

Starfish: Fault-Tolerant Far Memory with Low Resource and Performance Overhead

Yanwen Xia
SKLP, Institute of Computing
Technology, CAS; University of
Chinese Academy of Sciences
Beijing, Beijing, China
xiayanwen25z@ict.ac.cn

Yuzheng Wang
Peking University
Beijing, Beijing, China
yuzhengwang@stu.pku.edu.cn

Xiaobing Feng
SKLP, Institute of Computing
Technology, CAS; University of
Chinese Academy of Sciences
Beijing, Beijing, China
fxb@ict.ac.cn

Benyong Deng
SKLP, Institute of Computing
Technology, CAS; University of
Chinese Academy of Sciences
Beijing, Beijing, China
u202312400@hust.edu.cn

Quanxi Li
SKLP, Institute of Computing
Technology, CAS; University of
Chinese Academy of Sciences
Beijing, Beijing, China
liquanxi20z@ict.ac.cn

Huimin Cui
SKLP, Institute of Computing
Technology, CAS; University of
Chinese Academy of Sciences
Beijing, Beijing, China
cuihm@ict.ac.cn

Hong Huang
SKLP, Institute of Computing
Technology, CAS; University of
Chinese Academy of Sciences
Beijing, Beijing, China
huanghong24s@ict.ac.cn

Xingda Wei
Shanghai Jiao Tong University
Shanghai, Shanghai, China
wxdwfc@sjtu.edu.cn

Chenxi Wang
SKLP, Institute of Computing
Technology, CAS; University of
Chinese Academy of Sciences
Beijing, Beijing, China
wangchenxi@ict.ac.cn

Abstract

Fault-tolerant (FT) far memory systems protect remote state against memory node failures. Existing designs optimize erasure coding (EC) around coarse memory units, forcing objects to fit those units and causing redundant protection work and amplified remote accesses. Starfish is object-first: it first reuses valid recovery paths, through retained remote backups and recomputation, and then uses object behavior to choose the protection method, granularity, and layout for the remaining remote object states.

Across 6 applications and 8 workloads, Starfish improves performance by $2.1\times$ over Hydra and $1.9\times$ over Carbink on average. It reaches 88.3% of the performance of the same system without FT. Starfish also reduces network traffic by $8.5\times$ and lowers memory node CPU usage by $7.5\times$ over Carbink with no extra remote memory cost, while preserving efficient recovery.

CCS Concepts: • Software and its engineering → Distributed memory; • Computer systems organization → Reliability.

Keywords: Far memory, Fault tolerance, Runtime systems

ACM Reference Format:

Yanwen Xia, Benyong Deng, Hong Huang, Yuzheng Wang, Quanxi Li, Xingda Wei, Xiaobing Feng, Huimin Cui, and Chenxi Wang. 2026. Starfish: Fault-Tolerant Far Memory with Low Resource and Performance Overhead. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026)*, November 15–18, 2026, Hong Kong, China. ACM, New York, NY, USA, 18 pages. <https://doi.org/10.1145/3828523.3857010>

1 Introduction

Memory disaggregation lets applications use memory from other nodes as *remote memory*, meeting growing capacity demand and improving datacenter memory utilization. Recent high-speed interconnects such as InfiniBand [49] and CXL [26] make this design increasingly practical [9, 21, 41, 54, 56, 83, 84, 98, 114, 119].

Fault tolerance (FT) becomes more challenging when application memory moves from one local node to multiple remote nodes. Once an application’s remote memory state spans multiple memory nodes, even a single memory node failure may make part of that state unavailable, causing application crashes or forcing expensive recovery [55, 121]. Production experience from Google and Microsoft shows that node failures are common and disruptive [14, 64, 94]. Traditional approaches tolerate such failures by periodically checkpointing application state to durable storage; recent systems instead maintain *in-memory* redundancy, as recovery from memory is much faster than from storage [23, 62]. Recent far memory



This work is licensed under a Creative Commons Attribution 4.0 International License.

SIGOPS ATC 2026, Hong Kong, China

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2818-1/2026/11

<https://doi.org/10.1145/3828523.3857010>

systems also treat FT as a core design concern and maintain redundancy in remote memory [47, 55, 71, 75, 121].

Unfortunately, such in-memory FT is not cheap. Keeping remote memory recoverable requires redundant state across memory nodes. Replication (Rep) must keep $m+1$ full copies to tolerate m memory node failures, so recent far memory systems use erasure coding (EC) to reduce redundancy space [47, 55, 121]. Reed–Solomon (RS) coding [81] stores m parity blocks for every n data blocks, reducing total memory consumption from $(m+1) \times (n+m)/n$.

State-of-the-art solutions. However, EC-based systems trade lower space overhead for parity maintenance and layout constraints. We use two representative systems, Hydra [55] and Carbink [121], to illustrate these costs. Hydra protects remote pages, while Carbink packs runtime-managed objects into larger spans and protects spansets; §2.3 describes both mechanisms. We refer to the operation that reads remote state into local memory as *fetch*, and the operation that writes local state back to remote memory under memory pressure as *eviction*.

Our experiments in §3.1 show that, across four applications, Hydra and Carbink slow overall performance by $1.6\times$ and $1.5\times$ over Carbink-w/oFT, respectively. Carbink-w/oFT is a version of Carbink that keeps the same span-based memory management but disables FT. The main performance overhead comes from FT-amplified network traffic: under $RS(n, m)$, each eviction writes the evicted data plus m/n parity data, slowing local memory reclamation and later fetches. Carbink further consumes memory node CPU resources up to 59.0% of compute node CPU usage, mostly for compaction.

Root cause of inefficiency. These inefficiencies stem from organizing objects around EC layouts. To make EC efficient, Hydra and Carbink first choose a fixed protection unit and then force objects to fit this unit (page or span). Object access and protection layout are coupled by the EC organization, rather than decided from object-level information.

Target. This paper targets high application performance and low FT resource costs while maintaining the same failure tolerance as existing systems. These costs include network traffic, memory node CPU usage, and remote memory usage.

Major insights. Our key insight is to make FT decisions from an object-first view, instead of first choosing a fixed protection unit and forcing objects to fit it. Recent runtime-based systems show that objects expose useful application semantics and runtime information [43, 61, 83, 92, 104]. Object-level information can guide FT decisions in two directions.

First, an object-first view helps decide whether a remote object state can reuse a valid recovery path, instead of generating a new protected version. We identify two such opportunities. The first is *clean eviction*: if an object has been fetched but remains unmodified before eviction, the runtime can retain its previous remote copy as a valid backup and avoid the writeback. This opportunity is important under EC-based FT where each eviction also triggers parity maintenance. In our evaluated workloads, 4 of 8 have clean-eviction ratios above 90% [109]. The second is recomputation: some remote object

states are derivable from protected inputs, so the runtime can record how to regenerate them after failures, instead of continuously protecting them with EC or Rep.

Second, an object-first view helps reduce the cost of new protection to states. Object size, lifetime, and update intensity provide signals for choosing how to protect each state. For example, frequently updated or short-lived states may be better protected with Rep, while more stable states can be batched into EC stripes for lower space overhead. These signals also guide the protection granularity and EC layout, such as whether to pack multiple small objects together or split a large object into fragments. Fine-grained access workloads further expose this opportunity: many applications use only a small fraction of each fetched coarse unit, leaving room for object-granular management to reduce access amplification.

Challenges. However, exploiting object-first management for FT introduces three challenges. First, reusing existing recovery paths must remain space-efficient and correct. Retaining remote backups for all fetched objects would turn local memory into an inclusive cache and waste remote memory, while recomputation must ensure that declared inputs are recoverable and unchanged when the recipe is used. Second, remote object states that still need new protection cannot simply use existing EC layouts at object granularity. A Hydra-style design would split each small object into EC chunks across memory nodes, making fetches sensitive to stragglers, while a Carbink-style design makes compaction more frequent and expensive (see §3.2). Third, object-first FT requires runtime object behavior information, but tracking every object event or access would be expensive. The runtime must therefore obtain useful object-level signals at low overhead.

To address these challenges, we propose Starfish, an object-first fault-tolerant runtime-managed far memory system. Starfish uses object behavior and semantics to choose among backup reuse, recomputation, replication, and EC. Reusing Valid Recovery Paths selectively retains useful remote backups under a bounded budget and uses runtime abstractions to track recomputable states and their protected inputs. Behavior-Guided Protection then uses behavior profiles to choose the protection method, granularity, and layout for states that require new remote protection. Starfish obtains these signals at low overhead by grouping objects with similar behavior.

Results. We evaluate Starfish with 6 applications and 8 workloads covering diverse types and memory access patterns. Starfish improves overall performance by $2.1\times$ over Hydra and $1.9\times$ over the state-of-the-art runtime-managed far memory FT system Carbink on average, and by up to $3.9\times$ over Carbink. It reaches 88.3% of the performance of a system without FT. Starfish also reduces network traffic by $8.5\times$ over Carbink, lowers memory node CPU usage by $7.5\times$, and uses 4.9% less remote memory. These results show that object-first FT can preserve strong recoverability while substantially reducing the performance and resource costs of far memory FT. Starfish is available at <https://github.com/ICTPLSys/Starfish>.

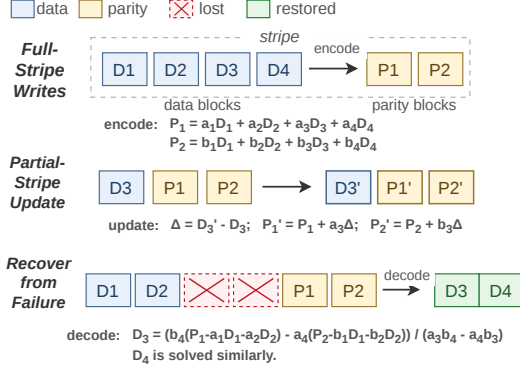


Figure 1. Erasure coding protects data blocks by redundancy information in parity blocks. In this RS(4,2) example, a full-stripe write groups data blocks D_1 – D_4 and computes two parity blocks, P_1 and P_2 . A partial-stripe update to D_3 must read the old D_3 , P_1 and P_2 , then write D_3' , P_1' , and P_2' . When both D_3 and D_4 are lost, *decoding* uses the surviving data and parity blocks (D_1 , D_2 , P_1 , and P_2) to recover them. The m parity blocks can tolerate m simultaneous node failures.

2 Background and Existing Approaches

2.1 Far Memory Systems

Far memory systems expand a compute node’s local memory with capacity from remote memory nodes, and tend to place frequently accessed data in local memory for better performance. These systems share common operations: *allocating* application state, *fetching* remote state into newly allocated local memory (swap in), *evicting* local state back to remote memory to reclaim local memory under pressure (swap out), and *freeing* the associated local and remote memory space.

Current far memory systems fall into three categories: OS paging-based[8, 41, 74, 77, 84, 100, 106, 117], hardware-assisted[21, 39, 44, 56, 58], and runtime-managed[20, 24, 43, 46, 66, 83, 92, 98, 99, 101, 116]. OS paging-based designs fetch and evict data at page granularity, which incur swap-path overhead and access amplification for fine-grained accesses[43, 61, 83]. Runtime-managed designs instead expose remote memory as a remote heap and manage remote state through application-integrated runtimes [83]; they fetch and evict data at object granularity, avoiding access amplification and enabling finer-grained control over remote state [43, 61, 92, 116]. Hardware-assisted designs improve access efficiency with specialized hardware support and are orthogonal to the software mechanisms studied in this paper.

2.2 Common Fault Tolerance Mechanisms

Fault tolerance protects systems from failures by adding redundancy. Common mechanisms include replication [5, 11, 29, 34, 96, 107, 111, 112], checkpointing [6, 25, 51, 68, 69, 76], and erasure coding (EC) [7, 10, 37, 48, 52, 86, 87, 105]. Replication tolerates f failures by storing $f + 1$ copies across separate failure domains. Checkpointing periodically saves

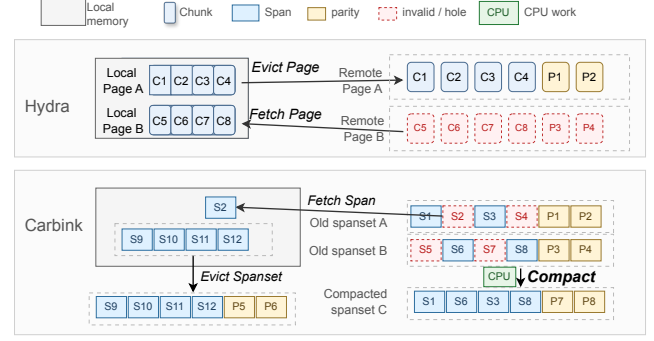


Figure 2. Data paths of Hydra and Carbink.

application state and restores from the latest checkpoint, making recovery coarse-grained.

Erasure coding divides data into n equal-sized *data* blocks and computes m equal-sized *parity* blocks using algorithms such as Reed–Solomon (RS) coding [81]. Together, the $n + m$ blocks form an EC *stripe*; RS(n, m) denotes a stripe with n data blocks and m parity blocks. As shown in Figure 1, *encoding* computes parity from data blocks, and *decoding* reconstructs up to m lost blocks from any n surviving blocks in the same stripe. EC stores only m/n extra parity data, much less than replication under the same failure tolerance.

However, EC makes updates and recovery expensive. An *in-place partial-stripe update* modifies part of data blocks in an existing stripe, and must update all m parity blocks, requiring m parity reads and m parity writes. An *out-of-place full-stripe write* creates a new stripe by writing n data and m parity blocks, incurring only m/n parity-write overhead. Recovery is costly: reconstructing a lost data block requires fetching n surviving blocks and decoding them. Therefore, production systems mainly use EC for warm or cold data with infrequent updates to reduce space overhead [23, 36, 48, 62, 70, 78, 79, 122].

2.3 Existing Fault-Tolerant Far Memory Designs

Hydra [55] and Carbink [121] are two representative EC-based far memory designs. Hydra is paging-based and protects remote pages, while Carbink is runtime-managed and protects remote objects.

Hydra. Hydra protects each remote page as one EC stripe. On eviction, the compute node splits a page into n data chunks, computes m parity chunks, and writes the $n + m$ chunks across memory nodes. On fetch, the compute node reads n chunks and decodes the page. This fanout makes fetches sensitive to stragglers, which Hydra mitigates with late binding [55]. Each eviction remains a full-stripe write with m/n parity traffic. Encoding and decoding run on the compute node, so Hydra imposes no EC computation on memory nodes.

Carbink. Carbink protects remote objects by first packing them into larger fixed-size *spans* and then grouping every n data spans with m parity spans into an EC *spanset*. On eviction, Carbink writes spans out of place: it batches n evicted data spans, computes m parity spans, and distributes the $n + m$

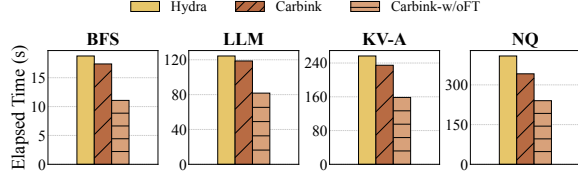


Figure 3. Elapsed time of applications. Lower is better.

spans across memory nodes. On fetch, Carbink reads only the data span containing the target object from one node.

This design reduces straggler sensitivity compared to Hydra, but complicates remote space management. Each eviction is a full-stripe write and adds m/n parity traffic. As Carbink uses exclusive remote memory, fetching a span into local memory invalidates its original remote span slot, leaving a dead span inside the spanset. Such dead spans cannot be reclaimed immediately if other live spans in the same spanset still depend on the spanset for recovery. Directly filling such dead spans with new spans would require span-sized partial-stripe updates with high parity traffic, so Carbink performs *compaction*: it merges fragmented spansets into new full spansets and reclaims dead spans. Compaction avoids frequent partial parity updates but adds extra remote memory usage and memory node computation.

3 Motivation

We quantify the performance and resource overheads of existing fault-tolerant far memory systems. We then discuss the opportunities enabled by object-first runtime design and explain why simply applying it to existing systems is insufficient.

3.1 Problem: Overhead From Fault Tolerance

Baselines and Workloads. We use two state-of-the-art fault-tolerant far memory systems, Hydra [55] and Carbink [121], as baselines to demonstrate the application performance degradation, and resource overhead in CPU, memory and network traffic. We compare them against *Carbink-w/oFT*, a system that uses the same span-based memory management as Carbink but disables fault tolerance. We evaluate four real-world applications: GAPBS BFS (BFS [15]), llama.cpp (LLM [38]), in-memory key-value store (KV [45]), and Nhop (NQ [40]), which have different memory access patterns and data structures. The local memory is configured as 25% (25% of applications’ working set resides in local memory). Refer to § 6.1 for more setup details.

Overall performance overhead. We first evaluate the overall performance by using application throughput under the same workload. As shown in Figure 3, Carbink performs better than Hydra by 1.1× to 1.2× across applications. However, when compared to *Carbink-w/oFT*, performance decreased by 1.6× and 1.5× on average. The next paragraph explains why.

Network Traffic Overhead. Network traffic explains why FT turns eviction into a new bottleneck for far memory. In *Carbink-w/oFT*, fetch and eviction are roughly balanced: evicting one

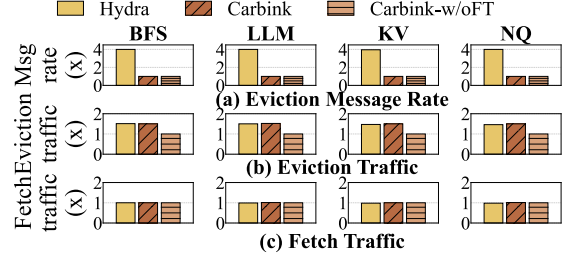


Figure 4. Network traffic and network message rate across applications. Normalized to Carbink-w/oFT and lower is better.

unit of object data frees one unit of local memory, which can then be used to fetch one unit of remote data. Under Hydra and Carbink, however, each eviction also writes parity data for protection. With RS(4,2), evicting one unit of object data generates 1.5× write traffic. These applications already saturate the NIC bandwidth at about 12 GB/s in Carbink-w/oFT. Therefore, the extra parity traffic directly lowers the memory reclamation rate to only 67% of the non-FT rate. Later fetches are limited by this rate, as shown in Figure 4(b) and (c). Thus, FT makes eviction the bottleneck of the far memory data path. Besides, Carbink outperforms Hydra because it is less exposed to fetch-path stragglers, as discussed in §2.3. Figure 4(a) shows that Carbink reduces the message rate by approximately 4.0× compared with Hydra.

Despite the overall performance gain compared to Hydra, Carbink also consumes more remote space and CPU resources on the memory nodes for compaction.

Space Overhead. Both Hydra and Carbink use erasure-coding parity to protect far memory data. As shown in Figure 5, they consume up to 52.3% and 76.7% more remote memory space than *Carbink-w/oFT*, respectively. This overhead comes from two sources. First, RS(4,2) parity adds up to 50% space overhead over the original remote data. Second, memory fragmentation adds another 2.3% in Hydra and 26.7% in Carbink. Carbink suffers more fragmentation because of its coarse-grained swapping design. To avoid expensive EC partial updates, Carbink evicts cold spans out of place and writes them to new memory node locations to form new spansets. It then relies on compaction to reclaim dead spans, as discussed in § 2.3. This reduces partial-update traffic, but consumes nontrivial memory node CPU resources.

CPU Overhead. As shown in Figure 5, Carbink consumes substantial memory node CPU resources. On average, its memory node CPU usage reaches 35.9% of the compute node CPU usage, and rises to 59.0% for KV. Most (75.8%) of this CPU time is spent on compaction, which reclaims stale spans left by Carbink’s out-of-place eviction path. The cost varies significantly across applications. KV and NQ consume, on average, 4.1× more memory node CPU resources than BFS and LLM. KV and NQ’s random access patterns create more dead spans, increasing compaction CPU cost.

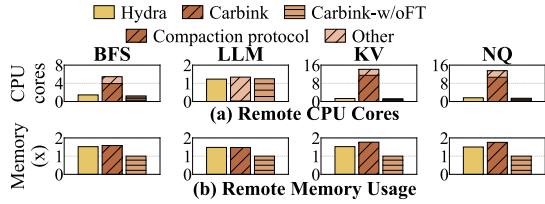


Figure 5. Remote CPU usage and remote memory usage across applications. Remote memory usage is normalized to Carbink-w/oFT. CPU usage is represented as average CPU cores utilized per second. Lower is better.

In this motivation experiment, we assign all 24 cores of each memory node for compaction to minimize fragmentation. Reducing the compaction budget to 6 remote cores, 25.0% of the application CPU resources, increases Carbink’s remote memory consumption from 1.7 $\times$ to 2.3 $\times$. This confirms that Carbink controls fragmentation by spending substantial memory node CPU resources. This cost matters because disaggregated memory nodes are typically provisioned with weak compute [44, 84], and CXL memory devices may expose no usable CPU at all [39, 56].

3.2 Opportunities and Challenges of Object-First FT

Opportunities. Runtime-based far memory systems already manage memory at object granularity. Objects expose application semantics and runtime behavior for optimization. We argue that the same object view can also reduce FT overhead. First, it helps decide whether a remote object state can reuse a valid recovery path instead of generating a new protected version. A clean object with a valid remote backup can be evicted without new data or parity writes, and recomputable states can be rebuilt from protected inputs instead of being protected. In our workloads, 4 of 8 have clean-eviction ratios above 90%. Second, for remote object states that do require new protection, object size, lifetime, update intensity, and access behavior can guide the protection method, granularity, and layout. For fine-grained access workloads, an object-granular runtime without FT improves performance by 2.7 $\times$ over *Carbink-w/oFT* on average, showing the benefit of preserving object granularity before adding FT. Carbink is well matched to its target workloads, where large objects occupy the majority of memory [121, §3.3]; our point is that modern workloads also include many fine-grained objects, where span-based management and protection perform poorly [109].

Challenges. These opportunities are hard to realize with existing FT designs. Simply keeping a remote backup for every local object is too expensive: under a 50% local memory setting, an inclusive remote cache requires extra remote space equal to 1.0 $\times$ of the local memory footprint. The runtime needs object-level knowledge to keep useful backups. Recomputation must guarantee correctness: if inputs of recomputation recipes have changed, regenerating the object may produce incorrect state.

Naively making existing EC layouts object-granular is also insufficient. We build two variants, Hydra-Obj and

Carbink-Obj, over the same object-level runtime without FT; we call this baseline Non-FT. Hydra-Obj treats each object as an EC stripe, which increases message rate by 4.0 $\times$ and P99 latency by 9.8 $\times$ over Non-FT. Carbink-Obj packs n equal-size objects into an EC stripe, but object-size heterogeneity makes stripe reclamation harder; it increases remote memory usage by 1.2 $\times$ and CPU usage by 2.3 $\times$ over Carbink. Finally, tracking every object event would add critical-path overhead and make object behavior profiling a new bottleneck.

Together, these observations motivate Starfish: an object-first FT design that avoids unnecessary protection work and reduces the cost of new protection.

4 Starfish Design

4.1 Overview

Deployment Scope. Starfish targets far memory systems with multiple memory nodes in separate failure domains, where nodes can fail independently. These nodes can be RDMA-connected memory nodes or separate physical CXL memory devices connected through a CXL switch. A single CXL memory expansion card attached to a single host has only one memory-device failure domain and is therefore outside our scope. Our prototype uses RDMA-connected memory nodes.

Fault Model. Like prior fault-tolerant far memory systems [47, 55, 71, 121], Starfish targets fail-stop failures of memory nodes while the compute node remains alive. It protects remote application state against a configured number of simultaneous memory node failures, but does not protect compute node state or tolerate Byzantine failures. Under this fault model, Starfish provides the same application-visible recoverability as Hydra and Carbink while reducing fault-free performance and resource overheads.

Recovery-path invariant. Starfish treats fault tolerance as maintaining a valid recovery path for each remote object state. A recovery path can be an existing protected remote state, a recompute recipe to regenerate the state, or a new state protected by Rep or EC. During normal execution, object events such as fetch, eviction, mutable access, and free may change an object’s location or value. Starfish preserves correctness by ensuring that, before a remote object state leaves local memory, the state either still has a valid existing recovery path or has a new protected state.

Workflow. Figure 6 shows this workflow. For each object event, Starfish first checks whether the resulting remote object state can reuse an existing recovery path. Clean evictions reuse retained remote backups, and recomputable states use a recipe to regenerate the state from protected inputs. The remaining states become protection requests, for which Starfish creates new protected state and uses object behavior to choose the protection method, granularity, and layout. The rest of this section follows this flow: §4.2 describes the runtime substrate, §4.3 explains recovery-path reuse, §4.4 describes new protected state creation, and §4.5 describes recovery.

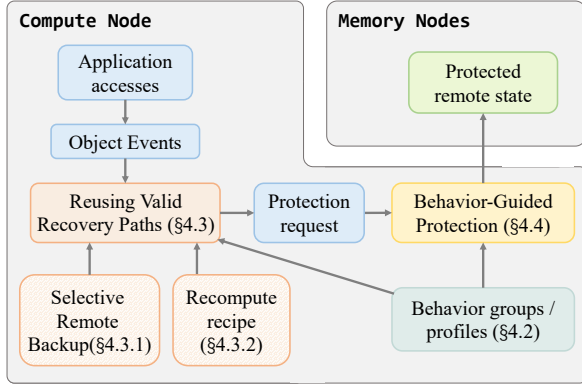


Figure 6. Starfish design overview.

4.2 Runtime Substrate: Behavior Groups and Metadata

Starfish uses two runtime structures to manage recovery paths at object granularity. *Behavior groups* summarize object behavior at region granularity, and *object metadata* records the current recovery path and location of each remote object state. Together, they let later stages choose backup reuse, recomputation, Rep, or EC without making regions or stripes the unit of object access.

4.2.1 Behavior Groups and Profiles. Starfish collects behavior signals at region granularity. Region-based organization is common in managed runtimes and allocators [17, 18, 31]; Starfish uses it as a profiling and placement substrate for FT. A *region* is a contiguous allocator-managed area used for allocation, deallocation, and profiling; by default, a region is 256KB. A remote region is placed on a memory node, and its compute-side descriptor records the memory node, remote base address, and pointer to recovery metadata. A *behavior group* contains regions whose objects show similar access, update, size, and lifetime behavior. Regions and groups are not units of access: objects are still fetched, evicted, protected, invalidated, and reclaimed independently.

A group profile tracks the signals needed by later policies: fetch and refetch behavior, clean/dirty eviction ratio, update intensity, object-size bucket, and protected-state lifetime, i.e., the time before a protected version is invalidated, overwritten, or reclaimed. Prior far memory runtimes also use program or object behavior to guide placement and migration [43, 61, 92, 99]; Starfish uses similar signals to guide recovery-path reuse and protection layout. These signals tell Reusing Valid Recovery Paths whether a fetched state is likely to be evicted cleanly, and tell Behavior-Guided Protection whether a state is transient, stable, small enough to pack, or large enough to split. Aggregating them at region and group granularity avoids storing excessive metadata in every object reference.

Starfish initially assigns regions to groups using allocator-visible keys, such as allocation site, object type, and size class. It updates group profiles over profiling windows, where each window is a batch of object lifecycle events such as fetches, evictions, dirty evictions, frees, and protection invalidations.

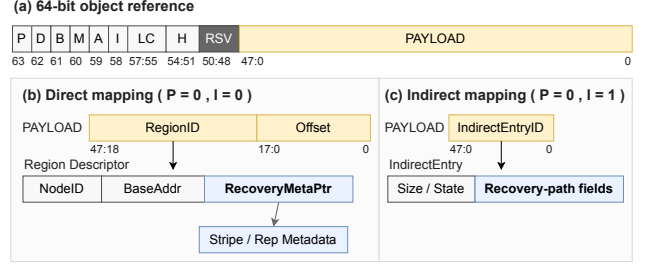


Figure 7. Starfish object metadata. (a) A 64-bit reference packs state bits. For remote objects, the payload either (b) stores a region ID and offset directly or (c) points to an indirect entry holding recovery-path fields that do not fit in the reference.

The fast path updates only lightweight counters using the event type and object metadata bits. At window boundaries, Starfish refines a group only when one policy would make conflicting decisions for different regions; future allocations or protection requests use a more specific predicate, without migrating existing objects solely for profiling.

4.2.2 Object State and Recovery Metadata. Starfish stores the metadata needed by common object operations in a compact 8-byte object reference, and uses auxiliary metadata only when a recovery path needs more information than the reference can encode. Each object has a runtime-managed reference. Figure 7(a) shows the reference layout. The first fields are common in far memory runtimes: *P* marks whether the object is local, *D* marks a dirty local copy, *M* marks relocation or layout transition, *A* is a sampled access bit, and *H* stores a small hotness hint. Starfish adds *B* to mark a valid retained remote backup, *I* to indicate that the payload points to an indirect entry, and *LC* to record the recovery-path class: no protection, backup, recompute, Rep, or EC. *RSV* contains reserved bits. If the object is local and does not need indirection, the reference payload directly stores the local address. Runtime dereference operations may return multiple local C/C++ pointers to the same object (see §4.3), but each object has a single runtime-managed reference that holds its state bits. When the application code requests mutable access, the runtime sets the *D* bit in that object’s reference. The returned local C/C++ pointers do not maintain separate dirty bits.

For remote objects, the reference payload either stores a remote region ID plus an offset, or stores an indirect-entry ID. The direct case locates the remote address through the region descriptor and follows its pointer to recovery metadata when Rep/EC metadata is needed. The indirect case records extra recovery information that does not fit in the reference, such as multiple replica locations, a retained backup plus a local copy, or a stripe ID, offset, length, and fragment list for EC.

An EC stripe contains data fragments and parity fragments. When EC packs small remote object states, a data fragment is divided into fixed-size *slots*; each live slot stores one remote object state or one fragment of a split state. Large remote object

states can occupy multiple fragments and use an indirect entry to record the fragment list. Fetch, eviction, mutable access, free, and recovery update object metadata, but they do not make a spanset or stripe the unit of access. Replicas and EC stripes remain protection and recovery units; remote objects remain independently fetched, invalidated, and reclaimed.

4.3 Reusing Valid Recovery Paths

Before creating new protected state, Starfish checks whether the remote object state already has a valid recovery path. A retained remote backup can recover a fetched object that is evicted cleanly, and a recompute recipe can regenerate derived state from protected inputs. When neither path remains valid, the state becomes a protection request for §4.4.

Programming model. Starfish builds on the programming model of existing runtime-managed far memory systems [43, 61, 83, 92]. Applications access remote objects through runtime APIs. `deref` returns read-only access, while `deref_mut` returns writable access and marks the object dirty. A dereference scope bounds returned local pointers. Starfish uses these existing events to maintain recovery paths for fault tolerance.

4.3.1 Selective Remote Backup. After fetching a remote object, Starfish may keep its existing protected remote version instead of immediately invalidating it. This retained version is a *remote backup*. A backup is useful only while it still represents the current remote object state, so Starfish records backup validity in object metadata.

The backup remains valid as long as the fetched object is accessed only through `deref`. When an object is evicted while still clean, Starfish reuses the backup by discarding the local copy and restoring the object mapping to the retained remote version. This avoids a new Rep/EC write. For EC-protected objects, it also keeps the old stripe slot valid, so a clean fetch–eviction cycle does not change the stripe organization.

`deref_mut` invalidates the backup because the local copy may be changed. Starfish clears the backup state and releases the retained version according to its old protection layout: replicas can be freed directly, while EC slots are marked invalid and reclaimed through stripe management. If the dirty object is later evicted, it must either use another valid recovery path or enter the new-protection path.

This reuse is hard to exploit in span-granular systems: a span becomes dirty once any packed object is modified, so span-granular cleanliness tracking loses most reuse opportunities, especially for small objects.

Budgeted selection. Starfish retains backups under a configurable remote memory budget, set to 10% of total memory by default. The policy is group-level: a group receives backup budget when its profile predicts high saved protection work per retained byte. Starfish ranks groups by

$$S_g = \frac{P_{\text{refetch}}(g) \cdot P_{\text{clean}}(g) \cdot C_{\text{avoid}}(g)}{B_g},$$

```
// Application computation code.
void recompute_a(DerefScope& s, Obj a, Obj b, Obj
c) {
    MUT_PTR* out = a.deref_mut(s); // output
    IMMUT_PTR* in_b = b.deref(s);   // input
    IMMUT_PTR* in_c = c.deref(s);   // input
    *out = compute(*in_b, *in_c);
}

// Actual application call.
DerefScope s;
recompute_a(s, a, b, c);
// Extra recovery declaration.
// mark_recomputable(output, inputs, entry,
// entry_args)
s.mark_recomputable(a, {b, c}, recompute_a, {a, b,
c});
```

Figure 8. Marking a recomputable remote object state. The application reuses the computation function as the recompute entry and adds one call to declare the output, inputs, and recovery arguments.

where P_{refetch} estimates whether a backup will be reused, P_{clean} estimates whether the fetched object remains clean until eviction, C_{avoid} estimates the avoided Rep/EC write, parity update, and message cost, and B_g is the retained backup bytes. When the budget is tight, Starfish reclaims backup budget from groups with lower benefit per byte; per-object history, such as recent clean or dirty evictions, only refines decisions within groups that receive backup budget. Thus, Starfish captures useful clean eviction reuse without tracking every local access or making every fetch run a heavy policy.

4.3.2 Recomputation Recipes. For derived remote object states, Starfish can use a *recompute recipe* as the recovery path. A recipe records the output, protected inputs, and recompute entry point. Starfish stores this recipe instead of continuously protecting the output with Rep/EC, trading normal-execution protection work for recovery-time computation.

Insight. Starfish treats a `DerefScope` as a recomputation boundary. Runtime-managed far memory already groups program code blocks that include remote memory access into scopes, and distinguishes read-only and mutable dereferences. Starfish uses this existing boundary: read-only dereferences identify recipe inputs, mutable dereferences identify recipe outputs, and later mutable dereferences provide the events needed to invalidate stale recipes. Thus, Starfish maintains recomputation correctness without separate dependency tracking.

Programming interface. Figure 8 shows the required code modification: the only extra line is `mark_recomputable`, whose first two fields describe the recovery dependency and whose last two fields describe the recovery call; the computation function and its call remain normal application code. An object can appear twice: in Figure 8, `a` is the output state

and also the destination argument of `recompute_a`, while `b` and `c` are both protected inputs and function arguments. The application reuses the normal computation body and does not manage recipe metadata or invalidation. The same interface could also let applications mark *discardable* objects that need no recovery path after eviction; we leave this extension as future work.

Runtime support. When `mark_recomputable` is called, Starfish creates a recipe that maps the output to the declared inputs and `recompute` entry. It also indexes recipes by their declared inputs. This index covers only explicit recipes; Starfish does not infer dependencies from arbitrary memory accesses. The recipe becomes the output’s recovery path only after the input states have valid recovery paths. If the output is evicted while the recipe is valid, Starfish does not protect the output contents through Rep/EC; after a memory node failure, Starfish restores the protected inputs and reruns the `recompute` entry. Starfish uses reference counting [12, 103] to manage lifetimes for inputs of recipes. Each recipe holds references to its declared inputs until it is invalidated. Thus, a `free(input)` call from application code will not reclaim the input while an output still relies on it for recovery.

Correctness. Correctness requires a recipe to regenerate the current output state. A later mutable access to the output invalidates the recipe and turns the output into a normal dirty state; its next eviction uses Rep/EC unless another recovery path applies. Input modification must also preserve the recovery-path invariant. If an input write would invalidate a recipe whose output has already been evicted with only that recipe as its recovery path, Starfish first regenerates the output and protects it through Rep/EC. Only then does Starfish invalidate the recipe and allow the input write to proceed. Thus, invalidation never removes the last recovery path of a remote object state.

Scope. Starfish only supports deterministic recipes whose inputs, output, and recovery entry are explicitly declared. Recipes target derived states with stable inputs; for example, WC’s map-phase hashmaps derive from read-only input text, so input writes that force output regeneration are rare in practice. This is a limitation: Starfish does not infer arbitrary dependencies, side effects, or nondeterministic executions, so such states fall back to Rep/EC. Techniques from lineage and checkpointing systems could reduce this manual burden by inferring dependency graphs or materializing selected intermediate outputs to bound long recipe chains [42, 57, 69, 102, 118]; applying them to Starfish is future work.

4.4 Behavior-Guided Protection

When a remote object state cannot reuse a valid recovery path, Starfish creates a protection request. The request carries object metadata and the behavior group profile, which guide the protection method, EC granularity, and EC layout. Algorithm 1 summarizes the policy. Starfish uses EC by default for low space overhead. It uses Rep only for groups likely to invalidate protected state soon, such as those with a high dirty eviction

Algorithm 1 Starfish behavior-guided protection

Require: protection request for object o , group profile p , budgets b

Ensure: new protected state

```

1: if TRANSIENT( $p$ ) and REPBUDGETSLACK( $b$ ) then
2:   protect  $o$  with bounded Rep
3:   return
4: if FREQUENTLIFECYCLE( $p$ ) then
5:   use fine EC slots for object/fragment operations
6: else if STABLECOLD( $p$ ) then
7:   use coarser packed EC to amortize coding and metadata
8: else
9:   use size-class packed EC
10: if LARGEOBJECT( $o$ ) and LOWFANOUTSENSITIVITY( $p$ ) then
11:   split  $o$  into fragments
12: construct EC through batched region scan when batching
    is profitable

```

rate or a short protected-state lifetime, and only when remote memory, network bandwidth, and message rate budgets have slack. Rep is a temporary choice for selected transient groups, not a system-wide replication mode. Background threads later consolidate Rep-protected states into EC when the group stabilizes or the memory budget becomes tight.

The predicates in Algorithm 1 are computed from the group profile. Replication avoids EC computation but consumes more remote memory and network bandwidth (§2.2). Starfish uses Rep to protect new states only when at least 50% of both budgets remain available (REPBUDGETSLACK); otherwise, it uses EC. TRANSIENT means a high dirty eviction rate or a short protected-state lifetime; FREQUENTLIFECYCLE means frequent fetch, dirty eviction, or free events.

Fine-grained EC reduces read/write amplification and improves access performance, but requires more EC computation and more work to maintain metadata. Our measurements of existing systems show that EC computation typically consumes less than 15% of the application’s CPU allocation and causes little interference with application execution. We therefore conservatively limit fine-grained EC computation to 10% of the application’s CPU allocation. If this budget is exceeded, Starfish falls back to coarse-grained EC for objects in larger size classes. STABLECOLD means a long protected-state lifetime and a low access/update rate; and LOWFANOUTSENSITIVITY means the profile can tolerate fetching multiple fragments for one object.

Batched EC construction. Starfish improves EC construction efficiency by exploiting group-level object organization. Regions in the same behavior group often contain many objects that are likely to be evicted together. Instead of copying these objects into a separate encoding buffer, Starfish scans the region and encodes contiguously, improving cache locality and encoding throughput. The scan range is an encoding choice, not an application access unit.

EC granularity. Starfish does not force all remote object states into one fixed EC granularity. Coarser granularity amortizes coding and metadata cost, so it fits cold or stable states even when individual objects are small. Finer granularity preserves object-level fetch, dirty eviction, and free operations, so it fits frequently accessed or frequently updated states even when they could be batched. Packed slots are protection layout units, not fetch units: a fetch resolves the requested object or fragment from object metadata rather than fetching neighboring objects in the same stripe. Object size still constrains the layout: size classes keep packed units regular, while very large states may be split into fragments when the group profile indicates that the extra fetch fan-out is acceptable.

Partial EC maintenance. A mutable dereference only marks the local object dirty and invalidates any retained backup; it does not update remote EC parity. Remote EC changes happen when a dirty eviction creates a new protected version, or when Starfish overwrites or reclaims an existing EC slot. In these cases, Starfish reads the old slot when needed, computes the data delta, writes the new slot, and applies the corresponding parity deltas. A free or invalidation marks the old slot dead in remote region metadata. The dead slot is unavailable to the application, but its bytes remain in the stripe until Starfish reclaims it, so parity remains valid for other live slots. Starfish reclaims dead slots when a region needs free slots or its dead-slot ratio crosses a configured threshold; it uses partial-stripe updates for small changes and full-stripe writes when rewriting the stripe is cheaper, following the standard EC tradeoff between small partial updates and batched rewrites [22, 59, 88, 120]. Starfish applies batches of slot and parity-delta writes through two-phase commits (2PC), keeping every stripe decodable when failures hit ongoing updates. Thus, packed EC units still create dead slots as spans do, but Starfish reclaims them from the compute node at slot granularity, preserving object-level lifecycle operations without memory node compaction.

4.5 Fault Recovery

Recovery metadata. The compute node remains alive under our fault model, so Starfish keeps the metadata needed to interpret recovery paths on the compute side: object metadata, region descriptors, indirect entries, stripe metadata, and recompute recipes. Memory nodes store remote object contents, replicas, and EC data/parity fragments. Small EC-protected remote object states usually need no indirect entry: the reference stores the remote region and offset, and the region descriptor points to the stripe metadata. Starfish uses indirect entries only when the recovery path does not fit in the reference, such as Rep with multiple copies, split large objects, recompute recipes, or EC stripes with explicit per-object entries. A 16-byte indirect entry can hold two extra 48-bit remote locations for two-failure-tolerant Rep. For EC, stripe metadata is amortized across the stripe: an RS(4,2) stripe needs about 32 bytes with compact region and chunk IDs. We omit the exact bit layout due to paper space.

Recovery process. Like Hydra and Carbink, Starfish uses foreground recovery and background repair. Foreground recovery

reads the object reference; it either obtains the remote region and offset directly or follows an indirect entry. Background repair starts from failed memory nodes, enumerates compute-side descriptors for remote regions placed on those nodes, and follows pointers to recovery metadata for affected objects or fragments. Existing protected versions, including those kept across clean fetches, use their recorded Rep/EC path: Rep reads a surviving replica, and EC decodes the missing object or fragment. Recompute restores protected inputs and reruns the recompute entry. If foreground recovery modifies an object before background repair reaches it, Starfish keeps the newer state and discards stale repair output.

5 Limitations

Starfish’s approach has several limitations.

Expensive recovery for long recipe chains. Recipe inputs (§4.3.2) may themselves be outputs of other recipes. Multiple levels of such dependencies can form long recipe chains. Starfish currently requires programmers to choose recomputable outputs and consider the performance cost. The runtime follows these declarations, so long chains can make recovery expensive. Protecting selected intermediate outputs with Rep/EC could shorten these chains [42, 57, 69, 102, 118]; we leave this to future work.

Limited precision of group profiling. Compared to per-object profiling, group profiling (§4.2.1) reduces overhead but may mix objects with different behaviors. Finer-grained groups and more frequent refinement can improve protection decisions but incur higher overhead. We leave adaptive adjustment of group granularity and refinement frequency to future work.

Extra programming effort. Like AIFM [83] and Carbink [121], Starfish requires applications to identify remote objects and access them through runtime APIs. Starfish could integrate techniques from Mira [43] and TrackFM [92] to reduce this effort.

6 Evaluation

Our evaluation seeks to answer the following questions:

- **Overall performance:** How much does Starfish improve application performance over existing FT designs, and how close does it get to a non-FT runtime? (§6.2)
- **Fault-tolerance cost:** How effectively does Starfish avoid unnecessary protection work and reduce the cost of new protection? (§6.3)
- **Failure recovery:** Can Starfish achieve comparable failure recovery performance to existing designs? (§6.4)
- **Design analysis:** How do Starfish’s design choices behave under different configurations? (§6.5)

6.1 Experimental Setup

Compared Systems and Implementation. We compare Starfish with Hydra [55], Carbink [121], and Non-FT. We implement

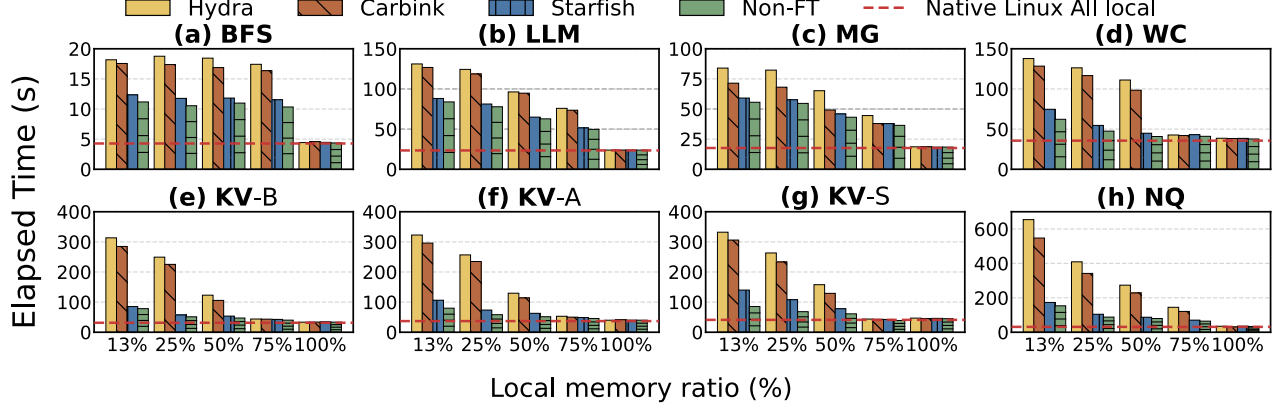


Figure 9. Performance comparison across applications. Lower is better.

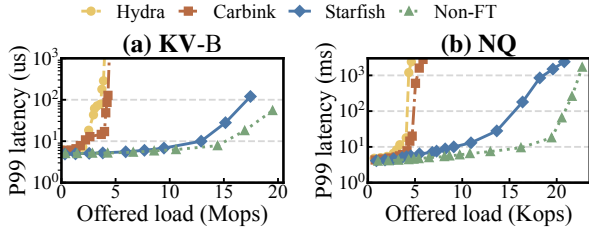


Figure 10. Tail latency versus offered load. Lower is better.

Hydra’s and Carbink’s FT designs in the same runtime framework, so all systems differ only in their FT strategy; the runtime eviction path is optimized to sustain NIC line rate.

Hydra-style EC (Hydra). Since the original Hydra implementation is tightly coupled with an old Linux kernel [89], we follow Carbink’s methodology [121] to implement a Hydra-like intra-page EC design, using 8KB pages as in Hydra [55]. *Carbink-style EC (Carbink).* Since Carbink is not publicly available, we implement it from the paper [121], including its span-based layout, out-of-place updates, and compaction policies. *Starfish.* This is our runtime with Reusing Valid Recovery Paths and Behavior-Guided Protection.

Non-FT. Non-FT is Starfish’s object-level runtime without remote memory protection, and represents Starfish’s performance upper bound. It differs from Carbink-w/oFT in § 3.1, which keeps Carbink’s span-based memory management to isolate existing FT overhead. We do not include full n-way Rep as a default baseline because two-failure tolerance would require three full remote copies and substantially more eviction bandwidth; instead, our default uses EC plus the bounded budget for retained remote backups. §6.5 separately evaluates Starfish’s optional Rep path when extra remote memory is available. We use Intel ISA-L v2.30.0 [50] for erasure-code computation and implement about 41K lines of C++.

Hardware Setup. We conduct experiments on a nine-node cluster. Each node has two Intel Xeon Gold 6342 CPUs (24 cores per CPU), 256 GB DDR4 memory, and one Mellanox ConnectX-5 100 Gbps InfiniBand NIC [72]; all nodes are connected through a 100 Gbps InfiniBand switch. We use MLNX_OFED [73] and InfiniBand verbs [49] for inter-node communication. By default, each application uses 24 cores and a 25% local-memory ratio. We use one node as the compute node, four as data memory nodes, and two as parity memory nodes, forming a (4, 2) erasure-code configuration, following the default setting in Carbink [121] and Hydra’s source code [90]. The remaining two nodes serve as standby failover nodes for recovery experiments.

Workloads. We evaluate Starfish using 6 applications and 8 workloads with representative features. The batch-processing applications include GAPBS BFS (BFS) [15], llama.cpp (LLM) [38], NPB MG (MG) [1], and WordCount (WC) [91]. The latency-sensitive applications include Key-Value Store (KV) [45] and Nhop (NQ) [40]. Our supplementary material [109] validates this choice with clean-eviction data and broader fine-grained-access measurements.

BFS is a graph benchmark from GAPBS. It traverses a graph using direction-optimized breadth-first search [16]. This algorithm has two phases: one irregularly traverses vertices through adjacency lists, while the other sequentially scans the adjacency arrays to traverse the remaining vertices. Therefore, BFS exhibits mixed random and sequential memory-access behavior [16]. MG is a scientific-computing application from the NAS Parallel Benchmarks [13]. It solves a 3D grid problem using a multigrid algorithm, traverses multiple large grid arrays, and also exhibits a mixed memory-access pattern [28]. LLM is a large language model inference application. It repeatedly streams large model weight vectors during inference, leading to a mostly sequential access pattern [60]. WC [30] is an analytic application that counts word frequencies in input text. All these applications are bandwidth-hungry and can

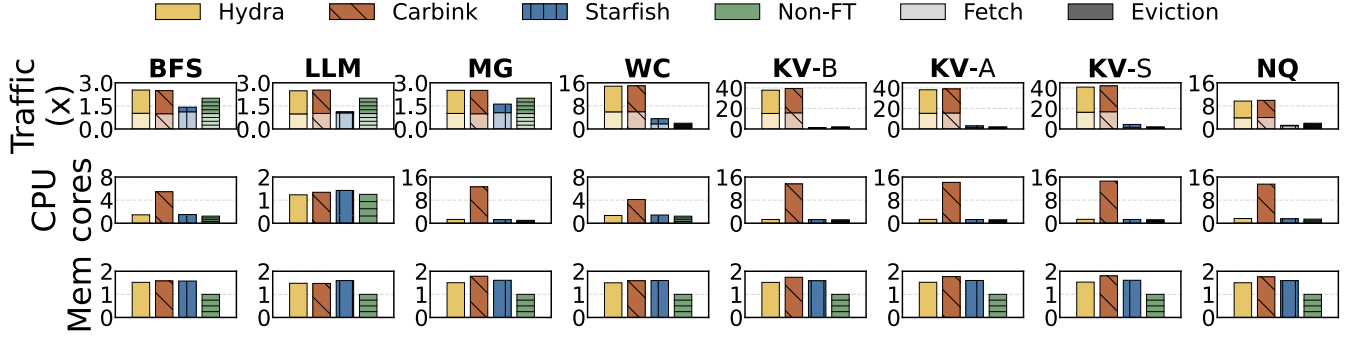


Figure 11. Fault-tolerance tax across applications. Lower is better.

Application	Dataset / Configuration
Batch-processing applications	
GAPBS BFS (BFS [15])	Friendster social graph [2, 115], 26.91 GiB
llama.cpp (LLM [38])	LLaMA 2 7B Chat [93] FP32, 27 GiB
NPB MG (MG [13])	NPB Class D [1, 13], 24.14 GiB
WordCount (WC [91])	English Wikipedia [108], 24.62 GiB
Latency-sensitive applications	
Key-Value Store (KV [45])	
KV-B: YCSB Workload B [27]	95% read, 5% update, 1000M operations
KV-A: YCSB Workload A [27]	50% read, 50% update, 1000M operations
KV-S: Synthetic	95% update, 5% read, 1000M operations
Nhop Query (NQ [40])	Friendster social graph [2, 115], 26.91 GiB, 2M queries

Table 1. Applications and datasets.

saturate the NIC bandwidth, so their overall performance is sensitive to extra network traffic induced by FT.

KV is a common building block for large-scale online services such as caching [19]. It organizes 512B key-value records in hash tables, and serves get and update operations under different request mixes. We use YCSB [27] as the workload generator and set the Zipfian skewness to 0.99 to model a common skewed request distribution [3]. We denote YCSB Workload B as **KV-B**, YCSB Workload A as **KV-A**, and a high-update synthetic workload as **KV-S**. NQ [40] is a graph-query service for social-network workloads [35]. It stores each vertex’s neighbor list as an adjacency list and issues 2-hop neighborhood queries. These queries repeatedly follow adjacency lists across multiple hops, leading to highly random accesses. Together, KV and NQ stress fine-grained object accesses and are sensitive to access amplification.

6.2 Overall Performance

In this section, all systems run under normal fault-free execution, but fault tolerance remains enabled for all FT systems.

6.2.1 Throughput. We evaluate overall performance using the execution time of the same task; lower execution time therefore represents higher throughput. Figure 9 shows that Starfish consistently outperforms existing FT designs under 13%–75% local memory ratios, where remote accesses are frequent. On average, Starfish improves throughput by 1.9× over Carbink and 2.1× over Hydra, while reaching 88.3%

of Non-FT’s throughput. The gain is largest under stronger memory pressure: at 13% and 25% local memory, Starfish improves throughput by 2.2× and 2.4× over Carbink, and by 2.4× and 2.6× over Hydra, respectively.

Recovery-path reuse and recomputation. Starfish benefits most when remote object states can avoid new protection. BFS traverses stable graph arrays and LLM streams mostly stable model weights; their fetched states are frequently evicted but remain clean. Starfish exploits this property through Selective Remote Backup, improving BFS and LLM throughput by 1.4×/1.4× over Carbink and 1.5×/1.5× over Hydra, while reducing eviction traffic by 13.4× over Carbink and 13.2× over Hydra on average. MG has less room for this optimization because 55.8% of its evictions are dirty, but Starfish still improves throughput by 1.1× over Carbink and 1.4× over Hydra. WC shows the benefit of recomputation: its map-phase hashmaps are temporary and rebuildable from the input text, so marking them recomputable improves throughput by 1.8×/1.9× over Carbink/Hydra on average.

Behavior-guided protection. Starfish also improves fine-grained workloads by preserving object-granular access in the FT path. Across KV and NQ, Starfish improves throughput by 2.3× over Carbink and 2.6× over Hydra, while reaching 61.0%–94.2% of Non-FT’s throughput. KV exposes the effect of update intensity: Starfish improves KV-B, KV-A, and KV-S by 2.6×/2.8×, 2.2×/2.4×, and 1.8×/2.0× over Carbink/Hydra, respectively. NQ continues to benefit even at 75% local memory because its graph footprint remains large. Through Behavior-Guided Protection, Starfish avoids span-granular FT updates on the request path and reduces the total network traffic of these workloads by 14.5× over Carbink and 14.0× over Hydra; after optimization, the three KV workloads and NQ consume only 2.8 GB/s and 1.9 GB/s of network traffic on average, respectively. All throughput results above use 13%–75% local memory. At 100% local memory, all systems use the same local-memory access path, and Starfish’s runtime overhead over native Linux is 5.9%.

6.2.2 Latency. We evaluate P99 latency for KV-B and NQ by varying the offered load. Figure 10 shows that Starfish

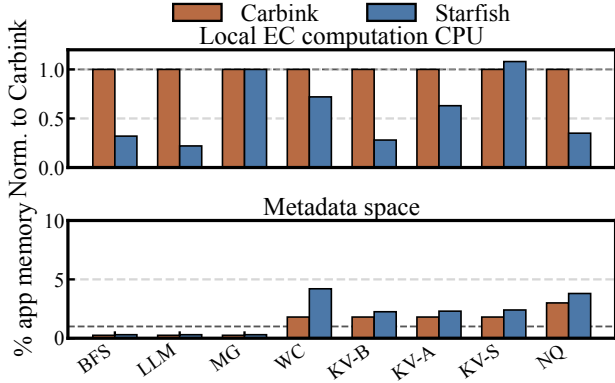


Figure 12. Starfish compute-node overhead compared with Carbink. Lower is better.

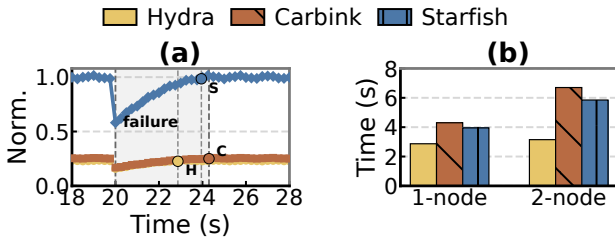


Figure 13. Foreground throughput (a) and background recovery time (b) under failures.

delays the latency cliff and stays close to Non-FT over a wider load range. For KV-B, at 4.0 Mops, Starfish keeps P99 latency at 5.3 us, while Carbink reaches 23.4 us and Hydra reaches 1.7 ms. For NQ, Starfish reaches 20.8 Kops, 3.3× higher than Carbink’s 6.2 Kops and 4.3× higher than Hydra’s 4.9 Kops; at 13.7 Kops, Starfish still keeps P99 at 28.0 ms. This is because Carbink and Hydra amplify fine-grained requests into larger network transfers and hit the bandwidth limit, increasing queuing delay.

6.3 Fault Tolerance Costs

In this section, we quantify how much Starfish reduces the cost of fault tolerance and whether its own design mechanisms introduce noticeable compute node side cost.

Network traffic. Figure 11 shows that Starfish reduces total traffic by 8.5× over Carbink and 8.2× over Hydra on average. For BFS and LLM, Selective Remote Backup avoids re-protecting clean evictions and reduces eviction traffic by 13.4× over Carbink and 13.2× over Hydra; in contrast, Hydra and Carbink write both data and parity on each eviction, making eviction traffic about 1.5× of fetch traffic. For KV and NQ, Behavior-Guided Protection keeps accesses object-granular and avoids span-level access amplification. NQ exposes this

granularity effect: each vertex’s adjacency list is a variable-size remote object state, with median size 80B, p90 1.25KB, and 99.5% of objects below an 8KB span. The runtime chooses protection granularities automatically; in our run, the smallest EC slot is 256B, and NQ fetches about 0.65KB per remote access for 0.49KB of useful adjacency-list data, so roughly 74% of fetched bytes are useful. Starfish reduces fetch traffic for KV by 12.3×/12.0× over Carbink/Hydra, and for NQ by 3.5×/3.4×.

Memory node resources. Starfish also reduces memory node resource usage. Across applications, Starfish lowers memory node CPU usage by 7.5× and remote memory usage by 4.9% over Carbink on average; on KV, Carbink consumes up to 14.6 cores and 1.8× remote memory. Starfish’s usage already includes its retained backup budget and FT metadata, while Carbink loses more space to dead spans awaiting compaction. CPU overhead comes from Carbink’s compaction: irregular updates scatter invalid spans across spansets, requiring memory node CPU work. Starfish instead handles object invalidation and reclamation through object-granular EC updates or full writes from the compute node. Compared with Hydra, Starfish uses 5.9% more remote memory on average because Selective Remote Backup retains additional remote backups, but this overhead stays within 3.8%–7.9% across workloads.

Finally, Figure 12 shows that Starfish adds little compute node cost. Starfish uses 42.5% less local EC-computation CPU than Carbink on average because clean evictions avoid most re-encoding work. Its metadata, including runtime object references and auxiliary FT/runtime metadata, uses 2.0% of application memory on average, slightly above Carbink’s 1.4% due to region metadata, backup/recompute indirection, and fine-grained FT mappings.

6.4 Failure Recovery

We evaluate recovery by running KV to steady state and injecting one data memory node failure at 20 s under the default (4, 2) EC configuration. The failed node is removed and reconstructed on a standby node while the application continues running. During recovery, all systems reserve 80% of network bandwidth for foreground service and 20% for background reconstruction. Figure 13(a) plots foreground throughput, normalized to Starfish’s pre-failure throughput. Starfish keeps the highest throughput throughout recovery; at the failure point, it drops by 41.5% from its own pre-failure throughput, while Hydra and Carbink drop by 32.0% and 34.0% from their lower baselines.

Figure 13(b) shows background recovery time. Repair competes with foreground degraded reads for bandwidth. Hydra keeps degraded reads close to the normal path, so repair progresses fastest. Carbink reconstructs span-granular data for foreground reads, amplifying foreground traffic and leaving less bandwidth for repair. Starfish recovers in 3.9 s for one-node failure and 5.9 s for two-node failure, between Hydra and Carbink, because object-granular degraded reads avoid most of this amplification.

6.5 Design Analysis

Within-group differences and refinement intervals. Group profiling reduces overhead but may lead to less accurate protection decisions than per-object profiling. Frequent refinement can also increase profiling overhead. Using MG and KV-A, we measure within-group differences and refinement intervals. We choose them instead of LLM and BFS, where elements in the same large array have similar access patterns.

We add per-object access and eviction counters to measure access frequency and dirty-eviction ratios (§4.3 and §4.4). For access share, we select the more frequently accessed 50% of objects in each group and measure their share of group accesses. For dirty-ratio deviation, we measure the average absolute difference between each object’s dirty-eviction ratio and its group mean. Higher values indicate greater within-group differences. Both metrics are weighted averages across groups. The access shares are 65.3% for MG and 59.7% for KV-A. The dirty-ratio deviations are 7.2% and 13.3%, respectively. These results show that relatively small differences exist within groups. We argue that more frequent refinement can reduce these differences.

The mean interval between consecutive runtime refinement decisions is 643.5 ms for MG and 212.8 ms for KV-A. KV-A refines more frequently because its key-value pairs can have different access and update patterns. In MG, neighboring grid elements have similar access patterns across iterations. These intervals are on the order of hundreds of milliseconds, suggesting that refinement is relatively infrequent.

Backup budget. We vary the remote backup budget from 5% to 50% with 50% local memory, using MG and LLM as examples. Performance is not very sensitive to this budget: MG is 14.0% slower than its best time at 5% and 11.0% slower at 10%, while LLM stays within 2.0% across the range. The policy prioritizes backups that save more protection work per byte, so a small budget retains the most useful states. We therefore use a 10% default budget.

Hybridizing Rep. We next give applications an extra 50% remote memory so the runtime can choose Rep for selected states instead of always using EC. This tests the optional Rep branch in §4.4. With the default 25% local memory, this improves one-node recovery by 1.1×, from 3.9 seconds to 3.6 seconds, under the same failure as in §6.4.

Scalability. We use BFS to test different memory node counts and $RS(n, m)$ configurations. Starfish achieves the largest gain under the default $RS(4, 2)$ configuration; under $RS(4, 1)$ and $RS(6, 2)$, the gain is smaller but still reaches 1.2× in both cases. Larger n/m reduces the relative amount of parity writes, leaving less FT-maintenance overhead for Starfish to eliminate.

7 Related Work

Fault tolerance for disaggregated memory. Several systems target memory node failures in disaggregated settings [32,

47, 55, 71, 121]. FaRM [32] and SWARM [71] replicate distributed objects, Aceso [47] combines EC, Rep, and checkpointing for disaggregated key-value stores, and Hydra and Carbink use EC for general remote memory. These systems either depend on specific storage/transaction interfaces or organize protection around coarse memory units. Starfish instead uses runtime object semantics to reuse existing recovery paths and choose lower-cost protection for general far memory. Other systems address different fault models, including client-side metadata corruption, compute server failures, and storage durability [65, 75, 85].

Avoiding and reducing protection work. Prior FT systems reduce overhead from different directions. Lineage-based analytics systems avoid protecting some intermediate data by rebuilding it from recorded dependencies [42, 57, 102, 118]. Incremental and fine-grained checkpointing systems reduce checkpoint work by tracking changed state, selecting the state to materialize, or adapting checkpoint frequency [51, 69, 95]. Erasure-coded storage systems assume new protected data must be maintained, and reduce the update cost with parity logging, partial writes, delta updates, and update-aware stripe organization [22, 59, 87, 88, 120]. Starfish differs by applying an object-first design to far memory and combining both goals: runtime object semantics identify remote object states that can avoid new protection by reusing a retained backup or recompute recipe, and object behavior reduces the cost of protecting the remaining states.

Tiered memory systems. Tiered memory systems [4, 33, 53, 54, 63, 67, 80, 82, 97, 110, 113, 114, 119] extend application memory with slower tiers, such as remote NUMA and CXL-attached memory. They optimize placement and migration, not fault tolerance. A clean demotion with a valid slow-tier copy resembles Selective Remote Backup, and Nomad [110] uses shadow pages to reduce migration writes. Starfish applies this reuse principle to fault tolerance.

8 Conclusion

This paper presents Starfish, an object-first runtime that first reuses valid recovery paths and then reduces the cost of new protection. Across diverse applications, Starfish improves performance, reduces network traffic and memory node resource usage, and preserves efficient recovery.

Acknowledgments

We thank the reviewers for their comments and are particularly grateful to our shepherd for his feedback. The corresponding author is Chenxi Wang. This work was supported by the National Key Research and Development Program of China (No. 2024YFE0204100), the Pioneer Initiative Talent Program B of the Chinese Academy of Sciences (Nos. E545030 and E401430), and the National Natural Science Foundation of China under Grant Nos. 62302479, U23B2020, and 625B2002.

References

- [1] [n. d.]. Problem Sizes and Parameters in NAS Parallel Benchmarks. https://www.nas.nasa.gov/software/npb_problem_sizes.html. Accessed: 2026-05-24.
- [2] [n. d.]. SNAP: Friendster Social Network and Ground-Truth Communities. <http://snap.stanford.edu/data/com-Friendster.html>. Accessed: 2026-05-24.
- [3] [n. d.]. YCSB ZipfianGenerator.java. <https://github.com/brianfrankcooper/YCSB/blob/master/core/src/main/java/site/ycsb/generator/ZipfianGenerator.java>. Default Zipfian constant is 0.99; accessed: 2026-06-01.
- [4] Neha Agarwal and Thomas F Wenisch. 2017. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 631–644.
- [5] Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J Marathe, Athanasios Xygkis, and Igor Zablotchi. 2020. Microsecond consensus for microsecond applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 599–616.
- [6] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. 2000. Failure detection and consensus in the crash-recovery model. *Distributed computing* 13, 2 (2000), 99–125.
- [7] Marcos Kawazoe Aguilera, Ramaprabhu Janakiraman, and Lihao Xu. 2005. Using erasure codes efficiently for storage in a distributed system. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*. IEEE, 336–345.
- [8] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 843–857.
- [9] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput?. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.
- [10] Sanjith Athlur, Timothy Kim, Saurabh Kadekodi, Francisco Maturana, Xavier Ramos, Arif Merchant, KV Rashmi, and Gregory R Ganger. 2025. Okapi: Decoupling Data Striping and Redundancy Grouping in Cluster File Systems. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 897–914.
- [11] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)* 42, 1 (1995), 124–142.
- [12] David F Bacon, Perry Cheng, and VT Rajan. 2004. A unified theory of garbage collection. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 50–68.
- [13] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications* 5, 3 (1991), 63–73. doi:10.1177/109434209100500306
- [14] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2018. *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*. Number 46 in Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers. doi:10.2200/S00874ED3V01Y201809CAC046
- [15] Scott Beamer, Krste Asanović, David Patterson, and contributors. 2015. GAP Benchmark Suite. <https://github.com/sbeamer/gapbs>. Reference implementation. Accessed: 2026-05-20.
- [16] Scott Beamer, Krste Asanovic, and David A. Patterson. 2012. Direction-Optimizing Breadth-First Search. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, Article 12, 10 pages. doi:10.1109/SC.2012.50
- [17] 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 Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*. Association for Computing Machinery, New York, NY, USA, 117–128. doi:10.1145/378995.379232
- [18] Stephen M. Blackburn and Kathryn S. McKinley. 2008. Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 22–32. doi:10.1145/1375581.1375586
- [19] Alex D. Breslow, Dong Ping Zhang, Joseph L. Greathouse, Nuwan Jayasena, and Dean M. Tullsen. 2016. Horton Tables: Fast Hash Tables for In-Memory Data-Intensive Computing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 281–294. <https://www.usenix.org/conference/atc16/technical-sessions/presentation/breslow>
- [20] Irina Calciu, M Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 79–92.
- [21] Irina Calciu, Ivan Puddu, Aasheesh Kolli, Andreas Nowatzky, Jayneel Gandhi, Onur Mutlu, and Pratap Subrahmanyam. 2019. Project pberry: Fpga acceleration for remote memory. In *Proceedings of the workshop on hot topics in operating systems*. 127–135.
- [22] Jeremy CW Chan, Qian Ding, Patrick PC Lee, and Helen HW Chan. 2014. Parity Logging with Reserved Space: Towards Efficient Updates and Recovery in Erasure-coded Clustered Storage. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*. 163–176.
- [23] Haibo Chen, Heng Zhang, Mingkai Dong, Zhaoguo Wang, Yubin Xia, Haibing Guan, and Binyu Zang. 2017. Efficient and available in-memory KV-store with hybrid erasure coding and replication. *ACM Transactions on Storage (TOS)* 13, 3 (2017), 1–30.
- [24] Lei Chen, Shi Liu, Chenxi Wang, Haoran Ma, Yifan Qiao, Zhe Wang, Chenggang Wu, Youyou Lu, Xiaobing Feng, Huimin Cui, et al. 2024. A Tale of Two Paths: Toward a Hybrid Data Plane for Efficient Far-Memory Applications. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 77–95.
- [25] Vijay Chidambaram, Thanumalayan Sankaranarayanan Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2013. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 228–243.
- [26] Compute Express Link Consortium. [n. d.]. About CXL. <https://computeexpresslink.org/about-cxl/>. Accessed: 2026-05-17.
- [27] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. 143–154. doi:10.1145/1807128.1807152
- [28] Eduardo Henrique Molina da Cruz, Marco Antonio Zanata Alves, Alexandre Carissimi, Philippe Olivier Alexandre Navaux, Christiane Pousa Ribeiro, and Jean-François M'ehaut. 2011. Using Memory Access Traces to Map Threads and Data on Hierarchical Multi-core Platforms. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*. IEEE, 551–558. doi:10.1109/IPDPS.2011.197
- [29] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. Netpaxos: Consensus at network speed.

- In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. Association for Computing Machinery, New York, NY, USA, 1–7.
- [30] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*. USENIX Association, 137–150.
 - [31] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-First Garbage Collection. In *Proceedings of the 4th International Symposium on Memory Management (ISMM '04)*. Association for Computing Machinery, New York, NY, USA, 37–48. doi:10.1145/1029873.1029879
 - [32] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Berkeley, CA, USA, 401–414.
 - [33] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, et al. 2023. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. Association for Computing Machinery, New York, NY, USA, 727–741.
 - [34] Partha Dutta, Rachid Guerraoui, Ron R Levy, and Arindam Chakraborty. 2004. How fast can a distributed atomic read be?. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*. 236–245.
 - [35] Qi Fan, Zhengkui Wang, Chee-Yong Chan, and Kian-Lee Tan. 2016. Towards Neighborhood Window Analytics over Large-Scale Graphs. In *Database Systems for Advanced Applications: 21st International Conference, DASFAA 2016, Dallas, TX, USA, April 16–19, 2016, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9643)*. Springer, 201–217. doi:10.1007/978-3-319-32049-6_13
 - [36] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. 2010. Availability in Globally Distributed Storage Systems. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association, Vancouver, BC.
 - [37] Jian Gao, Jiwei Shu, Bin Yan, Yuhao Zhang, and Keji Huang. 2025. Stripeless Data Placement for Erasure-Coded In-Memory Storage. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 821–838.
 - [38] Georgi Gerganov and contributors. 2023. llama.cpp: LLM inference in C/C++. <https://github.com/ggml-org/llama.cpp>. Accessed: 2026-05-20.
 - [39] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct access, High-Performance memory disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 287–294.
 - [40] graph-benchmark contributors. [n.d.]. Graph Benchmark: Graph benchmarks for Kuzu, Ladybug and lance-graph. <https://github.com/prrao87/graph-benchmark>. Includes n-hop path-finding queries q8 and q9. Accessed: 2026-05-20.
 - [41] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 649–667.
 - [42] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. 2010. Nectar: Automatic Management of Data and Computation in Datacenters. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association, Vancouver, BC.
 - [43] Zhiyuan Guo, Zijian He, and Yiyang Zhang. 2023. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 692–708.
 - [44] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiyang Zhang. 2022. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 417–433.
 - [45] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch Hashing. In *Proceedings of the 22nd International Symposium on Distributed Computing (DISC '08)*. Springer, 350–364. doi:10.1007/978-3-540-87779-0_24
 - [46] Chunchen Hu, Chenxi Wang, Sa Wang, Ninghui Sun, Yungang Bao, Jieru Zhao, Sanidhya Kashyap, Pengfei Zuo, Xusheng Chen, Liangliang Xu, et al. 2023. Skadi: Building a distributed runtime for data systems in disaggregated data centers. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. 94–102.
 - [47] Zhisheng Hu, Pengfei Zuo, Yizou Chen, Chao Wang, Junliang Hu, and Ming-Chang Yang. 2024. Aceso: Achieving efficient fault tolerance in memory-disaggregated key-value stores. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 127–143.
 - [48] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure coding in windows azure storage. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 15–26.
 - [49] InfiniBand Trade Association. [n.d.]. InfiniBand Architecture Specification. <https://www.infinibandta.org/ibta-specification/>. Accessed: 2026-05-17.
 - [50] Intel Corporation. 2020. Intel(R) Intelligent Storage Acceleration Library (ISA-L), v2.30.0. <https://github.com/intel/isa-l/releases/tag/v2.30.0>. Version 2.30.0; accessed: 2026-06-01.
 - [51] Tanzima Zerin Islam, Kathryn Mohror, Saurabh Bagchi, Adam Moody, Bronis R De Supinski, and Rudolf Eigenmann. 2012. MCREngine: A scalable checkpointing system using data-aware aggregation and compression. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, IEEE, Washington, DC, USA, 1–11.
 - [52] Osama Khan, Randal C Burns, James S Plank, William Pierce, and Cheng Huang. 2012. Rethinking erasure codes for cloud file systems: minimizing i/o for recovery and degraded reads. In *FAST*. 20.
 - [53] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, et al. 2019. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, USA, 317–330.
 - [54] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 17–34.
 - [55] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G Shin. 2022. Hydra: Resilient and Highly Available Remote Memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, 181–198.
 - [56] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 574–587.
 - [57] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2014. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *Proceedings of the ACM Symposium on*

- Cloud Computing (SoCC '14)*. Association for Computing Machinery, Seattle, WA, USA, Article 6, 15 pages. doi:10.1145/2670979.2670985
- [58] Haifeng Li, Ke Liu, Ting Liang, Zuojun Li, Tianyue Lu, Hui Yuan, Yinben Xia, Yungang Bao, Mingyu Chen, and Yizhou Shan. 2023. Hopp: Hardware-software co-designed page prefetching for disaggregated memory. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1168–1181.
- [59] Huiba Li, Yiming Zhang, Zhiming Zhang, Shengyun Liu, Dongsheng Li, Xiaohui Liu, and Yuxing Peng. 2017. PARIX: Speculative partial writes in Erasure-Coded systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 581–587.
- [60] Jindong Li, Tenglong Li, Guobin Shen, Dongcheng Zhao, Qian Zhang, and Yi Zeng. 2025. Pushing up to the Limit of Memory Bandwidth and Capacity Utilization for Efficient LLM Decoding on Embedded FPGA. In *2025 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1–7. doi:10.23919/DATE64628.2025.10993087
- [61] Quanxi Li, Hong Huang, Ying Liu, Yanwen Xia, Jie Zhang, Mosong Zhou, Xiaobing Feng, Huimin Cui, Quan Chen, Yizhou Shan, and Chenxi Wang. 2025. Beehive: A Scalable Disaggregated Memory Runtime Exploiting Asynchrony of Multithreaded Programs. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 167–187.
- [62] Shenglong Li, Quanlu Zhang, Zhi Yang, and Yafei Dai. 2017. BCStore: Bandwidth-efficient in-memory KV-store with batch coding. In *2017 IEEE 33rd Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, Washington, DC, USA, 1–12.
- [63] Jinshu Liu, Hamid Hadian, Hanchen Xu, and Huaicheng Li. 2025. Tiered memory management beyond hotness. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 731–747.
- [64] Xianting Lu, Yunong Wang, Yu Fu, Qi Sun, Xuhua Ma, Xudong Zheng, and Cheng Zhuo. 2024. MISP: A Multimodal-based Intelligent Server Failure Prediction Model for Cloud Computing Systems. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '24)*. Association for Computing Machinery, New York, NY, USA, 5509–5520. doi:10.1145/3637528.3671568
- [65] Xuhao Luo, Ramnathan Alagappan, and Aishwarya Ganesan. 2024. SplitFT: Fault tolerance for disaggregated datacenters via remote memory logging. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 590–607.
- [66] Haoran Ma, Shi Liu, Chenxi Wang, Yifan Qiao, Michael D Bond, Stephen M Blackburn, Miryung Kim, and Guoqing Harry Xu. 2022. Mako: A low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 92–107.
- [67] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 742–755.
- [68] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnappalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding Crash-Consistency bugs with bounded Black-Box crash testing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 33–50.
- [69] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. CheckFreq: Frequent, Fine-Grained DNN checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 203–216.
- [70] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. 2014. f4: Facebook's Warm BLOB Storage System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 383–398. <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/muralidhar>
- [71] Antoine Murat, Clément Burgelin, Athanasios Xygkis, Igor Zablatchi, Marcos Kawazoe Aguilera, and Rachid Guerraoui. 2024. Swarm: Replicating shared disaggregated-memory data in no time. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 24–45.
- [72] NVIDIA ConnectX-5 Documentation. [n.d.]. NVIDIA Mellanox ConnectX-5 Adapters. <https://www.nvidia.com/en-in/networking/ethernet/connectx-5/>. Accessed: 2026-05-25.
- [73] NVIDIA MLNX_OFED Documentation. [n.d.]. NVIDIA MLNX_OFED Documentation. <https://docs.nvidia.com/networking/display/mlnxofedv24100700>. Accessed: 2026-05-25.
- [74] Yueyang Pan, Yash Lala, Musa Unal, Yujie Ren, Seung-seob Lee, Abhishek Bhattacharjee, Anurag Khandelwal, and Sanidhya Kashyap. 2025. Scalable Far Memory: Balancing Faults and Evictions. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. 136–152.
- [75] Adarsh Patil, Vijay Nagarajan, Nikos Nikoleris, and Nicolai Oswald. 2023. Āpta: Fault-tolerant object-granular CXL disaggregated memory for accelerating FaaS. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 201–215.
- [76] Thanumalayan Sankaranarayanan Pillai, Ramnathan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Application crash consistency and performance with CCFS. *ACM Transactions on Storage (TOS)* 13, 3 (2017), 1–29.
- [77] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiyang Zhang, Miryung Kim, and Guoqing Harry Xu. 2023. Hermit: Low-Latency, High-Throughput, and Transparent Remote Memory via Feedback-Directed Asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 181–198.
- [78] K Vinayak Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. 2016. EC-Cache: Load-Balanced, Low-Latency cluster caching with online erasure coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 401–417.
- [79] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruva Borthakur, and Kannan Ramchandran. 2014. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. In *ACM SIGCOMM 2014 Conference*. ACM, 331–342. doi:10.1145/2619239.2626325
- [80] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 392–407.
- [81] Irving S Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics* 8, 2 (1960), 300–304.
- [82] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. 2024. MTM: Rethinking memory profiling and migration for multi-tiered large memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 803–817.
- [83] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. 2020. AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 315–332.
- [84] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiyang Zhang. 2018. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 69–87.

- [85] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R Lyu. 2023. FUSEE: A fully Memory-Disaggregated Key-Value store. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 81–98.
- [86] Zhirong Shen, Yuhui Cai, Keyun Cheng, Patrick PC Lee, Xiaolu Li, Yuchong Hu, and Jiwu Shu. 2025. A survey of the past, present, and future of erasure coding for storage systems. *ACM Transactions on Storage* 21, 1 (2025), 1–39.
- [87] Zhirong Shen, Patrick PC Lee, Jiwu Shu, and Wenzhong Guo. 2017. Correlation-aware stripe organization for efficient writes in erasure-coded storage systems. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 134–143.
- [88] Daniel Stodolsky, Garth Gibson, and Mark Holland. 1993. Parity logging overcoming the small write problem in redundant disk arrays. *ACM SIGARCH Computer Architecture News* 21, 2 (1993), 64–75.
- [89] SymbioticLab. 2022. Hydra: Resilient and Highly Available Remote Memory. <https://github.com/SymbioticLab/Hydra>. GitHub repository, accessed May 10, 2026.
- [90] SymbioticLab. 2022. Hydra source code: EC configuration in infiniswap.h. https://github.com/SymbioticLab/Hydra/blob/9918789f1faa5169307b1d4ae7a1c86303dc3563/resilience_manager/infiniswap.h#L125-L132. Defines NDATAS = 4 and NDISKS = NDATAS + 2, corresponding to a default (4,2) erasure-code configuration..
- [91] Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. 2011. Phoenix++: Modular MapReduce for Shared-Memory Systems. In *Proceedings of the Second International Workshop on MapReduce and Its Applications (MapReduce '11)*. Association for Computing Machinery, New York, NY, USA, 9–16. doi:10.1145/1996092.1996095
- [92] Brian R Tauro, Brian Suchy, Simone Campanoni, Peter Dinda, and Kyle C Hale. 2024. TrackFM: Far-out compiler support for a far memory world. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 401–419.
- [93] Hugo Touvron, Louis Martin, Kevin Stone, et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv preprint arXiv:2307.09288* (2023). doi:10.48550/arXiv.2307.09288
- [94] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. 2010. Characterizing Cloud Computing Hardware Reliability. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 193–204. doi:10.1145/1807128.1807161
- [95] Dirk Vogt, Cristiano Giuffrida, Herbert Bos, and Andrew S. Tanenbaum. 2015. Lightweight Memory Checkpointing. In *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2015)*. IEEE Computer Society, 474–484. doi:10.1109/DSN.2015.45
- [96] Marko Vukolić. 2015. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In *International workshop on open problems in network security*. Springer, 112–125.
- [97] Midhul Vuppapapati and Rachit Agarwal. 2024. Tiered memory management: Access latency is the key!. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 79–94.
- [98] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2020. Semeru: a Memory-Disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 261–280.
- [99] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. 2022. Mem-Liner: Lining up tracing and application for a Far-Memory-Friendly runtime. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 35–53.
- [100] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2023. Canvas: Isolated and adaptive swapping for Multi-Applications on remote memory. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 161–179.
- [101] Chen-Xi Wang, Yi-Zhou Shan, Peng-Fei Zuo, and Hui-Min Cui. 2023. Reinvent cloud software stacks for resource disaggregation. *Journal of Computer Science and Technology* 38, 5 (2023), 949–969.
- [102] Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. 2019. Lineage Stash: Fault Tolerance Off the Critical Path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, Huntsville, ON, Canada, 338–352. doi:10.1145/3341301.3359653
- [103] Stephanie Wang, Eric Liang, Edward Oakes, Ben Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. 2021. Ownership: A distributed futures system for Fine-Grained tasks. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 671–686.
- [104] Zixuan Wang, Jinyu Gu, Xingda Wei, and Yubin Xia. 2026. OneSidedMW: Managing Disaggregated Memory Efficiently, Flexibly, and Securely with RNIC Offloading. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. USENIX Association, Renton, WA, 2705–2720. <https://www.usenix.org/conference/nsdi26/presentation/wang-zixuan>
- [105] Zizhong Wang, Tongliang Li, Haixia Wang, Airan Shao, Yunren Bai, Shangming Cai, Zihan Xu, and Dongsheng Wang. 2020. CRaft: An Erasure-coding-supported Version of Raft for Reducing Storage Cost and Network Cost. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 297–308.
- [106] Zixuan Wang, Xingda Wei, Jinyu Gu, Hongrui Xie, Rong Chen, and Haibo Chen. 2025. ODRP: On-Demand Remote Paging with Programmable RDMA. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 1101–1115. <https://www.usenix.org/conference/nsdi25/presentation/wang-zixuan>
- [107] Matthias Wiesmann, Fernando Pedone, André Schiper, Bettina Kemme, and Gustavo Alonso. 2000. Database replication techniques: A three parameter classification. In *Proceedings 19th IEEE Symposium on Reliable Distributed Systems SRDS-2000*. IEEE, 206–215.
- [108] Wikimedia Foundation. [n. d.]. English Wikipedia Data Dump. <https://dumps.wikimedia.org/enwiki/>. Accessed: 2026-06-08.
- [109] Yanwen Xia, Benyong Deng, Hong Huang, Yuzheng Wang, Quanxi Li, Xingda Wei, Xiaobing Feng, Huimin Cui, and Chenxi Wang. 2026. Starfish: Supplementary Material. Retrieved October 14, 2026 from <https://ssk015.github.io/files/papers/starfish-supplementary.pdf>
- [110] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. 2024. Nomad: Non-Exclusive Memory Tiering via Transactional Page Migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, 19–35.
- [111] Yang Xiao, Ning Zhang, Wenjing Lou, and Y Thomas Hou. 2020. A survey of distributed consensus protocols for blockchain networks. *IEEE communications surveys & tutorials* 22, 2 (2020), 1432–1465.
- [112] Cong Xie, Oluwasanmi Koyejo, and Indranil Gupta. 2020. Fall of empires: Breaking byzantine-tolerant sgd by inner product manipulation. In *Uncertainty in artificial intelligence*. PMLR, 261–270.
- [113] Dong Xu, Junhee Ryu, Kwangsik Shin, Pengfei Su, and Dong Li. 2024. FlexMem: Adaptive page profiling and migration for tiered memory. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 817–833.
- [114] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 331–345.
- [115] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-Truth. In *2012 IEEE 12th International Conference on Data Mining*. 745–754. doi:10.1109/

- [116] Anil Yelam, Stewart Grant, Saarth Deshpande, Nadav Amit, Radhika Niranjana Mysore, Amy Ousterhout, Marcos K. Aguilera, and Alex C. Snoeren. 2025. Eden: Developer-Friendly Application-Integrated Far Memory. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 1067–1083. <https://www.usenix.org/conference/nsdi25/presentation/yelam>
- [117] Wonsup Yoon, Jisu Ok, Jinyoung Oh, Sue Moon, and Youngjin Kwon. 2023. Dilos: Do not trade compatibility for performance in memory disaggregation. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 266–282.
- [118] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 15–28.
- [119] Yuhong Zhong, Daniel S Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D Hill, Mosharaf Chowdhury, et al. 2024. Managing memory tiers with CXL in virtualized environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 37–56.
- [120] Hai Zhou, Dan Feng, Yuchong Hu, Wei Wang, and Huadong Huang. 2024. CoRD: Combining Raid and Delta for Fast Partial Updates in Erasure-Coded Storage Clusters. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.
- [121] Yang Zhou, Hassan MG Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E Culler, and Henry M Levy. 2022. Carbink: Fault-tolerant far memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 55–71.
- [122] Ruidong Zhu, Zili Zhang, Bingyang Wu, Xin Yao, Renhai Chen, Gong Zhang, Xuanzhe Liu, and Xin Jin. 2026. Epiphron: Resource-Efficient Distributed Key-Value Storage. *IEEE Transactions on Knowledge and Data Engineering* 38, 5 (2026), 2779–2792. doi:10.1109/TKDE.2026.3669983