Lupin: Tolerating Partial Failures in a CXL Pod

Zhiting Zhu, Newton Ni, Yibo Huang, Yan Sun, Zhipeng Jia, Nam Sung Kim, Emmett Witchel

Lepton AI

TEXAS
The University of Texas at Austin

Google

ILLINOIS

Single-host software vs. distributed software

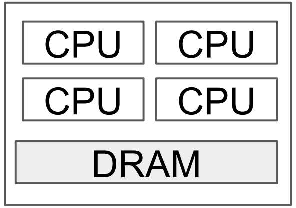
One Host

Distributed (many hosts)

Single-host software vs. distributed software

One Host

- Shared mutable state
- Centralized state
- Many efficient algorithms
- Limited scalability
- Database
- In memory MapReduce

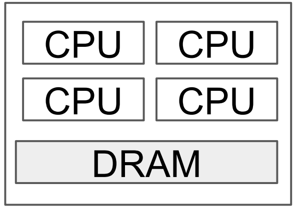


Distributed (many hosts)

Single-host software vs. distributed software

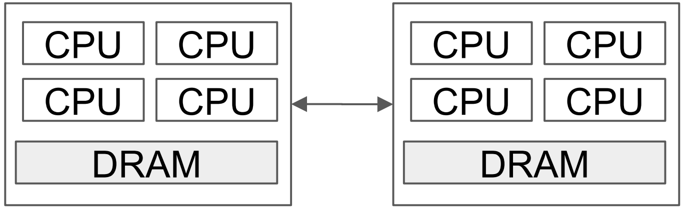
One Host

- Shared mutable state
- Centralized state
- Many efficient algorithms
- Limited scalability
- Database
- In memory MapReduce



Distributed (many hosts)

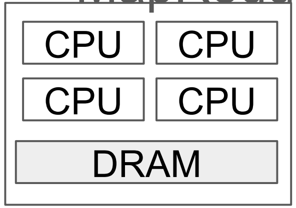
- Partitioned state
- Scalable
- Fast failover
- Difficult to construct and maintain (performance)
- Key-value store
- MapReduce



Single-host software vs. distributed software

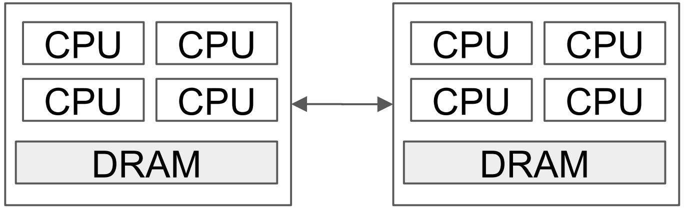
One Host

- Shared mutable state
- Centralized state
- Many efficient algorithms
- Limited scalability
- Database
- In memory MapReduce



Distributed (many hosts)

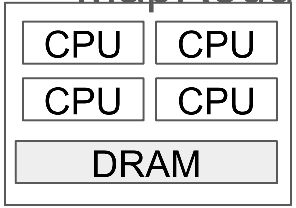
- Partitioned state
- Scalable
- Fast failover
- Difficult to construct and maintain (performance)
- Key-value store
- MapReduce



Single-host software vs. distributed software

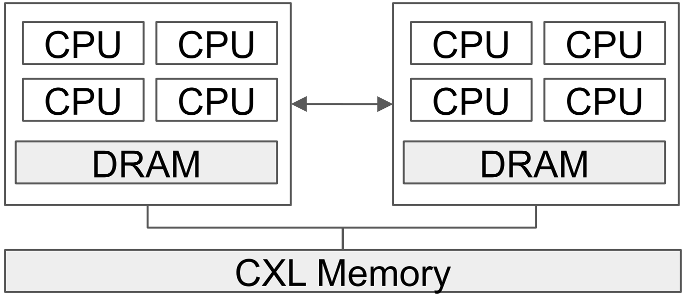
One Host

- Shared mutable state
- Centralized state
- Many efficient algorithms
- Limited scalability
- Database
- In memory MapReduce



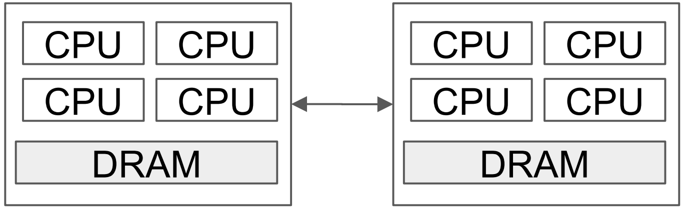
CXL Pod

- Machines connected to CXL memory

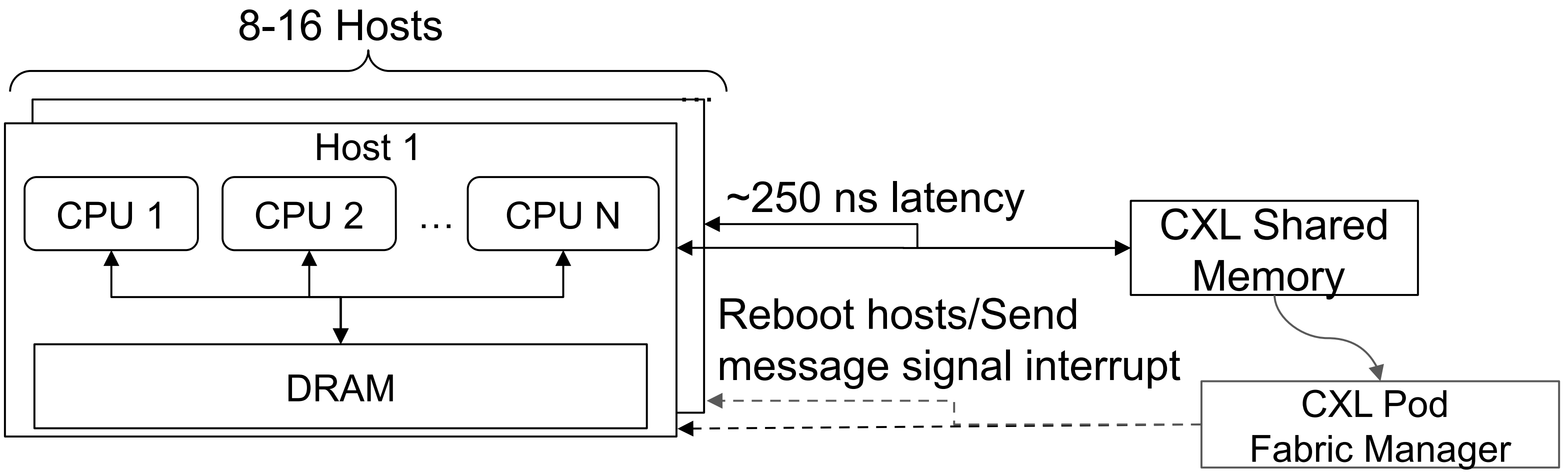


Distributed (many hosts)

- Partitioned state
- Scalable
- Fast failover
- Difficult to construct and maintain (performance)
- Key-value store
- MapReduce



CXL memory accessible to multiple hosts via PCIe

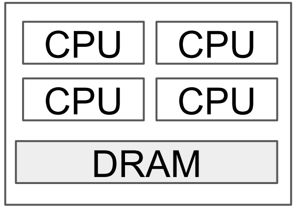


- 8-16 Hosts physically connected to a CXL memory module
  - CXL 3.1 allows fine-grained memory sharing
  - Multi-host HW cache coherence on entire physical CXL memory
    - Probably not realizable
  - Pod fabric manager is control software

A tale of two climates

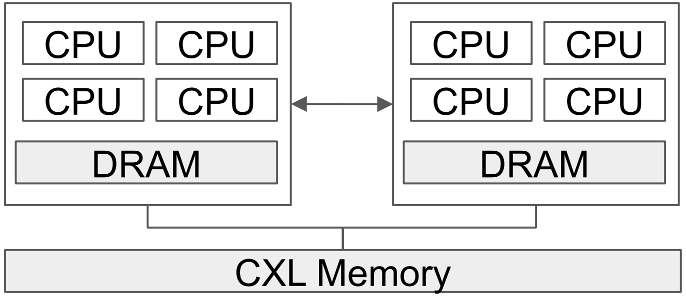
One Host

- Shared mutable state
- Centralized state
- Many efficient algorithms
- Limited scalability
- Database
- In memory MapReduce



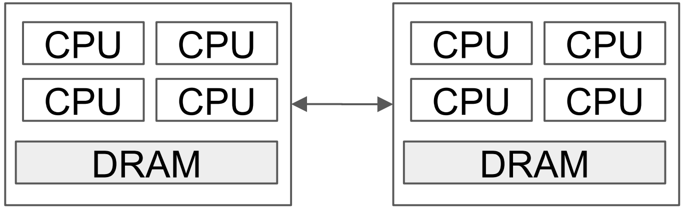
CXL Pod

- Reuse efficient single host algorithms
- Shared state across machines
- Low tail latency



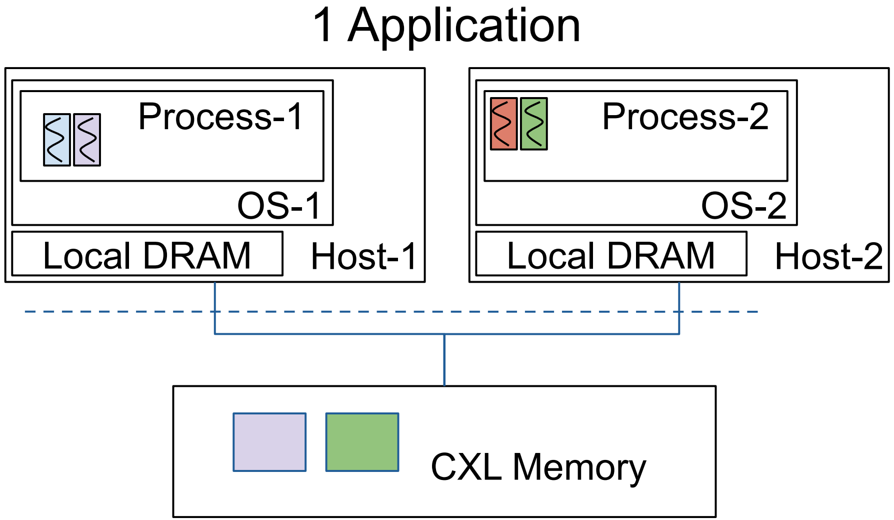
Distributed (many hosts)

- Replicated state machines
- Scalable
- Fast failover
- Difficult to construct and maintain (performance)
- Key-value store
- MapReduce



What application will benefit from a CXL pod?

- A shared-memory MapReduce
  - High performance
  - Limited scalability by single host



Challenges: Efficiently tolerate partial failure

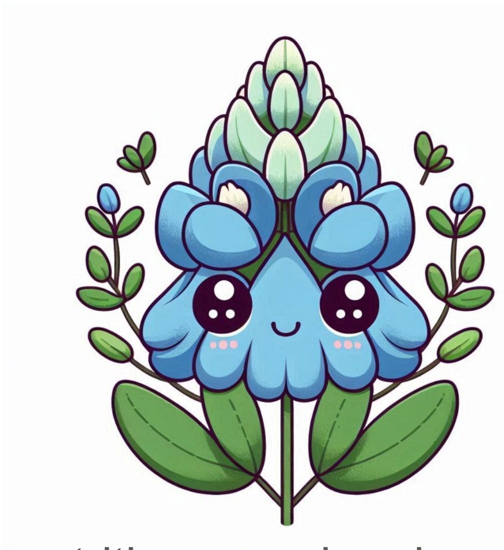
- Partial failure
  - One or more process/OS dies
  - Other processes or OSes remain live
- Efficiently tolerate partial failure
  - Do I have to restart all OSes (or all processes)?
  - Full restart is bad for availability
  - OS reboot takes minutes (79s - 2.5 mins)

Challenges: Correctly tolerate partial failure

- Shared data structures go in shared CXL
  - Shared data structures need synchronization
- OSes & applications have to synchronize on CXL memory
  - Spinlocks, futexes, mutexes, semaphores are not fault-tolerant
  - Die with a lock held ⇒Deadlock
- Recovery needs to ensure input are processed exactly once
  - Duplicated output/update
  - Missing output /update

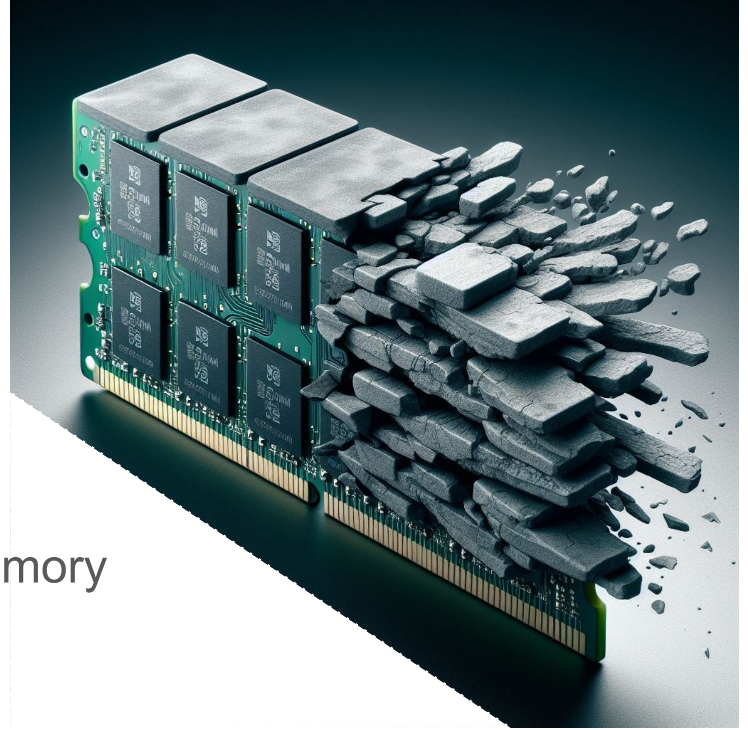
Lupin: Software infrastructure for partial failure tolerance

- Efficiency
  - Applications should remain available during recover
  - Don’t have to pause application or other OS until dead OS reboot
- Correctness
  - No deadlock
  - Recovery needs to ensure that operation executes exactly once



*The Lupin (bluebonnet) is known for its nutritious seed pod

CXL pod partial failure model



- Make CXL memory persistent
  - Give it independent power supply
  - Protect integrity with ECC
- Efficient recovery
  - Application can restore state from CXL memory

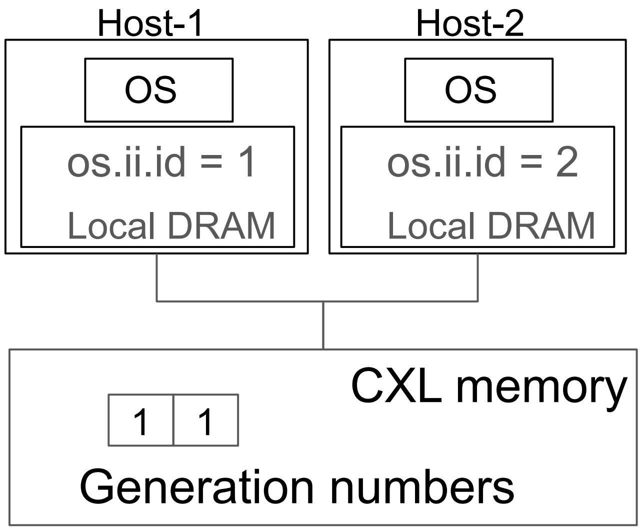
- How do we tolerate partial failure?

Lupin: Software infrastructure for partial failure tolerance

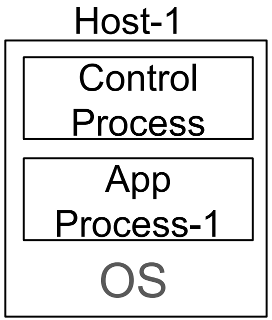
- Failure detection and notification
    - Instance identifier
    - CXL control group
    - Partial failure detection
    - Partial failure notification
- Cooperative recovery
- Partial failure tolerant kernel memory allocation

Instance identifiers

- Instance identifiers (ii) for OS and processes
  - Stable ID (ii.id) +
  - Generation number (ii.gen)
- Resources in Lupin are owned by instance ids
  - E.g., recoverable locks have ii field
  - Current generation numbers in CXL memory
    - Read by any host as generation[ii.id]
  - Instance ids will be useful for recovery

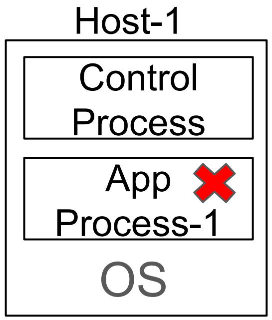


CXL control group (CxlCG)



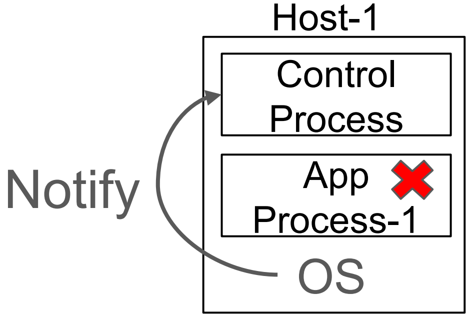
- CXL control groups (CxlCG)
  - A cross-host process group
  - OS data structures in CXL memory
  - Group member can get notification when process dies/rejoins
- Control process and application process
  - Control process handles failure notification

CXL control group (CxlCG)



- CXL control groups (CxlCG)
  - A cross-host process group
  - OS data structures in CXL memory
  - Group member can get notification when process dies/rejoins
- Control process and application process
  - Control process handles failure notification

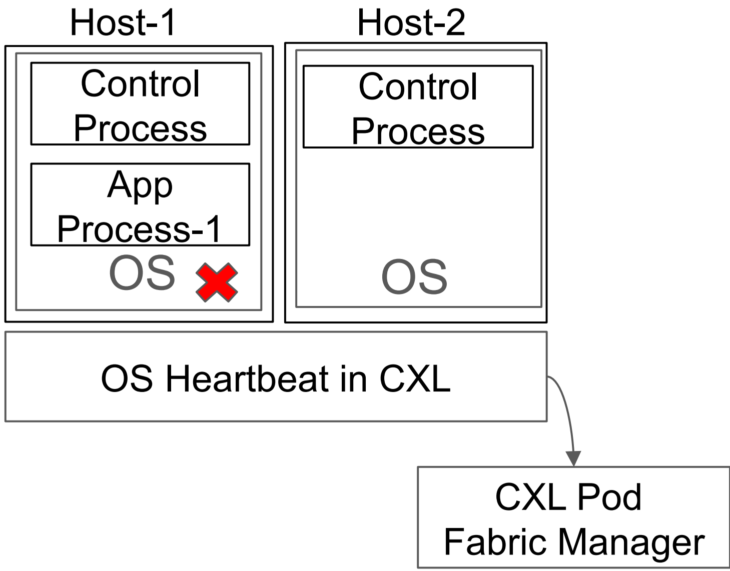
CXL control group (CxlCG)



- CXL control groups (CxlCG)
  - A cross-host process group
  - OS data structures in CXL memory
  - Group member can get notification when process dies/rejoins
- Control process and application process
  - Control process handles failure notification

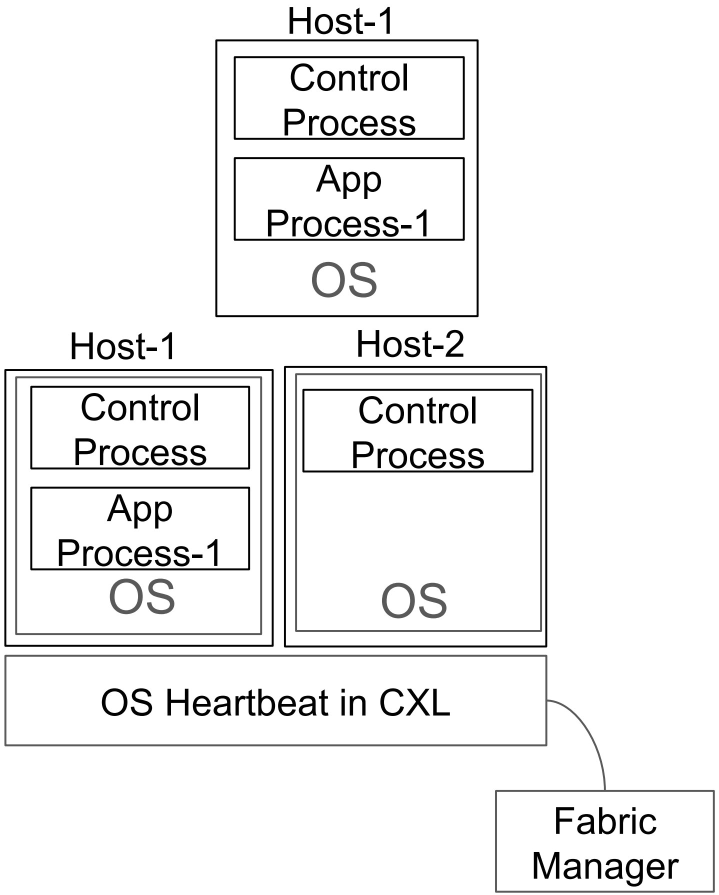
Failure detection

- Process failure detection
  - Already done by OS!
  - OS notifies CXL control group about failures
  - Detection takes: 175 μs
- OS failure detection
  - Heartbeats via CXL memory
  - Fabric manager monitors OS heartbeats
  - Fabric manager power cycles (dead|slow) host
- Fabric manager is reliable failure detector
  - Power cycle makes sure OS is dead!
  - After OS dies, signal other OSes in the pod via message signal interrupt (MSI-X)



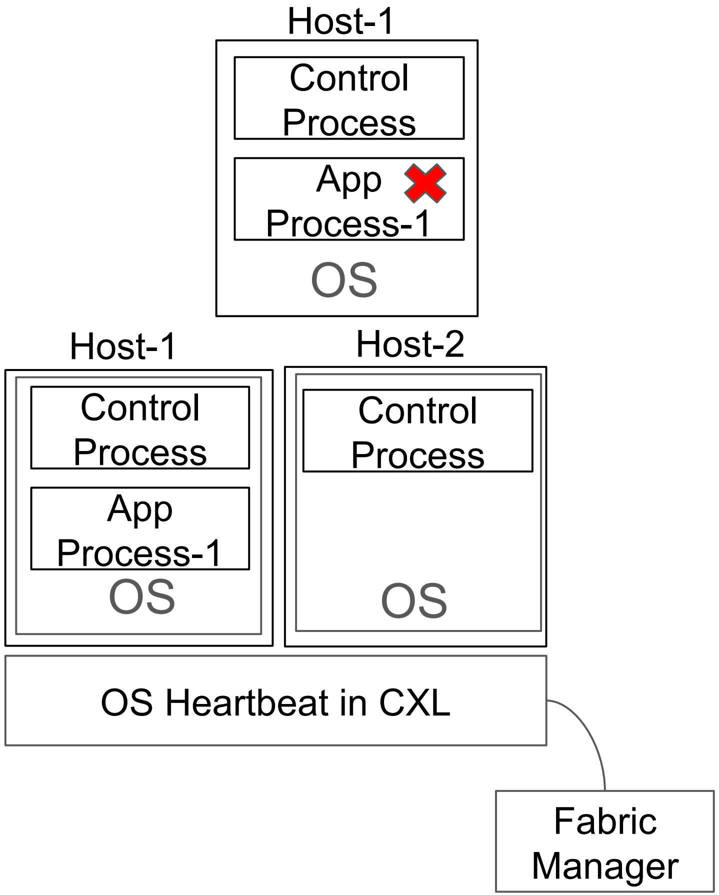
Failure notification

- Notification via CXL control groups (CxlCG)
  - OS talks to process via netlink messages
  - Notification: 106 µs
- Mechanism to notify application
  - Application defines policy
- Control process can restart application process, or migrate



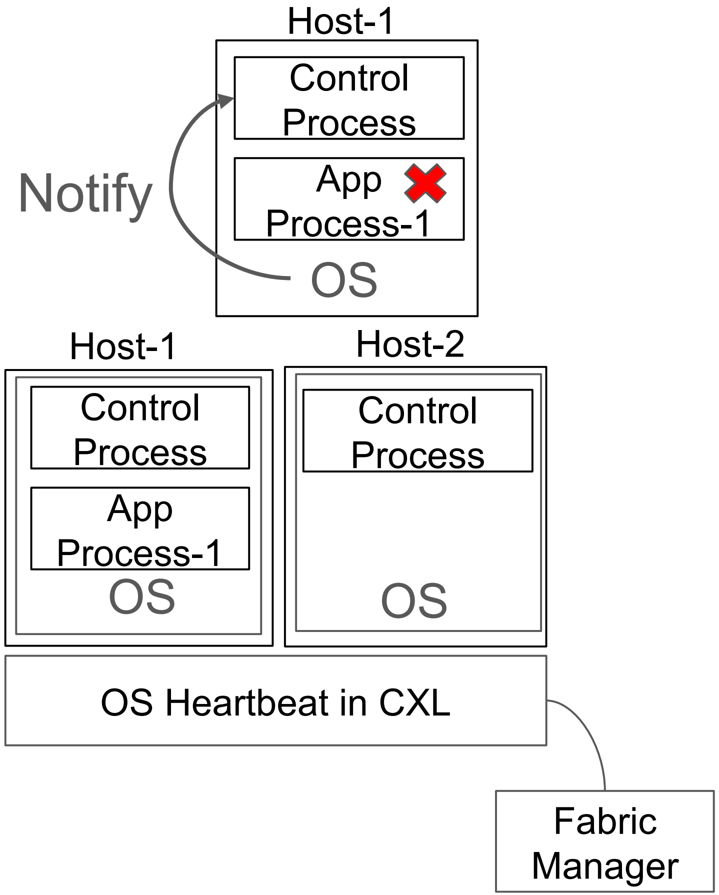
Failure notification

- Notification via CXL control groups (CxlCG)
    - OS talks to process via netlink messages
    - Notification: 106 μs
- Mechanism to notify application
    - Application defines policy
- Control process can restart application process, or migrate



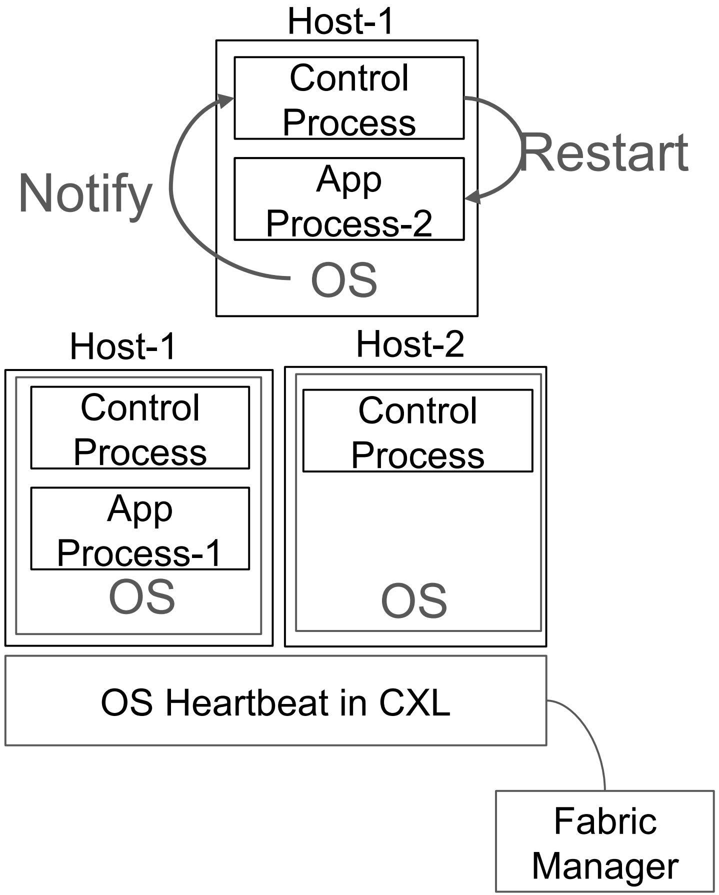
Failure notification

- Notification via CXL control groups (CxlCG)
  - OS talks to process via netlink messages
  - Notification: 106 µs
- Mechanism to notify application
  - Application defines policy
- Control process can restart application process, or migrate



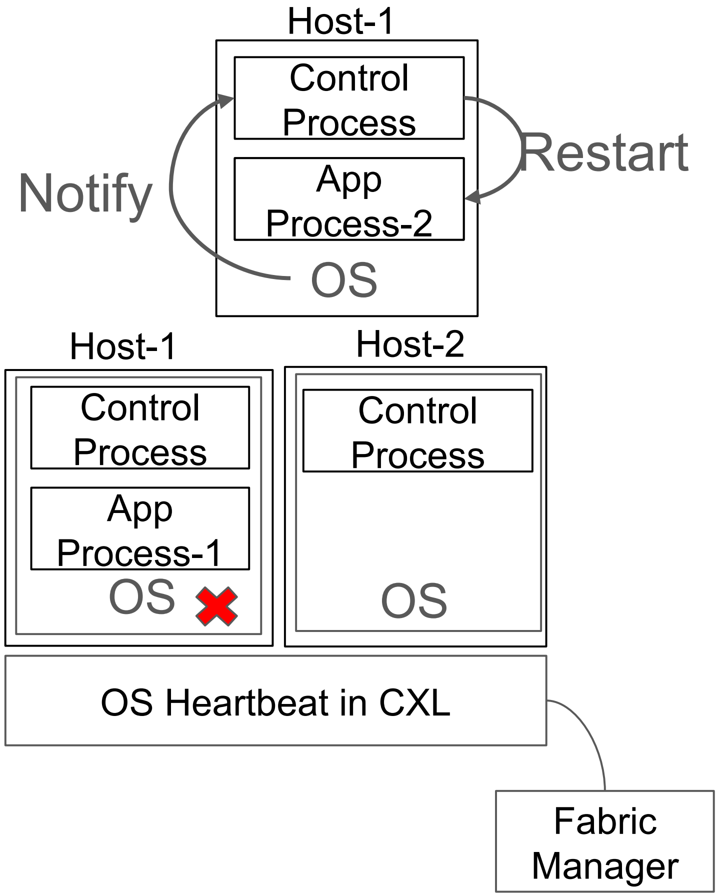
Failure notification

- Notification via CXL control groups (CxlCG)
  - OS talks to process via netlink messages
  - Notification: 106 µs
- Mechanism to notify application
  - Application defines policy
- Control process can restart application process, or migrate



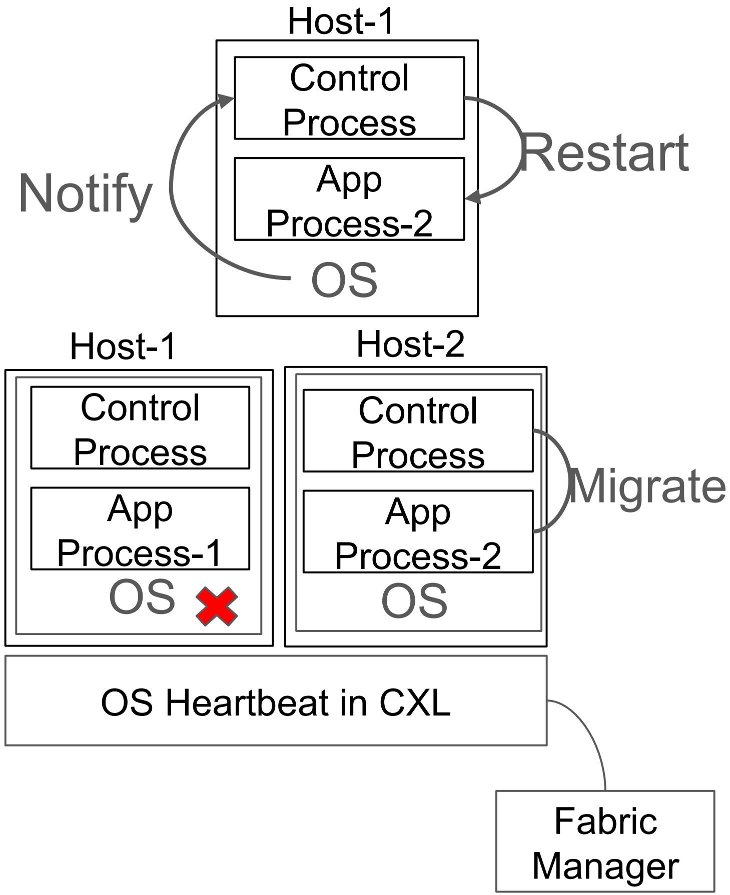
Failure notification

- Notification via CXL control groups (CxlCG)
  - OS talks to process via netlink messages
  - Notification: 106 μs
- Mechanism to notify application
  - Application defines policy
- Control process can restart application process, or migrate



Failure notification

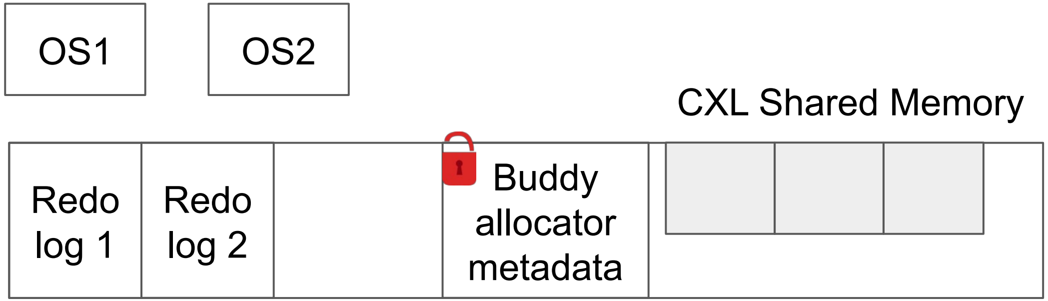
- Notification via CXL control groups (CxlCG)
  - OS talks to process via netlink messages
  - Notification: 106 μs
- Mechanism to notify application
  - Application defines policy
- Control process can restart application process, or migrate



Recovery and cooperative recovery

- Self recovery: OS1 must recover OS1's failure
  - Crashed processes and OSes recover themselves
  - But OS reboot is slow (1+ minutes)
- Cooperative recovery: OS2 can recover OS1's failure
  - Live OS/process recovers the failed process/OS by executing its recovery method
  - Efficiently recover OS without waiting failed OS to reboot
  - Safety: only one OS/process runs the recovery method for a failed OS/process

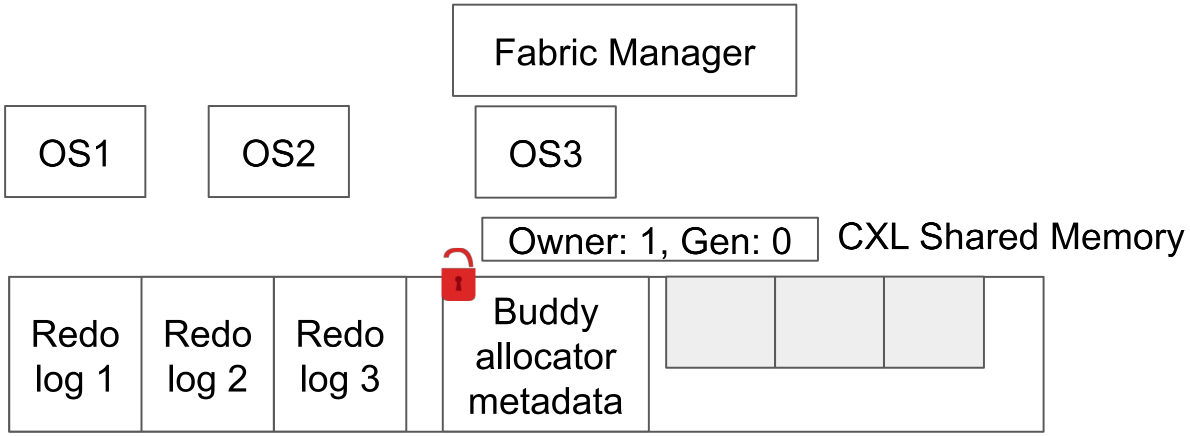
OS memory allocator



- Cooperative allocator for all OSes in pod
- A single recoverable test-and-test-and-set lock protects the metadata
- Atomic recoverable allocation/free with redo log
  - Record the operation, parameters and new values

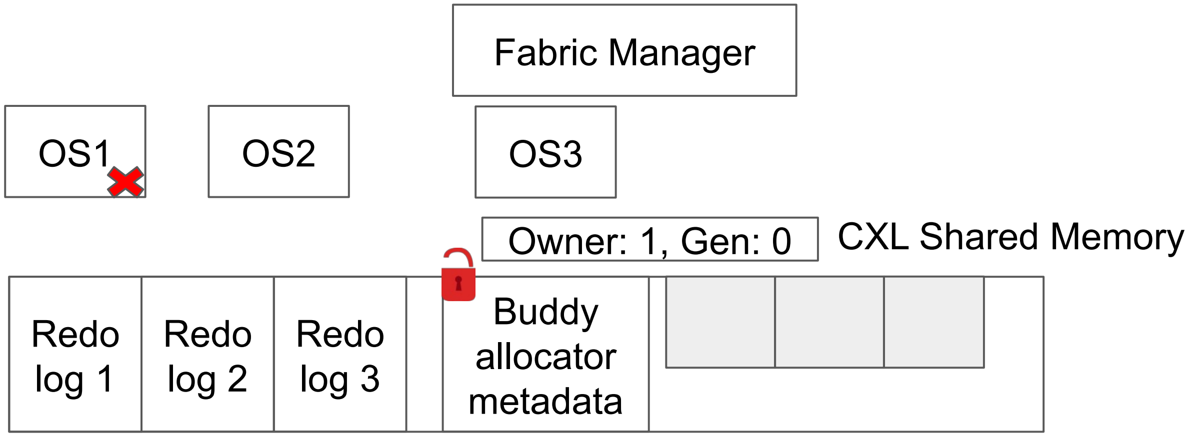
Recovery for kernel memory allocator

- Safe lock transfer algorithm: only one OS can be in the critical section
  - Fabric manager is reliable failure detector
    - Because it power cycles machine before notifying failure
  - Change owner of lock via atomic compare and swap
    - Only one OS will succeed and help complete or abort the current operation



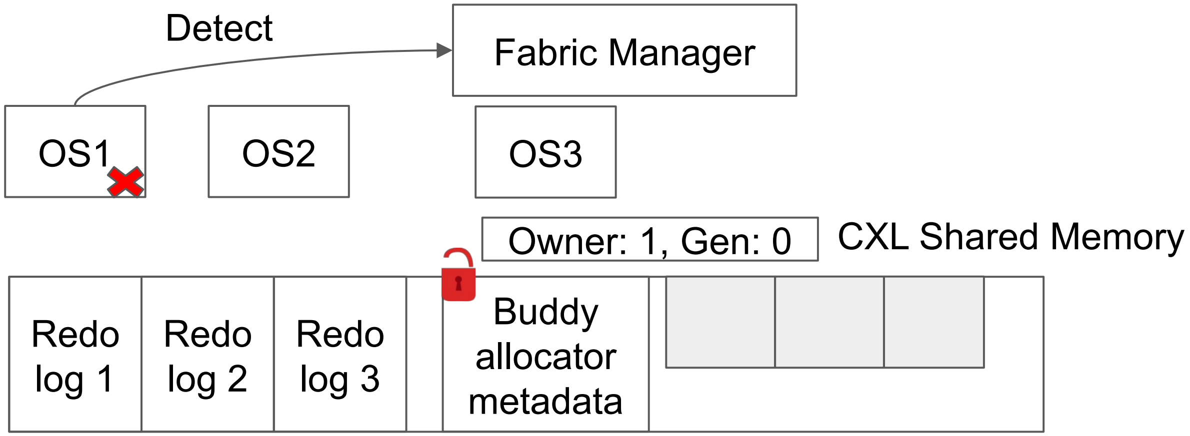
Recovery for kernel memory allocator

- Safe lock transfer algorithm: only one OS can be in the critical section
  - Fabric manager is reliable failure detector
    - Because it power cycles machine before notifying failure
  - Change owner of lock via atomic compare and swap
    - Only one OS will succeed and help complete or abort the current operation



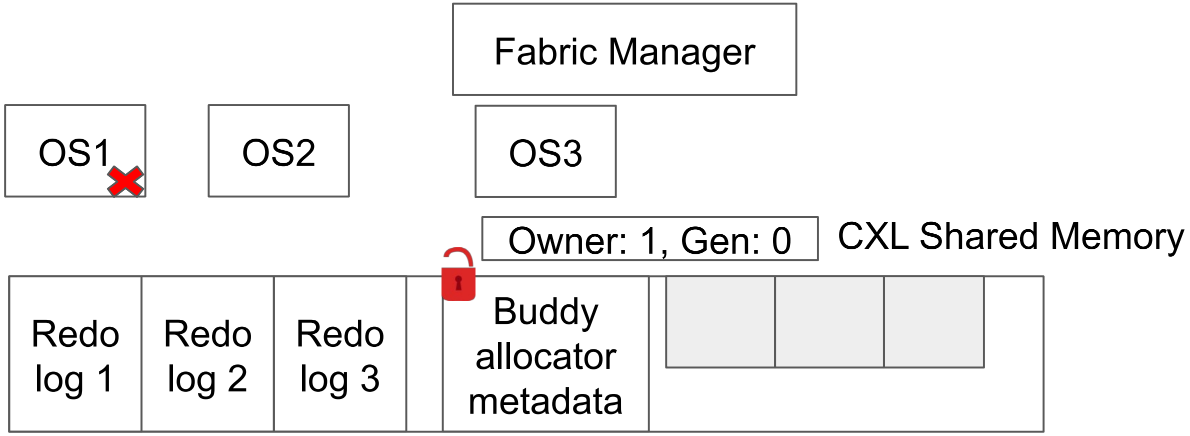
Recovery for kernel memory allocator

- Safe lock transfer algorithm: only one OS can be in the critical section
  - Fabric manager is reliable failure detector
    - Because it power cycles machine before notifying failure
  - Change owner of lock via atomic compare and swap
    - Only one OS will succeed and help complete or abort the current operation



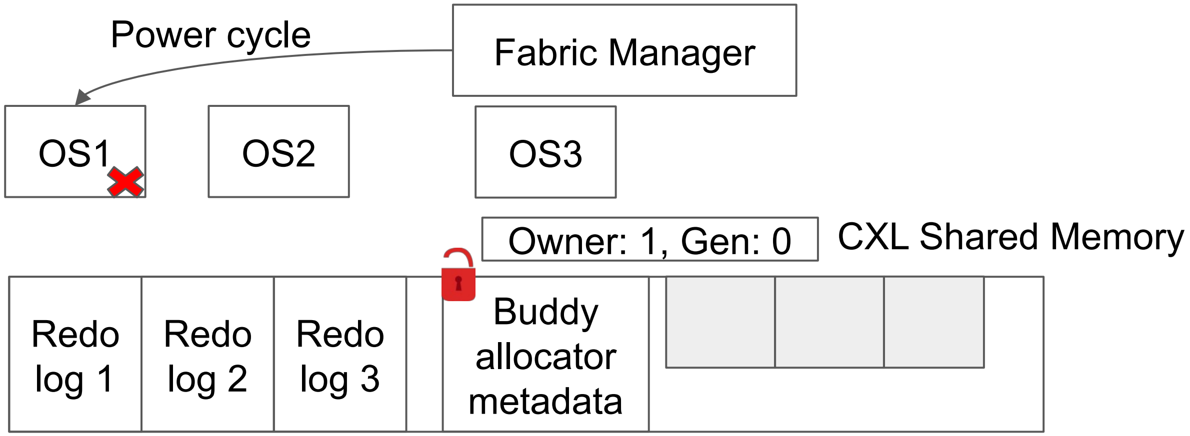
Recovery for kernel memory allocator

- Safe lock transfer algorithm: only one OS can be in the critical section
  - Fabric manager is reliable failure detector
    - Because it power cycles machine before notifying failure
  - Change owner of lock via atomic compare and swap
    - Only one OS will succeed and help complete or abort the current operation



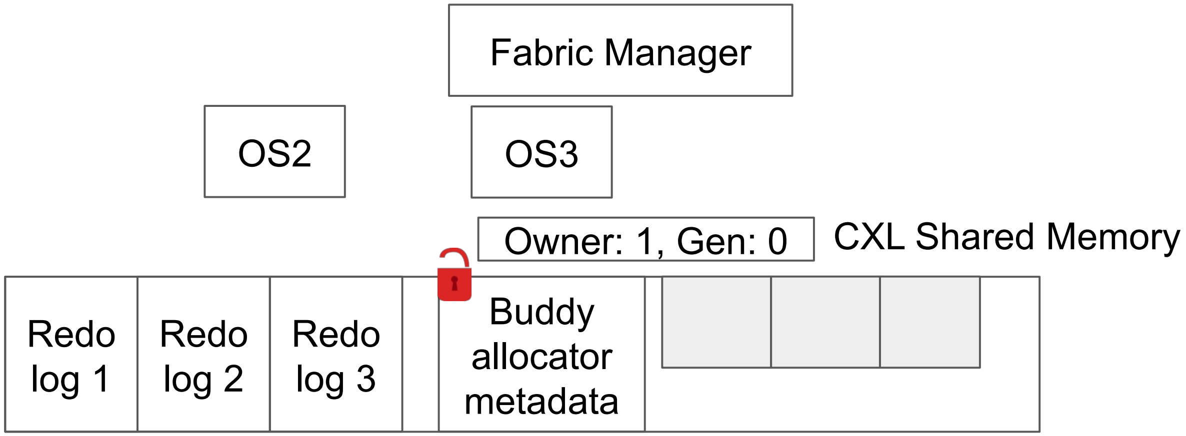
Recovery for kernel memory allocator

- Safe lock transfer algorithm: only one OS can be in the critical section
  - Fabric manager is reliable failure detector
    - Because it power cycles machine before notifying failure
  - Change owner of lock via atomic compare and swap
    - Only one OS will succeed and help complete or abort the current operation



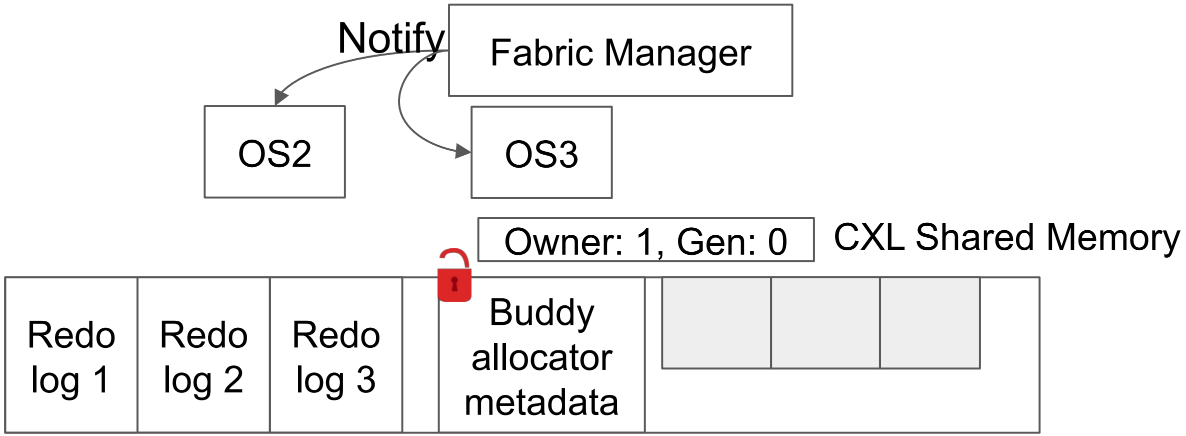
Recovery for kernel memory allocator

- Safe lock transfer algorithm: only one OS can be in the critical section
  - Fabric manager is reliable failure detector
    - Because it power cycles machine before notifying failure
  - Change owner of lock via atomic compare and swap
    - Only one OS will succeed and help complete or abort the current operation



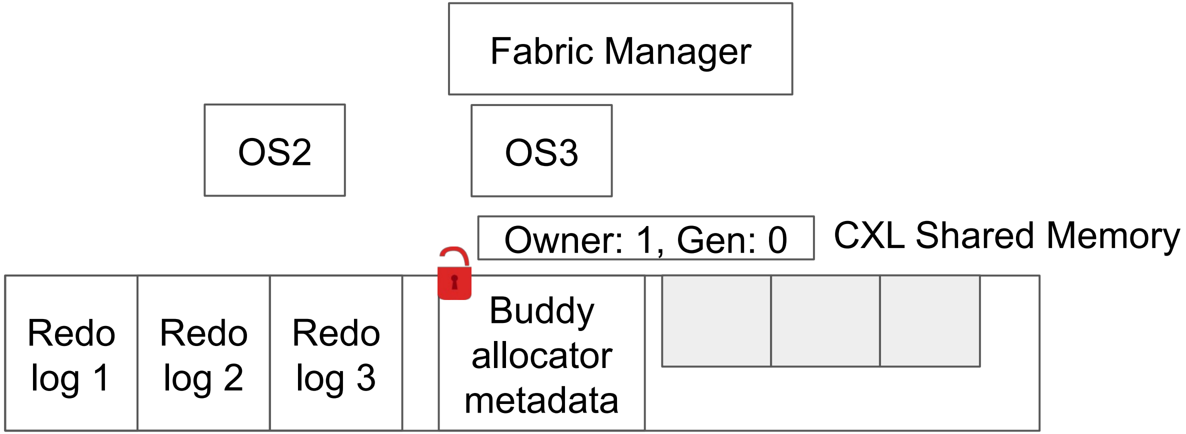
Recovery for kernel memory allocator

- Safe lock transfer algorithm: only one OS can be in the critical section
  - Fabric manager is reliable failure detector
    - Because it power cycles machine before notifying failure
  - Change owner of lock via atomic compare and swap
    - Only one OS will succeed and help complete or abort the current operation



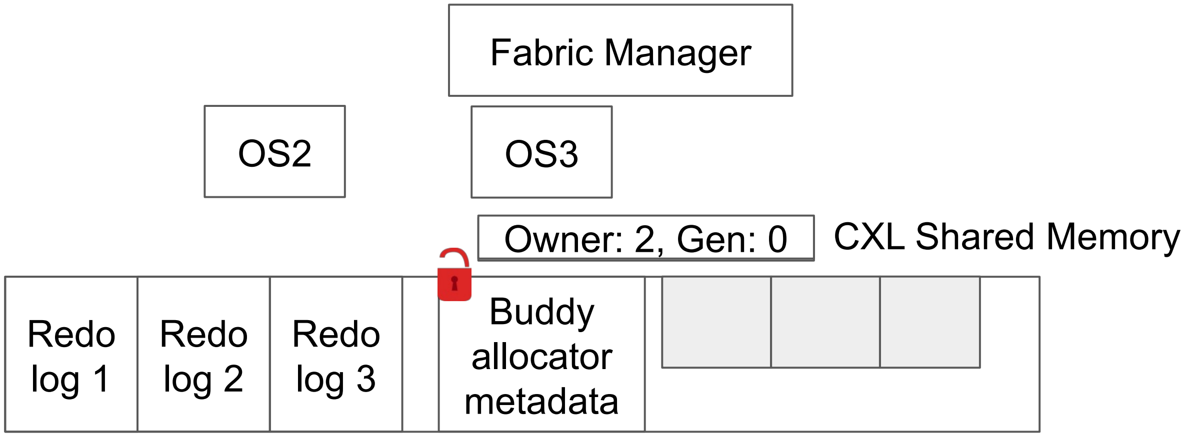
Recovery for kernel memory allocator

- Safe lock transfer algorithm: only one OS can be in the critical section
  - Fabric manager is reliable failure detector
    - Because it power cycles machine before notifying failure
  - Change owner of lock via atomic compare and swap
    - Only one OS will succeed and help complete or abort the current operation



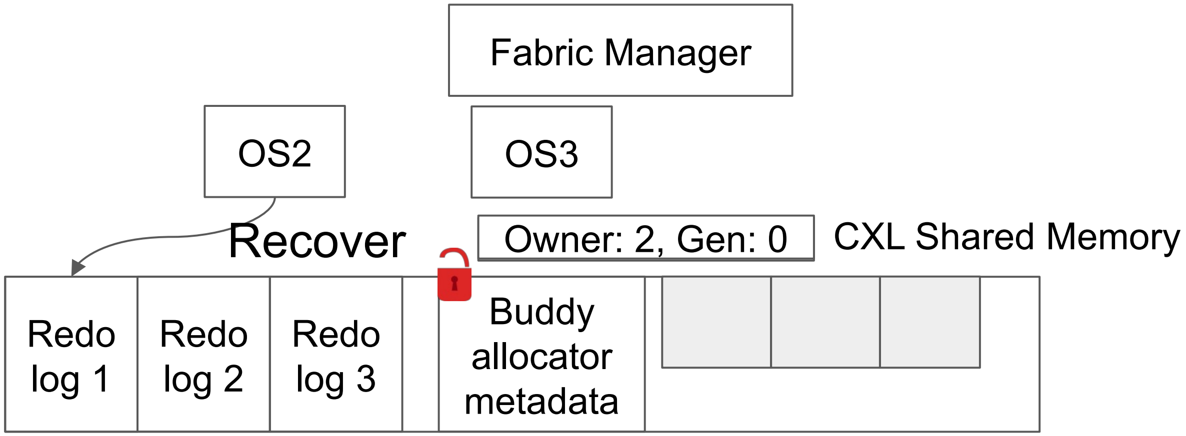
Recovery for kernel memory allocator

- Safe lock transfer algorithm: only one OS can be in the critical section
  - Fabric manager is reliable failure detector
    - Because it power cycles machine before notifying failure
  - Change owner of lock via atomic compare and swap
    - Only one OS will succeed and help complete or abort the current operation

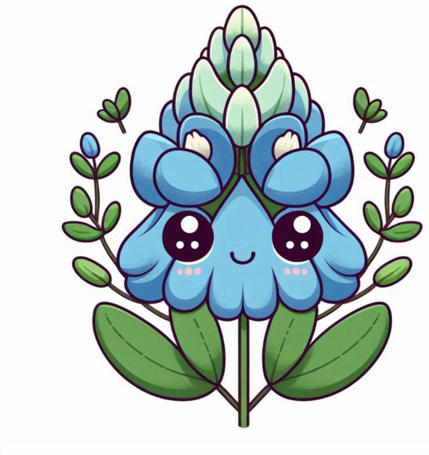


Recovery for kernel memory allocator

- Safe lock transfer algorithm: only one OS can be in the critical section
  - Fabric manager is reliable failure detector
    - Because it power cycles machine before notifying failure
  - Change owner of lock via atomic compare and swap
    - Only one OS will succeed and help complete or abort the current operation



Evaluation



- 16 virtual machines, each 2 vCPU in simulated CXL pod
  - Ubuntu 22.04.2 LTS (Linux kernel v5.19)
  - Danger: in-host cache coherence (CC) simulates cross-host CC
  - CXL: VMs run on the same NUMA node as CXL memory
    - ~250ns
- CPU (Intel SPR): 2× Intel® Xeon 8460H CPU @2.2 GHz
- 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
- Application: MapReduce
  - Global result array and thread-local hash table stored in CXL memory

Overhead for recoverable locks

- MCS vs. TATAS
  - Higher latency
  - Lower variability
- Instance identifiers are lightweight
- JJ focus too much on strong fairness properties over efficiency

	Mean	Std. Dev
Test-and-test-and-set	5.4μs	2.3μs
Recoverable TATAS	5.6μs	2.3μs
MCS queue lock	7.5μs	0.2μs
Recoverable MCS	8.1μs	0.1μs
JJ (Jayanti and Joshi, 2022)	95.7μs	0.2μs

Slowdown due to crash recovery — MapReduce

Crashes	Word Count	K Means	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%

- Crashes are spread evenly across the executions
- Failure detection, notification, and recovery is fast
    - Black Scholes and Dedup from PARSEC

Thank you