LUPIN: Tolerating Partial Failures in a CXL Pod

Zhiting Zhu[*]
zhitingz@lepton.ai
Lepton AI Inc.
Cupertino, California, United States

Newton Ni
Yibo Huang
nwtnni@cs.utexas.edu
ybhuang@cs.utexas.edu
The University of Texas at Austin
Austin, Texas, United States

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

Zhipeng Jia
zhipengjia@google.com
Google LLC
Seattle, Washington, United States

Nam Sung Kim
nam.sung.kim@gmail.com
University of Illinois
Urbana-Champaign
Urbana, Illinois, United States

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

Abstract

A compute express link (CXL) pod is a collection of hosts attached to a CXL memory module. It provides an opportunity to port single-host shared-memory programs to execute on multiple hosts in a CXL pod, where the ported application achieves higher performance than a distributed application that uses network for coordination. The cost of performance scaling on a CXL pod is that applications should tolerate partial failures, where one process or operating system fails or reboots. LUPIN is system software that includes kernel modifications and user-level libraries to help applications remain available while they recover from partial failures using the contents of CXL memory.

CCS Concepts: **• Computer systems organization → Availability**; **Availability**; **• Hardware → Memory and dense storage**; **Memory and dense storage**; **• Software and its engineering → Operating systems**; **Operating systems**.

Keywords: CXL, Partial failure tolerance

ACM Reference Format:
Zhiting Zhu, Newton Ni, Yibo Huang, Yan Sun, Zhipeng Jia, Nam Sung Kim, and Emmett Witchel. 2024. LUPIN: Tolerating Partial Failures in a CXL Pod . In *2nd Workshop on Disruptive Memory Systems (DIMES '24), November 3, 2024, Austin, TX, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3698783.3699377

[*]Work down while at The University of Texas at Austin.

1 Introduction

The emergence of Compute Express Link (CXL) memory provides a new technology useful for the construction of distributed systems. CXL memory allows a DRAM module to be physically connected to many hosts (e.g., at least 16 [43]) via the PCIe bus. We call such a collection of machines a **CXL pod**. Initial interest in CXL is driven by cloud vendors seeking disaggregated memory solutions to allow denser packing of client virtual machines and additional DRAM capacity [43, 46]. In this context, CXL memory is another tier in the memory hierarchy, like far memory [37, 40, 44, 49, 56, 65]. But proposed features of CXL memory allow it to be dynamically shared by multiple hosts [1], providing an intermediate hardware configuration between a network of machines, each with their private memory, and a shared memory multiprocessor with a unified memory.

Single-host multi-threaded applications are typically more efficient than distributed applications because they use shared memory rather than the network for coordination, but that comes with a cost of limited scalability. The CXL pod hardware provides an opportunity to scale multi-threaded applications written for a single host to execute on multiple hosts sharing CXL memory, by allowing applications to explicitly allocate CXL memory. For example, in-memory MapReduce [45, 48] is a popular way to write multi-threaded programs using a simple programming model that achieves high performance [25]. Running in-memory MapReduce on a CXL pod allows it to achieve better scalability while preserving single-host programming simplicity and performance.

However, applications running on a CXL pod need to deal with **partial failures**—what happens when a thread, process, or operating system on one host terminates abnormally while the application continues running on other hosts. Recent work shares this failure model [51, 59, 64]. For example, highly available applications should continue processing even during operating system restarts, which can take minutes, without pausing the entire application. Availability requires synchronization primitives (used extensively in the OS and multi-threaded applications to protect shared data structures) that tolerate partial failures, as well as data structures that safely permit accesses concurrent with threads that have failed or are in the process of recovering from a failure.

Lupin[1] is system software that helps applications running on a CXL pod tolerate partial failures. It addresses the following three fundamental problems introduced by partial failures.

- **Failure detection and notification.** The Lupin OS detects application process and kernel thread failures. Each Lupin OS runs a protocol to detect the failure of other OSes. (§3.3) Processes can enroll in CXL control groups (§3.4) to receive failure notifications from another host.
- **Failure recovery.** Lupin provides recoverable locks (§3.5) and a scheduling library to enable applications to structure their recovery actions. Lupin also establishes correctness conditions for recovery—crash consistency and detectable execution [24] (§2).
- **Memory management.** Lupin provides a kernel-level memory allocator that tolerates partial failures using a recoverable lock.

We develop and evaluate Lupin using single-host multi-threaded software including two programs from the PARSEC benchmarks [12] and four programs running on a shared-memory MapReduce framework [45, 48]. These workloads are originally written for a single host shared-memory multiprocessor. While the MapReduce [22] design comes from distributed systems, shared-memory MapReduce [45, 48] does not tolerate partial failures. Lupin's motivation to tolerate partial failures on a CXL pod applies to any highly available application and any long-running application with significant intermediate state that is expensive to checkpoint.

Our evaluation shows that Lupin enables applications to be resilient to partial failures. When running MapReduce and PARSEC programs, failing and restarting half of the processes slows the performance down by at most 7.8%.

2 Motivation and background

Lupin supports porting single-host, shared-memory applications to a CXL pod and all of the applications we port are a single process with multiple threads. Single-host systems are easier to develop, maintain and deploy and do not suffer average and tail latency problems from network-based communication. Lupin supports multi-process applications that share memory, but to simplify exposition we assume that single-host systems have a single process with multiple threads, whereas the CXL pod version will have multiple processes, each running on a different host and each containing multiple kernel threads (also known as light-weight processes or pthreads–they execute user code but are scheduled by the kernel).

Recovering state from CXL memory. The CXL memory is shared by and accessible to all the operating systems running on the connected hosts. The memory device itself is volatile DRAM, but it remains powered on and can retain state while any host that is attached reboots. The contents of CXL memory persists across process and OS restarts (planned and unplanned). If system software can restore the state of its computation from CXL memory, a CXL pod will have less downtime and can support higher utilization by restarting individual processes rather than entire applications.

Lupin's goal is increased scalability and availability, which we believe can be provided at a reasonable cost by CXL memory. Just as DRAM can be made reliable with error correcting codes (ECC), a CXL memory module supports strong, custom ECC implemented in the CXL controller [16]. Other recent work assumes software can recover its state from CXL memory [59, 64]. Orthogonal techniques are necessary to make results persistent, replicated, or geo-replicated.

Why tolerate partial failures? Use of common synchronization primitives, like locks, that are not fault-tolerant requires applications and operating systems to fully restart, converting partial failures to total system failures. Locks and many common synchronization primitives like blocking mutexes are not fault-tolerant. Linux and Windows use many locks and other non-fault tolerant synchronization and if they share CXL memory then they will have to fully restart on a partial failure—if any operating system reboots, the entire pod must reboot.

It is possible for simple applications to ignore partial failures and have an "all-or-nothing" approach to fault tolerance where if any application process fails then the entire distributed application fails. Such a system will restart from persistent storage, and if it has certain processing guarantees like doing operations exactly once, it will have to regularly checkpoint or log to persistent storage (usually at significant performance cost [61]).

We argue that highly-available and long-running applications will want to tolerate partial failures, so the system should support such applications. Tolerating partial failures allows one of the machines in a CXL pod to reboot without having to reboot all the machines, for example, and also allows applications to simply restart a failed thread or process. Other work shares the motivation of high availability [51, 64].

Failure model. Lupin assumes crash-stop failures where application threads and processes and also operating systems

[1]A Lupin (also known as a bluebonnet) is a genus of plant with attractive flowers that have a high-protein seed pod.

Metric	Intel SPR Local	Intel SPR CXL	Intel ICX Local	Intel ICX NUMA
Read lat. (ns)	1	(2.3×)	73	192 (2.63×)
Total BW (GB/s)	1	(0.072×)	147	47 (0.32×)
BW/ch (GB/s)	1	(0.58×)	18.4	5.9 (0.32×)

Table 1. Intel SPR is Sapphire Rapids and Intel ICX is Ice Lake. The CXL memory module is a commercial prototype and performance measurements have been normalized at the request of the hardware manufacturer, so we provide non-normalized measurements of ICX for context. The ratio of remote to local is in parentheses.

may fail at any time, but such failures are not Byzantine. Similar to other fault-tolerant systems [62], Lupin does not tolerate random scribbles into shared CXL memory. Under this failure model, the Lupin OS guarantees crash consistency via durable linearizability [31] where individual operations appear to be executed atomically (at a linearization point) between their invocation and response, and recovering from a failure returns to a prefix of the sequence of linearized operations.

Lupin also requires detectable execution [24] for partial failures, as is common in persistent memory systems [9, 10, 18]. Detectable execution ensures the application has written enough state in advance of its operations to recover fully if it crashes: where recovering fully means it knows what operation it was attempting and whether that operation succeeded before the crash. Informally, a detectably recoverable data structure enables a programmer to ensure that all operations occur exactly once, even in execution histories that contain concurrent operations and failures.

Threat model. We assume a setting where administrators are trusted, as is system software. For this work, system software is the OS, but it could be the hypervisor. Different OSes in a CXL pod share data structures and trust the information they store there, though the kernel protects all its data structures from user programs. Administrative tasks like assigning unique 32-bit IDs to hosts (§3.1) can be accomplished by trusted administrators in many ways such as: setting the hostname or using the physical port number on the CXL device.

2.1 Hardware model

Compute Express Link (CXL) is an interconnect offering high-bandwidth, low-latency connectivity between host CPUs, heterogeneous memory, and I/O devices running on PCIe 5.0 and 6.0. The CXL.mem protocol allows a host CPU to access CXL memory at cacheline granularity.

There are hardware prototypes of CXL standard 1.1 [52]. The CXL 3.1 specification [1] allows CXL memory to be accessed by multiple hosts concurrently and coherently. Current proposals can scale to 16 connected hosts without undue latency and bandwidth penalties [43].

Performance assumptions. Table 1 shows the latency and bandwidth of the CXL memory device we use for evaluation. Both local DRAM and our CXL memory device are based on DDR5 4800 DRAM. In our system, local DRAM has 8 memory channels with a maximum bandwidth of 8×38.4 GB/s, whereas the CXL memory module has a single memory channel connected to a PCIe 5.0 x8 link. We measure the latency and bandwidth values of local DRAM and CXL memory using Intel's Memory Latency Checker (MLC) [29]. In the table, we present unloaded latency and bandwidth with a 3:1 read-write ratio. CXL has higher read latency and lower bandwidth than local DRAM. The CXL memory access latency and bandwidth are in line with the latest study on characterizing commercial CXL memory devices [52].

Coherence and consistency. This work assumes that CXL memory is as coherent and consistent as local DRAM, with hosts and devices participating in all hardware coherence traffic [1]. It now seems likely that CXL devices will only support hardware cache coherence on a small part of the memory space, which will require greater porting effort for software-based coherence [32].

For our CXL memory measurements, we use a CXL 1.1 module attached to a single machine, so its single-machine cache coherence is emulating what will be multi-machine cache coherence when CXL 3.1 memory becomes available. The cost of cache coherence for CXL memory on real CXL pod hardware will likely exceed the cost of single-node coherence. However, we currently have no way to model the multi-node coherence cost that will characterize this future hardware. Our experiments do use real CXL hardware, not a NUMA memory node, so our emulation has higher fidelity than some recent work [43, 46].

Global persistent flush. For Lupin's evaluation, we assume global persistent flush (GPF) [50] which guarantees that a processor has enough energy to write the data from all committed instructions back to memory even if the power fails, alleviating programmers from having to explicitly flush data into the hardware's persistence domain. We think GPF is likely given Intel's history with persistent memory. Initially, Intel required explicit cache flush instructions and a pcommit instruction to ensure durability for persistent memory [8]. It deprecated pcommit in 2016, and then announced extended asynchronous DRAM Refresh (eADR) [27] in late 2020, which removed the requirement of explicit cache flush operations. GPF is similar to eADR at the microarchitectural level, but eADR is specific to persistent memory and does not apply to CXL memory. A recently announced CXL 3.0 device consisting of DRAM and NAND flash says it will support GPF [20].

2.2 Performance constraints on recovery

To determine the design constraints on Lupin, we measure the restart times of processes: Cavalia [60] is 1.5ms, memcached [6] is 1.1ms, and the Apache http server [5] is 12.3ms. We conclude that some applications can tolerate 1-15ms for a thread or process to restart, while others should provide

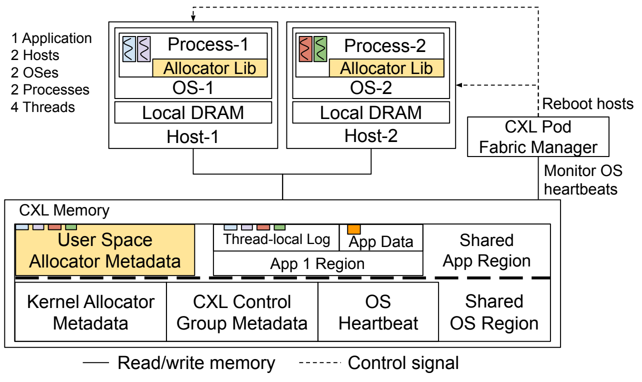

Figure 1. Architectural overview of Lupin. The OS heartbeat section has all the instance identifiers (§3.1) and generation numbers for all OSes and applications using shared CXL.

cooperative recovery where a live thread recovers the state of the dead thread.

We measure the restart times of operating systems: KVM virtual machine is 79s, Intel SPR (local server) is 147s, and Intel ICX (cloud server) is 137s. OS restart takes far too long to pause application progress and the times are long enough to motivate pod-wide rescheduling decisions.

We summarize the performance constraints on Lupin.

- Applications can wait for a thread/process restart, or its threads/processes can cooperate for faster recovery.
- Applications cannot wait for operating system reboot. Applications should be notified about OS reboot and might want to take scheduling action (e.g., create a process on a non-failed node).

3 Detecting, notifying and recovering from faults

Figure 1 shows an overview of Lupin with one application running on two hosts, each with its own OS. Each OS runs a single process inside of which are two threads. The processes are linked with an allocation library whose metadata is in shared CXL memory. Processes can use local DRAM for data structures or they can use shared CXL memory. Each thread maintains a log in CXL that records operations for recovery. Shared CXL memory is split between a section used by the application (above the dashed line) and a section used by the OS (below the dashed line). The Lupin OS uses standard page tables to provide protection–applications can read and write their CXL heap shared between hosts, but cannot read and write the CXL heap of other applications or the kernel.

Beside application state, the Lupin OS uses CXL memory for kernel data structures shared between OSes on different hosts (depicted below the dashed line in Figure 1). The OSes support failure notification and detection by maintaining group information for applications executing on more than one host (§3.4). The OSes share an area for a heartbeat area monitored by the CXL Pod fabric manager for fast failure detection (§3.3). The fabric manager can power cycle hosts in the CXL pod through a baseboard management controller [1, 53].

Lupin provides kernel-level CXL memory allocation. A user-level memory allocator that supports detectable execution is beyond the scope of this work.

3.1 Instance identifiers

All resources in Lupin are owned by instance ids. Instance identifiers allow operating systems and user-level threads to reason about failures. In our prototype, we assign unique thread ids via program arguments. These arguments are still available if a process needs to be restarted because of a partial failure, so thread ids are stable across partial failures. They do not need to be stable across different application instances (where the entire application is restarted). If an application wants to avoid malicious processes masquerading as recovering threads, the OS provides a cross-host process control group that can check permissions (see §3.4).

The fabric manager (or a trusted administrator) assigns unique `ids` to each OS (§2) that remains fixed across reboots. There are many methods the fabric manager could use to generate these ids, like physical port numbers or MAC addresses. An instance identifier (`ii`) is the combination of the unchanging `ii.id` and a monotonic generation number `ii.gen` which is incremented at least once for every failure.

The current generation numbers for all `id`'s are stored in CXL memory, and any OS or process can read them (denoted by `generation[id]`). If a resource (like a lock) is owned by an instance identifier `ii`, and a thread finds that `ii.gen != generation[ii.id]`, then it knows that the instance that owned this resource has failed. For `ii`s stored in data structures (like locks), if we use 32-bit `ids` and 32-bit `iis`, then a 64-bit load/store on any architecture can read/write both quantities atomically.

3.2 CXL memory mapping (offset pointers)

CXL memory regions can be mapped to different addresses in user memory during different executions (as is the case with persistent regions [19, 57]). Lupin adopts offset pointers [17] to make CXL-resident data structures position-independent. An offset pointer stores the offset between the pointer location and the object it points to, which means that non-CXL code can be ported to run in CXL simply by changing all pointer types to offset pointers.

3.3 Failure detection

Detecting thread/process failures. In Lupin, each OS can detect the failure of a local process or thread within that process, as is standard in Linux. Lupin does not track user-level coroutines, it only supports kernel-level threads (also known as light-weight processes–they execute user code but are scheduled by the kernel).

Detecting OS failures. Lupin detects OS failure using heartbeats in shared CXL memory, which are monitored by the

CXL pod fabric manager [1]. All OSes increment their heartbeat counter in CXL memory (every 8ms in our prototype). The fabric manager polls the heartbeat (every 16ms in our prototype). We assume the fabric manager can reset any machine in the CXL pod, which allows it to be a reliable failure detector. If an OS misses a heartbeat, the fabric manager power cycles the failed (or slow) OS and increments its generation number The reset ensures that the currently running OS has been terminated and a new OS instance is booting.

Killing an OS when we are unsure if it is slow or failed is a technique used by previous reliable failure detectors [26, 41]. After the fabric manager restarts a host, it notifies the other operating systems of the failure via a message signaled interrupt (MSI) sent to all OSes via the PCIe bus. The MSI is a low latency notification so remaining OSes can start recovery, but it can be masked or lost–other OSes do not need to receive the MSI to remain correct. Lupin can deploy primary and backup fabric managers to guard against fabric manager failures [30]. Fabric managers can define their own heartbeat protocol, but if the manager reads and copies counters incremented by each OS, then a failure of the primary fabric manager can only delay the detection of a true OS failure, it cannot create false positives.

3.4 Failure notification

CXL control groups (CxlCG). A group of processes sharing CXL memory benefit from additional system services for the group, so Lupin adds a system call to create and configure a CXL control group (CxlCG). Security is beyond the scope of this paper, but creating and joining a CXL control group will require credentials that allow admission control.

Membership. A cross-host process group can be informed of membership changes–a thread or operating system in the group has exited or rejoined. As explained in Section 3.3, Lupin detects the failure of threads and operating systems. If a CxlCG member thread abruptly terminates, that process' operating system will notify the group. As soon as an OS detects another OS's failure, it notifies the group about the OS failure and the failure of all user threads running on that OS.

Scheduling in response to failure. Lupin provides a simple library to make control plane decisions based on failures. An application can use the library to create a control thread that responds to local thread failures by restarting the thread. If the application needs to tolerate process failures, it creates a control process that can restart the data processing process if it fails. If the control process fails, an OS service like `systemd` on Linux can restart it. This mode of operation is similar to how system daemons attain high availability. Lupin applications can also take action on operating system failures, for example, rescheduling all threads on OSes that have not failed.

3.5 Failure recovery (locks and applications)

While the literature on recoverable locks is extensive, we found that most of the designs placed too much emphasis on fairness over efficiency. They also differ in recovery assumptions. For example, the Katzan and Morrison [36] lock requires a "capsule recovery" where the program's entire memory state must be restored for recovery. This is infeasible for an operating system.

Lock recovery in Lupin is simpler than in some published designs because global persistent flush (§2.1) ensures that all successful atomic operation are globally visible and Lupin has reliable failure detection based on the CXL fabric manager.

Lupin modifies two simple and popular locks to be recoverable: a test-and-test-and-set (TATAS) lock and the MCS queuing lock [47]. In both cases, Lupin stores the lock owner's instance ID in the lock structure and in the MCS case we use a published design [13]. The TATAS lock does not provide fairness, while the MCS lock provides first-come-first-served ordering.

Lupin assumes that every thread and OS has a log in a known location in CXL memory that will be available during recovery. The log will identify all locks that were held when the thread or OS failed.

Application recovery. PARSEC and MapReduce share recovery code. We structure all these applications to partition their inputs into work units and compare-and-swap results into a results data structure. Recovered threads simply continue processing input work units. Re-execution of work units might do useless work, but the compare-and-swap for the results ensure that all input is processed exactly once. In our prototype, we find that tasks executing for at least 5 ms do not create significant scheduling overhead while minimizing wasted work.

3.6 Kernel-level memory allocation

The kernel-level allocator is used relatively infrequently as its role is to provide large chunks of memory that are allocated at a fine-grain by the user-level memory allocator. Lupin uses a single, global, recoverable lock, trading parallelism for ease of recovery.

The allocator is based on a buddy allocator [4] that stores allocation metadata in a binary tree encoded in a bitset. The following changes are sufficient to allow recovery after the kernel has rebooted.

- Protect metadata with a global recoverable lock (§3.5).
- Implement transactional semantics by adding a redo log in CXL memory for each OS. Mechanically convert each operation from a sequence of direct memory writes into log entry appends, concluding with a checksum commit entry before executing the log.
- Add a recovery code path that confirms lock ownership and re-executes the log.

Lock transfer. Lupin protects the kernel memory allocator with a single recoverable lock, so it can atomically transfer lock ownership to a different OS to increase availability. The fabric manager sends the MSI interrupt only after it makes sure that the failed or slow OS has reset and is rebooting. All live OSes that gets the MSI signal will try to gain ownership of the lock by atomic compare swap on the instance identifier in the lock. If other OSes mask the interrupt or ignore it, the failed OS itself will eventually recover. Only one OS will successfully complete the lock transfer, and it will complete or abort the in-progress action started by the failed OS, based on its log.

4 Implementation

CxlCG kernel/user communication. Our Lupin prototype uses netlink for group notification and communication, so processes that want timely notification need to have a thread dedicated to blocking on the netlink socket, waiting for communication. Netlink is not a reliable transport [38], but the OS will know if a message cannot be delivered, so it will retry.

MapReduce. MapReduce originates in distributed computing [22]. However, Phoenix [48] and Metis [45] show that MapReduce is also an efficient paradigm for structuring shared memory applications. A recent comprehensive survey of synchronization in shared memory applications [25] identifies MapReduce as a popular framework for writing shared-memory, parallel programs. Shared-memory MapReduce does not tolerate partial faults, but the Lupin port does tolerate partial faults using CXL-resident crash consistent hash tables.

The Lupin port of MapReduce is based on CXL-MapReduce [64], a single-machine, shared-memory MapReduce framework similar to Phoenix [48]. For simplicity, our MapReduce (and CXL-MapReduce) does not sort or merge final results like Phoenix.

CXL-MapReduce [64] only supports a fixed-number of tasks (e.g., one task for each worker) and thus does not support dynamic load balancing. Inspired by Metis [45], we extend it to use a global task list for work distribution. Only WordCount and KMeans run on CXL-MapReduce, so we also extend it to support Matrix Multiply and Histogram.

5 Evaluation

Our evaluation machine has the following configuration, which uses a commercial, physical CXL memory module:

- OS: Ubuntu 22.04.2 LTS (Linux kernel v5.19)
- CPU: 2× Intel® Xeon 8460H CPUs @2.2 GHz [28] 40 cores and 105 MB LLC per CPU
- RAM: 8× DDR5-4800 channels on each socket (16 in total), 1× DDR5-4800 CXL memory with PCIe 5.0 ×8
- NIC: BlueField-2 ConnectX-6 Dx, 100 Gbps

We disable hyper-threading and turbo boost and set the CPU frequency governor to performance. Unless otherwise specified, we use 16 VMs (each with 2 CPU cores) to emulate 16 machines connected to a shared CXL memory module, which is our model of a CXL pod.

	DRAM (μs)		CXL (μs)	
	mean	std	mean	std
TATAS	5.43	2.27	5.42	2.34
recoverable TATAS	5.50	2.37	5.56	2.33
MCS (qspin)	7.42	0.11	7.49	0.21
recoverable MCS [13]	7.86	0.17	8.09	0.07
JJ [34]	99.97	4.38	95.71	0.21

Table 2. Measure the overhead of adding recoverability to a spin lock and an MCS lock. The locks guard a shared counter contended by 32 threads. Mean is the mean of 10 trials and std is the standard deviation. TATAS stands for test-and-test-and-set.

Porting effort. Porting applications to build on Lupin took two days. MapReduce and PARSEC use a lock-free singly linked list and hash table, and they have identical recovery code. Using these data structures took one day for PARSEC and three days for MapReduce.

Recoverable locks. Table 2 compares the performance of Lupin's recoverable locks with their original implementation. The MCS lock is ported from the Linux kernel to userspace [3]. The experiments are run at user-level with 32 threads incrementing a shared counter a fixed number of times. All threads are pinned to their own core, and we use Linux's SCHED_FIFO scheduler as recommended by the pthread manual [2]. In DRAM and CXL, lock performance is similar, with TATAS having much higher standard deviation than MCS, demonstrating its weak fairness properties. The JJ [34] lock has the high latency characteristic of more complex recoverable locks. We conclude that tracking the owner's instance identifier (§3.1) for recoverability in Lupin is efficient and prefer the recoverable TATAS lock in Lupin.

Failure detection and notification. We measured the latency of Lupin's process failure notification via netlink at an average of 106μ with 20μs standard deviation (from 7 trials). Process failure detection is fast and stable (average 175μs, standard deviation 4.4μs) because the OS tracks process exits.

For OS failure detection, the fabric manager has to spend at least 16ms to detect an OS failure due to its polling interval. Sending an MSI interrupt to one OS takes about 3-20μs [21] and sending MSIs to all remaining OSes takes 45-300μs.

Memory allocation. We evaluate the kernel allocator latency by allocating and freeing 100 objects ranging from 1-16MB. The average latency of allocation is 1.6μs and free is 2.0μs. The latency of the allocation is independent of the size of the allocation, but the overhead to initialize the page tables that map the allocation into the process' address space is proportional to the allocation size.

MapReduce and PARSEC. Table 3 shows that our port of MapReduce (Lupin-MapReduce) has excellent scalability for our CXL pod configuration (the Lupin column) and it

	Phoenix	Metis	CXL-SHM	Lupin-1	Lupin
WordCount	9.54	11.09	2.46	0.67 (28.5×)	0.68 (28.0×)
KMeans	3.99	3.83	4.27	3.78 (29.4×)	3.86 (28.8×)
Matrix Multiply	5.23	1.41	6.48	5.13 (32.4×)	5.24 (31.7×)
Histogram	0.25	0.09	0.19	0.23 (13.2×)	0.25 (12.2×)
BlackScholes				2.24 (27.9×)	2.36 (26.5×)
Dedup				0.49 (23.8×)	0.73 (15.9×)

Table 3. All times are in seconds. Lupin runs in 16 VMs, each with 2 CPUs, the remainder, including Lupin-1 run in 1 VM with 32 CPUs. Phoenix and Metis run in DRAM. CXL-SHM, Lupin-1 and Lupin use the commercial CXL module instead of local DRAM. Parentheses are speedups relative to single-threaded performance.

is only 3.6% slower than the 1 VM case (Lupin-1) using the geometric mean. It outperforms Phoenix, Metis and CXL-MapReduce in all workloads except Histogram, where the performance is bottlenecked by the limited read bandwidth of CXL memory [52]. We measure Histogram executing at near linear speedup on different hardware while consuming 33GB/s, which is more bandwidth than our CXL device can support. By default, Phoenix and Metis sort intermediate keys, which causes their poor performance on WordCount (the input of WordCount contains 1 million keys).

Table 3 shows that the Lupin port of BlackScholes has near linear scalability and similar performance between the single VM (Lupin-1) and the pod (Lupin), Dedup has serial compute sections, so its speedup is limited by Amdahl's law. Both applications have serial initialization phases (parsing the input data and copying it into CXL memory) which we omit from our measurements; we only report the speedup for the concurrent phase.

Tolerating process failures. Table 4 shows the performance impact of process failures on end-to-end runtimes while executing on our emulated CXL pod. We crash 0, 1, and 8 of the processes during execution. The crashes are evenly spaced during the workload. For Lupin-MapReduce, each process crashes right before the commit point, which means the maximum amount of work is lost. The performance consequences, even for high process restart rates, is under 8%. 0% slowdown means that even with failures, the performance of the benchmark was unchanged.

Small tasks reduce the overheads from process failures, but if tasks are too small, Lupin incurs overhead to schedule them. All individual tasks in our benchmarks execute for less than 5 ms, but all of them are long enough, so they do not cause scheduling overheads.

6 Related work

CXL tiered memory management. Recent work explores using CXL to enable memory disaggregation and pooling for improved utilization and reduced costs in datacenter servers [11, 42, 43]. Key research directions include optimizing CXL memory pool configurations for performance and cost savings [46], developing resilient memory managers

Crashes	Word Count	KMeans	Matrix Multiply	Histogram	Black Scholes	Dedup
0	0.68s	3.86s	5.24s	0.23s	2.36s	0.74s
1	0.0%	0.0%	0.0%	0.0%	0.0%	3.4%
8	2.9%	2.5%	4.6%	7.8%	0.0%	6.3%

Table 4. Performance slowdowns for Lupin-MapReduce and Lupin-PARSEC with different numbers of process failures.

and intelligent page placement policies to mitigate CXL's higher access latency [39, 52, 64], and leveraging CXL's expanded memory capacity and bandwidth for large-scale applications [33]. These approaches treat CXL as a memory tier that is not directly visible to user software, though some recent work has looked at how to provide an extensible interface [54, 55].

CXL and partial failures. Other recent work [51, 59, 64] makes the same observation as this work that CXL systems can observe partial failures, with FUSEE [51] and rTX [59] focusing on RDMA and remote memory nodes. CXL-SHM tolerates partial failures for CXL memory, but requires reference counting, whereas Lupin relies on explicit memory management and recoverability.

RDMA. RDMA [7, 14, 23, 35, 58, 63] is similar to CXL memory in that it provides access to greater memory capacity, but CXL memory is cacheable by the CPU and has lower latency (e.g., hundreds of ns [43]) and higher bandwidth. Some work has been done to make RDMA-accessible memory cacheable [14, 58].

Far memory. Several systems provide fault-tolerant far memory via replication [40, 65], which is an orthogonal problem to tolerating application and OS failures within a CXL pod.

Recoverability. The availability of persistent memory has sparked interest in recoverable data structures [9, 10, 18, 24] which inspired our data structure design. None of the published systems support detectable recoverability in the operating system. Persistent memory systems consider any OS failure a total failure. They do not have the notion of multiple, cooperating OSes [15, 19, 57].

7 Conclusion

Lupin introduces software infrastructure to support porting single-host applications to run on multi-host CXL pods with high performance and tolerance of partial failures. Lupin demonstrates that a wide range of applications can be adapted to CXL pods for higher performance, better scalability, and fault tolerance.

8 Acknowledgements

This work was supported in part by the Center for Processing with Intelligent Storage and Memories (PRISM), one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Compute Express Link (CXL) Specification, Revision 3.1. (Accessed: April 2024). URL: https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-3.1-Specification.pdf.

[2] pthread_spin_init(3) — Linux manual page. (Accessed: April 2024). URL: https://man7.org/linux/man-pages/man3/pthread_spin_init.3.html.

[3] queued_spinlock. (Accessed: April 2024). URL: https://github.com/ARM-software/synchronization-benchmarks/blob/master/ext/linux/queued_spinlock.h.

[4] A single header buddy memory allocator for c. (Accessed: Dec 2023). URL: https://github.com/spaskalev/buddy_alloc.

[5] Apache HTTP Server Project. https://httpd.apache.org/, accessed in 2023.

[6] Memcached. https://memcached.org/, accessed in 2023.

[7] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote memory in the age of fast networks. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 121–127, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3127479.3131612.

[8] Andy M Rudoff. Deprecating the PCOMMIT Instruction. https://www.intel.com/content/www/us/en/developer/articles/technical/deprecate-pcommit-instruction.html, 2016.

[9] Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. Detectable recovery of lock-free data structures. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page 262–277, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3503221.3508444.

[10] Naama Ben-David, Guy E. Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, page 253–264, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3323165.3323187.

[11] Daniel S. Berger, Daniel Ernst, Huaicheng Li, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Lisa Hsu, Ishwar Agarwal, Mark D. Hill, and Ricardo Bianchini. Design tradeoffs in cxl-based memory pools for public cloud platforms. *IEEE Micro*, 43(2):30–38, 2023. doi:10.1109/MM.2023.3241586.

[12] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 72–81, New York, NY, USA, 2008. Association for Computing Machinery. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/1454115.1454128, doi:10.1145/1454115.1454128.

[13] P. Bohannon, D. Lieuwen, and A. Silbershatz. Recovering scalable spin locks. In *Proceedings of SPDP '96: 8th IEEE Symposium on Parallel and Distributed Processing*, pages 314–322, 1996. doi:10.1109/SPDP.1996.570349.

[14] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with rdma and caching. *Proc. VLDB Endow.*, 11(11):1604–1617, jul 2018. doi:10.14778/3236187.3236209.

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

[16] Prakash Chauhan, Chris Petersen, Brian Morris, and Jerome Glisse. Hyperscale cxl tiered memory expander specification v1.2, 2023. URL: https://www.opencompute.org/documents/hyperscale-cxl-tiered-memory-expander-for-ocp-base-specification-1-pdf.

[17] Guoyang Chen, Lei Zhang, Richa Budhiraja, Xipeng Shen, and Youfeng Wu. Efficient support of position independence on non-volatile memory. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 191–203, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3123939.3124543.

[18] Kyeongmin Cho, Seungmin Jeon, Azalea Raad, and Jeehoon Kang. Memento: A framework for detectable recoverability in persistent memory. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023. doi:10.1145/3591232.

[19] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, 2011.

[20] Samsung corporation. Samsung cxl solutions – cmm-h. 2024. URL: https://semiconductor.samsung.com/news-events/tech-blog/samsung-cxl-solutions-cmm-h.

[21] Craig Howard. What is a ballpark figure for PCIe interrupt-to-userspace latency?, 2011. (Accessed: Jul 2024). URL: https://community.osr.com/t/whats-a-ballpark-figure-for-pcie-interrupt-to-userspace-latency/41717.

[22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, jan 2008. doi:10.1145/1327452.1327492.

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

[24] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. *SIGPLAN Not.*, 53(1):28–40, feb 2018. doi:10.1145/3200691.3178490.

[25] Rachid Guerraoui, Hugo Guiroux, Renaud Lachaize, Vivien Quéma, and Vasileios Trigonakis. Lock–unlock: Is that all? a pragmatic analysis of locking in software systems. *ACM Trans. Comput. Syst.*, 36(1), mar 2019. doi:10.1145/3301501.

[26] Trinabh Gupta, Joshua B. Leners, Marcos K. Aguilera, and Michael Walfish. Improving availability in distributed systems with failure informers. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 427–441, Lombard, IL, April 2013. USENIX Association. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/leners.

[27] Intel Corporation. eADR: New Opportunities for Persistent Memory Applications. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html, 2021.

[28] Intel Corporation. Intel Xeon Platinum 8460H Processor. https://www.intel.com/content/www/us/en/products/sku/231744/intel-xeon-platinum-8460h-processor-105m-cache-2-20-ghz/specifications.html, accessed in 2023.

[29] Intel Corporation. Intel® memory latency checker v3.10. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html, accessed in 2023.

[30] IntelliProp. Introducing omega fabric based on cxl, 2024. URL: https://www.intelliprop.com/products-page.

[31] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In Cyril Gavoille and David Ilcinkas, editors, *Distributed*

Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings, volume 9888 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 2016. doi:10.1007/978-3-662-53426-7_23.

[32] Sunita Jain, Nagaradhesh Yeleswarapu, Hasan Al Maruf, and Rita Gupta. Memory sharing with cxl: Hardware and software design approaches, 2024.

[33] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. 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)*, pages 585–600, Boston, MA, July 2023. USENIX Association. URL: https://www.usenix.org/conference/atc23/presentation/jang.

[34] Prasad Jayanti and Anup Joshi. Recoverable mutual exclusion with abortability. *Computing*, 104(10):2225–2252, aug 2022. URL: https://doi.org/10.1007%2Fs00607-022-01105-1, doi:10.1007/s00607-022-01105-1.

[35] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, scalable and simple distributed transactions with Two-Sided (RDMA) datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, Savannah, GA, November 2016. USENIX Association. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia.

[36] Daniel Katzan and Adam Morrison. Recoverable, abortable, and adaptive mutual exclusion with sublogarithmic RMR complexity. In Quentin Bramas, Rotem Oshman, and Paolo Romano, editors, *24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14-16, 2020, Strasbourg, France (Virtual Conference)*, volume 184 of *LIPIcs*, pages 15:1–15:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.OPODIS.2020.15.

[37] Kimberly Keeton, Sharad Singhal, and Michael Raymond. The openfam api: A programming model for disaggregated persistent memory. In Swaroop Pophale, Neena Imam, Ferrol Aderholdt, and Manjunath Gorentla Venkata, editors, *OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Extreme Heterogeneity*, pages 70–89, Cham, 2019. Springer International Publishing.

[38] Linux kernel documentation. Kernel connector. Date retrieved: December 2023. URL: https://docs.kernel.org/driver-api/connector.html.

[39] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 17–34, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3600006.3613167.

[40] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. Hydra : Resilient and highly available remote memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 181–198, Santa Clara, CA, February 2022. USENIX Association. URL: https://www.usenix.org/conference/fast22/presentation/lee.

[41] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting failures in distributed systems with the falcon spy network. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 279–294, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2043556.2043583.

[42] Philip Levis, Kun Lin, and Amy Tai. A case against cxl memory pooling. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, HotNets '23, page 18–24, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3626111.3628195.

[43] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 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*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/3575693.3578835, doi:10.1145/3575693.3578835.

[44] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. 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*, ASPLOS '20, page 757–773, New York, NY, USA, 2020. Association for Computing Machinery. URL: https://doi-org.ezproxy.lib.utexas.edu/10.1145/3373376.3378511, doi:10.1145/3373376.3378511.

[45] Yandong Mao, Robert Morris, and Frans Kaashoek. Optimizing mapreduce for multicore architectures. Technical report, Technical Report MIT-CSAIL-TR-2010-020, MIT, 2010.

[46] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3582016.3582063.

[47] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, feb 1991. doi:10.1145/103727.103729.

[48] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Feb 2007. doi:10.1109/HPCA.2007.346181.

[49] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332. USENIX Association, November 2020. URL: https://www.usenix.org/conference/osdi20/presentation/ruan.

[50] Andy Rudoff, Chet Douglas, and Tiffany Kasanicky. Persistent memory in CXL. 2021. URL: https://www.snia.org/sites/default/files/PM-Summit/2021/snia-pm-cs-summit-Rudoff-PM-in-CXL-2021.pdf.

[51] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. FUSEE: A fully Memory-Disaggregated Key-Value store. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 81–98, Santa Clara, CA, February 2023. USENIX Association. URL: https://www.usenix.org/conference/fast23/presentation/shen.

[52] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. Demystifying CXL memory with genuine CXL-ready systems and devices, 2023.

[53] Inc Super Micro Computer. What is a Baseboard Management Controller? (BMC), 2024. (Accessed: Jul 2024). URL: https://www.supermicro.com/en/glossary/baseboard-management-controller.

[54] Bijan Tabatabai, Mark Mansi, and Michael M. Swift. FBMM: using the VFS for extensibility in kernel memory management. In Malte Schwarzkopf, Andrew Baumann, and Natacha Crooks, editors, *Proceedings of the 19th Workshop on Hot Topics in Operating Systems, HOTOS 2023, Providence, RI, USA, June 22-24, 2023*, pages 181–187. ACM, 2023. doi:10.1145/3593856.3595908.

[55] Bijan Tabatabai, James Sorenson, and Michael M. Swift. FBMM: Making memory management extensible with filesystems. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 785–798, Santa Clara, CA, July 2024. USENIX Association. URL: https://www.usenix.o

rg/conference/atc24/presentation/tabatabai.

[56] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated Key-Value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 33–48. USENIX Association, July 2020. URL: https://www.usenix.org/conference/atc20/presentation/tsai.

[57] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, 2011.

[58] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed shared memory with {In-Network} cache coherence. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 277–292, 2021.

[59] Xingda Wei, Haotian Wang, Tianxia Wang, Rong Chen, Jinyu Gu, Pengfei Zuo, and Haibo Chen. Transactional indexes on (rdma or cxl-based) disaggregated memory with repairable transaction, 2023. arXiv:2308.02501.

[60] Yingjun Wu and Kian-Lee Tan. Scalable In-Memory transaction processing with HTM. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 365–377, Denver, CO, June 2016. USENIX Association. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/wu.

[61] Yu Xia, Xiangyao Yu, Andrew Pavlo, and Srinivas Devadas. Taurus: lightweight parallel logging for in-memory database management systems. *Proc. VLDB Endow.*, 14(2):189–201, oct 2020. doi:10.14778/3425879.3425889.

[62] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 478–496, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3132747.3132761.

[63] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 51–68, Santa Clara, CA, February 2022. USENIX Association. URL: https://www.usenix.org/conference/fast22/presentation/zhang-ming.

[64] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. Partial failure resilient memory management system for (cxl-based) distributed shared memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 658–674, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3600006.3613135.

[65] 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. Carbink: Fault-Tolerant far memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 55–71, Carlsbad, CA, July 2022. USENIX Association. URL: https://www.usenix.org/conference/osdi22/presentation/zhou-yang.