AEP: Achieving Hierarchical Fault Tolerance in DSM Through Atomic Execution Protection

Zixuan Wang[1], Qi Wu[1], Hang Huang[1*], Jia Rao[2*], Hui Lu[2*], Hao Fan[1], Zhuo Huang[1], Song Wu[1*], Hai Jin[1]

[1]National Engineering Research Center for Big Data Technology and System
Services Computing Technology and System Lab, Cluster and Grid Computing Lab
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China
[2]The University of Texas at Arlington, USA
Email: {wangzixuan, duckwu, hanghuang, haofan, huangzhuo, wusong, hjin}@hust.edu.cn, {jia.rao, hui.lu}@uta.edu

Abstract

Recent advances in CXL 3.0 have revitalized Distributed Shared Memory (DSM), enabling zero-copy sharing across machine nodes at near-NUMA latency and making shared-memory communication practical for distributed applications. However, DSM memory allocators remain vulnerable to partial failures: client crashes during critical, non-atomic memory operations can corrupt metadata and stall the entire system. Existing solutions either block all clients during recovery or impose heavy runtime overhead due to distributed fault-tolerance protocols. We propose Atomic Execution Protection (AEP), a kernel-assisted fault tolerance mechanism that guarantees atomic completion of user-space critical operations by deferring termination until protected regions finish. AEP tolerates intra-node partial failures without costly distributed coordination. To extend beyond a single node, we further design AEP-DSM, a hierarchical DSM allocator that combines AEP-protected node-level allocators with cross-node coordination. Our Linux AEP prototype integrated with a cross-node DSM consistency protocol (CXL-SHM) achieves near-native local performance and improves throughput by up to 13.3x over state-of-the-art DSM allocators.

CCS Concepts: • **Software and its engineering** → **Operating systems**; • **Computer systems organization** → *Distributed architectures.*

Keywords: Operating Systems, Distributed Shared Memory, CXL, Memory management

ACM Reference Format:
Zixuan Wang[1], Qi Wu[1], Hang Huang[1*], Jia Rao[2*], Hui Lu[2*], Hao Fan[1], Zhuo Huang[1], Song Wu[1*], Hai Jin[1]. 2026. AEP: Achieving Hierarchical Fault Tolerance in DSM Through Atomic Execution Protection. In *21st European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3767295.3803602

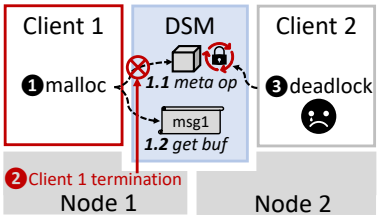


Figure 1. Partial failure in DSM system.

1 Introduction

Modern distributed applications are built on fine-grained microservices [20, 51] and serverless Function-as-a-Service (FaaS) platforms [4–6, 34, 52], which provide independent and elastic scalability. However, their performance is often constrained by inefficient inter-service communication, particularly state transfer. This challenge is especially acute in *stateful serverless functions* (SSFs) [49, 50, 55], where maintaining and moving persistent state can dominate execution time – accounting for up to 95% in some workloads [30]. Conventional communication approaches, whether message passing or using external storage, incur high overhead due to costly (de)serialization and heavy I/O stack [36, 44, 48].

Recent advances in remote memory technologies such as CXL 3.0 [3, 42] have renewed interest in Distributed Shared Memory (DSM) [67]. With CXL's cache-coherent interconnects and hardware-assisted memory sharing, multiple nodes can concurrently map and update the same physical memory region through a CXL switch, enabling zero-copy sharing, in-place updates, and pass-by-reference RPC at latencies close to NUMA access [35, 38, 56, 62, 67]. With DSM clients (processes or threads) running over CXL-based shared memory, distributed applications gain low-latency direct access to shared objects across the cluster, avoiding serialization and I/O costs while approaching the simplicity and efficiency of single-node multithreaded programs.

Despite its promise, DSM is particularly fragile under **partial failures** [67, 70], where individual clients or nodes

EUROSYS '26, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2212-7/2026/04
https://doi.org/10.1145/3767295.3803602

may crash while others continue running. Because multiple clients share the DSM memory allocator, a single client failure (due to software bugs or resource exhaustion) can compromise allocator integrity and system availability. As shown in Figure 1, a client crashing (step ❷) while holding a lock in a critical section can cause allocator deadlock, stalling all participating clients (step ❸). This risk is amplified by the need to *atomically* execute multi-step memory operations. For example, in smart pointer-like semantics [24, 40], a reference operation involves both pointer assignment and reference count increment. Failing to update the reference count can lead to memory leaks, while omitting the pointer assignment can cause double-free errors. These coupled operations must be *failure-atomic* (i.e., all-or-nothing); otherwise, partial failures can result in inconsistent allocator metadata.

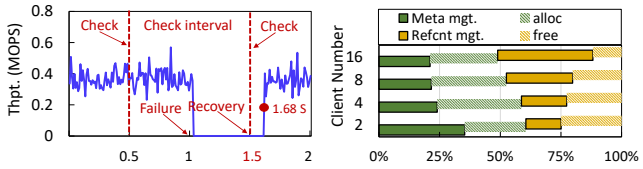
To address partial failures, prior work has explored three directions: log-based recovery, memory partitioning, and lock-free coordination. First, some lock-based DSMs extend write-ahead logging into the allocator, enabling state and lock recovery via log replay [21, 65, 70]. This approach has low runtime overhead and is simple to implement but it incurs a *stop-the-world* recovery phase to restore global consistency, resulting in high latency unsuitable for latency-critical services. Another line of work avoids partial failures by *statically partitioning* memory across clients, eliminating synchronization points [36]. Partitioning removes inter-client dependencies but applies only to workloads with predictable, static memory access patterns. More recently, lock-free DSMs such as CXL-SHM [67] employ distributed consistency protocols (e.g., an era-based scheme [45]) to coordinate allocator metadata operations. This eliminates system-wide halts on failures but imposes substantial runtime overhead and scalability bottlenecks due to frequent cache flushes and memory barriers on critical paths.

In this paper, we introduce a lightweight fault-tolerance approach to DSM allocators that preserves non-blocking resilience to partial failures while sustaining high performance. Our approach is guided by two key insights. First, in modern datacenters, many partial failures are confined to a single node: orchestrators frequently co-locate related services on machines with hundreds of cores [63], and studies show that most faults manifest at the intra-node level [11]. Second, state-of-the-art DSMs (e.g., CXL-SHM [67]) fail to distinguish between intra-node and cross-node failures, applying heavyweight distributed protocols uniformly across all memory operations. As a result, even intra-node allocations pay the cost of global synchronization, incurring frequent cache flushes and memory barriers on the critical path.

To close this gap, we propose **Atomic Execution Protection** (AEP), a kernel-assisted mechanism that guarantees atomic completion of user-space critical operations. AEP leverages the kernel's control over process termination to defer kill signals or exits until a protected region has completed, ensuring that critical sections execute as indivisible units. AEP preserves atomicity for multi-step memory operations in DSM allocators, preventing inconsistent metadata under *intra-node* client crashes, while avoiding heavy synchronization costs of distributed fault-tolerance protocols.

While kernel-assisted atomic execution is feasible, enabling DSM allocators to fully exploit AEP requires addressing two challenges. First, *how to exposing AEP to user space*. The kernel cannot autonomously identify which operations require atomicity or determine when a critical section has completed. To bridge this gap, we adopt a user–kernel co-design. AEP introduces a shadow-entry mechanism that defers process termination until the protected region completes (§ 3.2), ensuring atomic execution. It further provides address-space virtualization that allows co-located clients to efficiently share an AEP context while maintaining isolation among separate tenants (§ 3.3). Finally, we develop a lightweight Musl-compatible library (AEP-lib) coupled with a customized binary loader (AEP-loader), which resides in the AEP context to bridge the gap between kernel-level AEP and user-level DSM systems (§ 3.4). Second, *how to extende AEP beyond a single node*. AEP alone addresses only intra-node failures. To achieve system-wide resilience, we design AEP-DSM, a hierarchical DSM that combines AEP with distributed coordination (§ 4). At inter-node layer, an era-based protocol manages global allocation across nodes; at intra-node layer, each node runs an AEP-protected allocator for memory management within a node. To support this hierarchy, AEP-DSM builds a hierarchical messaging system (§ 4.2) that preserves standard DSM interfaces (e.g., send_to, recv) while optimizing for locality. Specifically, it introduces metadata decoupling (§ 4.1) and hierarchical reference counting (§ 4.2), which reduce (or eliminate) costly remote metadata access, barriers, and flushes from critical paths. These optimizations substantially lower runtime overhead compared to modern DSM systems.

We have implemented AEP in Linux with minimal kernel modifications (under 500 lines of code) while achieving full integration with our AEP-DSM built upon existing state-of-the-art lock-free DSM systems (CXL-SHM) through three key components: (1) AEP-lib, derived from the Musl standard C library [43]; (2) a lightweight AEP-loader designed to bypass conventional address space limitations; and (3) AEP-allocator, an enhanced version of the Musl memory allocator. Comprehensive evaluations across micro/macro-benchmarks and real-world distributed applications demonstrate the effectiveness of AEP-DSM in node-private memory management. AEP-DSM achieves performance similar to Musl-based shared memory systems while significantly outperforming CXL-SHM [67]. Furthermore, performance advantages of AEP-DSM over CXL-SHM exhibit positive scaling with increased node-private memory operations. Our analysis



(a) Recovery latency in Lightning. (b) Runtime overhead in CXL-SHM.

Figure 2. Runtime-recovery trade-offs in DSM systems.

demonstrates that AEP-DSM effectively complements contemporary DSM systems, particularly as data-centric scheduling advances, where such optimizations increasingly determine system-wide performance. Our implementation is available at https://github.com/CGCL-codes/AEP.

2 Motivation

2.1 Distributed Shared Memory (DSM)

DSM provides a software abstraction that unifies the virtual address space across multiple compute nodes into a logically single memory space. By masking the underlying complexity of inter-node communication and synchronization, DSM enables developers to program distributed applications with the same simplicity as shared-memory multithreaded programs. Historically, DSM distributed applications incur higher overhead than single-host multithreaded programs, because traditional network-based DSM (over Ethernet or Infiniband) introduces nontrivial coordination costs to maintain coherence and consistency of shared objects [12, 19, 31].

Emerging cache-coherent interconnects, such as Compute Express Link (CXL) [3], make DSM far more promising. CXL 3.0's memory sharing allows multiple nodes to concurrently map and update the same physical memory region through a CXL switch – all within a single coherence domain. This reduces DSM communication overhead by enabling zero-copy sharing, in-place updates, and pass-by-reference RPC, bringing inter-machine communication close to intra-machine (inter-thread) latency [35, 38, 56, 61, 62, 67].

2.2 Partial Failures and Tolerance

While DSM memory management, or the allocator, presents programmers with simple shared-memory semantics (e.g., allocation, sharing, and reclamation), it internally must handle distributed concurrency and failures while sustaining high performance. This is challenging under *partial failures*, where individual clients (processes) or nodes may terminate unexpectedly while others remain active – i.e., common in distributed environments due to software bugs, resource exhaustion, or machine restarts. Partial failures can leave *allocator metadata* in an inconsistent state, undermining both correctness and progress. Because the system continues running, these inconsistencies may silently propagate, leading to deadlocks, leaks, or double frees that stall DSM allocators and impact all participants. For example, memory allocation typically requires two non-atomic steps: (1) allocating a memory object and (2) storing its pointer in a reference. If a client fails between these steps, the result is either a leak (i.e., allocation without a reference) or a wild pointer (i.e., reference without a valid allocation).

A possible approach is to leverage hardware support. Modern processors provide transactional execution mechanisms to ensure atomic memory updates within a single transaction; however, these mechanisms are limited to short, instruction-level critical sections and operate under strict constraints. For example, Intel TSX (Transactional Synchronization Extensions) [1] requires the transaction footprint to fit within the L1 cache and may abort due to interrupts, system calls, capacity overflows. In contrast, DSM operations such as malloc or send_to involve complex control flows spanning multiple instructions and may even include system calls. Consequently, instruction-level atomic primitives are ill-suited for such complex DSM operations. Existing DSM systems therefore rely on software mechanisms to achieve interface-level atomic execution completeness, thereby eliminating partial failures.

A software-based *resilient* DSM allocator must tolerate partial failures – i.e., even if some clients crash mid-operation, memory management must remain correct (avoiding leaks, double frees, or wild pointers) and non-blocking. Existing DSM systems adopt diverse fault-tolerance mechanisms to tolerate partial failures broadly categorized as *lock-based* and *lock-free*.

In *lock-based* designs [54, 69, 70], a cluster-wide failure monitor (e.g., a fabric manager [69]) tracks client liveness. When it detects a crash, it broadcasts a failure event and the runtime enters a brief quiescent phase, i.e., *stop-the-world*: participants stop issuing allocator operations, and a recovery routine replays that client's operation log to restore allocator metadata to a consistent state. The design, i.e., mutual exclusion with redo/undo logging, is straightforward and relatively easy to implement, but at the cost of coordination pauses – the stop-the-world phase reduces system availability. More specifically, the total outage equals the failure-detection latency (often bounded by the monitor's polling interval) plus recovery time. As shown in Figure 2(a), our emulation with a lock-based DSM [70] observes a crash at t=1.0s, detection at t=1.5s, and recovery completion at t=1.68s – 0.68s of unavailability. This magnitude is large enough to compromise service level objectives (SLOs) and user-visible latency: e.g., a 0.68s pause exceeds the per-hour error budget for 99.99% availability at Google [2] (e.g., 0.36s); due to tail-latency amplification in fan-out services (common in microservices), a single 0.68s outlier at any shard can dominate end-to-end latency and blow p99 targets [18].

To overcome the stop-the-world limitation inherent in lock-based DSM systems, recent work has explored *lock-free* (or non-blocking) mechanisms that avoid global locks and instead rely on distributed consistency protocols to maintain memory consistency during partial failures. Lock-free

designs allow surviving clients to continue making progress even when others fail, enabling asynchronous recovery without blocking the system. A recent example is CXL-SHM [67], which introduces an era-based algorithm to provide partial failure resilience. In CXL-SHM, memory references are maintained through reference counting, where each reference attach and release operation is treated as a transaction that comprises (1) a reference count update and (2) a pointer update. The era-based protocol designates the successful reference count update as the commit point, while pointer updates are idempotent and can be safely retried or completed by recovery services if a client crashes. As a result, CXL-SHM is lock-free, guaranteeing that no client can indefinitely block others, even if it fails mid-operation.

However, the sophistication of the era-based mechanism also introduces significant runtime overheads. Because reference count updates must be carefully ordered, the protocol enforces strict memory operation constraints (e.g., fences and flushes) that stall CPU pipelines and reduce instruction-level parallelism. In addition, memory ownership management through the `send_to` interface requires precise reference count maintenance and frequent metadata synchronization, both of which add to the cost of normal execution. As a result, the runtime overhead of CXL-SHM grows with the number of participating clients. As illustrated in Figure 2(b), when client count increases from 2 to 16, the proportion of additional overhead attributed to CXL-SHM (shown by the dark-colored portion) rises from 49.86% to 60.31%.

Achieving non-blocking partial-failure tolerance while sustaining high performance in DSM memory management remains a critical challenge.

2.3 Key Insights

We observe that the distributed, complex era-based mechanism in CXL-SHM is *only* needed during cross-node memory sharing and reclamation. However, as CXL-SHM treats clients on the same node in the same manner as those on different nodes, it incurs frequent, unnecessary synchronization and metadata coordination overhead even for *intra-node memory operations* (across clients co-located on the same node) that could otherwise be handled more efficiently or even completely eliminated (more in §3).

This limitation is particularly significant given that co-locating multiple services on the same node has become the norm in modern datacenters. Today's machines pack hundreds of cores (e.g., AMD EPYC with up to 192 cores and 384 threads) and large memory capacities, enabling a single node to host thousands of service instances from one distributed application. As a result, orchestration systems often co-locate services on the same node to avoid cross-node communication and reduce latency [9, 10, 25, 39, 53]. For example, Kubernetes' kube-scheduler [7] incorporates affinity rules to encourage co-location, while Nightcore [25] ensures that over 60% of inter-function communication occurs within a single node. Pheromone [63] goes further with a data-centric scheduler that delays placement to optimize locality and minimize cross-node transfers. Meanwhile, dense co-location raises failure risks due to resource contention and orchestration overheads. Prior work [32, 33] shows that container-heavy deployments increase cgroup creation and scheduling latency, leading to pod timeouts triggering Kubernetes-initiated terminations. This implies that **intra-node failures** (i.e., when a single client or process terminates on a node while other co-located clients continue running) are both more common and often dominate overall failure scenarios, making lightweight intra-node fault tolerance desirable.

In fact, the Linux kernel demonstrates that such lightweight intra-node fault tolerance is possible in a similar shared-memory setting, albeit for kernel-critical data structures rather than DSM allocator metadata. Specifically, the kernel maintains globally shared data structures (e.g., process tables, inodes, filesystem buffers, device states) that must remain accessible across all processes. To handle unexpected process terminations (i.e., intra-node partial failures), the kernel employs a *delayed signal handling mechanism* that defers termination until the current operation completes and control returns to user space, ensuring *kernel execution paths finish without disruption*. This simple yet effective design preserves kernel state consistency and provides fault tolerance for kernel memory without complex recovery logic. Unlike era-based fault-tolerance algorithms, it achieves resilience with zero runtime overhead, making it exceptionally lightweight. These kernel-level insights inspire our design in this paper: by enforcing atomic completion of critical operations before a process can be terminated, DSM can achieve lightweight, intra-node fault tolerance for node-private memory management (i.e., DSM memory operations confined to threads/processes within a single physical node) without the heavy overheads of distributed protocols.

2.4 Failure Model

We assume independent client and node failures, with intra-node failures being dominant due to the dense co-location of services on modern datacenter nodes. While partial failures can arise throughout the DSM stack (e.g., coherence protocols, metadata management, synchronization primitives, transport layers, and task runtimes), we focus specifically on DSM allocator operations (e.g., allocation, reclamation, and reference management), ensuring they remain atomic and non-blocking despite client or node crashes. We do not consider Byzantine failures such as arbitrary corruption of shared memory, but we assume failures may occur at any point within a critical section, potentially interrupting allocator operations. Similar to CXL-SHM [67], we consider only client and node crashes – not failures of backend DSM memory devices – and assume that once recovery begins, failed clients cannot further modify the shared memory.

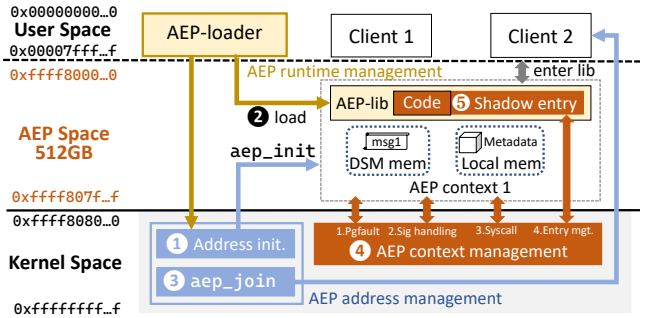


Figure 3. AEP system architecture.

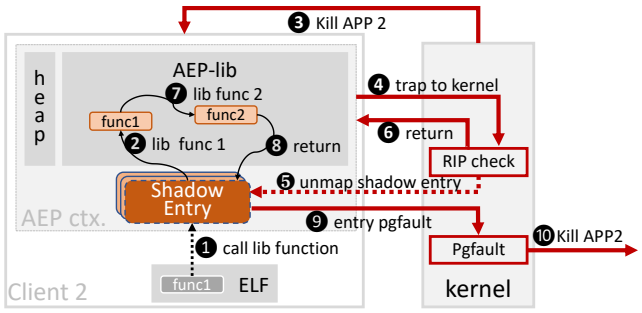


Figure 4. Delayed process termination.

3 Atomic Execution Protection (AEP)

Inspired by the prevalence of process co-location in modern datacenters and the kernel's delayed signal handling mechanism, as discussed in § 2, we propose *Atomic Execution Protection* (AEP), a lightweight fault-tolerance mechanism for *intra-node* partial failures. AEP guarantees that the termination of clients on a single node does not affect the consistency of DSM memory allocation if the physical node remains healthy. In contrast, *node-level* failures, where the entire node crashes, are handled separately in a hierarchical manner (§4).

The key idea behind AEP is to ensure atomic execution of user-space processes within critical sections by deferring their termination until the protected region has completed. This design prevents DSM allocators from entering an inconsistent state under intra-node partial failures, while eliminating the runtime overhead associated with distributed fault-tolerance protocols. The design of AEP is inspired by the delayed process termination mechanism in modern operating systems, which defers killing a process executing in kernel mode to preserve kernel data integrity. AEP extends this mechanism to ensure the multiple steps in DSM allocation are atomic with respect to process failures.

3.1 AEP Architecture

Figure 3 illustrates the AEP architecture and its workflow. AEP comprises two primary components: (1) a dedicated user-accessible address space, the *AEP context*, shared among clients on the same host to support atomic execution of DSM critical sections; and (2) a user-space runtime, consisting of `AEP-lib` and `AEP-loader`, which facilitates application development and execution. To enable AEP, a system administrator first creates and initializes the AEP context (❶) and loads the functions associated with DSM allocations into this context. When a client, such as a container in a serverless computing environment that uses DSM, joins, the administrator adds it to the AEP context (❸). During runtime, the AEP context enforces atomicity at the DSM management interface granularity based on pre-registered functions (e.g., malloc, free, send, and recv), requiring no modifications to user programs. AEP also introduces a dedicated kernel-level context management subsystem (❹) to support the AEP context and guarantee that each function invocation executes to completion without interruption or partial failure, even in the presence of client crashes. The subsystem handles AEP context–related events such as exceptions, signals, and system calls, and in particular provides entry management for registered functions (❺). Together, these kernel enhancements safely defer process termination and reprocess pending termination events after execution completes, thereby ensuring correct and atomic operation completion.

3.2 Atomic Execution

The shadow entry mechanism is central to AEP's atomic execution guarantee. Processes enter the AEP context through the shadow entry, which is a 4KB memory page physically isolated from other program code and data segments. While program execution within the AEP context is entirely in the userspace, the OS kernel can take control over program execution and handle process termination by manipulating the address mapping of the shadow entry page.

Figure 4 illustrates how AEP delays the killing of a process while it is executing within the AEP context. Upon invoking an AEP-registered DSM management interfaces, e.g., `malloc`, program execution transits to the AEP context (Step ❶). During the atomic execution, if `func1` encounters a failure (Step ❷) and subsequently receives a `SIGKILL` signal (Step ❸), the program execution is interrupted and trapped into the kernel (Step ❹). Unlike the conventional Linux handling of user-space exceptions, AEP inspects the faulting program counter `RIP` to determine whether it lies within the AEP context's address space. If so, AEP defers process termination and unmaps the shadow entry page from its page table (Step ❺) and allows the program execution to continue. Otherwise, the kernel follows the standard return path and terminates the process immediately (Step ❻). Under AEP protection, `func1` can resume its execution and call `func2` (Step ❼). When both `func1` and `func2` complete, the program execution returns to the shadow entry (Step ❽), where the access to the unmapped page triggers a page fault (Step ❾). During a shadow entry page fault, AEP instructs the kernel to verify the completion of the faulting process and terminate it afterward.

Table 1. Events handling in Linux and AEP context.

Event Type		Event location		When to handle the event	
				Vanilla Linux	AEP enabled
External event	Termination: destroy, suspend		User AEP	in place -	in place ❶ After AEP
		Nested	User->Kernel AEP->Kernel	After K -	After K ❷ After K & AEP
	Interrupt		All	in place	❸ in place
Internal event	PF exception		User AEP	in place -	in place ❹ in place
	Program exception		All	in place	❺ in place

The efficiency and simplicity of the shadow entry mechanism stem from several key properties. *First*, because the AEP space operates entirely in user mode, entering the shadow entry requires neither privilege transitions nor special instructions; a regular function call suffices, resulting in minimal overhead. *Second*, when a client enters AEP-lib, it continues to use its own stack to execute AEP-lib code, thereby eliminating the need for stack switching within the shadow entry. *Third*, the shadow entry contains only the minimal logic required to dispatch into AEP-lib and is carefully designed to avoid modifying any core metadata before execution is fully established. In our implementation, the shadow entry page includes a lightweight interface dispatcher that selects the target operation (e.g., malloc, free, send_to, or recv) based on the invocation identifier and transfers control accordingly.

Event handling. In addition to termination signals such as `SIGKILL`, other events may also interrupt program execution in the AEP context. We illustrate how AEP correctly handles these events properly. Table 1 summarizes the handling of various events in both Vanilla Linux and the AEP context. The table lists the different types of events, their points of occurrence, and whether they are handled immediately in place or deferred until after exiting the current context. As shown in the figure, AEP handles most events in the same way as vanilla Linux (❸–❺), since these events either do not terminate the process or result from inherent program errors (e.g., exceptions). Program exceptions within the AEP context indicate errors in the DSM allocators and are therefore not considered partial failures caused by external management. For process termination signals (❶–❷), AEP defers their handling only when they occur within the AEP context. Otherwise, since the process does not involve the DSM allocator's metadata, no protection is required. For termination signals that occur while executing on the kernel stack within the AEP context (❷), AEP defers their handling until it returns from the kernel and completes execution of the AEP context.

3.3 AEP Context Management

The AEP context not only serves as an atomic execution environment for DSM memory operations but also facilitates data sharing between clients on the same host. For example, to transfer the access permission of an object from one client

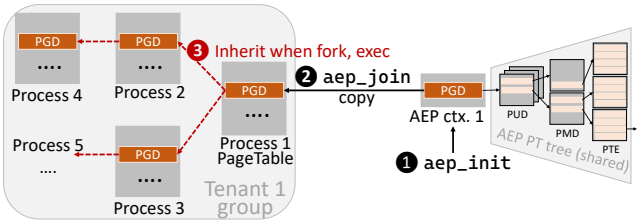


Figure 5. AEP context management and sharing.

to another, one needs to copy the reference of the object in DSM across process boundaries via the `send_to` primitive. AEP facilitates such DSM operations between clients on the same host via memory sharing within the AEP context. Specifically, the AEP address space is a dedicated 512 GB globally shared virtual address space, positioned immediately below the kernel address space, as illustrated in Figure 3. It includes three types of mappings: (1) the shadow entry page and AEP-lib code; (2) a tenant's virtual address range for allocated CXL memory; and (3) memory metadata, comprising intra-node metadata stored in DRAM and inter-node metadata stored in CXL memory. Mappings (1) and (2) are established when a tenant joins the DSM, while mappings in (3) are created and updated dynamically as memory allocations occur. In our implementation, the tenant's CXL address range in (2) is statically mapped into the AEP, allowing clients across multiple hosts to allocate memory from the tenant's budget. Metadata in (3) is mapped on demand as inter- and intra-node allocations take place.

Unlike traditional process address space management, where each process has an isolated address space, processes belonging to the same DSM user must share references to common DSM objects and operate within a shared AEP context to enable atomic execution. Inspired by kernel page table management—where all processes share a common kernel page table when executing in kernel space, AEP adopts a similar approach, allowing all processes of the same user to share a single page table for the AEP context. In a multi-user environment requiring strong isolation, AEP assigns each user a distinct AEP address space with its own page table, ensuring proper separation across users.

Figure 5 illustrates the management of an AEP context for a single user. After the AEP context is initialized (Step ❶), a process joins the context (Step ❷) and shares the AEP page table with other processes belonging to the same user. Furthermore, AEP also introduces an inheritance mechanism within the AEP context – all processes created with `fork` share the same AEP page table with their parents. Note that although processes share the AEP page table, this sharing is limited to operations within the AEP context; the rest of each process's address space remains isolated.

3.4 Userspace Runtime

AEP exposes a comprehensive set of operations to the user space via an extended `aep_ops` system call interface, with

Table 2. List of AEP APIs used by `AEP-loader` and `AEP-lib`.

Capabilities	Interface	Required Privileges	RIP restriction
AEP context initialization	`aep_init`	Root privilege	No restriction
Add to an AEP context	`aep_join`	Root privilege	No restriction
Load AEP-lib	`aep_mmap_exec`	Root privilege	No restriction
Memory expanding	`aep_brk`	No restriction	AEP context
	`aep_mmap_rw_nx`	No restriction	AEP context
AEP grace period	`aep_grace_period`	Root privilege	No restriction

each operation identified by a distinct opcode. To enable convenient and controlled access to these capabilities, we developed a user-space library (`AEP-lib`) together with a specialized loader (`AEP-loader`). Table 2 lists the AEP APIs and their access permissions.

Interfaces and privileges. The `aep_init` interface initializes the AEP context, while `aep_mmap_exec` handles subsequent AEP-lib loading. By separating these two functionalities, we enable the implementation of an AEP loader entirely in user space. This design avoids re-implementing a dedicated AEP-ELF loader inside the kernel, thereby minimizing kernel modifications. Address-space virtualization and context sharing are enabled by `aep_join`, allowing multiple clients within a tenant to share the same AEP context. These operations require root privileges to ensure controlled setup. For runtime operations such as heap expansion (`aep_brk`) and memory mapping (`aep_mmap_rw_nx`, map memory with read and write permissions but no execute permission), AEP provides unprivileged interfaces that must be invoked strictly within the AEP context; requests from normal user space are rejected to protect context integrity. To prevent indefinite execution, e.g., from bugs or infinite loops, AEP implements a `aep_grace_period` sysfs interface to enforce maximum execution time and terminate stalled processes.

AEP-lib is a statically linked library derived from Musl, designed to avoid dynamic loading failures and concurrency hazards. Its code segment is positioned at the start of the AEP virtual address space to guarantee correct execution. In addition to the code segment, AEP-lib introduces a dedicated shadow entry segment to support the shadow entry mechanism. This special segment is 4KB-aligned and mapped to a predefined AEP address. Developers can use compiler annotations to place a carefully designed entry function into this segment, thereby enforcing atomic execution of AEP-lib interface invocations. The entry code avoids direct access to user memory, enforcing a strict boundary between user and AEP contexts. Similar to system call conventions, clients pass operation codes and data through a controlled interface, with a `copy_from_cli` routine used to safely transfer user data, preventing faults from corrupting AEP state.

AEP-loader maps `AEP-lib` into the AEP context with executable permissions, bypassing the limitations of standard Linux loaders. This ensures compatibility with AEP's specialized address-space layout and enables robust, atomic execution for DSM allocators.

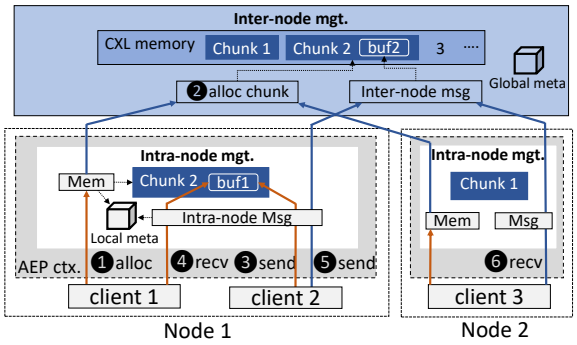
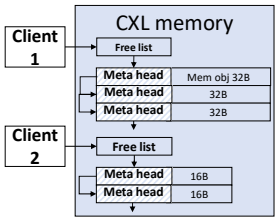


Figure 6. Hierarchical DSM management.

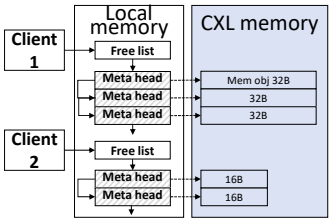
4 Hierarchical DSM Management

We further introduce AEP-DSM, a hierarchical DSM management system that integrates seamlessly with AEP's intra-node fault-tolerance mechanisms. Unlike existing CXL-enabled DSM fault-tolerance mechanisms, such as CXL-SHM, which use complex era-based algorithms for all clients regardless of whether they are co-located on the same host, AEP-DSM exploits intra-node locality and employs a two-level DSM management scheme. As shown in Figure 6, inter-node DSM management, such as memory allocation, reclamation, and ownership tracking, is implemented using the era-based algorithm, and therefore incurs relatively high runtime overhead. In contrast, intra-node DSM management leverages efficient AEP to maintain allocation consistency within a node. Software client failures and intra-node DSM operations are isolated from the global DSM management. The core idea is to allocate DSM using a two-level strategy: (1) when a node issues its first DSM allocation request, the global DSM manager assigns a large memory chunk to that node; and (2) subsequent allocation requests from the same node are served from this preallocated chunk until additional chunks are needed.

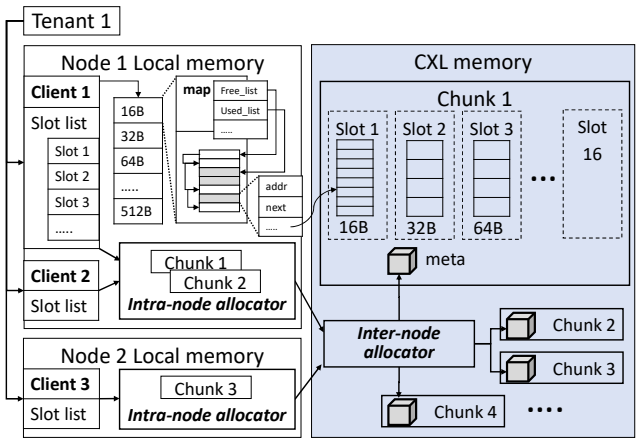
Although AEP guarantees atomic execution of DSM memory operations within a node, two key challenges remain when integrating AEP with global DSM management: (1) **Metadata management** refers to the tracking of free and allocated DSM. Traditional memory allocators [8, 43, 67] store allocation metadata alongside the corresponding memory objects on DSM (Figure 7 (a)). Consequently, even node-local memory allocations must update metadata in the remote DSM. Our experiments with CXL-SHM reveal that these metadata updates can contribute up to 32.2% of the throughput loss in the `threadtest` benchmark. (2) **Reference counting**. To prevent memory leaks, double-free errors, and dangling pointers, current DSM system [67] perform precise reference counting [24, 40] for each memory object, introducing substantial reference update overhead and degrading performance. Continuing to adopt such fine-grained reference counting would fail to fully leverage the inherent advantages of AEP-DSM's hierarchical architecture.



(a) Free list metadata structure in traditional memory allocator.



(b) Free list metadata structure in AEP-DSM.



(c) Overall metadata structure in AEP-DSM.

Figure 7. Metadata structure management.

4.1 Metadata Management

To minimize frequent remote DSM accesses, the core insight of AEP-DSM is to decouple memory objects from their management metadata. This design allows AEP-DSM to construct fine-grained allocation metadata and free lists entirely in local memory (as shown in Figure 7(b)), thereby avoiding remote metadata operations. Furthermore, this local metadata is protected by AEP, making it resilient to arbitrary client crashes. However, unlike DSM memory, node-local memory is volatile. Upon a full-node failure, locally maintained metadata would be lost and could potentially lead to inconsistencies. To ensure that such local metadata can be safely discarded without compromising global correctness, AEP-DSM adopts a two-level metadata management scheme (Figure 7(c)). At the inter-node level, memory is tracked at chunk granularity, with metadata stored in DSM and updated using an era-based fault-tolerant algorithm. At the intra-node level, finer-grained metadata is maintained in local DRAM to construct free lists for allocation. In the event of a full-node hardware failure, the intra-node free-list metadata can simply be discarded, and recovery only requires reclaiming the chunks assigned to that node at the inter-node layer. In contrast, if an individual client crashes, the AEP mechanism preserves the intra-node free-list metadata, allowing it to remain consistent and independent of global inter-node management. As a result, intra-node allocation and deallocation operate solely on local free-list metadata, without cross-node synchronization, cache flushes, or remote metadata accesses.

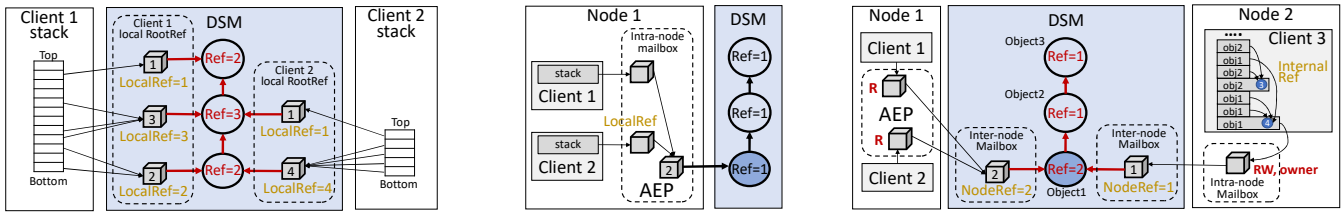
Our current intra-node memory manager uses a fixed-size block management scheme [8, 43], serving memory requests through free-list searches in a manner similar to the mimalloc [8]. This design eliminates the need for fault-tolerance protocols, memory barriers, or cache flushes, while fully utilizing `AEP-lib`'s rich API support (e.g., `mmap`, locks, hash tables). Consequently, it provides a simple development model that allows users to customize memory allocation strategies without worrying about fault tolerance, which is guaranteed by AEP's atomic execution semantics.

4.2 Hierarchical Reference Counting

Reference counting is a major challenge in DSM management. As illustrated in Figure 8(a), traditional reference tracking technique [24, 40, 67] maintains two types of reference information to track references to a three-element linked list: (1) a `LocalRef`, which counts the number of references to an object from a single client, and (2) a global `Ref`, which tracks the total number of clients referencing the object. To ensure correct reference counting in the presence of arbitrary client crashes, all reference-counting metadata is stored in CXL memory and maintained using the era-based algorithm, which enforces strictly ordered updates through memory fences and cache flushes. Consequently, whenever any client accesses a memory object, it must incur these expensive ordering and persistence operations. As a result, precise reference tracking introduces substantial overhead.

Baseline reference counting in AEP-DSM. To reduce the overhead of reference counting, AEP-DSM decouples intra-node reference counting from inter-node reference counting. As shown in Figure 8 (b), AEP-DSM moves intra-node reference counting entirely to the local memory of a node, within the AEP context. As such, expensive updates to the global reference counts are triggered only when the node-level refcount reaches zero. For example, if both client 1 and 2 stop referencing object 1, the global Ref needs to be updated. In addition, AEP-DSM uses an intra-node mailbox, a shared memory region within the AEP context and local DRAM, to record per-client reference information for each object. This information is used to determine how the global reference count should be updated, particularly in scenarios where one or more clients have failed. Note that the recorded reference counts remain consistent even in the presence of client failures, as AEP's atomic execution guarantee ensures that clients update their reference counts correctly before termination.

Root-only reference counting in AEP-DSM. Although the baseline reference counting in AEP-DSM significantly reduces inter-node reference count updates, the amount of reference count information, in the form of LocalRef and NodeRef, grows with the number of clients , nodes, and the



(a) Precise reference counting. (b) Baseline reference counting in AEP-DSM. (c) Root-only reference counting in AEP-DSM.

Figure 8. Reference count management for a three-element linked list. Object 1 is the link head (root). Red arrows indicate refcount updates via the era-based protocol.

number of objects, especially for complex data structures like binary trees. To address this reference counting overhead, AEP-DSM proposes a root-only reference counting scheme that only tracks the reference count to the root node of a complex data structure, such as the head node in a linked list and the root of a tree. A root object can be explicitly specified via the `send_to` interface. For example, when transmitting a binary tree, the sender only needs to send the root node through `send_to`, the remaining objects can be accessed by traversing the internal pointers of the data structure. As shown in Figure 8(c), only the root node's NodeRef is persisted and registered in inter-node mailbox (located and accessed by the recovery service in the CXL memory.), and its updates are synchronized using the era-based algorithm. This root-only reference counting design is motivated by several observations. *First*, precise reference counting is a sufficient but not necessary condition for preventing memory errors. AEP-DSM can relax strict counting precision to significantly improve performance while still avoiding critical errors such as use-after-free. *Second*, shared-object access control typically follows a single-writer, multiple-reader model. Under a hierarchical architecture, this property provides substantial design flexibility for AEP-DSM to simplify reference counting, thereby reducing runtime overhead.

Based on these observations, AEP-DSM adopts a root-only counting mechanism. It grants write permission for a shared data structure to a single client, which is responsible for tracking the reference counts of the structure's internal nodes. All other clients, whether on the same host or across hosts, are restricted to read-only access for memory-related operations. Note that all clients can modify the content of the objects, but only one can allocate to or free memory from the shared data structure. The write permission can be transferred to a different client once the current one completes its memory allocation/reclamation. If a writer crashes, the AEP records its internal reference counts and synchronizes the counts to the global Ref in DSM.

This design introduces a subtle challenge. Even after an object's internal reference count drops to zero, some readers may still be accessing it. To handle this safely, AEP-DSM employs a volatile reference count protected by AEP, ensuring correctness under client-level failures. However, following a node-level failure, these counts are not automatically decremented and may temporarily exceed the actual number of live readers. Such transient overestimation does not result in use-after-free errors, but it may cause minor memory leakage. To address this issue,

AEP-DSM defers the reclamation until all readers exit. These objects are placed into a local cache pool and withheld from reuse to ensure safety. Such a deferred reclamation strategy is commonly adopted on in today's memory allocators, such as mimalloc [8]. In essence, root-only reference counting embodies a deliberate design trade-off compared to traditional schemes [67]. It maintains only the degree of reference-count precision necessary to guarantee safety, rather than enforcing exact accuracy. By tolerating a bounded overestimation of live readers under node failures, it eliminates the need to precisely track a large number of distributed objects across hosts in DSM, thereby substantially reducing coordination overhead and improving scalability.

Memory ownership and reference passing. Ownership transfer and reference passing are important mechanism in DSM to share memory objects across isolated address spaces. Building on the reference-counting design, AEP-DSM implements its memory send primitive using a hierarchical mailbox mechanism to transfer and share ownership. For intra-node sends, AEP-DSM simply creates a LocalRef in local memory and links it to the root object's NodeRef, requiring neither CXL memory access nor global coordination. In contrast, inter-node sends create a new NodeRef in the inter-node mailbox and link it to the root object, invoking the era-based fault-tolerant protocol (as in CXL-SHM) to ensure idempotence. Although this step introduces higher overhead, it is incurred only once per cross-node transfer; subsequent references to sub-objects in the group are managed entirely locally at the receiving node. As a result, the overall efficiency remains high despite the initial coordination cost.

4.3 Failure recovery

Since AEP-DSM employs hierarchical reference counting, its recovery mechanism is likewise designed hierarchically. Traditional approaches rely on a centralized recovery service; for instance, in CXL-SHM, when a client fails, the recovery service must scan the client's rootref set in CXL memory,

clear each rootref, and decrement the corresponding global reference counters. In the event of a node failure, this process must be repeated for all clients on the failed node, further increasing recovery overhead. In contrast, AEP-DSM deploys lightweight recovery daemons on each node to handle client-level failures locally within the intra-node layer, while a separate inter-node recovery service is responsible only for node-level failures. This design minimizes global coordination and reduces recovery latency.

When a client fails, the intra-node recovery daemon locates and clears the client's LocalRef in the node's mailbox, then decrements the corresponding NodeRef. If the NodeRef reaches zero, indicating no remaining references from that node, the global reference count is decremented. When the global reference count also reaches zero, the entire memory group and its deferred reclamation pool are reclaimed. In the case of a node failure, the global recovery service simply clears the node's NodeRef in the inter-node mailbox and follows the same procedure. This hierarchical strategy confines recovery to the appropriate layer, avoids unnecessary cross-layer coordination, and achieves inter-node recovery with the same time complexity that CXL-SHM incurs for client-level recovery. Moreover, because intra-node metadata such as mailboxes resides entirely in local memory, AEP-DSM largely eliminates remote CXL memory accesses during recovery, further improving efficiency.

5 Evaluation

We evaluated **AEP-DSM**, our hierarchical DSM allocator that combines AEP-protected intra-node allocators with lightweight cross-node coordination. To demonstrate its effectiveness, we compared it against two representative baselines: (1) **Lightning** [70], a lock-based DSM with log-based recovery and stop-the-world fault tolerance, and (2) **CXL-SHM** [67], a state-of-the-art lock-free DSM that employs a distributed era-based protocol for non-blocking partial-failure resilience. Notably, Lightning is a key-value store rather than a memory allocator and therefore does not expose allocator interfaces; as a result, we compare it only in key-value workload evaluations. In addition, Lightning relies on Intel MPK to protect against stray writes, whereas AEP-DSM and CXL-SHM do not provide such protection. Our evaluation seeks to answer four key questions: (1) Does AEP-DSM achieve high performance on basic memory operations compared to state-of-the-art DSMs? (2) How much benefit does AEP's intra-node fast path provide for memory allocation and transmission? (3) Can AEP-DSM sustain these gains under realistic applications such as key-value stores and serverless workflows? (4) How effectively does AEP-DSM recover from client- and node-level failures compared to prior approaches?

Testbed. We conducted our experiments on a dual-socket server with Intel Xeon Gold 6416H 2.10GHz processors, providing 36 physical cores (72 logical threads) and 128,GB of DRAM across two NUMA nodes. Since commercial CXL 3.0 hardware has been not yet available, we emulated disaggregated shared memory using a Montage CXL memory controller [41] (CXL 1.1) attached to a 32GB memory pool. Table 3 summarizes the key memory access characteristics of our test platform.

Table 3. Memory throughput and latency characteristics.

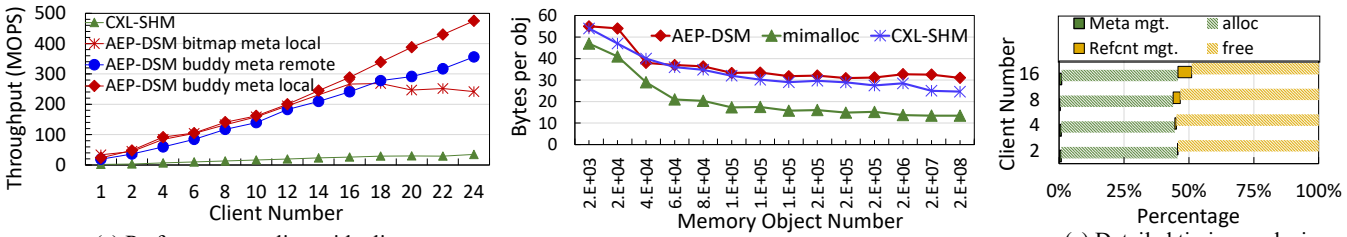
Mem type	Seq (MB/s)	Rand (MB/s)	CAS (MOPS)	Latency (ns)
Local numa	109782.6	57830.3	65.9	115
Remote numa	35701.2	36334.7	65.9	179
CXL	28300.2	25893.2	65.9	253

5.1 Memory Allocation/Deallocation

We evaluated whether decoupling intra-node memory management and using AEP for lightweight fault tolerance can improve throughput of *allocation and deallocation* – the two fundamental primitives in DSM allocators – by avoiding costly cross-node coordination and remote metadata accesses. We measured throughput using the threadtest benchmark [13], where multiple clients continuously allocate and free large objects from a shared DSM pool. Note that, in AEP-DSM, most allocations are satisfied locally by the intra-node allocator, while inter-node coordination is invoked only when node-private CXL memory is exhausted (e.g., roughly one inter-node allocation for every 16,000 intra-node allocations). We compared AEP-DSM against CXL-SHM, and further analyzed AEP-DSM variants with different metadata placements (local vs. remote) and intra-node allocation schemes (buddy vs. bitmap).

Figure 9(a) shows that AEP-DSM (with a *buddy*-based Level-2 allocator and *local* metadata) sustains over **13×** higher allocation/deallocation throughput than CXL-SHM across thread counts. Even when using the alternative bitmap allocator, AEP-DSM outperforms CXL-SHM, although the buddy allocator yields higher performance by avoiding global locking and bitmap traversal costs. In addition to decoupling intra-node operations from cross-node DSM management, the advantage of AEP-DSM also stems from maintaining metadata in local memory on each node. This design delivers a 1.33× throughput improvement over an otherwise identical configuration that keeps metadata in remote (CXL) memory (i.e., "AEP-DSM buddy meta remote"). In contrast, CXL-SHM, lacking any notion of node-local metadata or atomic intra-node execution, must perform expensive remote writes and flushes on every allocation, severely limiting its throughput.

Similar to CXL-SHM, AEP-DSM must maintain additional metadata to support fault tolerance (e.g., per-object reference counts), while this overhead is modest. As shown in Figure 9(b), compared to mimalloc [8], a modern allocator without fault-tolerance mechanisms, allocating 500M objects in AEP-DSM requires only about 16 extra bytes of local metadata per



(a) Performance scaling with client count. (b) Memory consumption in AEP-DSM. (c) Detailed timing analysis.

Figure 9. Performance comparison of memory allocation/deallocation between AEP-DSM and CXL-SHM.

Table 4. Memory primitives execution time breakdown (numbered annotations match Figure 6 steps)

	AEP-DSM	CXL-SHM
Alloc	slow path ❶ + ❷: 589.1ns fast path ❶: 42.2ns	581.3ns
Free	33.72ns	462.0ns
Send + Recv	fast path ❸ + ❹: 86.1ns slow path ❺ + ❻: 2027.3ns	2019.6ns

object. Overall, the metadata footprint of AEP-DSM remains comparable to that of CXL-SHM, exceeding it by only 8 bytes due to the need to maintain a NodeRef—a cost most noticeable when a memory group contains only a single object. In contrast, AEP-DSM supports storing metadata locally, whereas CXL-SHM places it in remote CXL memory.

To better understand where AEP-DSM's gains come from, we profiled allocator execution using Linux perf. As shown in Figure 9(c), the vast majority of CPU cycles in AEP-DSM are spent in the core allocation and free routines, with negligible time in metadata or reference-count management. In contrast, profiling CXL-SHM reveals that a large fraction of cycles are consumed by metadata synchronization, particularly cache flushes and fence instructions required to persist updates in CXL memory (Figure 2(b)). This difference clearly reflects the impact of AEP-DSM's design advantage – by offloading intra-node fault tolerance to the AEP context, AEP-DSM can eliminate flush and fence operations from the intra-node allocation fast path. Timing analysis in Table 4 further confirms this benefit: AEP-DSM's fast-path allocation completes in just 42.2 ns – only 7% of the time taken by CXL-SHM (581.3 ns) – while its slow-path allocation (triggered infrequently when node-private pools are exhausted) performs similarly to CXL-SHM, since both fall back to inter-node global allocation. Given this superior performance, we adopt the "AEP-DSM buddy meta local" configuration as the default for all subsequent experiments unless otherwise noted.

5.2 Memory Transmission

Beyond basic allocation/deallocation, AEP-DSM provides a high-performance *memory transmission* primitive for data sharing between clients – via `send`/`receive` operations that are functionally comparable to those in CXL-SHM but optimized for *intra-node* clients. By leveraging hierarchical reference counting, AEP-DSM removes flush operations from the intra-node fast path and minimizes global metadata updates, enabling faster transfers. We evaluated these primitives using a microbenchmark and an RPC server implementation, comparing AEP-DSM to CXL-SHM.

First, Table 4 shows that AEP-DSM completes a fast-path `send+receive` in 86.1ns, compared to 2019.6ns for CXL-SHM – a 23.4× speedup. This corresponds to the time saved by avoiding cache flushes (steps ❸ and ❹ in Figure 6). For inter-node transfers, AEP-DSM performs the same as CXL-SHM (within 0.3%), as the cost is dominated by remote CXL memory access with inter-node synchronization. This shows that AEP-DSM's intra-node fast path optimization does not interfere with the performance of inter-node transfers.

We further stress-tested both DSM systems by implementing a remote procedure call (RPC) server to measure the performance of single-object and structured-object transfers. As shown in Figure 10(a), AEP-DSM delivers 14.4× higher throughput than CXL-SHM for intra-node single-object transfers, while the two systems perform comparably in the cross-node case (11.74 MOPS vs. 12.93 MOPS). For structured objects such as linked lists or trees, CXL-SHM suffers from severe degradation, as shown in Figure 10(b) [1], because every object access creates a new reference and triggers the era-based protocol, incurring at least two cache flushes per operation. In contrast, AEP-DSM bypasses inter-node reference count updates for intra-node transfers, thereby eliminating flushes from the fast path.

For cross-node transfers, AEP-DSM invents a Root-only reference counting mechanism that confines updates to the root object of a structured data group, requiring just a single cache flush per structure rather than per object. As shown in Figure 10(b), this design reduces traversal time for a binary tree of height 20 by 12× in cross-node transfers compared to CXL-SHM. Although AEP-DSM remains 3.72× slower than Local-SHM (a baseline that traverses the tree entirely in local memory without fault tolerance) in the intra-node case,

[1]Both subfigures are derived from the same evaluation experiment. We plot them separately due to scale differences: placing all lines on a single axis makes them indistinguishable. The red line (AEP-DSM cross-node) appears in both subfigures and represents the same test case.

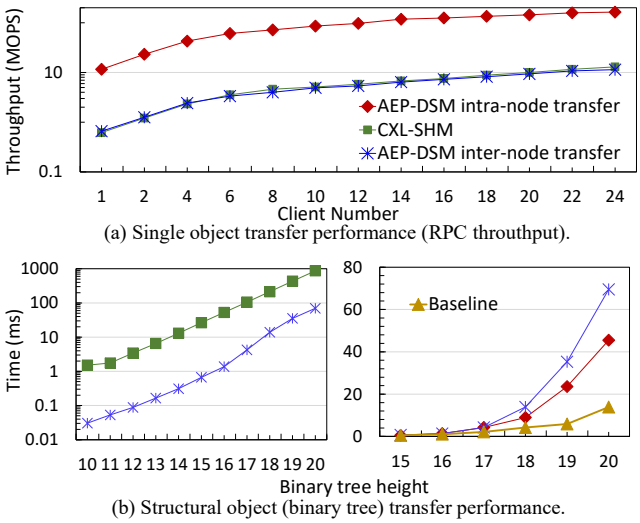


Figure 10. Performance of memory transmission.

the gap reflects the extra work of constructing root-level reference counters on the receiver side – overhead that is essential for maintaining fault tolerance.

5.3 Real-World Applications

We conducted evaluations using two representative distributed applications, including a memory-intensive key-value (KV) store and interaction-intensive serverless workflows.

KV store. As AEP-DSM requires application adaptation to use the `AEP-allocator` and `AEP-lib` components, we implemented a custom KV store. This implementation adopts the single-writer/multiple-reader concurrency model from CXL-SHM, ensuring exclusive memory group ownership for safe reclamation after partial failures. Similar to CXL-SHM, the design employs partitioned key spaces where writers insert exclusively into assigned hash tables. The AEP mechanism provides atomicity for storage operations like `put/get`, allowing the KV store to focus on core logic without handling partial failures (the implementation comprises fewer than 600 lines of code). In addition, we used a single-node KV store built on the AEP-DSM buddy allocator as a baseline, with all fault-tolerance mechanisms removed to approximate the upper-bound performance.

We adopted the workload generator from CXL-SHM to evaluate KV-store performance under varying read/write ratios. As shown in Figure 11(a–d), the performance benefit of AEP-DSM over CXL-SHM grows with write intensity: from modest gains under read-heavy workloads to as much as 2.36× higher throughput (45.69 vs. 16.26 MOPS) in the write-only case (Figure 11d), enabled by AEP-DSM's lightweight intra-node fault tolerance. Figure 11(e) further demonstrates that AEP-DSM significantly outperforms Lightning [70] across client counts and access patterns. Lightning scales poorly due to severe global lock contention from its lack of memory pooling, and its log-based recovery introduces frequent memory barriers that further degrade performance.

Serverless workflow. We further evaluated AEP-DSM on realistic serverless workflows through a 40-function chain benchmark from ServerlessBench [64] and a PageRank implementation from FunctionBench [27]. Pre-warming was applied to eliminate cold-start overhead, and the function scheduling policy was adjusted to control the ratio of local to remote communication. We first examined the 100% intra-node case and then the 100% inter-node case.

When functions solely communicate within a single node. Figure 12(a) shows that both AEP-DSM and CXL-SHM outperform TCP-based communication, underscoring the advantage of DSM for function interaction. In the chain benchmark, AEP-DSM yields a 1.20x speedup over CXL-SHM, whereas a more substantial 2.46x improvement is observed for PageRank. As depicted in Figure 12(b), the execution time breakdown for PageRank shows that AEP-DSM reduces the transfer latency from 68.2µs to 1.6µs, a 42.6× reduction, yet the transfer cost remains only a small fraction of the total execution time. The major benefit comes from AEP-DSM's efficient memory allocation, which dominates the communication cost and drives the end-to-end speedup. As shown in Figure 12(c), the advantage grows with larger PageRank working sets, as more allocation is required during communication. Overall, these results demonstrate that AEP-DSM delivers substantial performance gains for intra-node DSM-based communication in realistic serverless workflows.

As function communication shifts from fully local to fully remote, the overall performance improvement of AEP-DSM decreases, as shown in Figure 12(d) and (e), due to its fast-path optimizations being less used. In the chain benchmark, AEP-DSM incurs overheads similar to those of CXL-SHM. Nevertheless, AEP-DSM still achieves a 2.12× speedup over CXL-SHM in PageRank. As noted earlier, this improvement stems from the dominant role of memory allocation in PageRank. Thus, even when scheduling policies prevent intra-node optimizations, AEP-DSM achieves performance gains through efficient memory management.

5.4 Failure Recovery

To evaluate AEP-DSM's recovery efficiency, we deployed four clients across two nodes. Clients 1, 2, and 3 ran on node A while Client 4 operated on node B. We actively triggered failures by injecting explicit exit calls into client code and recorded service performance through in-memory throughput measurements. During client failure tests, we crashed only Client 1. For node failure simulations, we simultaneously crashed all three clients on node A (Clients 1-3).

Client-level failure recovery. During client faults (left subfigure of Figure 13), AEP-DSM sustains continuous service with minimal performance fluctuation, owing to its optimized intra-node metadata management. In our experiment,

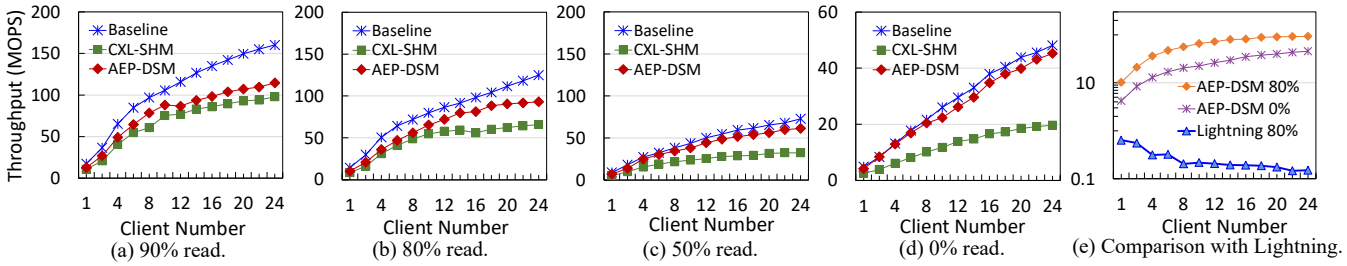


Figure 11. Performance comparisons of KV store under various read/write ratios among AEP-DSM, CXL-SHM, and Lightning.

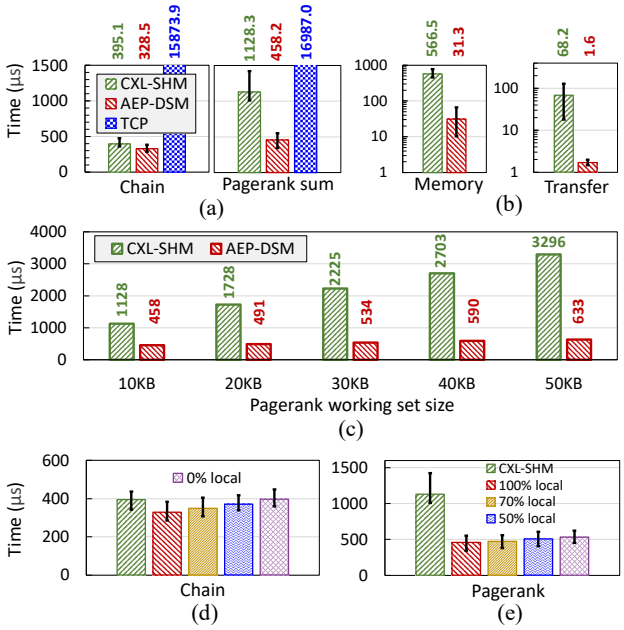


Figure 12. Performance of serverless workflow.

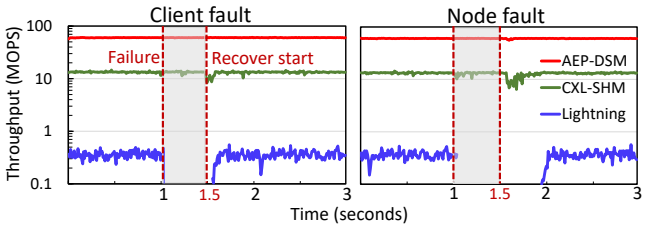


Figure 13. Service behavior during client and node fault.

each client allocates 2 million objects before triggering a fault. During recovery, AEP-DSM achieved a throughput of 253.4 million objects per second, compared to only 17.4 million obj/s for CXL-SHM. The overhead in CXL-SHM primarily stems from cache flushes and CXL memory accesses incurred by global reference count updates. These results demonstrate that AEP-DSM is particularly well-suited for high-density client deployments.

Node-level failure recovery. As shown in the right subfigure of Figure 13, CXL-SHM causes significant throughput fluctuation (>10%) whereas AEP-DSM maintains minimal fluctuation (<5%). This stems from AEP-DSM's hierarchical design isolating notra-node allocators from cross-node sharing, confining interference exclusively to the shared internode allocator. CXL-SHM experiences prolonged degradation (about 0.43s for 4M objects) during node recovery since it recovers each client individually, while AEP-DSM performs single node-level recovery at intra-node layer with complexity matching one CXL-SHM client recovery, minimizing service disruption.

6 Related Work

Inter-service communication. Modern distributed applications enable scalable distributed applications but intensify inter-service communication demands. This has driven optimizations including RPC stack redesign [68], low-latency channel construction [25, 46, 57], connection reuse [16, 66], and RDMA-enhanced networking [17, 36, 60]. Recent work further eliminates I/O overhead through DSM for in-memory intermediate data sharing [29, 58].

Memory consistency protocol. The emergence of interconnect technologies such as CXL has enabled unified DSM management, causing research across software [14, 23, 38], hardware [22, 26, 28, 59], and hybrid [61] approaches to implement various consistency protocols at different layers of the system stack, offering applications a unified memory abstraction to reduce software complexity and data movement overhead. For example, Infiniswap [23] leverages RDMA along with the OS paging mechanisms to expose a unified memory view to applications. CTXNL [59], on the other hand, tailors CXL's cache coherence protocol for transactional systems to minimize cross-node coherence overhead.

Our AEP-DSM system is built atop CXL 3.0, leveraging hardware support to construct a unified memory view across nodes. Importantly, the AEP mechanism is agnostic to the underlying coherence protocol.

DSM management. A broad spectrum of DSM-based memory management schemes [15, 19, 37, 47, 67, 69, 70] seeks to improve fault tolerance by refining recovery protocols or tailoring distributed primitives for specific applications. Ralloc [15] pioneered fault-tolerant allocators in heterogeneous memory, laying a theoretical foundation but leaving challenges unresolved for multi-client partial-failure problem. FaRM [19] reduces Paxos overhead with a four-phase commit

protocol optimized for distributed transactions. Lupin [69] integrates epoch-based protocols and logging to provide recoverable locks, covering error detection, recovery, and garbage collection. DRust [37] leverages Rust's ownership model to improve memory consistency in DSM, though its reliance on Rust constrains compatibility with other languages.

To the best of our knowledge, no prior work has exploited the hierarchical nature of distributed memory to eliminate the overhead of heavy-weight fault tolerance algorithms and cross-node consistency protocol.

7 Conclusion

We presented Atomic Execution Protection (AEP), a lightweight kernel-assisted mechanism that guarantees atomic completion of DSM allocator operations, tolerating intra-node partial failures without costly distributed coordination. Building on AEP, we designed AEP-DSM, a hierarchical DSM allocator that separates intra-node fast paths from cross-node coordination through metadata decoupling and hierarchical reference counting. Our prototype outperforms state-of-the-art resilient DSMs, improving intra-node allocation throughput and object transfers by over an order of magnitude, and delivering substantial gains in KV-stores and serverless workflows, while supporting efficient client- and node-level recovery. These results demonstrate that lightweight OS support for intra-node failures, combined with distributed protocols for cross-node sharing, provides a practical path toward high-performance, failure-resilient DSM in CXL-enabled memory sharing scenarios.

8 Acknowledgment

We thank the anonymous reviewers and our shepherd, Chloe Alverti, for their constructive feedback and suggestions. This work is supported by the National Key Research and Development Program of China under grant 2022YFB4502001, Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China under grant JYB2025XDXM103, and Science and Technology Plan Project of Hubei Province under grant 2025CSA056. The corresponding authors are Hang Huang, Jia Rao, Hui Lu and Song Wu.

References

[1] 2015. *Analyzing Intel SDE's TSX-related log data for capacity aborts*. https://www.intel.com/content/www/us/en/developer/articles/technical/analyzing-intel-sdes-tsx-related-log-data-for-capacity-aborts.html.
[2] 2017. *Site Reliability*. https://sre.google/sre-book/availability-table/.
[3] 2025. *About CXL*. https://computeexpresslink.org/about-cxl/.
[4] 2025. *AWS Lambda*. https://aws.amazon.com/lambda.
[5] 2025. *Azure Functions*. https://azure.microsoft.com/products/functions.
[6] 2025. *Google Cloud Functions*. https://cloud.google.com/functions.
[7] 2025. *Kubernetes scheduler*. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/.
[8] 2025. *Mimalloc*. https://github.com/microsoft/mimalloc.
[9] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose Faleiro, Gohar Irfan Chaudhry, Inigo Goiri, Ricardo Bianchini, Daniel S Berger, and Rodrigo Fonseca. 2023. Palette Load Balancing: Locality Hints for Serverless Functions. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. New York, NY, USA, 365–380.
[10] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. Boston, MA, 923–935.
[11] Marco Barletta, Marcello Cinque, Catello Di Martino, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2024. Mutiny! How Does Kubernetes Fail, and What Can We Do About It?. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 1–14.
[12] 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 (PPoPP)*. New York, NY, USA, 168–176.
[13] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA, 117–128.
[14] 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.
[15] 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 ACM SIGPLAN International Symposium on Memory Management (ISMM)*. New York, NY, USA, 60–73.
[16] Qiong Chen, Jianmin Qian, Yulin Che, Ziqi Lin, Jianfeng Wang, Jie Zhou, Licheng Song, Yi Liang, Jie Wu, Wei Zheng, et al. 2024. Yuanrong: A production general-purpose serverless system for distributed applications in the cloud. In *Proceedings of the Annual Conference of ACM Special Interest Group on Data Communication (SIGCOMM)*. 843–859.
[17] Marcin Copik, Konstantin Taranov, Alexandru Calotoiu, and Torsten Hoefler. 2023. rFaaS: Enabling high performance serverless with RDMA and leases. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 897–907.
[18] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.
[19] 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 Symposium on Operating Systems Principles (SOSP)*. New York, NY, USA, 54–70.

[20] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering* (2017), 195–216.

[21] Dimitra Giantsidi, Emmanouil Giortamis, Nathaniel Tornow, Florin Dinu, and Pramod Bhatotia. 2023. Flexlog: A shared log for stateful serverless computing. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. 195–209.

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

[23] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Boston, MA, 649–667.

[24] Jennifer Hamilton. 1997. Montana Smart Pointers: They're Smart, and They're Pointers. In *COOTS*. 21–40.

[25] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA, 152–166.

[26] P. Keleher, A.L. Cox, and W. Zwaenepoel. 1992. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. 13–21.

[27] Jeongchul Kim and Kyungyong Lee. 2019. Functionbench: A suite of workloads for serverless cloud function service. In *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD)*. 502–504.

[28] Kyungsan Kim, Hyunseok Kim, Jinin So, Wonjae Lee, Junhyuk Im, Sungjoo Park, Jeonghyeon Cho, and Hoyoung Song. 2023. SMT: Software-Defined Memory Tiering for Heterogeneous Computing Systems With CXL Memory Expander. *IEEE Micro* 43, 2 (2023), 20–29.

[29] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 427–444.

[30] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 805–820.

[31] 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 Symposium on Operating Systems Principles (SOSP)*. New York, NY, USA, 488–504.

[32] Xiang Li, Linfeng Wen, Minxian Xu, and Kejiang Ye. 2023. An Interference-aware Approach for Co-located Container Orchestration with Novel Metric. In *Proceedings of the IEEE International Conferences on Internet of Things (iThings) and IEEE Green Computing Communications (GreenCom) and IEEE Cyber, Physical Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*. 600–607.

[33] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. 2022. RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. Carlsbad, CA, 53–68.

[34] Changyuan Lin, Nima Mahmoudi, Caixiang Fan, and Hamzeh Khazaei. 2022. Fine-grained performance and cost modeling and optimization for faas applications. *IEEE Transactions on Parallel and Distributed Systems* 34, 1 (2022), 180–194.

[35] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S. Berger, Marie Nguyen, Xun Jian, Sam H. Noh, and Huaicheng Li. 2025. Systematic CXL Memory Characterization and Performance Analysis at Scale. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA, 1203–1217.

[36] Fangming Lu, Xingda Wei, Zhuobin Huang, Rong Chen, Minyu Wu, and Haibo Chen. 2024. Serialization/Deserialization-free State Transfer in Serverless Workflows. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. Association for Computing Machinery, New York, NY, USA, 132–147.

[37] Haoran Ma, Yifan Qiao, Shi Liu, Shan Yu, Yuanjiang Ni, Qingda Lu, Jiesheng Wu, Yiying Zhang, Miryung Kim, and Harry Xu. 2024. DRust: Language-Guided Distributed Shared Memory with Fine Granularity, Full Transparency, and Ultra Efficiency. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Santa Clara, CA, 97–115.

[38] Teng Ma, Zheng Liu, Chengkun Wei, Jialiang Huang, Youwei Zhuo, Haoyu Li, Ning Zhang, Yijin Guan, Dimin Niu, Mingxing Zhang, and Tao Ma. 2024. HydraRPC: RPC in the CXL Era. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. Santa Clara, CA, 387–395.

[39] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware data passing for chained serverless applications. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 285–301.

[40] Ethan L. Miller, George Neville-Neil, Achilles Benetopoulos, Pankaj Mehra, and Daniel Bittman. 2023. Pointers in Far Memory. *Commun. ACM* 66, 12 (2023), 40–45.

[41] Montage 2025. *Montage CXL Memory Expander Controller*. Montage, https://www.montage-tech.com/MXC/M88MX5891&5851.

[42] 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 USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[43] Musl 2025. *A simplified C standard library*. Musl, https://musl.libc.org/.

[44] Russell Power and Jinyang Li. 2010. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[45] Pedro Ramalhete and Andreia Correia. 2017. Brief Announcement: Hazard Eras - Non-Blocking Memory Reclamation. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. New York, NY, USA, 367–369.

[46] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. FaaST: A Transparent Auto-Scaling Cache for Serverless Applications. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. New York, NY, USA, 122–137.

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

[48] Bo Sang, Pierre-Louis Roman, Patrick Eugster, Hui Lu, Srivatsan Ravi, and Gustavo Petri. 2020. Plasma: programmable elasticity for stateful cloud computing applications. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. 1–15.

[49] Divyanshu Saxena, William Zhang, Madhav Tummala, Saksham Goel, and Aditya Akella. 2023. Invited Paper: Towards Efficient Microservice Communication. In *Proceedings of the Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating Algorithms for Distributed Systems (ApPLIED)*. New York, NY, USA, Article 8, 5 pages.

[50] Larissa Schmid, Marcin Copik, Alexandru Calotoiu, Laurin Brandner, Anne Koziolek, and Torsten Hoefler. 2025. SeBS-Flow: Benchmarking Serverless Cloud Function Workflows. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. New York, NY, USA, 902–920.

[51] Isak Shabani, Endrit Mëziu, Blend Berisha, and Tonit Biba. 2021. Design of modern distributed systems based on microservices architecture. *International Journal of Advanced Computer Science and Applications* 12, 2 (2021).

[52] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural implications of function-as-a-service computing. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1063–1075.

[53] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 205–218.

[54] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. 2023. FUSEE: A Fully Memory-Disaggregated Key-Value Store. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. Santa Clara, CA, 81–98.

[55] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592* (2020).

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

[57] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 443–457.

[58] 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 (HotOS)*. 191–198.

[59] Zhao Wang, Yiqi Chen, Cong Li, Yijin Guan, Dimin Niu, Tianchan Guan, Zhaoyang Du, Xingda Wei, and Guangyu Sun. 2025. CTXNL: A Software-Hardware Co-designed Solution for Efficient CXL-Based Transaction Processing. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA, 192–209.

[60] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. No Provisioned Concurrency: Fast RDMA-codesigned Remote Fork for Serverless Computing. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Boston, MA, 497–517.

[61] Tong Xing and Antonio Barbalace. 2025. Rethinking Applications' Address Space with CXL Shared Memory Pools. In *Proceedings of the Workshop on Heterogeneous Composable and Disaggregated Systems (HCDS)*. New York, NY, USA, 52–59.

[62] Yujie Yang, Lingfeng Xiang, Peiran Du, Zhen Lin, Weishu Deng, Ren Wang, Andrey Kudryavtsev, Louis Ko, Hui Lu, and Jia Rao. 2025. Architectural and System Implications of CXL-enabled Tiered Memory. *arXiv preprint arXiv:2503.17864* (2025).

[63] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2023. Following the data, not the function: Rethinking function orchestration in serverless computing. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 1489–1504.

[64] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing serverless platforms with serverlessbench. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. 30–44.

[65] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-tolerant and transactional stateful serverless workflows. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 1187–1204.

[66] Jie Zhang, Xuzheng Chen, Yin Zhang, and Zeke Wang. 2024. DmRPC: Disaggregated Memory-aware Datacenter RPC for Data-intensive Applications. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 3796–3809.

[67] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. 2023. Partial Failure Resilient Memory Management System for (CXL-based) Distributed Shared Memory. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. New York, NY, USA, 658–674.

[68] Xiangfeng Zhu, Yang Zhou, Yuyao Wang, Xiangyu Gao, Arvind Krishnamurthy, Sam Kumar, Ratul Mahajan, and Danyang Zhuo. 2025. Rethinking RPC Communication for Microservices-based Applications. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*. 158–164.

[69] Zhiting Zhu, Newton Ni, Yibo Huang, Yan Sun, Zhipeng Jia, Nam Sung Kim, and Emmett Witchel. 2024. Lupin: Tolerating Partial Failures in a CXL Pod. In *Proceedings of the Workshop on Disruptive Memory Systems (DIMES)*. New York, NY, USA, 41–50.

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