CXLALLOC: Safe and Efficient Memory Allocation for a CXL Pod

Newton Ni
nwtnni@cs.utexas.edu
The University of Texas at Austin
Austin, Texas, USA

Yan Sun
yans3@illinois.edu
University of Illinois Urbana-Champaign
Champaign, Illinois, USA

Zhiting Zhu*
zhitingz@nvidia.com
The University of Texas at Austin
Austin, Texas, USA
NVIDIA
Santa Clara, California, USA

Emmett Witchel
witchel@cs.utexas.edu
The University of Texas at Austin
Austin, Texas, USA

Abstract

A Compute Express Link (CXL) pod is a group of hosts that share CXL-attached memory. A memory allocator for a CXL pod faces novel challenges: (1) CXL devices may not fully support inter-host hardware cache coherence (HWcc), (2) the allocator may be concurrently accessed from different processes, and (3) with more hosts, failures become more likely.

We present CXLALLOC, a user-space memory allocator that addresses these challenges through careful metadata layout and new protocols to maintain cache coherence in software, coordinate memory mappings across processes, and recover from crashes. CXLALLOC uses compare-and-swap (CAS) for efficient synchronization; to support CXL devices with no HWcc, we present a memory-based CAS (mCAS) primitive implemented in an FPGA.

Experiments with in-memory key-value store workloads demonstrate that CXLALLOC retains competitive performance while enabling new use-cases. Experiments with a commercial CXL device show that CXLALLOC can achieve 80% of its maximum allocation throughput using mCAS.

CCS Concepts: • **Software and its engineering** → **Memory management**; • **Computer systems organization** → *Processors and memory architectures.*

Keywords: Memory allocation; CXL; shared memory; mCAS; cache coherence

*Work done while at The University of Texas at Austin.

ASPLOS '26, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790149

ACM Reference Format:
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 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3779212.3790149

1 Introduction

Compute Express Link (CXL) memory provides a load/store interface that allows processors to access memory across a CXL link, most commonly transported over PCIe. CXL is maturing into a practical substrate for lower-cost, disaggregated memory in data centers, where system software manages tiering between fast local memory and slower CXL-attached memory [45, 46, 50, 61, 65, 69]. Initial prototypes for hardware that allow multiple machines to share a single CXL memory device are now available, with plans for broader commercialization [5, 12]. A small number of hosts (e.g., 8–16) connected directly to a single multi-headed CXL memory module and sharing memory at cacheline granularity is called a CXL pod [12, 33, 71]. Applications that want to dynamically allocate and share memory in a CXL pod require a memory allocator.

CXL pods present some novel challenges that make it difficult to use existing memory allocators: **limited inter-host hardware cache coherence (HWcc)**, **cross-process sharing**, and **partial failure**. We contribute a new memory allocator, CXLALLOC, that address these challenges.

Limited HWcc. Version 3 of the CXL specification [2] defines an inter-host cache coherence protocol, but given implementation cost and complexity, it is unclear if HWcc will become widely supported in practice. HWcc is important for software, as it enables threads on different hosts to synchronize with atomic operations like compare-and-swap (CAS). Recent work assumes a range of HWcc models: full HWcc [9], HWcc in a limited memory region [33, 35], and no HWcc [56, 70].

CXLALLOC is compatible with all three of these HWcc models. CXLALLOC's algorithms reduce the amount of HWcc metadata while efficiently maintaining cache coherence in software for the remaining metadata. CXLALLOC's memory layout separates HWcc metadata into its own contiguous region. If there is full HWcc, CXLALLOC remains correct. If HWcc is limited to a small region (Figure 1(A)), CXLALLOC minimizes its HWcc usage. If there is no HWcc (Figure 1(B)), we demonstrate how to implement a memory-based compare-and-swap (mCAS) operation in near-memory processing logic, using hardware available today. CXLALLOC can use mCAS instead of CAS.

Cross-process sharing. Memory allocators typically assume that their metadata will be accessed within a single process, but shared CXL memory may be accessed concurrently from different processes on different hosts. To discuss correctness, it is useful to define two properties:

Definition 1.1 (Spatial pointer consistency (PC-S))**.** A pointer refers to the same physical memory in each sharing process.

Definition 1.2 (Temporal pointer consistency (PC-T))**.** A pointer to memory allocated in one process can immediately be dereferenced in any sharing process.

Together, we refer to these properties as pointer consistency (PC). PC must be maintained when memory allocators manipulate memory mappings—for example, to increase the size of the heap, or to back a new 1GiB allocation. In a single-process setting, PC is guaranteed by the OS: concurrent `mmap` calls return memory mappings that do not overlap and are immediately visible to all threads. With cross-process sharing, the OS can no longer provide these guarantees. For example, concurrent `mmap` calls in different processes may return memory mappings with the same virtual address, and each memory mapping is initially invisible to other processes.

CXLALLOC is the first memory allocator to provide PC for cross-process shared memory, without trade-offs like fixed heap size [1, 72] or maximum allocation size [68]. CXLALLOC provides PC-S by using offset pointers (§2.3) and placing heap metadata and data at consistent offsets in every process. CXLALLOC provides PC-T by using a signal handler to asynchronously install memory mappings in each process, and introduces a hazard pointer [51] based protocol to safely reclaim memory mappings.

Partial failure. Compared to a single-process application, a multi-process application in a CXL pod faces a higher probability of partial failure [28, 68], where a single thread or process may crash (e.g., due to the OS's out-of-memory killer). Tolerating partial failures is useful for high-availability applications [71] like transactional databases [33], machine learning training [48], and file systems [7], to ensure that a single software bug or other problem cannot bring down the whole system. Memory allocators should not cause live application threads to block, even if a thread crashes inside an allocator function.

Prior work [68] achieves partial failure tolerant memory allocation using lock-free data structures to ensure metadata shared between threads is always consistent, and reference counting to recover memory from dead threads. Reference counting works well for message passing workloads like RPC, which involve relatively few and uncontended reference count updates, but less so for shared memory data structures, where reference counts updates can cause high contention even under read-heavy access patterns. Moreover, reference counts (which require HWcc) are embedded in each allocation, making them non-trivial to adapt to other HWcc models.

CXLALLOC also uses lock-free data structures, but introduces a new recovery protocol to reduce overhead. Each allocator operation starts by updating 8 bytes of state atomically, in place, like a single-element redo log. This state provides enough information to idempotently redo an interrupted operation on recovery.

Contributions. We identify and explain the constraints of shared CXL memory allocation: limited HWcc, cross-process sharing, and partial failure. We present the design of CXLALLOC, the first memory allocator to satisfy all of these constraints, and an FPGA implementation of mCAS that can be used for inter-host synchronization for CXL devices that do not support HWcc.

We evaluate CXLALLOC against shared memory allocators, persistent memory allocators, and CXL memory allocators (§5). Using YCSB [20] and real-world traces of memcached requests [66], we establish that CXLALLOC has best-in-class performance for time and space. Using a commercial CXL device, we demonstrate that CXLALLOC can achieve up to 80% of its maximum allocation throughput with no HWcc (using mCAS). Our allocator is open source and available here: https://github.com/nwtnni/cxlalloc.

2 Background and motivation

We will discuss CXL hardware and memory allocation.

2.1 Compute Express Link (CXL)

CXL is a communication protocol that defines semantics for accessing memory across a serial link. CXL can run over PCIe links, allowing processors that support the CXL protocol (e.g., Intel's Sapphire Rapids) to access memory across the PCIe bus. The CXL memory device contains commodity DDR DRAM modules. CXL memory has evolved from a single device (1.0), to a switched pool (2.0), to fine-grained sharing (3.0, and in the current 3.2 specification [2]).

Cache coherence. The CXL standard supports inter-host hardware cache coherence (HWcc) via back-invalidation [2]. HWcc is important for shared memory programs because it allows threads to efficiently synchronize across hosts using standard atomic operations like compare-and-swap (CAS). Unfortunately, full HWcc seems unlikely due the cost of

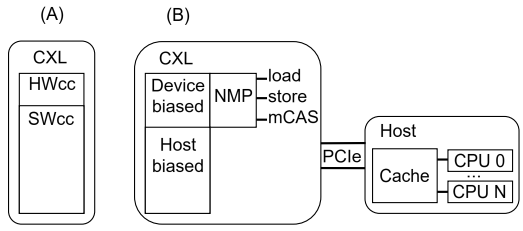


Figure 1. (A) shows shared CXL memory split into a section that is kept coherent by hardware (HWcc) and a much larger region that must rely on software for cache coherence (SWcc). **(B)** shows shared CXL memory connected to a host via a PCIe bus. One region of CXL memory is device biased, meaning its contents cannot be cached by a CPU. This region is fully managed by near memory processing logic (NMP), which handles load, store, and mCAS operations sent from the CPU to an address region marked uncachable. Host-biased CXL memory can be cached by the host.

snoop filters [35]. Two current hardware prototypes that support inter-host memory sharing do not support HWcc at all [5, 12]. Other work proposes limiting HWcc to a small memory region [35].

These two HWcc models are illustrated in Figure 1. Figure 1(A) shows a CXL device that supports HWcc in a small contiguous region (HWcc). Figure 1(B) shows a CXL device with no HWcc: we implement a custom memory-based compare-and-swap (mCAS) primitive using near memory processing (NMP) logic (§4). The NMP intercepts operations to a small contiguous region (device biased) to ensure that mCAS operations are serialized.

A critical difference between these HWcc models is that the HWcc region in Figure 1(A) can be cached by CPUs, but the device biased region in Figure 1(B) must be marked uncachable (via `/proc/mtrr`, for example), as the NMP has no way of tracking or invalidating CPU caches.

Failure model. Like related work [68, 71], we assume our CXL device is reliable. It keeps its state while processes can crash and operating systems can reboot. Such reliability can be achieved using an independent power supply or batteries.

2.2 Traditional memory allocation

We will start with some background, and then discuss why traditional memory allocators fail to support CXL shared memory.

Slab allocation. Slab allocation [41] is a common [16, 27, 43] design where memory is statically split into coarse-grained, fixed size slabs, which are then dynamically split into fine-grained, equally sized blocks. This balances fragmentation and performance.

Remote free. A remote free [43, 47] is when memory allocated by one thread is freed by another, which can result in false cache line sharing or even unbounded memory usage [13].

Limitations. It is non-trivial to adapt traditional memory allocators to limited HWcc because their layouts intersperse thread-local and global metadata. Leaving the global metadata in place wastes HWcc memory, but moving it out creates other problems, like correlating the thread-local and global metadata.

Cross-process sharing also makes managing memory mappings more challenging. Traditional memory allocators use absolute virtual addresses as pointers and assume a single-process address space. They rely on the OS to guarantee that new memory mappings do not overlap, so that a pointer refers to the same physical memory (PC-S). But new memory mappings in different processes may have overlapping virtual addresses, violating PC-S. They also rely on the OS to guarantee that memory mapping updates are immediately visible to all threads, so that pointers can be safely dereferenced (PC-T). But a new memory mapping in one process is invisible to other processes, so a pointer passed between processes may fault when dereferenced, violating PC-T.

Finally, traditional memory allocators do not provide failure tolerance. Many use locks for synchronization [27], which can block live threads if a thread crashes in a critical section, and none provide APIs to allow the application to recover the in-memory state of a crashed thread.

2.3 Persistent memory (PM) allocation

We again start with background, and then discuss why PM allocators fail to support CXL shared memory. PM is byte-addressable, like DRAM, but preserves its contents upon power loss. CXL memory is not persistent by default (though it can be made persistent using a backing SSD [22]), but data in shared CXL memory can survive application and even OS restarts, if shared by different hosts.

Recoverability. PM allocators can restore their internal metadata to a consistent state after a crash. There are broadly two approaches to recoverability: garbage collection [14, 16] scans the heap for memory leaks on recovery, while redo logging [23] replays log entries on recovery.

Offset pointers. Offset pointers [17] are a ubiquitous [1, 19, 23, 31, 52] alternative to traditional pointers (absolute virtual addresses) that allow in-memory data structures containing pointers to be mapped at different virtual addresses. Traditional pointers require memory mappings to always be placed at stable addresses, which can conflict with OS and application memory mappings. Offset pointers instead require stable offsets between memory mappings. Other alternatives, like indirection [15] and relocation [49], are possible, but require complex runtime support.

Limitations. Like traditional memory allocators (§2.2), PM allocators are difficult to adapt to limited HWcc: their layouts also intersperse thread-local and global metadata.

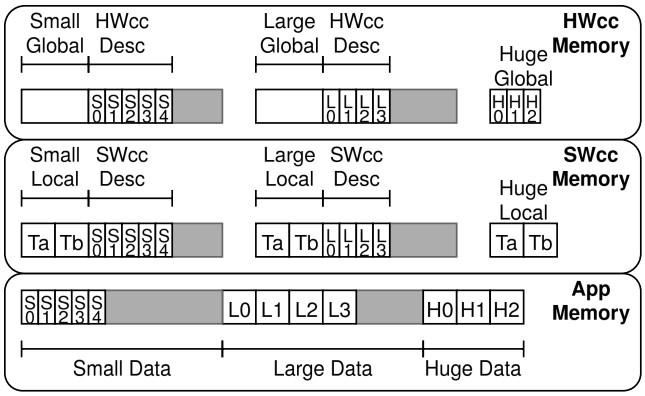


Figure 2. A sketch of CXLALLOC's memory layout. Heap metadata is partitioned into HWcc and SWcc metadata to support limited HWcc (§3.2), and separated from application data, which can reside in either HWcc or SWcc memory. Grey shaded regions indicate virtual address space reservations (§3.3), used to maintain pointer consistency (§1) across processes. S0 and L0 indicate slab metadata and data for a small or large slab, respectively, with index 0. H0 indicates a virtual address region in the huge heap (§3.1.2). Ta indicates thread-local metadata for thread a.

PM allocators assume sequential access from one process at a time, but cross-process sharing implies concurrent access. Even though PM allocators use offset pointers, they still rely on the OS to guarantee that concurrent memory mapping updates do not have overlapping offset ranges (PC-S). And even though PM allocators can replay memory mappings on recovery, they still rely on the OS to make mapping updates visible to all threads during normal execution (PC-T).

Partial failure also breaks some common design choices in PM allocators. PM allocators assume a total failure model, where all sharing threads crash at once, and there is a quiescent period during recovery where no thread is accessing the heap. As a result, many PM allocators synchronize their data structures using locks [23, 52], which can block live threads if a thread crashes in an allocator critical section. And many PM allocators recover their metadata by using non-concurrent garbage collection [14, 16], which blocks live threads from accessing the heap during recovery. Any blocking is undesirable for highly available applications.

3 Design

We begin by describing CXLALLOC 's core data layout and algorithms, and then explain how this core is extended to address limited HWcc, cross-process sharing, and partial failure.

3.1 Architecture

CXLALLOC comprises three heaps: the small, large, and huge heaps, which manage allocations of size 8B-1KiB, 1KiB-512KiB, and 512KiB+, respectively. Figure 2 sketches how these heaps are arranged in memory. We note two immediate differences in how CXLALLOC lays out metadata compared to a traditional memory allocator: (1) CXLALLOC partitions heap metadata into HWcc and SWcc (§3.2), and (2) data regions are contiguous in virtual address space to support offset pointers (§3.3). We next describe each heap in more detail.

```
struct SmallHeap {
    hwcc: (SmallGlobal, [HWccDesc]),
    swcc: ([SmallLocal; NUM_THREAD], [SWccDesc]),
    data: [u8] }
struct SmallGlobal { len: u32, free: u32 }
struct SmallLocal {
    unsized: u32, sized: [u32; NUM_SIZE_CLASS] }
struct HWccDesc { remote: u16 }
struct SWccDesc {
    next: u32, owner: ThreadId,
    class: u8, free: BitSet }
```

Figure 3. Pseudocode type definitions for the small heap.

3.1.1 Small and large heap.

The small and large heaps share the same slab allocation design (§2.2), so we omit a separate discussion of the large heap. Figure 3 lists the core data structures in the small heap.

At a high level, the data region is divided into fixed-size slabs. The heap length (`SmallGlobal.len`) indicates the current number of slabs in the heap, and can be increased (§3.3.1). Slabs are organized into free lists, and each slab is linked to at most one free list (we will explain how a slab may be unlinked during allocation). There are three kinds of free lists: the global free list (`SmallGlobal.free`) and thread-local unsized free lists (`SmallLocal.unsized`) contain inactive slabs, which have no size class and all memory available for allocation; the thread-local sized free lists (`SmallLocal.sized`) contain non-full slabs, which have a size class and at least one block available for allocation.

Each slab has some associated metadata, which is split across two descriptors: `SWccDesc` and `HWccDesc`. A slab has an owning thread (`SWccDesc.owner`) that has exclusive write access to the slab's `SWccDesc`, which includes the size class (`SWccDesc.class`), a bitset of available blocks (`SWccDesc.free`), and a slab index (`SWccDesc.next`) to link into free lists (implemented as intrusive linked lists). The owner of a slab is the only thread that can allocate blocks from that slab. To handle when a thread frees to a slab it does not own (remote free), each slab also has a counter of remote frees (`HWccDesc.remote`). This counter counts down from the total number of blocks due to our SWcc protocol (§3.2.2). Figure 4 shows the various state transitions that a slab undergoes during the (de)allocation algorithms described next.

Allocation. To allocate, a thread first checks its thread-local sized free list. If it is empty, the thread tries to transfer

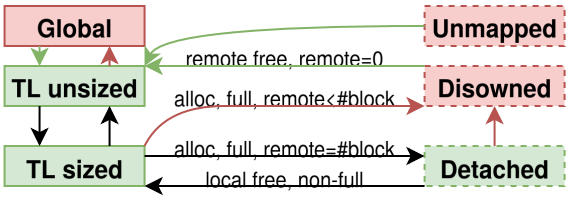


Figure 4. The state transition diagram of a slab in the small heap. Green and red indicate when a slab does or does not have an owner, respectively, and a dashed outline indicates that a slab is unlinked from all free lists. A global, thread-local (TL) unsized, or thread-local (TL) sized slab is linked to the corresponding free list. An unmapped slab is past the heap length. A detached slab is full, has an owner, and is unlinked, while a disowned slab is full, has no owner, and is unlinked (also see §3.2.1). The more complex transitions are labelled with conditions.

over a slab from the following sources, in order: the thread-local unsized free list, global free list, and heap length (i.e., extending the heap). After transferring a slab, the thread initializes the slab by setting `SWccDesc.owner` to its own ID, `SWccDesc.class` to the requested size class, `SWccDesc.free` to the full set of blocks, and `HWccDesc.remote` to the total number of blocks.

At this point, the thread-local sized free list must contain at least one slab, so the thread allocates a block from `SWccDesc.free`. If the slab is still not full, allocation is done. Otherwise, the thread must maintain our invariant that thread-local sized free lists only contain non-full slabs (which allows future allocations to avoid traversing full slabs, and remote frees to avoid coordination).

In the common case with no remote frees (i.e., `HWccDesc.remote` is equal to the total number of blocks), the thread keeps ownership of the slab, but still unlinks it from the thread-local sized free list, transitioning the slab to the detached state in Figure 4. Otherwise, at least one remote free has occurred, and the thread clears `SWccDesc.owner` before unlinking, transitioning the slab to the disowned state in Figure 4. We discuss the reasoning behind these two states in more detail (§3.2.1).

Deallocation. To deallocate, a thread identifies the slab containing the freed pointer (by dividing the pointer's offset within the data region by the slab size) and then checks the owner. If the deallocating thread is the owner, it takes the local free path and updates `SWccDesc.free` in place. If this slab was previously full, it must have been in the detached state, and the thread pushes it onto the thread-local sized free list. If this slab is now empty, the thread transfers it to the thread-local unsized free list.

Otherwise, the thread is not the owner, and it takes the remote free path and uses CAS to decrement `HWccDesc.remote`. If this counter reaches 0, this slab must have been detached or disowned, so the thread steals ownership of the slab, transferring it to its own thread-local unsized free list. Stealing is safe here because (1) a detached or disowned slab is not linked to any free list, and (2) if the counter is 0, then every block has been remotely freed, and there can be no more allocation from or deallocation to this slab. This code path is the only way that remotely freed memory can be reclaimed; we discuss tradeoffs in §3.2.1.

In either case, if the thread-local unsized free list reaches a configurable threshold length, the thread transfers some slabs to the global free list.

```
struct HugeHeap {
    hwcc: HugeGlobal,
    swcc: [HugeLocal; NUM_THREAD],
    data: [u8] }
struct HugeGlobal {
    reservations: [ThreadId; NUM_RESERVATION] }
struct HugeLocal {
    free: IntervalTree,
    descs: u64,
    hazards: [u64; NUM_HAZARD] }
struct HugeDesc {
   next: u64, offset: u64, size: u64, free: bool }
```

Figure 5. Pseudocode type definitions for the huge heap.

3.1.2 Huge heap. Allocations in the huge heap are backed by individual memory mappings, necessitating a different design than the small heap. The main data structures are listed in Figure 5. At a high level, the reservation array (`HugeGlobal.reservations`) tracks ownership of coarse-grained virtual address regions; an entry grants a thread exclusive permission to install new mappings in the corresponding region. Each thread tracks its owned regions using an interval tree (`HugeLocal.free`). We note that any deterministic data structure will work here. Whenever a thread creates a memory mapping to back an allocation, it also allocates a new descriptor (`HugeDesc`) and links the descriptor to an intrusive linked list (`HugeLocal.descs`). Descriptors record the allocation's offset (relative to `HugeHeap.data`) and size. The free bit (`HugeDesc.free`) is used in tandem with hazard offsets (`HugeLocal.hazards`) to safely reclaim memory in a cross-process setting.

Allocation. To allocate, a thread finds a contiguous region of the requested size using `HugeLocal.free` (requesting more virtual address space from the reservation array if necessary). The thread then allocates a descriptor, initializes the descriptor's size and offset with free bit unset, and links the descriptor to its descriptor list. Finally, the thread installs the mapping and returns the resulting pointer to the application.

Deallocation. To deallocate, a thread computes the virtual address region containing the freed pointer (by subtracting `HugeHeap.data` from the pointer and dividing by the

region size), and then looks up the owner in the reservation array. The thread then traverses the owner's descriptor list to find the descriptor with the same offset, and sets the `HugeDesc.free` bit. Setting the free bit does not require CAS because huge descriptors are never updated concurrently. Finally, the thread unmaps this memory mapping.

3.2 Limited HWcc

CXLALLOC supports limited HWcc by minimizing and separating metadata that requires HWcc (`SmallHeap.hwcc` in Figure 3 and `HugeHeap.hwcc` in Figure 5). For the small and large heaps, CXLALLOC uses only 2B of HWcc memory (`HWccDesc`) per slab—a small slab is 32KiB and a large slab is 512KiB—with 8B constant overhead (`SmallGlobal`). The huge heap uses a constant amount of HWcc memory (`HugeGlobal`), which is 8KiB in our prototype. We will first discuss CXLALLOC's small heap remote free protocol, as metadata to manage remote frees is the only HWcc metadata that scales with the size of the heap, and then explain CXLALLOC's SWcc protocol.

3.2.1 Remote free. CXLALLOC's slab allocation design is similar to mimalloc [43] in that (1) each slab has its own free bitset to decrease contention and improve spatial locality, and (2) each slab has separate local and remote free metadata, allowing local frees to take a fast unsychronized path, while only remote frees need to synchronize via CAS for platforms with HWcc, mCAS otherwise.

CXLALLOC introduces two major changes for remote frees. The first change is the detached state (Figure 4), which allows a slab that is entirely remotely freed (e.g., in a producer-consumer workload) to be stolen by a thread without coordinating with the slab's previous owner. The second change is using a counter to track remote frees, which minimizes HWcc memory overhead compared to, say, a bitset or intrusive free list. However, a counter loses information about which specific blocks have been freed, which prevents remotely freed blocks from being reused until the entire slab has been remotely freed. To ensure that slabs with a mix of local and remote frees are eventually reclaimed, CXLALLOC introduces the disowned state (Figure 4); any slab that has at least one remote free and is being actively allocated from will be disowned instead of detached, forcing all subsequent frees to take the remote free path (§3.1.1.Deallocation) and allowing the whole slab to be reclaimed.

There are pathological cases for our remote free protocol, but these cases require a thread to allocate many slabs, locally free a few blocks in each slab (while the rest are freed remotely), and then stop allocating from those size classes. We do not expect this pattern to be common in normal workloads, and our evaluation does not show excessive fragmentation.

3.2.2 SWcc protocol. CXLALLOC assumes that SWcc CXL memory does not have hardware inter-host cache coherence, but does let hosts keep state in their CPU caches. CXLALLOC also assumes that any application written for SWcc memory will pin threads to cores, to avoid inconsistent cache contents due to the OS scheduling a thread on different core. CXLALLOC manually controls cache state by flushing and fencing.

We begin with the small heap. The two sources of SWcc data are thread-local free lists (`SmallLocal`) and `SWccDesc`. Thread-local free lists are only read and written by a single thread, and trivially do not require any flushing or fencing. `SWccDescs` require more care: while they are only written by their owner, ownership can change, and `SWccDescs` can also be read by many threads.

For `SWccDesc` writers, we observe that flushing and fencing is necessary only when ownership may change (see Figure 4); the owner may otherwise keep `SWccDesc` in cache. For example, a thread must flush and fence a `SWccDesc` before transferring the slab from the thread-local unsized free list to the global free list. More subtly, a flush and fence is required before a slab transitions to detached or disowned states, since ownership may change due to remote frees.

For `SWccDesc` readers, there are two locations where a non-owning thread can read a `SWccDesc`. The first is pushing and popping from the global free list, which reads `SWccDesc.next`. Since global free list operations are rare, readers simply flush and fence before each load. The value of `SWccDesc.next` cannot change without popping the slab from the global free list, so a stale load will be detected by a CAS (or mCAS) conflict on `SmallGlobal.free`. The second location is freeing, which starts by loading `SWccDesc.owner`. Crucially for performance, it is safe for a thread to cache `SWccDesc.owner`; no flush or fence is required. To understand why, we do case analysis on the value of `SWccDesc.owner` in thread t's cache and in memory.

1. Owner is t in cache and memory. This is the common case of a thread freeing to a slab it owns, which is safe.
2. Owner is t in memory, but not in cache. Same as (1).
3. Owner is t in cache, but not in memory. This is impossible, since it implies t gave up ownership of this slab without flushing and fencing its `SWccDesc`.
4. Owner is not t in cache or memory. In this case, the thread performs a remote free by CASing (or mCASing) to decrement `HWccDesc.remote`. Importantly, this decrement is correct even if the cached owner is inconsistent with memory (e.g., if the `SWccDesc` is transferred between two other threads while this thread holds a cached copy of `SWccDesc.owner`). This is possible because `HWccDesc.remote` counts down to 0 instead of up to the block count, so remote free logic does not depend on the potentially inconsistent value of `SWccDesc.class` (which is in the same cache line as `SWccDesc.owner`).

For the huge heap, performance is less critical, so we simply treat all SWcc data (`HugeLocal` and `HugeDesc`) as uncachable, and flush and fence after every write and before

every read. This does not cause any data races, because `HugeLocal` and `HugeDesc` are never concurrently updated.

3.3 Cross-process sharing

CXLALLOC provides pointer consistency (PC, §1) across processes by coordinating the location, installation, and removal of memory mappings. We first introduce two basic mechanisms that are used by both the small and huge heap, and then explain how each heap maintains PC.

Virtual address space reservation. In order to provide PC-S for offset pointers, CXLALLOC must create memory mappings at exactly the same offset in each process, which requires calling `mmap` with the `MAP_FIXED` flag. To avoid overwriting existing memory mappings, CXLALLOC reserves large contiguous regions of virtual address space during heap initialization, in each process, by calling `mmap` with the `PROT_NONE` flag. The absolute address of a reservation does not matter, and the OS may choose different addresses for each process. What matters is that a reservation gives CXLALLOC a contiguous range of offsets where it can its manage its own memory mappings. Reservations are visible as the gray regions in Figure 2.

Signal handler. In order to provide PC-T, CXLALLOC must ensure that new memory mappings in one process are made visible to sharing processes. For example, if a thread in one process requests a huge allocation (backed by a new memory mapping), and writes a pointer to this new memory mapping into a shared data structure, a thread in a different process should be able to dereference the pointer without crashing.

CXLALLOC updates memory mappings asynchronously by installing a signal handler in each process that intercepts `SIGSEGV` signals when a thread dereferences an unmapped pointer. A SIGSEGV might be a program bug or it might be a thread trying to access a region that has been mapped by CXLALLOC in some processes, but not in the current one. The signal handler inspects heap metadata to determine if the pointer is within the heap and if it should be backed by a valid memory mapping. If so, the signal handler installs the memory mapping for the current process and reissues the faulting instruction; if not, the signal is forwarded to the default signal handler.

We considered and rejected a synchronous design, where processes participate in a barrier when updating memory mappings. A barrier would (a) introduce global overhead for memory mappings that are only accessed by a subset of processes, (b) prevent concurrent updates of memory mappings, and (c) block live threads under partial failure.

3.3.1 Small heap. CXLALLOC extends the small heap by atomically increasing the heap length (`SmallGlobal.len`), which requires creating three new memory mappings: one for each of the `HWccDesc`, `SWccDesc`, and `SmallHeap.data` regions. To provide PC-S, the small heap reserves virtual address space for each of these regions at initialization time, ensuring they have room to extend, and then places new memory mappings linearly within the reservations (Figure 2). There can be no overlapping memory mappings because the heap length is changed atomically. To provide PC-T, CXLALLOC's signal handler checks the heap length to see if a pointer is within the heap.

We simplify heap extension by having it be monotonic—CXLALLOC never unmaps small heap memory mappings. The underlying memory can be returned to the OS by calling `MADV_REMOVE` (or any equivalent mechanism) when transferring a slab to the global free list.

3.3.2 Huge heap. The huge heap must create new memory mappings to back allocations. To provide PC-S, the reservation array ensures that each thread creates memory mappings in disjoint regions. To provide PC-T, CXLALLOC's signal handler walks huge descriptor lists to see if a pointer is within a huge allocation. However, unlike the small heap, memory mappings can be unmapped when a huge allocation is freed. Frees are challenging in a cross-process setting because memory must be unmapped in *all* processes when a huge allocation is freed in *any* process; only then are its resources (physical memory, huge descriptor, virtual address region) safe to reclaim and reuse. CXLALLOC introduces a hazard offset protocol to determine when reclamation is safe.

Hazard offsets. Hazard offsets are a variant of hazard pointers [51], which are used for safe memory reclamation in lock-free data structures. Hazard pointers work roughly as follows: before dereferencing a pointer, a thread publishes the pointer to a globally readable list, which prevents this pointer from being reclaimed while the thread is accessing the memory. In CXLALLOC, before installing a memory mapping, a thread publishes the offset to a globally readable list, which prevents the memory mapping from being reclaimed while a process has the memory mapped. Our protocol follows three simple rules:

- Publish hazard offset before mapping a huge allocation.
- Remove hazard offset after unmapping a huge allocation.
- Reclaim huge allocation if `HugeDesc.free` is set and `HugeDesc.offset` is not published in any hazard offset list.

Together, these imply that a huge allocation will be reclaimed if it has been freed and no process has this allocation mapped. To implement these rules, we update allocation and CXLALLOC's signal handler to publish hazard offsets, and deallocation to remove hazard offsets. We do not expect the length of the huge allocation hazard list to pose a performance issue because huge allocations are relatively rare and long-lived.

CXLALLOC must clean up unused memory mappings and huge descriptors: a memory mapping can be unmapped and its hazard offset removed if the corresponding huge descriptor's free bit is set. A huge descriptor can be reclaimed if its

free bit is set and its offset is not published in any hazard offset list. CXLALLOC cleans up asynchronously by having each thread occasionally walk its hazard offset list and huge descriptor list.

Finally, we point out one subtlety: hazard pointers require a validation step after publishing a hazard pointer, to make sure the pointer wasn't freed in between loading the pointer and publishing it as a hazard. The equivalent race condition for hazard offsets would require one thread to dereference a huge allocation (to publish a hazard offset) while another thread frees the huge allocation, which is a use after free violation and can be ruled out for correct programs. Accordingly, hazard offsets do not require this validation step.

3.4 Partial failure

CXLALLOC avoids blocking live threads during crashes by using lock-free data structures, and recovers without blocking using a combination of detectable CAS [10] and atomic state changes.

3.4.1 Non-blocking crashes. Single-writer, single-reader data structures, like thread-local sized free lists, may be inconsistent after a crash. These transient inconsistencies are not a problem, because these data structures are not visible to other threads. They can be repaired by the recovered thread and will not block other live threads. Single-writer, multiple-reader data structures, like hazard offsets or huge descriptors, are updated through atomic writes and are always consistent. Multiple-writer, multiple-reader data structures, of which there are four—heap length, global free list, `HwccDesc`, and reservation array—are lock-free; furthermore, every operation on these data structures requires only a single CAS, making them much easier to reason about. Since lock-free data structures transition atomically between consistent states, CXLALLOC remains available even in the presence of thread crashes.

3.4.2 Non-blocking recovery. To recover without having to scan the heap for memory leaks [14, 16], each thread atomically updates 8 bytes of state in place, which records which operation the thread is currently performing, and contains enough information to recover the operation in an idempotent manner.

Small heap. For the small heap, each operation roughly corresponds to a state transition in Figure 4. For example, before transferring a slab from the thread-local unsized free list to the thread-local sized free list, a thread records the operation ID (4 bits), slab index (32 bits), and size class (8 bits). On recovery, the thread ensures this slab has been popped from the thread-local unsized free list and pushed onto the correct thread-local sized free list.

For operations involving lock-free data structures, like transferring a slab from the global free list to the thread-local unsized free list, we use detectable CAS [10] as a primitive to help implement idempotence. In short, detectable CAS allows a thread to attach a version to a CAS operation; upon recovery, the thread can query a global help array to see if its operation succeeded, i.e., became visible to another thread. For example, before popping from the global free list, a thread records the operation ID (4 bits), the slab index (32 bits) to pop, and a version (16 bits). On recovery, the thread checks if its CAS succeeded. If the CAS did not succeed, the thread retries the CAS; otherwise, the thread ensures that the popped slab index has been pushed to the thread-local unsized free list.

Detectable CAS requires embedding a thread ID and logical version in each CAS target: our CAS targets are at most 32 bits, so we use a 16-bit thread ID and version to support systems with only 8-byte CAS. This strategy increases our HWcc (or mCAS) overhead for remote free metadata from 2B to 6B (8B aligned) per slab (§3.2)

Huge heap. For the huge heap, on recovery, a thread can deterministically reconstruct its thread-local allocation state (`HugeLocal.free`) from the reservation array and its huge descriptor list. Since the huge heap is much simpler than the small heap, and its data structures almost all have a single writer, recording the huge descriptor offset (the offset of the huge descriptor itself, not the offset of its memory mapping) is sufficient to recover its operations.

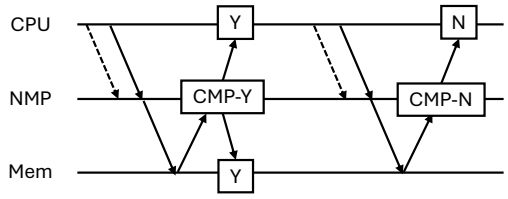
4 Implementation

Prototyping mCAS. We design customized near-memory processing (NMP) logic in the FPGA of Intel's Agilex 7 board [21] to provide an mCAS operation for architectures that do not support inter-host HWcc (Figure 1(B)).

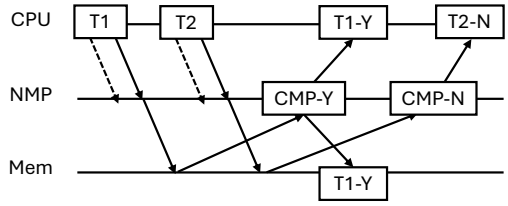
We partition the CXL physical address space into two regions: device-biased and host-biased [2]. The NMP unit is positioned between the *CXL interface (IP)* and the *CXL memory controller*, and it manages the device-biased memory. All load, store, and mCAS requests for the device-biased memory go through the NMP unit (see Figure 1(B)). Only memory within this device-biased region can be mCASed and because the memory is device-biased, it must never be cached by a CPU [2]. These mCAS restrictions create barriers to porting software to make use of mCAS. Any memory location that might be used in an mCAS should be sequestered from other data structures, to minimize the amount of memory that needs to be marked uncachable.

Our mCAS interface avoids MMIO to reduce latency. Instead, we reserve two address ranges for threads to interact with the NMP: the special write (`spwr`) region and special read (`sprd`) region. Each thread accesses different cache lines within these regions according to its thread ID.

To initiate an mCAS, a thread writes 64B containing the expected value, swap value, and target address to the `spwr` region. To retrieve the response, a thread reads 16B from the `sprd` region, which returns a success or failure bit and the previous value at the target address. To ensure these



(a) mCAS returning success (Y) upon a matching value (CMP-Y) on the left and failure (N) upon a mismatch (CMP-N) on the right. The dashed line indicates the `spwr` while the solid line indicates the `sprd`.



(b) Thread 1 (T1) issues `spwr-sprd` pair before thread 2 (T2) to the same target address. T1 succeeds (T1-Y) in the comparison and blocks T2's operation (CMP-N), resulting in T2 failing its mCAS operation (T2-N).

Figure 6. mCAS timing diagram

writes and reads reach the NMP, we mark the `spwr` and `sprd` regions uncachable.

On the NMP side, upon receiving a `spwr`, the NMP unit stores the operands in its internal register array and waits for a `sprd` to trigger the mCAS operation. When the NMP unit receives a `sprd`, it reads the target address and compares the value with the expected value. At the end of each `sprd`, the NMP unit checks its register array to see if any other `spwr` or `sprd` is in progress and has a matching target address. If there is a match, the NMP unit will return mCAS failure for the competing `spwr` or `sprd`. On an mCAS success, all subsequent `sprd` and `spwr` operations are stalled until the swap value is written to the memory. These checks ensure that for a given address, only one `spwr-sprd` pair can be in progress at a time. Figure 6 shows the timing diagram of the mCAS operation.

Heap initialization. Most allocators require some single-thread initialization of the heap [1, 16, 43, 68] which creates a bootstrapping problem for cross-process applications. Some external coordination becomes necessary to allow one process to initialize the heap before any other processes can access it.

CXLALLOC is carefully constructed so that zeroed memory constitutes a valid and initialized heap. Processes do not need to coordinate to initialize a shared heap.

5 Evaluation

Our design seeks to answer the following questions:

- Is CXLALLOC correct (§5.1)?
- How does CXLALLOC affect performance of end-to-end key-value store workloads (§5.2.1)?
- How does CXLALLOC perform on low contention and high contention microbenchmarks (§5.2.2)?
- How scalable are huge allocations (§5.3)?
- How is performance affected by HWcc support (§5.4)?

We evaluate on two machines: a Chameleon [39] instance that does not have a CXL device, but does have 80 cores to better compare scalability, and another machine with a CXL Type-2 device, 32 cores, and our mCAS prototype to measure the effect of HWcc.

Allocator	Mem.	XP	mmap	Fail	Rec.	Str.
mimalloc [43]	M	×	✓	NB	×	×
boost [1]	XP	✓	×	B	×	×
lightning [72]	XP	✓	×	B	B	GC
cxl-shm [68]	CXL	✓	×	NB	NB	GC
ralloc [16]	PM	×	×	NB	B	App
CXLALLOC	XP, CXL	✓	✓	NB	NB	App

Table 1. Properties of memory allocators in our evaluation. **Mem.** shows what kind of memory the allocator was designed to manage (M is "normal", volatile, in-process memory, XP is cross-process memory, CXL is compute express link memory, and PM is persistent memory). **XP** means supports cross-process allocations by using pointer alternatives (§2.3). **mmap** means allocator can use `mmap` for large allocations or to extend the heap. **Fail** means behavior on failure (blocking (B), non-blocking (NB)) **Rec.** means behavior on recovery (blocking (B), non-blocking (NB), or not recoverable(×)) **Str.** is the recovery strategy (garbage collect allocations from dead threads (GC), allow application to recover (App), or not recoverable (×)).

Baselines. We choose our baselines (summarized in Table 1) for the following reasons: mimalloc [43] is a state-of-the-art traditional memory allocator that provides the best performance for most allocation benchmarks [4]. Boost [1] is an industry C++ library and one of the only explicit cross-process shared memory allocators we found. Lightning [72] is a shared-memory key-value store, one of our motivating use-cases. We extract its internal, cross-process memory allocator. Ralloc [16] is a lock-free recoverable allocator for PM. Finally, cxl-shm [68] is the state-of-the-art partial fault tolerant memory management system for CXL shared memory.

None of our baselines optimize for limited HWcc. All allocators (except mimalloc) support pointer consistency for cross-process shared memory, but only trivially, because they do not allow heap extension or huge allocation, and do not update memory mappings. Mimalloc does not support cross-process sharing at all, but serves as an indicator of maximum allocator performance.

Experimental setup. All benchmarks are run for 10 trials, and include error bars for standard deviation (these are often too close to be visible because of low performance variability). All CPU performance governors are set to performance, and the NMI watchdog, NUMA balancing, KSM, turbo boost, and hyperthreading disabled. Threads are always pinned to a core. We configure all benchmarks to perform fixed amounts of work as the thread count varies; we choose amounts that (a) can be divided evenly across all thread counts and (b) run long enough for throughput to be stable across trials. Each memory allocator is backed by a 64 GiB shared memory file. We configure ralloc and cxl-shm to remove flushing and fencing.

5.1 Correctness

We compile CXLALLOC with a host of runtime invariant checks, for example: `SWccDesc.owner` is null when popping a slab from the global free list, all slabs in thread-local sized free lists are non-full, all free lists are acyclic. We run all of our benchmarks with these checks enabled and observe no errors. We evaluate the correctness of recovery using a mix of black-box tests with random thread crashes, and white-box tests with defined thread crash points, again with invariant checks enabled.

5.2 Performance

We run our first set of benchmarks on a Chameleon [39] compute_icelake_r650 instance running Ubuntu 22.04.5 LTS and Linux kernel version 5.15. It has two Intel Xeon Platinum 8380 CPUs running at 2.30GHz, with 40 cores, 120 MiB LLC, and 128GiB DDR4 3200 DRAM per socket. We bind all memory to NUMA node 0 because NUMA-awareness is not a stated goal of any of the benchmarked systems, and to avoid introducing bias from NUMA interleaving correlating with allocator data structure layout.

CXLALLOC is a multi-process allocator, and we want to test cross-process allocation. (All allocators are cross-process except mimalloc.) Our understanding is that most multi-process applications are also multi-threaded, but how many threads should run per process? We experimentally verified that performance of the allocators generally decreases with increasing process counts, though there was no universal trend. We choose to run cross-process allocators in 10 processes because it provides good performance (relative to, say 2 processes or 80), and it allows us to vary the number of threads per process from 1 to 8. We report the total proportional set size (PSS) across all processes to directly compare the cross-process allocators with mimalloc.

5.2.1 YCSB and memcached traces. We benchmark a key-value store by port YCSB [20] and running production traces from Twitter memcached clusters [66], which are summarized in Table 2. The throughput and sum of the proportional set size (PSS) for each workload are reported in Figure 8. For our index data structure, we adapt cxl-shm's non-resizable lock-free hash table to support all allocators, configuring it with 32M buckets. In order to support deletion, we also adapt it to use token-passing epoch-based reclamation [40]. Because we are comparing the impact of the underlying allocator, and not the index data structure, we omit workloads that do not involve allocation (e.g., most read-biased YCSB workload mixes).

Workload	Ins. %	Key Distr.	Key Size	Value Size
YCSB-Load	100	Uniform	8B	960B
YCSB-A	25	Skew	8B	960B
YCSB-D	5	Skew	8B	960B
MC-12	79.7	Uniform	44 B	0-307 KiB
MC-15	99.9	Uniform	14-19 B	0-144 B
MC-31	93.0	Uniform	40-46 B	0-15 B
MC-37	38.8	Skew	68-82 B	0-325 KiB

Table 2. Summary statistics for in-memory key-value store workloads. Ins. % is the percentage of operations that insert data (causing an allocation). We configure YCSB with the default Zipfian constant of 0.99, and modify YCSB-A from 50% update to 25% insert and 25% delete operations to stress the memory allocator. All other workloads consist entirely of read and insert operations.

We configure YCSB with the default Zipfian constant of 0.99, 8 byte keys, and 960 byte values, with an initial index size of 8.4M key-value pairs for YCSB-A and YCSB-D (0 for YCSB-Load), and run each workload for 8.4M total operations.

For memcached, we use the original paper's representative clusters for write-heavy workloads, and execute 8.4M operations from each trace (840K for MC-37, which requires more memory). Each trace consists solely of read and insert operations.

General analysis. Figure 8 shows that Boost and Lightning are fundamentally unscalable, as they both acquire a global mutex. Lightning's PSS usage is not included in the figure, because it uses a large array to track each individual allocation for garbage collection, and requires an order of magnitude more memory.

Cxl-shm's performance suffers on skewed workloads like YCSB-A and YCSB-D because its reference counting creates additional contention on hot items, even though YCSB-D is read-heavy. Cxl-shm also requires 24B of inline header metadata for each allocation, which causes noticeable overhead in workloads with small allocations like MC-15 and MC-31.

Mimalloc, ralloc, and cxlalloc generally perform similarly. Across all workloads and thread counts, cxlalloc achieves 93.9% of mimalloc's performance on average, while ralloc achieves 90.9%.

HWcc memory. Besides CXLALLOC, ralloc is the only baseline that separates heap metadata from data. It can naively

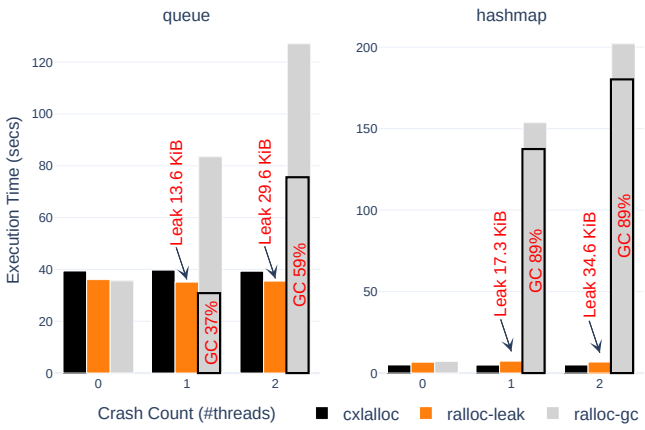


Figure 7. Execution time of inserting and removing 1M objects from Memento [18] recoverable data structures under 0, 1, or 2 thread crashes. This experiment demonstrates how PM allocators that recover using garbage collection, like ralloc, must choose to block heap access to run GC (ralloc-gc) or leak memory (ralloc-leak). CXLALLOC recovers without leaking or blocking.

support limited HWcc by placing only its metadata in the HWcc region, rather than the entire heap, so we will use it as a reference point for our HWcc optimizations. Across all workloads and thread counts, CXLALLOC uses only 0.02% of HWcc memory relative to total memory usage on average, and 7.1% relative to ralloc's HWcc memory usage.

Partial failure. We evaluate the overhead of CXLALLOC's partial failure tolerance by comparing a variant of CXLALLOC (CXLALLOC-nonrecoverable) that disables recovery state updates and uses a normal CAS instead of a detectable CAS. Across all workloads and thread counts, CXLALLOC is only 0.3% slower than CXLALLOC-nonrecoverable.

We also show some simple experiments with a recoverable queue and hash table from Memento [18]. We insert 1M objects with sizes chosen uniformly randomly between 8B-1KiB into each data structure, and then remove them, crashing 0, 1, or 2 threads during the insertion phase. After a crash, ralloc has to block heap access to run recovery garbage collection, or else leak memory; CXLALLOC recovers without leaking or blocking.

5.2.2 Allocator microbenchmarks.

We next evaluate two microbenchmarks in Figure 9: thread-test and xmalloc. Thread-test estimates the highest possible allocator throughput using a fixed allocation size and entirely thread-local operations. Xmalloc is a producer-consumer workload that stresses the remote free code path, which requires synchronization.

General analysis. Threadtest reveals mimalloc's highly optimized fast path, which uses an intrusive linked list. CXLALLOC achieves only 47% of mimalloc's throughput on average, while ralloc achieves 41%. Xmalloc shows that mimalloc, ralloc, and CXLALLOC's designs are effective at reducing contention for remote free operations. CXLALLOC achieves 81% of mimalloc's throughput, while ralloc achieves 106%. Ralloc falls off at higher thread counts because it returns partially full slabs to the global free list, which introduces contention.

HWcc memory. Overall memory usage is low for these benchmarks, so CXLALLOC's HWcc optimizations are less effective. CXLALLOC still only requires 2.5% and 0.09% HWcc memory relative to total memory usage for thread-test and xmalloc, respectively, which is 9.4% and 9.5% of ralloc's HWcc memory usage. Xmalloc shows how CXLALLOC's split HWcc and SWcc metadata design reduces HWcc usage, but can cause increased total memory usage.

Partial failure. CXLALLOC achieves 94.7% of CXLALLOC-nonrecoverable's throughput for thread-test, which shows the low overhead of CXLALLOC's recovery logic in the fast path. And CXLALLOC achieves 88.4% of CXLALLOC-nonrecoverable's throughput for xmalloc, which shows the cost of using detectable CAS to perform remote frees.

5.3 Huge allocations

CXLALLOC's support for cross-process huge allocations (§3.3.2) is a novel feature. We evaluate its performance by configuring both threadtest and xmalloc with a fixed object size of 1GiB, and run them for 9.6M total operations, with results shown in Figure 10. There are no baselines because every other allocator crashes or does not complete within 30 minutes.

Threadtest performance is pretty flat with increasing thread count within one NUMA node indicating that the scalability limit is from the OS memory mapping work, which shows greater throughput as work is split across more processes. Xmalloc stresses remote frees, and for larger process counts, as we increase the number of threads, performance increases, because there is enough OS level parallelism. For low process counts, increasing the number of threads slightly decreases performance as the workload bottlenecks on OS maintainance of a small number of process address spaces. Going to multiple NUMA nodes (80 threads) decreases performance as threads access remote memory.

Memory consumption in all cases is modest because these are allocator microbenchmarks and they do not access all of the allocated data. Therefore memory consumption is proportional to the size of allocator metadata.

While thread-test does not cause cross-process faults, xmalloc does exercise both CXLALLOC's cross-process faults and hazard offset operations, and demonstrates its efficient reuse of huge descriptors and address space. Overall, these results validate our huge allocation design (§3.3.2) by showing stable performance even for a punishingly unrealistic workload that unnaturally stresses huge allocations.

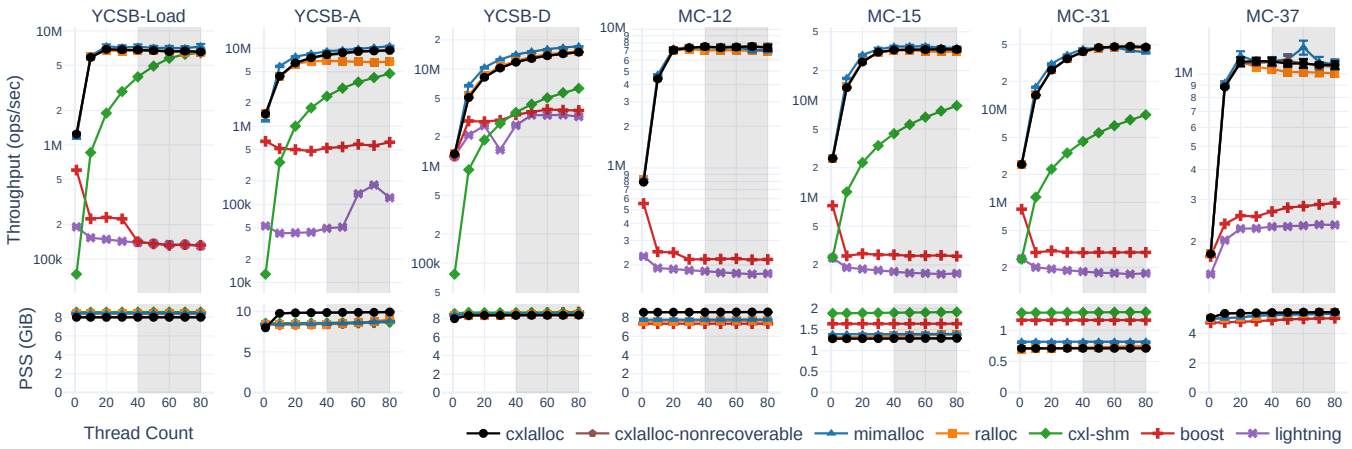


Figure 8. Throughput (logarithmic Y-axis) and memory consumption for different memory allocators running in-memory key-value store workloads from YCSB and Twitter memcached traces [66]. Experiments up to and including 40 threads run on a single socket in a single NUMA node. Experiments with more than 40 threads (shaded in gray) run on two sockets using two NUMA nodes. Cxl-shm crashes for workloads MC-12 and MC-37 because it does not support allocations larger than 1KiB.

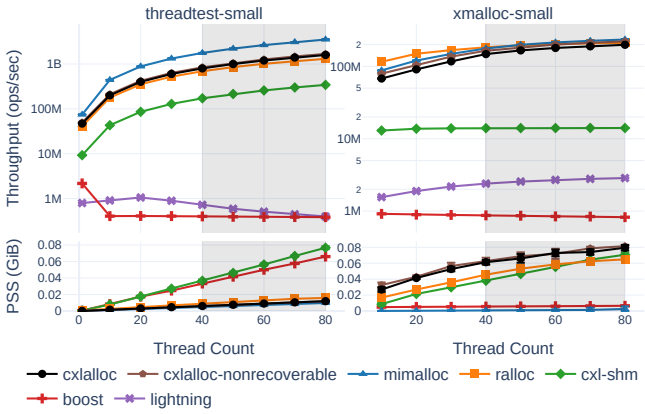


Figure 9. Throughput (logarithmic Y-axis) and memory consumption for small heap allocation microbenchmarks across all allocators with increasing numbers of threads distributed among 10 processes, run on the Intel ICX machine. Cxlalloc, mimalloc, and ralloc are the highest performing options.

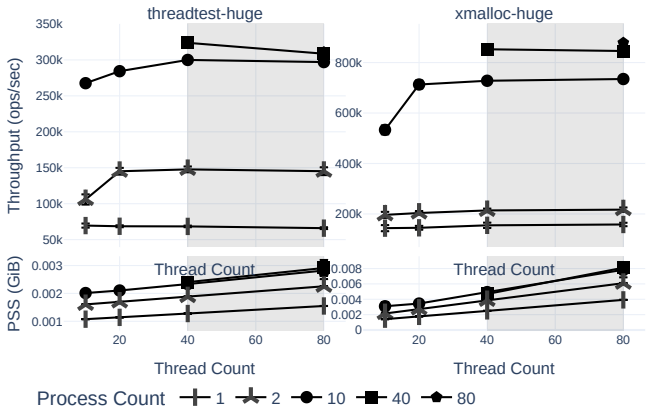


Figure 10. Throughput and memory consumption for huge allocation microbenchmarks as the number of threads are increased for different numbers of processes, run on the Intel ICX machine. Performance (and memory consumption) improves monotonically for increasing process counts. Note that there is only a single data point for 80 threads in 80 processes.

5.4 CXL memory hardware

Our test machine runs Ubuntu 24.04.2 LTS, Linux kernel version 6.8. It has an Intel Xeon 8568 CPU running at 2.0GHz, with 48 cores and 300MB LLC. The machine is equipped with 8-channel DDR5 4800 DRAM. We use a commercially available CXL Type-2 device, the Intel Agilex 7 [21], connected to the CPU via a PCIe 5.0 x16 link. It integrates an FPGA with ASIC-based CXL IPs. The memory controller is an ASIC, the FPGA is only for implementing near memory processing logic (NMP). While the device supports CXL 2.0, our Intel Emerald Rapids CPU only supports 1.1 [37, 58].

We measure the latency and bandwidth values of local DRAM and CXL memory using Intel's Memory Latency Checker (MLC) [34] with a 3:1 read-write ratio. CXL read latency is 357ns, compared with 112ns for local memory. Its bandwidth is 19.9 GB/s (using two channels), compared with 114 GB/s for local memory (using four channels). The CXL memory access latency and bandwidth are in line with the latest study on characterizing commercial CXL memory devices [37]. We disable hyper-threading and turbo boost and set the CPU frequency governor to performance.

5.4.1 CXL: mCAS prototyping. Figure 11 shows the latency of a compare-and-swap (CAS) operation on a CXL

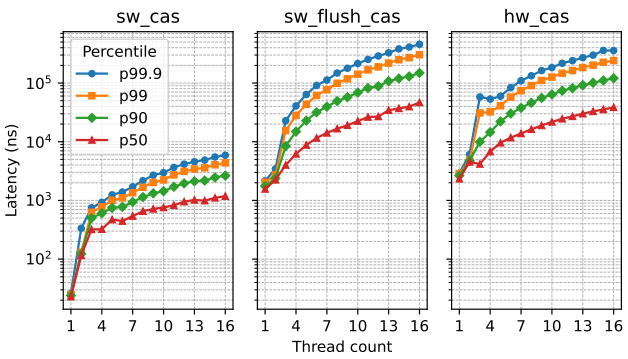


Figure 11. The latency of CAS operation with GCC CAS (sw_cas), cacheline flush and CAS (sw_flush_cas), and hardware NMP enabled CAS (hw_cas).

memory location for various implementations. For the software CAS (sw_cas), the CAS instruction is issued by the CPU to CXL memory. Its performance benefits from the low-latency CPU cache, and its atomicity is guaranteed by the cache-coherence protocol. The sw_flush_cas configuration models an mCAS by having the CPU first flush the target from the cache, then issue the CAS. For systems without NMP, this is a software emulation for a system that does not have HWcc.

It is important to remember that neither sw_cas nor sw_flush_cas would be safe in a CXL pod without inter-host hardware cache coherence. CAS safety comes from coherence. Two hosts could CAS the same location and both succeed because each would have exclusive access to the line relative to all other caches in its coherence domain.

The hw_cas measures our NMP mCAS implementation, which works without HWcc. At 16 threads (the maximum in our experiments) hw_cas achieves 17.4% lower p50 latency, and 20% lower p99 latency than sw_flush_cas. However, for 1 thread, hw_cas is slower than sw_flush_cas, with a p50 latency of 2.3μs and a p99 of 2.8μs. We believe an ASIC version of the NMP will further reduce this latency gap when the contention is low. However, many projects have used sw_flush_cas to model mCAS [33, 68], and our measurements show that the latencies of these two primitives are comparable.

5.4.2 CXL: allocator microbenchmarks. Figure 12 shows throughput for small heap allocations on CXL memory. We compare against ralloc as a baseline since its heap metadata is separate from application data—though it does not separate HWcc and SWcc metadata—so it can somewhat reduce HWcc usage by placing only its metadata in the HWcc region. We do not compare against cxl-shm because it embeds a HWcc reference count in each allocation; with our mCAS implementation, this would require the whole heap to be marked uncachable, making a fair comparison impossible.

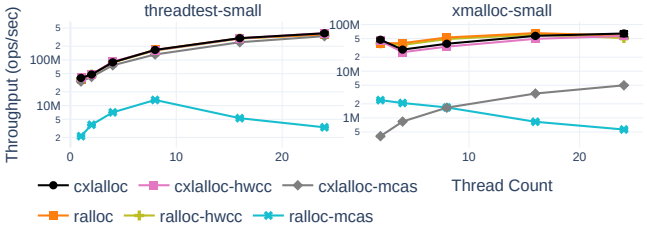


Figure 12. Throughput of small heap allocator microbenchmarks for different CXL HWcc architectural assumptions and increasing thread counts. Experiments are run on a machine with physical CXL memory (§5.4). CXLALLOC and ralloc use local DRAM; -hwcc variants use CXL memory and assume HWcc; -mcas variants use our NMP mCAS prototype (§4).

Overall performance is similar for local DRAM and HWcc CXL memory. For threadtest, CXLALLOC-mcas achieves 80% of CXLALLOC-hwcc's throughput, and 10-99x of ralloc-mcas's throughput: our SWcc protocol allows local operations to keep metadata cached, while ralloc must read a size class from uncachable memory on every free. For xmalloc, CXLALLOC-mcas drops to 1% of CXLALLOC-hwcc's throughput, as every remote free requires an mCAS. Below 8 threads, ralloc-mcas has higher throughput because it shares partial slabs between threads, allowing remote frees to go into thread-local caches. However, slab sharing increases mCAS contention on slab metadata, causing ralloc-mcas to scale poorly; CXLALLOC-mcas attains 9.9x higher throughput at 24 threads.

6 Related Work

CXLALLOC takes inspiration from previous allocators for different types of memory: volatile, persistent, cross-process, and CXL, but contributes novel huge allocation management and combines previous techniques in a new way that provides strong performance across a variety of use cases.

Persistent memory allocation. Ralloc [16] and zallocator [64] are lock-free allocators, while libpmem [6], makalu [14], nvm_malloc [54], and nvalloc [23] use locks. All of them do garbage collection during a blocking recovery period after a failure. Offline GC is attractive because it allows optimizing the common case of avoiding cache flushes and fences for allocator metadata during allocation.

Cxl-shm [68]. Cxl-shm is another memory management system for CXL that tolerates partial failures. Cxl-shm makes several design choices that are incompatible with our constraints. First, it embeds a 24B header into each allocation to support reference counting, 8B of which requires HWcc. This metadata is scattered throughout the heap, inflating HWcc usage. Second, it provides only basic pointer consistency: the heap is created with a fixed size and cannot be extended, and it does not support allocation sizes larger than 1KiB, so it never modifies memory mappings. Thirdly, it requires reference counting, which is suitable for message passing

applications that rarely modify reference counts, but not for applications with shared data structures, for which reference count modifications artificially increase contention, even for read-only workloads.

Tigon [33]. Tigon is an in-memory transactional database that uses explicit allocation of CXL memory to share and synchronize data used in cross-partition transactions. The authors of Tigon used an early version of CXLALLOC in their system. Tigon assumes inter-host hardware cache coherent memory and does not tolerate partial failures. Adapting Tigon to mCAS is interesting future work.

Memory protection. There is recent work on memory allocators [24, 53] that protect the heap from buggy or malicious programs using hardware memory protection mechanisms like Intel's memory protection keys (MPK) [3]. CXLALLOC does not currently implement these mechanisms because we assume processes sharing memory are correct and trusted. That being said, our design does separate heap metadata and heap data, and can therefore be extended with protection mechanisms in the future.

CXL tiered memory management. Recent work explores using CXL to enable memory disaggregation and pooling for improved utilization and reduced costs in datacenter servers [11, 26, 44, 45]. Key research directions include optimizing CXL memory pool configurations for performance and cost savings [50], developing resilient memory managers and intelligent page placement policies to mitigate CXL's higher access latency [37, 42, 57, 58, 61, 68], reducing process and container startup time [8, 32] and leveraging CXL's expanded memory capacity and bandwidth for large-scale applications [30, 36, 62]. These approaches treat CXL as a memory tier that is not directly visible to user software, though some recent work has looked at how to provide an extensible interface [59, 60]. Comparisons on genuine CXL hardware reveal differences from emulated CXL that compel revisiting prior assumptions [58].

CXL and partial failures. Other recent work [55, 63, 68, 71] makes the same observation as this work that CXL systems can observe partial failures, with FUSEE [55] and rTX [63] focusing on RDMA and remote memory nodes.

RDMA. Systems built using RDMA [25, 38, 67] have a disaggregated view of memory, but remote allocation is not controlled directly by a malloc/free interface used by the application. These systems use message passing to coordinate state, not explicit memory allocation. Also, the latency for memory access through RDMA networks is still one to two orders of magnitude higher than local memory [29], while it is 2.3× higher latency in our experimental testbed (§5.4).

7 Conclusion

CXLALLOC is the first memory allocator appropriate for a CXL pod. It is efficient, it supports memory sharing among processes with pointer consistency, and it supports the limited inter-host hardware cache coherence of CXL. CXLALLOC also tolerates partial failures [28], which makes it resilient to thread or process failures. Our evaluation demonstrates CXLALLOC's performance.

8 Acknowledgements

We thank the anonymous reviewers, Vijay Chidambaram, Hayley Leblanc, and Arthur Peters for their insightful comments. Results presented in this paper were obtained using the Chameleon testbed [39] supported by the National Science Foundation. This work was supported in part by the Center for Processing with Intelligent Storage and Memories (PRISM), one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] [n. d.]. Boost.Interprocess. https://www.boost.org/doc/libs/1_77_0/doc/html/interprocess.html. (Accessed: April 2025).

[2] [n. d.]. Compute Express Link (CXL) Specification, Revision 3.2. https://computeexpresslink.org/cxl-specification/. (Accessed: April 2025).

[3] [n. d.]. Memory Protection Keys. https://docs.kernel.org/core-api/protection-keys.html (Accessed: August 2025).

[4] [n. d.]. mimalloc-bench. https://github.com/daanx/mimalloc-bench (Accessed: 2024).

[5] [n. d.]. SK hynix Presents CXL Memory Solutions Set to Power the AI Era at CXL DevCon 2024. https://news.skhynix.com/sk-hynix-presents-ai-memory-solutions-at-cxl-devcon-2024/. (Accessed May 2025).

[6] [n. d.]. The libpmem library. https://pmem.io/pmdk/libpmem. (Accessed: April 2025).

[7] Marcos K. Aguilera, Emmanuel Amaro, Nadav Amit, Erika Hunhoff, Anil Yelam, and Gerd Zellweger. 2023. Memory disaggregation: why now and what are the challenges. *SIGOPS Oper. Syst. Rev.* 57, 1 (June 2023), 38–46. doi:10.1145/3606557.3606563

[8] 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* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 210–226. doi:10.1145/3676641.3715988

[9] Gal Assa, Michal Friedman, and Ori Lahav. 2024. A Programming Model for Disaggregated Memory over CXL. arXiv:2407.16300 (July 2024). doi:10.48550/arXiv.2407.16300 arXiv:2407.16300 [cs].

[10] Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. 2018. Nesting-Safe Recoverable Linearizability: Modular Constructions for Non-Volatile Memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*. ACM, Egham United Kingdom, 7–16. doi:10.1145/3212734.3212753

[11] Daniel S. Berger, Daniel Ernst, Huaicheng Li, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Lisa Hsu, Ishwar Agarwal, Mark D. Hill, and Ricardo Bianchini. 2023. Design Tradeoffs in CXL-Based Memory Pools for Public Cloud Platforms. *IEEE Micro* 43, 2 (2023), 30–38. doi:10.1109/MM.2023.3241586

[12] Daniel S. Berger, Yuhong Zhong, Pantea Zardoshti, Shuwei Teng, Fiodar Kazhamiaka, and Rodrigo Fonseca. 2025. Octopus: Scalable Low-Cost CXL Memory Pooling. https://arxiv.org/abs/2501.09020. (Accessed: April 2025).

[13] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: a scalable memory allocator for multithreaded

applications. *SIGPLAN Not.* 35, 11 (Nov. 2000), 117–128. doi:10.1145/356989.357000

[14] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. 2016. Makalu: Fast Recoverable Allocation of Non-Volatile Memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 677–694. doi:10.1145/2983990.2984019

[15] Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell D. E. Long, and Ethan L. Miller. 2021. Twizzler: A Data-centric OS for Non-volatile Memory. *ACM Trans. Storage* 17, 2 (June 2021), 11:1–11:31. doi:10.1145/3454129

[16] Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. 2020. Understanding and optimizing persistent memory allocation. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management* (London, UK) *(ISMM 2020)*. Association for Computing Machinery, New York, NY, USA, 60–73. doi:10.1145/3381898.3397212

[17] Guoyang Chen, Lei Zhang, Richa Budhiraja, Xipeng Shen, and Youfeng Wu. 2017. Efficient Support of Position Independence on Non-Volatile Memory. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 191–203. doi:10.1145/3123939.3124543

[18] Kyeongmin Cho, Seungmin Jeon, Azalea Raad, and Jeehoon Kang. 2023. Memento: A Framework for Detectable Recoverability in Persistent Memory. *Proc. ACM Program. Lang.* 7, PLDI, Article 118 (jun 2023), 26 pages. doi:10.1145/3591232

[19] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *SIGARCH Comput. Archit. News* 39, 1 (March 2011), 105–118. doi:10.1145/1961295.1950380

[20] 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*. ACM, Indianapolis Indiana USA, 143–154. doi:10.1145/1807128.1807152

[21] Intel Corporation. [n. d.]. Agilex™ 7 FPGA I-Series Development Kit. https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/agilex/agi027.html. (Accessed: April 2025).

[22] Samsung corporation. 2024. Samsung CXL Solutions – CMM-H. (2024). https://semiconductor.samsung.com/news-events/tech-blog/samsung-cxl-solutions-cmm-h

[23] Zheng Dang, Shuibing He, Peiyi Hong, Zhenxin Li, Xuechen Zhang, Xian-He Sun, and Gang Chen. 2022. NVAlloc: Rethinking Heap Metadata Management in Persistent Memory Allocators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 115–127. doi:10.1145/3503222.3507743

[24] Anthony Demeri, Wook-Hee Kim, R. Madhava Krishnan, Jaeho Kim, Mohannad Ismail, and Changwoo Min. 2020. Poseidon: Safe, Fast and Scalable Persistent Memory Allocator. In *Proceedings of the 21st International Middleware Conference*. ACM, Delft Netherlands, 207–220. doi:10.1145/3423211.3425671

[25] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, Ethan L. Miller and Steven Hand (Eds.). ACM, 54–70. doi:10.1145/2815400.2815425

[26] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. 2023. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 727–741. doi:10.1145/3582016.3582031

[27] Jason Evans. 2006. A Scalable Concurrent malloc(3) Implementation for FreeBSD. (2006). https://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf

[28] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A Persistent Lock-Free Queue for Non-Volatile Memory. *SIGPLAN Not.* 53, 1 (feb 2018), 28–40. doi:10.1145/3200691.3178490

[29] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network requirements for resource disaggregation. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA, 249–264.

[30] Yufeng Gu, Alireza Khadem, Sumanth Umesh, Ning Liang, Xavier Servot, Onur Mutlu, Ravi Iyer, and Reetuparna Das. 2025. PIM Is All You Need: A CXL-Enabled GPU-Free System for Large Language Model Inference. 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, 862–881. doi:10.1145/3676641.3716267

[31] Morteza Hoseinzadeh and Steven Swanson. 2021. Corundum: statically-enforced persistent memory safety. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Virtual USA, 429–442. doi:10.1145/3445814.3446710

[32] Jialiang Huang, MingXing Zhang, Teng Ma, Zheng Liu, Sixing Lin, Kang Chen, Jinlei Jiang, Xia Liao, Yingdi Shan, Ning Zhang, Mengting Lu, Tao Ma, Haifeng Gong, and YongWei Wu. 2024. TrEnv: Transparently Share Serverless Execution Environments Across Different Functions and Nodes. 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, 421–437. doi:10.1145/3694715.3695967

[33] 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 2025, Boston, MA, USA, July 7-9, 2025*, Lidong Zhou and Yuanyuan Zhou (Eds.). USENIX Association, 109–128. https://www.usenix.org/conference/osdi25/presentation/huang-yibo

[34] Intel Corporation. accessed in 2024. Intel® Memory Latency Checker v3.10. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

[35] Sunita Jain, Nagaradhesh Yeleswarapu, Hasan Al Maruf, and Rita Gupta. 2024. Memory Sharing with CXL: Hardware and Software Design Approaches. arXiv:2404.03245 (April 2024). http://arxiv.org/abs/2404.03245 arXiv:2404.03245 [cs].

[36] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. 2023. CXL-ANNS: Software-Hardware Collaborative Memory Disaggregation and Computation for Billion-Scale Approximate Nearest Neighbor Search. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 585–600. https://www.usenix.org/conference/atc23/presentation/jang

[37] Houxiang Ji, Srikar Vanavasam, Yang Zhou, Qirong Xia, Jinghan Huang, Yifan Yuan, Ren Wang, Pekon Gupta, Bhushan Chitlur, Ipoom

Jeong, and Nam Sung Kim. 2024. Demystifying a CXL Type-2 Device: A Heterogeneous Cooperative Computing Perspective. In *57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1504–1517. doi:10.1109/MICRO61859.2024.00110

[38] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 185–201. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia

[39] 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.

[40] Daewoo Kim, Trevor Brown, and Ajay Singh. 2024. Are Your Epochs Too Epic? Batch Free Can Be Harmful. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '24)*. Association for Computing Machinery, New York, NY, USA, 30–41. doi:10.1145/3627535.3638491

[41] Christoph Lameter. 2007. SLUB: The unqueued slab allocator V6. https://lwn.net/Articles/229096/

[42] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 17–34. doi:10.1145/3600006.3613167

[43] Daan Leijen, Benjamin Zorn, and Leonardo De Moura. 2019. *Mimalloc: Free List Sharding in Action*. Lecture Notes in Computer Science, Vol. 11893. Springer International Publishing, Cham, 244–265. doi:10.1007/978-3-030-34175-6_13

[44] Philip Levis, Kun Lin, and Amy Tai. 2023. A Case Against CXL Memory Pooling. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks* (, Cambridge, MA, USA,) *(HotNets '23)*. Association for Computing Machinery, New York, NY, USA, 18–24. doi:10.1145/3626111.3628195

[45] 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 2023)*. Association for Computing Machinery, New York, NY, USA, 574–587. doi:10.1145/3575693.3578835

[46] 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. https://www.usenix.org/system/files/osdi25-liu.pdf

[47] Paul Liétar, Theodore Butler, Sylvan Clebsch, Sophia Drossopoulou, Juliana Franco, Matthew J. Parkinson, Alex Shamis, Christoph M. Wintersteiger, and David Chisnall. 2019. snmalloc: a message passing allocator. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*. ACM, Phoenix AZ USA, 122–135. doi:10.1145/3315573.3329980

[48] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyan Yang, Mike Rabbat, Brandon Lucia, and Carole-Jean Wu. 2021. Understanding and Improving Failure Tolerant Training for Deep Learning Recommendation with Partial Recovery. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 637–651. https://proceedings.mlsys.org/paper_files/paper/2021/file/f0f9e98bc2e2f0abc3e315eaa0d808fc-Paper.pdf

[49] Suyash Mahar, Mingyao Shen, Tj Smith, Joseph Izraelevitz, and Steven Swanson. 2024. Puddles: Application-Independent Recovery and Location-Independent Data for Persistent Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*. ACM, Athens Greece, 575–589. doi:10.1145/3627703.3629555

[50] 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

[51] M.M. Michael. 2004. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (June 2004), 491–504. doi:10.1109/TPDS.2004.8

[52] PMem.io. [n. d.]. Persistent memory development kit (PMDK). https://pmem.io/pmdk/

[53] Antonin Reitz, Aymeric Fromherz, and Jonathan Protzenko. 2024. StarMalloc: Verifying a Modern, Hardened Memory Allocator. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 333 (Oct. 2024), 30 pages. doi:10.1145/3689773

[54] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. 2015. nvm_malloc: Memory Allocation for NVRAM. *ADMS@VLDB* 15 (2015), 61–72.

[55] 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

[56] Joshua Suetterlein, Joseph Manzano, and Andres Marquez. 2024. Synchronization for CXL Based Memory. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '24)*. Association for Computing Machinery, New York, NY, USA, 178–185. doi:10.1145/3695794.3695810

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

[58] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 105–121. doi:10.1145/3613424.3614256

[59] Bijan Tabatabai, Mark Mansi, and Michael M. Swift. 2023. FBMM: Using the VFS for Extensibility in Kernel Memory Management. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems, HOTOS 2023, Providence, RI, USA, June 22-24, 2023*, Malte Schwarzkopf, Andrew Baumann, and Natacha Crooks (Eds.). ACM, 181–187. doi:10.1145/3593856.3595908

[60] Bijan Tabatabai, James Sorenson, and Michael M. Swift. 2024. FBMM: Making Memory Management Extensible With Filesystems. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 785–798. https://www.usenix.org/conference/atc24/presentation/tabatabai

[61] 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

[62] 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

[63] Xingda Wei, Haotian Wang, Tianxia Wang, Rong Chen, Jinyu Gu, Pengfei Zuo, and Haibo Chen. 2023. Transactional Indexes on (RDMA or CXL-based) Disaggregated Memory with Repairable Transaction. arXiv:2308.02501 [cs.DB]

[64] You Wu and Lin Li. 2022. Zallocator: A High Throughput Write-Optimized Persistent Allocator for Non-Volatile Memory. *J. Emerg. Technol. Comput. Syst.* 18, 4, Article 80 (oct 2022), 20 pages. doi:10.1145/3549528

[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 *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 2, 17 pages.

[66] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2021. A Large-scale Analysis of Hundreds of In-memory Key-value Cache Clusters at Twitter. *ACM Trans. Storage* 17, 3 (Aug. 2021), 17:1–17:35. doi:10.1145/3468521

[67] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. 2022. FORD: Fast One-sided RDMA-based Distributed Transactions for Disaggregated Persistent Memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 51–68. https://www.usenix.org/conference/fast22/presentation/zhang-ming

[68] 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

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

[70] 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*. ACM, Banff AB Canada, 58–66. doi:10.1145/3713082.3730393

[71] 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

[72] Danyang Zhuo, Kaiyuan Zhang, Zhuohan Li, Siyuan Zhuang, Stephanie Wang, Ang Chen, and Ion Stoica. 2021. Rearchitecting in-memory object stores for low latency. *Proceedings of the VLDB Endowment* 15, 3 (Nov. 2021), 555–568. doi:10.14778/3494124.3494138

A Artifact Appendix

A.1 Abstract

The main benchmark harness is the `cxlalloc-bench` crate, which reads workload configuration files in the `cxlalloc-bench/workloads` directory and runs the YCSB and memcached macrobenchmarks and thread-test and xmalloc microbenchmarks in our evaluation. We use one external data set for the memcached traces [66].

A.2 Artifact check-list (meta-information)

- **Program:** YCSB, included
- **Compilation:** rustc 1.88.0, included by installation
- **Data set:** Memcached traces, 6.7GiB
- **Run-time environment:** Linux, dependencies managed by Nix
- **Execution:** Hardware settings set during installation
- **Metrics:** Throughput, peak memory usage
- **Output:** NDJSON files containing throughput and memory usage, pdf plots
- **How much disk space required (approximately)?:** 10GiB
- **How much time is needed to prepare workflow (approximately)?:** 10min
- **How much time is needed to complete experiments (approximately)?:** 11h per iteration
- **Publicly available?:** https://github.com/nwtnni/cxlalloc
- **Code licenses (if publicly available)?:** MIT
- **Data licenses (if publicly available)?:** SNIA Trace Data Files Download License
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.18234672

A.3 Description

A.3.1 How to access.

```
git clone https://github.com/nwtnni/cxlalloc.git
```

A.3.2 Hardware dependencies. Our figures 4, 5, and 6 (main macrobenchmarks and microbenchmarks) are evaluated on a machine with 80 physical cores (no hyper-threading). The workload configurations can be freely adjusted to use fewer threads and processes, but the figures won't match exactly. The default heap size is 64GiB; it can be reduced based on the hardware, but some baselines may crash.

Our figure 9 is evaluated with a 32-core Intel SPR machine with CXL FPGA. We can provide the FPGA RTL and bitstream, but the current bitstream only works with Altera Agilex 7 I-series FPGA, version R1BES.

A.3.3 Software dependencies. Managed by Nix during installation process.

A.3.4 Data sets. We use the first subtrace of clusters 12, 15, 31, and 37 from [66]. The trace data can be downloaded via this SNIA link. We convert each CSV trace to Parquet. After installation (§A.4), run the following to convert:

```
mv cluster{12,15,31,37}.000.zst cxlalloc/twitter/
./cxlalloc/twitter/convert.sh 12
```

```
./cxlalloc/twitter/convert.sh 15
./cxlalloc/twitter/convert.sh 31
./cxlalloc/twitter/convert.sh 37
```

The resulting parquet files occupy about 6.7 GiB.

A.4 Installation

Run `cxlalloc/script/setup.sh` to install nix and direnv (which we use for dependency management), clone submodules, and set up the hardware for reproducibility (e.g., disabling CPU frequency scaling). Make sure the `cluster{12,15,31,37}.000.parquet` files (§A.3.4) are moved into the `cxlalloc/twitter/` directory.

Basic test. From the root of the `cxlalloc` repository, run `./script/run.sh cxlalloc-bench/workloads/mini.toml`, which compiles and runs a small subset of the macro- and micro-benchmarks. The results will be in `mini.ndjson`, which contains throughput and memory usage information for each benchmark.

A.5 Experiment workflow

Relative paths assume we are at the root of the `cxlalloc` repository. The run script `./script/run.sh` takes a path to a workload configuration file as an argument, and appends results to an `ndjson` file. The workloads from our paper are defined in `./cxlalloc-bench/workloads/`.

A.6 Evaluation and expected results

To reproduce the main figures (8,9, and 10) in our paper, run the following commands:

```
./script/run.sh ./cxlalloc-bench/workloads/main.toml
./script/run.sh ./cxlalloc-bench/workloads/huge.toml
python3 ./plot/macro.py main.ndjson # macro.pdf
python3 ./plot/micro.py main.ndjson # micro.pdf
python3 ./plot/huge.py huge.ndjson # huge.pdf
```

To reproduce the MCAS results (Figure 12), assuming appropriate hardware, run:

```
./script/ablation.sh
python3 ./plot/ablation.py ablation.ndjson
```

A.7 Experiment customization

The fields of the configuration files should be mostly self-explanatory, but we note that our benchmark harness iterates over the cartesian product of the fields within each experiment. The reviewer can vary these fields according to their hardware (e.g., thread count). Note that mimalloc can only be run in a single process, which is why the configuration files are essentially duplicated (once for mimalloc, once for all other allocators).