Partial Failure Resilient Memory Management System for (CXL-based) Distributed Shared Memory

Mingxing Zhang[♠*] Teng Ma[♡*] Jinqi Hua[♠*] Zheng Liu[♡] Kang Chen[♠♣]
Ning Ding[♡] Fan Du[♢] Jinlei Jiang[♠] Tao Ma[♡] Yongwei Wu[♠]
[♠]Tsinghua University [♡]Alibaba Group [♣]Zhongguancun Laboratory [♢]Intel

Abstract

The efficiency of distributed shared memory (DSM) has been greatly improved by recent hardware technologies. But, the difficulty of distributed memory management can still be a major obstacle to the democratization of DSM, especially when a partial failure of the participating clients (e.g., due to crashed processes or machines) should be tolerated.

In this paper, we present CXL-SHM, an automatic distributed memory management system based on reference counting. The reference count maintenance in CXL-SHM is implemented with a special era-based non-blocking algorithm. Thus, there are no blocking synchronization, memory leak, double free, and wild pointer problems, even if some participating clients unexpectedly fail without freeing their possessed memory references. We evaluated our system on real CXL hardware with both micro-benchmarks and end-to-end applications, which demonstrate the efficiency of CXL-SHM and the simplicity/flexibility of using CXL-SHM to build efficient distributed applications.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; • **Hardware** → *External storage*.

Keywords: CXL, Distributed Shared Memory, Non-blocking

ACM Reference Format:
Mingxing Zhang[♠*] Teng Ma[♡*] Jinqi Hua[♠*] Zheng Liu[♡] Kang Chen[♠♣], Ning Ding[♡] Fan Du[♢] Jinlei Jiang[♠] Tao Ma[♡] Yongwei Wu[♠]. 2023. **Partial Failure Resilient Memory Management System for (CXL-based) Distributed Shared Memory. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3600006.3613135**

∗ The first three authors contributed equally to this research.

SOSP '23, October 23–26, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0229-7/23/10.
https://doi.org/10.1145/3600006.3613135

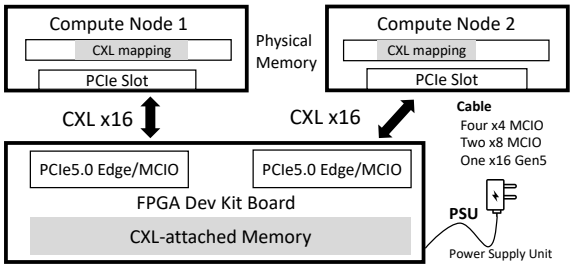


Figure 1. A CXL-based RDSM test platform with two compute nodes sharing a single external CXL memory device.

1 Introduction

1.1 Motivation

Recent highly efficient and byte addressable remote memory access technologies have led to the resurgence of Distributed Shared Memory (DSM) systems [24, 59, 68, 73]. We believe this trend of DSM will continue because the efficiency of distributed cache coherence will be greatly improved by new hardware technologies such as CXL 3.0 [1, 38].

However, since the original DSM systems mainly focus on improving the efficiency of accessing and coordinating remote memory, their memory management systems are usually implemented in a straightforward manner, and do not pay much attention to the automatic management of allocated memory spaces. As an illustration, many existing DSM systems [22, 37, 78, 89] employ a typical two-tier allocator, which first allocates a large chunk of memory via global synchronization, and then performs fine-grained allocations locally. This simple method leaves the burden of memory lifecycle management to the programmers themselves. Programmers must ensure that each allocation of the large chunk is freed once and only once after all the machines have stopped accessing any part of that chunk. This may not be a big problem for data analysis applications where only a few large chunks of data need to be allocated and the processing lifecycle is well modeled by a DAG. But, it will **become a major obstacle** in promoting DSM to a wider range of application scenarios.

For example, refactoring a monolithic application into multiple microservices is a common need today. But, the isolation between different microservices comes at the cost of high-latency inter-process communication, especially when using the original *pass-by-value RPC*. In contrast, with a

highly efficient, byte-addressable, and cache-coherent DSM, different processes/machines can communicate by exchanging zero-copy data references through a shared concurrent memory queue [80]. It avoids the cost of (de)serializing and I/O stack processing of input arguments and return values. As we will show in §6.3, a CXL-based *pass-by-reference RPC* framework can greatly reduce the cost of inter-process and inter-machine communication to a level that is even comparable to the cost of inter-thread communication.

However, the flexibility of microservices places a significant burden on application programmers to manage the lifecycle of shared data objects exchanged via pass-by-reference RPC. Some of their references may even be lost silently (without explicit destruction) due to network and microservice failures. In addition, requiring a microservice process to track the global reference count of distributed objects in its own business logic breaks the isolation semantic of microservice architecture. Thus, we argue that an automatic distributed memory management system that can tolerate possible network, process, and machine failures **is necessary** in this and many other important scenarios (§2.2).

1.2 Challenges

To formalize the above challenge, we propose the model of **Partial Failure Resilient DSM (RDSM)**, where the shared distributed objects allocated from RDSM and the clients (threads/processes/machines) that hold their references have **separate failure domains**. In RDSM, clients are free to join, exit, and even fail during the processing. They are also free to create, release, and even exchange references of fine-grained remote memory spaces that are allocated from the RDSM.

Consider the CXL-based RDSM test platform shown in Figure 1: all three devices have independent power supply unit (PSU) and PCIe/MCIO connections [6], so they have separate failure domains. In this scenario, a memory allocation from Client A on Compute Node 1: *1)* cannot simply be released, even if Compute Node 1 unexpectedly crashes, because it could lead to *double free* or *wild pointer* problems if its reference has been passed to another client that is still alive on another machine (i.e. Compute Node 2); *2)* should be reclaimed to avoid a *memory leak*, when all the clients that originally held its reference have terminated, even if some of those clients terminated unexpectedly. More details about this test platform are given in §2.1.

Since memory management is a well-known difficult task, there are many related design efforts [23, 47, 73, 77, 78, 84] to solve this problem automatically in various scenarios. However, according to our investigation (§4), existing works mainly focus on the complete failure scenario. Thus, even for persistent [25, 39] and shared [90] memory allocators that also need to tolerate unexpected failures, their recovery is typically implemented in a blocking manner, where all participating clients could be blocked by the crash of a single client. But, a **non-blocking** system is desired to tolerate partial failures, because clients sharing a RDSM may serve long-running services that handle latency-sensitive requests.

1.3 Our Contribution

In this paper, we present the design of CXL-SHM (Figure 1), an efficient automatic memory management system for RDSM backed by CXL or other hardware distributed cache coherency technologies. CXL-SHM can **recover from a partial failure of processing clients in a non-blocking manner**, as long as the underlying RDSM provides compare-and-swap (CAS) instructions to atomically update the shared distributed memory space. Our implementation is based on a modern memory allocator, mimalloc [8], which does not involve any cross-thread synchronization in the fast path. It turns out that it is particularly challenging to preserve this property for ensuring the performance, and simultaneously guarantee partial failure safety in a non-blocking manner.

In CXL-SHM, we use reference counting to avoid the overhead of global distributed garbage collection. But, maintaining per-object reference count leads to another challenge of executing distributed transactions. As we will demonstrate later in §4, an existing implementation [90] that uses a redo log for atomicity and locks for concurrency control may lead to indefinite blocking, if the client unexpectedly exits without releasing the locks.

Fortunately, we found that, in this reference count maintenance transaction, only the increment/decrement of reference count is not idempotent. All the other steps (e.g., modifying the value of reference), if designed carefully, can be transformed into idempotent operations. This gives us an opportunity to design a non-blocking **era-based algorithm** (§4.3), which is inspired by the Hazard Era algorithm [64]. The crux is that we use the successful execution of reference count increment/decrement as a **commit point**. This increment/decrement will never be redone. In contrast, once the commit point is passed, the following idempotent steps will be performed at least once either by the client itself, or by the recovery service as a helper if the client fails during the execution, or both.

According to our experiments (§6.1) on real CXL hardware and representative benchmarks, CXL-SHM can allocate/deallocate tens of millions of distributed objects per second. Even after our extension of partial failure recovery, the speed of CXL-SHM is still comparable to state-of-the-art persistent memory allocators, and only an order of magnitude slower than single-machine volatile allocators. The slowness is mainly because, although there is no cross-thread synchronization, at least one memory fence is required to enable recovery from a partial failure by enforcing the execution order of several instructions in the same thread.

The contribution of this paper is two-fold. First, we highlight the importance of handling partial failures directly in low-level concurrent data structure design, where existing solutions from traditional distributed systems become overly

Table 1. Comparisons between local NUMA memory, remote NUMA memory and CXL-attached memory. Demonstrated by millions operations (8 byte accesses) executed per second.

Type	Seq	Rand	Rand CAS	Latency
local NUMA	5200	562	3.3	110 ns
remote NUMA	4312	350	3.3	200 ns
CXL	1487	250	3.3	390 ns

burdensome. We model this problem through the RDSM abstraction and use a variety of examples to illustrate RDSM's advantages. RDSM is different from traditional DSMs that primarily target less flexible applications (e.g., those where object ownership can be easily modeled as a DAG). We argue that RDSM may emerge as a distinctive topic worthy of further exploration.

Second, we design a novel non-blocking algorithm, which is based on our observations of the idempotent part of the reference count maintenance procedure. Based on it, we develop CXL-SHM, which is the first realization of the above RDSM model. We also use several examples to demonstrate the simplicity and flexibility of using CXL-SHM to build efficient distributed applications, including a pass-by-reference RPC framework (described in §2.2.1 and evaluated in §6.3) and a shared-everything distributed key-value store (described in §2.2.2 and evaluated in §6.4). Our implementation is available at https://github.com/madsys-dev/sosp-paper19-ae.

2 The Potentials and Challenges of RDSM

In this section, we demonstrate the potential of RDSM by *1)* reporting the performance of CXL, the most promising technique for implementing RDSM; and *2)* several important scenarios that would benefit from a partial failure resilient automatic memory management system over RDSM.

2.1 Compute Express Link

Compute Express Link (CXL) [3] is an interconnect protocol based on the PCI Express (PCIe) interface. It can attach remote byte addressable CXL-memory into the physical address space of the host machine, which appears to the program as a CPU-less NUMA node. Recently, the CXL Consortium announced version 3.0 of CXL [1], which supports memory sharing. As demonstrated by Figure 2, unlike memory pooling of CXL 2.0, memory sharing of 3.0 will allow the CXL Switch to map the same region of a remote memory pool to multiple host machines' physical addresses at the same time, which can be concurrently updated in the same coherency domain (i.e., a CXL-hardware-assisted DSM).

Since CXL 3.0 has just been announced without any real-world realization, our evaluation is based on a host platform where two compute nodes share an external CXL 2.0 device (Figure 1). However, since the current hardware implementation is not yet capable of creating a single cache coherency domain, additional cache invalidation mechanisms are needed to share the data between different machines. We used only two compute nodes due to real-world device limitations. In contrast, our software design doesn't impose a limit on the number of compute nodes. Our model treats different clients on the same machine in the same manner as those on different machines and hence we can use multiple clients to simulate the scenario of multiple compute nodes.

As we can see from Table 1, the latency of randomly accessing this remote CXL-memory is higher than (but still comparable to) the latency of accessing another NUMA node's memory of the same machine. Similar evaluation results have been reported by other researchers [38, 72]. Major CPU vendors confirm that a future CXL 3.0 device that implements hardware-assisted distributed cache coherence will lead to even smaller memory access latency, with the same topology and parallelism [11, 69].

2.2 Typical Application Scenarios of RDSM

Given the promising performance of modern hardware assisted distributed cache coherence, we argue that the current resurgence of DSM will continue. However, the lack of partial failure resilience deters many important scenarios from stopping applications from using DSM, which leads to the design of RDSM. In this section, we will use two typical examples to demonstrate the benefits and challenges.

2.2.1 Pass-by-reference RPC. Unlike the traditional pass-by-value RPC, pass-by-reference can avoid the expensive data copying in many cases and is therefore widely used in modern distributed processing systems [57, 58, 71, 80]. In these systems, large input arguments and returning values are first stored in a distributed object store [9, 10] and then only the references are passed through the RPC.

The advantage of using RDSM instead of a distributed object store to implement pass-by-reference RPC is two-fold. First, as demonstrated by Lightning [90], the traditional object store architecture mandates that clients interact with the server via RPC/IPC. This interface poses a significant performance bottleneck for low-latency workloads and can be largely mitigated with a shared memory based design. Although Lightning is only a POSIX SHM based single-machine multi-process in-memory object store, it is possible to extend its architecture to a distributed environment with RDSM. The main challenge will be refactoring Lightning's current blocking recovery method into a non-blocking algorithm, which will be further discussed later in §4.2.

Second, unlike the original object interface, RDSM provides a memory allocator interface which allows the construction of dynamic data structures with link pointers and in-place updates. These features further eliminate serialization and deserialization costs for complex data structures. Even the I/O stack processing costs can be avoided by using a concurrent memory queue as a communication channel, where the memory queue is also a shared object.

However, as discussed in §1.1 and [80], the lack of a partial failure resilient automatic memory management system

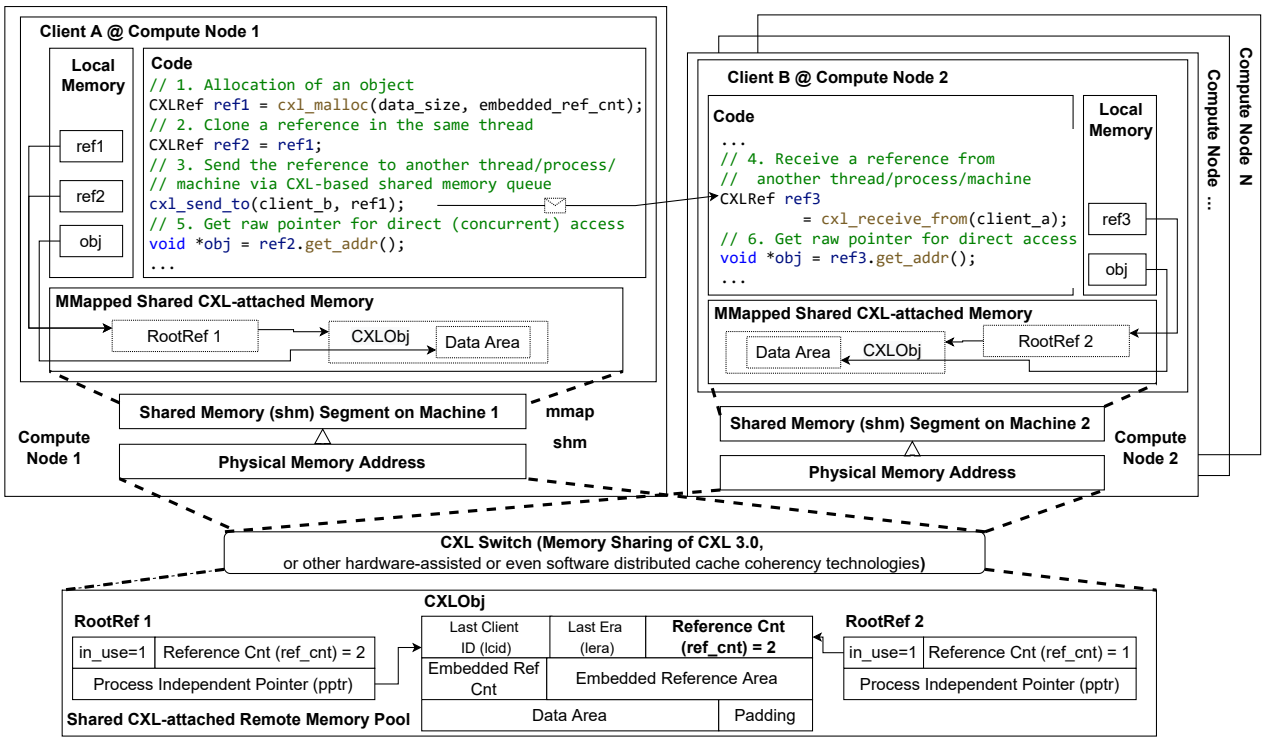


Figure 2. The architecture of CXL-based RDSM and CXL-SHM, a distributed allocator and smart pointer alike system for RDSM. The shared CXL-attached memory pool is mapped to multiple compute nodes' physical memory as a direct access device. The clients are free to join and exit via POSIX shm/mmap APIs without blocking other clients.

places a significant burden on application programmers to manage the lifecycle of distributed objects. In many scenarios such as microservices and serverless, the processing logic cannot be fully modeled by a DAG and therefore requires a generic automatic memory management method. Even worse, despite the challenge of non-blocking, it is also difficult to decide whether a sent reference is received or not in a distributed environment, which leads to a very ambiguous situation when the sender crashes immediately after sending the reference. We will discuss our solution to this problem in §5.2, which also takes advantage of the atomic operation provided by RDSM to implement exactly-once RPC.

2.2.2 Shared-Everything Distributed Processing.

Unlike the shared-nothing architecture, RDSM's efficient hardware-assisted cache coherency enables broader adoption of the shared-everything architecture. As demonstrated by existing studies [79], a shared-everything service typically can achieve better load balance and higher availability. For example, in §6.4, we will demonstrate a non-blocking distributed key-value store over RDSM, where all reader clients from different machines can read the entire store directly, so there is less read imbalance. Writer clients are still sharded into disjoint partitions to ensure a single-writer-multi-reader concurrency model (needed by our current non-blocking algorithm), but repartitioning triggered by write load imbalance or writer failure can be performed very efficiently because no data transfer is required. Only light metadata modification and notification are required to complete the repartitioning.

However, the above shared-everything architecture also comes with many complexities that should be hidden by RDSM's memory management system. Besides the lifecycle management problem discussed in §2.2.1, an important feature necessary for the implementation of a non-blocking concurrent data structure is the capability of atomically modifying link pointers embedded in shared objects (called embedded references in this paper). Although atomic operation is already provided by RDSM, the atomic modification of link pointers is much more complex because it also involves the maintenance of reference counters. More details about our solution are given in §5.4.

3 Overview of CXL-SHM

To solve the challenges inherent in RDSM, we design and implement CXL-SHM, the first realization of the RDSM model. Our current implementation is backed by CXL, but the implementation of distributed cache coherency is orthogonal to our design. It can tolerate partial failures in a non-blocking way as long as the underlying RDSM provides the basic load/store/fence/flush and, most importantly, CAS instructions to read/update the shared distributed memory space. In this section, we give an overview of our system, including the user interface, main components, and limitations of the

system. Then, §4 presents the core non-blocking algorithm used in CXL-SHM and §5 discusses more details that are necessary for using this algorithm in a real-world memory management system.

3.1 Interface

Figure 2 presents an example of using CXL-SHM's API. A client can use *cxl_malloc* to frequently allocate fine-grained *CXLObj* objects from the memory pool, similar to malloc. Along with the size of the data to allocate, a second parameter specifies the number of embedded references in this object. In §5.4, we describe how this count enables the recycling of embedded objects during recovery from partial failure.

The invocation of *cxl_malloc* returns a *CXLRef* that works like a *shared pointer* except that *1)* it enables the toleration of partial failure and hence can be used to maintain the reference counting across different host machines; and *2)* it is not thread safe and hence clone a reference in the same thread. Thus, an explicit copy procedure should be used if another client (may or may not be in the same process/machine) wants to use the same memory object. As demonstrated by step 3/4 in Figure 2, in order to explicitly copy a reference across the boundary of threads/processes/machines, the client needs to send a *CXLRef* with an explicit *cxl_send_to* function. At the receiver side, the receiver client needs to invoke *cxl_receive_from*, which contains necessary steps for increasing the reference counting and will also return a *CXLRef* that points to the same remote CXL memory space (more details in §5.2). Each thread can use their *CXLRef* to obtain a standard process-dependent *void** raw pointer for concurrently and directly accessing the remote memory via memory load/store instructions or CAS instructions.

Moreover, besides the allocated *CXLObj* object stored on CXL shared memory pool and the returned *CXLRef* object stored on local memory, an implicit *RootRef* object is created by every invocation of *cxl_malloc* for tolerating failures. More details are given in §5.1

3.2 Recovery

Similar to Lightning [90], CXL-SHM uses a standalone monitor to detect client failures (due to process or machine failures) and initiate the recovery process asynchronously. The mechanism of detecting failed clients is orthogonal to our main contribution, and a hardware Reliability, Availability, and Serviceability (RAS) feature [2] is desired to ensure that the failed client cannot modify the shared memory pool after its recovery has started. Moreover, in this paper, we consider only the failure of processing clients and computing nodes, not the failure of backend RDSM nodes. Client-side [37, 88, 89] or hardware-based replication [48] or a precise membership protocol [28, 30] can be used to achieve data availability, which is another orthogonal problem in our disaggregated architecture.

In CXL-SHM, recovery does not block the execution of other threads and the recovery service itself is asynchronous, stateless, and fail-safe. Thus, if the recovery service crashes (e.g., due to the machine failure), it can be **simply restarted** on another machine without any issues.

3.3 Memory Management

The implementation of CXL-SHM is based on a modern multi-thread memory allocator mimalloc [8]. In mimalloc, a thread-exclusive *segment* is first allocated from the entire memory *arena* via global synchronization. The region after the header of each segment is further partitioned into *pages* that are dedicated for allocating objects of specific size classes. For example, a page for 16 bytes size class is partitioned into fixed-size 16 bytes data *blocks*. The free blocks in a page can be organized as an intrusive linked list [7]. The head of this list is stored as a *free* pointer in the page meta (a field in the segment header). It points to the first free block in this page. Each free block contains a pointer to the next free block, or *NULL* if it is the last free block in this page. With this design, any further fine-grained allocations from a segment are performed locally and hence do not need cross-thread synchronization, i.e., the fast path.

Similar to mimalloc, the layout of CXL-SHM is also partitioned into four layers (*arena*, *segment*, *page*, *block*) as shown in Figure 3. Specifically, the whole shared memory pool is organized as an *arena*, and two global variables *SegmentAllocationVec* and *ClientLocalVec* are stored at the head of *arena*. The following spaces are partitioned into fixed-size (64MB) *segments*. For allocation, each thread will first allocate an exclusive *segment* from *arena*, and hence further allocation from a *segment* does not need cross-thread synchronization.

The aforementioned *SegmentAllocationVec* is a meta vector for coordinating concurrent segment allocations with other threads. Each meta item in this vector represents a segment and the thread can use CAS to update the "occupied client ID" field for allocation. Objects larger than a single segment are supported by a simple retry and rollback method that occupies continuous segments, because this is a rare case even in slow path. The other three fields are used in memory reclamation that will be further discussed in §5.3.

Most fields in *ClientLocalState* are the same as mimalloc, such as a size class list that points to pages that still have free space for further allocation. The main differences are:

- Our size class starts from 16 bytes rather than 8, because every *CXLObj* will be attached with a header.
- Besides normal size classes, there is a special size class for pages that allocate *RootRef* only. This is a critical design because, after a failure, we can use the content in and only in these pages to destroy the *RootRef* references possessed by this failed thread.
- There is a transaction meta field and a fixed-size redo log area in the thread local state. They are used for

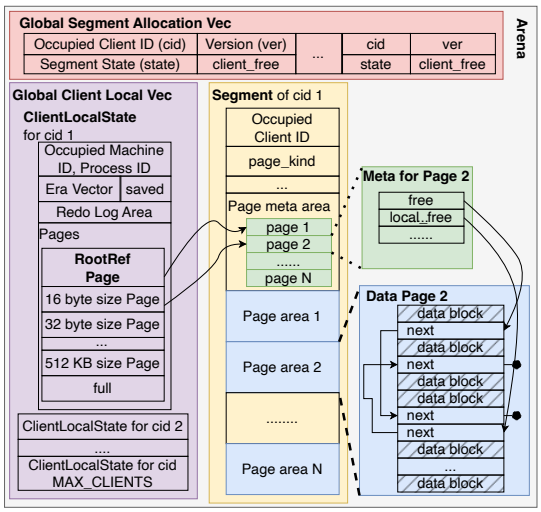


Figure 3. Structure of shared remote memory pool.

recovery from failed transactions, which will be described later in §4.3.

4 Partial Failure Resilient Automatic Memory Management

As we have discussed before, the development of a partial failure resilient automatic memory management system is necessary for many important potential application scenarios of RDSM. But, although automatic memory management is a well studied topic that can be achieved by either tracing garbage collection or reference counting, the existence of partial failures makes it a complex problem in RDSM, which is the key problem we face when designing CXL-SHM.

In this section, we will first introduce the problem and solution of failure recovery in a persistent memory management environment, which is similar to RDSM but does not require the recovery algorithm to be non-blocking. Then, we further demonstrate the challenge of achieving non-blocking automatic memory management with two straw-man implementations. Finally, we will describe our novel era-based algorithm that is used in the implementation of CXL-SHM.

4.1 Failure Recovery in Persistent Memory

Since all the memory of a process can be safely reclaimed by the operating system after the process fails, volatile memory management systems do not need to recover from a failure. In contrast, a persistent memory (pmem) allocator [19, 25, 39, 61, 75, 83, 84] must consider the problem of "crash consistency". Memory allocation and reference linking (storing the value of the allocated space's pointer into the reference) are two separate operations that cannot be implemented with a single atomic instruction. Thus, there is a persistent memory leak if the allocation is done but the linking is not. In the other case, if the linking is done but the allocation is not, there is a wild pointer problem.

To solve this problem, persistent memory allocators typically provide a *setRoot()* function for users to indicate that an object is a root object. Root objects are recorded in special locations that can still be found after a failure, and hence pmem allocators can perform garbage collection to reclaim spaces that are not linked from any root object. However, pmem allocators typically assume a complete failure model and hence use "stop-the-world" garbage collection [25] to recover from the failure of a single process. As we will show in §6.2.1, this recovery can result in a global pause of several or even tens of seconds when there are a large number of objects. This assumption simplifies their implementation and is reasonable in the single-machine scenario of persistent memory, because garbage collection is only executed during the initialization process of the application restart.

However, as described in §2.1, remote CXL memory can be powered by a separate PSU and connected to processing machines via independent PCIe/MCIO connections. A partial failure **will not** affect the accessibility of other alive compute nodes. Thus, a non-blocking recovery method is desired.

Why Not Use a Mostly-concurrent Garbage Collector? Recent developments in mostly-concurrent garbage collectors, such as Shenandoah [32] and ZGC [86], have greatly reduced the duration of stop-the-world pauses. However, these algorithms *1)* still involve blocking procedures during root scanning and relocation, which is undesirable for RDSM where the number of objects can be very large and the collector itself may fail during the blocking procedure; *2)* need to insert memory fences during the dereferencing, which can lead to high overhead without custom hardware [84]; and *3)* periodically scan the huge remote memory pool, which can become a performance bottleneck [77]. Thus, we focus on developing a non-blocking reference counting based method, and leave the exploration of tracing garbage collectors as future work. Essentially, we view reference counting and GC not as replacements for one another, but as distinct tools, each having its unique applications, even within the RDSM context (akin to their roles in multi-threaded programming).

4.2 Challenges

Unlike complex mostly-concurrent garbage collectors, the implementation of a reference counting based automatic memory management system is much simpler, if the possibility of failure is not considered. With a reference count attached to each allocated object, the most basic reference count maintenance procedure – "attach/release a reference" involves only two steps: *1) ModifyRefCnt*: increment or decrement the reference count by one; and *2) ModifyRef*: link the reference by setting the value of the reference to the pointer of the object, or unlink the reference by setting the value of the reference to *NULL*. However, although there are only two steps, this basic procedure *1)* modifies two separate spaces;

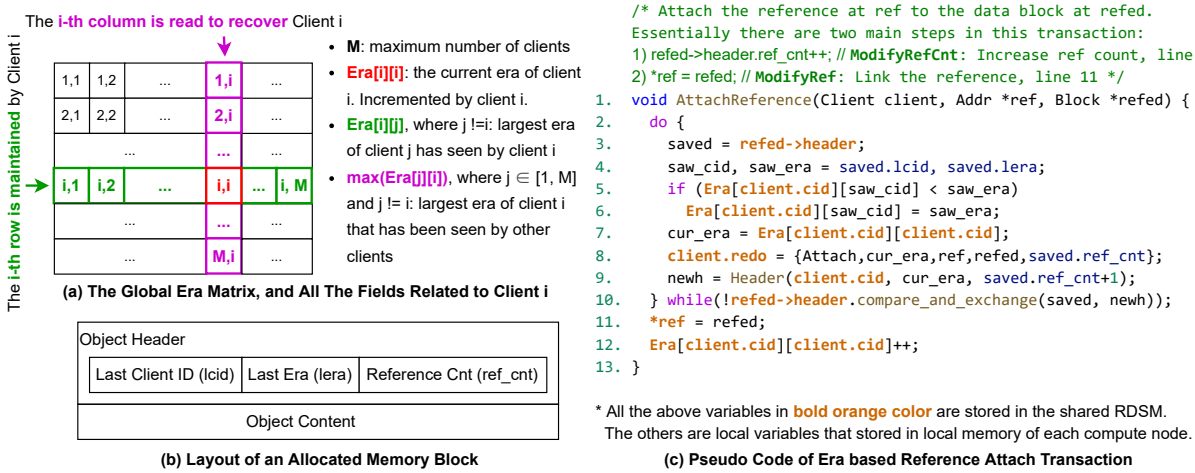


Figure 4. The era-based non-blocking algorithm for reference count maintenance.

2) involves multiple read/write steps; *3)* is not idempotent; and *4)* can be concurrently executed from multiple machines on the same shared object. As a result, it is already enough to form a distributed read-write transaction that requires a lot of considerations to tolerate partial failures in an efficient and non-blocking way.

Here, we use two straw-man implementations to further demonstrate the difficulty.

The first straw-man implementation uses redo logs. The entire procedure can be: *1)* record the execution in a local redo log; *2)* perform the reference count maintenance procedure; and *3)* mark the log as committed. As *ModifyRefCnt* is not idempotent, simply redo it after failure will leads to error if it is already executed by the failed client. But, if the client fails after step 1 and before step 3, there is no way for the recovery service to decide whether or not to redo the *ModifyRefCnt* step.

The second straw-man implementation further uses locks, where each object is associated with not only a reference count but also a spinlock. With the lock, the thread can *1)* lock the object and read its current reference count; *2)* record the new count according to the current value in the redo log; *3)* perform the reference count maintenance procedure; and *4)* release the lock and mark the log as committed. This implementation does not cause memory leak and double free problems because the recorded new count makes *ModifyRefCnt* an idempotent step. In fact, this is the method used in Lightning [91]. Although Lightning only tolerates process failures, extending this method to tolerate partial machine failures is straightforward with the hardware-assisted distributed cache coherence provided by RDSM.

However, this second straw-man implementation is still blocking, and therefore may block other participating clients indefinitely if the current client unexpectedly exits without releasing the lock. The blocked clients will not be able to resume their execution until the recovery service detects the failure and releases the lock during recovery. This problem is acknowledged in Lightning's paper as a major drawback of its current implementation.

4.3 An Era based Algorithm

To solve the above problem, we propose an era-based algorithm that is inspired by the Hazard Era algorithm[1]. It is based on the observation that the entire reference count maintenance procedure can be divided into two phases where *1)* the *ModifyRefCnt* phase is not idempotent but can be executed atomically; and *2)* the following *ModifyRef* phase is idempotent before the application tries to modify the value of the reference once again. Thus, in our algorithm, *ModifyRefCnt* is never redone and therefore we can use the successful execution of *ModifyRefCnt* as a **commit point**.

In order to identify whether *ModifyRefCnt* has been executed or not, each client *1)* is assigned with a unique client ID (cid); *2)* maintains a client-local strictly increasing era that is incremented after each commitment of the reference count maintenance transaction; *3)* maintains an array of the largest era it has seen for every other client. As we can see from Figure 4 (a), this auxiliary information is organized as a M×M two-dimensional era matrix *Era[i][j]* that is stored in a well-known shared memory location (the Era fields of each *ClientLocalState* as illustrated in Figure 3) where the maximum number of clients is M. In this matrix, *Era[i][i]* stores the current era of client *i* and will be incremented by one after each commitment of the reference count maintenance transaction from client cid_1. In contrast, *Era[i][j]* where *i* is not equal to *j* stores the largest era number of client *j* that has been seen by client *i*. We also pack two more fields into the same cache line of each object's reference count, which are *1)* the client ID of the last client (lcid) that successfully

[1]Hazard Era is a non-blocking memory reclamation algorithm [64].

executed *ModifyRefCnt* on this object; and *2)* the era of that last reference count maintenance transaction (lera).

With the above information, the main process of our era-based algorithm is demonstrated by Figure 4 (c) with attaching reference as an example (the process of releasing a reference is similar). As we can see from the figure, if a *client* aims to perform the reference count maintenance transaction on an object (a data block) *refed*, it will first read the current header of *refed* into a local variable *saved*. This read header is used to: *1)* update the largest era of *saved.lcid* that has been seen by the current client through updating *Era[client.cid][saved.lcid]* if it is smaller than *saved.lera* (line 4-6); and *2)* record a redo log entry in its client-local log area with the new count calculated from the read *saved.ref_cnt* (line 8). A separate log area is allocated for each client and all these log areas are stored on well known spaces of the shared RDSM.

Then, the client will try to overwrite the header of *refed* through an atomic CAS instruction to ensure that **no other client** has interleaved in between (*ModifyRefCnt*, line 9-10). Finally, the client can modify the value of the reference (*ModifyRef*, line 11) and finish the transaction by incrementing its local era *Era[client.cid1][client.cid]* by one (line 12).

This algorithm is non-blocking (lock-free but not wait-free) because only a single CAS instruction is used, and at least one of the concurrent executions will make progress. But, its correctness depends on the following two questions.

How to identify a client that fails immediately after executing *ModifyRefCnt* and before *ModifyRef*? As described above, the successful execution of *ModifyRefCnt* is the commit point of the entire transaction. Thus, the recovery service should redo *ModifyRef* if the client fails in the middle of these two steps. In our algorithm, if ID of the failed client is *i*, the recovery service should first check the header of the last object (*lo*) that is modified by the failed client. The address of *lo* is recorded in client i's redo log entry. If *lo.lcid == i* **and** *lo.lera == Era[i][i]*, a redo is needed (Condition 1). Otherwise, the recovery service also needs to compare the failed client's era, with the maximum era of client *i* that has been seen by any other clients. If *Era[i][i] <= max(Era[j][i], j!=i)*, a redo is also needed (Condition 2).

The above conditions are sufficient because if the *ModifyRefCnt* operation is successfully executed on object *lo* from client *i* at era *Era[i][i]* (i.e., the commit point is passed): *1)* Condition 1 will be true, if no other client has overwritten *lo*'s header during the time between client *i*'s failure and its recovery; otherwise *2)* if the header is overwritten by another client *j*, it must have updated its *Era[j][i]*, which guarantees the truth of Condition 2. Essentially, the *Era* matrix can also be viewed as a collection of distributed vector clocks [45]. We use the happen-before relationship established during the reference count maintainaince operation to determine whether an event has happened or not.

A rare corner case occurs when the above condition checking process of the recovery service is running concurrently with a reference count maintaining transaction on the same object *lo* from client *j*. However, this data race will not affect the correctness of our algorithm as long as the recovery service strictly checks Condition 1 before Condition 2 via a memory fence.

Is there any prerequisite for the correctness? The most important assumption of our algorithm is that all the steps following *ModifyRefCnt* are idempotent. This assumption is needed to ensure that it is OK for the asynchronous recovery service to redo the following steps even if the failed client has actually already executed them before the failure. However, this assumption is true only with the cooperation of the memory management system.

First, each reference pointer should be **owned (and hence modified) by only a single writer**. This single-writer-multi-reader concurrency model ensures that the value of the reference will not be modified after the writer's failure, and hence the idempotence of *ModifyRef*. As illustrated later, this restriction does not affect our support for the scenarios described in §2.2, because *1)* only the reference should follow the single-writer-multi-reader model, **the referenced data objects can still be updated concurrently**. Multiple references (owned by different clients) can point to the same shared object; *2)* our memory management system provides specific functionality to atomically transfer the ownership of references between clients.

Second, unlike adding a reference, releasing a reference may further trigger a space reclaim operation if the reference count is decremented to zero. This reclaim operation is not idempotent in existing implementations. Our solution will be discussed in §5.3.

5 Implementation of CXL-SHM

§4 outlines the core algorithm used in CXL-SHM, which is based on the idempotency of *ModifyRef*. However, it is only a simplified model that cannot be directly used in a real-world memory management system. Additional design considerations are necessary to ensure the efficiency and idempotency in many corner cases. In this section, we will discuss the detailed implementation of CXL-SHM separately in memory *1)* allocation; *2)* sharing; and *3)* reclaiming. For allocation, we design a special algorithm for the initialization of reference count, which is involved in the fast path of allocation to further improve the efficiency of CXL-SHM. For sharing, we handle the problem of achieving exactly-once reference transferring described in §2.2.1. For recovery, we solve the problem of non-idempotent memory reclamation. Finally, we also discuss the design of embedded reference, which is important for designing dynamic data structures.

5.1 Memory Allocation

CXL-SHM Allocation The great feature of no cross-thread synchronization in the fast path is preserved in CXL-SHM. However, to tolerate partial failures, we integrate a built-in reference count mechanism in CXL-SHM and hence a header (Figure 4 (b)) is added to each allocated object. Moreover, even the fast path of CXL-SHM allocation involves 4 separate steps that must be executed in a **carefully designed order** to tolerate failures.

As we can see from Figure 2, besides the *CXLObj* object in the shared memory pool and the returned *CXLRef* object in the client's local memory, *cxl_malloc* also implicitly allocates a *RootRef* object in the shared memory pool. The pointer contained in *CXLRef* points to *RootRef*, which contains a pointer to the *CXLObj* object. These are process-independent pointers that can be implemented by an offset value from the beginning of the *arena* like PMDK [39], or a self-contained off-holder pointer [27].

The use of *RootRef* is similar to pmem allocators' root objects, but created automatically to guarantee atomicity and avoid blocking recovery, which is achieved by the following carefully designed step order. First, the client allocates a *RootRef* object from special segment pages (Figure 3) that are dedicated to allocate only *RootRef* objects, and sets the *in_use* bit to 1. These pages are implemented by a specialized size class so that, after a failure, the recovery service can use the content in and only in these pages to release the *RootRef* possessed by the failed client. The allocated *RootRef* object can be safely reclaimed if the client fails immediately after this first step because they are all client local objects (single-writer).

The next step is to find a free data block for the allocated *CXLObj* and write the address of this block into *RootRef* to establish the link. Then, in the third step, the client that is allocating the *CXLObj* should advance the thread-exclusive *free* pointer of the corresponding page. The execution of these two steps should **strictly follow this order** (via a memory fence). If the allocation is performed before linking, the allocated *CXLObj* may become a memory leak if the thread fails just in between these two steps. In contrast, linking first may lead to a double free problem, because the *RootRef* is pointing to a *CXLObj* that has not yet been allocated. However, this double free problem can be detected and avoided by checking the *free* pointer of each page possessed by the failed client during the recovery. A releasing is skipped if the pointer stored in *RootRef* is equal to the *free* pointer of a page. This *free* pointer remains the same if the client fails just in between the above two substeps of allocation, because the page is exclusively owned only by the failed client and will not change before its recovery. The final step is to increment the reference count of *CXLObj* by one. A volatile *CXLRef* object that points to the allocated *RootRef* is also constructed and returned to users.

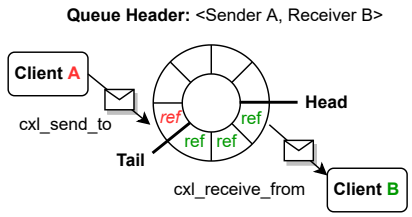


Figure 5. Exchange References between Clients.

With the above order, the fast path of CXL-SHM's allocation does not need any cross thread/process/machine synchronization. Slow path is invoked when there is no free space and hence new pages or even new segments are needed, which is protected by redo logs for atomicity.

5.2 Cross Thread/Process/Machine Sharing

To reduce coordination overhead, we restrict the use of *RootRef* to thread local. Thus, cloning a new reference in the same thread simply increases the *ref_cnt* of the *RootRef* by one. For example, the *ref_cnt* of *RootRef 1* in Figure 2 is 2 because it is referenced by both *ref 1* and *ref 2* of Client 1. In contrast, the sharing of reference across threads/processes is much more complex because it needs to change the *ref_cnt* field in the header of shared *CXLObj*. As an illustration, the *ref_cnt* of *CXLObj* in Figure 2 is also 2 because it is referenced by both *RootRef 1* and *RootRef 2* on different machines.

CXL-SHM provides a *cxl_send_to()* function to wrap a *CXLRef* object in bytes and send it to another thread (another client that has a separate failure domain). The serialization format of a *CXLRef* contains only an offset-based machine independent pointer that directly points to the *CXLObj*.

However, as discussed in §2.2.1, there is **an ambiguous situation** in the transfer of reference count based shared objects in a distributed environment. It is possible that when the receiver calls the receive procedure, the original *CXLObj* is already released, leading to a wild pointer problem. To avoid this, the application should hold at least one reference before there is no further remote reference attachment to this object.

A straw-man solution can ask the receiver to send an ack back to the sender and require that the sender never release its reference before receiving the ack. But this straw-man solution may still lead to wild pointer problems if the sender crashes after sending its reference. In our partial failure model, the recovery service may detect the sender's failure and release all of its references to avoid a memory leak. It is possible that this recovery is performed before the receiver receives the reference.

Thus, in CXL-SHM, the exchange of references between two clients is handled by explicit calling built-in APIs, which are based on shared single-producer-single-consumer (SPSC) fixed-size non-blocking queues. As shown in Figure 5, before transferring any reference from client A to another client

B, client A should first allocate a queue on the shared memory pool and then record the client IDs of both sender and receiver in the queue's header. All these queues should be stored in well known locations so that the recovery service can read them.

Then, to send a reference of a specific shared *CXLObj o* through the queue, A must first attach the reference of *o* to the current tail of the queue. This step is implemented with the same era based reference attaching procedure described in Figure 4 (c). It increments the reference count of *o* by one, and then records the offset-based machine-independent pointer of *o* to the tail of the queue as an atomic transaction. Finally, client A can send the recorded reference to client B by advancing the tail offset of the non-blocking queue.

On the receiver side, the receiver should *1)* use the same era based reference attaching procedure to link the reference stored at the head offset of the queue to a local reference; *2)* release the reference recorded in the queue; and finally *3)* advance the head offset of the queue.

The crux of the above transfer protocol is that ownership of the reference recorded in the queue is transferred atomically from A to B at the point where A advances the tail offset of the queue. This is different from network transfer, where the ownership of an in-flight reference is ambiguous. With a SPSC queue stored on the shared memory and manipulated with atomic instructions, **the transfer of ownership is precise and atomic**. Thus, the recovery service can use this information to identify whether a reference has been transferred or not and hence avoid wild pointer problems.

In summary, CXL-SHM uses a two-tiered reference count. Within the same thread, cloning/destroying an object reference only alters the corresponding RootRef object's count without using atomic instruction and flush, which is good for efficiency. All cross-thread/process/machine exchanges will create new RootRef objects, leading to modifying the reference count in CXLObj's header. These modifications are primarily encapsulated by the *cxl_receive_from* function, used in both cross-thread and cross-process/machine communication. Further reading/writing to the data with the reference does not need to modify the reference count.

5.3 Release a Reference

To release a *CXLRef*, the client will decrement the *ref_cnt* of the *RootRef* that this *CXLRef* points to. The release process terminates if the *ref_cnt* is not zero after decrementing. In contrast, the client needs to further unlink the *RootRef* from the *CXLObj* it points to, which is also an era-based distributed transaction, and reclaim spaces as needed.

However, as discussed in §4.3, the correctness of our era-based algorithm relies on the assumption that all steps following *ModifyRefCnt* are idempotent. But, unlike setting the value of the reference to *NULL*, reclaiming memory space is not idempotent and hence can lead to double free problems if it is redone. Thus, in this rare case, instead of redoing the reclamation, the recovery service will mark the state of the segment that contains that object as a special *POTENTIAL_LEAKING* state by modifying the Segment State fields of the Global Segment Allocation Vec, shown in Figure 3. Segments in *POTENTIAL_LEAKING* state can only be recycled asynchronously with a block-local full scan.

The asynchronous scan will check the *ref_cnt* field of all the data blocks in this segment, and recycle the segment only if they are all zero. This is possible because each page of the segment is partitioned into fixed-size blocks and hence the location of all the *ref_cnt* fields can be found by calculation. To avoid data race problems, this asynchronous full scan is performed periodically in the slow path of the client that had exclusive possession of this segment.

This periodic asynchronous recycling is different from a typical garbage collection method's full scan and hence is acceptable in two main ways. First, it is only required when a client crashes between executing two specific instructions, which is an unlikely event. More importantly, is only a segment-local garbage collection that does not block other clients and the cost of this scan **can be amortized**. It only scans a single segment (64MB in our configuration, not the whole memory space) and the cost of postponing recycling is also only postponing the recycling of a single segment. Therefore, there is unnecessary to perform the full scan frequently. More details will be discussed later in §6.2.1.

5.4 Embedded Reference

In §5.1~§5.3, we only talk about plain objects that do not contain embedded references in a *CXLObj*, where all the references are held by *CXLRef* in local memory. However, embedded references are essential for implementing dynamic data structures that originally use pointers.

In our implementation, the value of an embedded reference is simply an 8 bytes machine-independent pointer without any other header information. Similar to *RootRef*, an embedded reference is stored on the shared memory pool (as it is embedded in a *CXLObj* that is stored on the shared memory pool), and hence it can directly point to a *CXLObj*. Thus, the number of embedded references will also be added to the reference counting (*ref_cnt*) field of the *CXLObj*'s header. Adding and removing an embedded reference is also performed by the era based algorithm. As discussed in §4.3, upper-layer applications should ensure that that there are no multiple threads concurrently modifying the same embedded reference (i.e., single-writer-multi-reader).

Although linking and removing an embedded reference is supported by the era based algorithm described in §4.3, directly changing the value of an embedded reference from pointing to a *CXLObj* A to another *CXLObj* B cannot be implemented by first removing and then linking the embedded reference separately if atomicity is needed.

CXL-SHM provides a specific change function that executes the following steps:

1. Record both *CXLObj* A and B in redo log.
2. Decrement *CXLObj* A's reference count via CAS.
3. Increment the client's era by one.
4. Increment *CXLObj* B's reference count via CAS.
5. Change the value of the embedded reference.
6. Increment the client's era by one once more.

This function is implemented as two *ModifyRefCnt* and a following idempotent *ModifyRef*. The crux is that the era is added twice so that the recovery service can use information to decide whether the failed client has executed or not.

Moreover, if the readers are allowed to concurrently read the linked list without notifying the writer, there is a classical ABA problem for the reclamation of deleted values. This problem can be solved with a standard Hazard era based reclamation [64], because the era is already maintained by our era based reference count algorithm.

Since the destructor function of a specific *CXLObj* is not available during the recovery, we disable the use of custom destructor in CXL-SHM. The application can specify the number of embedded references in its allocated *CXLObj* and make sure that these embedded references are stored at the first few of bytes in the data area. This information of the number of embedded references will also be stored in the header meta of the object (Figure 2) and hence the recovery can use a depth-first search to recursively unlink and release these references if it is needed.

6 Evaluation

Our experiments are based on the same CXL platform described in §2.1. Even though CXL 2.0 driver is not fully upstreamed, developers can leverage `EFI_MEMORY_SP` attribute [4] marked by BIOS. We use `daxctl` to initialize CXL-attached memory in `devdax` (device direct access), where CXL-attached memory is configured into `dax` device. Thus, we can freely use CXL-attached memory by mapping the `dax` device via `mmap`. However, the detailed implementation of distributed cache coherency is orthogonal to our design, as long as the underlying RDSM provides CAS instructions to update the shared distributed memory space atomically.

To show the efficiency of CXL-SHM's memory management system, we first compare it with state-of-the-art allocators, which also demonstrates CXL-SHM's overhead. Then, we study the correctness and execution time of CXL-SHM's recovery procedure. Finally, we demonstrate the usability and ease-of-use of CXL-SHM with several real-world applications. Specifically, we present our PoC implementation of a pass-by-reference RPC and a shared everything distributed key-value store to demonstrate the potential of using RDSM in real-world scenarios.

6.1 Overhead of CXL-SHM

To demonstrate the overhead of CXL-SHM, we compare it with: *1)* mimalloc [8] and jemalloc [31], two state-of-the-art volatile allocators; and *2)* Ralloc [25], a state-of-the-art pmem allocator that can recover from failures through a "stop-the-world" garbage collection and is reported to be much faster than PMDK [39]. Our evaluation is based on two well-known allocator benchmarks: *1)* Threadtest, which allocates and deallocates a large number of 64-byte objects without any sharing between threads. It is introduced by Hoard [17] as a representative micro-benchmark; and *2)* Shbench [56], which allocates and deallocates variable-size (from 64 to 400 bytes) objects. It is designed as a stress test for small-size allocation and reclaiming. All the benchmarks are executed for ten times to obtain converged performance results.

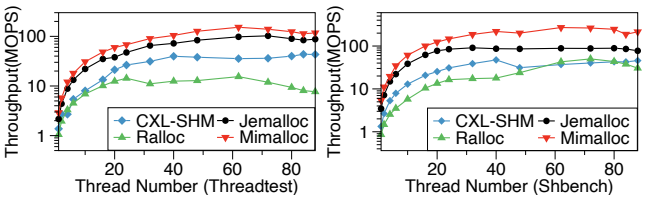


Figure 6. The comparison between CXL-SHM and other state-of-the-art allocators under threadtest/shbench.

Our experimental results in Figure 6 show that CXL-SHM is able to allocate/deallocate tens of millions of fine-grained objects per second (e.g., peak throughput is 43.2/47.4 MOPS in Threadtest/Shbench) for both fixed-size (Threadtest) and variable-size (Shbench) scenarios. This performance is comparable with Ralloc, but the objects allocated by Ralloc cannot be shared across processes/machines.

Moreover, the throughput of CXL-SHM is about an order of magnitude lower than mimalloc and jemalloc. There are three noticeable sources of overheads are added by CXL-SHM to mimalloc's fast path: additional allocations of CXL-Ref/RootRef, a memory fence, and a cache flush, resulting in 2× (4 threads)~5× (62 threads) slowdown when compared to pure-DRAM mimalloc. To measure the effect of these operations in CXL-SHM, they were gradually removed to evaluate the resulting time difference (Figure 7).

Although there is no cross-thread/machine synchronization, at least one flush and memory fence (a *sfence* instruction [5]) is added in the fast path of CXL-SHM's allocation to enforce the order, as described in §5.1. But, as demonstrated by Figure 7, step-by-step breakdown analysis show that the fence account for less than 5% of the slowdown. We also add an additional *CLWB* instruction to flush the cache line of *RootRef* to persist the modification because our current implementation is still based on CXL 2.0, which can occupy 27%~50% of the total time. This flush may not be required in a CXL 3.0 based implementation that implements hardware distributed cache coherence (e.g., with an eADR-like assistance to flush cache data at the time of node failure).

6.2 Recovery of CXL-SHM

6.2.1 Performance of Recovery. We also evaluate the cost of recovery from a client's failure for varying numbers

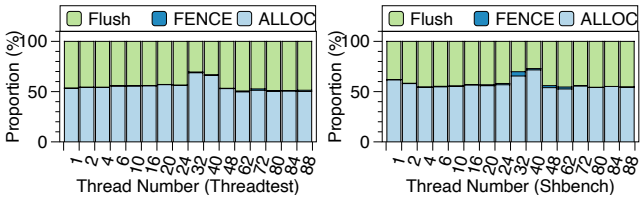


Figure 7. Breakdown of the costs of CXL-SHM.

of *RootRef* possessed by the failed client. According to our experiments, recovery throughput is around 23.5 million objects per second to recover a client that fails without correctly destroying references it possesses. This is much faster than the garbage collection based pmem allocators, which, according to their experiments [25], require 10-100 seconds to recover from the same amount of references (e.g., less than 10k objects per second). The main difference is that CXL-SHM is based on reference counting and hence avoids the cost of conservative garbage collection that may scan the whole memory space. In other words, the cost of recovery is proportional to the number of *RootRef* possessed by the failed client. In contrast, it is proportional to the size of the whole remote memory pool if a conservative garbage collection is used.

Most importantly, the recovery procedure of CXL-SHM is started by an asynchronous monitor and the whole process can proceed asynchronously **without** blocking other processes. This is the main advantage of using per-object reference counting mechanism and only becomes possible with our non-blocking algorithm for executing the reference count maintaining procedure as a distributed transaction.

Moreover, as described in §5.3, an additional periodical segment-local scan may be triggered by the recovery service if a client crashes between executing two specific instructions — an unlikely event not seen in our normal evaluations (except the correctness experiment that deliberately injects this kind of failure). It only scans a single 64MB segment (not a full scan). It takes less than 20us and doesn't need to be performed more than once per second, leading to a minimal impact on throughput.

6.2.2 Correctness of Recovery. Besides theoretical correctness analysis of our algorithm, we also use a fault-injection test to validate the correctness of our implementation. The fault-injection test is designed according to the common practice of pmem crash-consistency validation studies [20, 84]. We develop a test program that starts multiple clients on different machines and randomly creates, releases, and exchanges a certain number of distributed references. This test program is compiled with a special compilation flag, hence a code snippet that randomly brings down the current client is injected in all the critical points of memory allocation/deallocation, reference count increment/decrement, and reference exchange procedures. Then, we execute the fault-injected program more than 100 thousand times and execute post-crash validation to check whether there is leaked memory, double-free, or wild pointers. Our implementation passed all the tests in both memory benchmarks and applications like RPC.

6.3 Pass-by-Reference RPC

Besides the basic memory allocator benchmarks, we implement CXL-RPC, a pass-by-reference RPC described in §2.2.1 to demonstrate the simplicity and flexibility of using CXL-SHM to build efficient distributed applications. We will first introduce the support of embedded reference in CXL-SHM, which is necessary for implementing dynamic data structures. Then, we will describe the implementation of our RPC system and evaluate it with both micro benchmarks and a MapReduce framework as an end-to-end application.

6.3.1 RPC Protocol. With the support of embedded reference, a pass-by-reference remote function call that accepts *I* input arguments can be implemented very straightforwardly in CXL-RPC. The client just needs to *1)* allocate a *CXLObj rpc_msg* that contains the function ID and *I+1* embedded references; *2)* link the first *I* embedded references to input arguments; *3)* allocate a *CXLObj output* and link the last embedded reference of *rpc_msg* to *output*; and finally *4)* send the reference of *rpc_msg* to RPC server via *cxl_send_to*. The server simply polls for *rpc_msg* with *cxl_recv_from* and directly uses the embedded references to access input arguments and modify the output, hence **no data copy** is needed.

As we can see from Figure 8, the throughput of CXL-RPC is 3.83~4.62× higher than RDMA-based RPC with 64-bytes payload (similar to Herd RPC [42] under RC mode and tested with a commercial 50GBps ConnectX-5 RDMA NIC), because it avoids the overhead of serialization, deserialization, network transfers, etc. The performance of CXL-RPC is also insensitive to the size of payload because only zero-copy references are exchanged. In fact, our evaluation shows that when the number of threads is low, the main bottleneck is still object allocation and destruction, not transfer. We also implement inter-thread communication, which can be considered as the performance upper bound of CXL-RPC, which spawns 2-64 pairs of threads and assigns a lock-free SPSC queue [55] to each pair. One thread in each pair allocates the same number of objects and pushes pointers to those objects into the queue; the other thread simultaneously pops pointers from the queue, executes the function, and then deallocates the objects. The results show that CXL-RPC can achieve a performance that is only 46.1%~52.7% lower than pure SPSC reference exchange. We argue that this performance improvement and the ability to exchange distributed objects that can be updated in-place will further enhance the current trend of RPC by reference [21, 80].

6.3.2 CXL-RPC based MapReduce. We also build CXL-MapReduce, a MapReduce framework based on CXL-RPC, as an end-to-end application. Our implementation is similar

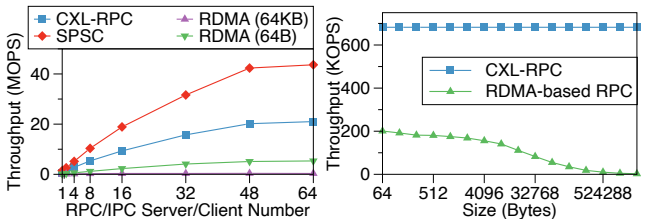


Figure 8. Comparisons between CXL-SHM and RDMA based RPC with different numbers of client/server (payload size is 64Bytes in the left subfigure) and payload size (single client/server in the right subfigure). Simple RPC protobuf is used to achieve the maximum throughput.

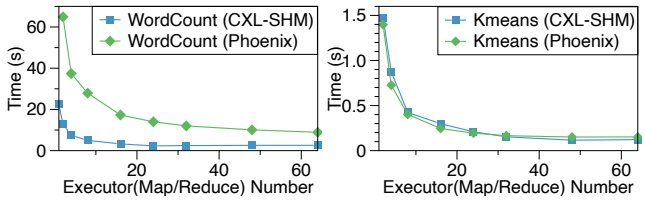


Figure 9. Performance of CXL-MapReduce.

to Phoenix [65], a single-machine, shared-memory MapReduce implementation. Figure 9 shows our evaluation results on word count and kmeans, two of the most popular data-intensive benchmarks. We use a 1GB text dataset for word count and a randomly generated dataset (1k clusters and 500k 8-dimension points) for kmeans. Similar to Phoenix, both map and reduce phase share the same (RDSM) memory region, thus avoiding data copying. As we can see, the performance of CXL-MapReduce based kmeans is comparable with Phoenix 1.0. In the case of word count, the execution time is reduced by 80.0%~87.2% [2]. At the same time, CXL-MapReduce can *1)* take advantage of remote storage; and *2)* scale out to multiple machines. CXL-MapReduce also demonstrates good scalability. For example, the execution time is reduced by 8.31× (this number is 9.54×) when the number of map executors is increased from 2 to 64.

6.4 Shared-everything Key-Value Store

In this section, we use distributed key-value store as an end-to-end example of using CXL-SHM to build shared-everything distributed applications as described in §2.2.2. We will first describe the implementation of our key-value store and then the evaluation results.

6.4.1 Key-Value Store. Distributed key-value store is one of the most important distributed applications that is widely used in the real-world. With CXL-SHM, a rack-scale distributed key-value store can be implemented in the same way as concurrent multi-thread data structures with less than 500 lines of code. The main data structure of our implementation is a concurrent fixed-size latch-free hash index that holds embedded references to key-value records. Hash collisions are handled by organizing records as linked lists. This implementation is possible in CXL-SHM because we allow: *1)* frequent allocation of fine-grained shareable distributed objects; *2)* atomic in-place updates of the distributed object; and *3)* machine independent pointers that can be embedded in other distributed objects.

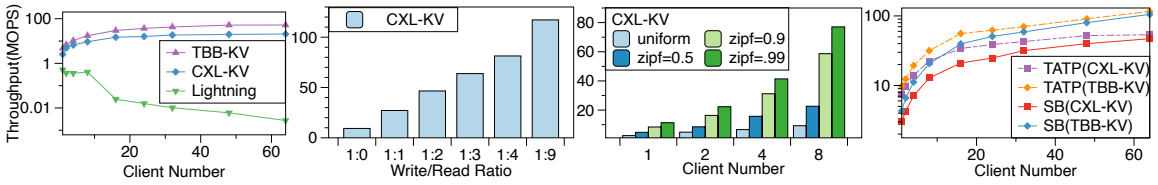
Since only single-writer-multiple-reader is supported in CXL-SHM, the keys are partitioned into disjoint ranges, each assigned to a specific writer. The liveness of these writers can be maintained with application-level heartbeats and leases, so that a dead writer can be taken over by another writer after its lease is expired. This application-level takeover procedure does not necessitate coordination with the recovery service. The single-writer enforcement is due to the atomic swapping of pointers required by many concurrent data structures. In CXL-SHM, such a swap triggers additional modifications to the reference count, precluding atomic execution. The recovery service, on the other hand, isn't bounded by this constraint as it doesn't need to swap pointers. This takeover is quick because there is no need for copy based data repartitioning in our shared-everything architecture. Unlike the disjoint writers, readers can directly read the whole key-value store to achieve a better load balance and read scalability.

Moreover, some persistent root objects (akin to pmem allocators) are needed if users intend to keep alive certain data even if all clients are temporarily crashed. This functionality can be implemented by adding a special API to CXL-SHM.

6.4.2 Evaluation Results. Figure 10 (a) and (b) present the comparison between CXL-SHM based key-value store (CXL-KV) with *1)* a single-process multi-thread key-value store implemented in Intel TBB [63] (TBB-KV), and *2)* Lightning [90], the state-of-the-art single-machine multi-process object store that uses shm to avoid the cost of IPC. CXL-KV can also achieve one to three orders of magnitude higher throughput than Lightning. This performance gap is because memory allocation is not the main focus of Lightning and hence Lightning's memory management is based on a simple lock-based buddy system. CXL-KV is only 1.40~2.61× slower than the concurrent multi-thread hash map, which is mainly due to the performance gap between CXL-attached memory and local memory. Figure 10b shows the throughput of CXL-KV will increase with a lower write/read ratio. With 8 clients, it can reach 117.20 MOPS in 1:9 write/read ratio which is 12.57× larger than the all write case (9.20 MOPS). This is because the writing operations involve memory allocations that execute memory fences. In contrast, the read operations are pure CXL memory loads.

Both CXL-KV and Lightning use reference counting and an asynchronous recovery system for fail-safe memory management. But, in Lightning, all the clients must wait for the recovery even if only one client crashes (the entire recovery process is 9 ms for 10000 objects.), which is avoided in

[2]The PoC implementation of CXL-MapReduce is simpler than Phoenix, which is the source of performance improvements in WordCount. This experiment is only a demonstration of the potential of low overhead and scalability of CXL-SHM, not an apples-to-apples comparison.



(a) Different Thread Number **(b)** Different W/R Ratio **(c)** YCSB w/ different zipf **(d)** TATP/Smallbank test

Figure 10. The performance of CXL-KV under different workloads (YCSB and transaction benchmarks).

CXL-KV through our era based non-blocking algorithm. In contrast, Lightning's isolation technique (not implemented in CXL-KV) is not the reason for Lightning's lower performance [62]. Thus it can be a future work to combine CXL-KV and Lightning or other security protection mechanism [85] to implement a multi-processes object store that is more secure (isolated from incorrect modifications) and scalable.

We also evaluate CXL-KV with several other benchmarks, such as *1)* a key-value oriented benchmark YCSB generating offline workloads, which use our own custom configuration (different zipf parameters); and *2)* transaction benchmarks TATP and Smallbank (only the read write workload is used because currently we have not implemented transaction support in CXL-KV). As shown in Figure 10c, the throughput of CXL-KV increases with a higher zipf value. This is because the key-value store can benefit from a skewed distribution that has better cache locality. Figure 10d shows the throughput of transactions under TATP and Smallbank. Compared to using intel TBB, CXL-KV achieves a comparable performance which is 46.1.2~78.8%/40.7~70.4% in TATP and Smallbank (SB) of TBB-KV respectively. As we can see, CXL-SHM leads to good scalability on all the above scenarios.

7 Related Works

Our work is based on previous development of DSM [15, 18, 29, 34, 47, 49, 78]. However, existing works typically focus on improving the efficiency of software-based [24] or hardware-assisted [47, 78] cache coherency, or avoid the need of cache coherency in upper layer applications [29, 53]. Our system relies on hardware advancements, such as CXL 3.0, to solve the cache coherency problem. The main focus of CXL-SHM is automatic memory management, which is orthogonal to existing works. CXL has demonstrated potential in various applications such as HPC [76], AI/ML training [40, 43] and IO/memory-intensive applications [41, 44, 70, 87]. At the same time, new processing paradigm such as Fabric-Centric Computing (FCC) [52] has been proposed. There are also several works that use CXL to mitigate the memory utilization problem in data centers [16, 35, 36, 50, 54].

The advent of high-speed network technologies has led to another trend of building memory disaggregation systems from the perspective of OS [51, 67], network [33], data structures [13, 60], key-value store [46, 74] and runtime [26]. For example, FaRM [29, 30] offers low latency and high throughput updates to DSM via RDMA. It also implements a precise membership protocol [28] to recover from a backend server's failure, which can be integrated into CXL-SHM as currently we focus only on tolerating compute nodes' failure.

In principle, CXL-SHM is also a kind of object-based memory disaggregation. Thus, it can achieve the same benefit of improving memory utilization as other works [12, 14, 37]. AIFM [66] is an object-based memory disaggregation system that also provides a remote memory allocator abstraction. It allows users to integrate remote memory with application data structures for fine-grained partial remote accessing of data structures without amplification or high overheads. There are many other object-based memory disaggregation systems [14, 29, 81] that focus on intelligently swapping only part of the data to remote memory and keeping the hot data in local memory. These works can be integrated into CXL-SHM by adding an intelligent local cache within the *CXLRef* object. rTX [82] is a recent work that provides transactional indexes on disaggregated memory. It can handle multi-writer scenarios via logging, but at a cost of larger overheads.

8 Conclusion

In this paper, we present CXL-SHM, an efficient, partial failure resilient, and non-blocking memory management system. We also demonstrate the simplicity/flexibility of using CXL-SHM to build efficient distributed applications, through several end-to-end applications such as pass-by-reference RPC and shared-everything distributed key-value store.

Acknowledgments

We thank the anonymous reviewers and our shepherd Prof. Angela Demke Brown for their helpful suggestions. We thank the colleagues from Intel and Alibaba Group for their enthusiastic assistance. The authors from Tsinghua University are all in the Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China. This work is supported by National Key Research & Development Program of China (2022YFB4502004), Natural Science Foundation of China (62141216, 61877035), Young Elite Scientists Sponsorship Program by CAST (2022QNRC001). This work was also supported by Alibaba Group through Alibaba Innovative Research Program. Correspondence to: Mingxing Zhang (zhang_mingxing@mail.tsinghua.edu.cn), Tao Ma (boyu.mt@taobao.com).

References

[1] 2022. Compute Express Link 3.0. https://www.computeexpresslink.org/_files/ugd/0c1418_a8713008916044ae9604405d10a7773b.pdf.

[2] 2022. Compute Express Link CXL 3.0 is the Exciting Building Block for Disaggregation. https://www.servethehome.com/compute-express-link-cxl-3-0-is-the-exciting-building-block-for-disaggregation/.

[3] 2022. Compute Express Link™: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/home.

[4] 2022. EFI Special Purpose Memory Support. https://lwn.net/Articles/784971/.

[5] 2022. Intel 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.

[6] 2022. Intel® Agilex™ I-Series FPGA Development Kit User Guide. https://www.intel.com/content/www/us/en/docs/programmable/683288/current/overview.html.

[7] 2022. Intrusive linked lists. https://www.data-structures-in-practice.com/intrusive-linked-lists/.

[8] 2022. Mimalloc. https://github.com/microsoft/mimalloc.

[9] 2022. The Plasma In-Memory Object Store. https://arrow.apache.org/docs/python/plasma.html.

[10] 2022. Vineyard (v6d) an in-memory immutable data manager. https://v6d.io/.

[11] 2023. Compute Express Link CXL Latency How Much is Added at HC34. https://www.servethehome.com/compute-express-link-cxl-latency-how-much-is-added-at-hc34/.

[12] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, et al. 2018. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 775–787.

[13] Marcos K Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. 2019. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 120–126.

[14] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 843–857.

[15] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. 1990. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming* (Seattle, Washington, USA) *(PPOPP '90)*. Association for Computing Machinery, New York, NY, USA, 168–176. https://doi.org/10.1145/99163.99182

[16] Daniel S Berger, Daniel Ernst, Huaicheng Li, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Lisa Hsu, Ishwar Agarwal, Mark D Hill, et al. 2023. Design Tradeoffs in CXL-Based Memory Pools for Public Cloud Platforms. *IEEE Micro* 43, 2 (2023), 30–38.

[17] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. 2000. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices* 35, 11 (2000), 117–128.

[18] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. 1993. *The Midway Distributed Shared Memory System.* Technical Report. USA.

[19] Kumud Bhandari, Dhruva R Chakrabarti, and Hansjuergen Boehm. 2016. Makalu: fast recoverable allocation of non-volatile memory. *conference on object oriented programming systems languages and applications* 51, 10 (2016), 677–694.

[20] Koustubha Bhat, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. 2021. FIRestarter: Practical Software Crash Recovery with Targeted Library-level Fault Injection. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 363–375. https://doi.org/10.1109/DSN48987.2021.00048

[21] Daniel Bittman, Robert Soulé, Ethan L. Miller, Vishal Shrivastav, Pankaj Mehra, Matthew Boisvert, Avi Silberschatz, and Peter Alvaro. 2021. Don't Let RPCs Constrain Your API. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks* (Virtual Event, United Kingdom) *(HotNets '21)*. Association for Computing Machinery, New York, NY, USA, 192–198. https://doi.org/10.1145/3484266.3487389

[22] Jeff Bonwick et al. 1994. The Slab Allocator: An Object-Caching Kernel Memory Allocator.. In *USENIX summer*, Vol. 16. Boston, MA, USA.

[23] Roberto Brega and Gabrio Rivera. 2000. Dynamic Memory Management with Garbage Collection for Embedded Applications.. In *WIESS*. 81–82.

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

[25] 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. https://doi.org/10.1145/3381898.3397212

[26] Irina Calciu, M Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 79–92.

[27] Guoyang Chen, Lei Zhang, Richa Budhiraja, Xipeng Shen, and Youfeng Wu. 2017. Efficient Support of Position Independence on Non-Volatile Memory. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 191–203.

[28] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. 2001. Group Communication Specifications: A Comprehensive Study. *ACM Comput. Surv.* 33, 4 (dec 2001), 427–469. https://doi.org/10.1145/503112.503113

[29] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. 401–414.

[30] Aleksandar Dragojević, 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* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 54–70. https://doi.org/10.1145/2815400.2815425

[31] Jason Evans. 2006. A Scalable Concurrent malloc(3) Implementation for FreeBSD. (01 2006).

[32] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An Open-Source Concurrent Compacting Garbage Collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (Lugano, Switzerland) *(PPPJ '16)*. Association for Computing Machinery, New York, NY, USA, Article 13, 9 pages. https://doi.org/10.1145/2972206.2972210

[33] 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 *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 249–264.

[34] Kourosh Gharachorloo. 1999. The Plight of Software Distributed Shared Memory. *In Invited talk at 1st Workshop on Software Distributed Shared Memory (WSDSM'99)* (1999).

[35] Donghyun Gouk, Miryeong Kwon, Hanyeoreum Bae, Sangwon Lee, and Myoungsoo Jung. 2023. Memory pooling with cxl. *IEEE Micro* 43, 2 (2023), 48–57.

[36] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access High-Performance Memory Disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 287–294.

[37] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 649–667.

[38] Minho Ha, Junhee Ryu, Jungmin Choi, Kwangjin Ko, Sunwoong Kim, Sungwoo Hyun, Donguk Moon, Byungil Koh, Hokyoon Lee, Myoungseo Kim, Hoshik Kim, and Kyoung Park. 2023. Dynamic Capacity Service for Improving CXL Pooled Memory Efficiency. *IEEE Micro* 43, 2 (2023), 39–47. https://doi.org/10.1109/MM.2023.3237756

[39] Intel. 2022. Persistent Memory Development Kit. https://pmem.io/pmdk/.

[40] 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)*. 585–600.

[41] Myoungsoo Jung. 2022. Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*. 45–51.

[42] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2014. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*. 295–306.

[43] Miryeong Kwon, Junhyeok Jang, Hanjin Choi, Sangwon Lee, and Myoungsoo Jung. 2023. Failure Tolerant Training With Persistent Memory Disaggregation Over CXL. *IEEE Micro* 43, 2 (2023), 66–75.

[44] Miryeong Kwon, Sangwon Lee, and Myoungsoo Jung. 2023. Cache in Hand: Expander-Driven CXL Prefetcher for Next Generation CXL-SSD. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*. 24–30.

[45] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (jul 1978), 558–565. https://doi.org/10.1145/359545.359563

[46] Sekwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K Aguilera, Kimberly Keeton, and Vijay Chidambaram. 2022. DINOMO: An Elastic, Scalable, High-Performance Key-Value Store for Disaggregated Persistent Memory. *Proceedings of the VLDB Endowment* 15, 13 (2022), 4023–4037.

[47] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. MIND: In-Network Memory Management for Disaggregated Data Centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 488–504. https://doi.org/10.1145/3477132.3483561

[48] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. 2022. Hydra : Resilient and Highly Available Remote Memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 181–198. https://www.usenix.org/conference/fast22/presentation/lee

[49] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam. 1992. The Stanford Dash multiprocessor. *Computer* 25, 3 (1992), 63–79. https://doi.org/10.1109/2.121510

[50] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 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*. 574–587.

[51] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 267–278.

[52] Ming Liu. 2023. Fabric-Centric Computing. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. 118–126.

[53] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. 2020. AsymNVM: An efficient framework for implementing persistent data structures on asymmetric NVM architecture. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 757–773.

[54] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2022. TPP: Transparent Page Placement for CXL-Enabled Tiered Memory. *arXiv preprint arXiv:2206.02878* (2022).

[55] Maged M Michael and Michael L Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. 267–275.

[56] Inc MicroQuill. 2022. shbench. http://www.microquill.com/smartheap/shbench/.

[57] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 561–577.

[58] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. 2011. CIEL: A Universal Execution Engine for Distributed Data-Flow Computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI'11)*. USENIX Association, USA, 113–126.

[59] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. {Latency-Tolerant} Software Distributed Shared Memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 291–305.

[60] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, et al. 2019. Storm: a fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage*. 97–108.

[61] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. 2017. Memory management techniques for large-scale persistent-main-memory systems. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1166–1177.

[62] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. Libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 241–254.

[63] Chuck Pheatt. 2008. Intel® Threading Building Blocks. *J. Comput. Sci. Coll.* 23, 4 (apr 2008), 298.

[64] Pedro Ramalhete and Andreia Correia. 2017. Brief Announcement: Hazard Eras - Non-Blocking Memory Reclamation. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures* (Washington, DC, USA) *(SPAA '17)*. Association for Computing Machinery, New York, NY, USA, 367–369. https://doi.org/10.1145/3087556.3087588

[65] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. 13–24. https://doi.org/10

.1109/HPCA.2007.346181

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

[67] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 69–87.

[68] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed Shared Persistent Memory. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 323–337. https://doi.org/10.1145/3127479.3128610

[69] Debendra Das Sharma, Robert Blankenship, and Daniel S. Berger. 2023. An Introduction to the Compute Express Link (CXL) Interconnect. arXiv:2306.11227 [cs.AR]

[70] Joonseop Sim, Soohong Ahn, Taeyoung Ahn, Seungyong Lee, Myunghyun Rhee, Jooyoung Kim, Kwangsik Shin, Donguk Moon, Euiseok Kim, and Kyoung Park. 2022. Computational CXL-Memory Solution for Accelerating Memory-Intensive Applications. *IEEE Computer Architecture Letters* 22, 1 (2022), 5–8.

[71] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2438–2452. https://doi.org/10.14778/3407790.3407836

[72] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Ipoom Jeong, Ren Wang, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. arXiv:2303.15375 [cs.PF]

[73] Konstantin Taranov, Salvatore Di Girolamo, and Torsten Hoefler. 2021. CoRM: Compactable Remote Memory over RDMA. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1811–1824. https://doi.org/10.1145/3448016.3452817

[74] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. 33–48.

[75] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing non-volatile memory in database systems. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 1541–1555.

[76] Jacob Wahlgren, Maya Gokhale, and Ivy B Peng. 2022. Evaluating Emerging CXL-enabled Memory Pooling for HPC Systems. In *2022 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE, 11–20.

[77] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2020. Semeru: A Memory-Disaggregated Managed Runtime. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 15, 20 pages.

[78] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. 2021. Concordia: Distributed Shared Memory with {In-Network} Cache Coherence. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 277–292.

[79] Ruihong Wang, Jianguo Wang, Stratos Idreos, M. Tamer Özsu, and Walid G. Aref. 2022. The Case for Distributed Shared-Memory Databases with RDMA-Enabled Memory Disaggregation. *Proc. VLDB Endow.* 16, 1 (sep 2022), 15–22. https://doi.org/10.14778/3561261.3561263

[80] Stephanie Wang, Benjamin Hindman, and Ion Stoica. 2021. In Reference to RPC: It's Time to Add Distributed Memory. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 191–198. https://doi.org/10.1145/3458336.3465302

[81] Xingda Wei, Rong Chen, and Haibo Chen. 2020. Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 117–135.

[82] 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]

[83] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. 2018. Espresso: Brewing Java For More Non-Volatility with Non-Volatile Memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 70–83. https://doi.org/10.1145/3173162.3173201

[84] Yuanchao Xu, Chencheng Ye, Yan Solihin, and Xipeng Shen. 2022. FFCCD: Fence-Free Crash-Consistent Concurrent Defragmentation for Persistent Memory. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 274–288. https://doi.org/10.1145/3470496.3527406

[85] Bin Yan, Youyou Lu, Qing Wang, Minhui Xie, and Jiwu Shu. 2023. Patronus: High-Performance and Protective Remote Memory. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) *(FAST'23)*. USENIX Association, USA, Article 20, 16 pages.

[86] Albert Mingkun Yang and Tobias Wrigstad. 2022. Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK. *ACM Trans. Program. Lang. Syst.* 44, 4, Article 22 (sep 2022), 34 pages. https://doi.org/10.1145/3538532

[87] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin-yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S Kim. 2023. Overcoming the Memory Wall with {CXL-Enabled}{SSDs}. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 601–617.

[88] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–18.

[89] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. 2022. Carbink: Fault-Tolerant Far Memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 55–71. https://www.usenix.org/conference/osdi22/presentation/zhou-yang

[90] Danyang Zhuo, Kaiyuan Zhang, Zhuohan Li, Siyuan Zhuang, Stephanie Wang, Ang Chen, and Ion Stoica. 2022. Rearchitecting In-Memory Object Stores for Low Latency. *Proc. VLDB Endow.* 15, 3 (feb 2022), 555–568. https://doi.org/10.14778/3494124.3494138

[91] Danyang Zhuo, Kaiyuan Zhang, Zhuohan Li, Siyuan Zhuang, Stephanie Wang, Ang Chen, and Ion Stoica. 2022. Rearchitecting in-memory object stores for low latency. In *Proceedings of the VLDB Endowment.*