. It is a direct consequence of it. Because Borges already provides low-latency publication of consistent prefixes and efficient replay from shared storage, the log itself can serve as both the durable record and the authoritative state machine for exactly-once execution.

4 Evaluation

This section asks whether Borges' gains come from CXL alone or from its CXL-native design, and whether they reduce both append and replay cost. We answer four questions.

- Is moving coordination onto a CXL device sufficient by itself, or does Borges' single-writer metadata and dual-index publication reduce append cost? (§4.2, §4.3)
- Is replay in Borges cheap enough for latency-sensitive objects while remaining robust to skew? (§4.4, §4.5)
- Do these benefits carry through to end-to-end services such as a key-value store? (§4.6, §4.7)
- Are Borges' design choices practical with respect to configuration, sequencing, and recovery? (§4.8, §4.9 §4.10)

4.1 Experimental Setup and Fairness

Modeling our target hardware. Following prior work on disaggregated-memory evaluation [25], we emulate a pod using virtual machines on a server with an Intel® Xeon 6761P CPU, eight DDR5-6400 memory channels, and a physical 1 TB Samsung CMM-H attached over PCIe 5.0 x8. The device characteristics were summarized earlier in § 2.2. We run eight VMs, each with 5 vCPUs and 10 GB of DRAM, and use SR-IOV on an Intel X550 10 GbE NIC for client/server communication through the Linux TCP/IP stack. As in all current pod prototypes [53, 80], we assume no inter-host hardware cache coherence; all shared-memory coordination therefore uses non-temporal accesses with `clflush` and `fence`. Each host has only one socket, so it experiences no NUMA effects. Borges is backward-compatible with CXL shared memory that does not include a NAND tier.

Borges requires at least two CXL devices for replication, but we only have one physical CMM-H. We therefore split the device into two logical devices with distinct address ranges and duplicated metadata. This is conservative: both logical replicas contend for the bandwidth of one physical device, whereas a real deployment would provide more aggregate bandwidth and less interference between replicas.

Baselines. We compare against Scalog [19] and Boki [29], both reimplemented in our framework for an apples-to-apples comparison. For each baseline we provide a '-NET' variant that communicates as in the original design and a '-CXL' variant that replaces server-side network messages with a CXL message queue, similar to prior CXL prototypes [25, 78]. This lets us separate the benefit of transport from the benefit of software design. For Boki, we preserve its original client-side log engine, the design choice that makes Boki's append path relatively fast; later gaps therefore do not come from weakening its implementation.

We also implement Borges-RDMA, which keeps Borges' software mechanism unchanged but replaces CXL shared load/store access with RDMA memory nodes and one-sided RDMA operations. Our testbed uses Mellanox ConnectX-6 NICs with 25 Gb/s bandwidth and 1.86μs latency. We use Borges-RDMA only in the early append-path discussion to isolate the contribution of the CXL substrate to Borges' own design. Later subsections focus on Scalog-CXL, vScalog-CXL, and Boki-CXL, because those systems represent distinct coordination and replay tradeoffs, whereas Borges-RDMA uses the same mechanism as Borges and already loses on the primitive append path. All experiments are repeated five times; the standard deviation is below 5% of the mean.

4.2 Why a Naive CXL Port Is Still Slow

We first ask two control questions: how much does CXL help Borges' own design, and how far can existing systems improve by simply replacing server-side messages with CXL queue operations? Table 3 answers both.

CXL also benefits Borges itself. We use 2 shards, 2 replicas, and cache-line-padded records with 100 B payloads. Comparing Borges with Borges-RDMA isolates the substrate effect while keeping the software mechanism fixed. RDMA increases end-to-end append latency from 175μs to 252μs. The gap appears primarily in coordination steps: reporting the local cut rises from 13μs to 29μs, updating and switching the index roughly double, and receiving the global cut rises from 2μs to 14μs. This shows that the CXL shared-memory substrate materially benefits Borges' own design. We notice that the throughput gap is not caused by saturating RDMA bandwidth but the per-operation coordination latency.

Transport alone is not enough. Comparing the '-NET' and '-CXL' variants shows that CXL message passing helps

		Borges		Borges-RDMA		Scalog-NET		Scalog-CXL		Boki-NET		Boki-CXL	
Replication (μs)	primary copy	26		56		24		27		122		125	
	backup copy	26		56		122		81		122		125	
	effective (max)	26	(14.86%)	56	(22.22%)	122	(20.64%)	81	(18.33%)	122	(25.85%)	125	(27.17%)
Sequencing (μs)	batch wait for entries	-		-		70	(11.84%)	60	(13.57%)	40	(8.47%)	57	(12.39%)
	report local cut	13	(7.43%)	29	(11.51%)	101	(17.09%)	56	(12.67%)	91	(19.28%)	56	(12.17%)
Committing (μs)	batch wait for local cuts	-		-		58	(9.81%)	51	(11.54%)	61	(12.92%)	60	(13.04%)
	update index	6	(3.43%)	12	(4.76%)	-		-		-		-	
	write global cut	13	(7.43%)	22	(8.73%)	17	(2.88%)	14	(3.17%)	18	(3.81%)	14	(3.04%)
	switch index	6	(3.43%)	11	(4.37%)	-		-		-		-	
	receive global cut	2	(1.14%)	14	(5.56%)	102	(17.26%)	67	(15.16%)	95	(20.13%)	97	(21.09%)
Processing (μs)		4	(2.29%)	3	(1.19%)	14	(2.37%)	11	(2.49%)	14	(2.97%)	19	(4.13%)
Client_IO (μs)	send and receive request	105	(60.00%)	105	(41.67%)	107	(18.10%)	102	(23.08%)	31	(6.57%)	32	(6.96%)
Total (μs)		175		252		591		442		472		460	

Table 3. Append latency breakdown showing the mean latency of each part. Parentheses report each component's share of the end-to-end total. Borges-RDMA keeps Borges' mechanism but replaces CXL shared load/store access with RDMA; Scalog/Boki '-NET' and '-CXL' isolate the effect of changing transport without redesigning metadata ownership.

	Borges	Borges-RDMA	Scalog-CXL	Boki-CXL
T-put (req/s)	161.91K	38.38K (4.22×)	74.96K (2.16×)	60.76K (2.66×)
P50 (μs)	157.24	270.01 (1.72×)	406.36 (2.58×)	402.59 (2.56×)
P99 (μs)	184.21	305.78 (1.66×)	528.57 (2.87×)	511.39 (2.78×)

Table 4. End-to-end append latency and throughput. Parenthesized values show slowdown relative to Borges.

Scalog and Boki, but only modestly changes the overall picture. Scalog-CXL is faster than Scalog-NET, and Boki-CXL is only slightly faster than Boki-NET. This small Boki gap is expected: its original client-side log engine already makes append relatively fast. More broadly, neither approach closes the gap to Borges, because both retain metadata protocols that are poorly matched to non-coherent shared memory.

This mismatch is clearest in Scalog-CXL. Without hardware coherence, each queue operation becomes a flush-read-modify-write cycle on shared metadata. In our measurements, lock acquisition alone contributes 15.2%, 15.9%, and 15.7% of replication, sequencing, and publication latency, respectively, and 24.1% of end-to-end p99 append latency. By contrast, Borges keeps each shared word single-writer and uses ordinary loads and stores. As a result, compared with Scalog-CXL, Borges reduces replication latency by 3.1×, sequencing latency by 8.9×, and commit latency by 4.9×.

Client-side headroom. Table 3 identifies Linux TCP/IP, rather than the CXL path, as Borges' dominant bottleneck: `Client_IO` consumes 105μs (60%) of its 175μs append latency, leaving a measured 70μs server-side path. Borges variants and Scalog variants receive client requests through the kernel TCP/IP stack. Boki instead colocates its log engine with the client, avoiding this network hop and thus having much faster client-I/O. For comparison, client I/O accounts for 41.67%, 23.08%, and 6.96% of Borges-RDMA, Scalog-CXL, and Boki-CXL, leaving server-side paths of 147μs, 340μs, and 428μs, respectively. Under an idealized zero-client-I/O bound, Borges' 70μs path would yield speedups of 2.1×, 4.9×, and 6.1× over these systems, widening the gap between Borges and prior work.

Takeaway. CXL helps in two ways: it speeds Borges itself relative to Borges-RDMA, and it modestly improves naive ports of prior systems. The larger win still comes from redesigning coordination and publication for non-coherent shared memory.

4.3 Log Append

We next evaluate the end-to-end append path, the most direct test of Borges' single-writer metadata and publication design.

End-to-end append latency and throughput. Table 4 reports low-load latency and saturation throughput. We use one client for low-load latency and increase the number of clients until throughput saturates (160 for Borges, 120 for Scalog-CXL, and 100 for Boki-CXL). Borges achieves both the best latency and the best throughput.

The comparison to Borges-RDMA isolates the value of the CXL substrate for the Borges design itself. Even though the mechanism is unchanged, Borges improves over Borges-RDMA by 1.72× at p50, 1.66× at p99, and 4.22× in throughput. This is why we do not carry Borges-RDMA through the rest of the section: once the primitive append path already shows that the same design is slower on RDMA, the more informative later comparisons are against systems with different coordination and replay strategies.

The comparison to Scalog-CXL and Boki-CXL captures the architectural benefit of Borges. Boki-CXL is already an append-friendly baseline because Boki's original client-side log engine keeps log append relatively fast, yet Borges still improves over Scalog-CXL by 2.16× in throughput and 2.87× at p99, and over Boki-CXL by 2.66× in throughput and 2.78× at p99. The gap is larger than the Borges-vs-Borges-RDMA gap because they pay the cost of coordination mechanisms that are poorly matched to non-coherent shared memory.

	Borges	vScalog-CXL	Boki-CXL
T-put (req/s)	37.12K	8.50K (4.37×)	0.38K (97.68×)
P50 (μs)	326.58	1007.47 (3.08×)	4484.73 (13.73×)
P99 (μs)	370.15	1239.19 (3.35×)	10271.47 (27.75×)

Table 5. End-to-end latency and throughput for a linearizable lock built on each log.

Latency under load. To test whether Borges retains its advantage near saturation, we raise offered load to 40 K, 50 K, and 60 K appends/s. At these loads, Borges' p99 latencies are 271.9μs, 298.2μs, and 326.4μs, respectively. By comparison, Scalog-CXL's corresponding p99 latencies are 798.9μs, 949.6μs, and 1.3ms (2.9×, 3.2×, and 4.0× those of Borges), while Boki-CXL's are 0.9ms, 1.0ms, and 1.4ms (3.3×, 3.4×, and 4.3× those of Borges). The advantage widens under load because Borges does not rely on batching to amortize coordination; its fast path stays small even as queues form.

Takeaway. Borges' lock-free design delivers low latency on append path without batching, and its advantage grows under load, precisely when latency matters most.

4.4 Linearizable Lock

We now turn to replay. A linearizable lock is a useful stress test because each request must append a record, replay the relevant stream, and confirm lock state before returning.

We use the YCSB key-generation setting with 102,400 keys and Zipf parameter $\theta = 0.99$. A `Lock` request appends a lock record and then replays the stream to determine whether the acquisition succeeds, as in Boki [29]. An `Unlock` request appends an unlock record. The end-to-end latency of a lock operation includes both acquisition and release; clients back off and retry on conflict. Scalog does not natively support locks, so we implement Boki's lock protocol on top of vScalog [19], a stream-aware variant that binds each stream to a single shard. This makes vScalog-CXL the natural baseline for the single-shard-replay design point.

Table 5 shows that Borges is 3.1× faster than vScalog-CXL at p50 and 13.7× faster than Boki-CXL under two clients. As we increase the number of clients to 4, 8, and 16 (the point at which vScalog-CXL saturates), Borges shows p99 latencies of 830.0μs, 1.5ms, and 2.7ms, whereas vScalog-CXL reaches 1.9ms (2.3×), 3.7ms (2.5×), and 15.0ms (5.6×), respectively.

The two baselines lose for opposite reasons, which is precisely the replay tradeoff that Borges is designed to change. Boki-CXL spreads a stream across shards, but replay still goes through remote communication with its client-side log engine; on our platform, a remote record fetch costs 673μs at p50 and exceeds 2.5 ms at p99. Borges instead reads committed remote records directly from shared CXL memory, reducing cross-shard fetch to 10.1μs at p99. vScalog-CXL avoids remote replay, but its placement is too rigid: binding each stream to one shard forces random per-record lookups within that shard-local history and causes a lookup latency 2.5× higher than Borges.

		Uniform	$\theta = 0.7$	$\theta = 0.9$	$\theta = 0.99$
Borges	T-put	147.3 (100%)	146.7 (99.6%)	144.5 (98.1%)	144.2 (97.9%)
	P50	620.1 (100%)	623.5 (100.5%)	640.4 (103.3%)	642.1 (103.5%)
	P99	1222.0 (100%)	1249.4 (102.2%)	1289.4 (105.5%)	1317.0 (107.8%)
vScalog	T-put	60.4 (100%)	53.6 (88.7%)	46.1 (76.3%)	42.0 (69.5%)
	P50	1574.4 (100%)	1807.8 (114.8%)	2273.7 (144.4%)	2338.9 (148.6%)
	P99	2842.0 (100%)	3129.9 (110.1%)	3511.1 (123.5%)	3693.2 (130.0%)

Table 6. A counter workload under increasing Zipf skew (§ 4.5). Throughput is measured in Kreq/s, while the p50 and p99 latencies are measured in μs. Values in parentheses are normalized to the uniform case.

Publication cost. We observe that dual-index publication costs 3.4μs at p50 and 9.8μs at p99 with four readers, rising to 8.9μs and 22.6μs with twelve readers. This suggests modest reader-sensitive overhead added in the pod-scale setting.

Takeaway. The lock benchmark shows that Borges is not just a faster append engine. The benefit comes from both halves of its design: the dual-index protocol and per-stream indexing that makes replay path cheap enough for latency-sensitive, serialized objects.

4.5 Skew Tolerance

The lock result shows that Borges makes replay cheap; the next question is whether replay stays cheap when Borges uses that flexibility to spread a stream across shards under skew. We therefore compare Borges with vScalog-CXL on a counter workload where each request appends a delta and then replays the counter's history to obtain the current value.

As in YCSB, keys are drawn from a Zipf distribution. In vScalog-CXL, only the owning shard can serve a given counter. In Borges, either shard can append to and replay that counter because committed records remain directly readable from the shared CXL device. We first find the maximum number of clients each system can support under a uniform workload (140 for Borges, 120 for vScalog-CXL), then hold the client count fixed and increase skew.

Table 6 shows that Borges remains almost flat as skew rises: from uniform access to $\theta = 0.99$, throughput drops by only 2.1%. In contrast, vScalog-CXL loses nearly 30% of throughput. The reason is load imbalance: under high skew, the hot shard in vScalog-CXL reaches 2.25× the CPU utilization of the other shard.

To examine the latency effect of this imbalance, we fix the client count at 96 for both systems. This setting keeps both systems near saturation across all skew levels while avoiding overload-induced latency spikes, and ensures a fair comparison with the same client concurrency. As skew increases, vScalog-CXL's p50 and p99 latency rise by 48.6% and 30.0%, whereas Borges' latency changes only slightly.

Boki-CXL is not included in this skew table because it already chooses the cross-shard placement side of the design space; its weakness was exposed in § 4.4, where remote

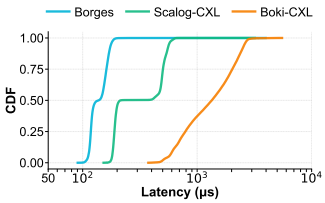

Figure 6. YCSB-A.

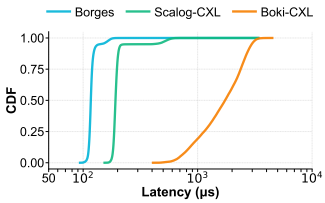

Figure 7. YCSB-B.

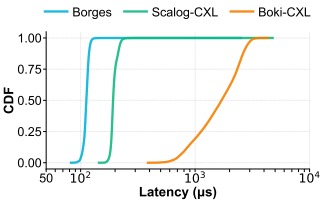

Figure 8. YCSB-C.

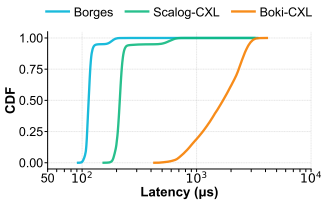

Figure 9. YCSB-D.

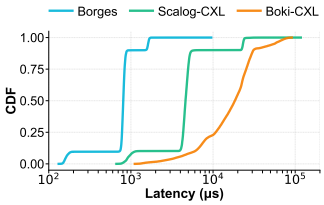

Figure 10. Retwis.

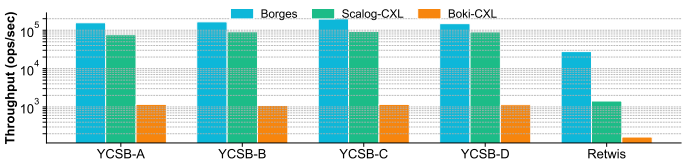

Figure 11. Throughput for YCSB A–D, and Retwis

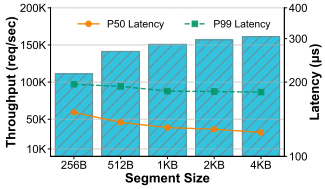

Figure 12. Segment size study.

replay dominated latency. The skew experiment therefore isolates the remaining question: what happens if a system keeps replay local by binding each stream to one shard?

Takeaway. Allowing streams to span shards is valuable only if cross-shard replay stays cheap. Borges achieves both, which is why it avoids the usual skew-vs-replay tradeoff.

4.6 Key-Value Store

We next test whether Borges' append and replay advantages carry to a linearizable key-value store. We run YCSB [17] workloads A–D, i.e., 50/50 read-write, 95/5 read-write, read-only, and 95/5 read-insert. We use 102,400 8-byte keys from a Zipf distribution with $\theta = 0.99$ and 1024-byte values, representative of skewed serving workloads [15, 20, 52].

We implement ScalogStore on Scalog-CXL and ensure that all systems provide linearizable semantics. Boki-CXL requires special care: because its index is updated asynchronously, a read cannot trust that index alone. Each read therefore appends a read record and replays the log to that point, reintroducing the same remote-replay penalty seen in § 4.4. We observe p50/p99 read-side latency of 445μs/1.7 ms just from waiting for index updates and remote replay.

Figures 6–9 show that Borges consistently reduces tail latency across all four workloads. Relative to Scalog-CXL, Borges cuts p99 latency by 1.8×–3.3× across YCSB A–D. Relative to Boki-CXL, the reduction is larger still, ranging from 14.9× to 25.2×.

Figure 11 shows that throughput improves as well: Borges sustains 156.4 K, 165.5 K, 198.6 K, and 148.4 K requests/s on YCSB A–D, improving over Scalog-CXL by 1.6×–2.1×. These gains are smaller than the latency gains because Scalog-CXL uses group commit to trade latency for throughput, whereas Borges largely avoids that tradeoff.

4.7 Fault-Tolerant Workflow

Finally, we evaluate the log-native workflow design from § 3.8. We adapt Halfmoon [48]'s version of Retwis [49], a Twitter-like application decomposed into fault-tolerant workflows that issue read and write operations. We use the same setting as Halfmoon: 10,000 users, each with eight followers, and a unique workflow identifier for every request.

BokiFlow [29] provides a fault-tolerant workflow implementation, which we also adapt to Scalog-CXL. In both baselines, workflow state lives in an external MongoDB instance whose write and read latency are 1.02 ms and 0.65 ms, respectively. To obtain exactly-once semantics, each database access is wrapped in a test-and-append style protocol that incurs both a log append and log playback. Borges instead keeps authoritative state in the log itself and uses replay-time deduplication, avoiding both the external DB round trip and an extra deduplication round on the critical path.

Figure 10 shows a large application-level gap. Borges achieves 806.7μs p50 latency and 1.7 ms p99. Scalog-CXL reaches 4.6 ms/23.9 ms p50/p99, while Boki-CXL reaches 18.1 ms/70.7 ms. Figure 11 shows the corresponding throughput on a log scale: Borges sustains 27,722 requests/s, versus 1,426 for Scalog-CXL.

Takeaway. Borges' low-latency publication and replay are strong enough to serve as both durable record and authoritative state machine for exactly-once workflows.

4.8 Configuration Tradeoffs

We next ask whether Borges' design choices are brittle by showing how key parameters affect performance.

Segment size. Segments are the unit that lets Borges preserve mostly sequential replay. Figure 12 shows the tradeoff. Increasing the segment size from 256 B to 4 KB raises throughput by 45.1% while reducing p50 and p99 latency by 16.7% and 6.8%. The benefit comes from less frequent allocation and fewer segment links on the fast path. The tradeoff is fragmentation: with 4 KB segments, total internal fragmentation is 25.03 MB (19.58%) across 8.8 K allocations, whereas with 256 B segments it drops to 18.30 MB (11.53%) but requires 382 K allocations. Our 2 KB default balances these costs and yields 16.7% fragmentation.

	Record size				Shards			Replicas		
	100B	512B	1KB	2KB	1	2	4	1	2	3
T-put (req/s)	177.56K	165.74K	125.67K	71.81K	84.27K	164.82K	223.77K	170.64K	160.83K	149.68K
P50-lat (us)	153.21	158.62	175.83	180.24	141.61	153.19	157.77	155.26	153.78	166.21
P99-lat (us)	179.66	186.12	192.91	214.67	164.78	184.87	189.21	176.83	184.02	190.14

Table 7. Append throughput and latency under varying record size, shard count, and replica count.

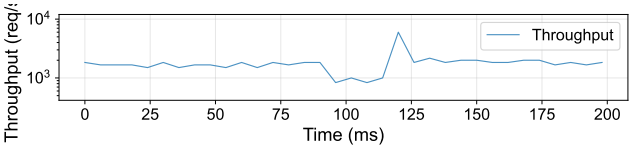

Figure 13. Throughput during shard failure and recovery.

Record size. The left portion of Table 7 shows that throughput decreases by 2.47× as the record size grows from 100 B to 2 KB with 1 replica, while p50 and p99 latency increase by only 17.6% and 19.5%. This is the expected signature of Borges' design goal: it keeps per-operation coordination small, so larger records mainly reduce throughput rather than sharply increasing control-path latency.

Shard count. The middle portion of Table 7 shows that increasing the number of shards from 1 to 4 with 2 replicas raises throughput by 2.7× because more shard servers can accept and replicate requests in parallel. The corresponding p50/p99 latency increase of 11.4%/14.8% reflects higher contention for shared storage and more metadata flowing through the sequencer. This is a reasonable tradeoff for the pod-scale setting Borges targets: shard parallelism scales throughput faster than it degrades latency.

Replica count. The right portion of Table 7 shows that durability is relatively cheap. Increasing the replica factor from 1 to 3 reduces throughput by only 12.3% and increases p50/p99 latency by 7.1%/7.5%. This suggests that the replication path is not dominated by extra coordination metadata; most of the cost remains in copying the data itself.

4.9 Single-Sequencer Scalability

We next examine if a single sequencer can keep up with the metadata load of a pod-scale log.

Our physical testbed can host only six shard servers, so to emulate larger pods we add synthetic shard streams that deliver records to the sequencer at the same rate as real shards but do not originate from clients. As we scale to 8, 12, and 16 shards, sequencer throughput grows to 341.9 K, 502.7 K, and 612.1 K requests/s. The corresponding end-to-end latency, computed as emulated shard latency plus client I/O, remains modest: p50/p99 rise from 300.0μs/340.6μs at 8 shards to 360.1μs/378.2μs at 12 shards and 382.4μs/450.7μs at 16 shards. Even at 16 shards, the single five-core sequencer uses only about 40% CPU. This is exactly the behavior Borges was designed for: at pod scale, sequencing is metadata-only work over shared memory rather than a network-bound control-plane bottleneck.

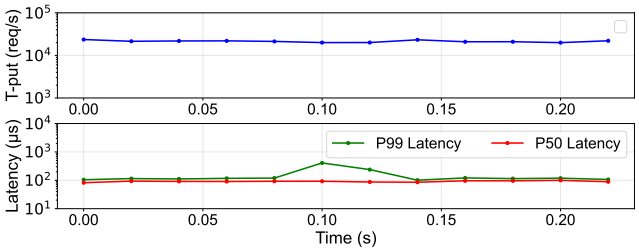

Figure 14. Throughput and latency during reconfiguration.

4.10 Recovery and Reconfiguration

We finally evaluate operational behavior. We do not separately benchmark sequencer restart because the sequencer is stateless and reconstructs its state from the last committed cut/index pair. We instead focus on the events that directly perturb the data path: shard recovery and reconfiguration.

Shard recovery. While running a two-shard system, we kill one shard server and measure throughput. The sequencer detects failure through lease expiration, seals the failed shard, and launches a replacement instance. The replacement recovers its progress from the most recent global cut and associated index. Figure 13 shows that throughput briefly overshoots as the system drains the backlog and then returns to steady state. In this run, no reader was still holding an old index pointer. If a failed reader had still been advertising an old pointer, the system would wait for the corresponding lease to expire before retiring that index, preserving the publication invariant.

Reconfiguration. We evaluate reconfiguration by adding a new shard server. The joining shard posts a request, the sequencer observes it before a sequencing round, updates the cluster configuration in CXL memory, and publishes the new epoch in the next global cut. Existing shard servers learn the new membership only through that published cut, so configuration change reuses the same visibility boundary as ordinary log data. Figure 14 shows a temporary p99 latency increase while shard servers fetch the new configuration, but throughput remains stable because the reconfiguration metadata is small and pending cuts are processed quickly.

5 Related Work

Distributed shared logs. Shared-log systems provide a total-order abstraction, but existing designs either keep ordering and replication on the network path or relax when global order is fixed. CORFU [8] introduced the abstraction; Tango [9] and vCorfu [64] built replicated data structures

and cloud storage on top of it; Scalog [19], Delos [7], and Boki [29] improved scalability, reconfiguration, and serverless support. FuzzyLog [40] offers partial order, while LazyLog [41] and SpecLog [11] reduce perceived latency by deferring or speculating on global order. In contrast, Borges targets low *actual* commit and replay latency under strict total order by placing log state in shared CXL storage and turning cross-shard replay into a direct shared-memory read. Mu [1] and Derecho [28] also pursue low latency, but they build consensus-based state machine replication based on RDMA, not a shared-log abstraction. Their designs rely on NIC-managed one-sided verbs and NIC-level ordering, which do not apply to CXL's non-coherent shared memory.

CXL-based memory and storage systems. At the infrastructure layer, CXL systems and related tiered-memory work explore resource pooling [34, 78, 79], page placement and tiering [39, 44, 62, 65, 77], and CXL-memory characterization and cost optimization [38, 57]. Other work builds multi-host shared-memory mechanisms for RPC, DSM, process cloning, and data processing [3, 42, 45, 56]. At the storage layer, Overcoming the Memory Wall [66] explores memory-semantic flash, while SkyByte [75], ByteFS [35], and RomeFS [73] redesign device, OS/firmware, and file-system stacks; Cylon [70] adds fast full-system emulation. XHarvest [47] and Espresso [68] instead share controller resources between a host and SSD or across SSDs while retaining block-I/O data paths. ANCHOR [81] uses a memory-semantic SSD as the persistence and error-repair anchor for a single host, whereas DJFS [69] and Transparent DAX Mappings [31] redesign file-system and mapping stacks for CMM-H. CXL-backed database systems build on these mechanisms for transaction processing and for pod-scale or in-memory databases [2, 25, 63], while Database Kernels and the Key-Space Partitioned LSM Tree adapt database and storage engines to memory-semantic devices [33, 36]. PolarCXLMem [67] uses CXL as a mutable buffer-pool/page layer with software coherence; Borges's append-only log avoids fast-path coherence via single-writer metadata and dual-index publication. Complementary work formalizes CXL cache coherence and partial-crash semantics [6, 55], model-checks crashing CXL programs [23], or provides failure-aware shared-memory management [46, 76, 83].

Log-centric data systems. Shared logs underpin several data services. Impeller stores stream data and exactly-once progress in one; Delos composes production database protocols above one; and Conflux runs a production metadata service over a cloud-backed shared log, separating durability (LogDrive) from sequencing (AtomicLog) [10, 61, 82]. CalvinFS orders file-system metadata transactions through a global log, while Ethane delegates file-system durability, linearizability, and cache coherence to a shared operation log [13, 59]. XLL instead shares one persistent log across TiKV's consensus and local-database layers to eliminate duplicate writes [51]. MEGALON [24] is closest: its CXL-resident log orders replicated-index updates and software coherence, but supports replication rather than recovery and treats any host or CXL-memory failure as system-wide. Socrates, LogStore, and Clover use logs more narrowly for WAL, operational-log storage, or per-key access, respectively [4, 14, 60]. Unlike them, Borges exposes a fault-tolerant shared log whose global order, cuts, indexes, and replay state are application-visible over a non-coherent CXL memory/SSD hybrid.

6 Conclusion

Borges shows that CXL DRAM/SSD hybrids enable a new design point for distributed shared logs. By replacing network coordination with lock-free single-writer metadata and dual-index publication, Borges preserves strict total order and fault tolerance while substantially reducing append and replay latency. Our results show clear gains over prior systems on append latency, workflows, and skewed workloads.

7 Acknowledgment

Our work is supported in part by PRISM, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. We used the Chameleon testbed [30] supported by the National Science Foundation for development and artifact evaluation.

References

[1] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. 2020. Microsecond Consensus for Microsecond Applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Virtual Event, 599–616. https://www.usenix.org/conference/osdi20/presentation/aguilera

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

[3] 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 ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ACM, New York, NY, USA, 210–226. doi:10.1145/3676641.3715988

[4] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, New York, NY, USA, 1743–1756. doi:10.1145/3299869.3314047

[5] Apache Software Foundation. 2026. *ZooKeeper Administrator's Guide: A Guide to Deployment and Administration.* https://zookeeper.apache.org/doc/r3.9.5/zookeeperAdmin.html#sc_zkMulitServerSetup Apache ZooKeeper 3.9.5, section "Clustered (Multi-Server) Setup"; accessed August 22, 2026.

[6] Gal Assa, Moritz Lumme, Lucas Bürgi, Michal Friedman, and Ori Lahav. 2026. A Programming Model for Disaggregated Memory over CXL. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ACM, New York, NY, USA, 41–58. doi:10.1145/3779212.3790121

[7] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. 2020. Virtual Consensus in Delos. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Virtual Event, 617–632. https://www.usenix.org/conference/osdi20/presentation/balakrishnan

[8] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. 2012. CORFU: A Shared Log Design for Flash Clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 1–14. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/balakrishnan

[9] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. 2013. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 325–340. doi:10.1145/2517349.2522732

[10] Mahesh Balakrishnan, Chen Shen, Ahmed Jafri, Suyog Mapara, David Geraghty, Jason Flinn, Vidhya Venkat, Ivailo Nedelchev, Santosh Ghosh, Mihir Dharamshi, Jingming Liu, Filip Gruszczynski, Jun Li, Rounak Tibrewal, Ali Zaveri, Rajeev Nagar, Ahmed Yossef, Francois Richard, and Yee Jiun Song. 2021. Log-structured Protocols in Delos. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 538–552. doi:10.1145/3477132.3483544

[11] Shreesha G. Bhat, Tony Hong, Xuhao Luo, Jiyu Hu, Aishwarya Ganesan, and Ramnatthan Alagappan. 2025. Low End-to-End Latency atop a Speculative Shared Log with Fix-Ante Ordering. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA, 465–481. https://www.usenix.org/conference/osdi25/presentation/bhat

[12] Mike Burrows. 2006. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*. USENIX Association, Seattle, WA, 335–350. https://www.usenix.org/conference/osdi-06/chubby-lock-service-loosely-coupled-distributed-systems

[13] Miao Cai, Junru Shen, and Baoliu Ye. 2024. Ethane: An Asymmetric File System for Disaggregated Persistent Memory. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 191–207. https://www.usenix.org/conference/atc24/presentation/cai

[14] Wei Cao, Xiaojie Feng, Boyuan Liang, Tianyu Zhang, Yusong Gao, Yunyang Zhang, and Feifei Li. 2021. LogStore: A Cloud-Native and Multi-Tenant Log Database. In *Proceedings of the 2021 International Conference on Management of Data*. ACM, New York, NY, USA, 2464–2476. doi:10.1145/3448016.3457565

[15] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 209–223. https://www.usenix.org/conference/fast20/presentation/cao-zhichao

[16] Compute Express Link Consortium, Inc. 2024. *Compute Express Link (CXL) Specification, Revision 3.2, Version 1.0.* https://computeexpresslink.org/wp-content/uploads/2024/11/CXL-Specification_rev3p2_ver1p0_2024October2_evalcopy.pdf Evaluation copy; accessed August 2025.

[17] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. doi:10.1145/1807128.1807152

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

[19] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. 2020. Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 325–338. https://www.usenix.org/conference/nsdi20/presentation/ding

[20] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Virtual Event, 33–49. https://www.usenix.org/conference/fast21/presentation/dong

[21] Donghyun Gouk, Seungkwan Kang, Hanyeoreum Bae, Eojin Ryu, Sangwon Lee, Dongpyung Kim, Junhyeok Jang, and Myoungsoo Jung. 2024. Breaking Barriers: Expanding GPU Memory with Sub-Two Digit Nanosecond Latency CXL Controller. In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems* (Santa Clara, CA, USA) *(HotStorage '24)*. Association for Computing Machinery, New York, NY, USA, 108–115. doi:10.1145/3655038.3665953

[22] C. Gray and D. Cheriton. 1989. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP '89)*. Association for Computing Machinery, New York, NY, USA, 202–210. doi:10.1145/74850.74870

[23] Simon Guo, Conan Truong, and Brian Demsky. 2026. CXLMC: Model Checking CXL Shared Memory Programs. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ACM, New York, NY, USA, 546–562. doi:10.1145/3779212.3790150

[24] Jiyu Hu, Seokjoo Cho, Landon Johnson, Kiran Hombal, Shreesha Gopalakrishna Bhat, Marcos K. Aguilera, Ramnatthan Alagappan, and Aishwarya Ganesan. 2026. MEGALON: Efficient Data Sharing for Partly Coherent CXL Memory. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*. USENIX Association, Seattle, WA, 2281–2298. https://www.usenix.org/conference/osdi26/presentation/hu-jiyu

[25] 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 *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA, 109–128. https://www.usenix.org/conference/osdi25/presentation/huang-yibo

[26] Yibo Huang, Newton Ni, Vijay Chidambaram, Emmett Witchel, and Dixin Tang. 2025. Pasha: An Efficient, Scalable Database Architecture for CXL Pods. In *15th Annual Conference on Innovative Data Systems Research (CIDR '25)*. www.cidrdb.org, Amsterdam, The Netherlands, 8 pages. https://vldb.org/cidrdb/papers/2025/p8-huang.pdf

[27] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. USENIX Association, Boston, MA, 14 pages. https://www.usenix.org/conference/usenix-atc-10/zookeeper-wait-free-coordination-internet-scale-systems

[28] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Mae Milano, Weijia Song, Edward Tremel, Robbert van Renesse, Sydney Zink, and Kenneth P. Birman. 2019. Derecho: Fast State Machine Replication for Cloud Services. *ACM Transactions on Computer Systems* 36, 2, Article 4 (2019), 49 pages. doi:10.1145/3302258

[29] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful Serverless Computing with Shared Logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 691–707. doi:10.1145/3477132.3483541

[30] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association.

[31] Yussuf Khalil, Daniel Habicht, Pascal Ellinger, Frank Bellosa, Javier González, Adam Manzanares, and Vivek Shah. 2025. Transparent DAX Mappings: Towards Automatic Kernel Bypass with CXL-Based Hybrid SSDs. In *Proceedings of the 3rd Workshop on Disruptive Memory Systems* (Seoul, Republic of Korea) *(DIMES '25)*. Association for Computing Machinery, New York, NY, USA, 54–62. doi:10.1145/3764862.3768178

[32] Hayan Lee, Jungwoo Kim, Wookyung Lee, Juhyung Park, Sanghyuk Jung, Jinki Han, Bryan S. Kim, Sungjin Lee, and Eunji Lee. 2025. Revisiting Trim for CXL Memory. In *Proceedings of the 17th ACM Workshop on Hot Topics in Storage and File Systems* (Boston, MA, USA) *(HotStorage '25)*. Association for Computing Machinery, New York, NY, USA, 24–30. doi:10.1145/3736548.3737825

[33] Sangjin Lee, Alberto Lerner, Philippe Bonnet, and Philippe Cudré-Mauroux. 2024. Database Kernels: Seamless Integration of Database Systems and Fast Storage via CXL. In *14th Annual Conference on Innovative Data Systems Research (CIDR '24)*. www.cidrdb.org, Santa Cruz, CA, USA, 8 pages. https://www.vldb.org/cidrdb/papers/2024/p43-lee.pdf

[34] 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 ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 574–587. doi:10.1145/3575693.3578835

[35] Shaobo Li, Yirui (Eric) Zhou, Hao Ren, and Jian Huang. 2025. ByteFS: System Support for (CXL-based) Memory-Semantic Solid-State Drives. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 116–132. doi:10.1145/3669940.3707250

[36] Seungho Lim, Seung Won Yoo, Joontaek Oh, Wonseb Jeong, Hyunsub Song, Hyeonho Song, Donghun Lee, and Youjip Won. 2024. Key-Space Partitioned LSM Tree for CMM-H. In *2024 13th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, Gangwon-do, Republic of Korea, 1–6. doi:10.1109/NVMSA63038.2024.10693654

[37] Xiancheng Lin, Jing Peng, Rongkai Liu, and Xiang Gao. 2024. Research on the CXL Memory. In *2024 IEEE 6th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*. IEEE, Chongqing, China, 1428–1432. doi:10.1109/IMCEC59810.2024.10575852

[38] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S. Berger, Marie Nguyen, Xun Jian, Sam H. Noh, and Huaicheng Li. 2025. Systematic CXL Memory Characterization and Performance Analysis at Scale. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1203–1217. doi:10.1145/3676641.3715987

[39] Jinshu Liu, Hamid Hadian, Hanchen Xu, and Huaicheng Li. 2025. Tiered Memory Management Beyond Hotness. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA, 731–747. https://www.usenix.org/conference/osdi25/presentation/liu

[40] Joshua Lockerman, Jose M. Faleiro, Juno Kim, Soham Sankaran, Daniel J. Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. 2018. The FuzzyLog: A Partially Ordered Shared Log. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 357–372. https://www.usenix.org/conference/osdi18/presentation/lockerman

[41] Xuhao Luo, Shreesha G. Bhat, Jiyu Hu, Ramnatthan Alagappan, and Aishwarya Ganesan. 2024. LazyLog: A New Shared Log Abstraction for Low-Latency Applications. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 296–312. doi:10.1145/3694715.3695983

[42] Teng Ma, Zheng Liu, Chengkun Wei, Jialiang Huang, Youwei Zhuo, Haoyu Li, Ning Zhang, Yijin Guan, Dimin Niu, Mingxing Zhang, and Tao Ma. 2024. HydraRPC: RPC in the CXL Era. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 387–395. https://www.usenix.org/conference/atc24/presentation/ma

[43] Shunyu Mao, Jiajun Luo, Yixin Li, Jiapeng Zhou, Weidong Zhang, Zheng Liu, Teng Ma, and Shuwen Deng. 2025. CXL-INTERPLAY: Unraveling and Characterizing CXL Interference in Modern Computer Systems. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. IEEE, San Francisco, CA, USA, 1–7. doi:10.1109/DAC63849.2025.11132607

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

[45] Qiutong Men, Tao Wang, Jongryool Kim, Hane (Stella) Yie, Emmanuel Amaro, Marcos K. Aguilera, and Aurojit Panda. 2026. Duhu: Shared Disaggregated Memory for Distributed Data Processing Frameworks. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*. USENIX Association, Seattle, WA, 921–938. https://www.usenix.org/conference/osdi26/presentation/men

[46] 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 ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (USA) *(ASPLOS '26)*. Association for Computing Machinery, New York, NY, USA, 528–545. doi:10.1145/3779212.3790149

[47] Li Peng, Wenbo Wu, Shushu Yi, Xianzhang Chen, Chenxi Wang, Shengwen Liang, Zhe Wang, Nong Xiao, Qiao Li, Mingzhe Zhang, and Jie Zhang. 2025. XHarvest: Rethinking High-Performance and Cost-Efficient SSD Architecture with CXL-Driven Harvesting. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. ACM, New York, NY, USA, 434–449. doi:10.1145/3695053.3731028

[48] Sheng Qi, Xuanzhe Liu, and Xin Jin. 2023. Halfmoon: Log-Optimal Fault-Tolerant Stateful Serverless Computing. In *Proceedings of the*

29th Symposium on Operating Systems Principles (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 314–330. doi:10.1145/3600006.3613154

[49] Salvatore Sanfilippo. 2009. Redis Patterns Example: Building a Twitter Clone (Retwis). https://redis.io/docs/latest/develop/clients/patterns/twitter-clone/ Original article written in 2009; accessed August 2025.

[50] Srinath Setty, Chunzhi Su, Jacob R. Lorch, Lidong Zhou, Hao Chen, Parveen Patel, and Jinglei Ren. 2016. Realizing the Fault-Tolerance Promise of Cloud Storage Using Locks with Intent. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 501–516. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/setty

[51] John Shawger, Arnav Jhingran, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2026. XLL: Cross-Layer Logging for Data Deduplication in Consensus-Based Storage. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. USENIX Association, Renton, WA, 2619–2634. https://www.usenix.org/conference/nsdi26/presentation/shawger

[52] 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 *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 81–98. https://www.usenix.org/conference/fast23/presentation/shen

[53] SK hynix. 2024. SK hynix Presents CXL Memory Solutions Set to Power the AI Era at CXL DevCon 2024. https://news.skhynix.com/en/sk-hynix-presents-ai-memory-solutions-at-cxl-devcon-2024/ Published May 2, 2024; accessed August 2025.

[54] Mohammadreza Soltaniyeh, Gongjin Sun, Xuebin Yao, Amir Beygi, Ramdas Kachare, Dongwan Zhao, Hingkwan Huen, Andrew Chang, Senthil Murugesapandian, and Caroline Kahn. 2025. Revisiting Memory Hierarchies with CMM-H: Use Device-side Caching to Integrate DRAM and SSD for a Hybrid CXL Memory. In *Proceedings of the 17th ACM Workshop on Hot Topics in Storage and File Systems* (Boston, MA, USA) *(HotStorage '25)*. Association for Computing Machinery, New York, NY, USA, 45–51. doi:10.1145/3736548.3737828

[55] Chengsong Tan, Alastair F. Donaldson, and John Wickerson. 2025. Formalising CXL Cache Coherence. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. ACM, New York, NY, USA, 437–450. doi:10.1145/3676641.3715999

[56] Wenda Tang, Ying Han, Tianxiang Ai, Guanghui Li, Bin Yu, and Xin Yang. 2024. Yggdrasil: Reducing Network I/O Tax with (CXL-Based) Distributed Shared Memory. In *Proceedings of the 53rd International Conference on Parallel Processing* (Gotland, Sweden) *(ICPP '24)*. Association for Computing Machinery, New York, NY, USA, 597–606. doi:10.1145/3673038.3673138

[57] Yupeng Tang, Ping Zhou, Wenhui Zhang, Henry Hu, Qirui Yang, Hao Xiang, Tongping Liu, Jiaxin Shan, Ruoyun Huang, Cheng Zhao, Cheng Chen, Hui Zhang, Fei Liu, Shuai Zhang, Xiaoning Ding, and Jianjun Chen. 2024. Exploring Performance and Cost Optimization with ASIC-Based CXL Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 818–833. doi:10.1145/3627703.3650061

[58] The etcd Authors. 2025. etcd Frequently Asked Questions. https://etcd.io/docs/v3.7/faq/ [Accessed Aug. 2026].

[59] Alexander Thomson and Daniel J. Abadi. 2015. CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 1–14. https://www.usenix.org/conference/fast15/technical-sessions/presentation/thomson

[60] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Virtual Event, 33–48. https://www.usenix.org/conference/atc20/presentation/tsai

[61] Gardner Vickers, Lucas Bradstreet, Mahesh Balakrishnan, Prince Mahajan, David Mao, Xavier Léauté, Ismael Juma, Nikhil Bhatia, Jack Vanlightly, Prateek Jindal, Sumit Arrawatia, Andrew Grant, Dhruvil Shah, Dimitar Dimitrov, Gaurav Badoni, Shimiao Zhang, and Yang Yu. 2026. The LogDrive: Composable Durability for Cloud-Based Shared Logs. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*. USENIX Association, Seattle, WA, 1663–1682. https://www.usenix.org/conference/osdi26/presentation/vickers

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

[63] Zhao Wang, Yiqi Chen, Cong Li, Yijin Guan, Dimin Niu, Tianchan Guan, Zhaoyang Du, Xingda Wei, and Guangyu Sun. 2025. CTXNL: A Software-Hardware Co-designed Solution for Efficient CXL-Based Transaction Processing. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 192–209. doi:10.1145/3676641.3716244

[64] Michael Wei, Amy Tai, Christopher J. Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, Michael J. Freedman, and Dahlia Malkhi. 2017. vCorfu: A Cloud-Scale Object Store on a Shared Log. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 35–49. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/wei-michael

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

[66] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S. Kim. 2023. Overcoming the Memory Wall with CXL-Enabled SSDs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 601–617. https://www.usenix.org/conference/atc23/presentation/yang-shao-peng

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

[68] Shushu Yi, Yuda An, Li Peng, Xiurui Pan, Qiao Li, Jieming Yin, Guangyan Zhang, Wenfei Wu, Chenxi Wang, Diyu Zhou, Zhenlin Wang, Xiaolin Wang, Yingwei Luo, Ke Zhou, and Jie Zhang. 2026. Espresso: Constructing Cost-Efficient CXL JBOF via Inter-SSD Computing Resource Sharing. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*. USENIX Association, Seattle, WA, 957–976. https://www.usenix.org/conference/osdi26/presentation/yi

[69] Seung Won Yoo, Joontaek Oh, Myeongin Cheon, Bonmoo Koo, Wonseb Jeong, Hyunsub Song, Hyeonho Song, Donghun Lee, and Youjip Won. 2025. DJFS: Directory-Granularity Filesystem Journaling for CMM-H

SSDs. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. USENIX Association, Santa Clara, CA, 35–51. https://www.usenix.org/conference/fast25/presentation/yoo

[70] Dongha Yoon, Hansen Idden, Jinshu Liu, Berkay Inceisci, Sam H. Noh, and Huaicheng Li. 2026. Cylon: Fast and Accurate Full-System Emulation of CXL-SSDs. In *24th USENIX Conference on File and Storage Technologies (FAST 26)*. USENIX Association, Santa Clara, CA, 313–327. https://www.usenix.org/conference/fast26/presentation/yoon

[71] Jianping Zeng, Shuyi Pei, Da Zhang, Yuchen Zhou, Amir Beygi, Xuebin Yao, Ramdas Kachare, Tong Zhang, Zongwang Li, Marie Nguyen, Rekha Pitchumani, Yang Soek Ki, and Changhee Jung. 2025. Performance Characterizations and Usage Guidelines of Samsung CMM-H. *IEEE Micro* 45, 5 (Sept. 2025), 94–102. doi:10.1109/MM.2025.3591577

[72] Jianping Zeng, Shuyi Pei, Da Zhang, Yuchen Zhou, Amir Beygi, Xuebin Yao, Ramdas Kachare, Tong Zhang, Zongwang Li, Marie Nguyen, Rekha Pitchumani, Yang Soek Ki, and Changhee Jung. 2025. Performance Characterizations and Usage Guidelines of Samsung CXL Memory Module Hybrid Prototype. arXiv:2503.22017 [cs.AR] doi:10.48550/arXiv.2503.22017 arXiv preprint, version 1.

[73] Yekang Zhan, Haichuan Hu, Xiangrui Yang, Shaohua Wang, Qiang Cao, Hong Jiang, and Jie Yao. 2024. RomeFS: A CXL-SSD Aware File System Exploiting Synergy of Memory-Block Dual Paths. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, New York, NY, USA, 720–736. doi:10.1145/3698038.3698539

[74] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Virtual Event, 1187–1204. https://www.usenix.org/conference/osdi20/presentation/zhang-haoran

[75] Haoyang Zhang, Yuqi Xue, Yirui Eric Zhou, Shaobo Li, and Jian Huang. 2025. SkyByte: Architecting an Efficient Memory-Semantic CXL-based SSD with OS and Hardware Co-design. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Las Vegas, NV, USA, 577–593. doi:10.1109/HPCA61900.2025.00051

[76] 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 Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 658–674. doi:10.1145/3600006.3613135

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

[78] Yuhong Zhong, Daniel S Berger, Pantea Zardoshti, Enrique Saurez, Jacob Nelson, Dan RK Ports, Antonis Psistakis, Joshua Fried, and Asaf Cidon. 2025. Oasis: Pooling PCIe Devices Over CXL to Boost Utilization. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Seoul, Republic of Korea) *(SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 101–119. doi:10.1145/3731569.3764812

[79] Yuhong Zhong, Daniel S Berger, Pantea Zardoshti, Enrique Saurez, Jacob Nelson, Antonis Psistakis, Joshua Fried, and Asaf Cidon. 2025. My CXL Pool Obviates Your PCIe Switch. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Banff, AB, Canada) *(HOTOS '25)*. Association for Computing Machinery, New York, NY, USA, 58–66. doi:10.1145/3713082.3730393

[80] Yuhong Zhong, Fiodar Kazhamiaka, Pantea Zardoshti, Shuwei Teng, Rodrigo Fonseca, Mark D. Hill, and Daniel S. Berger. 2026. Octopus: Enhancing CXL Memory Pods via Sparse Topology. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. USENIX Association, Renton, WA, 1303–1322. https://www.usenix.org/conference/nsdi26/presentation/zhong

[81] Yuchen Zhou, Jianping Zeng, and Changhee Jung. 2026. Anchoring Whole-System Persistence and Resilience in CXL. In *Proceedings of the 40th ACM International Conference on Supercomputing*. ACM, New York, NY, USA, 811–827. doi:10.1145/3797905.3800523

[82] Zhiting Zhu, Zhipeng Jia, Newton Ni, Dixin Tang, and Emmett Witchel. 2025. Impeller: Stream Processing on Shared Logs. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 637–653. doi:10.1145/3689031.3717485

[83] 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* (Austin, TX, USA) *(DIMES '24)*. Association for Computing Machinery, New York, NY, USA, 41–50. doi:10.1145/3698783.3699377